A study of visibility graphs for time series representations

Carlos Bergillos Varela
Bachelor Degree in Informatics Engineering
Specialization in Computing

supervised by
Argimiro Arratia Quesada
Computer Science Department - UPC

June 22, 2020
Abstract

[EN] In this project we study and present the field of visibility graphs as a way to represent and characterize time series as introduced by Lucas Lacasa et al. in 2008. With visibility graphs, time series are converted into graphs that inherit in their topology some of the structural properties of the time series, allowing the novel analysis of time series via graph theory and complex network theory tools. We study and analyze the computational complexity of a selection of different visibility graph algorithms and subsequently develop and present the ts2vg Python package that provides efficient and easy-to-use implementations for some of these visibility graph algorithms. Additionally, in this project we explore and discuss some of the potential practical applications of these visibility graphs.

[ES] En este proyecto estudiamos y presentamos el campo de los grafos de visibilidad (visibility graphs) como una forma de representar y caracterizar series temporales tal y como fue introducido por Lucas Lacasa et al. en 2008. Con los grafos de visibilidad las series temporales se convierten en grafos que heredan en su topología algunas de las propiedades estructurales de las series temporales, permitiendo así el novedoso análisis de series temporales a través de herramientas propias de la teoría de grafos y la teoría de redes complejas. Estudiamos y analizamos la complejidad computacional de algunos algoritmos para obtener grafos de visibilidad y, posteriormente, desarrollamos y presentamos el paquete de Python ts2vg que proporciona implementaciones eficientes y fáciles de usar para algunos de estos algoritmos. Además, en el proyecto exploramos y discutimos algunas de las posibles aplicaciones prácticas de estos grafos de visibilidad.

[CA] En aquest projecte estudiem i presentem el camp dels grafos de visibilitat (visibility graphs) com una forma de representar i caracteritzar sèries temporals tal com va ser introduït per Lucas Lacasa et al. el 2008. Amb els grafos de visibilitat les sèries temporals es converteixen en gràfics que hereten en la seva topologia algunes de les propietats estructurals de les sèries temporals, permetent així el nou anàlisi de sèries temporals a través d’eines pròpies de la teoria de gràfics i la teoria de xarxes complexes. Estudiem i analitzem la complexitat computacional d’alguns algorismes per obtenir gràfics de visibilitat i, posteriorment, desenvolupem i presentem el paquet de Python ts2vg que proporciona implementacions eficients i fàcils d’usar per a alguns d’aquests algorismes. A més, al projecte explorem i discutim algunes de les possibles aplicacions pràctiques d’aquests gràfics de visibilitat.

Keywords— visibility graphs, time series, graph theory, complex networks, Python
## Contents

1 Introduction ............................................. 3  
   1.1 Context .......................................... 3  
   1.2 Description of the project ......................... 3  
      1.2.1 Goals and scope of the project .......... 3  
   1.3 Motivations ..................................... 4  
   1.4 State of the art and alternatives ................. 4  

2 Definitions and Theoretical Background .............. 6  
   2.1 Time series ..................................... 6  
      2.1.1 Wiener process (Brownian motion) .......... 7  
      2.1.2 Fractional Brownian motion ................. 8  
   2.2 Graphs .......................................... 9  
      2.2.1 Types ...................................... 9  
      2.2.2 Measures ................................... 9  
   2.3 Real-world networks ............................... 11  
   2.4 Graph clustering ................................ 12  
   2.5 Visibility graphs ................................ 12  
      2.5.1 Horizontal visibility graphs .......... 14  
   2.6 Power law fitting ................................ 15  

3 Algorithms and Complexity ............................ 16  
   3.1 Naive strategy .................................. 17  
   3.2 Slope strategy .................................. 19  
   3.3 Divide-and-conquer strategy ....................... 22  
   3.4 Streaming algorithm .............................. 26  

4 Technologies and Implementation ...................... 27  
   4.1 Programming language ........................... 27  
   4.2 Time series implementation ....................... 27  
   4.3 Visibility graph algorithms implementation .... 28  
      4.3.1 Backend optimizations with C and Cython 28  
      4.3.2 Iterative algorithm for recursion .... 29  
      4.3.3 Performance analysis ...................... 29  
   4.4 ts2vg package ................................ 29  
      4.4.1 Potential future considerations .......... 30
# Contents

4.5 Auxiliary functionalities .......................... 31
   4.5.1 Graph analysis functionalities .................. 31
   4.5.2 Generating Wiener process and fractional Brownian motion series ....... 31
   4.5.3 Estimating power-law exponents .................. 31

5 Applications and Experiments .................. 32
   5.1 Performance evaluation of the different algorithms .................. 32
   5.2 Periodic series result in regular visibility graphs .................. 33
   5.3 Fractal series result in scale-free visibility graphs ................. 35
   5.4 Estimating the Hurst exponent via the visibility graph ................. 36
   5.5 Segmentation of time series via graph clustering .................. 38

6 Discussion and Future Work .................. 40

Bibliography ........................................ 42

Appendices ........................................... 46

A Conditions and Specifications ................. 47

B Algorithm’s Source Code .................. 48
   B.1 Naive strategy implementation .................. 49
   B.2 Slope strategy implementation .................. 50
   B.3 Divide-and-conquer strategy implementation .................. 51
   B.4 Streaming algorithm implementation .................. 52
   B.5 ts2vg package .................................. 53

C Project Management .................. 54
   C.1 Tasks ........................................ 54
   C.2 Time planning .................................. 56
   C.3 Budget ........................................ 58
      C.3.1 Human resources .................. 58
      C.3.2 Hardware resources .................. 58
      C.3.3 Software resources .................. 59
      C.3.4 Indirect costs .................. 59
      C.3.5 Contingencies .................. 59
      C.3.6 Unexpected deviations .................. 59
      C.3.7 Total budget .................. 60
   C.4 Economical impact ................................ 60
   C.5 Environmental impact ............................ 60
   C.6 Social impact .................................. 61
   C.7 Regulations .................................... 61
   C.8 Technical competences ............................ 61
1 Introduction

1.1 Context

This work is developed by Carlos Bergillos Varela as a Final Degree Project for the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics (FIB) of the Universitat Politècnica de Catalunya (UPC). The director of the project is Argimiro A. Arratia Quesada, Professor from the Department of Computer Science at UPC. The project is framed within the Computing specialization of the mentioned degree, and as such, aims to display and make use of technical competencies associated with this specialization.

1.2 Description of the project

Time series are an extensively used and simple data representation (in most cases they can be thought of as just an ordered list of numbers) for which a new frame of study has recently emerged, with what is known as visibility graphs. In this project we will explore the techniques introduced by Lucas Lacasa et al. 2008 [30], which allow the study of time series through a completely new perspective aided by graph theory and complex network theory ideas.

1.2.1 Goals and scope of the project

The goals of the project are multiple and can be grouped and summarized as:

• to serve as an opportunity for the author to research, study, and later present the field of visibility graphs, its algorithms, its potential applications, and associated complex network theory ideas

• to study and define the computational complexity, both in terms of time and memory, of a selection of different algorithmic strategies useful for obtaining and working with visibility graphs

• to implement correct and efficient versions of the different studied algorithms with the end-goal of developing an easy-to-use and efficient software framework for visibility graphs

• to reproduce known experiments and applications of visibility graphs using the implemented algorithms and assess the obtained results and behaviors

• to explore and document new possibilities and applications in the field of visibility graphs
1.3 Motivations

Time series (or signals) are simple data representations used extensively throughout lots of different human and natural phenomena, found in all kinds of fields like economics, physics, chemistry, biology, engineering, etc. Thus, time series are an extremely simple representation idea but used to capture what sometimes are very complex systems.

The field of visibility graphs allows transforming time series data into graphs via an abstract mapping, thus building a bridge between time series analysis and graph theory [43]. Time series analysis, although a mature field, has some limitations when it comes to studying complex non-linear systems and behaviors. Graph theory is also a very mature field in mathematics and computer science with many decades of study, with hundreds of associated studies, theories, and algorithms. With this, visibility graphs now open many new possibilities by allowing to study time series from a completely different and powerful non-linear perspective, whose full potential is still not fully known and is worth exploring.

More traditional time series analysis techniques should and will continue to be used in many applications, as they have proven to be very effective for many specific tasks, and perhaps more importantly, are more mature and mathematically well-founded. Time series analysis via visibility graphs should therefore obviously not be a replacement for other well established and more direct time series analysis techniques when these techniques already produce successful results. But research and exploration of the mathematical properties and potential applications of visibility graphs is certainly worth being continued and could result in benefits not yet discovered, by potentially providing new tools effective at solving problems that might be very hard to approach otherwise.

With this project we intend to gain significant knowledge on the field of visibility graphs and their associated algorithms, and to study some of their useful properties and practical applications. We will also focus on studying and evaluating the computational complexity of different algorithms used to obtain visibility graphs from an input time series, worth taking into consideration when working with algorithms that will be in charge of processing potentially very large amounts of data.

This project also emerges as a response for the need to have available tools to compute and work with visibility graphs, as no adequate open tools that fulfill all the potential requirements in time series research have been found. The algorithm implementations that result from this project can be a useful tool for future research in the field.

1.4 State of the art and alternatives

The study of visibility graphs applied to time series analysis is a relatively recent field and a decent amount of studies have already been presented that attempt to establish mathematical foundations for visibility graphs and their use [30, 31, 35, 43], but further work on strengthening these mathematical ideas and building new bridges with other areas of mathematics is still required.

Due to their novelty, many researchers have also started to explore the use of visibility graphs applied to time series analysis in a wide variety of many different scientific fields and applications with different levels of success. For example, the use of visibility graphs has been explored in medical applications [48, 56], the study of earthquakes series [28], periodicity of series [45], forecasting of series [9], air quality patterns in cities [55], turbulent fluid flows [25, 47], hydrogeological patterns [27], sunspots and solar wind activity [49, 57], financial series [8, 42], image processing [26], and even human behavior [15] and touristic activity [3], among others.

Computing and working with visibility graphs presents a unique and interesting computational challenge, and despite the amount of research on visibility graphs presented, research on the computational aspects and efficiency of the algorithms used for visibility graphs work is rarer.
Due to the apparent demand for visibility graphs research, the availability of efficient and accessible software implementations of the algorithms prove to be of big importance for researchers. Some available functions and implementations of visibility graphs for different programming languages already exist, such as the one provided by Lucas Lacasa in the Fortran programming language [29], and other implemented functions for Python [19] and MATLAB [24]. These implementations might be limited in terms of usability, performance, and future extensions. In this project we have developed and presented a software library and implementation that attempts to improve the situation by providing an open, accessible, easy-to-use and very efficient tool to obtain and work with visibility graphs in the Python programming language.
2 Definitions and Theoretical Background

2.1 Time series

A time series is a sequence of data points listed in time order, usually taken in regular equally spaced points in time. In other words, a time series is any data sequence taken in discrete time. The data could, for example, come from real-life measurements (like the temperature at a certain location) or be derived from a mathematical formula or model (e.g. see Brownian Motion in Section 2.1.1).

Time series are commonly visualized using line plots, area plots, or bar plots where the $x$ axis corresponds to time, and the height ($y$ axis) corresponds to the value of the data for the corresponding point in time, see Figure 2.1 for an example.

We can mathematically formalize any time series $T$ using a set of pairs:

$$T = \{(t_1, y_1), (t_2, y_2), \ldots, (t_n, y_n)\}$$

where each pair $(t_i, y_i)$ corresponds to one observation (or data point) in which $y_i$ is the data value associated to the point in time $t_i$, and such that $t_1 < t_2, \ldots, < t_n$. For simplicity, we will sometimes refer to a given observation $(t_i, y_i)$ as just data point $i$. We will use the term length to refer to the number of observations of a given time series.

![Figure 2.1](image-url) Figure 2.1: Example of a time series plot. Values of the IBEX 35 stock market index from the year 2015 until the beginning of 2020. Data provided by ICE Data Services and accessed via Yahoo Finance [53, 54].
Additionally, if the series observations are equally spaced in time, we can define a time series as just an ordered list of values:

\[ T = (y_1, y_2, \ldots, y_n) \]

assuming that any relevant relative time information is implicitly accounted for in the order of the elements.

### 2.1.1 Wiener process (Brownian motion)

In addition to studying real-world time series, it is also very useful to explore some known mathematical series that might help understand other more complex real-world behaviors. One such mathematical series is the one resulting from the widely studied Wiener process, also known as Brownian motion in physics and other areas of science.

The Wiener process has large practical and theoretical significance, it can be used as a model for many different real-world phenomena. For example, it can be used to model the movement of physical particles or to model the values of the stock market [14].

The Wiener process \( W(t), t \in \mathbb{R}, t \geq 0 \), is a mathematical continuous-time stochastic process with the following properties [32]:

1. \( W(0) = 0 \)
2. \( W(t) \) is a continuous function in \( t \)
3. \( W \) has independent increments
   - For \( 0 \leq s < t < u < v \), the increments \( W(t) - W(s) \) and \( W(v) - W(u) \) are independent.
4. \( W \) has Gaussian increments
   - For \( 0 \leq s < t \), the increment \( W(t) - W(s) \) is normally distributed with mean 0 and variance \( t - s \):
     \[ W(t) - W(s) \sim \mathcal{N}(0, t - s) \]

The Wiener process as defined is a continuous-time signal. We can easily construct a discrete-time version \( W_t, t \in \mathbb{N} \), using an iterative random-walk approach defined as:

\[
\begin{align*}
W_0 &= 0 \\
W_i &= W_{i-1} + \xi_i 
\end{align*}
\]

Where \( \xi_1, \xi_2, \ldots \) are independent and identically distributed (i.i.d.) random variables with mean 0 and variance 1. So, for any \( i \): \( \xi_i \sim \mathcal{N}(0, 1) \).

When \( n \to \infty \), \( W_n \) behaves similarly to the continuous Wiener process described before [32]. From now on, we will use the name Wiener process to refer to this discrete version. With this method we can construct time series as formalized in Section 2.1:

\[ T_W = \{(1, W_1), (2, W_2), \ldots, (n, W_n)\} \]

This iterative approach can be easily implemented programmatically, thus allowing us to generate Wiener process time series of any length \( n \) we want (only limited by practical computer limitations).

Because \( W_t \) is a stochastic process (the stochastic nature emerges from the use of randomized increments) we can generate infinitely many different time series \( T_W \).

Another way to look at Wiener process series is that they are constructed by integrating a Gaussian series with distribution \( \mathcal{N}(0, 1) \), integrating what is known as Gaussian white noise (see Figure 2.2 for an example).
2.1.2 Fractional Brownian motion

Fractional Brownian motion (fBm) is a generalization of the previously described Brownian motion (or Wiener process, see Section 2.1.1). Unlike in the classical Brownian motion, in the fractional Brownian motion the increments are not required to be independent. The amount of dependency (or covariance) of the increments is a function of a parameter $H$ in the interval $(0, 1)$, called the Hurst index or Hurst parameter [34, 36, 37]. Depending on the value of $H$, the behavior of the fBm process can be divided into three classes:

- if $H < 1/2$ then the increments of the process are negatively correlated
- if $H = 1/2$ then the process is a regular Wiener process, with non-correlated increments
- if $H > 1/2$ then the increments of the process are positively correlated

The closer $H$ is to 0 and the closer $H$ is to 1 the more negatively and positively correlated the increments will be respectively.

Therefore, for example, if an fBm process with a value of $H$ greater than $1/2$ is increasing in value in a certain interval, then it is likely to keep increasing in a similar way in the next interval. Furthermore, the extreme case of $H = 1$ would result in a completely predictable linear function of constant trend. Examples of fBm processes with different values of $H$ are shown in Figure 2.3.

Mathematically, one way to define fBm processes is via the Riemann–Liouville fractional integral, resulting in the following stochastic representation [34, 37]:

$$B_H(t) = \frac{1}{\Gamma(H + 1/2)} \int_0^t (t - s)^{-H-1/2} dW(s) \quad (2.2)$$

where $W(s)$ is a regular Wiener process (or Brownian motion) as previously described in Section 2.1.1 and $\Gamma$ is the gamma function, which can be defined as $\Gamma(n) = (n - 1)!$ for positive integer values. Note how when $H = 1/2$ then $\Gamma(H + 1/2) = 1$ and $B_H(t)$ becomes the regular Wiener process $W(t)$.

The Hurst index can be said to describe the raggedness, volatility or long-term memory of the fBm series [37], and the same idea can be extrapolated to measure these properties for any kind of time series (not only the ones resulting from fBm processes) via the indicator known as the Hurst exponent.
2.2 Graphs

In the field of graph theory, a graph (also referred as a network) is a mathematical structure consisting of a set of objects in which some pairs of these objects are connected or related in some sense. For a given graph $G$, this is generally formalized as $G = (V, E)$, where $V$ is the set of vertices (or nodes) that represent the objects, and $E$ is the set of edges (or links) that represent the connections among pairs of objects. Two nodes that are connected with an edge are also referred to as neighbor nodes.

2.2.1 Types

Directed - Undirected A directed graph is a graph where each edge has a notion of direction, this is, edges have an origin and a destination, so for a given pair of nodes $a, b$ there are two possible edges $(a, b)$ and $(b, a)$ that can be part of the graph. On the other hand, in an undirected graph there is no notion of direction and the connection between nodes is defined to be symmetrical, so the edge $(a, b)$ is the same as the edge $(b, a)$.

Weighted - Unweighted A weighted graph is a graph where each of its edges has an associated numerical weight. This weight corresponds to any relevant information of the connection, such as a distance, cost, or capacity. On the other hand, an unweighted graph is a graph with no associated weight information and therefore all node connections are considered equal.

A graph can be any combination of directed/undirected and weighted/unweighted as these properties are independent.

2.2.2 Measures

For any graph $G = (V, E)$ we define the following properties and measures:
**Degree** $k$  For any node $i$ in a graph, its degree $k_i$ is the number of nodes it is connected to, this is, the number of adjacent nodes (and edges) of node $i$.

**Average degree** $\langle k \rangle$  A quick and simple measure for the connectivity of a graph is the average $\langle k \rangle$ of all the degrees $k_i$ in the graph:

$$\langle k \rangle = \frac{1}{n} \sum_{i=1}^{n} k_i$$  (2.3)

Notably, since $\sum_i k_i = 2m$ (where $m$ is the number of edges in the graph), another straightforward way of obtaining $\langle k \rangle$ is:

$$\langle k \rangle = \frac{2m}{n}$$  (2.4)

**Degree distribution** $P(k)$  For a given degree number $k$, $P(k)$ provides the probability that a node picked at random from the network will have degree $k$. For a given graph $G$, this probability $P(k)$ can be approximated as the ratio between the number of nodes with degree $k$ and the total number of nodes:

$$P(k) \approx \frac{N(k)}{N}$$  (2.5)

where $N(k)$ is the number of nodes in the graph with degree $k$.

As with any probability distribution (and knowing that degrees are always non-negative), it follows that:

$$\sum_{k=0}^{+\infty} P(k) = 1$$  (2.6)

**Diameter** $d$  The diameter $d$ of a graph is the maximum length of all the shortest paths found between any pair of nodes in the graph. In other words, the length of longest shortest path between any two nodes in the graph. Mathematically:

$$d = \max_{u,v \in V} d(u,v)$$  (2.7)

where $d(u,v)$ is the length of the shortest path from $u$ to $v$, usually defined as the number of edges that form this shortest path.

**Average shortest path length** $\langle l \rangle$  Another very useful connectivity measure for graphs is their average shortest path length $\langle l \rangle$, obtained by:

$$\langle l \rangle = \frac{1}{n} \sum_{u,v \in V} d(u,v)$$  (2.8)

**Clustering coefficient** $C_i$  For any node $i$ in a graph, its clustering coefficient $C_i$ is the ratio of the number of edges between its neighbors and the number of possible such edges. For nodes $i$ with $k_i = 0, 1$, we define $C_i = 0$. We know that a node $i$ has $k_i$ neighbors, and for a set of $k_i$ nodes there are $\binom{k_i}{2}$ possible edges (number of edges in a complete graph of $k_i$ edges). Thus, we can calculate the clustering coefficient as:

$$C_i = \frac{E_i}{\binom{k_i}{2}} = \frac{2E_i}{k_i(k_i-1)}$$  (2.9)

Where $E_i$ is the number of edges between $i$’s neighbors (number of edges in the subgraph induced by $i$’s neighbors. With this, we can also find the average clustering coefficient $\langle C \rangle$ of the network:

$$\langle C \rangle = \frac{1}{n} \sum_{i=1}^{n} C_i$$  (2.10)
2.3 Real-world networks

Graphs used to model many real-world phenomena have been found to share a surprising number of properties, even across vastly different scales and fields [40]. Examples of such graphs are the ones obtained from human and animal social networks, biological structures (such as gene regulation and protein interactions), the Internet, transport networks, among others. These common properties are being studied extensively and have resulted in the concept of what are known as real-world networks, small-world networks, or scale-free networks. These real-world networks can be simulated with models that attempt to replicate their degree distribution or other topological characteristics with varying degrees of success. One of the most popular of such models is the Barabási–Albert [4]. In summary, real-world networks have the following properties [2]:

**Small diameter** Real-world networks have a very small diameter and a very small average shortest path length and therefore, most nodes can reach any other node with a very small number of steps through the network. For example, reaching any specific person from any other one in a social network or real-world community of people requires a very small number of intermediate connections. This is known as the *small-world phenomenon* and was famously exemplified in the 1967 Milgram experiment [39].

**High clustering coefficient** In a real-world network, neighbors of any given node are likely to also be neighbors of each other, this results in a large number of triangles of connected nodes in the network. This property is also known as *transitivity*. This fact gets reflected in a high average clustering coefficient of the network. As a practical example, in a friendship social network, two of someone’s friends are likely to also be friends of each other.

**Power-law degree distribution** The majority of nodes in a real-world network have a very low degree, only a very small number of them have a significantly large degree, these nodes with large degrees are sometimes referred as *hub* nodes. Therefore, node degrees in these networks are far from following a uniform or normal distribution, instead, their degree distribution is highly skewed favoring nodes with a low degree, resulting in what are colloquially called *fat-tail* or *heavy-tail* distributions. More precisely, the tail of the degree distribution of these networks can be often approximated with a power-law distribution, this is, the tail of their degree distribution $P(k)$ can be often accurately approximated with the form:

$$P(k) = ck^{-\gamma}$$  \hspace{1cm} (2.11)

See Figure 2.4 for an illustrated example. Networks with a power-law distribution of degrees are also often called scale-free networks.
2.4 Graph clustering

Graph clustering (also commonly known as network community detection) is a big topic of study in the graph theory and network science fields. It consists in grouping the nodes in a network into sets of nodes that are densely connected internally and sparsely connected externally across the different groups. For example, real social networks can often be divided into somewhat distinct communities that emerge based on common locations, interests, occupations, etc. of its members. The clustering of an example network is illustrated in Figure 2.5.

Finding such communities is often a computationally difficult task and many different ideas and techniques are being studied and used, such as algorithms based on minimum cuts, modularity optimization, simulated annealing, eigenvector decomposition (spectral graph theory), among others [18].

2.5 Visibility graphs

As described in [30], the (natural) visibility graph of a time series $T$ is the graph with $n$ nodes (corresponding to the $n$ observations in $T$) and where an edge between two nodes exists if and only if there is no obstruction when tracing a line between the top of the corresponding bars in a bar plot of $T$. Intuitively, if we consider the bar plot to be a landscape, for each node, we link it with all the other nodes whose bar can be viewed from the top of the considered one (see Figure 2.6 for an illustrated example).
In other words, in the visibility graph $G_T = (V, E)$ of $T$, for every pair of nodes $a, b \in V$ with associated data points $(t_a, y_a)$ and $(t_b, y_b)$, with $t_a < t_b$, there is an edge $(a, b) \in E$ if and only if there is no intermediate data point that obstructs the line-of-sight line from $(t_a, y_a)$ to $(t_b, y_b)$. Mathematically, an edge $(a, b)$ is in $E$ if and only if all points $(t_c, y_c) \in T$ with $t_a < t_c < t_b$, fall below the 2-dimensional line that passes through the points $(t_a, y_a)$ and $(t_b, y_b)$. This is, all intermediate points $(t_c, y_c)$ must fulfill the following inequality:

$$y_c < y_b - y_a \frac{t_b - t_c}{t_b - t_a} + y_a \quad (2.12)$$

In some literature (like in Lacasa et al. 2008 [30]) the following equivalent inequality is instead used:

$$y_c < y_b + (y_a - y_b) \frac{t_b - t_c}{t_b - t_a} \quad (2.13)$$

All visibility graphs as described have the following properties:

**Connected** Any pair of contiguous observations in $T$ (with times $t_x$ and $t_{x+1}$) have no possible obstruction laying between them, so all observations are connected to their immediate neighbors, and therefore, the resulting visibility graph has always exactly 1 connected component.

**Undirected** There is no notion of direction of the edges in the visibility graph implementation we are working with. Intuitively, a node $A$ in the bar plot sees another node $B$ if and only if node $B$ sees node $A$. Therefore, the resulting graph is undirected. With that said, additional extensions to the basic visibility graph model can deviate from this by, for example, assigning direction related to the time axis direction of the input data, but exploring extensions like this is outside the scope of this project.

**Unweighted** The edges in the resulting graph have no notion of weight, value, or category assigned to them. With that said, additional extensions to the basic model could, for example, assign a weight to edges based on slope or length of the visibility lines, but exploring extensions like this is outside the scope of this project.

**Invariant under affine transformations of the series data** The visibility criterion is invariant under horizontal and vertical rescaling, horizontal and vertical translations, and any composition of linear transformations and translations (affine transformations) of the data [30]. See Figure 2.7 for examples.
Lossy  The affine transformation invariance just described also implies that the visibility graph mapping is not bijective, and two different time series can have the same visibility graph. Thus, we are losing information in the conversion (information such as magnitudes and absolute values of the time series) and recovering the original time series from its visibility graph is not possible.

2.5.1 Horizontal visibility graphs

Visibility graphs as described in Section 2.5 (which are also sometimes known as natural visibility graphs to avoid ambiguity) are not the only way to map time series into graphs. Another of such techniques is the one known as horizontal visibility graphs [35, 43]. In this case, a graph of \( n \) nodes is constructed (again, corresponding to the \( n \) observations of the input time series), and any two of such nodes are connected if and only if there is obstruction-free horizontal visibility between the two data points (see an illustrated example in Figure 2.8). Mathematically, two observations \((t_a, y_a)\) and \((t_b, y_b)\), with \( t_a < t_b \), are connected via an edge \((a, b)\) in a horizontal visibility graph if and only if:

\[
y_a, y_b > y_c \text{ for all } c \text{ such that } t_a < t_c < t_b
\]  

(2.14)

If two nodes have horizontal visibility they will also have direct natural visibility (but not the other way around), therefore the horizontal visibility graph of a given time series will always be a subgraph of its natural visibility graph.

Figure 2.8: Bar plot for an example time series (left), horizontal obstruction-free lines traced (center), and resulting horizontal visibility graph (right). The same example time series is used as in Figure 2.6. Notice that the horizontal visibility graph is a subgraph of the previous visibility graph.
2.6 Power law fitting

As previously stated, a power-law distribution is one that follows the form:

\[ P(k) = ck^{-\gamma} \]  \hspace{1cm} (2.15)

The constant \( \gamma \) is called the exponent of the power law and characterizes the slope of the distribution in a log-log plot. The constant \( c \) is not as interesting to work with, since, once \( \gamma \) is fixed, \( c \) is determined by the requirement that the distribution \( P(x) \) sums to 1.

When given a set of observations \( x_1, \ldots, x_n \) (for example, the sequence of degrees of the \( n \) nodes of a graph) and believing that its probability distribution follows a power law (following a straight line in a log-log plot as a necessary but not sufficient condition) we can use various techniques to try to estimate the value for \( \gamma \). This is, finding the value for the exponent \( \gamma \) that results in the ideal power-law distribution that most accurately models the distribution of our empirical observations.

Note that in real data the power-law behavior often only manifests in the tail of the distributions, this is, only for values of \( x \) larger than some \( x_{\text{min}} \). One of the most reliable methods to obtain \( \gamma \) is to use Maximum Likelihood Estimation (MLE), in this case given by the following formula [10, 41]:

\[ \gamma = 1 + n \left[ \sum_{i=1}^{n} \log \frac{x_i}{x_{\text{min}}} \right]^{-1} \]  \hspace{1cm} (2.16)

where \( x_i, i = 1, \ldots, n \) are the observed values of \( x \) such that \( x_i \geq x_{\text{min}} \).
We explored different algorithms to build visibility graphs. The input to all the algorithms should be a time series as described in Section 2.1, and they should return an undirected and unweighted graph complying with the visibility graph definition given in Section 2.5. Therefore, for all of the upcoming algorithms (unless otherwise specified) the input and output will be of the following form:

**INPUT**

Time series $T$: $T = \{(t_1, y_1), (t_2, y_2), \ldots, (t_n, y_n)\}$

**OUTPUT**

Graph $G$: $G = (V, E)$

As a quick summary, the computational cost in time of three algorithms is provided in Table 3.1.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>$O(n^2)$</td>
<td>$O(n^3)$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>Slope</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

**Table 3.1:** Asymptotic computational time complexity of three different later described algorithms to obtain the visibility graph from an input time series of length $n$.

These algorithms are described in detail in the upcoming sections in this chapter.
3.1 Naive strategy

Algorithm 1 Visibility Graph - Naive Strategy

1: function VisibilityGraph($T = \{(t_1, y_1), \ldots, (t_n, y_n)\}$) 
2: \hspace{1em} $V \leftarrow \{1, \ldots, n\}$ 
3: \hspace{1em} $E \leftarrow \emptyset$ 
4: \hspace{1em} for each $(t_a, y_a), (t_b, y_b) \in T$ with $t_a < t_b$ do 
5: \hspace{2em} if NoObstructions($T$, $a$, $b$) then 
6: \hspace{3em} add edge $(a, b)$ to $E$ 
7: \hspace{1em} return $G = (V, E)$ 
8: end function 

9: function NoObstructions($T$, $a$, $b$) 
10: \hspace{1em} for each $(t_c, y_c) \in T$ with $t_a < t_c < t_b$ do 
11: \hspace{2em} if $y_c \geq y_b + (y_a - y_b) \frac{t_b - t_c}{t_a - t_c}$ then 
12: \hspace{3em} return False 
13: \hspace{1em} return True 
14: end function 

A naive and straightforward approach to obtain the visibility graph consists on exhaustively traversing all pairs of nodes (all potential edges), and for each one, check if any obstruction is found between the two corresponding data points by checking if the inequality (2.13), in page 13, is broken by any in-between point and add the corresponding edge when appropriate. For a given pair of nodes, as soon as one obstructing in-between data point is found (see Figure 3.1 for an example) we can stop checking the rest of the in-between points since at that moment we are already certain that no edge should be present between the two nodes. On the other hand, for two nodes to be connected in the visibility graph, we need to guarantee that all in-between points have been checked and no obstruction has been found.

![Figure 3.1](image_url): Nodes corresponding to observations at $t_1$ and $t_6$ in this example will not be connected in their visibility graph because the data point at $t_4$ lays in the obstruction shaded area.

**Correctness** The algorithm exhaustively visits all potential edges in the graph and checks inequality (2.13) (in page 13) for all of them, therefore directly complying with the visibility graph definition given in Section 2.5.

**Time complexity** For any given time series of size $n$ and its resulting graph with $n$ nodes, we need to check all potential edges in the graph, and to do so, we are traversing all possible pairs of data points. We know that the number of such potential edges (and the number of such pairs...
of data points) is \( \binom{n}{2} = \frac{n(n-1)}{2} \). This results in \( \frac{n(n-1)}{2} \) iterations \( O(n^2) \) of the external for-loop found in line 4. For each iteration of this loop with a given potential edge \((a, b)\) we need to check whether all intermediate points \(c\) (with \( t_a < t_c < t_b \)) satisfy the required inequality.

Note that, and as explained before, the number of inequality checks that need to be performed is not fixed and depends on the particular structure of the time series (as soon as one intermediate point is found to not satisfy the inequality we can stop checking for that edge). In the worst case, for all pairs \((a, b)\), all intermediate points \(c\) will need to be checked \((b-a-1)\) intermediate points in each case, in the order of \( O(n) \) asymptotically, resulting in a total algorithm time complexity of \( O(n^3) \).

On the other hand, in the best case (e.g. in a flat time series, for this algorithm), each pair of nodes gets checked in constant time, as for all potential edges \((a, b)\), either \(a\) and \(b\) are immediate neighbors (without intermediate points \(c\), and therefore connected), or \(a\) and \(b\) are not immediate neighbors but the first intermediate point \(c\) that gets checked will already present an obstruction not satisfying the inequality. In such time series checking for any given potential edge \((a, b)\) will always require constant time, resulting in a total algorithm time complexity of \( O(n^2) \) in this best case.

The average time complexity of this algorithm is difficult to estimate, as there are infinitely many possible input time series, and the exact running time behavior depends on the structural properties of the input series. With that said, we can make the assumption that on average, for each potential edge \((a, b)\), an unknown fixed fraction of the \((b-a-1)\) intermediate points will get checked, and therefore the average time complexity does not change asymptotically, remaining at \( O(n^3) \).

**Space complexity** The algorithm does not require any additional data structures apart from the input time series of size \(n\), and the output graph consisting of \(n\) nodes and, at most, \( \frac{n(n-1)}{2} \) edges, \( O(n^2) \) asymptotically. Therefore the cost of the space required in memory for this algorithm will be dictated by the size of the output graph, in the order of \( O(n^2) \).
3.2 Slope strategy

Algorithm 2 Visibility Graph - Slope Strategy

1: function VisibilityGraph(T = \{(t_1, y_1), \ldots, (t_n, y_n)\})
2:     V ← \{1, \ldots, n\}
3:     E ← \emptyset
4:     for a ← 1 to n − 1 do
5:         slope\_max ← −\infty
6:         for b ← a + 1 to n do
7:             slope_{a,b} ← (y_b − y_a)/(t_b − t_a)
8:             if slope_{a,b} > slope\_max then
9:                 add edge (a, b) to E
10:         slope\_max ← slope_{a,b}
11:     return G = (V, E)
12: end function

We can improve the previously described naive strategy by benefiting from information provided by the slope of the visibility lines as explained in [33]. Mathematically, for a time series T, a pair of data points \((a, b)\) ∈ T (with \(t_a < t_b\)) are connected nodes in the visibility graph of T if and only if the slope of the visibility line from \(a\) to \(b\) is larger than the maximum slope found between \(a\) and all intermediate data points \(c\) with \(t_a < t_c < t_b\) (see proof of this in the upcoming Theorem 3.2.1). This way we can find all edges for the node corresponding to time \(t_1\) in a single pass through the other points by just storing the value for the maximum slope found so far (see Figure 3.2 for an example), and similarly for the remaining edges for the node at \(t_2\), etc.

![Figure 3.2: The slope of the visibility line is strictly increasing for the connected points as \(t\) increases.](image)

**Correctness** In order to prove that this approach complies with the visibility graph definition proposed in Section 2.5, we will first prove that the slope idea is more than just an intuition. To do so, we will define and prove the following:

**Theorem 3.2.1.** For a given time series \(T\) and \((a, y_a), (b, y_b)\) ∈ \(T\) with \(t_a < t_b\), the edge \((a, b)\) belongs to the visibility graph \(G_T\) of \(T\) if and only if the slope of the visibility line from \(a\) to \(b\) is larger than the maximum slope found between \(a\) and all other observations \(c\) with \(t_a < t_c < t_b\).

**Proof.** Let \(C'\) be the set of the intermediate points: \(C' = \{c \mid (t_c, y_c) \in T \text{ and } t_a < t_c < t_b\}\). Let \(s(i, j)\) be the slope of the line that passes through some given data
points \((t_i, y_i)\) and \((t_j, y_j)\), calculated as:

\[
s(i, j) = \frac{y_j - y_i}{t_j - t_i}
\]  

With this, and using the visibility graph definition and its inequality (2.12) (in page 13) we can obtain:

\[
(a, b) \text{ is an edge in the visibility graph } \iff \forall c \in C': y_c < y_b - y_a \cdot \frac{t_c - t_a}{t_b - t_a} + y_a
\]

\[
\iff \forall c \in C': y_c < s(a, b) \cdot (t_c - t_a) + y_a
\]

\[
\iff \forall c \in C': \frac{y_c - y_a}{t_c - t_a} < s(a, b)
\]

\[
\iff \forall c \in C': s(a, c) < s(a, b)
\]

\[
\iff \max_{c \in C'} s(a, c) < s(a, b)
\]

By storing and updating the value for largest slope encountered so far \((slopes_{\text{max}})\) when linearly traversing the data points to the right of a given data point \(a\), we know that if the slope from \(a\) to a new data point \(b\) is larger than \(slopes_{\text{max}}\) then the slope from \(a\) to \(b\) is larger than all the slopes from \(a\) to the previous points between \(a\) and \(b\) and therefore there is direct line-of-sight visibility from \(a\) to \(b\) and an edge \((a, b)\) is added to the graph. In doing this linear pass starting from all possible nodes \(a\), we will have checked the visibility for all potential edges (all pairs of nodes) in the graph.

**Time complexity**  We can quickly see that this algorithm makes use of two for-loops, one nested inside the other (lines 4 and 6). Both of these for-loops do a number of iterations in the order of \(O(n)\), resulting in a total number of iterations of the inner loop of \(O(n^2)\). More precisely, the number of times that the block of code (of constant time) found inside the inner loop (lines 7 to 10) is executed is described by the following formula:

\[
\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} 1 = \sum_{a=1}^{n-1} (n - a)
\]

\[
= (n - 1)n - \sum_{a=1}^{n-1} a
\]

\[
= (n - 1)n - \frac{(n - 1)(n)}{2}
\]

\[
= \frac{n^2 - n}{2}
\]

Which, not surprisingly, also corresponds to the number of potential edges in a graph with \(n\) nodes.

This algorithm has a running time that only depends on the size of the input \(n\) and does not depend on the structure and values of the input time series like in the previous naive algorithm. Therefore its best-case, worst-case, and average-case time complexity is always \(O(n^2)\).
Space complexity  This algorithm only requires one extra auxiliary variable (slope_{max}), of constant size in memory $O(1)$. Therefore, as with the previous naive algorithm, the memory required is mainly dictated by the size of the input time series and the size of the output graph, again, in the order of $O(n^2)$. 
### 3.3 Divide-and-conquer strategy

**Algorithm 3** Visibility Graph - Divide-and-Conquer Strategy

1. function VisibilityGraph\( (T = (t_1, y_1), \ldots, (t_n, y_n)) \)
2. \( V \leftarrow \{1, \ldots, n\} \)
3. \( E \leftarrow \text{VisibilityGraphRecursive}(T, 1, n) \)
4. return \( G = (V, E) \)

5. end function

6. function VisibilityGraphRecursive\( (T, \text{left}, \text{right}) \)
7. \( E' \leftarrow \emptyset \)
8. if \( \text{left} < \text{right} \) then
9. \( i_{\text{max}} \leftarrow \text{index } i, \text{left} \leq i \leq \text{right}, \text{with highest } y_i \)
10. for \( a \leftarrow \text{left} \) to \( \text{right} \) except \( i_{\text{max}} \) do
11. if node \( i_{\text{max}} \) ‘sees’ node \( a \) then \( \triangleright \) i.e. using the slope strategy locally
12. add edge \( (i_{\text{max}}, a) \) to \( E' \)
13. \( E' \leftarrow E' \cup \text{VisibilityGraphRecursive}(T, \text{left, } i_{\text{max}} - 1) \)
14. \( E' \leftarrow E' \cup \text{VisibilityGraphRecursive}(T, i_{\text{max}} + 1, \text{right}) \)
15. return \( E' \)
16. end function

When looking for an even more optimal strategy we can see that we can benefit from the fact that very large data points in the time series act as a kind of barrier or wall between the data points found to their left and those found to their right \[33\], preventing most nodes from the two different sides from being connected in the visibility graph. We can make use of this property to prune many of the candidate edges, substantially reducing the number of potential edges that need to be checked. In particular, the highest data point in the time series will prevent any data point to its left from having line-of-sight visibility with any node to the right of this highest point. We can, therefore, divide the input time series into two segments split by this highest data point, and consider the two resulting smaller time series as two independent visibility graph subproblems, without having to check for edges corresponding to nodes that belong to different sides, as we already know that none will exist. The visibility graph of the original time series will be the result of taking these two smaller visibility graphs together and adding the node of the highest data point and its edges (which, in this case, can indeed reach both sides, and will need to be checked using, ideally, the slope strategy described in Section 3.2). This idea can be recursively applied to obtain the visibility graph of each of the two mentioned smaller time series, whose highest data point can again split the segment into two even smaller subproblems. This can continue until the time series segments cannot be further divided. See Figure 3.3 for an example.

**Correctness** This algorithm builds on top of the previously described algorithms and greatly reduces the number of potential edges that need to be checked. Let \( (t_{\text{max}}, y_{\text{max}}) \) be the observation with the highest value of the input time series \( T \) (such that \( y_{\text{max}} \geq y_i \) for all \( (t_i, y_i) \in T \)). Then we know for certain that there cannot be an edge connecting two nodes \( a, b \) if they are on different sides of \( t_{\text{max}} \) (e.g. \( t_a < t_{\text{max}} < t_b \)). As, by definition, all points other than \( (t_{\text{max}}, y_{\text{max}}) \) have a value not higher than \( y_{\text{max}} \) and thus will not have line-of-sight visibility with any point at the other side of \( (t_{\text{max}}, y_{\text{max}}) \) because \( (t_{\text{max}}, y_{\text{max}}) \) will always be an intermediate data point causing an obstruction.
Figure 3.3: Divide-and-conquer strategy demonstrated in an example time series. In this case, the maximum value of the time series corresponds to \(y_6\), so we connect its corresponding node with all other nodes in sight, and then we recursively repeat the same strategy on both smaller time series formed by the points to the left and to the right of \(t_6\).

Let \(T_L\) be the smaller time series formed by all data points to the left of \((t_{\text{max}}, y_{\text{max}})\), and equivalently \(T_R\) be the smaller time series formed by all data points to the right (note that \((t_{\text{max}}, y_{\text{max}})\) is not included in either). No node from \(T_L\) is connected to a node in \(T_R\) in the visibility graph of \(T\), and therefore the visibility graph of \(T\) can be constructed by the union of the nodes and edges of the visibility graphs of \(T_L\) and \(T_R\), with the addition of the node corresponding to \((t_{\text{max}}, y_{\text{max}})\) and its edges. Both \(T_L\) and \(T_R\) can be considered independent visibility graph subproblems and we can therefore recursively apply the same algorithmic idea again to obtain their visibility graphs. The recursion stops when the time series segments become too small and cannot be further divided.

**Time complexity** We explored the different behaviors experienced in the worst and best-case scenarios for this divide-and-conquer strategy. Due to the nature of the algorithm, the time required for its execution can vary significantly even when comparing different time series of the same input size \(n\). The running time of the algorithm will depend on the placement of the maximum values of the time series at each recursion step and how balanced each split happens to be.

In any case, the running time of each subproblem of input size \(n'\) is \(O(n')\). This is, the function in line 7 in Algorithm 3, excluding the two subsequent recursive calls (excluding lines 14 and 15), when given an input time series of size \(n'\) has cost \(O(n')\). This arises from the linear cost of finding the maximum value of the time series (in line 10), and the also linear cost of finding all the appropriate edges attached to the node corresponding to this maximum value (in line 11), which can be achieved in linear cost by taking advantage of the slope strategy as described in Section 3.2.
Figure 3.4: Monotonically increasing time series (a) and bowl-shaped time series (b) are examples of the worst-case running time of the Divide-and-Conquer strategy, as the size of the input only decreases by one at each recursion step. On the other hand, a fractal-like balanced time series like in (c) results in the best possible running time, as the size of the input gets exactly halved at each recursion step.

Worst case The worst-case behavior of this algorithm happens, for example, with monotonically increasing (or decreasing) time series. In these time series, at each recursion step the split is always as unbalanced as it can be, as the maximum value of the time series always happens to be either at the very beginning or very end of the time series. As a consequence, at each recursion step, for a certain partial input size $n'$, the two subsequent time series have size $0$ and $n' - 1$.

Therefore, in these worst cases, for an input time series of size $n$, the algorithm will require a recursion of total depth $n$, where the size of the subproblem gets only decreased by one at each step (although there is only one resulting subproblem instead of two). Thus, the total running time $T(n)$ of the algorithm in these worst cases is:

$$T(n) = O(n) + O(n - 1) + O(n - 2) + \cdots + O(1) = O(n^2)$$

Note that the behavior, in this case, is equivalent to that of the slope strategy previously described in Section 3.2.

Best case The best-case behavior of this algorithm happens when the divide-and-conquer strategy splits the input into two approximately equal parts in all recursion steps. This happens when the maximum values of the time series are appropriately distributed (see Figure 3.4). To find the total running time of this algorithm with these conditions, we will use the master theorem for divide-and-conquer problems [11]. According to the master theorem, for a recursive running time $T(n)$ of the form:

$$T(n) = aT(\lceil n/b \rceil) + O(n^d), \quad a > 0, \ b > 1 \text{ and } d \geq 0$$

then:

$$T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a 
\end{cases}$$

In our case, for the best case behavior of this divide-and-conquer algorithm, we have:

$$T(n) = 2T(\lceil n/2 \rceil) + O(n)$$
Where \( a = 2, b = 2, d = 1 \), and therefore we are in the second case \( (d = \log_b a) \), so the asymptotic running time \( T(n) \) of this divide-and-conquer algorithm in its best case is \( O(n \log n) \).

Note that for flat time series (time series \( T \) where \( y_i = y_j \) for all \( (t_i, y_i), (t_j, y_j) \in T \)) this divide-and-conquer algorithm has no defined running time complexity, it could either behave as in the worst case, or as in the best case, or anywhere in between, depending on the implementation and how the maximum values are selected at each step. In general, when multiple maximum data point candidates exist in a time series, the exact behavior of the algorithm is left up to the implementation. Ideally, the maximum data point that should be picked is the one that is closer to the center of the time series segment (resulting in the most balanced split possible). Multiple equal maximum data points are very rarely encountered in most real-world time series (specially in real-valued series) so this should not be a major concern in most algorithm implementations.

**Average case**

Estimating the average time behavior of this algorithm is not straight-forward. It requires determining how well balanced an average or typical time series is, which is a very difficult question to answer in itself, as there are infinitely many possible time series for any given length \( n \). Time series from real-world phenomena can come in many different structures and levels of balancedness, varying greatly from one domain to another. From empirical analysis on different groups of time series it could be reasonable to estimate the average time complexity for this algorithm as \( O(n \log n) \) [33].

**Space Complexity**

On top of the size required by the input time series \( (O(n)) \), the output graph \( (O(n^2)) \), and the auxiliary variable \( \text{slope}_{\text{max}} \ (O(1)) \), this algorithm also requires additional space in memory to store the recursive auxiliary structures and stack. There will be a total of \( n \) recursive calls made during the whole execution of the algorithm (essentially one function call is made for each data point in the time series). In the worst case, the recursion stack will need to be of size \( O(n) \) at one same single moment, but in balanced time series the recursion stack might only reach a maximum concurrent size of as little as \( O(\log n) \).

The size of the stack and the required auxiliary structures can be substantially reduced in some cases with the use of certain tail call optimizations in the programming language used for the implementation. Additionally, a non-recursive version of this algorithm can be constructed by replacing the recursive function calls with the use of an explicit stack or queue data structure combined with an iterative loop (more information on this implementation strategy is provided in Section 4.3.2). In these cases, significantly lower general memory usage is expected, but the asymptotic space in memory required for the stack or queue might still be in the order of \( O(n) \) in the (not common) worst case.

In summary, the memory requirements of this divide-and-conquer algorithm are greater than those of the two previous algorithmic strategies, but the additional space required is still small when compared with the potential space in memory required for the output graph and thus, the total asymptotic memory cost of this divide-and-conquer algorithm remains at \( O(n^2) \).
3.4 Streaming algorithm

Algorithm 4 Visibility Graph - Streaming Algorithm

1: \( G \leftarrow (V = \emptyset, E = \emptyset) \)
2: \( T \leftarrow \emptyset \)
3: \( i_{\text{max}} \leftarrow -1 \)
4: \( y_{\text{max}} \leftarrow -\infty \)
5: 
6: function \textsc{UpdateVisibilityGraph}(\text{newpoint} = (t_{n+1}, y_{n+1}))
7: \quad \text{add node } n+1 \text{ to } V
8: \quad \text{limit} \leftarrow i_{\text{max}}
9: \quad \text{if } y_{n+1} \geq y_{\text{max}} \text{ then}
10: \quad \quad \text{if } y_{n+1} > y_{\text{max}} \text{ then}
11: \quad \quad \quad \text{limit} \leftarrow 0
12: \quad \quad \quad i_{\text{max}} \leftarrow n+1
13: \quad \quad \quad y_{\text{max}} \leftarrow y_{n+1}
14: \quad \quad \text{for } a \leftarrow n \text{ down to limit do}
15: \quad \quad \quad \text{if node } n+1 \text{ ‘sees’ node } a \text{ then} \quad \triangleright \text{ i.e. using the slope}
16: \quad \quad \quad \quad \text{strategy locally}
17: \quad \quad \quad \quad \text{add edge } (n+1, a) \text{ to } E
18: \quad \quad \quad \text{return } G = (V, E)
19: end function

Streaming algorithms are a big part of today’s big data world, these algorithms are used to process large data streams where the input is given sequentially and should be processed in one or just a few passes. In many models, these algorithms have access to limited memory such that the whole input cannot be stored.

In this section we present the pseudocode for a simple strategy to obtain a visibility graph from a stream of time series data, which can also be used to update and expand existing visibility graphs when new time series observations become available (Algorithm 4).

In this algorithm the graph gets built iteratively, adding nodes one at a time. In a naive streaming approach, when a new observation arrives and a node \( n+1 \) gets added to an existing graph, we would have to check all \( n \) potential edges by checking the visibility with all previously existing \( n \) data points. But we can also take advantage of the same ideas learned with the divide-and-conquer strategy previously described, by which we know that the highest data point in the series acts as an effective barrier in the visibility of nodes at each side of this highest value. Therefore, if we store the index for the highest value in the time series \( i_{\text{max}} \), we will only need to check the visibility with the data points in the range \([i_{\text{max}}, n]\), as no data point to the left of \( i_{\text{max}} \) will ever be visible. In the particular case that the newly added observation is higher than the previous value at \( i_{\text{max}} \) then the whole time series needs to be checked and the value for \( i_{\text{max}} \) will need to be updated with \( n+1 \).

Another more advanced strategy optimal for stream-like applications has been very recently presented by Fano Yela et al. 2020 [16] which expands on the previous ideas and makes use of an auxiliary binary tree structure.
We implemented the different visibility graph algorithms described in Chapter 3, with the wider goal of developing a software framework containing different usable functions for converting time series data into visibility graphs. The implementation had to be very efficient, allow for working with time series of any nature of potentially very large lengths, and be easy to use and integrate with other statistics, data science, graph analysis, and visualization tools.

4.1 Programming language

Choosing the programming language for our implementations was an important decision and could not be easily changed at a later point in time. After considering different options (e.g., C, C++, R, MATLAB), the Python programming language was chosen due to its flexibility, simple syntax, the familiarity of the author with the language, and availability of many other existing and easy-to-use libraries and frameworks to work on statistics, data science, graph analysis, data visualization, and many other fields that could be explored in the future.

Python is a very flexible scripting, dynamically-typed, and interpreted programming language that was first introduced in 1990 [21] and that has gained significant popularity in recent years. Python is also being increasingly used in scientific applications and research [38].

4.2 Time series implementation

In order to simplify our implementation and make the code more efficient we will implement and store any time series as a single one-dimensional vector or array, containing the ordered heights $y_i$ of the time series:

$$ T = [y_1, y_2, \ldots, y_n] $$

For this to be a valid representation of time series we assume that we will be working only with time series whose data points are equally spaced in time, and therefore, all required time information to construct the visibility graph is implicit in the order of the elements in the array. For information and data visualization purposes we might want to sometimes store a separate auxiliary array containing the timestamps of the observations: $t_1, t_2, \ldots, t_n$.

We are using the numpy package [46] and its array structures to store and work with time series as they provide significant performance benefits over the simpler native Python data structures. Arguably, numpy has become the standard tool for storing and manipulating arrays and matrices in all kinds of data science projects in Python.
4.3 Visibility graph algorithms implementation

We implemented all the different visibility graph algorithmic strategies described in Chapter 3 and integrated them in the Python programming language. We decided to implement the three versions (and not just the apparently optimal divide-and-conquer strategy) in order to be able to assess our theoretical complexity study, better understand their practical behavior and compare their estimated performance in real-world situations.

A selection of the source code developed for our functions implementations is included in Appendix B. In Figure 4.1 we show the visualization of a visibility graph obtained from an example time series and as computed with our implemented visibility graph functions.

![Generated visibility graph of a Wiener process time series of length $10^3$ (same time series as the one previously shown in Figure 2.2 in Section 2.1.1) and as obtained with our implementation of the algorithms. The resulting visibility graph consists of 1000 nodes and 6236 edges.](image)

**Figure 4.1:** Generated visibility graph of a Wiener process time series of length $10^3$ (same time series as the one previously shown in Figure 2.2 in Section 2.1.1) and as obtained with our implementation of the algorithms. The resulting visibility graph consists of 1000 nodes and 6236 edges.

4.3.1 Backend optimizations with C and Cython

The flexible and interpreted nature of Python is in many cases an advantage over other programming languages, and arguably helped Python become more popular, but it also introduces noticeable drawbacks. It is well known that Python is not the most computationally efficient of programming languages, and is often not the right choice when high performance with large amounts of data is wanted. Native Python scripts are far from providing the same level of computational performance as programs written in statically-typed and compiled programming languages like C, C++, or Fortran, but these can often be more difficult to work with, requiring significant additional programming complexity, featuring lower-level abstraction, and offering more difficult integration with other existing statistics, data science, visualization, etc. tools and libraries.

Different libraries and tools have emerged through the years to overcome the performance shortcomings of Python with different degrees of success and performance results. One of the more robust and standard of such tools is Cython [5, 6], which attempts to provide a bridge between Python and C. To do this, Cython introduces its own programming language, a superset of the Python language, that additionally supports easy integration of calls to C functions at any point in the code and provides static-typing features for its variables and class attributes, allowing for the code to then be compiled into efficient C code. Cython allows our functions to achieve a performance equivalent to that of native C code, while at the same time preserving the benefits of Python’s high-level nature and the integration with its many libraries.

Many well-known Python packages and libraries (such as NumPy [46], SciPy [52] or Pandas [38]) also implement most of their backend functions in the C, C++, or Fortran programming languages to do the heavy lifting (with native code in those languages or aided by tools like Cython).
4.3.2 Iterative algorithm for recursion

In the case of the divide-and-conquer algorithm as described in Section 3.3, obtaining the visibility graph from time series depends on a function being recursively called. Recursive function calls often entail notable overhead, both in additional time and memory cost. Furthermore, programming languages might have significant practical limitations in their maximum recursive depth allowed. In particular, Python has a recursion limit of 1000 nested function calls by default and does not provide any kind of tail recursion optimization [51] (essentially limiting our input time series to a length of 1000 in the worst case). Although these limitations can be in part reduced with the use of Cython, the results are still not the most efficient.

As is the case with many recursive algorithms, a more direct (but perhaps less intuitive) iterative approach can instead be used. We implemented the recursive divide-and-conquer strategy with the use of a while loop iterating over an explicit queue until it is emptied, and replacing the original recursive calls by pushes of a pair (left, right) to said queue.

4.3.3 Performance analysis

Figure 4.2 shows the execution times of our implementation of the divide-and-conquer algorithm for visibility graphs for time series of different lengths \( n \) and two groups of time series (Gaussian white noise and Wiener process series). The results shown are averaged over 10 executions of different series. We can see that our implementation will perform in near-instant time in most practical use cases. See Appendix A for additional detailed information on the software and hardware used for these tests. A detailed empirical performance study and comparison of the three algorithmic strategies is also later presented in Section 5.1.

![Figure 4.2: Execution times of our implementation of the divide-and-conquer algorithm on Gaussian white noise and Wiener process time series with different lengths \( n \) (averaged over 10 samples each). The Gaussian white noise series consistently require less time due to their structure in which the maximum values of the series are more uniformly distributed, resulting in more balanced splits.](image)

4.4 ts2vg package

From the lessons learned and using the algorithms implemented and described for this project, we developed and published a Python package named ts2vg (`time series to visibility graphs`), an object-oriented library to easily and efficiently compute visibility graphs from time series.
The package benefits from an efficient C backend for its functions (with the use of Cython) while still providing native integration in the Python environment. Therefore, ts2vg can easily work with input data from many sources and providers as available with existing Python tools, as well as allowing the study and analysis of the resulting visibility graphs with any of the many existing graph analysis, data science, and visualization packages and tools available for Python.

A minimum working example, useful for many use cases, of ts2vg as used in a Python script can be:

```python
from ts2vg import NaturalVisibilityGraph
ts = [0.87, 0.48, 0.36, 0.83, 0.87, 0.48, 0.36, 0.83]
edges = NaturalVisibilityGraph(ts).edgelist()
```

Similarly, an igraph object representation of the graph [12] (see Section 4.5.1) can be very easily obtained with:

```python
vg = NaturalVisibilityGraph(ts).as_igraph()
```

The divide-and-conquer strategy is used in all cases by default. ts2vg can also be used as a stand-alone command line program useful to quickly obtain visibility graphs from time series without having to type any Python code. As a basic example, the following command can be used from the console to obtain the edge list (as saved to a file `out.edg`) of the visibility graph resulting from an input time series found in a local `timeseries.txt` file:

```bash
>>> ts2vg ./timeseries.txt -o out.edg
```

Full source code, documentation, and installation instructions for the ts2vg Python package and program can be found at:

https://github.com/CarlosBergillos/ts2vg

### 4.4.1 Potential future considerations

The current state of our implementation and the associated ts2vg package, although stable, and successfully fulfilling the initial requirements, need not be final, it allows for future additions and improvements, some of which are out of the scope of the project or proved not viable due to technical or time limitations. It could be expanded and improved, for example, in the following ways:

- Adding new functionalities by providing implementations for other visibility graph algorithms (e.g. horizontal visibility graphs) and/or enrichening and expanding the existing ones (e.g. directed and weighted versions of visibility graphs).
- Exploring advanced performance improvements to the existing algorithms, for example, with the use of parallel and multithreaded strategies or with the use of GPU accelerated implementations, exploring tools like CUDA [44].

The above are just presented as ideas worthy of consideration, but in no case are planned future developments and might even prove to not be practical.
4.5 Auxiliary functionalities

To conduct experiments and explore potential visibility graph applications (see Chapter 5) we also needed the help of many auxiliary functionalities, whose implementation was outside of the scope of this project, and in most cases pointless due to different already existing and well-established Python libraries suited for the tasks.

4.5.1 Graph analysis functionalities

In order to conduct different graph analysis tasks, we used the Python igraph package [12]. This package provides many easy-to-use graph construction, manipulation, and analysis functionalities, such as, to name a few, obtaining their degree distribution, obtaining path lengths and many other measures, applying different clustering algorithms and plotting the graphs using different visualization strategies. The Python igraph package is (for the most part) only an interface for a core implementation of igraph written in C, again, benefiting from more efficient and portable methods.

Full official documentation for igraph has been used and can be found in [50].

4.5.2 Generating Wiener process and fractional Brownian motion series

A major part of our experimentation with visibility graphs required the availability of stochastic time series as input for the visibility graph algorithms. Understanding the behavior of Wiener process series and fractional Brownian motion series can help in forming conclusions that might then be useful for real-world time series.

We used the Python stochastic package [17] to construct these series. The implementations of fractional Brownian motion series provided by this package use the methods described by Hosking 1984 [22] and by Davies and Harte 1987 [13].

4.5.3 Estimating power-law exponents

We used our own Python implementation (with the support of numpy functions) of the formula for estimating the exponent $\gamma$ of power-law data using MLE (equation (2.16) shown in Section 2.6 and based on [41]). For more complex distribution estimations we also studied and considered using the Python powerlaw package [1].
As part of our research on visibility graphs we will empirically test some of the ideas described in previous chapters, we will further study some of the properties of visibility graphs and we will explore some of their potential applications.

We will be using our visibility graph algorithms implementations (using the Python programming language and Cython, as described in Chapter 4). Detailed information on relevant software packages and hardware used for all the experiments conducted and described in this chapter can be found in Appendix A.

5.1 Performance evaluation of the different algorithms

In Chapter 3 we described and mathematically evaluated the time and space complexity of three different algorithm strategies that can be used to compute visibility graphs from an input time series. To verify such theoretical analysis, we decided to test and measure the time performance of our implementations of the algorithms.

In Table 5.1 we show the execution time of the different visibility graph algorithms used on Gaussian white noise time series of different length. As expected, there is a clear difference in the running time of the different algorithmic strategies, the divide-and-conquer strategy being the fastest one, and, on the other hand, the naive algorithm the slowest one.

It is important to consider that the visibility graph algorithms can require drastically different execution times for different input time series, even when of the same length \( n \). For example, the execution time of the divide-and-conquer algorithm depends both on the time series length \( n \), and on the distribution of maximum values through the time series (how balanced the time series is). Gaussian white noise time series can be considered a very well balanced time series, and thus greatly benefit from the divide-and-conquer algorithm.

Table 5.2 shows the execution time of the same algorithms but this time with Wiener process time series as input, a less balanced group of time series, but maybe closer to more realistic use cases of visibility graphs. In this case, as expected the slope algorithm shows no significant time difference when compared to the previous white noise case, but both the naive and the divide-and-conquer algorithms require noticeable longer execution times.

From these results and the theoretical analysis from previous sections, we can see and conclude that there is no real reason to prefer the naive or the slope algorithms over the divide-and-conquer algorithm for any practical application. The divide-and-conquer algorithm shows significantly better time performance at the expense of a very small additional memory penalty.
### 5.2 Periodic series result in regular visibility graphs

Periodic time series are those formed by the repetition of one same subsequence over and over:

\[ T = (y_1, y_2, \ldots, y_p, y_1, y_2, \ldots, y_p, y_1, y_2, \ldots, y_p, \ldots) \]

The length \( p \) of this subsequence that gets repeated is called the *period* of the series. Two examples of periodic time series are shown in Figure 5.1.

The visibility graph resulting from a periodic time series will be formed by the many repetitions of one same graph *motif*, as each instance of the repeated subsequence results in the same equivalent subgraph.

Therefore, the resulting visibility graphs present a very regular degree distribution, as there will be at most \( p \) unique degree values for all \( n \) nodes in the graph (excluding the ones from the nodes in the two endpoint segments, but the effect of those on the degree distribution is negligible.

---

#### Table 5.1: Execution times of the different algorithmic strategies (naive, slope, and divide-and-conquer) when used to obtain the visibility graph of Gaussian white noise time series of different lengths \( n \) (averaged over 10 samples each).

<table>
<thead>
<tr>
<th>( n \times 10^4 )</th>
<th>Naive Algorithm (s)</th>
<th>Slope Algorithm (s)</th>
<th>D&amp;C Algorithm (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.6141</td>
<td>0.0964</td>
<td>0.0066</td>
</tr>
<tr>
<td>2</td>
<td>2.6288</td>
<td>0.3736</td>
<td>0.0140</td>
</tr>
<tr>
<td>3</td>
<td>6.1971</td>
<td>0.8321</td>
<td>0.0213</td>
</tr>
<tr>
<td>4</td>
<td>11.3760</td>
<td>1.4703</td>
<td>0.0289</td>
</tr>
<tr>
<td>5</td>
<td>18.3866</td>
<td>2.2898</td>
<td>0.0366</td>
</tr>
<tr>
<td>6</td>
<td>26.6579</td>
<td>3.2924</td>
<td>0.0439</td>
</tr>
<tr>
<td>7</td>
<td>36.8757</td>
<td>4.4717</td>
<td>0.0511</td>
</tr>
<tr>
<td>8</td>
<td>48.6662</td>
<td>5.8315</td>
<td>0.0588</td>
</tr>
<tr>
<td>9</td>
<td>62.4979</td>
<td>7.3732</td>
<td>0.0668</td>
</tr>
<tr>
<td>10</td>
<td>78.4211</td>
<td>9.0952</td>
<td>0.0750</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>-</td>
<td>0.7977</td>
</tr>
</tbody>
</table>

#### Table 5.2: Execution times of the different algorithmic strategies (naive, slope, and divide-and-conquer) when used to obtain the visibility graph of Wiener process time series of different lengths \( n \) (averaged over 10 samples each).

<table>
<thead>
<tr>
<th>( n \times 10^4 )</th>
<th>Naive Algorithm (s)</th>
<th>Slope Algorithm (s)</th>
<th>D&amp;C Algorithm (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.4416</td>
<td>0.1099</td>
<td>0.0246</td>
</tr>
<tr>
<td>2</td>
<td>23.6012</td>
<td>0.3977</td>
<td>0.0565</td>
</tr>
<tr>
<td>3</td>
<td>66.0205</td>
<td>0.8730</td>
<td>0.0866</td>
</tr>
<tr>
<td>4</td>
<td>134.4352</td>
<td>1.5266</td>
<td>0.1242</td>
</tr>
<tr>
<td>5</td>
<td>227.7911</td>
<td>2.3682</td>
<td>0.1526</td>
</tr>
<tr>
<td>6</td>
<td>417.3478</td>
<td>3.3837</td>
<td>0.2018</td>
</tr>
<tr>
<td>7</td>
<td>653.0515</td>
<td>4.5799</td>
<td>0.2308</td>
</tr>
<tr>
<td>8</td>
<td>703.4591</td>
<td>5.9542</td>
<td>0.2841</td>
</tr>
<tr>
<td>9</td>
<td>907.9010</td>
<td>7.5241</td>
<td>0.3352</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>9.2722</td>
<td>0.4089</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>-</td>
<td>8.4202</td>
</tr>
</tbody>
</table>
when $n$ is large, with $p \ll n$). It can also be easily seen that the degree distribution will stay essentially constant independently of the length $n$ of the periodic time series (for a fixed, repeated subsequence of length $p$, with $p \ll n$).

Additionally, these degrees will be quite small as the highest data point in the sequence appears repeatedly every $p$ observations, effectively segmenting the whole time series into repeated independent regions. In particular, each of the nodes corresponding to the highest data point in the segments will have degree at most $2p$, and all other non-maximum nodes will have degree at most $p$.

Clearly, there are no hub nodes, no power-law distribution, and the diameter and average shortest path length are very large (on the order of $n/p$).

In Figure 5.2 we show the degree distributions of the visibility graphs obtained from the two time series shown in Figure 5.1 (when continued up to length $10^5$). Arguably, the discrete degree distributions resulting from periodic time series can be considered as a sort of fingerprint of the time series periods [30].

\textbf{Figure 5.1}: First 500 observations of two example periodic time series, both of repeating period 32.
Figure 5.2: Degree distributions of the two periodic time series, A and B, shown in Figure 5.1, continued up to length $10^5$. Note that for both time series no node has degree larger than 32 (the period). The distributions show clearly distinct discretized structures that could be used as a sort of fingerprint of the periodic time series. This distribution will stay essentially constant with other varying lengths $n$ of the series, as long as the repeated period subsequence stays the same.

5.3 Fractal series result in scale-free visibility graphs

An interesting finding in the field is that visibility graphs resulting from fractal series (like Wiener process series, fractional Brownian motion series, and other mathematical and real-world series of similar fractal behavior) present small-world and real-world network properties [30]. We generated different samples of Wiener process series (such as the one shown in Figure 5.3), obtained their visibility graph, and confirmed that they match the properties previously presented in Section 2.3, usually associated to real-world networks.

Figure 5.3: First 1000 observations of an example Wiener process time series (also known as Brownian motion).

These properties are:

**Small diameter** These visibility graphs present diameter lengths and average shortest path lengths that are orders of magnitude smaller than the number of nodes in the graphs. In Figure 5.4 we show empirical measurements for these values as a function of the time series length $n$. Both the diameter and the average shortest path lengths seem to grow logarithmically with respect to the graph size $n$. We observe very small path lengths joining any two nodes, i.e. paths consisting of less than 10 edges even in graphs with millions of nodes. This is possible, in part, thanks to the presence of some nodes with a very high degree, commonly known as hub nodes that allow quick navigation through the network.

**High clustering coefficient** In the case of the Wiener time series shown in Figure 5.3, it presents a remarkably high average clustering coefficient (also known as transitivity),
with values of $0.685 \pm 0.01$ for many different lengths $n$ considered for the series. This finding also holds for other Wiener process series studied.

**Power-law degree distribution** As shown in Figure 5.5 the Wiener process time series presents a degree distribution clearly compatible with a power-law distribution (starting after a certain $x_{\text{min}}$ value, and getting progressively noisy at the end of its tail). From this, it is clear that most of the nodes in the graph have a very small degree, and only a small amount of nodes have a large degree. These nodes of large degree (hub nodes) might correspond to regional peaks in the input time series (data points with higher $y$ value than their surroundings, and therefore having direct visibility with many other data points in the series).

![Figure 5.5](image.png)

**Figure 5.5:** Linear plot (left) and log-log plot (right) of the degree distribution of the visibility graph obtained from the Wiener time series shown in Figure 5.3 and as continued to length $10^5$. The tail of the degree distribution visually follows a distinct power-law shape.

5.4 Estimating the Hurst exponent via the visibility graph

The Hurst exponent of time series is an important and useful measure used in time series analysis. It provides information on the volatility of the series, and can have practical applications, for example, on financial time series analysis, and it could aid in trying to predict the future behavior of the series. Visibility graphs obtained from time series have been shown to provide a new unique
way of estimating the Hurst exponent of such time series. In particular, the exponent of the power law that approximates the degree distribution of the visibility graph is used for this estimation. More so, a linear dependency between this power-law exponent and the Hurst exponent of the time series has been proven to exist [31, 42].

Fractional Brownian motion series (characterized by the Hurst exponent) can be used to model certain financial series and real-world time series. In Figure 5.6 we show the degree distributions of three visibility graphs obtained from fractional Brownian motion series characterized by different values of $H$. As it can be seen, the three visibility graphs exhibit scale-free behavior and can be characterized as power laws $P(k) = ck^{-\gamma}$ of different shape. For each of the distributions, we have estimated its best-fitting exponent $\gamma$ using Maximum Likelihood Estimation (MLE) as explained in Section 2.6 and using $x_{\text{min}} = 8$. And at first glance, we can observe that the values for $\gamma$ decrease as $H$ increases.

![Figure 5.6: Degree distributions (in log-log axes) of three visibility graphs each obtained from a different fBm series of length $10^5$ with $H = 0.2$ (left), $H = 0.5$ (center) and $H = 0.8$ (right). The tails of the three degree distributions follow a power law $P(k) = ck^{-\gamma}$, each one with a different shape and with noticeably different exponents $\gamma$.](image)

Lacasa et al. 2009 [31] showed that the exponent $\gamma$ of visibility graphs is linearly related to the Hurst exponent $H$ of the series in the form:

$$\gamma(H) = 3 - 2H$$  \hspace{1cm} (5.1)

We conducted a concise experiment trying to replicate these findings. In Figure 5.7 we show the result of empirical tests of this relationship between $\gamma$ and $H$ and we compare it with the previous theoretical formula (5.1). The empirical tests seem to follow the theoretical formula quite accurately, but it can be observed that the values of $\gamma$ start to deviate for values of $H$ close to 0 and close to 1, and being the most accurate near $H = 1/2$. These small deviations could be due to biases in the fBm series generated, limitations of working with series of finite length, and inaccuracies in the values of $\gamma$ obtained through MLE.
Chapter 5  Applications and Experiments

5.5  Segmentation of time series via graph clustering

Automatic segmentation of time series can also be a useful application of visibility graphs. Time series segmentation consists on dividing an input time series into a sequence of discrete and disjoint segments according to some notable underlying property. For example, we might be interested in segmenting an almost periodic time series (i.e. a periodic time series that might have been subjected to noise on the $y$ and/or $t$ dimension). This can be achieved on certain time series by applying clustering algorithms to their resulting visibility graphs, and extrapolating the discovered clusters (or communities) to the original time series data points. Different clustering algorithms can be used to solve the task, achieving potentially different results. A clustering algorithm based on modularity optimization presented by Blondel et al. 2008 [7] has been used in all our experiments presented in this section.

In Figure 5.8-A we show the result of segmenting an almost-periodic time series using this technique. We also show the result of two other time series examples (B and C) whose periodicity is much more diffuse, but the technique still manages to divide the time series into segments of distinct behavior.

Note that due to the nature of visibility graphs, clusters of nodes are likely to correspond to valleys of the time series (where direct visibility among data points is easier), and regional peaks of the series are likely to divide segments. Therefore, the resulting segmentation of the time series will also reflect this behavior. It is easy to see that peak-centered segments (separated by valleys) could also be obtained by multiplying the time series heights by $-1$ before obtaining its visibility graph.

It is worth noting that the clustering algorithms used have no notion of time in their graph analysis, but still sequential-in-time clusters naturally emerge in many practical cases. More complex techniques could be used that include a weighting of the edges based on the difference in time between two connected nodes, therefore aiming to avoid broken-in-time clusters as it happened in Figure 5.8-C. Additionally, it can be theorized that different possible segmentations of one same time series (corresponding to the potentially overlapped different frequencies of the time series) could be obtained by optimizing for clusters of different sizes in each case.
Figure 5.8: Results of time series segmentation via visibility graph clustering used in three different input time series. Each color corresponds to a distinct cluster/segment obtained in the time series. An example almost-periodic time series is used in A, and two chosen Wiener-process-like series in B and C.
In this project we have presented a study and overview of the so-called visibility graphs and their use applied to time series analysis as it has emerged in recent years. These visibility graphs are characterized by a relatively simple mathematical definition, but still present a notable computational challenge in their implementation when efficiency is a factor. We have seen how the simple and most intuitive algorithms to obtain visibility graphs prove to not be usable in time series of realistic sizes, and we compared its performance with that of other more advanced algorithms, such as a divide-and-conquer algorithm that achieves a running time complexity on the order of $O(n \log n)$.

Potential applications of visibility graphs have recently started to be explored in very different fields such as medicine, physics, or finances, and this exploration will very likely continue in the upcoming future. For this reason, the availability of accessible and efficient implementations of visibility graph algorithms seems to be of significant importance. We developed and introduced the ts2vg software library which provides accessible and efficient implementations of visibility graph algorithms for the Python programming language with an optimized and fast backend in the C programming language.

We empirically tested and compared the performance of different algorithms to compute the visibility graph of time series and later studied and replicated some of the known analysis on the properties of visibility graphs. We have seen how the degree distribution of these visibility graphs can provide key information on the structural properties of time series, we have seen how visibility graphs can be used to identify periodic series, we have seen how fractal time series such as Wiener processes or fractional Brownian motion series result in real-world-like and scale-free visibility graphs and whose Hurst exponent can be estimated from the graph’s degree distribution, and we explored how clustering algorithms, when applied to visibility graphs, can help in the segmentation of certain time series.

With this, we consider that we successfully met our initial goals of this project, which could be very quickly summarized as researching mathematical and computational aspects of visibility graphs, developing suitable tools and algorithms, and exploring possible applications for them. If more time had been available to work on this project we would have liked to explore in further detail other properties of visibility graphs and we would have liked to experiment with real-world time series and data sets.

Additionally, there are many lines of work that can be continued in future research and developments. There are some significant ways in which the ts2vg Python library can be expanded, such as by adding implementations for other visibility algorithms (e.g. horizontal visibility graph algorithms) or considering variations of the existing ones such as to obtain directed and/or weighted graphs (e.g. based on line-of-sight angle or distance) and studying the implementation of other computational strategies such as the one very recently proposed by Ghosh and Dutta 2019 [20].
described as a sort-and-conquer algorithm. And, although maybe unnecessary for current applications, it could also be interesting to explore more advanced performance techniques for the algorithms, such as the use of parallel and multithreaded strategies or GPU accelerated implementations.

It is worth mentioning, and as it has been nicely explained by Nuñez et al. 2012 [43], that before being able to reliably use visibility graphs for real-world applications in time series analysis, a sound and rigorous mathematical theory must be developed and constructed that explains the behavior and properties of visibility graphs and provides a justified mathematical connection with the properties of the time series they are obtained from. Such a broad and general mathematical theory is still to be completed.
Bibliography


Appendices
A

Conditions and Specifications

To conduct all the performance analysis, visibility graph tests, and experiments presented throughout this project we have used one same computer with the characteristics and specifications shown in Table A.1.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel® Core™ i5-4690K, 4 Cores @ 3.50GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GB</td>
</tr>
<tr>
<td>Memory</td>
<td>Crucial® MX500 1000GB SSD</td>
</tr>
<tr>
<td>OS</td>
<td>Manjaro Linux 19.0.2 Kyria (rolling release 03-2020)</td>
</tr>
</tbody>
</table>

Table A.1: Details and specifications of the computer used to conduct the experiments presented in this project.

Details on the release versions used for Python and for any other relevant software tools are shown in Table A.2.

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>3.8.2</td>
</tr>
<tr>
<td>Cython</td>
<td>0.29.17</td>
</tr>
<tr>
<td>gcc</td>
<td>9.3.0</td>
</tr>
<tr>
<td>numpy</td>
<td>1.18.2</td>
</tr>
<tr>
<td>igraph</td>
<td>0.8.0</td>
</tr>
<tr>
<td>stochastic</td>
<td>0.4.0</td>
</tr>
</tbody>
</table>

Table A.2: Details on the Python release version and on the associated relevant software tools and libraries used for the implementations and experiments presented in this project.

Additionally, all diagrams and figures shown throughout this whole project have been created by the author with the help of Python and Matplotlib [23].
Source code for working implementations of the visibility graph algorithms that have been described and studied in Chapter 3 are presented here. These implementations, as provided here, have been used for all experiments conducted and presented in Chapter 5. The programming language used is Cython, a superset of the Python programming language with additional support for static typing and easy integration of C methods.

Additionally, we later provide information on the source code for the ts2vg package.
B.1 Naive strategy implementation

```python
def visibility_graph_naive(np.float64_t[:] ts):
    """
    Computes the visibility graph of a time series using a naive strategy.
    
    Args:
    ts (npy 1d array): Time series data.
    
    Returns:
    g (igraph.Graph): The visibility graph of 'ts'.
    """

cdef uint n = ts.size
    g = Graph()
    g.add_vertices(n)
    edges = []

cdef uint a, b

    for a in range(n-1):
        for b in range(a+1, n):
            if _no_obstructions(ts, a, b):
                edges.append((a, b))

    g.add_edges(edges)

    return g

cdef bint _no_obstructions(np.float64_t[:] ts, uint a, uint b):
    """
    True if there is direct line-of-sight visibility (no intermediate obstructions) between data points at positions 'a' and 'b' in the time series 'ts'. False otherwise.
    """

cdef float y_a = ts[a]
cdef float y_b = ts[b]
cdef float y_c
cdef uint c

    for c in range(a+1, b):
        y_c = ts[c]
        if y_c >= y_b + (y_a-y_b) * (b-c) / (b-a): return False

    return True
```

49
B.2 Slope strategy implementation

```python
def visibility_graph_slope(np.float64_t[:,:] ts):
    """
    Computes the visibility graph of a time series
    using a slope strategy.
    
    Args:
    ts (numpy 1d array): Time series data.
    
    Returns:
    g (igraph.Graph): The visibility graph of 'ts'.
    """

cdef uint n = ts.size
g = Graph()
g.add_vertices(n)
edges = []

cdef uint a, d
cdef float y_a, y_b
cdef float max_slope, slope

for a in range(n-1):
y_a = ts[a]
max_slope = -INFINITY
for d in range(1, n-a):
y_b = ts[a+d]
slope = (y_b-y_a) / d # d = b-a
if slope > max_slope:
edges.append((a, a+d))
max_slope = slope

g.add_edges(edges)

return g
```
B.3 Divide-and-conquer strategy implementation

```python
def visibility_graph_dc(np.float64_t[:]] ts):
    """
    Computes the visibility graph of a time series using a divide-and-conquer strategy.
    """
    cdef uint n = ts.size
    g = Graph()
    g.add_vertices(n)
    edges = []

cdef uint left, right, i, d
cdef float y_i, y_a
cdef float max_slope, slope

cdef PairQueue queue = PairQueue()
queue.push((0, n))

while not queue.is_empty():
    (left, right) = queue.pop()
    if left + 1 < right:
        i = _argmax(ts, left, right)
        y_i = ts[i]

        max_slope = -INFINITY
        for d in range(1, i-left+1):
            y_a = ts[i-d]
            slope = (y_a-y_i) / d # d = a-i
            if slope > max_slope:
                edges.append((i, i-d))
                max_slope = slope

        max_slope = -INFINITY
        for d in range(1, right-i):
            y_a = ts[i+d]
            slope = (y_a-y_i) / d # d = a-i
            if slope > max_slope:
                edges.append((i, i+d))
                max_slope = slope

        queue.push((left, i))
        queue.push((i+1, right))

    g.add_edges(edges)

return g
```

51
B.4 Streaming algorithm implementation

```python
class VisibilityGraphStream:
    """
    Class representation for a visibility graph updated via a stream of data.
    The graph object can be accessed at any point via the variable 'vg'.
    """

    def __init__(self):
        """
        Method executed on initialization.
        """
        self.vg = Graph()
        self.ts = np.empty(0)
        self.n = 0
        self.__max_i = -1
        self.__max_v = -INFINITY

    def update(self, new_y):
        """
        Updates the visibility graph with one new time series observation.
        A new node is added to the graph, as well as all required new edges.
        """
        self.vg.add_vertex()
        self.ts = np.append(self.ts, new_y)
        edges = []
        max_slope = INFINITY
        limit = self.__max_i

        if new_y >= self.__max_v:
            if new_y > self.__max_v:
                limit = 0
                self.__max_i = self.n
                self.__max_v = new_y

        for a in reversed(range(limit, self.n)):
            slope = (self.ts[a] - self.ts[self.n])/(a - self.n)
            if slope < max_slope:
                edges.append((self.n, a))
                max_slope = slope

        self.vg.add_edges(edges)
        self.n += 1
```

52
B.5 ts2vg package

A selection of the previously shown functions and algorithms has been included and adapted in the ts2vg Python package that we have developed. ts2vg aims to be an easy-to-use and fast tool for anyone to obtain and work with visibility graphs both in Python and directly through a command-line interface.

The full source code for the ts2vg package is public and can be found at:

https://github.com/CarlosBergillos/ts2vg
C

Project Management

C.1 Tasks

A summary of the tasks required in order to successfully complete this project is provided in Table C.1, a more detailed description of tasks and their subtasks is provided later.

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Est. Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T.0 Project Management (GEP)</strong></td>
<td>Project planning, management and assessment as required for the GEP course.</td>
<td>90</td>
</tr>
<tr>
<td><strong>T.1 Literature Review &amp; Research</strong></td>
<td>Research and study on visibility graphs topics and all the related time series, complex networks, and data science ideas.</td>
<td>100</td>
</tr>
<tr>
<td><strong>T.2 Algorithms Study</strong></td>
<td>Studying different visibility graphs algorithms, evaluating their correctness and computational complexity.</td>
<td>80</td>
</tr>
<tr>
<td><strong>T.3 Algorithms Implementation</strong></td>
<td>Implementing and testing the visibility graph algorithms and related algorithms and functions.</td>
<td>100</td>
</tr>
<tr>
<td><strong>T.4 Applications &amp; Experiments</strong></td>
<td>Conducting case studies and exploring real-world applications for the visibility graph ideas.</td>
<td>100</td>
</tr>
<tr>
<td><strong>T.5 Documentation</strong></td>
<td>Writing the final report of the project and any related documentation, including the creation of any required tables and figures.</td>
<td>60</td>
</tr>
<tr>
<td><strong>T.6 Defense Preparation</strong></td>
<td>Preparing the required slides and material for the oral presentation, rehearsing it and preparing for potential questions.</td>
<td>10</td>
</tr>
</tbody>
</table>

| **Total:** 540                     | **Table C.1:** Summary of the different tasks planned for the project and their estimated durations in hours. |           |
Appendix C  Project Management

T.0  **PROJECT MANAGEMENT (GEP)**  [90 hours]
Project planning, management, and assessment as required for the GEP course.

- **T.0.1 Deliverable 1**
  Work on the first deliverable for the GEP course, this deliverable will focus on the context and scope of the project.

- **T.0.2 Deliverable 2**
  Work on the second deliverable for the GEP course (this document), for this deliverable we will define the project tasks and resources and establish a viable time planning for the project.

- **T.0.3 Deliverable 3**
  Work on the third deliverable for the GEP course, this deliverable will focus on the budget and sustainability aspects of the project.

- **T.0.4 Deliverable 4**
  Work on the fourth deliverable for the GEP course, this deliverable will combine the work and improve on the three previous deliverables.

T.1  **LITERATURE REVIEW & RESEARCH**  [100 hours]
The project requires deep research and study on many concepts and mathematical ideas. Before starting to write any code, we need to conduct research on visibility graphs topics and all the related time series, complex networks, and data science ideas.

- **T.1.1 Visibility Graphs Theory**
  Exploring literature for visibility graphs concepts.

- **T.1.2 Time Series Theory & Models**
  Exploring literature for time series ideas and models such as the Brownian Motion Series.

- **T.1.3 Complex Networks Theory**
  Exploring literature for complex networks theory ideas, their measures, and the properties of real-world networks.

- **T.1.4 Data Science Theory**
  Exploring literature and researching on different data science concepts such as regression models.

T.2  **ALGORITHMS STUDY**  [80 hours]
Studying different visibility graphs algorithms, evaluating their correctness and computational complexity.

- **T.2.1 State-of-the-Art Algorithms**
  Study and conceptualize different state-of-the-art visibility graph algorithms.

- **T.2.2 Complexity Study**
  Analyze and compare the computational complexity of the previously studied algorithms.

T.3  **ALGORITHMS IMPLEMENTATION**  [100 hours]
Implementing and testing the visibility graph algorithms and related algorithms and functions.

- **T.3.1 Research on Tools & Libraries**
  Properly choosing which programming language, tools, libraries and technologies to use for the implementation and programming development is a very important decision, as this cannot be easily changed further down the line once the development has started. For this reason, we are devoting substantial time to research and compare different options available to us.
T.3.2 Visibility Algorithm Implementation
Implementing the functions and algorithms required to obtain visibility graphs from time series.

T.3.3 Other Algorithms & Optimizations
Implementing alternative strategies for obtaining visibility graphs and some of their variants, and later implementing potential optimizations for the algorithms.

T.3.4 Code Correction & Testing
Reviewing, evaluating the correctness of the algorithms with simple tests, cleaning up the code and making it easy to use.

T.4 Applications & Experiments [100 hours]
Conducting case studies and exploring real-world applications for the visibility graph ideas. For example, studying real-world financial series.

T.4.1 Experiments Planning
Planning of the applications and experiments that will need to be conducted, and specifying their requirements.

T.4.2 Conducting Experiments
Conducting the experiments and exploring the applications described and then analyzing their results.

T.5 Documentation [60 hours]
Writing the final report and other related documentation is a very important part of this project. And related figures, tables etc.

T.6 Defense Preparation [10 hours]
In concluding the project, we are showing the results of our work in an oral presentation in front of a panel of examiners. For this, we require previous planning and summarization of the project, working on presentation slides and auxiliary material, rehearsing the presentation, and preparing for potential questions.

C.2 Time planning
A detailed Gantt chart describing the proposed temporal planning for the project and each of its tasks and subtasks is provided in Figure C.1.
Figure C.1: Gantt chart showing the proposed temporal planning for the project.
C.3 Budget

In this section we present the budget requirements for this project. We identified all the different sources of expense involved (both direct and indirect), estimated their cost, and considered and planned for any potential deviations.

Some of the resources we made use of have a limited lifespan. The estimated amortization value for these resources can be obtained using the following formula:

\[ Amortization\ value = \frac{C \cdot D}{L} \]

where: 
- \( C \) = Total monetary cost of the resource
- \( D \) = Duration of the project
- \( L \) = Estimated useful life duration (lifespan) of the resource

Our project has a planned duration \( (D) \) of 5 months. More detailed information on the time planning of this project can be found in Section C.2.

C.3.1 Human resources

We analyzed the overall money that would need to be spent on all the human workforce required for the project. We consider different job roles, each one with its own hourly wage and assigned to different tasks of the project.

The values for hourly wages considered are based on estimated real-world wages and they include social security expenses and all other required taxes which accounted for an additional 30%-35% increase.

The planned budget for human resources of the project is detailed in the following table:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Estimated Time (hours)</th>
<th>Price per Hour (€)</th>
<th>Estimated Cost (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Manager</td>
<td>T.0</td>
<td>90</td>
<td>25</td>
</tr>
<tr>
<td>Researcher</td>
<td>T.1, T.2, T.4.1</td>
<td>200</td>
<td>20</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>T.3.1 - T.3.3</td>
<td>75</td>
<td>20</td>
</tr>
<tr>
<td>Tester</td>
<td>T.3.4, T.4.2</td>
<td>105</td>
<td>15</td>
</tr>
<tr>
<td>Documenter</td>
<td>T.5, T.6</td>
<td>70</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>540 h</td>
<td></td>
<td>10375 €</td>
</tr>
</tbody>
</table>

Table C.2: Planned budget for human resources of the project.

C.3.2 Hardware resources

No computer science project can be completed without the help of a computer and the associated hardware. For this project we required both a PC and a laptop computer.

<table>
<thead>
<tr>
<th>Useful Life (months)</th>
<th>Direct Cost (€)</th>
<th>Amortized Cost (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop PC System</td>
<td>72</td>
<td>1300</td>
</tr>
<tr>
<td>Laptop</td>
<td>60</td>
<td>700</td>
</tr>
<tr>
<td>Total</td>
<td>1700 €</td>
<td>150 €</td>
</tr>
</tbody>
</table>

Table C.3: Planned budget for hardware resources of the project.
C.3.3 Software resources

There are various software components that are required to make this project possible. We decided on using free and open-source software whenever possible. This resulted in a planned budget of 0€ for the software resources. Nonetheless, for information purposes, more detailed information for the software budget is given in the following table:

<table>
<thead>
<tr>
<th></th>
<th>Useful Life (months)</th>
<th>Direct Cost (€)</th>
<th>Amortized Cost (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux OS</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Python</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GitHub</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Other Software</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>0 €</strong></td>
<td><strong>0 €</strong></td>
</tr>
</tbody>
</table>

Table C.4: Planned budget for software resources of the project.

C.3.4 Indirect costs

We consider as indirect costs all the expenses that cannot be easily identified as a product or service directly associated with tasks of the project.

In this case, we need to consider the costs associated with a workplace or office, electricity consumption and Internet access. We use an estimation of 300€/month to cover all these indirect costs, resulting in a total of 1500€ required for the whole duration of the project.

C.3.5 Contingencies

We devote an extra 15% of the already estimated budget as a safety margin for any potential unplanned cost overruns. The planned budget so far (human, hardware, software resources, and indirect costs) require a total of 12025€, so as our contingency margin we will allocate an extra rounded amount of 1800€ to the budget.

C.3.6 Unexpected deviations

We need to account for the unplanned events that might require us to spend additional budget and/or re-plan our strategy and planning. These are, all the possible risks and obstacles that we described in previous sections for which we already defined a course of action if encountered.

Each deviation situation defined got assigned a risk percentage, which is an estimation of its likeliness of occurring (being 100% a fully certain event). The cost of each event as we assume it in our budget is factored by this likeliness percentage.

One such deviation would be one caused by one of the computers becoming unusable at some point during the project work, due to a major software or hardware failure. We also need to plan for a potential project duration extension.

The deviation situations considered, their risk, and their cost is presented in the following table:
### C.6 Risk Management

<table>
<thead>
<tr>
<th>Risk</th>
<th>Direct Cost ($\text{\euro}$)</th>
<th>Budget Cost ($\text{\euro}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Breakdown</td>
<td>5</td>
<td>1000</td>
</tr>
<tr>
<td>Project Duration Extension</td>
<td>15</td>
<td>3000</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>4000\euro</strong></td>
</tr>
</tbody>
</table>

Table C.5: Planned budget for software resources of the project.

### C.3.7 Total budget

After considering all the costs previously described, the total planned budget for this project amounts to **14 475 \euro**, see table C.6.

### C.4 Economical impact

This project is considered to be very sustainable in the economical area. We have tried to minimize the budget required, for example, by deciding to extensively and consistently use free and open-source software throughout the project, and by reducing the amount of other direct and indirect costs when possible, resulting in a relatively low budget.

Additionally, due to the non-profit nature of the project, we are not expecting to make any profit as a result of the work on this project.

### C.5 Environmental impact

This project, due to its research nature, will have no direct environmental impact, and little to no indirect environmental impact. We need, though, to consider the energy consumed by the computer components used. Electronic devices such as computers or laptops require an electricity source to function. We cannot guarantee that this electricity is provided by renewable sources, so the electric consumption derived from the project most probably entails associated CO$_2$ emissions to the atmosphere. There is no way to conduct the project without the use of computers, we therefore assume the compromise of causing this small environmental damage which we think is compensated by the potential benefits of a research project like this.

Most of the literature and research material to be consulted, as well as the documentation to be created as a result of this project will be in digital format, therefore avoiding unnecessary paper waste. We will use recycled paper whenever paper use is unavoidable.

With this, we believe we have taken all reasonable measures to minimize any unnecessary environmental damage.
C.6 Social impact

A research-driven project like this does not have a clear measurable social impact. Although demonstrations of the practical use of visibility graphs for data analysis have already been explored in previous works, the complete power of the techniques are still not completely known and their use in the future is hard to predict. But we do not think these techniques pose a social risk higher than any other existing data analysis tools, and can in fact result in significant social benefits through applications in fields like medicine.

The use of open-source software and tools to conduct the project is good for the continuation and progress of these projects and for the overall community. The public release of the project report and documentation will hopefully get more people interested in the topics presented in the project allowing for future research on these to grow, and ultimately resulting in potentially more real-life applications for visibility graphs. Furthermore, with the public release of the \ts2vg\ Python package and its source code, we directly contribute to the development, study, and research on visibility graphs, its properties, and its algorithms by providing new and accessible tools for anyone to use for research and allowing for people to provide improvements and additions to the tool itself.

C.7 Regulations

The project actions are not governed by any particular law or regulation. The project consists on the study, research, development and exploration of algorithmic techniques and tools associated with visibility graphs, and as such, the execution of the project will not have any consequence directly subject to significant regulations.

C.8 Technical competences

Here we list and justify the technical competences belonging to the computing specialization that were chosen to best suit the work conducted in this project.

- **CCO1.1: To evaluate the computational complexity of a problem, know the algorithmic strategies which can solve it and recommend, develop and implement the solution which guarantees the best performance according to the established requirements.**

  This project has had a clear and strong focus on exploring different algorithmic strategies for solving a specific problem (computing a visibility graph from an input time series), and in doing so, we conducted a comprehensive evaluation of the computational complexity (in both time and space in memory) of these different algorithms, as presented in Chapter 3 (Algorithms and Complexity).

- **CCO1.3: To define, evaluate and select platforms to develop and produce hardware and software for developing computer applications and services of different complexities.**

  Evaluating and selecting adequate tools for our algorithm implementations was an important part of the project, as presented in Chapter 4 (Technologies and Implementation). Careful consideration was required to satisfy requirements regarding efficiency, ease of use, accessibility, integration with other existing data science tools and libraries, and the potential for future extensions. This work concluded in the development of the \ts2vg\ Python package.

- **CCO2.2: Capacity to acquire, obtain, formalize and represent human knowledge in a computable way to solve problems through a computer system in any ap-**
Aplicable field, in particular in the fields related to computation, perception and operation in intelligent environments.

In this project we have worked with time series as a formalization and representation of real-world measurements, and, on the other hand, we have also worked with graphs as a formalization and representation of the same data. Visibility graphs as presented provide a unique bridge between the two (traditionally independent) methods.

- **CCO2.4:** To demonstrate knowledge and develop techniques about computational learning; to design and implement applications and system that use them, including these ones dedicated to the automatic extraction of information and knowledge from large data volumes.

This project required a complex and diverse theoretical mathematical background regarding techniques suitable to extract information from potentially very large data sources (e.g. using data science ideas, time series analysis and complex network theory) as presented in Chapter 2 (Definitions and Theoretical Background) and later put into practice in Chapter 5 (Applications and Experiments).

- **CCO3.1:** To implement critical code following criteria like execution time, efficiency and security.

One of the initial goals of the project was to implement efficient solutions to compute visibility graphs even for very large input sizes. Execution time and efficiency of our code has therefore been an important requirement in our implementations, and as described in Chapter 4 (Technologies and Implementation), special attention has been payed to choose adequate tools and libraries and to develop optimized implementations of the algorithmic strategies chosen.