Politecnico di Torino

Master's degree in Computer Engineering

Master Degree Thesis

Development of an Open-Source Python Software for Structural Health Monitoring with Advanced Signal Processing Techniques

Supervisors:
Prof. G. C. Marano

Co-supervisors:
Ph.D. Marco Martino Rosso
Ph.D. Dag Pasquale Pasca
Ph.D. Angelo Aloisio

Candidate:
Diego Federico Margoni

Student's ID:
S295483

December 2024
A.Y. 2023/2024

Abstract

The current dissertation document presents the development of an open-source Python library designed for Operational Modal Analysis (OMA). The main objective of `pyOMA2` package is to provide an open-source, user-friendly software ecosystem whereby researchers and engineers may benefit from easy access to advanced and state-of-the-art OMA techniques. The growing demand for OMA in the built environment, as an essential instrument for structural health monitoring, has underlined the need for easily available computational tools; `pyOMA2` attempts to cover this gap by using the Python programming language's versatility and simplicity.

This dissertation document begins with a starting overview of the current panorama of OMA methods and the already available tools nowadays, pointing out their constraints in terms of accessibility, cost, and adaptability. This background helps the reader to appreciate why an open-source strategy is advantageous, particularly one that is developed using the vast library ecosystem of Python programming language. The modular design of `pyOMA2`'s allows users to combine several analysis approaches, therefore offering a flexible toolkit for preprocessing, analysis, and visualization of OMA data.

The software design and architecture of `pyOMA2`'s are thoroughly discussed, especially focusing on system components including data ingestion, preprocessing modules, algorithmic layers, and visualizing tools. A significant effort was invested in the software's best practices and design principles which ensure the ease of code extension and maintainability, therefore enabling the package to be fit for both professional and educational use. Unified Modeling Language (UML) software diagrams have been included to illustrate the architecture of the main component of the library and the main workflow of the most common operations applied to structural data. Moreover, UML diagrams have also been conceived to be a valid reference for future developers and researchers. Afterward, the document analyzes an overview of the development process, including version control, continuous integration, and deployment, unit, and integration testing.

Furthermore, this document discusses the primary subject of licensing in the open-source field, including its benefits and highlighting how collaborative development can accelerate innovation in the development of the package. The dissertation ends with the presentation of a real case study and a discussion of possible opportunities for further improvements to `pyOMA2`, including the expansion of the package with new OMA techniques and the implementation of modifications to optimize its reliability.

Acknowledgements

Before I begin, I would like to mention that these acknowledgments are written in Italian as they are mainly intended for those who have accompanied me on this journey, that are Italian speakers.

Desidero iniziare questi ringraziamenti esprimendo la mia più sincera gratitudine al Professor G.C. Marano, supervisore di questa tesi, per aver accettato di affidarmi questo progetto.

Un ringraziamento speciale va al dott. Marco Martino Rosso, che con pazienza e grande disponibilità mi ha aiutato a comprendere le complesse basi teoriche dell'Operational Modal Analysis astraendole il più possibile dalla loro complessità, rendendo più chiaro ogni passaggio di questo lungo percorso.

Inoltre, desidero ringraziare il dott. Dag Pasquale Pasca, con il quale ho lavorato fianco a fianco per circa un anno alla scrittura della libreria, un'esperienza che è stata fondamentale non solo per il risultato finale ma anche per la mia crescita professionale. Vorrei anche menzionare il dott. Angelo Aloisio, il cui contributo, nella ricerca del settore ha avuto impatto anche su questo lavoro.

Rivolgo un pensiero speciale ai miei figli, Isabella, Mattia e Christian. A loro chiedo scusa per ogni minuto sottratto al loro tempo e dedicato al raggiungimento di questo obiettivo. Nonstante questo, spero che possano trarre da questa impresa un importante insegnamento: non esiste un tempo giusto per fare le cose, ognuno deve seguire i propri ritmi, senza sentirsi obbligato a rispettare quelli dettati dalla società, sempre con la docuta responsabilità e consapevolezza. Vorrei inoltre che capiscano che con il giusto impegno e sacrificio qualsiasi obiettivo è raggiungibile, se ci si crede abbastanza.

Un sentito grazie va alla mia famiglia, mia Sorella Emanuela e mio fratello Fabio, a mia nonna e a tutte le persone che mi hanno sostenuto lungo questo percorso. In particolare, alla mia mamma, che mi ha trasmesso la passione per l'informatica. So che avrebbe voluto essere qui per condividere questo traguardo con me.

Infine chiedo se sia pensabile che un obiettivo così individuale si possa raggiungere in due; sì, nel mio caso è assolutamente vero. Senza il sostegno e l'aiuto di Corinne, la mia compagna di vita e mamma dei miei figli, non sarei mai riuscito a completare questo percorso. A lei va il mio più profondo ringraziamento per aver creduto in me e per aver reso possibile questo sogno.

Contents

List of Figures

Chapter 1

Introduction

1.1 Significance of Operational Modal Analysis (OMA)

In structural engineering, Operational Modal Analysis (OMA) is defined as a technique used to determine the dynamic characteristics of a structure, such as *natural frequencies*, *damping ratios*, and *mode shapes*, based only on output measurements [1]. It has become a fundamental tool, especially for structural health monitoring (SHM). Whereas Experimental Modal Analysis (EMA) depends on both input and output measurements under controlled settings, OMA depends just on the output data acquired during the normal structural operations. OMA is the recommended technique in many practical applications since this basic difference offers several important benefits.

OMA's capacity to perform modal analysis free from artificial excitation is one of its main advantages, in fact this makes it especially appropriate for massive and complicated buildings such bridges, high-rise buildings, and industrial machinery, where providing regulated input forces is impractical or impossible. OMA can continually monitor the dynamic behavior of buildings by means of ambient vibrations and operational loads, therefore offering real-time information on their health and performance [2, 3].

OMA's effectiveness and affordability help to emphasize even more its relevance. Traditional EMA techniques, usually require significant downtime and investments in resources to set up and run tests, while on the other hand, OMA can be carried out without interfering with the usual operation of the construction, therefore lowering expenses and minimizing disruptions, making OMA the best approach for applications where preserving the structural operational integrity is crucial.

Moreover, OMA is quite efficient in determining the modal parameters of buildings under active operating conditions, this produces more accurate and consistent outcomes, where EMA might only reflect the behavior of the structure under particular test settings. By means of monitoring structures in their natural context, engineers can identify changes in modal parameters as diagnosis of damage or deterioration, therefore enabling proper maintenance and repairs interventions [4, 3].

Another reason that motivates the wide adoption of OMA techniques, is the overall enhancement in data processing methods and sensor technologies, this allows modern OMA technologies to collect and evaluate structural responses with great accuracy by using advanced algorithms and highly precise sensors. Developed to improve the dependability and robustness of OMA, different algorithms have been studied by the researchers of this field, some of these, that will be recalled later, are Stochastic Subspace Identification (`SSI`), Frequency Domain Decomposition (`FDD`), and the Poly-reference Least Squares Complex Frequency (`pLSCF`).

In addition OMA can be employed to monitor critical sensitive structures, such as nuclear power reactors, offshore platforms, and aeronautical constructions. This additional OMA's capacity for continuous observation in hazardous or impractical environments enhances public safety and mitigates the risk of catastrophic failures, which rely on the structural integrity of these edifices. Take bridge monitoring as an example, which is among the most striking illustrations of OMA's importance, where dynamic stresses such as traffic, wind, and seismic activity are prevalent on bridges, which can cause vibrations and compromise structural stability. OMA allows for continuous monitoring of these rapid responses, enabling early detection of potential issues such as cracks, corrosion, or other forms of deterioration. This proactive maintenance technique greatly increases the lifetime of bridges and lower the possibility of abrupt breakdown [4, 3, 5].

Moreover, the integration of OMA with contemporary data collecting and processing tools has produced the design of cutting-edge SHM systems. These systems may gather, process, and automatically create data analytics which gives engineers real-time knowledge of structural condition of tracked resources, including machine learning algorithms, wireless sensor networks, and cloud-based data storage has fundamentally improved OMA's capability, establishing it as a fundamental instrument in structural engineering.

In conclusion, by means of early identification of possible problems and real-time monitoring, OMA increases the lifetime of buildings, lowers maintenance costs, and helps to prevent disastrous collapses.

1.2 Importance of Open Source Software for OMA Tools

In recent years, with the spreading acceptance of OMA in all engineering fields, considerably more software tools are being created to support its implementation. Broadly speaking, however, most of the tools have historically been proprietary — which means that they are less available to a wider audience and prone to becoming stranded in different implementations. The landscape has changed greatly with the introduction of these tools in the open-source software environment, which allows people to collaborate on projects, making everything transparent and subsequently lowers cost, with the added benefit of being able to customize the software and rely on a community of

users for support.

To accelerate the time that it takes to validate an algorithm, open-source tools offer three key benefits: customization, transparency, and cost reduction.

Customization helps in providing researchers and practitioners to take a further step by moulding the capabilities to suit their individual requirements. This level of customization can be extremely valuable in the realm of OMA, as there are many applications with unique demands. Open-source software helps in fostering user driven innovation by allowing them to change the source code directly and develop new methodologies and techniques more quickly.

Open-source software enhances transparency by allowing users to examine the code and understand all the steps involved; this is crucial for verifying results and trusting analyses. Additionally, it fosters community trust, encourages learning best practices, and further promotes openness.

The other major advantage is the decrease in costs. As open-source tools are free to access and users do not need refined analysis-specific licenses for every simulation, the more complex OMA techniques can be available instead of restricting their use only to industry or larger engineering organizations. This makes the technology increasingly accessible to educational programs and general use in OMA practices. We will explore these aspects further in Chapter 7.

One representation of these advantages are the `PyOMA` (https://github.com/dagghe/PyOMA) and its more advanced version, `pyOMA2` (https://github.com/dagghe/pyOMA2) (object of this dissertation), which are open-source libraries for OMA, fully implemented in Python. These Python-based libraries leverage robust class structures and rich scientific computing capabilities, providing a comprehensive environment for OMA. `pyOMA2` specifically has a lot of improvements introduced with respect to its predecessor, it is indeed designed to process data from both single and multi-setup measurements, provides interactive plotting options as well supports complex visualizations, not forgetting the rework done on the whole project structure to make it more modular.

Open-source projects, because they are by nature collaborative and transparent, give rise to a more dynamic community of end users and developers. By being community-driven, this method constantly improves the software whenever users contribute to code base or report a bug or even requests for new functionality. This collective wisdom allows the software to grow in response to user needs and advancing technologies.

This transition of OMA tools from being mainly proprietary software to open-source is certainly remarkable. This continued evolution and increased uptake of open-source OMA tools such as `pyOMA2` continues to demonstrate the crucial role that open-source software plays in furthering engineering applications. [2].

1.3 Thesis Objectives and Scope

The primary objective of this thesis is to develop an open-source Python library designed for Structural Health Monitoring utilizing advanced modal analysis techniques, especially for Operational Modal Analysis; by incorporating existing signal processing techniques, this library seeks to be a modular, expandable, user-friendly tool for SHM community researchers and practitioners, delivering a strong and useful implementation of fundamental OMA techniques including Frequency Domain Decomposition (FDD), Enhanced Frequency Domain Decomposition (EFDD), and Stochastic Subspace Identification (SSI).

Important library development objectives include the design, aimed to be effective and able of managing real-world situations, the library will provide implementations of the fundamental OMA techniques mentioned earlier, as well as additional methods. Following software engineering best practices means that they will be used all through the development process. This covers creating comprehensive user documentation, keeping track and managing changes using version control, and running exhaustive tests to guarantee the dependability and stability of the library.

Practical case studies examples will help to validate the library's effectiveness and consistency by means of comparisons with several SHM situations, these case studies will show the actual relevance of the library.

This thesis therefore aims to create a state-of-the-art Python open-source library for SHM, based on recent signal processing developments in OMA. The project intends to serve as a useful resource for the SHM community through adopting best practices in software engineering and benchmarking it with practical case studies, while acknowledging its ongoing requirement for further improvements and optimizations concerning limitations related to initial handiness level of users and algorithm implementations.

1.4 Document Structure

This thesis is structured as follows:

Chapter 2: Literature Review: Provides an overview of existing OMA methods and tools, with a focus on Python and non-Python libraries relevant to OMA

Chapter 3: Why Python?: Discusses the advantages of using Python for developing OMA tools, including its environment, package management, and dependency handling capabilities.

Chapter 4: Software Design and Architecture: Outlines the system architecture, including modular design and design patterns applied in the development of the library.

Chapter 5: Software Unified Modeling Language (UML) diagrams: Presents the UML diagrams of the library's architecture and workflow, including class diagrams and sequence diagrams.

Chapter 6: Development Process and Testing: Covers the technology stack, development stages, version control, and testing methodologies adhered to during the software development.

Chapter 7: Open Source Licensing and Benefits: Discusses the importance of open-source licensing and the benefits of collaborative development in the context of the library.

Chapter 8: Case Study and Examples: Presents real-world applications to demonstrate the library's capabilities.

Chapter 9: Conclusion and Future Work: Summarizes the key contributions of the thesis and discusses potential improvements and future developments.

Chapter 2

Literature Review

2.1 Existing OMA Methods and Tools

The dynamic characteristics of structures are identified by OMA using only output data obtained during the structure's normal operation. In this chapter we'll review the key OMA methods and tools, focusing on Peak Picking (PP), Frequency Domain Decomposition (FDD), Enhanced Frequency Domain Decomposition (EFDD), Frequency-Spatial Domain Decomposition (FSDD), poly-reference Least Square Complex Frequency (pLSCF), and Stochastic Subspace Identification (SSI) methods. We'll also have a look to single setup and multi-setup techniques including PoSER and PreGER merging strategies [2, 4, 3, 6].

1. **Peak Picking (PP)**: Among the first and simplest methods applied in OMA is the Peak Picking (PP) method. Operated on frequency domain, it detects peaks in the Power Spectral Density (PSD) plot to determine the natural frequencies of the system. This approach is most appreciated when structural modes shapes are well-separated (therefore the natural frequencies of these modes are significantly distinct from each other [7]) and damping is minimal (damping refers to energy dissipation within the structure during vibration, usually due to internal friction, material qualities, or external forces like air resistance). However, in the presence of closely spaced modes—common in complicated constructions like bridges—it can produce inaccurate results. More sophisticated methods such Frequency Domain Decomposition (FDD) have been developed to get over this limitation.

2. **Frequency Domain Decomposition (FDD)**: Frequency Domain Decomposition (FDD) improves the Peak Picking technique by finding the natural frequencies and mode shapes of the system by means of the Singular Value Decomposition (SVD) of the PSD matrix. FDD is an approach known as non-parametric whereby it does not depend on a predefined mathematical model of the system. This approach still suffers in identifying closely spaced modes shapes. FDD, however,

cannot compute damping ratios despite its simple and strong characteristics; the damping ratio is a dimensionless measurement that describes how oscillations in a system decay after a disturbance [8], and it is an important parameter in structural dynamics, so this limitation has led to the development of more advanced methods.

3. **Enhanced Frequency Domain Decomposition (EFDD)**: The Enhanced Frequency Domain Decomposition (EFDD) technique was developed as a natural consequence of the limitation of FDD that we just presented. EFDD offers an approach to more precisely estimate damping ratios, hence expanding on the fundamental Frequency Domain Decomposition (FDD) technique. It begins by identifying peaks in the frequency domain, corresponding to the resonance frequencies for the structure, these peaks are associated to specific modes of vibration; EFDD uses these frequency-domain signals and an inverse Fourier Transform to translate them back into the time domain so obtaining more detail picture of each mode. This phase lets engineers see over time how every mode performs, by analyzing these time-domain signals, they can determine the natural frequencies by counting how often the signal crosses zero within a certain period. About damping ration, EFDD looks at how rapidly the vibrations decrease over time, this is accomplished by computing the logarithmic reduction in amplitude of the signal, therefore measuring the oscillation rate; here a faster decrease denotes stronger damping. EFDD provides better distinction between closely spaced mode compared to the FDD technique and it is generally more accurate in determining natural frequencies, however, it still has difficulties in accurately calculating modal damping.

4. **Frequency-Spatial Domain Decomposition (FSDD)**: Built on the Enhanced Frequency Domain Decomposition (EFDD) approach, Frequency-Spatial Domain Decomposition (FSDD) is an improved method in operational modal analysis, it is more precise and detailed in the output results with respect to EFDD. Its primary objective is to increase the precision of damping ratio estimation, particularly in cases involving closed space modes. FSDD improves the analysis by adding both spatial information from the measured data together and frequency.

5. **Poly-reference Least Square Complex Frequency (pLSCF)**: The (pLSCF) method—also known as PolyMAX— is an effective instrument for determining modal parameters, it works particularly well for buildings with closely spaced modes or when handling a high modal frequency density. pLSCF works fitting a mathematical model to the observed vibration data via least-squares approach, in the complex frequency domain; it improves its capacity to precisely depict the dynamics of the structure by simultaneously considering several references or measuring points, thus known as "poly-reference". The way pLSCF generates

simple, understandable stabilization diagrams is one of its strongest points. These diagrams enable researchers to find consistent modal parameters across several model orders, therefore facilitating the identification of real structural modes from numerical artifacts. This method is extensively used in practical applications since of its great accuracy and computational efficiency.

6. **Stochastic Subspace Identification (SSI)**: In operational modal analysis, stochastic subspace identification (SSI) is a powerful time-domain technique, especially useful in circumstances where data may be noisy or when modes are closely spaced. As unlike frequency-domain methods it operates directly with time-series measurements. SSI uses two basic strategies:: Data-Driven SSI (SSI-DATA) and Covariance-Driven SSI (SSI-COV).

   (a) **Data-Driven SSI (SSI-DATA)**: Data-driven SSI (SSI-DATA) is a technique that estimates a state-space model of the system directly using measured response data. One of its most relevant benefits is that it eliminates the necessity of having pre-processing processes like, for instance, the computation of correlation functions. This method can produce consistent estimates of modal parameters—such as natural frequencies, damping ratios, and mode shapes—by matching the model straight to the raw data, which makes it well-suited for situations with significant noise in the data; this direct approach eliminates possible biases from intermediary computations, therefore enabling the capture of the actual dynamics of the structure.

   (b) **Covariance-Driven SSI (SSI-COV)**: Covariance-Driven SSI (SSI-COV) generates the state-space model by means of the covariance of the time-series data. Especially in large datasets, SSI-COV becomes computationally efficient by concentrating on the statistical characteristics of the measurements. By means of the covariance matrix analysis, it allows to efficiently detect the dynamic characteristics of the system, in particular this approach is pretty useful in context of systems having large dimensionality, so it can be considered in situations where computational resources could be a limiting factor since it uses statistical methods to reduce the necessary modal information; data dimensionality is however often reduced with techniques like Principal Component Analysis (PCA).

   Despite of the specific method, both SSI-DATA and SSI-COV call for several important steps: parameter tuning, pole estimation, stabilization diagram analysis, and result evaluation. Current developments in SSI techniques are trying to automate these tasks in order to increases the resilience and efficiency of the identification methods by reducing the possible human error and manual work.

Using the outcomes of these OMA techniques leads us on considering the different data collecting setup strategies.

Capturing the whole dynamic behavior of buildings depends on both *single* and *multi-setup* methods, particularly in big or complicated systems, these methods are meant to guarantee accurate modal parameter estimation and thorough data collecting, even when a single setup is not enough to capture the whole dynamic behavior of the structure [9, 10].

1. **Single Setup Techniques**: Single setup methods are those where data from the examined structure is gathered using a predefined set of sensors. Although these techniques are rather easy to apply and evaluate, they might not be able to fully capture the whole dynamic behavior of large and complicated structures, it is usually utilized for smaller buildings or in cases when the dynamic behavior of the structure is fairly uniform across the structure. The main advantage of these single setup methods lies in their straightforward application and analysis, but their limitation is often related to the number and placement of sensors. In larger or more complex structures, a fixed number of sensors might not be enough to record all the important modes of vibration.

2. **Multi-Setup Techniques**: Multi-setup techniques imply using multiple sets of sensors, where some sensors are kept fixed, and others are relocated between setups, so data is gathered iteratively during different acquisitions. This method is indeed more appropriate for large or complex structures where a single configuration cannot capture the complete dynamic behavior and is especially useful when the available set of sensors is not enough to cover all the modes of the structure.

   Multi-setup methods necessitate complex data merging and normalizing processes to guarantee consistency across setups since they demand accurate knowledge of the dynamics of the structure.

   Merging Strategies: Merging techniques are therefore needed for combining data from several configurations which are absolutely essential to obtain detailed global modal parameters. Two widely used merging techniques are PreGER (Pre Global Estimation Re-scaling) and PoSER (Post Separate Estimation Re-scaling).

   (a) **Post Separate Estimation Re-scaling**: In the PoSER approach, modal parameters are estimated independently for every configuration and subsequently re-scaled and merged together to generate a global mode shape. Although this is a simple approach, it can be laborious in cases of many configurations or when working with closely spaced modes. The re-scaling is typically performed using a least-squares method on the reference sensor part (hence on the sensors that are common to all setups) of each partial mode shape to ensure consistency across setups.

   (b) **PreGER (Pre Global Estimation Re-scaling)**: The PreGER method scales the Power Spectral Densities (PSDs) from several configurations to a common reference, before modal parameter estimation, so with respect to

the PoSER method, PreGER applies immediately a global estimation of the modal parameters. This approach guarantees that all configurations have the same poles, therefore facilitating the data merging process and obtaining the global modal parameter. PreGER has the advantage in processing all configurations concurrently, which can be more resilient against non-stationary excitation levels and can be also more efficient. PreGER lowers the probability of discrepancies resulting from different excitation conditions among setups by leveling the data before the identification stage.

The decision on adopting one between these OMA techniques ultimately depends on the particular requirements of the analysis, such as the computational resources available, the complexity of the structure under study, and the amount of detail needed to estimate damping. The goal of creating a new Python library for OMA is to address existing limitations and improve the capabilities of OMA experts in a more approachable way by offering a unified, user-friendly tool that combines the best features of these various approaches.

2.2 Relevant Libraries for OMA (Python and Non-Python)

Operational modal analysis has developed significantly and is now used in many civil engineering areas thanks to the development of multiple software tools supporting it. These commercial and open-source software tools can process data obtained from different kinds of sensors and return the inherent frequencies, modal shapes, and damping factors and visualization tools of the structures that are being studied.

In this section, we will examine some of the most widely used software in order to present a picture of the current market and the main alternatives available besides the software developed in this thesis. We will briefly outline the functional characteristics and key features of each of them.

- **PyOMA** (PyOMA-1) https://github.com/dagghe/PyOMA: the first version of the library presented in this dissertation, worth of mentioning despite the improvements and new features of the second version, it is still a valuable open-source Python library designed specifically for Operational Modal Analysis. `PyOMA` provides a comprehensive suite of tools for conducting OMA, including algorithms for FDD and SSI family of algorithms. The library supports only single-setup configurations and offers interactive plotting capabilities for mode shape visualization. The usage of the library is supported by a graphical user interface (GUI) that simplifies the process of conducting OMA studies in particular for users with limited programming experience.

  Features:

  - Algorithms: `PyOMA` includes different OMA techniques such as FDD, EFDD, FSDD, and both covariance-driven and data-driven SSI.
  - Interactive Plotting: The library facilitates interactive plotting, allowing users to choose modes straight from the plots produced by the algorithm.
  - Geometry Definition: Additionally, it allows users to define the geometry of the structures they are analyzing, which improves the visualization and comprehension of mode shapes.
  - Open-Source
  - GUI: The most relevant additional value of this library is that it includes a graphical user interface (GUI) that simplifies the process of conducting OMA studies.

- **PyEMA** https://pyema.readthedocs.io/en/latest/: `PyEMA` is a Python library focused on Experimental Modal Analysis (EMA), even if it is not OMA focused it is worth mentioning. It has been mentioned as an inspiration for `PyOMA`, as stated in [2]. `PyEMA` provides tools for modal parameter estimation using various techniques, making it a valuable resource for EMA applications.

  Features:

  - Algorithms: `PyEMA` supports several EMA techniques, which can be adapted for OMA applications.
  - Interactive Plotting: Similar to `PyOMA`, `PyEMA` offers interactive plotting capabilities for mode selection and visualization.
  - Open-Source

- **Artemis** https://www.svibs.com/operational-modal-analysis: `Artemis` is a commercial software tool developed by Structural Vibration Solutions. Unlike the open-source libraries, `Artemis` is a paid tool, offering a wide range of sophisticated features and algorithms for modal analysis.

  Features:

  - Algorithms: `Artemis` supports a complete set of OMA techniques, including FDD, EFDD, SSI, and more. It is known for its robust performance and accuracy in modal parameter estimation.
  - User Interface: The software provides a user-friendly interface with advanced visualization tools, making it accessible for both novice and experienced users.
  - Support and Updates: As a commercial product, `Artemis` comes with professional support and regular updates, ensuring users have access to the latest advancements in OMA technology.

- **KOMA** https://github.com/knutankv/koma: `Koma` is an open-source Python library available on GitHub. It provides tools for OMA, focusing on simplicity and ease of use. `Koma` is designed to be a lightweight alternative to more general libraries like `PyOMA`, making it suitable for smaller projects or educational purposes.

  Features:

  - Algorithms: `Koma` includes basic OMA techniques such as FDD and SSI, providing essential functionalities for modal analysis.
  - Ease of Use: The library is designed to be user-friendly, with straightforward functions and minimal setup requirements.
  - Open-Source

As we just saw, the field of structural dynamics has been substantially improved by the advancements made in the creation of both commercial and open-source OMA softwares. Open-source libraries have encouraged creativity and cooperation by making complex OMA techniques more widely available to a larger audience, on the other hand, commercial software, offers advanced features and professional support. Every tool has its own advantages, and the user's particular needs and available resources will determine which program is best for them.

Chapter 3

Why Python?

Several elements influenced our choice to build our OMA module with Python. Above all, Python's natural benefits—flexibility, simplicity of learning, and strong community support—made it a clear pick. Python's interpreted nature and dynamic typing enable fast development cycles, therefore enabling quick feature validation and minimization of development overhead.

Still another important factor, was the continuity with the initial library version `PyOMA` that was itself developed in Python. By using the same programming language, we not only maintain the consistency and familiarity for current users and contributors, but also build the new library on a strong basis, improving and refining features as we go forward.

Using well-known Python libraries like `numpy` for numerical operations and `pandas`, between other, for data manipulation and analysis, we could leverage the full power of Python's great ecosystem.

Furthermore, Python's large standard library offers a wide range of modules and tools right out of the box, therefore drastically cutting the time needed to apply basic features from fresh start, and in particular the vast choice of scientific libraries, for example, high-quality graphs and visualizations, made possible by modules such as `matplotlib` or `pyvista`, were crucial for properly evaluating and presenting modal analysis data.

The active Python community adds still another great benefit to the language, since participating in a dynamic ecosystem guarantees ongoing development and access to a wealth of knowledge via websites, repositories, and cooperative tools. The main enhancements, feature proposal and bug fixes are well documented in the Python Enhancement Proposals (PEP) and the Python Issue Tracker, which makes it easy to follow the development of the language and its libraries. The PEP are relevant documents that outline new features, enhancements, or changes to the Python programming language, they serve as a central repository for Python's development process, providing a structured and organized way to propose, discuss, and document improvements. Each PEP is thoroughly reviewed and discussed by the Python community and core

developers before being accepted or rejected.

PEPs can cover a wide range of topics, including language syntax and semantics, library and module enhancements, and even processes and conventions for Python development. Well-known and worth mentioning PEPs include:

1. PEP 8 https://www.python.org/dev/peps/pep-0008/: Style Guide for Python Code, which provides guidelines for writing clean and readable Python code.

2. PEP 20 https://www.python.org/dev/peps/pep-0020/: The Zen of Python, which outlines the philosophy and design principles of Python.

3. PEP 257 https://www.python.org/dev/peps/pep-0257/: Docstring Conventions, which defines conventions for writing and formatting docstrings in Python code.

4. PEP 484 https://www.python.org/dev/peps/pep-0484/: Type Hints, which introduces optional type annotations for Python code.

The purpose of PEPs is to ensure transparency and community involvement in the evolution of Python. By documenting proposals and their rationales in PEPs, the Python community ensures that changes are well-considered, discussed extensively, and agreed upon before implementation.

All things considered, our OMA library's natural strengths and our aim to keep consistency with our first development efforts drove our choice of Python, this decision lets us make use of Python's extensive environment, support group development, and guarantee that our tools stay easily available and potent for the engineering community.

3.1 Introduction to the Python Environment

Python is a high-level, interpreted, and general-purpose programming language, it was created by Guido van Rossum in 1991 and it was immediately known for its code readability and simplicity. These characteristics are probably the reason of its popularity in particular among Machine Learning and Data Scientist community since it is able to abstract the complexity of the subject from the complexity of writing code. [11].

Here we will explore more technical features of Python that make it the best fit for the development of our library.

Python is, firstly and most importantly, an interpreted language, this means that Python code runs line by line, which makes it possible to have instantaneous feedback on the code being executed which turns into to a facilitating development and debugging process, despite the fact that interpreted languages are generally slower than compiled languages, however, speed in the development process is a major benefit since it lets developers test and iterate fast, therefore allowing fast prototyping and troubleshooting.

As anticipated, Python applications, on the other hand, typically run slower than compiled languages like C++ or Java and other compiled languages, this is because, in

these languages, code is translated into machine language before the execution, leading to faster execution time and lighter resource usage. Development speed and runtime performance are the trade-off here, although for many OMA applications the agility in development and the various and wide support of Python libraries exceeds the cost of somewhat lowered execution speed.

About typing, that in programming languages is the process of assigning a type to a variable, the dynamic type mechanism of Python adds still more versatility. Python finds types at runtime unlike statically-typed languages, in which each variable must be declared with a type that indicate what type of data the variable can hold, the same applies to functions in terms of what type of arguments they can accept and what type of values they will return. This saves boilerplate and enables more compact code; however, it also means that problems or inconsistencies caused by wrong types can only be found during execution. This problem can be mitigated by using type hints, a feature introduced in Python 3.5, with the PEP 484 [12] that introduced optional type annotations for Python code; in the development of `pyOMA2` we decided to widely adopts this feature, in order to improve code readability and maintainability, indeed type hinting is becoming a synonym of good practice and well-documented code. Although using type hints in Python is a non-enforced annotations, but it only indicated the "expected types", it is a valuable practice to enhance code readability and receive support from IDEs (Integrated Development Environments) for assisting with static type checking, and improving support tools such as autocompletion and refactoring. Some growing and popular libraries are in a way bringing this python feature to the next level, for example, `Pydantic` [13], that is widely used in `pyOMA2`, is a data validation and settings management library using and enforcing type hints at runtime.

All things considered, Python's technical qualities—including its interpreted character, extensive standard library, and strong ecosystem of open source third-party libraries—make it a perfect choice for creating advanced and effective OMA tools. Using these characteristic, guarantees that our OMA library stays strong, easy to use, and cutting edge technologically advanced.

3.2 Dependency and Package Management in Python

In the modern software development world, efficient management of dependencies is definitely crucial to guarantee the robustness and reliability of a project, in order to make it repeatable, and free of conflicts. As a flexible programming language, Python offers a wide range of tools and approaches to address problems with dependency. Among these, package managers and virtual environments are especially used to facilitate the workflow. Any Python development process depends on their ability to clearly handle package installations and separate project-specific dependencies.

Maintaining a strong and reliable codebase depends strongly on managing dependencies, so it does not only represent a good practice but also a necessity.

Integrity and dependency of software depend on addressing the several difficulties related to dependency management. The common problems that these practices address include version constraints, bloated dependencies, and missing dependencies each of which will be covered in this section. These difficulties can come from many different causes, including, but not limited to, erroneous declarations in configuration files, accumulation of pointless dependencies that unnecessarily mess the environment, and conflicts resulting from incompatible version constraints. Ignoring such problems might cause unexpected behavior and increases the maintenance difficulties. Thus, reducing these problems and promoting a reliable development environment, depends on methodical and efficient dependence management strategies, which are absolutely important [14].

Virtual Environments

Managing dependencies and reducing conflicts in Python development relies mostly on virtual environments and the `PIP` (Pip Installs Packages) dependencies management. Virtual environments are fundamental in the Python ecosystem for effectively handling dependent-based problems, these are basically isolated directories that allows you to manage dependencies independently for every project, and keep also different versions of Python within them, depending on the project and needs.

This isolation guarantees that a particular versions needed for one project do not interact or contradict those of another project or globally installed in the user machine. Using virtual environments allows developers to guarantee that the exact versions of dependable required is always reproducible and that the project is not influenced by the global environment.

Python's normal package management is the utility known as `PIP`, short for "`Pip Installs Packages`".

Python libraries are available as packages, which are assembles of modules and other files each of which offers specific functionalities. These packages, are indexed and made available for download from the `Python Package Index` (`PyPI`) that is the official and most common third-party software repository for Python. So `PyPI` is the tool by which it is possible to search, download, and install Python packages, inside our virtual environment. An example of how to create a virtual environment and install a package in a unix-like operative system, is shown below:

```
1  # Create a virtual environment
2  python -m venv myenv
3
4  # Activate the virtual environment
5  source myenv/bin/activate
6
```

```
# Install a package
pip install pyoma-2
```

Listing 3.1: Creating a virtual environment and installing a package

Rather common in the scientific communities, `Conda` is a valid substitute for `PIP`. `Conda` is a large, open-source package manager and somewhat different from the `PIP` package system, in fact, unlike `PIP`, which only allows to work with Python packages from `PyPI`, `Conda` is language-agnostic, consequently, it can also control dependencies for other languages such `R`, `Ruby`, and `Perl`.

Currently, `pyOMA2`, is available both ad `PyPI` and `Conda` repositories, specifically in the `conda-forge` channel, that is a channel maintained by the community; the choice of making the library available on `Conda` was made to make the library available to a wider audience, in particular to the scientific community that is commonly more familiar with `Conda`.

Here an example of how to create a virtual environment and install a package using `Conda`:

```
# Create a virtual environment
conda create --name myenv

# Activate the virtual environment
conda activate myenv

# Install a package (pyoma-2 is currently available in the conda-forge channel)
conda install conda-forge::pyoma-2
```

Listing 3.2: Creating a virtual environment and installing a package with Conda

`Conda` tends to handle binary dependencies more efficiently. When installing packages, `Conda` deals not only with Python packages but also with system-level dependencies (like C and Fortran libraries) that do not necessarily come from `PyPI`, this capability makes `Conda` especially useful for Python packages that depend on non-Python libraries, such as those used in scientific computing, we itself experienced less problems when we release the library on `Conda` compared to `PyPI`, the problems with the latter born particularly from the compilation of graphical libraries that are really dependent on the Operative System and the version of the libraries installed on the system.

Despite its advantages, `Conda` does have some limitations [15]:

1. *Package Availability*: `Conda` might not have every package available on `PyPI`, this can be a limitation when a required library is not available in `Conda` channels and must be installed separately via `PIP`.

2. *Larger Package Size*: `Conda` packages can be larger because they contain compiled binaries and all necessary dependencies and this can lead to larger installations

and higher storage requirements.

3. *Performance*: `Conda`'s resolver can be slower than `PIP`'s, especially when dealing with a large number of packages or solving complex dependency trees. This can often cause it to require way longer installation times.

PEP 518 and PEP 582: Enhancing Dependency Management

A relevant improvement was introduced in Python 3.8, with the introduction of PEP 518 [16] and PEP 582 [17] specifically for projects intended to be distributed via `PyPI` or to be generally installable via `PIP`.

PEP 518 presents a technique for specifying the minimal build system needs for Python programs. This PEP's main goal is to improve the consistency and dependency of Python projects by requiring that projects specifically state the tools and dependencies needed to construct them from source. The `pyproject.toml` file is a standardized configuration file that lets developers list build dependencies using a consistent and standardized format.

Having also uniform metadata for build tools and dependencies, the `pyproject.toml` file also allows developers to specify the build back-end to be used for the project and other projects metadata like the project name, version, authors, and so on.

Here is an example of a `pyproject.toml` file taken from the `pyOMA2` project repository, the representation and explanation of each part of the file:

```
[project]
name = "pyOMA_2"
version = "0.5.2"
description = "Python module for conducting Operational Modal Analysis"
authors = [
    {name = "Dag Pasca", email = "dpa@treteknisk.no"},
    {name = "Angelo Aloisio", email = "angelo.aloisio1@univaq.it"},
    {name = "Marco Martino Rosso", email = "marco.rosso@polito.it"},
    {name = "Diego Federico Margoni", email = "diegofederico.margoni@studenti.
    polito.it"},
]
dependencies = [
    "numpy<1.25; python_version < '3.9'",
    "numpy>=1.25; python_version >= '3.9'",
    "pandas>=2.0.3",
    "scipy>=1.9.3",
    "pydantic>=2.5.1",
    "tqdm>=4.66.1",
    "matplotlib>=3.7.4",
]
requires-python = ">=3.8,<3.13"
readme = "README.md"
license = {text = "MIT"}

[project.urls]
```

```
Homepage = "https://github.com/dagghe/pyOMA2"
Documentation = "https://pyoma.readthedocs.io/en/latest/"
Repository = "https://github.com/dagghe/pyOMA2"
Changelog = "https://github.com/dagghe/pyOMA2/blob/main/CHANGELOG.md"
Contributing = "https://github.com/dagghe/pyOMA2/blob/main/CONTRIBUTING.md"

[build-system]
requires = ["pdm-backend"]
build-backend = "pdm.backend"

[tool.pdm.dev-dependencies]
docs = [
    "sphinx>=7.1.2",
    "sphinx-rtd-theme>=2.0.0",
    "ghp-import>=2.1.0",
    "nbsphinx>=0.9.3",
    "pandoc>=2.3",
]
qa = [
    "pre-commit>=3.5.0",
    "ipdb>=0.13.13",
    "pytest>=7.4.4",
    "pytest-cov>=4.1.0",
    "notebook>=7.1.2",
    "tox>=4.14.2",
]
```

Listing 3.3: `pyproject.toml` Configuration File example

- `[project]`: This section contains metadata about the project.
  - `name`: The name of the project, `pyOMA2`.
  - `version`: The version of the project, here it is `0.5.2`.
  - `description`: A brief description of the project.
  - `authors`: A list of authors and their email addresses.
  - `dependencies`: Project dependencies with specific version constraints. For example:
    - `numpy<1.25; python_version < '3.9'`: Numpy version less than 1.25 for Python versions below 3.9.
    - `numpy>=1.25; python_version >= '3.9'`: Numpy version 1.25 or greater for Python versions 3.9 and above.
    - `pandas>=2.0.3`, `scipy>=1.9.3`, etc.: Other dependencies with their minimum required versions.
  - `requires-python`: Specifies the supported Python versions.
  - `readme`: Points to the README file.

  - `license`: The license under which the project is released, here it is the MIT license.

- `[project.urls]`: This section provides URLs relevant to the project.
  - `Homepage`: The project's homepage URL.
  - `Documentation`: URL for the project's documentation.
  - `Repository`: URL for the project's source code repository.
  - `Changelog`: URL for the project's changelog.
  - `Contributing`: URL for the contribution guidelines.

- `[build-system]`: This section defines the build system requirements.
  - `requires`: Specifies the build dependencies, here it is `pdm-backend`.
  - `build-backend`: Specifies the backend to use, here it is `pdm.backend`.

- `[tool.pdm.dev-dependencies]`: This section specifies the development dependencies.
  - `docs`: Dependencies required for documentation.
    - `sphinx>=7.1.2`: Minimum version requirement for Sphinx.
    - `sphinx-rtd-theme>=2.0.0`, etc.: Other documentation-related dependencies.
  - `qa`: Quality assurance and testing dependencies.
    - `pre-commit>=3.5.0`: Minimum version requirement for pre-commit hooks.
    - `pytest>=7.4.4`, `tox>=4.14.2`, etc.: Other QA-related dependencies.

Following PEP 518 helps developers to guarantee that the method of building environment for their project is clearly defined, therefore reducing variations between several systems and development approaches. For pipelines of continuous integration and deployment where fixed and reproducible build environments are absolutely vital, this standardization is especially helpful. For example, if a project requires specific versions of build tools such as a static analyzer or setuptools, the `pyproject.toml` file can be used to specify these dependencies, ensuring their universal recognition and installation before the start of the building process.

Adoption of PEP 518 by tools and package managers represents a major step toward delivering more robust and repeatable Python projects, thus enabling developers to manage build dependencies more effectively.

PEP 582: Python local packages directory

PEP 582, on the other hand, introduces a new approach for controlling project-specific dependencies free from virtual environment activation. This is achieved by identifying a particular folder in the project root directory, exactly called `__pypackages__`. All the dependencies, separated by the specific python version used for the project, would be placed in this directory, thus isolating packages linked to that project.

PEP 582's main objective is to lighten the dependency management process by removing the overhead associated with building and activating virtual environments. Especially for newbies to the Python ecosystem, this method is more straightforward since it directly combines dependency management into the directory structure of the project.

PEP 582 also seeks to simplify project setup since, depending on a repository and installing dependencies would now require fewer actions. Just looking at the `__pypackages__` directory can help developers and tools to identify the necessary dependencies, hence simplifying the setup process.

In the Python environment different tool were developed to ease the management of dependencies, in line with the presented PEPs, just to cite some of them we can mention `Poetry`, `PDM`, `Pipenv`, and the most recent `Hatch` and `uv`, a powerful and performant package manager built on top of `Rust` programming language.

When it came the time to decide which tool to use for the development of this library, we opted for `PDM` [18] (Python Development Master), a modern Python package and dependency manager that adopts PEP 582. Despite the fact that `Poetry` is the most popular and long-living tool, `PDM` is a more recent tool that gave the impression to have learnt from the mistakes of the past tools, and it is more in line with the most recent PEPs, in particular PEP 582. Furthermore, I freely admit that it was decided to utilize `PDM`, rather new, dependency management technology, seizing the opportunity to learn and experiment with it during the project's development. Our project's choice of `PDM` was also driven in part by its capacity to natively support platform-specific dependencies, since scientific and graphical libraries can vary significantly across operating systems due to their reliance on system-level and low-level libraries that can lead to OS-specific versions, `PDM` offers the ability to specify and lock requirements unique to Linux, macOS, or Windows, pinning the specific or a range of version for each of them.

Adopting PEP 582, `PDM` is, in conclusion, a contemporary Python package and dependency manager meant to tackle some of the typical problems with conventional dependency management tools. `PDM` simplifies developers' workflow by using the local packages directory specified in PEP 582, therefore eliminating the requirement to activate virtual environments.

Chapter 4

Software Design and Architecture

4.1 System Architecture

One of the main reasons why it was necessary to re-engineer the `PyOMA` library in the present version of `pyOMA2`, in addition to the natural extension of the set of algorithms and visualization tools, was to ensure that data processing, manipulation, result saving and result visualization were improved and more fluid.

Let's examine in detail how we organized the system and the overall workflow.

Central to `pyOMA2` are the Setup classes. Consider these as the directorship of the whole OMA process. They act a an orchestrator, their role is to integrate all the various components and ensure their seamless operation. The primary setup classes available are `SingleSetup` for the analysis of individual datasets and multi-setup ones, that are the `MultiSetup_PoSER` and `MultiSetup_PreGER` that allows to manage several experimental configurations.

4.1.1 Acquisition and Preprocessing of Data

The Setup classes mark the starting point of the data journey. They receive the raw vibration data and sample frequency, therefore enabling the provision of this information to the remaining components of the system, such as the algorithms and visualization tools. However, their role does not end there. Given the inherent imperfections of real-world data, Setup classes, are provided with a collection of convenient preprocessing functions, whether it involves removing an unwanted trend or for example downsampling to focus on lower frequencies or simple filtering, the classes offer methods and comprehensive decimation algorithms to ensure that the data reaching the algorithms is cleaned and pertinent.

4.1.2 Algorithmic Layer

After data acquisition, the real analytical work starts in the algorithmic layer.

Different OMA techniques have been applied; each one belongs to a different Python

class and, since all of these classes inherit from a common base class, they provide a consistent interface, while enabling the implementation of specific functionalities.

Every algorithm class is supported by a unique set of input parameters, properly arranged inside a `RunParams` class; this approach helps users maximize the performance of the algorithm without feeling overburdened by its complexity of use. Every algorithm's result is stored in a corresponding `Result` class, therefore enabling easy access and comprehension of the produced output. This architecture has intrinsic strength because was thought and designed to be easily extended, both in the specific functionalities of the single algorithms or in the extension of new OMA techniques.

4.1.3 Data Visualization and Interaction

Lastly, but surely not less important, we have our visualization tools. They are essential components of the analysis process, since they allow users to interpret and interact with the results or the operative data in a more perceptive way. A variety of plotting functions have been developed to accurately represent a wide range of data, including time histories and stability diagrams.

For instance, an outstanding quality of this library is its interactive mode shape visualization capability. This feature lets users observe structural behavior with animations showing vibrations at several natural frequencies. While interpreting numerical data on a computer screen can give insight, visualizing these analytical results using dynamic representations helps one to better grasp the behavior of the system.

Furthermore, we have included interactive plotting capabilities that enable users to directly choose pertinent data from the plots. The seamless integration of visualization and analysis enhances the overall intuitiveness and user-friendliness of the process.

4.1.4 Data flow

How therefore does it all cohere? Here is a typical workflow for every setup type:

- `SingleSetup`
    1. The user initialize Setup Class with some data.
    2. Preprocessing techniques may be applied on the data if necessary.
    3. One or more algorithm instances is added to the Setup.
    4. The Setup class runs the algorithms, passing them the prepared data.
    5. Results are stored, by the Setup Class, in each algorithm instance.
    6. The user should now provides the peaks to the algorithm class either by providing them manually or by selecting them utilizing the interactive visualization plots.
    7. The user can now, optionally, define the Geometry of the sensors by proving the coordinates of the sensors, along with their orientation.

8. Finally it is possible to use visualization methods to interpret and interact with the results.

- `MultiSetup_PoSER`

  1. The user inputs data into a set of `SingleSetup` instances.
  2. Each `SingleSetup` instance should have their set of algorithms already ran as seen before.
  3. The `MultiSetup_PoSER` class takes these `SingleSetup` instances along with a set of reference sensors, and runs the merging method on their results.
  4. As in the `SingleSetup` case, the user can now provide the peaks and the geometry of the sensors and use visualization methods to interpret and interact with the results.

- `MultiSetup_PreGER`

  1. The user inputs a set of data and a set of reference sensors into a `MultiSetup_PreGER` instance.
  2. A global algorithm is ran on the data, merging the results.
  3. A set of algorithms can be added to the `MultiSetup_PreGER` instance.
  4. Each algorithm is ran on the merged data.
  5. Also here it is possible now to provide the peaks and the geometry of the sensors

This architecture allows OMA methods to be applied with a flexible and modular approach. Users may simply compare outcomes, mix and combine several techniques, and incrementally improve their work, indeed this design was specifically thought to be intuitive for beginners while providing the power and flexibility that advanced users need. We have essentially developed a system that reflects the OMA process itself: gathering unprocessed data, applying advanced analysis techniques, and generating outcomes that help in understand the dynamic behavior of structures. And just like the structures we analyzed, we've built `pyOMA2` to be robust, flexible, and ready to respond to the requirements of the OMA community.

4.2 Modular Design and description

The `pyOMA2` library has been engineered with a strong emphasis on modular design, hence improving its extensibility, maintainability, and flexibility.

The package structure, which arranges related components into certain subpackages, reflects the modular design. Let's investigate the main features of the modular design of `pyOMA2` underlying the package structure.

4.2.1 pyoma2.algorithms

Containing all the implemented OMA techniques, this package is the core of pyOMA2.

This is the structure of the package:

- base.py: Here is where the BaseAlgorithm class is defined, this class serves as a template for every OMA algorithm.
- fdd.py, plscf.py, ssi.py Implement particular OMA techniques (Frequency Domain Decomposition, Polyreference Least Squares Complex Frequency-domain, and Stochastic Subspace Identification, respectively), each orchestrated by the SingleSetup and MultiSetup cases.
- data/: A subfolder containing:
  - result.py: Specifies result classes for each algorithm.
  - run_params.py: Specifies input parameters for each algorithm.

This framework guarantees consistency among several implementations and makes new algorithm addition simpler.

4.2.2 pyoma2.setup

The Setup classes in the setup package handle the general OMA process. It includes the following:

- base.py: Contains the base class for textttSingleSetup and MultiSetup_PreGER.
- single.py: Implements the SingleSetup class for individual dataset analysis.
- multi.py: Contains MultiSetup_PoSER and MultiSetup_PreGER classes for handling multiple experimental configurations.

These classes act as orchestrators, coordinating data input, preprocessing, algorithm execution, and result visualization.

4.2.3 pyoma2.support

This package includes utility functions and support classes for data plotting and visualization:

- sel_from_plot.py: Provides interactive selection of results from plots.
- geometry/: A subfolder containing:
  - data.py: Defines BaseGeometry, Geometry1, and Geometry2
  - where the latter two are subclasses of the former, and are used as data classes for different ways of defining the geometry of the sensors.

  - `mixin.py`: Provides methods for geometry definition and visualization, implementing the `GeometryMixin` class, this mixins offers these functionalities to the setup classes.
  - `plotter.py`: Defines the abstract base class `BasePlotter` for plotting geometry and mode shapes, using generic typing for flexibility across different geometry types.
  - `mpl_plotter.py`: Implements Matplotlib-based plotting functions through `Geo1MplPlotter`, `Geo2MplPlotter`, and `MplPlotter` classes, which likely inherit from `BasePlotter`.
  - `pyvista_plotter.py`: Provides PyVista-based 3D visualization capabilities through the `PvGeoPlotter` class, which likely inherits from `BasePlotter`.

- `utils/`: A subfolder containing:
  - `logging_handler.py`: Manages logging functionality.
  - `typing.py`: Defines custom data types used throughout the library.

This package gives necessary utility purposes and improves the visualizing capacity of the library.

4.2.4 `pyoma2.functions`

The functions package contains preprocessing functions specific to each algorithm as well as tools for result visualization:

- `gen.py`: Houses general-purpose functions used across different algorithms.
- `fdd.py`, `plscf.py`, `ssi.py`: Contain algorithm-specific preprocessing and visualization functions.
- `plot.py`: Implements common plotting functions used throughout the library.

This design keeps algorithm-specific operations near to the algorithms themselves while separating supporting functions and setup module from central algorithm logic.

Based on its package architecture, `pyOMA2`'s modular design presents various benefits.

- **Extensibility**: It is possible to easily add new algorithms or visualizing techniques without affecting current code.
- **Maintainability**: Every module can be developed, debugged or tested separately, therefore streamlining maintenance.
- **Flexibility**: Users can combine and match several elements to fit their particular requirements.

- **Readability**: The clear package structure makes the codebase more organized and easier to understand and read.

- **Collaboration**: Different team members might work on separate modules concurrently, encouraging collaborative development.

Finally, `pyOMA2`'s modular architecture and careful package layout not only improve its present performance but also open the path for next developments and adaptations. This design concept guarantees that `pyOMA2` may develop with the field of operational modal analysis, always fulfilling the evolving needs of its users.

4.3 Design Patterns

Another fundamental aspect that has been taken into consideration in re-engineering `pyOMA2` library, was the incorporation of design patterns and best practices to guarantee the creation of cleaner, more maintainable and adaptable code.

The library is still under active and continuous development; therefore, the smooth growth and maintenance of the code depends on the inclusion of design patterns. Still, there are other areas that might be improved.

Here is presented the analysis of the design patterns included into the `pyOMA2` library.

4.3.1 Template Method Pattern

Found in `pyoma2.algorithms.base.py`, all OMA techniques are modeled from the abstract class `BaseAlgorithm`. It outlines the skeleton of the function in its `run()` method, that is the method used by the setup classes to run the algorithms [19].

This pattern guarantees a uniform architecture and structure across all OMA methods and helps each particular implementation to be customized.

In python the Template Method Pattern is implemented by using the `abc` module, which allows the creation of abstract classes and methods.

```python
# pyoma2/algorithms/base.py

class BaseAlgorithm(abc.ABC):
    # ...

    @abc.abstractmethod
    def run(self) -> T_Result:
        """
        Abstract method to execute the algorithm.
        """
```

Listing 4.1: Template Method Pattern snippet from `pyOMA2`

4.3.2 Strategy Pattern Method

The Strategy Pattern is shown by the capacity to include several algorithms to the Setup classes (`SingleSetup`, `MultiSetup_PoSER`, `MultiSetup_PreGER`). This pattern allows for the selection and interchange of several methodologies (algorithms) without changing the Setup class itself. This gives freedom in selecting and aggregating several OMA methods based on needs [20].

As shown in the snippet below, the `add_algorithms()` method in the `BaseSetup` class takes a generic list of algorithms and configures them with the data and sampling frequency.

Although it doesn't deal with the actual creation of the algorithm instance, this method acts as a Factory Method, as it captures the logical framework of object initialization, passing necessary data to the main `run()` method [21].

```python
# pyoma2/setup/base.py

class BaseSetup:
    # ...

    def add_algorithms(self, *algorithms: BaseAlgorithm):
        """
        Adds algorithms to the setup and configures
        them with data and sampling frequency.
        """
        self.algorithms = {
            **getattr(
                self, "algorithms", {}
            ),
            **{
                alg.name: alg._set_data(
                    data=self.data, fs=self.fs
                ) for alg in algorithms
            },
        }
```

Listing 4.2: Strategy Pattern snippet from `pyOMA2`

4.3.3 Composition over Inheritance

The Setup classes integrate several algorithms via composition rather than inheriting from them. It simply means that for utilizing the method of a class, this class will be passed as an argument to the Setup class, rather than being a subclass. Since it's simpler to add or delete algorithms at runtime and this architecture avoids the complexity of multiple inheritance making the code cleaner and applying the `Single Responsibility Principle` that we'll discuss later [22].

```python
# pyoma2/setup/base.py
```

```
2
3  class BaseSetup:
4      # ...
5      algorithms: typing.Dict[str, BaseAlgorithm]
```

Listing 4.3: Composition over Inheritance snippet from `pyOMA2`

4.3.4 Generic Typing and Type Hinting

As known, Python is a dynamically typed language, but the use of type hinting and generic typing can improve code readability and identify type-related defects at an early stage [23, 24].

The library uses extensively the `typing` Python module [25] to specify the types of the parameters and return values of the functions and methods.

Addition constraint on typing are supported by the `Pydantic` [13] library, that is one of the dependencies of the `pyOMA2` library that allows the definition of data structures with constraints and validation at runtime.

Generic Typing on the other hand is used to specify the types of the `RunParams`, `Result` and `Data` components of the `BaseAlgorithm` class, this allows the definition of a generic type at the level of the base class, but at the same time allow its re-definition in the subclasses, so is pretty useful when a given type can change from one subclass to another.

```
1
2  # pyoma2/algorithms/base.py
3
4  import typing
5
6  T_RunParams = typing.TypeVar("T_RunParams", bound=BaseRunParams)
7  T_Result = typing.TypeVar("T_Result", bound=BaseResult)
8  T_Data = typing.TypeVar("T_Data", bound=typing.Iterable)
9
10 class BaseAlgorithm(typing.Generic[T_RunParams, T_Result, T_Data]):
11     # ...
12     result: typing.Optional[T_Result] = None
13     run_params: typing.Optional[T_RunParams] = None
14     RunParamCls: typing.Type[T_RunParams]
15     ResultCls: typing.Type[T_Result]
16
17     data: typing.Optional[T_Data]
18
19     def __init__(
20         self,
21         run_params: typing.Optional[T_RunParams] = None,
22         name: typing.Optional[str] = None,
23         *args,
24         **kwargs,
```

```
    ):
        """
        Initialize the algorithm
        """
        # ...

    def run(self) -> T_Result:
        """
        Abstract method to execute the algorithm.
        """

    def set_run_params(self, run_params: T_RunParams) -> "BaseAlgorithm":
        """
        Set the run parameters for the algorithm.
        """

    def _set_data(self, data: T_Data, fs: float) -> "BaseAlgorithm":
        """
        Set the input data and sampling frequency for the algorithm.
        """

# pyoma2/algorithms/fdd.py
class FDD(BaseAlgorithm[FDDRunParams, FDDResult, Iterable[float]]):
    # ...

    RunParamCls = FDDRunParams
    ResultCls = FDDResult

    def run(self) -> FDDResult:
        """
        Execute the Frequency Domain Decomposition algorithm.
        """

    # ...
```

Listing 4.4: Generic Typing snippet from `pyOMA2`

4.3.5 Facade Pattern

Facade pattern is a structural design pattern that provides a simplified interface to a complex subsystem, in our case, this role is played by the Setup classes, those classes, in fact, offer a streamlined interface to the complex subsystem composed by the algorithms, data processing, and visualization tools, acting as a facade. This design simplifies the library's usage for customers who do not require an understanding of the underlying complexities but also serves as a single entry point to run the core functionalities of the library [26].

4.3.6 SOLID Concepts

SOLID stand for (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) and are a set of principles that help to design maintainable and scalable software [27].

The SOLID principles are used in different parts of the code; here are some examples of its application in the `pyOMA2` library:

- **Single Responsibility Principle**: Every class and module have a clearly-defined responsibility, for example, while the Algorithm classes are responsible for implementing specific OMA techniques, the Setup classes manage the overall analysis process.
- **Open/Closed Principle**: The modular design lets the library's functionality to be easily extended (like for instance adding new algorithms) without modifying existing code.
- **Liskov Substitution Principle**: Subclasses of BaseAlgorithm can be used interchangeably, as they all implement the same interface.
- **Interface Segregation Principle:** The use of specific `RunParams` and `Result classes` for each algorithm ensures that clients only need to know about the methods they actually use.
- **Dependency Inversion Principle**: High-level modules (like Setup classes) depend on abstractions (like BaseAlgorithm, BaseResult, etc.), not concrete implementations.

These design principles and concepts help to define `pyOMA2`'s whole architecture, so increasing its modularity, expandable capability, and maintenance simplicity. Without major modifications to the current software, they enable simple inclusion of new algorithms, visualizing tools, and data processing approaches. This design strategy guarantees that `pyOMA2`' may develop alongside the area of operational modal analysis, therefore allowing new approaches and user needs as they arise.

In the end, we aim to position `pyOMA2` as a strong, adaptable, and forward-looking tool in the context of operational modal analysis software, thanks to its strategic architectural choices and the integration of best practices in its design.

The library's ability to evolve and adapt ensures that it will remain a valuable resource for scholars and practitioners seeking to analyze and interpret structural dynamics with greater accuracy and clarity.

Chapter 5

Software UML Diagrams

The Unified Modeling Language (UML) is a standardized visual language used in software engineering to model and document the design of software systems. It offers a set of diagrams and symbols to depict the structure and behavior of systems, so facilitating clear communication between developers and stakeholders during the development process and acting as a source of documentation for next maintenance, also presenting a high-level overview of the interactions between the system and its components.

UML diagrams support the entire software development process, from requirements analysis (Use Case Diagram and Class Diagram in the Glossary), as well as the various interactions between stakeholders and the application (Context Diagram and Sequence Diagram), to the definition of the main components that will make up the code (Class Diagram), up to the representation of the flows of the different processes (Activity Diagram, State Diagram, and Sequence Diagram), and finally to the representation of the architecture on which the application will be deployed (Deployment Diagram).

In the following sections, we present the Class and Sequence Diagrams, briefly describing what they are and how they are used, and applying them on the main components of the `pyOMA2` library.

5.1 Class Diagrams

Class diagrams are the most common diagrams used in UML to represent the structure of a system, since with their representation, they let to visualize the classes and interfaces of the system, together with their attributes, functions, and relationships [28].

Classes in UML Diagrams: At the core of class diagrams we can find the classes themselves, depicted as rectangles and divided into three compartments: the part at the top shows the class name, the middle one lists its attributes with the relative type, and the bottom outlines its methods.

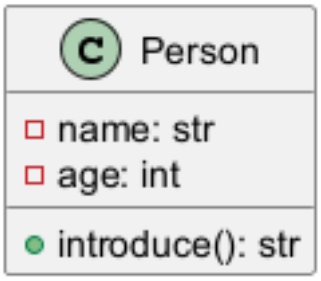


Figure 5.1: Class Diagram class

Interaction between classes is represented by different kinds of lines and arrows, which indicate the relationships between the classes, let's see some of them:

- **Association:** A relationship between two classes that indicates that one class is aware of the other but none of them is part of the other. It is represented by a solid line, optionally with an arrowhead indicating the direction of the relationship.

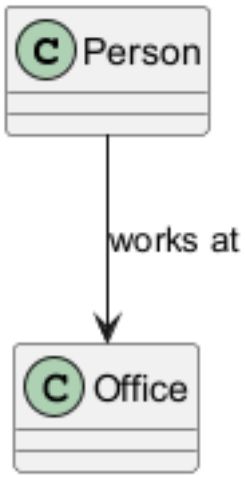


Figure 5.2: Class Diagram association

- **Aggregation:** Shows that a class is a part of another class and this is shown by a line with a hollow diamond at the class that holds the other class.

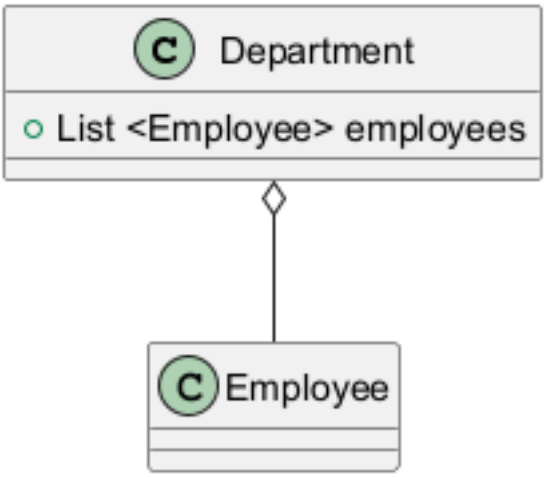


Figure 5.3: Class Diagram aggregation

- **Composition:** This is a stronger kind of aggregation that reflects a whole-part relationship where the component cannot exist without its whole composition. It is shown as a line with a solid diamond on the side of the class containing the other class.

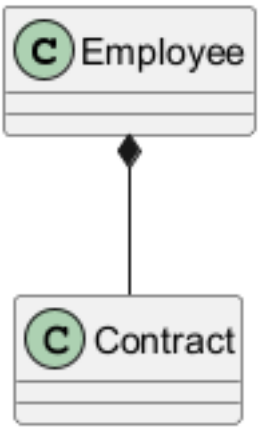


Figure 5.4: Class Diagram composition

- **Inheritance:** Is the relationship between two classes in which one class is a subclass of the other. It is represented by a solid line with a hollow triangle arrowhead.

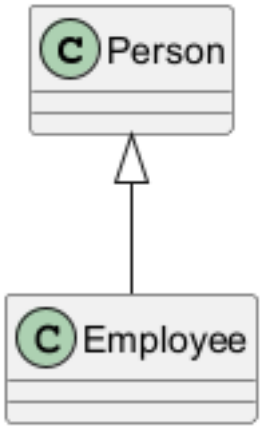


Figure 5.5: Class Diagram inheritance

- **Realization:** When a class realizes an interface, means that the class implements the interface, so has the concrete implementation of the methods exposed by that interface. Is the relationship between two classes in which one class is a subclass of the other, that means that the subclass implements the interface exposed by the superclass. It is represented by a dashed line with a hollow triangle arrowhead.

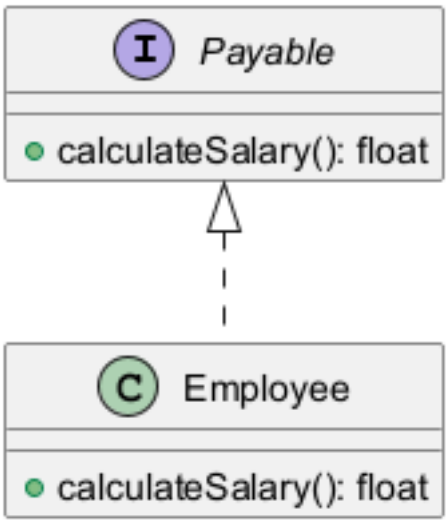


Figure 5.6: Class Diagram realization

Class diagrams' capabilities are not limited to the representation of classes and their relationships. They can also represent the visibility of attributes and methods, as well as the multiplicity of relationships between classes, we won't deepen these concepts here to avoid burdening the representation.

Here is a class diagram of the `pyOMA2` library, as anticipated, while simultaneously providing a general overview of the composition of the main classes in the library, the signatures and return types of the class methods have been omitted.

The representation and placement of the elements in the following class diagram (5.7) have been designed to graphically separate the different components of the package. In fact, it is possible to clearly notice, in addition to the hierarchies and relationships between the classes, the various layers composed of Setups, Algorithms, and Classes reserved for data storage.

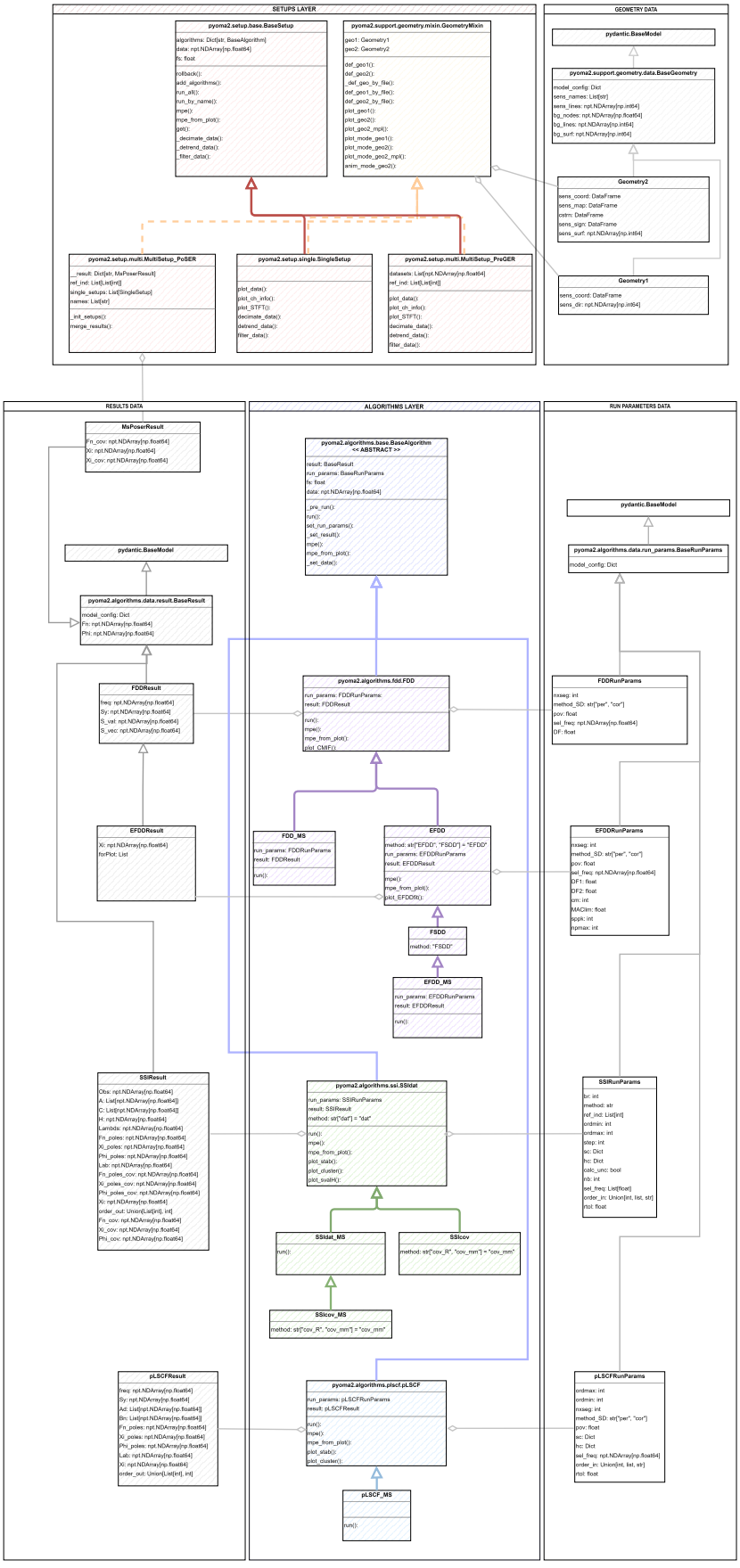


Figure 5.7: Class Diagram

5.2 Sequence Diagrams

Sequence diagrams are a type of interaction diagram that focus on the time ordering and messages exchanges between the different components, so they are pretty useful to represent the dynamic behavior of the system in the context of a specific scenario, considering also time and order of activation [29].

Elements in Sequence Diagrams: The main elements of a sequence diagram are the objects, which are represented by rectangles at the top of the diagram, and the lifelines, which are vertical dashed lines that represent the object's existence during the interaction. A similar component is the actor, which is represented by a stick figure and represents the user or external system, the main difference between the two is that the actor object is an active and internal part of the system, while the actor is an external entity.

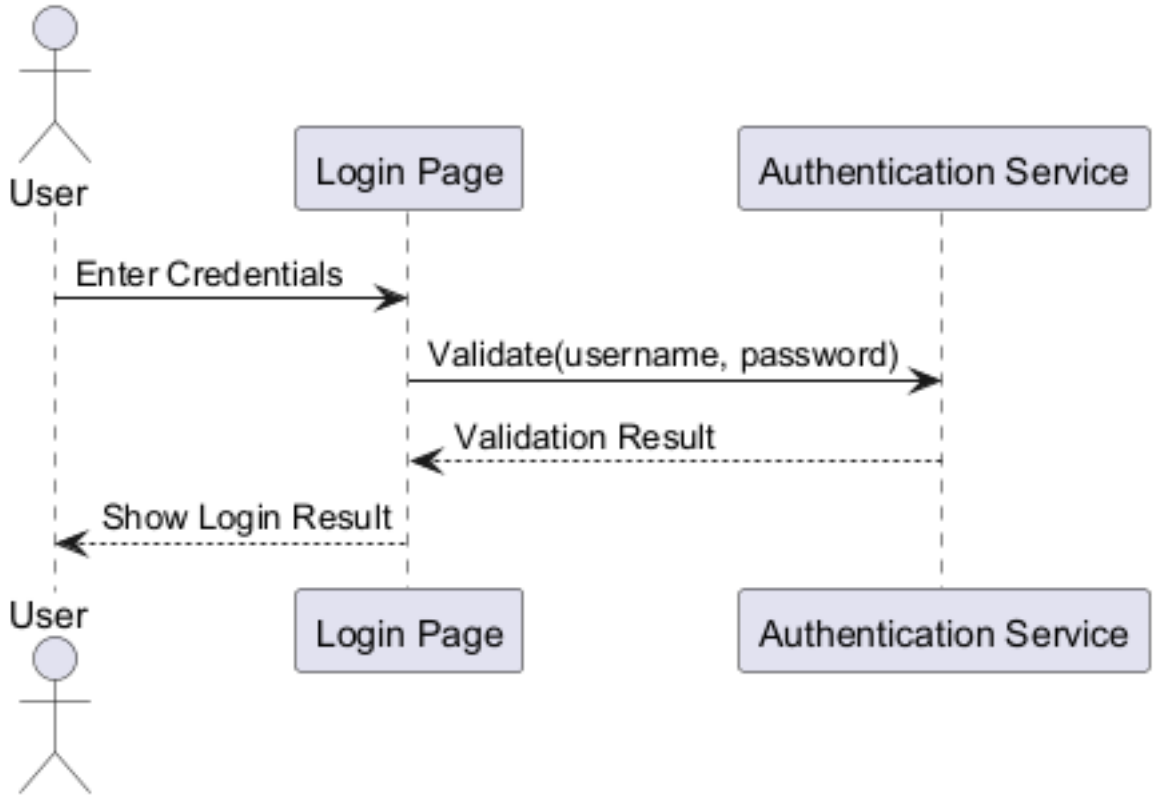


Figure 5.8: Sequence Diagram actor object

In the figure 5.8, the sequence diagram illustrates also the interaction between the actor and the object; in this kind of UML diagram, the interaction is represented by arrows that connect the objects and actors, and the messages are represented by the arrows themselves, which are labeled with the name of the message and the parameters passed.

These are the meaning of the different types of messages that can be represented in a sequence diagram:

- **Synchronous Message:** A message that blocks the sender until the receiver processes it, is represented by a solid line.
- **Asynchronous Message:** A message that does not block the sender, it is represented by a dashed line.
- **Return Message:** A message that represents the return of a method call, it is represented by a dashed line with an open arrowhead.
- **Self Message:** A message that represents a method call to the same object, it is represented by a looped arrow.
- **Create Message:** A message that represents the creation of a new object, it is represented by a dashed line with an open arrowhead and corresponds to the constructor of the object.
- **Destroy Message:** A message that represents the destruction of an object, it is represented by a dashed line with an X at the end.

To represent conditional behavior in the sequence diagram, as illustrated in the next example figure 5.9, we will also see other tools in the `pyOMA2` sequence diagrams such the alternatives and optional fragments.

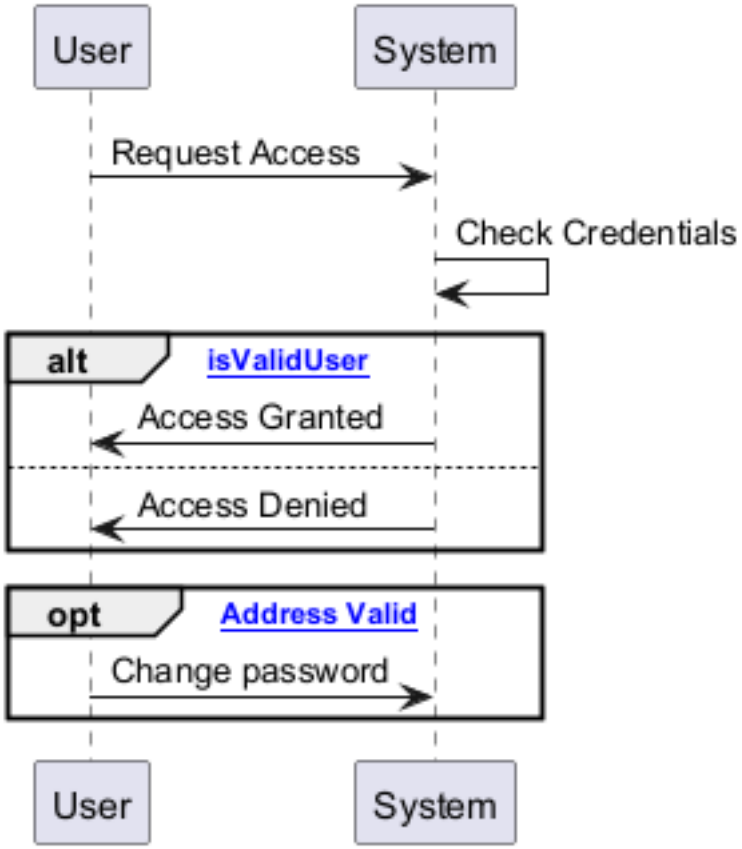


Figure 5.9: Sequence Diagram fragments

5.2.1 Single Setup Sequence

This 5.18 sequence diagram shows the typical workflow for performing OMA using the `Single Setup` approach.

The main components involved in this process are the `SingleSetup` class and algorithms classes.

The most relevant steps of the workflow are the following:

1. **Data acquisition:** 5.10 The main script retrieves sample data using utility functions.

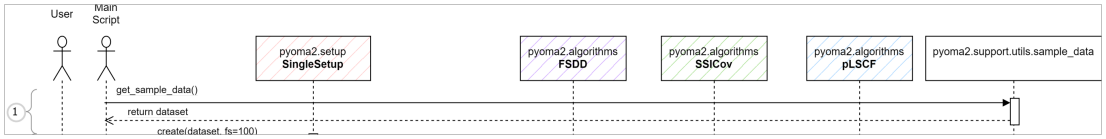


Figure 5.10: Sequence Diagram Data Acquisition

2. **SingleSetup Instantiation:** 5.11 A SingleSetup object is created with the dataset and sampling frequency and initial data preprocessing (filtering, decimation) is performed.

Figure 5.11: Sequence Diagram SingleSetup Instantiation

3. **Geometry Definition:** 5.12 The geometry of the structure is defined using `def_geo1()` and `def_geo2()`. Optional visualization of the geometry is available

Figure 5.12: Sequence Diagram Geometry Definition

4. **Channel Definition and Data Visualization:** 5.13 Time history plots of all or specific channels can be generated.

Figure 5.13: Sequence Diagram Channel Definition and Data Visualization

5. **Algorithm Definition:** 5.14 Instances of analysis algorithms (FSDD, SSIcov, pLSCF) are created with specific parameters.

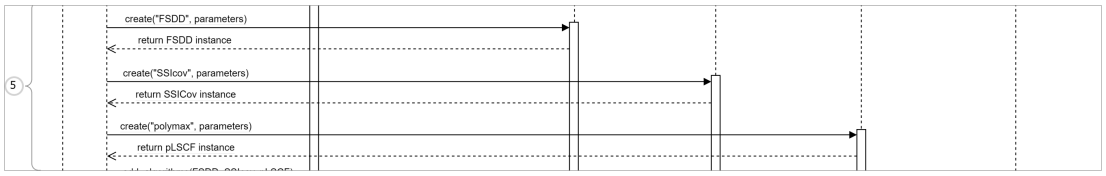


Figure 5.14: Sequence Diagram Algorithm Definition

6. **Adding Algorithms and Execution:** 5.15 Algorithms are added to the SingleSetup instance that takes care of executing them sequentially.

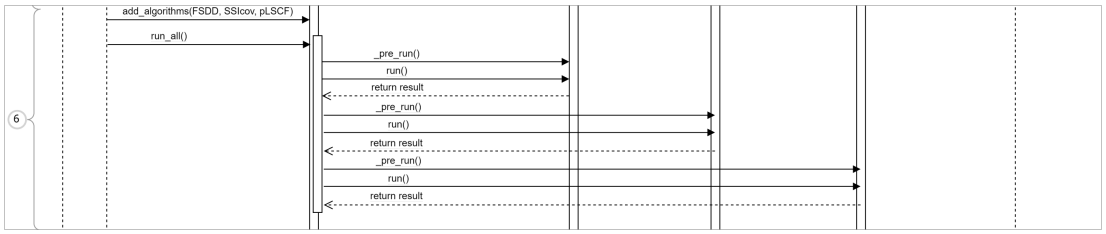


Figure 5.15: Sequence Diagram Adding Algorithms and Execution

7. **Modal Parameter Extraction:** 5.16 Modal parameters can be extracted either by providing specific data (`mpe()`) or through interactive plot selection (`mpe_from_plot()`).

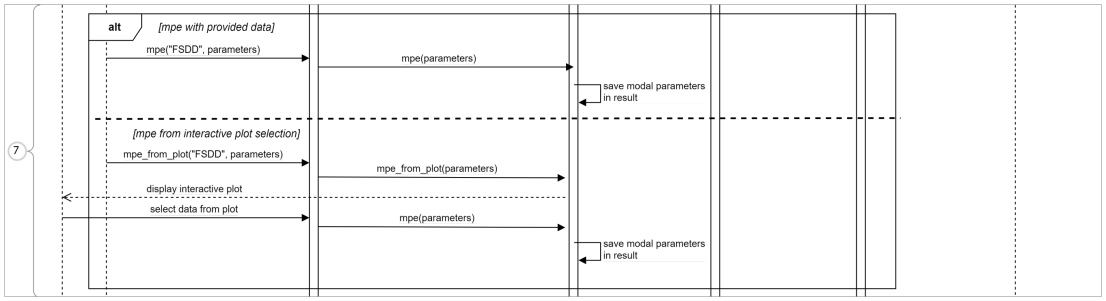


Figure 5.16: Sequence Diagram Modal Parameter Extraction

8. **Mode Shape Visualization:** 5.17 Various methods for plotting and animating mode shapes are available, using the results from any of the algorithms. In the diagram is only shown for the `FSDD` algorithm.

Figure 5.17: Sequence Diagram Mode Shape Visualization

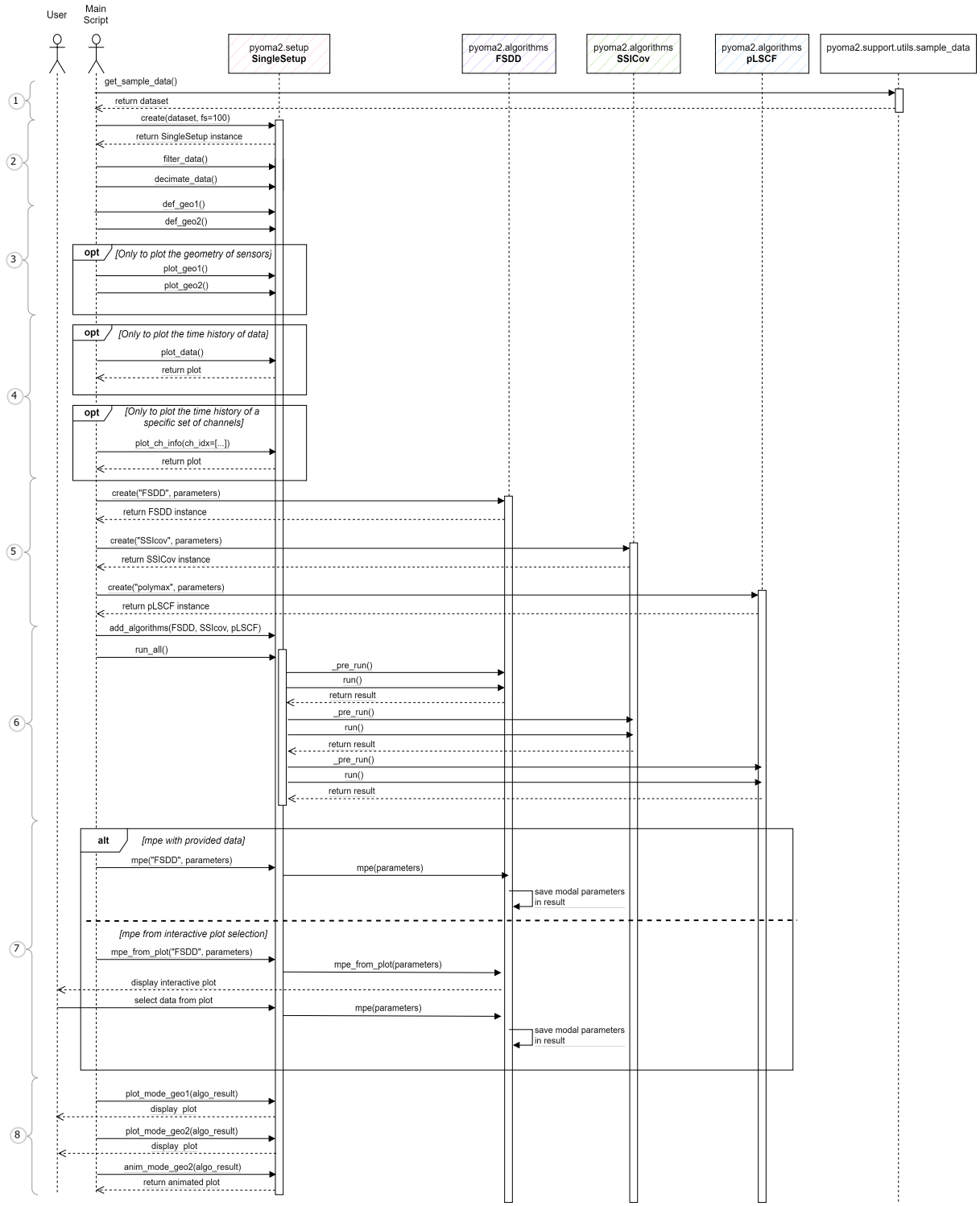


Figure 5.18: Single Setup Sequence Diagram

5.2.2 Multi Setup PoSER

Now let's deepen the workflow of the `Multi Setup PoSER` approach, which is shown in the sequence diagram 5.25. As we saw in chapter 4, the PoSER approach is like an iteration on the Single Setup approach, but with the possibility of using multiple setups, hence here the main classes involved are `MultiSetup_PoSER`, `SingleSetup` and the algorithms classes.

Let's see the main steps of the workflow:

1. **Data Acquisition:** 5.19 For simplicity in the diagram, the data acquisition step is omitted as it is similar to the Single Setup approach.

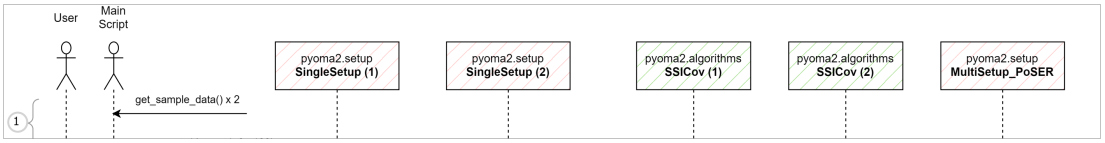


Figure 5.19: Sequence Diagram Data Acquisition

2. **SingleSetup Instantiation:** 5.20 Multiple SingleSetup objects are created, one for each dataset.

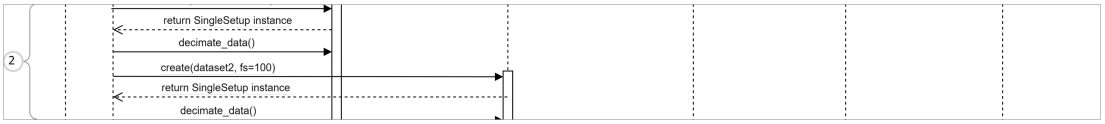


Figure 5.20: Sequence Diagram SingleSetup Instantiation

3. **Algorithm Creation and Execution:** 5.21 For each setup the same pattern as the Single Setup approach is followed. so different algorithms are created, attached to the Single Setups and ran.

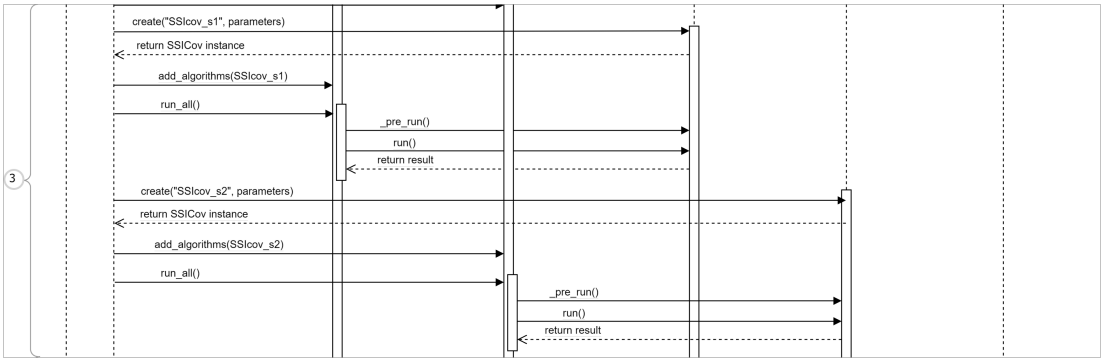


Figure 5.21: Sequence Diagram Algorithm Creation and Execution

4. **Modal Parameter Evaluation:** 5.22 For each setup, the modal parameters are extracted.

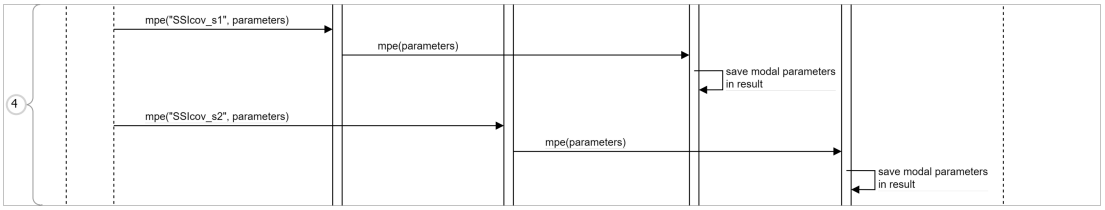


Figure 5.22: Sequence Diagram Modal Parameter Evaluation

5. **MultiSetup_PoSER Creation and Result Merging:** 5.23 A MultiSetup_PoSER instance is finally created, incorporating all SingleSetup instances. The `merge_results()` method is called to combine and process results from all setups.

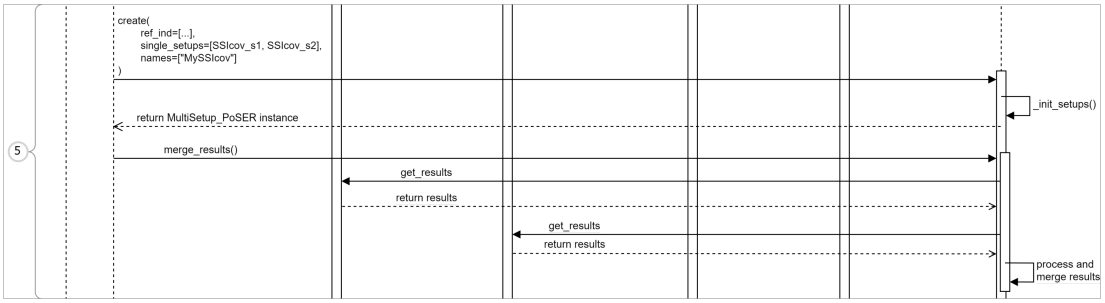


Figure 5.23: Sequence Diagram MultiSetup_PoSER Creation and Result Merging

6. **Geometry Definition and Visualization:** 5.24 The global geometry is defined for the merged results, and various methods for plotting and animating mode shapes are available using the merged results.

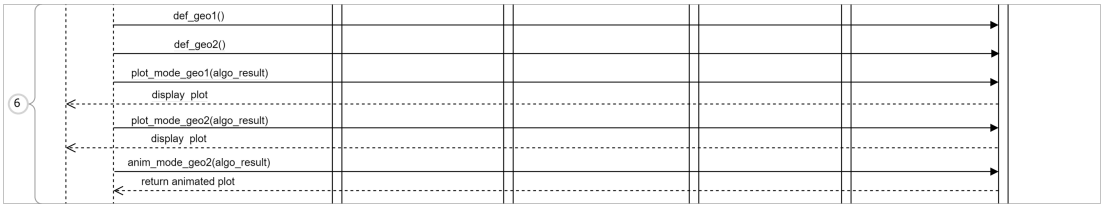


Figure 5.24: Sequence Diagram Geometry Definition and Visualization

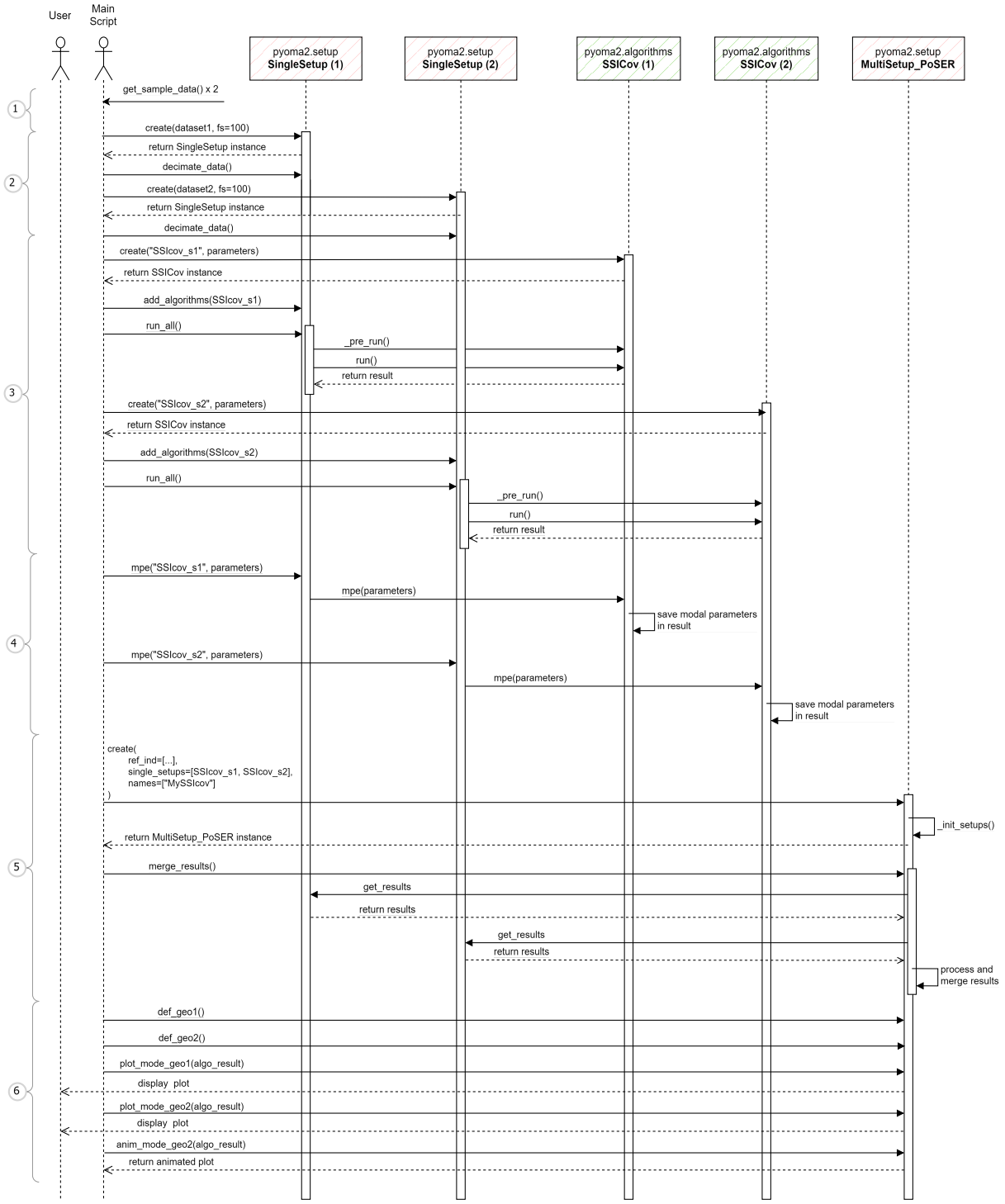


Figure 5.25: Multi Setup PoSER Diagram

5.2.3 Multi Setup PreGER

Finally, let's see the workflow of the `Multi Setup PreGER` approach, which is shown in the sequence diagram 5.31. They key difference between the PoSER and PreGER approaches is that the PreGER combines all datasets before processing, allowing for a single algorithm run.

The main classes involved in this process are `MultiSetup_PreGER` and the involved algorithms classes.

The main steps of the workflow are the following:

1. **Data Acquisition:** 5.26 The main script retrieves sample data for multiple setups (three in this example), each for a different dataset.

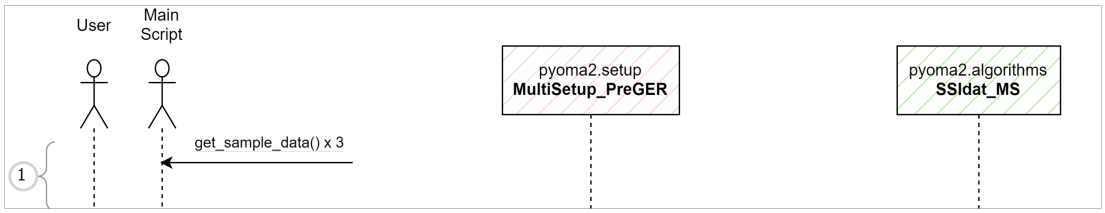


Figure 5.26: Sequence Diagram Data Acquisition

2. **MultiSetup_PreGER Instantiation:** 5.27 A `MultiSetup_PreGER` object is created, and data initialization is performed.

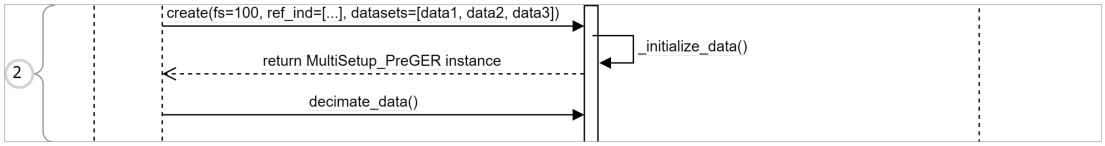


Figure 5.27: Sequence Diagram MultiSetup_PreGER Instantiation

3. **Algorithm Creation and Execution:** 5.28 An `SSIdat_MS` algorithm instance is created, added to the `MultiSetup_PreGER` instance, and executed on the combined dataset.

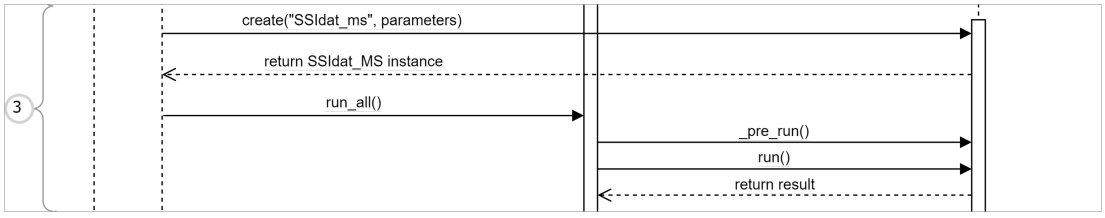


Figure 5.28: Sequence Diagram Algorithm Creation and Execution

4. **Modal Parameter Evaluation:** 5.29 Modal parameters are extracted using the `mpe()` method.

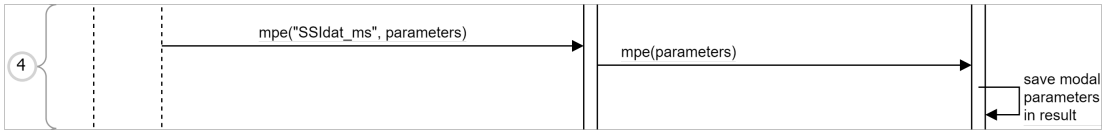


Figure 5.29: Sequence Diagram Modal Parameter Evaluation

5. **Geometry Definition and Visualization:** 5.30 The global geometry is defined for the results, and various methods for plotting and animating mode shapes are available.

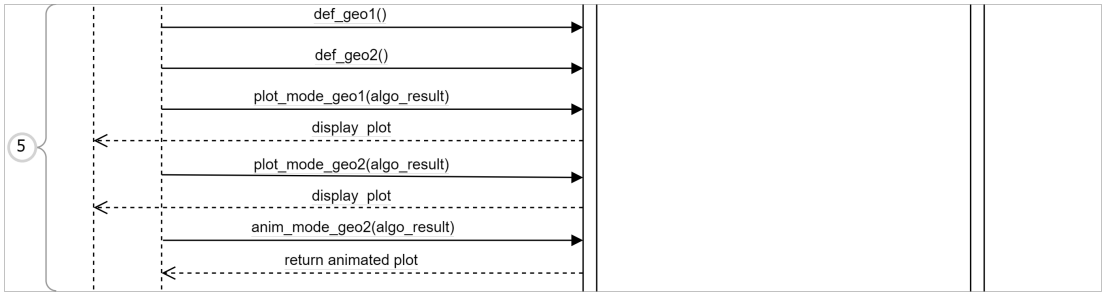


Figure 5.30: Sequence Diagram Geometry Definition and Visualization

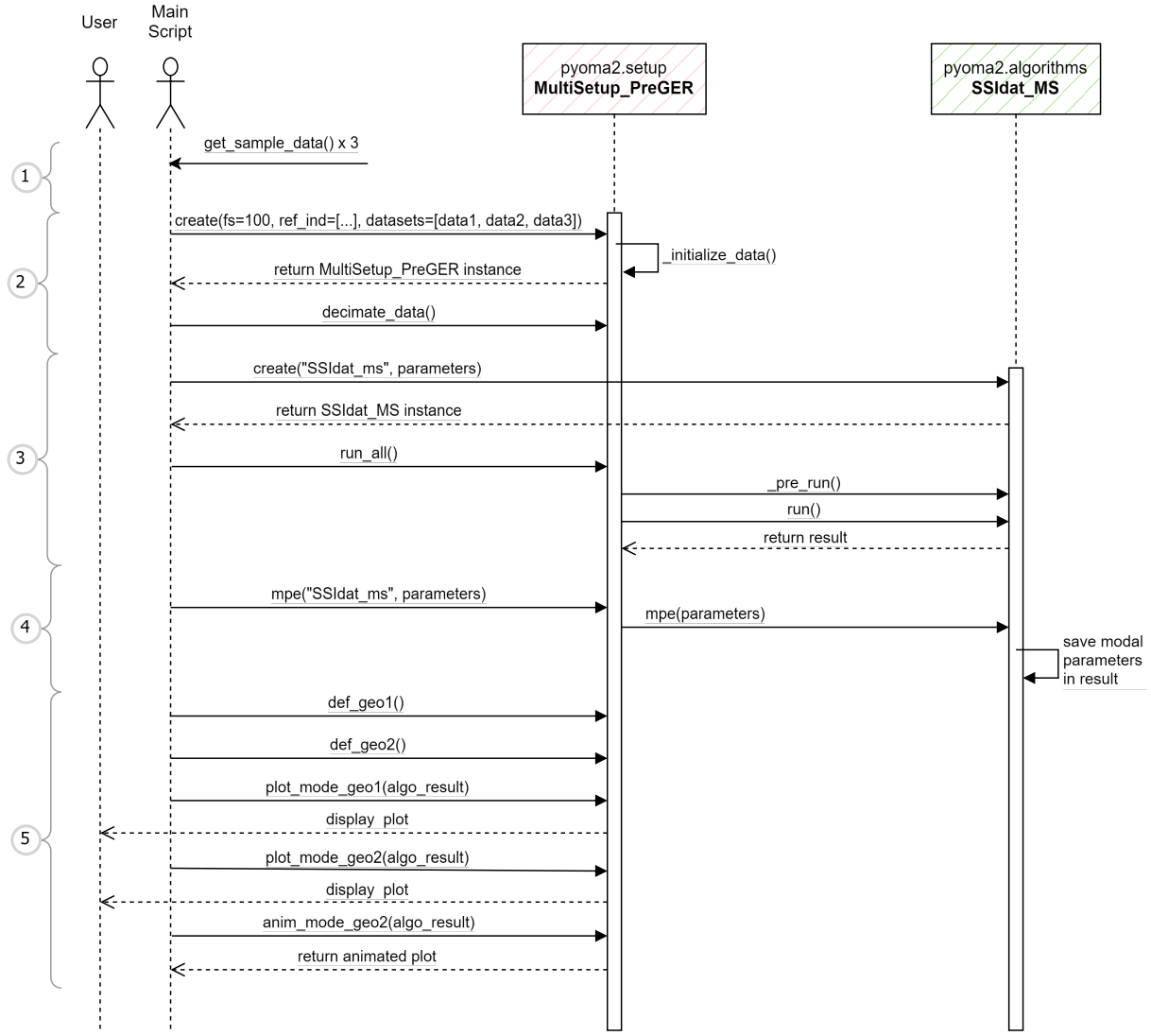


Figure 5.31: Multi Setup PreGER Diagram

Chapter 6

Development Process and Testing

6.1 Development Workflow

The development workflow is a crucial aspect of the project. It is important to have a clear and well-defined process in place to ensure that the project progresses efficiently. In this section, we will discuss the development workflow that we followed for this project.

6.1.1 Requirements Analysis

Unfortunately, the requirements analysis was not properly set up during the initial phase of the project, which made it difficult to establish an efficient method for tracking the requirements and the level of development of the library during the development process. The features and requirements developed during the year of development were always communicated verbally or through informal documentation, which made it difficult to keep track of the progress of the project. Despite this, several phases of prototyping were carried out, mainly aimed at redesigning the structure and interactions of the code. However, the standard for documenting requirements was not followed, nor was there a division into tasks that would have certainly facilitated traceability and organization of the work.

6.1.2 Version Control

We followed industry-standard version control and package distribution guidelines and used a modern technological stack throughout the `pyOMA2` development. Managing the lifetime of the project depended much on `Git` and `GitHub`; below is a brief description to introduce these two tools [30].

Git: Designed as a distributed version control system, `Git` lets developers track changes in their code across time. It seems like a sophisticated "save" function that not only saves several iterations of your files but also lets you alternate between different versions, create separate lines of work (called branches), and combine changes from

many sources.

In software development, `Git` is really useful since it lets basic errors to be corrected, tracks changes, and helps numerous engineers collaborate. While `GitHub` is a web-based tool hosting `Git` repositories and providing additional collaborative capabilities, `Git` is the fundamental version control tool.

GitHub: Because of its centralized code repository, pull requests and issue tracking features, and project management tools, we chose `GitHub` to host `pyOMA2`, moreover, in the open-source community, `GitHub` is widely used so we supposed it would be easier to help find and involve possible project contributors.

6.2 Continuous Integration and Deployment

Continuous Integration and Continuous Deployment (CI/CD) are a set of practices and tools that help automate the process of building, testing, and deploying software. For `pyOMA2`, we implemented CI/CD pipelines using `GitHub` Actions to automate various activities including for instance ensuring code quality, run tests across several environments and automate the release process to `PyPI`. `Github` actions are workflows defined in `YAML` files needed mainly to automate different tasks, these actions can be triggered by various hooks such as a push on the repository, opening a pull request, or other manuel scheduled events, they run on `GitHub`-hosted machines and can be used to automate different tasks.

Here is an overview of the CI/CD pipeline for pyOMA2:

CI:

- **Pre-commit hooks:** let us to automatically verify code formatting, style, and other problems before commits are made. This guarantees all through the project uniform and continuous coding quality and styling.

- **Automated tests:** in the context of tests, `GitHub` Actions, allow us to run the test suite on cloud machines, hosting different operating systems and different Python versions, specifically, we run the test across all the supported Python versions (from 3.8 to 3.12 included) and all the main operating systems (Ubuntu, Windows, and macOS), this is configured to be triggered on every push to the main branch and for all pull requests.

CD:

- **Automated releases:** Our workflow automatically creates the package and publishes it to `PyPI` when a new release is generated on `GitHub`, therefore instantly providing the most recent version to the users.

6.3 Documentation

A lot of effort was put into make the documentation as clear and ready to use as possible. We use automated tools alongside personal touches to make sure our documentation is comprehensive and easy to navigate. We primarily used `Sphinx` [31] to generate our documentation, it is a powerful tool that makes it easy to create intelligent and automated documentation extracting information from the codebase, docstrings and type annotations. This is a great way not to waste time writing documentation separately from the code, and at the same time encourage developers to write more detailed docstrings and type annotations.

Although we depend a lot on this automated method, we've also dedicated time to personally tailoring specific chapters of the documentation. This careful selection enabled us to offer broader insights and clarifications that may not seamlessly integrate into code documentation, like the integration with examples and user guides that show how to effectively use `pyOMA2`.

Our documentation can be found on `Read the Docs` [32], a well-known platform that helps in hosting and automatically building documentation for open source projects. Read the Docs provides a variety of advantages, including a reliable and easy-to-reach URL for our documentation, versioning, and also enabling users to access documentation for previous versions of `pyOMA2`. It provides a polished and professional appearance, complete with integrated search features.

At the moment, we handle our documentation build process on Read the Docs manually, this means that, after making important updates to the documentation or launching a new version of `pyOMA2`, a maintainer needs to manually initiate a new build on Read the Docs. This process allows us to have greater control over the timing of updates, but it does need ongoing attention to keep the online documentation up to date. Looking ahead, we might think about automating this process so that the online documentation stays aligned with the latest release or with the current state of the main branch. We might need to set up webhooks or find a way to integrate the documentation build process into our CI/CD pipeline.

6.4 Testing Methodologies

The development of `pyOMA2`, for what concerns the testing phase, followed a bottom-up approach. This means that we started by writing tests for the smallest components of the library, and progressively moved up to more complex tests that involve multiple components, so we started from those that are called unit tests and then moved to integration tests.

6.4.1 Unit Testing

Unit tests are the most basic form of testing, where individual units or components of a software are tested in isolation, to achieve this, since is rare to find functions or methods that don't depend on other parts of the code, it is necessary to use mocks and stubs to isolate the unit under test, these components are "hand made" objects that simulate the behavior of the real objects, from which the unit under test depends.

6.4.2 Integration Testing

On the other hand integration tests are used to examine the interaction between different components of the software, these are particularly useful for verifying the expected behavior of an entire workflow, in our case, this approach was for example used to test the correct behavior of the setups classes alongside the algorithms.

A significant challenge was handling the graphical functionalities during tests, in fact, these required extensive mocking to ensure they could be tested without relying on a graphical environment. This mocking strategy was crucial for both unit and integration tests, enabling us to run our test suite in various environments, including continuous integration pipelines.

Another aspect that was needed to be considered in order to develop reliable and replicable tests, given the specialized nature of OMA, was to have realistic working examples of the expected behavior of the library, to be able to build tests upon them; for this reason, it was essential to have a close collaboration with our Civil Engineering contributors that provided invaluable example datasets and expected outputs.

6.4.3 Testing Tools

The testing tool that was utilized to write and run the tests is `pytest` [33], a popular testing library for Python designed to makes it easier to write simple and scalable tests. One more powerful tool that we used to allow contributors to run local tests on different python versions is `tox` [34], a generic virtual environment management and test command line tool that allows to test the package, running the tests on different Python versions therefore enabling early discovery of compatibility problems.

Our test coverage is assessed with `pytest-cov` [35]. As of right now, we have attained a 75% statement coverage. Although this is a decent basis, we understand the need of always enhancing our test set. The project still has a lot of work ahead in raising the coverage percentage as well as the test quality.

Our testing strategy offers various advantages.

It aids in early in the development process bug and regression catchment. Crucially for a scientific computing tool, it gives trust on the accuracy of the library's implementations. It offers a safety net against inadvertent changes, hence enabling refactoring

and feature additions. The example-based tests provide extra documentation showing how the library ought to be used in useful situations.

Our goals going ahead are:

- Targeting currently under-tested portions of the codebase, expand our test coverage.
- Make sure our tests cover possible failure mechanisms and edge situations, therefore enhancing their quality.
- Increase our range of integration tests depending on practical use scenarios to close the theory-application gap.

We want to make sure `pyOMA2` stays a dependable instrument for the OMA community by keeping a strong testing program.

Chapter 7

Open Source Licensing and Benefits

One may question the relevance of this chapter. I considered it important to incorporate this chapter, as it is crucial to comprehend how the decision to make the library open source—within a predominantly proprietary market—has impacted the selection of technologies and development methodologies, following a comprehensive examination of OMA techniques' characteristics and objectives, as well as the implementation choices and development flow of the library. The objective is to make this technology accessible to anyone, with the expectation that the community will enhance and foster its development.

To better understand the context, we'll take a brief look at the main open-source licenses, as well as the benefits of open-source software (OSS).

7.1 Introduction to Open Source Licensing

The open source movement has profoundly changed the world of software development. Just consider the Linux operating system, which today is found on most servers, the Apache web server, the MySQL database management system and many others, all of which are open-source. From this we can derive that OSS drives most of the technological infrastructure used worldwide today.

At the core of this movement we found its licensing system, which guarantees precise rights and conditions for the use, modification, and distribution of software while balancing legal clarity with freedom so enabling developers to build on one another's work.

7.1.1 Open Source Definition

Let's dive in these concepts as excellently detailed in [36].

1. **Free distribution:** The idea of free redistribution guarantees that users may

distribute software without limitations, therefore encouraging larger acceptance and use. Open-source licenses guarantee that the software reaches a wide audience by removing obstacles on distribution, therefore enabling both individuals and companies to adopt and build upon it. On the other hand, proprietary software sometimes imposes strict requirements that can limit redistribution in order to maintain control over its use.

The `pyOMA2` itself aims to grow its user base in the engineering and scientific computing domains, and this strongly depend on free redistribution.

2. **Source code availability:** Still another fundamental element of open source is access to source code. Software that is opaque prevents users from accessing and modifying the underlying code, thus obscuring its operations and functionalities. In scientific disciplines, where repeatability is a major issue, this transparency is absolutely essential.

   Scientific software traditionally relies on proprietary packages, limiting users from fully examining or enhancing the algorithms driving their research. While `pyOMA2` enables users to customize the software to particular use cases by guaranteeing source code availability, therefore assuring it stays current and efficient across several fields of OMA.

3. **Derived works:** One great benefit of open-source licenses is the capacity to produce derivative works. Not only does it let users customize programs to fit their needs, but it also encourages creativity by letting users test and enhance the original code.

   This means that users can modify the software to suit their needs, and use it as part of their own projects, without the need to reinvent the wheel, but always meeting the requirements of the original license that we'll deepen in the next section.

4. **Integrity of the author's source code:** Open-source licenses could have clauses preserving the author's work integrity. They might, for instance, let just unedited versions of the program be shared but provide "patch files" the user might use to change the program at build. This guarantees that the original work stays recognizable even with improvements and customizing allowed.

5. **No discrimination against persons or groups:** The license must not exhibit discrimination against any individual or group of individuals.

6. **No discrimination against fields of endeavor:** No discrimination among fields of endeavor: so the license shall not prohibit any individual from utilizing the program in a particular business or field of activity.

7. **Distribution of license:** The rights associated with the software shall extend to all recipients upon redistribution, without requiring the execution of an extra license by those parties. This facilitates the distribution of the software and its derivatives, taking off the user from the burden of negotiating licenses with each new user.

8. **License must not be specific to a product:** this guarantees that the rights acquired by the license are not connected to a specific program or distribution of a software. A license stating, "This software is free to use only when installed on XYZ Linux distribution," for instance would contradict this idea.

9. **License must not restrict other software:** This idea keeps the license from restricting other programs released alongside the licensed ones. The license should only cover the software it is tied to, not unrelated software that happens to be distributed with it, so for instance it cannot impose the software that uses the licensed one to be open source as well.

10. **License must be technology-neutral:** Technology-neutral licenses guarantee that they neither favor nor discriminate against any one technology, platform, or interface. The rights given by the license should be applicable independent of the technological stack the program is applied in.

7.1.2 Open Source Licenses

As can already be guessed from the previous section, choosing an open-source license is a crucial step in the development of an open-source project; the basic ideas of open-source licenses—free redistribution, access to source code, and the ability to create derivative works—are fundamental as we have described; yet, the particular conditions and restrictions can vary greatly among different licenses.

Key concepts to understand better the following discussion are **copyleft**, **copyright** and **permissive licenses**.

Copyleft is a general strategy for creating a program (or other work) free; however, it also depends on any updated and expanded versions of the program to be free as well. This guarantees that derivative works maintain the liberties given upon by the original author.

Copyright grants the author of original work exclusive rights to use and distribute her work. In the context of software, it implies that the creator can regulate the way the program is applied, changed, and distributed.

Permissive licenses are those that allow users to do almost anything with the software, as long as they include the original license and notice in any distribution.

Let's briefly examine, as deeply detailed in [36], some of the most often used open-source licenses, the `MIT` License, the `GPL`, and the `Apache` License, with examples to

better grasp the consequences of every strategy.

1. `MIT License:` This is one of the more permissive open-source licenses now in use. It allows users to do almost anything they want with the software, as long as the original copyright notice is included in each distributed copy of the software, it practically gives almost total freedom to use, copy, alter, merge, publish, sublicense and distribute software, even for commercial uses.

   Key points: allows both personal and business application of the software, permits proprietary changes (you can alter the code and release it under another license, even as a closed-source product). Disclaims any warranties, hence users cannot hold the developers accountable for any issues the program generates.

   Example: An open source MIT licensed library can be used in a proprietary software, the only requirement is to include the original MIT license in the distribution, but the proprietary software can be closed source, can modify the library, and not share the changes.

2. `GNU General Public License (GPL):` Being a copyleft license, the GNU GPL requires any derivative work derived from GPL-licensed software to be also distributed under the GPL license. This guarantees that any altered forms of the software stay free and open-source.

   Key points: guarantees open-source nature for derived works and changes as well. Users that share the software—even with modifications—must make the source code available. Strong copyleft: any component of the derivative work employ GPL-licensed code, the whole work must be under the GPL.

   Example: A GPL licensed library can be used in a proprietary software, the proprietary software itself can be closed source, but the library must be distributed under the GPL license, and the source code of the library must be made available to the users of the proprietary software, both if it was modified or not.

3. `Apache License:` Is a permissive open-source license similar to the `MIT` License, with an additional patent clause meant to guard consumers from patent claims by contributors. This guarantees that, regarding their contributions, contributors cannot subsequently sue users or redistributors for patent infringement.

   Key points: allows redistribution and changes including commercial usage. While more complex than the `MIT` License, it includes an express grant of patent rights from contributors and requires preservation of copyright and license notices. It also does not require explicit permission for derivative works.

   Example: For instance, a company adds proprietary modules to software licensed under the `Apache 2.0` License, distributes the modified software, and uses it to run a web server; they must keep the original copyright notices even though

they are not obliged to publish their proprietary alterations as open-source, so that they can't sue users for patent infringement related to the use of the software.

For the `pyOMA2` library, the `MIT` License was chosen, it was a critical decision (probably still open to discussion). We had to take into account possible integration with other softwares in the engineering and scientific computing ecosystem while also complementing the objectives of the project that of encouraging community cooperation and general acceptance.

Making `pyOMA2` open source in an area where private software has always predominated, represents a larger change in scientific and engineering software development. It recognizes the need of openness and repeatability in scientific computing as well as the force of community-driven development.

7.2 Benefits of Open Source

Although one might believe that selling software could result in more immediate income, especially in an engineering and scientific environment, the advantages of using an OSS approach can be somewhat more significant in the long-term.

Here we will explore some of the most important benefits of open-source software.

OSS has several main benefits, one of that is the way it encourages teamwork and the quick technological advancement, as reported in [37], opening the source code invites a broad community of developers to participate to the project, thereby helping identify weaknesses and imperfection, allowing them to proposing solutions at a pace that is hard to match in closed, proprietary software development. This approach is ofter referred to as the "bazaar" model, [38] a concept popularized by Eric Raymond [39] that uses this metaphor to describe the open-source development model, where the software is developed by a large group of developers, in contrast to the "cathedral" model, where the software is developed by a small group of developers.

Reusing software components adds still another major advantage. OSS encourages mass reuse so that several projects may grow on top of one another. Since developers don't have to start from scratch for every new project, they can concentrate more on innovation than repetition [37], this efficiency has real economic value since time saved in development directly translates to reduced expenses for businesses and individuals using the software.

Open-source software is also usually more transparent. In fields including scientific computing or engineering, reproducibility of experiments and the verifiability of algorithms is strictly related to the availability of the source code. This transparency is another key benefit for the scientific community that ensures the reliability of the results since it is under the scrutiny of many.

We can't forget the security aspect, that is one of the strongest points of OSS, despite it might be counterintuitive, since the source code is available to everyone, this

leads it to be under constant analysis by a large community of developers, increasing the probability of identifying and fixing security vulnerabilities that in proprietary software might remain hidden for a long time.

Lastly, longevity is another crucial factor to take into account; proprietary software companies may fail and leave their consumers helpless without updates or support. But since they are not dependent on the success of one company, open-source projects often have longer lifetime. The community can preserve and upgrade the software as long as the project retains attention.

Making `pyOMA2` open-source for our project fits the larger tendency in scientific computing, where openness and teamwork are really appreciated.

Chapter 8

Case Study and Examples

8.1 Experimental Case Study 1: a Laboratory Timber Beam

In this chapter, the application of `pyOMA2` to a real-world laboratory case study is reported. Specifically, a simply supported timber beam has been analyzed to illustrate the library's capabilities, and operability, evidencing the engineering-worthy results that may be obtained with `pyOMA2` (the full script used for this analysis can be found in the appendix A.1).

As depicted in Figures 8.1-8.2, the herein analyzed case study has been taken from the study of Pasca et al. [40]. In that scientific contribution, the scholars performed some OMA tests campaign on a 5.00 m long glulam timber beam with a cross-section dimension of 115×315 mm^2. Characterized by the structural timber grade class GL30C (according to the in-force European norm EN 14080), its nominal average Young's Modulus is 13 GPa, and its mean weight is 430 kg/m^3. This beam structure has been considered in simply supported boundary conditions under white noise excitation, being a typical situation for this kind of laboratory test. However, due to some documented practical difficulties in creating the actual suspension supports, the scholars employed a layer of rockwool insulation placed under the beam to simulate freely suspended conditions. These rockwool panels with size $300 \times 300 \times 100$ mm^3 were located at

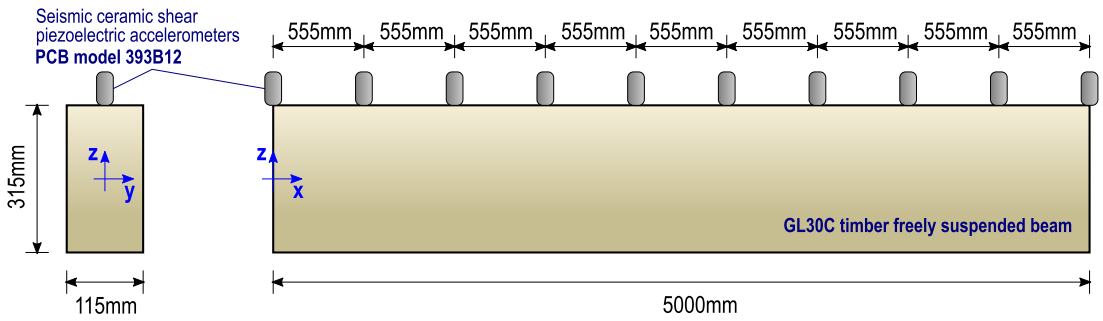


Figure 8.1: `pyOMA2` scheme of experimental application: timber beam dynamic identification.

Figure 8.2: Photo of the experimental setup conducted on the timber beam dynamic identification application.

the beam extremal points and under the midspan of the beam to ensure consistency, repeatability, and clearness in spectral density and stabilization diagram, accepting the possible scattering effects on the damping ratio true evaluations. Furthermore, besides the rockwhool panel effects, it is widely acknowledged that typically high level of uncertainties are related to the damping parameter estimate within the operational output-only identification framework [41].

The acceleration response data were collected using 10 equally spaced seismic uniaxial piezoelectric accelerometers (model PBC 393B12) to monitor vibration responses in the vertical z-direction (gravity direction), thus according to the strong-axis (SA) of the beam cross-section using the typical civil engineering terminology. For the sake of simplicity, in the present thesis document, only the strong-axis measurement campaign has been considered, i.e. employing the test setup specifically illustrated in Figures 8.1-8.2. The piezoelectric accelerometer is wired and connected to a compact Data Acquisition module (cDAQ), acting as a data logger for storing data at a specified sampling frequency. Therefore, 5 minutes of vibration response data were recorded with a sampling frequency of 1200 Hz.

The collected data has been preprocessed by detrending them, and no further preprocessing has been performed. Indeed, since the starting numerical model revealed in [40] a fundamental mode of interest close to 300 Hz, no decimation preprocessing procedures have been applied. In this way, the Nyquist frequency being equal to 600 Hz is located far enough from this mode of interest, thus avoiding jeopardizing its clear identification. These steps are shown in code snippet Listing 8.1.

```python
# Example - Real data set (Corvara Bridge)
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pyoma2.algorithms import FSDD, SSIcov, pLSCF
from pyoma2.functions.plot import plot_mac_matrix
from pyoma2.setup import SingleSetup

# Sampling frequency of the collected data
fs = 1200  # [Hz] Sampling Frequency

# Data was loaded from an external file (e.g., CSV, MAT, etc.)
data: np.ndarray = ...

# Initialize the SingleSetup class with our data and sampling frequency
timber_ss = SingleSetup(data=data, fs=fs)
# detrend data to be centered around zero
timber_ss.detrend_data(type='constant')
```

Listing 8.1: Initialization of the Timber Beam.

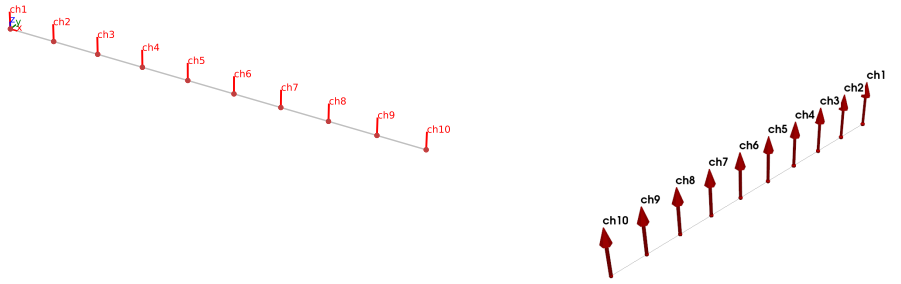
We can optionally plot the geometry of the structure, showing the sensors placement and the direction of the measurements, in 2D or 3D plot using `plot_geo1` and `plot_geo2` methods respectively, as depicted in figures 8.3 and in the example Listing 8.2.

```python
# Define the geometry of the structure using Excel files that contain
# sensor placement and structural layout information

# Load both geometry definitions
# geo1 allows visualization with arrows showing sensor directions
timber_ss.def_geo1_by_file(path=...)
# geo2 enables more advanced 3D visualization and animations
timber_ss.def_geo2_by_file(path=...)

# Plot geometry1 showing sensor placements and directions
fig, ax = timber_ss.plot_geo1()

# Plot geometry2 using PyVista for interactive 3D visualization
_ = timber_ss.plot_geo2()
```

Listing 8.2: Geometry definition of the Timber Beam.

Before performing dynamic identification routines, the best practice includes visualizing collected data, and performing some preliminary evaluation on the quality of the data and the respectfulness of the OMA basic hypothesis, both in qualitative and quantitative terms.

An example of channel information has been depicted in Figure 8.4 from the code execution Listing 8.3.



(a) 2D geometry definition. **(b)** 3D geometry definition.

Figure 8.3: Timber beam case study: geometry definition.

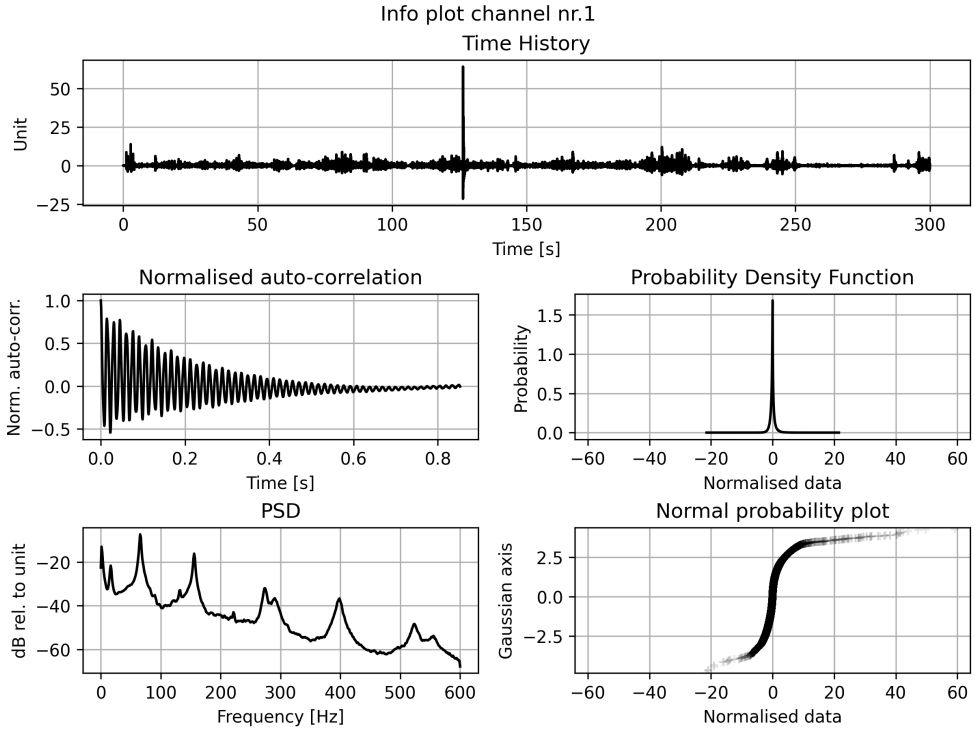


Figure 8.4: Timber beam case study: channel information.

```
# Plot TH, PSD and KDE of the (selected) channels
fig, _ = timber_ss.plot_ch_info(ch_idx=[1])
```

Listing 8.3: Plotting the channel information of the Timber Beam.

Foremost, this graph illustrates the time history of the recorded signal of a selected channel (in this case sensor number 1). Moreover, some statistics and signal processing

metrics are depicted providing to the analysts a quite complete overview of the quality of the information collected. The normalized auto-correlation graph represents the correlation between the signal and a time-lagged version of itself, and it is a key footprint of hidden information from a first glance to the time series data only, e.g. making evident possible trends and other features. The PSD graph represents the power spectral density, which is related to the energy content of the signal spread in the frequency domain axis. This sub-figure permits the inspection of the visible resonance frequencies of the system under study captured from every single channel. The probability density function graph reports the response signal empirical probability distribution. Indeed, if the system is linear, under a white noise excitation, the output response should be also a Gaussian process [41]. This graph permits the diagnosis of the statistics of the recorded signals, evidencing if it is a zero mean Gaussian-distribution-like, or if some dispersions and skewness are present in the monitored responses. The last graph is the normal probability chart, representing a graphical way of conducting a goodness-of-fit statistical test. The test aims to check the null hypothesis that the data were drawn by a specific distribution family, i.e. a Gaussian distribution in this case. If the resulting graph can be well approximated with a linear behavior, this means it is possible to assume that the collected data belong to a Gaussian distribution. Typically, the tails of the distribution are not well represented by a Gaussian distribution, since the extreme value distribution is more suitable to focus on these parts. Nonetheless, it is typically accepted slight variations of the tails, if the majority of the data are well represented by a linear trend, such as in this case (notice the transparency of markers which are darker where many points are overlapped). The analyst assessment of the Gaussianity and stationarity of the monitored vibration response, at least qualitatively using these kinds of graphs, is an indispensable prerequisite for permitting a valid adoption of OMA methods. When these kinds of checks are not completely satisfied, then OMA is probably not the best technique that should be employed for the data analysis, and its results may be insignificant from a physics/engineering standpoint.

After inspecting the statistics and signal processing footprint metrics of the output-only response data, the analysis proceeeds with the initialization of some of the available OMA methods (in this study we'll use the `FSDD`, `SSIcov`, and `pLSCF` algorithms) and we'll start exploring some of the results that can be plotted after the execution of them (Listing 8.4).

```
1
2       # Initialize the OMA algorithms with specific parameters
3       # FSDD for frequency domain analysis
4       fsdd = FSDD(name="FSDD", nxseg=1024,)
5       # SSI for time domain analysis
6       ssicov = SSIcov(name="SSIcov", br=50, ordmax=50)
7       # pLSCF (PolyMAX) for frequency domain analysis
8       plscf = pLSCF(name="polymax", ordmax=30)
9
10      # Add all algorithms to our setup
11      timber_ss.add_algorithms(ssicov, fsdd, plscf)
12
13      # # Run all algorithms all together or one by one
14      # timber_ss.run_by_name("SSIcov")
15      # timber_ss.run_by_name("FSDD")
16      # timber_ss.run_by_name("polymax")
17      timber_ss.run_all()
18
19      # Now we have access to some results
20      # (still need to run Modal Parameter Estimation (MPE))
21      ssi_res = ssicov.result.model_dump()  # or
22      fsdd_res = dict(fsdd.result)
```

Listing 8.4: Initialization and execution of the algorithms for the Timber Beam.

The analysis started with the EFDD algorithm. As reported in Figure 8.5, the singular values decomposition (SVD) of the power spectral density (PSD) have been computed from Listing 8.5 in order to evidence the common peaks among all the measured channels. Thereafter, the peak-peaking method can be performed in order to extract the modal parameter estimates related only to those natural frequencies of actual interest. As documented in [40], disregarding the DC component, it is possible to notice that some small peaks mildly appear between the main peaks of interest at 65.14 Hz, 155.6 Hz, and 271.11 Hz. Those small peaks at about 25 Hz, 62 Hz, and 124 Hz, are referred to as the vibration mode along the weak axis. However, since the herein-considered data are only the ones associated with the strong-axis direction experimental test, the peaks associated with the weak axis are not considered. This highlights the real potentials of the signal processing methodologies of well-enstablished OMA implemented methods. Indeed, the natural frequencies associated with all the fundamental modes of a structural system can be accurately identified, however, the mode shape reconstruction can be difficult or even impossible like in the current case. Indeed, since all the 10 uniaxial accelerometers located along the x-axis of the beam were oriented toward the vertical z-direction, the mode shape that can be retrieved only in this vertical direction, i.e. visualizing them in the x-z plane. Therefore, despite the weak-axis fundamental frequencies can be observed in this SVD graph, the extraction of their mode shapes is impossible considering the current strong-axis-related dataset only, since no channel was placed to capture the x-y plane movements.

```
1
2       # Plot CMIF (Complex Mode Indication Function) from FSDD
3       _, _ = fsdd.plot_CMIF()
4
5       # from these plot we can extract the frequencies that correspond
6       # to the relevant natural frequencies
7       freq = [65.14, 155.6, 271.11]  # [Hz]
```

Listing 8.5: Plot FSDD analysis results for the Timber Beam.

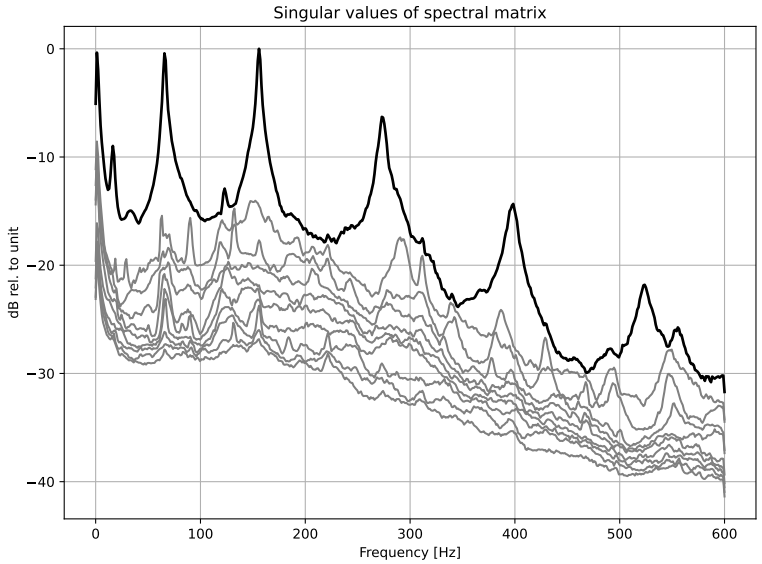


Figure 8.5: Timber beam case study: SVD of the PSD within the EFDD method.

Subsequently, leveraging the EFDD method, it was possible to extract the SDOF bells from the first singular value graph, and finally estimate the remaining modal parameters associated with the selected natural frequencies, respectively referred to as the first, second, and third vibration flexural modes along the strong-axis of the beam (as presented in the code snippet Listing 8.6).

```
1
2       # Modal Parameter Estimation (MPE) for FSDD
3       timber_ss.mpe("FSDD", sel_freq=freq, DF1=0.6)
4       # FDD Frequency Domain Identification (FIT)
5       _, _ = timber_ss[fsdd.name].plot_EFDDfit()
```

Listing 8.6: Modal Parameter Estimation for the Timber Beam and FDD Frequency Domain Identification (FIT).

These identified natural frequencies are 65.14 Hz, 155.6 Hz, and 271.11 Hz, which are in perfect agreement with the numerical and experimental results validated in [40]. The SDOF bell of the first two modes of interest are reported in Figure 8.6, and the

damping estimates are equal to 0.18% and 0.03%, respectively. These values of damping are quite different from the ones illustrated in the reference study [40]. However, this is commonly acceptable, since the damping estimate is always the most uncertain quantity in the identification process, being affected both by epistemic uncertainty, due to modeling errors, and aleatory ones, due to the randomness in the noise in the output-only response signals [41]. It is worth recalling that, in the EFDD method, the single-degree-of-freedom (SDOF) bells are extracted from the multi-degree-of-freedom (MDOF) SVD plot, to simulate a simple free-vibration damped oscillator which oscillates at the selected natural frequency of interest. Then, the time history response is simulated for this SDOF system, and its autocorrelation function is studied to characterize the decreasing law of its peaks. The damping estimate is therefore performed with a logarithmic decrement method, i.e. providing a linear fitting of the peaks over time in the semi-logarithmic plane. The results from EFDD of the third mode in terms of damping estimate were not reported since the autocorrelation did not provide any significant correlation, thus preventing any representative estimate. The mode shape

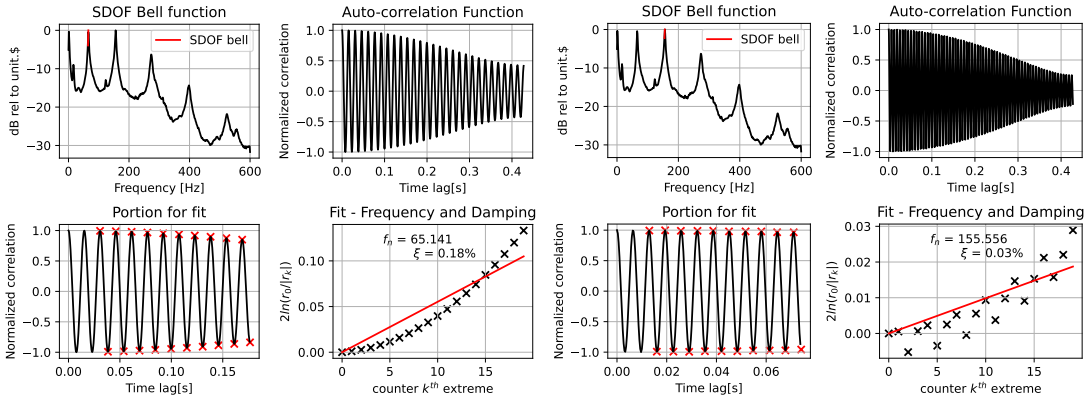


Figure 8.6: Timber beam case study: SVD of the PSD within the EFDD method.

estimates provided by the EFDD are finally evaluated in Listing 8.7 and reported in Figure 8.7, and they are in perfect agreement with the ones in [40], clearly showing the theoretically expected first, second, and third flexural vibration mode shapes of a simply supported beam structure.

```
1
2       # Plot mode shapes
3       MODES_NR = 3
4       for mode_nr in range(MODES_NR):
5           # 2D mode shapes of FSDD
6           _, _ = timber_ss.plot_mode_geo1(
7               algo_res=fsdd.result, mode_nr=mode_nr + 1, view="xz", scaleF=0.5
8           )
9           # 3D mode shapes of FSDD
10          _, _ = timber_ss.plot_mode_geo2_mpl(
```

```
            algo_res=fsdd.result, mode_nr=mode_nr + 1, view="xz", scaleF=2
        )
        # Animate mode shapes of FSDD
        _ = timber_ss.anim_mode_geo2(
            algo_res=fsdd.result, mode_nr=mode_nr + 1, scaleF=2, saveGIF=True
        )
        # the same can be plotted for the other algorithms
```

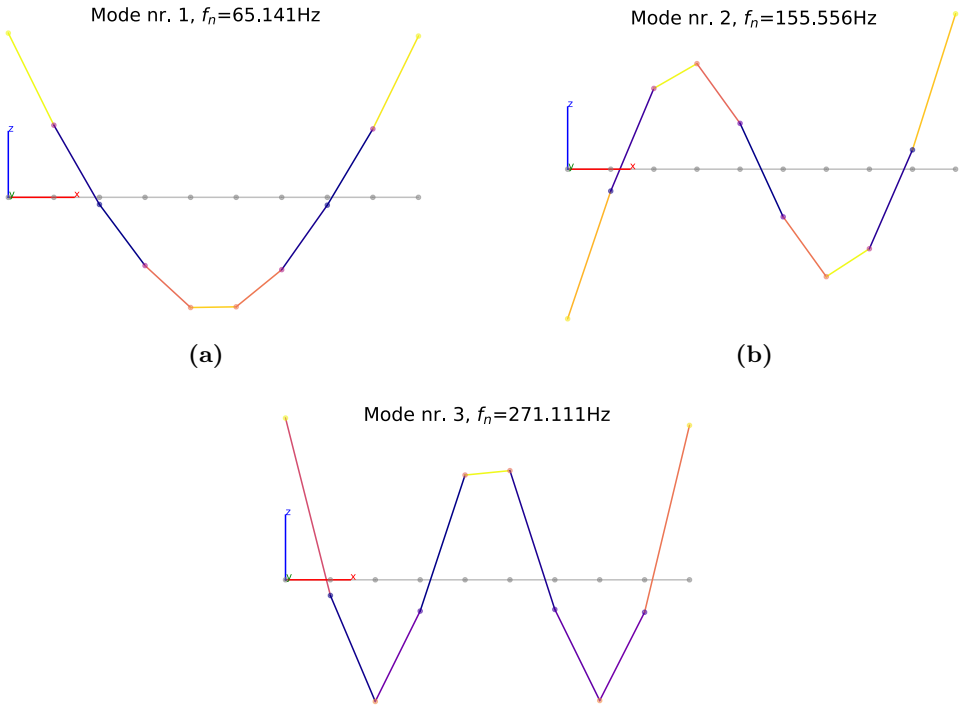
Listing 8.7: Plotting the mode shapes of the Timber Beam (FSDD).

The modal parameter estimates have been also conducted using the time-domain method covariance-based version of the stochastic subspace identification (SSI-cov) algorithm. The stabilization diagram, depicted in Figure 8.8 (a) and Listing 8.8, has been computed setting both the number of block rows and the maximum order equal to 50.

```

    # Plot SSI stabilization diagram
    _, _ = ssicov.plot_stab(freqlim=(0, timber_ss.fs / 2), hide_poles=False)
    # plot frequecy-damping clusters for SSI
    _, _ = ssicov.plot_cluster()

    # and for the stabilization diagram we can extract the minimum order where
    # the poles are stable in correlation with the frequencies
    orders_ssi = [10, 10, 10]  # [int]
```

Listing 8.8: Plot SSIcov analysis results for the Timber Beam.

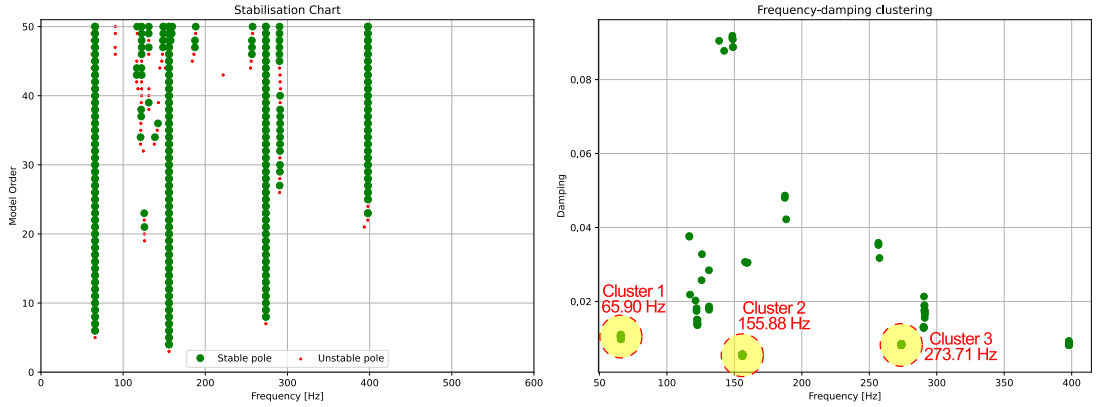
The red and green colors of the poles denote respectively the unstable and the completely stable mathematical solutions, i.e. the ones respecting all the stability physical-based criteria in terms of frequency, damping, and mode shape modal assurance criterion (MAC), which is a correlation measure between mode shapes arrays. Figure 8.8 (a) highlights clear stable pole alignments with the progressive increase of model order of analysis in correspondence with the strong-axis natural frequency of interest, i.e. 65.90 Hz, 155.88 Hz, and 273.71 Hz. Furthermore, Figure 8.8 (b) shows very concentrated clusters of damping values for the identified stable poles related only to the three before-mentioned natural frequencies of interest, providing a clear estimate of the damping ratios for the three vertical flexural modes equal to 1.06 %, 0.56 %, and 0.83 %, respectively. These latter results are in better agreement with the reference results in [40] rather than the ones obtained with the previous EFDD method. The mode shapes derived with the SSIcov method are very similar to the ones illustrated in Figure 8.7, and therefore have not been reported here. This similarity between the mode shapes obtained from the EFDD and SSIcov methods has been illustrated in Figure 8.9 with the cross MAC matrix computation (in code Listing 8.9.



(a) **(b)**

(c)

Figure 8.7: Timber beam case study: mode shape estimates from EFDD method.



(a) **(b)**

Figure 8.8: Timber beam case study: visualization of the results obtained from SSIcov method, i.e. stabilization diagram (a) and frequency vs damping graph (b).

```
from pyoma2.functions.plot import plot_mac_matrix

# Select modes manually, extracting them from plot
timber_ss.mpe_from_plot("SSIcov")
# or directly
timber_ss.mpe("SSIcov", sel_freq=freq, order=orders_ssi)


```

```
9       ssi_res = ssicov.result.model_dump()
10      fsdd_res = fsdd.result.model_dump()
11
12      figure, axes = plot_mac_matrix(ssi_res["Phi"].real, fsdd_res["Phi"].real)
```

Listing 8.9: Modal Parameter Estimation for the Timber Beam and MAC matrix comparison.

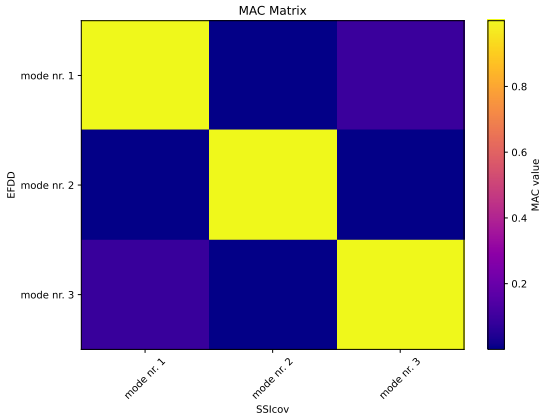


Figure 8.9: Timber beam case study: cross MAC matrix for evaluating the correlation between mode shapes estimates provided by EFDD and SSIcov methods.

Eventually, the same analysis has been conducted again with the polymax method, and the stabilization results is reported in Figure 8.10. With this latter technique, the

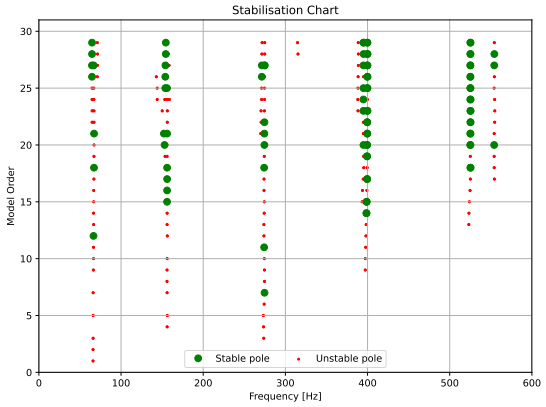


Figure 8.10: Timber beam case study: visualization of the stabilization diagram obtained from the polymax method.

stability criteria are pretty much stringent, and less stable poles are returned. Anyway, the stable poles alignments are in a good agreement with the ones located with the SSIcov method, thus delivering the first natural frequencies at 67.06 Hz, 156.04 Hz, and 274.12 Hz. Nevertheless, the relative damping ratio estimates for these three fundamental modes of interest are in less agreement with the reference study [40], thus providing generalized lower values equal to 0.67 %, 0.47 %, and 0.60 % respectively.

The comparative results between the reference literature study [40] and the currently analyzed data have been finally summarized in Table 8.1. It is worth noting that

	Pasca et al. [40]				
	Experimetal campaign				Numerical
	EFDD		SSIcov		Model
Mode	Freq. (Hz)	Damping (%)	Freq. (Hz)	Damping (%)	Freq. (Hz)
1-Flex-SA	65.98	1.00	65.50	1.03	67.57
2-Flex-SA	155.97	0.54	154.07	0.59	166.79
3-Flex-SA	273.68	0.73	273.90	0.80	286.77

	pyOMA2					
	EFDD		SSIcov		Polymax	
Mode	Freq. (Hz)	Damping (%)	Freq. (Hz)	Damping (%)	Freq. (Hz)	Damping (%)
1-Flex-SA	65.14	0.18	65.90	1.06	67.06	0.67
2-Flex-SA	155.56	0.03	155.88	0.56	156.04	0.47
3-Flex-SA	271.11	-	273.71	0.83	274.12	0.60

Table 8.1: Comparison of modal parameters estimates obtained with pyOMA2 and the reference study Pasca et al. [40] for the strong-axis case only.

	Pasca et al. [40]		pyOMA2		Relative Differences	
Mode	Freq. (Hz)	Damping (%)	Freq. (Hz)	Damping (%)	Freq. (%)	Damping (%)
1-Flex-SA	65.50	1.03	65.90	1.06	-0.60	-3.11
2-Flex-SA	154.07	0.59	155.88	0.56	-1.18	4.88
3-Flex-SA	273.90	0.80	273.71	0.83	0.07	-3.67

Table 8.2: Relative difference of modal parameters estimates obtained with SSIcov method between pyOMA2 and the reference study Pasca et al. [40] for the strong-axis case only.

the best agreement for both natural frequencies and damping ratios has been found with the use of the SSIcov algorithm, whose relative differences are below 5% as testified in Table 8.2. For all the other cases, the damping estimates exhibit a quite big scatter between the current analysis and the reference study. Nevertheless, it is worth reminding that the reason for these differences can be related to the use of rockwhool panels to simulate the suspension boundary conditions of the timber beam. Indeed, as reported by the authors in their study [40], the use of rockwhool requires accepting the possible scattering effects on the damping ratio true evaluations, besides the already inherent uncertainties associated with this modal parameter estimate in the operational output-only identification process.

8.2 Experimental Case Study 2: The Corvara Bridge

To further demonstrate pyOMA2's capabilities in analyzing more complex real-world structures, another case study has been analyzed. Therefore, in the present Section, a brief examination of an existing concrete roadway bridge viaduct dynamic identification is illustrated. In particular, this second case study is denoted as Corvara Bridge, and it is located in the municipality of Pescara, Italy. The bridge dates back to 1987 and, despite the rain-water drainage system is sometimes defective and dedicated bearing support devices are missing at the top of piers, this structure is characterized by an overall good conservation state. It is worth underlining that the present case study

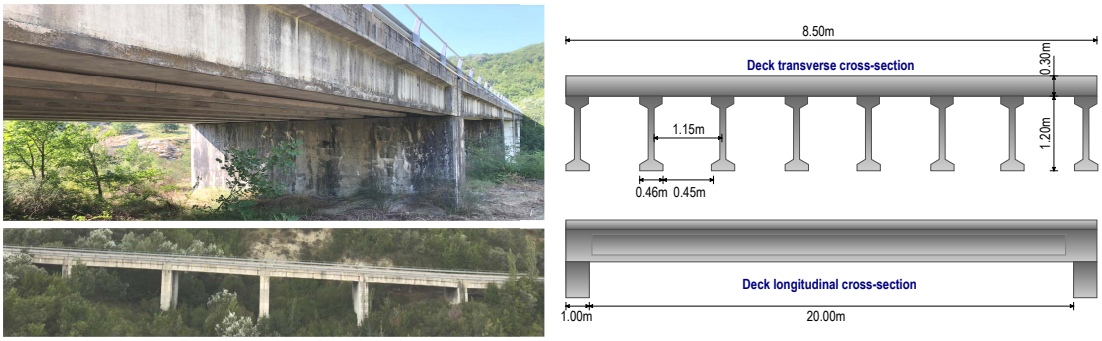


Figure 8.11: Corvara bridge case study: Transverse cross-section of the Corvara bridge.

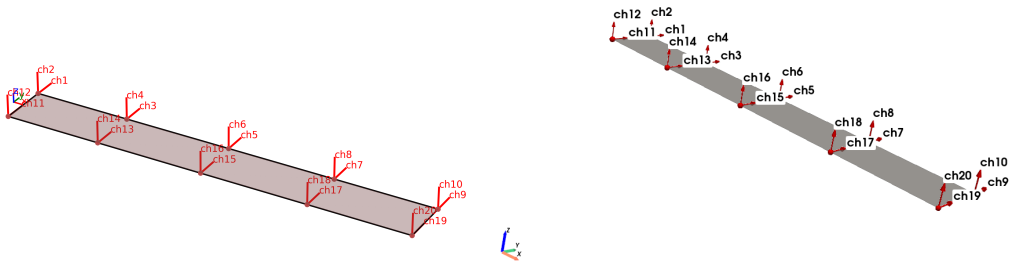
(a) **(b)**

Figure 8.12: Corvara bridge case study: wired data acquisition setup.

provides an important contrast to the controlled conditions of the laboratory environment for the previous timber beam experiment, illustrating the additional complexities encountered when working with in-field data, collected from more sophisticated, and large-scale structural systems. The present bridge structure is the subject of two literature studies [42, 43].

As illustrated in Figure 8.11, the bridge is totally composed of 7 equal spans, and the deck of each span presents a transverse cross-section composed of 8 girders with a concrete topping slab. The monitoring campaign has been repeated for every single span, testing a single span at a time, thus collecting 1 hour of vibration response data at a sampling frequency of 200 Hz under natural environmental excitation with traffic closure. The setup deployed for each single span includes 10 bi-axial accelerometers strategically placed over two measurement chains to capture both vertical and torsional vibration modes, as illustrated in Figure 8.12. Figure 8.13 presents the geometric configuration and sensor placement scheme, highlighting the more complex three-dimensional nature of the structure compared to the previous laboratory specimen.

The initial examination of the acceleration data reveals the inherent challenges of field measurements. As shown in Figure 8.14, the channel information graphs display more complex patterns than those observed in the laboratory setting, including



(a) Plot of the geometry of the Corvara bridge span. **(b)** 3D visualization of the Corvara bridge span.

Figure 8.13: Corvara bridge case study: Geometry definition.

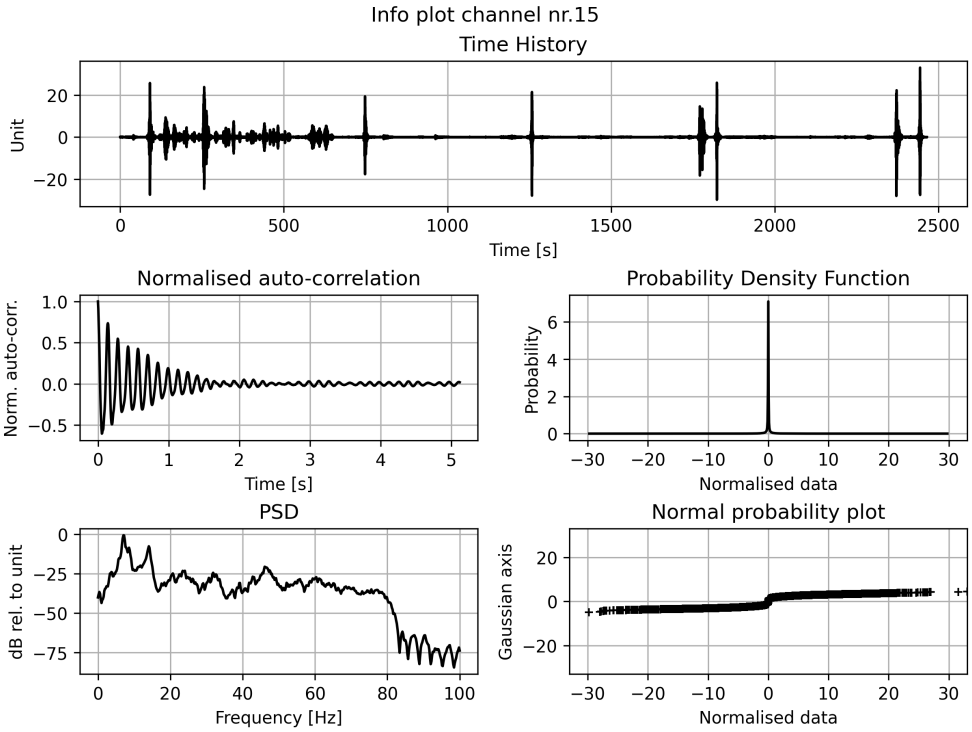


Figure 8.14: Corvara bridge case study: Channel information before decimation.

higher noise levels due to environmental factors and possibly the presence of various non-stationary components in the signal, and also some spikes. Nevertheless, the PSD graph evidenced the presence of some resonant peaks within 20 Hz. Therefore, these characteristics necessitate more careful preprocessing, including the application of decimation to improve signal-to-noise ratios and focus on the frequency range of interest within 20 Hz. In this case, a decimation factor equal to 4 has been used, and the resulting information of the midspan vertical channel has been represented in Figure 8.15.

The subsequent SVD of the PSD graph in Figure 8.16 demonstrated the increased difficulty in identifying structural modes compared to the laboratory case. While the timber beam exhibited clear, well-separated peaks, the bridge data shows multiple

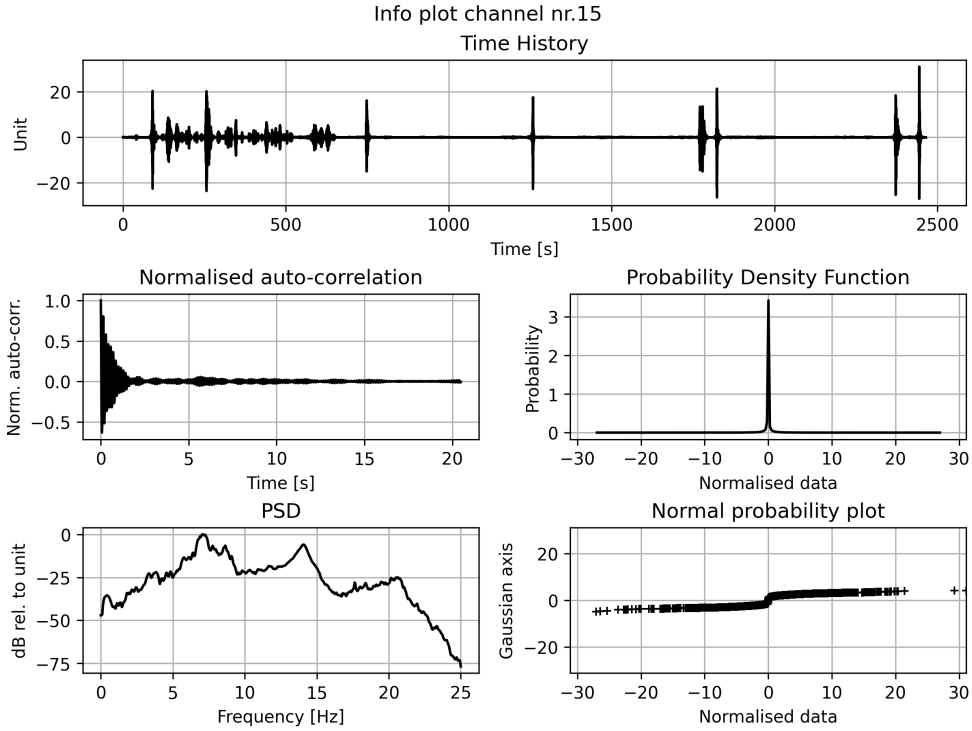


Figure 8.15: Corvara bridge case study: Channel information after decimation.

closely-spaced modes and less distinct peak separation. Nevertheless, even in this more complex case, the acknowledged OMA methods implemented in `pyOMA2`, were able to provide the analysts with all the instruments for conducting their analysis. The beneficial effects of the decimation preprocessing permitted the identification of at least the three main peaks over the first singular value line, located at 7.19 Hz, 8.66 Hz, and 14.13 Hz, respectively. These three identified natural frequencies are in good agreement with the literature studies [42, 43]. Similar findings can be retrieved from the SSIcov algorithm stabilization chart shown in Figure 8.17. Even in this case, the greater complexity of this OMA case study rather than the laboratory timber beam results in a more crowded stabilization diagram, in which not always a simple interpretation nor a regular detection of stable poles' alignments is possible with ease. In this case, the SSIcov algorithm still permitted the identification of fundamental frequencies similar to the FDD method, i.e. at 7.10 Hz, 8.69 Hz, and 14.09 Hz. In order to provide a more reliable identification of the best modal parameters truly representative of the intrinsic dynamical behavior of the structural system under study, a best practice consists of overlapping the stabilization diagram with at least the first singular value line. This approach allows for a mutual validation of the two dual techniques. Indeed, despite their different theoretical/practical basis, since they operate in the two dual time and frequency domains, both methods' ultimate goal is the same, i.e. seeking the true modal parameters of the structure. Therefore, considering Figure 8.18, the previously

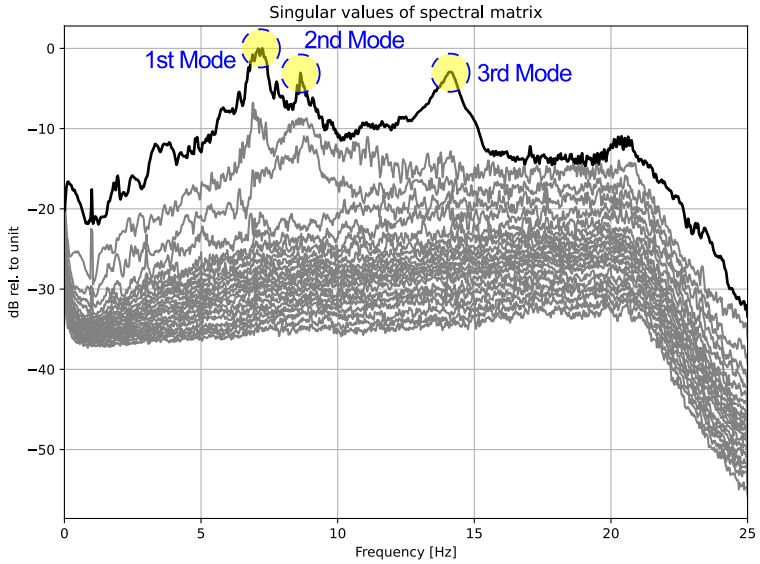


Figure 8.16: Corvara bridge case study: SVD of the PSD within the EFDD method.

identified frequencies are better confirmed by both techniques. The mode shape parameters estimates using for instance the EFDD method have been reported in Figure 8.19, finally delivering damping ratio estimates equal to 1.00 %, 0.03 %, and 0.27 % for the three identified modes.

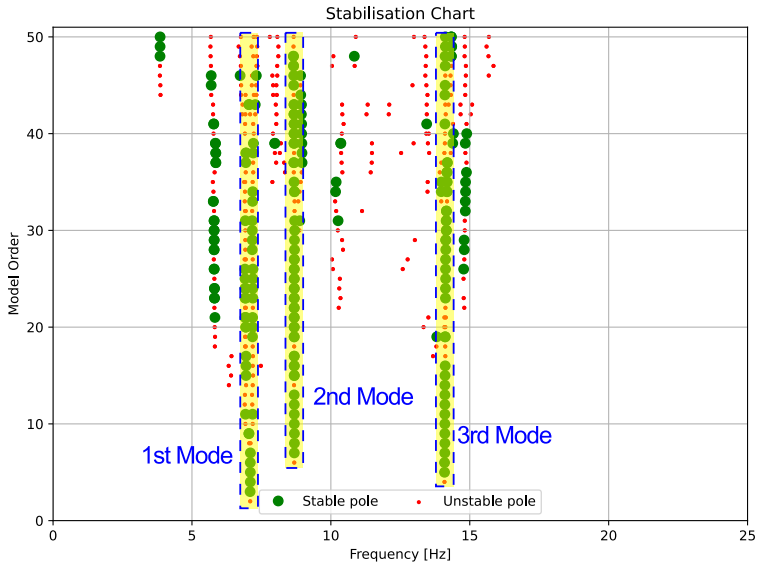


Figure 8.17: Corvara bridge case study: SSIcov Stabilization Diagram.

In the end, the identified mode shapes associated with the three identified modes have been visualized through both static and animated representations (Figures 8.20, 8.21, 8.22). They reveal the complex three-dimensional behavior of the bridge structure. The animations particularly highlight the coupling between vertical and torsional movements, a phenomenon not observed in the simpler timber beam case. From the

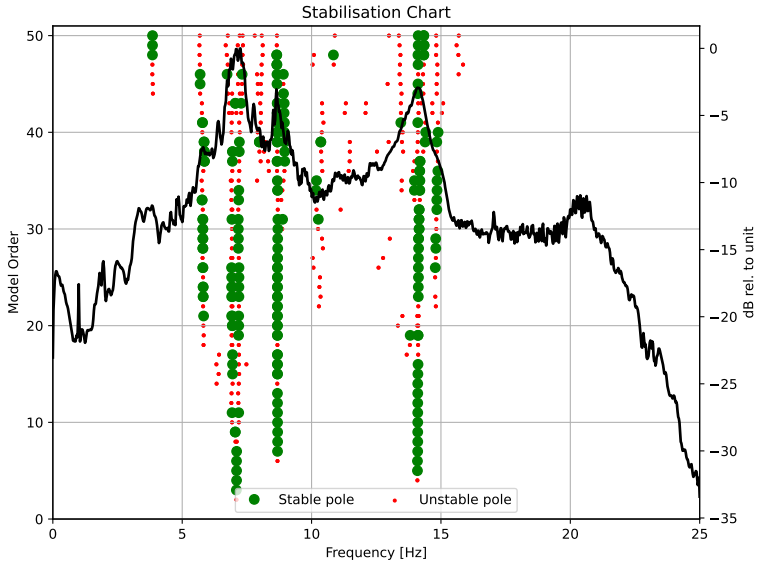


Figure 8.18: Corvara bridge case study: SSIcov Stabilization Diagram with overlap the first singular value line of the PSD.

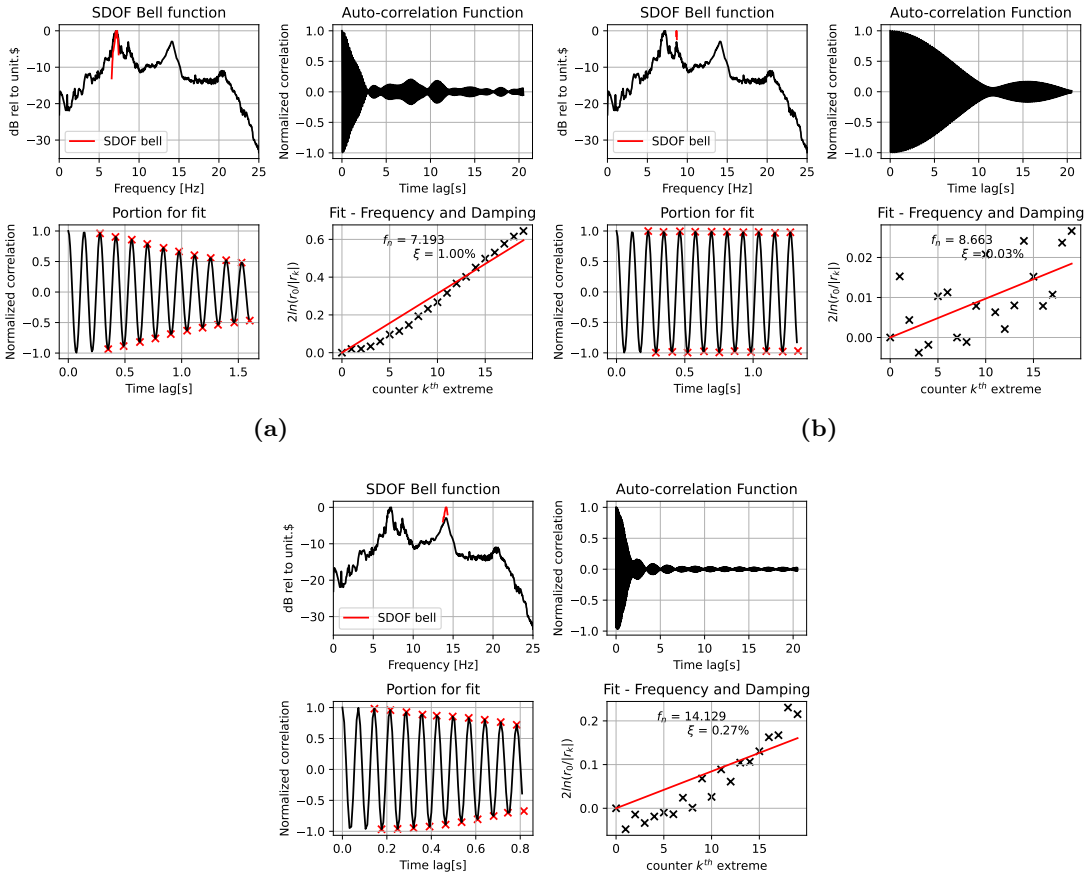
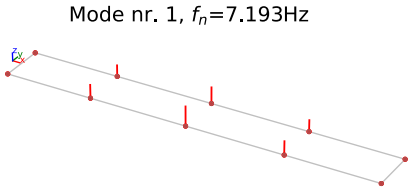
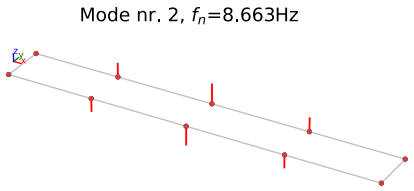


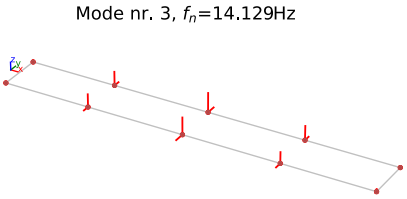
Figure 8.19: Corvara bridge case study: modal parameter estimates using EFDD method.



(a) FSDD mode nr. 1 - 2D mode shape.

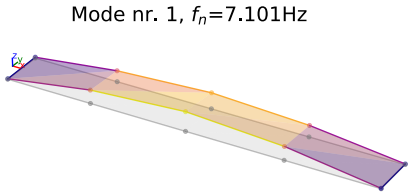


(b) FSDD mode nr. 2 - 2D mode shape.

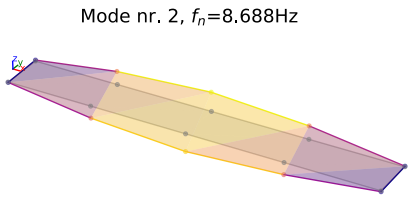


(c) FSDD mode nr. 3 - 2D mode shape.

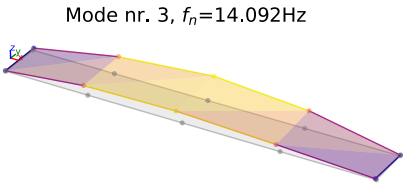
Figure 8.20: 2D mode Shapes of the Corvara Bridge Span using FSDD



(a) SSIcov mode nr. 1 - 3D mode shape.



(b) SSIcov mode nr. 2 - 3D mode shape.

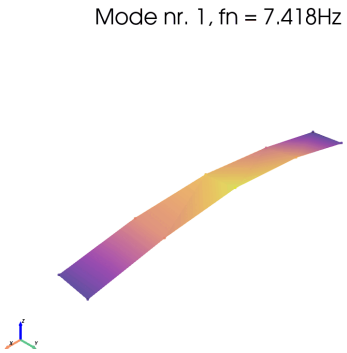


(c) SSIcov mode nr. 3 - 3D mode shape.

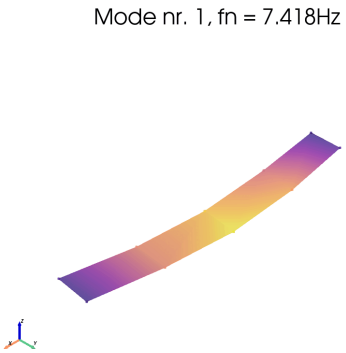
Figure 8.21: 3D Mode Shapes of the Corvara Bridge Span using SSIcov

2D plot, we can easily identify the directions of the vibrations, from these we can see that mode 1 is a vertical vibration, mode 2 is a torsional vibration, and mode 3 is a translational coupled vertical and transverse vibration. The 3D mode shape plots of the structure, provide a more detailed view of the mode shapes, here we show both the static 3D mode shape plot and some frames of the animated mode shape plot. In the animated mode shapes, we can see the structure's movement at each natural frequency, the animation provides a more intuitive understanding of the mode shapes.

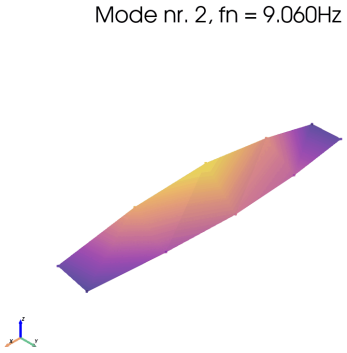
In conclusion, we have shown the capabilities of the `pyOMA2` library in the analysis of a more complex real-world existing large-scale structure, showing the results that can be obtained with the library. It is worth underlining that in the reference literature studies [42, 43], the scholars adopted the first version of the PyOMA module. The accuracy of the OMA results obtained with the new library `pyOMA2` didn't change from the literature study, however, the usability of the newly implemented package interface and the new functionalities completely revolutionized the analysts' experience.



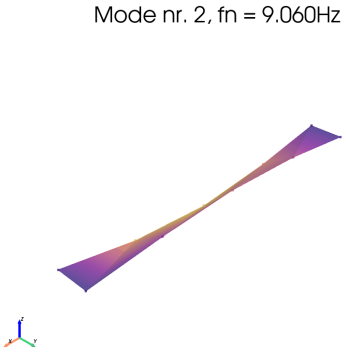
(a) FSDD mode nr. 1 - Frame 1



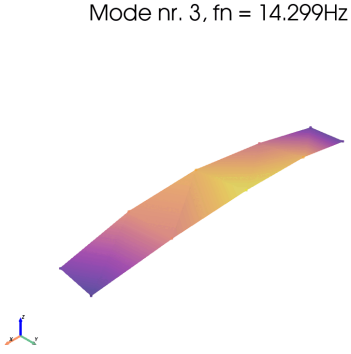
(b) FSDD mode nr. 1 - Frame 2



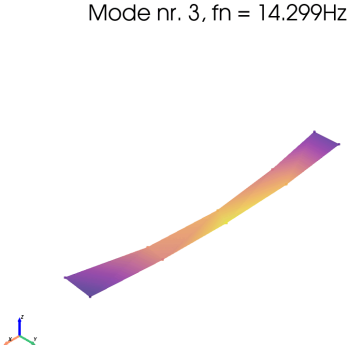
(c) FSDD mode nr. 2 - Frame 1



(d) FSDD mode nr. 2 - Frame 2



(e) FSDD mode nr. 3 - Frame 1



(f) FSDD mode nr. 3 - Frame 2

Figure 8.22: Animated Mode Shape Frames for Modes 1–3 using FSDD

Chapter 9

Conclusion and Future Work

In this dissertation, we set out the ambitious goal of creating an open-source Python library for Operational Modal Analysis (OMA), a crucial branch of dynamic identification of structures, for practical and both research interests, especially for civil engineering, mechanical engineering, and aerospace engineering fields among the others. Dynamic identification refers to the field of study of characterizing the intrinsic properties of vibrating structural systems, denoted as modal parameters, able to completely describe their dynamical behavior. Specifically, OMA comprises those techniques dedicated to the dynamic characterization based on output-only response vibration signals measured from the structure of interest, such as acceleration, velocity, and displacement time series, i.e. without the knowledge of the input excitation that generated those responses. Our ambitious goal was concretized in the `pyOMA2` package, which allowed the creation of a modular, extensible, and user-friendly tool to address the topic of Structural Health Monitoring, making it a flexible answer for both practical SHM engineers and academic researchers. The main objective was not only to provide access to these functionalities but also to make them available for the open-source community, where there are still few projects in Python that address this important topic.

Foremost, for the sake of clearness and completeness, these initial paragraphs report a short review of the followed path during this project, in order to provide a stand-alone and cohesive perspective on the conducted research. Designed around Setup classes, the modular architecture of `pyOMA2` has shown a strong framework able to coordinate the whole OMA process. This covers running sophisticated algorithms, preparing raw data, and displaying results via understandable visuals. Furthermore, the library is made to be readily maintainable and scalable by following software engineering best practices, such as the SOLID principles and some of the most common design patterns around Object Oriented Programming. Regardless of these advantages and the extensive efforts invested in this project together with our satisfaction with the outcomes—as evidenced by its sponsorship at conferences ([44]), increasing number of downloads on `PyPI` (No. downloads from GitHub), and initial community contributions, several improving aspects still necessitate our attention and future potential enhancement.

As a matter of fact, the project is still in its early stages, and we expect a lot of feedback, such as bug reports, and feature requests, from the community as the number of users increases. For instance, even if the modularity guarantees extension, the present approach still requires users to be generally familiar with Python and its ecosystem, this technical need could be a barrier for some users. Moreover, even if the implemented algorithms have been tested extensively, their performance in real-world situations with high noise levels or unusual setups could indicate the need for more improvement and new functionalities to further extend the library's versatility. Firstly, as discussed in 6.4.2, improving the scope and quality of automated testing aims not only to increase test coverage but also to address the correctness of what is tested. This is an important factor in ensuring the dependability and stability of the code. Moreover, the future integration of a graphical user interface (GUI), in line with what was done with the previous version (`PyOMA`), could show a notable improvement; this would thus enable users to use the software without the need of having coding knowledge, so making modal analysis more accessible to a larger audience including those without programming experience. From the very beginning of this specific dissertation project, it was decided to exclude the graphical interface component from the `pyOMA2` package to preserve the project's modularity, and we considered, in the near future, the feasibility of creating an independent GUI project that utilizes `pyOMA2` as a dependency and functions as a "back-end". A platform-independent GUI should be chased to reach as many users as possible, regardless of the specific Operating System they are used to working on. For instance, this cross-platform feature could also permit potential integrations with existing infrastructure asset management platforms, ensuring the compatibility of `pyOMA2` in real-world continuous monitoring systems. This may ensure typical requirements of communication and decentralized control related to contracts with public owners and stakeholders which often use different applications and platforms for collecting, storing, and managing monitoring data and analyses. Besides all of these aspects, user-friendly and intuitive GUI programming should be the key factors to focus on for pursuing this important milestone in promising future releases.

Another area of interest for future developments should focus on enhancing the performance of diverse computations, beginning with lower-level tasks—like parallel matrix computations, and then extending to higher-level operations like in Setup classes, where we might parallelize algorithm computations and data merging. For the first version of the project, the main effort was focused on modularity, extensibility, and ease of use; for the next version, we could work also on the performance aspects. Finally, it is impossible not to mention the possibility of updating the families of supported algorithms, following the ongoing research being conducted in the field of OMA and dynamic characterization of structures.

In conclusion, the `pyOMA2` project marks a major first in widespread access to sophisticated SHM tools. Modern algorithms married with Python's adaptability create the foundation for a new paradigm in OMA software development. The future-forward

is still to be written, but we are optimistic about the contributions and impact that `pyOMA2` will continue to make to the SHM and engineering communities.

Appendix A

Timber Beam Example - Runnable Script

```
"""
Appendix X: Modal Analysis Implementation
=========================================

pyOMA2 - version 1.0.0

This script implements modal analysis procedures for structural dynamics using multiple identification
methods: FSDD (Frequency Spatial Domain Decomposition), SSIcov (Covariance-driven Stochastic Subspace
Identification), and pLSCF (poly-Least Squares Complex Frequency-domain).

Author: [Margoni Diego Federico]
Date: [December 2024]
"""

# ============================================================================
# Section 0: Import Dependencies
# ============================================================================

from pathlib import Path

import matplotlib.pyplot as plt
import mplcursors
import numpy as np
import pandas as pd
from pyoma2.algorithms import FSDD, SSIcov, pLSCF
from pyoma2.functions.plot import plot_mac_matrix
from pyoma2.setup import SingleSetup

# ============================================================================
# Section 1: Path Setup and Configuration
# ============================================================================
from pathlib import Path

# Define input/output paths and measurement parameters
DATA_PATH = Path("TRAVE1")
DATA_FILENAME = "TRAVE1_DATA"
DATA_FILE_EXT = ".xlsx"
OUTPUT_PATH = Path("RESULTS") / "trave1_results_1cuscino"
SAMPLING_FREQ = 1200  # Sampling Frequency in Hz

# ============================================================================
# Section 2: Data Loading and Preprocessing
# ============================================================================
```

```python

    # Create output directory if it doesn't exist
    OUTPUT_PATH.mkdir(parents=True, exist_ok=True)

    # Define full paths for data files
    excel_file = DATA_PATH / f"{DATA_FILENAME}{DATA_FILE_EXT}"
    parquet_file = DATA_PATH / f"{DATA_FILENAME}.parquet"

    # Load data from parquet file if exists, otherwise create from Excel
    if not parquet_file.exists():
        print("Data file not found, creating it...")
        data = pd.read_excel(excel_file, header=None).dropna()
        data.to_parquet(parquet_file)

    data = pd.read_parquet(parquet_file).to_numpy()

    # Initialize single setup analysis
    timber_ss = SingleSetup(data, fs=SAMPLING_FREQ)
    timber_ss.detrend_data(type="constant")  # Remove DC offset

    # Load geometry definitions
    _geo1 = "Geo1_timber.xlsx"
    _geo2 = "Geo2_timber.xlsx"
    timber_ss.def_geo1_by_file(_geo1)
    timber_ss.def_geo2_by_file(_geo2)

    # ================================================================================
    # Section 3: Geometry Visualization
    # ================================================================================

    # Plot and save geometry representations
    _, _ = timber_ss.plot_geo1(scaleF=0.2)
    plt.savefig(OUTPUT_PATH / "01_plot_geo1.png", dpi=300, bbox_inches="tight")
    plt.savefig(OUTPUT_PATH / "01_plot_geo1.pdf", bbox_inches="tight")

    pyvista_plotter = timber_ss.plot_geo2(scaleF=0.8)
    pyvista_plotter.screenshot(OUTPUT_PATH / "01_plot_geo2.png")

    # Plot time history, PSD and KDE for selected channels
    _, _ = timber_ss.plot_ch_info(ch_idx=[1])
    plt.savefig(
        OUTPUT_PATH / "03_plot_ch_info_no_decim.png", dpi=300, bbox_inches="tight"
    )
    plt.savefig(OUTPUT_PATH / "03_plot_ch_info_no_decim.pdf", bbox_inches="tight")

    # ================================================================================
    # Section 4: Modal Analysis Algorithm Implementation
    # ================================================================================

    # Initialize modal analysis algorithms
    fsdd = FSDD(name="FSDD", nxseg=1024)  # Frequency Spatial Domain Decomposition
    ssicov = SSIcov(name="SSIcov", br=50, ordmax=50)  # Stochastic Subspace Identification
    plscf = pLSCF(name="polymax", ordmax=30)  # poly-Least Squares Complex Frequency

    # Add algorithms to analysis setup
    timber_ss.add_algorithms(ssicov, fsdd, plscf)

    # Execute all algorithms
    timber_ss.run_by_name("SSIcov")
    timber_ss.run_by_name("FSDD")
    timber_ss.run_by_name("polymax")

    # Store results
    ssi_res = ssicov.result.model_dump()
    fsdd_res = dict(fsdd.result)

    # ================================================================================
    # Section 5: FSDD Analysis
    # ================================================================================

```

```python
    # Plot and save Singular Value Decomposition results
    _, _ = fsdd.plot_CMIF()
    mplcursors.cursor()
    plt.savefig(OUTPUT_PATH / "04_plot_SVD_matrix.png", dpi=300, bbox_inches="tight")
    plt.savefig(OUTPUT_PATH / "04_plot_SVD_matrix.pdf", bbox_inches="tight")

    # Define frequencies of interest and perform modal parameter estimation
    freq = [65.6, 155.9, 273.0]
    timber_ss.mpe("FSDD", sel_freq=freq, DF1=0.6)

    # Save FSDD results
    fsdd_res = dict(fsdd.result)
    np.savetxt(OUTPUT_PATH / "05_fsdd_res_Fn.txt", fsdd_res["Fn"], fmt="%.6f")
    np.savetxt(OUTPUT_PATH / "05_fsdd_res_Xi.txt", fsdd_res["Xi"], fmt="%.6f")
    np.savetxt(OUTPUT_PATH / "05_fsdd_res_Phi.txt", fsdd_res["Phi"], fmt="%.6f")

    # ================================================================================
    # Section 6: Mode Shape Visualization (FSDD)
    # ================================================================================

    # Plot EFDD fits
    plt.close("all")
    _, _ = timber_ss[fsdd.name].plot_EFDDfit()

    # Save individual mode fits
    for ii in plt.get_fignums():
        plt.figure(ii)
        plt.savefig(
            OUTPUT_PATH / f"06_plot_EFDDfit_Mode_{ii+1}.png",
            dpi=300,
            bbox_inches="tight",
        )
        plt.savefig(
            OUTPUT_PATH / f"06_plot_EFDDfit_Mode_{ii+1}.pdf", bbox_inches="tight"
        )

    # Plot mode shapes for geometry 1
    for mode_nr in range(1, 4):
        _, _ = timber_ss.plot_mode_geo1(
            algo_res=fsdd.result, mode_nr=mode_nr, view="3D", scaleF=0.5
        )
        plt.savefig(
            OUTPUT_PATH / f"07_plot_EFDD_Mode_SHAPE_{mode_nr}.png",
            dpi=300,
            bbox_inches="tight",
        )
        plt.savefig(
            OUTPUT_PATH / f"07_plot_EFDD_Mode_SHAPE_{mode_nr}.pdf",
            bbox_inches="tight",
        )

    # Plot mode shapes for geometry 2
    for mode_nr in range(1, 4):
        figure, _ = timber_ss.plot_mode_geo2_mpl(
            algo_res=fsdd.result, mode_nr=mode_nr, view="3D", scaleF=2
        )
        figure.savefig(
            OUTPUT_PATH / f"08_plot_EFDD_Mode_SHAPE_{mode_nr}_geo2.png",
            dpi=300,
            bbox_inches="tight",
        )
        figure.savefig(
            OUTPUT_PATH / f"08_plot_EFDD_Mode_SHAPE_{mode_nr}_geo2.pdf",
            bbox_inches="tight",
        )

    # Generate animated visualizations
    for mode_nr in range(1, 4):
        _ = timber_ss.anim_mode_g2(
            algo_res=fsdd.result, mode_nr=mode_nr, scaleF=2, saveGIF=True
```

```
        )

    # ================================================================================
    # Section 7: SSI Analysis
    # ================================================================================

    # Plot stabilization diagram
    _, _ = ssicov.plot_stab(freqlim=(0, timber_ss.fs / 2), hide_poles=False)
    mplcursors.cursor()
    plt.savefig(
        OUTPUT_PATH / "09_ssicov_plot_stab.png", dpi=300, bbox_inches="tight"
    )
    plt.savefig(OUTPUT_PATH / "09_ssicov_plot_stab.pdf", bbox_inches="tight")

    # Plot frequency-damping clusters
    _, _ = ssicov.plot_cluster()
    mplcursors.cursor()
    plt.savefig(OUTPUT_PATH / "10_plot_cluster.png", dpi=300, bbox_inches="tight")
    plt.savefig(OUTPUT_PATH / "10_plot_cluster.pdf", bbox_inches="tight")

    # Modal parameter estimation
    orders_ssi = [10, 10, 10]
    timber_ss.mpe("SSIcov", sel_freq=freq, order=orders_ssi)

    # ================================================================================
    # Section 8: SSI Mode Shape Visualization
    # ================================================================================

    # Save SSI results
    ssi_res = dict(ssicov.result)
    np.savetxt(OUTPUT_PATH / "11_ssi_res_Fn.txt", ssi_res["Fn"], fmt="%.6f")
    np.savetxt(OUTPUT_PATH / "11_ssi_res_Xi.txt", ssi_res["Xi"], fmt="%.6f")
    np.savetxt(OUTPUT_PATH / "11_ssi_res_Phi.txt", ssi_res["Phi"], fmt="%.6f")

    # Plot and save mode shapes for both geometries
    for mode_nr in range(1, 4):
        # Geometry 1
        _, _ = timber_ss.plot_mode_geo1(
            algo_res=ssicov.result, mode_nr=mode_nr, view="3D", scaleF=0.5
        )
        plt.savefig(
            OUTPUT_PATH / f"12_plot_SSIcov_Mode_SHAPE_{mode_nr}.png",
            dpi=300,
            bbox_inches="tight",
        )
        plt.savefig(
            OUTPUT_PATH / f"12_plot_SSIcov_Mode_SHAPE_{mode_nr}.pdf",
            bbox_inches="tight",
        )

        # Geometry 2
        figure, _ = timber_ss.plot_mode_geo2_mpl(
            algo_res=ssicov.result, mode_nr=mode_nr, view="3D", scaleF=2
        )
        figure.savefig(
            OUTPUT_PATH / f"13_plot_SSIcov_Mode_SHAPE_{mode_nr}_geo2.png",
            dpi=300,
            bbox_inches="tight",
        )
        figure.savefig(
            OUTPUT_PATH / f"13_plot_SSIcov_Mode_SHAPE_{mode_nr}_geo2.pdf",
            bbox_inches="tight",
        )

    # Generate animated mode shapes
    for mode_nr in range(1, 4):
        _ = timber_ss.anim_mode_g2(
            algo_res=ssicov.result, mode_nr=mode_nr, scaleF=2, saveGIF=True
        )

```

```python
    # ================================================================================
    # Section 9: pLSCF Analysis
    # ================================================================================

    # Plot stabilization diagram
    _, _ = plscf.plot_stab(freqlim=(0, timber_ss.fs / 2), hide_poles=False)
    mplcursors.cursor()
    plt.savefig(OUTPUT_PATH / "14_plscf_plot_stab.png", dpi=300, bbox_inches="tight")
    plt.savefig(OUTPUT_PATH / "14_plscf_plot_stab.pdf", bbox_inches="tight")

    # Modal parameter estimation
    orders_plscf = [18, 18, 18]
    timber_ss.mpe("polymax", sel_freq=freq, order=orders_plscf)

    # Save results
    plscf_res = dict(plscf.result)
    np.savetxt(OUTPUT_PATH / "15_plscf_res_Fn.txt", plscf_res["Fn"], fmt="%.6f")
    np.savetxt(OUTPUT_PATH / "15_plscf_res_Xi.txt", plscf_res["Xi"], fmt="%.6f")
    np.savetxt(OUTPUT_PATH / "15_plscf_res_Phi.txt", plscf_res["Phi"], fmt="%.6f")

    # ================================================================================
    # Section 10: Results Comparison and Validation
    # ================================================================================

    # Generate MAC matrix comparing SSI and FSDD results
    figure, axes = plot_mac_matrix(ssi_res["Phi"].real, fsdd_res["Phi"].real)
    axes.set_xlabel("SSIcov")
    axes.set_ylabel("EFDD")
    plt.savefig(OUTPUT_PATH / "18_plot_mac_matrix.png", dpi=300, bbox_inches="tight")
    plt.savefig(OUTPUT_PATH / "18_plot_mac_matrix.pdf", bbox_inches="tight")

    print("Analysis complete")
```

Listing A.1: Complete Modal Analysis Implementation using pyOMA2.

Bibliography

[1] Morteza Ghalishooyan and Ahmad Shooshtari. "Operational modal analysis techniques and their theoretical and practical aspects: A comprehensive review and introduction". In: *6th International Operational Modal Analysis Conference, IOMAC 2015* (Jan. 2015) (cit. on p. 1).

[2] Dag Pasquale Pasca, Diego Federico Margoni, Marco Martino Rosso, and Angelo Aloisio. "pyOMA2: An Open-Source Python Software for Operational Modal Analysis". In: *Proceedings of the 10th International Operational Modal Analysis Conference (IOMAC 2024)*. Ed. by Carlo Rainieri, Carmelo Gentile, and Manuel Aenlle López. Cham: Springer Nature Switzerland, 2024, pp. 423–434. ISBN: 978-3-031-61421-7 (cit. on pp. 1, 3, 7, 12).

[3] Hamed Hasani and Francesco Freddi. "Operational Modal Analysis on Bridges: A Comprehensive Review". In: *Infrastructures* 8.12 (2023). ISSN: 2412-3811. DOI: 10.3390/infrastructures8120172. URL: https://www.mdpi.com/2412-3811/8/12/172 (cit. on pp. 1, 2, 7).

[4] Fahad Bin Zahid, Zhi Chao Ong, and Shin Yee Khoo. "A review of operational modal analysis techniques for in-service modal identification". In: *Journal of the Brazilian Society of Mechanical Sciences and Engineering* 42.8 (July 2020), p. 398. ISSN: 1806-3691. DOI: 10.1007/s40430-020-02470-8. URL: https://doi.org/10.1007/s40430-020-02470-8 (cit. on pp. 1, 2, 7).

[5] Fahad Bin Zahid, Zhi Chao Ong, and Shin Yee Khoo. "A review of operational modal analysis techniques for in-service modal identification". In: *Journal of the Brazilian Society of Mechanical Sciences and Engineering* 42.8 (July 2020), p. 398. ISSN: 1806-3691. DOI: 10.1007/s40430-020-02470-8. URL: https://doi.org/10.1007/s40430-020-02470-8 (cit. on p. 2).

[6] Marco Martino Rosso, Angelo Aloisio, Jafarali Parol, Giuseppe Carlo Marano, and Giuseppe Quaranta. "Intelligent automatic operational modal analysis". In: *Mechanical Systems and Signal Processing* 201 (2023), p. 110669. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2023.110669. URL: https://www.sciencedirect.com/science/article/pii/S0888327023005770 (cit. on p. 7).

[7] Yan-Long Xie, Siu-Kui Au, and Binbin Li. "Asymptotic identification uncertainty of well-separated modes in operational modal analysis with multiple setups". In: *Mechanical Systems and Signal Processing* 152 (2021), p. 107382. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2020.107382. URL: https://www.sciencedirect.com/science/article/pii/S0888327020307688 (cit. on p. 7).

[8] A Kareem and K Gurley. "Damping in structures: its evaluation and treatment of uncertainty". In: *Journal of wind engineering and industrial aerodynamics* 59.2-3 (1996), pp. 131–157 (cit. on p. 8).

[9] M. Döhler, P. Andersen, and L. Mevel. "Data Merging for Multi-Setup Operational Modal Analysis with Data-Driven SSI". In: *Structural Dynamics, Volume 3*. Ed. by Tom Proulx. New York, NY: Springer New York, 2011, pp. 443–452. ISBN: 978-1-4419-9834-7 (cit. on p. 10).

[10] Edwin Reynders, Filipe Aes, Guido De Roeck, and Alvaro Cunha. "Merging Strategies for Multi-Setup Operational Modal Analysis: Application to the Luiz I steel Arch Bridge". In: *Conference Proceedings of the Society for Experimental Mechanics Series* (Jan. 2009) (cit. on p. 10).

[11] C. Dierbach. "Python as a first programming language". In: (2014) (cit. on p. 16).

[12] *PEP 484 – Type Hints — peps.python.org*. https://peps.python.org/pep-0484/. (Accessed on 10/10/2024) (cit. on p. 17).

[13] Samuel Colvin. *Pydantic*. Version 1.8.2. Available at: https://github.com/pydantic/pydantic. 2021. DOI: 10.5281/zenodo.3351098. URL: https://github.com/pydantic/pydantic (cit. on pp. 17, 32).

[14] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. "Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects". In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 1741–1765. DOI: 10.1109/TSE.2022.3191353 (cit. on p. 18).

[15] Itamar Turner-Trauring. *Pip vs Conda: an in-depth comparison of Python's two packaging systems — pythonspeed.com*. https://pythonspeed.com/articles/conda-vs-pip/. Available at: https://pythonspeed.com/articles/conda-vs-pip/ (cit. on p. 19).

[16] *PEP 518 Specifying Minimum Build System Requirements for Python Projects — peps.python.org*. https://peps.python.org/pep-0518/. (Accessed on 08/22/2024) (cit. on p. 20).

[17] *PEP 582 Python local packages directory — peps.python.org*. https://peps.python.org/pep-0582/. (Accessed on 08/22/2024) (cit. on p. 20).

[18] *Introduction - PDM*. https://pdm-project.org/latest/. (Accessed on 08/22/2024) (cit. on p. 23).

[19] *Template Method - Python Design Patterns - GeeksforGeeks*. https://www.geeksforgeeks.org/template-method-python-design-patterns/. (Accessed on 09/02/2024) (cit. on p. 30).

[20] *Strategy Method - Python Design Patterns - GeeksforGeeks*. https://www.geeksforgeeks.org/strategy-method-python-design-patterns/. (Accessed on 09/02/2024) (cit. on p. 31).

[21] *Factory Method - Python Design Patterns - GeeksforGeeks*. https://www.geeksforgeeks.org/factory-method-python-design-patterns/. (Accessed on 09/03/2024) (cit. on p. 31).

[22] *Inheritance and Composition in Python - GeeksforGeeks*. https://www.geeksforgeeks.org/inheritance-and-composition-in-python/. (Accessed on 09/02/2024) (cit. on p. 31).

[23] *Type Hints in Python - GeeksforGeeks*. https://www.geeksforgeeks.org/type-hints-in-python/. (Accessed on 09/03/2024) (cit. on p. 32).

[24] *Python generics - GeeksforGeeks*. https://www.geeksforgeeks.org/python-generics/. (Accessed on 09/03/2024) (cit. on p. 32).

[25] *typing — Support for type hints — Python 3.13.0 documentation*. https://docs.python.org/3/library/typing.html. (Accessed on 10/13/2024) (cit. on p. 32).

[26] *Facade Method Design Pattern in Python - GeeksforGeeks*. https://www.geeksforgeeks.org/facade-method-python-design-patterns/. (Accessed on 09/03/2024) (cit. on p. 33).

[27] *SOLID Principles in Programming: Understand With Real Life Examples - GeeksforGeeks*. https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/. (Accessed on 09/03/2024) (cit. on p. 34).

[28] *Class Diagram — Unified Modeling Language (UML) - GeeksforGeeks*. https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/. (Accessed on 09/25/2024) (cit. on p. 35).

[29] *Sequence Diagrams — Unified Modeling Language (UML) - GeeksforGeeks*. https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/. (Accessed on 09/25/2024) (cit. on p. 41).

[30] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014 (cit. on p. 53).

[31] *Sphinx — Sphinx documentation*. https://www.sphinx-doc.org/en/master/. (Accessed on 09/28/2024) (cit. on p. 55).

[32] *Full featured documentation deployment platform - Read the Docs*. https://about.readthedocs.com/?ref=readthedocs.org. (Accessed on 09/28/2024) (cit. on p. 55).

[33] *pytest documentation*. https://docs.pytest.org/en/stable/index.html. (Accessed on 09/28/2024) (cit. on p. 56).

[34] *tox*. https://tox.wiki/en/4.20.0/. (Accessed on 09/28/2024) (cit. on p. 56).

[35] *pytest-cov 5.0.0 documentation*. https://pytest-cov.readthedocs.io/en/latest/. (Accessed on 09/29/2024) (cit. on p. 56).

[36] Andrew M. Saint-Laurent. "Understanding open source and free software licensing - guide to navigation licensing issues in existing and new software". In: 2004. URL: https://api.semanticscholar.org/CorpusID:106668315 (cit. on pp. 59, 61).

[37] Shuhui Wu. "The Impact of Open Source Software". In: 2003. URL: https://api.semanticscholar.org/CorpusID:16582897 (cit. on p. 63).

[38] *The Cathedral and the Bazaar - Wikipedia*. https://en.wikipedia.org/wiki/The_Cathedral_and_the_Bazaar. (Accessed on 10/04/2024) (cit. on p. 63).

[39] *Eric Steven Raymond - Wikipedia*. https://it.wikipedia.org/wiki/Eric_Steven_Raymond. (Accessed on 10/04/2024) (cit. on p. 63).

[40] Dag Pasquale Pasca, Angelo Aloisio, Massimo Fragiacomo, and Roberto Tomasi. "Dynamic characterization of timber floor subassemblies: Sensitivity analysis and modeling issues". In: (2021) (cit. on pp. 65, 66, 70–73, 75, 76).

[41] Carlo Rainieri and Giovanni Fabbrocino. *Operational modal analysis of civil engineering structures*. Vol. 142. Springer, 2014, p. 143 (cit. on pp. 66, 69, 72).

[42] Angelo Aloisio, Dag Pasquale Pasca, Luca Di Battista, Marco Martino Rosso, Raffaele Cucuzza, Giuseppe Carlo Marano, and Rocco Alaggio. "Indirect assessment of concrete resistance from FE model updating and Young's modulus estimation of a multi-span PSC viaduct: Experimental tests and validation". In: *Structures* 37 (2022), pp. 686–697. ISSN: 2352-0124. DOI: https://doi.org/10.1016/j.istruc.2022.01.045. URL: https://www.sciencedirect.com/science/article/pii/S2352012422000455 (cit. on pp. 77, 79, 83).

[43] Marco Martino Rosso, Raffaele Cucuzza, Giuseppe Carlo Marano, Angelo Aloisio, and Dag Pasquale Pasca. "Indirect estimate of concrete compression strength framework with FE model updating and operational modal analysis". In: 118 (2022), pp. 1611–1618. ISSN: 2221-3783. DOI: 10.2749/prague.2022.1611. URL: http://dx.doi.org/10.2749/prague.2022.1611 (cit. on pp. 77, 79, 83).

[44] *IOMAC 2024 – International Operational Modal Analysis Conference*. https://iomac2024.com/. (Accessed on 10/06/2024) (cit. on p. 85).