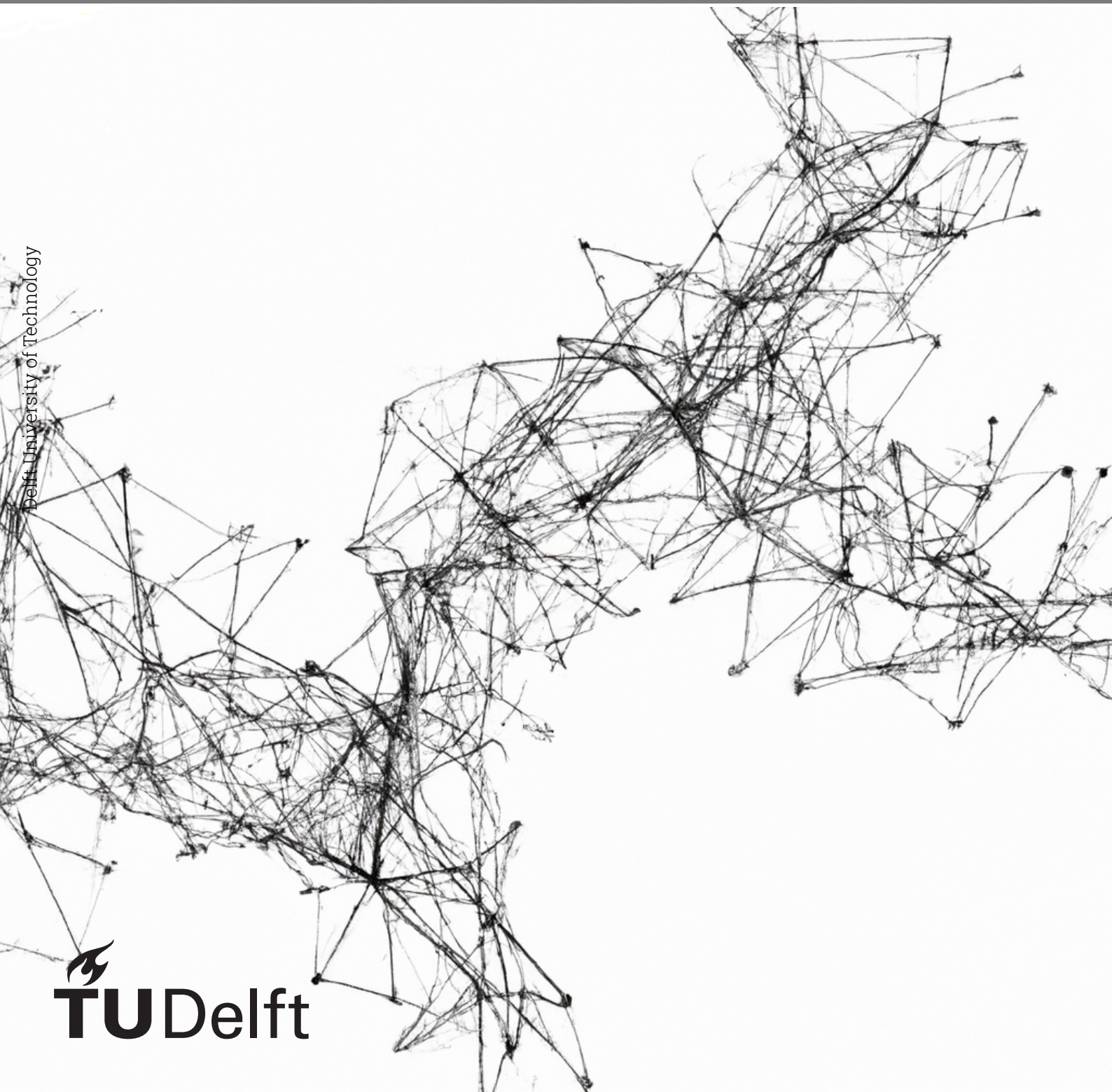
PyTSPL

A Python Library for Topological Signal Processing and Learning

Irtaza Hashmi



Delft University of Technology

TU Delft

PyTSPL

A Python Library for Topological Signal Processing and Learning

by

Irtaza Hashmi

to obtain the degree of Master in Computer Science with Specialization in Data Science at Technische Universiteit Delft, to be publicly defended on Monday, August 12th 2024 at 15:00

Thesis Supervisor:	Dr. E. Isufi, TU Delft
Daily Supervisor:	Maosheng Yang, TU Delft
Committee Members:	Dr. E. Isufi, TU Delft (chair)
	Dr. N.Tömen, TU Delft
	Maosheng Yang, TU Delft
Faculty:	Multimedia Computing Group, Faculty of EEMCS, TU Delft

TU Delft

Preface

While this master's thesis marks the end of my formal education at TU Delft, I look forward to continuously learning in the field of computer science, specifically data science and artificial intelligence. Reflecting on the past year, I started this thesis with a blank slate with no previous experience in graph learning, a domain that always fascinated me. It was one of my goals to learn and apply graph-based learning to real-world applications and thankfully, I was able to achieve that during this master's thesis. Over the past nine months, I have transformed into a knowledgeable individual in the domain of graph learning and software engineering, and such an endeavour would not have been possible without the support and guidance I received from numerous people.

First and foremost, I would like to extend my gratitude to Maosheng Yang for the countless hours he invested in this thesis. He was patient with me as he guided me through the mathematics behind the concepts. As I was new to the domain, it required more effort than usual and I thank him for that. Through numerous meetings, discussions and GitHub issues, we kept refining the requirements of the thesis to make sure we built a valuable product together. Moreover, I sincerely thank Dr. Elvin Isufi for giving me time and invaluable feedback on this thesis. He supported me through my work and was always there to help. Furthermore, I would like to thank Dr. Nergis Tömen for being part of my thesis committee.

Next, I would like to express my heartfelt gratitude to my friends and family for their unwavering support throughout my journey. To my friends, Radek and Niyousha, thank you for your support during my time in Delft. The countless study sessions, sports classes, barbecues, and discussions we shared were invaluable and kept us motivated. Lastly, I extend my sincere gratitude to my parents, Dr. Murtaza Hashmi and Faiqa Hashmi, for their constant support and encouragement.

Irtaza Hashmi
Delft, August 2024

Summary

Graph-based machine learning has seen significant growth during the past years with great advancements and applicability. These approaches mostly focus on pairwise interactions, neglecting the patterns of higher-order interactions which are common to complex systems. In real-world applications, we often encounter these signals that naturally associate with nodes, edges or sets of nodes (e.g. triangles). While the node signals have been well-studied by graph-based methods, the other signals have been researched in the recently emerging field of topological signal processing and topological machine learning. In this thesis, we are particularly interested in edge flow, which models the signals over the edges of a network by signal processing and learning tasks, centring on simplicial complexes. Examples of such networks can be traffic flows in a road network or water flows in a hydrological network. Recent literature in topological signal processing shows simplicial complex as a powerful and principled higher-order network model for edge flows.

In this thesis, we introduce `PyTSPL`, a Python library that provides reliable and user-friendly building blocks for interacting with simplicial complexes. The library aims to provide a unified platform to read network data in different formats, preprocess them and store them in a data structure such that their properties can be easily retrieved. Users can visualize the simplicial complex simply and effectively, enhancing the interpretability of complex structures and data flows. Additionally, the library provides functionality to analyze the simplicial complexes using various advanced signal processing techniques.

The motivation behind developing this master's thesis is to provide practical bridges to analysing and processing network data based on recent research methods with a unified Python library. While various tools exist for specific aspects of network analysis, there is a lack of unified platforms that integrate reading, processing, visualization, and advanced analysis of network data through topological frameworks, specifically for simplicial complexes. This library is a comprehensive solution encapsulating the entire workflow in a single environment.

The source code of `PyTSPL` is available under MIT licence in this **repository** with its **documentation**. The library is also available on **PyPI** for installation.

Contents

Preface i

Summary ii

1 Introduction 1

2 Background 4
2.1 Graph Signal Processing . . . 4
2.1.1 Graph Theory . . . 4
2.1.2 Graph Signals . . . 5
2.1.3 Graph Filters . . . 5
2.1.4 Graph Fourier Transform . . . 6
2.2 Higher-order Interactions with Simplicial Complexes . . . 7
2.2.1 Introduction to Simplicial Complexes . . . 7
2.2.2 Signal Processing on Simplicial Complexes . . . 7
2.2.3 Simplicial Convolutional Filters . . . 11
2.2.4 Simplicial Trend Filtering . . . 14
2.2.5 Hodge-compositional Gaussian Process . . . 16

3 Review of Existing Libraries 18
3.1 Libraries for Graphs . . . 18
3.2 Libraries for Higher-Order Networks . . . 19
3.3 Comparison and Interaction of Libraries . . . 20

4 Design Goals and Description of PyTSPL 21
4.1 Design Goals of PyTSPL . . . 21
4.1.1 Functional Requirements . . . 21
4.1.2 Non-functional Requirements . . . 22
4.2 Key Modules and Submodules . . . 22
4.2.1 `io` . . . 22
4.2.2 `simplicial_complex` . . . 23
4.2.3 `plot` . . . 23
4.2.4 `decomposition` . . . 25
4.2.5 `filters` . . . 26
4.2.6 `hodge_gp` . . . 28
4.3 Interaction Between the Modules . . . 29
4.4 Datasets . . . 29

5 Pedagogical Tutorials 31
5.1 Loading a SC . . . 31
5.2 Building a SC . . . 34
5.3 Plotting . . . 35
5.4 Algebraic Properties . . . 37
5.5 Signal Processing . . . 37
5.5.1 Hodge Decomposition . . . 37
5.5.2 Eigendecomposition . . . 38
5.5.3 Simplicial Fourier Transform . . . 39
5.6 Filters . . . 39
5.6.1 Simplicial Shifting . . . 39
5.6.2 Simplicial Convolutional Filters . . . 41
5.6.3 Simplicial Trend Filter . . . 47

5.7 Hodge-compositional Edge Gaussian Processes 49

6 Configuration Management **53**
6.1 Version Control 53
6.2 Dependency Management 53
6.3 Quality Assurance 54
6.3.1 Testing 54
6.3.2 Static Code Analysis 55
6.4 CI/CD Pipelines 55
6.5 Documentation 56
6.6 Contributing and Changelog 57

7 Library Usability Evaluation **59**
7.1 Installation 59
7.2 Quick Start 61
7.3 Tutorials and API Reference 63
7.4 Overall Feedback 64

8 Conclusion **65**
8.1 Thesis Summary 65
8.2 Future Work 65

References **67**

1

Introduction

We are immersed in a world where data has become an invaluable asset for companies and organizations. This wealth of information drives decision-making, fuels innovation, and provides a competitive edge. It has taken a significant role in the digital age, impacting our lives and altering how we live. Each of us generates massive amounts of data across various levels, from public data like social media activities to private data such as health records and banking information. The complexity of such data with its interactions means the data resides on irregular and complex structures [1]. To effectively analyze this data, standard tools are insufficient; we need to model the data in a more complex manner.

Graphs are a powerful model for representing systems consisting of entities that are interacting with each other. These entities can be represented as nodes, with their interactions encoded as edges in a graph. This representation allows for the analysis and visualization of complex relationships within a system, enabling the identification of key nodes, the discovery of patterns, and the understanding of overall structure and dynamics. Graphs are widely used in various fields to model complex structures and relationships, such as public transportation networks [2], social sciences [3] and many more types of systems. For example, in a social network, the users can be modelled as nodes while their friend connections can be modelled as edges as shown in Figure 1.1 below. We can add properties to the nodes (users) and model those signals over a graph. For example, we can analyze the influence and information spread in a social network by examining the activity level of each user as a graph signal. By doing so, we can identify key influencers and highlight nodes (people) that play a critical role in spreading the information. Many of these applications can be addressed in terms of graph signal processing (GSP), which provides a framework for processing signal data on graphs.

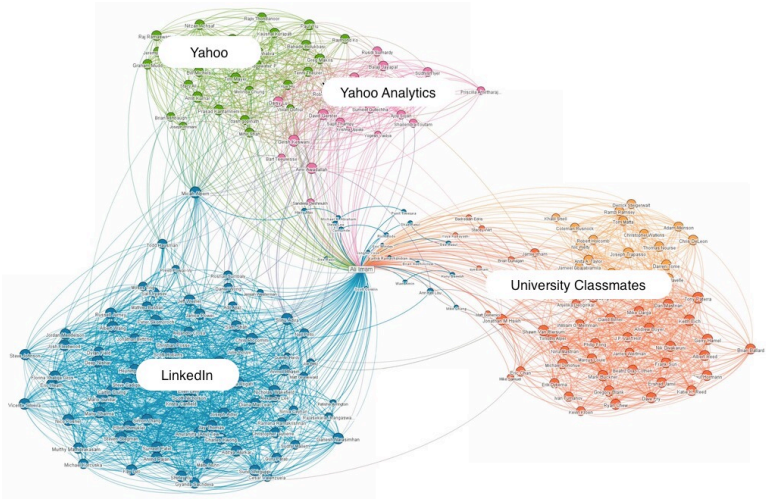


Figure 1.1: A social network of a user from LinkedIn [4]. The user is located in the centre while the edges represent the user's connections. The colour-coded clusters represent the different groups from the user's professional career such as university classmates or colleagues from previous workplaces.

However, we often encounter situations where the signals naturally associate with edges or sets of nodes (triangles) in real-world applications. Such applications include the data flows in social networks [5], water flows in the hydrological network or traffic flows in a road network [6]. Typically, these signals are modelled as edge flows over a network, which have been used in analyzing delivery networks [7], games [8], etc. Regular graphs fail to capture the interactions between more than two nodes even though such multi-way interactions are common in real-world applications. For instance, people interacting in a small social group [9] or a co-authorship network where a paper has more than two authors [10]. Graph-based representations compress the higher-order interactions into pairwise interactions and therefore lose high-dimensional information. In these applications, it can be fruitful to utilize the relationships between the node-relationships, that are the edges or higher-order edges, themselves. To represent such polyadic interactions, several modelling frameworks have been proposed in the literature including simplicial complexes (SCs) [11], hypergraphs [12] and others [13]. Recently, these frameworks have attracted much attention to analyze polyadic relations and study edge flows. In this master's thesis, we focus on the development of a library focusing on the functionalities of using signal processing on SC.

The field of SCs is rapidly evolving and there is no unified way to process these structures, halting the reproducibility and research advances. The goal of this library is to enable researchers to quickly interact or be hands-on with the tools, propose new functionalities and datasets etc. For instance, `NetworkX` is a powerful, open-source Python library that enables direct interaction with graphs. It offers a unified package that includes functionalities for creating, manipulating, analyzing, and visualizing complex networks. This unified approach provides researchers and engineers with an invaluable tool to work with network data seamlessly, without the need to rely on disparate code pieces from different sources.

Before developing the library, extensive research was conducted to identify existing libraries in the domain of higher-order interactions. The functionalities they offer were thoroughly evaluated, and gaps between them were identified. Furthermore, the components of the library were scattered across different programming languages. Research was conducted to determine how these components could be cohesively integrated to ensure good usability, maintainability, and extendability of the library. During the implementation of the library components, various libraries and functionalities were experimented with to identify the most efficient solutions. For example, multiplying large sparse matrices using `numpy` arrays were not very efficient and required research before implementing it using `scipy`'s sparse matrices.

The goal of this library is to provide a unified platform for researchers and engineers to interact with higher-order networks. It includes features for creating, building, manipulating, analyzing, and visual-

izing these networks. Additionally, the library offers advanced signal processing techniques such as simplicial convolutional filters, simplicial trend filtering, and Hodge-compositional edge Gaussian processes. In future work, we aim to support additional higher-order structures, such as hypergraphs and cell complexes, as well as topological signal processing and learning functionalities.

The thesis is structured as follows: Chapter 2 provides the fundamental background on the subject, including a brief review of prior work directly related to signal processing on graphs and SCs. This chapter also explores the concepts on which the library is built. Chapter 3 surveys the functionalities of existing libraries. Chapter 4 delves into the design goals and key modules of the proposed library. In Chapter 5, we provide pedagogical tutorials on how to use the library's functionalities. Chapter 6 examines the configuration management of the library and its development using the best software development practices. Chapter 7 evaluates the usability of the library and assesses the user experience during the installation process and the quick-start phase. Finally, Chapter 8 presents a summary of the thesis and discusses potential future work to further improve and develop the library.

2

Background

This chapter establishes the foundational knowledge required for the reader. As the subsequent chapters assume a certain level of familiarity with the topic, this chapter will systematically develop the necessary background information.

2.1. Graph Signal Processing

Before delving into signal processing on higher-order networks, we will review the fundamental principles of signal processing on graphs. This section will concentrate on essential components of graph theory, including graph signals, filters, and the Fourier transform. This foundational understanding will provide the necessary context and guide our subsequent discussion on higher-order signal processing.

2.1.1. Graph Theory

A graph G is a triplet $G = (V, E, W)$ where $V = \{v_1, \ldots, v_N\}$ is a finite set of nodes or vertices with cardinality N, $E \subseteq V \times V$ is a set of edges defined as ordered pairs (i, j) and $W : E \to \mathbb{R}$ is a mapping from the set of edges to scalar values, w_{ij}. The weights w_{ij} represent the relationship between nodes i and j. The adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a square matrix used to represent a graph and its elements indicate whether pairs of nodes are adjacent in the graph. The matrix $\mathbf{A}$ is defined as

$$A_{ji} = \begin{cases} w_{ij}, & \text{if } (i,j) \in E; \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

Graphs can be either weighted or unweighted. In unweighted graphs, the adjacency matrix element $\mathbf{A}_{ij}$ is 1 if nodes i and j are connected, and 0 otherwise. If the adjacency matrix $\mathbf{A}$ is symmetric, the graph is considered undirected; otherwise, it is directed.

The degree matrix $\mathbf{D} \in \mathbb{R}^{N \times N}$ is a diagonal matrix where element D_{ii} represents the degree of node i, $deg(i)$. The degree of a node i, $deg(i)$, is the sum of the weights of the edges incident to the node and is defined as

$$deg(i) = \sum_{j \in N(i)} w_{ij} \tag{2.2}$$

where $N(i)$ is the neighbourhood nodes for node i.

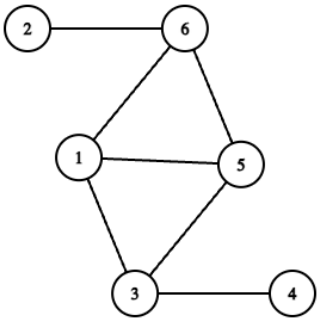
Given an undirected graph G with adjacency matrix $\mathbf{A}$ and a degree matrix $\mathbf{D}$, the Laplacian matrix $\mathbf{L} \in \mathbb{R}^{N \times N}$ is defined as

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{2.3}$$

Alternatively, we can define the Laplacian matrix $\mathbf{L}$ using the graph's incidence matrix $\mathbf{B} \in \mathbb{R}^{N \times E}$ as

$$\mathbf{L} = \mathbf{B}\mathbf{B}^{\top} \tag{2.4}$$

Given an arbitrary graph $G = (V, E, W)$, a graph-shift operator $\mathbf{S} \in \mathbb{R}^{N \times N}$ is a matrix that takes nonzero values on the edges of the graph G or its diagonal. Formally, S_{ij} = 0 for all $i \neq j$ and $(i, j) \notin E$, where E is the the set of edges of the graph. This focuses on the connections between nodes rather than arbitrary pairs of nodes. An example of a graph with its corresponding graph-shift operator is shown in Figure 2.1 below.



$$\mathbf{S} = \begin{pmatrix} S_{11} & 0 & S_{13} & 0 & S_{15} & S_{16} \\ 0 & S_{22} & 0 & 0 & 0 & S_{26} \\ S_{31} & 0 & S_{33} & S_{34} & S_{35} & 0 \\ 0 & 0 & S_{43} & S_{44} & 0 & 0 \\ S_{51} & 0 & S_{53} & 0 & S_{55} & S_{56} \\ S_{61} & S_{62} & 0 & 0 & S_{65} & S_{66} \end{pmatrix}$$

Figure 2.1: An example of a six node graph with its corresponding graph-shift operator $\mathbf{S}$. The graph-shift operator is nonzero only where there is an edge or in the diagonal.

There are different choices for the graph-shift operator including the adjacency matrix $\mathbf{A}$ [14], the Laplacians [15], or variations of these matrices each presenting different trade-offs [16]. The Laplacian matrix $\mathbf{L}$ has specific spectral properties, in particular, $\mathbf{L}$ is a semi-definite matrix for an undirected graph. This means all its eigenvalues are non-negative and have an eigenvalue of 0. Since $\mathbf{L}$ is a real and symmetric matrix for an undirected graph, it has a complete set of orthonormal eigenvectors with associated eigenvalues. Due to these properties, using $\mathbf{L}$ as a graph-shift operator avoids a certain number of numerical difficulties [1]. However, one should consider the different graph-shift operators and choose the one specific to their application with the best trade-off [17].

2.1.2. Graph Signals

A graph signal is a map from the set of nodes V to the set of real numbers $\mathbb{R}$, defined as

$$s : V \to \mathbb{R} \tag{2.5}$$

Graph signals can be represented by the component vector

$$\mathbf{s} = \begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_{N-1} \end{bmatrix} \in \mathbb{R}^N \tag{2.6}$$

Note that each value s_n is indexed by node v_n for a given graph G.

2.1.3. Graph Filters

A general graph filter is represented by a matrix $\mathbf{H}$, which takes in a signal $\mathbf{s_{in}}$, processes it via matrix-vector multiplication and produces another graph signal $\mathbf{s_{out}}$ defined as

$$\mathbf{s_{out}} = \mathbf{H}\mathbf{s_{in}} \tag{2.7}$$

A basic filter defined on a graph $G = (V, \mathbf{S})$, called the *graph shift*, is a local operation that replaces each signal value s_n at node v_n with the weighted sum of graph signal at the neighbours of node v_n

$$\tilde{s_n} = \sum_{i \in N_n} S_{ni} s_i \tag{2.8}$$

where the weights S_{ni} are from the graph-shift operator $\mathbf{S}$ and N_n is a set containing the neighbourhood nodes of v_n. Therefore, the output of the graph shift is given as

$$\tilde{\mathbf{s}} = \begin{bmatrix} \tilde{s_0} \\ \tilde{s_1} \\ \vdots \\ \tilde{s_{N-1}} \end{bmatrix} = \mathbf{S}\mathbf{s} \tag{2.9}$$

The graph shift operation is the most elementary filtering operation in GSP. Linear, shift-invariant graph filters, in particular, play a crucial role in GSP. Graph filters are considered linear if, for a given linear combination of inputs, they produce the same linear combination of outputs. Graph filters are shift-invariant if the processing of a signal by multiple graph filters does not depend on the order of processing. In other words, shift-invariant filters commute with each other.

$$\mathbf{S}(\mathbf{H}\mathbf{s}) = \mathbf{H}(\mathbf{S}\mathbf{s}) \tag{2.10}$$

All linear, shift-invariant graph filters $\mathbf{H}$ are polynomials in the graph-shift operator $\mathbf{S}$ of the form [14]

$$\mathbf{H} = h(\mathbf{S}) = h_0\mathbf{I} + h_1\mathbf{S} + \cdots + h_L\mathbf{S}^L \tag{2.11}$$

Hence, the output of the filter is signal $\tilde{\mathbf{s}}$ is

$$\tilde{\mathbf{s}} = \mathbf{H}(\mathbf{s}) = h(\mathbf{S})\mathbf{s} \tag{2.12}$$

2.1.4. Graph Fourier Transform

The graph Fourier transform (GFT) allows us to analyze graph signals in the frequency domain. Given the eigenvalue decomposition of the shift operator as

$$\mathbf{S} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top \tag{2.13}$$

where $\mathbf{U}$ represents the eigenvectors of $\mathbf{S}$, and $\mathbf{\Lambda}$ is a diagonal matrix containing the associated eigenvalues of $\mathbf{S}$. These eigenvalues are known as the graph frequencies and form the graph spectrum. The eigenvectors corresponding to the mth frequency λ_m are referred to as the frequency components. Given the eigenvalue decomposition and a filtering weight function as $h : \mathbb{R} \to \mathbb{R}$, any shift-invariant filter can be expressed as [1]

$$\mathbf{H} = \sum_{k=1}^{N} h(\lambda_k) u_k u_k^\top = \mathbf{U} h(\mathbf{\Lambda}) \mathbf{U}^\top \tag{2.14}$$

where $h(\mathbf{\Lambda})$ is shorthand for diag$(h(\lambda_1), \ldots h(\lambda_N))$, $\mathbf{U}$ are the eigenvectors of the shift operator that define the GFT and $h(\mathbf{\Lambda})$ is the frequency response of the filter $\mathbf{H}$. Thus, the GFT of a graph signal $\mathbf{s}$ is defined as

$$\tilde{\mathbf{s}} = \mathbf{U}^\top \mathbf{s} \tag{2.15}$$

where s_n are the values of the signal's graph Fourier transform characterizing the frequency content of signal $\mathbf{s}$. The inverse of the GFT reconstructs the original signal using the frequency contents, each weighted by the corresponding Fourier transform coefficients defined as

$$\mathbf{s} = \mathbf{U}\tilde{\mathbf{s}}. \tag{2.16}$$

2.2. Higher-order Interactions with Simplicial Complexes

In this section, we revisit the definitions of a simplicial complex, signal processing on simplicial complexes, simplicial convolutional filters, and the applications of simplicial signal processing. Furthermore, we will define Simplicial Trend Filtering and Hodge- compositional Gaussian Process, along with their respective applications.

2.2.1. Introduction to Simplicial Complexes

Given a finite set of vertices V, a k-simplex S^k is a subset of V with $k+1$ nodes. The face of S^k is the subset of k-simplex S^k with cardinality k. A coface of S^k is a simplex S^{k+1} that includes it. A simplicial complex $\mathcal{X}$ is a finite collection of simplices satisfying the inclusion property. The inclusion property states that for any $S^k \in \mathcal{X}$, all its faces $S^{k-1} \subset S^k$ are also part of the simplicial complex [18, 19]. The order of the SC $\mathcal{X}$ is the largest order of its simplices. Then, a node is a 0-dimensional complex, an edge is a 1-dimensional complex and a triangle (shaded) is a 2-dimensional complex and so on. Note that an "unshaded" triangle formed by three nodes and three pairwise relations between them is not a 2-simplex. Henceforth, we will address 2- simplices as triangles for simplicity. Figure 2.2 shows an SC with an order 2 including nodes, edges and triangles (shaded). The edge {0, 1} has nodes {0} and {1} as its faces and triangle {0, 1, 2} as its coface.

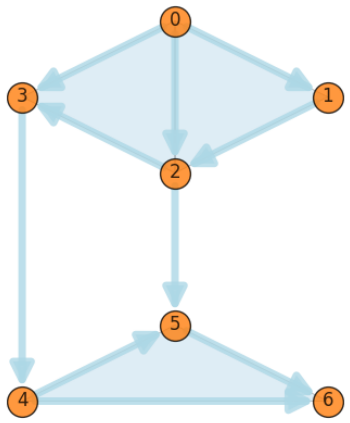


Figure 2.2: A 2-dimensional simplicial complex, $\mathcal{X}$, containing seven nodes, ten edges and three shaded triangles (2-simplex).

2.2.2. Signal Processing on Simplicial Complexes

For computational purposes, we fix the reference orientation for each simplex according to the lexicographic order of its vertices [20]. Based on this reference for each simplex, we define a *k-simplicial signal* as

$$\mathbf{s}^k = \begin{bmatrix} s_1^k \\ s_2^k \\ \vdots \\ s_{N_k}^k \end{bmatrix} \in \mathbb{R}^{N_k} \tag{2.17}$$

where the value s_i^k is the signal of the ith k-simplex S_i^k. By convention, if the signal s_i^k is positive, then the corresponding signal is aligned with the reference orientation. Otherwise, the orientation is in the opposite direction. Figure 2.3 illustrates an arbitrary edge flow on a SC. The colour map on

the right describes the strength of the flow. For future reference, we denote a node signal as $\mathbf{v} = [v_1, v_2, \ldots, v_{N_0}]^\top \in \mathbb{R}^{N_0}$ and edge flow as $\mathbf{f} = [f_1, f_2, \ldots, f_{N_1}]^\top \in \mathbb{R}^{N_1}$.

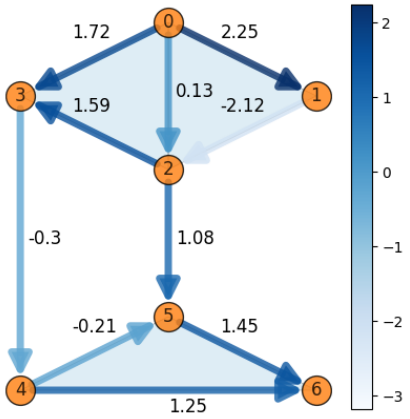


Figure 2.3: An arbitrary edge flow $\mathbf{f}$ on SC $\mathcal{X}$. The negative flow indicates that the actual flow is in the opposite direction to the reference orientation and the magnitude is denoted by the color map.

The relationship between $(k-1)$-simplices and k-simplices can be described via linear maps known as boundary operators. Boundary operators record the higher-order interactions in the networks. Since we considered finite simplicial complexes, the boundary operators are represented by matrices $\mathbf{B}_k$. The rows of the matrix $\mathbf{B}_k$ correspond to $(k-1)$ dimensional simplices and the columns correspond to (k)- dimensional simplices. Specifically, $\mathbf{B}_1$ is the node-to-edge incidence matrix and $\mathbf{B}_2$ is the edge-to-triangle incidence matrix. Incidence matrices, by definition, have a particular property [18, 21]

$$\mathbf{B}_k \mathbf{B}_{k+1} = \mathbf{0}. \tag{2.18}$$

We established that an appropriate shift operator is needed to process graph signals in Section 2.1.1. Similarly, we need to establish an appropriate shift operator to process signals on a SC. Naturally, the Hodge Laplacian serves as an appropriate shift operator, extending the concept of the graph Laplacian we discussed earlier. Based on the incidence matrices defined above, we can describe a simplicial complex $\mathcal{X}$ of order *K* via the Hodge Laplacians defined as

$$\mathbf{L}_k = \mathbf{B}_k^\top \mathbf{B}_k + \mathbf{B}_{k+1} \mathbf{B}_{k+1}^\top, \quad k = 1, 2, \ldots K-1 \tag{2.19}$$

where the graph Laplacian $\mathbf{L}_0$ is defined as

$$\mathbf{L}_0 = \mathbf{B}_1 \mathbf{B}_1^\top \quad with \quad \mathbf{B}_0 = \mathbf{0} \tag{2.20}$$

The kth-Hodge Laplacian $\mathbf{L}_k$ can be decomposed into the *lower Laplacian* and the *upper Laplacian* defined as

$$\mathbf{L}_{k,l} \triangleq \mathbf{B}_k^\top \mathbf{B}_k \tag{2.21}$$

$$\mathbf{L}_{k,u} \triangleq \mathbf{B}_{k+1} \mathbf{B}_k^\top \tag{2.22}$$

respectively. The lower Laplacian captures how simplices are connected through their shared faces, while the upper Laplacian captures their connections through shared cofaces. Specifically, $\mathbf{L}_{1,l}$ represents edge adjacencies based on their incident nodes, while $\mathbf{L}_{1,u}$ represents adjacencies based on the triangles they share. For example, two edges would be lower adjacent if they share a common node and upper adjacent if they are faces of a common triangle. Like the graph Laplacian, the Hodge Laplacian

is a positive and semi-definite matrix. This ensures we can interpret its eigenvalues as non-negative frequencies.

Hodge Decomposition

The Hodge decomposition states that the space of k-simplex signals can be decomposed into three orthogonal subspaces that can be written as

$$\mathbb{R}^{N_k} = im(\mathbf{B}_{k+1}) \oplus im(\mathbf{B}_k^\top) \oplus ker(\mathbf{L}_k) \tag{2.23}$$

where im is the image space and ker is the kernel space of the respective matrices. The subspace $im(\mathbf{B}_{k+1})$ is known as the gradient subspace, $im(\mathbf{B}_k^\top)$ as the curl subspace and $ker(\mathbf{L}_k)$ as the harmonic subspace [18, 20]. Therefore, we can decompose any edge flow $\mathbf{f} \in \mathbb{R}^{N_1}$ into the orthogonal components

$$\mathbf{f} = \mathbf{f}_G + \mathbf{f}_C + \mathbf{f}_H, \tag{2.24}$$

where $\mathbf{f}_G$ is the *gradient component* $\mathbf{f}_G \in im(\mathbf{B}_1^\top)$, $\mathbf{f}_C$ is the *curl component* $\mathbf{f}_C \in im(\mathbf{B}_2)$, and $\mathbf{f}_H$ is the *harmonic component* $\mathbf{f}_H \in ker(\mathbf{L}_1)$. By decomposing the signal into three different components, we can extract the different properties of the flow. For instance, we can study the effect of an external source or sink by extracting the gradient component of the edge flow [21]. However, the incidence matrices $\mathbf{B}_1$ and $\mathbf{B}_2$ can be interpreted as follows [20, 21].

Divergence. Divergence is defined as the netflow passing through the ith vertex. The incidence matrix $\mathbf{B}_1$ acts as a divergence operator and the divergence of an edge flow $\mathbf{f}$ can be calculated as

$$div(\mathbf{f}) = \mathbf{B}_1\mathbf{f} \tag{2.25}$$

where the $i$the entry represents the netflow of the ith vertex. A flow is known to be *divergence-free* if the edge flow $\mathbf{f}$ has zero divergence at each vertex i.e. $\mathbf{B}_1\mathbf{f} = \mathbf{0} \iff \mathbf{f} \in ker(\mathbf{B}_1)$.

Curl. Curl is defined as the netflow circulating along a ith triangle. The curl of an edge flow $\mathbf{f}$ can be calculated as

$$curl(\mathbf{f}) = \mathbf{B}_2^\top\mathbf{f} \tag{2.26}$$

where the $i$the entry represents the netflow of the ith triangle. A flow is known to be *curl-free* if the edge flow $\mathbf{f} \in ker(\mathbf{B}_2^\top)$ has zero curl on each triangle.

Gradient flow. The adjoint operator $\mathbf{B}_1^\top$ is called the *gradient operator*. It induces a *gradient flow* $\mathbf{f_G}$ by taking the difference between the node signals along the oriented edges calculated as follows

$$\mathbf{f_G} = \mathbf{B}_1^\top\mathbf{v} \in im(\mathbf{B}_1^\top) \tag{2.27}$$

The subspace $im(\mathbf{B}_1^\top)$ is defined as the *gradient space*, as any gradient flow can be induced from a node signal via the gradient operator.

Curl flow. The *curl flow* $\mathbf{f_C}$ corresponds to a flow locally circling along the edges of a triangle (2-simplex) and is calculated as follows

$$\mathbf{f_C} = \mathbf{B}_2\mathbf{t} \tag{2.28}$$

where $\mathbf{t} \in \mathbb{R}^{N_2}$ is a triangle signal. The subspace $im(\mathbf{B}_2)$ is defined as the *curl space*, as any curl flow can be induced from a triangle signal.

Harmonic flow. The space $ker(\mathbf{L}_1)$ is defined as the *harmonic space* and any flow $\mathbf{f_H} \in ker(\mathbf{L}_1)$ that satisfies $\mathbf{L}_1\mathbf{f_H} = \mathbf{0}$ is *harmonic*. Harmonic flow is both divergence- and curl-free.

Eigendecomposition

As a generalization of the approach in equation (2.13), we can represent the signals of different order over bases built with the eigenvectors of the higher-order Laplacian matrices. Thus, the eigendecomposition of the Hodge Laplacians is defined as

$$\mathbf{L}_k = \mathbf{U}_k \mathbf{\Lambda}_k \mathbf{U}_k^\top \tag{2.29}$$

where $\mathbf{U}_k$ is an orthonormal matrix that collects the eigenvectors and $\mathbf{\Lambda}_k$ is a diagonal matrix with the associated eigenvalues. The Hodge decomposition establishes a correspondence between $\mathbf{U}_1$ and the three orthogonal subspaces, that is, the eigenvectors of $\mathbf{U}_1$ fully span the three orthogonal subspaces, namely gradient, curl and harmonic, given by the Hodge decomposition [22]. Given the 1-Hodge Laplacian of an SC

$$\mathbf{L}_1 = \mathbf{L}_{1,l} + \mathbf{L}_{1,u} \tag{2.30}$$

the following holds:

1. *Gradient eigenvectors* $\mathbf{U_G} = [\mathbf{u_{G,1}} \dots \mathbf{u}_{\mathbf{G},N_\mathbf{G}}] \in \mathbb{R}^{N_1 \times N_\mathbf{G}}$ of $\mathbf{L}_{1,l}$, associated with their corresponding nonzero eigenvalues, span the gradient space $im(\mathbf{B}_1^\top)$ with dimension $N_\mathbf{G}$.
2. *Curl eigenvectors* $\mathbf{U_C} = [\mathbf{u_{C,1}} \dots \mathbf{u}_{\mathbf{C},N_\mathbf{C}}] \in \mathbb{R}^{N_1 \times N_\mathbf{C}}$ of $\mathbf{L}_{1,u}$, associated with their corresponding nonzero eigenvalues span the curl space $im(\mathbf{B}_2)$ with dimension $N_\mathbf{C}$.
3. *Harmonic eigenvectors* $\mathbf{U_H} = [\mathbf{u_{H,1}} \dots \mathbf{u}_{\mathbf{H},N_\mathbf{H}}] \in \mathbb{R}^{N_1 \times N_\mathbf{H}}$ of $\mathbf{L}_1$, associated with their corresponding zero eigenvalues span the harmonic space $ker(\mathbf{L}_1)$ with dimension $N_\mathbf{H}$.

Furthermore, the matrices $[\mathbf{U_H}\,\mathbf{U_C}]$ and $[\mathbf{U_H}\,\mathbf{U_G}]$ provide the eigenvectors of the lower Laplacian $\mathbf{L}_{1,l}$ and the upper Laplacian $\mathbf{L}_{1,u}$ associated with their corresponding zero eigenvalues, respectively.

Simplicial Fourier Transform

Given a k-simplicial signal $\mathbf{s}^k$, the simplicial Fourier Transform (SFT) is defined as

$$\tilde{\mathbf{s}}^k \triangleq \mathbf{U}_k^\top \mathbf{s}^k \tag{2.31}$$

which is a projection onto the eigenvectors $\mathbf{U}_k$ where each entry $\mathbf{s}_i^k$ represents the weight eigenvector $\mathbf{u}_i^k$ has on $\mathbf{s}^k$. The inverse SFT is given by

$$\mathbf{s}^k \triangleq \mathbf{U}_k \tilde{\mathbf{s}}^k \tag{2.32}$$

Similarly to GFT, the eigenvalues of $\mathbf{L}_k$ represent the notion of simplicial frequencies, but in a more meaningful way. The eigenvalues of $\mathbf{L}_k$ measure three types of simplicial frequencies [22].

1. *Gradient frequency:* The magnitude of an eigenvalue λ_G measures the amount of total divergence in an SC. The divergence is a measure of the net flow of a signal out of a node. The gradient eigenvectors associated with large eigenvalues have a large total divergence.
2. *Curl frequency:* The magnitude of an eigenvalue λ_C measures the amount of total curl in an SC i.e. rotation variation. The rotation variation is the measure of the extent of circular or rotational flow in the network. The curl eigenvectors associated with large eigenvalues have a large total curl.
3. *Harmonic frequency:* The harmonic eigenvectors $\mathbf{U}_H$ are both divergence- and curl-free. A harmonic flow is defined as the SFT of an edge flow that has nonzero components only at the harmonic frequencies, which correspond to zero eigenvalues.

Simplicial Embeddings

Given the three component eigenvectors, we define the three embeddings of an edge flow $\mathbf{f} \in \mathbb{R}^{N_1}$ as follows

$$\begin{cases} \tilde{\mathbf{f}}_{\mathbf{H}} = \mathbf{U}_{\mathbf{H}}{}^{\top}\mathbf{f} = \mathbf{U}_{\mathbf{H}}{}^{\top}\mathbf{f} \in \mathbb{R}^{N_{\mathbf{H}}}, & \textit{harmonic embedding} \\ \tilde{\mathbf{f}}_{\mathbf{G}} = \mathbf{U}_{\mathbf{G}}{}^{\top}\mathbf{f} = \mathbf{U}_{\mathbf{G}}{}^{\top}\mathbf{f} \in \mathbb{R}^{N_{\mathbf{G}}}, & \textit{gradient embedding} \\ \tilde{\mathbf{f}}_{\mathbf{C}} = \mathbf{U}_{\mathbf{C}}{}^{\top}\mathbf{f} = \mathbf{U}_{\mathbf{C}}{}^{\top}\mathbf{f} \in \mathbb{R}^{N_{\mathbf{C}}}, & \textit{curl embedding} \end{cases} \tag{2.33}$$

These embeddings are the result of the orthogonality of there three components given by the Hodge decomposition. Using these simplicial embeddings, we can rewrite the SFT of $\mathbf{f}$ as

$$\tilde{\mathbf{f}} = \left[\tilde{\mathbf{f}}_{\mathbf{H}}{}^{\top}, \tilde{\mathbf{f}}_{\mathbf{G}}{}^{\top}, \tilde{\mathbf{f}}_{\mathbf{C}}{}^{\top}\right]^{\top} \tag{2.34}$$

where $\tilde{\mathbf{f}}_{\mathbf{H}}{}^{\top}$ is the *harmonic embedding*, $\tilde{\mathbf{f}}_{\mathbf{G}}{}^{\top}$ is the *gradient embedding* and $\tilde{\mathbf{f}}_{\mathbf{C}}{}^{\top}$ is the *curl embedding*. Each entry of an embedding represents the weight the flow has on the corresponding eigenvector. This offers a compressed representation of the edge flow and allows us to cluster them based on their types.

2.2.3. Simplicial Convolutional Filters

This section describes the theoretical foundations and definitions involved in creating a simplicial convolutional filter based on the Hodge Laplacian, intended for processing simplicial signals.

The simplicial convolutional filters are based on Hodge Laplacian and rely on the basic building block of simplicial shifting. We define the simplicial convolutional filter, intended to process a k-simplicial signal $\mathbf{s}^k$, using the kth-Hodge Laplacian $\mathbf{L}_k$. The filter is defined as

$$\mathbf{H}_k = h_0\mathbf{I} + \sum_{l_1=1}^{L_1} \boldsymbol{\alpha}_{l_1}(\mathbf{B}_k^{\top}\mathbf{B}_k)^{l_1} + \sum_{l_2=1}^{L_2} \boldsymbol{\beta}_{l_2}(\mathbf{B}_{k+1}^{\top}\mathbf{B}_{k+1})^{l_2}, \tag{2.35}$$

Here, $\mathbf{H}_k$ represents a matrix polynomial of the lower and upper Hodge Laplacian. The polynomial coefficients are given by h_0, with vectors $\boldsymbol{\alpha}$ = $[\boldsymbol{\alpha}_1 \dots \boldsymbol{\alpha}_{L_1}]^{\top}$, and $\boldsymbol{\beta}$ = $[\boldsymbol{\beta}_1 \dots \boldsymbol{\beta}_{L_2}]^{\top}$. The symbols L_1 and L_2 represent the filter orders. Specifically, when $k = 0$, this leads to the graph convolutional filter $\mathbf{H}_0 := \mathbf{H}(\mathbf{L}_0)$ which is constructed using the graph Laplacian [22].

Assigning two different sets of coefficients to the lower and upper Laplacian parts in $\mathbf{H}_k$ allows us to treat the lower and upper adjacencies differently. This leads to more flexibility and control over the frequency response. On the other hand, if the coefficients are the same for the lower and upper Laplacian part, that is $L_1 = L_2 = L$ and $\boldsymbol{\alpha} = \boldsymbol{\beta}$, the filter becomes $\mathbf{H}_k = \sum_{l=0}^{L} h_l \mathbf{L}_k^l$. In such a case, the filter cannot differentiate between the two types of adjacencies and loses some expressive power.

Simplicial shifting. Applying $\mathbf{H}_k$ to a k-simplicial signal $\mathbf{s}^k$ yields an output $\mathbf{H}_k\mathbf{s}^k$ which is a shift-and-sum operation. The filter $\mathbf{H}_k$ shifts the signal L_1 times over the lower neighbourhoods and L_2 times over the upper neighbourhoods. Next, the shifted results are summed together according to the corresponding coefficients. Similarly, consider an edge filter $\mathbf{H}_1$ applied to an edge flow $\mathbf{f}$ with the output

$$\mathbf{f}_o = \mathbf{H}_1\mathbf{f} = h_0\mathbf{f} + \sum_{l_1=1}^{L_1} \boldsymbol{\alpha}_{l_1}\mathbf{L}_{1,l}^{l_1}\mathbf{f} + \sum_{l_2=1}^{L_2} \boldsymbol{\beta}_{l_2}\mathbf{L}_{1,u}^{l_2}\mathbf{f}, \tag{2.36}$$

where we apply different powers of the lower and upper Hodge Laplacian to the edge flow. This basic operation is known as *simplicial shifting*. The one-step lower and upper shifting are defined below, respectively

$$\mathbf{f}_l^{(1)} \triangleq \mathbf{L}_{1,l}\mathbf{f}, \tag{2.37}$$

$$\mathbf{f}_u^{(1)} \triangleq \mathbf{L}_{1,u}\mathbf{f}, \tag{2.38}$$

Then, k-step shifting is defined as the weighted linear combination of the lower and upper shifted simplicial signals after k steps

$$\mathbf{f}_o = h_0\mathbf{f}^{(0)} + \sum_{l_1=1}^{L_1} \boldsymbol{\alpha}_{l_1}\mathbf{f}_\ell^{(l_1)} + \sum_{l_2=1}^{L_2} \boldsymbol{\beta}_{l_2}\mathbf{f}_\mathsf{u}^{(l_2)}, \tag{2.39}$$

Filter designs. Given a dataset of input-output edge flow relations $T = \{(\mathbf{f}_1, \mathbf{f}_{o,1}), \ldots, (\mathbf{f}_{|T|}, \mathbf{f}_{o,|T|})\}$, we learn the filter coefficients by aligning the filter's output $\mathbf{H_1 f}$ with the observed output $\mathbf{f}_o$. The filter coefficients here are learned in a data-driven manner. The optimization utilizes a mean squared error (MSE) cost function, enhanced with a regularization term $\gamma r(h_0, \boldsymbol{\alpha}, \boldsymbol{\beta})$, to prevent overfitting. The problem is formulated as

$$\min_{h_0;\boldsymbol{\alpha};\boldsymbol{\beta}} \frac{1}{|T|} \sum_{(\mathbf{f}_i;\mathbf{f}_o,i)\in T} \|\mathbf{H}_1\mathbf{f}_i - \mathbf{f}_{o;i}\|_2^2 + \gamma r(h_0, \boldsymbol{\alpha}, \boldsymbol{\beta}), \tag{2.40}$$

where γ > 0 acts as the regularization coefficient, balancing fit and complexity to improve model generalization. In the next section, we aim to design simplicial filters given a desired frequency response. First, we considered the standard least-squares (LS) filter design and following that, we consider a universal filter design that avoids the eigenvalue computation given a continuous frequency response. In particular, we consider grid-based and Chebyshev polynomial filter designs.

Least-Squares Filter Design

To determine the filter coefficients h_0, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ in a data-driven manner to approximate the desired response using the filter frequency response $H_1(\lambda)$, the problem can be formulated as follows

$$\begin{cases} h_0 \approx g_0, & \text{for } \lambda_i = 0, \\ h_0 + \sum_{l_1=1}^{L_1} \alpha_{l_1}\lambda_i^{l_1} \approx g_G(\lambda_i), & \text{for } \lambda_i \in \mathcal{Q}_G, \\ h_0 + \sum_{l_2=1}^{L_2} \beta_{l_2}\lambda_i^{l_2} \approx g_C(\lambda_i), & \text{for } \lambda_i \mathcal{Q}_C. \end{cases} \tag{2.41}$$

The equation (2.41) can be solved to obtain the filter coefficients by solving the following LS problem

$$\min_{h_0,\boldsymbol{\alpha},\boldsymbol{\beta}} \left\| \left[\begin{array}{c|cc} \mathbf{1} & \multicolumn{2}{c}{\mathbf{0}} \\ \hline \mathbf{\Phi}_\mathrm{G} & \mathbf{0} \\ \mathbf{0} & \mathbf{\Phi}_\mathrm{C} \end{array}\right] \begin{bmatrix} h_0 \\ \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{bmatrix} - \mathbf{g} \right\|_2^2, \tag{2.42}$$

In this context, $\mathbf{1}$ ($\mathbf{0}$) represents a matrix or vector filled with ones (zeros) of a suitable dimension. The matrices $\mathbf{\Phi}_G \in \mathbb{R}^{D_G \times L_1}$ and $\mathbf{\Phi}_C \in \mathbb{R}^{D_C \times L_2}$ are Vandermonde matrices, characterized by their entries $[\mathbf{\Phi}_G]_{ij} = \lambda_{G,i}^j$ for $\mathbf{\Phi}_G$ and $[\mathbf{\Phi}_C]_{ij} = \lambda_{C,i}^j$ for $\mathbf{\Phi}_C$. This is known as the LS problem and can be solved using the direct pseudo-inverse of the system matrix or decoupled way or through the decoupled solution method [23].

Grid-Based Filter Design

The grid-based filter design is a type of universal filter and avoids computing the eigenvalues of $\mathbf{L}_1$. The filter matches the desired frequency response in a continuous interval and determines where the exact frequencies lie. Given harmonic g_0, gradient $g_G(\lambda)$ for $\lambda \in [\lambda_{G,\mathsf{min}}, \lambda_{G,\mathsf{max}}]$, and curl $g_C(\lambda)$ for $\lambda \in [\lambda_{C,\mathsf{min}}, \lambda_{C,\mathsf{max}}]$ frequency responses, we aim to achieve the following conditions such that

$$\begin{cases} h_0 - g_0 \approx 0 \\ \int_{\lambda_{G,\min}}^{\lambda_{G,\max}} \left| h_0 + \sum_{l_1=1}^{L_1} \alpha_{l_1} \lambda^{l_1} - g_G(\lambda) \right|^2 d\lambda \approx 0 \\ \int_{\lambda_{C,\min}}^{\lambda_{C,\max}} \left| h_0 + \sum_{l_2=1}^{L_2} \beta_{l_2} \lambda^{l_2} - g_C(\lambda) \right|^2 d\lambda \approx 0 \end{cases}$$

which is a continuous version of (2.41). This problem can be transferred and solved as an LS problem of the form (2.42) by sampling M_1 and M_2 grid-points uniformly from the interval $[\lambda_{G,\text{min}}, \lambda_{G,\text{max}}]$ and $[\lambda_{C,\text{min}}, \lambda_{C,\text{max}}]$. The largest true eigenvalue can be approximated by using efficient algorithms such as power iteration and the smallest eigenvalue can be set to a value greater than 0 as the lower bound [24, 25].

Chebyshev Polynomial Filter Design

The filters described earlier depend on resolving the LS problem and experience numerical instability due to the Vandermonde matrix. To tackle this issue, we consider a filter design based on Chebyshev polynomials [26, 27]. Let us consider a continuous gradient frequency response $g_G(\lambda), \lambda \in [0, \lambda_{G,\text{max}}]$ and a continuous curl frequency response $g_C(\lambda), \lambda \in [0, \lambda_{C,\text{max}}]$. We require that $g_G(\lambda) = g_C(\lambda) = g_0$, that is, the continuous gradient and curl frequency are defined starting at 0. At this point, they are equal to the harmonic frequency g_0. The strategy is to obtain the gradient and curl frequency responses separately and sum these two polynomials together to obtain the final filter.

In the first step, we approximate the gradient frequency response via a truncated series of shifted Chebyshev polynomials $\mathbf{H}_l := \mathbf{H}_l(\mathbf{L}_{1,l})$. Let $\bar{P}_l(\lambda), \lambda \in [-1, 1]$ be the l-th Chebyshev polynomial of the first kind [28]. To shift the domain to $[0, \lambda_{G,\text{max}}]$ as mentioned previously, we perform a transformation $P_l(\lambda) := \bar{P}_l\left(\frac{\lambda-\omega}{\omega}\right)$ with $\omega := \frac{\lambda_{G,\text{max}}}{2}$. Next, we approximate the operator $g_G(\mathbf{L}_{1,l})$ that has the gradient frequency response $g_G(\lambda)$ by $\mathbf{H}_l$ of order L_1

$$\mathbf{H}_l = \frac{1}{2} c_{l,0} \mathbf{I} + \sum_{l_1=1}^{L_1} c_{l,l_1} P_{l_1}(\mathbf{L}_{1,l}) \tag{2.43}$$

where $P_0(\mathbf{L}_{1,l}) = \mathbf{I}$ and $P_1(\mathbf{L}_{1,l}) = \frac{2}{\lambda_{G,\text{max}}} \mathbf{L}_{1,l} - \mathbf{I}$. The l_1th Chebyshev term for $l_1 \geq 2$ can be calculated as

$$P_{l_1}(\mathbf{L}_{1,l}) = 2P_1(\mathbf{L}_{1,l}) P_{l_1-1}(\mathbf{L}_{1,l}) - P_{l_1-2}(\mathbf{L}_{1,l}), \tag{2.44}$$

where the Chebyshev coefficients $c_{;l_1}$ can be computed as

$$c_{l,l_1} = \frac{2}{\pi} \int_0^{\pi} \cos(l_1 \phi) g_G\left(\omega(\cos\phi + 1)\right) \, \mathrm{d}\phi. \tag{2.45}$$

The frequency response $\tilde{H}_l(\lambda)$ of $\mathbf{H}_l$ is

$$\begin{cases} p_{l,0} := \frac{1}{2} c_{l,0} + \sum_{l_1=1}^{\lfloor L_1/2 \rfloor} (c_{l,2l_1} - c_{l,2l_1-1}), & \text{for } \lambda \in \mathcal{Q}_{\mathbf{H}} \cup \mathcal{Q}_{\mathbf{C}}, \\ \tilde{H}_{l,G}(\lambda) := \frac{1}{2} c_{l,0} + \sum_{l_1=1}^{L_1} c_{l,l_1} P_{l_1}(\lambda), & \text{for } \lambda \in \mathcal{Q}_{\mathbf{G}}, \end{cases} \tag{2.46}$$

with coefficients $p_{l,0}$ on the identity term of $\mathbf{H}_l$. This is the frequency response at the harmonic and curl frequencies associated with the kernel of $\mathbf{L}_{1,l}$. For a large value of L_1 we have $p_{l,0} \approx g_0$ and $\tilde{H}_{l,G} \approx g_G(\lambda)$, for $\lambda \in \mathcal{Q}_G$.

In the second step, we approximate the curl frequency response $g_C(\lambda)$ following the same logic to obtain the Chebyshev polynomial $\mathbf{H}_u := \mathbf{H}_u(\mathbf{L}_{\mathbf{1},\mathbf{u}})$ of order L_2

$$\mathbf{H}_u = \frac{1}{2} c_{u,0} \mathbf{I} + \sum_{l_2=1}^{L_2} c_{u,l_2} P_{l_2}(\mathbf{L}_{1,u}) \tag{2.47}$$

The frequency response $\tilde{H}_u(\lambda)$ of $\mathbf{H}_u$ is

$$\begin{cases} p_{u,0} := \frac{1}{2}c_{u,0} + \sum_{l_2=1}^{\lfloor L_2/2 \rfloor}(c_{u,2l_2} - c_{u,2l_2-1}), & \text{for } \lambda \in \mathcal{Q}_{\mathbf{H}} \cup \mathcal{Q}_{\mathbf{G}}, \\ \tilde{H}_{u,C}(\lambda) := \frac{1}{2}c_{u,0} + \sum_{l_2=1}^{L_2} c_{u,l_2} P_{l_2}(\lambda), & \text{for } \lambda \in \mathcal{Q}_{\mathbf{C}}. \end{cases} \tag{2.48}$$

with coefficients $p_{u,0}$ on the identity term of $\mathbf{H}_u$. This is the frequency response at the harmonic and gradient frequencies associated with the kernel of $\mathbf{L}_{\mathbf{1},\mathbf{u}}$. For a large value of L_2 we have $p_{u,0} \approx g_0$ and $\tilde{H}_{u,C} \approx g_C(\lambda)$, for $\lambda \in \mathcal{Q}_C$.

Finally, we sum $\mathbf{H}_l$ and $\mathbf{H}_u$ to obtain the filter that approximates the gradient and curl frequency responses. However, we notice from equation (2.46) that $\mathbf{H}_l$ generates a frequency response at both harmonic and curl frequencies, lifting the curl frequency unwantedly by $p_{l,0}$. Similarly, $\mathbf{H}_u$ lifts the gradient frequency response by $p_{u,0}$. To tackle this, we require that $g_G(0) = g_C(0) = g_0$ and subtract a term $g_0\mathbf{I}$ from the summation of the filters. Hence, the final Chebyshev polynomial design $\mathbf{H}_1$ with orders L_1 and L_2 is given by

$$\mathbf{H}_1 = \mathbf{H}_l + \mathbf{H}_u - g_0\mathbf{I}, \tag{2.49}$$

with the frequency response

$$\begin{cases} p_{l,0} + p_{u,0} - g_0, & \text{for } \lambda \in \mathcal{Q}_{\mathbf{H}} \\ \tilde{H}_{1,G}(\lambda) + p_{u,0} - g_0, & \text{for } \lambda \in \mathcal{Q}_{\mathbf{G}} \\ \tilde{H}_{u,C}(\lambda) + p_{l,0} - g_0, & \text{for } \lambda \in \mathcal{Q}_{\mathbf{C}}. \end{cases} \tag{2.50}$$

The approximation error of the Chebyshev polynomial design given in equation (2.49) is bounded.

2.2.4. Simplicial Trend Filtering

Simplicial trend filtering is applied for reconstructing simplicial signals from (partial) noisy observation [29]. The simplicial signals are the signals defined on nodes, edges triangles etc. of a higher-order network. The type of flow data includes, for instance, flows in traffic networks, exchange rates of a foreign currency (forex), flows in water, and so on [30]. These topological structures can be represented through SCs. Algebraically, these problems can be defined via their Hodge Laplacian matrices, utilizing two types of simplicial adjacency: lower adjacency and upper adjacency. Lower adjacency occurs when the two edges are adjacent if they share a common node. Conversely, upper adjacency occurs when two edges are adjacent if they belong to the same triangle. The reconstruction of a nonparametric signal is typically an ill-posed problem that requires regularizing with a term that introduces a particular bias into the solution. In GSP, the Tikhonov regularizer is used to recover the smooth signals of a graph from noisy observations by penalizing large node signal variations in adjacent nodes. This can be represented via the ℓ_2 norm of the signal differences in the adjacent nodes.

In real-world applications, we often encounter signals that are naturally associated with edges or sets of nodes (triangles). To address this, the simplicial signal reconstruction problem, equivalent to the signal reconstruction problem in GSP, is extended to the simplex. The Tikhonov regularizer has been extended in the following paper [31] for edge flow denoising and in [32] for interpolation of missing values. Both of the works penalize the fitting part with a term based on ℓ_2 norm of signal smoothness. Since SCs have two types of adjacency, we penalize them separately. The smoothness is penalized based on connectivity. For lower-connectivity, this is done using the ℓ_2 norm of the divergence of the flow, which corresponds to the net flow at the nodes. For upper-connectivity, it is penalized using the ℓ_2 norm of the curl, which corresponds to the net flow circulating within triangles. However, the ℓ_2 regularizers cannot preserve the divergence- or curl-free nature of the real-world edge flows, such as the forex markets [30] where we aim to have divergence or the curl of the edge flow to be zero.

ℓ_1 regularizer

Motivated by this, a regularized filtering is proposed in [29] which penalizes the ℓ_1 norm of the divergence and curl referred to as simplicial trend filtering (STF). Consider a noisy edge flow observation $\mathbf{y} = \mathbf{f}^* + \mathbf{n}$ where $\mathbf{n}$ is the additive noise.

The 0-order SFT estimates $\hat{\mathbf{f}}$ by solving

$$\hat{\mathbf{f}} = \arg\min_{\mathbf{f}\in\mathbb{R}^{N_1}} \|\mathbf{y}-\mathbf{f}\|_2^2 + \alpha\|\mathbf{B}_1\mathbf{f}\|_1 + \beta\left\|\mathbf{B}_2^\top\mathbf{f}\right\|_1, \tag{2.51}$$

where the parameters $\alpha \geq 0$ and $\beta \geq 0$. These parameters control the trade-off between the data fidelity and the ℓ_1 norm of the divergence and curl. When $\alpha = 0$ or $\beta = 0$, the SFT only accounts the regularization on the curl or the divergence, respectively. To solve the 0-STF convex problem, we can use the off-the-shelf algorithms, such as ADMM or Newton method [33].

As discussed in Section 2.2.2, the Hodge Laplacian acts as an shift operator for simplicial signals. By applying the lower Laplacian $\mathbf{L}_{1,l}$ and the upper Laplacian $\mathbf{L}_{1,u}$ to a flow $\mathbf{f}$, we obtain the shifted edge flow as discussed in Section 2.2.3. Similarly, for a general shifting $\mathbf{L}_{1,l}^p\mathbf{f}$ or $\mathbf{L}_{1,u}^q\mathbf{f}$, we consider the p-hop lower neighbours and q-hop upper neighbours [22, 34]. Thus, to account the information from multi-hop neighbours using a general (p, q)-order SFT, we can solve the following

$$\hat{\mathbf{f}} = \arg\min_{\mathbf{f}\in\mathbb{R}^{N_1}} \|\mathbf{y}-\mathbf{f}\|_2^2 + \alpha\left\|\Delta_\ell^{(p)}\mathbf{f}\right\|_1 + \beta\left\|\Delta_u^{(q)}\mathbf{f}\right\|_1, \tag{2.52}$$

where the operators $\Delta_\ell^{(p)}$ and $\Delta_u^{(p)}$ have the forms

$$\Delta_\ell^{(p)} = \begin{cases} \mathbf{B}_1^\top\Delta_\ell^{(p-1)} = \mathbf{L}_{1,l}^{\frac{p+1}{2}}, & \text{for odd } p, \\ \mathbf{B}_1\Delta_\ell^{(p-1)} = \mathbf{B}_1\mathbf{L}_{l,l}^{\frac{p}{2}}, & \text{for even } p, \end{cases} \tag{2.53}$$

with $\Delta_\ell^{(0)} = \mathbf{B}_1$ and

$$\Delta_u^{(q)} = \begin{cases} \mathbf{B}_2\Delta_u^{(q-1)} = \mathbf{L}_{1,u}^{\frac{q+1}{2}}, & \text{for odd } q, \\ \mathbf{B}_2^\top\Delta_u^{(q-1)} = \mathbf{B}_2^\top\mathbf{L}_{1,u}^{\frac{q}{2}}, & \text{for even } q, \end{cases} \tag{2.54}$$

with $\Delta_u^{(0)} = \mathbf{B}_2^\top$. In case where $p = 0$ and $1 = 0$, we obtain the 0-STF (2.51). When $p = 1$, we obtain the regularizer $\mathbf{L}_{1,l}\mathbf{f}$ which is a one-step lower shifting. When $q = 1$, we obtain the regularizer $\mathbf{L}_{1,u}\mathbf{f}$ which is a one-step upper shifting. At an edge i, the regularizers have the following entries, respectively

$$[\mathbf{L}_{1,l}\mathbf{f}]_i = 2\mathbf{f}_i + \sum_{j\in\mathcal{N}_{i,\ell}^{1,+}}\mathbf{f}_j - \sum_{k\in\mathcal{N}_{i,\ell}^{1,-}}\mathbf{f}_k, \quad \textit{(one-step lower shifting)} \tag{2.55}$$

$$[\mathbf{L}_{1,u}\mathbf{f}]_i = d_{i,u}\mathbf{f}_i + \sum_{j\in\mathcal{N}_{i,u}^{1,+}}\mathbf{f}_j - \sum_{k\in\mathcal{N}_{i,u}^{1,-}}\mathbf{f}_k, \quad \textit{(one-step upper shifting)} \tag{2.56}$$

where the edges j and k are positive and negative lower (upper) neighbours. The regularizers $\left\|\Delta_\ell^{(p)}\mathbf{f}\right\|_1$ and $\left\|\Delta_u^{(q)}\mathbf{f}\right\|_1$ can promote the divergence- and curl free property.

For interpolation, the proposed STF is adapted by replacing the data fitting term $\|\mathbf{y}-\mathbf{f}\|_2^2$ with $\|\mathbf{y}-\mathbf{C}\mathbf{f}\|_2^2$ where $\mathbf{y}\in\mathbb{R}^M$ and $\mathbf{C}\in\{0,1\}^{M\times N_1}$ is the selection matrix [29].

ℓ_2 regularizer

This section discusses the ℓ_2 regularizer from the frequency domain, which aims to generate a globally smooth edge flow. The ℓ_2 regularization problem has the form [20, 31]

$$\min_{\mathbf{f}\in\mathbb{R}^{N_1}} \|\mathbf{y}-\mathbf{f}\|_2^2 + \alpha\|\mathbf{B}_1\mathbf{f}\|_2^2 + \beta\left\|\mathbf{B}_2^\top\mathbf{f}\right\|_2^2, \tag{2.57}$$

where the regularizers $\|\mathbf{B}_1\mathbf{f}\|_2^2$ and $\left\|\mathbf{B}_2^\top\mathbf{f}\right\|_2^2$ are the edge flow variation measures [22, 34]. The closed-form solution can be written as

$$\hat{\mathbf{f}} = \left(\mathbf{I} + \alpha\mathbf{B}_1^\top\mathbf{B}_1 + \beta\mathbf{B}_2\mathbf{B}_2^\top\right)^{-1}\mathbf{y}. \tag{2.58}$$

where the operator $\mathbf{H} := \left(\mathbf{I} + \alpha\mathbf{B}_1^\top\mathbf{B}_1 + \beta\mathbf{B}_2\mathbf{B}_2^\top\right)^{-1}$ is a low-pass simplicial filter.

2.2.5. Hodge-compositional Gaussian Process

Gaussian processes (GPs) are a popular class of statistical models known for their ability to quantify the uncertainty in their predictions [35]. These models are defined by covariance kernels, which encode prior knowledge about the unknown function. Selecting an appropriate kernel can be particularly challenging, especially when the input space is non-Euclidean [36]. Specifically, we focus on functions defined over the edges of a network such as flows of signal. The Hodge compositional GPs are built as a combination of three GPs. Each of the GP models a specific part of the Hodge decomposition of an edge function, namely gradient curl and harmonic parts [37].

Gaussian processes. A random function $f : X \to \mathbb{R}$ defined over a set X is a Gaussian process $f \sim \mathcal{GP}(\mu, k)$ with mean function $\mu(\cdot)$ and kernel $k(\cdot,\cdot)$ if, for any finite set of points $\mathbf{x} = (x_1, \ldots, x_n)^\top \in X^n$, the random vector $f(\mathbf{x}) = (f(x_1), \ldots, f(x_n))^\top$ is multivariate Gaussian with mean vector $\mu(\mathbf{x})$ and covariance matrix $k(\mathbf{x},\mathbf{x})$. The kernel k of a *prior* encodes the prior knowledge about the unknown function. The mean of the kernel μ is assumed to be zero. Thus, a GP on graphs $f_0 \sim \mathcal{GP}(0, K_0)$ assumes f_0 is a random function with zero mean and a graph kernel K_0 which encodes the covariance between pairs of nodes.

Edge Gaussian processes. We can define GPs on the edges of simplicial 2-complexes SC_2, specifically, $\boldsymbol{f}_1 \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}_1)$ with zero mean and edge kernel $\boldsymbol{K}_1$. These are referred to as *edge GPs*. The edge Gaussian processes (GPs) are derived from stochastic partial differential equations (SPDEs) on edges. The detailed derivation can be found in [37].

The two edge GPs can be defined as

$$\begin{aligned}
\boldsymbol{f}_{1,\text{Matérn}} &\sim \mathcal{GP}\left(\mathbf{0}, \left(\frac{2\nu}{\kappa^2}\mathbf{I} + \mathbf{L}_1\right)^{-\nu}\right), \\
\boldsymbol{f}_{1,\text{diffusion}} &\sim \mathcal{GP}\left(\mathbf{0}, e^{-\frac{\kappa^2}{2}\mathbf{L}_1}\right),
\end{aligned} \tag{2.59}$$

which are the *edge Matérn* and *diffusion* GPs, respectively. These edge GPs impose a structured prior covariance that encodes the dependencies between edges. Given the eigendecomposition in Section 2.2.2, we obtain special classes of edge GPs by using certain types of eigenvectors when building edge kernels in equation (2.59). These types can be defined as the *gradient*, *curl* and *harmonic edge* GPs as follows

$$\boldsymbol{f}_G \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}_G), \quad \boldsymbol{f}_C \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}_C) \quad \boldsymbol{f}_H \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}_H) \tag{2.60}$$

where the gradient kernel, curl kernel and the harmonic kernel are

$$\boldsymbol{K}_G = \boldsymbol{U}_G\boldsymbol{\Psi}_G(\boldsymbol{\Lambda}_G)\boldsymbol{U}_G^\top, \quad \boldsymbol{K}_C = \boldsymbol{U}_C\boldsymbol{\Psi}_C(\boldsymbol{\Lambda}_C)\boldsymbol{U}_C^\top \quad \boldsymbol{K}_H = \boldsymbol{U}_H\boldsymbol{\Psi}_H(\boldsymbol{\Lambda}_H)\boldsymbol{U}_H^\top. \tag{2.61}$$

Hodge-compositional Edge GPs. Many real-world edge functions are indeed div- or curl-free, but not all. We combine the gradient, curl and harmonic GPs to define the Hodge-compositional (HC) edge GPs as follows. A Hodge-compositional edge Gaussian process $\boldsymbol{f}_1 \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}_1)$ is a sum of gradient, curl, and harmonic GPs, i.e., $\boldsymbol{f}_1 = \boldsymbol{f}_G + \boldsymbol{f}_C + \boldsymbol{f}_H$, where

$$\boldsymbol{f}_\square \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}_\square) \quad \text{with} \quad \boldsymbol{K}_\square = \boldsymbol{U}_\square\boldsymbol{\Psi}_\square(\boldsymbol{\Lambda}_\square)\boldsymbol{U}_\square^\top$$

for $\square = H, G, C$, where their kernels do not share hyperparameters. Given this definition, the following property of HC edge GP holds. Let $\boldsymbol{f}_1 \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}_1)$ be an edge GP. Its realizations then give all possible edge functions. It further holds that $\boldsymbol{K}_1 = \boldsymbol{K}_H + \boldsymbol{K}_G + \boldsymbol{K}_C$, and the three Hodge GPs are mutually independent.

The HC GP encodes the prior covariance $\mathrm{Cov}(f_1(e), f_1(e'))$ between edge functions over two edges e, e' as follows:

1. The covariance is the sum of three covariances $\mathrm{Cov}_\square = \mathrm{Cov}(f_\square(e), f_\square(e'))$ for $\square = H, G, C$.
2. Each $\mathrm{Cov}_\square$ encodes the covariance between the corresponding Hodge parts of f_1 without affecting the others.
3. No covariance is imposed across different Hodge components, e.g., $\mathrm{Cov}(f_G(e), f_C(e')) = 0$.

The eigenvalues $\boldsymbol{\Psi}_\square$ of HC GP's kernels are associated with three different Hodge spaces. These have individual parameters, which enable us to capture the different Hodge components of the edge functions. On the other hand, the Non-HC GPs require us to solve the Hodge decomposition in equation (2.24) and are strictly incapable of practical need when an eigenvalue is associated with both gradient and curl spaces.

3

Review of Existing Libraries

This chapter reviews the existing libraries in the domain of graphs and higher-order networks. As these fields continue to evolve, examining the tools and resources available to researchers and practitioners becomes increasingly crucial. By exploring the existing frameworks and libraries, the aim is to provide insights into the capabilities and limitations of the tools. Through this exploration, we identify missing functionality in each library.

3.1. Libraries for Graphs

There are several existing libraries for analysing graphs such as `NetworkX` [38], `PyTorch Geometric (PyG)` [39], `Deep Graph Library (DGL)` [40], and `KarateClub` [41].

`NetworkX`[1] is a Python-based library designed for creating, manipulating, visualizing, and studying the structure of complex networks. It offers efficient data structures for various graph types, including simple graphs, directed graphs, and multigraphs, along with many standard graph algorithms. This library is particularly useful for analyzing network structures and performing measures on large datasets. It is open-source and supported by a large community of contributors who maintain the core library and extend its functionality through a third-party ecosystem.

`PyG`[2] is a library built on `PyTorch`, designed for working with Graph Neural Networks (GNNs) and geometric deep-learning tasks. It provides a comprehensive toolkit for developing and training GNNs across a wide range of applications involving structured data. `PyG` includes various deep- learning methods for graphs, derived from numerous published papers, offering a flexible interface for building GNNs. It is optimized for handling large datasets, supporting scalable GNNs for graphs with millions of nodes.

`DGL`[3] is also a geometric deep-learning library and similar to `PyG` in terms of functionality. It is built for developing and training GNNs on top of existing Deep Learning frameworks such as `PyTorch`, `MXNet` and `TensorFlow`. It supports a wide range of graph types, including directed and undirected graphs, as well as graphs with multiple node and edge types. The library also offers multi-GPU and CPU training that can be scaled to graphs with millions of nodes. This allows DGL to be suitable for real-world applications.

`KarateClub`[4] is a Python framework that combines more than 30 state-of-the-art graph mining algorithms for unsupervised machine learning tasks. The main features include community detection, clustering, node and whole graph embedding techniques. The library is designed as an extension library for `NetworkX`. It's a scalable and efficient library that supports the analysis of large real-world datasets.

[1] https://github.com/networkx
[2] https://github.com/pyg- team/pytorch_geometric
[3] https://github.com/dmlc/dgl
[4] https://github.com/benedekrozemberczki/karateclub

Table 3.1 below compares the features across the different libraries for analysing graphs. However, none of these libraries supports higher-order interactions, especially for SCs. The rapid growth of the sub-field of higher-order network science has pushed scientists to take a different approach to tackle this problem [42].

Table 3.1: Comparison of features across different libraries for graph-based learning

Library	Loading datasets	Manipulating network	Analyzing network	Visualizing network	Unsupervised ML	GNNs
NetworkX	✓	✓	✓	✓	✓	
PyG	✓	✓				✓
DGL	✓	✓				✓
KarateClub	✓	✓	✓	✓	✓	

3.2. Libraries for Higher-Order Networks

In this section, we will explore the existing libraries for interacting with higher-order networks. Several existing libraries support higher-order networks, namely `HyperNetX` [43], `CompleX Group Interactions (XGI)` [44], `Reticula` [45], `Geometry Understanding in Higher Dimensions (GUDHI)` [46], `Deep HyperGraph (DHG)` [47], and `TopoX` [48].

`HyperNetX`[5] is a Python library that provides functionality for the analysis and visualization of hypergraphs, which are networks composed of nodes and hyperedges. Unlike traditional networks where the edge only connects two nodes, hyperedges can connect three or more nodes simultaneously. Key features of the library include hyperedge analysis of hypernetworks, clustering, community detection, and visualization. The library was developed by the Pacific Northwest National Laboratory.

`XGI`[6] is a Python library for higher-order interactions. It provides similar functionality to `HyperNetX` with a distinction: it also allows users to interact with SCs and directed hypergraphs. Users can create hypergraphs and SCs, analyze the structures, visualize them and provide a collection of higher-order datasets. The main features of the library include finding the degree of assortativity, computing the centralities of nodes and edges, clustering, and computing the shortest path in a hypergraph.

`Reticula`[7] provides a comprehensive set of tools to work with directed and undirected temporal networks, hypergraphs, and hypergraph temporal networks [45]. The library is built on C++ and is optimized for fast and efficient analysis of complex network structures. A Python version of the library is also available. The main features of the library include generating and randomising different types of networks, calculating and studying various properties of networks, such as calculating static or temporal network readability and forming event graphs.

`GUDHI`[8] is another Python-based library providing a wide range of methods for Topological Data Analysis and Higher Dimensional Geometry Understanding. It provides a wide range of methods, including data structures and algorithms for various types of SCs to compute geometric approximations of shapes and persistent homology. The types of SCs include alpha complex, cubical complex, rips complex, and witness complex. The algorithms are used for statistical analysis, computing persistence homology and bottleneck distance.

`DHG`[9] is a Python library built on `PyTorch` for deep-learning with GNNs and Hypergraph Neural Networks (HGNN). The library supports different low-order and high-order network structures. Low-order structures include graphs, directed graphs and bipartite graphs. Higher-order structures include hypergraphs and directed hypergraphs. The library has various spectral-based and spatial-based operations for different structures and provides performance metrics for the evaluation of different tasks. The library also has datasets and implemented models that can be utilized for research.

[5] https://github.com/pnnl/HyperNetX
[6] https://github.com/xgi-org/xgi
[7] https://github.com/reticula-network/reticula
[8] https://github.com/GUDHI
[9] https://github.com/iMoonLab/DeepHypergraph

TopoX[10] is a suite of Python libraries designed for machine learning and deep learning in topological domains. The library provides a tool for higher-order interactions using hypergraphs, simplicial, cellular, path and combinatorial complexes. The suite has three libraries, TopoNetX, TopoEmbedX and TopoModelX. TopoNetX focuses on the construction and computation of hypergraphs, SCs, cellular, path and combinatorial complexes. This includes working with the nodes, edges and higher-order cells. TopoEmbedX provides the functionality to embed topological domains into vector spaces, similar to the well-known graph-based embedding algorithms such as Node2Vec. These algorithms include DeepCell and Cell2Vec. TopoEmbedX builds on top of KarateClub and utilizes its functionality. TopoModelX is built on PyTorch and provides a comprehensive toolbox of higher-order message-passing functions for neural networks operating on topological domains. This library allows users to work with topological deep learning (TDL) techniques on topological data and leverage neural networks for pattern recognition.

Table 3.2 below compares the features of various libraries for interacting with higher-order networks. To facilitate comparison, we categorize these libraries into three distinct clusters based on their functionalities and offerings. In the next section, we will compare the libraries in more detail. Our implemented library, PyTSPL, is also listed in the table with the features it offers. In Chapter 4, we will delve into more detail about the features and implementation.

Table 3.2: Comparison of features across different libraries for higher-order interactions. Our implemented library, PyTSPL is also listed for direct comparison.

Library	SC	Hypergraphs	Other topological structure	Loading datasets	Analyzing structure	Visualizing structure	TDL	Signal Processing on SC	Linear filters	Hodge-compositional Edge Gaussian Process
HyperNetX		✓		✓	✓	✓				
XGI	✓	✓		✓	✓	✓				
Reticula		✓	✓	✓	✓	✓				
GUDHI	✓			✓	✓	✓				
DHG		✓		✓	✓	✓	✓			
TopoX	✓	✓	✓	✓	✓	✓	✓			
PyTSPL	✓			✓	✓	✓		✓	✓	✓

3.3. Comparison and Interaction of Libraries

The first cluster comprises of NetworkX, HyperNetX, XGI, and TopoNetX. These libraries support the functionality computations on graphs and hypergraphs. NetworkX facilitates computations on graphs, while HyperNetX and XGI offers functionalities for hypergraphs. Additionally, XGI offers additional support for SCs and directed hypergraphs [48]. TopoNetX has similar APIs to the libraries mentioned above with the adoption of topological domains including SCs, cell complexes, combinatorial complexes, and hypergraphs. We can also include Reticula in the cluster. The second cluster comprises TopoEmbedX and KarateClub. TopoEmbedX is built on top of KarateClub, extending the functionality of unsupervised ML of graphs to the topological domain via embeddings. The third cluster comprises deep learning libraries for graphs. This includes PyG and DGL, the two most popular libraries for geometric deep learning on graphs. These two libraries are closely related to TopoModelX and DHG that support deep learning on hypergraphs.

If we now examine the libraries strictly in terms of SC functionality, currently, XGI, GUDHI, and TopoX are the only libraries that offer support for it. XGI provides a data structure to store an SC, visualize it, and perform a quick analysis using the simplicial Kuramoto model. While these features are a good start, they are limited in terms of functionality. GUDHI offers similar SC functionality to XGI. TopoX surpasses in terms of functionality relative to the previous two libraries. TopoNetX offers functionality to calculate the shift operators of the SC that can be used for analyzing the structure and TopoModelX offers TDL for SCs that can be used for node and complex classification.

[10] https://github.com/pyt-team/TopoModelX

4

Design Goals and Description of PyTSPL

This chapter delves into the design goals and description of PyTSPL. We will explore the goals of the library, providing a clear understanding of its intended purpose and the problems it aims to solve. Following this, we will examine the key components and modules in detail, explaining their roles, functionalities, and how they interact with each other to deliver the desired outcomes.

4.1. Design Goals of PyTSPL

The primary goal of PyTSPL is to provide a unified and comprehensive toolset for the analysis, visualization and processing of signals on higher-order interactions, specifically SCs. As illustrated in Table 3.2, various libraries offer functionalities for SCs, but each with its limitations. These functionalities are scattered across different libraries, making it challenging to utilize multiple features simultaneously. By integrating and extending these functionalities within a single library, PyTSPL aims to facilitate research and practical applications in various domains. Thus, we define the functional and non-functional requirements of the library in the next sections.

4.1.1. Functional Requirements

Functional requirements are the features of the product that the software engineers must implement to meet the specific needs and expectations of the end user. These requirements detail the system's behaviour, functions, and interactions with users and other systems, ensuring that the product performs its intended tasks effectively. They are critical for guiding the development process and ensuring that the final product aligns with user needs.

The functional requirements of PyTSPL are as follows:

- **Support for SCs:** Enable support for the construction, manipulation and analysis of SCs. There are very limited libraries that support SC. Some libraries have a data structure to store and manipulate an SC, such as TopoX, however, the functionalities are limited.
- **Dataset loading:** Facilitate the loading and preprocessing of various real-world datasets and directly convert them into an SC without any custom scripts, streamlining the workflow from raw data to analysis-ready formats.
- **Structural analysis of SCs**: Extensive functionalities for analyzing the structure of SC. This includes calculating the shift-operators e.g. Laplacian matrix, extracting eigenvalues and eigenvectors using eigendecomposition, and applying k-step shifting using simple function calls.
- **Visualizing SCs**: Provide visualization tools to help users intuitively understand and present the structural characteristics of the SC. Even though the libraries have the functionality to visualize network data, none of the libraries provide comprehensive functionality to visualize an SC.

- **Signal processing on SCs**: Include functionalities for signal processing on SCs, supporting advanced analysis techniques such as Hodge decomposition, Simplicial Fourier Transform and simplicial embeddings.
- **Applying linear filters on SCs**: Offer linear filtering methods to facilitate various signal processing tasks e.g. edge flow denoising, subcomponent extraction, on higher-order structures, such as Least-squares filter, Grid-based filter, Chebyshev polynomial filter and Simplicial Trend filtering.
- **Hodge-compositional Edge Gaussian Processes:** Provide functionality of edge Gaussian processes to provide probabilistic modelling and inference capabilities on edges within the analyzed structures.

4.1.2. Non-functional Requirements

The non-functional requirements of the system are defined as the quality constraints that the system must satisfy to enhance the overall user experience. These requirements focus on how the system performs rather than what it does.

The non-functional requirements of `PyTSPL` are as follows:

- **Unified Framework**: Consolidate multiple functionalities into a single, cohesive library. This includes streamlining the process of loading built-in datasets defined over a network, plotting them, and applying advanced signal processing techniques to analyze the SCs.
- **Usability**: Design the library to be user-friendly for researchers and practitioners across different fields. New users should be able to quickly learn and navigate the library with minimal effort. This includes providing comprehensive documentation, intuitive interfaces, and helpful error messages. Additionally, the library should offer examples and tutorials to assist users in understanding and utilizing its features effectively.
- **Scalability**: Ensure the library can handle large datasets defined over a network efficiently. This involves optimizing algorithms and data structures to perform well with increasing data size.
- **Flexibility**: Allow for the easy extension and customization of features by the open-source community e.g. via GitHub issues. The library should be designed with a modular architecture, enabling users to add new functionalities or modify existing ones without altering the core components. This includes providing extensive documentation, tutorials and API references of existing functionalities.
- **Maintainability**: Design the library such that it is easy to maintain and improve over time. This includes writing in-code documentation and the following best coding practices. Additionally, having good test coverage is implemented to catch and fix bugs at the early stage, ensuring the reliability of the library. Furthermore, there are continuous development/deployment (CI/CD) pipelines in place to streamline the process of integrating code changes into a repository. There are clear guidelines on how to contribute to the project such that the open-source community can easily contribute.
- **Reusability**: Ensure the library's components can be easily reused in different components. This involves designing modular and well-encapsulated functions and classes that adhere to standard programming practices. By promoting reusability, the library can save development time and effort, and encourage consistent use of best practices across different modules.

4.2. Key Modules and Submodules

`PyTSPL` is organized into key modules and submodules to enhance the overall design, maintainability, and functionality of the system. Currently, the library has six key modules.

4.2.1. `io`

The `io` module is responsible for reading raw datasets and preprocessing them such that they can be passed on to the next module to build an SC. The raw datasets can be read using the `network_reader` submodule. The data can be in various formats such as comma-separated values (CSV), tab-delimited text files (TNTP) or plain incidence matrices $\mathbf{B}_1$ and $\mathbf{B}_2$. There is additional functionality to read the coordinates and edge flow of the SC as well. Most of the network datasets are in CSV or TNTP

format, so they can be read using the built-in functions. Additionally, there is functionality to load built-in datasets using the `dataset_loader` submodule. The submodule provides various options for datasets such as forex exchange and transportation networks from different cities. The user can view the available datasets as well. Finally, there is also a quick way to generate a random SC using the `sc_generator` submodule.

The method and their respective descriptions of the `io` module are given in Table 4.1 below.

Table 4.1: Methods and their descriptions of the `io` module.

Method	Description
`list_datasets()`	List the currently available datasets that can loaded directly into an SC.
`load_dataset(...)`	Load the dataset by passing in the available dataset name.
`read_csv(...)`	Read custom data defined over a network using the CSV format.
`read_tntp(...)`	Read custom data defined over a network using the TNTP format.
`read_B1_B2(...)`	Read custom data defined over a network using incidence matrices $\mathbf{B}_1$ and $\mathbf{B}_2$.
`read_coordinates(...)`	Read the coordinates of the network. This is useful e.g. when plotting the SC.
`read_flow(...)`	Read the edge flow of the network. This is useful e.g. when applying filter designs to the SC.

4.2.2. `simplicial_complex`

Building an SC. Once the raw data is read using the `io` module, it is passed to the `sc_builder` submodule. This submodule constructs an SC from the raw data. The 2- simplices are created based on user-defined criteria: *triangle-based* and *distance-based*. The *triangle-based* method finds all the triangles in the graph and considers them as 2-simplices. The *distance-based* method finds all the triangles and only keeps the ones where the distance between the nodes is less than a threshold ϵ. By default, when we load a dataset using the `load_dataset` function, the SC is built using the triangle-based method. The Table 4.2 illustrates the two ways to build the SC.

Table 4.2: The two ways to build the SC using the `sc_builder` module.

Method	Description
`to_simplicial_complex (condition="all")`	Build the 2-simplices of the SC using the *triangle-based* method.
`to_simplicial_complex (condition="distance", dist_threshold=0.8)`	Build the 2-simplices of the SC using the *distance-based* method.

SC data structure class. Once the SC is built, it is stored in the `simplicial_complex` data structure class. This data structure allows us to efficiently compute the algebraic properties of the SC. This includes computing the adjacency matrix, incidence matrices and the Laplacian matrices. Also, it allows the computation of k-step lower and upper shifting of the SC. The methods and their implementation details provided by the `simplicial_complex` class are shown in Table 4.3 below.

4.2.3. `plot`

The `plot` module offers the functionality to plot an SC in a quick and meaningful way. The user can draw nodes, edges, and their respective labels in a custom way that meets their needs. The plots are drawn using the network coordinates provided. If the coordinates are not provided by the user or don't exist in the corresponding dataset, the module automatically generates coordinates using `NetworkX`'s `spring_layout`.

The methods and descriptions offered by `plot` module are given in Table 4.4 below.

Table 4.3: Methods and descriptions of the of `simplicial_complex` module.

Method	Description
`print_summary()`	Prints the summary of the SC including the number of nodes, edges, triangles, shape, and maximum dimension with the coordinates and flow.
`generate_coordinates()`	Generates random coordinates for an SC if they are not provided, for instance for plotting purposes. The coordinates are generated using `NetworkX`'s `spring_layout`.
`get_node_features()`	Returns the node features of the SC. These features are directly loaded from the dataset.
`get_edge_features()`	Returns the edge features of the SC. These features are directly loaded from the dataset.
`get_faces()`	Returns the faces of the SC.
`shape`	Returns the shape of the SC. This includes the number of 0-, 1- and 2-simplices.
`max_dim`	Returns the maximum dimension of the SC.
`simplices`	Returns the simplices of the SC.
`incidence_matrix(...)`	Returns the incidence matrix of the SC using the rank.
`adjacency_matrix()`	Returns the adjacency matrix of the SC defined in equation 2.1.
`laplacian_matrix()`	Returns the Laplacian matrix of the SC defined in equation 2.3.
`hodge_laplacian_matrix(...)`	Returns the Hodge Laplacian matrix of the SC using the rank defined in the equation 2.19.
`lower_laplacian_matrix(...)`	Returns the lower Laplacian matrix of the SC using the rank defined in the equation 2.21.
`upper_laplacian_matrix(...)`	Returns the upper Laplacian matrix of the SC using the rank defined in the equation 2.22.
`apply_lower_shifting(...)`	Applies lower shifting to an edge flow for k steps defined in the equation 2.37.
`apply_upper_shifting(...)`	Applies upper shifting to an edge flow for k steps defined in the equation 2.38.
`apply_k_step_shifting(...)`	Applies k-step shifting to an edge flow defined in the equation 2.39.
`get_total_variance(...)`	Calculates the total variance of the SC.
`get_divergence(...)`	Calculates the total divergence of the edge flow defined in the equation 2.25.
`get_curl(...)`	Calculates the total curl of the edge flow defined in the equation 2.26.
`get_simplicial_embeddings(...)`	Calculates the simplicial embeddings of the edge flow $\mathbf{f}$ and returns the harmonic, curl and gradient embeddings using the equation 2.33.
`get_component_eigenpair(...)`	Calculates the eigenpairs of harmonic, curl or gradient components using the equation 2.29.
`get_component_flow(...)`	Calculates the harmonic, curl or gradient flow based on the component parameter passed using the equations 2.28 and 2.27

An example plot of the *chicago-sketch* transportation network dataset drawn using the `draw_network` function is shown in Figure 4.1 below.

Table 4.4: Methods and descriptions offered by the `plot` module.

Method	Description
`draw_sc_nodes(...)`	Draw the nodes of the SC and customize the plot using parameters like node size, node colour, camp, edge font size etc. This method is designed to customize the drawing of nodes.
`draw_sc_edges(...)`	Draw the edges of the SC and customize the plot using parameters like edge colour, edge width, edge colour etc. This method is designed to customize the drawing of edges.
`draw_network(...)`	Draw the SC with or without edge flow and labels. The user can pass in parameters to customize the plot according to their own needs, such as node size, edge width etc. The method uses `draw_sc_edges()` and `draw_sc_nodes()` to plot the SC.
`draw_hodge_decomposition(...)`	Plot the Hodge decomposition for the given component and edge flow. If the component name is not passed as a parameter, it draws all three components i.e. harmonic, curl and gradient flows.
`draw_eigenvectors(...)`	Plot the eigenvectors and their corresponding eigenvalues for a given component. The user can pass in the indices and only plot certain eigenpairs.

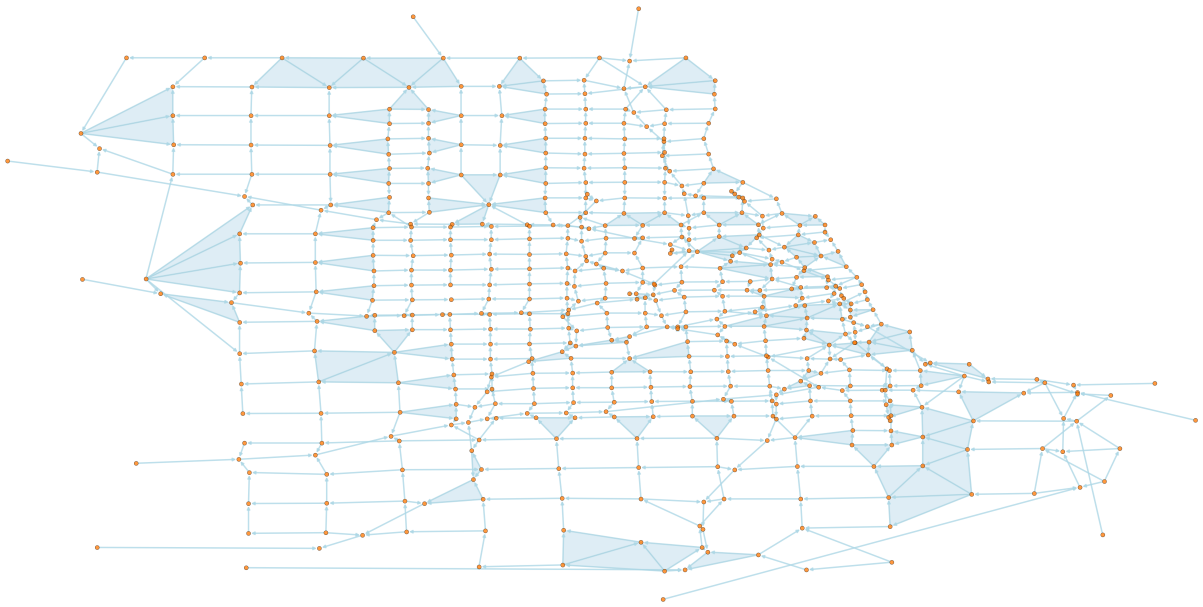


Figure 4.1: SC plot of the chicago-sketch transportation network dataset.

4.2.4. `decomposition`

The `decomposition` module offers functionality to decompose signals using *eigendecomposition* and *Hodge decomposition*. The *eigendecomposition* functionality allows users to extract the gradient, harmonic and curl eigenvalues and eigenvectors, whereas, the *Hodge decomposition* offers the users to get the gradient, harmonic and curl component of the edge flow.

The methods and descriptions offered by `eigendecomposition` and `Hodge decomposition` submodule are given in Table 4.5 and 4.6 below, respectively. These methods are encapsulated in their module and used by others as a service for code cohesion.

Table 4.5: Methods and descriptions offered by the `eigendecomposition` submodule.

Method	Description
`get_total_variance(...)`	Calculates the total variance of the SC and returns it.
`get_divergence(...)`	Calculates the divergence of the SC and returns it using the equation 2.25.
`get_curl(...)`	Calculates the curl of the SC and returns it using the equation 2.26.
`get_harmonic_eigenpair(...)`	Calculates the harmonic eigenpairs as discussed in Section 2.2.2.
`get_curl_eigenpair(...)`	Calculates the curl eigenpairs as discussed in Section 2.2.2.
`get_gradient_eigenpair(...)`	Calculates the gradient eigenpairs as discussed in Section 2.2.2.
`get_eigendecomposition(...)`	Calculates the eigendecomposition as discussed in Section 2.2.2.

Table 4.6: Methods and descriptions offered by the `Hodge decomposition` submodule.

Method	Description
`get_harmonic_flow(...)`	Calculates the harmonic component of an edge flow as discussed in Section 2.2.2.
`get_curl_flow(...)`	Calculates the curl component of an edge flow using the equation 2.28.
`get_gradient_flow(...)`	Calculates the gradient component of an edge flow using the equation 2.27.

4.2.5. `filters`

The `filters` module aims to provide simplicial convolutional filters for the SC, namely, Least-Squares filter (2.2.3), Grid-Based filter (2.2.3), and Chebyshev polynomial filter design (2.2.3). The applications for these filters include subcomponent extraction and edge flow denoising. Additionally, the module offers functionality for simplicial trend filtering (2.2.4) for reconstructing simplicial signals from (partial) noisy observations.

The purpose and functionality of each of the filter classes are explained below.

`BaseFilter`: Each filter is implemented as a filter class that extends from the `BaseFilter` parent class. This programming concept is known as *inheritance* and is a good choice when we want to reuse code from the base class and make global changes to the extended classes by changing a base class. In this case, we define various methods in the `BaseFilter` class that are reused in the child classes for code reusability. Additionally, we update the *history* attribute using one method from the base class, which stores the built *filter, estimated frequency, frequency responses, extracted component error and filter error*. The inheritance diagram for the `filters` module is shown in Figure 4.2 below.

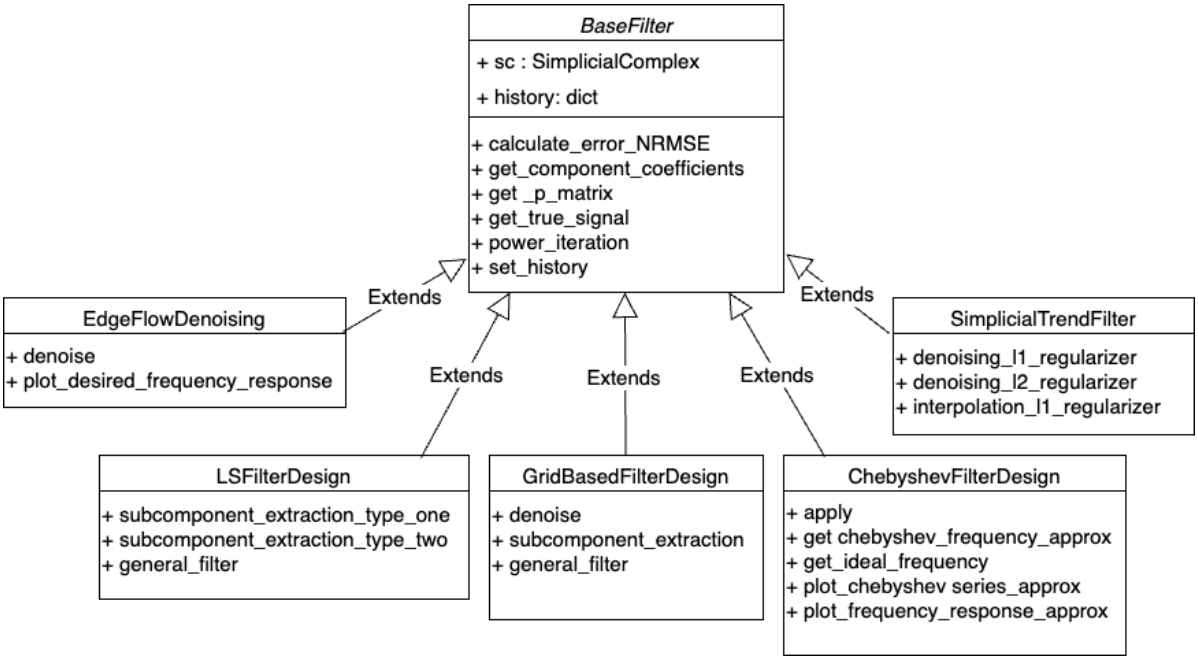


Figure 4.2: Class inheritance diagram for the `filters` module.

`EdgeFlowDenoising`: This class implements a low-pass filter $\mathbf{H_P}$ for edge flow denoising. Given a noisy edge flow $\mathbf{f} = \mathbf{f}^0 + \epsilon$, where $\mathbf{f}^0$ is the true edge flow and ϵ is a zero-mean white Gaussian noise, we can estimate the edge flow $\hat{\mathbf{f}}$ by solving the following regularized problem $\min_{\hat{\mathbf{f}}} \|\hat{\mathbf{f}} - \mathbf{f}\|_2^2 + \mu \hat{\mathbf{f}} \mathbf{P} \hat{\mathbf{f}}$ [20, 31]. The optimal solution to the regularized problem is given by $\hat{\mathbf{f}} = \mathbf{H_P}\hat{\mathbf{f}} := (\mathbf{I} + \mu\mathbf{P})^{-1}\mathbf{f}$. The `denoise` function in the class finds the optimal solution given the noise-free flow $\mathbf{f_0}$, noisy flow $\mathbf{f}$, the $\mathbf{P}$ choice matrix *p_choice* and the $\mathbf{mu}$ values *mu_vals*. Once the solution is found, it builds the filter and calculates the frequency response of the error for each filter order. The users can also calculate and plot the desired frequency response given the $\mathbf{P}$ choice matrix using the `plot_desired_frequency_response` function.

`LSFilterDesign`: This class implements subcomponent extraction of type I and type II filters using the LS-filter design. In type I filter, we have $L_1 = L_2 = L$ and $\boldsymbol{\alpha} = \boldsymbol{\beta}$. To apply the type I filter, users can use the `subcomponent_component_extraction_type_one` function. The type I filter does not differentiate between lower and upper adjacencies, resulting in a loss of expressive power as it cannot distinguish between them. To address this limitation, we additionally offer the type II filter using the `subcomponent_component_extraction_type_two` function, which treats lower and upper adjacencies separately, where L_1 != L_2 and $\boldsymbol{\alpha}$!= $\boldsymbol{\beta}$. This approach provides greater flexibility and control over the frequency response. Finally, the `general_filter` function extracts and returns the subcomponents for all three components: harmonic, gradient, and curl. The LS problem is solved via the pseudo inverse of the system matrix using the LS problem using the `np.linalg.lstsq` function provided by `numpy`. The function takes in the system matrix and the alpha coefficients as input and returns the coefficients of the filter, which are then used to build the filter. Once the filter is built, we calculate the estimated subcomponent flow $\hat{\mathbf{f}}$ and calculate the errors and frequency responses for each filter size.

`GridBasedFilterDesign`: This class implements edge flow denoising and subcomponent extraction using the grid-based filter design. The `subcomponent_extraction` function uses a grid-based simplicial filter $\mathbf{H}_1$ for edge flow denoising and the `general_filter` function extracts all three components: harmonic, gradient and curl. The filter design avoids the eigenvalue computation of the 1-Hodge Laplacian $\mathbf{L_1}$ and aims to match the desired frequency response in a continuous interval where the exact frequencies lie. To achieve this, we sample M_1 and M_2 grid points from the interval $[\lambda_{G,\text{min}}, \lambda_{G,\text{max}}]$ and $[\lambda_{C,\text{min}}, \lambda_{C,\text{max}}]$ and solve the problem as an LS problem of form equation 2.42. The LS problem is solved via the pseudo inverse of the system matrix using the `np.linalg.lstsq` function provided by `numpy`. To approximate the largest eigenvalue, we implement the `power_iteration` function. For the smallest eigenvalue, we set it to a small value greater than 0 or as 0.

`ChebyshevPolynomialFilterDesign`: This class implements the Chebyshev polynomial filter design to tackle the issue of numerical stability by solving the LS problem using the Vandermonde matrix as a system matrix. In real-world datasets, some of the simplicial frequencies are close together, leading to an ill-conditioning of the LS-based filter design. This issue is tackled by the Chebyshev polynomial design, which requires a continuous desired frequency response to extract the gradient subcomponent. Ideally, this is an indicator function $\mathbf{1}_{\lambda>0}$ with $\lambda \in [0, \lambda_{G,\text{max}}]$. In this case, we use the logistic function $g_{\mathbf{G}}(\lambda) = \frac{1}{1+\exp^{-k(\lambda-\lambda_0)}}$ with a growth rate $k > 0$ and the midpoint is λ_0. Given the smallest gradient frequency close to 0, we require large k and a small λ_0 to achieve a good approximation of the ideal indicator function. Given an edge flow $\mathbf{f}$, matrix choice $\mathbf{P}$ and *component*, we can apply the Chebyshev filter design using the `apply` function. First, we approximate the Chebyshev series using the *number of points*, minimum and maximum *domain*, and the *cut off frequency*. For this, we calculate the Chebyshev coefficients using the `chebpy`[1] library. Next, we approximate the component frequency using either equation 2.43 or 2.47. Finally, we calculate the extracted component error and the filter error. The users can plot the Chebyshev series approximation using the `plot_chebyshev_series_approx` and the frequency response approximation using the `plot_frequency_response_approx` function.

`SimplicialTrendFilter`: This class implements the edge flow denoising and the interpolation tasks using the STF (see Section 2.2.4). The users can apply the `denoising_l1_regularizer` function to denoise the edge flow using the ℓ_1 regularizer (see Section 2.2.4). The function takes in parameters such as the edge flow $\mathbf{f}$, the order of the STF, component name (divergence or curl) and the regularization parameter α and β. The order of the STF and the component name determine the operators $\Delta_\ell^{(p)}$ and $\Delta_u^{(p)}$ as given in equation 2.53 and 2.54, respectively. Next, it solves the equation 2.52 using the `cvxpy`[2] library and calculates the frequency response of the filter, the divergence or curl flow, with its corresponding NRMSE error and correlation. Finally, it updates the history of the filter so users can retrieve the filter results. Similarly, users can apply the `denoising_l2_regularizer` function to denoise the edge flow using the ℓ_2 regularizer (see Section 2.2.4). The function uses the closed-form of equation 2.58 and approximates $\hat{\mathbf{f}}$. The class also offers functionality for interpolation tasks. The user can apply the `interpolation_l1_regularizer` function to interpolate. This function solves the equation 2.52 using the `cvxpy` library but replaces the data fitting term by $\|\mathbf{y} - \mathbf{f}\|_2^2$ with $\|\mathbf{y} - \mathbf{C}\mathbf{f}\|_2^2$ as discussed in Section 2.2.4.

4.2.6. `hodge_gp`

Hodge-compositional edge Gaussian processes are used to model functions defined over the edge set of an SC (see Section 2.2.5). This method intends to learn the flow-type data on networks where edge flows can be characterized by discrete divergence and curl. This module is built on `PyTorch` and allows users to directly train their model and make predictions on real-world datasets e.g. forex dataset.

The types of kernels are defined in the `Kernels` enum class. These kernels include diffusion, diffusion non-HC, Matérn, Matérn non-HC, Laplacian and Laplacian non-HC (see equation 2.59). The kernels are built using the `gpytorch.kernels.Kernel` base class provided by `gpytorch`[3]. These kernels can be defined separately for each dataset to ensure the best performance. When initializing a kernel, the users can use the `KernelSerializer` class by specifying the kernel type and the dataset name. This ensures the initialization of different kernels is encapsulated from the users. For training a model, the `HodgeGPTrainer` class can be leveraged by passing in the loaded SC and the edge flow defined on the SC. The trainer object encapsulates the methods required to train the models. Users can use the built-in class function `train_test_split` to split the data into training and testing sets into different ratios. Once the data is spit, the users can initialize the `ExactGP` model, inherited from `gpytorch.models.ExactGP` which is a base class for any Gaussian process latent function to be used in conjunction with exact inference. The model takes the data, the likelihood function and the kernel as the parameters. Next, we train the `ExactGP` model, the likelihood function and pass these as parameters to the `HodgeGPTrainer` along with the training data to start the training process. The model is trained by calling the `train` function with parameters such as the training iterations, learning rate and the optimizer e.g. Adam. Once the training is complete, users can get the model's learned parameters

[1] https://github.com/chebpy/chebpy
[2] https://www.cvxpy.org/
[3] https://gpytorch.ai/

using the `get_model_parameters` function. To build a kernel, users can use the built-in class methods and obtain the kernel e.g. Matérn kernel can be built using the `build_matern_kernel` function which uses the equation 2.59.

4.3. Interaction Between the Modules

The flow and interaction between the key modules can be seen in Figure 4.3 below. The process begins with reading network data either by directly loading build-in datasets or by loading custom datasets using the `io` module. The module preprocesses this data to ensure it is in a suitable format. The preprocessed data is then used to construct an SC in different ways and stored in a data structure using the `simplicial_complex` module.

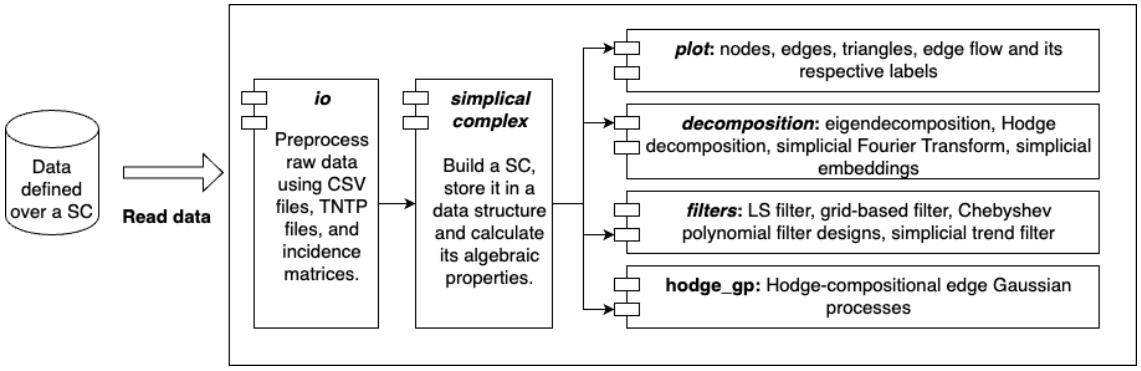


Figure 4.3: The flow and interaction between the key modules.

Once the SC is built, the user has various options including

1. **Plotting the SC**: Visualize the nodes, edges, triangles, edge flow and their respective labels within the SC to understand the structure and relationships in the network data.
2. **Decomposing the Signals**: Use techniques such as eigendecomposition and Hodge decomposition to analyze and understand the properties of the signals within the SC. This also includes calculating simplicial Fourier Transform and simplicial embeddings.
3. **Applying Various Filters** for:
    - Subcomponent Extraction
    - Edge Flow Denoising
    - Reconstructing simplicial signals from (partial) noisy observation
4. **Applying the Hodge-compositional edge Gaussian processes**

In the next chapter, we will dive into each module and provide pedagogical code examples for them.

4.4. Datasets

The library offers pre-loaded datasets to get started. Currently, the library contains various datasets, namely *foreign exchange*, *lastfm-dataset- 1k*[4] *transportation networks*[5] and so on. All the available datasets are listed in Table 4.7 below.

The signals in the datasets are defined solely on the edges and the nodes have coordinates as features. The features of the edges are detailed in Table 4.8 below. Note that the transportation networks datasets include all the city names from Table 4.7. Some datasets lack node coordinates or edge flow. If the node coordinates don't exist, they are automatically generated using `NetworkX`'s `spring_layout`. In case the edge flow doesn't exist, the user can manually define it and use it. These datasets can be

[4] https://github.com/eifuentes/lastfm-dataset-1K
[5] https://github.com/bstabler/TransportationNetworks

Table 4.7: List of built-in datasets in `PyTSPL`.

Dataset Name	Num. of Nodes	Num. of Edges	Num. of Triangles	Coordinates	Edge Flow
anaheim	416	634	54	✓	✓
barcelona	1020	1798	164	✓	✓
chicago-regional	12982	20627	807	✓	✓
chicago-sketch	546	1088	112	✓	✓
forex	25	210	770		✓
goldcoast	4807	5952	269	✓	
lastfm-1k	657	1997	1276	✓	✓
opashi-powergrid	4940	6593	0		
paper (data)	7	10	3	✓	✓
siouxfalls	24	38	2	✓	✓
winnipeg	1052	1595	185	✓	✓
wsn	114	164	2	✓	✓

directly loaded via the `dataset_loader` module. An example of this is given in the next chapter. For future work, the aim is to add additional datasets.

Table 4.8: List of features in the built-in datasets.

Dataset Name	Edge Features
transportation networks	capacity, length, free flow time, b, power, speed, toll, link type, volume, cost
forex	bid, ask, midpoint
lastfm-1k	artist, music
opashi-powergrid	
paper (data)	arbitrary flow
wsn	water flow

5

Pedagogical Tutorials

In this chapter, we delve into the foundational aspects of using the library. By exploring elementary examples, the aim is to provide a clear and practical understanding of how to navigate through the basic functionalities and tools available. These examples are carefully chosen to illustrate key concepts and operations that form the building blocks for more advanced analyses. For clarity, the important functions implemented are color-coded in blue.

5.1. Loading a SC

The `io` module handles the loading of an SC (see Section 4.2.1). Let's start by loading a SC using an existing dataset defined over a network. The library offers built-in datasets that can be directly loaded or users can load their custom dataset using the provided functions. For experimental purposes, the users can also generate a random SC. Once the SC is loaded up, we can use it for further analysis.

Built-in Dataset

Let's load an existing built-in dataset using the `dataset_loader` module. To list the available built-in datasets, we can use the `list_datasets` function as follows

```
from pytspl import list_datasets

list_datasets()
```

Listing 5.1: List existing built-in datasets.

The list of the available built-in datasets will be shown as follows

```
["barcelona",
 "chicago-regional",
 "siouxfalls",
 "anaheim",
 "test_dataset",
 "goldcoast",
 "winnipeg",
 "chicago-sketch",
 "paper",
 "forex",
 "lastfm-1k-artist"]
```

Listing 5.2: List of available built-in datasets.

For this example, we will load the *paper* dataset. Once the dataset is loaded, it will return the SC data structure, coordinates, and the edge flow defined over the SC. The existing built-in dataset can be loaded using the `load_dataset` function

```
from pytspl import load_dataset

```

```
sc, coordinates, flow = load_dataset(dataset="paper")
```

Listing 5.3: Loading the existing built-in *paper* dataset.

Once the dataset is loaded, the summary of the dataset will be printed as follows

```
Num. of nodes: 7
Num. of edges: 10
Num. of triangles: 3
Shape: (7, 10, 3)
Max Dimension: 2
Coordinates: 7
Flow: 10
```

Listing 5.4: Summary of the *paper* dataset.

Note that the number of coordinates should match the number of nodes and the flow length should correspond to the number of edges. Both coordinates and flow are represented as *dict*, where the keys are the edges, and the values are the respective coordinates or flow.

Custom Dataset

Users can also load their datasets by utilizing the built-in functions to preprocess data and build an SC. They can load an SC from a CSV file, a TNTP file, or through incidence matrices $\mathbf{B}_1$ and $\mathbf{B}_2$. Once the SC is loaded, the user can print the summary using the `print_summary` function.

CSV and TNTP. To load the datasets from a CSV format, the user can use the `read_csv` function as follows

```
from pytspl import read_csv

filename = f"{PAPER_DATA_FOLDER}/edges.csv"
delimiter = " "
src_col = "Source"
dest_col = "Target"
feature_cols = ["Distance"]

# reading from a CSV file
sc = read_csv(
        filename=filename,
        delimiter=delimiter,
        src_col=src_col,
        dest_col=dest_col,
        feature_cols=feature_cols
    ).to_simplicial_complex(condition="all")

# print summary
sc.print_summary()
```

Listing 5.5: Loading an SC using a CSV file.

Similarly, the datasets can be loaded from TNTP format using the `read_tntp` function.

Incidence matrices. We can use the `read_B1_B2` function to load the incidence matrices and the `to_simplicial_complex` function to build the SC.

```
from pytspl import read_B1_B2

B1_filename = "pytspl/data/paper_data/B1.csv"
B2_filename = "pytspl/data/paper_data/B2t.csv"

# extract the triangles
scbuilder, triangles = read_B1_B2(
    B1_filename=B1_filename,
    B2_filename=B2_filename
)

# build the SC using the extracted triangles
sc = scbuilder.to_simplicial_complex(triangles=triangles)

```

```
# print summary
sc.print_summary()
```

Listing 5.6: Loading an SC using incidence matrices $\mathbf{B}_1$ and $\mathbf{B}_2$.

Coordinates. The coordinates of the SC can be loaded using the `read_coordinates` function as follows

```
from pytspl.io.network_reader import read_coordinates

# load coordinates
coordinates_path = "pytspl/data/paper_data/coordinates.csv"

coordinates = read_coordinates(
    filename=coordinates_path,
    node_id_col="Id",
    x_col="X",
    y_col="Y",
    delimiter=" "
)
```

Listing 5.7: Loading coordinates of a SC.

The user can specify the *node_id_col*, *x_col*, and *y_col* names as well as the *delimiter* to read the coordinates. Different data files may have different column names and delimiters.

Edge flow. The edge flow defined over the SC can be loaded using the `read_flow` function as follows

```
from pytspl.io.network_reader import read_flow

# load flow
flow_path = "pytspl/data/paper_data/flow.csv"

flow = read_flow(filename=flow_path, header=None)
```

Listing 5.8: Loading edge flow defined over a SC.

Generating a SC

The users can randomly generate an SC using the `generate_random_simplicial_complex` function in the following way

```
from pytspl import generate_random_simplicial_complex

# generate a random SC
sc, coordinates = generate_random_simplicial_complex(
    num_of_nodes=7,
    p=0.25,
    dist_threshold=0.8
    seed=42,
)

# print summary
sc.print_summary()
```

Listing 5.9: Generate a random SC.

The user defines the number of nodes, the probability of an edge between two nodes and the distance threshold for building the 2-simplices. An example of a randomly generated SC with seven nodes is shown in Figure 5.1 below.

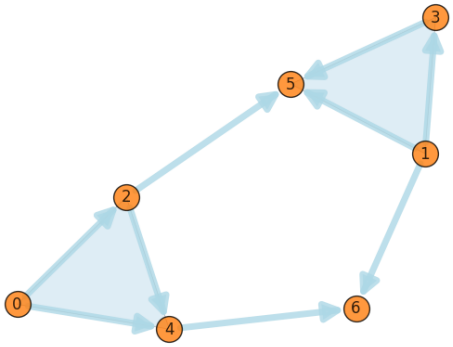


Figure 5.1: A randomly generated SC with seven nodes.

5.2. Building a SC

There are two ways to build an SC using `PyTSPL`, namely the *triangle-based* method and *distance-based* method using the `to_simplicial_complex` function. More details about the building methods are given in Section 4.2.2. By default, when we load a dataset using the `load_dataset` function, the SC is built using the triangle-based method.

Triangle-based method. To build a SC using all the triangles, we can do the following

```
from pytspl import read_csv

sc = read_csv(
        filename=filename,
        delimeter=delimeter,
        src_col=src_col,
        dest_col=dest_col, feature_cols=feature_cols
    ).to_simplicial_complex(condition="all")
```

Listing 5.10: Building the SC using the triangle-based method.

where the `condition` parameter is set to *all*. This parameter specifies to use all the triangles as the 2-simplices.

Distance-based method. To build a SC using the distance-based method, we can do the following

```
from pytspl import read_csv

# reading a csv file
sc = read_csv(
        filename=filename,
        delimeter=delimeter,
        src_col=src_col,
        dest_col=dest_col, feature_cols=feature_cols
    )
    .to_simplicial_complex(
        condition="distance",
        dist_col_name: str = "distance",
        dist_threshold: float = 1.5,
    )
```

Listing 5.11: Building the SC using the distance-based method.

where the `condition` parameter is set to *distance*, `dist_col_name` specifies which data attribute has the distance values for comparison and the `dist_threshold` specifies the value of ϵ.

Similarly, we can build an SC in two ways by reading *TNTP* and *incidence matrices* files using the `to_simplicial_complex` function.

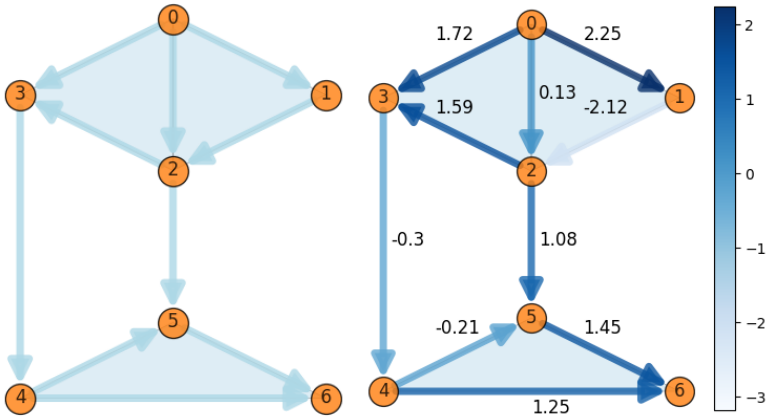
5.3. Plotting

Once the SC is loaded, we can plot it using the `plot` module (see Section 4.2.3) by passing in its coordinates. To plot the SC, we can use the `draw_network` function.

```
from pytspl import SCPlot

# init the plot instance
scplot = SCPlot(
    simplicial_complex=sc,
    coordinates=coordinates
)

# configure figure size
fig, ax = plt.subplots(figsize=(8, 10))

# draw network without edge flow
scplot.draw_network(node_size=400, arrowsize=30, ax=ax)

# draw network with edge flow
scplot.draw_network(edge_flow=flow, node_size=700, arrowsize=30, ax=ax)
```

Listing 5.12: Plot the SC.

The triangles are shaded as light blue to represent the 2-simplices shown in Figure 5.2. The SC can be drawn with or without the edge flow, shown in Figures 5.2b and 5.2a below, respectively.



(a) Without edge flow. **(b)** With edge flow.

Figure 5.2: SC plots with and without edge flow for the *paper* dataset.

To see how a real-world dataset would look like, the *chicago-sketch* transportation network is plotted in Figure 5.3 below.

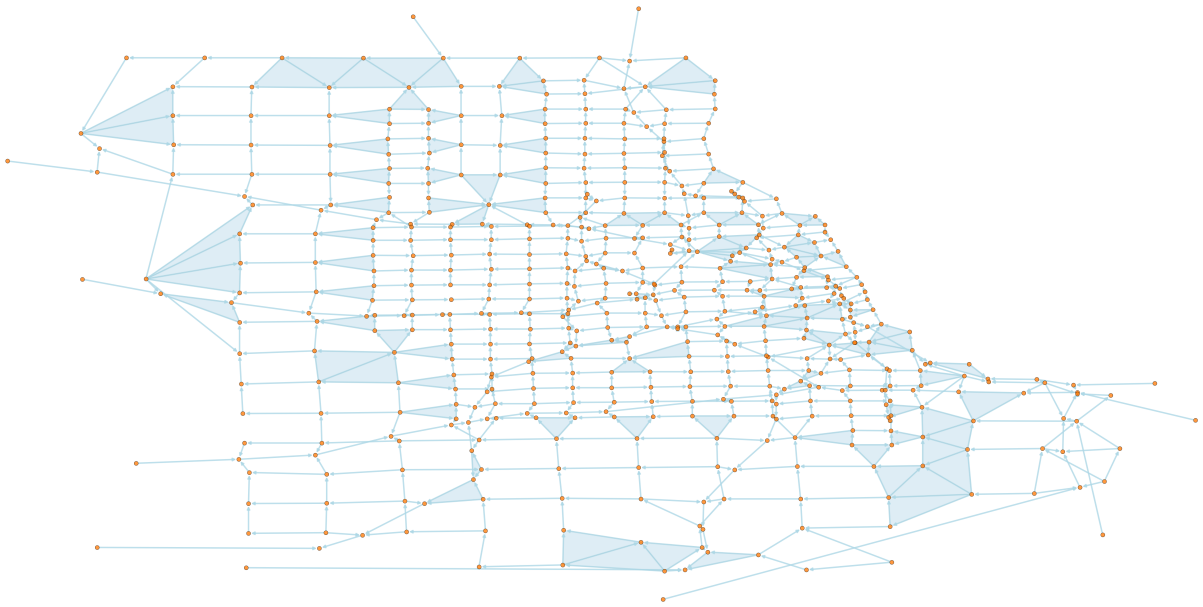


Figure 5.3: SC plot of the chicago-sketch transportation network.

The data defined over an SC may in size depending on the dataset, therefore, the users have the option to draw the network that suits their needs. The user can customize the plot using the parameters of the `draw_network` function, for instance by changing the node size, arrow size, edge width etc. The function supports the parameters to customize the nodes, edges and edge labels straight from the `draw_network` function. Table 5.1 below shows the list of supported parameters.

Table 5.1: List of customizable parameters provided by the `draw_network` function .

Node Parameters	Edge Parameters	Edge Label Parameters
node size	edge color	font size
node color	edge width	font color
node edge colors	arrow size	font weight
font size	edge cmap	offset
font color	edge vmin	alpha
font weight	edge vmax	
cmap	directed	
vmin	alpha	
vmax		
alpha		
margins		

For this, we provide custom plotting functionality to draw the SC by defining the nodes and edges separately using the `draw_sc_nodes`, `draw_sc_edges` and `draw_edge_labels` functions

```
import matplotlib.pyplot as plt

# configure figure size
fig, ax = plt.subplots(1, 1, figsize=(10, 10))

# draw nodes with custom parameters
scplot.draw_sc_nodes(
    node_size=1000,
    font_size=25,
    with_labels=True
)

```

```python
# draw edges with custom parameters
scplot.draw_sc_edges(
    edge_flow=flow,
    edge_width=5,
    arrowsize=30,
    edge_cmap=plt.cm.Blues,
)

# draw edge labels with custom parameters
scplot.draw_edge_labels(
    edge_labels=flow,
    font_size=20
)
```

Listing 5.13: Plot the SC in by plotting the components separately.

5.4. Algebraic Properties

The `SimplicialComplex` data structure class offers the functionality to calculate different algebraic properties of the SC. The list of functions provided by the class is given in Table 4.3. For instance, the incidence matrix and 1-Hodge Laplacian matrix can be calculated as follows

```python
# get the B1 and B2 incidence matrices
B1 = sc.incidence_matrix(rank=1)
B2 = sc.incidence_matrix(rank=2)

# get the 1-Hodge Laplacian matrix
L1 = sc.hodge_laplacian_matrix(rank=1)
```

Listing 5.14: Calculate the incidence matrices and the Hodge Laplacian matrix of the SC.

The output is a `scipy` sparse matrix. Since these matrices are sparse, the `scipy`'s sparse module allows for efficient computations, for instance, matrix multiplication. Once an `scipy` sparse matrix is converted to an array, the output looks like the following for the incidence matrix $\mathbf{B}_1$

```
array([[-1., -1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0., -1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  1., -1., -1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  1.,  0., -1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  1., -1., -1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  0., -1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.]])
```

Listing 5.15: $\mathbf{B}_1$ matrix output.

5.5. Signal Processing

The `decomposition` module (see Section 4.2.4) offers *Hodge decomposition* and *eigendecomposition* in a quick and user-friendly manner. The *eigendecomposition* then can be used to get the SFT of a flow $\mathbf{f}$.

5.5.1. Hodge Decomposition

The *Hodge decomposition* can be used to extract the gradient, curl and harmonic frequencies and analyze the spectral properties of the simplicial filters. For additional functionality of the submodule, please see Table 4.6. To get the harmonic, gradient and curl frequencies of the edge flow, we can use the `get_componenet_flow` function

```python
# create a synthetic flow
synthetic_flow = np.array([0.03, 0.5, 2.38, 0.88, -0.53, -0.52, 1.08, 0.47, -1.17, 0.09])

# get component flow
f_h = sc.get_component_flow(flow=synthetic_flow, component="harmonic")
f_g = sc.get_component_flow(flow=synthetic_flow, component="gradient")
f_c = sc.get_component_flow(flow=synthetic_flow, component="curl")

# gradient component
```

```
print("Gradient:", f_g)
# curl component
print("Curl:", f_c)
# harmonic component
print("Harmonic:", f_h)
```

Listing 5.16: Extract the the three frequency components of the edge flow using Hodge decomposition.

```
Gradient: [0.25, 1.34, 1.32, 1.1, -0.02, 0.03, 0.53, -0.47, -0.78, -0.3]
Curl: [-0.15, -0.7, 0.85, -0.15, -0.85, 0.0, 0.0, 0.58, -0.58, 0.58]
Harmonic: [-0.07, -0.14, 0.21, -0.07, 0.34, -0.55, 0.55, 0.37, 0.18, -0.18]
```

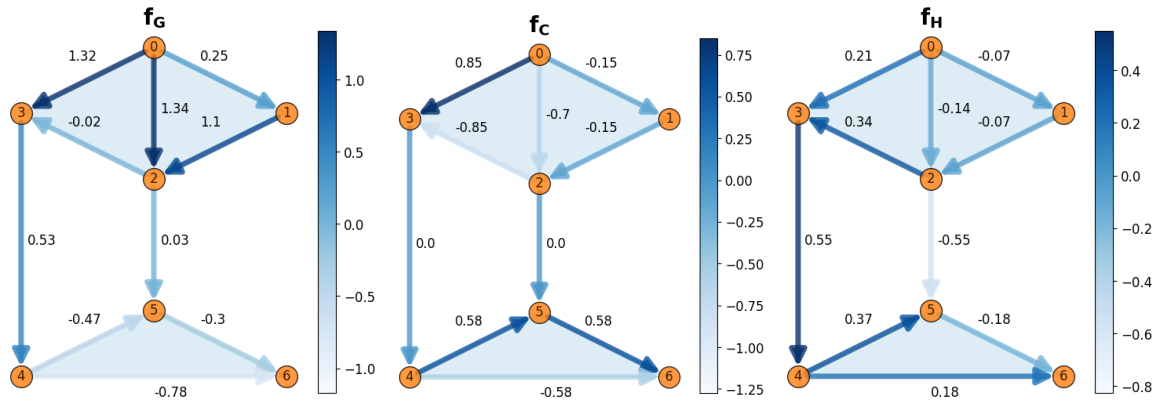
Listing 5.17: The three Hodge components extracted.

The user also has the option to directly plot either a single frequency component or all three components using the `draw_hodge_decomposition` function

```
# create a synthetic flow
synthetic_flow = np.array([0.03, 0.5, 2.38, 0.88, -0.53, -0.52, 1.08, 0.47, -1.17, 0.09])

# plot only a specific component - harmonic
scplot.draw_hodge_decomposition(flow=synthetic_flow, component="harmonic")

# plot all three components
scplot.draw_hodge_decomposition(flow=synthetic_flow)
```

Listing 5.18: Plot the frequency components.

The plots of all three frequency components, gradient, curl and harmonic, are shown in Figure 5.4 below.



(a) Gradient component. **(b)** Curl component. **(c)** Harmonic component.

Figure 5.4: The gradient, curl and harmonic component of the edge flow.

5.5.2. Eigendecomposition

The functionality offered by the `eigendecomposition` module is listed in Table 4.5. As an example, the gradient, curl and harmonic eigenvectors with their respective eigenvalues can be extracted using the `get_component_eigenpair` function

```
# harmonic
u_h, eigenvals_h = sc.get_component_eigenpair(component="harmonic")
# curl
u_c, eigenvals_c = sc.get_component_eigenpair(component="curl")
# gradient
u_g, eigenvals_g = sc.get_component_eigenpair(component="gradient")
```

Listing 5.19: Extract the eigenvectors and eigenvalues of the three components.

Using the `draw_eigenvectors` function, we can plot the eigenvectors with their respective eigenvalues. The users have the option to plot all the eigenvectors or pass indices of the eigenvalues they want to plot as illustrated below

```python
# plot curl eigenvectors
scplot.draw_eigenvectors(component="curl")

# plot only the selected curl eigenvectors
eig_vec_indices = np.array([0, 2])
scplot.draw_eigenvectors(component="curl", eigenvector_indices=eig_vec_indices)
```

Listing 5.20: Plot the eigenvectors and eigenvalues of a component.

The first and third eigenvectors and their associated eigenvalues of the curl component is shown in Figure 5.5 below.

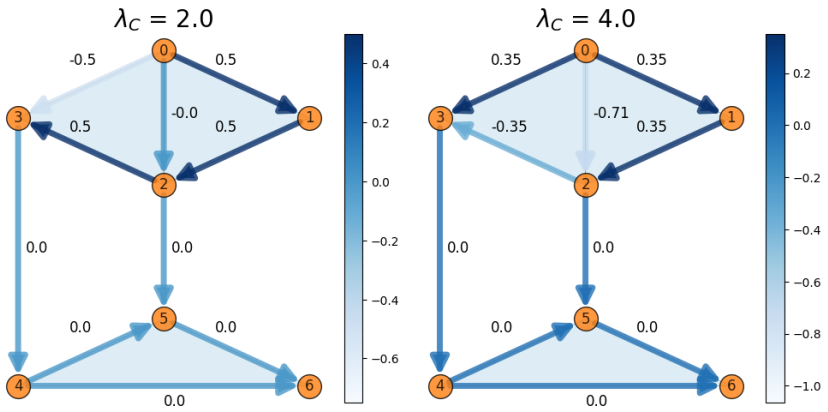


Figure 5.5: The first and third eigenvectors in the curl space with the corresponding curl frequencies.

5.5.3. Simplicial Fourier Transform

The SFT (see Section 2.2.2) can be written in terms of *simplical embeddings* as discussed in Section 2.2.2. The frequency embeddings can be calculated using the `get_simplicial_embeddings` function in the following way

```python
# define a synthetic flow
synthetic_flow = [0.03, 0.5, 2.38, 0.88, -0.53, -0.52, 1.08, 0.47, -1.17, 0.09]

# get simplicial embeddings
f_tilda_h, f_tilda_c, f_tilda_g = sc.get_simplicial_embeddings(flow=synthetic_flow)

print("embedding_h:", f_tilda_h)
print("embedding_g:", f_tilda_g)
print("embedding_c:", f_tilda_c)
```

Listing 5.21: Get the simplicial embeddings of flow $\mathbf{f}$.

```
embedding_h: [-1.00084785]
embedding_g: [-1.00061494, -1.00127703, 1.00173495 -1.00287539  0.99531105, 1.00412064]
embedding_c: [-1, 0.99881597, 0.99702056]
```

Listing 5.22: The extracted simplicial embeddings from flow $\mathbf{f}$.

5.6. Filters

The tutorials in this section illustrate the functionality of simplicial shifting, simplicial convolution filters (see Section 2.2.3) and the simplicial trend filter (see Section 2.2.4).

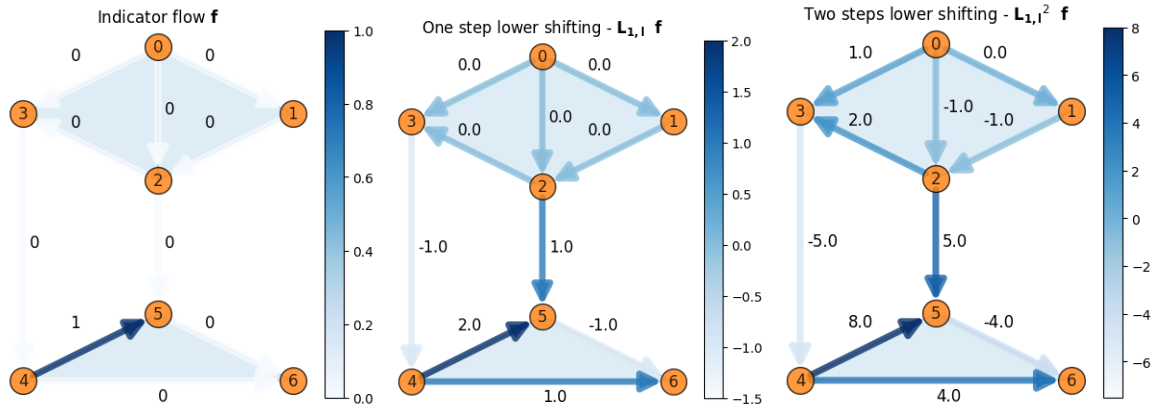
5.6.1. Simplicial Shifting

Simplicial shifting allows k-step shifts on a signal over the lower and upper neighbourhoods. In other words, the edge collects the flows from its lower and upper neighbours k-hops away and updates its information. The user can apply k-step lower and upper shifting using the `apply_lower_shifting` and `apply_upper_shifting` functions, respectively.

```python
# create a synthetic flow
synthetic_flow = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]

# one step lower shifting
one_step = sc.apply_lower_shifting(synthetic_flow, steps=1)

# two step lower shifting
two_step = sc.apply_lower_shifting(synthetic_flow, steps=2)
```

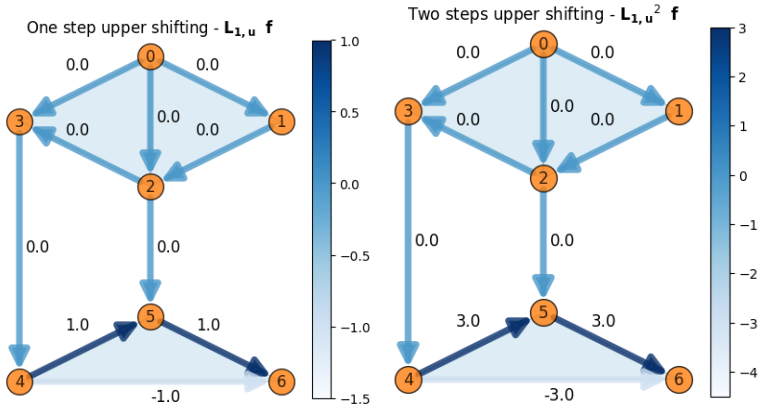
Listing 5.23: Applying lower and upper k-step simplicial shifting on an indicator flow $\mathbf{f}$.

An example of lower and upper k-step shifting is illustrated in Figure 5.6 and 5.7 below, respectively.



(a) An edge flow indicator $\mathbf{f}$ of edge (4, 5). **(b)** One-step lower shifting $\mathbf{L}_{1,\mathrm{l}}\mathbf{f}$. **(c)** Two-step lower shifting $\mathbf{L}_{1,\mathrm{l}}{}^2\mathbf{f}$.

Figure 5.6: One- and two-step lower shifting on an edge flow indicator $\mathbf{f}$.



(a) One-step upper shifting $\mathbf{L}_{1,\mathrm{u}}\mathbf{f}$. **(b)** Two-step upper shifting $\mathbf{L}_{1,\mathrm{u}}{}^2\mathbf{f}$.

Figure 5.7: One- and two-step upper shifting on an edge flow indicator $\mathbf{f}$.

The user can also apply k-step shifting using the `apply_k_step_shifting` function

```python
# apply two-step shifting
k = 2
flow = sc.apply_k_step_shifting(synthetic_flow)
```

Listing 5.24: Applying two-step simplicial shifting on an indicator flow $\mathbf{f}$.

The resultant edge flow after applying a two-step shift is illustrated in Figure 5.8 below

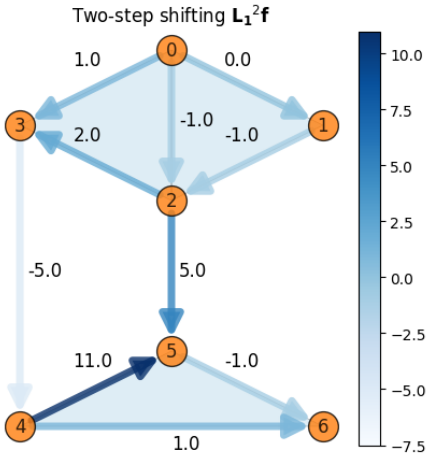


Figure 5.8: Two-step simplicial shifting $\mathbf{L_1}^2\mathbf{f} = \mathbf{L_{1,l}}^2\mathbf{f} + \mathbf{L_{1,u}}^2\mathbf{f}$ on an indicator flow $\mathbf{f}$, as the sum of 5.6c and 5.7b

5.6.2. Simplicial Convolutional Filters

The `filters` module offers functionality for three simplicial convolution filters (see Section 2.2.3), namely *LS-Filter*, *Grid-Based Filter* and *Chebyshev Polynomial Filter* Designs (see Sections 2.2.3, 2.2.3, 2.2.3).

Least-Squares Filter Design. The LS filter design offers subcomponent extraction using type I and II filters. In type I filter, we have $L_1 = L_2$ and $\alpha = \beta$. The type I filter can be applied using the `subcomponent_component_extraction_type_one` function

```python
from pytspl.filters import LSFilterDesign

# init LS filter
filter_size = 16
component = "gradient"
ls_filter = LSFilterDesign(simplicial_complex=sc)

# subcomponent extraction using type I filter
ls_filter.subcomponent_extraction_type_one(
    f=f,
    component=component,
    L=filter_size
)
```

Listing 5.25: Subcompoenent extraction using type I filter.

The type I filter does not differentiate between lower and upper adjacencies, resulting in a loss of expressive power. To address this limitation, we propose the type II filter, which treats lower and upper adjacencies separately. This approach provides greater flexibility and control over the frequency response.

Subcomponent extraction using the type II filter, where we have L_1 != L_2 and α != β can be applied using the `subcomponent_component_extraction_type_two` function as follows

```python
# subcomponent extraction using type II filter
ls_filter.subcomponent_extraction_type_two(
    f=f,
    component=component,
    L=filter_size
)
```

Listing 5.26: Subcomponent extraction using type II filter.

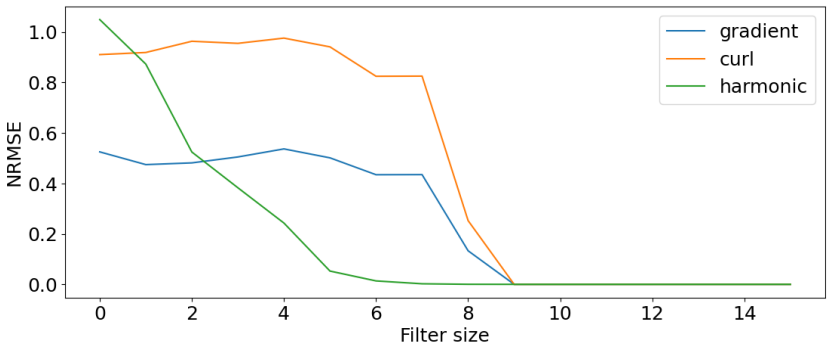
The *history* of the filter stores the results of the built filter $\mathbf{H}_1$ and can be retrieved using the class attribute `history`

```python
# retrive all history
history = ls_filter.history
```

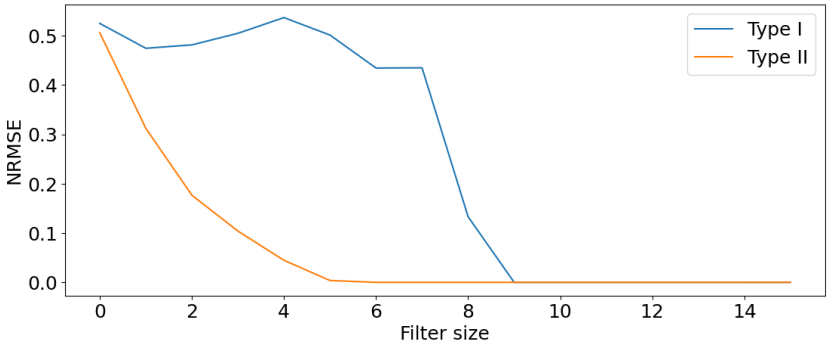
```

# retrive history components
H = history["filter"]
f_estimated = history["f_estimated"]
f_responses = history["frequency_responses"]
extracted_comp_error = history["extracted_component_error"]
```

Listing 5.27: Retriving the history of the built filter $\mathbf{H}_1$.

where *filter* is the built filter $\mathbf{H}_1$, *f_estimated* is the estimated frequency of the extracted component, *frequency_responses* are the frequency response for each filter size and *extracted_component_error* is the *normalized root mean squared error* (NRMSE) for each filter size. The filter error can be plotted over each filter size as shown in Figure 5.9 below. The extraction improves as the filter size increases as shown in Figure 5.9a. As seen in Figure 5.9b, for the gradient component, setting $L_1 = L_2$ and $\alpha = \beta$ worsens the performance because of the limited expressive power.



(a) Performance of type I filter for the gradient, curl and harmonic component. The extraction improves as the filter size increases.



(b) Performance of type I and II filters for the gradient component. Type I filter worsens the performance by setting $L_1 = L_2$ and $\alpha = \beta$.

Figure 5.9: Subcomponent extraction performance by filter $\mathbf{H}_1$

Grid-Based Filter Design: Similar to LS filter design, the grid-based filter design offers type I and II filters, however, it aims to match the desired frequency response in a continuous interval. In this case, the exact frequencies lie such that the eigenvector and eigenvalue computation of $\mathbf{L}_1$ can be avoided. We can denoise with low-pass filter $\mathbf{H_P}$ using the `denoise` function as follows

```
# denoise
gbf.denoise(
    f=f,
    f_true=f0,
    p_choice="L1",
    L=filter_size,
    mu_vals=[0.5]
)
```

Listing 5.28: Denoising with the low-pass filter $\mathbf{H}_P$.

where *f* is the noisy flow, *f_true* is the original flow, *p_choice* is the shift matrix choice used for the filter and *mu* is regularizer. Futhermore, we can denoise with gradient based simplicial filter $\mathbf{H}_1$ using the `subcomponent_extraction` function as follows

```
from pytspl.filters import GridBasedFilterDesign

# init filter
filter_size = 10
gb_filter = GridBasedFilterDesign(simplicial_complex=sc)

# subcomponent extraction
gb_filter.subcomponent_extraction(
    f=f,
    f_true=f0,
    p_choice="L1L",
    L=filter_size
)
```

Listing 5.29: Denoising with the gradient based simplicial filter $\mathbf{H}_1$.

The frequency responses of the denoising filter $\mathbf{H_P} = (\mathbf{I} + 0.5\mathbf{L}_1)^{-1}$ based on the grid filter design is shown in Figure 5.10 below.

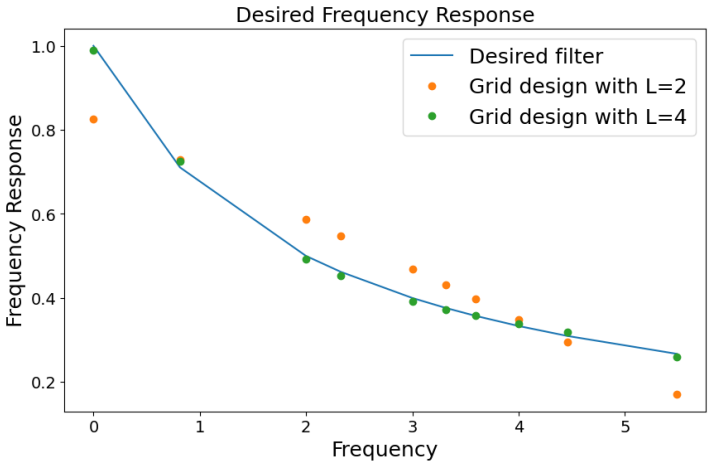
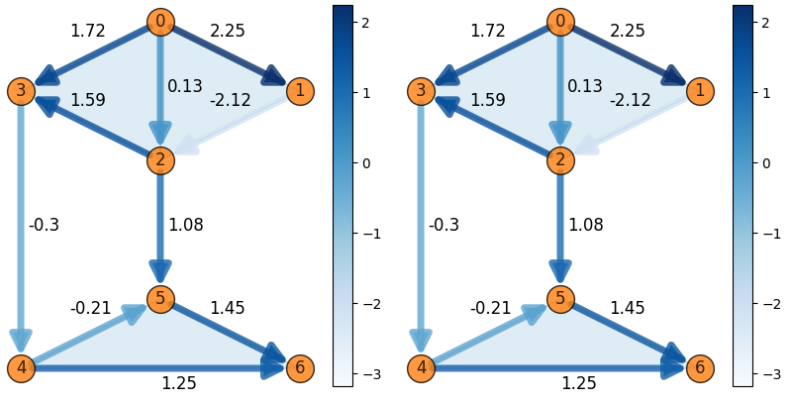


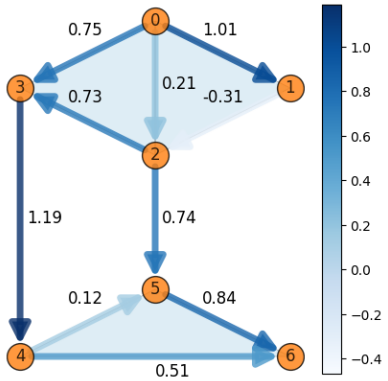
Figure 5.10: Frequency responses of the denoising filter $\mathbf{H_P}$ based on the grid filter design.

We can see the plotted results of denoising in Figure 5.11 below. Figure 5.11a represents an edge flow $\mathbf{f}_0$ induced by a node signal with a flat spectrum. Figure 5.11b is the noisy observation $\mathbf{f}$. Figure 5.11c and 5.11d is the denoising with the low-pass filter $\mathbf{H}_P$ with $\mathbf{P} = \mathbf{L}_1$ and $\mathbf{P} = \mathbf{L}_{1,l}$, with error $e = 0.70$ and $e = 0.73$, respectively. In Figure 5.11e, we denoise by the gradient based simplicial filter $\mathbf{H}_1$ with an order $L_1 = L_2 = 4$ that yields a much better performance with error $e = 0.39$.

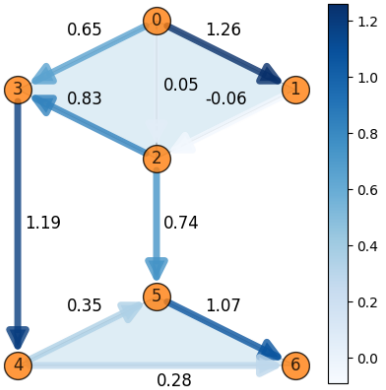


(a) Noise-free flow $\mathbf{f_0}$

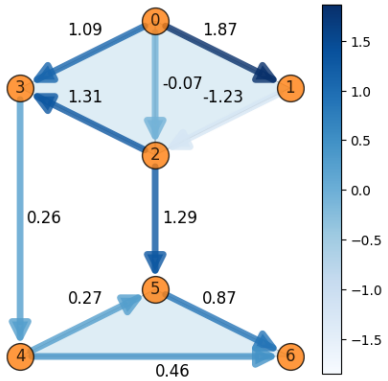
(b) Noisy flow $\mathbf{f}$



(c) $\mathbf{H} = (\mathbf{I} + 0.5\mathbf{L}_1)^{-1}$



(d) $\mathbf{H} = (\mathbf{I} + 0.5\mathbf{L}_{1,l})^{-1}$



(e) $\mathbf{H}_1, \boldsymbol{\alpha} = \boldsymbol{\beta}$

Figure 5.11: Gradient flow denoising.

Chebyshev polynomial filter design. The goal of using Chebyshev polynomial based filter design is to tackle the numerical instability issue and design a filter for large networks. For this filter design, we will use a large network to demonstrate its capabilities. Specifically, we will use the Chicago road network dataset with 546 nodes, 1088 edges and 112 triangles. We perform the gradient component extraction of the measured traffic flow via the filter $\mathbf{H_1}$ built on the lower Hodge Laplacian $\mathbf{L}_{1,l}$ using the `apply` function as follows

```
from pytspl import load_dataset, SCPlot
from pytspl.filters import ChebyshevFilterDesign

# load chicago-sketch dataset
sc, coordinates, flow = load_dataset("chicago-sketch")

```

```
# convert the flow to an numpy array
flow = np.asarray(list(flow.values()))

# init filter
chebyshev_filter = ChebyshevFilterDesign(simplicial_complex=sc)

# apply filter
filter_size = 50
chebyshev_filter.apply(
    f=flow,
    p_choice="L1L",
    component="gradient",
    L=filter_size,
    cut_off_frequency=0.01,
    steep=100,
    n=100
)
```

Listing 5.30: Subcomponent extraction using the Chebyshev polynomial based filter design.

where the order of the Chebyshev polynomial is 50.

Next, we can plot the filter error and the extracted component error as follows

```
# plot errors
plt.plot(chebyshev_filter.history["extracted_component_error"])
plt.plot(chebyshev_filter.history["filter_error"])
```

Listing 5.31: Plot filter error and extracted component error.

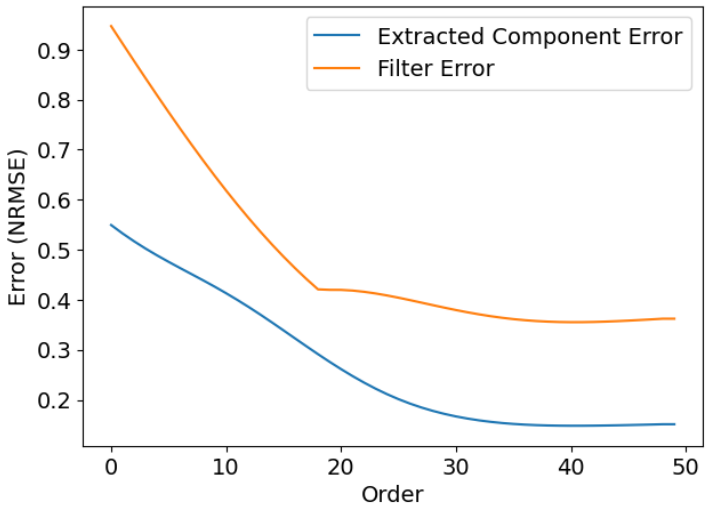


Figure 5.12: The approximation errors of Chebyshev filters of different orders and the extracted gradient component.

To compare the true component with the extracted component, we can plot the frequency response using the `plot_frequency_response_approx` function as follows

```
chebyshev_filter.plot_frequency_response_approx(flow=flow, component="gradient")
```

Listing 5.32: Plot frequency approximation of the extracted component.

As shown in Figure 5.13 below, the Chebyshev polynomial design effectively extracts the gradient component.

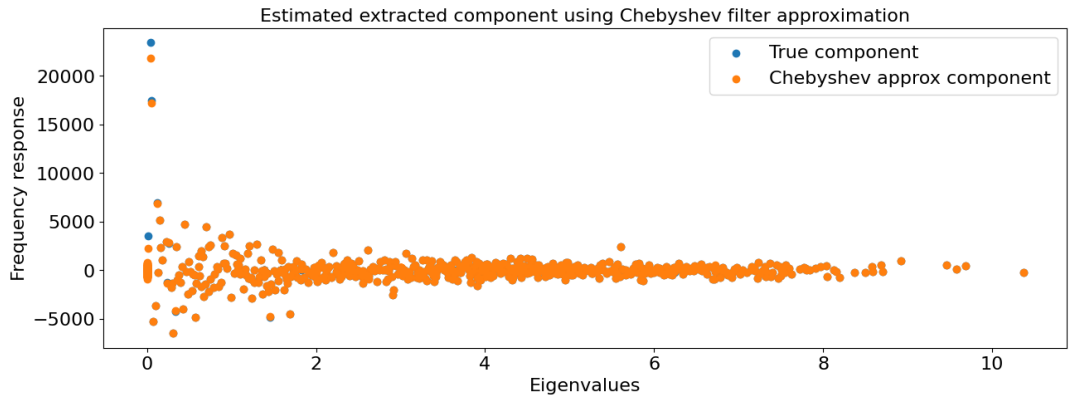


Figure 5.13: Frequency response of the true and extracted gradient component using the Chebyshev polynomial design.

To compare the three filter designs, we applied them to the Chicago road network. For the LS-based filter design, we set a filter order of 10. Setting a low filter order avoids the ill-conditioning. For the grid-based filter design, we uniformly sampled 100 points in the interval $[0, \lambda_{G,\text{max}}]$ with $\lambda_{G,\text{max}} = 10.8$ approximated using the power-iteration algorithm with steps = 50. The cut-off frequency $\lambda_0 = 0.01$ and the steep k = 100 for the logistic function in the Chebyshev polynomial design.

As seen in Figure 5.14 below, the Chebyshev polynomial of order 50 only has a couple of frequencies smaller than 0.9 at the smallest gradient frequency. The remaining frequencies can preserve the gradient component well. The LS-based and grid-based filter designs have a poorer performance, especially at small gradient frequencies.

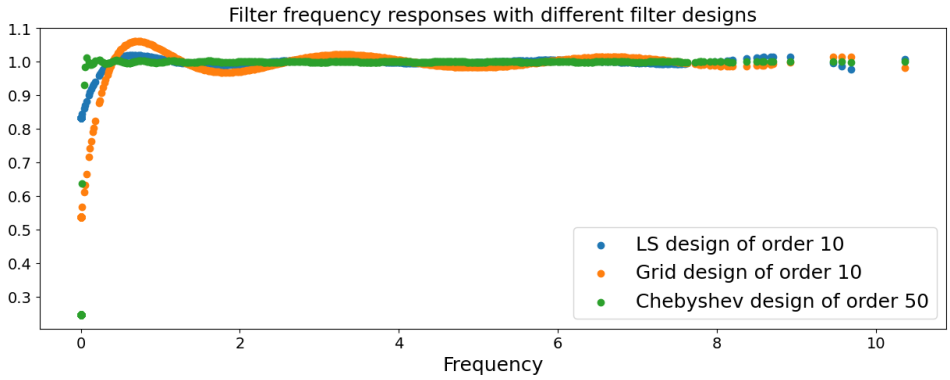


Figure 5.14: Filter frequency responses for all three filter designs.

Finally, we can calculate the SFT of the extracted gradient component and compare it with the grid-based filter design. The comparison is plotted in 5.15 below. As we can see, the Chebyshev polynomial filter has a good extraction ability and performs well at very small frequencies where the grid-based design fails.

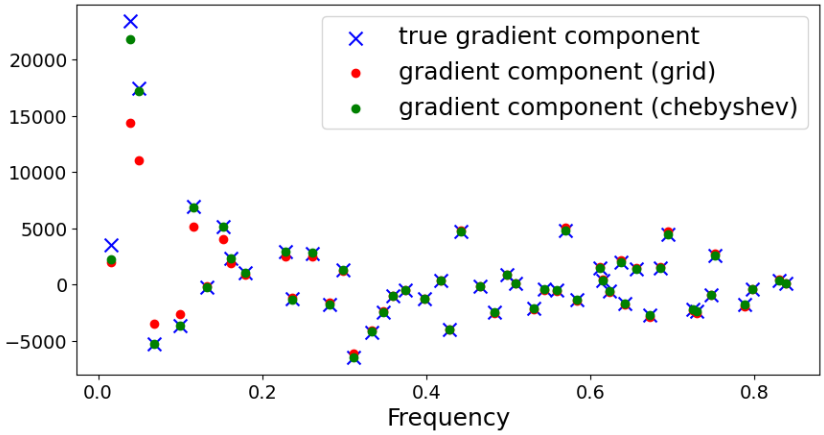


Figure 5.15: SFT of the extracted gradient component.

5.6.3. Simplicial Trend Filter

Reconstructing simplicial signals of a network from (partial) noisy observation can be achieved via simplicial trend filter (STF) by regularizing the total divergence and the curl as discussed in Section 2.2.4. This process is also known as simplicial signal denoising and interpolation. For this pedagogical example, we will primarily focus on the edge flow case. To initialize an STF and get the divergence and curl, the users can use the `get_divergence_flow` and `get_curl_flow` functions respectively

```
from pytspl.filters import SimplicialTrendFilter

#init STF
trend_filter = SimplicialTrendFilter(simplicial_complex=sc)

# get the divergence and curl flow
print("Divergence:", trend_filter.get_divergence_flow(flow))
print("Curl:", trend_filter.get_curl_flow(flow))
```

Listing 5.33: Initialize STF and get the divergence and curl flow.

```
Divergence: 1.4141
Curl: 220.6808
```

Listing 5.34: Output: divergence and curl flow.

Denoising: We denoise a noisy signal by using the regularizers ℓ_1 or ℓ_2 (see Sections 2.2.4 and 2.2.4). To denoise using a ℓ_1 regularizer, we can use the `denoising_l1_regularizer`

```
# define parameters
num_realizations = 50
snr_db = np.arange(-12, 12.5, 2)

# order of the STF
order = 0

# denoise using l1 regularizer
trend_filter.denoising_l1_regularizer(
    flow=flow,
    order=order,
    component="divergence",
    num_realizations=num_realizations,
    snr_db=snr_db,
    regularization=0.5
)
```

Listing 5.35: Denoise using the ℓ_1 regularizer.

where the *order* of the STF and the component name determine the operators $\Delta_\ell^{(p)}$ and $\Delta_u^{(p)}$ as given in equation 2.53 and 2.54, respectively.

```
SNR: 0.0631 dB - l1 error: 0.2922 - corr: 0.9591
SNR: 1.0 dB - l1 error: 0.0197 - corr: 0.9998
SNR: 15.8489 dB - l1 error: 0.0032 - corr: 1.0
```

Listing 5.36: Output: denoising with ℓ_1 regularizer using STF.

Similarly, we can denoise with ℓ_2 regularizer using the `denoising_l2_regularizer` function

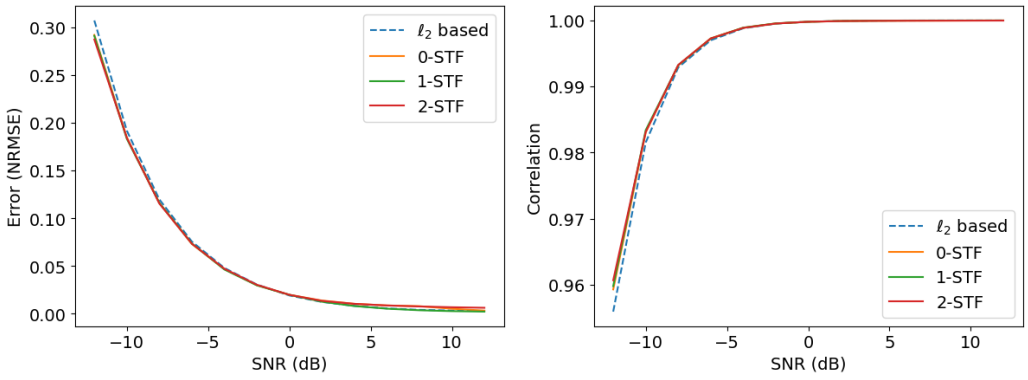
```
# define parameters
num_realizations = 50
snr_db = np.arange(-12, 12.5, 2)

# denoise using l2 regularizer
trend_filter.denoising_l2_regularizer(
    flow=flow,
    component="divergence",
    num_realizations=num_realizations,
    snr_db=snr_db,
    mu=0.5
)
```

Listing 5.37: Denoise using the ℓ_2 regularizer.

```
SNR: 0.0631 dB - error noisy: 0.1185 - l2 error: 0.3047 - corr:  0.9558
SNR: 1.0 dB - error noisy: 0.0112 - l2 error: 0.0193 - corr:  0.9998
SNR: 15.8489 dB - error noisy: 0.0014 - l2 error: 0.003 - corr:  1.0
```

Listing 5.38: Output: denoising with ℓ_2 regularizer using STF.

Next, we can plot the errors and correlations for the ℓ_2-based regularizer, 0-STF, 1-STF and 2-STF. The errors and correlations are shown in Figures 5.16a and 5.16b below.



(a) Denoising NRMSE performance.

(b) Correlation performance.

Figure 5.16: Reconstruction performance for the lastfm-1k-artist dataset.

We have set $\beta = 0$ to only penalize the divergence and the regularization parameter μ was set to 0.5. For denoising with SNRs ranging from [-12dB, 12dB], we observe that the STF performs comparably with the ℓ_2 regularizer in terms of NRMSE.

Interpolation: For interpolation tasks, the users can use the `interpolation_l1_regularizer` function as follows

```
# define parameters
num_realizations = 50
ratio = np.arange(0.05, 1.05, 0.3)

# order of the STF
order = 0

# interpolate
trend_filter.interpolation_l1_regularizer(
```

```
    flow=flow,
    order=order,
    regularization=0.5,
    component="divergence",
    ratio=ratio,
    num_realizations=num_realizations
)
```

Listing 5.39: Interpolate using the ℓ_1 regularize.

```
Ratio: 0.05 - error: 0.9549 - corr: 0.2814
Ratio: 0.35 - error: 0.6522 - corr: 0.7502
Ratio: 0.65 - error: 0.3355 - corr: 0.9381
Ratio: 0.95 - error: 0.0151 - corr: 0.9996
```

Listing 5.40: Output: interpolating with ℓ_1 regularizer using STF.

5.7. Hodge-compositional Edge Gaussian Processes

Gaussian processes (GPs) are used for modelling functions defined over the edge set of a simplicial 2-complex, a structure similar to a graph in which edges may form triangular faces. This method is aimed at learning flow-type data on networks, where edge flows can be described by discrete divergence and curl [37]. More details in Section 2.2.5.

To get started, we can load the *forex exchange* dataset and initialize the `HodgeGPTrainer` class as follows

```
from pytspl import load_dataset
from pytspl.hogde_gp import HodgeGPTrainer

# read dataset
sc, _, flow = load_dataset("forex")

# get the y from flow dict
y = np.fromiter(flow.values(), dtype=float)

# init trainer
hogde_gp = HodgeGPTrainer(sc=sc, y=y)
```

Listing 5.41: Initilialize HodgeGPTrainer.

Next, we split our data into training and testing data using the `train_test_split` function as follows

```
# test split ratio
train_ratio = 0.2

# normalize data
data_normalization = False

# split data
x_train, y_train, x_test, y_test, x, y  = hogde_gp.train_test_split(
    train_ratio=train_ratio,
    data_normalization=data_normalization,
    seed=4
)
```

Listing 5.42: Split the data into training and test set.

In the next step, we decide the kernel type, likelihood function and model used in training. We can retrieve the eigenpairs using the `get_eigenpairs` function as follows

```
from pytspl.hogde_gp.kernel_serializer import KernelSerializer
import gpytorch
from pytspl.hogde_gp import ExactGPModel

# set the kernel parameters
kernel_type = "matern" # kernel type
data_name = "forex" # dataset name

```

```
# get the eigenpairs
eigenpairs = hogde_gp.get_eigenpairs()

kernel = KernelSerializer().serialize(
    eigenpairs=eigenpairs,
    kernel_type=kernel_type,
    data_name=data_name
)

# choose a likelihood function
likelihood = gpytorch.likelihoods.GaussianLikelihood()

# choose a model
model = ExactGPModel(
    x_train,
    y_train,
    likelihood,
    kernel,
    mean_function=None
)

# view model architecture
print(model)
```

Listing 5.43: Choose parameters for training and build the model.

```
ExactGPModel(
  (likelihood): GaussianLikelihood(
    (noise_covar): HomoskedasticNoise(
      (raw_noise_constraint): GreaterThan(1.000E-04)
    )
  )
  (mean_module): ConstantMean()
  (covar_module): ScaleKernel(
    (base_kernel): MaternKernelForex(
      (raw_kappa_down_constraint): Positive()
      (raw_kappa_up_constraint): Positive()
      (raw_kappa_constraint): Positive()
      (raw_mu_constraint): Positive()
      (raw_mu_down_constraint): Positive()
      (raw_mu_up_constraint): Positive()
      (raw_h_constraint): Positive()
      (raw_h_down_constraint): Positive()
      (raw_h_up_constraint): Positive()
    )
    (raw_outputscale_constraint): Positive()
  )
)
```

Listing 5.44: Output: model architecture.

We can train the models with the training data using the `train` function as follows

```
# train models
model.train()
likelihood.train()
hogde_gp.train(model, likelihood, x_train, y_train)
```

Listing 5.45: Training the model.

```
Iteration 1/1000 - Loss: 5.387
Iteration 2/1000 - Loss: 4.940
Iteration 3/1000 - Loss: 4.554
Iteration 4/1000 - Loss: 4.221
Iteration 5/1000 - Loss: 3.936
Iteration 6/1000 - Loss: 3.691
Iteration 7/1000 - Loss: 3.482
...
```

Listing 5.46: Example training output.

Finally, we can evaluate the model and make predictions using the `predict` function as follows using the test data

```
# make predictions
hogde_gp.predict(
    model=model,
    likelihood=likelihood,
    x_test=x_test,
    y_test=y_test
)
```

Listing 5.47: Evaluating the model and making predictions using the test dataset with the trained model.

```
# metrics
Test MAE: 0.01143
Test MSE: 0.0019
Test R2: 0.9997
Test MLSS: -3.0126
Test NLPD: -3.5197

# type
MultivariateNormal(loc: torch.Size([168]))
```

Listing 5.48: Example prediction output.

To obtain the model's learned parameters, users can use the `get_model_parameters` function as follows

```
# get the trained model parameters
hogde_gp.get_model_parameters()
```

Listing 5.49: Obtain model's learned parameters.

```
{'raw_noise': 0.00010002510680351406,
 'raw_mu': 0.6931471824645996,
 'raw_mu_down': 0.6705738306045532,
 'raw_mu_up': 2.148306369781494,
 'raw_kappa': 0.6931471824645996,
 'raw_kappa_down': 39.97739791870117,
 'raw_kappa_up': 0.02517639473080635,
 'raw_h': 0.6931471824645996,
 'raw_h_down': 31.798500061035156,
 'raw_h_up': 0.053553808480501175,
 'raw_outputscale': 11.325119972229004
 }
```

Listing 5.50: Output: model's learned parameters.

To build the Matérn kernel K_1 and get the variance, we can use the `build_matern_kernel`. Figure 5.17 below shows the plot of the forex dataset with a training ratio of 20%.

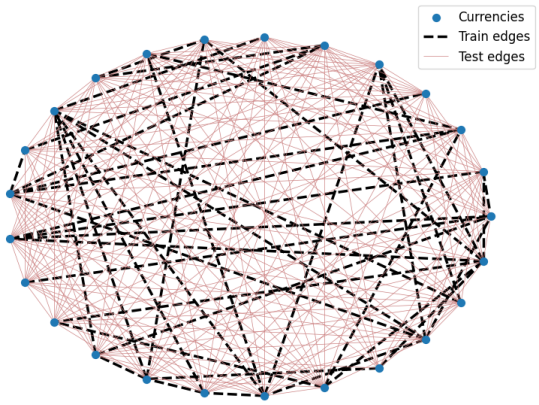
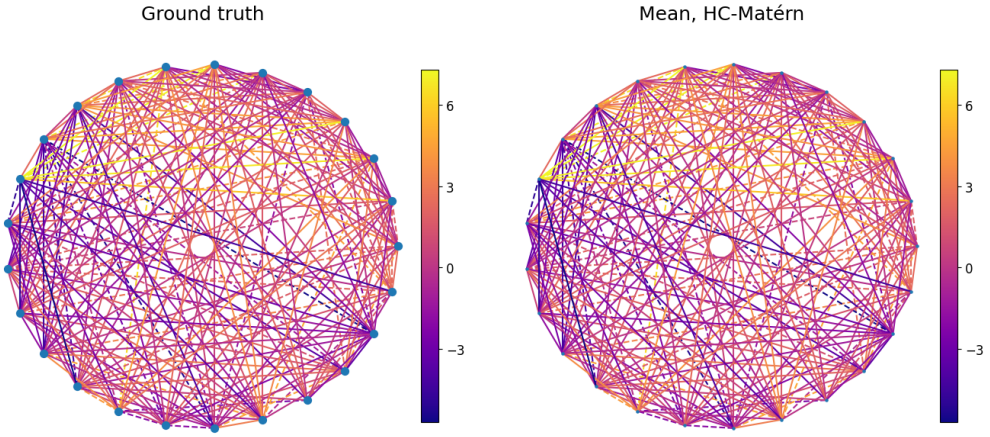


Figure 5.17: Forex dataset with a training ration of 20% where the the dashed (solid) edges are used for training (testing).

Next, we interpolate the forex market using the test set. The ground truth is shown in Figure 5.18a and the prediction is shown in Figure 5.18b.



(a) Ground truth. **(b)** Mean, HC Matérn

Figure 5.18: Interpolating the forex dataset with the HC Matérn.

6

Configuration Management

In this chapter, we dive into the crucial part of software development, the configure management. This includes version control, quality assurance, continuous integration and deployment (CI/CD) pipelines and documentation. Several tools were used to ensure the library conforms to certain standards and follows best practices.

6.1. Version Control

Version control is tracking and managing changes in a software repository. It keeps track of every modification in the code base. If a mistake is made, the developers can always revert to the last version and compare the changes. This helps them debug the code and fix the error, minimizing the delivery time of the software.

`GitHub`[1] is an online version control platform, that was used to develop the `PyTSPL` library. Its widespread adoption in the software development community ensures familiarity and ease of use for developers. Moreover, `GitHub` facilitates seamless collaboration through features like pull requests, issue tracking, and code review, enabling efficient communication and coordination among contributors. Additionally, its integration with CI/CD (`GitHub Actions`) tools streamlines the development process, enhancing productivity and code quality [49].

6.2. Dependency Management

Dependency management is the practice of identifying, resolving, and patching dependencies within a package. A dependency is an external package that your package relies on to function as intended. Dependencies can range from using a small function to a whole module from an external package. Managing these dependencies is crucial for your package for several reasons. Dependencies shorten your development time by utilizing existing code and avoid reinventing the wheel. Managing these dependencies ensures your package runs consistently across different environments without any issues. It prevents conflicts between different versions of existing packages and mitigates introducing bugs in your package due to incompatible versions between dependencies. Additionally, keeping dependencies up to date is crucial, as existing packages regularly release a new version by patching any security vulnerabilities. Updating the dependencies protects your package against potential attacks. A good practice is to automate dependency management by utilizing tools and processes, which significantly reduces manual effort and avoids any human error. The automation includes regularly checking for updates, resolving conflicts and applying patches [50].

`Poetry`[2] is a modern dependency management package for Python. It replaces multiple files like *setup.py, setup.cfg*, and *requirements.txt*. It has gained popularity due to its simplicity and versatility. `Poetry` offers a simpler and user-friendly way to handle dependency management compared to

[1] https://github.com/
[2] https://python-poetry.org/

pip or pipenv. Pip requires manual configuration to be used at maximum efficiency [51]. Poetry has an advanced dependency resolution mechanism and can handle dependency conflicts more effectively. It has a built-in virtual environment which is easily configurable, whereas pip requires venv to be installed and configured. The tool also facilitates the publishing of packages to Python Package Index (PyPI or private repositories using simple commands. Moreover, Poetry has a large community that actively develops and maintains the package by constantly fixing bug fixes and updating the functionality. Also, Poetry can be used with other tools, such as linters and pre-commit hooks by configuring the *pyproject.toml* file without the need for any external tools.

6.3. Quality Assurance

In this section, we dive into the crucial part of software development, Quality Assurance (QA) within the context of the developed library. QA is essential for software development and ensures that the final product meets the specified requirements and adheres to the highest performance, reliability, and usability standards. QA is a systematic and proactive approach that involves ensuring the software meets quality standards and user expectations. It comprises processes, techniques, and activities designed to prevent, detect, and correct defects, errors, or inconsistencies, ensuring optimal performance and reliability [52]. Through rigorous testing and static code analysis, QA identifies and fixes defects. By implementing strong QA processes, teams improve product reliability, performance, and user satisfaction, building trust in the software.

6.3.1. Testing

Software testing is the process of verifying and validating to see if the software is doing what it is supposed to do. It prevents future bugs and improves the performance. Delivering software with defects can significantly damage a brand's reputation, leading to frustrated users and potential loss of users. When the software fails to perform as intended, users lose trust and may seek alternative solutions elsewhere [53]. For quality assurance purposes, the testing of PyTSPL was started early. By testing at an early stage of software development, we identify bugs early so they can be fixed before they cause even larger bugs.

PyTest[3] is a Python framework to write and execute tests, that can scale and support complex functional testing of libraries. This framework was used to write tests for the PyTSPL library. Once the tests are run, we can see the test coverage. The test coverage measures the amount of testing performed by a set of tests. It informs us which lines of code are executed (tested) and which require additional tests. Identifying a quantitative measure of test coverage helps us indirectly measure the quality of the code. After extensive research, PyTest was chosen as the framework used for testing. It is simple to use, can run a specific subset of tests, and is open-source, which means it has robust community-driven support [54].

The test coverage of PyTSPL was above 90 percent at the end of the thesis, which is considered a high test coverage for a software library. This ensures that the majority of the codebase is tested and the reliability of the library increases. It helps identify and fix bugs in the early stage of the development process, leading to a more stable application. Additionally, maintaining and expanding the code becomes much easier. The future contributors of the library can refactor or add new features with confidence, knowing that the existing functionality is well-tested and any new potential issues will be quickly identified by the test suite.

Codecov[4] was used to report the test coverage after each push to the master branch. It is an all-in-one code coverage reporting solution that helps to visually analyze the code coverage of the codebase and improve it over time. Each time it is updated, it issues a badge which is displayed in the README of the repository as shown in Figure 6.1 below.

Figure 6.1: Codecov code coverage badge.

[3] https://docs.pytest.org
[4] https://about.codecov.io/

6.3.2. Static Code Analysis

Static code analysis is a method to debug code by examining it without actually executing the program. The process provides an understanding of the code structure to ensure the code adheres to industry standards. The method is good at finding issues such as programming errors, coding standing violations, syntax violations and security vulnerabilities [55].

A way to automatically analyze your source code for programmatic and stylistic errors is by using linters. `Flake8`[5] is a well-know linter used for `Python` projects. It is a highly extensible tool and can be integrated easily into most `Python` environments. It applies code style checking, syntax checking, and code complexity checking. The tool ensures you adhere to the best development practices in your code base. A *.flake8* file allows you to configure the tool's behaviour to your specific requirements by defining which rules should be enforced and which ones can be ignored. This configuration file empowers you to finely tune the linting process according to your project's standards and preferences [56].

6.4. CI/CD Pipelines

Continuous integration and development (CI/CD) pipelines are automated workflows that streamline the process of integrating code changes into a repository, testing those changes, and deploying them into development, test and production environments. It's a practice followed by developer teams to improve software delivery through the software development life cycle via automation. Automating CI/CD pipelines enables the development team to produce code of higher quality in a faster and more secure way.

Continuous Integration (CI): CI is the practice of integrating code changes regularly into your main branch, triggering an automated build and test process each time to verify that the new code doesn't introduce bugs into the code base. With CI, bugs and security vulnerabilities can be identified and fixed at an earlier stage in the development process [57]. Common stages for the CI pipeline include build, static code analysis and running the automated tests.

`Github Actions` is a CI/CD platform that allows you to automate build, test and deployment pipelines through workflows. The workflows can be created for every push or pull request to your repository. Additionally, you can create workflows that add appropriate labels whenever someone opens a new issue in the repository. The workflow can contain one or more jobs that run sequentially or parallel to other jobs. The jobs run inside a virtual machine or a container. The workflows are configured in a YAML file under `.github/workflows` in your repository and are triggered by an event. The workflow can also be triggered manually or at a defined schedule [58].

The CI pipeline workflow is illustrated in Figure 6.2 below. The pipeline is triggered by a push to the *dev* branch or a pull request to the *main* branch. The workflow consists of several key steps:

1. **Build**: The project is built by setting up the appropriate `Python` version and installing the necessary packages.
2. **Lint**: Static code analysis is performed to ensure adherence to best coding practices and to identify potential vulnerabilities.
3. **Test**: Automated tests are executed, and a test coverage report is generated. This report is then uploaded to `Codecov` for coverage analysis.
4. **Build documentation**: The documentation and tutorials are built and uploaded to Read the Docs for public access.

[5] https://flake8.pycqa.org/en/latest/

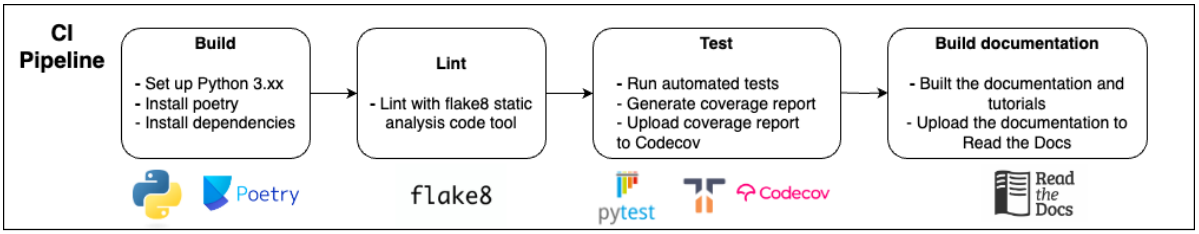


Figure 6.2: Github Actions CI pipeline.

Continuous Deployment (CD): CD enables development teams to deploy their code or applications automatically without any human intervention in the development, test or production environments. This automation reduces manual work, minimizes the risk of human error, and accelerates the delivery of new features or fixes to the end-users. While it's possible to do CI without CD, it's not possible to do CD without having CI in place. Deploying changes directly to a production environment without adhering to fundamental CI principles, such as integrating changes into a main branch or running automated tests before deployment, poses substantial risks. Such actions can lead to disruptions in the production environment, resulting in application malfunctions and user dissatisfaction, ultimately leading to the potential loss of users [59].

The CD pipeline workflow is illustrated in Figure 6.3 below. The pipeline is triggered on a *release tag*. The workflow consists of several key steps:

1. **Build**: The project is built by setting up the appropriate `Python` version and installing the necessary packages.
2. **Publish the package to PyPI**: The package is built and published to PyPI. Publishing requires the PyPI access token from a maintainer or owner of the project on PyPI.

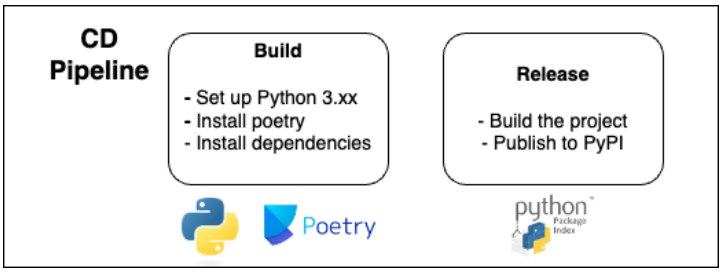


Figure 6.3: Github Actions CD pipeline.

6.5. Documentation

Documentation plays an equally vital role, serving as a comprehensive guide that outlines the functionalities, features, and usage instructions of the software. By paying close attention to these aspects, software development teams can improve their product quality, simplify development workflows, and facilitate effective communication among team members and users.

Building documentation. `Sphinx`[6] is a `Python` package to build documentation for a custom implemented library. Under the hood, it takes your reStructuredText (rst) or Markdown (MD) documents and converts them into HTML files or a PDF file. Many well-known libraries such as `NumPy`, `SciPy` and `scikit-learn` document their library using Sphinx. The library leveraged `Sphinx` to build the documentation and tutorials.

Hosting documentation. Once the documentation is built, a hosting platform that seamlessly integrates into our workflow is essential. `Read the Docs`[7] is an excellent tool for documentation versioning and hosting. It simplifies software documentation by treating it like code, making it easier to maintain.

[6] https://sphinx-rtd-tutorial.readthedocs.io/en/latest/
[7] `https://docs.readthedocs.io/en/stable/`

Additionally, it can host multiple versions of the documentation, such as version 1.0 and 2.0, by directly pulling them from Git.

Figure 6.4: Home page of the `PyTSPL` documentation hosted by `Read the Docs`.

The documentation also includes an installation guide, a quick start section, and tutorials to help users become familiar with the library's functionalities. For future work, the aim is to add additional tutorials.

6.6. Contributing and Changelog

Contributing: The `CONTRIBUTING.md` file in the repository provides comprehensive instructions on how to contribute to the library. It includes detailed guidelines on how to fork the repository, create a new branch, make changes, write and run tests, update documentation, and submit a pull request. These instructions ensure that contributors can easily follow a structured process to fix bugs, add new features, and improve existing functionality.

By following the guidelines, contributors can:

- *Fork the Repository*: Create a personal copy of the repository to work on.
- *Create a New Branch*: Work on a separate branch to keep changes organized and isolated.
- *Make Changes*: Improve or add functionality in the `pytspl` folder, ensuring all changes are well-documented.
- *Write and Run Tests*: Write unit tests to cover new code and run existing tests to maintain code quality.
- *Update Documentation*: Keep the `README.md` and `CHANGELOG.md` up to date with relevant changes and new features.
- *Submit a Pull Request*: Follow a streamlined process to submit changes for review and integration into the main codebase.

Additionally, the `CONTRIBUTING.md` file encourages contributors to add examples or tutorials in the `notebooks` folder, demonstrating the usage of new features. This comprehensive guide lowers the barriers to contribution, fostering an inclusive and collaborative open-source community. Over time, this collaborative effort helps the library evolve and improve, benefiting all users. By adhering to these

guidelines, contributors ensure their contributions are effectively integrated, maintaining the library's robustness and reliability.

Changelog: A `CHANGELOG.md` file is used to keep a chronological record of all notable changes made to a project. This file is particularly useful in the context of software development as it provides a clear and concise history of changes, improvements, and bug fixes, making it easier for users and developers to understand the evolution of the project [60].

Here are the primary purposes of a `CHANGELOG.md` file:

1. **Track Changes**: It provides a detailed log of what has been added, changed, fixed, deprecated, removed, and improved in each version of the project.
2. **Communication**: It communicates to users and contributors what has been done in each release. This transparency helps users understand new features, bug fixes, and any breaking changes that might affect their use of the software.
3. **Reference for Developers**: It serves as a reference for developers to understand the history of the project, aiding in debugging and understanding the context of changes.
4. **Release Notes**: It can be used to generate release notes for new versions of the software, summarizing the key changes and improvements.
5. **Project Management**: It helps in managing the project by keeping a structured record of progress and development activities over time.

7

Library Usability Evaluation

One of our primary goals was to create a usable library that would facilitate ease of use to the users and integration for developers. To evaluate the usability of the library, a questionnaire was designed to assess how well new users adapt to and utilize the functionality.

At the end of the thesis, eight users had answered the evaluation. The users can be divided into three groups.

- **Group 1**: The first group consisted of people who were unfamiliar with the field of graphs and learning on graphs, especially SCs. They have a computer science background and have been using various libraries throughout their academic and professional lives. Furthermore, they have at least some sort of industrial experience and are currently working in the industrial sector.
- **Group 2**: The second group consisted of people who are familiar with graphs and are doing their PhD in a similar subfield. They also have a computer science background and have been in academia for the most part.
- **Group 3**: The third group consists of people who have closely been working with the development of `PyTSPL`, and are educated individuals in the field of SCs and learning on SCs with various published journals and papers.

The questionnaire evaluated four key sections:

- **Installation**: This section assessed the ease of installing the library, including the clarity of instructions, the simplicity of the process, and any issues encountered during installation.
- **Quick Start**: This section focused on the initial setup and use of the library. It evaluated how easily users could get started, including their ability to understand and execute the basic functionality without extensive help.
- **Tutorials and API Reference**: This section examined the quality and comprehensiveness of the tutorials and API reference documentation. It looked at how well the users found it useful.
- **General Feedback**: This section gathered users' overall impressions and experiences with the library. It included open-ended questions to capture any additional comments, suggestions for improvements, and feedback on aspects not covered in the previous sections.

7.1. Installation

This section evaluates the installation process of `PyTSPL`. To evaluate the installation process, users were asked multiple questions with an open feedback form at the end.

Installation guide is available. Users were asked if the installation guide is available in the provided documentation. The evaluation result of the question is shown in Figure 7.1 below.

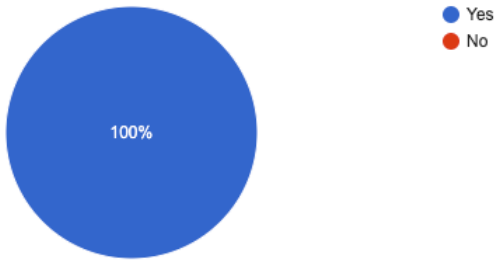


Figure 7.1: Question 1: Installation guide is available.

All of the users agreed that the installation guide is available in the provided documentation.

Installation guide is easy to follow. The second question focused on the clarity of the user guide, evaluating how well the instructions were presented and whether users could follow them without confusion. The evaluation result of the question is shown in Figure 7.2 below.

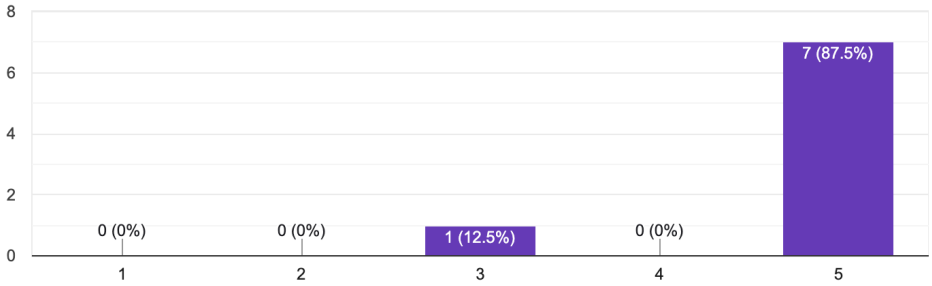


Figure 7.2: Question 2: Installation guide is easy to follow.

All bar graphs use a scale from 1 to 5, where 1 represents ”strongly disagree” and 5 represents ”strongly agree”. The majority of the users were happy with the clarity of the installation guide and considered them easy to follow. One of the users suggested adding the instructions for setting up the virtual environment, which was later added to the installation guide.

Installation setup completed successfully. The third and fourth questions asked users if they encountered any errors during the installation and whether the installation setup was completed successfully. The evaluation results of the questions are shown in Figure 7.3 and 7.4 below, respectively.

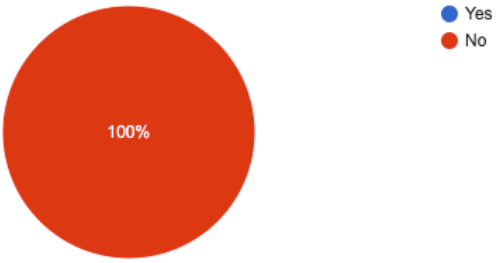


Figure 7.3: Question 3: Did you encounter any errors during the installation?

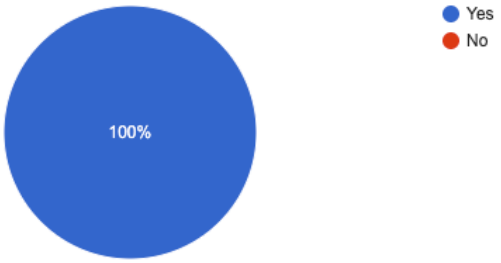


Figure 7.4: Question 4: Installation setup was completed successfully.

None of the users encountered any errors during installation and the installation setup was completed successfully.

Additional feedback. Most of the users gave positive feedback regarding the installation process. One of the users suggested only installing dependency packages when they are needed. This was a valuable suggestion and was tackled.

7.2. Quick Start

This section evaluates the quick start process of `PyTSPL`. To evaluate how quickly users can get started with the library, users were asked the following questions.

Quick start guide is available. Users were asked if the quick start guide is available in the provided documentation. The evaluation result of the question is shown in Figure 7.5 below.

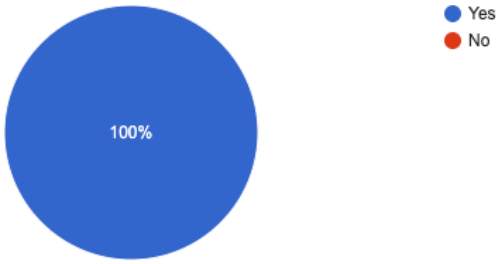


Figure 7.5: Question 5: Quick start guide is available.

All of the users agreed that the installation guide is available in the provided documentation.

Quick start guide is easy to follow. The sixth question focused on the clarity of the quick start guide, evaluating how well the instructions were presented and whether users could follow them without confusion. The evaluation result of the question is shown in Figure 7.6 below.

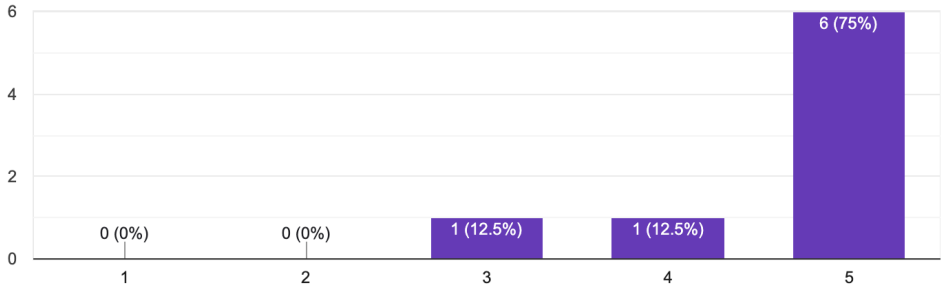


Figure 7.6: Question 6: Quick start guide is easy to follow.

The majority of the users were happy with the clarity of the quick start guide and considered them easy to follow. A couple of users didn't strongly agree, which is a clear indication that the quick start guide can be improved. The goal is to add additional documentation and examples that would introduce users to the basic functionality of the library.

Quick start completed successfully. The seventh and eighth questions asked users if they encountered any errors during the quick start and whether the quick start setup was completed successfully. The evaluation results of the questions are shown in Figure 7.7 and 7.8 below, respectively.

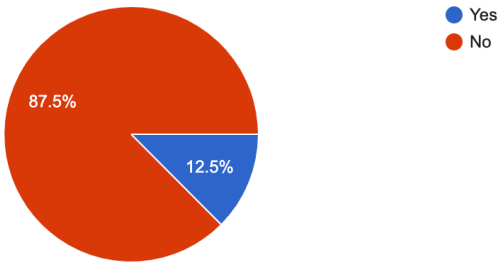


Figure 7.7: Question 7: Did you encounter any errors during the quick start?

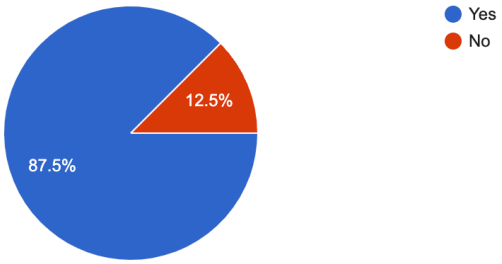


Figure 7.8: Question 8: Quick start setup was completed successfully.

The majority of the users were able to complete the quick start process without issues, however, one of the users encountered errors during the quick start. After investigating the problem, the user encountered a problem that was not related to the setup of the library. The problem was related to their environment which was not setup correctly. Due to this problem, they were encountering import errors. Once the environment was set properly, they were able to complete the quick start without any further issues. This was another reason why the instructions to setup the virtual environment were added to the installation guide.

Quick start examples were helpful. The users were asked if the quick start examples were helpful. The evaluation result of the question is shown in Figure 7.9 below.

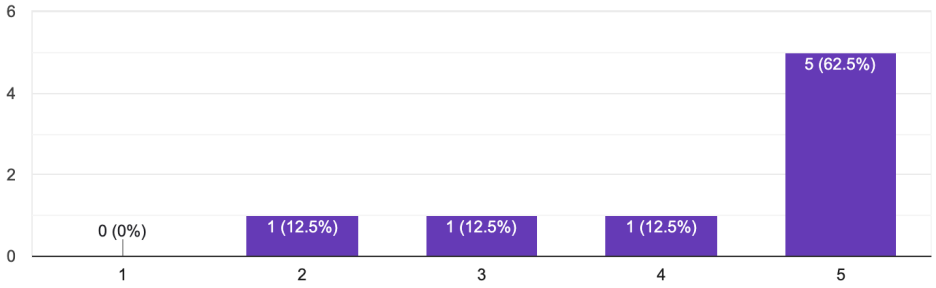


Figure 7.9: Question 8: Were the examples provided in the quick start guide helpful?

While most of the users found it helpful, a few users didn't strongly agree. After investigating the matter in terms of the groups, the hypothesis is that Group 2 users don't have any background on SCs and require more documentation on what the quick start examples do. A quick introduction to SCs and examples can be added to provide people with more background. While all the Group 2 users found the quick start helpful, Group 3 users didn't and suggested adding additional examples and documentation. Since they have a strong background regarding the domain, it would be useful to follow their suggestions and improve the quick start examples.

7.3. Tutorials and API Reference

This section evaluates the tutorials and documentation of `PyTSPL`. This includes clarity of instructions, comprehensiveness, usefulness of examples and ease of navigation. To evaluate the usefulness of the tutorials and documentation, the users were asked the following questions.

Tutorials. The users were asked if the tutorials provided a good starting point for new users to explore the different functionalities of the library. The evaluation result of the question is shown in Figure 7.10 below.

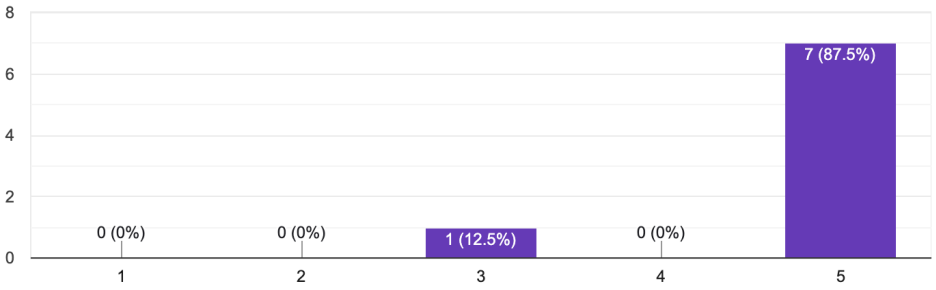


Figure 7.10: Question 9: The tutorials provide a good starting point for new users to explore the different functionalities of the library.

Most users found the tutorials helpful for new users of the library. However, users in Group 3 did not strongly agree with this assessment. The hypothesis is that Group 3 users are seeking improvements in the tutorials, specifically requesting more comprehensive documentation and additional examples to enhance their learning experience. This is valuable feedback and to improve the completeness of the library, this is an aspect to improve.

API Reference. The users were asked if the API Reference provides a useful tool to explore and look up documentation for different functionalities. The evaluation result of the question is shown in Figure 7.11 below.

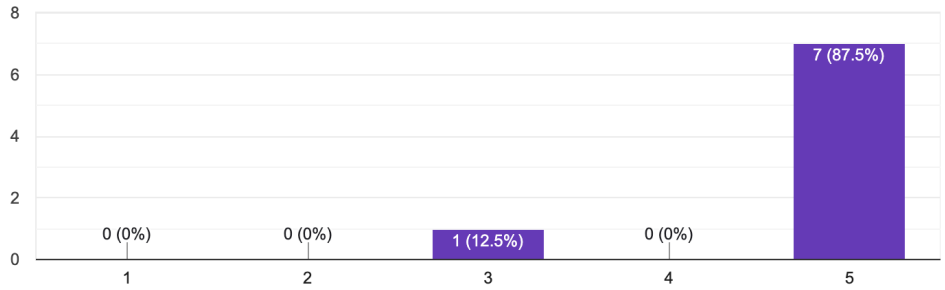


Figure 7.11: Question 10: The API Reference provides a useful tool to explore and look up documentation for different functionalities.

Similarly, most users found the API Reference as a useful tool to explore and look up documentation. In the open feedback, most of the users provided positive feedback regarding the tutorials and API references. Once again users from Groups 1 and 2 thought of the tutorials and API reference as useful and well-documented, while the users from Group 3 found it incomplete. Since it is an iterative process, the tutorials and API References will be improved over time.

7.4. Overall Feedback

This section evaluates the overall experience of the users after initially using the library. At the end of the evaluation, users were asked about their overall experience. The evaluation result of the question is shown in Figure 7.12 below.

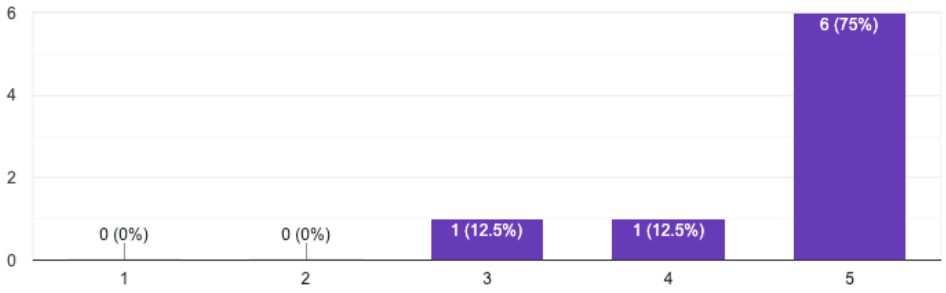


Figure 7.12: Question 11: Overall user experience of the library.

Users were also asked if they had any additional comments or suggestions for improving the library. Overall, users had a pleasant experience. Some valuable feedback was provided to decrease the number of installed packages and add additional examples to the tutorials. These suggestions are considered and will be implemented before the end of the thesis.

8

Conclusion

8.1. Thesis Summary

Simplicial complexes (SCs) have shown remarkable performance in capturing complex graph structures where the signals naturally associate with the edges or sets of nodes (triangles). As the field of SC is evolving at a fast pace, a unified process to process these structures was halting the responsibility of research advances. In this master's thesis, we introduced a Python library `PyTSPL` to bridge the gap between research and experiments. The library enables direct interactions with higher-order networks, specifically SCs. The library streamlines the workflow for loading datasets defined over a network, building SCs, and manipulating, analyzing and visualizing them. Furthermore, it offers functionality for advanced signal processing techniques such as eigendecomposition, Hodge decomposition, simplicial convolutional filters, simplicial trend filtering and Hodge-compositional edge Gaussian processes. For each of the modules, implementation details and pedagogical examples are provided. Additionally, the performance of each module is evaluated using different datasets. We showed how the Chebyshev polynomial filter design has a superior performance compared to LS-based and grid-based filter designs on large network datasets. We evaluated the performance of simplicial trend filtering using different regularization techniques for denoising and interpolation tasks. Further, we analyzed the performance of Hodge- compositional edge Gaussian processes for modelling functions defined over the edge set of simplicial 2- complexes. In Chapter 6, we delve into the software development process of the library, adhering to the best software development practices throughout the development cycle. We delved into each process and provided detailed explanations of why each step is necessary. For each phase, we discussed the importance of following best practices to ensure code quality, maintainability, and scalability. We also highlighted the benefits of using specific tools and methodologies, and how they contribute to the overall success of the software development lifecycle. In Chapter 7, we assessed the usability of the library for new users. This evaluation helped us identify potential issues and areas for future improvement, ensuring the library meets user needs more effectively. We concluded that `PyTSPL` serves as a foundational backbone for the future development of additional functionalities on SCs and other higher-order networks. Its comprehensive feature set provides a solid platform for extending capabilities, enabling researchers and developers to build upon and enhance the library to meet evolving needs in these advanced areas of study.

8.2. Future Work

`PyTSPL` is the first Python library that unifies the workflow of loading, building, manipulating, visualizing, and applying advanced signal processing techniques and learning on SCs. This is only the first step and there are still many ways this library can be improved. In this section, we will touch on a few which directly relate to our work.

Improving tutorials and documentation. One of the critical areas for improvement is the refinement of tutorials and documentation. While the current documentation provides a solid foundation, expanding it with more comprehensive examples and detailed explanations will help new users understand and

utilize the library more effectively. This includes step-by-step tutorials for each module with additional documentation.

Adding additional functionalities. Expanding the library's functionality is another important direction for future development. This could include integrating more advanced signal processing techniques, supporting a broader range of higher-order networks, and adding tools for specific applications such as analysis and machine learning on SCs. These additions will make `PyTSPL` even more versatile and powerful for researchers and developers.

Efficiency. Enhancing the efficiency of the library is crucial for handling large datasets and complex computations. Optimizing the existing algorithms, improving memory management, and leveraging parallel computing techniques can significantly boost performance. This will enable `PyTSPL` to process larger datasets more quickly and efficiently, improving the overall user experience.

References

[1] Ortega, Antonio et al. “Graph Signal Processing: Overview, Challenges, and Applications”. In: *Proceedings of the IEEE* 106 (May 2018), pp. 808–828. DOI: 10.1109/JPROC.2018.2820126.

[2] Derrible, Sybil and Kennedy, Christopher. “Applications of Graph Theory and Network Science to Transit Network Design”. In: *Transport Reviews* 31 (July 2011), pp. 495–519. DOI: 10.1080/01441647.2010.543709.

[3] Borgatti, Stephen et al. “Network Analysis in the Social Sciences”. In: *Science (New York, N.Y.)* 323 (Mar. 2009), pp. 892–5. DOI: 10.1126/science.1165821.

[4] Imam, Ali. *Visualize your LinkedIn network with InMaps*. [Online; accessed 18. Jul. 2024]. Jan. 2011. URL: https://www.linkedin.com/blog/member/product/linkedin-inmaps.

[5] Kee, Kerk et al. “Social Groups, Social Media, and Higher Dimensional Social Structures: A Simplicial Model of Social Aggregation for Computational Communication Research”. In: *Communication Quarterly* 61 (Jan. 2013). DOI: 10.1080/01463373.2012.719566.

[6] Leung, K.K., Massey, W.A., and Whitt, W. “Traffic models for wireless communication networks”. In: *IEEE Journal on Selected Areas in Communications* 12.8 (1994), pp. 1353–1364. DOI: 10.1109/49.329340.

[7] Gebhart, Thomas, Fu, Xiaojun, and Funk, Russell. *Go with the Flow? A Large-Scale Analysis of Health Care Delivery Networks in the United States Using Hodge Theory*. 2021. arXiv: 2110.09637 [cs.SI]. URL: https://arxiv.org/abs/2110.09637.

[8] Candogan, Ozan et al. “Flows and Decompositions of Games: Harmonic and Potential Games”. In: *Math. Oper. Res.* 36.3 (Aug. 2011), pp. 474–503. ISSN: 0364-765X. DOI: 10.1287/moor.1110.0500. URL: https://doi.org/10.1287/moor.1110.0500.

[9] Kee, Kerk et al. “Social Groups, Social Media, and Higher Dimensional Social Structures: A Simplicial Model of Social Aggregation for Computational Communication Research”. In: *Communication Quarterly* 61 (Jan. 2013). DOI: 10.1080/01463373.2012.719566.

[10] Patania, Alice, Petri, Giovanni, and Vaccarino, Francesco. “The shape of collaborations”. In: *EPJ Data Science* 6 (2017). URL: https://api.semanticscholar.org/CorpusID:25363328.

[11] Hatcher, A. *Algebraic Topology*. Algebraic Topology. Cambridge University Press, 2002. ISBN: 9780521795401. URL: https://books.google.fi/books?id=BjKs86kosqgC.

[12] Berge, Claude. “Hypergraphs”. In: (1989), p. 256.

[13] Frankl, P. “Extremal set systems”. In: *Handbook of Combinatorics (Vol. 2)*. Cambridge, MA, USA: MIT Press, 1996, pp. 1293–1329. ISBN: 0262071711.

[14] Sandryhaila, Aliaksei and Moura, José M. F. “Discrete Signal Processing on Graphs”. In: *IEEE Transactions on Signal Processing* 61.7 (Apr. 2013), pp. 1644–1656. ISSN: 1941-0476. DOI: 10.1109/tsp.2013.2238935. URL: http://dx.doi.org/10.1109/TSP.2013.2238935.

[15] Shuman, David I et al. “The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains”. In: *IEEE Signal Processing Magazine* 30.3 (2013), pp. 83–98. DOI: 10.1109/MSP.2012.2235192.

[16] Girault, Benjamin, Gonçalves, Paulo, and Fleury, Éric. “Translation on Graphs: An Isometric Shift Operator”. In: *IEEE Signal Processing Letters* 22.12 (2015), pp. 2416–2420. DOI: 10.1109/LSP.2015.2488279.

[17] Chen, Siheng et al. “Signal Recovery on Graphs: Variation Minimization”. In: *IEEE Transactions on Signal Processing* 63.17 (2015), pp. 4609–4624. DOI: 10.1109/TSP.2015.2441042.

[18] Lim, Lek-Heng. “Hodge Laplacians on Graphs”. In: *SIAM Review* 62.3 (2020), pp. 685–715. DOI: 10.1137/18M1223101. eprint: https://doi.org/10.1137/18M1223101. URL: https://doi.org/10.1137/18M1223101.

[19] Grady, L.J. and Polimeni, J.R. *Discrete Calculus: Applied Analysis on Graphs for Computational Science*. Springer London, 2010. ISBN: 9781849962902. URL: https://books.google.fi/books?id=E3-OSVSPbU0C.

[20] Schaub, Michael T. et al. “Signal processing on higher-order networks: Livin’ on the edge... and beyond”. In: *Signal Processing* 187 (Oct. 2021), p. 108149. ISSN: 0165-1684. DOI: 10.1016/j.sigpro.2021.108149. URL: http://dx.doi.org/10.1016/j.sigpro.2021.108149.

[21] Barbarossa, Sergio and Sardellitti, Stefania. “Topological Signal Processing Over Simplicial Complexes”. In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 2992–3007. DOI: 10.1109/TSP.2020.2981920.

[22] Yang, Maosheng et al. “Simplicial Convolutional Filters”. In: *IEEE Transactions on Signal Processing* 70 (2022), pp. 4633–4648. DOI: 10.1109/TSP.2022.3207045.

[23] Ebli, Stefania, Defferrard, Michaël, and Spreemann, Gard. *Simplicial Neural Networks*. 2020. arXiv: 2010.03633 [cs.LG]. URL: https://arxiv.org/abs/2010.03633.

[24] Watkins, David. *The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods*. Vol. 78. Jan. 2007. ISBN: 978-0-89871-641-2. DOI: 10.1137/1.9780898717808.

[25] Sleijpen, Gerard and Van der Vorst, Henk. “A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems”. In: *SIAM Journal on Matrix Analysis and Applications* 17 (Dec. 2000). DOI: 10.1137/S0036144599363084.

[26] Shuman, David, Vandergheynst, Pierre, and Frossard, Pascal. “Chebyshev Polynomial Approximation for Distributed Signal Processing”. In: *2011 International Conference on Distributed Computing in Sensor Systems and Workshops, DCOSS’11* (May 2011). DOI: 10.1109/DCOSS.2011.5982158.

[27] Druskin, V. L. and Knizhnerman, L. A. “Two polynomial methods of calculating functions of symmetric matrices”. In: *USSR Comput. Math. Math. Phys.* 29.6 (June 1991), pp. 112–121. ISSN: 0041-5553. DOI: 10.1016/S0041-5553(89)80020-5. URL: https://doi.org/10.1016/S0041-5553(89)80020-5.

[28] Mason, J.C. and Handscomb, D.C. *Chebyshev Polynomials*. CRC Press, 2002. ISBN: 9781420036114. URL: https://books.google.fi/books?id=8FHf0P3to0UC.

[29] Yang, Maosheng and Isufi, Elvin. “Simplicial Trend Filtering (Invited Paper)”. In: *2022 56th Asilomar Conference on Signals, Systems, and Computers*. 2022, pp. 930–934. DOI: 10.1109/IEEECONF56349.2022.10051892.

[30] Jiang, Xiaoye et al. *Statistical ranking and combinatorial Hodge theory*. 2009. arXiv: 0811.1067 [stat.ML]. URL: https://arxiv.org/abs/0811.1067.

[31] Schaub, Michael T. and Segarra, Santiago. “Flow Smoothing and Denoising: Graph Signal Processing in the Edge-Space”. In: *CoRR* abs/1808.02111 (2018). arXiv: 1808.02111. URL: http://arxiv.org/abs/1808.02111.

[32] Jia, Junteng et al. “Graph-based Semi-Supervised & Active Learning for Edge Flows”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 761–771. ISBN: 9781450362016. DOI: 10.1145/3292500.3330872. URL: https://doi.org/10.1145/3292500.3330872.

[33] Kim, Seung-Jean et al. “ℓ_1 Trend Filtering”. In: *SIAM Review* 51.2 (2009), pp. 339–360. DOI: 10.1137/070690274. eprint: https://doi.org/10.1137/070690274. URL: https://doi.org/10.1137/070690274.

[34] Yang, Maosheng et al. “Finite Impulse Response Filters for Simplicial Complexes”. In: *2021 29th European Signal Processing Conference (EUSIPCO)*. IEEE, Aug. 2021. DOI: 10.23919/eusipco54536.2021.9616185. URL: http://dx.doi.org/10.23919/EUSIPCO54536.2021.9616185.

[35] Rasmussen, Carl Edward and Williams, Christopher K. I. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[36] Duvenaud, David. “Automatic model construction with Gaussian processes”. PhD thesis. Apollo - University of Cambridge Repository, 2014. DOI: 10.17863/CAM.14087. URL: https://www.repository.cam.ac.uk/handle/1810/247281.

[37] Yang, Maosheng, Borovitskiy, Viacheslav, and Isufi, Elvin. *Hodge-Compositional Edge Gaussian Processes*. 2024. arXiv: 2310.19450 [stat.ML]. URL: https://arxiv.org/abs/2310.19450.

[38] Hagberg, Aric, Swart, Pieter, and Chult, Daniel. “Exploring Network Structure, Dynamics, and Function Using NetworkX”. In: *Proceedings of the 7th Python in Science Conference* (Jan. 2008).

[39] Fey, Matthias and Lenssen, Jan Eric. “Fast Graph Representation Learning with PyTorch Geometric”. In: *ArXiv* abs/1903.02428 (2019). URL: https://api.semanticscholar.org/CorpusID:70349949.

[40] Wang, Minjie et al. *Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks*. 2020. arXiv: 1909.01315 [cs.LG].

[41] Rozemberczki, Benedek, Kiss, Oliver, and Sarkar, Rik. *Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs*. 2020. arXiv: 2003.04819 [cs.LG].

[42] Battiston, Federico et al. “Networks beyond pairwise interactions: Structure and dynamics”. In: *Physics Reports* 874 (2020). Networks beyond pairwise interactions: Structure and dynamics, pp. 1–92. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2020.05.004. URL: https://www.sciencedirect.com/science/article/pii/S0370157320302489.

[43] Praggastis, Brenda et al. “HyperNetX: A Python package for modeling complex network data as hypergraphs”. In: *Journal of Open Source Software* 9.95 (2024), p. 6016. DOI: 10.21105/joss.06016. URL: https://doi.org/10.21105/joss.06016.

[44] Landry, Nicholas W. et al. “XGI: A Python package for higher-order interaction networks”. In: *Journal of Open Source Software* 8.85 (2023), p. 5162. DOI: 10.21105/joss.05162. URL: https://doi.org/10.21105/joss.05162.

[45] Badie-Modiri, Arash and Kivelä, Mikko. “Reticula: A temporal network and hypergraph analysis software package”. In: *SoftwareX* 21 (Feb. 2023), p. 101301. ISSN: 2352-7110. DOI: 10.1016/j.softx.2022.101301. URL: http://dx.doi.org/10.1016/j.softx.2022.101301.

[46] Maria, Clément et al. “The Gudhi Library: Simplicial Complexes and Persistent Homology”. In: *Mathematical Software – ICMS 2014*. Ed. by Hong, Hoon and Yap, Chee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 167–174. ISBN: 978-3-662-44199-2.

[47] Feng, Yifan et al. “Hypergraph Neural Networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 3558–3565. DOI: 10.1609/aaai.v33i01.33013558. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4235.

[48] Hajij, Mustafa et al. *TopoX: A Suite of Python Packages for Machine Learning on Topological Domains*. 2024. arXiv: 2402.02441 [cs.LG].

[49] Juviler, Jamie. “What Is GitHub? (And What Is It Used For?)” In: *HubSpot* (Apr. 2024). URL: https://blog.hubspot.com/website/what-is-github-used-for.

[50] Sarah Laoyan. *Understanding dependencies in project management*. [Online; accessed 16. May 2024]. May 2024. URL: https://asana.com/resources/project-dependencies.

[51] Jain, Kusum. “Python Poetry | Python Packaging and Dependency Management”. In: *FavTutor* (Nov. 2023). URL: https://favtutor.com/blogs/poetry-python.

[52] Nguyen, Taylor. “Quality Assurance | Benefits & Importance”. In: *Rikkeisoft - Trusted IT Solutions Provider* (Apr. 2023). URL: https://rikkeisoft.com/blog/importance-of-quality-assurance.

[53] *What Is Software Testing? | IBM*. [Online; accessed 6. May 2024]. Jan. 2024. URL: https://www.ibm.com/topics/software-testing.

[54] Gezer, Begum. “Why Pytest? - Beyn Technology - Medium”. In: *Medium* (Nov. 2022). URL: https://medium.com/beyn-technology/why-pytest-e7f04145155f.

[55] Gillis, Alexander S. “static analysis (static code analysis)”. In: *WhatIs* (July 2020). URL: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis.

[56] *Keeping Your Python Code Clean with Flake8: A Comprehensive Guide*. [Online; accessed 9. May 2024]. Jan. 2024. URL: https://statusneo.com/keeping-your-python-code-clean-with-flake8-a-comprehensive-guide.

[57] *What is a CI/CD pipeline?* [Online; accessed 9. May 2024]. May 2024. URL: https://www.redhat.com/en/topics/devops/what-cicd-pipeline.

[58] *Understanding GitHub Actions - GitHub Docs*. [Online; accessed 9. May 2024]. May 2024. URL: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

[59] GitLab. “What is CI/CD? | GitLab”. In: *GitLab* (Apr. 2023). URL: https://about.gitlab.com/topics/ci-cd.

[60] *Keep a Changelog*. [Online; accessed 8. Jul. 2024]. Feb. 2024. URL: https://keepachangelog.com/en/1.1.0.