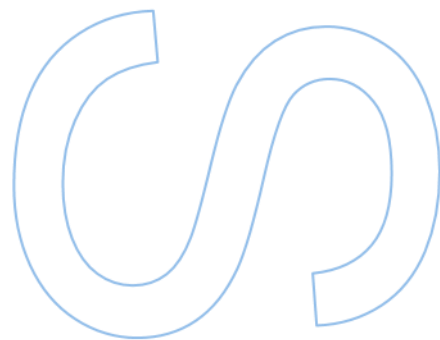
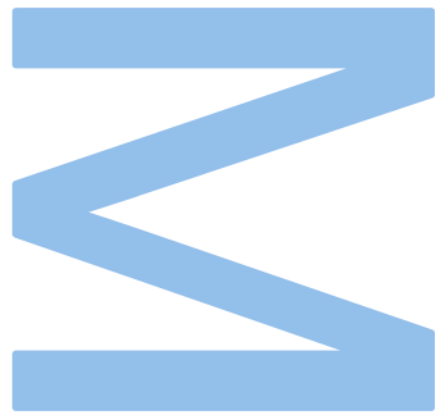


A Software Package Implementing Efficiently Time Series Mappings

Ana Filipa Pereira Torres
Master's degree in Computer Science
Computer Science Department
Faculty of Sciences of the University of Porto
2024



A Software Package Implementing Efficiently Time Series Mappings

Ana Filipa Pereira Torres

Dissertation carried out as part of the Master's degree in
Computer Science
Computer Science Department
2024

Supervisor

Vanessa Alexandra Freitas da Silva, Assistant Professor, Faculty of
Sciences, University of Porto

Co-supervisor

Fernando Manuel Augusto da Silva, Full Professor, Faculty of
Sciences, University of Porto

Acknowledgements

First, I would like to express my gratitude to my two supervisors, Vanessa Silva and Fernando Silva. Their support and guidance have been invaluable throughout this journey. Professor Vanessa, I am especially grateful for your constant availability and for helping me navigate both the theoretical aspects and practical challenges of this subject, always with great patience and clarity. Professor Fernando, your insights and encouragement have been essential in pushing me to achieve my best.

I would also like to acknowledge Professor Pedro Ribeiro and Professor Maria Eduarda Silva for contributing to deepening my understanding in the subjects of Complex Networks and Time Series Analysis. Professor Pedro encouraged me and my colleague Brenda to write a paper on our Network Science project, which greatly influenced my research direction. Professor Maria Eduarda provided valuable materials and feedback, which were extremely helpful in refining this thesis.

I would also like to thank my university colleagues and friends for sharing the ups and downs of this journey and for supporting me through any challenges. I want to especially highlight Brenda for inviting me to join her project, which led to a deeper exploration of time series mappings. Your encouragement, friendship, and academic collaboration have all been vital to this work.

Finally, I would like to express my deepest appreciation to my family, especially to my parents and brother, who have been my rock during the toughest times, offering unwavering encouragement, support and much needed study breaks — thank you for loving me, at all times.

Resumo

A análise de séries temporais foca-se em entender as dependências entre observações recolhidas sequencialmente ao longo do tempo. Com os recentes avanços tecnológicos, o volume de dados indexados temporalmente aumentou, tornando a sua análise ainda mais importante em áreas como finanças, saúde e engenharia. Embora as abordagens estatísticas tradicionais para a análise de séries temporais estejam bem estabelecidas, estas frequentemente enfrentam dificuldades ao lidar com a complexidade dos dados modernos de grande escala. Uma alternativa promissora envolve mapear dados de séries temporais em redes complexas, de modo a poder-se beneficiar da ciência de redes complexas e da teoria dos grafos para chegar a conhecimentos mais profundos. Apesar do potencial desta abordagem, as ferramentas e pacotes existentes para a análise de séries temporais via redes complexas carecem de uma integração abrangente. Este trabalho aborda essa lacuna ao desenvolver o pacote `pyTSMnet`, uma ferramenta em Python projetada para integrar métodos de mapeamento de séries temporais de maneira organizada, unificada e eficiente.

A nossa abordagem envolveu a criação de algoritmos do zero para vários métodos de mapeamento, principalmente para dados multivariados, de acordo com os fundamentos matemáticos descritos na literatura, e a integração de implementações de pacotes existentes para métodos mais populares. Focamos em otimizar o desempenho, utilizando estruturas de dados e operações eficientes e incluímos funções de visualização e extração de medidas topológicas de redes. Para demonstrar as capacidades do `pyTSMnet`, também conduzimos uma análise de dados gerados referentes a séries temporais univariadas e multivariadas, tendo-nos focado na extração e comparação de características de redes, para vários métodos de mapeamento. Também fornecemos exemplos de uso que ilustram a amplitude da funcionalidade do pacote.

As principais contribuições deste trabalho incluem a implementação bem-sucedida de uma ampla gama de métodos de mapeamento, com maior flexibilidade de parâmetros e a criação de uma ferramenta unificada que simplifica a análise de séries temporais univariadas e multivariadas. Uma documentação ampla e exemplos foram também desenvolvido para tornar o `pyTSMnet` acessível a usuários de diferentes níveis de habilidade. Este pacote estabelece as bases para avanços algorítmicos e inovações em métodos de mapeamento existentes, e tem potencial para expansão e otimização adicional de certos métodos.

Palavras-chave: Séries Temporais Univariadas, Séries Temporais Multivariadas, Redes Complexas, Métodos de Mapeamento, Pacote Python

Abstract

Time series analysis focuses on understanding the dependencies between data points collected sequentially over time. With recent technological advancements, the volume of time-indexed data has increased, making its analysis even more important in fields such as finance, healthcare, and engineering. Although traditional statistical approaches to time series analysis are well-established, they often struggle to handle the complexity of modern large-scale data. A promising alternative involves mapping time series data into complex networks, in order to leverage network science and graph theory to reach deeper insights. Despite the potential of this approach, existing tools and packages for time series analysis via complex networks lack comprehensive integration. This work addresses this gap by developing the **pyTSMnet** package, a Python tool designed to integrate time series mapping methods in an organised, unified, and efficient manner.

Our approach involved creating algorithms from scratch for several mapping methods, mainly for multivariate data, according to the mathematical foundations described in the literature, and integrating existing package implementations for more popular methods. We focused on optimising performance, by using efficient data structures and operations, and implementing visualisation functions and feature extraction capabilities for network topological measurements. To demonstrate the capabilities of **pyTSMnet**, we also conducted an analysis on both generated univariate and multivariate time series data, focusing on extracting and comparing network features for various mappings methods. We also provided usage examples that illustrate the scope of the package functionality.

The main contributions of this work include the successful implementation of a wide range of mapping methods with improved parameter flexibility, and the creation of a unified tool that simplifies the analysis of both univariate and multivariate time series data. A comprehensive documentation and examples have also been developed to make **pyTSMnet** accessible to users from different skill levels. This package lays the foundation for algorithmic advancements and innovation for existing mapping methods and has potential for further expansion and optimisation of certain methods.

Keywords: Univariate Time Series, Multivariate Time Series, Complex Networks, Mapping Methods, Python Package

Contents

Acknowledgements	i
Resumo	ii
Abstract	iii
Contents	vi
List of Tables	vii
List of Figures	ix
List of Listings	x
List of Abbreviations	xi
1 Introduction	1
1.1 Motivation	3
1.2 Goal and Contributions	4
1.3 Organisation	5
2 Concepts and Related Work	7
2.1 Time Series Analysis	7
2.2 Complex Networks	11
2.3 Time Series to Complex Networks	13

3	Univariate Mapping Methods	15
3.1	Visibility Networks	15
3.1.1	Natural Visibility Graph	15
3.1.2	Horizontal Visibility Graph	21
3.1.3	Difference Visibility Graph	26
3.1.4	Absolute Invisibility Graph	27
3.1.5	Circular Visibility Graph	27
3.2	Transition Networks	30
3.2.1	Quantile Graph	31
3.2.2	Ordinal Partition Transition Network	32
3.2.3	Coarse-Grained Phase Space Graph	33
3.2.4	Visibility Graphlets Network	35
3.3	Proximity Networks	36
3.3.1	Cycle Network	37
3.3.2	Correlation Network	38
3.3.3	Recurrence Network	38
4	Multivariate Mapping Methods	42
4.1	Single Layer Networks	42
4.1.1	Correlation Network	43
4.2	Multiple Layers Networks	43
4.2.1	Multiplex Visibility Network	44
4.2.2	Multiplex Recurrence Network	44
4.2.3	Multilayer Horizontal Visibility Graph	45
4.2.4	Multilayer Quantile Graph	47
5	Software Development	49
5.1	Programming Language	49
5.2	Package Structure	49

5.3	Modules and Classes	51
5.3.1	core module	52
5.3.2	univariate/visibility module	54
5.3.3	univariate/transition module	62
5.3.4	univariate/proximity module	67
5.3.5	multivariate/single_layer module	73
5.3.6	multivariate/multi_layer module	74
5.3.7	Summary	76
5.4	Supporting Functions and Utilities	78
5.4.1	Input and Output Operations	78
5.4.2	Feature Extraction Functions	78
5.4.3	Visualisation Tools	80
6	Applications and Analysis	81
6.1	Usage	81
6.2	Univariate Time Series Data	83
6.2.1	Feature Extraction	84
6.2.2	Visualisation	87
6.2.3	Discussion	89
6.3	Multivariate Time Series Data	90
6.3.1	Feature Extraction	91
6.3.2	Visualisation	92
6.3.3	Discussion	92
7	Conclusions	93
7.1	Main Contributions	94
7.2	Future Work	95
	Bibliography	97

List of Tables

2.1	Summary of existing packages of time series mapping methods	14
5.1	Mapping classes and associated time complexities	77
6.1	Summary of univariate time series models	83
6.2	Network features for time series from univariate mapping methods	86
6.3	Summary of bivariate time series models	90
6.4	Network features for time series from multivariate mapping methods	91

List of Figures

1.1	Time series mapping methods	2
2.1	Comparison between directed/undirected and weighted/unweighted graphs . . .	11
2.2	Comparison between general multilayer and multiplex networks	13
2.3	General overview of time series mapping methods	13
3.1	Illustration of the Natural Visibility Graph mapping method	16
3.2	Illustration of the Limited Penetrable Visibility Graph mapping method	18
3.3	Illustration of the Weighted Dual Perspective Visibility Graph mapping method .	20
3.4	Illustration of the Horizontal Visibility Graph mapping method	22
3.5	Illustration of the Directed Horizontal Visibility Graph mapping method	24
3.6	Illustration of the Limited Penetrable Horizontal Visibility Graph mapping method	24
3.7	Illustration of the Weighted Dual Perspective Horizontal Visibility Graph mapping method	25
3.8	Illustration of the Difference Visibility Graph mapping method	26
3.9	Illustration of the Absolute Invisibility Graph mapping method	27
3.10	Illustration of the Circular Visibility Graph mapping method	28
3.11	Illustration of the circle system for two arbitrary data points	28
3.12	Illustration of the Circular Limited Penetrable Visibility Graph mapping method	30
3.13	Illustration of the Quantile Graph mapping method	32
3.14	Illustration of the Ordinal Partition Transition Network mapping method	33
3.15	Illustration of Coarse-Grained Phase Space Graph mapping method	34

3.16	Illustration of Visibility Graphlets Network mapping method	36
3.17	Illustration of Cycle Network mapping method	37
3.18	Illustration of Correlation Network mapping method	39
3.19	Illustration of Recurrence Network mapping method	41
4.1	Illustration of Correlation Network mapping method	43
4.2	Illustration of Multiplex Visibility Network mapping method	44
4.3	Illustration of Multiplex Recurrence Network mapping method	45
4.4	Illustration of the Multilayer Horizontal Visibility Graph mapping method	46
4.5	Illustration of the Multilayer Quantile Graph mapping method	48
5.1	Overview of pyTSMnet package structure	50
5.2	Overview of the package's inheritance process	51
5.3	Overview of the core module	53
6.1	Network degree distributions from different visibility graphs	85
6.2	Network degree distributions from different transition graphs	85
6.3	Network degree distributions from different proximity graphs	87
6.4	Networks visualisation obtained from different visibility graphs	87
6.5	Networks visualisation obtained from different transition graphs	88
6.6	Networks visualisation obtained from different proximity graphs	88
6.7	Network degree distributions from different multilayer networks	91
6.8	Networks visualisation obtained by multivariate mappings	92

List of Listings

6.1	Creation of Horizontal Visibility Graph	81
6.2	Extraction of degree distribution from the Horizontal Visibility Graph	82
6.3	Creation of Multiplex Recurrence Network	82

List of Abbreviations

AIG	Absolute Invisibility Graph	LPVG	Limited Penetrable Visibility Graph
ANNN	Adaptive Nearest Neighbour Network	MHVG	Multilayer Horizontal Visibility Graph
AR	AutoRegression	MQG	Multilayer Quantile Graph
ARCH	AutoRegressive Conditional Heteroskedasticity	NNN	Nearest Neighbour Network
ARIMA	AutoRegressive Integrated Moving Average	NVG	Natural Visibility Graph
ARMA	AutoRegressive Moving Average	OPTN	Ordinal Partition Transition Network
CLPVG	Circular Limited Penetrable Visibility Graph	PNVG	Parametric Natural Visibility Graph
CVG	Circular Visibility Graph	PyPI	Python Package Index
cVWN	Correlated Vector White Noise	QG	Quantile Graph
DC	Divide-and-Conquer	sGARCH	Strong Vector Generalized AutoRegressive Conditional Heteroskedastic
DHVG	Directed Horizontal Visibility Graph	sVAR	Strong Vector Autoregression
DVG	Difference Visibility Graph	VAR	Vector Autoregression
GARCH	Generalized AutoRegressive Conditional Heteroskedastic	VG	Visibility Graph
HVG	Horizontal Visibility Graph	VGARCH	Vector Generalized AutoRegressive Conditional Heteroskedastic
INAR	Integer-Valued AutoRegressive	VWN	Vector White Noise
IoT	Internet-of-Things	WDPHVG	Weighted Dual Perspective Horizontal Visibility Graph
iVWN	Independent Vector White Noise	WDPVG	Weighted Dual Perspective Visibility Graph
LPHVG	Limited Penetrable Horizontal Visibility Graph		

wGARCH	Weak Vector Generalized AutoRegressive Conditional Heteroskedastic	WN	White Noise
		wVAR	Weak Vector Autoregression
WHVG	Weighted Horizontal Visibility Graph	WVG	Weighted Visibility Graph

Chapter 1

Introduction

Time series data consists of sequences of data points collected or recorded at successive points in time, often at uniform intervals. Unlike static data, time series data is inherently ordered, meaning that the temporal sequence of the data points is essential for its proper analysis [43]. This chronological ordering reflects the evolution of the underlying process being studied, making time a critical component that must be preserved and considered. Traditionally, time series analysis involves a set of statistical and computational techniques designed to understand, model, and extract meaningful insights from time-dependent data. Generally, the ultimate goal of time series analysis is to forecast future values based on historical data, enabling predictive insights that can inform decision-making in various fields such as finance, economics, meteorology, health, and engineering [18].

While time series forecasting is a primary goal, time series mining encompasses a variety of tasks aimed at extracting knowledge from temporal data. These tasks include indexing, clustering, classification, segmentation, motif discovery, anomaly detection, among others [21, 69]. Despite the importance and variety of traditional time series analysis methods, mostly involving statistical linear techniques, several challenges persist. These include handling high-dimensional data, managing high-frequency data, analysing non-stationary data or dealing with time series data with other non-trivial characteristics [44].

Another approach for the analysis of time series is called nonlinear time-series analysis. It relies on the concepts of phase-space reconstruction and dynamical systems theory, providing robust framework to deal with data modelled by deterministic dynamics within a multidimensional state space [9]. Moreover, time series analysis often involves several pre-processing steps, such as normalisation, detrending, and seasonal adjustment, to prepare the data for modelling. Also, the time series models tend to be highly parametric.

Time series analysis via complex networks offers a more recent and promising approach to address time series analysis challenges. By transforming time series data into complex networks, this approach allows leveraging network science and graph theory to uncover hidden structures and relationships within the data [44, 69]. This transformation is achieved through time series mapping functions that convert temporal data into network representations. Once mapped, advanced network analysis techniques can be employed to extract meaningful insights from the original temporal data, providing a novel perspective on time series analysis [69].

Network science concerns the study of networks—non-trivial structures that aim to model complex systems where elements interact with other elements. In a network (or graph), these elements are mathematically represented as nodes and their interactions are represented as edges that link pair of nodes. Complex systems are present everywhere. Examples include the World Wide Web, the nervous system, social groups, transportation systems, and ecological networks. By representing these systems as graphs, we can analyse the connectivity, dynamics, and structural properties of these systems, thereby gaining insights that are not easily accessible through traditional methods [7].

In the literature, there are several different functions for mapping time series data into one or more complex networks. These functions can be broadly categorised into three concepts: visibility, transition, and proximity. Depending on the mapping method, individual time points, sets of points, or even complete time series are mapped into graph nodes, and edges are established based on the specific mapping concept. In the case of multivariate data, each individual time series can be mapped to its own graph, transforming the original data into a multiple-layer network structure [44]. Figure 1.1 illustrates the complex networks resulting from existing mapping methods in the literature for two examples of time series (univariate and multivariate).

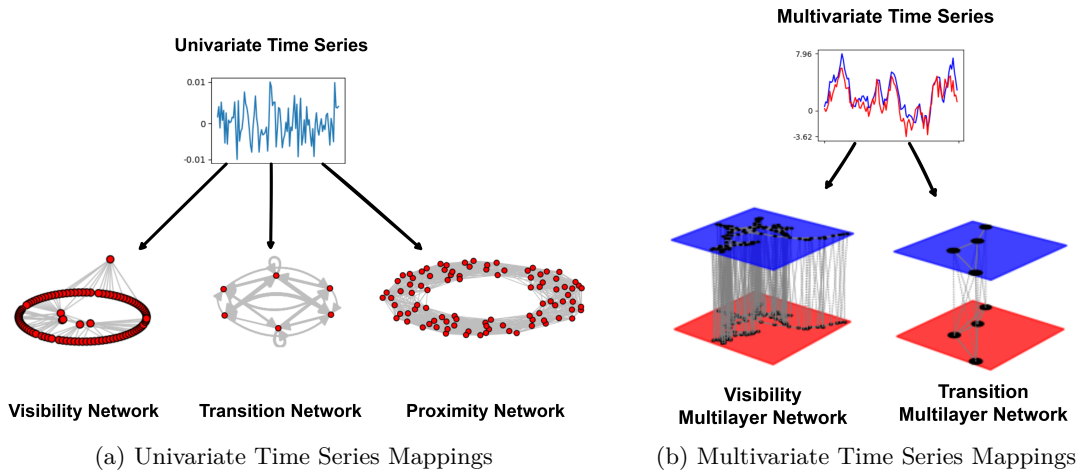


Figure 1.1: Illustrative example of the representations of time series mapping methods. On the left, three univariate time series mapping methods. On the right, two multivariate time series mapping methods.

1.1 Motivation

In recent years, there has been an exponential growth in the volume of data generated and collected. The proliferation of the Internet, Internet-of-Things (IoT) devices, and digital technologies has significantly contributed to this explosion. This vast amount of data presents both opportunities and challenges. On one hand, it offers unprecedented possibilities for insights and decision-making. On the other hand, it evidences the need to develop efficient and scalable analytical tools [68].

Time series, in particular, has grown in complexity and volume, making time series analysis pivotal in fields such as social sciences, biology, finance, and physics. However, traditional analysis methods are often insufficient to handle the scale and intricacies of modern time series data.

To address these challenges, the field of time series analysis via complex networks has emerged. The concepts behind this approach play a crucial role in understanding temporal data by leveraging network science theory. The first step, as introduced above, involves mapping the time series into networks. Several algorithmic approaches have been proposed for this purpose, applicable to both univariate and multivariate time series data.

These mapping methods, which can be categorised into concepts of visibility, transition, or proximity, produce different types of networks—both in terms of topology and associated dynamics—that highlight different properties of the original data, depending on the underlying mapping concept. It is important to understand the advantages and disadvantages of each method, in order to achieve more interesting and useful results in our analysis, and choose the method according to the scale of the data, as some methods perform dimensionality reduction, while others have higher computational complexity [44].

Although numerous time series mapping methods have been proposed in the literature, practical implementations remain limited. Existing R and Python software packages provide certain methods, such as Visibility Graphs and Recurrence Networks, but there is a significant gap in the availability of all established methods, their variations, and even newer algorithms described in the literature. Furthermore, the available methods are scattered across different packages and programming languages, leading to a lack of comprehensive integration. In other words, although some existing packages implement some of the most used existing mapping methods, there is no unified tool that covers all known methods.

To address this gap, we propose the development of a package that provides an efficient, comprehensive, and accessible tool for mapping time series into complex networks. This centralised solution aims to serve as a single, extensible code base for the community, reducing the time needed to experiment with different time series mapping techniques. As a result, we created a Python package, which we named `pyTSMnet`, with the name reflecting its focus on 'Time Series Mappings' (TSM) to networks (`net`) and its implementation in Python (`py`).

The **pyTSMnet** package aims to unify existing practical implementations and introduce new ones, enhancing the usability and effectiveness of time series analysis via complex networks. This tool is designed to assist researchers by providing practical and accessible implementations of these methods, aiding in tasks related to machine learning, data mining, and time series analysis. Additionally, it makes analyses based on complex networks more replicable and allows for easier comparison between different methods.

1.2 Goal and Contributions

The main research objective of this dissertation is to unify and expand existing mapping methods to transform time series data into complex networks into a single tool, maximising its accessibility and usability for the scientific community.

Concisely, our main goals regarding the development of the **pyTSMnet** package were:

- Implement each mapping algorithm existent in the literature, and missing from the related work, with precision, and as efficiently as possible, that is, maintaining the correctness of the mathematical foundations mentioned in the literature, while choosing the most adequate and space saving data structures and the most time efficient approaches for the algorithm;
- Unify the existent univariate mapping methods from the other packages and the new implemented algorithms into a cohesive software package, in an intuitive and easy to explore way;
- Implement recent multivariate time series mapping algorithms in multilayer networks, using appropriate data structures.
- Provide documentation and examples, to facilitate the user experience;
- Implement auxiliary functions to visualise mapping methods and resulting networks and to extract topological network features.

Ultimately, the main contributions of this work are the following:

- We designed and implemented the algorithms from scratch, based on the mathematical foundations described in the literature, for methods like the Circular Visibility Graph, Absolute Invisibility Graph, Visibility Graphlets Network, and others.
- We unified the existent packages for time series mappings, by using the **ts2vg**, **PyI0mica** and **pyunicorn** implementations for the Natural and Horizontal Visibility Graphs, Dual Perspective Visibility Graphs and Recurrence Networks, respectively.

- We used practical implementations and pseudo-code mentioned in the literature to write our algorithms for the Dual Tree method for Horizontal Visibility Graphs, and for the Multilayer Horizontal Visibility and Multilayer Quantile graphs.
- We extended some of the mapping concepts, allowing for extra parameters and flexibility.
- We developed visualisation functions for better analysis and interpretation, using the `igraph` (for univariate setting) and `pymnet` (for multivariate setting) packages.
- We created tools that allow conversion to formats compatible with popular graph libraries, such as `NetworkX`, `igraph` and `pymnet`.
- We provided full documentation for every class and use case examples for time series mappings, feature extraction and visualisation tasks.

The `pyTSMnet` package will be available as open-source on GitHub and on Python Package Index (PyPI), via `pip` installation, ensuring accessibility and ease of use for the research community.

1.3 Organisation

This section aims to outline the structure of this dissertation. The seven chapters can be briefly presented by the following:

Chapter 1: Introduction. Introduces time series analysis via complex networks and explains the motivation for developing our software package. It also outlines the main goals and contributions of the research and provides an organisation of the thesis to guide the reader.

Chapter 2: Concepts and Related Work. Provides the fundamental concepts of time series and complex networks, which are essential for understanding the remainder of the thesis. It provides a general overview of existing approaches for time series mappings. Additionally, it summarises existing Python packages that provide mapping methods.

Chapter 3: Univariate Mapping Methods. Reviews the literature on mapping methods tailored to univariate time series. It illustrates the methods with figures, using the same time series example for all methods for better comparison. Algorithmic descriptions are provided where applicable. The chapter is organised into three main mapping concepts: visibility, transition, and proximity.

Chapter 4: Multivariate Mapping Methods. Similarly to Chapter 3, this chapter presents mapping methods, but adapted to multivariate time series. It is organised into single-layer and multiple-layer networks based on the desired graph structure.

Chapter 5: Software Development. Details the development process of the software package. It covers the design principles, package structure, and implementation algorithms, ensuring the software meets the objectives of efficiency, accuracy, and user-friendliness. The chapter also includes a discussion of the tools and technologies used, as well as the challenges encountered during development.

Chapter 6: Analysis and Applications. Demonstrates the practical applications of the developed software. It includes examples that showcase the software capabilities in real-world scenarios, for visualisation purposes and feature extraction tasks, as well as a brief analysis of the results.

Chapter 7: Conclusion. Summarises the key findings and contributions of the thesis, reflecting on the research goals and discussing how they were achieved. It also outlines potential directions for future work.

Chapter 2

Concepts and Related Work

In this chapter, we present what we deem to be the theoretical background necessary to understand the methods and algorithms described in the following Chapters 3, 4 and 5, as well as the analysis conducted in Chapter 6. Essentially, we define and characterise time series, as well as the time series models used in the scope of this work, and we explore some of the fundamentals of complex networks theory, including the topological metrics of interest to this work. Additionally, we also provide a general overview of the mapping concepts existing in the literature and explore some of the available functionalities and methods in published Python or R software packages.

2.1 Time Series Analysis

A time series $\mathbf{Y} = \{(t_1, y_1), \dots, (t_n, y_n)\}$ is a finite sequence of observed data points, typically collected at regularly spaced intervals. It represents a realisation of a stochastic process, which is defined as a set of random values indexed by time, t [18]. This is referred to as *univariate time series* when the observations correspond to a single variable or process.

An essential characteristic of time series data is the temporal dependency between adjacent values. Unlike other data types, where individual data can be considered independently, time series data observations are often correlated with each other. In addition to the characteristic of serial dependence (over time), time series can exhibit other characteristics such as trends, seasonal patterns, and cycles. Therefore, often, the data are non-stationary—meaning that statistical properties like the mean, variance, and autocorrelation change over time—which can pose challenges to traditional data analysis techniques [8]. Furthermore, in non-linear contexts, concepts like non-linearity structure and chaos may also be present [9]. Therefore, the task of time series analysis involves using statistical techniques, such as stochastic and dynamic models, adapted to provide plausible descriptions of the characteristics of the time series data [43].

There are many time series statistical models available in the literature designed to model different properties of the time series data. These models can be broadly categorised into two types: linear and non-linear. Linear models assume that the conditional mean is a linear function of previous values of the time series, while non-linear models are used to model data with characteristics that cannot be adequately represented by linear models, such as volatility, asymmetry, different regimes, and clustering effects [18, 44].

We will now proceed to describe some of the univariate time series statistical models used to generate the simulated time series data for our analysis in Chapter 6, according to [43, 45].

- **White Noise (WN).** Corresponds to a sequence of random independent and identically distributed (i.i.d.) variables, with mean 0 and constant variance σ_ϵ^2 . It is the simplest process, acting as a base to the other time series models. A particular case, is the Gaussian White Noise, where the variables follow the normal distribution, $\epsilon_t \sim \mathcal{N}(0, 1)$.

Linear Models.

- **AutoRegression (AR).** Corresponds to an auto-regressive process of order p , where each value is a linear combination of the p most recent past values. This model is denoted as $AR(p)$, and it is represented by the following linear equation:

$$y_t = \sum_{i=1}^p \phi_i y_{t-i} + \epsilon_t, \quad (2.1)$$

where ϕ_i are the autoregressive constants with $i = 1, \dots, p$, and ϵ_t is a white noise process.

- **AutoRegressive Integrated Moving Average (ARIMA).** This model is a generalisation of the linear AutoRegressive Moving Average (ARMA) model for modelling non-stationary time series. ARIMA combines the autoregressive (AR) model presented before, with a moving average (MA) model that specifies a linear dependence between a value and past prediction errors, and an integrated (I) component which refers to the differentiation of observations to make the time series stationary (using the difference operator $\nabla^d y_t = (1 - B)^d y_t$, $d \in \mathbb{N}$). $ARIMA(p, d, q)$ model can be satisfied by the following equation:

$$\left(1 - \sum_{i=1}^p \phi_i B^i\right) (1 - B)^d y_t = \left(1 + \sum_{i=1}^q \theta_i B^i\right) \epsilon_t, \quad (2.2)$$

where ϕ_i are the autoregressive constants with $i = 1, \dots, p$, θ_i are the moving average constants with $i = 1, \dots, q$, parameter d is a constant that represents the amount of differences, and ϵ_t is a white noise process.

Non-linear Models. For the non-linear models presented, we consider ϵ_t as uncorrelated random variables, z_t as white noise with variance 1, and σ_t as the standard deviation of ϵ_t , such that $\epsilon_t = \sigma_t z_t$.

- **Generalized AutoRegressive Conditional Heteroskedastic (GARCH).** This process is based on the AutoRegressive Conditional Heteroskedasticity (ARCH) model, proposed to model processes where the conditional variance, or conditional volatility, is not constant, which consists of a quadratic function of lagged values. GARCH extends the ARCH concept, by not only considering the squares of past errors, but also the squares of their own past values. Thus, ϵ_t is a GARCH(p, q) process if it satisfies the following condition:

$$\sigma_t^2 = \omega + \sum_{i=1}^p \beta_i \sigma_{t-i}^2 + \sum_{i=1}^q \alpha_i \epsilon_{t-i}^2, \quad (2.3)$$

where $\omega > 0$, $\alpha_i, \beta_i \geq 0$, and $\sum_{i=1}^p \beta_i + \sum_{i=1}^q \alpha_i < 1$.

- **Integer-Valued AutoRegressive (INAR).** This process models integer-valued time series. It utilises a thinning (random) operation defined on the integers, being the binomial thinning the most common operation used. It is defined as follows: if X is a non-negative integer-valued random variable and $0 < \alpha < 1$, then $\alpha * X = \sum_{i=1}^X y_i$ where $\{y_i\}$ is a sequence of i.i.d. Bernoulli random variables, independent of X and $*$ defines the binomial thinning operation.

A time series y_t is said to follow an INAR(p) process if it satisfies:

$$y_t = \sum_{i=1}^p \alpha_i * y_{t-i} + \epsilon_t, \quad t = 0, \pm 1, \pm 2, \dots, \quad (2.4)$$

where $\alpha_i \in [0, 1]$ and ϵ_t is the integer-valued time series.

If at each time t , we have a vector of m observations, we have what is called a *multivariate time series* $\mathbf{Y} = \{((t_{1,1}, y_{1,1}), \dots, (t_{1,n}, y_{1,n})), \dots, ((t_{m,1}, y_{m,1}), \dots, (t_{m,n}, y_{m,n}))\}$. It can also be defined as a set of m univariate time series, here designated as the *multivariate time series components* [46].

In addition to the temporal dependency within each component $\mathbf{Y}_\alpha = \{y_{\alpha,t}\}_{t=1}^n, \alpha = 1, \dots, m$, in a multivariate time series there may also interdependence between different components, that is, between $y_{\alpha,t}$ and $y_{\beta,s}$, with $\alpha, \beta = 1, \dots, m$, with $\alpha \neq \beta$ and $t, s = 1, \dots, n$. This prompted the emergence of new concepts as an extension to the already known methods of the univariate time series theory, adjusted to not only work with the temporal dependence of the data but also with what we describe as cross-sectional dependence [54].

Unlike traditional statistical methods based on random samples that assume independence, multivariate time series analysis introduces complex tools, such as variance-covariance matrices, causality measures, and vector time series models [54]. These models, however, are often tailored to specific applications, such as for financial data, and come with challenges like high computational demands, the need to identify appropriate models, and data preprocessing to address non-stationarity, especially as the dimensionality of the data increases.

There are several linear and non-linear models in the literature also developed for multivariate time series analysis. Below we describe those used in this work to simulate multivariate temporal data, referencing [46, 54].

Linear Models.

- **Vector White Noise (VWN).** This process, ϵ_t , manages to be the simplest multivariate time series model and corresponds to a vector of sequences of i.i.d. random values with mean vector $\mathbf{0}$ and covariance matrix Σ (size $m \times m$), where Σ is symmetric and positive. The white noise components are serially uncorrelated, but can be contemporaneously correlated, that is, with cross-correlation at the same time t .
For processes that are not correlated, we call them Independent Vector White Noise (iVWN). On the other hand, if the white noise processes are contemporaneously correlated, we refer to it as a Correlated Vector White Noise (cVWN).

- **Vector Autoregression (VAR).** This process is an extension of its univariate counterpart, AR, following the same idea of the variables being linear dependent on past values. A VAR(p) process is generated according to:

$$\mathbf{y}_t = \boldsymbol{\varphi} + \sum_{i=1}^p \boldsymbol{\phi}_i \mathbf{y}_{t-i} + \boldsymbol{\epsilon}_t, \quad (2.5)$$

where $\boldsymbol{\epsilon}_t$ refers to the VWN, $\boldsymbol{\phi}_i$ is the matrix of autoregressive coefficients, and $\boldsymbol{\varphi}$ stands for the vector of intercept constants.

According to the coefficients used for the correlation, we could have two types of VAR models: Weak Vector Autoregression (wVAR) and Strong Vector Autoregression (sVAR).

Non-linear Models.

- **Vector Generalized AutoRegressive Conditional Heteroskedastic (VGARCH).** Also acting as a generalisation of an univariate method, the VGARCH extends the principle behind the GARCH model of considering volatility not constant to the multivariate setting. Thus, a VGARCH(p, q) process is generated based on the following condition:

$$\boldsymbol{\sigma}_t = \boldsymbol{\omega} + \sum_{i=1}^p \boldsymbol{\alpha}_i \boldsymbol{\epsilon}_{t-i}^2 + \sum_{i=1}^q \boldsymbol{\beta}_i \boldsymbol{\sigma}_{t-i}^2, \quad (2.6)$$

where $\boldsymbol{\sigma}_t$ refers to the volatility (or variance) in the variables $\mathbf{y}_t$, $\boldsymbol{\omega}$ is a vector of constant terms, $\boldsymbol{\alpha}_i$ are matrices that capture the effect of past innovations, and $\boldsymbol{\beta}_i$ are matrices that capture the effect of the past covariance matrices.

Again, depending on the vector parameters used for the model, we can have two different types of processes: Weak Vector Generalized AutoRegressive Conditional Heteroskedastic (wGARCH) and Strong Vector Generalized AutoRegressive Conditional Heteroskedastic (sGARCH).

2.2 Complex Networks

Network science aims to study the relationships between elements within a complex system and the collective behaviours these elements exhibit. It provides tools and methods for analysing the structure and dynamics of networks, offering insights into the underlying systems they represent.

Formally, a *graph*, $\mathbf{G}$, corresponds to the pair (V, E) , where V establishes the set of nodes (also called vertices), and $E \subseteq V \times V$ the set of edges (or links) between pairs of nodes defined in V . Given two nodes v_i and v_j of V , the edge $(v_i, v_j) \in E$, if the nodes are connected. In this case, we called them *adjacent* nodes or *neighbours*.

Graphs can have different characteristics based on the nature of their edges. They can be *directed*, meaning each edge has a specific direction from a source node to a target node, indicating a one-way relationship. Conversely, if the edges have no orientation, the graph is *undirected*. Additionally, graphs can be *weighted*, where each edge (v_i, v_j) has an associated weight $w_{i,j}$, representing the strength (or cost) of the connection. If there is no weight associated with the edges, the graph is called *unweighted*. See the simple representation in Figure 2.1 for a better understanding of these concepts. A graph is characterised as *simple* when it does not have self-loops—edges connecting a node to itself—or multiple edges between the same pair of nodes.

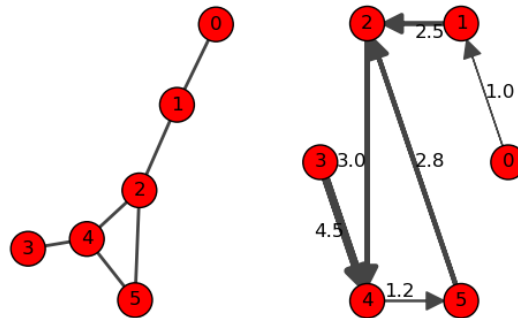


Figure 2.1: Illustration of two graphs. At the left, an undirected unweighted graph and, at the right, a directed weighted graph.

Path and *connectivity* are two important concepts in graphs. A path is an ordered sequence of nodes where each consecutive pair of nodes is linked. A connected graph has a path between any pair of nodes, ensuring the entire graph forms a single cohesive unit.

A graph is usually represented by an adjacency matrix $\mathbf{A}$, where $A_{ij} = 1$ (or $A_{ij} = w_{i,j}$, in case of a weighted graph) if there is an edge between nodes v_i and v_j , and $A_{ij} = 0$ otherwise. This matrix representation is a convenient and commonly used data structure for computationally handling graphs, as the elements are easily accessible and it allows for direct computation using linear algebra, which is often employed in graph theory methods. However, it has spatial complexity that depends on the number of nodes in the graph, n , requiring a $n \times n$ matrix. For very large graphs with relatively few edges (compared to the number of possible edges), other data structures, such as adjacency lists, are preferable to improve computational efficiency.

Network science helps in understanding complex systems by using topological metrics. Some of the available and widely used metrics are presented below. These were the metrics used in our analysis in Chapter 6.

Degree. The degree of a node, k_i , corresponds to the number of edges connected to it. In directed graphs, we can further distinguish the in-degree, $k_{i,in}$, and out-degree, $k_{i,out}$, the first relating to the number of edges pointing to node v_i , and the second to the number of edges leaving v_i . We can also focus on the degree distribution, which is the probability distribution of these degrees over the entire network, or the average degree of the network.

Density. The density, d , of a graph is the ratio of the number of edges to the number of possible edges which corresponds to the square of the total number of nodes. It measures how close the network is to being fully connected. If the value is close to 1, we call it a dense graph. Otherwise, we have what is called a sparse graph.

Shortest Path Length. The shortest path length, $d_{i,j}$, between two nodes (v_i and v_j) is the minimum number of edges that must be traversed to travel from one node to the other.

Diameter. The diameter, D , of the network corresponds to the largest shortest path length from every possible pair of nodes within the network. In other words, it is the shortest distance that separates the most distant nodes in the network [7].

There are also *multilayer graphs*, which are used to represent multiple systems and their interactions, using a layered structure. In a multilayer graph, nodes can belong to different layers, and edges can exist both within a layer (intra-layer connections) and between layers (inter-layer connections).

Mathematically, a multilayer networks can be defined as a pair $(\mathbf{G}, \mathbf{C})$, where $\mathbf{G} = \{G^\alpha; \alpha \in \{1, \dots, m\}\}$ is the family of graphs $G^\alpha = (V^\alpha, E^\alpha)$ corresponding to the m layers, and $\mathbf{C} = \{E^{\alpha\beta} \subseteq V^\alpha \times V^\beta; \alpha, \beta \in \{1, \dots, m\}, \alpha \neq \beta\}$ is the set of inter-layer connections between all pairs of layers α and β .

When $V^1 = V^2 = \dots = V^m = V$ and the only possible type of inter-layer edges are connections to counterpart nodes in other layers, we have what is called a *multiplex network*, which is a special type of multi-layer network. Formally, this means every node v_i^α is connected to every possible nodes v_i in successive layers [6]. We can see the difference between both types of multilayer networks in Figure 2.2.

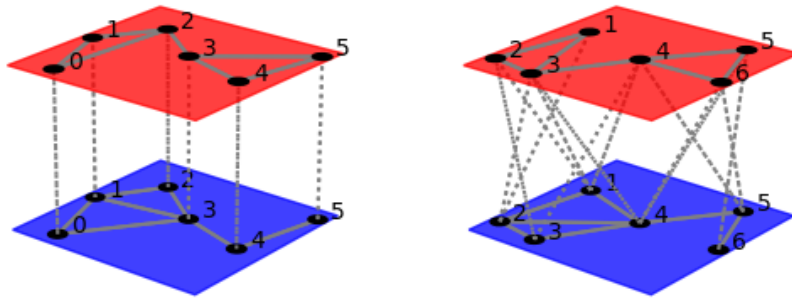


Figure 2.2: Illustration of two multilayer graphs. At the left, a multiplex network and, at the right, a general multilayer network.

2.3 Time Series to Complex Networks

Mapping methods are techniques used to transform time series data into network representations. These methods differ in how they interpret the temporal data and how they define connections between different segments or observations of the time series.

In this section, we present a hierarchy for the existent mapping methods found in the literature (see Figure 2.3), based on the dimensionality of the time series, the desired network structure, the mapping concept for the generated network, and the corresponding mapping method. This taxonomy is based on the conceptual division presented in [44] that is expanded to include new multivariate time series mapping methods recently proposed in the literature.

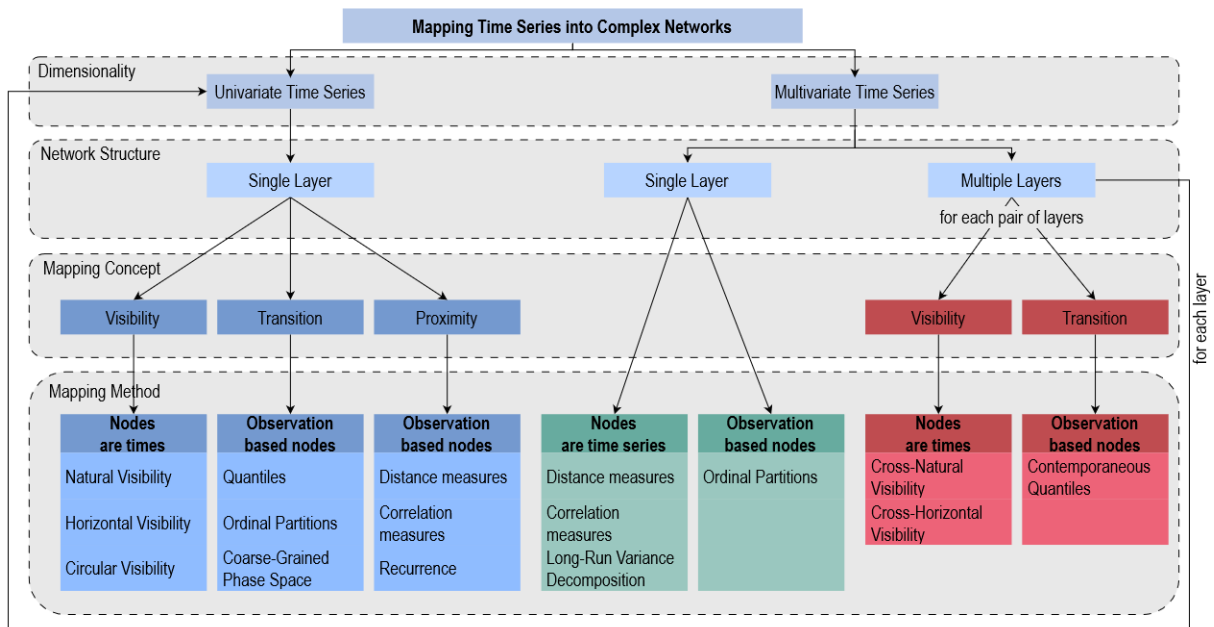


Figure 2.3: General overview of all existent time series mapping methods. *Adapted from [44].*

Univariate time series can be mapped into networks using three main concept types: visibility, transition, and proximity. Visibility mappings establish connections between timestamps using visibility lines, transition mappings utilise transition probabilities between states or partitions defined from time series data, and proximity mappings establish connections based on distance or similarity measures between time points or states.

Multivariate time series can be mapped into networks that either maintain a single-layer structure or adopt multiple layer structures. First one, each time series or temporal pattern serves as a node, with edges reflecting the relationships between these nodes based on a mapping concept. Alternatively, multiple layer networks assign each layer to individual time series or multivariate data partition, allowing for different univariate time series mapping methods to be used within each layer. Recently, new mappings have also been proposed to create non-categorical connections between nodes from different layers (highlighted in red in Figure 2.3), extending the range of multilayer mapping methods that previously lead to only multiplex networks.

More detailed explanations of each mapping method are available in Chapters 3 and 4.

In addition to discussing mapping methods for transforming time series into network representations, it is important to explore practical implementations available through software packages. Noteworthy Python tools include `pyunicorn`, `ts2vg`, and `PyIomica`. Options in R software are also available, such as the `ts2net` package. These software packages offer some, but not all of the mapping methods available, as we can see in Table 2.1. In Chapter 5, we will further explore how we also incorporated some of these Python tools in our own package implementation, to unify all the existent mapping methods, in the most efficient way possible to the community.

Package (Reference)	Language	Available methods
pyunicorn (Donges et al. [15])	Python	Univariate and multivariate time series mappings. Functional networks and recurrence networks.
ts2vg (Bergillos [3])	Python	Natural and Horizontal Visibility Graphs (with weighted, limited penetrable and parametric versions).
PyIomica (Domanskyi et al. [14])	Python	Natural and Horizontal Visibility Graphs (with dual perspective version).
vgspectra (Yela et al. [60])	Python	Spectral Visibility Graphs (for audio signals).
pytalib (Dennis Cheung)	Python	Natural and Horizontal Visibility Graphs.
ts2net (Leonardo Ferreira [24])	R	Natural and Horizontal Visibility Graphs, Quantile Graphs and Recurrence Networks (for univariate time series). Recurrence Networks (for multivariate time series).

Table 2.1: Summary of existent software packages providing time series mapping methods.

Chapter 3

Univariate Mapping Methods

This chapter aims to briefly explain the foundation of the univariate time series mapping methods existent in the literature and briefly expand on the usability and advantages of each method.

The following sections are organised by the different networks resulting from mapping methods, divided by the three possible concepts behind the mapping method: visibility, transition and proximity.

3.1 Visibility Networks

The concept of a visibility algorithm for mapping time series, using computational geometry, was first introduced by Lacasa and co-authors [30]. Essentially, the time series data is considered as a two-dimensional landscape and represented by vertical bars, where every bar can be linked to the others that can be visualised from the top of the considered one, without any obstruction.

Given the easy interpretation and implementation, this type of mapping rapidly attracted researchers, which resulted in variants of these methods beginning to emerge, with different geometric criteria associated. We divided this section into five topics to cover the various approaches for Visibility Graphs (VGs): *Natural Visibility Graph*, *Horizontal Visibility Graph*, *Difference Visibility Graph*, *Absolute Invisibility Graph*, and *Circular Visibility Graph*, each detailing specific variations and extensions of the original visibility algorithm.

3.1.1 Natural Visibility Graph

The Natural Visibility Graph (NVG), or simply VG, uses the visibility algorithm, precisely as described above (Section 3.1), pioneered by Lacasa and co-authors [30]. This method translates the time series into a complex network, with each node representing an observation value. Within this network, two nodes are connected if a straight line can connect their corresponding observations, without intersecting any intermediate data height.

Figure 3.1 illustrates the NVG concept.

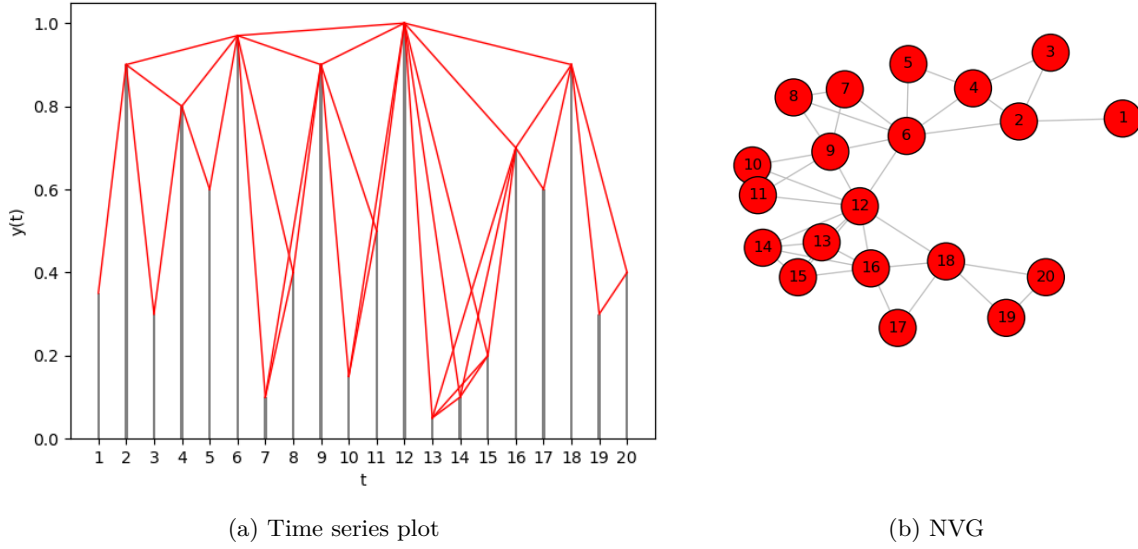


Figure 3.1: On the left, the time series plot, along the possible natural visibility lines. On the right, the correspondent NVG.

Formally, the set of nodes v_i within the NVG is indexed in chronological order, with each node v_i representing the i -th observation in the time series. Two nodes v_i and v_j are connected if, for all time series observations (t_k, y_k) , where $t_i < t_k < t_j$, the following condition is satisfied:

$$y_k < y_j + (y_i - y_j) \frac{(t_j - t_k)}{(t_j - t_i)}. \quad (3.1)$$

The original implementation (Algorithm 1) is simple to execute and use, since it is parameter free, but has quadratic complexity, that is, $\mathcal{O}(n^2)$. This results in a very slow time for longer time series. However, more recently, a more efficient technique based on Divide-and-Conquer (DC) was proposed by Lan et al. (2015) [33], which has $\mathcal{O}(n \log n)$ time complexity (see Algorithm 2).

Visibility graphs are always connected, explained by the fact that each node links with, at least, its nearest neighbour(s). They are also invariant under affine transformations of the data, as the visibility criterion remains consistent even when the axes are re-scaled or translated [30]. They can also have a direction associated, defined by the direction of the time axis. Particularly, NVGs are great at preserving the structural properties of the time series data.

New variations of the NVG have since emerged, from adding new parameters to changing the base algorithm. We will expand on these variations on the next sections.

Algorithm 1: Original Algorithm for NVG

```

1 function NaturalVisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ ):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \emptyset$ ;
4   for  $i$  from 1 to  $n - 1$  do
5     for  $j$  from  $i + 1$  to  $n$  do
6        $y_k \leftarrow y[i + 1 \dots j]$ ;
7        $t_k \leftarrow t[i + 1 \dots j]$ ;
8       if all  $(y_k[k] < (y_j + (y_i - y_j) \cdot (t_j - t_k[k]) / (t_j - t_i)))$  for  $k$  in  $\text{range}(\text{len}(y_k))$ 
9         then
10          add edge( $t_i, t_j$ ) to  $E$ ;
11 return  $G = (V, E)$ ;

```

Algorithm 2: DC Algorithm for VG

```

1 function NaturalVisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ ):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \text{NaturalVisibilityGraphRecursive}(ts, 1, n)$ ;
4   return  $G = (V, E)$ ;
5 function NaturalVisibilityGraphRecursive( $ts, left, right$ ):
6    $E' \leftarrow \emptyset$ ;
7   if  $left < right$  then
8      $imax \leftarrow \text{index } i, left \leq i \leq right, \text{ with highest } y_i$ ;
9     for  $a$  from  $left$  to  $right$  except  $imax$  do
10      if node  $imax$  sees node  $a$  then
11        add edge( $t_{imax}, t_a$ ) to  $E'$ ;
12       $E' \leftarrow E' \cup \text{NaturalVisibilityGraphRecursive}(ts, left, imax - 1)$ ;
13       $E' \leftarrow E' \cup \text{NaturalVisibilityGraphRecursive}(ts, imax + 1, right)$ ;
14 return  $E'$ ;

```

3.1.1.1 Limited Penetrable Visibility Graph

The Limited Penetrable Visibility Graph (LPVG) [65] introduces a new parameter to the original NVG, with the goal of improving the existent mapping to adapt to noise in the data, distinguishing it from chaos.

The penetrable limit distance parameter, l , lessens the restriction of the original visibility condition, as now, the visibility lines can intercept, but only up to l other data heights, as seen in Figure 3.2. When $l = 0$, the LPVG corresponds to the classic NVG.

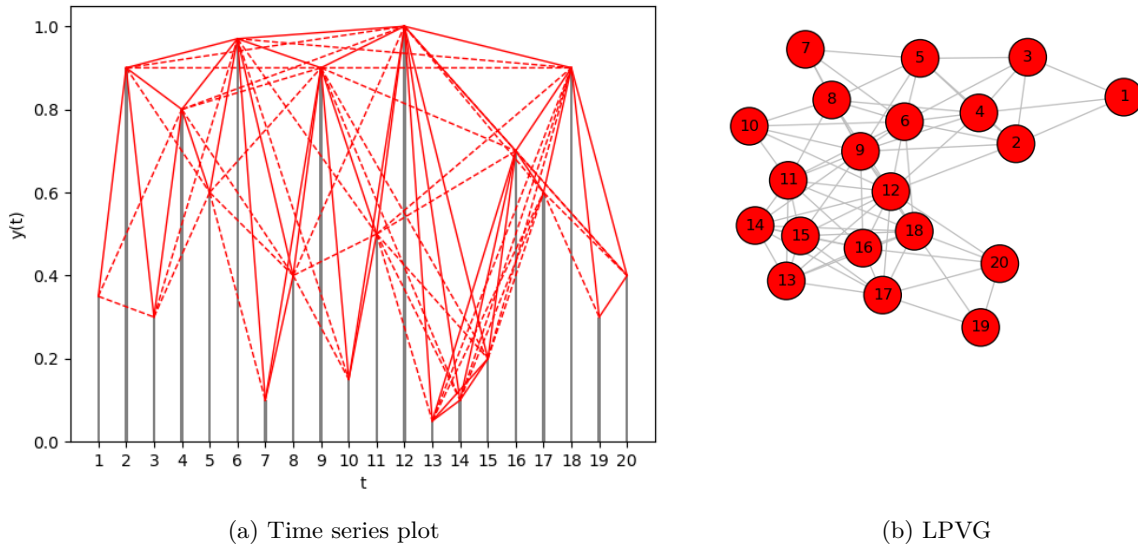


Figure 3.2: On the left, the time series plot, along the possible natural visibility lines. The dashed lines represent lines that would not be present in the Natural Visibility Graph. On the right, the correspondent LPVG, with $l = 1$.

This means that two nodes v_i and v_j are connected, if, and only if, there are no more than l sampling points (t_k, y_k) , where $t_i < t_k < t_j$, for which the following condition is true:

$$y_k > y_j + (y_i - y_j) \frac{(t_j - t_k)}{(t_j - t_i)}, \quad (3.2)$$

while all the other intermediary points satisfy Equation 3.1.

3.1.1.2 Weighted Visibility Graph

The Weighted Visibility Graph (WVG), first introduced in [52], follows the same principle as the original NVG algorithm, but assigns a direction and a weight value to each of the graph edges.

The weight corresponds to the view angle, calculated by the following:

$$w_{ij} = \arctan \left(\frac{y_j - y_i}{t_j - t_i} \right), \quad i < j. \quad (3.3)$$

This formula returns both negative and positive values, so the absolute value ($|w_{ij}|$) was used as the edges values instead, in radian function.

Other alternative calculations for obtaining the WVG weight values have been proposed in other works, such as the one in [55].

3.1.1.3 Parametric Natural Visibility Graph

By adding a restriction to the edge assignment process in the WVG, we can obtain a Parametric Natural Visibility Graph (PNVG) [4]. The PNVG uses the common NVG criteria, but sets for every link a direction and a weight. The weight corresponds to the angle α_{ij} (Equation 3.3) between the downward direction and the line connecting y_i to y_j on the time series plot. After the weight calculation phase, links are selected according to a new introduced parameter, that is, the view angle α :

$$(v_i, v_j) \in E \text{ of a PNVG if } \alpha_{ij} < \alpha. \quad (3.4)$$

The PNVG corresponds to a directed acyclic graph, and is always a subgraph of the respective NVG; it can either be a connected or a disconnected graph.

3.1.1.4 Weighted Dual Perspective Visibility Graph

The Weighted Dual Perspective Visibility Graph (WDPVG) was proposed in [64], in an attempt to overcome two disadvantages of the NVG, in applications in biological time series analysis: the NVG does not take into account uneven sampling and it does not capture time series changes below a zero baseline. If the time series is non-negative or if changes below the baseline are not of interest, the creation of a WDPVG is unnecessary.

The WDPVG works in two major steps. First, it constructs two weighted NVGs, one from the regular time series Y , but the other from the inverted time series Y' , obtained by reflecting the original series across the time axis, resulting in $\{y'_t = -y_t\}_{t=1}^n$. The weight can be calculated by either of the following equations:

$$w_{ij} = \sqrt{(y_i - y_j)^2 + (t_i - t_j)^2}, \quad (3.5)$$

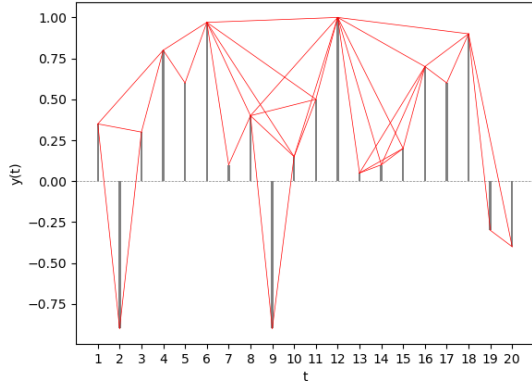
$$w_{ij} = \left| \frac{y_i - y_j}{t_i - t_j} \right| + 10^{-8}, \quad (3.6)$$

$$w_{ij} = |t_i - t_j|. \quad (3.7)$$

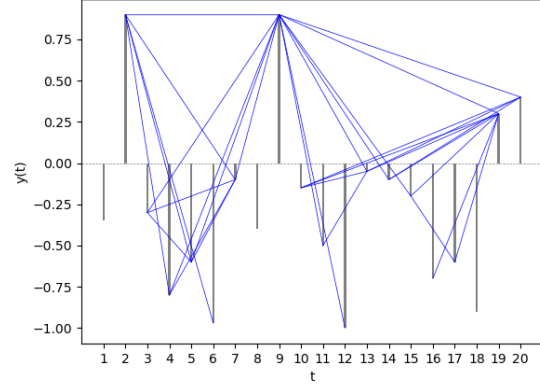
The second step is to combine the normal perspective NVG and the reflected perspective NVG into one WDPVG network. Considering the two respective adjacency matrices, A and A' , for any pair of elements i, j , A_{ij} and A'_{ij} have two possible relationships: either $A_{ij} = A'_{ij}$, or one of them is 0, while the other is non-zero. So, we can combine A and A' to get the WDPVG adjacency matrix A^d , by the following criterion:

$$A_{ij}^d = \max\{A_{ij}, A'_{ij}\}. \quad (3.8)$$

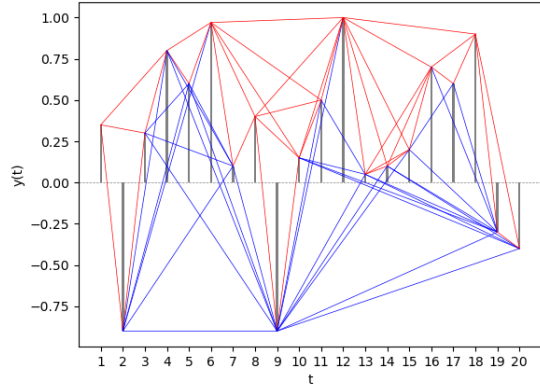
Figure 3.3 illustrates the process and the different visibility lines, according to the considered perspective.



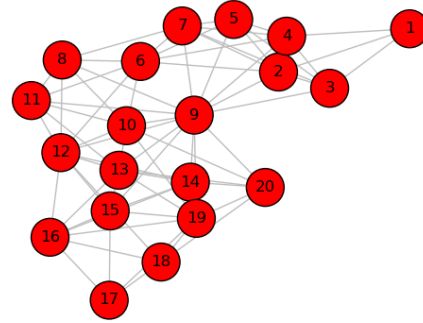
(a) Time series plot with normal perspective



(b) Time series plot with reflective perspective



(c) Time series plot with all perspectives



(d) WDPVG

Figure 3.3: On the upper left, the time series plot, along the possible natural visibility lines, according to normal perspective. On the upper right, the time series plot, along the possible natural visibility lines, according to reflective perspective. On the lower left, the time series plot, combining all possible natural visibility lines. On the lower right, the correspondent WDPVG.

The algorithm for the WDPVG is available on the `PyIOmica` package [14] and presented in Algorithm 3. We can see that the complexity associated with this algorithm is dependent of the time complexity used for the creation of the NVG (see lines 5 and 6) and also on the for-loops from lines 19 to 25, making the overall time complexity of the method to be $\mathcal{O}(n^2)$, dominated by these for-loops.

Algorithm 3: Algorithm for DPVG

```

1 function DualPerspectiveVisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ , withsign):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \emptyset$ ;
4    $inv\_ts \leftarrow \{(t_1, -y_1), \dots, (t_n, -y_n)\}$ ; ▷ Inverted time series
5    $A\_posi \leftarrow$  adjacency matrix of NaturalVisibilityGraph( $ts$ );
6    $A\_nega \leftarrow$  adjacency matrix of NaturalVisibilityGraph( $inv\_ts$ );
7   if  $\text{minimum}(y_1, \dots, y_n) < 0$  then
8     foreach element  $i$  in  $A\_posi$  and  $A\_nega$  do
9       if  $A\_posi[i] \geq A\_nega[i]$  then
10          $A\_dual[i] \leftarrow A\_posi[i]$ 
11       else
12          $A\_dual[i] \leftarrow -A\_nega[i]$ 
13   else
14      $A\_dual \leftarrow A\_posi$ 
15   if withsign is false then
16      $A\_dual \leftarrow$  absolute value of  $A\_dual$ 
17   for  $i$  from 1 to  $n$  do
18     for  $j$  from 1 to  $n$  do
19       if  $A\_dual[i][j] \neq 0$  then
20         add edge( $t_i, t_j$ ) to  $E$ ;
21   return  $G = (V, E)$ ;

```

3.1.2 Horizontal Visibility Graph

The horizontal visibility algorithm was initially introduced by Luque and co-authors [34], as an alternative to the original visibility algorithm, due to its geometrically straightforward nature. In this approach, if we once again consider the vertical bars of a time series plot as a two-dimensional landscape, a vertical bar can only see directly ahead horizontally. It connects to other observations it can sight without obstructions, and it does not need to see the top of the other bar, just any part of its height.

The Horizontal Visibility Graph (HVG) employs the horizontal visibility algorithm. Formally, two nodes v_i and v_j are connected, if, for all (t_k, y_k) data occurrences, such that $t_i < t_k < t_j$, the following condition is met:

$$y_i, y_j > y_k. \quad (3.9)$$

Figure 3.4 shows a visual representation of the algorithm, as well as the respective HVG.

The HVG is a sub-graph of the NVG and, so, it shares many similar characteristics to the NVG, like the invariance under affine transformations and the connectivity of the graph [34].

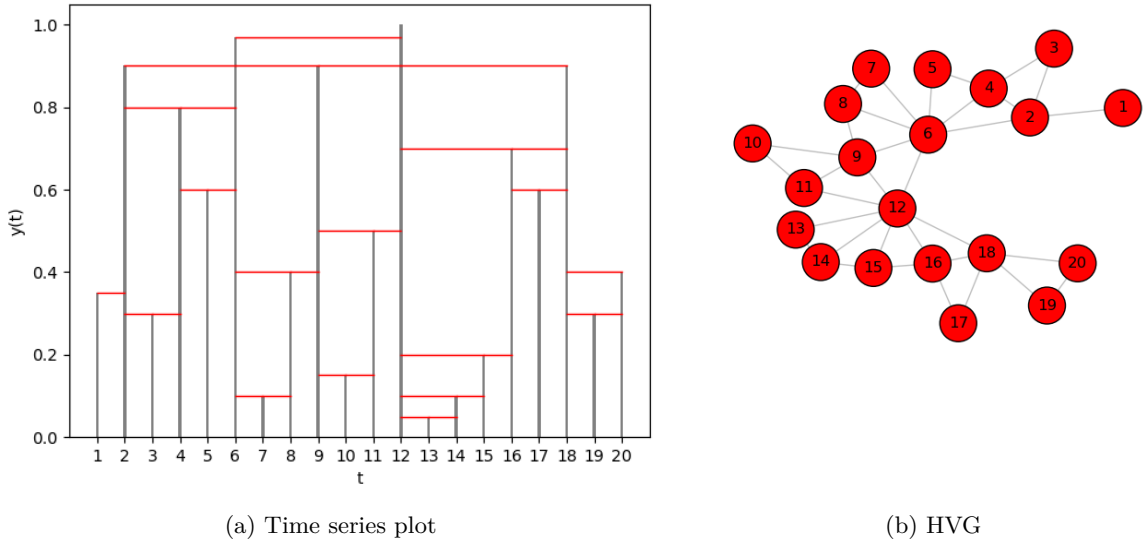


Figure 3.4: On the left, the time series plot, along the possible horizontal visibility lines. On the right, the correspondent HVG.

The original algorithm for the HVG (see Algorithm 4) had complexity of $\mathcal{O}(n^2)$, just like the NVG. However, since then, many approaches have been developed to improve time efficiency. The dual tree method [50] (illustrated in Algorithm 5) has linear time complexity $\mathcal{O}(n)$, making it particularly useful for large time series. For smaller time series, however, a DC approach might suffice, as suggested in [50].

Algorithm 4: Original Algorithm for HVG

```

1 function HorizontalVisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ ):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \emptyset$ ;
4   for  $i$  from 1 to  $n - 1$  do
5     for  $j$  from  $i + 1$  to  $n$  do
6        $y_k \leftarrow y[i + 1 \dots j]$ ;
7        $t_k \leftarrow t[i + 1 \dots j]$ ;
8       if all ( $y_k[k] < \min(y_i, y_j)$  for  $k$  in  $\text{range}(\text{len}(y_k))$ ) then
9         add edge( $t_i, t_j$ ) to  $E$ ;
10  return  $G = (V, E)$ ;
```

Algorithm 5: Linear-Time Algorithm for HVG

```

1 function HorizontalVisibility_GraphDualTree( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ ):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \emptyset$ ;
4    $\Gamma \leftarrow \text{Stack}()$ ; ▷ Regions with an edge in trailing branch of  $T_{ts}$ 
5    $\gamma \leftarrow -\infty$ ; ▷ Store max value seen so far
6    $\Lambda \leftarrow \text{Stack}()$ ; ▷ Regions with an edge in leading branch of  $T_{ts}$ 
7   foreach  $v$  in  $V$  do
8     if  $y_v > \gamma$  then
9        $\gamma \leftarrow y_v$ ;
10       $\Gamma.\text{push}((v, y_v))$ ;
11      while  $\Lambda$  is not empty do
12         $(u, y_u) \leftarrow \Lambda.\text{pop}()$ ;
13        add edge( $u, v$ ) to  $E$ ;
14        if  $y_u > y_v$  then
15           $\Lambda.\text{push}((u, y_u))$ ;
16        if  $y_u \geq y_v$  then
17          break;
18       $\Lambda.\text{push}((v, y_v))$ ;
19  return  $G = (V, E)$ ;

```

3.1.2.1 Directed Horizontal Visibility Graph

Similarly to the NVG, directed edges can also be applied to the horizontal visibility concept, resulting in a Directed Horizontal Visibility Graph (DHVG) [31], which can be observed in Figure 3.5.

3.1.2.2 Limited Penetrable Horizontal Visibility Graph

The Limited Penetrable Horizontal Visibility Graph (LPHVG) [25], similarly to the LPVG, also introduces a new parameter, but to the original HVG (see Figure 3.6).

This penetrable limit distance parameter, l , adds more edges in comparison with the original horizontal visibility condition, as now, the visibility lines can intercept other observations, but only up to l other data heights. When $l = 0$, the LPHVG represents the original HVG.

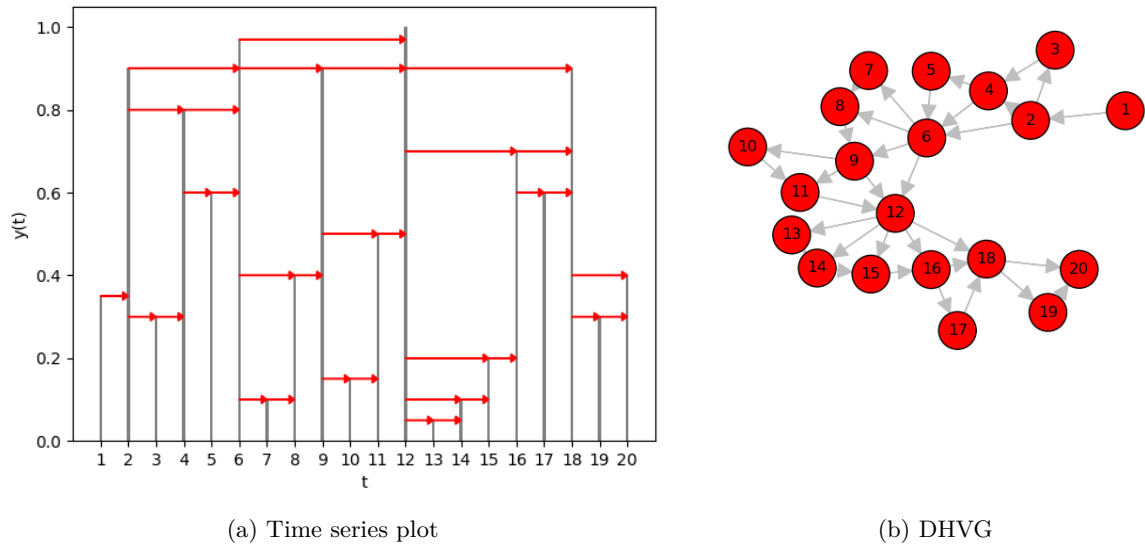


Figure 3.5: On the left, the time series plot, along the possible directed horizontal visibility lines. On the right, the correspondent DHVG.

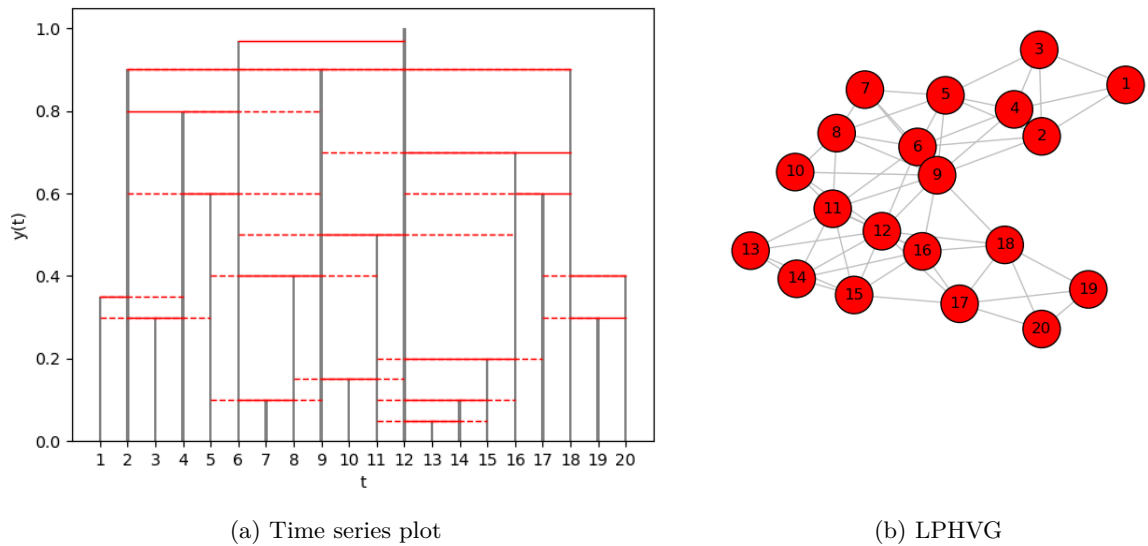


Figure 3.6: On the left, the time series plot, along the possible horizontal visibility lines. The dashed lines represent lines that would not be present in the Horizontal Visibility Graph. On the right, the correspondent LPHVG, with $l = 1$.

3.1.2.3 Weighted Horizontal Visibility Graph

The Weighted Horizontal Visibility Graph (WHVG) follows the same principal as the WVG, but applied to the horizontal algorithm: after constructing the HVG, we just need to designate weights to the edges, based on a chosen metric.

The same criterion for WVG also holds here. The weight corresponds to the view angle [52], calculated by the following:

$$w_{ij} = \left| \arctan \left(\frac{y_j - y_i}{t_j - t_i} \right) \right|, \quad i < j. \quad (3.10)$$

Another alternative way to calculate the weights can be found in [5], and is as follows:

$$w_{ij} = 1/\sqrt{(t_j - t_i)^2 + (y_i - y_j)^2}. \quad (3.11)$$

Other criteria for the weight calculation were also proposed in [29, 55, 67].

3.1.2.4 Weighted Dual Perspective Horizontal Visibility Graph

Instead of considering the natural visibility mapping criterion, if we use the horizontal algorithm, we have a Weighted Dual Perspective Horizontal Visibility Graph (WDPHVG) [64], as present in Figure 3.7.

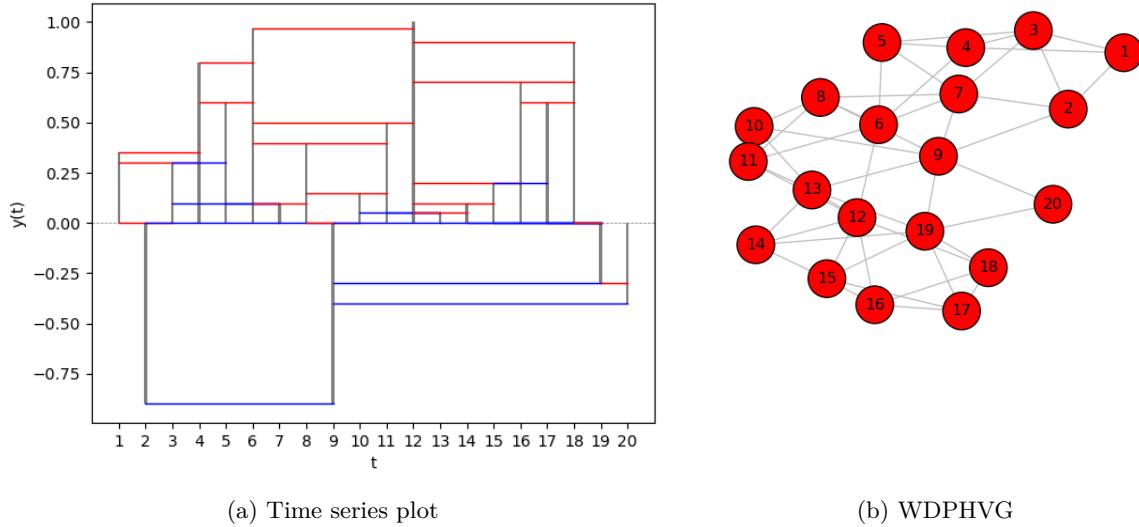


Figure 3.7: On the left, the time series plot, along the possible horizontal visibility lines: in red, the ones obtained from normal perspective, and, in blue, the ones obtained from reflective perspective. On the right, the correspondent WDPHVG.

Again, constructing a dual perspective graph involves two major steps: first, it constructs two weighted HVGs, a normal perspective HVG, constructed from the regular time series y , and a reflected perspective HVG, obtained from the inverted time series Y' , corresponding to the reflection of the original series across the time axis ($\{y'_t = -y_t\}_{t=1}^n$). The weight associated to each edge between a pair of nodes v_i and v_j can be calculated according to the same Equations presented in Section 3.1.1.4.

The second task is to combine both HVGs into one network. If we consider the two respective adjacency matrices, A and A' , for any pair of elements i, j , we can merge A and A' to get the Dual Perspective adjacency matrix A^d , by the following:

$$A_{ij}^d = \max\{A_{ij}, A'_{ij}\}. \quad (3.12)$$

This algorithm is also present in the PyIOMica framework [14], following the same steps of the WDPVG approach (see Algorithm 3), with $\mathcal{O}(n^2)$ time complexity.

3.1.3 Difference Visibility Graph

The Difference Visibility Graph (DVG) is a concept introduced by Zhu et al. [66], that relies on the fact that a HVG is a subset of the corresponding NVG, constructed from the same time series. It is obtained by removing the edges set of the HVG from the edges set of the NVG, thus isolating the unique connections that are present in the NVG, but excluded from the HVG. Figure 3.8 illustrate this method.

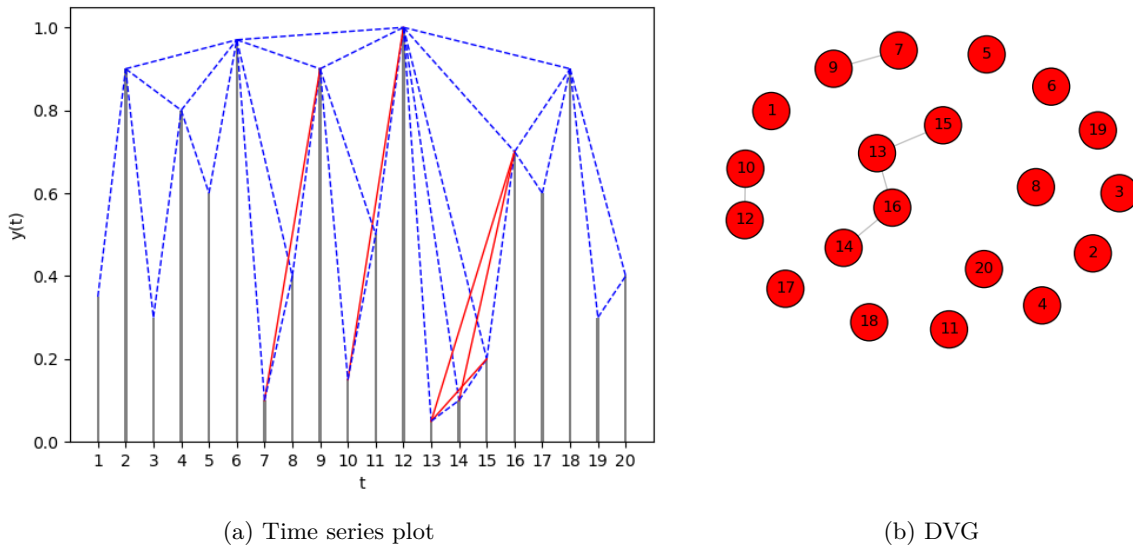


Figure 3.8: On the left, the time series plot, along the possible visibility lines, in red, and, in blue, the edges subtracted from the DVG, that would be present in the Natural Visibility Graph. On the right, the correspondent DVG.

Formally, if $G_{NVG}(V, E_1)$ and $G_{HVG}(V, E_2)$ correspond to NVG and HVG, respectively, then the DVG can be represented by $G_{DVG}(V, E_1 - E_2)$.

A notable advantage of DVGs is that they address a common issue in both NVGs and HVGs, where many nodes often have a degree value of $k = 2$, if they are only connected to their neighbours. This can result in a lack of variability in the degree distribution, making it harder to distinguish between different signals. For this reason, DVGs are also likely to be disconnected and may contain isolated nodes.

3.1.4 Absolute Invisibility Graph

The Absolute Invisibility Graph (AIG) [58] uses an absolute invisibility algorithm, which reverses the condition for the original NVG algorithm. So, essentially, two nodes v_i and v_j are connected, if, for every time series observation (t_k, y_k) , where $t_i < t_k < t_j$, the following condition holds:

$$y_k > y_j + (y_i - y_j) \frac{(t_j - t_k)}{(t_j - t_i)}. \quad (3.13)$$

This will lead to both NVG and AIG having some edges in common, as both connect each observation to their adjacent neighbours. For this reason, AIG is also considered a connected graph (see Figure 3.9).

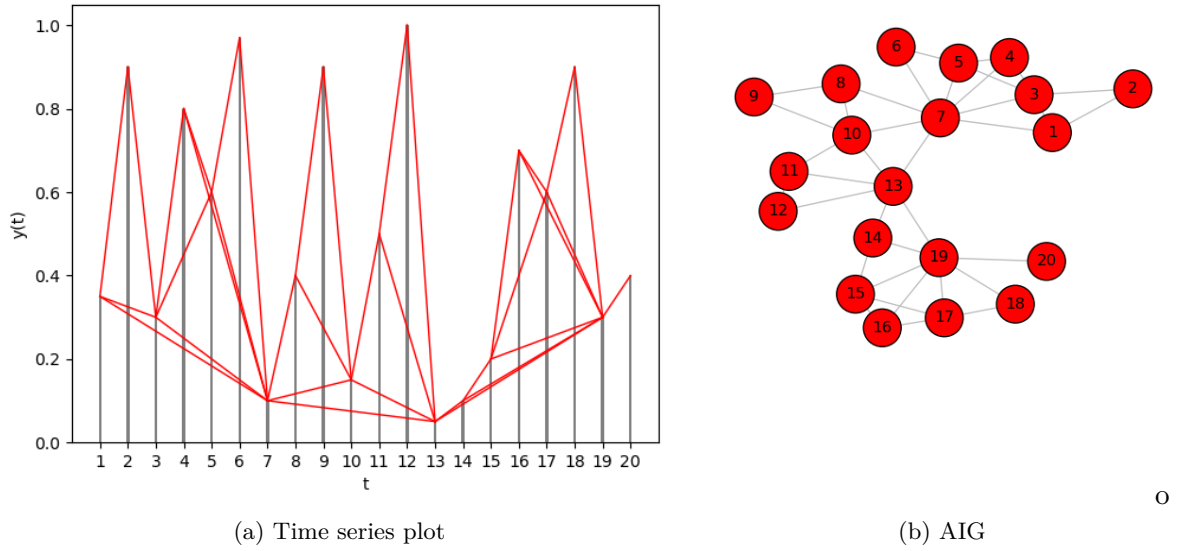


Figure 3.9: On the left, the time series plot, along the possible natural invisibility lines. On the right, the correspondent AIG.

3.1.5 Circular Visibility Graph

The Circular Visibility Graph (CVG) [57] introduces a novel approach in visibility algorithms by integrating the concept of a circle system. This method uses circular visibility arcs, instead of straight visibility lines to establish connections between data points in a time series, as illustrated in Figure 3.10.

We will start by explaining the concept of a circle system: for any given two points, there are several circles that can pass through them, as we can see in Figure 3.11. Each of these circles has a different central angle α associated that influences the radius of the associated circle: as α increases, the radius of the circle increases, resulting in smaller central angles for the arcs. The center of the circle is on the perpendicular bisector of the line segment between the two points.

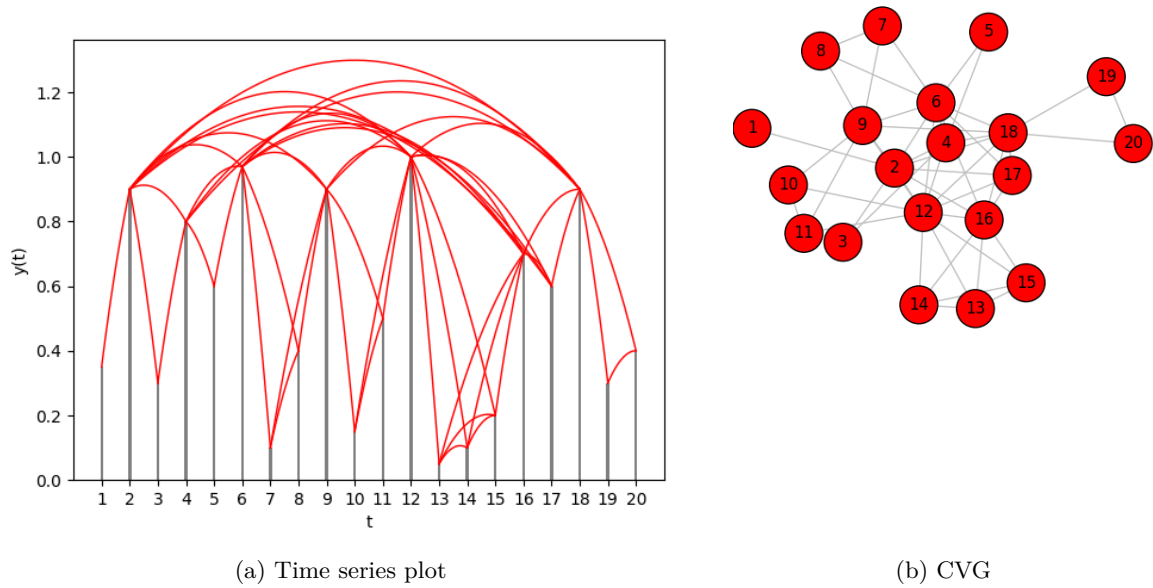


Figure 3.10: On the left, the time series plot, along the possible circular visibility arcs. On the right, the correspondent CVG, with $\alpha = -10$.

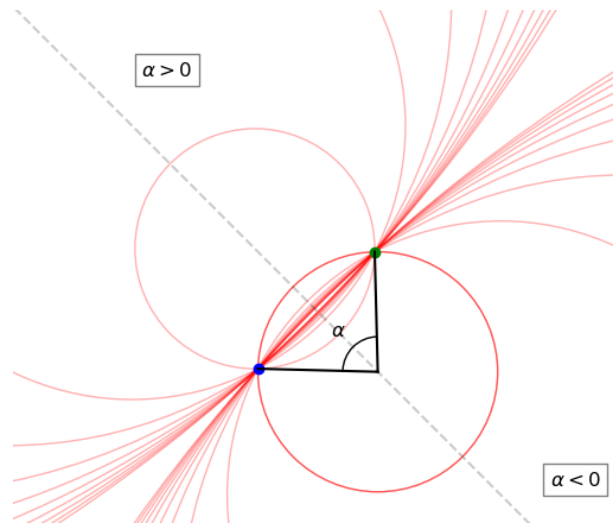


Figure 3.11: Illustration of the circle system for two arbitrary data points.

A circle system between two data points $A(t_a, y_a)$ and $B(t_b, y_b)$ can be described by the following parametric equation:

$$f(t, y) = (t - t_a)(t - t_b) + (y - y_a)(y - y_b) + \alpha [(t - t_a)(y_b - y_a) - (y - y_a)(t_b - t_a)] = 0. \quad (3.14)$$

For a certain α , there is only one circle that passes through the two data points. If we consider the smaller part of the circle, indexed by the angle, we have a minor arc between the two points. By using the minor arc, we ensure that the visibility arc is meaningful and that the connections are appropriately defined.

As expected, the parameter α plays a crucial role in determining the shape and size of the visibility arcs:

- When $\alpha < 0$, the circle midpoint is below the connecting visibility arc, making the arc concave down.
- When $\alpha = 0$, the line segment connecting the two points is the diameter of the circle.
- When $\alpha > 0$, the circle midpoint is above the line connecting the given two points, resulting in a concave up arc.
- When $\alpha \rightarrow \infty$, the circle approximates a straight line.

Given the Equation 3.14, the parameter α , and the coordinates of the two data points (t_i, y_i) and (t_j, y_j) , we can determine the parametric function of the circle arc that passes through the two observations. Then, for each observation timing t_k , such that $t_i < t_k < t_j$, we calculate the solution y_{cir} of this function. The respective nodes v_i and v_j will be linked, after the comparison with the respective data value y_k :

$$y_k < y_{cir}. \quad (3.15)$$

3.1.5.1 Circular Limited Penetrable Visibility Graph

The Circular Limited Penetrable Visibility Graph (CLPVG), also introduced in [57], applies the same limited penetrability concept, but for the CVG, as illustrated in Figure 3.12.

Similarly to LPVG and LPHVG, every sampling point (t_i, y_i) is represent as a vertex v_i , and a mutual link between two nodes v_i and v_j is created, if, and only if, no more than l sampling points (t_k, y_k) , with $t_i < t_k < t_j$, satisfy:

$$y_k > y_{cir}, \quad (3.16)$$

while all the other data points satisfy the criterion expressed in Equation 3.15.

The CLPVG shows effectiveness and robustness in the processing of noisy signals. Compared to the LPVG, it presents higher performance on time series classification [57].

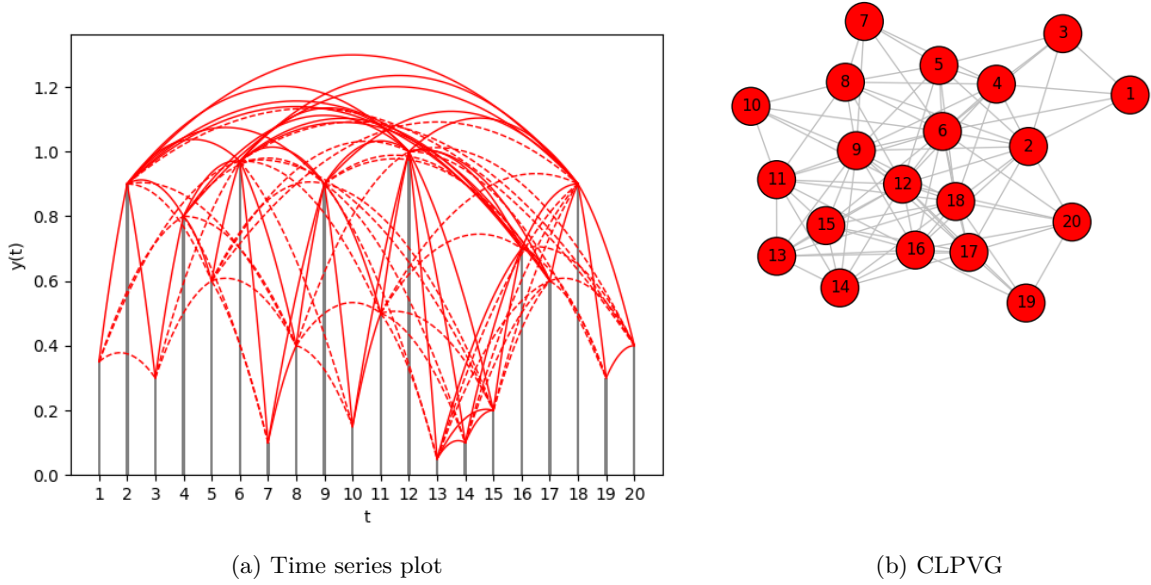


Figure 3.12: On the left, the time series plot, along the possible circular visibility arcs. The dashed lines represent lines that would not be present in the Circular Visibility Graph. On the right, the correspondent CLPVG, with $\alpha = -10$ and $l = 1$.

3.2 Transition Networks

While in the previous Section 3.1, we unveiled the visibility concepts available for time series mappings, now we will focus on a new approach: transition probabilities between discrete states.

In general, transition networks are constructed, first, by representing the time series values as a certain set of symbols, and second, by attributing as the edge weight between two symbols the transition probabilities between these states.

This first task ensures the symbolic encoding of the time series data, either by partitioning the time range, leading to the reconstruction of the time series into a new phase space, or by partitioning the support of the time series [44]. The first method is used in the construction of *Coarse-Grained Phase Space Graphs*, *Ordinal Partition Networks*, and *Visibility Graphlets Networks*, while the second leads to the *Quantile Graphs*.

Particularly, the reconstruction of phase-space consists of embedding the time series $(y_1, \dots, y_n)$ into an w -dimensional space, with a time delay τ between the states, resulting in a set of vectors $\vec{z}_t$, with $t = 1, \dots, n - (w - 1) \cdot \tau$, being n the length of the original time series. Each one of these vectors $\vec{z}_i$ can be calculated using the following formula:

$$\vec{z}_i = \left(y_i, y_{i+\tau}, y_{i+2\tau}, \dots, y_{i+(w-1)\tau} \right). \quad (3.17)$$

By constructing such vectors for different time points, we obtain a set of points in a w -dimensional space, which represents the phase space of the original time series. This method, known as time-delay embedding, is commonly used to analyse the underlying complex dynamics of the system that generated the time series, particularly in physics [44].

Once the time series is transformed into a set of symbols or states, the second step involves mapping these symbols and the transition function between them into nodes and edges, respectively. A weighted and directed edge is obtained by considering the non-zero transition probability between two states. Mathematically, this system can be described as a Markov chain.

Although, by default, weights and directions are desirable to understand the dynamics of these complex systems, an unweighted representation of the transition network may also be obtained. This can be achieved by omitting the information on the probability or quantity of transitions between different states. Additionally, a threshold can be applied to the transition probabilities, resulting in a less dense graph [69].

3.2.1 Quantile Graph

The Quantile Graph (QG) [10] involves the discretisation of the time series into Q quantiles, that is, assigning each time series value to one of the existing Q bins. So, in QGs, each node v_i represents a different quantile q_i , with $i = 1, \dots, Q$.

Each quantile q_i has an associated boundary given by the following equation:

$$q_i = y_{(n*i)/Q}, \quad (3.18)$$

being n the length of the time series. An observation belongs to q_i if its value is less than the calculated value for q_i , but bigger or equal that the one calculated for q_{i-1} .

Two nodes v_i and v_j are connected with a directed and weighted arc (v_i, v_j, w_{ij}) , where w_{ij} represents the transition probability in a Markov model between states from the time series, in this case, between quantiles (see Figure 3.13).

If the length n of the time series is greater than Q , the resulting network is also expected to be connected. This mapping is also insensitive to the distribution of values in the time series, requiring only the specification of Q .

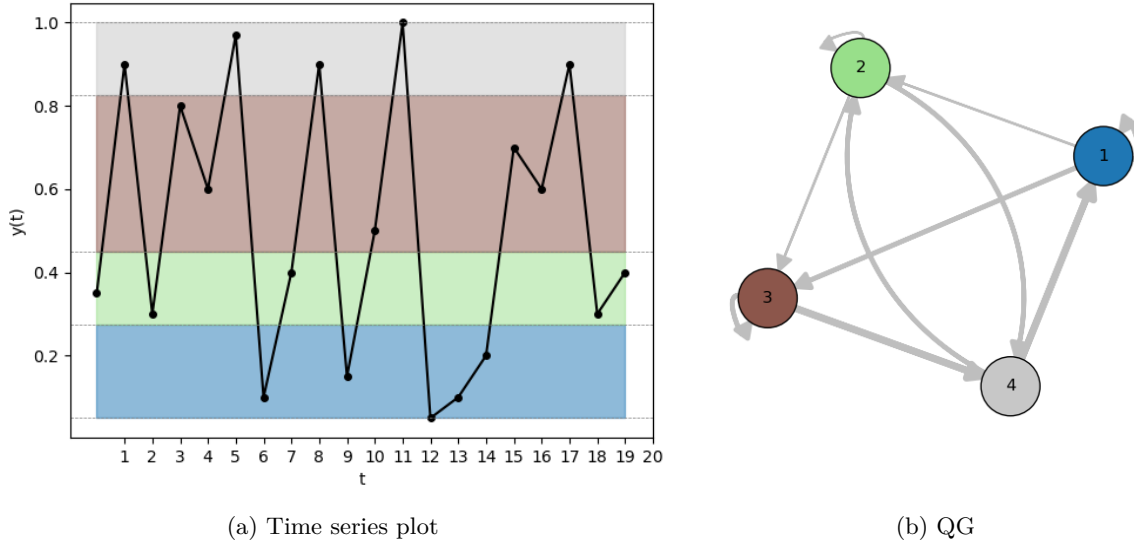


Figure 3.13: On the left, the time series plot, along the possible quantiles. On the right, the correspondent QG, with $Q = 4$. Each node has the same colour as the associated quantile.

3.2.2 Ordinal Partition Transition Network

By embedding the time series, in a w -dimensional space using a time delay τ , and then discretising the resulting vectors z_t based on the descending rank of the respective values within one vector, we obtain a set of sequential patterns π_t .

The Ordinal Partition Transition Network (OPTN) [48] works by considering each of these defined patterns as a node and assigning directed and weighted edges between two nodes, if the patterns are consecutive in the set of the discretised vectors. Thus, the assigned weight to an edge corresponds to the transition frequency between two sequential patterns.

Formally, to construct an OPTN for a univariate time series, the time series is embedded in a w -dimensional space using a time delay τ . For each of these vectors $\vec{z}_i$, we define the corresponding ordinal pattern by assigning ranks to the data y_t in descending order:

$$\pi_i = (r_1, r_2, \dots, r_w), \quad \text{where } r_j \in \{1, 2, \dots, w\}, \quad \text{and } r_j \neq r_k \text{ if } j \neq k. \quad (3.19)$$

As we explained, each node v_i corresponds to a pattern π_i and edges are allocated between nodes based on the transition sequence of the symbols π_i . This creates a directed and weighted graph that encapsulates the dynamics of the original time series. An example is present in Figure 3.14.

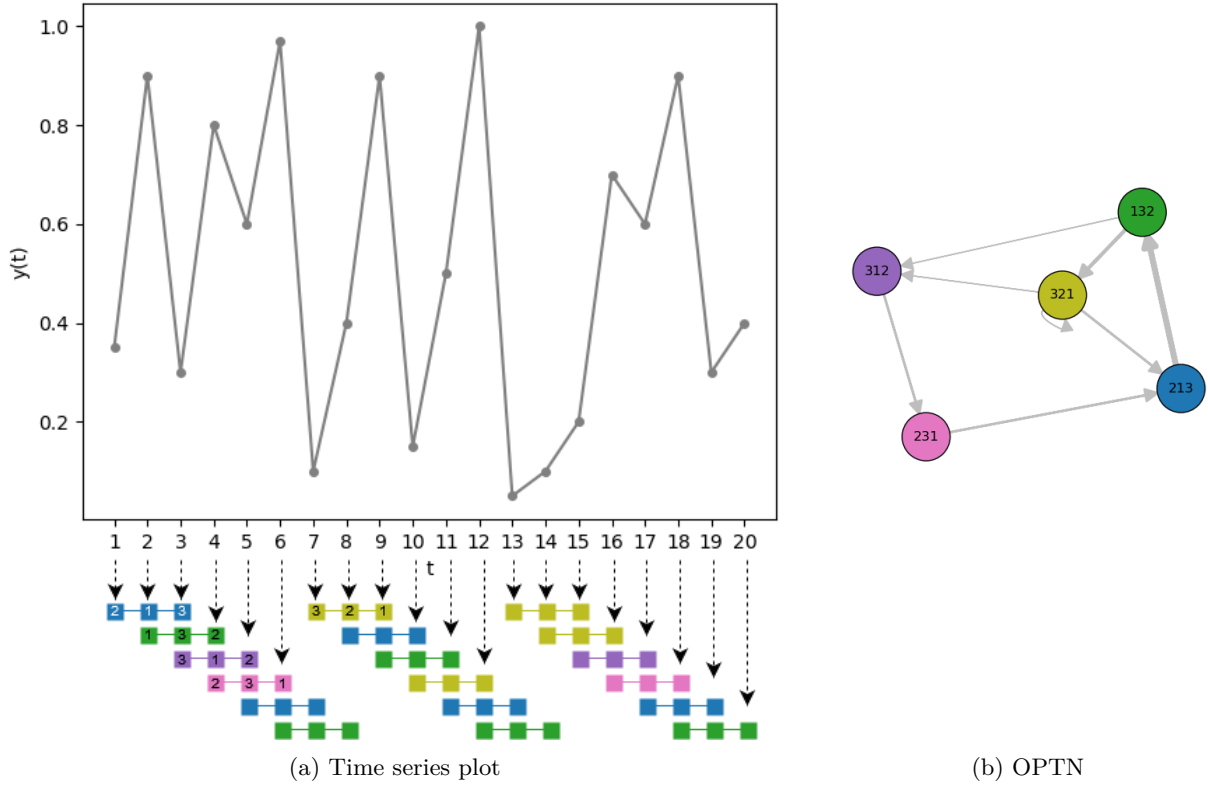


Figure 3.14: On the left, the time series plot, along the embedded vectors and respective discretised patterns. On the right, the correspondent OPTN, with $w = 3$ and $\tau = 1$. Each node has the same colour as the associated pattern.

3.2.3 Coarse-Grained Phase Space Graph

In the Coarse-Grained Phase Space Graph [26], we consider again the phase space resulting from the embedding of the time series, in a w -dimensional space using a time delay τ , instead of the original time series.

The process for coarse-graining a phase space involves a transformation similar to the one used in the QG (Section 3.2.1), as we also use a set of values to delimit our observations. However, while in the QG we mapped the entire time series support into calculated quantiles, which served as 'bins', here, we partition the values present in each vector of the set of vectors representing the embedded time series, according to a specified set of values, which results in a symbol group of w values [44].

Mathematically, let $\vec{z}_i = (y_i, y_{i+\tau}, \dots, y_{i+(w-1)\tau})$ represent an embedded vector from the time series. We define a set of m partitioning values $P = \{p_1, p_2, \dots, p_{m-1}\}$ where $p_1 < p_2 < \dots < p_{m-1}$. Each element $y_{i+k\tau}$ in the vector $\vec{z}_i$ is then assigned to a bin defined by the intervals created by P , resulting in each vector $\vec{z}_i$ being represented by a sequence of w symbols. Each symbol ψ_j in this sequence is determined by the following rule:

$$\psi_j = \begin{cases} \psi_1 & \text{if } y_t < p_1, \\ \psi_i & \text{if } v_{i-1} \leq y_t < p_i, \quad 1 < i < m-1, \\ \psi_m & \text{if } y_t \geq p_{m-1}. \end{cases} \quad (3.20)$$

Thus, each vector $\vec{z}_i$ is represented by a set of w symbols $\{\psi_j\}_{j=1}^w$, where each ψ_j corresponds to one of the partition intervals defined by P .

This ensures the embedded time series is represented by a sequence of groups composed by symbols, where each unique sequence corresponds to a node, and edges between nodes represent transitions between consecutive patterns in the set of embedded vectors. The weights of the edges reflect the frequency of these transitions. Figure 3.15 illustrates the process.

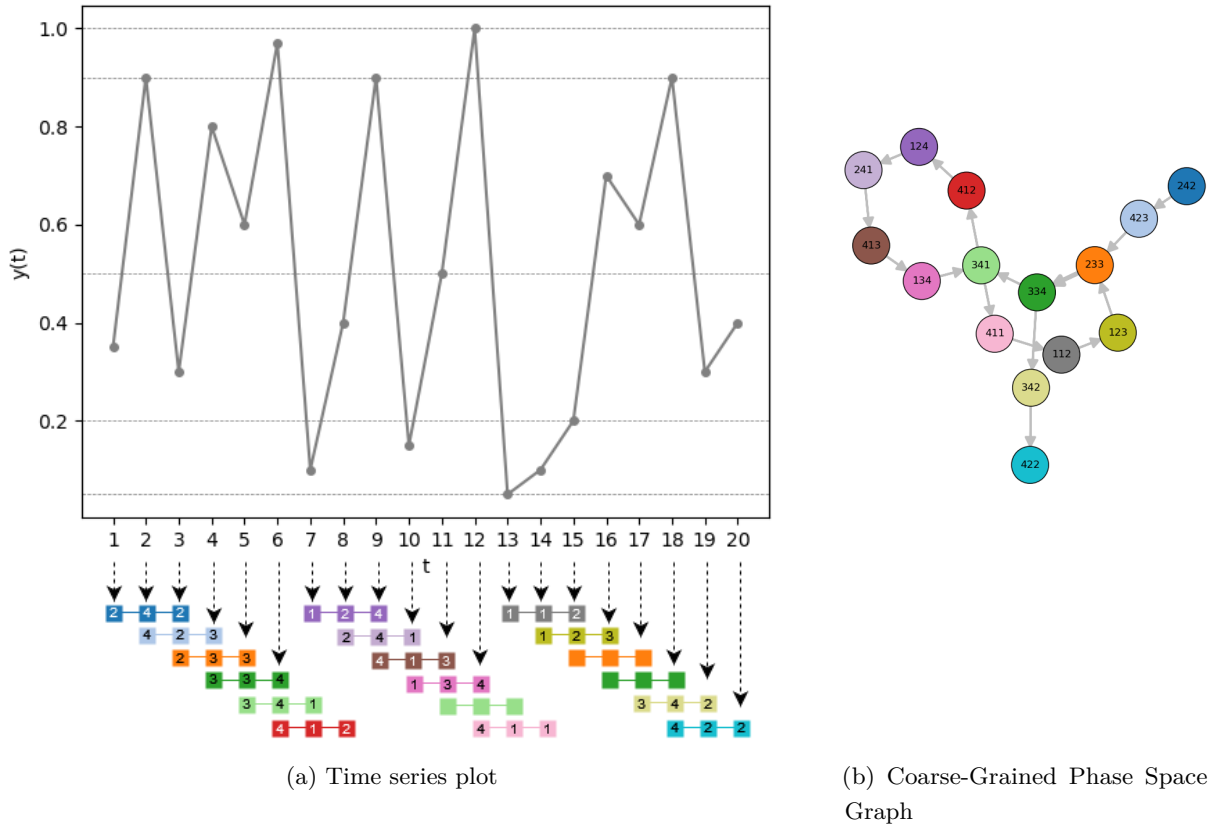


Figure 3.15: On the left, the time series plot, along the embedded vectors and respective discretised patterns. On the right, the correspondent Coarse-Grained Phase Space Graph, with $w = 3$, $\tau = 1$ and $P = \{0.2, 0.5, 0.9\}$. Each node has the same colour as the associated pattern.

3.2.4 Visibility Graphlets Network

In a Visibility Graphlets Network [51], we have a combination of two different mapping concepts: transition and visibility, as the local states are represented by visibility graphs, but the edges are defined by the transition probability between states. It is important to note that, although graphlets are mentioned in the name of this network, this method does not employ the traditional concept of graphlets used in network science, which refers to small, non-isomorphic, connected, induced subgraphs of a larger graph [39].

To construct a Visibility Graphlets Network, we start by incorporating the time series into a w -dimensional phase space, but without time delay ($\tau = 1$), to obtain a sliding window with size w across our time series. Each of the resulting vectors $\vec{z}_i = (y_i, y_{i+1}, \dots, y_{i+(w-1)})$, given $k = 1, 2, \dots, n - w + 1$, and n as the length of the time series, is then mapped to a directed visibility graph, according to the algorithm described in Subsection 3.1.1. This process results in a set of adjacency matrices $G = g_1, \dots, g_{n-w+1}$, each representing one of the generated visibility graphs.

The next step involves connecting these states to form a directed chain, linking each pair of successive states. This process creates a chain of directional states, represented as follows:

$$g_1 \rightarrow g_2 \rightarrow \dots \rightarrow g_{n-w+1}. \quad (3.21)$$

After establishing the initial connections between states, the next phase consists in finding all the distinguishable states by scanning G , comparing each of the adjacency matrices iteratively. If one adjacency matrix g_i is identical to other g_j , with $i < j$, the state g_j needs to be replaced with the state g_i , in the directional chain.

Finally, we define the visibility graphlets network, where the nodes represent unique graphlets generated from the embedding of the original time series. Directed and weighted edges are assigned between pairs of nodes, with the weight corresponding to the frequency of transitions between consecutive graphlets in the chain network. The motifs, graph and the process of how to obtain them is present in Figure 3.16.

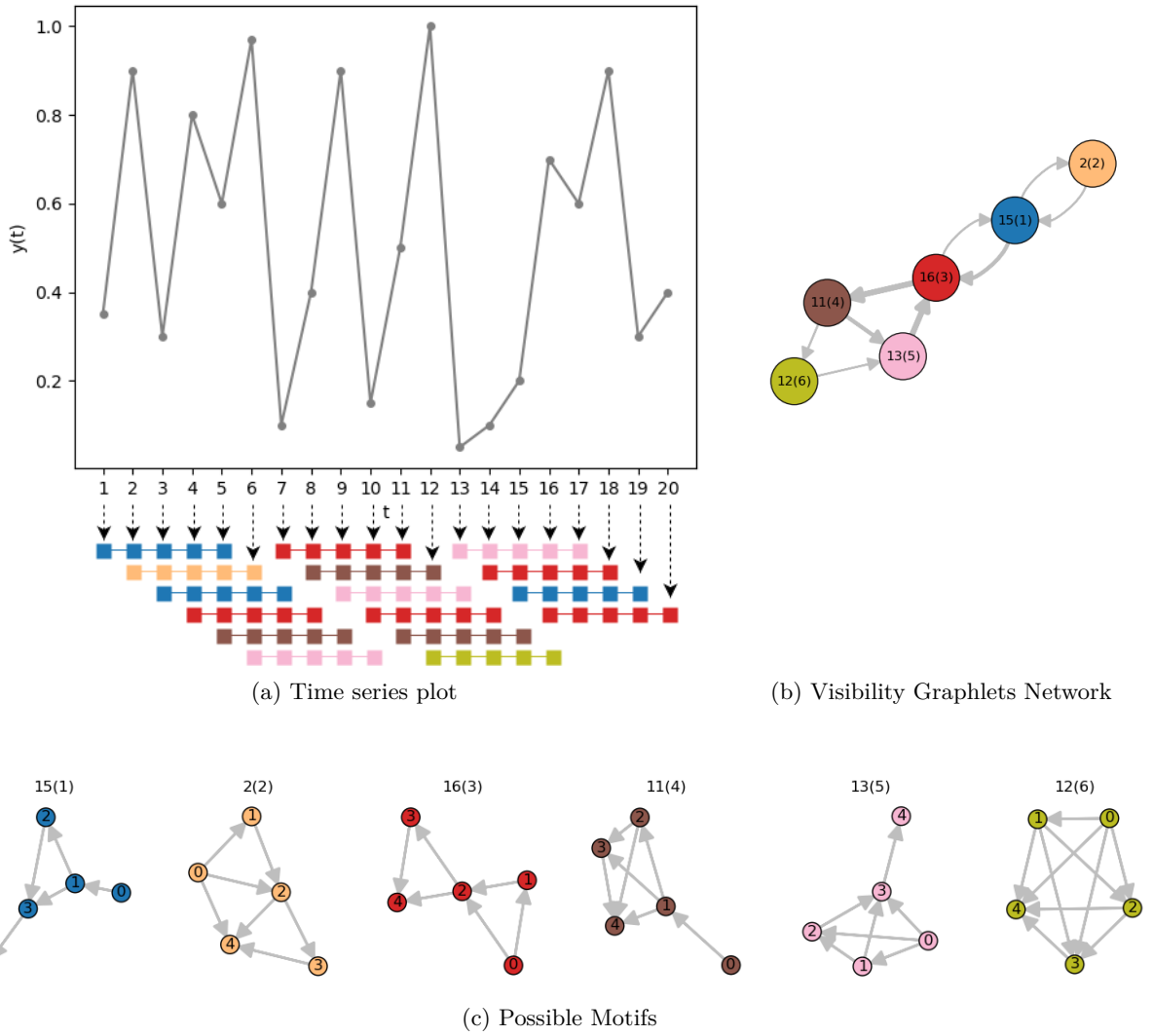


Figure 3.16: On the left, the time series plot, along the different embedded vectors, each colour-coded according to the motif identified. On the right, the correspondent Visibility Graphlets Network, with $w = 5$. Each node has the same colour as the associated motif. On the bottom, the motifs' graph representations.

3.3 Proximity Networks

This section focus on mapping methods based on the mutual proximity of different segments of a time series. These concept relies on distance or similarity measures to assign the edges between nodes, which, in turn, correspond to time series observations or states [44].

Proximity networks are mainly considered spatial networks, and are also invariant under vertex relabelling, as their topological characteristics remain unchanged despite permutation of vertices [17]. These networks are divided into *Cycle Networks*, *Correlation Networks*, and *Recurrence Networks*. The last ones include: *k-Nearest Neighbour Networks*, *Adaptive Nearest Neighbour Networks*, and ϵ -*Recurrence Networks*.

3.3.1 Cycle Network

Cycle Networks, first introduced by Zhang and co-authors [61, 63], are tailored to study the topological features of pseudo-periodic time series. To generate a cycle network, the first task is to segment the pseudo-periodic time series in question into x consecutive cycles $c_1, c_2, \dots, c_x$, according to the local minima or maxima. These cycles will serve as the nodes of our undirected network.

The association between pairs of nodes v_i and v_j is determined by the comparison to a threshold ϵ , by either an application of the correlation coefficient to cycles (of, possibly, different lengths):

$$\rho_{i,j} = \max_{l=0,1,\dots,l_j-l_i} \frac{\text{cov}(c_i[1:l_i], c_j[1+l:l_i+l])}{\sqrt{\text{var}(c_i[1:l_i])} \sqrt{\text{var}(c_j[1+l:l_i+l])}}, \quad (3.22)$$

where cov refers to the covariance and var to the variance, and l_i, l_j to the lengths of cycles c_i, c_j , respectively; or their phase space distance [16]:

$$d_{i,j} = \min_{l=0,1,\dots,l_j-l_i} \frac{1}{l_i} \sum_{k=1}^{l_i} \|c_i[k] - c_j[k+l]\|. \quad (3.23)$$

The cycle network method is illustrated in Figure 3.17.

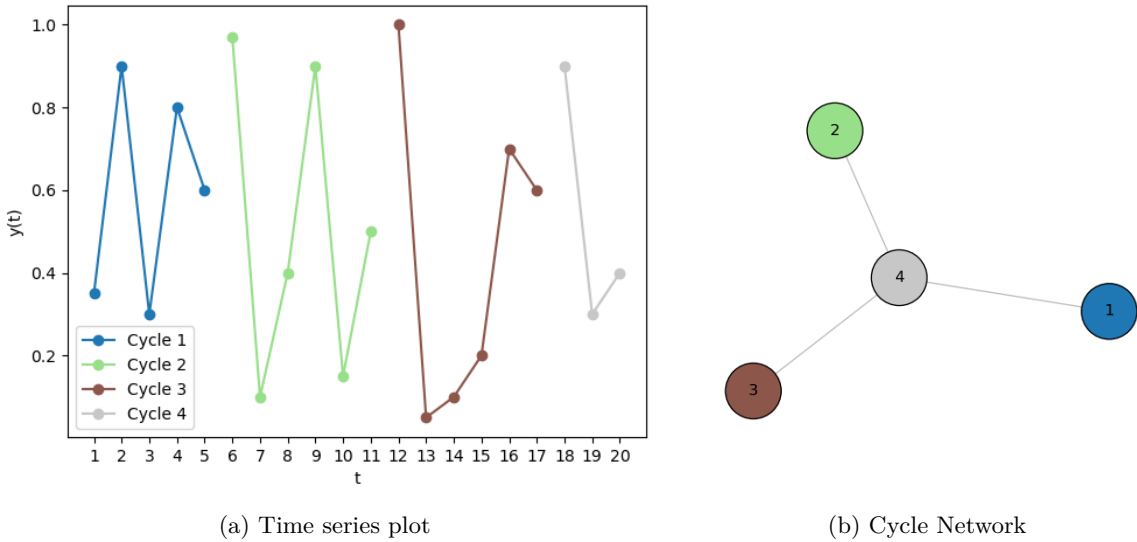


Figure 3.17: On the left, the time series plot, with each observation colour-coded according to the cycle it belongs to. On the right, the correspondent Cycle Network, with $\epsilon = 0.8$. Each node has the same colour as the associated cycle.

3.3.2 Correlation Network

An alternative to the use of correlation measures to cycles derived from pseudo-periodic time series, suggested by Yang and Yang [59], involves applying correlation measures directly to embeddings of a time series, without the need of identifying cycles.

Basically, the time series is reconstructed into a w -dimensional phase space, with a τ time delay associated, resulting in a set of vectors $\vec{z}_i$. We consider each of these vectors as the nodes in our Correlation network and we assign connectivity between a pair of nodes v_i and v_j , based on the computation of the Pearson correlation between the two correspondent vectors $\vec{z}_i$ and $\vec{z}_j$:

$$\rho_{ij} = \frac{\sum_{k=1}^w [\vec{z}_i[k] - \langle \vec{z}_i \rangle] \cdot [\vec{z}_j[k] - \langle \vec{z}_j \rangle]}{\sqrt{\sum_{k=1}^w [\vec{z}_i[k] - \langle \vec{z}_i \rangle]^2} \sqrt{\sum_{k=1}^w [\vec{z}_j[k] - \langle \vec{z}_j \rangle]^2}}, \quad (3.24)$$

where $\langle \vec{z}_i \rangle = \sum_{k=1}^w \vec{z}_i[k]/n$. Each result ρ_{ij} takes values between -1 and 1. After obtaining our correlation matrix, we can transform it to generate the adjacency matrix of the network, by applying the follow:

$$A_{ij} = \begin{cases} 1, & (|\rho_{ij}| \geq r_c) \\ 0, & (|\rho_{ij}| < r_c) \end{cases} \quad (3.25)$$

where r_c corresponds to a defined threshold. The choice of this parameter is critical to the structure of our generated network.

Figure 3.18 shows the correlation network mapping, according to the different embedded vectors found.

3.3.3 Recurrence Network

Recurrence Networks were introduced by Donner et al. [16] to address limitations of existing mapping methods in the study of dynamical (or chaotic) systems. The recurrence of states, where states return randomly close to previous ones after some time, is a fundamental characteristic of deterministic dynamical systems, particularly nonlinear or chaotic systems [69].

The algorithm for the Recurrence Networks starts with the reconstruction of phase space, so we have our initial states $\vec{z}_i$, that will serve as the nodes on our network, as we have seen in other types of networks before.

The second step, generally, involves using the determined $\vec{z}_i$ vectors to construct a recurrence matrix R , where each pair of elements i, j has a value defined by a proximity measure, which can be any norm in phase space (for example, Euclidean, Manhattan, or maximum norm). Then, the recurrence matrix R is converted, using certain criteria, into a binary matrix, which will be the adjacency matrix A of our unweighted and undirected complex network. The process is illustrated in Algorithm 6.

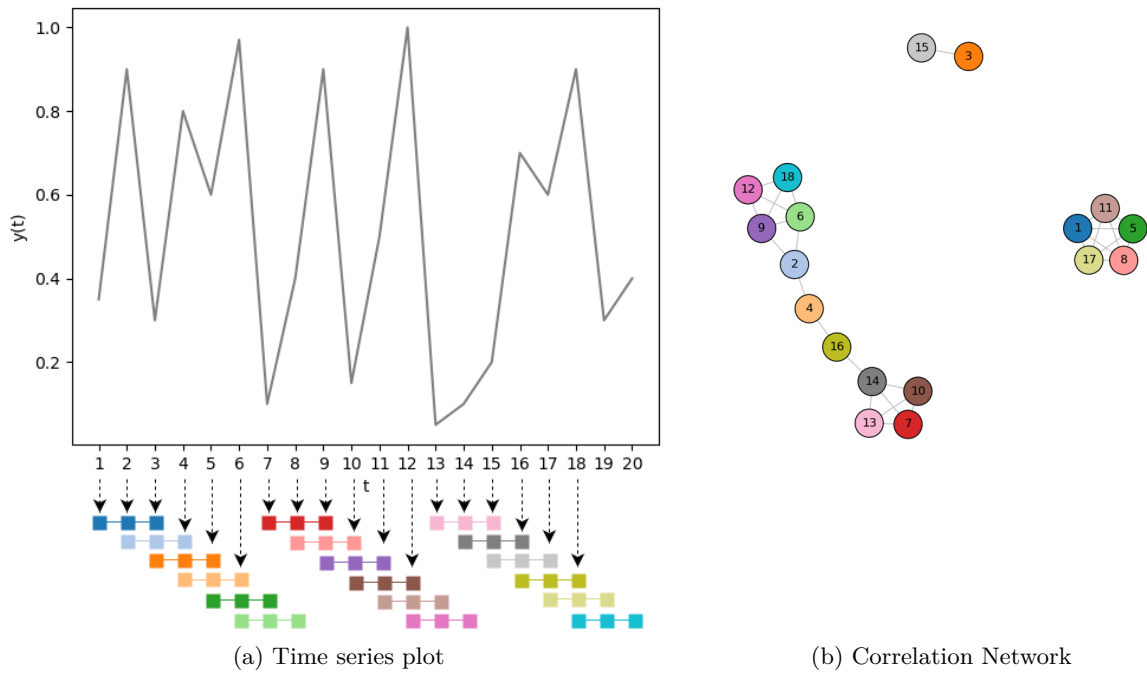


Figure 3.18: On the left, the time series plot, along the differently coloured embedded vectors. On the right, the correspondent Correlation Network, with $w = 3$, $\tau = 1$, and $\rho = 0.8$. Each node has the same colour as the associated vector.

Algorithm 6: Algorithm for Recurrence Network

```

1 function RecurrenceNetwork( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}, dim, \tau, \epsilon$ ):
2    $embedded\_ts \leftarrow \text{EmbedTimeSeries}(ts, dim, \tau)$ ;
3    $L \leftarrow \text{length of } embedded\_ts$ ;
4    $V \leftarrow \{1, \dots, L\}$ ;
5    $E \leftarrow \emptyset$ ;
6    $R \leftarrow$  recurrence matrix of size  $(L, L)$ ; ▷ Obtained using a specific criteria
7   for  $i$  from 0 to  $L - 1$  do
8     for  $j$  from 0 to  $L - 1$  do
9       if  $R[i][j] \geq \epsilon$  then
10        add edge( $i + 1, j + 1$ ) to  $E$ ;
11   return  $G = (V, E)$ ;
12 function EmbedTimeSeries( $ts, dim, \tau$ ):
13    $n \leftarrow \text{length of } ts$ ;
14    $embedded\_ts \leftarrow \emptyset$ ;
15   for  $i$  from 1 to  $n - (dim - 1) \cdot \tau$  do
16      $embedding \leftarrow \emptyset$ ;
17     for  $d$  from 1 to  $dim$  do
18        $embedding.append(y_{i+d \cdot \tau})$ ;
19      $embedded\_ts.append(embedding)$ ;
20   return  $embedded\_ts$ ;

```

We can see the temporal complexity of the method is dependent on the for-loops of lines 7-13, which iterate through the recurrence matrix R in $\mathcal{O}(L^2)$ time, and on the `EmbedTimeSeries` function, which has complexity $\mathcal{O}(L \cdot \dim)$. The criteria to obtain the recurrence matrix is also preponderant for the overall complexity of the algorithm, as different methods and norms may have different complexities associated.

Recurrence Networks methods can be further divided into 3 main types: k -Nearest Neighbour Network, Adaptive Nearest Neighbour Network and ϵ -Recurrence Network [17]. We will now explore the definitions and temporal complexities differences among the methods.

k -Nearest Neighbour Network (NNN) [49]: Each node is connected to the other k nodes that have the shortest mutual distances. It's important to know that if v_j is one of the closest points to v_i , the opposite may not be true. This results in a directed network, where all nodes have the same out-degree equal to k [44]. The algorithm for obtaining a recurrence matrix involves pairwise distance calculations and sorting to find the nearest neighbours for each node, which results in $\mathcal{O}(L^2 \cdot \dim + L^2 \cdot \log L)$.

Adaptive Nearest Neighbour Network (ANNN) [49, 56]: To address the limitation that all nodes have the same out-degree ($= k$) in the k -NNN, a new version, ANNN, was proposed. Here, if v_i is one of the closest neighbours of v_j , then v_j is also considered a neighbour of v_i , even if v_j is not one of the k -closest neighbours, according to the k -NNN algorithm. This results in a symmetric adjacency matrix and variable node degrees [44]. To obtain the recurrence matrix, the algorithm is similar to the k -NNN case, running also with complexity $\mathcal{O}(L^2 \cdot \dim + L^2 \cdot \log L)$.

ϵ -Recurrence Network [17]: Instead of limiting edges by imposing a constraint on the number, based on the neighbour concept, in the ϵ -Recurrence Network, a distance threshold is defined. This leads to a connection of two nodes v_i and v_j , but only if $d_{ij} \leq \epsilon$, being d_{ij} the proximity distance defined between the two associated vectors $\vec{z}_i$ and $\vec{z}_j$. In this case, after the pairwise distance calculation, we do not need sorting, but only a comparison to ϵ , which results in $\mathcal{O}(L^2 \cdot \dim)$ runtime.

Given the different possible complexities according to the recurrence criteria used and the considerations about Algorithm 6, we can see, in the worst case, a Recurrence Network runs in $\mathcal{O}(L^2 \cdot (1 + \dim + \log L))$. An example of an ϵ -Recurrence Network can be found in Figure 3.19.

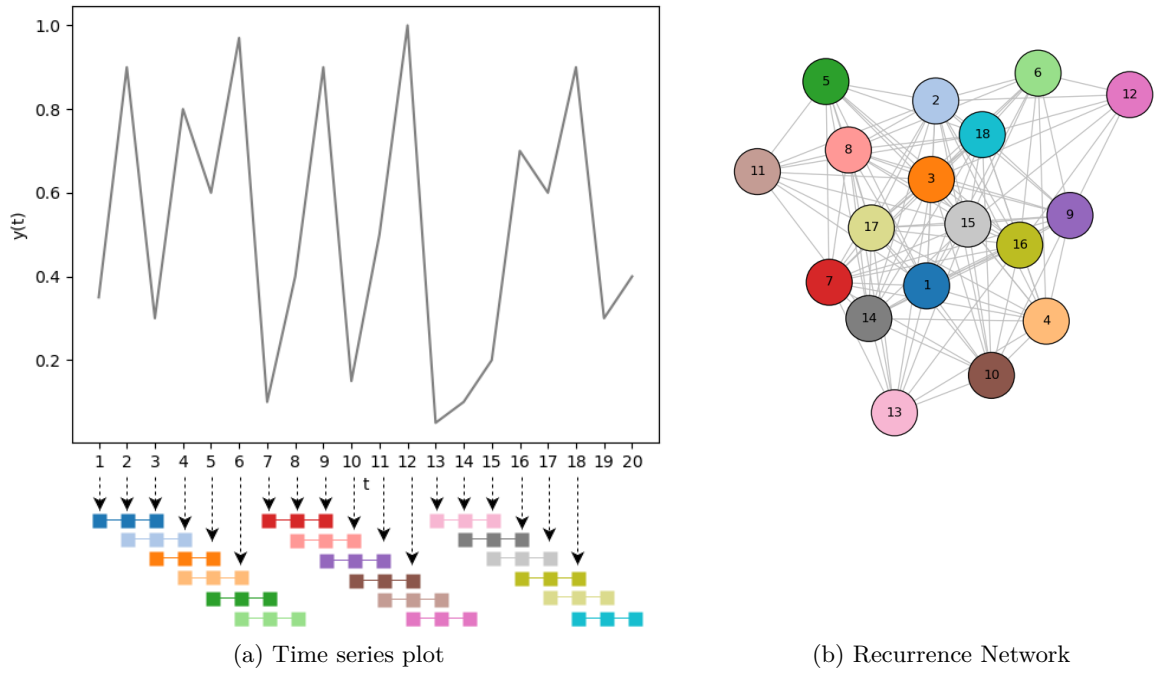


Figure 3.19: On the left, the time series plot, along the differently coloured embedded vectors. On the right, the correspondent Recurrence Network, with $w = 3$, $\tau = 1$ and $\epsilon = 0.8$. Each node has the same colour as the associated vector.

Chapter 4

Multivariate Mapping Methods

In the previous chapter, we focused on the description of mapping methods applied to univariate time series. However, the need for multivariate analysis tools has increased, leading to the emergence of more multivariate mapping methods.

In this chapter, we briefly explain the existing multivariate time series mapping methods, critical to the analysis of multivariate time series. We can divide the different multivariate methods into two categories, according to the generated network structure: Single Layer and Multiple Layers.

4.1 Single Layer Networks

In a single layer network representing multivariate time series data, we can have different multivariate time series components assigned to different nodes, or even nodes representing time observations or phase space vectors, like in the univariate setting. Edges are usually represented based on a certain relationship between the nodes, computed as correlation, for example.

There are different methods in this category. In this chapter, we only focus on the description of the Correlation Network, as proposed by Eguiluz et al. [19], as this was the only multivariate single layer method we were able to implement in our framework. Other notable methods include:

- Long-Run Variance Decomposition Network [13];
- Causal Effect Network [41];
- Ordinal Partition Transition Network [40, 62];
- Pattern Interdependent Network [38];
- Inter-System Recurrence Network [22];
- Joint Recurrence Network [23].

We recognise that the methods listed above are also valuable and, so, we have included them in our future work.

4.1.1 Correlation Network

The multivariate version of a Correlation Network [19] follows some of the same principles of the univariate counterpart (Subsection 3.3.2): it applies the same correlation measure and compares the results to a threshold to decide which edges to assign in the network. Here, instead of the nodes being the embedded vectors from the phase-space, they correspond to one of the components $y_{\alpha,t}$ of the multivariate time series.

Thus, a Correlation Network corresponds to an undirected and weighted graph $G = (V(G), E(G))$, such that $V(G) = \{Y_{i,t}\}_{i=1}^m$ and $E(G) = \{(v_i, v_j, w_{i,j}) \mid (v_i, v_j) \in V(G) \wedge w_{i,j} = \rho_{i,j}(0) \wedge i \neq j\}$. The inter-dependencies between the m time series components are represented by the (contemporaneous) correlation [44].

We can see the result for a bivariate time series in Figure 4.1.

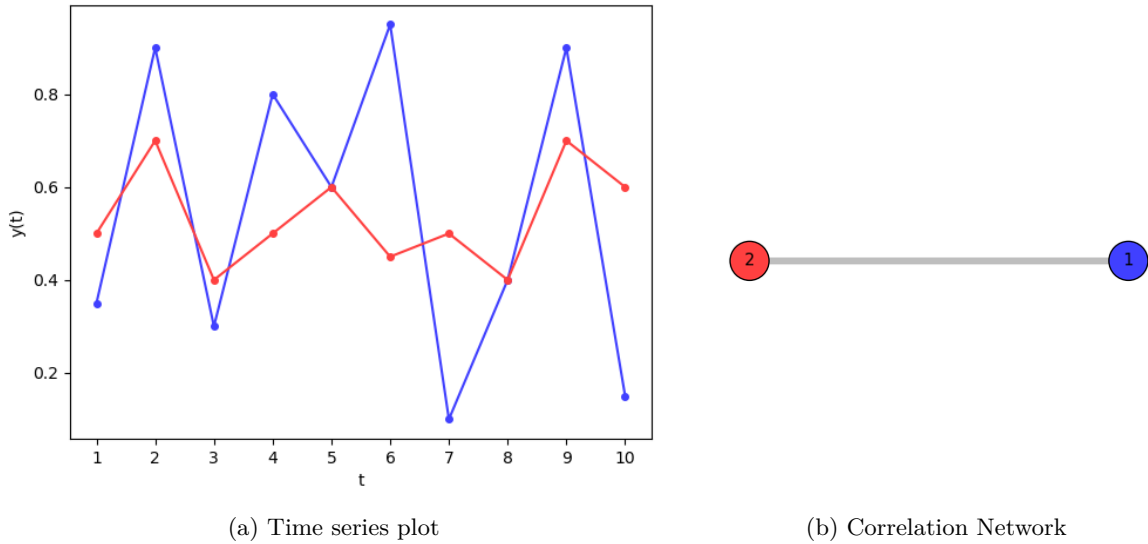


Figure 4.1: On the left, the time series plot. On the right, the correspondent Correlation Network, with $\rho = 0.2$.

4.2 Multiple Layers Networks

To map multivariate time series into a multi-layer network structure, each layer is constructed from an individual component of the multivariate time series data. To establish connections between these layers (i.e., inter-layer edges) we have two primary approaches.

The first approach involves creating multiplex networks. In this method, inter-layer edges are assigned categorically, meaning that nodes in one layer are connected to corresponding nodes in the other layers. This categorical approach facilitates a straightforward representation of how layers are connected.

Alternatively, we can use established visibility and transition concepts to map interactions between each pair of layers, leading to Multilayer Horizontal Visibility Graphs and Multilayer Quantile Graphs. By focusing on the relationships between pairs of layers, this approach provides a deeper insight into how the time series components interact across different layers.

4.2.1 Multiplex Visibility Network

In a m -layer Multiplex Visibility Network [32], each layer α is associated to a NVG, built from a time series component $y_{\alpha,t}$. Thus, intra-layer edges are defined by the univariate visibility algorithm, described in Section 3.1.1.

All m layers contain the same nodes, which represent the time series timings t . The inter-layer edges connect the same nodes across adjacent layers, i.e., they connect each pair v_i^α and $v_i^{\alpha+1}$, given $1 \leq i \leq n$ and $1 \leq \alpha \leq m - 1$. Figure 4.2 illustrates this method's result.

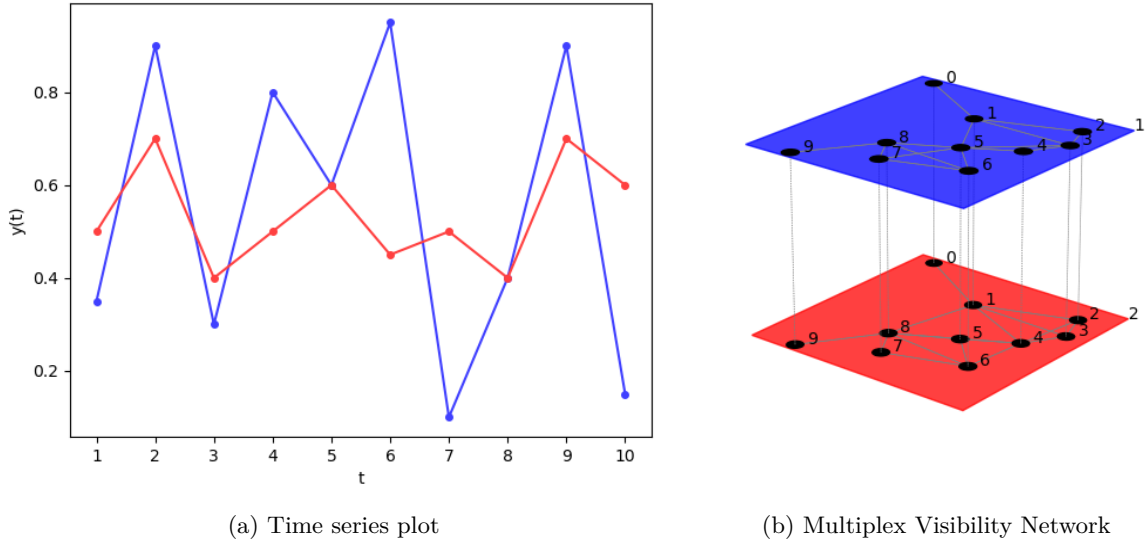


Figure 4.2: On the left, the time series plot. On the right, the correspondent Multiplex Visibility Network.

4.2.2 Multiplex Recurrence Network

The Multiplex Recurrence Network [20] follows a similar idea to the Multiplex Visibility Network, but using another base algorithm: m layers are constructed by the Recurrence Networks, explained in Section 3.3.3, built from the multivariate time series components $\{y_{\alpha,t}\}_{\alpha=1}^m$. The layers are also connected to each other by linking equivalent time labelled nodes, as seen in Figure 4.3.

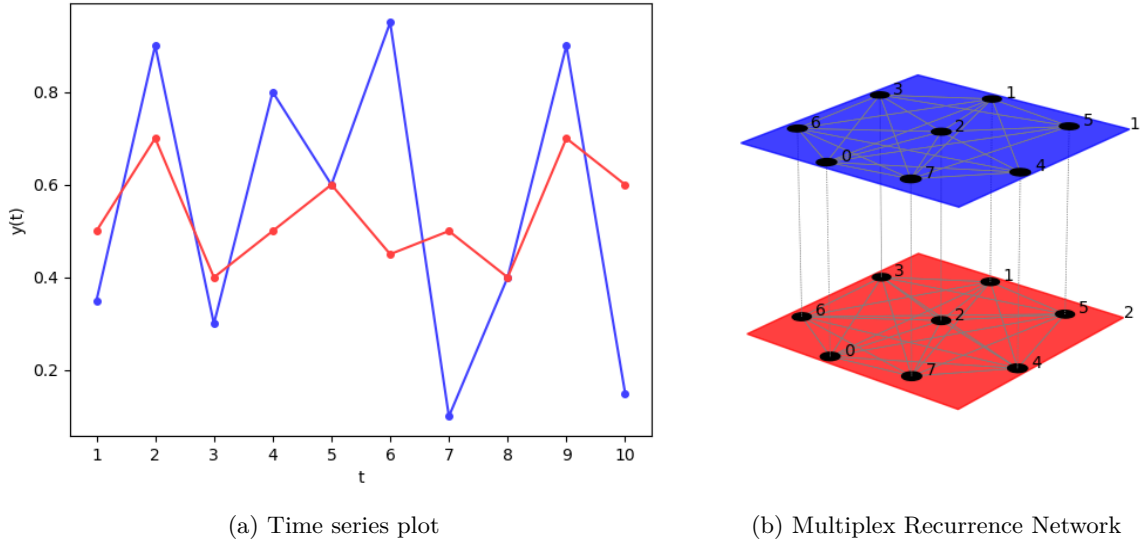


Figure 4.3: On the left, the time series plot. On the right, the correspondent Multiplex Recurrence Network.

4.2.3 Multilayer Horizontal Visibility Graph

While in the previous multilayer methods, the inter-layers corresponded, simply, to connections between equal nodes, the Multilayer Horizontal Visibility Graph (MHVG) [46] presents a new approach to assign edges across layers: the cross-horizontal visibility, inspired by the horizontal visibility concept (Section 3.1.2).

Considering two time series components $y_{\alpha,t}$ and $y_{\beta,t}$, two elements (t_i, y_{α,t_i}) and (t_j, y_{β,t_j}) present cross-horizontal visibility, if and only if:

$$y_{\alpha,t_i} \wedge y_{\beta,t_j} > \max(y_{\alpha,t}, y_{\beta,t}), \quad \forall t: \quad t_i < t < t_j, \quad i \neq j, \quad (4.1)$$

given $i, j = 1, \dots, n$, being n the length of the time series. As the definition implies, all time series components need to be in the same length.

In conclusion, using HVGs mapped from the time series components $\{y_{\alpha,t}\}_{\alpha=1}^m$ to build the m -layers, and using the cross-horizontal visibility method to assign the edges between each pair of layers, we get a MHVG, as shown in Figure 4.4.

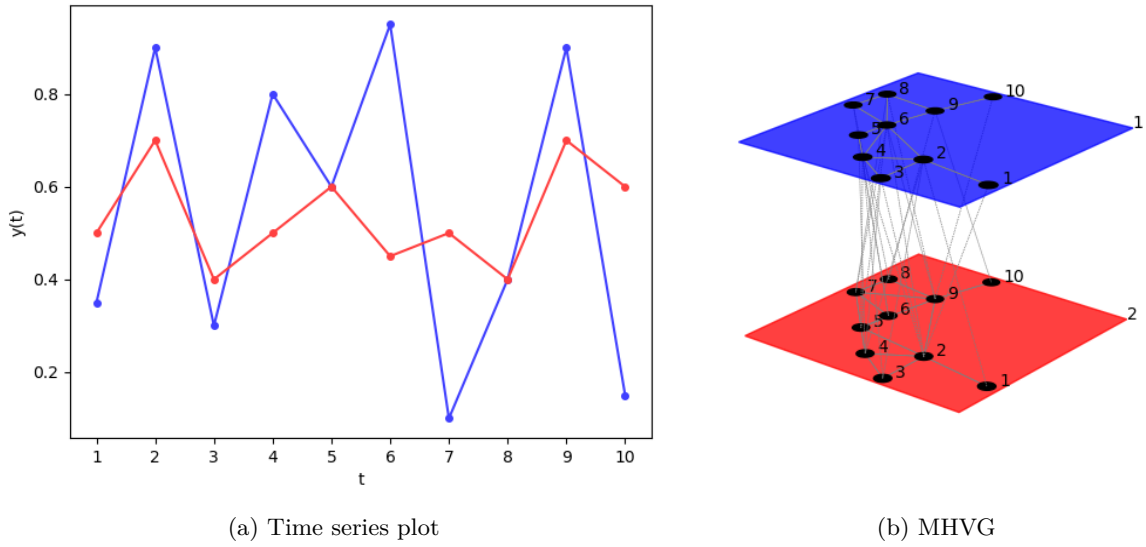


Figure 4.4: On the left, the time series plot. On the right, the correspondent MHVG.

Algorithm 7 presents the method to build the MHVG, while Algorithm 8 describes the calculation of the cross-horizontal visibility inter-layer edges, used in Algorithm 7. The cross-horizontal visibility function has a time complexity of $\mathcal{O}(n^2)$, determined by the for-loops in lines 2-13. This implies that Algorithm 7 has a complexity of $\mathcal{O}(m^2 \cdot n^2)$, as the second for-loop (lines 9-14) calls the **CrossHV** function for each pair of layers. The first for-loop (lines 4-8) does not influence the overall complexity, since it calls the HVG algorithm m times.

Algorithm 7: Algorithm for MHVG

```

1 function MHVG( $mts = \{((t_{1,1}, y_{1,1}), \dots, (t_{1,n}, y_{1,n})), \dots, ((t_{m,1}, y_{m,1}), \dots, (t_{m,n}, y_{m,n}))\}$ ):
2    $G \leftarrow \{\}$ ;
3    $C \leftarrow \{\}$ ;
4    $n\_mts \leftarrow \{\}$ ; ▷ List to store the rescaled time series components
5   for  $\alpha$  from 1 to  $m$  do
6      $layer_\alpha \leftarrow \text{HorizontalVisibilityGraph}(mts_\alpha = (t_{\alpha,1}, y_{\alpha,1}), \dots, (t_{\alpha,n}, y_{\alpha,n}))$ ;
7     add  $layer_\alpha$  to  $G$ ;
8      $n\_mts[\alpha] \leftarrow \text{MinMax}(mts[\alpha])$ ; ▷ Rescale time series  $y_\alpha$ 
9   for  $\alpha$  from 1 to  $m - 1$  do
10    for  $\beta$  from  $\alpha + 1$  to  $m$  do
11       $tsMax \leftarrow \text{MaxTS}(n\_mts[\alpha], n\_mts[\beta])$ ; ▷ Get the maximum rescaled time series
12       $\text{CrossHV}(n\_mts[\alpha], n\_mts[\beta], tsMax, G[\alpha], G[\beta])$ ; ▷ Map the pairwise time series  $y_\alpha$  and  $y_\beta$  on Cross-HV
13  return  $G$ ;

```

Algorithm 8: Cross-Horizontal Visibility Function

```

1 function CrossHV( $tsA$ ,  $tsB$ ,  $tsMax$ ,  $layerA$ ,  $layerB$ ):
2   foreach node  $i$  in  $layerA.V$  do
3     foreach node  $j$  from  $i + 1$  to  $n$  in  $layerB.V$  do
4       if node  $i$  can 'see'  $j$  then ▷ According to the MHVG criteria
5         add edge( $i, j, layerA, layerB$ ) to  $C$ ; ▷ Add inter-layer edge ( $v_{\alpha,i}, v_{\beta,j}$ )
6     foreach node  $j$  from  $i - 1$  to  $0$  in  $layerB.V$  do
7       if node  $i$  can 'see'  $j$  then
8         add edge( $i, j, layerA, layerB$ ) to  $C$ ;
9   return

```

4.2.4 Multilayer Quantile Graph

Another extension of an univariate method to a multivariate and multilayer setting is the Multilayer Quantile Graph [47]. Like the other multilayer methods before, each layer represents a graph generated from a univariate mapping method, in this case, the QG, with the nodes in each layer corresponding to the quantiles defined.

The connections between nodes across layers are made according to the contemporaneous quantiles concept: the quantile at each observation at time t in the α component, links to the quantile at the same observation at time t in the β component.

For each layer α , there are Q different quantiles defined. Formally, let $q_i^\alpha(t_k)$ be the quantile assigned to y_k^α , and $q_j^\beta(t_k)$ the quantile representing the observation at time t_k , for the β layer, y_k^β . Then, there is an inter-layer link between v_i^α and v_j^β , and the associated weight w_{ij} corresponds to the frequency of contemporaneous quantiles, for all observations t . Figure 4.5 illustrates the process.

Algorithms 9 and 10 use the concepts just explained to build the Multilayer Quantile Graph (MQG). The complexity of the first for-loop in Algorithms 9 (lines 3-6) is $\mathcal{O}(m \cdot n \log n)$, as it calls the QG algorithm m times, which has a complexity of $\mathcal{O}(n \log n)$ (see Section 5.3.3.1). The second for-loop (lines 7-11) iterates through each pair of layers, computing the contemporaneous quantile function (Algorithms 10), which runs in $\mathcal{O}(n)$ time. Thus, the overall complexity for the second loop is $\mathcal{O}(m^2 \cdot n)$, leading to a total complexity for Algorithm 9 of $\mathcal{O}(m \cdot n \log n + m^2 \cdot n)$.

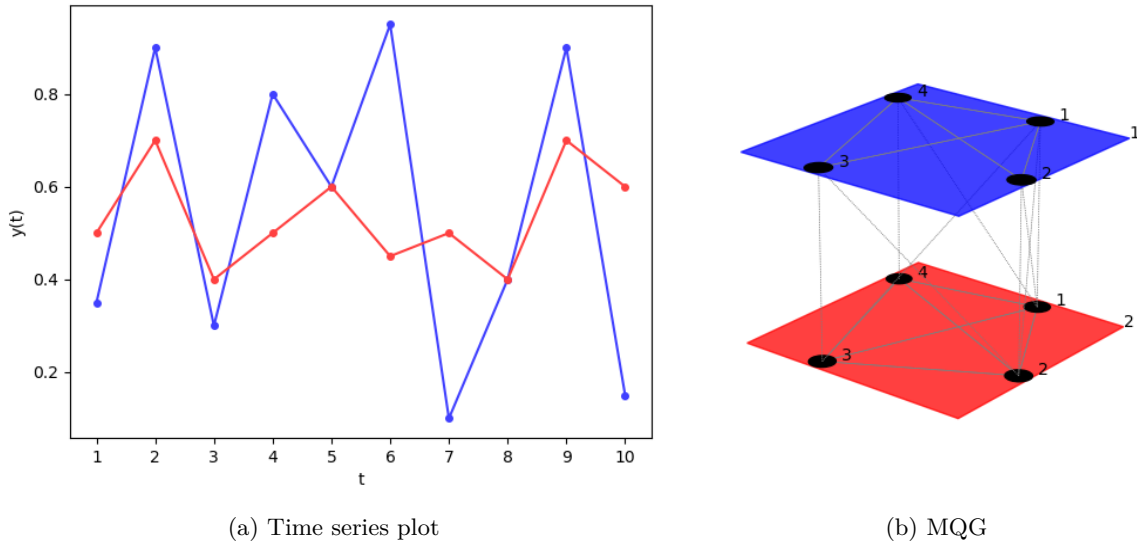


Figure 4.5: On the left, the time series plot. On the right, the correspondent MQG, with $Q = 4$.

Algorithm 9: Multilayer Quantile Graph (MQG)

```

1 function
  MQG( $mts = \{((t_{1,1}, y_{1,1}), \dots, (t_{1,n}, y_{1,n})), \dots, ((t_{m,1}, y_{m,1}), \dots, (t_{m,n}, y_{m,n}))\}, Q$ ):
2    $G \leftarrow \{\}$ ;
3    $C \leftarrow \{\}$ ;
4   for  $\alpha$  from 1 to  $m$  do
5      $layer_\alpha \leftarrow \text{QuantileGraph}(mts_\alpha = (t_{\alpha,1}, y_{\alpha,1}), \dots, (t_{\alpha,n}, y_{\alpha,n}), Q)$ ;
6     add  $layer_\alpha$  to  $G$ ;
7   for  $\alpha$  from 1 to  $m - 1$  do
8     for  $\beta$  from  $\alpha + 1$  to  $m$  do
9        $\text{CONTEMP\_Q}(G[\alpha], G[\beta])$ ;
10  return ( $G, C$ )

```

Algorithm 10: Contemporaneous Quantiles

```

1 function CONTEMP_Q( $layerA, layerB$ ):
2    $qA \leftarrow layerA.q\_sequence$ ;
3    $qB \leftarrow layerB.q\_sequence$ ;
4   for  $i$  from 1 to  $n$  do
5      $e \leftarrow \text{edge}(qA[i], qB[i], layerA, layerB)$ ;
6     if not  $e$  then
7       add  $\text{edge}(qA[i], qB[i], layerA, layerB, 1)$  to  $C$ ;
8     else
9        $w \leftarrow \text{get weight from } e$ ;
10      set  $\text{edge } e$  weight to  $w + 1$ ;
11  return

```

Chapter 5

Software Development

After considering the theoretical foundation of the various mapping methods, the next step was to transform these constructs into applicable tools, following the goals referenced in Section 1.2.

In this chapter, we present the key decisions made throughout the development process of our package, `pyTSMnet`, including the choice of programming language, overall structure, and the various functionalities provided by the framework. We also further explain how each of the mapping methods were implemented as a class, taking into account possible parameters while referencing external libraries, when used. Additionally, we show the pseudo-code created and analyse the respective time complexity. Finally, we present the supporting functions implemented, as well as utilities.

5.1 Programming Language

We decided to use Python to build our framework, due to the fact that language is user-friendly and highly popular among researchers of different skill levels. Furthermore, the wide availability of other useful Python packages that help solve tasks in data science, statistics, and network analysis helps in the goal of unifying existent software [42]. The author’s familiarity with the language also cemented this decision.

Specifically, we utilised Python 3.11 to ensure compatibility with other existing libraries, such as `ts2vg`, which we incorporated into our package.

5.2 Package Structure

We adopted an object-oriented logic for the implementation of our package and mapping algorithms, ensuring that each type of network is encapsulated within its own dedicated class. This design choice promotes code re-usability across different modules and establishes hierarchical relationships between classes. Moreover, it leads to a more clean, orderly and flexible interface [37].

To organise our package effectively, we drew inspiration from the taxonomy depicted in Figure 2.3. This structure ensures that users can easily locate the desired mapping method, by following an intuitive order. Our package is divided into 20 sub-modules, as illustrated in Figure 5.1. This organisation allows users to navigate through different categories such as univariate and multivariate methods, each containing specific techniques like visibility, transition, and proximity, providing a clear and intuitive framework for accessing our software’s capabilities.

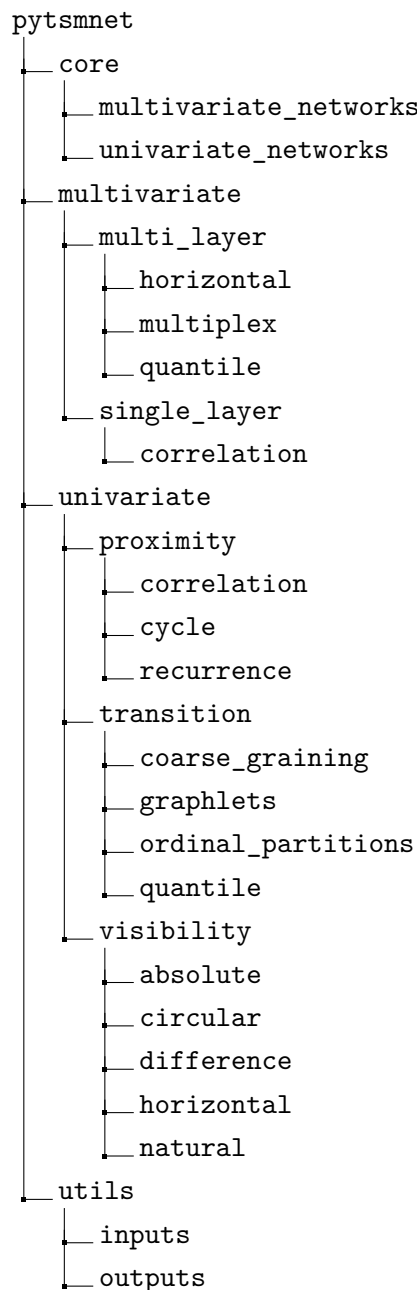


Figure 5.1: A directory tree structure, representing all modules and sub-modules present in the pyTSMnet package.

We can see the four main folders: `core`, `univariate`, `multivariate` and `utils`. As the name suggests, the `univariate` and `multivariate` modules contain all classes corresponding to the specific time series networks described in Chapters 3 and 4, respectively. Here, each sub-module refers to a mapping concept and may contain multiple classes, each representing variations of time series networks based on the specific mapping method.

The `core` module consists of various sub-modules that provide essential foundational classes supporting the aforementioned modules, while the `utils` module offers sub-modules for supplementary functions, primarily related to input and output operations.

The following sections provide a more detailed description of each module and corresponding sub-modules.

5.3 Modules and Classes

Figure 5.2 illustrates the general structure followed by each of the *Network Classes* in the `univariate` and `multivariate` modules. Each class begins by inheriting properties and functions from a parent class, corresponding to a class from the `core` module. It adds more attributes specific to the corresponding mapping method and includes functions to handle errors related to the parameters used to obtain these class attributes. A key function is the `build` function, which leverages auxiliary functions to populate the previous initialised network instance according to the defined mapping approach. If `build` is not called, the network remains empty.

Each of these network classes may add additional feature extraction or visualisation functions, or even overwrite the original inherited from the parent class. For more details on these supplementary functions, see below in Section 5.4.

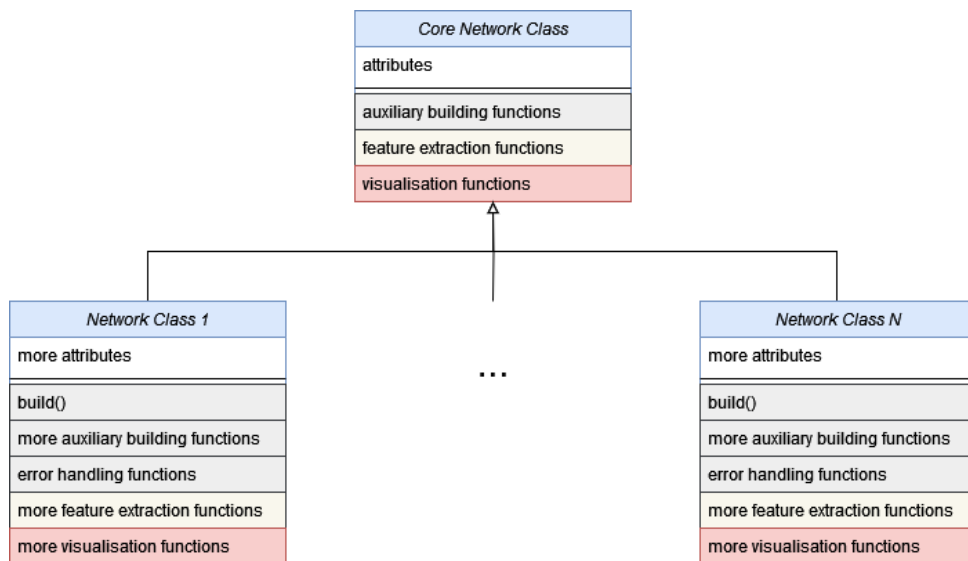


Figure 5.2: Overview of the package's inheritance process.

Several general principles were applied while designing our package classes: error handling strategies were implemented for parameters of type *string*, and NumPy [36] data structures were used wherever possible, given its optimised performance for numerical computations, especially with large datasets. Additionally, the use of vectorisation, which eliminates the need for explicit loops, enhances both performance and code readability.

In this section, we will proceed to detail the goal and composition of each module referenced in Figure 5.1, following the order established in the previous chapters. For each mapping algorithm, we present the initialisation parameters and the description of the class development process, as well as the correspondent pseudo-code for the build mapping algorithms and the respective time complexity, whether implemented from scratch or not available in the literature. Finally, in Section 5.3.7, we provide a summary table of time complexities for all mapping methods and clarify the nature of the mapping implementations, that is, if a method is implemented by scratch or if our implementation reuses other libraries or algorithms presented in the literature.

5.3.1 core module

The `core` module defines three base classes (see Figure 5.3) representing different types of networks based on the time series nature and graph structure: `Network` class, for univariate time series and available in the sub-module `univariate_network`, and `SinglelayerNetwork` and `MultilayerNetwork` classes, for multivariate data and available in the sub-module `multivariate_networks`.

Each of these classes has an associated graph/network structure (defined as attribute G), which is imported from other libraries. This approach leverages the efficiency, robustness and extensive features of well-established graph libraries like the Python interface for `igraph` [12] and the `pymnet` [35] packages. By utilising these packages, we can ensure high performance and reliability in graph operations, as well as access to a wide range of pre-built functionalities for visualisation, analysis, and manipulation of complex networks.

Network. Initialises with parameters `ts`, `xs` (optional), `weighted` (default *False*) and `directed` (default *False*). The `ts` parameter correspond to the y_t values of a time series $\{(t_1, y_1), \dots, (t_n, y_n)\}$. Optionally, we can provide the `xs` parameter, corresponding to the timestamps of the data (defined as attribute t). Note that `xs` needs to be of the same length as `ts`. If not provided, the class works with the default $t = [1, \dots, T]$. The attribute t is especially relevant to VG methods.

The `ts` and `xs` parameters are stored as NumPy arrays. We also allow flexibility by allowing the graph to be weighted or directed, if the corresponding boolean parameters are set to *True*.

The data structure used to store the created graph, G , is the `igraph Graph`. We also considered using the `NetworkX` graph structure [27], but ultimately decided on `igraph` due to its core library being written in C, often making it faster than `NetworkX`, which is purely Python-based.

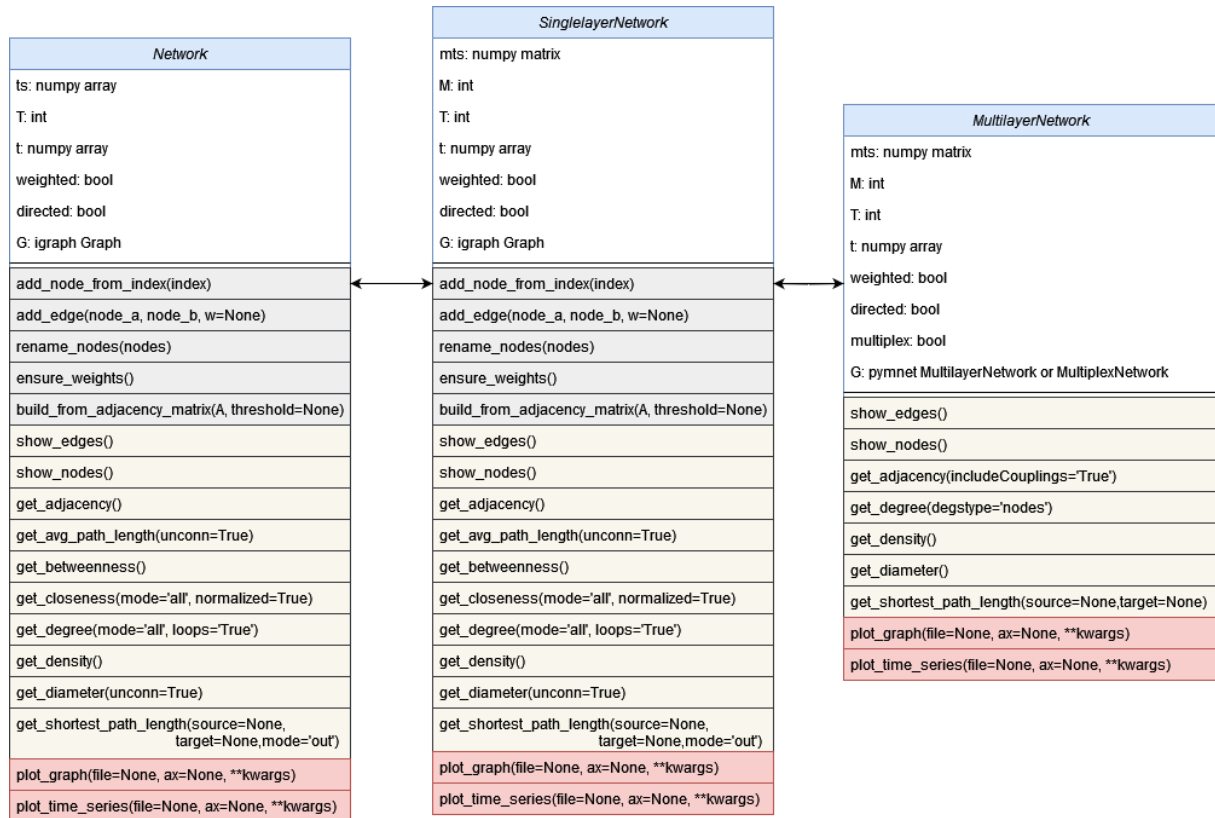


Figure 5.3: Overview of all core module classes: `Network`, `SinglelayerNetwork`, `MultilayerNetwork`.

Next, we will enumerate and explain the usability and versatility of the auxiliary building functions used in `Network` classes:

- `add_node_from_index`: simply adds t_i as a node, given an index i .
- `add_edge`: a very important function, as it is used in all of the children classes. It expands on the original add edge function of `igraph`, as it checks if the two nodes are already added and if the edge between the two already exists. Furthermore, it adapts the edge attributes of the network to the class parameters `weighted` and `directed`.
- `rename_nodes`: useful when working with other libraries to build our graph, as they may not allow to use a parameter similar to `xs`. It can also be called by the user, with a fitting list of node names.
- `ensure_weights`: ensures the graph has a `'weight'` attribute. If not, create it with all weights equal to 1. It is useful when calling other libraries' methods, that may not use the same attributes used in our package.
- `build_from_adjacency_matrix`: useful function, especially in the proximity-based networks, as we first calculate a distance matrix A , which we might next need to compare to a *threshold* value to obtain the graph edges.

Every class in the `univariate` module is a child class of `Network` and also uses the parameter `ts` for initialisation. In particular, the classes in the `visibility` sub-module also use the parameter `xs` for initialisation, as mentioned before.

SinglelayerNetwork. Similar to the `Network` class in all implementation aspects, except that it stores a list of time series values, `ts`, as a NumPy matrix, denoted as `mts` parameter, to accommodate multivariate time series data. It also includes a parameter `M`, which corresponds to the number of time series components. The class retains the same graph structure (that is, `igraph Graph`) and associated functions, as the objective is to obtain a single layer network. All classes in `multivariate/single_layer` are children of `SinglelayerNetwork` and are initialised with the `mts` parameter.

MultilayerNetwork. The initialisation is similar to `SinglelayerNetwork`, given that both use the `mts` parameter and were designed to handle multivariate time series data. The key difference lies in the graph structure: since a multilayer networks is required, we decided to use the `pymnet` library. In this case, if the `multiplex` parameter (default `False`) is set to `True`, the associated graph G is an instance of the `pymnet` class `MultiplexNetwork`; otherwise, G is of type `MultilayerNetwork`. All classes in `multivariate/multi_layer` inherit from the `MultilayerNetwork` class and are initialised with the `mts` parameter.

5.3.2 univariate/visibility module

The `univariate/visibility` module encompasses all classes that generate networks from univariate time series data using the visibility algorithms described in Section 3.1. Whenever possible, we tried to use the `ts2vg` package [3] for our implementations, due to its completeness in terms of possible natural and horizontal visibility graphs, especially because of the wide array of customisable arguments.

In following sections, we describe the sub-modules found inside the `univariate/visibility` module, as well as their respective classes.

5.3.2.1 natural sub-module

The `natural` sub-module provides two classes: `NaturalVisibilityGraph` and `DualPerspectiveVisibilityGraph`. The first class corresponds to the NVG method, referenced in Section 3.1.1, with several personalised parameters available, and uses the `ts2vg` [3] library. The second corresponds to the WDPVG method with optional weights, and uses the available `PyIOmica` [14] implementation for creating the dual perspective visibility graphs.

NaturalVisibilityGraph. Class designed to create the NVG. This class has the following parameters:

- **ts:** the time series values.
- **xs:** the time series timings. Default is `None`.
- **direction:** the orientation of the direction used for the edges. If not provided, an undirected graph will be returned. Otherwise, a directed graph will be created using one of the following options from `ts2vg`: `left_to_right` or `top_to_bottom`. Default is `None`.
- **weight:** the required weigh. If not provided, an unweighted graph is returned. Otherwise, a weighted graph will be created, using one of the available options of `ts2vg`: `distance`, `sq_distance`, `v_distance`, `abs_v_distance`, `h_distance`, `abs_h_distance`, `slope`, `abs_slope`, `angle`, `abs_angle`, `num_penetrations`. Default is `None`.
- **min_weight:** a threshold value for the weight of the edges. If provided, only edges with a weight higher than the argument value (non inclusive) will be included in the final graph. The graph must be weighted and the values used for weight will depend on the `weighted` parameter. This acts as a generalisation of parametric visibility graphs. Default is `None`.
- **max_weight:** similar to `min_weight`, except only edges lower than `max_weight` given parameter are included in the graph. Default is `None`.
- **penetrable_limit:** a integer value. If larger than 0, a LPVG is made. The value for `penetrable_limit` indicates the maximum number of data points that are allowed to obstruct the visibility between two connected nodes in the final graph. Default is 0.

The `ts2vg` package uses the DC approach (see Algorithm 2) for the NVG mapping method but provides five additional parameters (besides `xs`), allowing the customisation of the graph to fit the specific variations described in the literature, such as LPVG, WVG or PNVG. In our package, we used and adapted the `ts2vg` procedures to align with our framework, ensuring that the resulting graph's nodes retain the same '`weight`' and '`name`' attributes, as in our own implementations.

DualPerspectiveVisibilityGraph. Class designed to create the WDPVG. This class has the following parameters:

- **ts:** the time series values.
- **xs:** the time series timings. Default is `None`.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is `False`.
- **weight:** the required weigh. If not provided, an unweighted graph is returned. Otherwise, a weighted graph will be created, using one of the available options in `pyI0mica`: `time`, `tan`, `distance`. Default is `None`.
- **withsign:** a boolean value indicating whether to return the sign of the adjacency matrix, given that the link from normal perspective VG is positive, the link from reflected perspective VG is negative. Default is `True`.

The `PyIOMica` library provides this time series mapping method, with two options: `weight` and `withsign`. Because this package uses the original approach for calculating the NVGs, the algorithm for the WDPVG (see Algorithm 3) also takes $\mathcal{O}(n^2)$.

5.3.2.2 horizontal sub-module

The `horizontal` sub-module contains classes using the horizontal visibility algorithms with the use of the available specific variations. We have already shown the algorithms and the correspondent time complexity in Section 3.1.2, so we will only expand on how we implemented those.

HorizontalVisibilityGraph. Class designed to create the HVG. This class has the following parameters:

- `ts`: the time series values.
- `xs`: the time series timings. Default is `None`.
- `direction`: the orientation of the direction used for the edges. If not provided, an undirected graph will be returned. Otherwise, a directed graph will be created, using one of the following values, from `ts2vg`: `left_to_right` or `top_to_bottom`. Default is `None`.
- `weight`: the required weigh. If not provided, an unweighted graph is returned. Otherwise, a weighted graph will be created, using one of the `ts2vg` options: `distance`, `sq_distance`, `v_distance`, `abs_v_distance`, `h_distance`, `abs_h_distance`, `slope`, `abs_slope`, `angle`, `abs_angle`, `num_penetrations`. Default is `None`.
- `min_weight`: a threshold value for the weight of the edges. If provided, only edges with a weight higher than the argument value (non inclusive) will be included in the final graph. The graph must be weighted and the values used for weight will depend on the `weighted` parameter. This acts as a generalisation of parametric visibility graphs. Default `None`.
- `max_weight`: similar to `min_weight`, except only edges lower than given `max_weight` parameter are included in the graph. Default `None`.
- `penetrable_limit`: a integer value. If larger than 0, a LPHVG is made. The value for `penetrable_limit` indicates the maximum number of data points that are allowed to obstruct the visibility between two connected nodes in the final graph. Default is 0.

As the `HorizontalVisibilityGraph` also uses the `ts2vg` implementation, it shares the same parameters as the `NaturalVisibilityGraph` class. The `ts2vg` implementation follows a DC approach to the HVG, akin to Algorithm 2, but applying the horizontal visibility condition.

However, when the length of the time series exceeds 50000 values and the function is called parameter-free (`weight=None`, `min_weight=None`, `max_weight=None`, `penetrable_limit=0`), we print a warning to the user, recommending calling `HorizontalVisibilityGraph_DualTree`. The reason behind this decision comes from Stephen's findings [50], which demonstrate that the dual tree method maintains relatively flat runtime performance for large-scale datasets. This makes it significantly more efficient and scalable compared to the other existent approaches, whose runtime rapidly increases when the sequence length approaches 50000 data points.

HorizontalVisibilityGraph_DualTree. Class designed to create the HVG, following the dual tree approach. This class has the following parameters:

- **ts:** the time series values.
- **xs:** the time series timings. Default is `None`.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is `False`.

We used the implementation provided in the context of the work developed by [50] and adapted it to the object-oriented logic of our package.

DualPerspectiveHorizontalVisibilityGraph. Class designed to create the WDPHVG. This class has the following parameters:

- **ts:** the time series values.
- **xs:** the time series timings. Default is `None`.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is `False`.
- **weight:** the required weigh. If not provided, an unweighted graph is returned. Otherwise, a weighted graph will be created, using one of the `pyIOMica` options: `time`, `tan`, `distance`. Default is `None`.
- **withsign:** a boolean value indicating whether to return the sign of the adjacency matrix, given that the link from normal perspective VG is positive, the link from reflected perspective VG is negative. Default is `True`.

By using the `PyIOMica` package, the `DualPerspectiveHorizontalVisibilityGraph` class shares the same parameters as `DualPerspectiveVisibilityGraph`. It also has equal time complexity, that is, $\mathcal{O}(n^2)$.

5.3.2.3 difference sub-module

DifferenceVisibilityGraph. Class designed to create the DVG. This class has the following parameters:

- **ts:** the time series values.
- **xs:** the time series timings. Default is `None`.
- **direction:** the orientation of the direction used for the edges. If not provided, an undirected graph will be returned. Otherwise, a directed graph will be created, using one of the following values, from `ts2vg`: `left_to_right` or `top_to_bottom`. Default is `None`.
- **weight:** the required weigh. If not provided, an unweighted graph is returned. Otherwise, a weighted graph will be created, using one of the `ts2vg` options: `distance`, `sq_distance`, `v_distance`, `abs_v_distance`, `h_distance`, `abs_h_distance`, `slope`, `abs_slope`, `angle`, `abs_angle`, `num_penetrations`. Default is `None`.
- **min_weight:** a threshold value for the weight of the edges. If provided, only edges with a weight higher than the argument value (non inclusive) will be included in the final graph. The graph must be weighted and the values used for weight will depend on the `weighted` parameter. This acts as a generalisation of parametric visibility graphs. Default `None`.
- **max_weight:** similar to `min_weight`, except only edges lower than `max_weight` are included in the graph. Default `None`.
- **penetrable_limit:** a integer value. If larger than 0, a limited penetrable graph is made. The `penetrable_limit` value indicates the maximum number of points that are allowed to obstruct the visibility between two connected nodes in the final graph. Default is 0.

As described in Algorithm 11, the DVG construction procedure starts by creating two graphs: a NVG and a HVG. We used the implementations of `NaturalVisibilityGraph` and `HorizontalVisibilityGraph` mentioned above for this effect. Therefore, we can use several arguments to customise our DVG. Then, the algorithm considers the edge set for the DVG the same as the NVG, and it iterates through the edges of the HVG and removes each of them from the edge set of the DVG.

Algorithm 11: Algorithm for DVG

```

1 function DifferenceVisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ ):
2    $V, E_1 \leftarrow \text{NaturalVisibilityGraph}(ts)$ ;
3    $\_, E_2 \leftarrow \text{HorizontalVisibilityGraph}(ts)$ ;
4    $E \leftarrow E_1$ ;
5   foreach  $edge$  in  $E_2$  do
6     if  $edge$  in  $E_1$  then  $\triangleright$  Should always be true, as HVG is a subset of NVG
7        $\quad$  delete  $edge$  from  $E$ ;
8   return  $G = (V, E)$ ;
```

The time complexity of this mapping function (Algorithm 11) depends on the complexity of the construction of the starting visibility graphs (which, by using the DC approach, is $\mathcal{O}(n \cdot \log n)$), and the for-loop in lines 5 to 9 which takes $\mathcal{O}(|E_2| \cdot |E_1|)$. The overall time complexity of this algorithm is $\mathcal{O}(n \cdot \log n + |E_2| \cdot |E_1|)$.

5.3.2.4 absolute sub-module

Following the idea of the invisibility algorithm depicted in Section 3.1.4, we created two classes: **AbsoluteInvisibilityGraph** and **HorizontalAbsoluteInvisibilityGraph**. The first encapsulates perfectly what was described in [58], as it re-imagines the NVG algorithm. The second expands these same principles, but applied to the original HVG algorithm.

AbsoluteInvisibilityGraph. Class designed to create the AIG. This class has the following parameters:

- **ts**: the time series values.
- **xs**: the time series timings. Default is **None**.
- **directed**: boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **False**.

Essentially, the building function for the AIG follows Algorithm 12. For any pair of time series observations, we check the visibility condition and if it holds, we add the corresponded edge to the graph. Thus, making the associated time complexity $\mathcal{O}(n^2)$. Note that, a possible improvement could be using a DC approach to the algorithm.

Algorithm 12: Algorithm for AIG

```

1 function AbsoluteInvisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ ):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \emptyset$ ;
4   for  $i$  from 1 to  $n - 1$  do
5     for  $j$  from  $i + 1$  to  $n$  do
6        $y_k \leftarrow y[i + 1 \dots j]$ ;
7        $t_k \leftarrow t[i + 1 \dots j]$ ;
8       if all  $(y_k[k] > (y_j + (y_i - y_j) \cdot (t_j - t_k[k]) / (t_j - t_i))$  for  $k$  in range( $\text{len}(y_k)$ ))
9         then
10          add edge( $t_i, t_j$ ) to  $E$ ;
11 return  $G = (V, E)$ ;

```

HorizontalAbsoluteInvisibilityGraph. Class designed to create the horizontal AIG. This class has the following parameters:

- **ts:** the time series values.
- **xs:** the time series timings. Default is **None**.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **False**.

The building function for the Horizontal AIG follows Algorithm 13. Again, for every pair of time series observations, we test the horizontal invisibility condition, and we add the corresponded edge to the graph if it maintains. The associated time complexity is also $\mathcal{O}(n^2)$.

Algorithm 13: Algorithm for Horizontal AIG

```

1 function HorizontalAbsoluteInvisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}$ ):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \emptyset$ ;
4   for  $i$  from 1 to  $n - 1$  do
5     for  $j$  from  $i + 1$  to  $n$  do
6        $y_k \leftarrow y[i + 1 \dots j]$ ;
7        $t_k \leftarrow t[i + 1 \dots j]$ ;
8       if all ( $y_k[k] > \max(y_i, y_j)$  for  $k$  in range( $\text{len}(y_k)$ )) then
9         add edge( $t_i, t_j$ ) to  $E$ ;
10  return  $G = (V, E)$ ;
```

5.3.2.5 circular sub-module

CircularVisibilityGraph. Class designed to create the CVG. This class has the following parameters:

- **ts:** the time series values.
- **xs:** the time series timings. Default is **None**.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **False**.
- **alpha:** central angle of the minor visibility arc. Controls the circle radius. Default is -1 .
- **penetrable_limit:** a integer value. If larger than 0, a CLPVG is made. The value for **penetrable_limit** indicates the maximum number of data points that are allowed to obstruct the visibility between two connected nodes in the final graph. Default is 0.

Algorithms 14 and 15 demonstrates how to generate a CVG. For every pair of time series observations, we test the circular visibility condition, which involves finding the solution of the associated circle system. Again, the corresponded edge is added to the graph when the condition holds. The associated time complexity is then $\mathcal{O}(n^2)$, as `circle_system` has constant time, and we iterate through every pair of data points.

Algorithm 14: Algorithm for CVG

```

1 function CircularVisibilityGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}, \alpha, L$ ):
2    $V \leftarrow \{t_1, \dots, t_n\}$ ;
3    $E \leftarrow \emptyset$ ;
4    $count \leftarrow 0$ ;
5   for  $i$  from 1 to  $n - 1$  do
6     for  $j$  from  $i + 1$  to  $n$  do
7        $y_k \leftarrow y[i + 1 \dots j]$ ;
8        $t_k \leftarrow t[i + 1 \dots j]$ ;
9       for  $k$  in  $\text{range}(\text{len}(y_k))$  do
10        if  $y_k \geq \text{circle\_system}(t_i, y_i, t_j, y_j, t_k)$  then
11           $count \leftarrow count + 1$ ;
12          if  $count > L$  then
13            break;
14        if  $count \leq L$  then
15          add edge( $t_i, t_j$ ) to  $E$ ;
16  return  $G = (V, E)$ ;
```

Algorithm 15: Circle System Function

```

1 function circle_system( $t_i, y_i, t_j, y_j, t_k, \alpha$ ):
2    $a \leftarrow 1$ ;
3    $b \leftarrow \alpha \cdot t_i - \alpha \cdot t_j - y_i - y_j$ ;
4    $c \leftarrow \alpha \cdot (y_i \cdot t_j - y_j \cdot t_i + y_j \cdot t_c - y_i \cdot t_k) + y_i \cdot y_j + t_a \cdot t_j + t_k \cdot t_k - t_k \cdot t_j - t_k \cdot t_i$ ;
5    $\Delta \leftarrow b^2 - 4 \cdot a \cdot c$ ;
6   if  $\Delta < 0$  then
7     return None; ▷ There are no real roots
8    $x1 \leftarrow \frac{-b + \sqrt{\Delta}}{2 \cdot a}$ ;
9    $x2 \leftarrow \frac{-b - \sqrt{\Delta}}{2 \cdot a}$ ;
10   $roots \leftarrow (x1, x2)$ ;
11  if  $\alpha > 0$  then
12    return min( $roots$ );
13  else
14    return max( $roots$ );
```

5.3.3 univariate/transition module

In this section, we will explore the algorithms implemented for the univariate time series mapping methods based on the concept of probability transition, described in Section 3.2. Note that all algorithms presented here were implemented from scratch.

The sub-modules and respective classes included in this module are described below.

5.3.3.1 quantile sub-module

QuantileGraph. Class designed to create the QG. This class has the following parameters:

- **ts:** the time series values.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **True**.
- **weighted:** a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between the nodes. Default is **True**.
- **Q:** a integer value corresponding to the number of quantiles. Default is 4.
- **method:** corresponds to the NumPy statistical method available for the calculation of quantile samples. The options available are: **inverted_cdf**, **averaged_inverted_cdf**, **closest_observation**, **interpolated_inverted_cdf**, **hazen**, **weibull**, **linear**, **median_unbiased**, and **normal_unbiased**. Default is **linear**.

To create a QG, we need to, first, identify the upper boundary of each quantile, and then, for each observation, determine the quantile it belongs to, generating a sequence. We decide to use the **digitize** function of NumPy library for this step. Finally, we traverse this sequence and create edges between consecutive quantiles. Algorithm 16 presents this procedure.

The time complexity of the QG mapping method depends on the for-loop in lines 5 to 11 and the **get_adj_matrix** function (line 4). The for-loop has a complexity of $\mathcal{O}(Q^2)$, as it iterates through the adjacency matrix of size $Q \times Q$. The complexity of the **get_adj_matrix** function is influenced by the complexities of two NumPy functions, **quantile** and **digitize**, as well as a for-loop with n iterations in lines 19 to 23. The **quantile** function relies on two operations: sorting the time series array using, worst-case, a Quicksort algorithm, which has average complexity of $\mathcal{O}(n \log n)$, and selecting the quantile for each data point, which takes $\mathcal{O}(1)$ for the entire time series length. The **digitize** function iterates through the values of the time series, finding the appropriate quantile bin for each value using the Binary Search algorithm, resulting in a time complexity of $\mathcal{O}(n \log Q)$. Overall, the **get_adj_matrix** function runs in $\mathcal{O}(n \log n)$, and Algorithm 16 runs in $\mathcal{O}(Q^2 + n \log n)$.

Algorithm 16: Algorithm for QG

```

1 function QuantileGraph( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}, Q, method$ ):
2    $V \leftarrow \{1, \dots, Q\}$ ;
3    $E \leftarrow \emptyset$ ;
4    $A \leftarrow \text{get\_adj\_matrix}(ts, Q, method)[0]$ ;
5   for  $i$  from 0 to  $Q - 1$  do
6     for  $j$  from 0 to  $Q - 1$  do
7       if  $A[i][j] \neq 0$  then
8          $\text{add edge}(i + 1, j + 1)$  to  $E$ ;
9   return  $G = (V, E)$ ;
10 function get_adj_matrix( $ts, Q, method$ ):
11    $A \leftarrow$  empty matrix of size  $(Q, Q)$ ;
12    $q \leftarrow \text{linspace}(1/Q, 1, Q)$ ; ▷ linspace is a NumPy function
13    $q \leftarrow$  insert 0 at the beginning of  $q$ ;
14    $q\_boundaries \leftarrow \text{quantile}(y(t), q, method)$ ; ▷ quantile is a NumPy function
15    $q\_sequence \leftarrow \text{digitize}(ts, q\_boundaries, \text{right}=\text{False})$ ;
16   for  $i$  from 0 to  $Q - 1$  do
17      $from\_node \leftarrow q\_sequence[i]$ ;
18      $to\_node \leftarrow q\_sequence[i + 1]$ ;
19      $A[from\_node, to\_node] \leftarrow A[from\_node, to\_node] + 1$ ;
20   return  $A, q\_boundaries, q\_sequence$ ;

```

5.3.3.2 ordinal_partitions sub-module

OrdinalPartitionsTransitionNetwork. Class designed to create the OPTN. This class has the following parameters:

- **ts**: the time series values.
- **directed**: boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **True**.
- **weighted**: a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between the nodes. Default is **True**.
- **dim**: an integer value indicating the embedding dimension. Default is 3.
- **tau**: an integer value indicating the embedding time delay. Default is 1.

The main problem in the implementation of the OPTN, present in Algorithm 17, lied in how to perform the discretisation of the time series embedding. We decided to use a dictionary to map the elements from the embedding list to their corresponding ordinal ranks, making the process simpler to follow, as well as more efficient. Afterwards, we only needed to find the unique combinations or ordinal ranks, and connect successive ones in the time series data.

Algorithm 17: Algorithm for Ordinal Partitions Transition Network

```

1 function OrdinalPartitionsTransitionNetwork(ts, dim,  $\tau$ ):
2   embedded_ts  $\leftarrow$  EmbedTimeSeries(ts, dim,  $\tau$ );      ▷ Presented in Algorithm 6
3   discretized_embedded_ts  $\leftarrow$  DiscretizeEmbedding(embedded_ts);
4   L  $\leftarrow$  length of discretized_embedded_ts;
5   V  $\leftarrow$   $\emptyset$ ;
6   E  $\leftarrow$   $\emptyset$ ;
7   for i, data in enumerate(discretized_embedded_ts) do
8     node  $\leftarrow$  data transformed into a concatenated string;
9     if node  $\notin$  V then                                ▷ Encounter unique possible combinations
10      |   add node to V;
11   for i from 0 to L - 1 do
12     current_node  $\leftarrow$  discretized_embedded_ts[i] as a concatenated string;
13     next_node  $\leftarrow$  discretized_embedded_ts[i + 1] as a concatenated string;
14     add edge (current_node, next_node) to E;
15   return G = (V, E);
16 function DiscretizeEmbedding(embedded_ts):
17   discretized_data  $\leftarrow$  [ ];
18   foreach embedding in embedded_ts do
19     ordinal_rank  $\leftarrow$  list(range(len(embedding), 0, -1));    ▷ Create the ranks in
20     |   descending order
21     ordinal_mapping  $\leftarrow$  dict(zip(sorted(embedding), ordinal_rank));
22     discretized_embedding  $\leftarrow$  [ ];
23     foreach value in embedding do
24       |   discretized_embedding.append(ordinal_mapping[value]); ▷ Map each value
25       |   to its rank
26     discretized_data.append(discretized_embedding);
27   return discretized_data;

```

The time complexity of Algorithm 17 relies on the `EmbedTimeSeries` and `DiscretizeEmbedding` functions and the two for-loops presented in lines 7-12 and 13-17. `EmbedTimeSeries` function (presented in Algorithm 6) has time complexity of $\mathcal{O}(L \cdot \text{dim})$ (see Section 3.3.3), the `DiscretizeEmbedding` function iterates through each value of each vector in the embedded time series, resulting also in $\mathcal{O}(L \cdot \text{dim})$, and the two for-loops in Algorithm 17 have a linear complexity, $\mathcal{O}(L)$. The overall time complexity of Algorithm 17 is $\mathcal{O}(L \cdot \text{dim})$, with $L = n - (\text{dim} - 1) \cdot \tau$.

5.3.3.3 coarse_graining sub-module

CoarseGrainedPhaseSpaceGraph. Class designed to create the Coarse-Grained Phase Space Graph. This class has the following parameters:

- **ts:** the time series values.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **True**.
- **weighted:** a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between the nodes. Default is **True**.
- **thresholds:** array like of *float* values to obtain the bins.
- **dim:** an integer value indicating the embedding dimension. Default is 3.
- **tau:** an integer value indicating the embedding time delay. Default is 1.

The process of originating a Coarse-Grained Phase Space Graph combines steps from both the implementations of QG and OPTN. On one hand, we need to determine in which bin each value of the embedded time series belongs to according to the defined thresholds, using also NumPy `digitize`. On the other hand, we also have to determine the unique combinations of node names first, before linking consecutive ones in the data.

The complexity associated with the Coarse-Grained Phase Space Graph mapping method presented in Algorithm 18 is influenced by the two sequential for-loops, from line 8-13 and 14-18, that run in $\mathcal{O}(L)$ time (as checking if an element is within a set takes $\mathcal{O}(1)$), and by the complexity associated to `CoarseGrainVectors` and `EmbedTimeSeries` functions. `CoarseGrainVectors` relies on `digitize` function, which runs in time complexity of $\mathcal{O}(L \cdot \log D)$, as explained in Section 5.3.3.1. The `EmbedTimeSeries` function has $\mathcal{O}(L \cdot \dim)$, as explained in Section 3.3.3. Thus, the overall time complexity of the Algorithm 18 is $\mathcal{O}(L \cdot \log D + L \cdot \dim)$, with $L = n - (\dim - 1) \cdot \tau$.

5.3.3.4 graphlets sub-module

VisibilityGraphletsNetwork. Class designed to create the Visibility Graphlets Network. This class has the following parameters:

- **ts:** the time series values.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **True**.
- **weighted:** a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between nodes. Default is **True**.

Algorithm 18: Algorithm for Coarse-Grained Phase Space Graph

```

1 function CoarseGrainedPhaseSpaceGraph(ts, thresholds, dim,  $\tau$ ):
2   embedded_ts  $\leftarrow$  EmbedTimeSeries(ts, dim,  $\tau$ );      ▷ Presented in Algorithm 6
3   discretized_embedded_ts  $\leftarrow$  CoarseGrainVectors(embedded_ts, thresholds);
4   L  $\leftarrow$  length of discretized_embedded_ts;
5   D  $\leftarrow$  length of thresholds;
6   V  $\leftarrow \emptyset$ ;
7   E  $\leftarrow \emptyset$ ;
8   for i, data in enumerate(discretized_embedded_ts) do
9     node  $\leftarrow$  data transformed into a concatenated string;
10    if node  $\notin V$  then                                ▷ Encounter unique possible combinations
11      |   add node to V;
12    for i from 0 to L - 1 do
13      |   current_node  $\leftarrow$  discretized_embedded_ts[i] as a concatenated string;
14      |   next_node  $\leftarrow$  discretized_embedded_ts[i + 1] as a concatenated string;
15      |   add edge (current_node, next_node) to E;
16    return G = (V, E);
17 function CoarseGrainVectors(embedded_ts, thresholds):
18   bins  $\leftarrow$  concatenate(thresholds,  $\infty$ );
19   symbols  $\leftarrow$  digitize(embedded_ts, bins, right = False);  ▷ Gets indices of the
20   |   bins to which each value in input array belongs
21   symbols  $\leftarrow$  symbols + 1;                                ▷ Adjust symbols to start from 1
22   return symbols;

```

- **dim**: an integer value indicating the embedding dimension, that will also correspond to the number of nodes in the visibility graphlets. Default is 5.
- **tau**: a integer value indicating the embedding time delay. Default is 1.
- **first_occurrence_label**: a boolean value which decides how to name the nodes. If **True**, the nodes name is of type $x(y)$, with x being the first occurrence index and y the motif number. Otherwise, x corresponds to the index of the last occurrence of the graph.

For the implementation of the Visibility Graphlets Network, we need to first iterate through the embedded vectors and map each into a NVG. For each of these graphs, we compare it to the previously generated graphs that we stored in a dictionary as a value, alongside the first and last occurrences. Each key in the dictionary represents the motif number, by order of appearance. This comparison makes use of the `igraph isomorphic` function, that evaluates if two graphs are indistinguishable from each other, ignoring the vertex labels. This allows to store only the unique motifs, and to update the first and last occurrences, if necessary. Furthermore, as we traverse the embedded vectors, we also add edges between the consecutive correspondent motifs.

The complexity associated to Visibility Graphlets Network methods that is presented in the Algorithm 19 depends on several factors: the `isomorphic` (with a complexity depending on the method used and the graph properties), the `EmbedTimeSeries` function (with time complexity of $\mathcal{O}(L \cdot \text{dim})$), the `NaturalVisibilityGraph` algorithm (with complexity of $\mathcal{O}(\text{dim} \log \text{dim})$), and the for-loops of lines 11 to 31 that iterates L time the `NaturalVisibilityGraph` algorithm for each subgraph and the `isomorphic` function to each key graphlet in the dictionary.

According to the official documentation, if the graphlets have three or four vertices, the `igraph isomorphic` function has a complexity of $\mathcal{O}(1)$, by using pre-computed data; if the graphs are directed, it uses the *VF2 isomorphism algorithm*; otherwise, the *BLISS isomorphism algorithm* is the one used. For the VF2 method, the time complexity is $\mathcal{O}(|V|^2)$ in the best case and $\mathcal{O}(|V|! \cdot |V|)$ in the worst case [11]. For the BLISS method the complexity is not known [28]. In this mapping method case, the number of node $|V|$ corresponds to dim .

Therefore, and assuming the time complexity of $\mathcal{O}(\text{dim}!)$ for `isomorphic` function, the time complexity of Algorithm 19 is dominated by the for-loops, resulting in $\mathcal{O}(L \cdot \text{dim} \log \text{dim} + L \cdot x \cdot \text{dim}!)$, with x being the number of key graphs checked for isomorphism. Since factorial growth dominates and the x can be at most L (in the worst case, every graphlet is unique), the overall complexity is $\mathcal{O}(L^2 \cdot \text{dim}!)$.

5.3.4 univariate/proximity module

To finalise the exploration of our developed implementations for the univariate mapping methods, in this section, we focus on the methods based on proximity, described in the previous Section 3.3. We describe below the correspondent sub-modules and integrating classes.

5.3.4.1 cycle sub-module

CycleNetwork. Class designed to create the Cycle Network. This class has the following parameters:

- **ts:** the time series values.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **False**.
- **weighted:** a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between the nodes. Default is **False**.
- **threshold:** the threshold value for determining the presence of an edge between nodes based on their correlation or distance. Default is 0.5.
- **method:** corresponds to the method available for the similarity calculation between cycles. The options available are **correlation**, and **distance**. Default is **correlation**.

Algorithm 19: Algorithm for Visibility Graphlets Network

```

1 function VisibilityGraphletsNetwork( $ts, dim, \tau$ ):
2    $embedded\_ts \leftarrow \text{EmbedTimeSeries}(ts, dim, \tau)$ ;      ▷ Presented in Algorithm 6
3    $L \leftarrow \text{length of } embedded\_ts$ ;
4    $V \leftarrow \emptyset$ ;
5    $E \leftarrow \emptyset$ ;
6    $added\_graphs \leftarrow \{\}$ ;      ▷ To store motif number, graph, and occurrences
7    $graph \leftarrow \text{NaturalVisibilityGraph}(embedded\_ts[0])$ ;      ▷ Add first graph
8    $added\_graphs['1'] \leftarrow \{\text{'graph': } graph, \text{'first\_occurrence': } 1, \text{'last\_occurrence': } 1\}$ ;
9    $last\_key \leftarrow '1'$ ;
10  add  $last\_key$  to  $V$ ;
11  for  $i$  from 1 to  $L$  do
12     $graph \leftarrow \text{NaturalVisibilityGraph}(embedded\_ts[i])$ ;
13     $found\_isomorphic \leftarrow \text{False}$ ;
14     $current\_key \leftarrow \text{None}$ ;
15    foreach  $key, data$  in  $added\_graphs$  do
16      ▷ Check for isomorphic graph in added_graphs
17      if  $graph.isomorphic(data['graph'])$  then
18         $current\_key \leftarrow key$ ;
19         $added\_graphs[key]['last\_occurrence'] \leftarrow i + 1$ ;
20         $found\_isomorphic \leftarrow \text{True}$ ;
21        break;
22    if not  $found\_isomorphic$  then
23      ▷ Generate new key based on the current length of added_graphs
24       $new\_key \leftarrow \text{length of } added\_graphs + 1$ ;
25       $added\_graphs[new\_key] \leftarrow \{\text{'graph': } graph, \text{'first\_occurrence': } i + 1,$ 
26         $\text{'last\_occurrence': } i + 1\}$ ;
27       $current\_key \leftarrow new\_key$ ;
28      add  $current\_key$  to  $V$ ;
29      add edge ( $last\_key, current\_key$ ) to  $E$ ;
30       $last\_key \leftarrow current\_key$ ;
31  return  $G = (V, E)$ ;

```

- **cycles:** a matrix with the observation values for each cycle, in temporal order. Default is *None*.
- **radius:** the radius for local neighbourhood search in the cycle detection process. Default is 5.

The Cycle Network is tailored to pseudo-periodic time series, and the implementation requires the identification of the correspondent cycles in the data. This process is usually specific to the characteristics of the data in question, and so, we allow the user to provide these cycles as

a parameter. However, we also provide a function to detect cycles, based on the minima and maxima of the time series (found using NumPy `arglextrema` function), if the user chooses to not provide the previous mentioned parameter.

Algorithm 20 illustrates the process as whole, while Algorithm 21 shows the implementation of the distance/correlation measures. Overall, because of the for-loops, from lines 6 to 14, and because of the for-loops in `get_adj_matrix`, the algorithm takes $\mathcal{O}(L^2 \cdot f(n))$, with $f(n)$ being the complexity of the measure function.

If we consider i and j as the lengths of `cycle_i` and `cycle_j`, respectively, the `GetCorrelation` function runs in $\mathcal{O}((j - i) \cdot j) = \mathcal{O}(j^2)$ time, by using the NumPy `cov` and `var` functions, which take linear time. `GetDistance` has complexity $\mathcal{O}(j - i) = \mathcal{O}(j)$. Each cycle can have a different length than the others, though. So, if we consider the biggest cycle length as C , the overall worst-case complexity is $\mathcal{O}(L^2 \cdot C^2)$.

Algorithm 20: Algorithm for Cycle Network

```

1 function CycleNetwork( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}, cycles, r, method$ ):
2    $L \leftarrow$  length of  $cycles$ ;
3    $V \leftarrow \{1, \dots, L\}$ ;
4    $E \leftarrow \emptyset$ ;
5    $A \leftarrow$  get_adj_matrix( $cycles, L, method$ );
6   for  $i$  from 0 to  $L - 1$  do
7     for  $j$  from 0 to  $L - 1$  do
8       if  $i \neq j$  then
9         if  $|A[i][j]| \geq r$  then
10          add edge( $i + 1, j + 1$ ) to  $E$ ;
11   return  $G = (V, E)$ ;
12 function get_adj_matrix( $cycles, L, method$ ):
13    $A \leftarrow$  empty matrix of size  $(L, L)$ ;
14   for  $i$  from 0 to  $L - 1$  do
15     for  $j$  from 0 to  $L - 1$  do
16       if  $method == 'correlation'$  then
17          $A[i][j] \leftarrow$  GetCorrelation( $cycles[i], cycles[j]$ );
18       else
19          $A[i][j] \leftarrow$  GetDistance( $cycles[i], cycles[j]$ );
20   return  $A$ ;
```

Algorithm 21: Auxiliary Functions for Cycle Network

```

1 function GetCorrelation(cycle_i, cycle_j):
2   max_corr ← -1;
3   var_cycle_i ← var(cycle_i);
4   for lag ← 0 to |cycle_j| - |cycle_i| do
5     cov ← cov(cycle_i, cycle_j[lag : lag + |cycle_i|]);
6     var_cycle_j ← var(cycle_j[lag : lag + |cycle_i|]);
7     corr ←  $\frac{cov}{\sqrt{var\_cycle\_i \cdot var\_cycle\_j}}$ ;
8     if corr > max_corr then
9       max_corr ← corr;
10  return max_corr;
11 function GetDistance(cycle_i, cycle_j):
12  min_distance ← ∞ ; ▷ Initialize with a large value
13  for lag ← 0 to |cycle_j| - |cycle_i| do
14    distance ← 0 ;
15    for k ← 0 to |cycle_i| - 1 do
16      distance ← distance + ||cycle_i[k] - cycle_j[lag + k]|| ;
17    distance ← distance / |cycle_i| ;
18    if distance < min_distance then
19      min_distance ← distance ;
20  return min_distance;

```

5.3.4.2 correlation sub-module

CorrelationNetwork. Class designed to create the Correlation Network. This class has the following parameters:

- **ts**: the time series values.
- **directed**: boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **False**.
- **weighted**: a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between the nodes. Default is **False**.
- **dim**: the embedding dimension. Default is 3.
- **tau**: the embedding time delay. Default is 1.
- **threshold**: the threshold value for determining the presence of an edge between nodes based on their correlation. Default is 0.5.
- **method**: corresponds to the SciPy method available for the correlation calculation between vectors. The options available are **pearson**, and **spearman**. Default is **pearson**.

The implementation of the Correlation Network involves considering each pair of resulting embedded vectors from the time series and calculating the correlation of both. The original Correlation Network used the Pearson correlation coefficient, but we also provide the option of the Spearman Correlation measure, by leveraging the options from the `SciPy` library [53].

The `SciPy` `pearsonr` function involves calculating mean, variance and covariance, thus running in linear time. `spearmanr` is similar, but requires first the sorting of the data, using Quicksort, so it can, sequentially, compute its rankings, making the associated complexity $\mathcal{O}(dim \cdot \log dim)$, as dim corresponds to the size of the vectors.

We can see in Algorithm 22 in lines 7 to 15 and in lines 19 to 23 of the `get_adj_matrix` function, there are two sequential for-loops with range L . The operations inside the first only take $\mathcal{O}(1)$ time, while, on the second, the operations run in $\mathcal{O}(dim)$ or $\mathcal{O}(dim \cdot \log dim)$, according to the correlation methods. Worst case corresponds then to $\mathcal{O}(L^2 \cdot (dim \cdot \log dim))$.

Algorithm 22: Algorithm for Correlation Network

```

1 function CorrelationNetwork( $ts = \{(t_1, y_1), \dots, (t_n, y_n)\}, dim, \tau, r, method$ ):
2    $embedded\_ts \leftarrow \text{EmbedTimeSeries}(ts, dim, \tau)$ ; ▷ Presented in Algorithm 6
3    $L \leftarrow \text{length of } embedded\_ts$ ;
4    $V \leftarrow \{1, \dots, L\}$ ;
5    $E \leftarrow \emptyset$ ;
6    $A \leftarrow \text{get\_adj\_matrix}(embedded\_ts, L, method)$ ;
7   for  $i$  from 0 to  $L - 1$  do
8     for  $j$  from 0 to  $L - 1$  do
9       if  $i \neq j$  then
10        if  $|A[i][j]| \geq r$  then
11          add edge( $i + 1, j + 1$ ) to  $E$ ;
12   return  $G = (V, E)$ ;

13 function get_adj_matrix( $embedded\_ts, L, method$ ):
14    $A \leftarrow$  empty matrix of size  $(L, L)$ ;
15   for  $i$  from 0 to  $L - 1$  do
16     for  $j$  from 0 to  $L - 1$  do
17        $A[i][j] \leftarrow$  correlation of  $embedded\_ts[i]$  and  $embedded\_ts[j]$ , according to
         $method$ ;
18   return  $A$ ;
```

5.3.4.3 recurrence sub-module

RecurrenceNetwork. Class designed to create the Recurrence Network. This class has the following parameters:

- **ts:** the time series values.
- **directed:** boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **False**.
- **weighted:** a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between nodes. Default is **False**.
- **metric:** the metric used for measuring distances in phase space. Could be one of the following values: 'manhattan', 'euclidean', 'supremum'. Default is **supremum**.
- **normalize:** a boolean value to decide whether to normalise the time series to zero mean and unit standard deviation. Default is **False**.
- **missing_values:** a boolean value to toggle special treatment of missing values. Default is **False**.
- **silence_level:** inverse level of verbosity of the object. Default is 0.
- **threshold:** the recurrence threshold keyword for generating the recurrence network using a fixed threshold. Default is **None**.
- **threshold_std:** the recurrence threshold keyword for generating the recurrence plot using a fixed threshold in units of the time series' STD. Default is **None**.
- **recurrence_rate:** the recurrence rate keyword for generating the recurrence network using a fixed recurrence rate. Default is **None**.
- **local_recurrence_rate:** the local recurrence rate keyword for generating the recurrence plot using a fixed local recurrence rate (same number of recurrences for each state vector). Default is **None**.
- **adaptive_neighborhood_size:** the adaptive neighbourhood size parameter for generating recurrence plots based on the algorithm in [56]. Default is **None**.
- **dim:** an integer values indicating the embedding dimension. Default is **None**.
- **tau:** an integer values indicating the embedding delay. Default is **None**.
- **node_weights:** the sequence of weights associated with each node for calculating network measures. Default is **None**.

We used the implementation from the **pyunicorn** package, which provides plenty of customisable parameters, to adapt to the different types of Recurrence Networks, k -NNN, ANNN, and ϵ -Recurrence Network. Algorithm 6 shows how a Recurrence Network is built (see Section 3.3.3), and, as established before, runs in $\mathcal{O}(L^2 \cdot (1 + \dim + \log L))$.

5.3.5 multivariate/single_layer module

In this module, we encounter the implemented multivariate mapping methods, that originate single layer networks (see Section 4.1 for descriptions). As we have mentioned, only one of the existent methods was followed through in our package, but we hope to expand this module in the future.

5.3.5.1 correlation sub-module

MultivariateCorrelationNetwork. Class designed to create the Multivariate Correlation Network. This class has the following parameters:

- **mts**: matrix with all the values for all the time series components.
- **directed**: boolean value to denote if a directed graph is created, using the left to right orientation, or not. Default is **False**.
- **weighted**: a boolean value. If **False**, an unweighted graph is returned. Otherwise, a weighted graph will be created, using as weights the transition frequency between the nodes. Default is **False**.
- **threshold**: the threshold value for determining the presence of an edge between nodes based on their correlation. Default is 0.5.
- **method**: corresponds to the SciPy method available for the correlation calculation between vectors. The options available are **pearson**, and **spearman**. Default is **pearson**.

The Multivariate Correlation Network involves considering each pair of time series components and calculating the correlation between both. Algorithm 23 illustrates the concept and has an associated complexity of $\mathcal{O}(m^2 \cdot f(n))$, as, both from lines 15 to 19 of the `get_adj_matrix` function and from lines 5 to 11, there are two sequential for-loops with range m , and in the second, we call a SciPy function for the correlation calculation, which takes $f(n)$. By using `pearsonr`, $f(n) = \mathcal{O}(n)$, and for `spearmanr`, $f(n) = \mathcal{O}(n \log n)$. So, in worst case, the algorithm's complexity is $\mathcal{O}(m^2 \cdot n \log n)$.

Algorithm 23: Algorithm for Multivariate Correlation Network

```

1 function MultivariateCorrelationNetwork( $mts =$ 
   $\{((t_{1,1}, y_{1,1}), \dots, (t_{1,n}, y_{1,n})), \dots, ((t_{m,1}, y_{m,1}), \dots, (t_{m,n}, y_{m,n}))\}, r, method):$ 
2    $V \leftarrow \{1, \dots, m\};$ 
3    $E \leftarrow \emptyset;$ 
4    $A \leftarrow \text{get\_adj\_matrix}(mts, method);$ 
5   for  $i$  from 0 to  $m - 1$  do
6     for  $j$  from 0 to  $m - 1$  do
7       if  $|A[i][j]| \geq r$  then
8          $\text{add edge}(i + 1, j + 1)$  to  $E;$ 
9   return  $G = (V, E);$ 
10 function get_adj_matrix( $mts, method):$ 
11    $A \leftarrow$  empty matrix of size  $(m, m);$ 
12   for  $i$  from 0 to  $m - 1$  do
13     for  $j$  from 0 to  $m - 1$  do
14        $A[i][j] \leftarrow$  correlation of  $mts[i]$  and  $mts[j]$ , according to  $method;$ 
15   return  $A;$ 

```

5.3.6 multivariate/multi_layer module

This module provides classes for multivariate mapping methods, originating multi-layer networks, as described in Section 4.2. All classes assume equal lengths for all time series components.

5.3.6.1 multiplex sub-module

MultiplexNetwork. Class designed to create the Multiplex Network. This class has the following parameters:

- **mts:** matrix with all the values for all the time series components.
- **layer_method:** the method used for layer construction. Available options and the correspondent methods are:
 - ‘natural’: NaturalVisibilityGraph.
 - ‘dual_perspective’: DualPerspectiveVisibilityGraph.
 - ‘horizontal’: HorizontalVisibilityGraph.
 - ‘dual_perspective_horizontal’: DualPerspectiveHorizontalVisibilityGraph.
 - ‘horizontal_dual_tree’: HorizontalVisibilityGraph_DualTree.
 - ‘absolute’: AbsoluteInvisibilityGraph.
 - ‘absolute_horizontal’: HorizontalAbsoluteInvisibilityGraph.
 - ‘circular’: CircularVisibilityGraph.
 - ‘difference’: DifferenceVisibilityGraph.

- 'ordinal_partitions': OrdinalPartitionsTransitionNetwork.
- 'quantile': QuantileGraph.
- 'graphlets': VisibilityGraphletsNetwork.
- 'coarse_graining': CoarseGrainedPhaseSpaceGraph.
- 'correlation': CorrelationNetwork.
- 'cycle': CycleNetwork.
- 'recurrence': RecurrenceNetwork.

Default is 'natural'.

- **couplings**: pymnet parameter, determining the type of couplings between the layers. Options include categorical, ordinal or None. Default is categorical.
- ****kwargs**: all necessary or available parameters for the respective class of the chosen layer_method.

Algorithm 24 illustrates how to build a Multiplex Network using the **pymnet** functionalities. We expanded the original concepts of Multiplex Visibility and Multiplex Recurrence Networks to include every possible univariate method we have in our package. It starts by building each layer respective graph, according to the chosen **layer_method** parameter, and then uses the **pymnet couplings** option to provide categorical inter-layer edges. For this reason, the full algorithm runs in $\mathcal{O}(m \cdot n + m \cdot f(n))$, with $f(n)$ being the complexity of the correspondent mapping method, **layer_method**.

Algorithm 24: Algorithm for Multiplex Network

```

1 function MultiplexNetwork(mts =
  {((t1,1, y1,1), ..., (t1,n, y1,n)), ..., ((tm,1, ym,1), ..., (tm,n, ym,n))}, method, **kwargs):
2   G ← {};
3   C ← {};
4   for α from 1 to m do
5     | layerα ← graph built from mtsα, according to method and **kwargs;
6     | add layerα to G;
7   for α from 1 to m do
8     | for i from 1 to n do
9     | | add edge (i, i, mtsα, mtsα+1) to C;
10  return G;

```

5.3.6.2 horizontal sub-module

MultilayerHorizontalVisibilityGraph Class designed to create the MHVG. This class has the following parameters:

- **mts**: matrix with all the values for all the time series components.
- **xs**: the time series timings. Default is *None*.

The description of the correspondent algorithm was presented in the literature review portion of this dissertation, in Algorithm 7.

5.3.6.3 quantile sub-module

MultilayerQuantileGraph. Class designed to create the MQG. This class has the following parameters:

- **mts**: matrix with all the values for all the time series components.
- **Q**: an integer values corresponding to the number of quantiles and is equivalent to the algorithm parameter Q . Default is 4.
- **method**: corresponds to the NumPy statistical method available for the quantile calculation. The options available are `inverted_cdf`, `averaged_inverted_cdf`, `closest_observation`, `interpolated_inverted_cdf`, `hazen`, `weibull`, `linear`, `median_unbiased`, and `normal_unbiased`. Default is `linear`.

Similarly, the correspondent algorithm of the MQG was described in Algorithm 9. We provide different parameters, designed to specify how to build the QG for each layer.

5.3.7 Summary

In Section 5.3 and in Chapters 3 and 4, we have presented algorithms and the respective time complexities for all mapping methods comprehended in our package. In this section, we aim to provide, in a single table (see Table 5.1), a summary with all methods encountered in our framework, correspondent time complexities and if the algorithm re-uses other libraries' implementations or if it is implemented by scratch.

Additionally, we also reflect on possibly problematic methods, regarding larger datasets, and possible improvements for certain algorithms.

For the most part, transition methods seem to be the most time efficient. Because they work by reducing the dimensionality of the data, that makes them really efficient methods for larger time series datasets. VGs are also generally fast, but the implementations for the AIG and CVG could be improved, for example, by following a DC approach.

The proximity mappings are computationally more demanding.

Regarding the multivariate approaches, there are also possible improvements for the implementation of the MHVG.

Additionally, it is worth noting that if the variable n is larger than the variable m , as it is typically, the MQG is a more efficient way to use to study a multivariate time series than the MHVG, in terms of time.

Class Name	Time Complexity	Nature
univariate/visibility/		
NaturalVisibilityGraph	$\mathcal{O}(n \cdot \log n)$	R
DualPerspectiveVisibilityGraph	$\mathcal{O}(n^2)$	R
HorizontalVisibilityGraph	$\mathcal{O}(n \cdot \log n)$	R
HorizontalVisibilityGraph_DualTree	$\mathcal{O}(n)$	L
DualPerspectiveHorizontalVisibilityGraph	$\mathcal{O}(n^2)$	R
DifferenceVisibilityGraph	$\mathcal{O}(n \cdot \log n + E ^2)$	S
AbsoluteInvisibilityGraph	$\mathcal{O}(n^2)$	S
HorizontalAbsoluteInvisibilityGraph	$\mathcal{O}(n^2)$	S
CircularVisibilityGraph	$\mathcal{O}(n^2)$	S
univariate/transition/		
QuantileGraph	$\mathcal{O}(Q^2 + n \log n)$	S
OrdinalPartitionsNetwork	$\mathcal{O}(L \cdot \dim)$	S
CoarseGrainedPhaseSpaceGraph	$\mathcal{O}(L \cdot \log D + L \cdot \dim)$	S
VisibilityGraphletsNetwork	$\mathcal{O}(L^2 \cdot \dim!)$	S
univariate/proximity/		
CycleNetwork	$\mathcal{O}(L^2 \cdot C^2)$	S
CorrelationNetwork	$\mathcal{O}(L^2 \cdot \dim \cdot \log \dim)$	S
RecurrenceNetwork	$\mathcal{O}(L^2 \cdot (1 + \dim + \log L))$	R
multivariate/single_layer/		
MultivariateCorrelationNetwork	$\mathcal{O}(m^2 \cdot n \cdot \log n)$	S
multivariate/multi_layer/		
MultiplexNetwork	$\mathcal{O}(m \cdot n + m^2 \cdot f(n))$	S
MultilayerHorizontalVisibilityGraph	$\mathcal{O}(m^2 \cdot n^2)$	L
MultilayerQuantileGraph	$\mathcal{O}(m \cdot n \log n + m^2 \cdot n)$	L

Table 5.1: Class name, followed by the associated time complexities and the nature of the package implementation: **R** if the full core algorithm is re-used from other libraries, **S** if implemented from scratch and no literature description of the algorithm was available, and **L** if implemented following existent code or pseudo-code proposed in literature.

5.4 Supporting Functions and Utilities

In `utils`, we have two modules, regarding input and output operations. We present some useful functions as tools to help in the transformation of the time series.

Besides the functions available in `utils`, we also have some auxiliary in the classes we mentioned before, specially in the `core` classes, to help in the tasks of feature extraction and visualisation of the generated networks.

We provide some functions relying on the `igraph` and `pymnet` functionalities, but the user can access the class attribute `G` and perform extra available operations on the packages applied to the object.

We will expand and attempt to show the usefulness and customisation ability of these functions, in the analysis of different time series, on Chapter 6.

5.4.1 Input and Output Operations

We will proceed to describe the functions available in the two `utils` modules, for input and output operations.

- `read_csv`: reads data from a CSV file and returns it in a list of lists format, where each sub-list corresponds to a column in the CSV file. It has the option to skip a header, if necessary.
- `suppress_output`: acts as an auxiliary function to deal with prints from other libraries when calling our methods.
- `as_netx`: given a child class of `Network`, it transforms the attribute `G`, that uses an `igraph` Graph representation, returning a `NetworkX` Graph structure.

5.4.2 Feature Extraction Functions

To extract network features, meaningful for the analysis of the complex system, we provide functions for both networks generated from univariate time series, and multivariate time series, that use existent `igraph` and `pymnet` functions, adjusted to the type of network in question.

Network and SinglelayerNetwork functions:

- `show_edges`: shows the graph edges in format `(node1, node2)`.
- `show_nodes`: shows the graph nodes with their `name` attribute.
- `get_adjacency`: calculates the adjacency matrix of A the graph.
- `get_avg_path_length`: calculates the average path length of the graph, considering whether it is directed and if it allows for unconnected nodes.
- `get_betweenness`: calculates the betweenness centrality of the nodes, considering direction and weights.
- `get_closeness`: calculates the closeness centrality of the nodes, with options for mode, normalisation, and weights.
- `get_degree`: returns the degree of the nodes in the graph, with optional parameters for mode and loops.
- `get_density`: calculates the density of the graph, including loops.
- `get_diameter`: calculates the diameter of the graph, taking into account direction and weights.
- `get_shortest_path_length`: calculates the shortest path length between nodes, with optional parameters for source, target, and mode.

MultilayerNetwork functions:

- `show_edges`: shows the graph edges in format `(node1, node2, layer1, layer2, aspect)`.
- `show_nodes`: shows the graph nodes in format `(node, layer)`.
- `get_adjacency`: calculates the adjacency matrix of the graph, including the inter-layer edges (or not), and returns the supra-adjacency matrix and the list of node-layer pairs.
- `get_degree`: returns a dictionary with the degree of the nodes in the graph, either as a pair with node name as key and degree as value, or as a degree distribution.
- `get_density`: calculates the density of the graph, considering the all possible inter and intra-layer edges.
- `get_diameter`: calculates the diameter of the graph.
- `get_shortest_path_length`: calculates the shortest path lengths between nodes, with optional parameters for source and target.

5.4.3 Visualisation Tools

Each class includes an associated `plot_graph` function, which leverages existing functions from the `matplotlib` library and the graph structure package used, either `igraph` or `pymnet`. This function comes with default settings chosen by us but allows the user to customise them by passing available parameters according to the respective libraries' documentation.

We also developed the `plot_time_series` tool, as a simple way to plot the time series observations across time, available for all univariate and multivariate time series mapping methods.

Additionally, for all univariate visibility methods, we have created a visualisation tool named `plot_visibility_lines`. This tool distinguishes between lines that penetrate other observations (dashed lines) and lines that don't (solid lines). It also indicates direction when the graph is directed. For CVG, a similar procedure called `plot_visibility_arcs` was created, with the same purpose and functionalities.

The `plot_visibility_lines` function might vary according to the method being used. For instance, in DVG, we also plot the deleted edges from the corresponding NVG as blue dashed lines. In the WDPVG and Weighted Dual Perspective HVG, lines are red when obtained from the normal perspective and blue when generated using the reflected perspective.

For the WDPVG, we have also added `plot_normal_perspective_lines` and `plot_reflective_perspective_lines`. The former plots the normal perspective lines with the original time series, while the latter plots the inverted time series along with the visibility lines in blue.

For univariate transition methods like Coarse-Grained Phase Space Graph and QG, we have enabled the visualisation of the time series with the appropriate bins using `plot_time_series_with_bins`. As for the Visibility Graphlets Network, we created `plot_motifs` to visualise the different unique motifs encountered from the NVGs of the embedded time series vectors.

Finally, for the Cycle Graph, we make possible the visualisation of the time series observations, coloured by the cycle they belong to, using `plot_time_series_by_cycle`.

Chapter 6

Applications and Analysis

In this chapter, we begin by demonstrating how to use our framework to map time series into networks and how to create visualisations using `matplotlib`, by providing practical Python code examples, showcasing the essential steps and functions involved.

Following the initial demonstration, we generate univariate and multivariate time series data using the statistical models described in Section 2.1. Using this data, we create networks according to different mapping methods. Subsequently, we perform feature extraction from these networks and generate visualisations to analyse and interpret the results.

6.1 Usage

The `pyTSMnet` package will be publicly available on GitHub, and on the PyPI, making it accessible via pip installation, for easier integration into different tasks.

Using our package is straightforward. The first step involves initialising the desired class with the appropriate parameters. Then, calling the `build` method to populate the previously initialised empty graph. The following code snippet illustrates this process:

```
from pytsmnet.univariate.visibility import natural, horizontal
from pytsmnet.utils.inputs import read_csv
import matplotlib.pyplot as plt
import os

5
# Read the data
filename = os.path.join("data", "uts_example.csv")
xs, ts = read_csv(filename)

10
# Build the visibility graph
instance = horizontal.HorizontalVisibilityGraph(ts, xs, direction="
    left_to_right").build()
```

```

# Create plot
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
15
instance.plot_visibility_lines(ax=ax1)
ax1.set_title("Horizontal Visibility Lines")

instance.plot_graph(ax=ax2, layout="kk")
20 ax2.set_title("Horizontal Visibility Graph")

plt.savefig("examples/hvg.png")
plt.show()
plt.close()

```

Listing 6.1: Creation of Horizontal Visibility Graph

Furthermore, one can extract and visualise the degree distribution, as shown below:

```

k_values = instance.get_degree()

# Create plot
fig, ax = plt.subplots()
5
ax.hist(k_values)
ax.set_title("Degree Distribution of HVG")

plt.savefig("examples/hvg_degree.png")
10 plt.show()
plt.close()

```

Listing 6.2: Extraction of degree distribution from the Horizontal Visibility Graph

Finally, we also present how to produce visualisation for one of the multivariate mapping methods: the Multiplex Recurrence Network. It is essential, when dealing with the multi-layer methods, to ensure the `matplotlib` backend is interactive, before importing the `matplotlib.pyplot` capabilities (that is, if the user wants to call the `pyplot.show` function). We also recommend doing this step after importing the `pyTSMnet` modules, so there is no room for errors.

```

from pytsmnet.multivariate.multi_layer import multiplex
import matplotlib
matplotlib.use("TkAgg")
import matplotlib.pyplot as plt
5

# Bivariate Time Series
mts = [[0.35,0.90,0.30,0.80,0.60,0.95,0.10,0.40,0.90,0.15],
        [0.50,0.70,0.40,0.50,0.60,0.45,0.50,0.40,0.70,0.60]]

# Build the network
10 instance = multiplex.MultiplexNetwork(mts, layer_method='recurrence', dim=3,
    tau=1, threshold=0.8).build()

```

```

# Plot the network
fig = plt.figure()
15 ax = fig.add_subplot(111, projection='3d')
    instance.plot_graph(ax=ax, layerColorDict={1: 'blue', 2: 'red'},
        defaultEdgeWidth=0.5, layerLabelRule={}, nodeLabelRule={})

output_file = 'examples/mrn_graph.png'
plt.show()
20 plt.savefig(output_file)
plt.close()

```

Listing 6.3: Creation of Multiplex Recurrence Network

For the initialisation of other methods, a similar logic should be followed. Additionally, for more detailed examples and usage instructions, anyone will be able to refer to the documentation available on the GitHub repository.

6.2 Univariate Time Series Data

For our analysis process, we used a subset of the simulated data used in [45], that was formulated according to the processes described in Table 6.1.

Process	Formula	Main Property
WN	$\epsilon_t \sim N(0, 1)$	Noise effect
AR(2)	$y_t = 1.5y_{t-1} - 0.75y_{t-2} + \epsilon_t$	Pseudo-periodic
ARIMA(1, 1, 0)	$(1 - 0.7B)(1 - B)y_t = \epsilon_t$ ¹	Stochastic trend
GARCH(1, 1)	$\sigma_t^2 = 10^{-6} + 0.8\sigma_{t-1}^2 + 0.1\epsilon_{t-1}^2$	Persistent periods of high or low volatility
INAR(1)	$y_t = 0.5 * y_{t-1} + \epsilon_t, \epsilon_t \sim Po(1)$	Correlated counts

Table 6.1: Summary of synthetic univariate time series processes: parameters and the main characteristic of the data. *See [45] for more details.*

For each type of process, we generated two sets of time series data: one with a length of $n = 100$ and another with a length of $n = 1000$. Depending on the task, different time series were used:

- **Visualisation:** We used the time series with length $n = 100$. This shorter length is preferred, as it reduces the amount of data displayed, resulting in clearer, less cluttered graphs. Simplified data visualisations make patterns and trends more discernible.
- **Feature Extraction:** We utilised the time series of length $n = 1000$. The longer length provides a more detailed dataset, which is crucial for extracting robust features.

¹ B represents the backshift operator, where $BY_t = Y_{t-1}$.

We decided to use mappings obtained from all univariate methods. We used most of the default options for all the class parameters, but we detail the parameter selection in the following items, with the changed parameters in bold:

1. `NaturalVisibilityGraph`: `direction = None`, `weight = None`, `min_weight = None`, `max_weight = None`, `penetrable_limit = 0`.
2. `HorizontalVisibilityGraph`: `direction = None`, `weight = None`, `min_weight = None`, `max_weight = None`, `penetrable_limit = 0`.
3. `CircularVisibilityGraph`: `alpha=100`, `directed=False`, `penetrable_limit=0`.
4. `AbsoluteInvisibilityGraph`: `directed=False`.
5. `DifferenceVisibilityGraph`: `direction=None`, `weight=None`, `min_weight=None`, `max_weight=None`, `penetrable_limit=0`.
6. `CoarseGrainedPhaseSpaceGraph`: `thresholds={-100, -30, -10, -2, -0.1, 0, 0.1, 2, 10, 30, 100}`, `directed=True`, `weighted=True`, `dim=3`, `tau=1`.
7. `VisibilityGraphletsNetwork`: `directed=True`, `weighted=True`, `dim=5`, `tau=1`, `first_occurrence_label=True`.
8. `OrdinalPartitionsTransitionNetwork`: `directed=True`, `weighted=True`, `dim=3`, `tau=1`.
9. `QuantileGraph`: `directed=True`, `weighted=True`, `Q=4`, `method='linear'`.
10. `CycleNetwork`: `directed=False`, `weighted=False`, `threshold=0.7`, `method = 'correlation'`, `cycles=None`, `radius=5`.
11. `CorrelationNetwork`: `directed=False`, `weighted=False`, `dim=3`, `tau=3`, `threshold=0.7`, `method='pearson'`.
12. `RecurrenceNetwork`: `directed=False`, `weighted=False`, `metric='supremum'`, `normalize=False`, `missing_values=False`, `silence_level=0`, `threshold=0.7`, `threshold_std=None`, `recurrence_rate=None`, `local_recurrence_rate=None`, `adaptive_neighborhood_size=None`, `dim=3`, `tau=3`, `node_weights=None`.

6.2.1 Feature Extraction

We decided to analyse the degree distribution. The plots obtained are available in Figures 6.1, 6.2, and 6.3. We also computed the network diameter, as well as the density, available in Table 6.2.

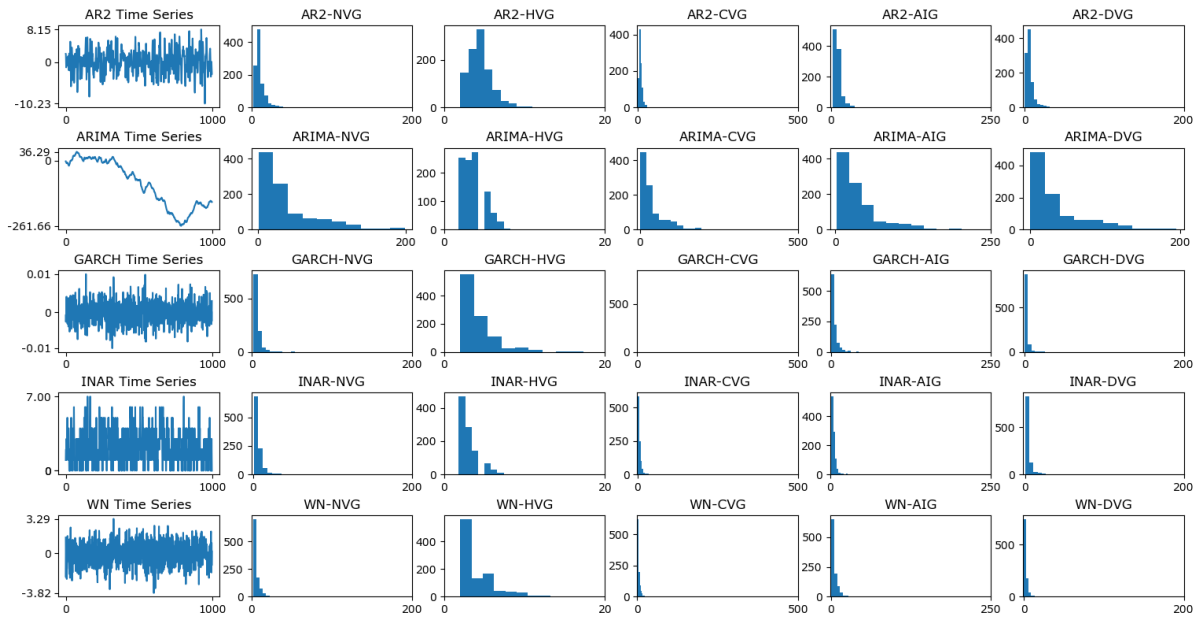


Figure 6.1: Network degree distributions extracted from several graphs obtained by different visibility mapping methods.

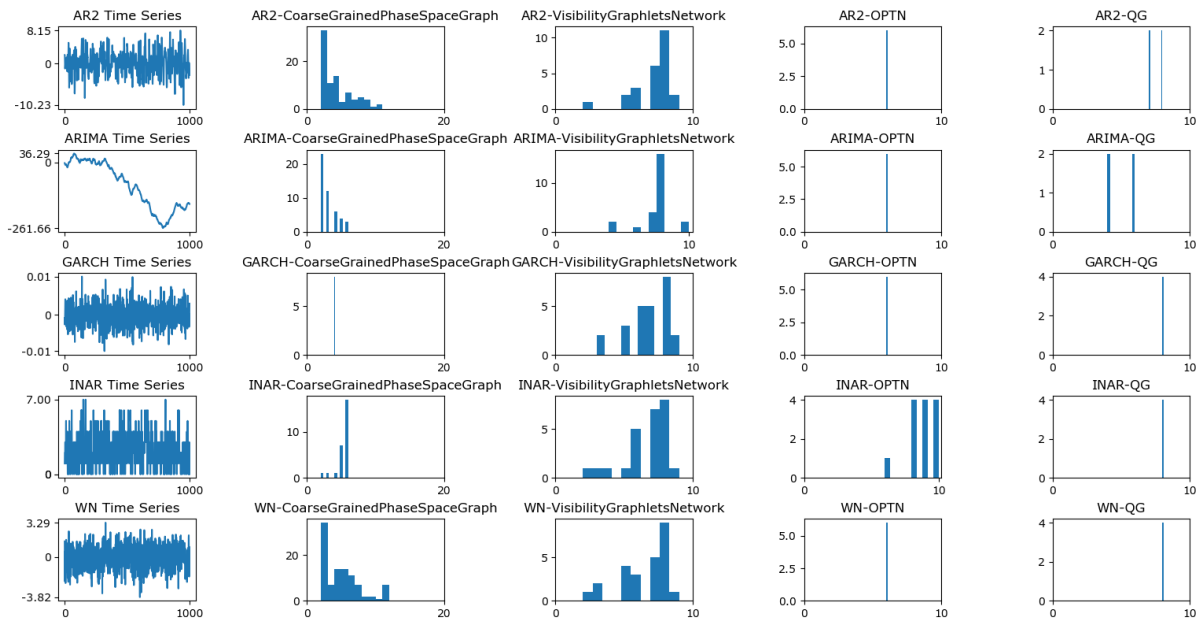


Figure 6.2: Network degree distributions extracted from several graphs obtained by different transition mapping methods.

	AR(2)	ARIMA	GARCH	INAR	WN
NVG	$D = 0.00991$ $d = 10$	$D = 0.03837$ $d = 9$	$D = 0.00607$ $d = 9$	$D = 0.00638$ $d = 11$	$D = 0.00588$ $d = 10$
HVG	$D = 0.00394$ $d = 29$	$D = 0.00361$ $d = 245$	$D = 0.00396$ $d = 18$	$D = 0.00293$ $d = 33$	$D = 0.00397$ $d = 18$
CVG	$D = 0.00968$ $d = 14$	$D = 0.03713$ $d = 9$	$D = 0.00221$ $d = 887$	$D = 0.00629$ $d = 23$	$D = 0.00567$ $d = 18$
AIG	$D = 0.00977$ $d = 10$	$D = 0.03723$ $d = 10$	$D = 0.00587$ $d = 11$	$D = 0.00525$ $d = 109$	$D = 0.00585$ $d = 10$
DVG	$D = 0.00596$ $d = \infty$	$D = 0.03475$ $d = \infty$	$D = 0.00211$ $d = \infty$	$D = 0.00344$ $d = \infty$	$D = 0.00190$ $d = \infty$
Coarse-Grained Phase Space Graph	$D = 0.02466$ $d = 37$	$D = 0.03125$ $d = \infty$	$D = 0.25000$ $d = 206$	$D = 0.10014$ $d = 54$	$D = 0.02316$ $d = 24$
Visibility Graphlets Network	$D = 0.14240$ $d = 28$	$D = 0.15200$ $d = 44$	$D = 0.13440$ $d = 35$	$D = 0.13280$ $d = 51$	$D = 0.12960$ $d = 40$
OPTN	$D = 0.50000$ $d = 108$	$D = 0.50000$ $d = 73$	$D = 0.50000$ $d = 124$	$D = 0.33728$ $d = 55$	$D = 0.50000$ $d = 117$
QG	$D = 0.93750$ $d = 69$	$D = 0.62500$ $d = 14$	$D = 1.00000$ $d = 67$	$D = 1.00000$ $d = 63$	$D = 1.00000$ $d = 72$
Cycle Network	$D = 0.80000$ $d = 2$	$D = 0.67015$ $d = 2$	$D = 0.44017$ $d = \infty$	$D = 0.32933$ $d = \infty$	$D = 0.43011$ $d = 2$
Correlation Network	$D = 0.25973$ $d = 4$	$D = 0.34818$ $d = 5$	$D = 0.25217$ $d = 5$	$D = 0.22196$ $d = \infty$	$D = 0.25231$ $d = 5$
Recurrence Network	$D = 0.00279$ $d = \infty$	$D = 0.00022$ $d = \infty$	$D = 0.99799$ $d = 1$	$D = 0.00886$ $d = \infty$	$D = 0.05044$ $d = \infty$

Table 6.2: Topological features, density (**D**) and diameter (**d**), extracted from networks generated using univariate mapping methods.

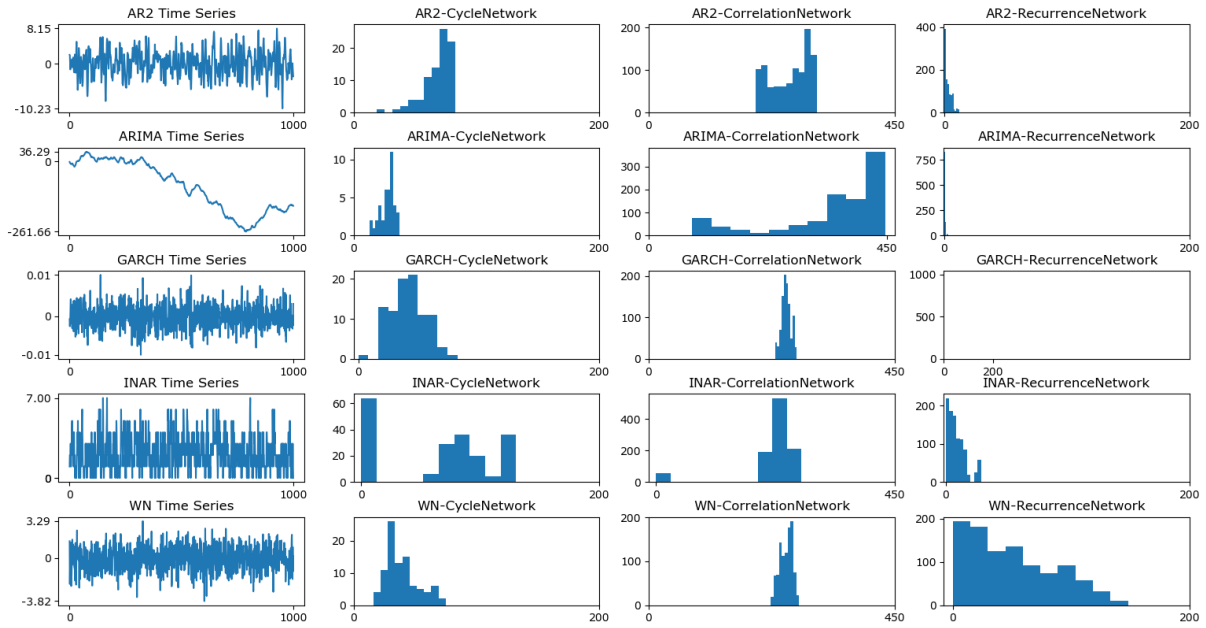


Figure 6.3: Network degree distributions extracted from several graphs obtained by different proximity mapping methods.

6.2.2 Visualisation

All visualisations were obtained by using the `plot_graph` function, and attributing the Kamada-Kawai algorithm as the layout parameter. They are available in Figures 6.4, 6.5 and 6.6.

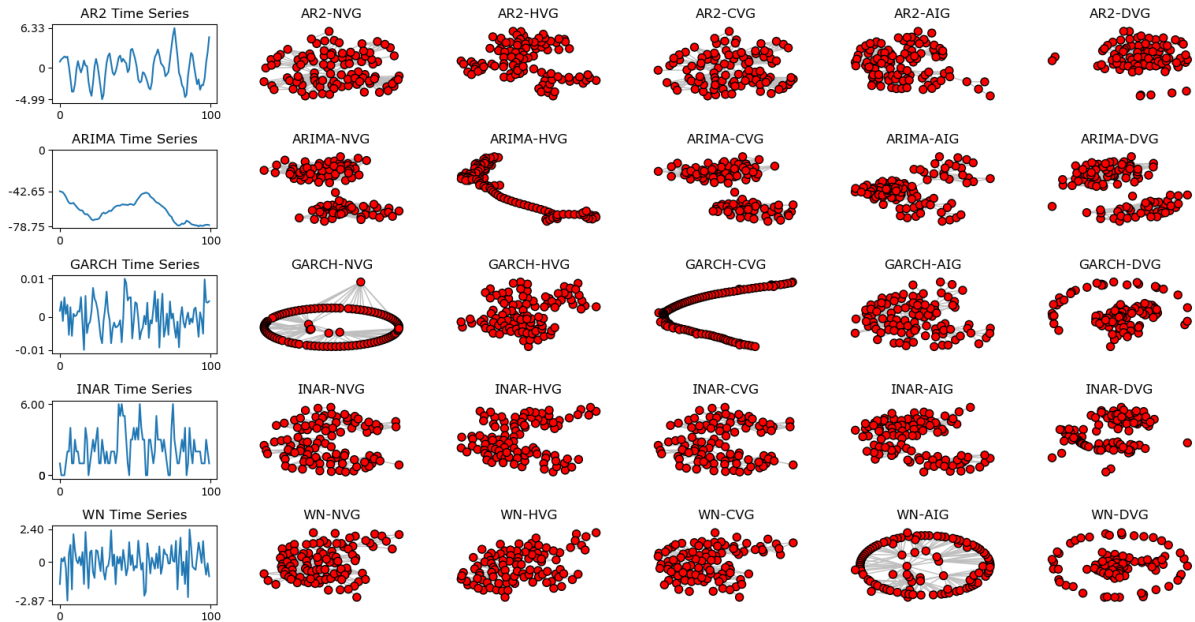


Figure 6.4: Visualisation for the several graphs obtained by visibility mapping methods.

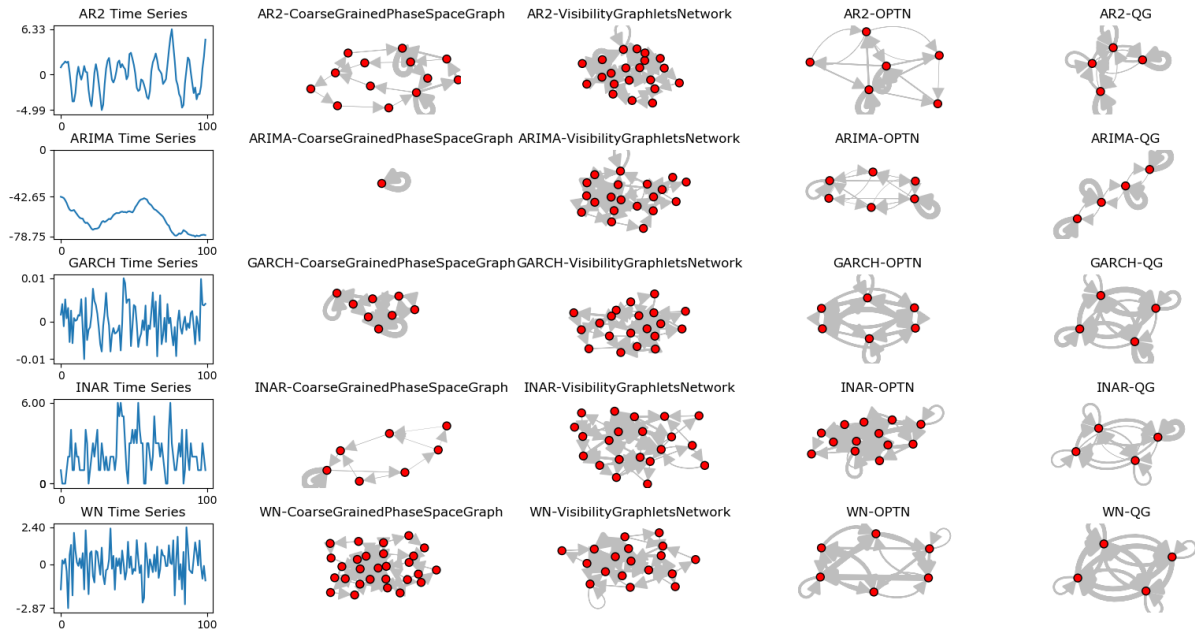


Figure 6.5: Visualisation for the several graphs obtained by transition mapping methods.

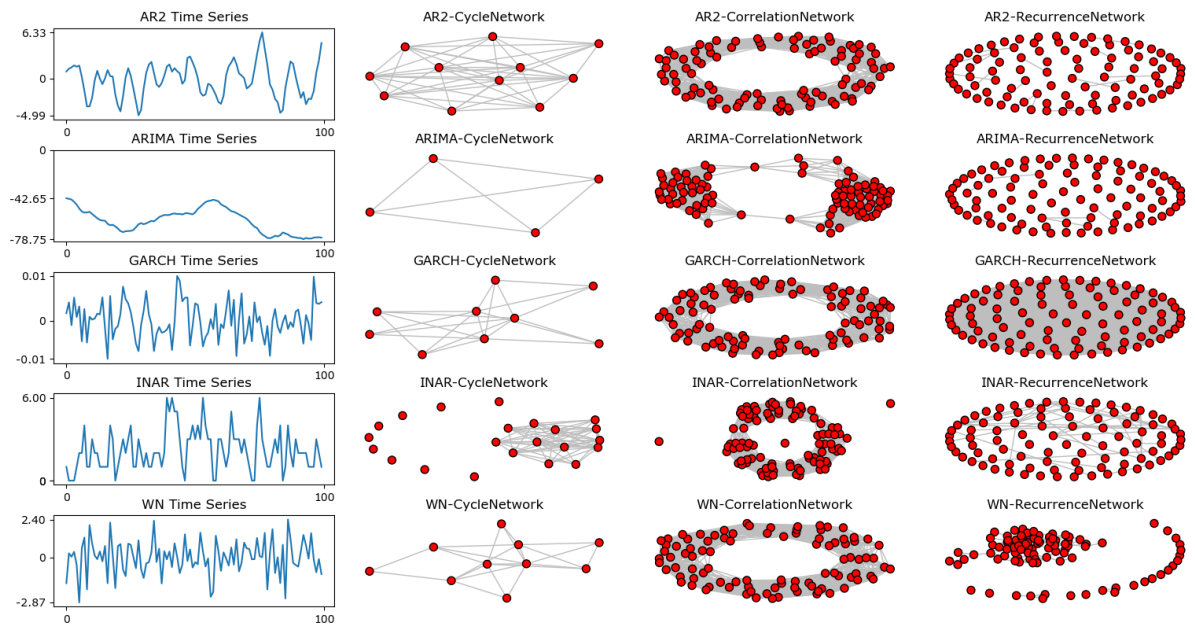


Figure 6.6: Visualisation for the several graphs obtained by proximity mapping methods.

6.2.3 Discussion

By combining the information extracted from the features and the shapes of the generated networks, we find interesting trends among different mapping methods. For example, the QG tends to be, or almost be, fully connected for all univariate time series processes, except for ARIMA, which also has a different shape: instead of a circular, all connected network, we have a chain of consecutive states. However, this network has a smaller diameter, in comparison to the other models. These results suggest that the QG is the method that leads to more dense networks, more consistently.

On the other hand, all VGs seem to constitute sparse networks, but with particular notice for the ARIMA time series, which generally leads to denser networks than the other processes, with the exception of the HVG. The structure and diameter of the ARIMA-HVG combination is also different, with a chain of successive observations between two clusters, and a much higher diameter. In fact, all VGs visualisations for the ARIMA series seem to present two clusters of data, with some nodes in between.

Most of the VGs degree distributions seem to fit an exponential distribution. Because of the *alpha* value used, the CVG showed similar properties to the NVG, except for the GARCH case, which may be related to the smaller time series values $y_t \in [-0.01, 0.01]$, and the method's sensitivity to the parameter choice. The diameter also showed an irregular value compared to the other cases, for the same mapping method. The structure seems to be also a chain of observations.

In terms of irregular results for the diameter, the combination AIG-INAR also stands out, with a superior value.

The Correlation Network for the ARIMA model also leads to higher density and to a different degree distribution, than the others, who present a Gaussian distribution. The diameter of Correlation Networks seems to be consistently low, except for the INAR case. Correlation Networks all seem to lead to a ring-like shaped structure.

An interesting case relates to the Recurrence Network. It seems all cases are sparse networks, except the one mapped from the GARCH time series, which is almost fully connected, with a diameter of 1. Obviously, for these types of networks, the choice of parameters is highly influential on the results, and what works for one time series process, might not work for others.

For the Coarse-Grained Phase Space Graph, the degree distributions are not really consistent across processes, nor the diameters. But we used the same `thresholds` parameter for all, which it may not be ideal for all time series.

The OPTN for the INAR series shows a different degree distribution comparing to the other processes: there are various degrees possible, whereas in the other, only one value for the degree is present.

Obviously, all networks with isolated nodes presented an infinite diameter. For the parameters chosen, the networks who present higher number of isolated nodes were the DVGs (as expected), and Recurrence Networks (again, because of the parameters).

6.3 Multivariate Time Series Data

The data used for our analysis process corresponds to the same data used in [46], and was formulated according to the models described in Table 6.3, for the bivariate time series data.

Process	Formula	Main Properties
iVWN	$\epsilon_t \sim \mathcal{N}(0, 1)$	Noise effect No correlation
cVWN	$\begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2,t} \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} 1.00 & 0.86 \\ 0.86 & 1.50 \end{bmatrix}\right)$	Noise effect No serial correlation Cross-correlation
sVAR	$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.70 & 0.02 \\ 0.30 & 0.80 \end{bmatrix} \begin{bmatrix} y_{1,t-1} \\ y_{2,t-1} \end{bmatrix} + \begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2,t} \end{bmatrix},$ $\epsilon_t \sim \begin{bmatrix} 1.00 & 0.86 \\ 0.86 & 1.50 \end{bmatrix}$	Strong correlation (serial and cross, lagged and contemporaneous)
sGARCH	$\begin{bmatrix} \sigma_{11,t} \\ \sigma_{22,t} \end{bmatrix} = \begin{bmatrix} 0.05 \\ 0.02 \end{bmatrix} + \begin{bmatrix} 0.10 & 0.00 \\ 0.00 & 0.05 \end{bmatrix} \begin{bmatrix} \epsilon_{1,t-1}^2 \\ \epsilon_{2,t-1}^2 \end{bmatrix} +$ $\begin{bmatrix} 0.85 & 0.00 \\ 0.00 & 0.88 \end{bmatrix} \begin{bmatrix} \sigma_{11,t} \\ \sigma_{22,t} \end{bmatrix}, \epsilon_t \sim \begin{bmatrix} 1.00 & 0.86 \\ 0.86 & 1.50 \end{bmatrix}$	Strong contemporaneous cross-correlation

Table 6.3: Summary of synthetic bivariate time series processes: parameters and the main characteristic of the data. *See [46] for more details.*

Again, for each type of process, two bivariate time series were produced: one with length $n = 100$, and other with length $n = 1000$, for the same purposes as described for the univariate time series data case.

We decided to analyse only some multivariate networks. We chose the MHVG, the MQG, the Multiplex HVG and the Multiplex QG, for comparison of the inter-layer edges. We used the following classes and respective parameters to create the graph instances:

1. `MultilayerHorizontalVisibilityGraph`.
2. `MultiplexNetwork`: `layer_method = 'horizontal'`, `couplings = 'categorical'`.
3. `MultilayerQuantileGraph`: `directed=False`, `Q=4`, `method='linear'`.
4. `MultiplexNetwork`: `layer_method = 'quantile'`, `couplings = 'categorical'`, `Q=4`, `method='linear'`.

6.3.1 Feature Extraction

We decided to also analyse the degree distribution of the multivariate networks. The plot is available in Figure 6.7. Again, we also computed the graph diameter, as well as the density, available in Table 6.4.

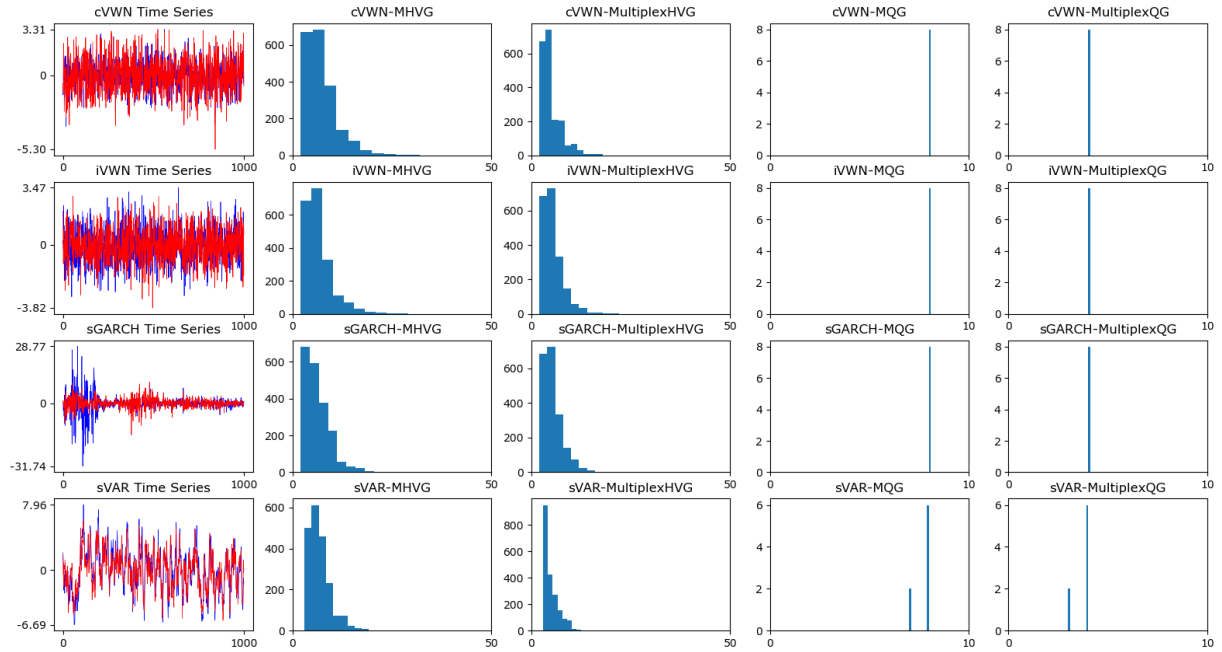


Figure 6.7: Network degree distributions extracted from several graphs obtained by different multivariate mapping methods.

	cVWN	iVWN	sGARCH	sVAR
Multiplex HVG	$D = 0.00248$ $d = 17$	$D = 0.00249$ $d = 11$	$D = 0.00248$ $d = 17$	$D = 0.00248$ $d = 21$
MHVG	$D = 0.00346$ $d = 16$	$D = 0.00333$ $d = 10$	$D = 0.00319$ $d = 16$	$D = 0.00336$ $d = 20$
Multiplex QG	$D = 0.57143$ $d = 2$	$D = 0.57143$ $d = 2$	$D = 0.57143$ $d = 2$	$D = 0.53571$ $d = 2$
MQG	$D = 0.85714$ $d = 2$	$D = 0.85714$ $d = 58$	$D = 0.85714$ $d = 13$	$D = 0.80357$ $d = 2$

Table 6.4: Topological features, density (**D**) and diameter (**d**), extracted from networks generated using multivariate mapping methods.

6.3.2 Visualisation

All visualisations were obtained by using the `plot_graph` function and available in Figure 6.8.

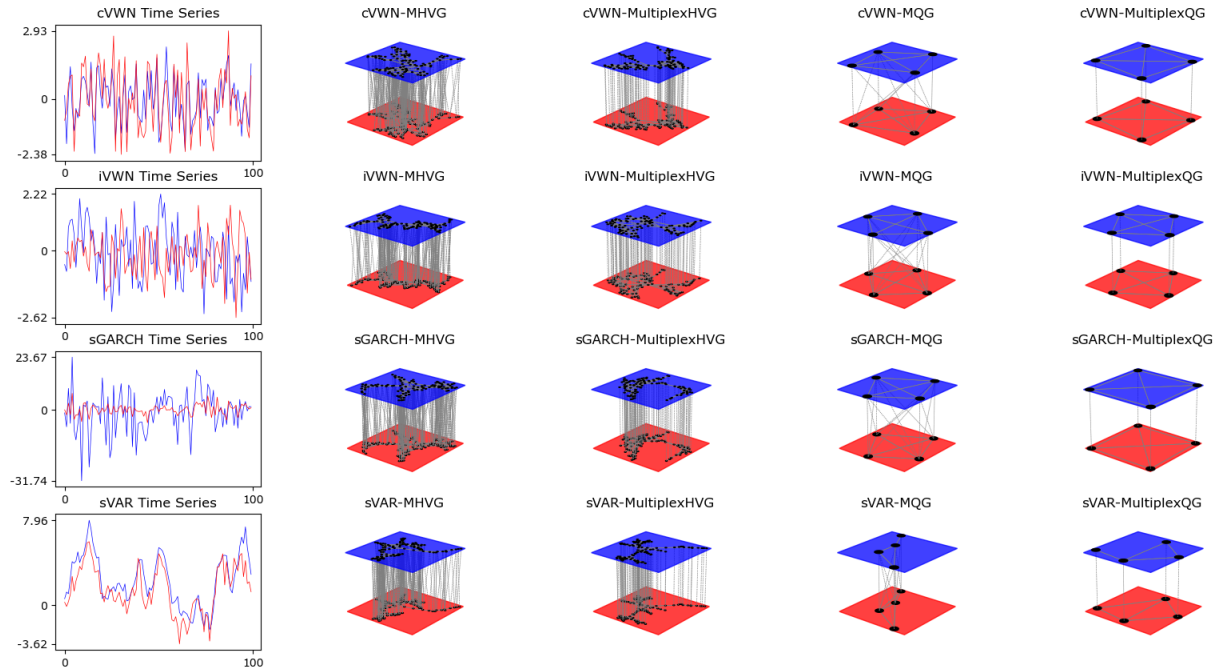


Figure 6.8: Visualisation for the several multiple layers graphs obtained by multivariate mapping methods.

6.3.3 Discussion

The multi-layer methods are slightly more dense than the multiplex counterparts, which was expected, because, while in multiplex we can only link simultaneous nodes, the multi-layer methods allow the possibility of connections between more pairs of nodes, given a certain criteria.

In terms of network diameter, the two types of multi-layer HVGs seem to present similar results. However, in the QG case, we see consistent results for the Multiplex QG ($d = 2$), while for the MQG case, the results for the iVWN and sGARCH series are much higher than for the other two series.

Chapter 7

Conclusions

Time series analysis is often limited when dealing with complex data types, such as high-dimensional, high-frequency, and non-stationary time series, as well as time series with other non-trivial characteristics. Mapping these time series data into complex networks is a complementary approach to traditional analysis and a promising option for dealing with such complex data. It allows us to leverage network science concepts and develop new approaches to uncover insights from the original data, either in univariate or multivariate format.

Univariate time series mapping methods can be broadly classified into three categories: visibility, transition and proximity. Some of the specifics of each concept are as follows:

- **Visibility Methods:** can reflect both local and global properties of the time series; do not require pre-processing of the data; can be used parameter-free, making them a very intuitive and easy to use option.
- **Transition Methods:** effective in identifying dynamic behaviour and state changes within the time series; great at reducing highly-dimensional time series, by partitioning the data, but require a good choice of parameters, in order to balance the loss of information.
- **Proximity Methods:** able to emphasise the relationship between similar time points; highly specific for mapping dynamical systems; even more parameter-sensitive than the transition methods.

When dealing with multivariate time series, the complexity increases, as we must account for the dependencies between multiple time series components. Thus, choosing the right structure for the network representation becomes crucial for this process. The three primary structures considered are single-layer, multiplex, and multi-layer networks. The main advantages and characteristics of each are:

- **Single-Layer Methods:** focuses on the relationships between different time series components, both contemporary and lagged; depending on the method, might need pre-processing.

- **Multiplex Methods:** allows the reuse of univariate methods' concepts; provides a simple way to compare dependencies within different time series components.
- **Multi-Layer Methods:** suitable for more complex, high-level systems; allows the preservation of individual characteristics while capturing interactions between layers (i.e., time series components).

Knowing the strengths and weaknesses of each method is crucial for selecting the most suitable approach to model our system and data effectively. By understanding the unique advantages and limitations of visibility, transition, proximity methods, as well as single-layer, multi-layer, and multiplex structures, we can ensure a more accurate and meaningful representation of the underlying dynamics of our data.

We will now reflect on whether we were able to achieve our initial goals for this package, as described in Section 1.2. By revisiting these objectives, we can assess the effectiveness of our implementation, while also helping identifying areas for improvement and potential future developments of our package.

7.1 Main Contributions

We successfully implemented all univariate time series mapping methods proposed in the literature, either by using existing libraries (namely, `ts2vg`, `PyIOmica`, `pyunicorn`) or by developing algorithms from scratch. For multivariate time series, we completed all multi-layer implementations and even extended multiplex network concepts. However, we were only able to implement the Correlation Network method among the single-layer approaches.

Overall, `pyTSMnet` provides a comprehensive range of time series mapping techniques in a single, object-oriented package, designed to enhance efficiency and usability. Key strengths include:

- **Development of Novel Algorithms:** we created and presented algorithms from scratch, for methods like Circular Visibility Graph, Absolute Invisibility Graph, Visibility Graphlets Network, Multilayer Horizontal Visibility Graph, and Multilayer Quantile Graph, among others following the mathematical foundations proposed in the literature.
- **Unification of Existing Tools:** we integrated implementations from `ts2vg`, `PyIOmica`, and `pyunicorn` for various visibility and recurrence networks.
- **Package Organisation:** we structured the package using an object-oriented approach, maintaining consistent structures for different methods. The hierarchy of the modules and classes is organised based on the dimensionality of the time series, the network structure, and the mapping concepts applied. This allows the user to find and experiment with different methods intuitively.

- **Enhanced Flexibility:** we expanded existing mapping concepts to allow for additional parameters, like, for example, direction and weights. Additionally, we also altered the AIG concept, by using the horizontal visibility algorithm, and, thus, creating the Horizontal AIG.
- **Algorithmic Optimisation:** we tried to utilise efficient data structures and operations from NumPy and SciPy, whenever possible, ensuring optimised performance and memory usage across various algorithms.
- **Visualisation Capabilities:** we developed visualisation functions using `igraph` and `pymnet` to facilitate better analysis and interpretation of the generated networks, while also allowing further customisation from the user.
- **Interoperability:** we provided tools for converting to formats compatible with popular graph libraries like `NetworkX`, `igraph`, and `pymnet`. Additionally, we also comprehend methods to read data from CSV files.
- **Comprehensive Documentation:** we offer extensive documentation and varied examples to ensure ease of use for all functionalities, including time series mappings, feature extraction, and visualisation tasks.

7.2 Future Work

The `pyTSMnet` package is a valuable contribution to the analysis of time series through complex networks, providing a unified and user-friendly tool for researchers across all fields. While the package has successfully met most of the original expectations, some goals were not fully realised, leaving room for future software improvements and extensions.

One of the identified possible improvements involves optimising the computational complexity of certain methods, including the MHVG, AIG (both the original and horizontal variations), and the CVG. Although these methods are functional and follow the literature’s plan, their current implementations could benefit from more efficient algorithms, such as DC strategies, to reduce their time complexity. Additionally, incorporating parallelisation techniques into the methods, that use DC algorithms, could significantly reduce runtime, especially when working with larger datasets.

A possibility for optimising computational efficiency would be the implementation of the package’s backend functions in lower-level programming languages such as C or C++. This approach is commonly used by other libraries like NumPy and `ts2vg`. By leveraging C or C++, we can achieve faster execution times and better memory management. The core computational functions could be rewritten in C, and then interfaced with Python using tools like Cython [2], maintaining the ease of use of the package while significantly boosting its performance.

Another option to explore, aimed at optimizing memory efficiency, would be using sparse matrices (`NumPy` data structure) to store the adjacency matrices of the graphs, as certain methods or parameter choices could lead to sparser networks.

Another crucial avenue for future development is expanding the scope of the package, by adding missing mapping methods. In particular, we aim to complete the `multivariate/single_layer` module by incorporating the following methods: Long-Run Variance Decomposition Network, Causal Effect Network, Ordinal Partition Transition Network, Pattern Interdependent Network, Inter-System Recurrence Network, and Joint Recurrence Network.

Furthermore, we aim to offer even more customisation options, such as making thresholds optional in certain methods, expanding parameter settings for the AIG and CVG methods and allowing directions and different time series components' lengths in the multi-layer methods.

Finally, we also intend to extend the visualisation and feature extraction functionalities of the package, exploring opportunities to provide more comprehensive insights and facilitate deeper analysis of time series data, as well as increase interoperability with tools such as Gephi [1], for example, by providing an export as `gml` option.

Bibliography

- [1] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: an open source software for exploring and manipulating networks. In *Proceedings of the international AAAI conference on web and social media*, volume 3, pages 361–362, 2009.
- [2] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2): 31–39, 2010.
- [3] Carlos Bergillos Varela. A study of visibility graphs for time series representations. B.S. thesis, Universitat Politècnica de Catalunya, 2020.
- [4] IV Bezsudnov and AA Snarskii. From the time series to the complex networks: The parametric natural visibility graph. *Physica A: Statistical Mechanics and its Applications*, 414:53–60, 2014.
- [5] Filippo Maria Bianchi, Lorenzo Livi, Cesare Alippi, and Robert Jenssen. Multiplex visibility graphs to investigate recurrent neural network dynamics. *Scientific reports*, 7(1):44037, 2017.
- [6] Stefano Boccaletti, Ginestra Bianconi, Regino Criado, Charo I Del Genio, Jesús Gómez-Gardenes, Miguel Romance, Irene Sendina-Nadal, Zhen Wang, and Massimiliano Zanin. The structure and dynamics of multilayer networks. *Physics reports*, 544(1):1–122, 2014.
- [7] Katy Börner, Soma Sanyal, Alessandro Vespignani, et al. Network science. *Annu. rev. inf. sci. technol.*, 41(1):537–607, 2007.
- [8] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.
- [9] Elizabeth Bradley and Holger Kantz. Nonlinear time-series analysis revisited. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(9), 2015.
- [10] Andriana SLO Campanharo, M Irmak Sirer, R Dean Malmgren, Fernando M Ramos, and Luís A Nunes Amaral. Duality between time series and networks. *PloS one*, 6(8):e23378, 2011.

- [11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.
- [12] Gabor Csardi and Tamas Nepusz. The igraph software. *Complex syst*, 1695:1–9, 2006.
- [13] Francis X Diebold and Kamil Yilmaz. On the network topology of variance decompositions: Measuring the connectedness of financial firms. *Journal of econometrics*, 182(1):119–134, 2014.
- [14] Sergii Domanskyi, Carlo Piermarocchi, and George I Mias. Pyiomica: longitudinal omics analysis and trend identification. *Bioinformatics*, 36(7):2306–2307, 2020.
- [15] Jonathan F Donges, Jobst Heitzig, Boyan Beronov, Marc Wiedermann, Jakob Runge, Qing Yi Feng, Liubov Tupikina, Veronika Stolbova, Reik V Donner, Norbert Marwan, et al. Unified functional network and nonlinear time series analysis for complex systems science: The pyunicorn package. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(11), 2015.
- [16] Reik V Donner, Yong Zou, Jonathan F Donges, Norbert Marwan, and Jürgen Kurths. Recurrence networks—a novel paradigm for nonlinear time series analysis. *New Journal of Physics*, 12(3):033025, 2010.
- [17] Reik V Donner, Michael Small, Jonathan F Donges, Norbert Marwan, Yong Zou, Ruoxi Xiang, and Jürgen Kurths. Recurrence-based time series analysis by means of complex network methods. *International Journal of Bifurcation and Chaos*, 21(04):1019–1046, 2011.
- [18] Randal Douc, Eric Moulines, and David Stoffer. *Nonlinear Time Series: Theory, Methods and Applications with R Examples*. Chapman and Hall/CRC, 1 edition, 2014.
- [19] Victor M Eguiluz, Dante R Chialvo, Guillermo A Cecchi, Marwan Baliki, and A Vania Apkarian. Scale-free brain functional networks. *Physical review letters*, 94(1):018102, 2005.
- [20] Deniz Eroglu, Norbert Marwan, Martina Stebich, and Jürgen Kurths. Multiplex recurrence networks. *Physical Review E*, 97(1):012312, 2018.
- [21] Philippe Esling and Carlos Agon. Time-series data mining. *ACM Computing Surveys (CSUR)*, 45(1):1–34, 2012.
- [22] Jan H Feldhoff, Reik V Donner, Jonathan F Donges, Norbert Marwan, and Jürgen Kurths. Geometric detection of coupling directions by means of inter-system recurrence networks. *Physics Letters A*, 376(46):3504–3513, 2012.
- [23] JH Feldhoff, Reik V Donner, Jonathan Friedemann Donges, Norbert Marwan, and Jürgen Kurths. Geometric signature of complex synchronisation scenarios. *Europhysics Letters*, 102(3):30007, 2013.

- [24] Leonardo N Ferreira. From time series to networks in r with the ts2net package. *Applied Network Science*, 9(1):32, 2024.
- [25] Zhong-Ke Gao, Qing Cai, Yu-Xuan Yang, Wei-Dong Dang, and Shan-Shan Zhang. Multiscale limited penetrable horizontal visibility graph for analyzing nonlinear time series. *Scientific reports*, 6(1):35622, 2016.
- [26] Zhongke Gao and Ningde Jin. Complex network from time series based on phase space reconstruction. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 19(3), 2009.
- [27] Aric Hagberg, Pieter J Swart, and Daniel A Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.
- [28] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 135–149. SIAM, 2007.
- [29] Tianjiao Kong, Jie Shao, Jiuyuan Hu, Xin Yang, Shiyiling Yang, and Reza Malekian. Eeg-based emotion recognition using an improved weighted horizontal visibility graph. *Sensors*, 21(5):1870, 2021.
- [30] Lucas Lacasa, Bartolo Luque, Fernando Ballesteros, Jordi Luque, and Juan Carlos Nuno. From time series to complex networks: The visibility graph. *Proceedings of the National Academy of Sciences*, 105(13):4972–4975, 2008.
- [31] Lucas Lacasa, Angel Nunez, Édgar Roldán, Juan MR Parrondo, and Bartolo Luque. Time series irreversibility: a visibility graph approach. *The European Physical Journal B*, 85:1–11, 2012.
- [32] Lucas Lacasa, Vincenzo Nicosia, and Vito Latora. Network structure of multivariate time series. *Scientific reports*, 5(1):15508, 2015.
- [33] Xin Lan, Hongming Mo, Shiyu Chen, Qi Liu, and Yong Deng. Fast transformation from time series to visibility graphs. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(8), 2015.
- [34] Bartolo Luque, Lucas Lacasa, Fernando Ballesteros, and Jordi Luque. Horizontal visibility graphs: Exact results for random time series. *Physical Review E*, 80(4):046103, 2009.
- [35] Tarmo Nurmi, Arash Badie-Modiri, Corinna Coupette, and Mikko Kivelä. pymnet: A python library for multilayer networks. *Journal of Open Source Software*, 9(99):6930, 2024.
- [36] Travis E Oliphant et al. *Guide to numpy*, volume 1. Trelgol Publishing USA, 2006.
- [37] Dusty Phillips. *Python 3 object oriented programming*. Packt Publishing Ltd, 2010.

- [38] Henggang Ren, Qianshun Yuan, Sherehe Semba, Tongfeng Weng, Changgui Gu, and Huijie Yang. Pattern interdependent network of cross-correlation in multivariate time series. *Physics Letters A*, 384(30):126781, 2020.
- [39] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021.
- [40] Yijing Ruan, Reik V Donner, Shuguang Guan, and Yong Zou. Ordinal partition transition network based complexity measures for inferring coupling direction and delay from time series. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 29(4), 2019.
- [41] Jakob Runge, Vladimir Petoukhov, Jonathan F Donges, Jaroslav Hlinka, Nikola Jajcay, Martin Vejmelka, David Hartman, Norbert Marwan, Milan Paluš, and Jürgen Kurths. Identifying causal gateways and mediators in complex spatio-temporal systems. *Nature communications*, 6(1):8502, 2015.
- [42] Akshansh Sharma, Firoj Khan, Deepak Sharma, Sunil Gupta, and FY Student. Python: the programming language of future. *Int. J. Innovative Res. Technol*, 6(2):115–118, 2020.
- [43] Robert H Shumway, David S Stoffer, and David S Stoffer. *Time series analysis and its applications*, volume 3. Springer, 2000.
- [44] Vanessa Freitas Silva, Maria Eduarda Silva, Pedro Ribeiro, and Fernando Silva. Time series analysis via network science: Concepts and algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 11(3):e1404, 2021.
- [45] Vanessa Freitas Silva, Maria Eduarda Silva, Pedro Ribeiro, and Fernando Silva. Novel features for time series analysis: a complex networks approach. *Data Mining and Knowledge Discovery*, 36(3):1062–1101, 2022.
- [46] Vanessa Freitas Silva, Maria Eduarda Silva, Pedro Ribeiro, and Fernando Silva. Mhvg2mts: Multilayer horizontal visibility graphs for multivariate time series analysis. *arXiv preprint arXiv:2301.02333*, 2023.
- [47] Vanessa Freitas Silva, Maria Eduarda Silva, Pedro Ribeiro, and Fernando Silva. Multilayer quantile graph for multivariate time series analysis and dimensionality reduction. *International Journal of Data Science and Analytics*, pages 1–13, 2024.
- [48] Michael Small. Complex networks from time series: Capturing dynamics. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2509–2512. IEEE, 2013.
- [49] Michael Small, Jie Zhang, and Xiaoke Xu. Transforming time series into complex networks. In *Complex Sciences: First International Conference, Complex 2009, Shanghai, China, February 23-25, 2009, Revised Papers, Part 2 1*, pages 2078–2089. Springer, 2009.

- [50] Colin Stephen. A scalable linear-time algorithm for horizontal visibility graph construction over long sequences. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 40–50. IEEE, 2021.
- [51] Mutua Stephen, Changgui Gu, and Huijie Yang. Visibility graph based time series analysis. *PloS one*, 10(11):e0143015, 2015.
- [52] Supriya Supriya, Siuly Siuly, Hua Wang, Jinli Cao, and Yanchun Zhang. Weighted visibility graph with complex network features in the detection of epilepsy. *IEEE access*, 4:6554–6566, 2016.
- [53] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.
- [54] William WS Wei. *Multivariate time series analysis and applications*. John Wiley & Sons, 2018.
- [55] Paiheng Xu, Rong Zhang, and Yong Deng. A novel visibility graph transformation of time series into weighted networks. *Chaos, Solitons & Fractals*, 117:201–208, 2018.
- [56] Xiaoke Xu, Jie Zhang, and Michael Small. Superfamily phenomena and motifs of networks induced from time series. *Proceedings of the National Academy of Sciences*, 105(50):19601–19605, 2008.
- [57] Qi Xuan, Jinchao Zhou, Kunfeng Qiu, Dongwei Xu, Shilian Zheng, and Xiaoni Yang. Clpvg: Circular limited penetrable visibility graph as a new network model for time series. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 32(1), 2022.
- [58] Wanfeng Yan and Edgar van Tuyl van Serooskerken. Forecasting financial extremes: a network degree measure of super-exponential growth. *PLoS One*, 10(9):e0128908, 2015.
- [59] Yue Yang and Huijie Yang. Complex network-based time series analysis. *Physica A: Statistical Mechanics and its Applications*, 387(5-6):1381–1386, 2008.
- [60] Delia Fano Yela, Dan Stowell, and Mark Sandler. Spectral visibility graphs: application to similarity of harmonic signals. In *2019 27th European Signal Processing Conference (EUSIPCO)*, pages 1–5. IEEE, 2019.
- [61] Jei Zhang, Xiaodong Luo, and Michael Small. Detecting chaos in pseudoperiodic time series without embedding. *Physical Review E*, 73(1):016216, 2006.

- [62] Jiayang Zhang, Jie Zhou, Ming Tang, Heng Guo, Michael Small, and Yong Zou. Constructing ordinal partition transition networks from multivariate time series. *Scientific reports*, 7(1):7795, 2017.
- [63] Jie Zhang and Michael Small. Complex network from pseudoperiodic time series: Topology versus dynamics. *Physical review letters*, 96(23):238701, 2006.
- [64] Minzhang Zheng, Sergii Domanskyi, Carlo Piermarocchi, and George I Mias. Visibility graph based temporal community detection with applications in biological time series. *Scientific reports*, 11(1):5623, 2021.
- [65] Ting-Ting Zhou, Ning-De Jin, Zhong-Ke Gao, and Yue-Bin Luo. Limited penetrable visibility graph for establishing complex network from time series. *Acta Physica Sinica*, 61(3):030506, 2012.
- [66] Guohun Zhu, Yan Li, and Peng Wen. Analysis and classification of sleep stages based on difference visibility graphs from a single-channel eeg signal. *IEEE journal of biomedical and health informatics*, 18(6):1813–1821, 2014.
- [67] Guohun Zhu, Yan Li, and Peng Paul Wen. Epileptic seizure detection in eegs signals using a fast weighted horizontal visibility algorithm. *Computer methods and programs in biomedicine*, 115(2):64–75, 2014.
- [68] Roberto V Zicari. Big data: Challenges and opportunities. *Big data computing*, 564:103, 2014.
- [69] Yong Zou, Reik V. Donner, Norbert Marwan, Jonathan F. Donges, and Jürgen Kurths. Complex network approaches to nonlinear time series analysis. *Physics Reports*, 787:1–97, 2019.