ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA MECÁNICA

DESARROLLO DE UNA METODOLOGÍA PARA ANÁLISIS MODAL OPERACIONAL UTILIZANDO SOFTWARE LIBRE

TRABAJO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO MECÁNICO

ARANDA PAZMIÑO JONATHAN PAÚL
jonathan.aranda@epn.edu.ec

DIRECTOR: ING. GUACHAMÍN ACERO WILSON IVÁN, Ph.D.
wilson.guachamin@epn.edu.ec

Quito, julio 2020

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por **Aranda Pazmiño Jonathan Paúl**, bajo mi supervisión.

____

Ing. Guachamín Acero Wilson Iván, Ph.D.

DIRECTOR DE PROYECTO

DECLARACIÓN

Yo, **Aranda Pazmiño Jonathan Paúl**, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedo mis derechos de propiedad intelectual correspondiente a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normativa institucional vigente.

____

Aranda Pazmiño Jonathan Paúl

DEDICATORIA

A mi padre Juan y a mi madre Janis (†), por nunca abandonarme, por todo el esfuerzo y sacrificio que realizaron para brindarme siempre lo mejor.

A mis hermanos Sandra, Juan Carlos y Diego, por su apoyo incondicional a lo largo de mi vida, por haberme forjado como la persona que soy en la actualidad, por ser mis mentores y por cuidar de mí cuando más lo necesité.

AGRADECIMIENTO

A Dios, por llenar mi vida de muchas bendiciones, a mi padre Juan y a mi madre Janis (†), por su amor, por el esfuerzo y sacrificio que realizaron para brindarme siempre lo mejor. Agradezco a mi padre, por sus consejos, por su apoyo incondicional, y en especial, por nunca abandonarme en los momentos más difíciles.

A mis hermanos Sandra, Juan Carlos y Diego, por su apoyo incondicional a lo largo de mi vida, por ser mis mentores y por cuidar de mí cuando más lo necesité, muchos de mis éxitos son gracias a ellos. También agradezco a mi cuñado César y a mi cuñada Verónica, por su valiosa ayuda durante mi niñez y juventud.

A Belén Arellano, por su amor, apoyo, paciencia y por ser mi compañera de vida durante tantos años. A mis sobrinas Liz, Paula y Victoria, por inspirarme a mejorar para lograr ser un ejemplo para ellas.

Agradezco a la Facultad de Ingeniería Mecánica de la Escuela Politécnica Nacional, por permitirme estudiar mi pasión y formarme como un profesional de excelencia. A mis compañeros, por su amistad y por llenar de gratos momentos mi etapa universitaria.

Finalmente, quiero expresar un agradecimiento especial al Ing. Wilson Guachamín, por su valiosa ayuda, por compartir sus conocimientos y experiencia, permitiéndome desarrollar exitosamente este trabajo de titulación.

ÍNDICE

CERTIFICACIÓN....................i
DECLARACIÓN....................ii
DEDICATORIA....................iii
AGRADECIMIENTO....................iv
ÍNDICE....................v
ÍNDICE DE FIGURAS....................vii
ÍNDICE DE TABLAS....................ix
GLOSARIO DE TÉRMINOS....................x
RESUMEN....................xii
ABSTRACT....................xiii
INTRODUCCIÓN....................1
Objetivo general....................3
Objetivos específicos....................3
Alcance....................3
1. MARCO TEÓRICO....................4
1.1. Introducción al análisis modal....................4
1.1.1. Sistemas discretos y continuos....................4
1.1.2. Análisis dinámico de sistemas continuos....................4
1.1.3. Análisis modal....................8
1.1.4. Tipos de análisis modal....................8
1.2. Análisis Modal Operacional....................10
1.2.1. Supuestos de OMA....................11
1.2.2. Ventajas y desventajas de OMA....................11
1.2.3. Códigos computacionales o softwares para OMA....................12
1.3. Técnicas de análisis modal operacional....................14
1.3.1. Identificación de picos (PP)....................15
1.4. Equipos analizadores de vibraciones....................17
1.4.1. Equipo de adquisición de señales dinámicas....................17
1.4.2. Analizador de vibraciones Fluke 810....................18
1.4.3. Medidor de vibraciones Lutron VB-8213....................18
2. METODOLOGÍA....................20
2.1. Modelo numérico....................21
2.1.1. Diseño del modelo geométrico....................21
2.1.2. Simulación del modelo numérico en ANSYS....................22

2.2. Modelo experimental........22
2.2.1. Fuente de excitación........22
2.2.2. Condiciones del modelo experimental........23
2.2.3. Disposición de acelerómetros........24
2.3. Adquisición de datos........25
2.3.1. Equipo de adquisición de señales dinámicas........25
2.3.2. Software MAINTraq Viewer........25
2.3.3. Software MAINTraq Analyzer........26
2.4. Procesamiento de datos........27
2.5. Análisis modal en OpenModal........29
2.5.1. Módulo “Geometry”........29
2.5.2. Módulo “Measurement”........30
2.5.3. Módulo “Analysis”........30
2.5.4. Módulo “Animation”........32
2.6. Validación de resultados experimentales........32
2.7. Cálculo del factor de amortiguamiento........33
3. RESULTADOS Y DISCUSIÓN........34
3.1. Modos de vibración en ANSYS........34
3.2. Filtrado de las señales de aceleración........36
3.3. Modos de vibración en OpenModal........42
3.4. Análisis de errores........45
3.5. Análisis de amortiguamiento........46
4. CONCLUSIONES Y RECOMENDACIONES........48
4.1. Conclusiones........48
4.2. Recomendaciones........49
REFERENCIAS BIBLIOGRÁFICAS........50
ANEXOS........54

ÍNDICE DE FIGURAS

Figura 1.1. Sistema discreto de masa-resorte con un grado de libertad. 4
Figura 1.2. Sistema continuo de una viga en voladizo. 4
Figura 1.3. Ejemplos de formas modales. (a) Modo 1; (b) Modo 2; (c) Modo 3. 5
Figura 1.4. Concepto de la FRF en EMA de un sistema lineal. 9
Figura 1.5. Concepto de la FRF en OMA de un sistema lineal. 10
Figura 1.6. Interfaz de usuario del software ARTeMIS Modal. 12
Figura 1.7. Interfaz de usuario del software OpenModal. 13
Figura 1.8. Interfaz de usuario del software PULSE Operational Modal Analysis. 14
Figura 1.9. Representación de los picos de la función PSD en la técnica PP. 16
Figura 1.10. Equipo de adquisición de señales dinámicas del LAEV. 18
Figura 1.11. Analizador de vibraciones Fluke 810. 18
Figura 1.12. Medidor de vibraciones Lutron VB-8213. 19
Figura 2.1. Metodología para determinar los modos de vibración. 20
Figura 2.2. Modelo geométrico de la viga en SolidWorks (Dimensiones en mm). 21
Figura 2.3. Fuente de excitación de la viga. 23
Figura 2.4. Sujeción de los extremos de la viga mediante pernos. 23
Figura 2.5. Sujeción de la viga a la fuente de excitación. 23
Figura 2.6. Ubicación de los acelerómetros sobre la viga. 24
Figura 2.7. Instalación del modelo experimental. 24
Figura 2.8. Configuración del ensayo. 25
Figura 2.9. Exportación de formas de onda de aceleración de los tres acelerómetros. 26
Figura 2.10. Configuración de los nodos de la viga en OpenModal. 29
Figura 2.11. Unión de nodos de la viga en OpenModal. 30
Figura 2.12. Configuración de la dirección de la respuesta dinámica. 30
Figura 2.13. Configuración de rango de frecuencias a analizar en OpenModal. 31
Figura 2.14. Técnica PP en la FRF en OpenModal. 31
Figura 2.15 Frecuencias naturales obtenidas en OpenModal. 32
Figura 2.16. Método del ancho de banda de media potencia. 33
Figura 3.1. Primer modo de vibración de la viga en ANSYS. 35
Figura 3.2. Segundo modo de vibración de la viga en ANSYS. 36
Figura 3.3. Tercer modo de vibración de la viga en ANSYS. 36
Figura 3.4. Respuesta en frecuencia del filtro Butterworth paso bajo de tercer orden. 38
Figura 3.5. Señal original y filtrada del acelerómetro 1. 38
Figura 3.6. Señal original y filtrada del acelerómetro 2. 38

Figura 3.7. Señal original y filtrada del acelerómetro 3..39
Figura 3.8. Componentes en frecuencia de la señal del acelerómetro 1. 40
Figura 3.9. Componentes en frecuencia de la señal del acelerómetro 2. 40
Figura 3.10. Componentes en frecuencia de la señal del acelerómetro 3. 41
Figura 3.11. PSD de la señal del acelerómetro 1. 42
Figura 3.12. Ajuste de curva de la FRF de la señal del acelerómetro 1. 43
Figura 3.13. Ajuste de curva de la FRF de la señal del acelerómetro 2. 43
Figura 3.14. Ajuste de curva de la FRF de la señal del acelerómetro 3. 44
Figura 3.15. Selección de frecuencias de media potencia en MAINTraq Analyzer. 47

ÍNDICE DE TABLAS

Tabla 1.1. Condiciones de borde comunes para vibración transversal en vigas................ 7
Tabla 2.1. Ubicación de los acelerómetros en la viga. 24
Tabla 3.1. Frecuencias naturales de modos flexionantes de vibración obtenidos en ANSYS y analíticamente para diferentes casos. 34
Tabla 3.2. Frecuencias naturales obtenidas en ANSYS y en MAINTraq Analyzer para diferentes frecuencias de excitación. 37
Tabla 3.3. Frecuencias naturales obtenidas en ANSYS y MAINTraq Analyzer. 41
Tabla 3.4. Modos de vibración obtenidos en OpenModal...44
Tabla 3.5. Error relativo de frecuencias naturales obtenidas en ANSYS y OpenModal. ...45
Tabla 3.6. Comparación de formas modales obtenidas en ANSYS y OpenModal............46
Tabla 3.7. Resultados para factores de amortiguamiento. 47

GLOSARIO DE TÉRMINOS

$\bar{\ }$	Compleja conjugada
^{T}	Transpuesta
β	Constante
ξ	Factor de amortiguamiento
ρ	Densidad
ϕ	Forma modal
A	Área de sección transversal
$[C]$	Matriz de amortiguamiento
C_1, C_2, C_3, C_4	Constantes
E	Módulo de Young
f_a, f_b	Frecuencias de media potencia
f_n	Frecuencia natural
$f(t)$	Fuerza de excitación
$G_{XX}(\omega)$	Función de densidad espectral de potencia de entrada
$G_{YY}(\omega)$	Función de densidad espectral de potencia de salida
$H(\omega)$	Función de respuesta en dominio de frecuencia
I	Momento de inercia de área
$[K]$	Matriz de rigidez
l	Longitud
$[M]$	Matriz de masa
$p(t)$	Coordenada modal
W	Deflexión transversal
$x(t)$	Desplazamiento en el dominio del tiempo
$X(\omega)$	Espectro de fuerza de excitación
$y(t)$	Respuesta dinámica en el dominio del tiempo
$Y(\omega)$	Espectro de respuesta dinámica

ACRÓNIMOS

AEF	Análisis de elementos finitos
ARMA	Modelo autorregresivo de media móvil
CAD	Diseño asistido por computador
CSV	Formato delimitado por comas
DFT	Transformada discreta de Fourier
EFDD	Descomposición en el dominio de la frecuencia mejorada

EMA	Análisis modal experimental
FDD	Descomposición en el dominio de la frecuencia
FFT	Transformada rápida de Fourier
FRF	Función de respuesta en dominio de frecuencia
IDFT	Transformada de Fourier discreta inversa
LAEV	Laboratorio De Análisis De Esfuerzos Y Vibraciones
LTI	Lineal invariante en el tiempo
MDOF	Múltiples grados de libertad
MIMO	Múltiples entradas múltiples salidas
NExT	Técnica de excitación natural
OMA	Análisis modal operacional
PP	Identificación de picos
PSD	Densidad espectral de potencia
SDOF	Un grado de libertad
SHM	Monitoreo de salud estructural
SSI	Identificación de subespacios estocásticos
SVD	Descomposición en valores singulares
UFF	Formato universal de archivos

RESUMEN

El presente proyecto de titulación tiene por objetivo desarrollar una metodología para análisis modal operacional (OMA), utilizando el software libre OpenModal y el equipo de adquisición de señales dinámicas del Laboratorio de Análisis de Esfuerzos y Vibraciones (LAEV). Como caso de estudio se analiza un perfil estructural tipo U, de material Acero ASTM A36, doblemente empotrado. Inicialmente, se realiza la simulación de un modelo numérico en el software ANSYS para conocer los modos de vibración que se esperan obtener experimentalmente. A continuación, se instala el modelo experimental y se realizan las mediciones de las respuestas dinámicas empleando acelerómetros y el equipo de adquisición de señales dinámicas. Posteriormente, se realiza el procesamiento de las señales en el lenguaje de programación Python, en el cual la señal es filtrada y usada para calcular la transformada discreta de Fourier, esto con el objetivo de adecuar las señales para realizar el análisis modal en el software libre OpenModal. En este software se determinaron los tres primeros modos de vibración de la estructura, resultados que fueron validados con los resultados obtenidos en ANSYS. Finalmente, se determinaron los parámetros dinámicos de la viga doblemente empotrada.

Palabras clave: Análisis modal operacional, ANSYS Modal, modos de vibración, OpenModal, viga doblemente empotrada.

ABSTRACT

This project deals with the development of a methodology for operational modal analysis (OMA), using the OpenModal free software and a dynamic signal acquisition equipment available at the Stress and Vibration Analysis Laboratory (LAEV). A double-embedded beam with a U cross section is used as a case study. A corresponding numerical model is developed in ANSYS in order to know in advance the expected modes of vibration. The experimental setup includes accelerometers and a dynamic signal acquisition equipment. Dynamic responses of the beam are measured and analyzed in the Python programming language, where time histories of the signals are filtered and used to calculate the discrete Fourier transform, with the purpose of preparing the signals for analysis in OpenModal. In this software, the first three modes of vibration of the structure are determined. Results were validated by comparing them against results obtained from ANSYS. Finally, the dynamic properties of the double-embedded beam are provided.

Keywords: ANSYS Modal, double-embedded beam, modes of vibration, operational modal analysis, OpenModal.

DESARROLLO DE UNA METODOLOGÍA PARA ANÁLISIS MODAL OPERACIONAL UTILIZANDO SOFTWARE LIBRE

INTRODUCCIÓN

La ingeniería estructural, en su continuo desarrollo, ha implementado técnicas experimentales con el objetivo de comprender lo que ocurre con las estructuras en la etapa posterior a la construcción, y que permitan la evaluación de su integridad estructural. Esto es posible conociendo las propiedades dinámicas reales de la estructura, que se pueden obtener a través de la técnica de análisis modal (Henao, Botero, & Muriá, 2014).

El conocimiento de respuestas dinámicas generadas por vibraciones es de vital importancia, ya que permite determinar la existencia de problemas estructurales y establecer patrones de funcionamiento (Rojas, 2014). Varios países cuentan con la tecnología necesaria para el diseño, optimización, monitoreo, control de vibraciones y detección de daños en estructuras tales como puentes y edificios (Masjedian & Keshmiri, 2009).

En la actualidad, los métodos de análisis modal más empleados para determinar las propiedades dinámicas de estructuras sometidas a vibraciones son el análisis modal experimental (experimental modal analysis - EMA) y el análisis modal operacional (operational modal analysis - OMA). La identificación de propiedades dinámicas de forma experimental se registra a mediados del siglo XX (Ewins, 2000), no obstante, la concepción de los modos de vibración (patrón o forma característica en el que vibra un sistema mecánico) se remonta al siglo XVIII con los estudios de Daniel y Johann Bernoulli, Euler y d'Alembert (Kline, 1990).

El primer y más significante método de EMA fue propuesto por C. C. Kennedy y C. D. Pancu en 1947, posteriormente, J. W. Cooly y J. W. Turkey en 1965 desarrollaron el algoritmo de la Transformada Rápida de Fourier (FFT), lo que permitió establecer los fundamentos para la aplicación de técnicas experimentales en el análisis dinámico estructural (He & Fu, 2001).

El OMA apareció por primera vez con el desarrollo del programa espacial de los Estados Unidos en los años de 1950 (Nikitas, Hugh, & Tsavdaridis, 2015). La identificación de propiedades dinámicas a partir de las mediciones de respuestas dinámicas, desarrolló varias aplicaciones en ingeniería mecánica y aeroespacial como se detalla en Ewins (1986), Ljung (1987), y en Juang (1994).

El OMA ha sido aplicado en el estudio estructural de varios puentes, por ejemplo: Golden Gate (Abdel & Scanlan, 1985), Fatih Sultan Mehmet (Brownjohn, Dumanoglu, & Severn, 1992), Tsing Ma (Xu, Ko, & Zhang, 1997), Hitsuishijima (Okauchi, Miyata, Tatsumi, & Sasaki, 1997), Vasco da Gama (Cunha, Caetano, & Delgado, 2001), Kap Shui Mun (Chang, Chang, & Zhang, 2001), Roebling (Ren, Harik, Lenett, & Basehearh, 2001), Girder (Ren, Zhao, & Harik, 2003), etc.

En los últimos años debido al constante avance tecnológico, en términos de sistemas de medición de vibraciones y métodos de análisis, el OMA ha tenido un papel importante en la mayoría de industrias, ya que se ha convertido en una herramienta con un extenso campo de aplicación en Ingeniería Civil, Mecánica y Aeroespacial.

En el campo de la investigación y la industria existen varios softwares empleados en el análisis modal: OpenModal (Slavic, Mrsnik, Pirnat, & Starc, 2015), ARTeMIS Modal (Structural Vibration Solutions, 2019), BK Connect Modal Analysis (Brüel & Kjaer, 2018), PULSE Operational Modal Analysis (Brüel & Kjaer, 2018), entre otros. Estos códigos son usados tanto en industria como en actividades de investigación a nivel mundial.

El Ecuador se encuentra localizado en una zona calificada de alto riesgo sísmico, lo que conlleva a mejorar los mecanismos de control en los procesos constructivos, optimizar el diseño y montaje estructural (Ministerio de Desarrollo Urbano y Vivienda, 2014). En Ecuador existen varias estructuras como son puentes y edificios que se encuentran en operación desde hace varios años, sin embargo, no se han evaluado de forma cuantitativa sus condiciones estructurales. Esto se debe a que en nuestro país el análisis modal es un tema nuevo y no se han desarrollado estudios previos para la creación de una herramienta que permita determinar propiedades dinámicas de estructuras o componentes mecánicos sometidos a vibraciones.

En este contexto, el presente proyecto plantea desarrollar una metodología, y por tanto una herramienta, que permita determinar propiedades dinámicas y modos de vibración de estructuras empleando el equipo de adquisición de señales dinámicas del Laboratorio de Análisis de Esfuerzos y Vibraciones (LAEV) junto con el método de OMA disponible en el software libre OpenModal, y de esta manera contribuir al conocimiento sobre integridad estructural en el país.

Objetivo general

Desarrollar una metodología para análisis modal operacional (OMA) utilizando el software libre OpenModal.

Objetivos específicos

- Potencializar el uso del equipo de adquisición de señales dinámicas del Laboratorio de Análisis de Esfuerzos y Vibraciones (LAEV).
- Desarrollar un código en Python para procesar los datos medidos con el equipo de adquisición de señales dinámicas, y exportar al software OpenModal.
- Determinar experimentalmente los modos fundamentales de vibración de una estructura mediante el software OpenModal.
- Validar los modos de vibración obtenidos experimentalmente mediante simulaciones de un modelo numérico en ANSYS.

Alcance

La metodología a desarrollar para análisis modal operacional permite determinar experimentalmente los modos de vibración de una estructura a partir de la medición de las respuestas dinámicas. Se analizan solamente modos de vibración flexionantes, no se consideran modos torsionales. El código a desarrollar en Python permite filtrar las señales de aceleraciones, calcular la transformada discreta de Fourier y adecuar estas señales para ser exportadas a OpenModal. Mediante este software se determinan los tres primeros modos fundamentales de vibración de una viga doblemente empotrada, por tanto, se determinan sus parámetros dinámicos.

1. MARCO TEÓRICO

En el presente proyecto se propone desarrollar una metodología para la implementación de OMA utilizando el software libre OpenModal. Para esto, en este capítulo se presenta la revisión bibliográfica del análisis dinámico, y posteriormente, de los fundamentos y técnicas del OMA.

1.1. Introducción al análisis modal

1.1.1. Sistemas discretos y continuos

En un sistema dinámico, el número de grados de libertad del sistema está definido por el número mínimo de coordenadas independientes requeridas para determinar completamente las posiciones de todas las partes del sistema en cualquier instante de tiempo (Rao, 2018).

Los sistemas discretos son aquellos con un número finito de grados de libertad, como se muestra en la Figura 1.1, la cual corresponde a un sistema masa-resorte con un grado de libertad (SDOF), donde la coordenada x es empleada para describir el movimiento del sistema.

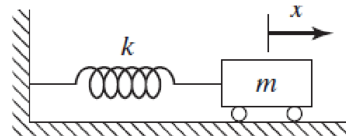


Figura 1.1. Sistema discreto de masa-resorte con un grado de libertad.
(Fuente: Rao, 2018)

Por otra parte, los sistemas continuos son aquellos con un número infinito de grados de libertad, como se muestra en la Figura 1.2, en donde se representa una viga en voladizo con un número infinito de coordenadas necesarias para describir su deflexión.

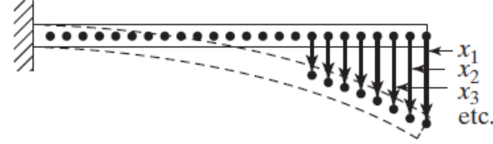


Figura 1.2. Sistema continuo de una viga en voladizo.
(Fuente: Rao, 2018)

1.1.2. Análisis dinámico de sistemas continuos

El análisis estructural, ya sean de manera analítica, experimental u operacional, consiste en la obtención de los parámetros modales de la estructura, los cuales gobiernan el comportamiento dinámico de la misma.

Los parámetros modales son: los modos de vibración, los factores de amortiguamiento y las frecuencias naturales. La obtención de estos parámetros permite la reconstrucción de los parámetros físicos de la estructura: masa, rigidez y amortiguamiento de la misma.

Los modos naturales de vibración son características inherentes de un sistema dinámico, y están determinados por sus propiedades dinámicas y sus distribuciones espaciales. Cada modo se describe en forma de sus parámetros modales como la frecuencia natural, factor de amortiguamiento y forma modal. En la vibración global de un sistema mecánico, la importancia de cada modo natural de vibración depende de la fuente de excitación y de las formas modales del sistema vibratorio (He & Fu, 2001). A continuación, se definen los parámetros modales.

- Forma modal: es un patrón de vibración, el cual se define como las posibles formas geométricas que adopta un sistema mecánico al estar sometido a vibraciones a una determinada frecuencia (Gómez & Herrera, 2011). En la Figura 1.3 se muestran las tres primeras formas modales de una viga doblemente empotrada.

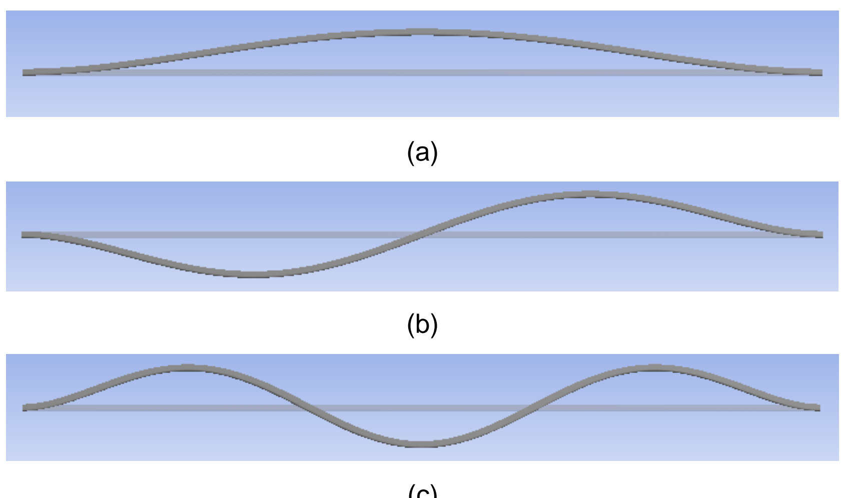


Figura 1.3. Ejemplos de formas modales. (a) Modo 1; (b) Modo 2; (c) Modo 3.
(Fuente: Propia)

- Frecuencia natural: se define como la frecuencia a la que un sistema mecánico continúa vibrando después de cesar la señal de excitación. Un sistema mecánico con múltiples grados de libertar (MDOF), posee múltiples frecuencias naturales, las cuales corresponden a las frecuencias presentes en las distintas formas modales (Harris, 2002).
- Amortiguamiento: en cualquier sistema dinámico real, el amortiguamiento es la disipación de energía mecánica, generalmente por la conversión en energía interna,

lo que provoca una disminución constante de la amplitud de la vibración libre hasta llegar a valores despreciables (Harris, 2002).

Los modos de vibración anteriormente descritos, se pueden determinar de manera analítica partiendo de la teoría elemental de flexión de vigas de Euler-Bernoulli, la cual se expresa en la Ecuación 1.1 para una viga uniforme.

$$EI\frac{\partial^4 W}{\partial x^4}(x,t) + \rho A\frac{\partial^2 W}{\partial t^2}(x,t) = f(x,t) \qquad \text{(Ec. 1.1.)}$$

Donde:

E: Módulo de Young

I: Momento de inercia de la sección transversal

ρ: Densidad del material

A: Área de la sección transversal

f: Fuerza externa por unidad de longitud

W: Deflexión transversal

Considerando una vibración libre, en donde la fuerza de excitación es nula, y aplicando el método de separación de variables, la Ecuación 1.2 muestra la función característica de la deflexión transversal de una viga. Las constantes desconocidas $C_1, C_2, C_3, C_4\ y\ \beta$ pueden ser determinadas a partir de las condiciones de borde de la viga como se muestra en la Tabla 1.1.

$$\begin{aligned} W(x) = {} & C_1(\cos\beta x + \cosh\beta x) + C_2(\cos\beta x - \cosh\beta x) \\ & + C_3(\sin\beta x + \sinh\beta x) + C_4(\sin\beta x - \sinh\beta x) \end{aligned} \qquad \text{(Ec. 1.2.)}$$

Por otra parte, las frecuencias naturales de la viga se pueden determinar mediante la Ecuación 1.3 como se muestra a continuación.

$$f_n = \frac{\beta^2}{2\pi}\sqrt{\frac{EI}{\rho A}} = \frac{(\beta l)^2}{2\pi}\sqrt{\frac{EI}{\rho A l^4}} \qquad \text{(Ec. 1.3.)}$$

Donde:

f_n: Frecuencia natural $[Hz]$

l: Longitud de la viga

Tabla 1.1. Condiciones de borde comunes para vibración transversal en vigas.

Condición de extremos	Ecuación de frecuencia	Forma modal (función normal)	Valor de $\beta_n l$
Apoyado - Apoyado	$sin(\beta_n l) = 0$	$W_n(x) = C_n[sin(\beta_n x)]$	$\beta_1 l = \pi$ $\beta_2 l = 2\pi$ $\beta_3 l = 3\pi$ $\beta_4 l = 4\pi$
Libre - Libre	$cos(\beta_n l) \cdot cosh(\beta_n l) = 1$	$W_n(x) = C_n\big[sin(\beta_n x) + sinh(\beta_n x) + \alpha_n\big(cos(\beta_n x) + cosh(\beta_n x)\big)\big]$ Donde: $\alpha_n = \frac{sin(\beta_n l) - sinh(\beta_n l)}{cosh(\beta_n l) - cos(\beta_n l)}$	$\beta_1 l = 4{,}730041$ $\beta_2 l = 7{,}853205$ $\beta_3 l = 10{,}995608$ $\beta_4 l = 14{,}137165$ ($\beta l = 0$ para modo de cuerpo rígido)
Empotrado – Empotrado	$cos(\beta_n l) \cdot cosh(\beta_n l) = 1$	$W_n(x) = C_n\big[sinh(\beta_n x) - sin(\beta_n x) + \alpha_n\big(cosh(\beta_n x) - cos(\beta_n x)\big)\big]$ Donde: $\alpha_n = \frac{sinh(\beta_n l) - sin(\beta_n l)}{cos(\beta_n l) - cosh(\beta_n l)}$	$\beta_1 l = 4{,}730041$ $\beta_2 l = 7{,}853205$ $\beta_3 l = 10{,}995608$ $\beta_4 l = 14{,}137165$
Empotrado - Libre	$cos(\beta_n l) \cdot cosh(\beta_n l) = -1$	$W_n(x) = C_n\big[sin(\beta_n x) - sinh(\beta_n x) - \alpha_n\big(cos(\beta_n x) - cosh(\beta_n x)\big)\big]$ Donde: $\alpha_n = \frac{sin(\beta_n l) + sinh(\beta_n l)}{cos(\beta_n l) + cosh(\beta_n l)}$	$\beta_1 l = 1{,}875104$ $\beta_2 l = 4{,}694091$ $\beta_3 l = 7{,}854757$ $\beta_4 l = 10{,}995541$
Empotrado – Apoyado	$tan(\beta_n l) - tanh(\beta_n l) = 0$	$W_n(x) = C_n\big[sin(\beta_n x) - sinh(\beta_n x) + \alpha_n\big(cosh(\beta_n x) - cos(\beta_n x)\big)\big]$ Donde: $\alpha_n = \frac{sin(\beta_n l) - sinh(\beta_n l)}{cos(\beta_n l) - cosh(\beta_n l)}$	$\beta_1 l = 3{,}926602$ $\beta_2 l = 7{,}068583$ $\beta_3 l = 10{,}210176$ $\beta_4 l = 13{,}351768$
Apoyado - Libre	$tan(\beta_n l) - tanh(\beta_n l) = 0$	$W_n(x) = C_n[sin(\beta_n x) + \alpha_n sinh(\beta_n x)]$ Donde: $\alpha_n = \frac{sin(\beta_n l)}{sinh(\beta_n l)}$	$\beta_1 l = 3{,}926602$ $\beta_2 l = 7{,}068583$ $\beta_3 l = 10{,}210176$ $\beta_4 l = 13{,}351768$ ($\beta l = 0$ para modo de cuerpo rígido)

(Fuente: Rao, 2018)

(Elaboración: Propia)

Sin embargo, también es posible determinar los modos de vibración de estructuras mediante la técnica de análisis modal, la cual se describe en el siguiente apartado.

1.1.3. Análisis modal

El análisis modal es el proceso de determinar las propiedades dinámicas inherentes de un sistema vibratorio, en forma de frecuencias naturales, factores de amortiguamiento y formas modales, y de esta manera formular un modelo matemático que represente su comportamiento dinámico (He & Fu, 2001). Las respuestas dinámicas de un sistema estructural dependen de los modos de vibración y las características de la fuerza de excitación. La respuesta vibratoria de un sistema dinámico lineal invariante en el tiempo (LTI), puede ser expresada como la combinación lineal de un conjunto de movimientos armónicos simples denominados modos naturales de vibración, a esto se conoce como principio de superposición (He & Fu, 2001).

El análisis modal es empleado en la detección de daños en estructuras, al comparar los parámetros modales originales con los parámetros modales de la estructura después de estar en servicio algún tiempo, por lo tanto, una variación en las propiedades modales indica la existencia de problemas estructurales.

1.1.4. Tipos de análisis modal

El análisis modal abarca tanto técnicas teóricas como experimentales. El análisis modal teórico se basa en un modelo físico de un sistema dinámico que comprende propiedades como la masa, rigidez y amortiguamiento, las cuales están dadas en forma de ecuaciones diferenciales. Además, para obtener un modelo físico más realista es necesario expresar las mismas propiedades anteriormente mencionadas, en términos de sus distribuciones espaciales, por consiguiente, toman el nombre de matrices de masa, rigidez y amortiguamiento, las cuales se incorporan en un conjunto de ecuaciones diferenciales normales de movimiento como se muestra en la Ecuación 1.4.

$$[M]\{\ddot{x}(t)\} + [C]\{\dot{x}(t)\} + [K]\{x(t)\} = \{f(t)\} \qquad \text{(Ec. 1.4.)}$$

Donde:

$[M], [C], [K]$: Matrices de masa, amortiguamiento y rigidez, respectivamente

$x(t), \dot{x}(t), \ddot{x}(t)$: Vectores desplazamiento, velocidad y aceleración, respectivamente

$\{f(t)\}$: Vector fuerza

Por otra parte, el EMA es una técnica empleada para determinar el modelo modal de un sistema vibratorio LTI. El EMA se basa en la función de transferencia $H(\omega)$ entre la respuesta dinámica (desplazamiento, velocidad o aceleración) $Y(\omega)$, y la fuerza de excitación $X(\omega)$, en función de la frecuencia de excitación (He & Fu, 2001), como se muestra en la Figura 1.4. Esta función de transferencia se denomina función de respuesta en dominio de frecuencia (FRF), y se obtiene al aplicar la transformada discreta de Fourier (DFT) tanto a la señal de la fuerza de excitación, como a la señal de la respuesta dinámica.

Figura 1.4. Concepto de la FRF en EMA de un sistema lineal.
(Fuente: Propia)

Como muestra la Figura 1.4, el EMA consiste en hacer vibrar la estructura al aplicar una fuerza de excitación artificial conocida, la cual puede ser medida al igual que la respuesta dinámica. Se determinan los correspondientes espectros de fuerza y de respuesta dinámica, $X(\omega)$ y $Y(\omega)$ respectivamente, aplicando la transformada discreta de Fourier. De esta manera se puede determinar la función de transferencia $H(\omega)$, y por tanto identificar los parámetros modales. Debido a que se realizan los ensayos bajo condiciones controladas, este método proporciona información precisa sobre el sistema vibratorio.

Varios tipos de estructuras son difíciles de excitar artificialmente debido a su tamaño, forma o localización, por lo que en estas aplicaciones los ingenieros se han visto obligados a buscar una alternativa al EMA, que es el OMA, el cual se describe en la siguiente sección.

1.2. Análisis Modal Operacional

A menudo, como en el caso de puentes y edificios, no es factible excitar de una forma controlada una estructura y medir con exactitud dicha fuerza, es decir no se puede aplicar el EMA. El OMA es una técnica para obtener una descripción modal de una estructura bajo sus condiciones de operación. El OMA está basado en la medida de la respuesta dinámica debido a excitaciones ambientales durante la operación de la estructura, por ejemplo: viento, tráfico, oleaje, etc.

Pese a que en OMA no se realizan mediciones de las fuerzas de entrada, esta técnica toma la señal de un sensor de referencia como entrada, y las FRFs se calculan para cada punto de medición con respecto a este sensor de referencia. En el contexto del OMA, $H(\omega)$ no significa la relación entre la respuesta dinámica y la fuerza de entrada, sino más bien representa la relación entre la respuesta medida por un sensor cualquiera $Y_i(\omega)$, y la respuesta medida por un sensor de referencia $Y_{ref}(\omega)$ (Ren & Zong, 2004), como se muestra en la Figura 1.5. Esta relación o diferencia en las respuestas dinámicas en varios puntos de una estructura, permite establecer las formas modales de la estructura.

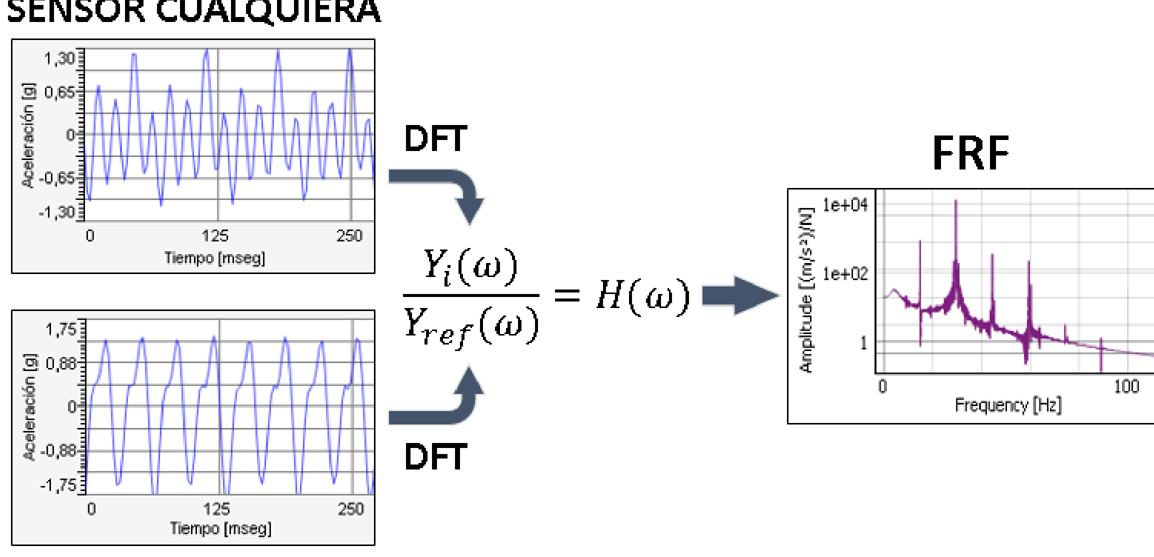


Figura 1.5. Concepto de la FRF en OMA de un sistema lineal.
(Fuente: Propia)

Actualmente, el OMA es una herramienta ampliamente empleada para la identificación de parámetros modales, con varias aplicaciones en ingeniería civil (puentes, edificios, puentes peatonales, estructuras históricas, plataformas marinas, turbinas de viento, represas, estadios, etc.), ingeniería mecánica (barcos, camiones, carrocerías, motores, maquinarias rotativas, etc.), e ingeniería aeroespacial (identificación modal en vuelo de aviones).

1.2.1. Supuestos de OMA

Según Rainieri y Fabbrocino (2014), el OMA está basado en los siguientes supuestos:

- Linealidad: dada una combinación de entradas, la respuesta del sistema es igual a la misma combinación de las correspondientes salidas.
- Estacionariedad: las características dinámicas de una estructura no varían en el tiempo, de modo que los coeficientes de las ecuaciones diferenciales que gobiernan la respuesta dinámica de la estructura son independientes del tiempo.
- Observabilidad: el arreglo de los sensores se diseña adecuadamente para observar los modos de interés.

1.2.2. Ventajas y desventajas de OMA

Desde principios de los años 90, el OMA ha sido el centro de atención en ingeniería civil, mecánica y aeroespacial, debido a las siguientes ventajas como manifiestan Masjedian y Keshmiri (2009):

- Al aplicar fuerzas reales aleatorias en diferentes partes de una estructura, se obtiene un modelo lineal real de las condiciones operacionales, en lugar de las condiciones experimentales.
- En OMA se obtienen las propiedades dinámicas de todo el sistema vibratorio, en lugar de solo una parte de él.
- Los ensayos operacionales son más baratos que los experimentales, ya que no se necesitan equipos de excitación y puede realizarse in situ.
- OMA es posible aplicar a estructuras difíciles de excitar artificialmente, debido a su tamaño, forma o localización.
- Es necesario menor tiempo para realizar el ensayo, ya que sólo se deben colocar los equipos de medida en un arreglo adecuado.
- El ensayo no interfiere ni interrumpe el funcionamiento normal de la estructura, por lo que puede permanecer en servicio durante el ensayo.
- Se evita la aplicación de cargas artificiales, lo que conlleva un riesgo de dañar la estructura.
- El análisis del OMA es de múltiples entradas y múltiples salidas (MIMO), por lo que es posible identificar fácilmente modos repetidos o cercanos. Por tanto, OMA es un método apropiado para estructuras complejas y complicadas.

- OMA puede ser empleado para el control de vibraciones de estructuras, así como la detección de daños estructurales y monitoreo de la salud estructural (SHM).

Sin embargo, este método presenta ciertas desventajas como son:

- Debido a que la fuerza de excitación es desconocida, el OMA es una técnica más difícil que el EMA.
- La intensidad de las señales de respuesta que se obtienen con vibraciones ambientales es baja y, a menudo, mezcladas con ruido.
- Se necesitan equipos de medición muy sensibles.
- Es necesario un análisis de datos cuidadoso.

1.2.3. Códigos computacionales o softwares para OMA

Con el constante avance tecnológico en términos computacionales, se han desarrollado softwares especializados para OMA, los cuales son indispensables en la etapa de procesamiento de datos. A continuación, se describen algunos softwares del estado del arte que se emplean en la industria e investigación:

- ARTeMIS Modal: es una plataforma abierta fácil de usar desarrollada por la empresa danesa Structural Vibration Solutions. Desde 1999, ha sido una herramienta potente y versátil para el OMA. A partir de la medición de vibraciones, este software proporciona los parámetros modales en términos de frecuencias naturales, factores de amortiguamiento y formas modales (Structural Vibration Solutions, 2019). En la Figura 1.6 se muestra la interfaz de usuario de este software.

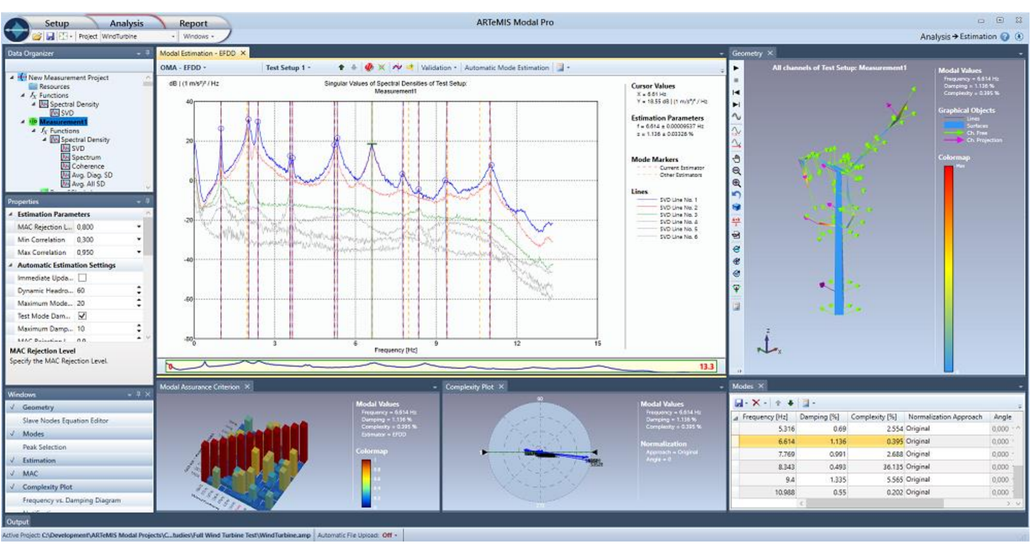


Figura 1.6. Interfaz de usuario del software ARTeMIS Modal.
(Fuente: Structural Vibration Solutions, 2019)

- OpenModal: es un software de análisis modal de código abierto escrito en lenguaje de programación Python, debido a esta característica, el presente proyecto se desarrolla empleando este software. OpenModal combina la generación de geometría, el módulo de medición, identificación y animación para determinar las propiedades dinámicas de las estructuras. Se puede desarrollar EMA y OMA, además el software brinda diferentes opciones de señales de excitación, como impulso o aleatorio (Slavic et al., 2015). En la Figura 1.7 se muestra la interfaz de usuario del software OpenModal, en donde a partir de la selección de los picos en la FRF, se determinan los parámetros modales.

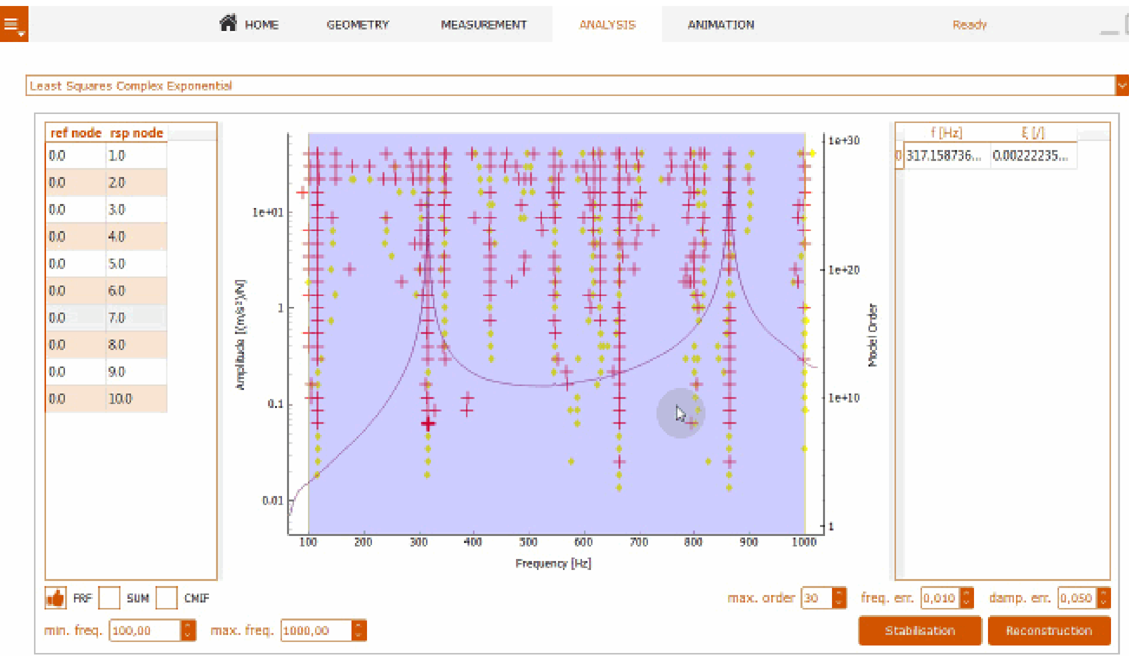


Figura 1.7. Interfaz de usuario del software OpenModal.
(Fuente: Slavic et al., 2015)

- PULSE Operational Modal Analysis: es un software desarrollado por Brüel & Kjaer (2018), para análisis modal mediante el post-procesamiento de los datos de medición en la salida de la estructura en condiciones reales de operación y donde es difícil excitar artificialmente la estructura. Este software para OMA contiene potentes algoritmos para una identificación modal precisa, además es muy fácil de usar gracias a su interfaz de usuario orientada a tareas, su flujo de trabajo intuitivo y su alto grado de automatización, como se muestra en la Figura 1.8.

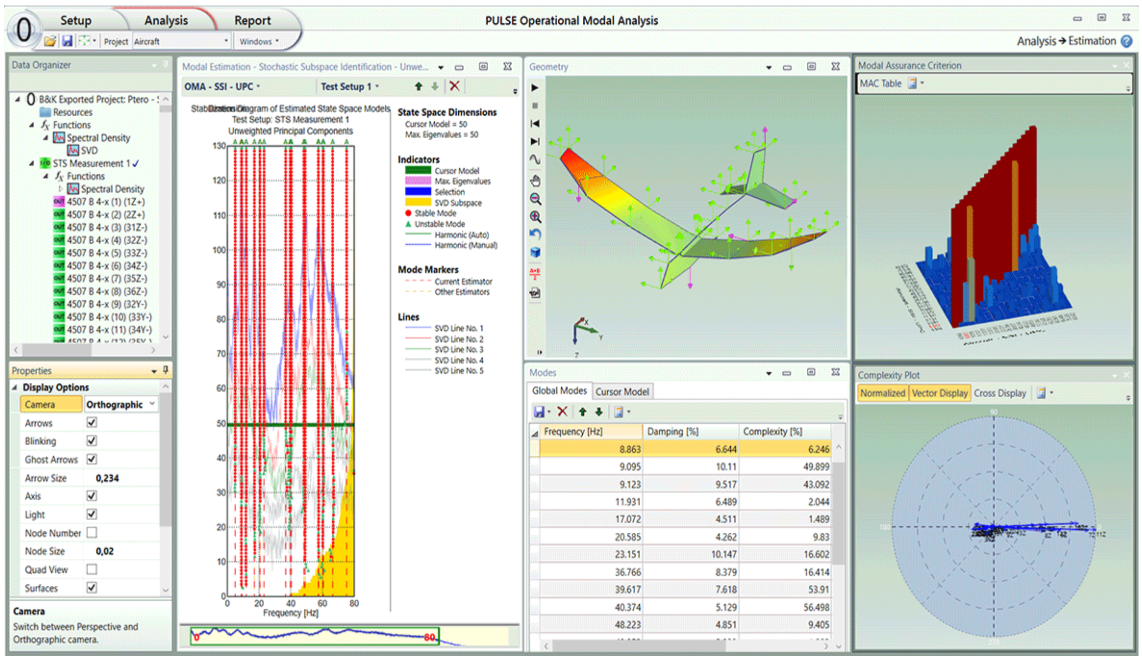


Figura 1.8. Interfaz de usuario del software PULSE Operational Modal Analysis.
(Fuente: Brüel & Kjaer, 2018)

1.3. Técnicas de análisis modal operacional

Existen varias técnicas para realizar OMA, ya sea en el dominio del tiempo o de la frecuencia. A continuación, se describen brevemente las técnicas que sobresalen en el dominio del tiempo:

- Técnica de Excitación Natural (NExT): se basa en que las funciones de correlación (obtenidas de la respuesta dinámica de una estructura bajo vibración ambiental), pueden ser expresadas como una suma de sinusoides amortiguadas. Cada sinusoide tiene una frecuencia natural amortiguada y un factor de amortiguamiento, los cuales están relacionados con su correspondiente modo de vibración (Rodríguez, 2005).
- Identificación de Subespacios Estocásticos (SSI): trabaja directamente con los datos temporales medidos sin necesidad de convertirlos en espectros. Es considerada la técnica más avanzada, sin embargo, la carga computacional es alta debido a su complejo desarrollo matemático (Zamora, 2016).
- Modelo autorregresivo de media móvil (ARMA): es una técnica empleada en sistemas LTI excitados con vibraciones ambientales, la cual representa el comportamiento dinámico de estructuras mediante ecuaciones diferenciales de orden superior (Hung, Thomas, Lakis, & Marcouiller, 2007).

Por otra parte, las técnicas en el dominio de la frecuencia se basan en las funciones de densidad espectral de potencia (PSD), las cuales se obtienen al convertir las medidas de las aceleraciones al dominio de la frecuencia mediante la DFT. A continuación, se describen brevemente las técnicas mayormente empleadas por la comunidad científica en el dominio de la frecuencia:

- Identificación de picos (PP): se basa en que la FRF alcanza valores máximos alrededor de las frecuencias naturales de la estructura. Las frecuencias naturales se determinan a partir de la observación de los picos en la gráfica de la función PSD (Ren & Zong, 2004).
- Descomposición en el dominio de la frecuencia (FDD): representa una mejora significativa de la técnica PP, eliminando sus desventajas, pero manteniendo la facilidad de uso. Consiste en factorizar a la PSD mediante el empleo de una técnica del álgebra lineal, llamada descomposición en valores singulares (SVD) (Gade, Moller, Herlufsen, & Konstantin-Hansen, 2005).
- Descomposición en el dominio de la frecuencia mejorada (EFDD): es una extensión mejorada de la técnica FDD, consiste en llevar la PSD al dominio del tiempo por medio de la transformada de Fourier discreta inversa (IDFT), y los factores de amortiguamiento se calculan empleando técnicas de decremento logarítmico (Masjedian & Keshmiri, 2009).

En el siguiente apartado se detalla únicamente la identificación de picos, ya que el presente proyecto se desarrolla mediante esta técnica debido a su simplicidad de implementación y velocidad de procesamiento.

1.3.1. Identificación de picos (PP)

La identificación de picos es la técnica más simple para la identificación de parámetros modales de estructuras sometidas a excitaciones ambientales. Esta técnica se basa en que la FRF alcanza valores máximos alrededor de las frecuencias naturales de la estructura. Las frecuencias naturales se determinan a partir de la observación de los picos en la gráfica de la función PSD (Ren & Zong, 2004).

La PSD representa la distribución de energía de la vibración en función de la frecuencia, cuyas unidades para datos de aceleración son $[g^2/Hz]$ o $[(m/s^2)^2/Hz]$ (Indian Institute of Technology, 2015). En la Figura 1.9 se representan los picos de la PSD, los cuales corresponden a las frecuencias naturales.

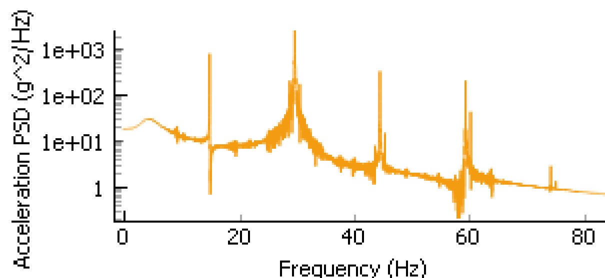


Figura 1.9. Representación de los picos de la función PSD en la técnica PP.
(Fuente: Propia)

Como afirman Rainieri y Fabbrocino (2014), si para cierto pico de resonancia solo el r-ésimo modo de vibración es dominante, la respuesta estructural es aproximadamente igual a la respuesta modal como indica la Ecuación 1.5.

$$y(t) \approx \phi_r \, p_r(t) \qquad \text{(Ec. 1.5.)}$$

Donde:

$y(t)$: Respuesta dinámica estructural

ϕ_r: Forma modal del r-ésimo modo

$p_r(t)$: Coordenada modal del r-ésimo modo

Por otra parte, la PSD se puede definir como muestra la Ecuación 1.6.

$$G_{YY}(\omega) = \overline{H}(\omega) \, G_{XX}(\omega) \, H(\omega)^T \qquad \text{(Ec. 1.6.)}$$

Donde:

$G_{YY}(\omega)$: Matriz PSD de salida

$G_{XX}(\omega)$: Matriz PSD de entrada

$\overline{H}(\omega)$: Matriz compleja conjugada de la FRF

$H(\omega)^T$: Matriz transpuesta de la FRF

La estimación de la frecuencia natural, del factor de amortiguamiento y de la forma modal están sujetos a aproximaciones y errores, por lo que no siempre proporcionan resultados fiables (Rodríguez, 2005).

Sin embargo, para estimar el factor de amortiguamiento es posible aplicar el método del ancho de banda de media potencia (Chopra, 2014).

La identificación de picos presenta las siguientes desventajas (Ren & Zong, 2004):

- Seleccionar los picos de las frecuencias naturales es una tarea subjetiva.
- Se obtienen las formas de deformación en lugar de las formas modales.
- Solo los modos reales o las estructuras con amortiguamiento proporcional pueden ser calculadas por este método.
- Las estimaciones de amortiguamiento no son confiables.

Pese a estas desventajas, la identificación de picos es muy empleada en estructuras en condiciones de operación, debido a su simplicidad de implementación y velocidad de procesamiento de los datos.

1.4. Equipos analizadores de vibraciones

En la actualidad existen varios equipos para realizar análisis de vibraciones en estructuras y maquinaria industrial, sin embargo, en nuestro ámbito nacional las opciones son muy limitadas. A continuación, se presentan los principales equipos disponibles en el mercado nacional.

1.4.1. Equipo de adquisición de señales dinámicas

Este equipo es un analizador de señales dinámicas desarrollado por IDEAR, consta de 16 canales para conectar acelerómetros, sensores de velocidad o sensores de proximidad. Trabaja junto con el software MAINTraq Viewer y MAINTraq Analyzer para realizar análisis de vibraciones y graficar formas de onda, espectros, órbitas, diagrama de Bode, diagrama polar, cascada de espectro vs RPM, posición de ejes, cepstrum y análisis cíclicos (IDM, 2019). Al ser una herramienta para la adquisición de señales dinámicas, y al estar disponible en el LAEV, este equipo se emplea en el presente proyecto en la adquisición de datos de aceleración. Una desventaja de este equipo es que no dispone de canales de salida para usar estos resultados en otros equipos. En la Figura 1.10 se muestra el equipo de adquisición de señales dinámicas disponible en el LAEV.

Figura 1.10. Equipo de adquisición de señales dinámicas del LAEV.
(Fuente: IDM, 2019)

1.4.2. Analizador de vibraciones Fluke 810

El analizador de vibraciones Fluke 810, mostrado en la Figura 1.11, es una herramienta para solucionar problemas avanzados de mantenimiento mecánico industrial, ya que cuenta con tecnología de diagnóstico para identificar y localizar rápidamente los problemas mecánicos más comunes, como son los cojinetes, alineación incorrecta, desequilibrio, holgura, etc. Este equipo es capaz de medir vibraciones, realizar diagramas espectrales, proporcionar una escala de gravedad, generar informes y brindar recomendaciones (FLUKE, 2019).

Figura 1.11. Analizador de vibraciones Fluke 810.
(Fuente: FLUKE, 2019)

1.4.3. Medidor de vibraciones Lutron VB-8213

El equipo Lutron VB-8213, mostrado en la Figura 1.12, es una herramienta con aplicación en el monitoreo de vibración industrial, ya que permite determinar problemas como desbalanceo, desequilibrio y desalineamiento, mediante las mediciones de desplazamiento, velocidad, aceleración, valores de desviación estándar (RMS) y valor pico.

Además, este equipo puede trabajar con el software Data Logger para la adquisición y procesamiento de datos (Valiometro, 2019).

Figura 1.12. Medidor de vibraciones Lutron VB-8213.
(Fuente: Valiometro, 2019)

Como se observa, los equipos anteriores han sido fabricados para medir vibraciones, más no para excitar estructuras o identificar formas modales de estructuras. Para el OMA, no es necesario excitar estructuras (generalmente realizado mediante un excitador), pero si se necesita identificar las formas modales. Por esta razón es necesario utilizar las señales registradas por el equipo de adquisición de señales dinámicas, procesarlas e ingresarlas a OpenModal.

2. METODOLOGÍA

La metodología que se propone en este proyecto se encuentra detallada en la Figura 2.1, donde se observan los pasos a seguir para determinar los tres primeros modos de vibración de una viga en condiciones de operación. Este trabajo se limita a este número de modos de vibración por dos razones: la primera es que el número de acelerómetros que dispone el LAEV es limitado, y segundo, los tres primeros modos de vibración son suficientes con el fin de probar la metodología propuesta. La viga doblemente empotrada se trata de un perfil estructural tipo U de material Acero ASTM A36.

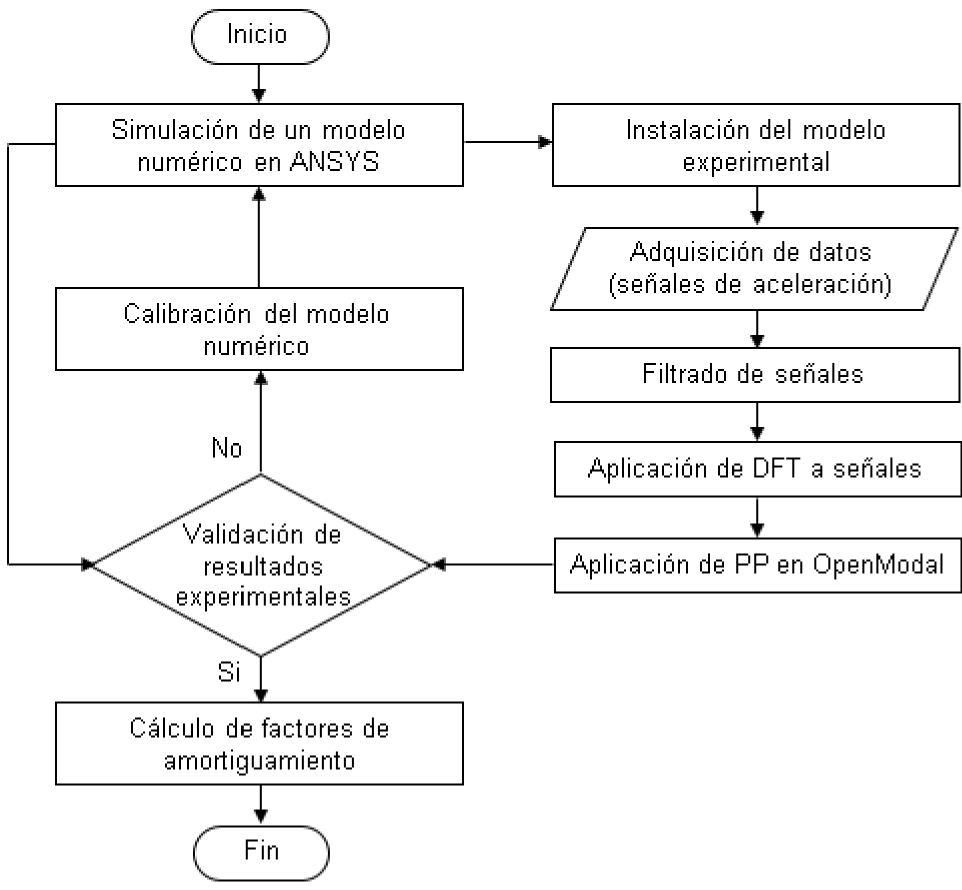


Figura 2.1. Metodología para determinar los modos de vibración.
(Fuente: Propia)

El proceso inicia con la simulación de un modelo numérico en el software ANSYS para conocer los modos de vibración que se esperan obtener experimentalmente. A continuación, se instala el modelo experimental y se realizan las mediciones de las señales de vibración empleando acelerómetros y el equipo de adquisición de señales dinámicas del LAEV. Posteriormente, se realiza el procesamiento de las señales en Python, que consiste en filtrar y aplicar la DFT a las mismas, esto con el objetivo de adecuar las señales para la aplicación de la técnica PP en el software OpenModal.

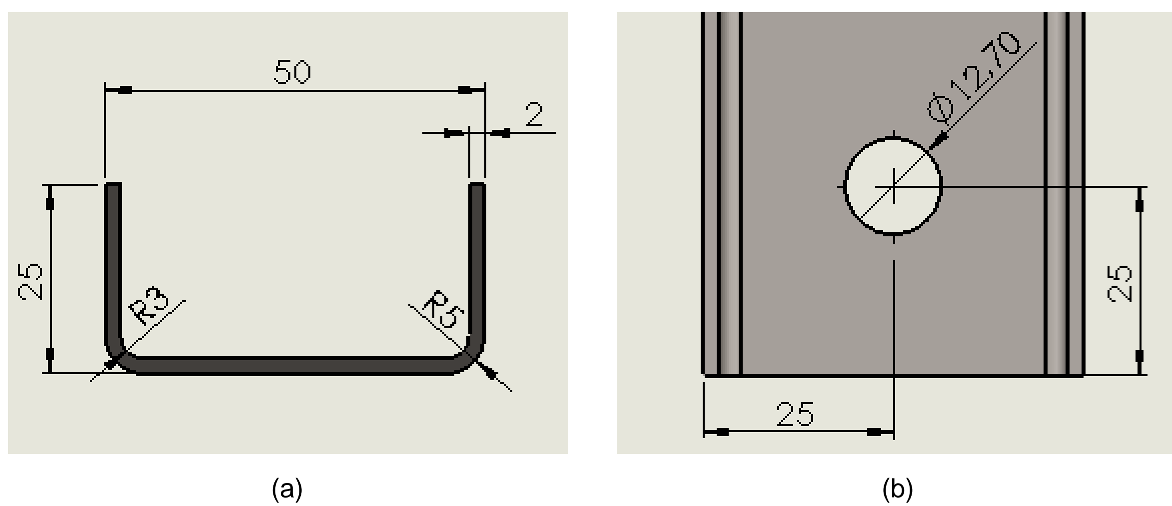
Para la validación del modelo experimental, se comparan los errores relativos entre las frecuencias naturales obtenidas experimentalmente y las obtenidas de la simulación del modelo numérico. En el presente proyecto se asume un error relativo máximo del 5%, en el caso de superar este límite entre ambos resultados, es necesario calibrar el modelo numérico (condiciones de borde), para replicar exactamente el modelo experimental. Finalmente, se calculan los factores de amortiguamiento mediante el método del ancho de banda de media potencia.

2.1. Modelo numérico

La simulación de un modelo numérico mediante el análisis de elementos finitos (AEF), permite determinar las frecuencias naturales y las formas modales de la viga, estos resultados se emplean posteriormente para validar los modos de vibración que van a ser determinados experimentalmente. Este proceso consta de dos etapas que se detallan a continuación.

2.1.1. Diseño del modelo geométrico

El diseño de la viga se realiza en un software CAD, en este proyecto se emplea el software SolidWorks 2017, ya que es una herramienta ingenieril para modelado mecánico en 2D y 3D. En la Figura 2.2 se muestra el modelo geométrico de la viga de 3 m de longitud, la cual posee un agujero en cada extremo para su sujeción durante el ensayo experimental.



(a) (b)

Figura 2.2. Modelo geométrico de la viga en SolidWorks (Dimensiones en mm).
(a) Vista frontal; (b) Vista superior.
(Fuente: Propia)

Se debe guardar el archivo con extensión igs, este paso es necesario para posteriormente realizar la simulación de la viga.

2.1.2. Simulación del modelo numérico en ANSYS

La simulación del modelo numérico se realiza en ANSYS Workbench 18.2, un software de simulación ingenieril que emplea la teoría de AEF. La simulación se realiza empleando el módulo “Modal”, el cual permite determinar las frecuencias naturales y las formas modales de la viga. Inicialmente es necesario configurar las propiedades del material, en este caso del Acero ASTM A36. A continuación, se procede a importar el modelo geométrico de la viga previamente diseñado en SolidWorks. Posteriormente, se configura el mallado y las condiciones de borde de la viga. Por último, se soluciona el modelo numérico, por tanto, se determinan las frecuencias naturales y las formas modales de la viga. Los pasos a seguir para realizar la simulación de la viga se describen de manera más detallada en el Anexo I.

2.2. Modelo experimental

La instalación del modelo experimental debe cumplir ciertos requisitos, como la fuente de excitación, condiciones de los extremos de la viga, número y disposición de acelerómetros. A continuación de describen las condiciones del modelo experimental.

2.2.1. Fuente de excitación

El objetivo principal del OMA es determinar las propiedades dinámicas de estructuras en sus condiciones de operación, sin la necesidad de realizar mediciones de las fuerzas de excitación. Sin embargo, es necesario generar vibraciones en la viga para poder medir la respuesta dinámica mediante acelerómetros. Por lo tanto, como fuente de excitación se emplea un motor desbalanceado, el cual está disponible en el LAEV y se muestra junto con su estructura en la Figura 2.3. Cabe señalar que la frecuencia máxima de operación de dicho motor es de 15 Hz, considerando esta limitante, se realizaron los ensayos experimentales para las siguientes frecuencias del motor: 5,68, 7,88, 9,97, 13,56 y 14,80 Hz, las cuales fueron escogidas arbitrariamente con el objetivo de poder observar cómo afectan a las formas modales obtenidas experimentalmente.

Figura 2.3. Fuente de excitación de la viga.
(Fuente: Propia)

2.2.2. Condiciones del modelo experimental

Se estudia un perfil estructural tipo U con ambos extremos sujetos mediante pernos, como se muestra en la Figura 2.4.

Figura 2.4. Sujeción de los extremos de la viga mediante pernos.
(Fuente: Propia)

Un extremo de la viga se sujeta a la parte inferior de la estructura del motor desbalanceado, como se muestra en la Figura 2.5, mientras que el otro extremo se sujeta a cualquier estructura fija del LAEV.

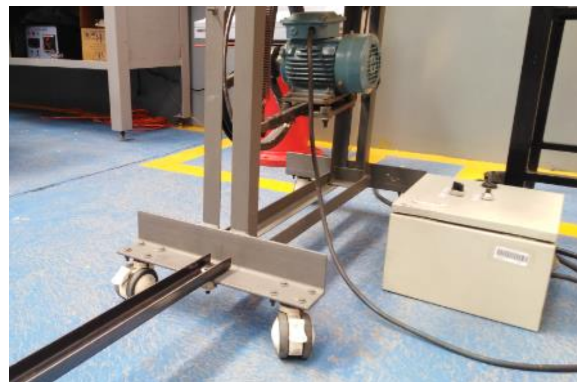


Figura 2.5. Sujeción de la viga a la fuente de excitación.
(Fuente: Propia)

2.2.3. Disposición de acelerómetros

Para medir las respuestas dinámicas se emplean los acelerómetros de montaje magnético que posee el LAEV, los cuales permiten medir desplazamiento, velocidad y aceleración. Para determinar los tres primeros modos de vibración de la viga, es necesario emplear tres acelerómetros, cuya ubicación se muestra en la Tabla 2.1. Cabe señalar que el primer acelerómetro es el más cercano a la fuente de excitación.

Tabla 2.1. Ubicación de los acelerómetros en la viga.

Acelerómetro	Ubicación
1	0,75 m
2	1,50 m
3	2,25 m

(Fuente: Propia)

Además, para realizar las mediciones adecuadamente, es necesario colocar los acelerómetros sobre el eje longitudinal de la viga como se muestra en la Figura 2.6.

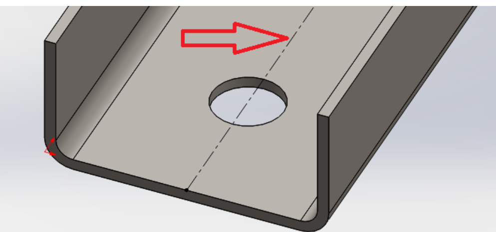


Figura 2.6. Ubicación de los acelerómetros sobre la viga.
(Fuente: Propia)

Cabe señalar que se debe manipular con cuidado los acelerómetros, ya que son delicados y no deben sufrir golpes. La Figura 2.7 muestra la instalación del modelo experimental.

Figura 2.7. Instalación del modelo experimental.
(Fuente: Propia)

2.3. Adquisición de datos

Esta etapa consiste en realizar las mediciones de las señales de vibración mediante los acelerómetros y el equipo de adquisición de señales dinámicas, ambos equipos disponibles en el LAEV.

2.3.1. Equipo de adquisición de señales dinámicas

Este equipo trabaja junto con el software MAINTraq Viewer para visualizar las señales en tiempo real, y con el software MAINTraq Analyzer para analizar datos guardados. En el Anexo II se detalla el procedimiento para la instalación y habilitación de los módulos.

Se configuró el equipo para registrar las mediciones con una frecuencia de muestreo de 375 Hz, frecuencia que permite observar los modos de vibración de interés, como se analiza en el siguiente capítulo. Los acelerómetros son conectados al equipo de adquisición de señales dinámicas, y éste a su vez se conecta al computador mediante cable de red Ethernet. Para una correcta comunicación entre el equipo de adquisición de señales dinámicas y el computador ver Anexo II.

2.3.2. Software MAINTraq Viewer

En primer lugar, se debe configurar la cantidad de acelerómetros a emplearse, los tipos de apoyos, las formas de onda y variables a ser medidas. La configuración empleada en el presente proyecto se muestra en la Figura 2.8.

Configuración del Ensayo

PUNTOS | TRIGGERS | APOYOS | FORMAS DE ONDA | VARIABLES | MEDICIÓN | DATOS

FORMAS DE ONDA: Nueva | Propiedades | Eliminar

Apoyo	Punto	Sensor	Magnitud	Rango de frecuencias
apoyo 1	punto 1	Acelerómetro	Aceleración	1,0Hz - 100,0Hz
apoyo 1	punto 1	Acelerómetro	Velocidad	1,0Hz - 100,0Hz
apoyo 1	punto 1	Acelerómetro	Desplazamiento	1,0Hz - 100,0Hz
apoyo 2	punto 2	Acelerómetro	Aceleración	1,0Hz - 100,0Hz
apoyo 2	punto 2	Acelerómetro	Velocidad	1,0Hz - 100,0Hz
apoyo 2	punto 2	Acelerómetro	Desplazamiento	1,0Hz - 100,0Hz
apoyo 3	punto 3	Acelerómetro	Aceleración	1,0Hz - 100,0Hz
apoyo 3	punto 3	Acelerómetro	Velocidad	1,0Hz - 100,0Hz
apoyo 3	punto 3	Acelerómetro	Desplazamiento	1,0Hz - 100,0Hz

Figura 2.8. Configuración del ensayo.
(Fuente: Propia)

Una vez realizada la configuración del ensayo, es posible ver en tiempo real los valores de las variables, así como visualizar las formas de onda de las señales de respuestas dinámicas. Por último, se deben grabar las formas de onda para su posterior análisis en el software MAINTraq Analyzer. Para obtener buenos resultados se determinó que es necesario grabar las señales un tiempo mínimo de un minuto para garantizar la estacionariedad de las señales, lo que significa que, en procesos estocásticos los parámetros estadísticos de la señal no cambian en el tiempo (Rincón, 2011).

2.3.3. Software MAINTraq Analyzer

Este software permite analizar las señales de vibración previamente grabadas, se pueden visualizar las formas de onda en el dominio del tiempo, y sus componentes en dominio de frecuencia.

Este módulo permite además exportar tanto las variables como las formas de onda, sin embargo, solamente se deben seleccionar las formas de onda de aceleración de los tres acelerómetros como se muestra en la Figura 2.9.

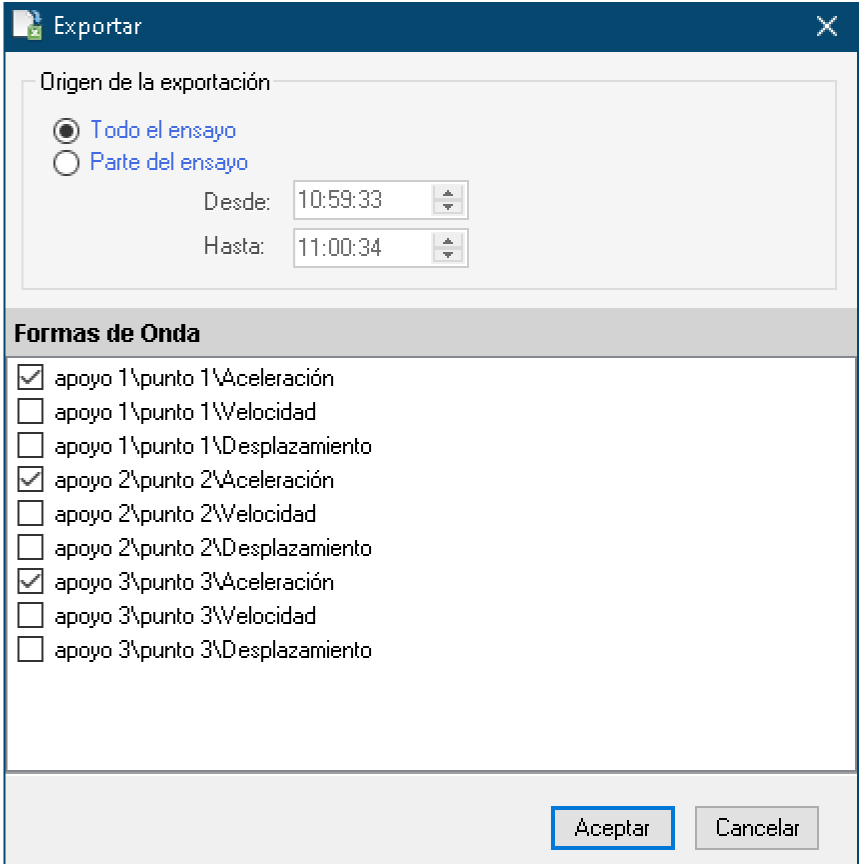


Figura 2.9. Exportación de formas de onda de aceleración de los tres acelerómetros.
(Fuente: Propia)

Por último, se deben exportar las formas de onda en formato delimitado por comas (CSV) para su posterior procesamiento en Python.

2.4. Procesamiento de datos

Para el procesamiento de las señales de aceleración se utiliza el lenguaje de programación Python versión 3.7.3, ya que este lenguaje se desarrolla bajo una licencia de código abierto. Este lenguaje de programación se puede utilizar a través de la plataforma SPYDER (Scientific Python Development Environment), el cual es un entorno científico escrito en Python, diseñado por y para científicos, ingenieros y analistas de datos (SPYDER, 2018).

Para procesar las señales de aceleración se desarrolla un código en Python, el cual se detalla en el Anexo III. En primer lugar, se lee la señal de entrada, es decir, el archivo en formato CSV que contenga las señales de aceleración. Para leer el archivo se emplea la función "pd.read_csv" disponible en la librería Pandas.

A continuación, es necesario filtrar las señales de aceleración con el objetivo de eliminar el ruido y otras interferencias de las señales. Para ello, se diseña un filtro paso bajo tipo Butterworth de tercer orden empleando la función "scipy.signal.butter", disponible en la librería SciPy. Para el diseño del filtro se aplica el Teorema de muestreo de Nyquist-Shannon, el cual afirma que la frecuencia de muestreo debe ser al menos dos veces más alta que la frecuencia máxima a analizar (Olshausen, 2000), como se expresa en la Ecuación 2.1.

$$f_s \geq 2f_{max} \qquad \text{(Ec. 2.1.)}$$

Donde:

f_s : Frecuencia de muestreo

f_{max} : Frecuencia máxima a analizar

Mediante la simulación del modelo numérico en ANSYS se determinó que el primer modo de vibración se produce a una frecuencia de 11,62 Hz, mientras que el tercer modo de vibración se produce a una frecuencia de 68,974 Hz, por lo tanto, con el fin de reconstruir correctamente las señales filtradas a partir de sus muestras, el teorema de Nyquist-Shannon se aplica a la frecuencia natural más alta, es decir, la correspondiente al tercer modo de vibración, como se muestra a continuación:

$$f_s \geq 2 * 68{,}974\ [Hz] \geq 137{,}948\ [Hz]$$

Como se observa, la frecuencia de muestreo debe ser mayor a 137,948 Hz, por lo tanto, se diseña el filtro con la misma frecuencia de muestreo de los acelerómetros, es decir, 375 Hz, frecuencia con la cual se obtuvo buenos resultados como se analiza en el siguiente capítulo.

A partir de la frecuencia de muestreo (f_s), es posible determinar la frecuencia de Nyquist (f_N), que se define como la frecuencia máxima que puede estar presente en la señal analógica sin que se produzca aliasing (solapamiento o reflejo de una señal). Esta frecuencia corresponde a la frecuencia de corte del filtro, y se determina mediante la Ecuación 2.2.

$$f_N = 0{,}5 * f_s \qquad \text{(Ec. 2.2.)}$$

$$f_N = 0{,}5 * 375\ [Hz] = 187{,}5\ [Hz]$$

Por lo tanto, se diseña el filtro paso bajo tipo Butterworth de tercer orden con una frecuencia de corte de 187,5 Hz. Una vez filtradas las señales de aceleración, se calcula la DFT de las mismas, esto se logra mediante la función “np.fft.fft”, disponible en la librería NumPy.

Para la aplicación de la técnica PP se emplea el software OpenModal, sin embargo, este software no permite importar archivos de datos en formato CSV. OpenModal trabaja con archivos de datos en formato UFF (Universal File Format), originalmente desarrollado por Structural Dynamics Research Corporation (SDRC), con el objetivo de estandarizar la transferencia de datos dinámicos experimentales entre la comunidad científica (SDRL, 2020). Por lo tanto, se procede a generar un archivo de datos en formato UFF mediante el empleo del módulo “pyuff” para OpenModal. Este módulo es de libre acceso, puede ser descargado de la plataforma GitHub y debe ser instalado en SPYDER, en el Anexo IV se detalla el código “pyuff”.

Cabe señalar que una vez ejecutado en SPYDER el código detallado en el Anexo III, automáticamente se crea un archivo de datos en formato UFF. Además, el código se debe ejecutar una sola vez, ya que, si se ejecuta varias veces, automáticamente se crean varios datos de mediciones en un mismo archivo.

2.5. Análisis modal en OpenModal

Como se mencionó en la sección 1.2.3, OpenModal permite realizar EMA y OMA, sin embargo, el presente proyecto se enfoca en realizar OMA, ya que no se conocen las fuerzas de excitación y solamente se realizan mediciones de las respuestas dinámicas, mediciones que corresponden a los datos de entrada en OpenModal.

Para trabajar en el software OpenModal es necesario ejecutar en SPYDER el código que se detalla en el Anexo V. A continuación, se procede a importar el archivo de datos en formato UFF.

El procedimiento para determinar los modos de vibración en OpenModal consiste en cuatro módulos ubicados en la parte superior de su interfaz gráfica. En los siguientes apartados se describe cada módulo de OpenModal.

2.5.1. Módulo "Geometry"

Una vez importado el archivo de datos en formato UFF se procede a crear el modelo discreto del elemento a analizar. En primer lugar, se ingresan las coordenadas de los nodos de la viga en la sección "Add nodes", como se muestra en la Figura 2.10. Según el modelo experimental se tienen cinco nodos, tres que corresponden a los acelerómetros y dos correspondientes a cada extremo de la viga.

Add nodes | Add lines | Add triangles

node nums	x	y	z	x to y	x to z	y to z	model ID
1	0.0	0.0	0.0	0.0	0.0	0.0	1
2	0.0	0.75	0.0	0.0	0.0	0.0	1
3	0.0	1.5	0.0	0.0	0.0	0.0	1
4	0.0	2.25	0.0	0.0	0.0	0.0	1
5	0.0	3.0	0.0	0.0	0.0	0.0	1

Figura 2.10. Configuración de los nodos de la viga en OpenModal.
(Fuente: Propia)

Por último, en la sección "Add lines" se deben unir los nodos con líneas, como se muestra en la Figura 2.11. Las líneas corresponden al modelo numérico de la viga, mientras que los nodos permitirán observar las formas modales.

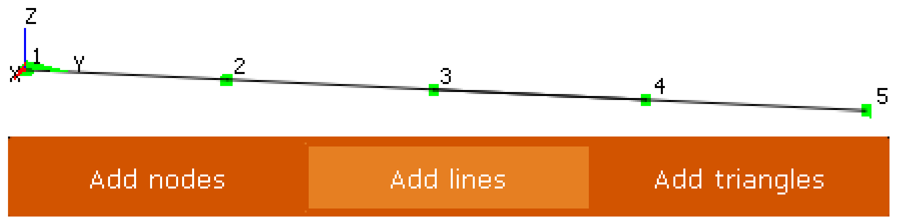


Figura 2.11. Unión de nodos de la viga en OpenModal.
(Fuente: Propia)

2.5.2. Módulo “Measurement”

Como se observa en la Figura 2.12 se deben configurar ciertas condiciones, “rsp dir” y ”ref dir” hacen referencia a la dirección de la respuesta dinámica, en este caso, el eje z. Además, “rsp node” hace referencia a la respuesta de los nodos, en este caso las respuestas dinámicas medidas por los acelerómetros corresponden a los nodos 2, 3 y 4, ya que los nodos 1 y 5 corresponden a los extremos fijos de la viga. Finalmente, en la columna “ref node” se debe ingresar cualquier nodo de los acelerómetros como referencia para el análisis modal.

rsp node	rsp dir	ref node	ref dir
2	z	2	z
3	z	2	z
4	z	2	z

Figura 2.12. Configuración de la dirección de la respuesta dinámica.
(Fuente: Propia)

Además, este módulo permite visualizar la PSD de las aceleraciones, así como sus gráficas de desfase en función de la frecuencia. Cabe mencionar que OpenModal no permite aplicar la técnica PP en la PSD, la selección de picos se realiza sobre la FRF como se explica en el siguiente apartado.

2.5.3. Módulo “Analysis”

En este módulo se visualizan las FRFs de cada acelerómetro y se configura el rango de frecuencias a analizar. Se determinó en ANSYS que la frecuencia del tercer modo de vibración es de 68,974 Hz, por lo tanto, para una mejor visualización es posible configurar la frecuencia máxima a analizar en 80 Hz y se procede a realizar el análisis modal presionando el ícono “Analyse”, como se observa en la Figura 2.13.

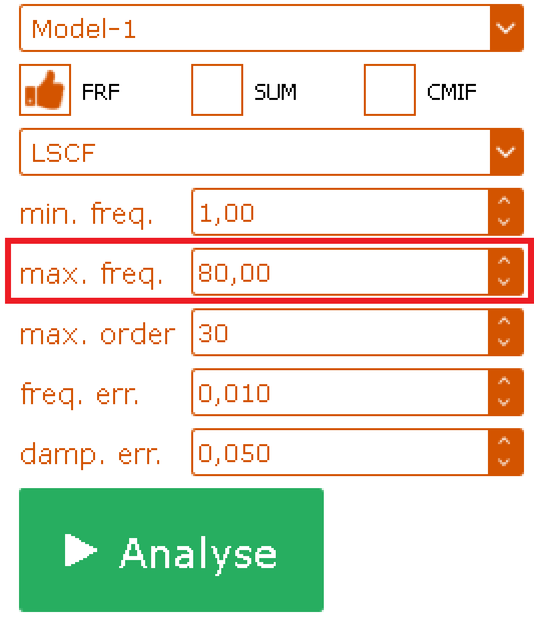


Figura 2.13. Configuración de rango de frecuencias a analizar en OpenModal.
(Fuente: Propia)

Posteriormente, se aplica la técnica PP, para lo cual el usuario debe seleccionar manualmente los picos de la FRF, como se observa en la Figura 2.14. Los signos rojos seleccionados corresponden a las frecuencias naturales de la viga. Por último, este módulo también muestra los factores de amortiguamiento de los picos seleccionados. Los factores de amortiguamiento que se obtienen dependen de la selección de los picos, es decir, al seleccionar picos altos se tiene bajo amortiguamiento, mientras que al seleccionar en un nivel más bajo se obtiene mayor amortiguamiento.

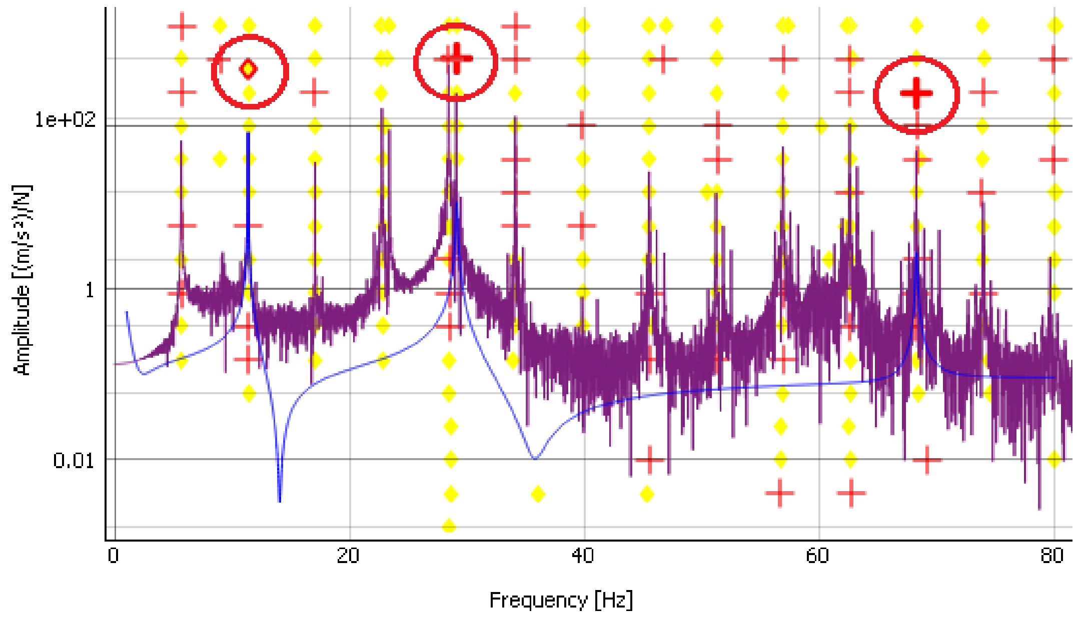
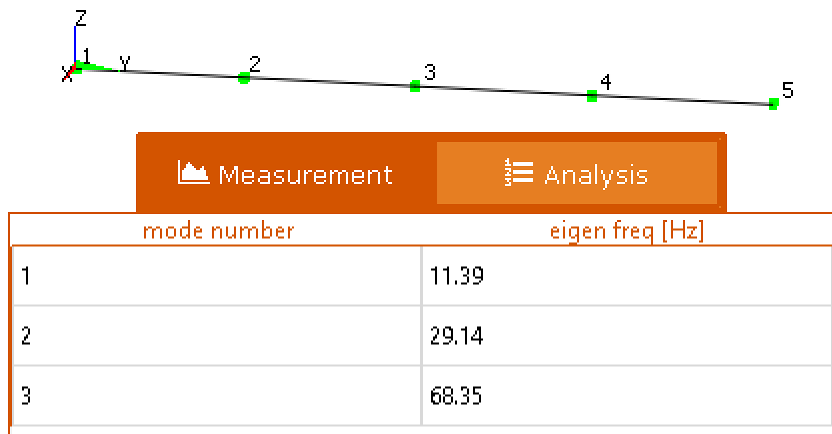


Figura 2.14. Técnica PP en la FRF en OpenModal.
(Fuente: Propia)

Cabe señalar que no se seleccionaron todos los picos de la FRF, debido a que los picos no seleccionados corresponden a modos torsionales de vibración, como se explica en la sección 3.2, cuyo análisis no se encuentra dentro del alcance del presente proyecto.

2.5.4. Módulo "Animation"

En la sección "Analysis" de este módulo, se muestran las frecuencias naturales de los picos seleccionados de la FRF, como se muestra en la Figura 2.15. Finalmente, es posible reproducir una animación de la forma modal correspondiente a la frecuencia natural seleccionada.



mode number	eigen freq [Hz]
1	11.39
2	29.14
3	68.35

Figura 2.15 Frecuencias naturales obtenidas en OpenModal.
(Fuente: Propia)

2.6. Validación de resultados experimentales

Como proceso de validación, se compara el error relativo entre las frecuencias naturales obtenidas experimentalmente y las obtenidas de la simulación en ANSYS. La Ecuación 2.3 expresa el error relativo absoluto entre los datos experimentales y simulados.

$$\%error\ relativo = \left|\frac{dato\ experimental - dato\ simulado}{dato\ experimental}\right| * 100 \qquad \text{(Ec. 2.3.)}$$

Un error relativo menor al 5% indica que los resultados experimentales son válidos, por el contrario, un error relativo mayor al 5% indica que el modelo numérico no es capaz de replicar con exactitud el modelo experimental, por tanto, es necesario calibrar el modelo numérico (condiciones de borde).

2.7. Cálculo del factor de amortiguamiento

Pese a que OpenModal proporciona los factores de amortiguamiento, éstos valores no son confiables y están sujetos a errores, como se menciona en la sección 1.3.1. Determinar el factor de amortiguamiento de un sistema estructural es complicado, sin embargo, existen teorías que permiten calcular el amortiguamiento de manera aproximada, una de ellas es el método del ancho de banda de media potencia. Como afirma Chopra (2014), este método es posible aplicar en el espectro de las componentes en dominio de frecuencia de la aceleración, cuando los modos de vibración no se encuentran superpuestos (juntos). Este método se debe aplicar en cada pico que corresponde a una frecuencia natural (f_n), como se muestra en la Figura 2.16.

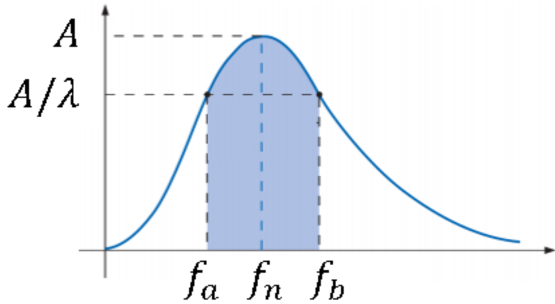


Figura 2.16. Método del ancho de banda de media potencia.
(Fuente: Propia)

Se encuentra el valor máximo de amplitud (A) y se divide por λ, generalmente se considera $\lambda = \sqrt{2}$. Se determinan las frecuencias de media potencia (f_a, f_b) que corresponden al valor A/λ, y mediante la Ecuación 2.4 se determina el factor de amortiguamiento (ξ).

$$\xi = \frac{f_b - f_a}{2f_n} \qquad \text{(Ec. 2.4.)}$$

3. RESULTADOS Y DISCUSIÓN

En este capítulo se detallan los resultados obtenidos después de haber desarrollado un procedimiento que emplea la técnica del OMA. En primer lugar, se analizan los modos de vibración obtenidos mediante la simulación de un modelo numérico en ANSYS. A continuación, se analiza el proceso de filtrado y el cálculo de la DFT de las señales de aceleración. Posteriormente, se analizan los modos de vibración obtenidos en OpenModal, se validan estos resultados experimentales al comparar con los resultados obtenidos de la simulación y de la solución analítica. Finalmente, se analiza el factor de amortiguamiento calculado mediante el método del ancho de banda de media potencia.

3.1. Modos de vibración en ANSYS

El modelo experimental consiste en una viga con sus ambos extremos sujetos mediante pernos a dos estructuras soporte del LAEV, esta configuración representa una condición de borde intermedia entre una articulación y un empotramiento, llamada parcialmente empotrada. Es por este motivo que se realizaron en ANSYS las simulaciones de tres casos distintos: el primero corresponde a la viga con las superficies internas de los agujeros fijas, el segundo corresponde a la viga con ambos extremos fijos (empotramientos), mientras que el último caso corresponde a un modelo numérico con un sistema de fijación igual al del modelo experimental, es decir, consta de pernos, tuercas, arandelas y los perfiles estructurales a los que se sujetó la viga. En la Tabla 3.1 se muestran las frecuencias naturales de los modos flexionantes de vibración para los 3 casos descritos anteriormente, y además se muestran las frecuencias naturales calculadas analíticamente empleando la Ecuación 1.3 y la Tabla 1.1, para el caso de viga articulada y empotrada.

Tabla 3.1. Frecuencias naturales de modos flexionantes de vibración obtenidos en ANSYS y analíticamente para diferentes casos.

	ANSYS			Solución analítica	
	Frecuencia natural [Hz]			Frecuencia natural [Hz]	
Modo	Caso 1	Caso 2	Caso 3	Articulada	Empotrada
1	10,604	15,534	11,62	6,758	15,321
2	31,72	42,764	29,928	27,034	42,232
3	68,441	83,679	68,974	60,826	82,791

(Fuente: Propia)

En el primer caso simulado, al fijar las superficies internas de los agujeros de la viga, se obtuvo frecuencias naturales mayores a las calculadas analíticamente para una viga con sus extremos articulados. Con respecto al segundo caso simulado, al fijar las superficies de los extremos de la viga, se obtuvo frecuencias naturales muy cercanas a las calculadas analíticamente para el caso de una viga con sus extremos empotrados. Por último, se puede observar que los resultados obtenidos de la simulación del tercer caso, se encuentran en un nivel intermedio entre los resultados analíticos para una viga articulada y empotrada, lo cual se esperaba debido a las condiciones de borde del modelo numérico.

A continuación, se muestran los resultados obtenidos para el último caso, ya que se asemeja más al modelo experimental y, por tanto, proporciona resultados más exactos. Se obtuvo los seis modos de vibración de la viga, entre los cuales incluyen modos flexionantes y torsionales, sin embargo, el presente proyecto se limita solo al análisis de los modos flexionantes. En el Anexo VI se detallan los seis modos de vibración obtenidos en ANSYS. Las frecuencias naturales determinadas en ANSYS se emplean en la sección 3.4 para verificar los resultados obtenidos experimentalmente en OpenModal.

En la Figura 3.1 se muestra el primer modo de vibración de la viga, el cual se produce a una frecuencia natural de 11,62 Hz. La mayor deflexión se presenta en la mitad de la viga, es decir, a los 1,50 m.

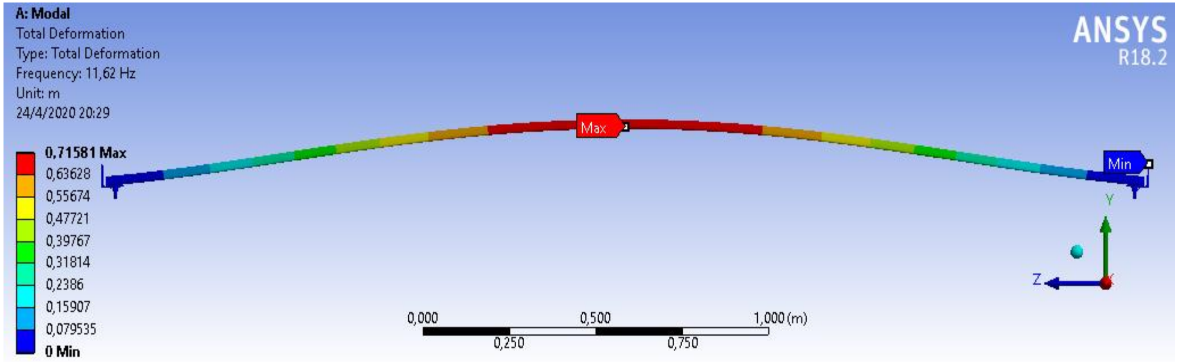


Figura 3.1. Primer modo de vibración de la viga en ANSYS.
(Fuente: Propia)

En la Figura 3.2 se muestra el segundo modo de vibración de la viga, el cual se produce a una frecuencia natural de 29,928 Hz. Las mayores deflexiones se presentan aproximadamente a los 0,77 m y 2,23 m, mientras que a los 1,50 m las deflexiones son nulas.

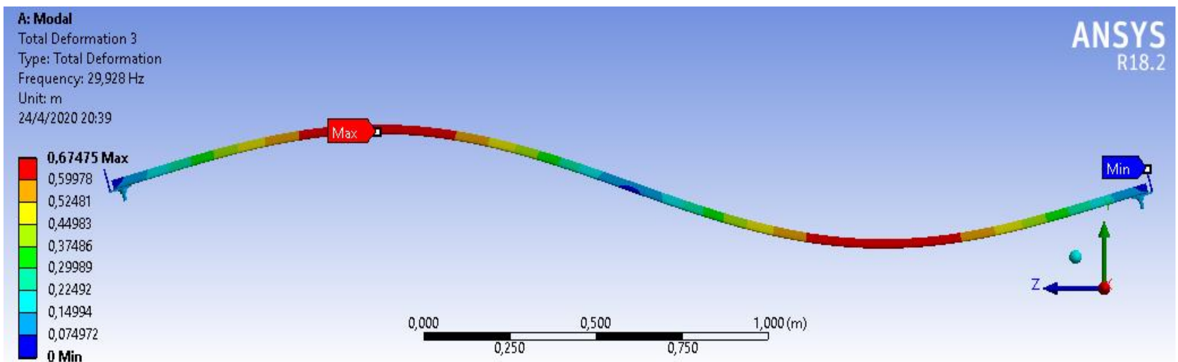


Figura 3.2. Segundo modo de vibración de la viga en ANSYS.
(Fuente: Propia)

En la Figura 3.3 se muestra el tercer modo de vibración de la viga, el cual se produce a una frecuencia natural de 68,974 Hz. Las mayores deflexiones se presentan aproximadamente a los 0,55 m, 1,50 m y 2,45 m.

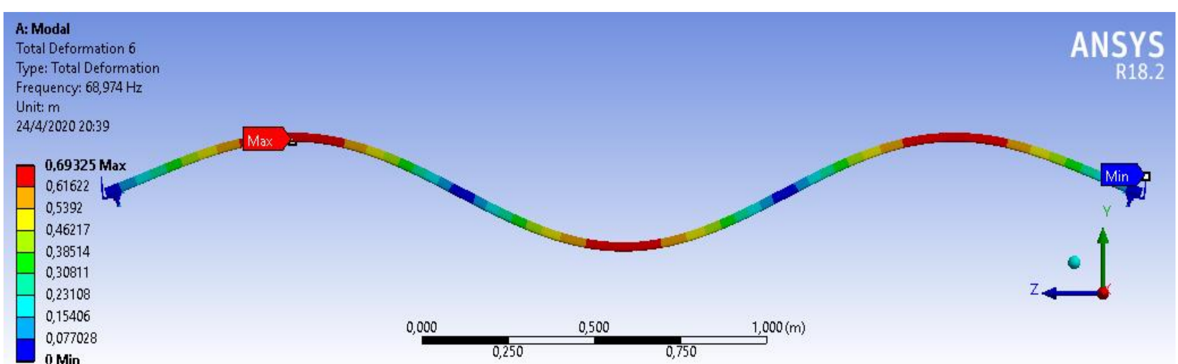


Figura 3.3. Tercer modo de vibración de la viga en ANSYS.
(Fuente: Propia)

3.2. Filtrado de las señales de aceleración

Se desarrolló un código en Python para el procesamiento de las señales de aceleración, ver Anexo III. Los datos de entrada corresponden a las señales de aceleración que fueron exportadas mediante el software MAINTraq Analyzer. Cada acelerómetro realizó las mediciones a una frecuencia de muestreo de 375 Hz, obteniéndose 24000 datos de aceleración por cada acelerómetro, durante aproximadamente el minuto que se realizó cada ensayo experimental. Se realizaron varios ensayos experimentales, con un barrido de frecuencias de operación del motor (5,68, 7,88, 9,97, 13,56 y 14,80 Hz), por lo que, en la Tabla 3.2 se muestran las frecuencias naturales de modos flexionantes de vibración, obtenidos en ANSYS y los obtenidos en MAINTraq Analyzer para cada frecuencia de excitación de los ensayos experimentales.

Tabla 3.2. Frecuencias naturales obtenidas en ANSYS y en MAINTraq Analyzer para diferentes frecuencias de excitación.

		MAINTraq Analyzer				
Modo	ANSYS [Hz]	5,68 [Hz]	7,88 [Hz]	9,97 [Hz]	13,56 [Hz]	14,80 [Hz]
1	**11,62**	**11,40**	-	-	-	-
2	**29,928**	**28,47**	31,95	29,66	27,37	29,71
3	**68,974**	**68,30**	63,90	69,20	68,39	74,25

(Fuente: Propia)

En la Tabla 3.2 se observan resaltadas las frecuencias naturales de los modos flexionantes de vibración obtenidos en ANSYS (11,62, 29,928 y 68,974 Hz), los cuales corresponden a la simulación del tercer caso. Se puede observar que no se obtienen los mismos resultados para las distintas frecuencias de excitación, se verifica que los mejores resultados se obtienen con una frecuencia de excitación de 5,68 Hz, cuyos resultados se analizan a continuación. Por otra parte, no se puede observar el primer modo de vibración con las demás frecuencias de excitación, esto se debe a que las condiciones de experimentación no son ideales, ya que de cierta manera influyen las vibraciones de los laboratorios cercanos al LAEV.

El archivo que contiene las señales de aceleración se encuentra en formato CSV, por lo tanto, para poder leer estos datos en Python se empleó la función "pd.read_csv" disponible en la librería Pandas. A continuación, se realizó el diseño del filtro paso bajo tipo Butterworth de tercer orden, con el objetivo de eliminar el ruido y otras interferencias de las señales de aceleración. El diseño del filtro se realizó empleando la función "scipy.signal.butter" disponible en la librería SciPy. Además, aplicando el teorema de muestreo de Nyquist-Shannon, se determinó que la frecuencia de muestreo del filtro es de 375 Hz y, la frecuencia de corte necesaria para evitar el fenómeno aliasing es de 187,5 Hz. En la Figura 3.4 se muestra la respuesta en dominio de frecuencia del filtro paso bajo tipo Butterworth de tercer orden, donde se puede observar la banda de paso definida entre 0 y 187,5 Hz.

Una vez diseñado el filtro, se aplicó a las señales de aceleración. En las Figuras 3.5, 3.6 y 3.7 se muestran las señales originales y filtradas de los 3 acelerómetros respectivamente. En las 3 figuras, graficadas en un intervalo de tiempo entre 0 y 1 segundo, se observa que no existe una gran diferencia entre las señales originales y filtradas, lo cual indica una correcta reconstrucción de la señal filtrada y la existencia de un nivel muy bajo de ruido.

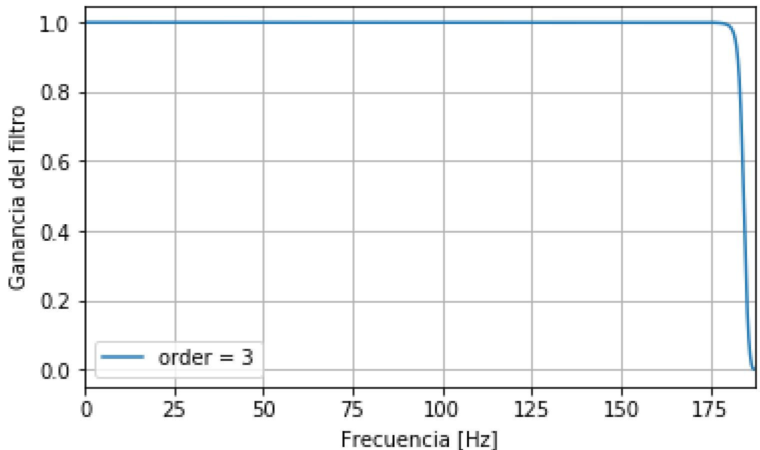


Figura 3.4. Respuesta en frecuencia del filtro Butterworth paso bajo de tercer orden.
(Fuente: Propia)

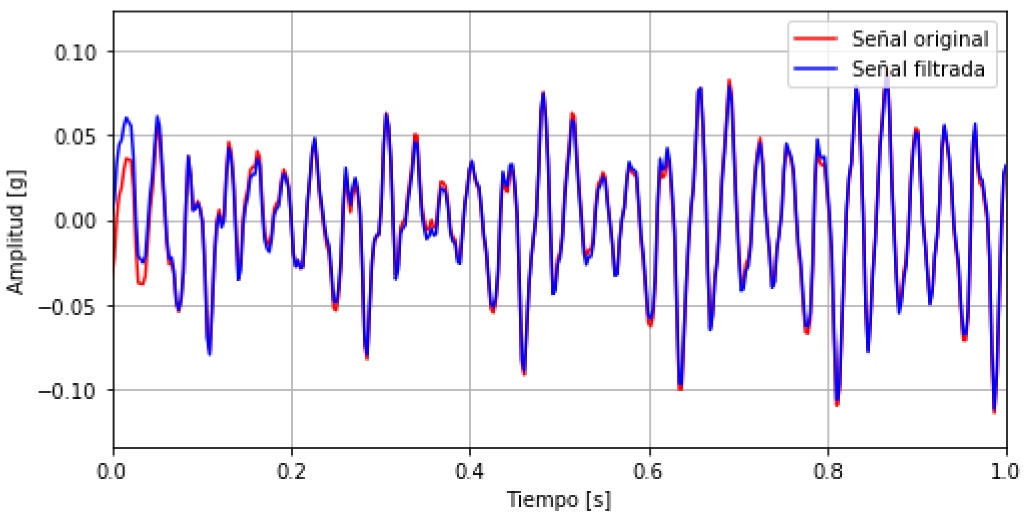


Figura 3.5. Señal original y filtrada del acelerómetro 1.
(Fuente: Propia)

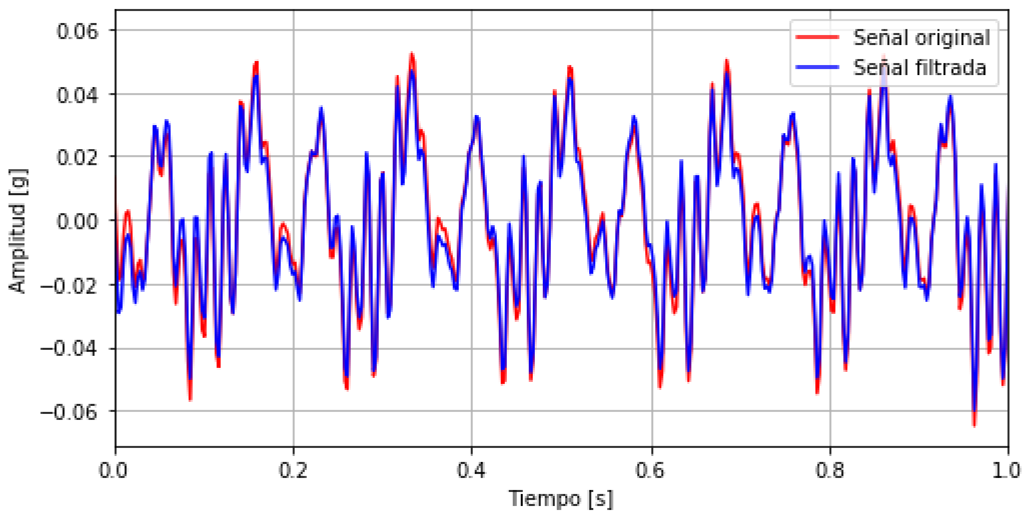


Figura 3.6. Señal original y filtrada del acelerómetro 2.
(Fuente: Propia)

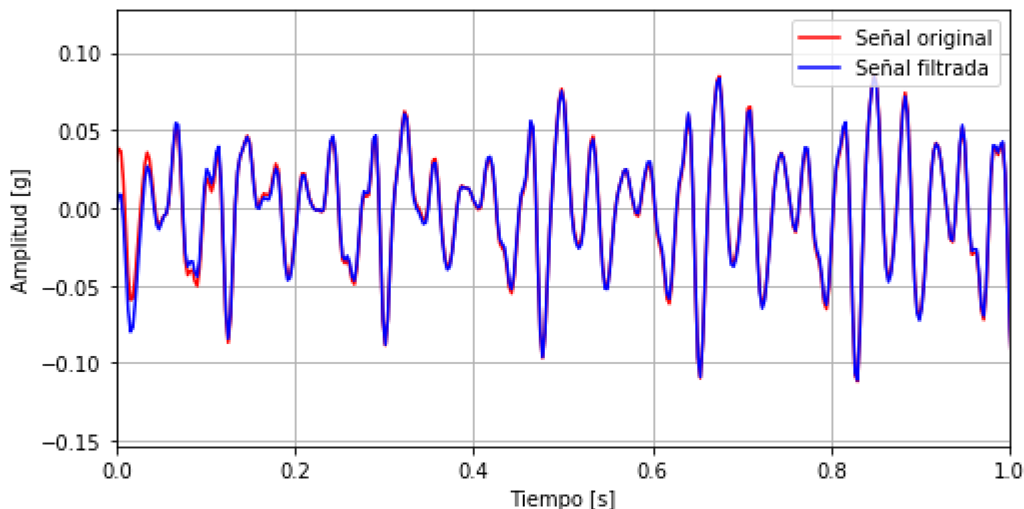
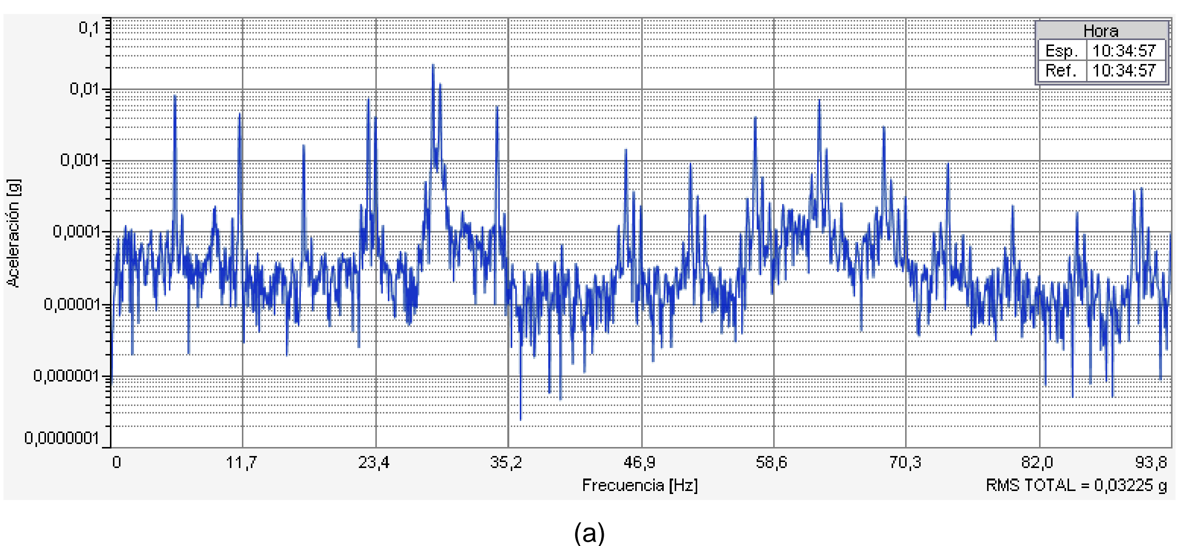


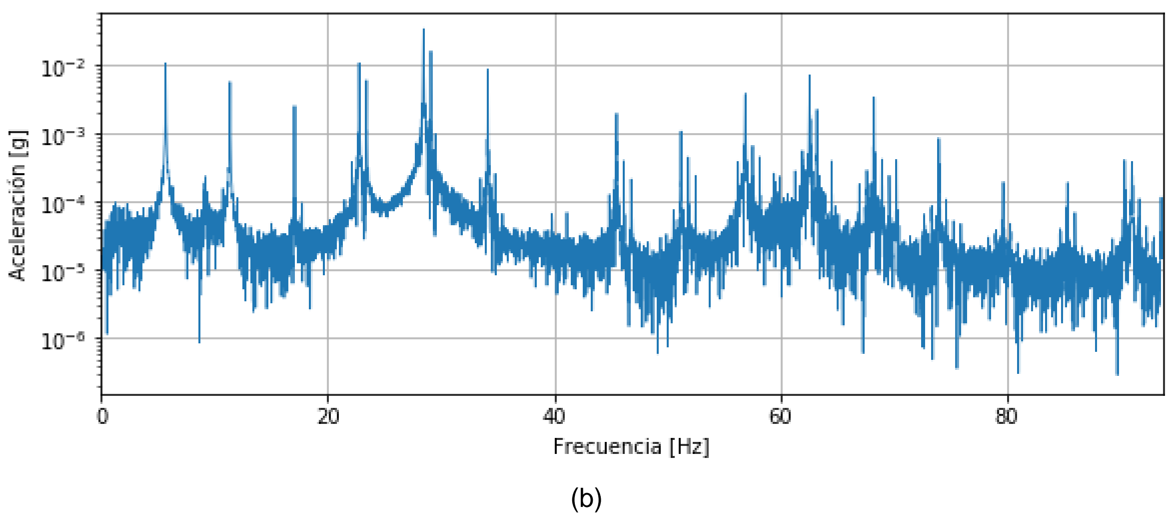
Figura 3.7. Señal original y filtrada del acelerómetro 3.
(Fuente: Propia)

Posteriormente, se calculó la DFT de las señales de aceleración filtradas mediante la función “np.fft.fft” disponible en la librería NumPy. Al realizar este cálculo se obtuvo como resultado las componentes en dominio de frecuencia de las señales, las cuales indican la amplitud de la aceleración en determinada frecuencia. El dominio de la frecuencia de los espectros obtenidos se sitúa en el rango entre 0 y 187,5 Hz.

Las Figuras 3.8, 3.9 y 3.10 muestran las componentes en dominio de frecuencia de las señales obtenidas en MAINTraq Analyzer y en Python. Los espectros han sido graficados en un rango entre 0 y 93.75 Hz para una mejor visualización.

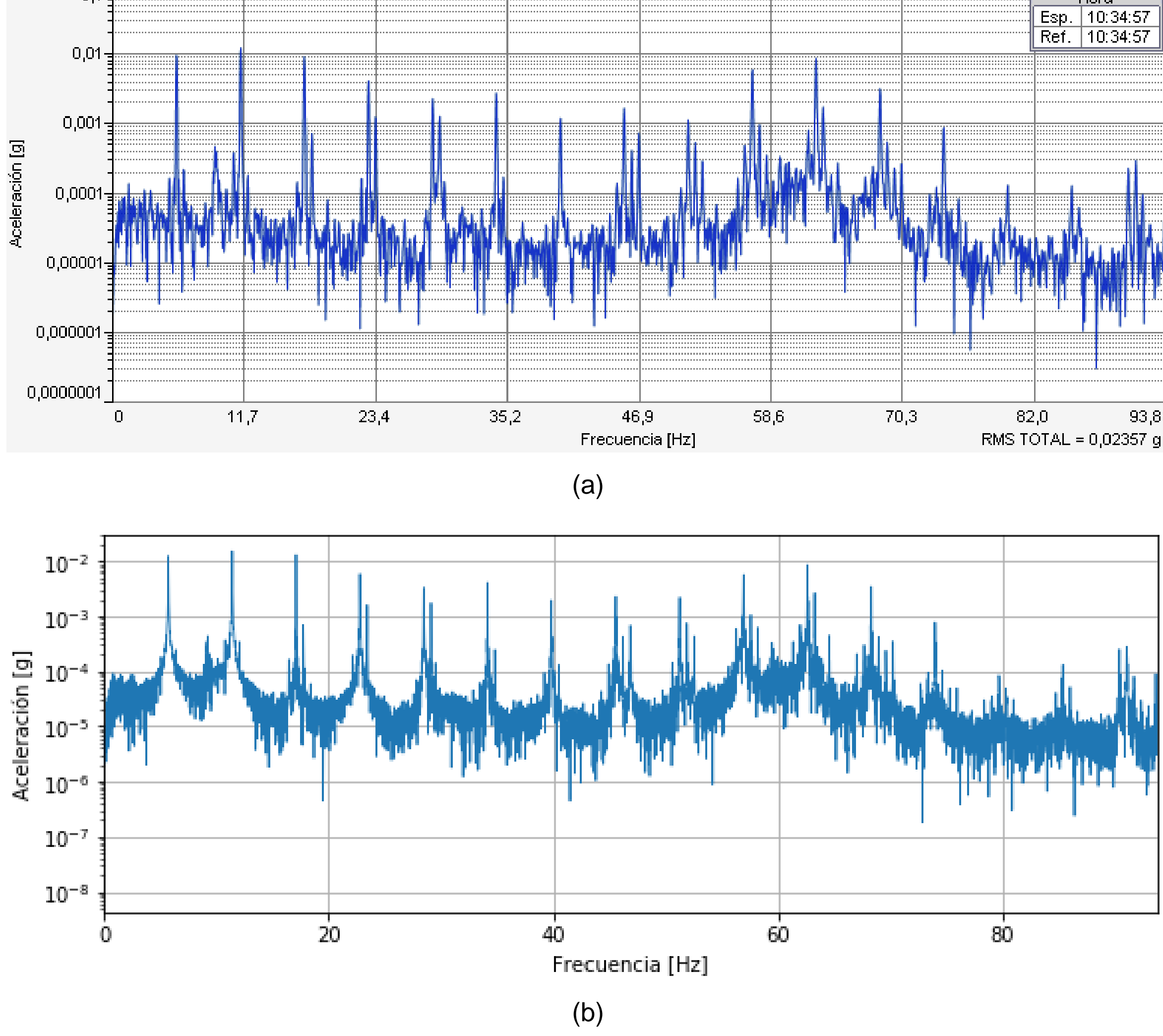


(a)



(b)

Figura 3.8. Componentes en frecuencia de la señal del acelerómetro 1.
(a) MAINTraq Analyzer; (b) Python.
(Fuente: Propia)



(a)

(b)

Figura 3.9. Componentes en frecuencia de la señal del acelerómetro 2.
(a) MAINTraq Analyzer; (b) Python.
(Fuente: Propia)

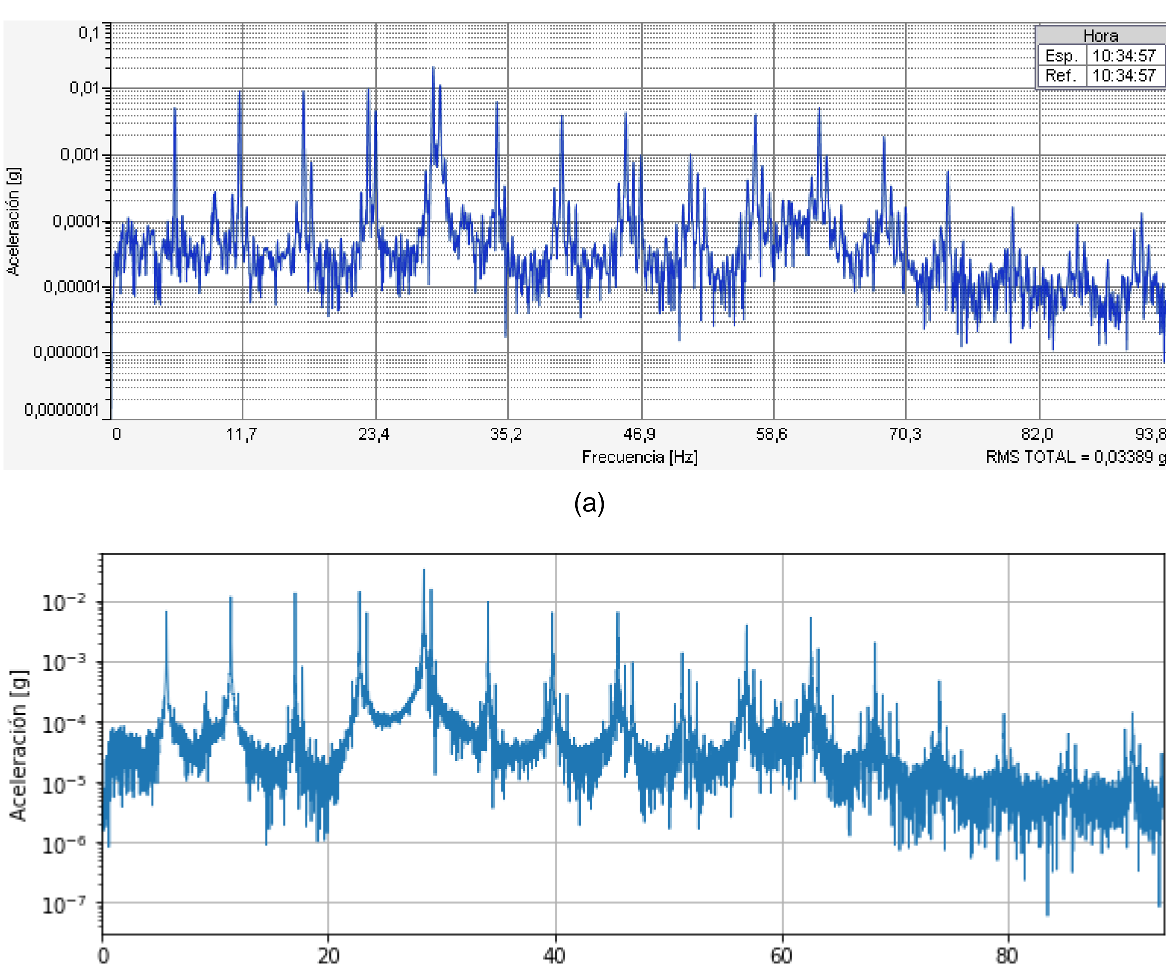


Figura 3.10. Componentes en frecuencia de la señal del acelerómetro 3.
(a) MAINTraq Analyzer; (b) Python.
(Fuente: Propia)

En la Tabla 3.3 se muestran las frecuencias naturales obtenidas mediante la simulación del modelo numérico en ANSYS y las obtenidas en MAINTraq Analyzer. Note que las frecuencias resaltadas corresponden a los modos flexionantes de vibración.

Tabla 3.3. Frecuencias naturales obtenidas en ANSYS y MAINTraq Analyzer.

Modo	ANSYS [Hz]	MAINTraq Analyzer [Hz]
1	**11,62**	**11,40**
2	18,466	17,07
3	**29,928**	**28,47**
4	39,426	39,84
5	51,241	51,22
6	**68,974**	**68,30**

(Fuente: Propia)

Mediante la simulación del modelo numérico en ANSYS se verificó que los 3 primeros modos flexionantes de la viga corresponden a las frecuencias de 11,40, 28,47 y 68,30 Hz, mientras que las demás frecuencias corresponden a modos torsionales de vibración, ver Anexo VI. Por lo tanto, estos modos torsionales son descartados del análisis de este proyecto, ya que como se mencionó anteriormente, este proyecto se limita solamente al análisis de modos flexionantes.

Los espectros de aceleración anteriormente graficados, contienen las componentes tanto de modos flexionantes como torsionales, esto se debe a que los acelerómetros al no estar ubicados al nivel del eje neutro de la viga, cualquier rotación de la estructura produce un desplazamiento horizontal y vertical de los acelerómetros y, por lo tanto, este desplazamiento vertical es captado por los acelerómetros y genera las componentes torsionales en los espectros de aceleración.

Por último, el código en Python detallado en el Anexo III permitió generar un archivo en formato UFF mediante el módulo “pyuff”, cuyo código se detalla en el Anexo IV.

3.3. Modos de vibración en OpenModal

El archivo de datos en formato UFF fue importado en OpenModal, en donde se pudo leer los espectros sin ningún problema. En OpenModal, se determinaron los tres primeros modos de vibración de la viga a partir de los datos procesados de aceleración.

En la Figura 3.11 se muestra la PSD de la señal del acelerómetro 1, con un rango de frecuencias de 0 a 187,5 Hz, y sin la presencia del fenómeno aliasing. Nótese que las unidades de la PSD son $[g^2/Hz]$.

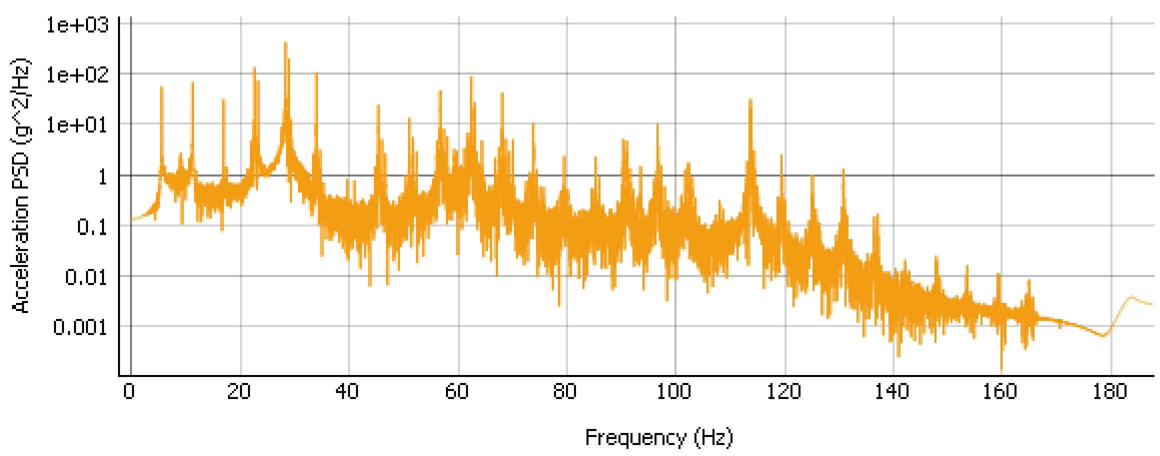


Figura 3.11. PSD de la señal del acelerómetro 1.
(Fuente: Propia)

Al realizar el análisis en un rango de frecuencias entre 1 y 80 Hz, OpenModal presenta la FRF de cada acelerómetro, sobre la cual se aplicó la técnica PP para determinar los modos de vibración de la viga. Cabe señalar que no es necesario aplicar la técnica PP en todas las FRFs, basta con aplicar a una. En la FRF de la señal del acelerómetro 1 se aplicó la técnica PP, se seleccionaron manualmente los picos de la FRF correspondientes a 11,39, 29,14 y 68,35 Hz, los cuales corresponden a las frecuencias naturales de la viga. Los demás picos no se seleccionaron debido a que se tratan de modos torsionales de vibración, como se explicó en la sección anterior.

En las Figuras 3.12, 3.13 y 3.14 se muestra un ajuste de curva generado a partir de la selección de los picos de la FRF de las señales de los 3 acelerómetros respectivamente.

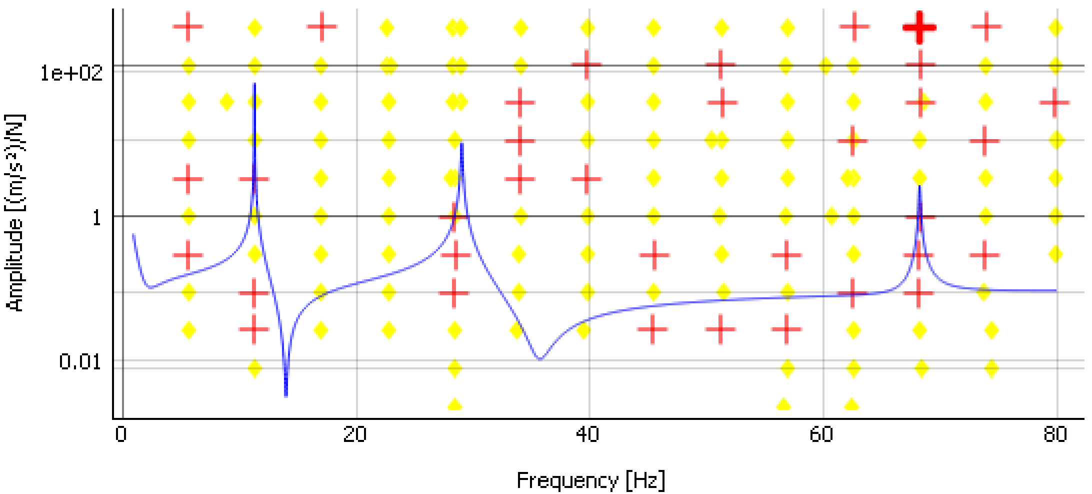


Figura 3.12. Ajuste de curva de la FRF de la señal del acelerómetro 1.
(Fuente: Propia)

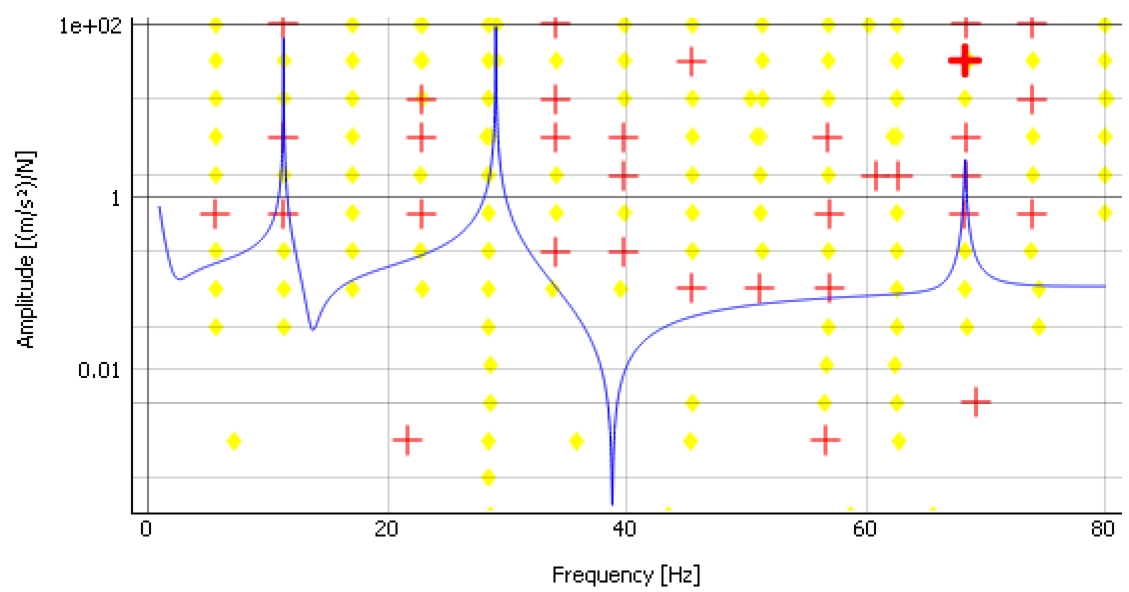


Figura 3.13. Ajuste de curva de la FRF de la señal del acelerómetro 2.
(Fuente: Propia)

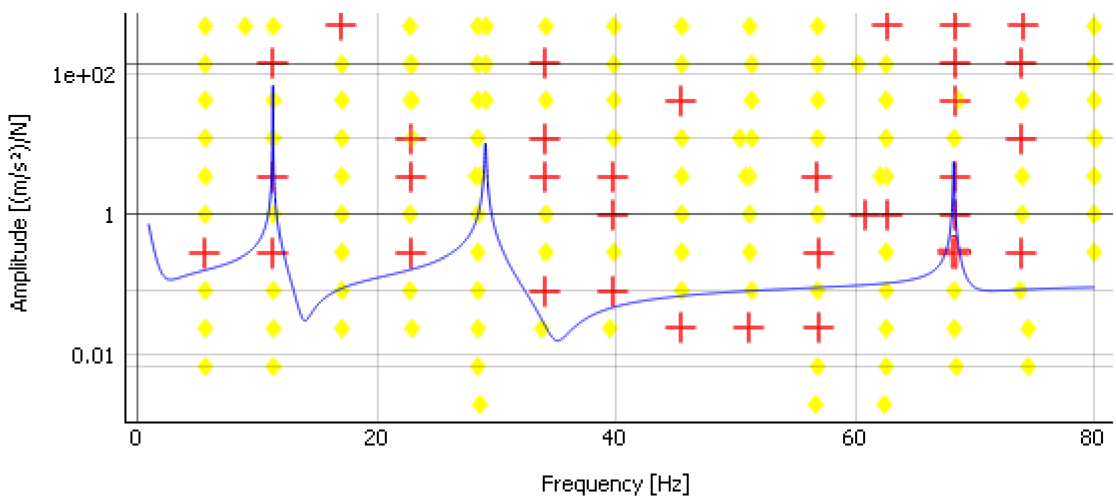


Figura 3.14. Ajuste de curva de la FRF de la señal del acelerómetro 3.
(Fuente: Propia)

Se observa que en la FRF las unidades del eje vertical son $[(m/s^2)/N]$, ya que se trata de una función de transferencia que relaciona la respuesta dinámica con la fuerza de excitación. Note que una función de transferencia depende de las propiedades dinámicas del sistema estructural como son: masa, factor de amortiguamiento y rigidez. Por esta razón es que, intrínsecamente el valor del factor de amortiguamiento puede ser determinado a partir del máximo valor del pico de la FRF.

Después de aplicar la técnica PP en la FRF, se obtuvo los modos de vibración de la viga doblemente empotrada, cuyos resultados se muestran en la Tabla 3.4.

Tabla 3.4. Modos de vibración obtenidos en OpenModal.

Modo	$f_n\ [Hz]$	$\xi\ [\%]$	Forma modal
1	11,386	0,003	Z X 1 Y 2 3 4 5
2	29,144	0,183	Z X 1 Y 2 3 4 5
3	68,348	0,083	Z X 1 Y 2 3 4 5

(Fuente: Propia)

Como se puede observar, el mayor factor de amortiguamiento se presenta en el segundo modo de vibración, mientras que el menor se presenta en el primer modo. Como ya se mencionó en la sección 1.3.1, los factores de amortiguamiento obtenidos mediante la técnica PP no son confiables y están sujetos a errores, es por este motivo que los resultados obtenidos en OpenModal no son totalmente válidos. Para determinar los factores de amortiguamiento se empleó el método del ancho de banda de media potencia, cuyos resultados se analizan en la sección 3.5.

3.4. Análisis de errores

Como proceso de validación de los resultados experimentales, se calcularon los errores relativos. Se aplicó la Ecuación 2.3 a las frecuencias naturales obtenidas en ANSYS y en OpenModal, y se obtuvo los resultados que se muestran en la Tabla 3.5.

Tabla 3.5. Error relativo de frecuencias naturales obtenidas en ANSYS y OpenModal.

Modo	ANSYS	OpenModal	Error (%)
1	11,62 Hz	11,386 Hz	2,055
2	29,928 Hz	29,144 Hz	2,69
3	68,974 Hz	68,348 Hz	0,915

(Fuente: Propia)

Como se puede observar, el mayor porcentaje de error relativo se presenta en el segundo modo de vibración, mientras que el menor se presenta en el tercer modo. Se obtuvo errores relativos menores al 5% en todos los modos flexionantes de vibración.

De manera general, los errores relativos obtenidos se deben en gran medida a la dificultad de modelar en ANSYS el sistema estructural real, pese a que el modelo numérico presenta un sistema de fijación igual al del modelo experimental y, por tanto, proporciona resultados más exactos. Los errores relativos muestran que es muy difícil resolver numéricamente y analíticamente un sistema dinámico bajo ciertas condiciones de borde y, por consiguiente, muestran la importancia de calibrar los modelos numéricos y analíticos a fin de capturar los efectos reales de los parámetros dinámicos. No obstante, la calibración de los modelos numéricos y analíticos no se encuentran dentro del alcance del presente proyecto.

Finalmente, como causa de error también se consideran aspectos de calibración de los acelerómetros, e incluso los modos de vibración inherentes de los acelerómetros (ruido), causas que de cierta manera generan un pequeño error.

Por otra parte, en la Tabla 3.6 se realiza una comparación entre las formas modales obtenidas de la simulación en ANSYS, y las formas modales obtenidas experimentalmente en OpenModal.

Tabla 3.6. Comparación de formas modales obtenidas en ANSYS y OpenModal.

Modo	ANSYS	OpenModal
1		
2		
3		

(Fuente: Propia)

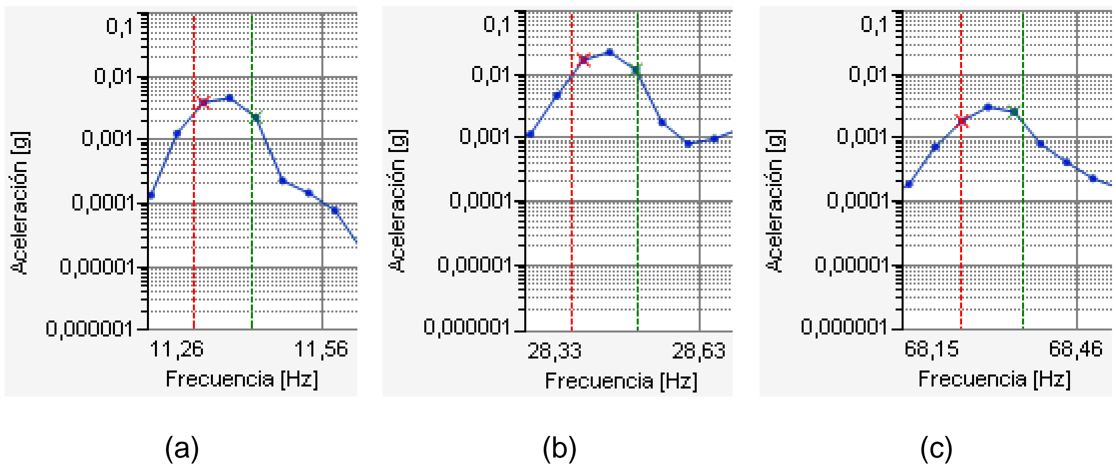
Se puede observar que las formas modales obtenidas experimentalmente en OpenModal tienen similitud con las formas modales resultantes de la simulación en ANSYS, por lo tanto, las formas modales son válidas. Las formas modales obtenidas en OpenModal se asemejan más a un sistema discreto debido a que se emplearon solamente los tres acelerómetros disponibles en el LAEV. Para obtener una mejor descripción de la forma modal es necesario emplear un mayor número de acelerómetros.

Por todo lo descrito en esta sección, las frecuencias naturales y formas modales obtenidas experimentalmente en OpenModal han sido validadas mediante las simulaciones de los modelos numéricos en ANSYS. En la siguiente sección se analizan los factores de amortiguamiento calculados mediante el método del ancho de banda de media potencia.

3.5. Análisis de amortiguamiento

Una vez validados los resultados experimentales obtenidos en OpenModal, se determinaron los factores de amortiguamiento, mediante la aplicación de la Ecuación 2.4, correspondiente al método del ancho de banda de media potencia. Este método se aplicó en el espectro de las componentes en dominio de frecuencia, obtenido en MAINTraq

Analyzer, Figura 3.8 (a), ya que este software permite determinar con mayor exactitud las frecuencias de media potencia (f_a, f_b). En la Figura 3.15 se muestra en detalle la selección de las frecuencias de media potencia para cada pico del espectro.



(a) (b) (c)

Figura 3.15. Selección de frecuencias de media potencia en MAINTraq Analyzer.
(a) Primer modo; (b) Segundo modo; (c) Tercer modo.
(Fuente: Propia)

En la Tabla 3.7 se muestran los resultados obtenidos de factores de amortiguamiento.

Tabla 3.7. Resultados para factores de amortiguamiento.

Modo	$f_n\ [Hz]$	$f_a\ [Hz]$	$f_b\ [Hz]$	$\xi\ [\%]$
1	11,40	11,342	11,425	0,364
2	28,47	28,424	28,501	0,135
3	68,30	68,263	68,354	0,066

(Fuente: Propia)

La estructura presenta un factor de amortiguamiento máximo de 0,364% correspondiente al primer modo de vibración, y un factor de amortiguamiento mínimo de 0,066% en el tercer modo de vibración (pico de menor amplitud). Los factores de amortiguamiento calculados dependen de la amplitud del pico y de la diferencia de las frecuencias de media potencia (ancho de banda). Estos valores de amortiguamiento se encuentran dentro del límite establecido por varios autores, como Adams & Askenazi (1999) y Avci & Davis (2015), quienes afirman que metales y en general estructuras, presentan un factor de amortiguamiento histerético (debido a la deformación del material), menor al 1%.

Finalmente, se observa que los factores de amortiguamiento obtenidos en OpenModal no son totalmente confiables y, por tanto, es necesario aplicar el método del ancho de banda de media potencia para determinar los factores de amortiguamiento.

4. CONCLUSIONES Y RECOMENDACIONES

4.1. Conclusiones

Se implementó una metodología para realizar OMA utilizando el software libre OpenModal, que permitió determinar los tres primeros modos flexionantes de vibración de una viga doblemente empotrada como caso de estudio. Además, esto permitió potencializar el uso del equipo de adquisición de señales dinámicas disponible en el LAEV.

El número de datos necesarios para garantizar la estacionariedad de las señales estocásticas, es decir, que sus parámetros estadísticos no cambien en el tiempo, se lograron con ensayos experimentales de un tiempo mínimo de un minuto. Además, los ensayos experimentales se deben realizar con una frecuencia de excitación menor a la frecuencia natural del primer modo de vibración.

Para el procesamiento de los datos medidos con el equipo de adquisición de señales dinámicas y su exportación al software OpenModal, se desarrolló un código en Python, el cual también permitió filtrar las señales de aceleración, mediante un filtro paso bajo tipo Butterworth de tercer orden, y posteriormente calcular la DFT de las mismas.

Se realizaron las simulaciones de diferentes modelos numéricos en ANSYS y se determinaron las frecuencias naturales y formas modales de la viga, cuyos resultados fueron comparados con los resultados experimentales. Los cálculos de error relativo de las frecuencias naturales permiten establecer que ANSYS no produce con exactitud el segundo modo de vibración, donde el error es aproximadamente 2,69%. Los errores pueden ser atribuidos a condiciones de experimentación no ideales, al modelo numérico, calibración de los acelerómetros y errores de instrumentación.

El factor de amortiguamiento de la estructura se calculó mediante el método del ancho de banda de media potencia y se obtuvo un amortiguamiento máximo de 0,364%, el cual se encuentra dentro del límite establecido por varios autores, quienes afirman que los metales y sistemas estructurales presentan de manera general factores de amortiguamiento histerético menores al 1%.

Finalmente, durante la investigación realizada para la elaboración de este proyecto, no se encontraron estudios previos desarrollados en el Ecuador sobre OMA, por lo que este proyecto de titulación es innovador y contribuye al conocimiento sobre integridad estructural en nuestro país, mediante el desarrollo de una metodología y, por lo tanto, una

herramienta que permita determinar propiedades dinámicas de estructuras y componentes mecánicos sometidos a vibraciones.

4.2. Recomendaciones

Una vez determinadas mediante ANSYS las frecuencias naturales que se esperan obtener experimentalmente, si las condiciones lo permiten se recomienda excitar armónicamente la estructura a frecuencias menores a la frecuencia natural del primer modo de vibración.

Para obtener formas modales más exactas en OpenModal, se recomienda emplear un mayor número de acelerómetros, de esta manera las formas modales se asemejan más a un sistema continuo.

Finalmente, para el cálculo de los factores de amortiguamiento mediante el método del ancho de banda de media potencia, se recomienda emplear MAINTraq Analyzer, ya que este software mediante sus cursores permite determinar con mayor exactitud las frecuencias de media potencia, lo cual no es posible en OpenModal.

REFERENCIAS BIBLIOGRÁFICAS

Abdel, G., & Scanlan, R. (1985). *Ambient vibration studies of Golden Gate Bridge. I: Suspended structure.* Journal of Engineering Mechanincs.

Adams, V., & Askenazi, A. (1999). *Building Better Products with Finite Element Analysis.* Santa Fe, USA: OnWord Press.

Avci, O., & Davis, B. (2015). *A Study on Effective Mass of One Way Joist Supported Systems.* ASCE Structures Conngress, Portland, USA.

Beer, F., Johnston, R., DeWolf, J., & Mazurek, D. (2010). *Mecánica de Materiales* (Quinta ed.). México D. F., México: McGraw-Hill.

Brownjohn, J., Dumanoglu, A., & Severn, R. (1992). *Ambient vibration survey of the Faith Sultan Mehmet (Second Bosporus) suspension bridge.* Earthquake Engineering & Structural Dynamics.

Brüel & Kjaer. (2018). *Análisis Modal Operacional de PULSE (OMA)*. Obtenido de Brüel & Kjaer: https://www.bksv.com/es-ES/products/Analysis-software/structural-dynamics-software/modal-measurements-and-analysis/operational-modal-analysis-8760-8761-8762

Chang, C., Chang, T. Y., & Zhang, Q. W. (2001). *Ambient vibration of long-span cable-stayed bridge.* Journal of Bridge Engineering.

Chopra, A. (2014). *Dinámica de Estructuras* (Cuarta ed.). Ciudad de México: Pearson.

Cunha, A., Caetano, E., & Delgado, R. (2001). *Dynamic tests on large cable-stayed bridge.* Journal of Bridge Engineering.

Ewins, D. J. (1986). *Modal Testing: Theory and Practice.* Baldock, Hertfordshire, England: Research Studies Press.

Ewins, D. J. (2000). *Modal testing: theory, practice, and application.* Baldock, Hertfordshire, England: Research Studies Press.

FLUKE. (2019). *Analizador de vibraciones Fluke 810*. Obtenido de Fluke: https://www.fluke.com/es-ec/producto/mantenimiento-mecanico/analisis-de-vibraciones/fluke-810

FLUKE. (2019). *Medidor de vibraciones Fluke 805*. Obtenido de FLUKE: https://www.fluke.com/es-ec/producto/mantenimiento-mecanico/analisis-de-vibraciones/fluke-805

Gade, S., Moller, N., Herlufsen, H., & Konstantin-Hansen, H. (2005). *Frequency Domain Techniques for Operational Modal Analysis.* International Operational Modal Analysis Conference (IOMAC), Copenhagen, Denmark.

Ghalishooyan, M., & Shooshtari, A. (2015). *Operational modal analysis techniques and their theoretical and practical aspects: A comprehensive review and introduction.* International Operational Modal Analysis Conference (IOMAC), Gijón, España.

Gómez, F., & Herrera, G. (2011). Análisis comparativo entre un Análisis Modal Experimental (EMA) y un Análisis Modal en Operación (OMA) realizado sobre un Rotorkit. *(Tesis de pregrado).* Universidad Pontificia Bolivariana, Bucaramanga.

Harris, C. (2002). *Harris´ Shock and Vibration Handbook.* New York: McGraw-Hill.

He, J., & Fu, Z.-F. (2001). *Modal Analysis.* Oxford: Butterworth Heinemann.

Henao, D., Botero, J. C., & Muriá, D. (2014). *Identificación de propiedades dinámicas de un modelo estructural sometido a vibración ambiental y vibración forzada empleando mesa vibradora.* Ingeniería Sísmica, México.

Hung, V., Thomas, M., Lakis, A., & Marcouiller. (2007). *Identification of modal parameters by operational modal analysis for the assessment of bridge rehabilitation.* International Operational Modal Analysis Conference (IOMAC), Copenhagen, Denmark.

IDM. (2019). *Analizadores de vibración y balanceadores*. Obtenido de IDEAR - Mantenimiento Predictivo: http://www.idmtto.com/index.php/idear-mtto-predictivo

Indian Institute of Technology. (2015). *Power Spectral Density Function (PSD)*. Obtenido de Cygnus Research International: https://www.cygres.com/OcnPageE/Glosry/SpecE.html

Isernia, D., & Rodríguez, J. (2011). El modelado de una grieta de fatiga en una estructura plana y su detección mediante la transformada de wavelet. *Ingeniería Mecánica, 14*, 74-86.

Juang, J. N. (1994). *Applied System Identification.* Englewood Cliffs, New Jersey, USA: Prentice Hall.

Kline, M. (1990). *Mathematical though from ancient to modern times.* Oxford: Oxford University.

Ljung, L. (1987). *System Identification: Theory for the User.* Englewood Cliffs, New Jersey, USA: Prentice Hall.

Masjedian, M., & Keshmiri, M. (2009). *A Review on Operational Modal Analysis Researches: Classification of Methods and Applications.* International Operational Modal Analysis Conference (IOMAC), Potonovo, Italy.

Material Mundial. (2020). *Acero ASTM A36*. Obtenido de Material Mundial: https://www.materialmundial.com/acero-astm-a36-propiedades-ficha-tecnica-estructural/

MathWorks. (2019). *Power Spectral Density Estimates Using FFT*. Obtenido de MathWorks: https://la.mathworks.com/help/signal/ug/power-spectral-density-estimates-using-fft.html

Ministerio de Desarrollo Urbano y Vivienda. (2014). *Norma Ecuatoriana de la Construcción: Estructuras de Acero.* Quito, Ecuador: MIDUVI.

Nikitas, N., Hugh, J., & Tsavdaridis, K. (2015). *Modal Analysis.* Encyclopedia of Earthquake Engineering, Crown.

Okauchi, I., Miyata, T., Tatsumi, M., & Sasaki, N. (1997). *Field vibration test of a long span cable-stayed bridge using large exciters.* Journal of Bridge Engineering, Tokyo.

Olshausen, B. (2000). *Aliasing.* University of California, California.

Papagiannopoulos, G., & Hatzigeorgiou, G. (2011). *On the use of the half-power bandwidth method to estimate damping in building structures.* Soil Dynamics and Earthquake Engineering, Greece.

Rainieri, C., & Fabbrocino, G. (2014). *Operational Modal Analysis of Civil Engineering Structures.* New York: Springer.

Rao, S. (2018). *Mechanical Vibrations* (Sexta ed.). Malasia: Pearson.

Ren, W.-X., & Zong, Z.-H. (2004). *Output-only modal parameter identification of.* Structural Engineering and Mechanics, China.

Ren, W.-X., Harik, I. E., Lenett, M., & Basehearh, T. (2001). *Modal properties of the Roebling Suspension Bridge - FEM modeling and ambient testing.* IMAC, Florida, USA.

Ren, W.-X., Zhao, T., & Harik, I. E. (2003). *Experimental and analytical modal analysis of a steel arch bridge.* Journal of Bridge Engineering.

Rincón, L. (2011). *Introducción a los Procesos Estocásticos.* México DF: Facultad de Ciencias UNAM.

Rodríguez, M. (2005). *Análisis Modal Operacional: Teoría y Práctica.* Sevilla.

Rojas, P. (2014). Análisis modal del banco de ensayo de vibraciones del Laboratorio del Departamento de Ingeniería Mecánica. *(Tesis de pregrado).* Universidad del Bío-Bío, Chile.

SIEMENS. (2019). *How to calculate damping from a FRF?* Obtenido de SIEMENS: https://community.sw.siemens.com/s/article/how-to-calculate-damping-from-a-frf

Slavic, J., Mrsnik, M., Pirnat, M., & Starc, B. (2015). *OPENMODAL*. Obtenido de OPENMODAL: http://www.openmodal.com/

SPYDER. (2018). *SPYDER: The Scientific Python Development Environment*. Obtenido de The Spyder Website Contributors: https://www.spyder-ide.org/

Structural Dynamics Research Lab. (2020). *Universal File Formats for Modal Analysis Testing*. Obtenido de SDRL Structural Dynamics Research Lab: http://www.sdrl.uc.edu/sdrl/referenceinfo/universalfileformats

Structural Vibration Solutions. (2019). *ARTeMIS Modal*. Obtenido de Structural Vibration Solutions: http://www.svibs.com/

Valiometro. (2019). *Vibrómetro para medición de aceleración, velocidad, y desplazamiento VB8213*. Obtenido de Valiometro: http://www.valiometro.pe/vibrometro-para-medicion-de-aceleracion-velocidad-y-desplazamiento-vb8213

Xu, Y. L., Ko, J. M., & Zhang, W. S. (1997). *Vibration studies of Tsing Ma Suspension Bridge.* Journal of Bridge Engineering.

Zamora, G. (2016). Análisis de Fourier vs. Análisis Modal Operacional - Fortalezas y debilidades en la evaluación de la salud estructural. *(Tesis de maestría).* Universidad Nacional Autónoma de México, México D. F.

ANEXOS

ANEXO I. Simulación del modelo numérico en ANSYS.

ANEXO II. Habilitación de módulos y comunicación del equipo de adquisición de señales dinámicas.

ANEXO III. Código en Python para procesamiento de datos.

ANEXO IV. Código en Python del módulo “pyuff”.

ANEXO V. Código en Python para inicializar OpenModal.

ANEXO VI. Modos de vibración obtenidos en ANSYS.

ANEXO I.

SIMULACIÓN DEL MODELO NUMÉRICO EN ANSYS

La simulación del modelo numérico se realiza en ANSYS Workbench 18.2, un software de simulación ingenieril que emplea la teoría de AEF. En la Figura I.1 se muestran algunos módulos de análisis disponibles en ANSYS, uno de ellos es el módulo "Modal", el cual permite determinar las frecuencias naturales y simular las formas modales de la viga en estudio. Se realizaron las simulaciones de tres casos distintos: el primero corresponde a la viga con las superficies internas de los agujeros fijas, el segundo corresponde a la viga con ambos extremos fijos, mientras que el último caso corresponde a un modelo numérico con un sistema de fijación igual al del modelo experimental. A continuación, se detallan los pasos necesarios para simular el tercer caso, con el cual se obtuvo resultados más exactos.

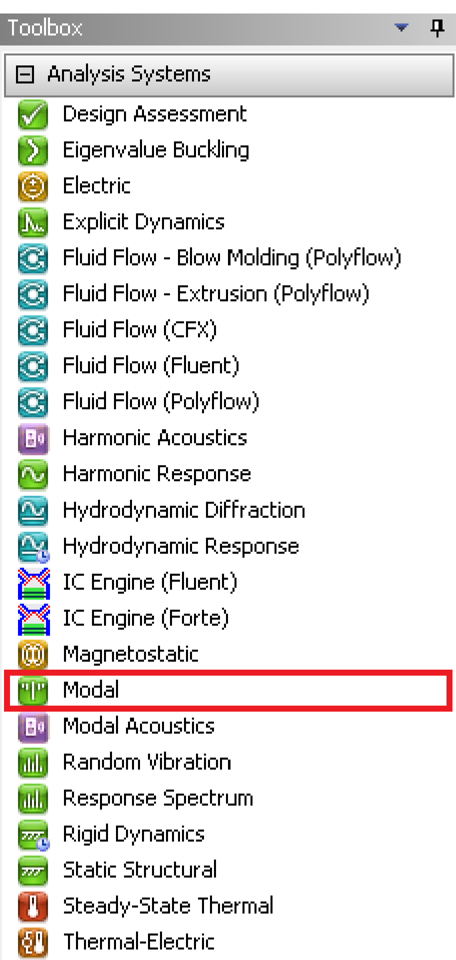


Figura I.1. Módulos de análisis en ANSYS Workbench.
(Fuente: Propia)

Para seleccionar el módulo a analizar, dar doble click y automáticamente se añade en el esquema de proyectos el respectivo bloque, como se muestra en la Figura I.2.

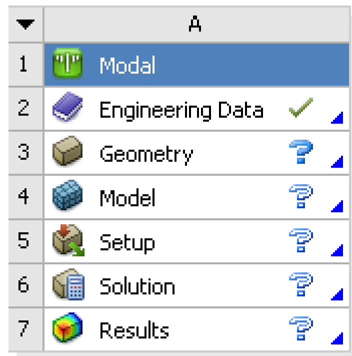


Figura I.2. Bloque del módulo "Modal" en ANSYS Workbench.
(Fuente: Propia)

Pese a que en SolidWorks es posible asignar el material correspondiente de la viga, en ANSYS Workbench es necesario editar el material, para ello, click derecho en “Engineering Data”, seleccionar “Edit” y, a continuación, se despliega una pantalla en donde es posible añadir un nuevo material como se muestra en la Figura I.3. Escribir el nombre del nuevo material, en este caso ASTM A36.

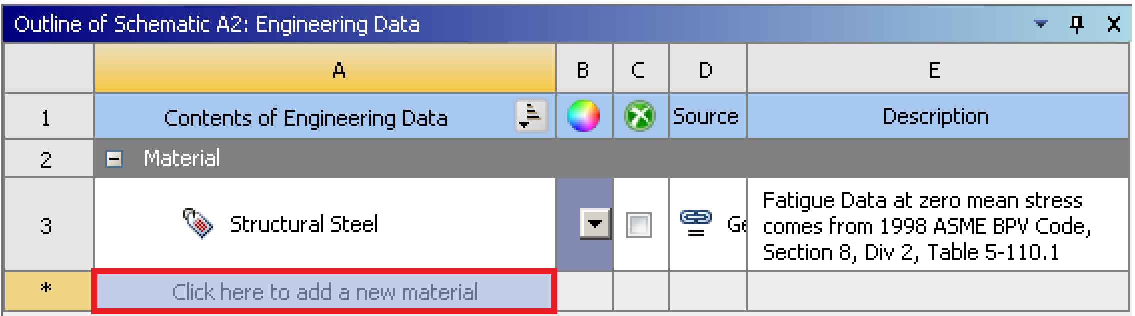


Figura I.3. Añadir nuevo material en ANSYS Workbench.
(Fuente: Propia)

A continuación, se deben añadir las propiedades “Density” e “Isotropic Elasticity” como se muestra en la Figura I.4. Para ello dar doble click sobre cada propiedad.

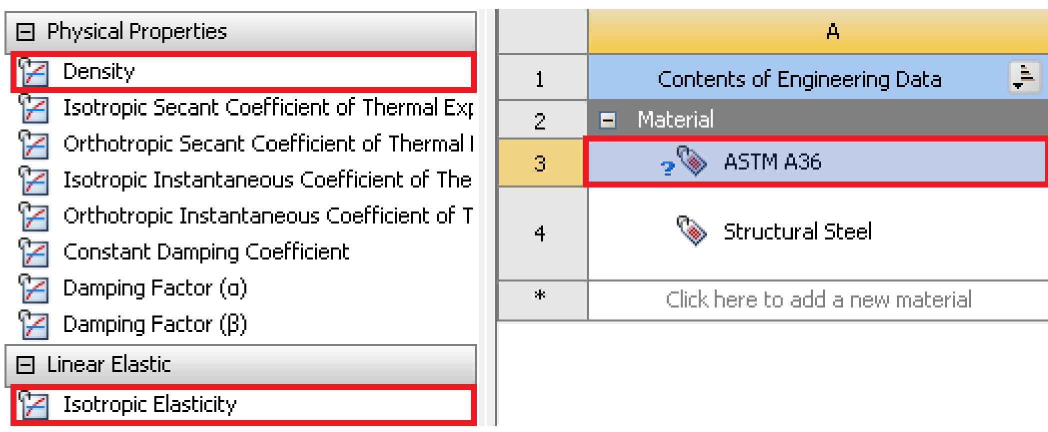


Figura I.4. Añadir propiedades al nuevo material en ANSYS Workbench.
(Fuente: Propia)

En la Figura I.5 se muestran las propiedades que deben ser llenadas, cuyos valores fueron tomados para el Acero ASTM A36 de Beer, Johnston, DeWolf y Mazurek (2010) y de Material Mundial (2020).

Properties of Outline Row 4: ASTM A36

	A	B	C	D	E
1	Property	Value	Unit		
2	Material Field Variables	Table			
3	Density	7850	kg m^-3		
4	Isotropic Elasticity				
5	Derive from	Young's Modulus ...			
6	Young's Modulus	2E+11	Pa		
7	Poisson's Ratio	0,26			
8	Bulk Modulus	1,3889E+11	Pa		
9	Shear Modulus	7,9365E+10	Pa		

Figura I.5. Propiedades del Acero ASTM A36 en ANSYS Workbench.
(Fuente: Propia)

Una vez añadidos todos los valores de las propiedades, volver al esquema de proyectos dando click en el ícono Project ubicado en la barra de herramientas. Posteriormente se procede a importar el modelo geométrico de la viga previamente diseñado en SolidWorks, para ello, click derecho en “Geometry”, seleccionar “Import Geometry” y a continuación “Browse”, como se muestra en la Figura I.6. Seleccionar el archivo de SolidWorks con extensión igs. Cabe indicar que los pernos, tuercas, arandelas y perfiles estructurales (ángulos tipo L) fueron obtenidos de las librerías de SolidWorks.

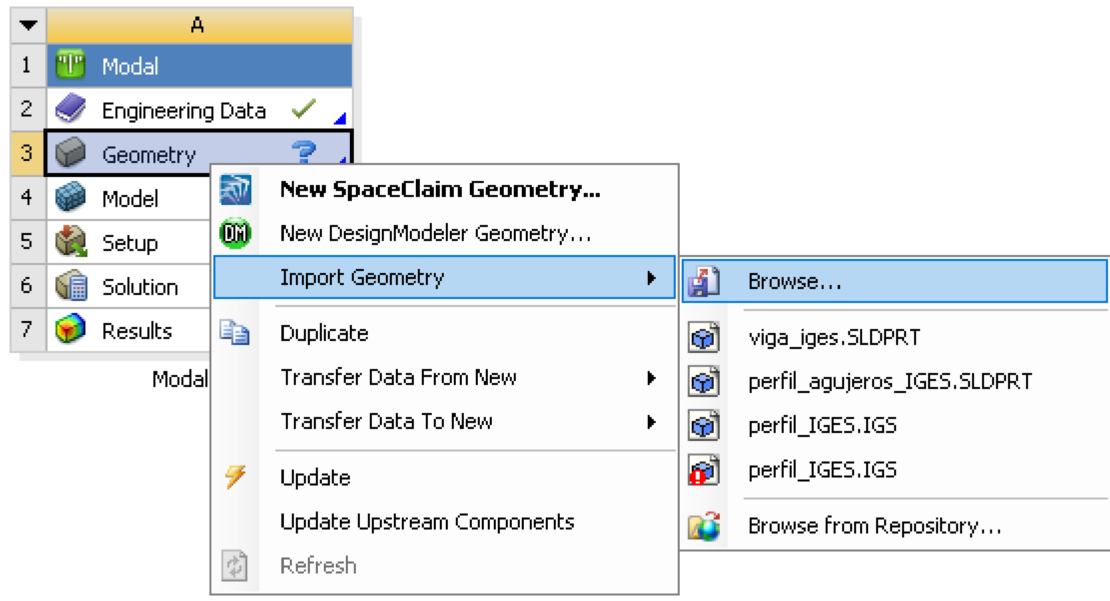


Figura I.6. Importación de geometría de la viga en ANSYS Workbench.
(Fuente: Propia)

Una vez añadidas las propiedades del nuevo material e importada la geometría, es necesario actualizar el proyecto, para lo cual dar click en el ícono Update Project ubicado en la barra de herramientas. Al actualizarse el proyecto, todos los componentes del bloque "Modal" aparecen con el ícono ✓. A continuación, doble click en "Model" y se despliega un nuevo espacio de trabajo llamado "Mechanical", en donde se realiza el análisis modal.

En la sección "Geometry" se debe asignar el material creado en los pasos anteriores, como se muestra en la Figura I.7. Cabe señalar que solamente a la viga se asigna el material ASTM A36.

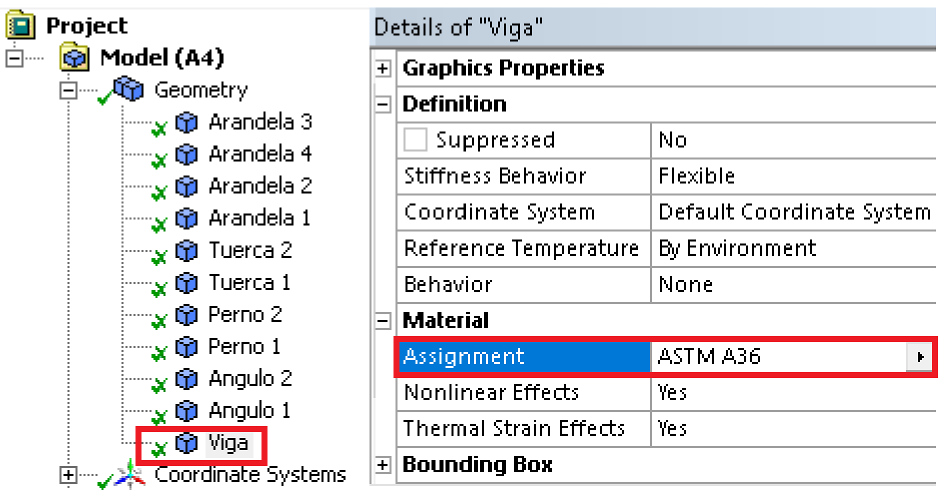


Figura I.7. Asignar material ASTM A36 a geometría en Mechanical.
(Fuente: Propia)

En la Figura I.8 se detallan los diferentes elementos del modelo geométrico ensamblados en SolidWorks. Cabe señalar que el Ángulo 1 corresponde a la estructura del motor desbalanceado con el cual se excita la viga.

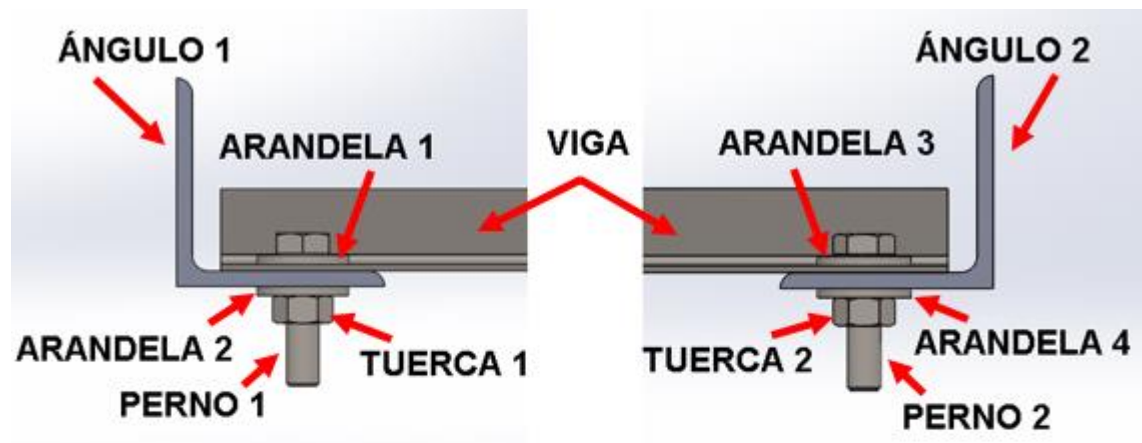


Figura I.8. Elementos del modelo geométrico ensamblados en SolidWorks.
(Fuente: Propia)

En la sección “Connections” se procede a configurar los contactos de los diferentes elementos ensamblados en SolidWorks. En la Tabla I.1 se muestran los tipos de contactos entre los diferentes elementos, mientras que en la Tabla I.2 se muestra la configuración para cada tipo de contacto.

Tabla I.1. Tipos de contactos entre los elementos del modelo numérico.

Elementos en contacto	Contacto	Elementos en contacto	Contacto
Viga – Ángulo 1	Frictional	Perno 1 – Tuerca 1	Bonded
Viga – Ángulo 2	Frictional	Perno 2 – Tuerca 2	Bonded
Viga – Arandela 1	Frictional	Perno 1 – Arandela 1	Bonded
Viga – Arandela 3	Frictional	Perno 2 – Arandela 3	Bonded
Ángulo 1 – Arandela 2	Frictional	Tuerca 1 – Arandela 2	Bonded
Ángulo 2 – Arandela 4	Frictional	Tuerca 2 – Arandela 4	Bonded

(Fuente: Propia)

Tabla I.2. Configuración de los contactos de los elementos del modelo numérico.

	Contacto tipo “Frictional”	Contacto tipo “Bonded”
Coeficiente de fricción	0,15	-
Comportamiento	Simétrico	Simétrico
Formulación	Lagrange aumentado	“Pure Penalty”
Método de detección	En el punto de Gauss	En el punto de Gauss

(Fuente: Propia)

En la siguiente sección, seleccionar “Mesh” y configurar el mallado. En este caso los parámetros configurados para obtener los mejores resultados se muestran en la Figura I.9. Una vez configurado el mallado, click derecho en “Mesh” y seleccionar “Generate Mesh”.

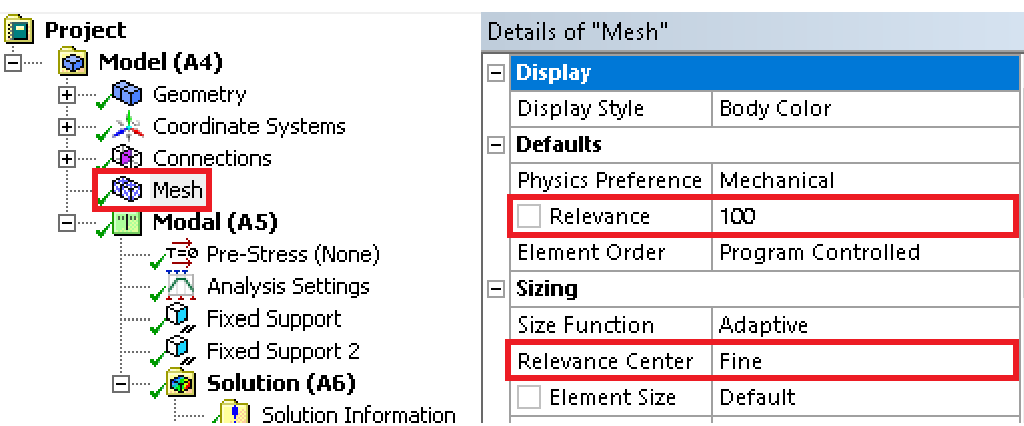


Figura I.9. Configuración de mallado en Mechanical.
(Fuente: Propia)

En el modelo experimental, la viga se sujetó a dos perfiles estructurales tipo L, los cuales se encuentran soldados, por lo tanto, para la simulación es necesario configurar las condiciones de borde del modelo numérico, para lo cual en la sección “Analysis Settings” seleccionar la opción “Fixed Support” y aplicar a los bordes superiores de los perfiles, como se muestra en la Figura I.10.

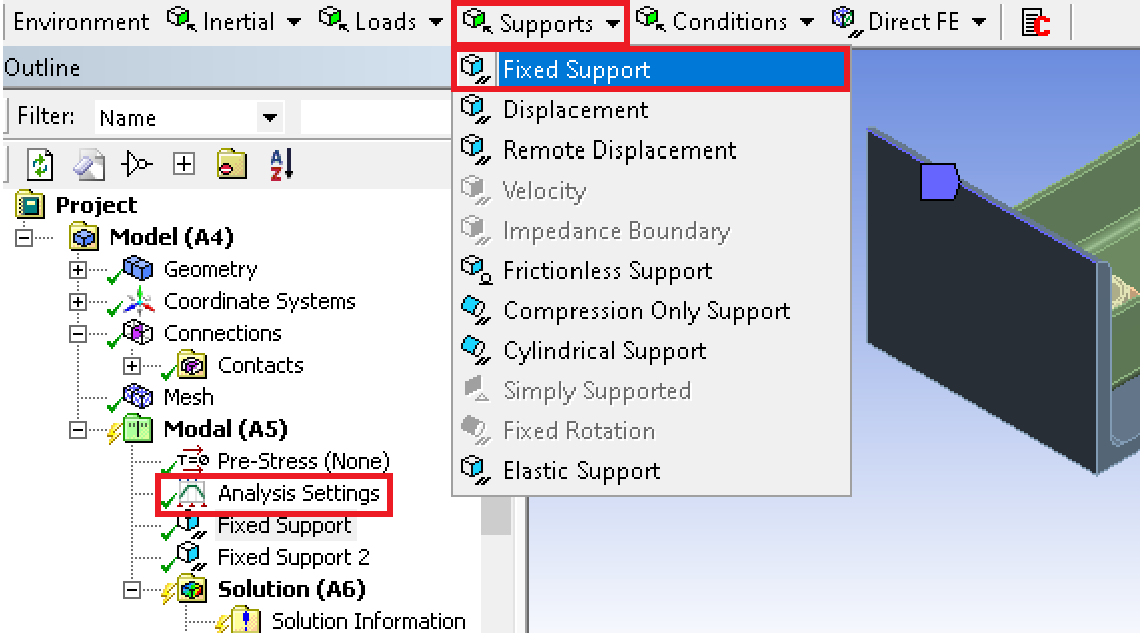


Figura I.10. Condiciones de los extremos de la viga.
(Fuente: Propia)

Además, en “Analysis Settings” es posible configurar el número de modos de vibración a determinar, como se observa en la Figura I.11.

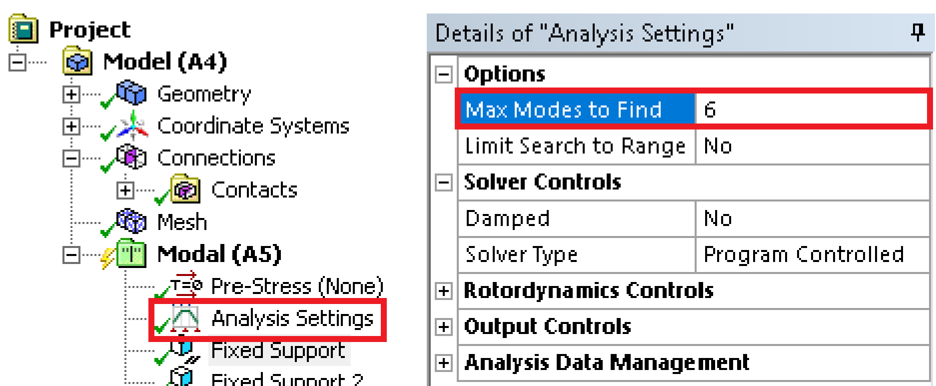


Figura I.11. Configuración del número de modos de vibración a determinar.
(Fuente: Propia)

Para determinar las frecuencias naturales de vibración de la viga, se procede a solucionar el modelo numérico mediante la tecla F5 o al dar click derecho en “Solution” y seleccionar “Solve”.

Para simular las formas modales, en el gráfico de las frecuencias naturales seleccionar todas las frecuencias, dar click derecho y seleccionar “Create Mode Shape Results”, como se muestra en la Figura I.12.

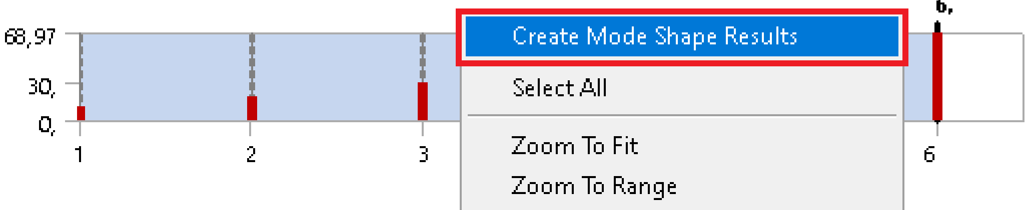


Figura I.12. Creación de resultados de formas modales de la viga.
(Fuente: Propia)

Finalmente, sobre cualquier “Total Deformation”, dar click derecho y seleccionar “Evaluate All Results”, como se indica en la Figura I.13.

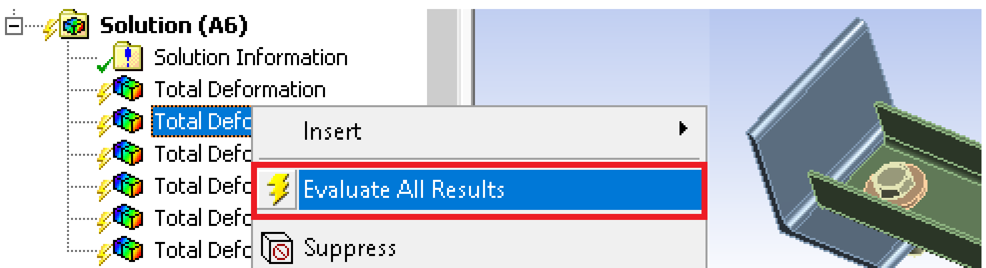
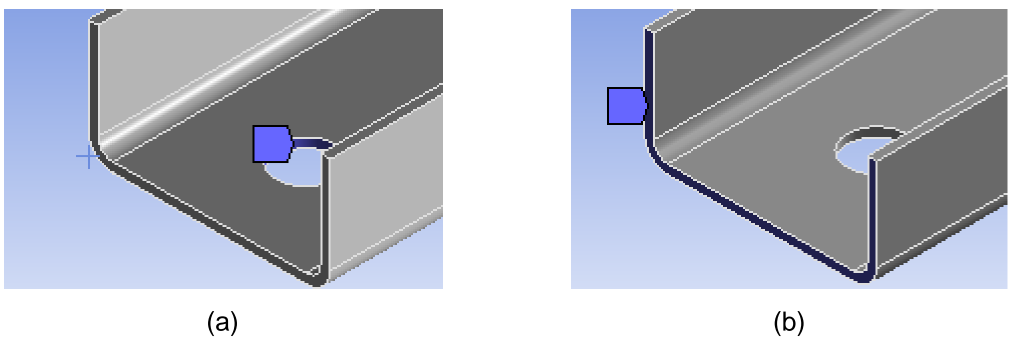


Figura I.13. Generación de formas modales de la viga.
(Fuente: Propia)

De esta manera se determinan las frecuencias naturales y las formas modales para el tercer caso del modelo numérico, cuyos resultados se detallan en el Anexo VI.

Cabe señalar que las simulaciones de los otros 2 casos no presentan contactos, por lo que el procedimiento se vuelve más sencillo. En la sección “Analysis Settings” seleccionar la opción “Fixed Support” y aplicar a las superficies internas de los agujeros (caso 1) y a las superficies de los extremos de la viga (caso 2), como se muestra en la Figura I.14.



(a) (b)

Figura I.14. Condiciones de borde de la viga. (a) Caso 1; (b) Caso 2.
(Fuente: Propia)

ANEXO II.
HABILITACIÓN DE MÓDULOS Y COMUNICACIÓN DEL EQUIPO DE ADQUISICIÓN DE SEÑALES DINÁMICAS

II.1. Instalación del software MAINTraq Analyzer

i. Abrir carpeta “INFORMACIÓN”
ii. Abrir carpeta “IDEAR DISCO 1.7”
iii. Abrir carpeta “Framework PuntoNet”
iv. Ejecutar archivo “DOTNETFX”
v. Volver a carpeta “INFORMACIÓN”
vi. Ejecutar el archivo “MAINTraq Analyzer 2.1 Setup”

II.2. Habilitación del módulo MAINTraq Viewer

i. Panel de control
ii. Firewall de Windows Defender
iii. Permitir una aplicación o una característica a través de Firewall de Windows Defender
iv. Permitir otra aplicación
v. Examinar, Disco Local C, Archivos de programa (x86)
vi. MAINTraq Analyzer
vii. Seleccionar “MTQ.Analyzer.Viewer”

II.3. Comunicación entre el equipo de adquisición de señales dinámicas y el computador

i. Panel de control
ii. Centro de redes y recursos compartidos
iii. Cambiar configuración del adaptador
iv. Ethernet, click derecho, Propiedades
v. Habilitar el protocolo de Internet versión 4 (TCP/IPv4), Propiedades
vi. Seleccionar “Usar la siguiente dirección IP”, escribir la dirección IP y la máscara de subred
vii. Aceptar y Cerrar

ANEXO III.

CÓDIGO EN PYTHON PARA PROCESAMIENTO DE DATOS

```
#Importación de librerías
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy
import pyuff
from scipy.signal import freqz, butter, filtfilt


#Lectura de datos del archivo en formato CSV
datos = pd.read_csv('Aceleraciones.csv', sep=';', header=0, decimal=b',')
t = datos.ix[:,0] #Columna 0 Tiempo
sensor_1 = datos.ix[:, 1] #Columna 1 aceleraciones del sensor 1
sensor_2 = datos.ix[:, 2] #Columna 2 aceleraciones del sensor 2
sensor_3 = datos.ix[:, 3] #Columna 3 aceleraciones del sensor 3


#Dar nueva forma a las matrices sin cambiar sus valores
t = np.reshape(t, len(t))
sensor_1 = np.reshape(sensor_1, len(sensor_1))
sensor_2 = np.reshape(sensor_2, len(sensor_2))
sensor_3 = np.reshape(sensor_3, len(sensor_3))


#Parámetros del filtro
N = len(t) #Número de datos
T = (t.max()-t.min())/N #Periodo de las señales
fs = 1/T #Frecuencia de muestreo del filtro
f = fs*np.arange(0,N//2+1)/N #Vector frecuencia remuestreado
cutoff = 0.49*fs #Frecuencia superior de corte
order = 3 #Orden del filtro
nyquist = 0.5*fs #Frecuencia de Nyquist
normal_cutoff = cutoff/nyquist


def butter_lowpass(cutoff, fs, order):
      b, a = butter(order, normal_cutoff, btype='low', analog=False)
      return b, a

def butter_lowpass_filter(data, cutoff, fs, order):
      b, a = butter_lowpass(cutoff, fs, order=order)
      y = filtfilt(b, a, data)
      return y


#Obtener los coeficientes del filtro
b, a = butter_lowpass(cutoff, fs, order)
w, h = freqz(b, a, worN=1000)


#Graficar la respuesta en frecuencia del filtro
fig1 = plt.figure(figsize=(6, 4))
m = fig1.add_subplot(1, 1, 1)
```

```
m.plot(0.5*fs*w/np.pi, np.abs(h), label="order = %d" % order)
plt.title('Respuesta en frecuencia del filtro Butterworth paso bajo')
plt.xlabel('Frecuencia [Hz]')
plt.ylabel('Ganancia del filtro')
plt.xlim(0,f.max())
plt.grid(True, axis='both')
plt.legend(loc='lower left')


#Filtrar y graficar las señales originales y filtradas

#Señal del acelerómetro 1
acel_1 = butter_lowpass_filter(sensor_1, cutoff, fs, order)
fig1 = plt.figure(figsize=(8, 3))
n = fig1.add_subplot(1, 1, 1)
n.plot(t, sensor_1, 'r-', label='Señal original')
n.plot(t, acel_1, 'b-', label='Señal filtrada')
plt.title('Señal Acelerómetro 1')
plt.xlabel('Tiempo [s]')
plt.ylabel('Amplitud [g]')
plt.xlim(0, 1)
plt.grid(True, axis='both')
plt.legend(loc='upper right')

#Señal del acelerómetro 2
acel_2 = butter_lowpass_filter(sensor_2, cutoff, fs, order)
fig2 = plt.figure(figsize=(8, 3))
o = fig2.add_subplot(1, 1, 1)
o.plot(t, sensor_2, 'r-', label='Señal original')
o.plot(t, acel_2, 'b-', label='Señal filtrada')
plt.title('Señal Acelerómetro 2')
plt.xlabel('Tiempo [s]')
plt.ylabel('Amplitud [g]')
plt.xlim(0, 1)
plt.grid(True, axis='both')
plt.legend(loc='upper right')

#Señal del acelerómetro 3
acel_3 = butter_lowpass_filter(sensor_3, cutoff, fs, order)
fig3 = plt.figure(figsize=(8, 3))
p = fig3.add_subplot(1, 1, 1)
p.plot(t, sensor_3, 'r-', label='Señal original')
p.plot(t, acel_3, 'b-', label='Señal filtrada')
plt.title('Señal Acelerómetro 3')
plt.xlabel('Tiempo [s]')
plt.ylabel('Amplitud [g]')
plt.xlim(0, 1)
plt.grid(True, axis='both')
plt.legend(loc='upper right')


#DFT de las señales filtradas

#Señal del acelerómetro 1
dft_1 = np.fft.fft(acel_1)
p2_1 = np.abs(dft_1/N)
p1_1 = p2_1[0:N//2+1]
p1_1[1:-2] = 2*p1_1[1:-2]
```

```
fig4 = plt.figure(figsize=(8, 3))
q = fig4.add_subplot(1, 1, 1)
q.semilogy(f, p1_1, linewidth=0.8)
plt.xlim(0, 0.5*max(f))
plt.title('Espectro de amplitudes del acelerómetro 1')
plt.xlabel('Frecuencia [Hz]')
plt.ylabel('Aceleración [g]')
plt.grid(True, axis='both')

#Señal del acelerómetro 2
dft_2 = np.fft.fft(acel_2)
p2_2 = np.abs(dft_2/N)
p1_2 = p2_2[0:N//2+1]
p1_2[1:-2] = 2*p1_2[1:-2]
fig5 = plt.figure(figsize=(8, 3))
r = fig5.add_subplot(1, 1, 1)
r.semilogy(f, p1_2, linewidth=0.8)
plt.xlim(0, 0.5*max(f))
plt.title('Espectro de amplitudes del acelerómetro 2')
plt.xlabel('Frecuencia [Hz]')
plt.ylabel('Aceleración [g]')
plt.grid(True, axis='both')

#Señal del acelerómetro 3
dft_3 = np.fft.fft(acel_3)
p2_3 = np.abs(dft_3/N)
p1_3 = p2_3[0:N//2+1]
p1_3[1:-2] = 2*p1_3[1:-2]
fig6 = plt.figure(figsize=(8, 3))
s = fig6.add_subplot(1, 1, 1)
s.semilogy(f, p1_3, linewidth=0.8)
plt.xlim(0, 0.5*max(f))
plt.title('Espectro de amplitudes del acelerómetro 3')
plt.xlabel('Frecuencia [Hz]')
plt.ylabel('Aceleración [g]')
plt.grid(True, axis='both')


#Generar archivo UFF
n_nyq = len(f)
dft_1 = np.array(dft_1, dtype=complex)
dft_2 = np.array(dft_2, dtype=complex)
dft_3 = np.array(dft_3, dtype=complex)
dft_1[0] = np.nan*(1+1.j)
matriz_acel = [dft_1[0:n_nyq], dft_2[0:n_nyq], dft_3[0:n_nyq]]
n_acel = (len(datos.columns)-1) #Número de acelerómetros
for i in range (n_acel): #Genera i=0,1,2,3,...n
      print ('Adding point {:}'.format(i+1))
      response_node = 1
      response_direction = 1
      reference_node = i+1
      reference_direction = 1
      acceleration_complex = matriz_acel[i]
      frequency = f
      name = 'TestCase'
      data = {'type':58,
      'func_type':4,
      'rsp_node': response_node,
```

```
        'rsp_dir': response_direction,
        'ref_dir': reference_direction,
        'ref_node': reference_node,
        'data': acceleration_complex,
        'x': frequency,
        'id1': 'id1',
        'rsp_ent_name': name,
        'ref_ent_name': name,
        'abscissa_spacing': 1,
        'abscissa_spec_data_type': 18,
        'ordinate_spec_data_type': 12,
        'orddenom_spec_data_type': 13}

uffwrite = pyuff.UFF('Señales_UFF.uff')
uffwrite._write_set(data,'add')
```

ANEXO IV.

CÓDIGO EN PYTHON DEL MÓDULO “PYUFF”

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Copyright (C) 2014-2017 Primož Čermelj, Matjaž Mršnik, Miha Pirnat, Janko
Slavič, Blaž Starc (in alphabetic order)
# This file is part of pyuff. #
# pyFRF is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 3 of the License.
#
# pyuff is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with pyuff.  If not, see <http://www.gnu.org/licenses/>.
"""
==========
pyuff module
==========

This module is part of the www.openmodal.com project and
defines an UFF class to manipulate with the
UFF (Universal File Format) files, i.e., to read from and write
to UFF files. Among the variety of UFF formats, only some of the
formats (data-set types) frequently used in structural dynamics
are supported: **151, 15, 55, 58, 58b, 82, 164.** Data-set **58b**
is actually a hybrid format [1]_ where the signal is written in the
binary form, while the header-part is slightly different from 58 but still in the
ascii format.

An UFF file is a file that can have many data-sets of either ascii or binary
data where data-set is a block of data between the start and end tags ``____-1``
(``_`` representing the space character). Refer to [1]_ and [2]_ for
more information about the UFF format.

This module also provides an exception handler class, ``UFFException``.

Sources:
    .. [1] http://sdrl.uc.edu/sdrl/referenceinfo/universalfileformats
    .. [2] Matlab's ``readuff`` and ``writeuff`` functions:
       http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?
objectId=6395

Acknowledgement:
    * This source (py2.7) was first written in 2007, 2008 by Primoz Cermelj
(primoz.cermelj@gmail.com)
    * As part of the www.openmodal.com project the first source was adopted for
Python 3 by
      Matjaz Mrsnik  <matjaz.mrsnik@gmail.com>
    * The package is maintained by Janko Slavič <janko.slavic@fs.uni-lj.si>
Notes:
    * 58 data-set is always written in double precision, even if it is
      read in single precision.
```

```
    * ``numpy`` module is required as all the vector/matrix-type data are read
      or written using ``numpy.array`` objects.
    Example:
    >>> import pyuff
    >>> uff_file = pyuff.UFF('beam.uff')
    >>> uff_file.file_exists()
    True
"""

import os
import struct
import sys
import time

import numpy as np

__version__= '1.22'
_SUPPORTED_SETS = ['151', '15', '55', '58', '58b', '82', '164', '2411', '2412',
'2420']

class UFFException(Exception):
    """An exception that prints a string describing the error.
    """

    def_init_(self, value):
        self.value = value

    def_str_(self):
        return self.value

class UFF:
    """
    Manages data reading and writing from/to the UFF file.

    The UFF class instance requires exactly 1 parameter - a file name of a
    universal file. If the file does not exist, no basic file info will be
    extracted and the status will be False - indicating that the file is not
    refreshed. Hovewer, when one tries to read one or more data-sets, the file
    must exist or the UFFException will be raised.

    The file, given as a parameter to the UFF instance, is open only when
    reading from or writing to the file. The UFF instance refreshes the file
    automatically - use ``UFF.get_status()`` to see the refresh status); note
    that this works fine if the file is being changed only through the UFF
    instance and not by other functions or even by other means, e.g.,
    externally. If the file is changed externally, the ``UFF.refresh()`` should
    be invoked before any reading or writing.

    All array-type data are read/written using numpy's ``np.array`` module.

    Appendix
    --------
    Below are the fileds for all the data sets supported. <..> designates an
    *optional* field, i.e., a field that is not needed when writting to a file
    as the field has a defualt value. Additionally, <<..>> designates fields that
    are *ignored* when writing (not needed at all); these fields are defined
    automatically.

    Moreover, there are also some fields that are data-type dependent, i.e.,
    some fields that are onyl used/available for some specific data-type. E.g.,
```

```
see ``modal_damp_vis`` at Data-set 55.

**Data-set 15 (points data)**:

    * ``'type'``               -- *type number = 15*
    * ``'node_nums'``          -- *list of n node numbers*
    * ``'x'``                  -- *x-coordinates of the n nodes*
    * ``'y'``                  -- *y-coordinates of the n nodes*
    * ``'z'``                  -- *z-coordinates of the n nodes*
    * ``<'def_cs'>``           -- *n deformation cs numbers*
    * ``<'disp_cs'>``          -- *n displacement cs numbers*
    * ``<'color'>``            -- *n color numbers*

**Data-set 82 (line data)**:

    * ``'type'``               -- *type number = 82*
    * ``'trace_num'``          -- *number of the trace*
    * ``'lines'``              -- *list of n line numbers*
    * ``<'id'>``               -- *id string*
    * ``<'color'>``            -- *color number*
    * ``<<'n_nodes'>>``        -- *number of nodes*

**Data-set 151 (header data)**:

    * ``'type'``               -- *type number = 151*
    * ``'model_name'``         -- *name of the model*
    * ``'description'``        -- *description of the model*
    * ``'db_app'``             -- *name of the application that
      created database*
    * ``'program'``            -- *name of the program*
    * ``'model_name'``         -- *the name of the model*
    * ``<'version_db1'>``      -- *version string 1 of the database*
    * ``<'version_db2'>``      -- *version string 2 of the database*
    * ``<'file_type'>``        -- *file type string*
    * ``<'date_db_created'>``  -- *date database was created*
    * ``<'time_db_created'>``  -- *time database was created*
    * ``<'date_db_saved'>``    -- *date database was saved*
    * ``<'time_db_saved'>``    -- *time database was saved*
    * ``<'date_file_written'>``-- *date file was written*
    * ``<'time_file_written'>``-- *time file was written*

**Data-set 164 (units)**:

    * ``'type'``               -- *type number = 164*
    * ``'units_code'``         -- *units code number*
    * ``'length'``             -- *length factor*
    * ``'force'``              -- *force factor*
    * ``'temp'``               -- *temperature factor*
    * ``'temp_offset'``        -- *temperature-offset factor*
    * ``<'units_description'>``-- *units description*
    * ``<'temp_mode'>``        -- *temperature mode number*

**Data-set 58<b> (function at nodal DOF)**:

    * ``'type'``               -- *type number = 58*
    * ``'func_type'``          -- *function type; only 1, 2, 3, 4
      and 6 are supported*
    * ``'rsp_node'``           -- *response node number*
    * ``'rsp_dir'``            -- *response direction number*
    * ``'ref_node'``           -- *reference node number*
```

```
        * ``'ref_dir'``            -- *reference direction number*
        * ``'data'``               -- *data array*
        * ``'x'``                  -- *abscissa array*
        * ``<'binary'>``           -- *1 for binary, 0 for ascii*
        * ``<'id1'>``              -- *id string 1*
        * ``<'id2'>``              -- *id string 2*
        * ``<'id3'>``              -- *id string 3*
        * ``<'id4'>``              -- *id string 4*
        * ``<'id5'>``              -- *id string 5*
        * ``<'load_case_id'>``     -- *id number for the load case*
        * ``<'rsp_ent_name'>``     -- *entity name for the response*
        * ``<'ref_ent_name'>``     -- *entity name for the reference*
        * ``<'abscissa_axis_units_lab'>``-- *label for the units on the abscissa*
        * ``<'abscissa_len_unit_exp'>``  -- *exp for the length unit on the
abscissa*
        * ``<'abscissa_force_unit_exp'>``-- *exp for the force unit on the
abscissa*
        * ``<'abscissa_temp_unit_exp'>`` -- *exp for the temperature unit on the
abscissa*
        * ``<'ordinate_axis_units_lab'>``-- *label for the units on the ordinate*
        * ``<'ordinate_len_unit_exp'>``  -- *exp for the length unit on the
ordinate*
        * ``<'ordinate_force_unit_exp'>``-- *exp for the force unit on the
ordinate*
        * ``<'ordinate_temp_unit_exp'>`` -- *exp for the temperature unit on the
ordinate*
        * ``<'orddenom_axis_units_lab'>``-- *label for the units on the ordinate
denominator*
        * ``<'orddenom_len_unit_exp'>``  -- *exp for the length unit on the
ordinate denominator*
        * ``<'orddenom_force_unit_exp'>``-- *exp for the force unit on the
ordinate denominator*
        * ``<'orddenom_temp_unit_exp'>`` -- *exp for the temperature unit on the
ordinate denominator*
        * ``<'z_axis_axis_units_lab'>``  -- *label for the units on the z axis*
        * ``<'z_axis_len_unit_exp'>``    -- *exp for the length unit on the z
axis*
        * ``<'z_axis_force_unit_exp'>``  -- *exp for the force unit on the z
axis*
        * ``<'z_axis_temp_unit_exp'>``   -- *exp for the temperature unit on the
z axis*
        * ``<'z_axis_value'>``           -- *z axis value*
        * ``<'spec_data_type'>``         -- *specific data type*
        * ``<'abscissa_spec_data_type'>``-- *abscissa specific data type*
        * ``<'ordinate_spec_data_type'>``-- *ordinate specific data type*
        * ``<'orddenom_spec_data_type'>``-- *ordinate denominator specific data
type*
        * ``<'z_axis_spec_data_type'>``  -- *z-axis specific data type*
        * ``<'ver_num'>``                -- *version number*
        * ``<<'ord_data_type'>>``        -- *ordinate data type*
        * ``<<'abscissa_min'>>``         -- *abscissa minimum*
        * ``<<'byte_ordering'>>``        -- *byte ordering*
        * ``<<'fp_format'>>``            -- *floating-point format*
        * ``<<'n_ascii_lines'>>``        -- *number of ascii lines*
        * ``<<'n_bytes'>>``              -- *number of bytes*
        * ``<<'num_pts'>>``              -- *number of data pairs for
          uneven abscissa or number of data values for even abscissa*
        * ``<<'abscissa_spacing'>>``     -- *abscissa spacing; 0=uneven,
          1=even*
        * ``<<'abscissa_inc'>>``         -- *abscissa increment; 0 if
```

```
      spacing uneven*

**Data-set 55 (data at nodes)**:

    * ``'type'``               -- *type number = 55*
    * ``'analysis_type'``      -- *analysis type number; currently
      only only normal mode (2), complex eigenvalue first order
      (displacement) (3), frequency response and (5) and complex eigenvalue
      second order (velocity) (7) are supported*
    * ``'data_ch'``            -- *data-characteristic number*
    * ``'spec_data_type'``     -- *specific data type number*
    * ``'load_case'``          -- *load case number*
    * ``'mode_n'``             -- *mode number; applicable to
      analysis types 2, 3 and 7 only*
    * ``'eig'``                -- *eigen frequency (complex number);
      applicable to analysis types 3 and 7 only*
    * ``'freq'``               --*frequency (Hz); applicable to
      analysis types 2 and 5 only*
    * ``'freq_step_n'``        -- *frequency step number; applicable
      to analysis type 5 only*
    * ``'node_nums'``          -- *node numbers*
    * ``'r1'..'r6'``           -- *response array for each DOF; when
      response is complex only r1 through r3 will be used*
    * ``<'id1'>``              -- *id1 string*
    * ``<'id2'>``              -- *id2 string*
    * ``<'id3'>``              -- *id3 string*
    * ``<'id4'>``              -- *id4 string*
    * ``<'id5'>``              -- *id5 string*
    * ``<'model_type'>``       -- *model type number*
    * ``<'modal_m'>``          -- *modal mass; applicable to
      analysis type 2 only*
    * ``<'modal_damp_vis'>``   -- *modal viscous damping ratio;
      applicable to analysis type 2 only*
    * ``<'modal_damp_his'>``   -- *modal hysteretic damping ratio;
      applicable to analysis type 2 only*
    * ``<'modal_b'>``          -- *modal-b (complex number);
      applicable to analysis types 3 and 7 only*
    * ``<'modal_a'>``          -- *modal-a (complex number);
      applicable to analysis types 3 and 7 only*
    * ``<<'n_data_per_node'>>``-- *number of data per node (DOFs)*
    * ``<<'data_type'>>``      -- *data type number; 2 = real data,
      5 = complex data*
"""


def_init_(self, fileName):
    """
    Initializes the uff object and extract the basic info:
    the number of sets, types of the sets and format of the sets (ascii
    or binary). To manually refresh this info, call the refresh method
    manually.

    Whenever some data is written to a file, a read-only flag
    indicates that the file needs to be refreshed - before any reading,
    the file is refreshed automatically (when needed).
    """
    # Some "private" members
    self._fileName = fileName
    self._blockInd = []  # an array of block indices: start-end pairs in rows
    self._refreshed = False
    self._nSets = 0  # number of sets found in file
```

```
        self._setTypes = np.array(())  # list of set-type numbers
        self._setFormats = np.array(())# list of set-format numbers
(0=ascii,1=binary)
        # Refresh
        self.refresh()

    def get_supported_sets(self):
        """Returns a list of data-sets supported for reading and writing."""
        return _SUPPORTED_SETS

    def get_n_sets(self):
        """
        Returns the number of valid sets found in the file."""
        if not self._refreshed:
            self.refresh()
        return self._nSets

    def get_set_types(self):
        """
        Returns an array of data-set types. All valid data-sets are returned,
        even those that are not supported, i.e., whose contents will not be
        read.
        """
        if not self._refreshed:
            self.refresh()
        return self._setTypes

    def get_set_formats(self):
        """Returns an array of data-set formats: 0=ascii, 1=binary."""
        if not self._refreshed:
            self.refresh()
        return self._setFormats

    def get_file_name(self):
        """Returns the file name (as a string) associated with the uff object."""
        return self._fileName

    def file_exists(self):
        """
        Returns true if the file exists and False otherwise. If the file does
        not exist, invoking one of the read methods would raise the UFFException
        exception.
        """
        return os.path.exists(self._fileName)

    def get_status(self):
        """
        Returns the file status, i.e., True if the file is refreshed and
        False otherwise.
        """
        return self._refreshed

    def refresh(self):
        """
        Extract/refreshes the info of all the sets from UFF file (if the file
        exists). The file must exist and must be accessable otherwise, an
        error is raised. If the file cannot be refreshed, False is returned and
        True otherwise.
        """
        self._refreshed = False
```

```
        if not self.file_exists():
            return False  # cannot read the file if it does not exist
        try:
            fh = open(self._fileName, 'rb')
        #             fh = open(self._fileName,'rt')
        except:
            raise UFFException('Cannot access the file %s' % self._fileName)
        else:
            try:
                # Parses the entire file for '    -1' tags and extracts
                # the corresponding indices
                data = fh.read()
                dataLen = len(data)
                ind = -1
                blockInd = []
                while True:
                    ind = data.find(b'    -1', ind + 1)
                    if ind == -1:
                        break
                    blockInd.append(ind)
                blockInd = np.asarray(blockInd, dtype='int64')

                # Constructs block indices of start and end values; each pair
                # points to start and end offset of the data-set (block) data,
                # but, the start '    -1' tag is included while the end one is
                # excluded.
                nBlocks = int(np.floor(len(blockInd) / 2.0))
                if nBlocks == 0:
                    # No valid blocks found but the file is still considered
                    # being refreshed
                    fh.close()
                    self._refreshed = True
                    return self._refreshed
                self._blockInd = np.zeros((nBlocks, 2), dtype='int64')
                self._blockInd[:, 0] = blockInd[:-1:2].copy()
                self._blockInd[:, 1] = blockInd[1::2].copy() - 1

                # Go through all the data-sets (blocks) and extract data-set
                # type and the property whether the data-set is in binary
                # or ascii format
                self._nSets = nBlocks
                self._setTypes = np.zeros(nBlocks)
                self._setFormats = np.zeros(nBlocks)
                for ii in range(0, self._nSets):
                    si = self._blockInd[ii, 0]
                    ei = self._blockInd[ii, 1]
                    try:
                        blockData = data[si:ei + 1].splitlines()
                        self._setTypes[ii] = int(blockData[1][0:6])
                        if blockData[1][6].lower() == 'b':
                            self._setFormats[ii] = 1
                    except:
                        # Some non-valid blocks found; ignore the exception
                        pass
                del blockInd
            except:
                fh.close()
                raise UFFException('Error refreshing UFF file: ' +
self._fileName)
            else:
```

```
                self._refreshed = True
                fh.close()
                return self._refreshed

    def read_sets(self, setn=None):
        """
        Reads sets from the list or array ``setn``. If ``setn=None``, all
        sets are read (default). Sets are numbered starting at 0, ending at
        n-1. The method returns a list of dset dictionaries - as
        many dictionaries as there are sets. Unknown data-sets are returned
        empty.

        User must be sure that, since the last reading/writing/refreshing,
        the data has not changed by some other means than through the
        UFF object.
        """
        dset = []
        if setn is None:
            readRange = range(0, self._nSets)
        else:
            if (not type(setn).__name__ == 'list'):
                readRange = [setn]
            else:
                readRange = setn
        if not self.file_exists():
            raise UFFException('Cannot read from a non-existing file: ' +
self._fileName)
        if not self._refreshed:
            if not self._refresh():
                raise UFFException('Cannot read from the file: ' +
self._fileName)
        try:
            for ii in readRange:
                dset.append(self._read_set(ii))
        except UFFException as msg:
            raise UFFException('Error when reading ' + str(ii) + '-th data-set: '
+ msg.value)
        except:
            raise UFFException('Error when reading data-set(s)')
        if len(dset) == 1:
            dset = dset[0]
        return dset

    ##    def read_all_sets(self):
    ##        """Reads all the sets from UFF file.
    ##        The method returns a list of dsets each containing as many
    ##        dictionaries as there are valid sets found in the file.
    ##        """
    ##        dset = []
    ##        if not self.file_exists():
    ##            raise UFFException('Cannot read from a non-existing file:
'+self._fileName)
    ##        if not self._refreshed:
    ##            if not self._refresh():
    ##                raise UFFException('Cannot read from the file:
'+self._fileName)
    ##        try:
    ##            for ii in range(0,self._nSets):
    ##                dset.append(self._read_set(ii))
    ##        except UFFException,msg:
```

```
    ##            raise UFFException('Error when reading '+str(ii)+'-th data-set:
'+msg.value)
    ##        except:
    ##            raise UFFException('Error when reading '+str(ii)+'-th data-
set')
    ##        return dset

    def write_sets(self, dsets, mode='add'):
        """
        Writes several UFF data-sets to the file.  The mode can be
        either 'add' (default) or 'overwrite'. The dsets is a
        list of dictionaries, each representing one data-set. Unsupported
        data-sets will be ignored. When only 1 data-set is to be written, no
        lists are necessary, i.e., only one dictionary is required.

        For each data-set, there are some optional and some required fields at
        dset dictionary. Also, in general, the sum of the required
        and the optional fields together can be less then the number of fields
        read from the same type of data-set; the reason is that for some
        data-sets some fields are set automatically. Optional fields are
        calculated automatically and the dset is updated - as dset is actually
        an alias (aka pointer), this is reflected at the caller too.
        """
        if (not type(dsets).__name__== 'list'):
            dsets = [dsets]
        nSets = len(dsets)
        if nSets < 1:
            raise UFFException('Nothing towrite')
        if mode.lower() == 'overwrite':
            # overwrite mode; first set is written in the overwrite mode, others
            # in add mode
            self._write_set(dsets[0],'overwrite')
            for ii in range(1, nSets):
                self._write_set(dsets[ii], 'add')
        elif mode.lower() == 'add':
            # add mode; all the sets are written in the add mode
            for ii in range(0, nSets):
                self._write_set(dsets[ii], 'add')
        else:
            raise UFFException('Unknown mode: ' + mode)

    def _read_set(self, n):
        # Reads n-th set from UFF file. n can be an integer between 0 and nSets-
1.
        # User must be sure that, since the last reading/writing/refreshing,
        # the data has not changed by some other means than through the
        # UFF object. The method returns dset dictionary.
        dset = {}
        if not self.file_exists():
            raise UFFException('Cannot read from a non-existing file: ' +
self._fileName)
        if not self._refreshed:
            if not self.refresh():
                raise UFFException('Cannot read from the file: ' + self._fileName
+ '. The file cannot be refreshed.')
        if (n > self._nSets - 1) or (n <0):
            raise UFFException('Cannot read data-set: ' + str(int(n)) + \
                               '. Data-set number to high or to low.')
        # Read n-th data-set data (one block)
        try:
```

```
            fh = open(self._fileName, 'rb')
        except:
            raise UFFException('Cannot access the file: ' + self._fileName + ' to
read from.')
        else:
            try:
                si = self._blockInd[n][0]  # start offset
                ei = self._blockInd[n][1]  # end offset
                fh.seek(si)
                if self._setTypes[int(n)] == 58:
                    blockData = fh.read(ei - si + 1)  # decoding is handled later
in _extract58
                else:
                    blockData = fh.read(ei - si + 1).decode('utf-8',
errors='replace')
            except:
                fh.close()
                raise UFFException('Error reading data-set #: ' + int(n))
            else:
                fh.close()
        # Extracts the dset
        if self._setTypes[int(n)] == 15:
            dset = self._extract15(blockData)
        elif self._setTypes[int(n)] == 2411:
            dset = self._extract2411(blockData)  # TEMP ADD
        elif self._setTypes[int(n)] == 2412:
            dset = self._extract15(blockData)  # TEMP ADD
        elif self._setTypes[int(n)] == 18:
            dset = self._extract18(blockData)  # TEMP ADD
        elif self._setTypes[int(n)] == 82:
            dset = self._extract82(blockData)
        elif self._setTypes[int(n)] == 2420:
            dset = self._extract2420(blockData)
        elif self._setTypes[int(n)] == 151:
            dset = self._extract151(blockData)
        elif self._setTypes[int(n)] == 164:
            dset = self._extract164(blockData)
        elif self._setTypes[int(n)] == 55:
            dset = self._extract55(blockData)
        elif self._setTypes[int(n)] == 58:
            dset = self._extract58(blockData)
        else:
            dset['type'] = self._setTypes[int(n)]
            # Unsupported data-set - do nothing
            pass
        return dset

    def _write_set(self, dset, mode='add'):
        # Writes UFF data (UFF data-sets) to the file.  The mode can be
        # either 'add' (default) or 'overwrite'. The dset is a
        # dictionary of keys and corresponding values. Unsupported
        # data-set will be ignored.
        #
        # For each data-set, there are some optional and some required fields at
        # dset dictionary. Also, in general, the sum of the required
        # and the optional fields together can be less then the number of fields
        # read from the same type of data-set; the reason is that for some
        # data-sets some fields are set automatically. Optional fields are
        # calculated automatically and the dset is updated - as dset is actually
        # an alias (aka pointer), this is reflected at the caller too.
```

```
        if mode.lower() == 'overwrite':
            # overwrite mode
            try:
                fh = open(self._fileName, 'wt')
            except:
                raise UFFException('Cannot access the file: ' + self._fileName +
' to write to.')
        elif mode.lower() == 'add':
            # add (append) mode
            try:
                fh = open(self._fileName, 'at')
            except:
                raise UFFException('Cannot access the file: ' + self._fileName +
' to write to.')
        else:
            raise UFFException('Unknown mode: ' + mode)
        try:
            # Actual writing
            try:
                setType = dset['type']
            except:
                fh.close()
                raise UFFException('Data-set\'s dictionary is missing the
required \'type\' key')
            # handle nan or inf
            if 'data' in dset.keys():
                dset['data'] = np.nan_to_num(dset['data'])

            if setType == 15:
                self._write15(fh, dset)
            elif setType == 82:
                self._write82(fh, dset)
            elif setType == 151:
                self._write151(fh, dset)
            elif setType == 164:
                self._write164(fh, dset)
            elif setType == 55:
                self._write55(fh, dset)
            elif setType == 58:
                self._write58(fh, dset, mode)
            elif setType == 2411:
                self._write2411(fh, dset)
            elif setType == 2420:
                self._write2420(fh, dset)
            else:
                # Unsupported data-set - do nothing
                pass
        except:
            fh.close()
            raise  # re-raise the last exception
        else:
            fh.close()
        self.refresh()

    def _write15(self, fh, dset):
        # Writes coordinate data - data-set 15 - to an open file fh
        try:
            n = len(dset['node_nums'])
            # handle optional fields
            dset = self._opt_fields(dset, {'def_cs': np.asarray([0 for ii in
```

```
range(0, n)], 'i'),
                                          'disp_cs': np.asarray([0 for ii in
range(0, n)], 'i'),
                                          'color': np.asarray([0 for ii in
range(0, n)], 'i')})
            # write strings to the file

            fh.write('%6i\n%6i%74s\n' % (-1, 15, ' '))
            for ii in range(0, n):
                fh.write('%10i%10i%10i%10i%13.5e%13.5e%13.5e\n' % (
                    dset['node_nums'][ii], dset['def_cs'][ii], dset['disp_cs']
[ii], dset['color'][ii],
                    dset['x'][ii], dset['y'][ii],dset['z'][ii]))
            fh.write('%6i\n' % -1)
        except KeyError as msg:
            raise UFFException('The required key \'' + msg.args[0] + '\' not
present when writing data-set #15')
        except:
            raise UFFException('Error writing data-set #15')

    def _write82(self, fh, dset):
        # Writes line data - data-set 82 - to an open file fh
        try:
            # handle optional fields
            dset = self._opt_fields(dset, {'id': 'NONE',
                                          'color': 0})
            # write strings to the file
            # removed jul 2017: unique_nodes = set(dset['nodes'])
            # removed jul 2017:if 0 in unique_nodes: unique_nodes.remove(0)
            # number of changes of node need to
            # nNodes = len(dset['nodes'])
            nNodes = np.sum((dset['nodes'][1:] - dset['nodes'][:-1]) != 0) + 1
            fh.write('%6i\n%6i%74s\n' % (-1, 82, ' '))
            fh.write('%10i%10i%10i\n' % (dset['trace_num'], nNodes,
dset['color']))
            fh.write('%-80s\n' % dset['id'])
            sl = 0
            n8Blocks = nNodes // 8
            remLines = nNodes % 8
            if n8Blocks:
                for ii in range(0, n8Blocks):
                    #                 fh.write( string.join(['%10i'%lineN for
lineN in dset['lines'][sl:sl+8]],'')+'\n' )
                    fh.write(''.join(['%10i' % lineN for lineN in dset['nodes']
[sl:sl + 8]]) + '\n')
                    sl += 8
            if remLines > 0:
                fh.write(''.join(['%10i' % lineN for lineN in dset['nodes']
[sl:]]) + '\n')
            #                 fh.write( string.join(['%10i'%lineN for lineN in
dset['lines'][sl:]],'')+'\n' )
            fh.write('%6i\n' % -1)
        except KeyError as msg:
            raise UFFException('The required key \'' + msg.args[0] + '\' not
present when writing data-set #82')
        except:
            raise UFFException('Error writing data-set #82')

    def _write151(self, fh, dset):
        # Writes dset data - data-set 151 - to an open file fh
        try:
```

```
            ds = time.strftime('%d-%b-%y', time.localtime())
            ts = time.strftime('%H:%M:%S', time.localtime())
            # handle optional fields
            dset = self._opt_fields(dset, {'version_db1': '0',
                                           'version_db2': '0',
                                           'file_type': '0',
                                           'date_db_created': ds,
                                           'time_db_created': ts,
                                           'date_db_saved': ds,
                                           'time_db_saved': ts,
                                           'date_file_written': ds,
                                           'time_file_written': ts})
            # write strings to the file
            fh.write('%6i\n%6i%74s\n' % (-1, 151, ' '))
            fh.write('%-80s\n' % dset['model_name'])
            fh.write('%-80s\n' % dset['description'])
            fh.write('%-80s\n' % dset['db_app'])
            fh.write('%-10s%-10s%10s%10s%10s\n'  %  (dset['date_db_created'],
                                                     dset['time_db_created'],
dset['version_db1'], dset['version_db2'],
                                                    dset['file_type']))
            fh.write('%-10s%-10s\n' % (dset['date_db_saved'],
dset['time_db_saved']))
            fh.write('%-80s\n' % dset['program'])
            fh.write('%-10s%-10s\n' % (dset['date_file_written'],
dset['time_file_written']))
            fh.write('%6i\n' % -1)
        except KeyError as msg:
            raise UFFException('The required key \'' + msg.args[0] + '\' not
present when writing data-set #151')
        except:
            raise UFFException('Error writing data-set #151')

    def _write164(self, fh, dset):
        # Writes units data - data-set 164 - to an open file fh
        try:
            # handle optional fields
            dset = self._opt_fields(dset, {'units_description': 'User unit
system',
                                           'temp_mode': 1})
            # write strings to the file

            fh.write('%6i\n%6i%74s\n' % (-1, 164, ' '))
            fh.write('%10i%20s%10i\n' % (dset['units_code'],
dset['units_description'], dset['temp_mode']))
            str = '%25.16e%25.16e%25.16e\n%25.16e\n' % (
                dset['length'], dset['force'], dset['temp'], dset['temp_offset'])
            str = str.replace('e+', 'D+')
            str = str.replace('e-', 'D-')
            fh.write(str)
            fh.write('%6i\n' % -1)
        except KeyError as msg:
            raise UFFException('The required key \'' + msg.args[0] + '\' not
present when writing data-set #164')
        except:
            raise UFFException('Error writing data-set #164')

    def _write55(self, fh, dset):
        # Writes data at nodes - data-set 55 - to an open file fh. Currently:
        #   - only normal mode (2)
        #   - complex eigenvalue first order (displacement) (3)
```

```
        #   - frequency response and (5)
        #   - complex eigenvalue second order (velocity) (7)
        # analyses are supported.
        try:
            # Handle general optional fields
            dset = self._opt_fields(dset,
                                    {'units_description': ' ',
                                     'id1': 'NONE',
                                     'id2': 'NONE',
                                     'id3': 'NONE',
                                     'id4': 'NONE',
                                     'id5': 'NONE',
                                     'model_type': 1})
            # ... and some data-type specific optional fields
            if dset['analysis_type'] == 2:
                # normal modes
                dset = self._opt_fields(dset,
                                        {'modal_m': 0,
                                         'modal_damp_vis': 0,
                                         'modal_damp_his': 0})
            elif dset['analysis_type'] in (3, 7):
                # complex modes
                dset = self._opt_fields(dset,
                                        {'modal_b': 0.0 + 0.0j,
                                         'modal_a': 0.0 + 0.0j})
                if not np.iscomplexobj(dset['modal_a']):
                    dset['modal_a'] = dset['modal_a'] + 0.j
                if not np.iscomplexobj(dset['modal_b']):
                    dset['modal_b'] = dset['modal_b'] + 0.j
            elif dset['analysis_type'] == 5:
                # frequency response
                pass
            else:
                # unsupported analysis type
                raise UFFException('Error writing data-set #55: unsupported
analysis type')
            # Some additional checking
            dataType = 2
            #              if dset.has_key('r4') and dset.has_key('r5') and
dset.has_key('r6'):
            if ('r4' in dset) and ('r5' in dset) and ('r6' in dset):
                nDataPerNode = 6
            else:
                nDataPerNode = 3
            if np.iscomplexobj(dset['r1']):
                dataType = 5
            else:
                dataType = 2
            # Write strings to the file
            fh.write('%6i\n%6i%74s\n' % (-1, 55, ' '))
            fh.write('%-80s\n' % dset['id1'])
            fh.write('%-80s\n' % dset['id2'])
            fh.write('%-80s\n' % dset['id3'])
            fh.write('%-80s\n' % dset['id4'])
            fh.write('%-80s\n' % dset['id5'])
            fh.write('%10i%10i%10i%10i%10i%10i\n' %
                     (dset['model_type'], dset['analysis_type'], dset['data_ch'],
                      dset['spec_data_type'], dataType, nDataPerNode))
            if dset['analysis_type'] == 2:
                # Normal modes
```

```
                fh.write('%10i%10i%10i%10i\n' % (2, 4, dset['load_case'],
dset['mode_n']))
                fh.write('%13.5e%13.5e%13.5e%13.5e\n' % (dset['freq'],
dset['modal_m'],
                                                        dset['modal_damp_vis'],
dset['modal_damp_his']))
            elif dset['analysis_type'] == 5:
                # Frequenc response
                fh.write('%10i%10i%10i%10i\n' % (2, 1, dset['load_case'],
dset['freq_step_n']))
                fh.write('%13.5e\n' % dset['freq'])
            elif (dset['analysis_type'] == 3) or (dset['analysis_type'] == 7):
                # Complex modes
                fh.write('%10i%10i%10i%10i\n' % (2, 6, dset['load_case'],
dset['mode_n']))
                fh.write('%13.5e%13.5e%13.5e%13.5e%13.5e%13.5e\n' % (
                    dset['eig'].real, dset['eig'].imag, dset['modal_a'].real,
dset['modal_a'].imag,
                    dset['modal_b'].real, dset['modal_b'].imag))
            else:
                raise UFFException('Unsupported analysis type')
            n = len(dset['node_nums'])
            if dataType == 2:
                # Real data
                if nDataPerNode == 3:
                    for k in range(0, n):
                        fh.write('%10i\n' % dset['node_nums'][k])
                        fh.write('%13.5e%13.5e%13.5e\n' % (dset['r1'][k],
dset['r2'][k], dset['r3'][k]))
                else:
                    for k in range(0, n):
                        fh.write('%10i\n' % dset['node_nums'][k])
                        fh.write('%13.5e%13.5e%13.5e%13.5e%13.5e%13.5e\n' %
                                 (dset['r1'][k], dset['r2'][k], dset['r3'][k],
dset['r4'][k], dset['r5'][k],
                                  dset['r6'][k]))
            elif dataType == 5:

                # Complex data; n_data_per_node is assumed being 3
                for k in range(0, n):
                    fh.write('%10i\n' % dset['node_nums'][k])
                    fh.write('%13.5e%13.5e%13.5e%13.5e%13.5e%13.5e\n' %
                             (dset['r1'][k].real, dset['r1'][k].imag, dset['r2']
[k].real, dset['r2'][k].imag,
                              dset['r3'][k].real, dset['r3'][k].imag))
            else:
                raise UFFException('Unsupported data type')
            fh.write('%6i\n' % -1)
        except KeyError as msg:
            raise UFFException('The required key \'' + msg.args[0] + '\' not
present when writing data-set #55')
        except:
            raise UFFException('Error writing data-set #55')

    def _write58(self, fh, dset, mode='add'):
        # Writes function at nodal DOF - data-set 58 - to an open file fh.
        try:
            if not (dset['func_type'] in [1, 2, 3, 4, 6]):
                raise UFFException('Unsupported function type')
            # handle optional fields - only those that are not calculated
            # automatically
```

```
            dict = {'units_description':'',
                    'id1': 'NONE',
                    'id2': 'NONE',
                    'id3': 'NONE',
                    'id4': 'NONE',
                    'id5': 'NONE',
                    'func_id': 0,
                    'ver_num': 0,
                    'binary': 0,
                    'load_case_id': 0,
                    'rsp_ent_name': 'NONE',
                    'ref_ent_name': 'NONE',
                    'abscissa_axis_lab': 'NONE',
                    'abscissa_axis_units_lab':'NONE',
                    'abscissa_len_unit_exp': 0,
                    'abscissa_force_unit_exp': 0,
                    'abscissa_temp_unit_exp': 0,
                    'ordinate_len_unit_exp': 0,
                    'ordinate_force_unit_exp': 0,
                    'ordinate_temp_unit_exp': 0,
                    'ordinate_axis_lab': 'NONE',
                    'ordinate_axis_units_lab':'NONE',
                    'orddenom_len_unit_exp': 0,
                    'orddenom_force_unit_exp': 0,
                    'orddenom_temp_unit_exp': 0,
                    'orddenom_axis_lab': 'NONE',
                    'orddenom_axis_units_lab':'NONE',
                    'z_axis_len_unit_exp': 0,
                    'z_axis_force_unit_exp': 0,
                    'z_axis_temp_unit_exp': 0,
                    'z_axis_axis_lab': 'NONE',
                    'z_axis_axis_units_lab':'NONE',
                    'z_axis_value': 0,
                    'spec_data_type': 0,
                    'abscissa_spec_data_type':0,
                    'ordinate_spec_data_type':0,
                    'z_axis_spec_data_type': 0,
                    'version_num': 0,
                    'abscissa_spacing': 0}
            dset = self._opt_fields(dset,dict)
            # Write strings to the file - always in double precision =>
ord_data_type = 2
            # for real data and 6 for complex data
            numPts = len(dset['data'])
            isR = not np.iscomplexobj(dset['data'])
            if isR:
                # real data
                dset['ord_data_type'] = 4
                nBytes = numPts * 8
                if 'n_bytes' in dset.keys():
                    dset['n_bytes'] = nBytes
                ordDataType = dset['ord_data_type']
            else:
                # complex data
                dset['ord_data_type'] = 6
                nBytes = numPts * 8
                ordDataType = 6

            isEven = bool(dset['abscissa_spacing'])  # handling even/uneven
abscissa spacing manually
```

```
            # handling abscissa spacing automatically
            # isEven = len( set( [ dset['x'][ii]-dset['x'][ii-1] for ii in
range(1,len(dset['x'])) ] ) ) == 1
            # decode utf to ascii
            for k, v in dset.items():
                if type(v) == str:
                    dset[k] = v.encode("utf-8").decode('ascii','ignore')

            dset['abscissa_min'] = dset['x'][0]
            dx = dset['x'][1] - dset['x'][0]
            fh.write('%6i\n%6i' % (-1, 58))
            if dset['binary']:
                if sys.byteorder == 'little':
                    bo = 1
                else:
                    bo = 2
                fh.write('b%6i%6i%12i%12i%6i%6i%12i%12i\n' % (bo, 2, 11, nBytes,
0, 0, 0, 0))
            else:
                fh.write('%74s\n' % ' ')
            fh.write('%-80s\n' % dset['id1'])
            fh.write('%-80s\n' % dset['id2'])
            fh.write('%-80s\n' % dset['id3'])
            fh.write('%-80s\n' % dset['id4'])
            fh.write('%-80s\n' % dset['id5'])
            fh.write('%5i%10i%5i%10i %10s%10i%4i %10s%10i%4i\n' %
                     (dset['func_type'], dset['func_id'], dset['ver_num'],
dset['load_case_id'],
                      dset['rsp_ent_name'], dset['rsp_node'], dset['rsp_dir'],
dset['ref_ent_name'],
                      dset['ref_node'], dset['ref_dir']))
            fh.write('%10i%10i%10i%13.5e%13.5e%13.5e\n' % (ordDataType, numPts,
isEven,
                                                           isEven *
dset['abscissa_min'], isEven * dx,
                                                           dset['z_axis_value']))
            fh.write('%10i%5i%5i%5i %-20s %-20s\n' %
(dset['abscissa_spec_data_type'],


dset['abscissa_len_unit_exp'], dset['abscissa_force_unit_exp'],

dset['abscissa_temp_unit_exp'], dset['abscissa_axis_lab'],

dset['abscissa_axis_units_lab']))
            fh.write('%10i%5i%5i%5i %-20s %-20s\n' %
(dset['ordinate_spec_data_type'],

dset['ordinate_len_unit_exp'], dset['ordinate_force_unit_exp'],

dset['ordinate_temp_unit_exp'], dset['ordinate_axis_lab'],

dset['ordinate_axis_units_lab']))
            fh.write('%10i%5i%5i%5i %-20s %-20s\n' %
(dset['orddenom_spec_data_type'],

dset['orddenom_len_unit_exp'], dset['orddenom_force_unit_exp'],

dset['orddenom_temp_unit_exp'], dset['orddenom_axis_lab'],
```

```
dset['orddenom_axis_units_lab']))
            fh.write('%10i%5i%5i%5i %-20s %-20s\n' %
(dset['z_axis_spec_data_type'],

dset['z_axis_len_unit_exp'], dset['z_axis_force_unit_exp'],

dset['z_axis_temp_unit_exp'], dset['z_axis_axis_lab'],

dset['z_axis_axis_units_lab']))
            if isR:
                if isEven:
                    data = dset['data'].copy()
                else:
                    data = np.zeros(2 * numPts, 'd')
                    data[0:-1:2] = dset['x']
                    data[1::2] = dset['data']
            else:
                if isEven:
                    data = np.zeros(2 * numPts, 'd')
                    data[0:-1:2] = dset['data'].real
                    data[1::2] = dset['data'].imag
                else:
                    data = np.zeros(3 * numPts, 'd')
                    data[0:-2:3] = dset['x']
                    data[1:-1:3] = dset['data'].real
                    data[2::3] = dset['data'].imag
            # always write data in double precision
            if dset['binary']:
                fh.close()
                if mode.lower() == 'overwrite':
                    fh = open(self._fileName, 'wb')
                elif mode.lower() == 'add':
                    fh = open(self._fileName, 'ab')
                # write data
                if bo == 1:
                    [fh.write(struct.pack('<d', datai)) for datai in data]
                else:
                    [fh.write(struct.pack('>d', datai)) for datai in data]
                fh.close()
                if mode.lower() == 'overwrite':
                    fh = open(self._fileName, 'wt')
                elif mode.lower() == 'add':
                    fh = open(self._fileName, 'at')
            else:
                n4Blocks = len(data) // 4
                remVals = len(data) % 4
                if isR:
                    if isEven:
                        fh.write(n4Blocks * '%20.11e%20.11e%20.11e%20.11e\n' %
tuple(data[:4 * n4Blocks]))
                        if remVals > 0:
                            fh.write((remVals * '%20.11e' + '\n') % tuple(data[4
* n4Blocks:]))
                    else:
                        fh.write(n4Blocks * '%13.5e%20.11e%13.5e%20.11e\n' %
tuple(data[:4 * n4Blocks]))
                        if remVals > 0:
                            fmt = ['%13.5e', '%20.11e', '%13.5e', '%20.11e']
                            fh.write((''.join(fmt[remVals]) + '\n') %
tuple(data[4 * n4Blocks:]))
```

```
                else:
                    if isEven:
                        fh.write(n4Blocks * '%20.11e%20.11e%20.11e%20.11e\n' %
tuple(data[:4 * n4Blocks]))
                        if remVals > 0:
                            fh.write((remVals * '%20.11e' + '\n') % tuple(data[4
* n4Blocks:]))
                    else:
                        n3Blocks = len(data) / 3
                        remVals = len(data) % 3
                        # TODO: It breaks here for long measurements. Implement
exceptions.
                        # n3Blocks seems to be a natural number but of the wrong
type. Convert for now,
                        # but make assertion to prevent werid things from
happening.
                        if float(n3Blocks - int(n3Blocks)) != 0.0:
                            print('Warning: Something went wrong when savning the
uff file.')
                        n3Blocks = int(n3Blocks)
                        fh.write(n3Blocks * '%13.5e%20.11e%20.11e\n' %
tuple(data[:3 * n3Blocks]))
                        if remVals > 0:
                            fmt = ['%13.5e', '%20.11e', '%20.11e']
                            fh.write((''.join(fmt[remVals]) + '\n') %
tuple(data[3 * n3Blocks:]))
            fh.write('%6i\n' % -1)
            del data
        except KeyError as msg:
            raise UFFException('The required key \'' + msg.args[0] + '\' not
present when writing data-set #58')
        except:
            raise UFFException('Error writing data-set #58')

    def _write2411(self, fh, dset):
        try:
            dict = {'export_cs_number': 0,
                    'cs_color': 8}

            dset = self._opt_fields(dset, dict)
            fh.write('%6i\n%6i%74s\n' % (-1, 2411, ' '))

            for node in range(dset['grid_global'].shape[0]):
                fh.write('%10i%10i%10i%10i\n' % (dset['grid_global'][node, 0],
dset['export_cs_number'],
                                                 dset['grid_global'][node, 0],
dset['cs_color']))

                fh.write('%25.16e%25.16e%25.16e\n' % tuple(dset['grid_global']
[node, 1:]))

            fh.write('%6i\n' % -1)

        except:
            raise UFFException('Error writing data-set #2411')

    # TODO: Big deal - the output dictionary when reading this set
    #    is different than the dictionary that is expected (keys) when
    #    writing this same set. This is not OK!
    def _write2420(self, fh, dset):
```

```
        try:
            dict = {'part_UID': 1,
                    'part_name': 'None',
                    'cs_type': 0,
                    'cs_color': 8}
            dset = self._opt_fields(dset, dict)

            fh.write('%6i\n%6i%74s\n' % (-1, 2420, ' '))
            fh.write('%10i\n' % (dset['part_UID']))
            fh.write('%-80s\n' % (dset['part_name']))

            for node in range(len(dset['nodes'])):
                fh.write('%10i%10i%10i\n' % (dset['nodes'][node],
dset['cs_type'], dset['cs_color']))
                fh.write('CS%i\n' % dset['nodes'][node])
                fh.write('%25.16e%25.16e%25.16e\n' % tuple(dset['local_cs'][node
* 4, :]))
                fh.write('%25.16e%25.16e%25.16e\n' % tuple(dset['local_cs'][node
* 4 + 1, :]))
                fh.write('%25.16e%25.16e%25.16e\n' % tuple(dset['local_cs'][node
* 4 + 2, :]))
                fh.write('%25.16e%25.16e%25.16e\n' % tuple(dset['local_cs'][node
* 4 + 3, :]))

            fh.write('%6i\n' % -1)
        except:
            raise UFFException('Error writing data-set #2420')

    def _extract15(self, blockData):
        # Extract coordinate data - data-set 15.
        dset = {'type': 15}
        try:
            # Body
            split_data = blockData.splitlines()
            split_data = ''.join(split_data[2:]).split()
            split_data = [float(_) for _ in split_data]
            dset['node_nums'] = split_data[::7]
            dset['def_cs'] = split_data[1::7]
            dset['disp_cs'] = split_data[2::7]
            dset['color'] = split_data[3::7]
            dset['x'] = split_data[4::7]
            dset['y'] = split_data[5::7]
            dset['z'] = split_data[6::7]
        except:
            raise UFFException('Error reading data-set #15')
        return dset

    def _extract2411(self, blockData):
        # Extract coordinate data - data-set 15.
        dset = {'type': 15}
        try:
            # Body
            splitData = blockData.splitlines(True)  # Keep the line breaks!
            splitData = ''.join(splitData[2:])  # ..as they are again needed
            splitData = splitData.split()
            values = np.asarray([float(str) for str in splitData], 'd')
            dset['node_nums'] = values[::7].copy()
            dset['def_cs'] = values[1::7].copy()
            dset['disp_cs'] = values[2::7].copy()
            dset['color'] = values[3::7].copy()
```

```
            dset['x'] = values[4::7].copy()
            dset['y'] = values[5::7].copy()
            dset['z'] = values[6::7].copy()
        except:
            raise UFFException('Error reading data-set #15')
        return dset

    def _extract18(self, blockData):
        '''Extract local CS definitions -- data-set 18.'''
        dset = {'type': 18}
        try:
            splitData = blockData.splitlines()

            # -- Get Record 1
            rec_1 = np.array(list(map(float, ''.join(splitData[2::4]).split())))

            dset['cs_num'] = rec_1[::5]
            # removed - clutter
            # dset['cs_type'] = rec_1[1::5]
            dset['ref_cs_num'] = rec_1[2::5]
            # left out here are the parameters color and definition type
            # -- Get Record 2
            # removed because clutter
            # dset['cs_name'] = splitData[3::4]
            # -- Get Record 31 and 32
            # ... these are the origins of cs defined in ref
            #             rec_31 = np.array(list(map(float,
''.join(splitData[4::4]).split())))
            lineData = ''.join(splitData[4::4])
            rec_31 = [float(lineData[i * 13:(i + 1) * 13]) for i in
range(int(len(lineData) / 13))]
            dset['ref_o'] = np.vstack((np.array(rec_31[::6]),
                                       np.array(rec_31[1::6]),
                                       np.array(rec_31[2::6]))).transpose()

            # ... these are points on the x axis of cs defined in ref
            dset['x_point'] = np.vstack((np.array(rec_31[3::6]),
                                         np.array(rec_31[4::6]),
                                         np.array(rec_31[5::6]))).transpose()

            # ... these are the points on the xz plane
            lineData = ''.join(splitData[5::4])
            rec_32 = [float(lineData[i * 13:(i + 1) * 13]) for i in
range(int(len(lineData) / 13))]
            #             rec_32 = np.array(list(map(float,
''.join(splitData[5::4]).split())))
            dset['xz_point'] = np.vstack((np.array(rec_32[::3]),
                                          np.array(rec_32[1::3]),
                                          np.array(rec_32[2::3]))).transpose()
        except:
            raise UFFException('Error reading data-set #18')
        return dset

    def _extract2420(self, blockData):
        '''Extract local CS/transforms -- data-set 2420.'''
        dset = {'type': 2420}
        #        try:
        splitData = blockData.splitlines(True)
```

```
        # -- Get Record 1
        dset['Part_UID'] = float(splitData[2])

        # -- Get Record 2
        dset['Part_Name'] = splitData[3].rstrip()

        # -- Get Record 3
        rec_3 = list(map(int, ''.join(splitData[4::6]).split()))
        dset['CS_sys_labels'] = rec_3[::3]
        dset['CS_types'] = rec_3[1::3]
        dset['CS_colors'] = rec_3[2::3]

        # -- Get Record 4
        dset['CS_names'] = list(map(str.rstrip, splitData[5::6]))

        # !! The following part should be made smoother
        # -- Get Record 5
        row1 = list(map(float, ''.join(splitData[6::6]).split()))
        row2 = list(map(float, ''.join(splitData[7::6]).split()))
        row3 = list(map(float, ''.join(splitData[8::6]).split()))
        # !! Row 4 left out for now - usually zeros ...
        #             row4 = map(float, splitData[7::6].split())
        dset['CS_matrices'] = [np.vstack((row1[i:(i + 3)], row2[i:(i + 3)],
row3[i:(i + 3)])) \
                               for i in np.arange(0, len(row1), 3)]
        #        except:
        #             raise UFFException('Error reading data-set #2420')
        return dset

    def _extract82(self, blockData):
        # Extract line data - data-set 82.
        dset = {'type': 82}
        try:
            splitData = blockData.splitlines(True)
            dset.update(
                self._parse_header_line(splitData[2], 3, [10, 10, 10], [2, 2, 2],
['trace_num', 'n_nodes', 'color']))
            dset.update(self._parse_header_line(splitData[3], 1, [80], [1],
['id']))
            splitData = ''.join(splitData[4:])
            splitData = splitData.split()
            dset['nodes'] = np.asarray([float(str) for str in splitData])
        except:
            raise UFFException('Error reading data-set #82')
        return dset

    def _extract151(self, blockData):
        # Extract dset data - data-set 151.
        dset = {'type': 151}
        try:
            splitData = blockData.splitlines(True)
            dset.update(self._parse_header_line(splitData[2], 1, [80], [1],
['model_name']))
            dset.update(self._parse_header_line(splitData[3], 1, [80], [1],
['description']))
            dset.update(self._parse_header_line(splitData[4], 1, [80], [1],
['db_app']))
            dset.update(self._parse_header_line(splitData[5], 2, [10, 10, 10, 10,
10], [1, 1, 2, 2, 2],
```

```
                                                    ['date_db_created',
'time_db_created', 'version_db1', 'version_db2',
                                                     'file_type']))
            dset.update(self._parse_header_line(splitData[6], 1, [10, 10], [1,
1], ['date_db_saved', 'time_db_saved']))
            dset.update(self._parse_header_line(splitData[7], 1, [80], [1],
['program']))
            dset.update(
                self._parse_header_line(splitData[8], 1, [10, 10], [1, 1],
['date_file_written', 'time_file_written']))
        except:
            raise UFFException('Error reading data-set #151')
        return dset

    def _extract164(self, blockData):
        # Extract units data - data-set 164.
        dset = {'type': 164}
        try:
            splitData = blockData.splitlines(True)
            dset.update(self._parse_header_line(splitData[2], 1, [10, 20, 10],
[2, 1, 2],
                                                    ['units_code',
'units_description', 'temp_mode']))
            splitData =''.join(splitData[3:])
            splitData = splitData.split()

            dset['length'] = float(splitData[0].lower().replace('d', 'e'))
            dset['force'] = float(splitData[1].lower().replace('d', 'e'))
            dset['temp'] = float(splitData[2].lower().replace('d', 'e'))
            dset['temp_offset'] = float(splitData[3].lower().replace('d', 'e'))
        except:
            raise UFFException('Error reading data-set #164')
        return dset

    def _extract55(self, blockData):
        # Extract data at nodes - data-set 55. Currently:
        #   - only normal mode (2)
        #   - complex eigenvalue first order (displacement) (3)
        #   - frequency response and (5)
        #   - complex eigenvalue second order (velocity) (7)
        # analyses are supported.
        dset = {'type': 55}
        try:
            splitData = blockData.splitlines(True)
            dset.update(self._parse_header_line(splitData[2], 1, [80], [1],
['id1']))
            dset.update(self._parse_header_line(splitData[3], 1, [80], [1],
['id2']))
            dset.update(self._parse_header_line(splitData[4], 1, [80], [1],
['id3']))
            dset.update(self._parse_header_line(splitData[5], 1, [80], [1],
['id4']))
            dset.update(self._parse_header_line(splitData[6], 1, [80], [1],
['id5']))
            dset.update(self._parse_header_line(splitData[7], 6, [10, 10, 10, 10,
10, 10], [2, 2, 2, 2, 2, 2],
                                                    ['model_type', 'analysis_type',
'data_ch', 'spec_data_type',
                                                     'data_type',
'n_data_per_node']))
            if dset['analysis_type'] == 2:
```

```
                # normal mode
                dset.update(self._parse_header_line(splitData[8], 4, [10, 10, 10,
10, 10, 10, 10, 10],
                                                    [-1, -1, 2, 2, -1, -1, -1,
-1],
                                                    ['', '', 'load_case',
'mode_n', '', '', '', '']))

                dset.update(self._parse_header_line(splitData[9], 4, [13, 13, 13,
13, 13, 13], [3, 3, 3, 3, -1, -1],
                                                    ['freq', 'modal_m',
'modal_damp_vis', 'modal_damp_his', '', '']))
            elif (dset['analysis_type'] == 3) or (dset['analysis_type'] == 7):
                # complex eigenvalue
                dset.update(self._parse_header_line(splitData[8], 4, [10, 10, 10,
10, 10, 10, 10, 10],
                                                    [-1, -1, 2, 2, -1, -1, -1,
-1],
                                                    ['', '', 'load_case',
'mode_n', '', '', '', '']))

                dset.update(self._parse_header_line(splitData[9], 4, [13, 13, 13,
13, 13, 13], [3, 3, 3, 3, 3, 3],
                                                    ['eig_r', 'eig_i',
'modal_a_r', 'modal_a_i', 'modal_b_r',
                                                     'modal_b_i']))
                dset.update({'modal_a': dset['modal_a_r'] + 1.j *
dset['modal_a_i']})
                dset.update({'modal_b': dset['modal_b_r'] + 1.j *
dset['modal_b_i']})
                dset.update({'eig': dset['eig_r'] + 1.j * dset['eig_i']})
                del dset['modal_a_r'], dset['modal_a_i'], dset['modal_b_r'],
dset['modal_b_i']
                del dset['eig_r'], dset['eig_i']
            elif dset['analysis_type'] == 5:
                # frequency response
                dset.update(self._parse_header_line(splitData[8], 4, [10, 10, 10,
10, 10, 10, 10, 10],
                                                    [-1, -1, 2, 2, -1, -1, -1,
-1],
                                                    ['', '', 'load_case',
'freq_step_n', '', '', '', '']))

                dset.update(self._parse_header_line(splitData[9], 1, [13, 13, 13,
13, 13, 13], [3, -1, -1, -1, -1, -1],
                                                    ['freq', '', '', '', '',
'']))
                # Body
            splitData = ''.join(splitData[10:])

            values = np.asarray([float(str) for str in splitData.split()], 'd')
            if dset['data_type'] == 2:
                # real data
                if dset['n_data_per_node'] == 3:
                    dset['node_nums'] = values[:-3:4].copy()
                    dset['r1'] = values[1:-2:4].copy()
                    dset['r2'] = values[2:-1:4].copy()
                    dset['r3'] = values[3::4].copy()
```

```
            else:
                    dset['node_nums'] = values[:-6:7].copy()
                    dset['r1'] = values[1:-5:7].copy()
                    dset['r2'] = values[2:-4:7].copy()
                    dset['r3'] = values[3:-3:7].copy()
                    dset['r4'] = values[4:-2:7].copy()
                    dset['r5'] = values[5:-1:7].copy()
                    dset['r6'] = values[6::7].copy()


                    elif dset['data_type'] == 5:
                # complex data
                if dset['n_data_per_node'] == 3:
                    dset['node_nums'] = values[:-6:7].copy()
                    dset['r1'] = values[1:-5:7] + 1.j * values[2:-4:7]
                    dset['r2'] = values[3:-3:7] + 1.j * values[4:-2:7]
                    dset['r3'] = values[5:-1:7] + 1.j * values[6::7]
                else:
                    raise UFFException('Cannot handle 6 points per node and
complex data when reading data-set #55')

            else:
                raise UFFException('Error reading data-set #55')

        except:
            raise UFFException('Error reading data-set #55')
        del values

        return dset

    def _extract58(self, blockData):
        # Extract function at nodal DOF - data-set 58.
        dset = {'type': 58, 'binary': 0}
        try:
            binary = False
            split_header = b''.join(blockData.splitlines(True)[:13]).decode('utf-
8',  errors='replace').splitlines(True)
            if len(split_header[1]) >= 7:
                if split_header[1][6].lower() == 'b':
                    # Read some addititional fields from the header section
                    binary = True
                    dset['binary'] = 1
                    dset.update(self._parse_header_line(split_header[1], 6,
                    [6,1, 6, 6, 12, 12, 6, 6, 12, 12],
                    [-1, -1, 2, 2, 2, 2, -1, -1, -1, -1],
                   ['', '', 'byte_ordering', 'n_bytes', '', '', '',
                   'fp_format', 'n_ascii_lines', '']))


                    dset['r5'] = values[5:-1:7].copy()
                    dset['r6'] = values[6::7].copy()
            elif dset['data_type'] == 5:
                # complex data
                if dset['n_data_per_node'] == 3:
                    dset['node_nums'] = values[:-6:7].copy()
                    dset['r1'] = values[1:-5:7] + 1.j * values[2:-4:7]
                    dset['r2'] = values[3:-3:7] + 1.j * values[4:-2:7]
                    dset['r3'] = values[5:-1:7] + 1.j * values[6::7]
```

```
                    else:
                    raise UFFException('Cannot handle 6 points per node and
complex data when reading data-set #55')

            else:
                raise UFFException('Error reading data-set #55')

        except:
            raise UFFException('Error reading data-set #55')
        del values
        return dset

    def _extract58(self, blockData):
        # Extract function at nodal DOF - data-set 58.
        dset = {'type': 58, 'binary': 0}

        try:
            binary = False
            split_header = b''.join(blockData.splitlines(True)[:13]).decode('utf-
8',  errors='replace').splitlines(True)

            if len(split_header[1]) >= 7:
                if split_header[1][6].lower() == 'b':

                    # Read some addititional fields from the header section
                    binary = True
                    dset['binary'] = 1
                    dset.update(self._parse_header_line(split_header[1], 6, [6,
1, 6, 6, 12, 12, 6, 6, 12, 12],
                                                                    [-1, -1, 2, 2, 2, 2, -1,
-1, -1, -1],
                                                                    ['', '', 'byte_ordering',
'fp_format', 'n_ascii_lines',
                                                                     'n_bytes', '', '', '',
'']))

            dset.update(self._parse_header_line(split_header[2], 1, [80], [1],
['id1']))
            dset.update(self._parse_header_line(split_header[3], 1, [80], [1],
['id2']))
            dset.update(self._parse_header_line(split_header[4], 1, [80], [1],
['id3']))  # usually for the date
            dset.update(self._parse_header_line(split_header[5], 1, [80], [1],
['id4']))
            dset.update(self._parse_header_line(split_header[6], 1, [80], [1],
['id5']))
            dset.update(self._parse_header_line(split_header[7], 1, [5, 10, 5,
10, 11, 10, 4, 11, 10, 4],
                                                            [2, 2, 2, 2, 1, 2, 2, 1, 2, 2],
                                                            ['func_type', 'func_id',
'ver_num', 'load_case_id','rsp_ent_name',
                                                             'rsp_node', 'rsp_dir',
'ref_ent_name',
                                                             'ref_node', 'ref_dir']))

            dset.update(self._parse_header_line(split_header[8], 6, [10, 10, 10,
13, 13, 13], [2, 2, 2, 3, 3, 3],
                                                            ['ord_data_type', 'num_pts',
```

```
'abscissa_spacing', 'abscissa_min',
                                                'abscissa_inc',
'z_axis_value']))


            dset.update(self._parse_header_line(split_header[9], 4, [10, 5, 5, 5,
21, 21], [2, 2, 2, 2, 1, 1],
                                               ['abscissa_spec_data_type',
'abscissa_len_unit_exp',
                                                'abscissa_force_unit_exp',
'abscissa_temp_unit_exp',
                                                'abscissa_axis_lab',
'abscissa_axis_units_lab']))



            dset.update(self._parse_header_line(split_header[10], 4, [10, 5, 5,
5, 21, 21], [2, 2, 2, 2, 1, 1],
                                               ['ordinate_spec_data_type',
'ordinate_len_unit_exp',
                                                'ordinate_force_unit_exp',
'ordinate_temp_unit_exp',
                                                'ordinate_axis_lab',
'ordinate_axis_units_lab']))



            dset.update(self._parse_header_line(split_header[11], 4, [10, 5, 5,
5, 21, 21], [2, 2, 2, 2, 1, 1],
                                               ['orddenom_spec_data_type',
'orddenom_len_unit_exp',
                                                'orddenom_force_unit_exp',
'orddenom_temp_unit_exp',
                                                'orddenom_axis_lab',
'orddenom_axis_units_lab']))



            dset.update(self._parse_header_line(split_header[12], 4, [10, 5, 5,
5, 21, 21], [2, 2, 2, 2, 1, 1],
                                               ['z_axis_spec_data_type',
'z_axis_len_unit_exp',
                                                'z_axis_force_unit_exp',
'z_axis_temp_unit_exp','z_axis_axis_lab',
                                                'z_axis_axis_units_lab']))
            # Body
```

```
            # splitData = ''.join(splitData[13:])
            if binary:
                split_data = b''.join(blockData.splitlines(True)[13:])
                if dset['byte_ordering'] == 1:
                    bo = '<'
                else:
                    bo = '>'
                if (dset['ord_data_type'] == 2) or (dset['ord_data_type'] == 5):

                # single precision - 4 bytes
                    values = np.asarray(struct.unpack('%c%sf' % (bo,
int(len(split_data) / 4)), split_data), 'd')
                else:
                    # double precision - 8 bytes
                    values = np.asarray(struct.unpack('%c%sd' % (bo,
int(len(split_data) / 8)), split_data), 'd')

            else:
                values = []
                split_data = blockData.decode('utf-8',
errors='replace').splitlines(True)[13:]
                if (dset['ord_data_type'] == 2) or (dset['ord_data_type'] == 5):
                    for line in split_data:  # '6E13.5'
                        values.extend([float(line[13 * i:13 * (i + 1)]) for i in
range(len(line) // 13)])
                elif ((dset['ord_data_type'] == 4) or (dset['ord_data_type'] ==
6)) and (dset['abscissa_spacing'] == 1):


                    for line in split_data:  # '4E20.12'
                        values.extend([float(line[20 * i:20 * (i + 1)]) for i in
range(len(line) // 20)])
                elif (dset['ord_data_type'] == 4) and (dset['abscissa_spacing']
== 0):
                    for line in split_data:  # 2(E13.5,E20.12)
                        values.extend(
                            [float(line[13 * (i + j) + 20 * (i):13 * (i + 1) + 20
* (i + j)]) \
                             for i in range(len(line) // 33) for j in [0, 1]])
                elif (dset['ord_data_type'] == 6) and (dset['abscissa_spacing']
== 0):
                    for line in split_data:  # 1E13.5,2E20.12
                        values.extend([float(line[0:13]), float(line[13:33]),
float(line[33:53])])


                else:
                    raise UFFException('Error reading data-set #58b; not proper
data case.')


                values = np.asarray(values)
                # values = np.asarray([float(str) for str in splitData],'d')
            if (dset['ord_data_type'] == 2) or (dset['ord_data_type'] == 4):
                # Non-complex ordinate data
                if (dset['abscissa_spacing'] == 0):
                    # Uneven abscissa
                    dset['x'] = values[:-1:2].copy()
                    dset['data'] = values[1::2].copy()
```

```
                else:
                # Even abscissa
                nVal = len(values)
                minVal = dset['abscissa_min']
                d = dset['abscissa_inc']
                dset['x'] = np.arange(minVal, minVal + nVal * d, d)
                dset['data'] = values.copy()
        elif (dset['ord_data_type'] == 5) or (dset['ord_data_type'] == 6):
            # Complex ordinate data
            if (dset['abscissa_spacing'] == 0):
                # Uneven abscissa
                dset['x'] = values[:-2:3].copy()
                dset['data'] = values[1:-1:3] + 1.j * values[2::3]
            else:
                # Even abscissa
                nVal = len(values) / 2
                minVal = dset['abscissa_min']
                d = dset['abscissa_inc']
                dset['x'] = np.arange(minVal, minVal + nVal * d, d)
                dset['data'] = values[0:-1:2] + 1.j * values[1::2]
        del values

        except:
        raise UFFException('Error reading data-set #58b')
    return dset

def _opt_fields(self, dict, fieldsDict):
    # Sets the optional fields of the dict dictionary. Optionaly fields are
    # given in fieldsDict dictionary.
    for key in fieldsDict:
        #           if not dict.has_key(key):
        if not key in dict:
            dict.update({key: fieldsDict[key]})
    return dict



def _parse_header_line(self, line, minValues, widths, types, names):
    # Parses the given line (a record in terms of UFF file) and returns all
    # the fields. line is a string representing the whole line.
    # width are fields widths to be read, types are field types
    #   1=string, 2=int, 3=float, -1=ignore the field
    # while names is a list of key (field) names.
    # Fields are split according to their widths as given in widths.
    # minValues specifies how many values (filds) are mandatory to read
    # from the line. Also, number of values found must not exceed the
    # number of fields requested by fieldsIn.
    # On the output, a dictionary of field names and corresponding
    # filed values is returned.
    fields = {}
    nFieldsReq = len(names)
    fieldsFromLine = []
    fieldsOut = {}

    # Extend the line if shorter than 80 chars
    ll = len(line)
    if ll < 80:
        line = line + ' ' * (80 - ll)
    # Parse the line for fields
```

```
        si = 0
        for n in range(0, len(widths)):
            fieldsFromLine.append(line[si:si + widths[n]].strip())
            si += widths[n]
        # Check for the number of fields,...
        nFields = len(fieldsFromLine)
        if (nFieldsReq < nFields) or (minValues > nFields):
            raise UFFException('Error parsing header section; too many or to
less' + \
                               'fields found')
        # Mandatory fields

        for key, n in zip(names[:minValues], range(0, minValues)):
            if types[n] == -1:
                pass
            elif types[n] == 1:
                fieldsOut.update({key: fieldsFromLine[n]})
            elif types[n] == 2:
                fieldsOut.update({key: int(fieldsFromLine[n])})
            else:
                fieldsOut.update({key: float(fieldsFromLine[n])})
        # Optional fields
        for key, n in zip(names[minValues:nFields], range(minValues, nFields)):
            try:
                if types[n] == -1:
                    pass
                elif types[n] == 1:
                    fieldsOut.update({key: fieldsFromLine[n]})
                elif types[n] == 2:
                    fieldsOut.update({key: int(fieldsFromLine[n])})
                else:
                    fieldsOut.update({key: float(fieldsFromLine[n])})
            except ValueError:
                if types[n] == 1:
                    fieldsOut.update({key: ''})
                elif types[n] == 2:
                    fieldsOut.update({key: 0})
                else:
                    fieldsOut.update({key: 0.0})
        return fieldsOut

def prepare_test_15(save_to_file=''):
    dataset = {'type': 15,  # Nodes
               'node_nums': [16, 17, 18, 19, 20],  # I10, node label
               'def_cs': [11, 11, 11, 12, 12],  # I10, definition coordinate
system number
               'disp_cs': [16, 16, 17, 18, 19],  # I10, displacement coordinate
system number
               'color': [1, 3, 4, 5, 6],  # I10, color
               'x': [0.0, 1.53, 0.0, 1.53, 0.0],  # E13.5
               'y': [0.0, 0.0, 3.84, 3.84, 0.0],  # E13.5
               'z': [0.0, 0.0, 0.0, 0.0, 1.83]}  # E13.5
    dataset_out = dataset.copy()

    if save_to_file:
        if os.path.exists(save_to_file):
            os.remove(save_to_file)
        uffwrite = UFF(save_to_file)
        uffwrite._write_set(dataset, 'add')
```

```
    return dataset_out

def prepare_test_55(save_to_file=''):
    if save_to_file:
        if os.path.exists(save_to_file):
            os.remove(save_to_file)
    uff_datasets = []
    modes = [1, 2, 3]
    node_nums = [1, 2, 3, 4]
    freqs = [10.0, 12.0, 13.0]
    for i, b in enumerate(modes):
        mode_shape = np.random.normal(size=len(node_nums))
        name = 'TestCase'
        data = {
            'type': 55,
            'model_type': 1,
            'id1': 'NONE',
            'id2': 'NONE',
            'id3': 'NONE',
            'id4': 'NONE',
            'id5': 'NONE',
            'analysis_type': 2,
            'data_ch': 2,
            'spec_data_type': 8,
            'data_type': 2,
            'data_ch': 2,
            'r1': mode_shape,
            'r2': mode_shape,
            'r3': mode_shape,
            'n_data_per_node': 3,
            'node_nums': [1, 2, 3, 4],
            'load_case': 1,
            'mode_n': i + 1,
            'modal_m': 0,
            'freq': freqs[i],
            'modal_damp_vis': 0,
            'modal_damp_his': 0,
        }

        uff_datasets.append(data.copy())
        if save_to_file:
            uffwrite = UFF(save_to_file)
            uffwrite._write_set(data, 'add')
    return uff_datasets

def prepare_test_58(save_to_file=''):
    if save_to_file:
        if os.path.exists(save_to_file):
            os.remove(save_to_file)

    uff_datasets = []
    binary = [0, 1, 0]  # ascii of binary
    frequency = np.arange(10)
    np.random.seed(0)
    for i, b in enumerate(binary):
        print('Adding point {}'.format(i + 1))
        response_node = 1
        response_direction = 1
        reference_node = i + 1
        reference_direction = 1
```

```
        # this is an artificial 'frf'
        acceleration_complex = np.random.normal(size=len(frequency)) + \
                               1j * np.random.normal(size=len(frequency))


        name = 'TestCase'

        data = {'type': 58,
                'binary': binary[i],
                'func_type': 4,
                'rsp_node': response_node,
                'rsp_dir': response_direction,
                'ref_dir': reference_direction,
                'ref_node': reference_node,
                'data': acceleration_complex,
                'x': frequency,
                'id1': 'id1',
                'rsp_ent_name': name,
                'ref_ent_name': name,
                'abscissa_spacing': 1,
                'abscissa_spec_data_type':  18,
                'ordinate_spec_data_type':  12,
                'orddenom_spec_data_type': 13}
        uff_datasets.append(data.copy())

        if save_to_file:
            uffwrite = UFF(save_to_file)
            uffwrite._write_set(data, 'add')

    return uff_datasets


def prepare_test_82(save_to_file=''):
    dataset = {'type': 82,  # Tracelines
               'trace_num': 2,  # I10, trace line number
               'n_nodes': 7,  # I10, number of nodes defining trace line (max
250)

               'color': 30,  # I10, color
               'id': 'Identification line',  # 80A1, Identification line
               'nodes': np.array([0, 10, 13, 14, 15, 16, 17]),
               # I10, nodes defining trace line:



               #  > 0 draw line to node
               #  = 0 move to node
               }
    dataset_out = dataset.copy()

    if save_to_file:
        if os.path.exists(save_to_file):
            os.remove(save_to_file)
        uffwrite = UFF(save_to_file)
        uffwrite._write_set(dataset, 'add')

    return dataset_out
```

```
def prepare_test_151(save_to_file=''):
    dataset = {'type': 151,  # Header
               'model_name': 'Model file name',  # 80A1, model file name
               'description': 'Model file description',  # 80A1, model file
description
               'db_app': 'Program which created DB',  # 80A1, program which
created DB
               'date_db_created': '27-Jan-16',  # 10A1, date database created
(DD-MMM-YY)
               'time_db_created': '14:38:15',  # 10A1, time database created
(HH:MM:SS)
               'version_db1': 1,# I10, Version from  database
               'version_db2': 2,# I10, Subversion from database
               'file_type': 0,  # I10, File type (0  Universal, 1 Archive, 2
Other)
               'date_db_saved': '28-Jan-16',  # 10A1, date database saved (DD-
MMM-YY)
               'time_db_saved': '14:38:16',  # 10A1, time database saved
(HH:MM:SS)
               'program': 'OpenModal',  # 80A1, program which created DB
               'date_db_written': '29-Jan-16',  # 10A1, date database written
(DD-MMM-YY)
               'time_db_written': '14:38:17',  # 10A1, time database written
(HH:MM:SS)
               }

    dataset_out = dataset.copy()

    if save_to_file:
        if os.path.exists(save_to_file):
            os.remove(save_to_file)
        uffwrite = UFF(save_to_file)
        uffwrite._write_set(dataset, 'add')

    return dataset_out


def prepare_test_164(save_to_file=''):
    dataset = {'type': 164,  # Universal Dataset
               'units_code': 1,  # I10, units code
               'units_description': 'SI units',  # 20A1, units description
               'temp_mode': 1,  # I10, temperature mode
               # Unit factors
               # for converting universal file units to SI.
               # To convert from universal file units to SI divide by
               # the appropriate factor listed below.
               'length': 3.28083989501312334,  # D25.17, length
               'force': 2.24808943099710480e-01,  # D25.17, force
               'temp': 1.8,  # D25.17, temperature
               'temp_offset': 459.67,  # D25.17, temperature offset
               }
    dataset_out = dataset.copy()

    if save_to_file:
        if os.path.exists(save_to_file):
            os.remove(save_to_file)
        uffwrite = UFF(save_to_file)
        uffwrite._write_set(dataset, 'add')

    return dataset_out
```

```
if __name__ == '__main__':
    uff_ascii = UFF('./data/beam.uff')
    a = uff_ascii.read_sets(0)
    print(a)
    prepare_test_55('./data/test_uff55.uff')
    # uff_ascii = UFF('./data/Artemis export - Geometry
RPBC_setup_05_14102016_105117.uff')
    uff_ascii = UFF('./data/no_spacing2_UFF58_ascii.uff')
    a = uff_ascii.read_sets(0)
    for _ in a.keys():
        if _ != 'data':
            print(_, ':', a[_])
    print(sum(a['data']))
```

ANEXO V.

CÓDIGO EN PYTHON PARA INICIALIZAR OPENMODAL

```
# Copyright (C) 2014-2017 Matjaž Mršnik, Miha Pirnat, Janko Slavič, Blaž Starc
#
# This file is part of OpenModal.
#
# OpenModal is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 3 of the License.
#
# OpenModal is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with OpenModal. If not, see <http://www.gnu.org/licenses/>.


import sys, time, os
import multiprocessing as mp

##
if __name__ == '__main__':
# executable = os.path.join(os.path.dirname(sys.executable), 'openmodal.exe')
# mp.set_executable(executable)
# mp.freeze_support()

from PyQt5 import QtGui, QtWidgets, QtWebEngineWidgets

class Logger(object):

        def __init__(self, filename):
                self.terminal = sys.stderr
                self.log = open(filename, 'w')

        def write(self, message):
                self.terminal.write(message)
                self.log.write(message)

        def flush(self):
                self.terminal.flush()
                self.log.close()

        if os.path.isdir('log'):
        pass
        else:
                os.mkdir('log')

#sys.stderr = Logger('log/{0:.0f}_log.txt'.format(time.time()))

sys.path.append('../')
```

```
if __name__ == '__main__':
        app = QtWidgets.QApplication(sys.argv)
        #TODO: do we need the following?
        #app.addLibraryPath('c:/Anaconda3/Lib/site-packages/PyQt5/plugins/')

        #pixmap = QtGui.QPixmap('gui/widgets/splash.png')
        #splash = QtGui.QSplashScreen(pixmap)
        #splash.show()

        #splash.showMessage('Importing modules ...')
         app.processEvents()


import gui.skeleton as sk

main_window = sk.FramelesContainer(app.desktop())
#splash.showMessage('Building environment ...')
app.processEvents()

main_window.show()

#splash.finish(main_window)


#sys.exit(app.exec_())

app.exec()
```

ANEXO VI.

MODOS DE VIBRACIÓN OBTENIDOS EN ANSYS

Primer modo de vibración en ANSYS, se trata de un modo flexionante a una frecuencia natural de 11,62 Hz.

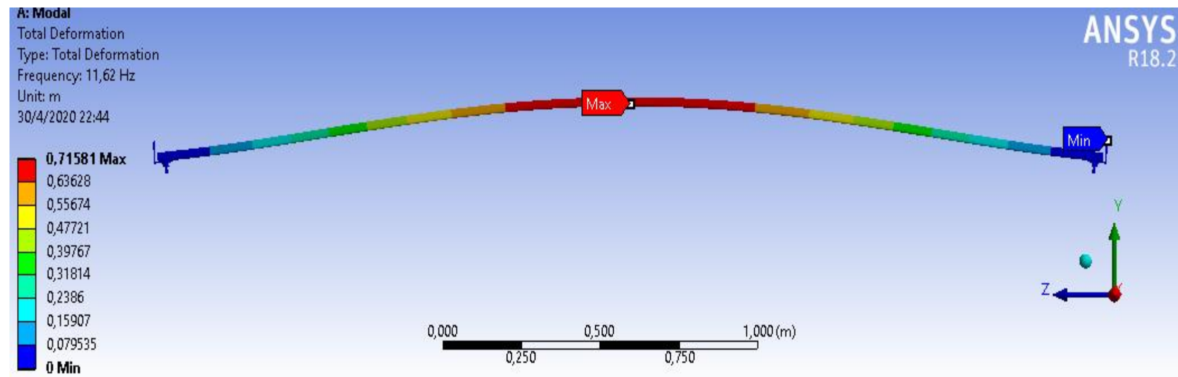


Figura VI.1. Vista lateral del primer modo de vibración en ANSYS.
(Fuente: Propia)

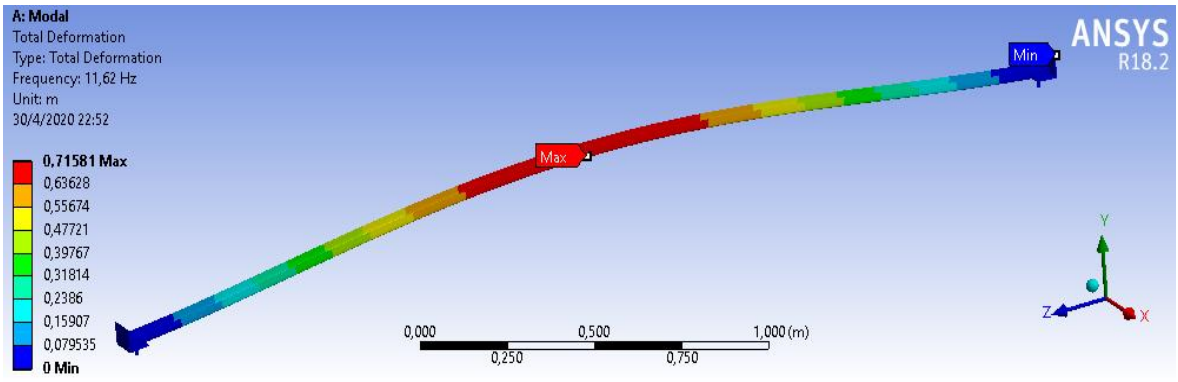


Figura VI.2. Vista isométrica del primer modo de vibración en ANSYS.
(Fuente: Propia)

Segundo modo de vibración en ANSYS, se trata de una combinación de un modo lateral y torsional, a una frecuencia natural de 18,466 Hz.

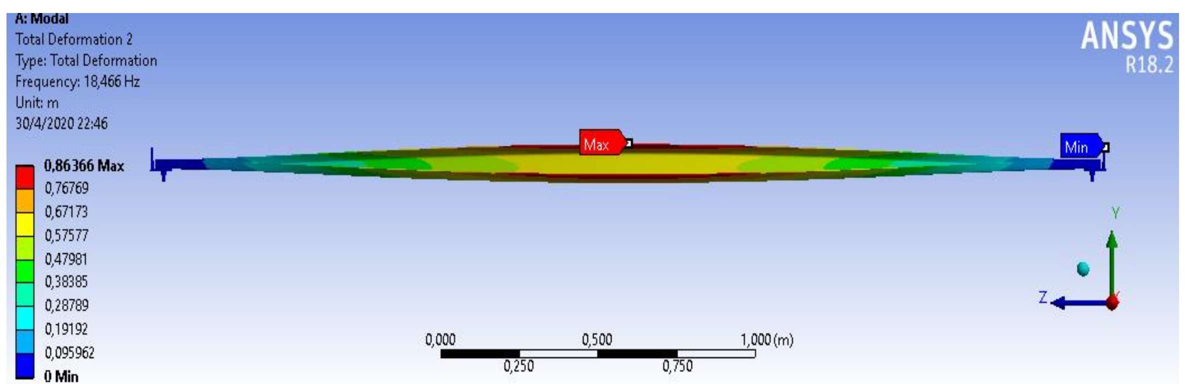


Figura VI.3. Vista lateral del segundo modo de vibración en ANSYS.
(Fuente: Propia)

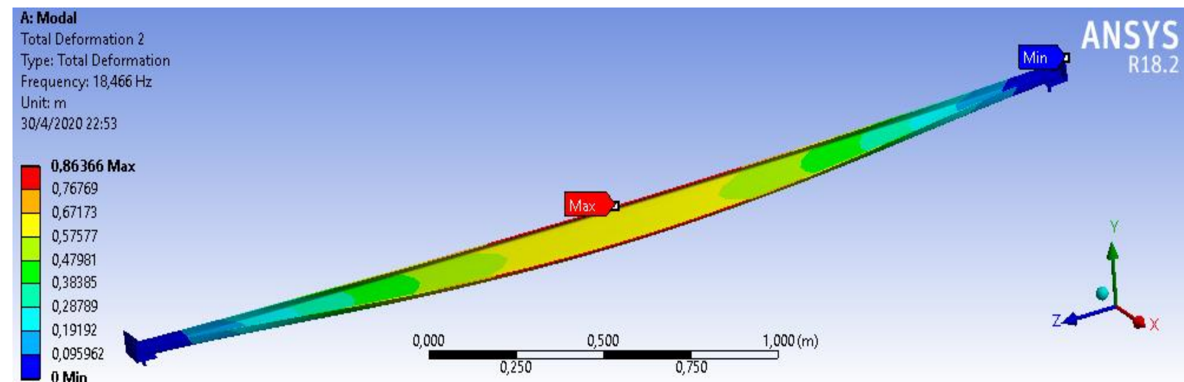


Figura VI.4. Vista isométrica del segundo modo de vibración en ANSYS.
(Fuente: Propia)

Tercer modo de vibración en ANSYS, se trata de un modo flexionante a una frecuencia natural de 29,928 Hz.

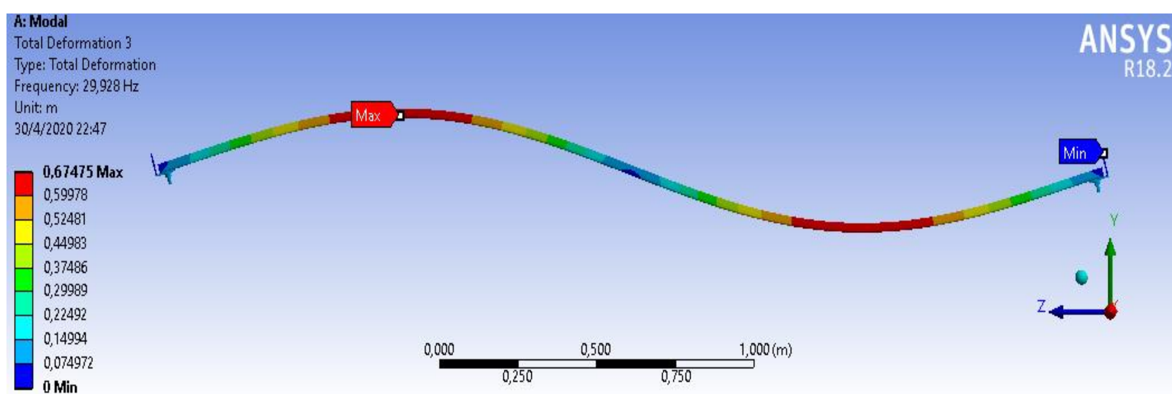


Figura VI.5. Vista lateral del tercer modo de vibración en ANSYS.
(Fuente: Propia)

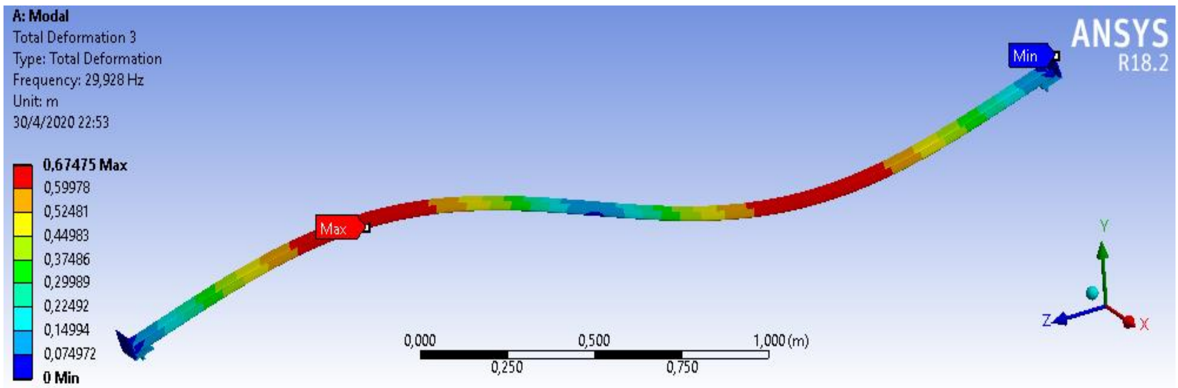


Figura VI.6. Vista isométrica del tercer modo de vibración en ANSYS.
(Fuente: Propia)

Cuarto modo de vibración en ANSYS, se trata de un modo torsional a una frecuencia natural de 39,426 Hz.

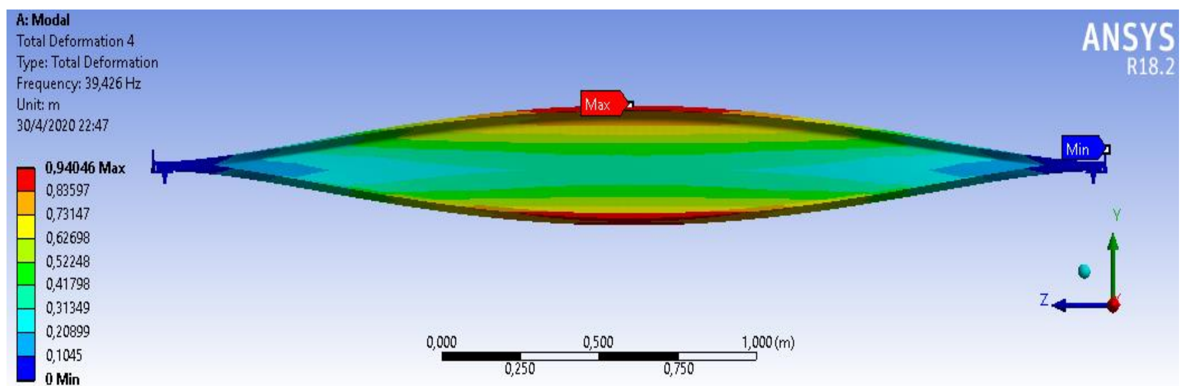


Figura VI.7. Vista lateral del cuarto modo de vibración en ANSYS.
(Fuente: Propia)

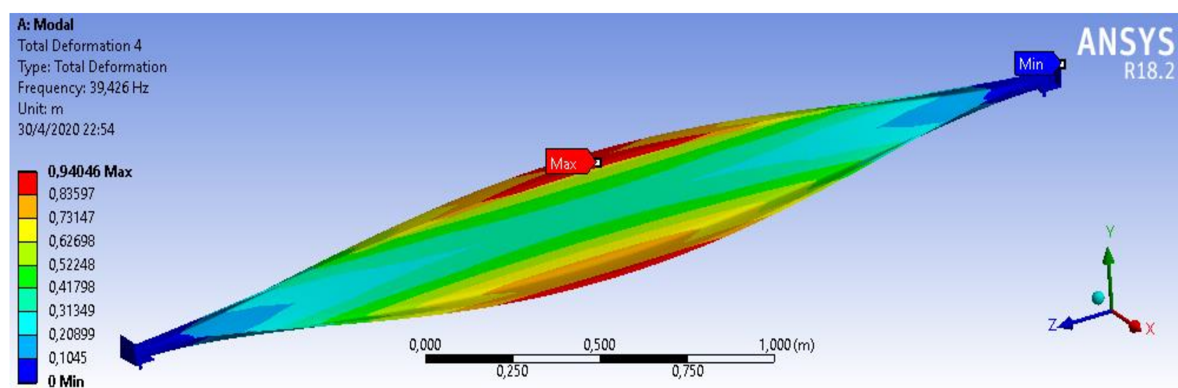


Figura VI.8. Vista isométrica del cuarto modo de vibración en ANSYS.
(Fuente: Propia)

Quinto modo de vibración en ANSYS, se trata de un modo torsional a una frecuencia natural de 51,241 Hz.

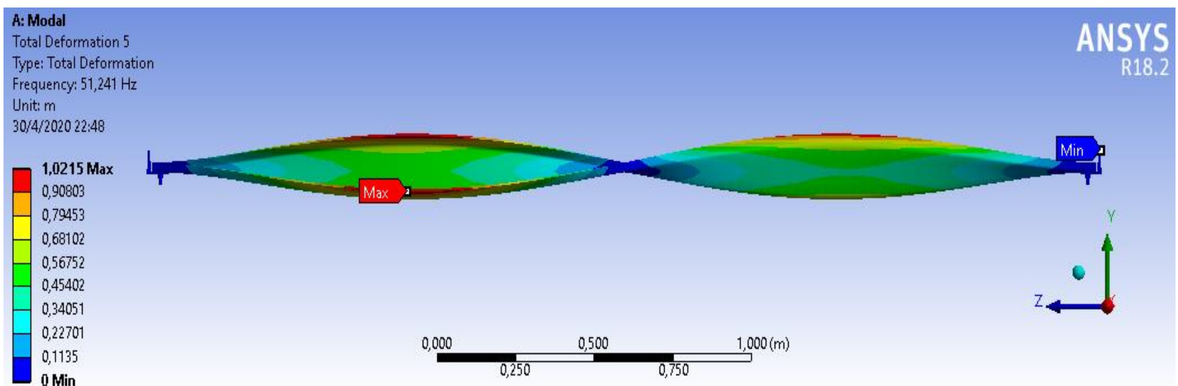


Figura VI.9. Vista lateral del quinto modo de vibración en ANSYS.
(Fuente: Propia)

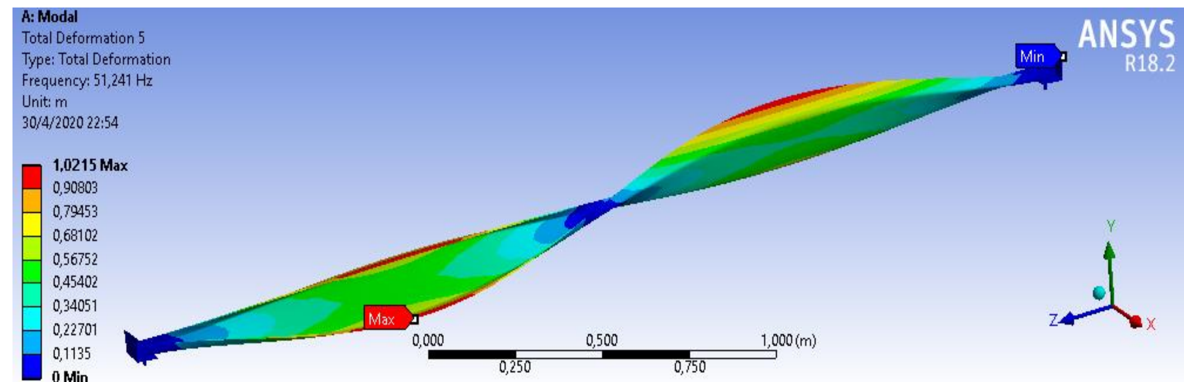


Figura VI.10. Vista isométrica del quinto modo de vibración en ANSYS.
(Fuente: Propia)

Sexto modo de vibración en ANSYS, se trata de un modo flexionante a una frecuencia natural de 68,974 Hz.

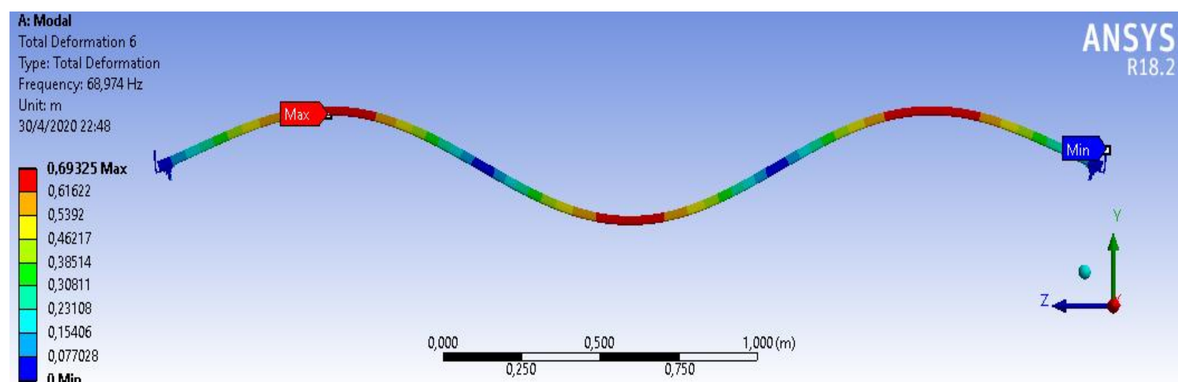


Figura VI.11. Vista lateral del sexto modo de vibración en ANSYS.
(Fuente: Propia)

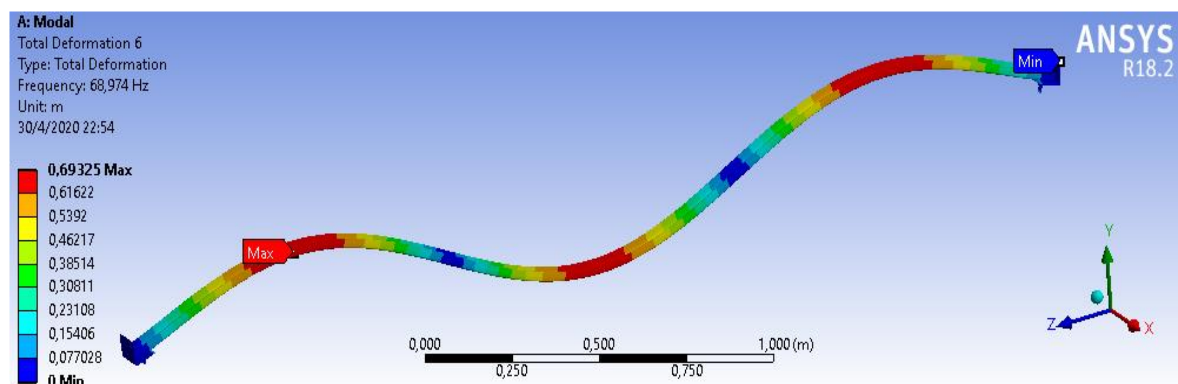


Figura VI.12. Vista isométrica del sexto modo de vibración en ANSYS.
(Fuente: Propia)