University of Nevada, Reno

The Horizontal Tunnelability Graph is Dual to Level Set Trees

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Statistics & Data Science

by

Ibraheem M. Khan

Dr. Ilya Zaliapin & Dr. Deena Schmidt/Thesis Advisor

August, 2023

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

Ibraheem Khan

entitled

The Horizontal Tunnelability Graph is Dual to Level Set Trees

be accepted in partial fulfillment of the
requirements for the degree of

Master of Science

Deena Schmidt
Advisor

Ilia Zaliapin
Co-advisor

Yinghan Chen
Committee Member

Alireza Tavakkoli.
Graduate School Representative

Markus Kemmelmeier, Ph.D., Dean
Graduate School

August, 2023

Abstract

Time series data, reflecting phenomena like climate patterns and stock prices, offer key insights for prediction and trend analysis. Contemporary research has independently developed disparate geometric approaches to time series analysis. These include tree methods, visibility algorithms, as well as persistence-based barcodes common to topological data analysis. This thesis enhances time series analysis by innovatively combining these perspectives through our concept of horizontal tunnelability. We prove that the level set tree gotten from its Harris Path (a time series), is dual to the time series' horizontal tunnelability graph, itself a subgraph of the more common horizontal visibility graph. This technique extends previous work by relating Merge, Chiral Merge, and Level Set Trees together along with visibility and persistence methodologies. Our method promises significant computational advantages and illuminates the tying threads between previously unconnected work. To facilitate its implementation, we provide accompanying empirical code and discuss its advantages.

Keywords: harris path, random self-similar trees, time series, duals, visibility, barcodes, topological data analysis, graphs

Contents

List of Tables

List of Figures

1 Introduction

Just as *Ted Harris* noted that "Walks and trees are abstractly identical objects" (Harris 10) so too is it commonly considered that planar graphs and their duals are "abstractly identical". The seminal observation by Harris led to the development of what is now commonly termed the Harris correspondence by which trees are associated to an appropriate random walk or time series and vice versa. This has allowed for the domain of time series analysis to be made applicable to studying trees. The converse is true as well as various stochastic processes have been shown to be related to certain distributions of random trees — the *level set trees* (Kovchegov and Zaliapin 14). We attempt to broaden this area of research by understanding such time-series derived level set trees as graphs first and hence as those possessing duals. There are two aims of this work: one, is to produce effective metrics for time-series analysis via such duals as has been produced before, and two is to potentially connect other theoretical approaches together and give a deeper understanding of many at-present unconnected methods.

Of particular consideration is the technology of horizontal visibility graphs for time series which captures its "dynamical fingerprints" (Luque et al. 19). Horizontal visibility graphs are not the first graph-theoretic approach to time-series analysis and are immediately preceded by visibility graphs/visibility algorithms. Unlike the Harris correspondence by which an isomorphism is established between the graph (the resulting tree) and the time series, these approaches typically do not yield such a direct connection. Despite this lack of theoretical fidelity, these methods have proven to be very effective at problems of signal vs. noise detection and distributional discrimination. Our research aims to provide justification for such methods by relating them back to the isomorphism that is first obtained by the Harris correspondence via duality.

Although not typically considered a graph-theoretic approach, Time Series Merge Trees also produce tree objects for time series analysis (Stephen 28). Typically these objects are studied in the domain of computational algebraic topology and tools such as persistent homology are used to study the time series (Goldfarb 9). There is insufficient cross-talk between this area of study and other similar approaches to time series analysis as we will show that Time Series Merge Trees are precisely the same as Chiral Merge Trees in other literature(Baryshnikov 3). Moreover, we will prove by way of Horton Pruning the connection between this approach and level set trees.

Recently, a paper by Colin Stephen has shown that Horizon Visibility Graphs, an extension of the aforementioned Horizontal Visibility Graphs, are dual to Time Series Merge Trees (Stephen 28). Prior to this, earlier results were found by Zaliapin and Kovchegov showing that the Harris path time series generating operation and the level set tree generating operations are inverse to each other, and moreover that the level set tree captures long-term information on the time series (such as the fact that there is a correspondence between leaves on the tree and the maxima on the time series)(Kovchegov, Zaliapin, and Foufoula-Georgiou 15). The dual construction in the Stephen paper is what motivated studying the relationship between horizontal visibility and the dual of a tree gotten as the level set tree of a corresponding time series.

The main tool that we introduce is the concept of *horizontal tunnelability* and *horizontal tunnelability graphs*. We will prove that horizontal tunnelability graphs on "U-shaped" segments of a given time series is precisely the dual of its level set tree. In doing so, we will additionally characterize the duals of all binary, rooted trees as well as prove the level set trees are isomorphic to chiral merge trees. Although these two results are simple, it was not found by the authors of this work anywhere in the literature. Furthermore, we will also prove that the first Horton pruning of the time

series merge tree is the level set tree of the vertically flipped time series. Hence, this connection allows us to formally explain the similarities between empirical analysis of time series via level-set tree and time-series merge tree methods in the realms of computational algebraic topology and topological data analysis. As such, our research devises a new method for tree-based time series analysis and proves its relations to many others, as well as relations between them.

Figure 1 summarizes the main relationships between our objects of study, in particular, the novel horizontal tunnelability graph. Starting from a time series X we compute its flipped version $-X$ (the time series multiplied by -1), its level set tree T (isomorphic to the time-series inverted chiral merge tree CMT^-, or that $T^- \cong \mathsf{CMT}$), the weighted path $\breve{X}$, the series appended with $\pm\infty$ (X_∞), as well as the time series merge tree $T_{\breve{X}}$. From there we can compute the duals of these trees, that is, the horizon visibility graph ($T^*_{\breve{X}} = \mathsf{HVG}_\infty$) and our horizontal tunnelability graphs ($T^* = \mathsf{HTG}$), which are related to the classic horizontal visibility graph (HVG). We can additionally go back either via double duals, the Harris Path operation ($H_T = X$), or by one Horton pruning ($\mathcal{R}$).

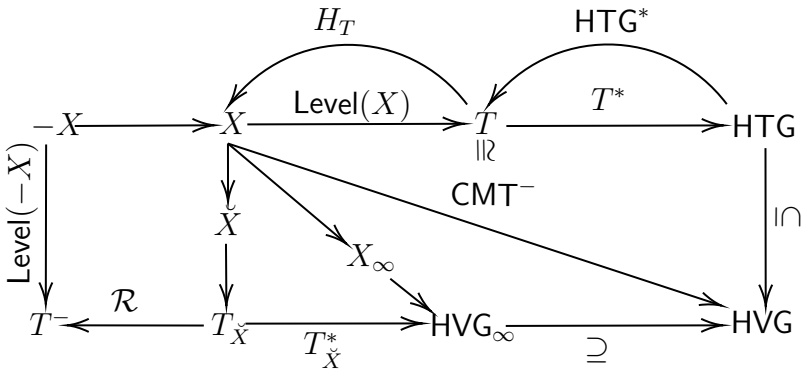


Figure 1: Here is an overview of the objects in our research and their relationships with each other. We begin with a time series X and compute various related objects including its associated trees and from their produce the associated graph objects (via duality).

Furthermore, our work is secondly concentrated on developing practical tools for

doing such analysis. As such, we have written code to compute such duals and made it generic (see Appendix). Packages for computing duals of any kind are sparse, clunky, and difficult to work with. At least for the computation of duals of binary, rooted trees, our package will be the first of its kind. Additionally, we produce code to compute various topological data analysis metrics for our duals, their various related-trees, and the time series in general. We do this in service of demonstrating the empirical efficacy of using our method, horizontal tunnelability, over others, for the specific problems of signal vs. noise detection and distributional discrimination. Lastly, we use our code to empirically investigate open problems regarding random trees and tree self-similarity. We propose various conjectures based on this empirical analysis and hope that our research may prove fruitful in future inquiry.

2 Background and Literature Review

In this section, we aim to elucidate the fundamental mathematical foundations integral to our research - a novel approach to time series analysis using the concept of graph duals. To appreciate the essence of this approach and the innovative dimension it contributes to the established landscape of time series analysis, a thorough understanding of several key mathematical concepts and methodologies is indispensable. Here, we will provide the necessary background into graphs, time series, duality, and other interrelated principles that anchor our theoretical framework.

2.1 Graphs and Duality

To begin, we will first discuss graphs in general and their basic properties. We will then discuss trees, the main graph object of focus in this thesis. Subsequently, we will then dive into the main graph theoretical tool we will work with in this thesis:

duality.

Definition 2.1 (Graph)**.** An undirected graph G is an ordered pair $G = (V, E)$ consisting of a set of vertices, V, and a set of edges, E, which are unordered pairs of vertices. A directed graph has the edges be ordered pairs indicating the direction of the link. Two vertices are said to be *connected* if there is an edge connecting them and a graph is connected if there are no vertices lacking an edge. A *cycle* in a graph is any path, that is a sequence of unique (except for the last) connected vertices, that start and end at the same vertex.

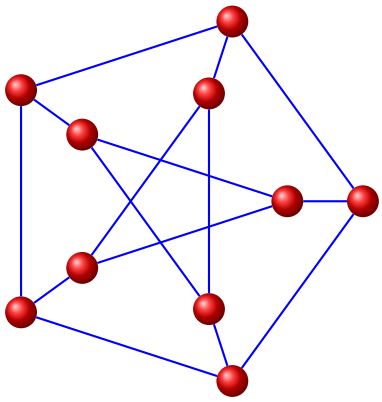


Figure 2: An example of a graph. In red, we have our vertices and in blue we have our edges.

Figure 2 shows the famous *Petersen* graph with its vertices in red and its edges in blue. We call parts of the graph that constitute in themselves another graph a *subgraph* and we additionally term a sequence of edge-connected vertices a *path*. Hence, the Petersen graph is a connected graph containing many cycle subgraphs.

In our research, our particular graphs will acquire probabilistic significance. In particular, we will be interested in vertex degrees, edge lengths, and various related distributions. We refer to those graphs without given edge lengths as *unweighted* such as the one in Figure 2.

Definition 2.2 (Degree of a Vertex; Degree Sequence)**.** The degree of a vertex v in a

graph G, denoted $\mathsf{deg}(v)$, is the number of edges connected to v. The degree sequence of the graph is the non-increasing sequence of vertex degrees.

Definition 2.3 (Degree Distribution)**.** The degree distribution of an undirected graph G, with maximum vertex degree K and N vertices, is the probability distribution of the vertex degrees. That is, it is the probability distribution

$$P(k) = \frac{\#\text{ vertices with degree } k}{N} \tag{1}$$

with $k = \{0, 1, ..., K\}$. We denote the degree function as the non-normalized distribution, that is $f(k) = \#$ vertices with degree k.

Given this, we are now ready to establish our main graph-theoretical tool: duality. In our case, however, we will only be interested in the typical scenario of duals for *planar* graphs. Something to note from Figure 2 is that the edges self-intersect, that is, the graph is not able to be properly *embedded* in the plane. For our research, we will only be interested in computing duals for graphs that can be drawn in the plane without self-intersecting edges.

Definition 2.4 (Graph Embedding)**.** An embedding of a graph is a drawing of it on some surface or manifold, often two-dimensional. That is, every vertex and edge of the graph are given geometric coordinates on some surface such that the edges become arcs that do not ever intersect other edge-arcs or contain points belonging to other edges and whose endpoints are associated with the graph's vertices. If the graph can be embedded on the plane, it is termed *planar*.

Definition 2.5 (Dual of a Planar Graph)**.** Given an embedding of the planar graph, the dual (based on this embedding) can be constructed by identifying the faces of the primal graph as its vertices. The edges of the dual are constructed wherever the face-vertices share a primal edge. The resulting dual graph is also itself planar.

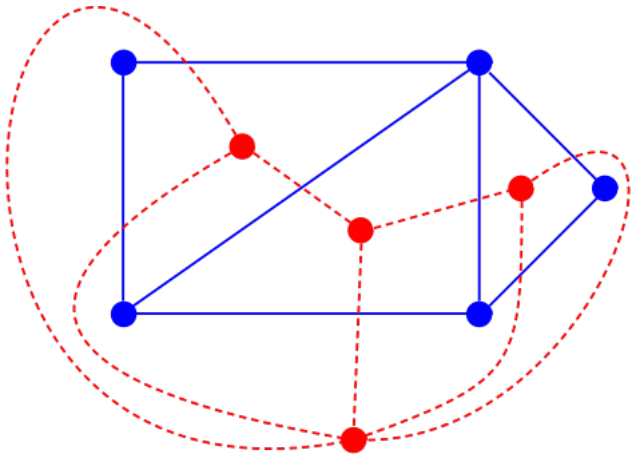


Figure 3: Here is the plane-embedded dual (in red) of a primal graph (in blue). Each red vertex is associated to a region of the plane partitioned by the primal graph. Note the bottom-most red vertex associated with the remainder of the infinite plane.

Although, a requirement for our work regarding duals is that the graphs be planar, we will not be working with typical plane embeddings. We give here a general idea of how duals are computed. As seen in Figure 3, our original graph in blue has a plane-embedded dual in red. Note that there is exactly one region outside of our graph representing the remainder of the infinite, real plane. Our embedding, however, is specific to the tree that we wish to compute the dual of, and is instead onto a closed disk in the real plane.

2.2 Trees

To begin, we will first introduce some basic terminology, following in the convention of earlier research(Haskell 11). We will first discuss trees, random walks, and how exactly it is that they are "abstractly identical". Then we will discuss time series and horizontal visibility. Finally, we will conclude this section by introducing our particular duality construction.

By "tree", we will be referring to finite, rooted (planar) trees. We are particularly interested in those generated by stochastic processes. That is, they are some particular

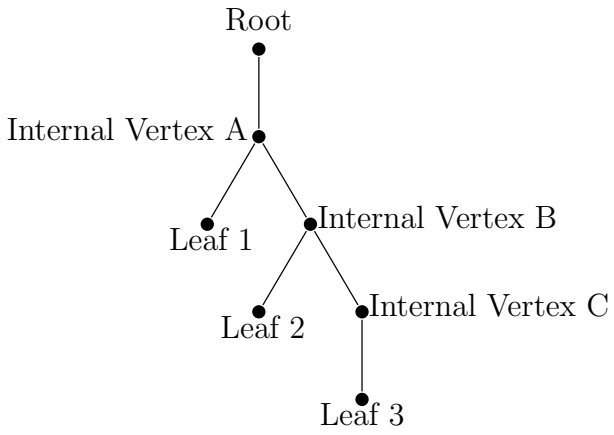


Figure 4: Here is an example of a rooted, binary tree with appropriately labeled leaves and internal vertices.

realizations of a related tree distribution (Pitman 22).

Definition 2.6 (Tree)**.** A connected acyclic undirected graph is called a *tree*.

First, trees are *graphs* possessing vertices and edges that connect them. Trees have hierarchical structure by which each vertex is connected to its *parent* or to nothing at all. Each vertex can also have *children* or *descendants*, that is the, the vertices connected to it excluding its singular parent. Those trees only having at most two children per vertex are termed *binary*. Should a tree possess a parent vertex with only 1 child and no parents, that vertex is denoted the *root* and we consider the tree *rooted*. Vertices with no children are deemed *leaves* of the tree. Finally, considering the tree as a graph, we denote its subgraphs as *subtrees* provided the subgraph is indeed a tree.

Here we will give a useful generalization of the tree concept that will allow us to consider the tree as a metric space. Later in this thesis, we will be dealing with trees with specified edge lengths and it is with this formalism that we will be able to define their size and traversability.

Definition 2.7 (Metric Tree)**.** A metric space (M, d) is called a tree if $\forall u, w \in M$ there exists a unique continuous path $\sigma_{u,w} : [0, d(u, w)] \to M$ that travels from u

to w at unit speed, and for any continuous, injective (simple) path $F : [0, L] \to M$ with $F(0) = u$ and $F(L) = w$, the ranges of F and $\sigma_{u,w}$ coincide. That is, the direct traversal without repetition or backtracking between any two points in the metric space is only possible through one unique path.

It is worth mentioning an alternative characterization of our metric trees, *Real* or $\mathbb{R}$-trees. Elements of our thesis attempt to bridge literature across many domains, in particular topological data analysis (TDA). There, it is common to work with *simplicial trees*, connected simplicial complexes with no cycles, and for whom $\mathbb{R}$-trees offer a simple generalization maintaining the applicability of various TDA constructions (Goldfarb 9).

Definition 2.8 (Real Trees)**.** A metric space (M, d) is called a $\mathbb{R}$-*tree* if it is connected and that it satisfies the 0-hyperbolicity condition:

$$\forall w, x, y, z \in M,\ d(w, x) + d(y, z) \leq max\{d(w, y) + d(x, z), d(x, y) + d(w, z)\} \quad (2)$$

0-hyperbolicity is characterized by the given four-point condition but is also equivalent to the three-point condition that any triplet be geodesically connected, however this formulation lacks an algebraic definition (Kovchegov and Zaliapin 14). Thus, our 0-hyperbolicity condition establishes two things: that our space is a geodesic metric space- that all points are connected through some minimal geodesic- and that "every triangle is a tripod", meaning that all points are indeed connected in a tree-like manner. This formalism makes it simpler to work with infinite trees and it can be seen that any finite metric tree satisfies our condition, where finite refers to the number of vertices.

As mentioned before, we will be interested in trees as considered as particular realizations of a tree distribution. The idea is that each tree in some set will be

assigned a particular probability such that the whole appropriately sums to 1. We will first provide some necessary terminology and then introduce an important example tree distribution.

Definition 2.9 (Forest)**.** A forest is a collection of trees.

Definition 2.10 (Finite, Rooted, Reduced Trees)**.** A tree T possessing some root (some labeled vertex) with a finite number of vertices is a finite, rooted tree. The root understood as a parent, induces a parent-offspring relation between any two adjacent vertices. Vertices with no offspring are termed leaves and the only vertex with no parent is the root. Equivalently, finite, rooted trees are those finite trees who have planar embedding (correspondent to the order on the tree, i.e. the choice of root). A tree is termed *reduced* if there are no vertices with degree 2 except for perhaps the root. We write $\#T$ for the number of non-root vertices or alternatively, the number of edges.

Definition 2.11 (Spaces of Finite, Rooted, Reduced Trees)**.** First, we will refer to *planted* trees as those whose root has degree 1 and otherwise call them *stemless*. Further, we will refer to *binary* trees as those with at most 2 offspring per vertex. Then we denote our spaces and projections as follows.

Note that the trees in $\mathcal{T}$ have no planar embedding because there is no defined order among offspring of the same parent. Also note that $\mathcal{L} = \mathcal{L}^{\vee} \cup \mathcal{L}^{|}$, $\phi = \mathcal{L}^{\vee} \cap \mathcal{L}^{|}$, and vice versa for $\mathcal{T}$. The empty tree is always a member of all of our above spaces. Additionally, it is worth noting that trees given planar embedding possess a *left-right* ordering given by their locations in the plane.

We are now prepared to discuss tree distributions and provide a concrete example of one: Galton-Watson Tree Distributions. Galton-Watson trees are incredibly pivotal to the study of random trees in general and have been used to describe the propagation

$\mathcal{T}$	Unlabeled, finite, rooted, reduced, non-planar trees
$\mathcal{L}$	*Weighted* trees from $\mathcal{T}$ with positive edge lengths ℓ_i
$\mathcal{S}_{\text{plane}}$	Subspace of trees from $\mathcal{S}$ with given planar embeddings
$\mathcal{T}^{\mid}, \mathcal{L}^{\mid}$	Planted trees from $\mathcal{T}$ and $\mathcal{L}$
$\mathcal{T}^{\vee}, \mathcal{L}^{\vee}$	Stemless trees from $\mathcal{T}$ and $\mathcal{L}$
$\mathcal{BS}$	Subspace of *binary* trees from $\mathcal{S}$
SHAPE	The projection $\mathcal{L} \to \mathcal{T}$
P-SHAPE	The projection $\mathcal{L}_{\text{plane}} \to \mathcal{T}_{\text{plane}}$

Table 1: Provided here is a summary of our denoted tree spaces.

of family names, the proliferation of free neutrons in a nuclear fission reaction, and river networks among many other applications (Lalley 17)(Watson and Galton 31). The Galton-Watson process is a branching stochastic process which can be represented as tree, the formalism for which we give below.

Definition 2.12 (Tree Distribution)**.** A tree distribution is a probability measure over some space of trees.

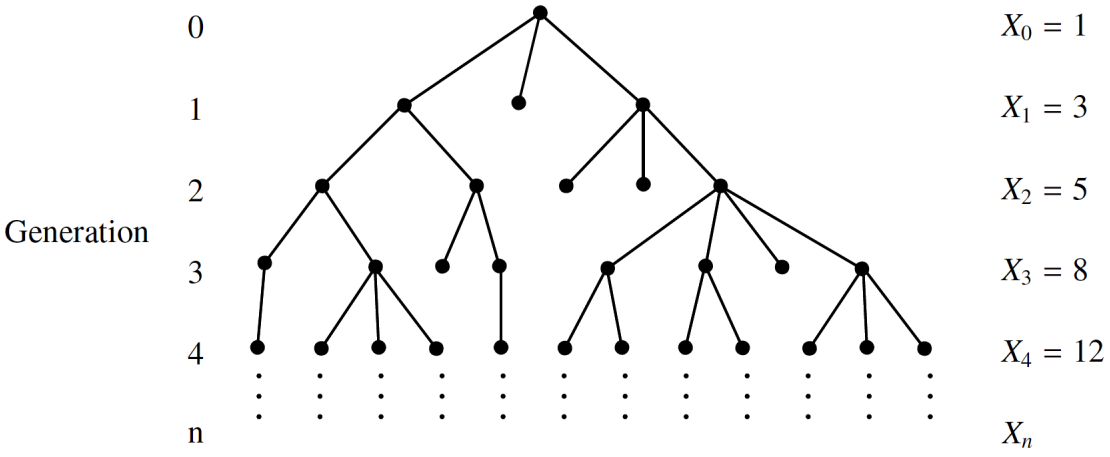


Figure 5: Observe this example Galton-Watson tree for modeling the number of children in each generation for a particular lineage. Note that this tree is not necessarily binary (McKenna 20)

.

Definition 2.13 (Galton-Watson Tree Distribution)**.** We denote the Galton-Watson Tree distribution over $\mathcal{T}^{|}$ as $\mathcal{GW}(q_k)$ with

$$k = 0, 1, ..., K,\ q_1 = 0 \text{ with } \sum_{k=0}^{K} q_k = 1 \text{ and } \sum_{k=0}^{K} kq_k \leq 1 \tag{3}$$

A random Galton-Watson tree starts with a root and evolves stepwise with steps $d = 0, 1, ..., d_{\max}$. At each step $d > 0$ every leaf at max depth $d-1$ will produce $k \geq 0$ offspring with probability q_k. The $q_1 = 0$ requirement assures that the generated tree is reduced. The tree is finite because $\sum_{k=0}^{K} kq_k \leq 1$ and terminates at step $d_{\max}$ when all leaves at depth $d_{\max} - 1$ produce no offspring. A Galton-Watson Process with n-individuals will beget an n-forest of Galton-Watson trees.

We will additionally require technology by which we will traverse and index our trees. Again, as we intend to use trees to aid time series analysis, we will need appropriate quantification tools. Traversing trees and keeping track of its vertices will accomplish what is correspondingly already achieved by such objects as the *linear interpolation* of time series. We will introduce our tree traversal function as evolving from the classical *depth-first search*.

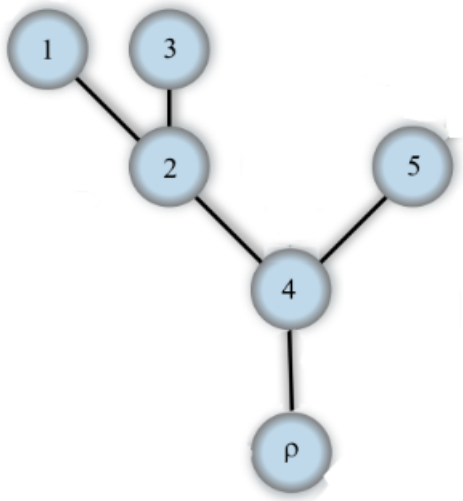


Figure 6: We give the numbered order of depth-first search in a binary tree with root ρ. We start from the leftmost leaf, progress to the root, and keep going until the tree is indexed (Haskell 11)

.

Definition 2.14 (Depth-First Search)**.** Given a tree $T \in \mathcal{BT}_{\text{plane}}$ and its set of vertices $\{v_i\}_{i=1}^{\#T}$, the depth-first search function $\mathcal{D} : \{v_i\}_{i=1}^{\#T} \to \{1, 2, ..., \#T\}$ uniquely assigns indices left-to-right in such a way that for any two vertices the rightmost one has larger index. Informally, the indices are assigned by traversing the tree along the leftmost branches until termination and then backtracking (hence, "depth-first").

Definition 2.15 (Edge Lengths)**.** Let T be a tree in $\mathcal{L}$ and let e_i denote the parental edge of the vertex v_i. Then the length of e_i and the length of the tree $\mathsf{Length}(T)$ is denoted as follows

$$\mathsf{Length}(e_i) = \ell_i, \mathsf{Length}(T) = \sum_{i=1}^{\#T} \mathsf{Length}(e_i) \tag{4}$$

Definition 2.16 (Edge Traversal)**.** Let T be a tree in $\mathcal{L}_{\text{plane}}$ indexed by depth-first search with index i and let e_i denote the parental edge of the vertex v_i. A continuous

function $f(t)$ is said to traverse e_i *towards the root* if for $t_2, t_1 \in \mathsf{Supp}(f)$ where $t_2 - t_1 = \ell_i$, $f(t_1) = v_i$ and $f(t_2) =$ parent of v_i. Likewise, $f(t)$ traverses e_i *away from the root* if $f(t_1) =$ parent of v_i and $f(t_2) = v_i$.

Definition 2.17 (Tree Traversal)**.** The tree traversal function of a tree $T \in \mathcal{L}_{\text{plane}}$ which is indexed by i via depth-first search is given by the unique continuous map $u_T : [0, 2 \cdot \mathsf{Length}(T)] \to T$. u_T *traverses* the edges of T at unit speed starting and ending at the root so as to traverse all the paths σ_{ij} between vertices v_i, v_j in the following order: $(\sigma_{\rho 1}, \sigma_{12}, ..., \sigma_{\#T\rho})$. Informally, this function traverses the edges of the tree in depth-first order twice — first traversing away from the root and then towards it.

Our thesis will be interested in examining certain simple metrics, among others, from given trees. These include, the number of leaves, the number of vertices, and the leaf-to-leaf distances.

Definition 2.18 (Leaf-to-Leaf Distance)**.** In a tree, the leaf-to-leaf distance is the minimum number of vertices required to traverse from one leaf to the other via a connected path.

The more complicated metrics primarily involve those arriving from a study of *self-similar random trees*. In particular, our context emerges from literature surrounding *Horton* self-similar trees and their various generalizations. Originating from studies in hydrogeomorphology, the Horton-Strahler ordering system has become increasingly useful for the study of random trees, in particular those which exhibit *Horton laws* related to the power-law distributions of various statistics in the trees (or more generally, networks) (Haskell 11). In the modest case of our thesis, we will primarily work with the *Horton pruning* operation and its implications. We will now introduce sufficient terminology for these more complicated metrics.

Definition 2.19 (Series Reduction)**.** For a rooted tree Δ, series reduction removes each degree-two non-root vertex by merging its adjacent edges into one (edge lengths are added appropriately).

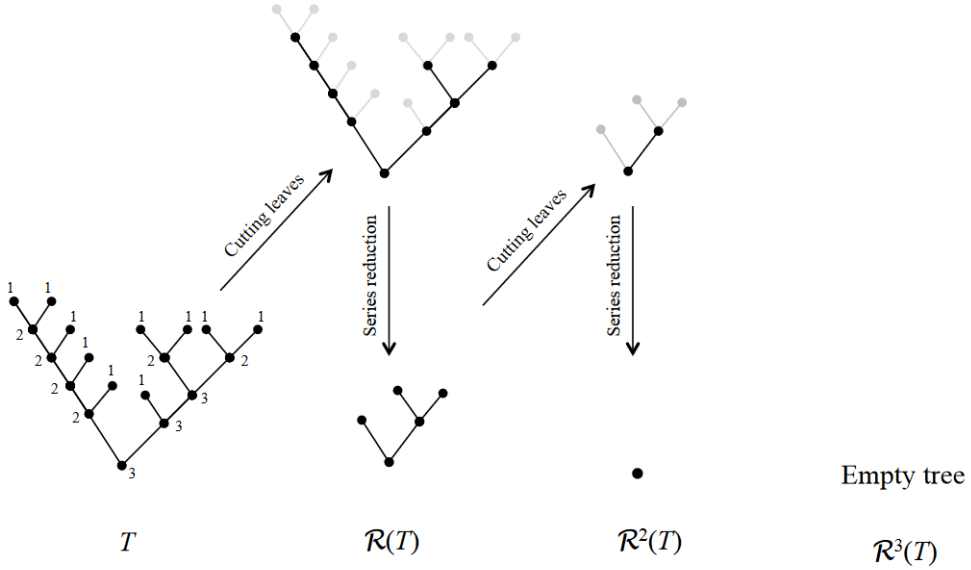


Figure 7: Described here is Horton Pruning on a tree. The operation is repeated until the entire tree is pruned from the left to right by cutting its leaves and performing series reduction to simplify singly-connected branches to one vertex (Haskell 11)

.

Definition 2.20 (Horton Pruning)**.** Let $\mathcal{T}$ denote the space of finite, rooted real trees. Then, Horton Pruning denoted, $\mathcal{R} : \mathcal{T} \to T$, is a surjection given by $\mathcal{R}(\varnothing) = \varnothing$ and for $\Delta \neq \varnothing$, $\mathcal{R}(\Delta) =$"the tree gotten by removing its leaves, their parental edges, and then performing series reduction".

It can be seen that the fixed point of the Horton pruning operation, a contraction map, is the empty tree and that any tree may be deleted via successive Horton prunings.

Definition 2.21 (Horton Self-Similarity)**.** A tree distribution is termed *Horton self-similar* if its distribution is not changed by Horton pruning. That, is for some tree

T in a specified space of rooted trees, and for some tree distribution μ on that space

$$\mu(\mathcal{R}^{-1}(T|T \neq \varnothing) = \mu(T) \tag{5}$$

Definition 2.22 (Horton-Strahler Orders)**.** The order of a leaf vertex is 1. For an internal vertex p with $m \geq 1$ offspring of orders $i_1, i_2, ..., i_m$ resp., the Horton-Strahler order for this vertex or alternatively its parental edge is

$$\text{ord}(p) = \begin{cases} r, & \text{if } |\{s : i_s = r\}| = 1 \\ r+1, & \text{otherwise} \end{cases} \tag{6}$$

The order of a nonempty tree is given as follows.

$$\text{ord}(\Delta) = \max_{v \in \Delta} \text{ord}(v) \tag{7}$$

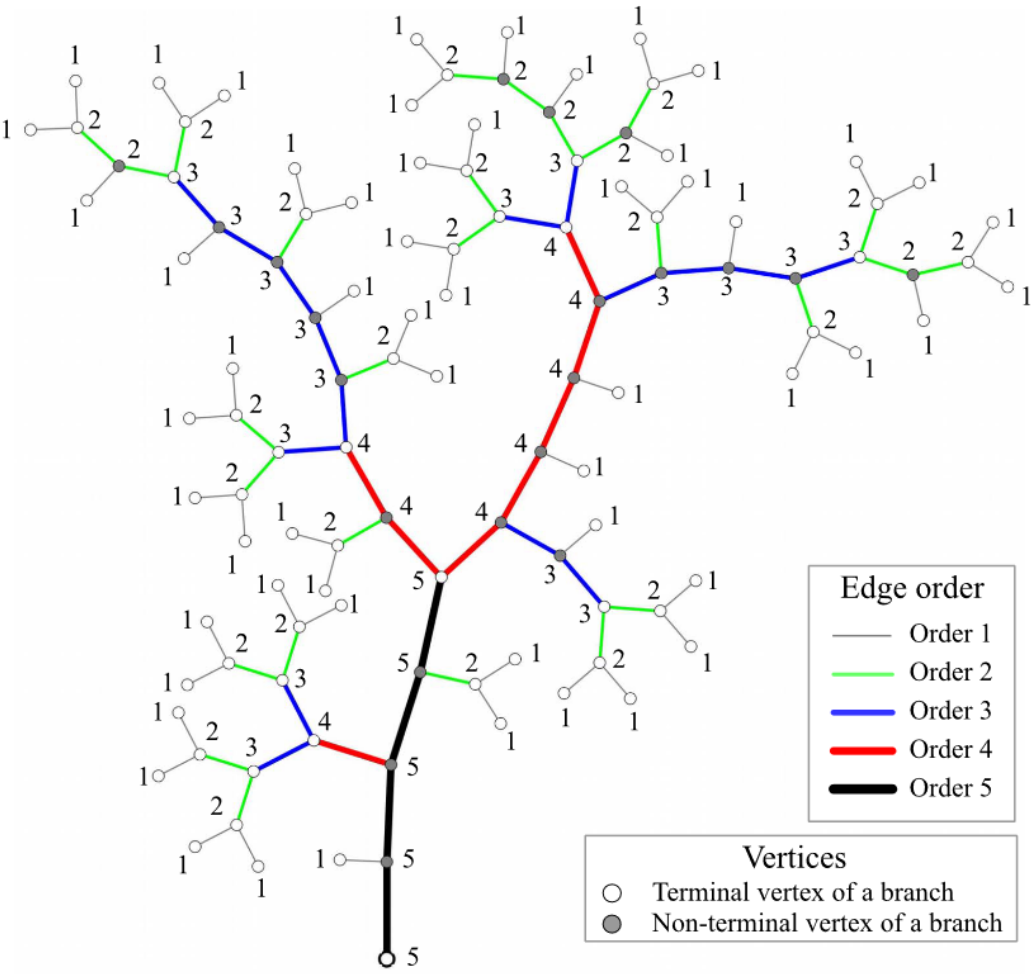


Figure 8: Presented here is the Horton-Strahler ordering on the edges and vertices of a binary (metric) tree (Kovchegov and Zaliapin 14)

.

The order of a tree coincides with the number of successive Horton pruning operations required to eliminate a planted (root is degree 1) tree. If the tree is stemless (not planted), then consider a phantom edge to a phantom root of degree one and the order is now just one more corresponding to this extra pruning operation.

In the course of this thesis, we will be working with random time series transformed into appropriate random trees which will then be transformed into appropriate (random) duals. We will make a case for how such notions as number of leaves, vertices,

leaf-to-leaf distances, and Horton-Strahler orders emerge in our dual representation and what can be gleaned by this understanding. However, recall that our duals are constructed by a particular tree-specific embedding into the disk. We provide this embedding below and will refer to "duals" from hereon out as those constructed as such.

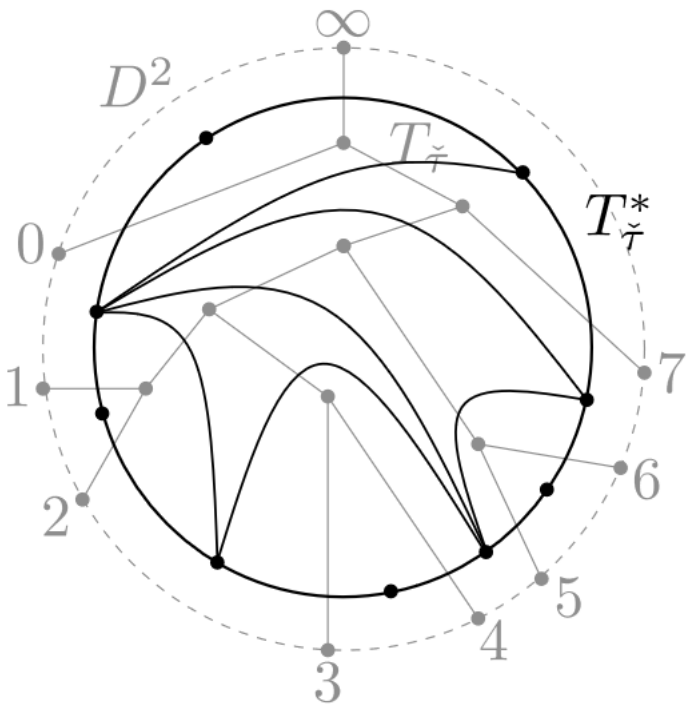


Figure 9: We construct a disk-embedded dual of a tree $T_{\check{\tau}}$ (in grey) as the solid, dark lines $T^*_{\check{\tau}}$(Stephen 28)

.

Definition 2.23 (Dual of a Planted Tree)**.** Given a tree $T \in \mathcal{L}^{|}_{\text{plane}}$, embed it into D^2 by adjoining all leaves and the root to the boundary. Then, define the dual graph G as follows. Internal vertices of G are connected regions of $D^2/(G \cup S^1)$ whose boundary does not include all of S^1 (the faces of the graph). Likewise, there is at most external vertex of G, also a connected region of $D^2/(G \cup S^1)$ whose boundary does includes all of S^1. Edges in G connect vertices whose primal regions share an edge (when two

faces share a tree branch), copying that primal edge's length in the case of a *metric dual*.

2.3 Time Series

Time series are ubiquitous in the field of statistics that primarily deals with analyzing data collected over time. Applications of time series are abundant and diverse and permeates numerous fields. In economics and finance, they are used for forecasting stock prices, economic indicators, and financial risk management(Box et al. 4). In environmental science, time series analysis aids in understanding climate change and weather patterns, and earthquake predictions(Cazalles et al. 6). The medical and health fields utilize time series for epidemiology studies, monitoring patient vital signs, and predicting disease trends(Unkel et al. 30). It is also an essential tool in signal processing and control engineering, enabling the understanding and prediction of signals and systems' behavior over time(Ljung 18). The utility of time series cannot be understated.

Our angle on time series will primarily involve being given them either as-is or understand as the result of some dynamical system or random walk process. Here we will provide some terminology and some of our own original notions that are of consequence to our main results.

Definition 2.24 (Time Series)**.** A time series X is an ordered sequence of n-data points $(X_{t_1}, X_{t_2}, ..., X_{t_n})$ indexed by times $t_1, ..., t_n$.

Definition 2.25 (Random Walk)**.** Suppose X_1,...,X_n are $\mathbb{R}^d$-valued independent and identically distributed random variables. A random walk starting at $z \in \mathbb{R}^d$ is a sequence $(S_n)_{n\geq 0}$ such that $S_0 = z$ and that

$$S_n = S_{n-1} + X_n,\ n \geq 1 \tag{8}$$

Thus, a random walk may be considered a process that generates a time series starting at z where the steps are given by X_n. As such, we may be interested in *parts* of this process as well as how we will geometrically give meaning to it. We will do this via *excursions* and *linear interpolations* respectively.

Definition 2.26 (Linear Interpolation)**.** Given a time series in the plane, its piecewise linear interpolation is the graph generated by connecting all points in the time series (x_t, y_t) with straight line segments.

Definition 2.27 (Excursion)**.** An excursion X_t is a function defined on a real interval $[0, a]$ such that $X_0 = 0, X_a = 0$ and $X_t \geq 0 \forall t \in [0, a]$. Let $\mathcal{E}$ denote the space of piecewise linear interpolations and let $\mathcal{E}^{\text{ex}}$ denote the space of piecewise linear excursions.

For the bulk of this thesis, we will be dealing with those time series that are piecewise linear excursions or that could be appropriately be made into piecewise linear excursions (either by cutting them short, vertically shifting them, or by adding in data points).

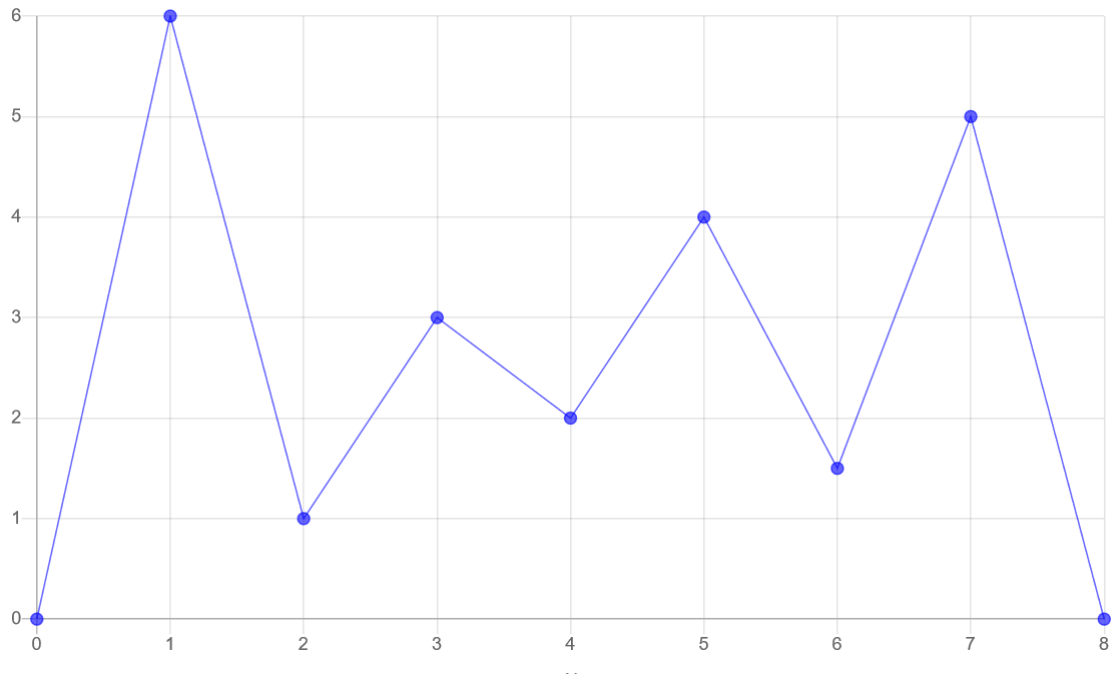


Figure 10: In blue is our piecewise linear interpolation of a time series. Note that it is also piecewise linear excursion as it starts and ends at 0.

Definition 2.28 (Time Series as Piecewise Linear Excursions)**.** A nonnegative time series X, starting and ending at the same point, is necessarily a piecewise linear excursion and has distinct, finite extrema. Furthermore, any real continuous function X_t defined on a closed real interval I has a time series approximation gotten by joining extremal and end points with linear segments. Let the function $\mathfrak{E} : C(I) \to \mathcal{E}^{ex}$ do precisely this but with the addition of shifting I to start at the origin and taking the absolute value.

There is a great deal of literature surrounding the mathematics and statistics of excursions. In particular, Brownian excursions have attracted much study due to the relevance to Wiener processes and their applicability in physics modeling(Revuz and Yor 23). However, for us, we will be working with excursions primarily for the fact that our research works only for the case of these excursions with limited versatility outside of this scope. This is because, we will be focusing on certain properties of

what we have termed *U-shaped* segments. This is a novel way of partitioning time series and reveals new and interesting geometric implications.

Definition 2.29 (U-shaped Segments)**.** Suppose we have a time series $X \in \mathcal{E}^{\text{ex}}$ with support $I = [t_i = 0, t_f]$ and distinct, finite extrema with n-maxima denoted from left to right at $t_1, ..., t_n$. *U-shaped segments* on X are those portions corresponding to subintervals $[t_i = 0, t_1], [t_2, t_3], ..., [t_n, t_f]$.

2.4 Tree and Time Series Duality

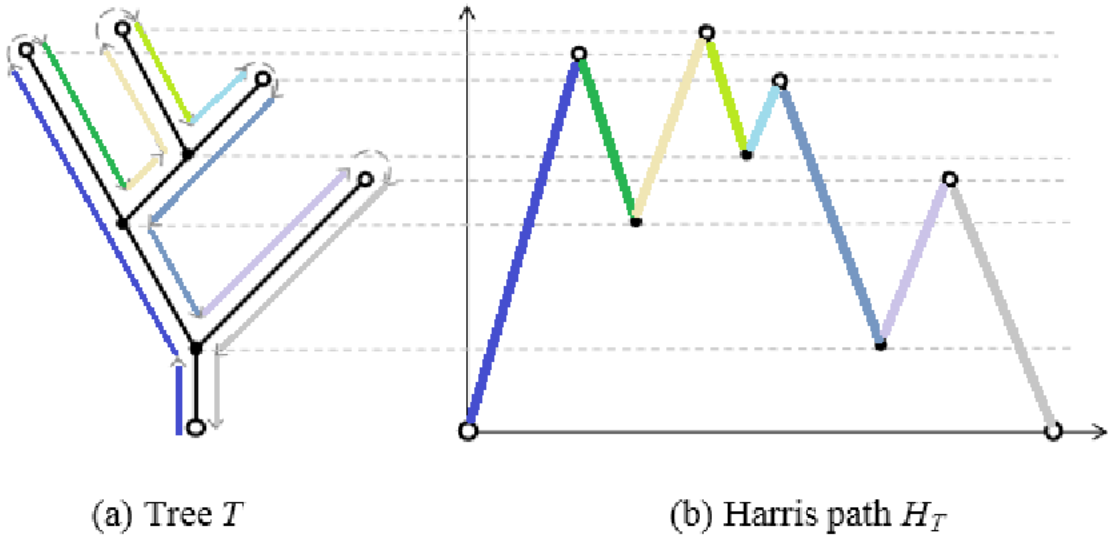
The program of our thesis is to establish a useful transformation of a time series to a tree and then to its dual. We now elucidate the first part of this program. That is, we now establish the Harris correspondence to identify the relationship between trees and excursions.

Definition 2.30 (Harris Path)**.** Given a metric plane tree $T \in \mathcal{L}_{\text{plane}}$, the Harris path is a linear piecewise function $H_T(t) : [0, 2 \cdot \mathsf{Length}(T)] \to \mathbb{R}$ s.t.

$$\forall t, H_T(t) := d_T(u_T(0), u_T(t)) \tag{9}$$

where $d_T(x, y)$ is the distance measured along edges between the vertices x and y on the metric space T. The graph of H_T is an element of $\mathcal{E}$ so we may think of the Harris path as an operation $H : \mathcal{L}_{\text{plane}} \to \mathcal{E}$. Likewise, if the original tree is planted then the graph of H_T is an element of $\mathcal{E}^{\text{ex}}$.

Remark. In the case of a planted binary tree with n leaves and unit edge lengths, the Harris Path has a simple construction as a lattice excursion with $2n$ steps. Start the excursion with a $+1$ step and begin traveling the tree vertices from left to right without repetition starting from the root. If the vertex is internal, the next step is $+1$, if it is a leaf it is -1.



(a) Tree T (b) Harris path H_T

Figure 11: From the tree on the left, we construct its Harris Path (time series) on the right and appropriately color the segments to match each other (Haskell 11)

.

2.5 Level-Set Trees

Just as we may start from a tree to yield a time series so too may we more appropriately begin from a time series and yield a tree. We will focus on three particular methods of achieving this. These include Time Series Merge Trees, Chiral Merge Trees, and Level-Set Trees. All three of these methods aim to describe via trees the hierarchical structure of time series as understood by their relative maxima and minima and how points are connected between them. In particular, the former two methods, although belonging to radically disparate literature, are actually one and the same.

Definition 2.31 (Level-Set Tree)**.** Let $I \subset \mathbb{R}$ be a closed interval and let $f : I \to \mathbb{R}$ be a continuous function with a finite number of extrema. Let $\underline{f}(u, v) = \inf_{x \in [u,v]} f(x)$ for any subinterval $[u, v] \in I$. Then, define the pseudometric d_f on I as

$$d_f(u, v) = (f(u) - \underline{f}(u, v)) + (f(v) - \underline{f}(u, v)), \text{ for } u, v \in I. \tag{10}$$

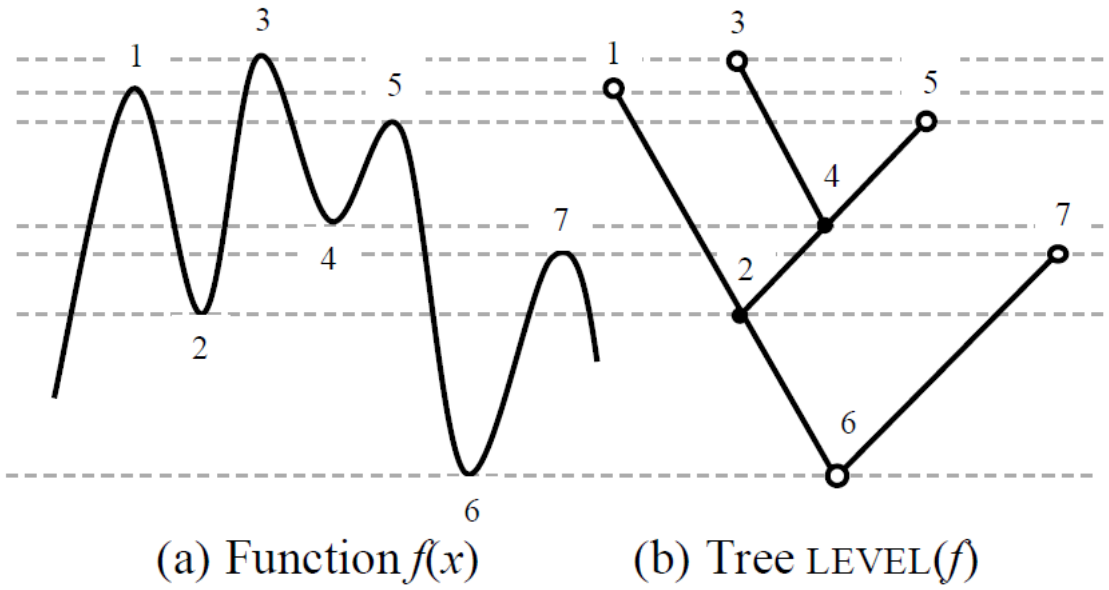


Figure 12: We construct a level set tree on the right from the given continuous function on the left. We additionally note extrema and note how it corresponds to vertices of the tree.

Thus, the quotient space $I_f = I/\sim_{d_f}$ is a metric space and moreover, it is a metric tree denoted $\mathsf{Level}(f)$. One may think of level-set trees as an operation $\mathsf{Level} : \mathcal{E} \to \mathcal{L}_{\text{plane}}$ or alternatively $\mathsf{Level} : \mathcal{E}^{\text{ex}} \to \mathcal{L}^{|}_{\text{plane}}$.

The Level-Set Tree is so-called because it describes the structure of level sets $\mathcal{L}_\alpha(f) = \{x \in I : f(x) \geq \alpha\}$ as a function of the threshold α. Level-Set Trees have some important properties that make them useful representations of time series. The equivalence classes, for example, bijectively correspond to various time series features.

1 point classes	Leaf vertices in the tree corresponding to local maxima.
2 point classes	Edge portions corresponding to the first positive excursions of $X(t) - X(t_i)$ to the right and left of a local minima at t_i. The left and rightmost edge portions may instead correspond to positive segments instead of outright excursions of $X(t) - X(t_i)$.
3+ point classes	Internal vertices corresponding to local minima not on the boundary.

Table 2: Here we summarize the various point classes (equivalence classes) of level-set trees.

Remark. The Level-Set Tree is always a *binary* tree —that is, any vertex has at most two children, provided that its generating time series is a piecewise linear excursion with no consecutively repeated values. In this case, construction is geometrically simple. Create the tree by placing leaves at local maxima, placing internal vertices at local minima, and placing the root at 0 in the center of the time series. Then, simply connect your tree vertices left-to-right.

There are a number of important uses to level-set trees. One such use is the fact that Horton self-similarity of a level-set tree corresponds to invariance of the time series under transition to local extrema(Kovchegov and Zaliapin 14). This is because Horton pruning itself corresponds to transition to the local extrema of a function.

Proposition 1 (Horton pruning of positive excursions(Kovchegov and Zaliapin 14)). The transition from a positive excursion X_t to the respective excursion $X_t^{(1)}$ of its local minima corresponds to the Horton pruning of the level set tree $\mathsf{Level}(X_t)$. In general,

$$\mathsf{Level}(X_t^{(m)}) = \mathcal{R}^m(\mathsf{Level}(X_t)),\ \forall m \geq 1 \tag{11}$$

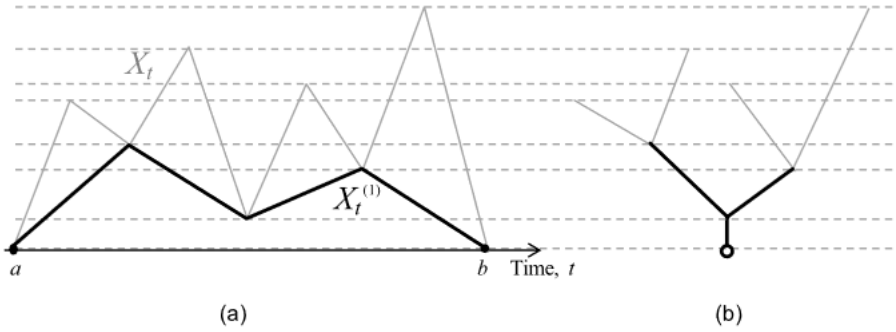


Figure 13: Depicted here is the transition of an excursion to its local minima (a) and its correspondent Horton-pruned level set tree (b) (Kovchegov and Zaliapin 14)

.

As a result of this, it becomes possible to geometrically display the *distributional* equivalence of time series. That is, the counterpart to Horton self-similarity for time series is *extreme-invariance*. This can be interpreted as the time-series ability to maintain its statistical structure regardless of its coarsening down of extrema. The general transformability of level set trees to time series and vice versa further exposes suitable study of such invariant behavior by either object.

Theorem 2.1 (The Harris Path Contour is the Inverse of the Level-Set Tree (Pitman 22))**.** *Consider a tree* $T \in \mathcal{L}_{plane}$ *and let* H_T *be the Harris Path Contour with support* $I = [0, 2 \cdot \mathsf{Length}(T)]$*. Then,* $(I/\sim_{H_T}, d_{H_T}) = \mathsf{Level}(H_T)$ *is a metric tree and moreover it is isometric to* (T, d_T)*. That is,* $\mathsf{Level}(H_T) \cong T$*. Similarly,* $H_{\mathsf{Level}(f)} = f$ *for* $f \in \mathcal{E}^{ex}$*.*

We will conclude this section with a short discussion of chiral merge trees and the related TDA-based literature. In topological data analysis and morse theory, *merge trees* are used to keep track of the development of connected components of sublevel sets of a Morse function in a similar fashion to level set trees (Brüggeman 5). If we restrict our Morse functions, that is functions on a smooth differentiable manifold with no degenerate critical points, to just those functions from $\mathbb{R} \to \mathbb{R}$ this will yield almost the same construction as level-set trees. The *chirality* refers to the left and

right handedness of the resulting tree which is given by the plane embedding of our level-set tree such that vertices are left or right children of their parents. The fact that these constructions are the same is not found in the literature and it is worth tying the two here. To be specific, chiral merge trees are isomorphic to inverted, rooted level-set trees.

The particular version of chiral merge trees which we establish our correspondence to level set trees with is given by J. Curry (Curry 7). There, they establish a useful notion of *graph-equivalence* by which continuous functions may be easily related to particular linear interpolations along extrema.

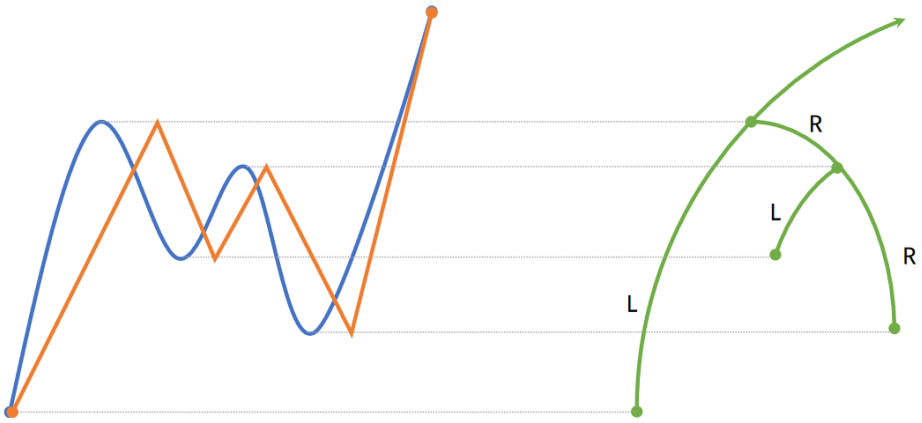


Figure 14: On the left we depict a continuous function (in blue) shown to be *graph-equivalent* to the time series linear interpolation (in orange). On the right, we depict their Chiral Merge Tree, which has preserved *chirality* labeled as right or left, describing the time-order of the original functions' merge set hierarchy (Curry 7)

.

2.6 Time Series Merge Trees

As discussed before, unlike the past two approaches for yielding a tree from a time series there is a third approach. We briefly recreate here the approach taken by Colin Stephen in producing his time series merge trees. This approach is not at all dissimilar from the usual approach in TDA for which a characterization is given by

Baryshnikov (Baryshnikov 3).

Recall our prior use of the term *weighted*. We will now generalize this for all graphs as such and in particular for time series as considered as graphs.

Definition 2.32 (Weighted Graph)**.** A weighted graph $G = (V, E, f)$ is a graph equipped with a weight function $f : E \to \mathbb{R}$.

Definition 2.33 (Weighted Time Series)**.** Given a time series $\tau = (x_1, ..., x_N)$ define the time series weighted path $\breve{\tau}$ to be the graph $\breve{\tau} = (V, E, f)$ where:

$$V = 0, 1, ..., N \qquad E = e_i = (i-1, i) \qquad f : E \to \mathbb{R};\ e_i \mapsto x_i \qquad 1 \leq i \leq N \quad (12)$$

Just as level-set tree before characterize sublevel sets before, for this construction we will formalize the notion of sublevel graphs.

Definition 2.34 (Sublevel Graph)**.** Given a graph $G = (V, E, f)$ and $a \in \mathbb{R}$ define the sublevel graph G_a to be the subgraph of G whose edges have weight no greater than $a : G_a := (V, E_a, f) \subseteq G$, where $E_a := e \in E : f(e) \leq a$.

Remark. Note that all vertices of G will be present in its sublevel graphs and that the weight function induces a strictly increasing sequence of sublevel graphs of G beginning at $(V, \varnothing)$ and ending at $G = (V, E)$.

Next, we will introduce notions of a-connectedness and maximally a-connectedness through which we will be able to aptly define Time Series Merge Trees. That is, maximally a-connectedness will establish the ready equivalence relation that will create our tree-like hierarchical structure from our sublevel graphs.

Definition 2.35 (a-Connected)**.** Given an acyclic graph $G = (V, E, f)$ and $a \in \mathbb{R}$ say that two vertices $v, w \in V$ are a-connected when any path in G between them contains no weight exceeding a. Additionally say v and w are maximally a-connected when any path in G extending an a-connected path between them is not itself a-connected.

Definition 2.36 (Time Series Merge Tree)**.** For a connected, weighted graph $G = (V, E, f)$ the relation of being maximally a-connected induces a refinement of partitions of V. Recall that a refinement of a partition P is P' such that $P \subseteq P'$. The refinement has the structure of a rooted tree called the merge tree T_G of G, with V as the root, $v : v \in V$ the leaves, and internal vertices being the maximally a-connected components of G induced by its edge weights.

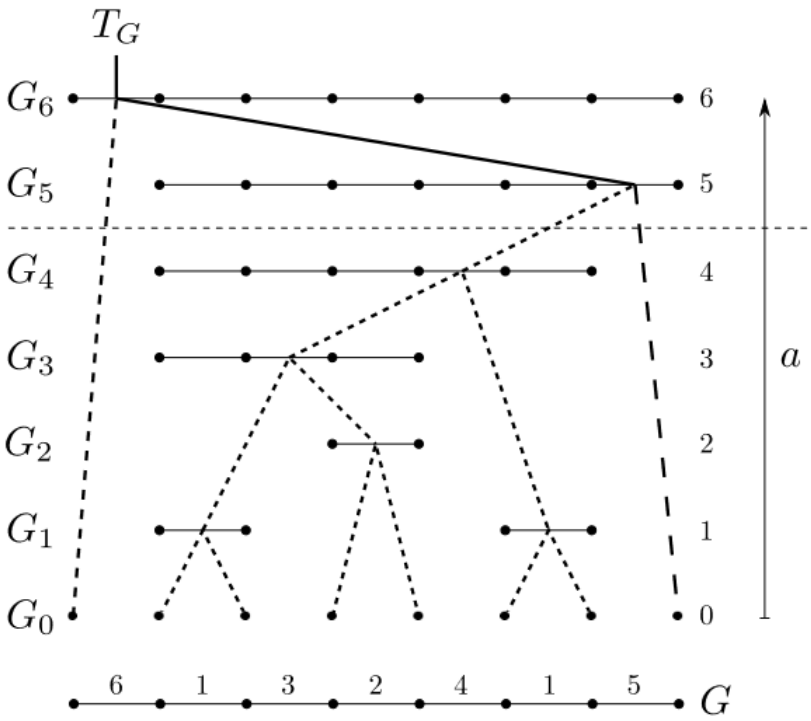


Figure 15: Here is a Time Series Merge Tree computed from the weighted time series G with y-values given by the edge weights. (Stephen 28)

2.7 Barcodes and Interchangeability

In TDA-literature, the computation of merge trees in general is often done to produce another object, namely, *barcodes* or *pile of stems* or its *persistence diagram*. The role of these diagrams lies in *persistent homology* which is a framework in computational topology to measure topological features of data that persists across scales

and has found wide applicability for studying data networks in fields as disparate as biology and the social sciences(Aktas, Akbas, and Fatmaoui 1).

The focus of this thesis will primarily be in working with and constructing these persistence diagrams. Here, we briefly show their construction which simply follows from a merge tree and drawing stems appropriately as long from the merge tree as seen in Figure 16.

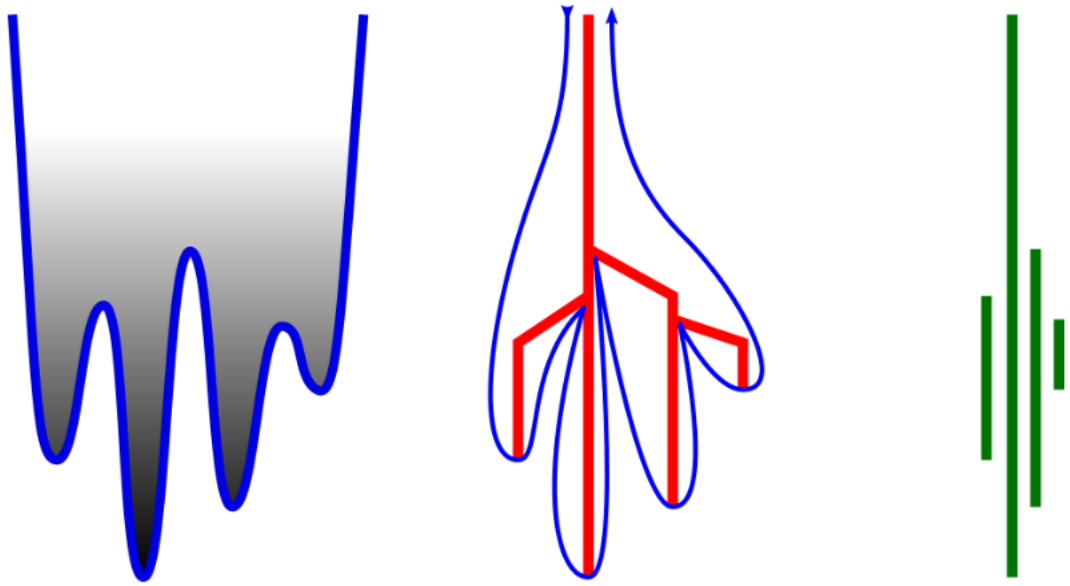


Figure 16: Here is an example of a pile of stems (right-most) from a merge tree (middle) of a continuous function (left, with grey-shading to indicate how to vertically construct the tree) (Baryshnikov 3)

.

There is a useful connection between time series merge trees and level set trees that has not yet been made apparent in the literature. Earlier, Colin Stephen had noted that Horton Pruning his *horizon visibility graphs* (via duality), to be discussed in the following section, yielded the branch structure of the merge tree associated to its finite time series' piecewise linear interpolation. Moreover, he showed that the *Elder Rule* on the first Horton pruning of the *persistence weighted* horizon visibility

graph (the graph with edge lengths copied over from the weighted merge tree which copies over edge lengths from the original weighted time series), yields the barcode of the piecewise linear interpolation (Stephen 28). This inspires our observation, which follows along the similar lines of proof. We use the work of Jussi Klemelä as reference for the alternative characterization of Level-Set Trees used in our proof (Klemelä 13).

Proposition 2 (Interchangeability of Time Series Merge Trees and Level Set Trees). The first Horton pruning of the time series merge tree is the Level-Set Tree of the inverted time series.

Proof. Let our time series' linear interpolation be denoted f from the closed interval $I \subset \mathbb{R}$ to $\mathbb{R}$. Recall that the Level-Set Tree of the inverted time series describes the structure of the level-sets $\mathcal{L}_\alpha$ as a function of the threshold α where

$$\mathcal{L}_\alpha(f) = \{x \in I : f(x) \geq a\} \tag{13}$$

$$\mathcal{L}_\alpha(-f) = \{x \in I : f(x) \leq a\} \tag{14}$$

That is, the Level-Set for any particular α is a disjoint union of subintervals of I. The nodes are identified with the connected components of the level sets and child-parent relations correspond to set inclusion, and so the root corresponds to the support I.

Now, recall the definition of time series merge trees as characterized by their sub-level sets. That is, time series merge trees describe the structure of merge sets of the weighted time series graph, $\breve{f}$, with (bijective) weight function W, under the relation, $\sim_{\hat{a}}$ of being maximally a-connected. Maximality implies that we only consider those paths $Path_{\breve{f}}(v, w)$ such that no edges extending it are also a-connected, we denote this with the relation loosely with $\leq_\sim$. The merge sets, M_α, given that $1 \leq i, j \leq N$,

are then of form

$$\begin{aligned} M_\alpha &= \{v_i, v_j \in V : v \sim_{\hat{a}} v_j\} \\ &= \{v_i, v_j \in V : \forall e_k \in \mathsf{Path}_{\check{\mathsf{T}}}(v_i, v_j), e_k \leq_\sim a\} \\ &= \{v_i, v_j \in V : \forall W^{-1}(x_k) \in W^{-1}(\{x_i, ..., x_j\}), W^{-1}(x_k) \leq_\sim a\} \end{aligned} \tag{15}$$

Now, if we bijectively associate each vertex to its immediate edge we have that

$$\begin{aligned} M_\alpha &\cong \{W^{-1}(e_{v_i}), W^{-1}(e_{v_j}) | v_i, v_j \in V : \forall W^{-1}(x_k) \in W^{-1}(x_i, ..., x_j), W^{-1}(x_k) \leq_\sim a\} \\ &\cong \{W^{-1}(e_i), W^{-1}(e_j) | e_i, e_j \in E : \forall W^{-1}(x_k) \in W^{-1}(x_i, ..., x_j), W^{-1}(x_k) \leq_\sim a\} \\ &\cong \{x_i, x_j \in x_1, ..., x_N : b \nparallel f(\{x_i, ..., x_j\}, b \leq_\sim a\} \end{aligned} \tag{16}$$

Applying the intermediate value theorem on account of the continuity of f we have that

$$\begin{aligned} M_\alpha &\cong \{x \in I : f(x) \leq a\} \\ M_\alpha &\cong \mathcal{L}_\alpha(-f) \end{aligned} \tag{17}$$

The time series merge tree is constructed similarly to level set trees with the root corresponding to the set of all vertices, or isomorphically the support I, the internal vertices corresponding to the connected components of the merge sets, child-parent relations being set inclusion, etc. However, time series merge trees additionally connect as the leaves of the tree all of the individual vertices as well. Horton pruning removes these extra leaves and we are left with what is isomorphic to the level set tree of the inverted time series. □

Essentially, this proposition reveals what is mysterious about the Horton pruning operation in the original paper by Colin Stephen. The mystery is nothing more than

the Time Series Merge Tree's additional construction of including the (extraneous) leaves. The reconstruction result of the Horton pruning on Stephen's horizon visibility graphs corresponds so well to the piecewise linear interpolation precisely because the merge tree adds this extra information which Horton pruning removes. We argue then, that the focus of such study should be turned towards the level-set tree object as opposed to the time series merge tree which obfuscates the real relations between things.

2.8 Visibility Approaches

Previously, we have discussed tree-based constructions that are amenable for time series analysis. However, there has been another geometric approach to time series analysis involving *visibility alogrithms* and their consequent *visibility graphs*. Our aim here is to provide introductory background to these methods and illuminate their connections to and their compatibility with tree methods as a prelude to our own *tunnelability* algorithms.

The first introduction of such methods began with the general *visibility algorithm* and *visibility* graph which were introduced by Lacasa et. al. in 2008 (Lacasa et al. 16). The basic idea is represented in Figure 17. Two points in a time series are considered *visible* if a straight line can be drawn between them without being intersected by any of the bars representing the data points. The resultant visibility graph summarizes the visibility of all of the points with each node in the graph corresponding to a data point and each edge corresponding to the visibility between them. The graph then is invariant under translation, horizontal and vertical rescaling, and critically, the addition of a linear trend in the data (Lacasa et al. 16).

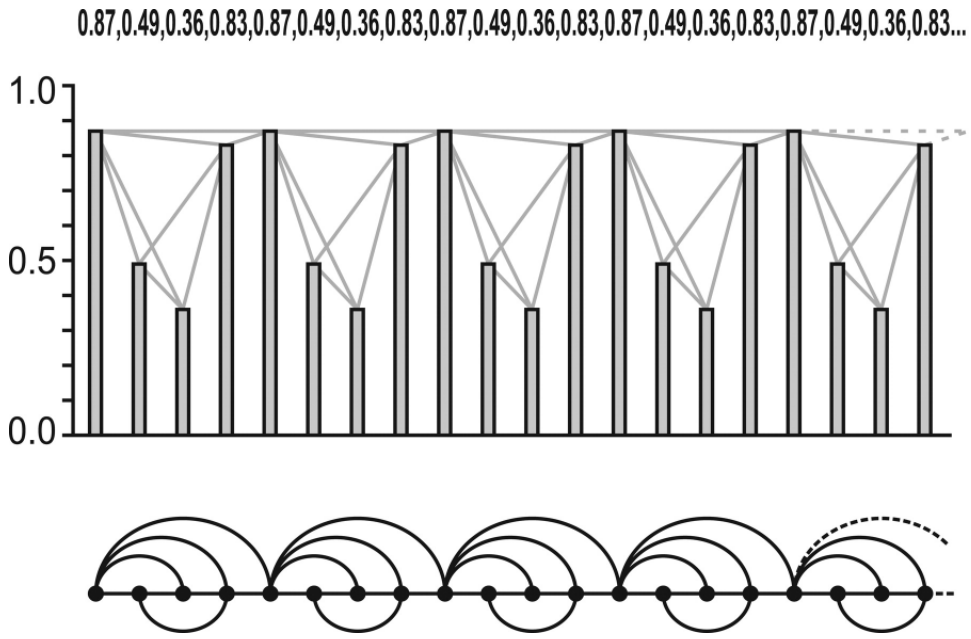


Figure 17: Here we depict the visibility of bars in a time series and its associated visibility graph below (Lacasa et al. 16)

.

Furthermore, it became possible to study time series through such a graph by its *degree distribution*, the *mean path length*, and *mean degree*, among other measures. Discriminatory analysis and distributional identification of time series is now possible through such graph analysis and we discuss some implications in Figures 18 and 19.

Definition 2.37 (Mean Degree)**.** The mean degree of a graph is the average degree of all of its nodes. As a function of n-nodes, the mean degree is computed as the mean degree of the subgraph of the first n nodes.

Definition 2.38 (Mean Path Length (Asif et al. 2))**.** The mean (average) or characteristic path length of a graph is the average number of steps along the shortest paths to traverse between all possible pairs of nodes. Suppose we have a graph G with vertices v_i, v_j denote $d(v_i, v_j)$ as the smallest number of connected edges needed to traverse from v_i to v_j. Then, the mean path length of the whole graph given it has

N total vertices is

$$l_G = \frac{1}{N(N-1)} \sum_{i \neq j} d(v_i, v_j) \tag{18}$$

The mean path length considered as a function of the total number of vertices is computed by computing l_G for the subgraph consisting of the first n nodes. That is, $l_G(n) = l_{G_n}$ where G_n is the subgraph of the first n vertices.

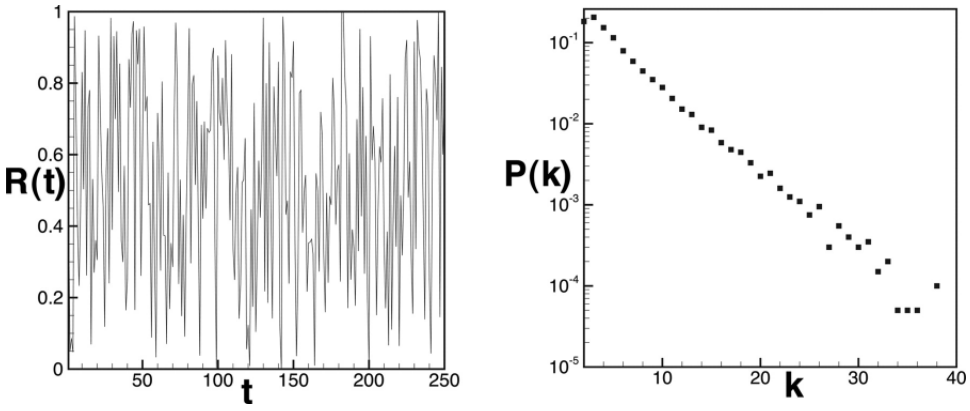


Figure 18: On the left we have a random time series. On the right we have its visibility graph's degree distribution showing the probability (in semilog) of observing a node of degree k. As observed by its tail, the degree distribution is exponential and in general we expect random time series to have exponential (visibility) degree distributions (Lacasa et al. 16).

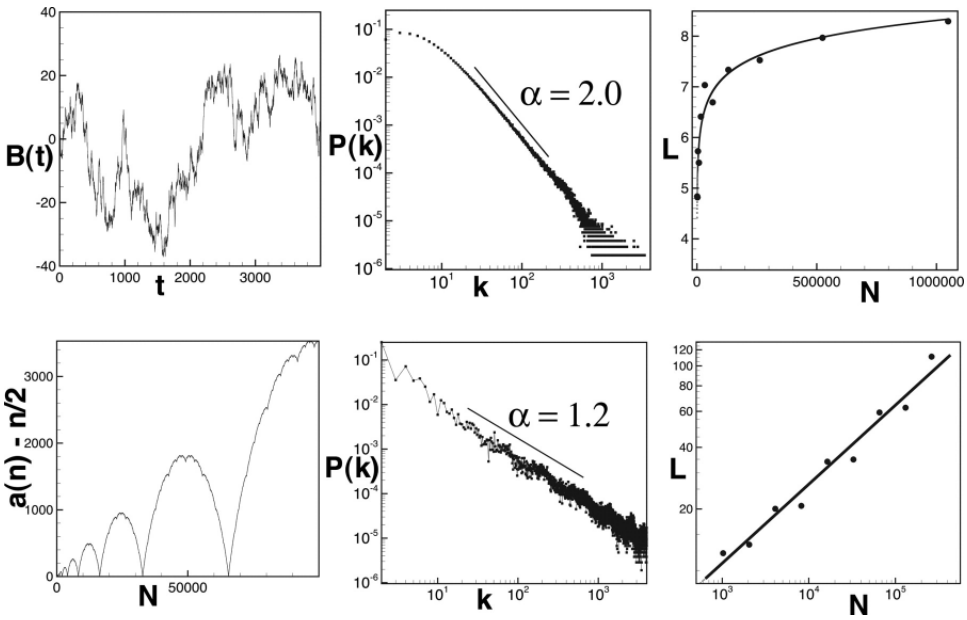


Figure 19: On the far left we have a Brownian and a Conway time series. In the middle we have their degree distributions, both power laws with different exponents α, and on the right we have their mean path lengths(Lacasa et al. 16).

Something worth considering about the graphs in Figure 19 is that the visibility graph for Brownian motion exhibits the *small-world property* and that both visibility graphs are *scale-invariant*. These are key metrics in computational graph theory and we give some brief introduction to these metrics here.

Definition 2.39 (Clustering Coefficient (Watts and Strogatz 32))**.** Let $|A|$ denote the number of items in the set A. For a graph $G = (V, E)$ with n-many vertices, the neighborhood N_i for a vertex v_i is its immediately connected neighbors.

$$N_i = \{v_j : e_{ij}, e_{ji} \in E\} \tag{19}$$

where e_{ab} denotes the edge connecting vertex v_a to vertex v_b. Then, the local clustering coefficient for a given vertex v_i is given as

$$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{|N_i|(|N_i| - 1)} \tag{20}$$

The global clustering coefficient, $\bar{C}$, of the graph is then the average of its local clustering coefficients.

$$\bar{C} = \frac{1}{n}\sum_{i=1}^{n} C_i \tag{21}$$

Definition 2.40 (Small-World Property(Mehlhorn and Schreiber 21))**.** A graph has the small-world property when it has a high clustering coefficient but a small mean path length. Typically, this means that the mean path length grows logarithmically but the clustering coefficient is large (usually more than 0.5).

Definition 2.41 (Scale-Invariant Graph)**.** A scale-invariant or a scale-free graph is one whose degree distribution follows (asymptotically) a power-law distribution.

Definition 2.42 (Graph Self-Similarity)**.** Let the diameter of a graph, $G = (V, E)$ with N vertices, be $\hat{d} = max_{v,w\in V}(d(v, w))$. If the graph possesses the small-world property, then

$$N \sim e^{\frac{d}{l_G}} \tag{22}$$

Similarly, the graph is considered self similar if it exhibits a power law distribution with some notion of *scale*, often d, with some positive exponent α.

$$N \sim d^{-\alpha} \tag{23}$$

Although, the two notions are seemingly disparate there is growing literature showing that under appropriate renormalization procedures complex graphs can be self-similar while also being small-world(Song, Havlin, and Makse 27).

For the purposes of this thesis, we will be primarily working with variations of this visibility idea that aim to make analysis simpler. Real-world time series have thousands to even millions or billions of data points. Generating a graph with this many nodes and even more edges makes the analysis computationally expensive if

not impossible. A simplification of the earlier approach begins with *horizontal* visibility(Luque et al. 19). This simplification is made apparent in Figure 20.

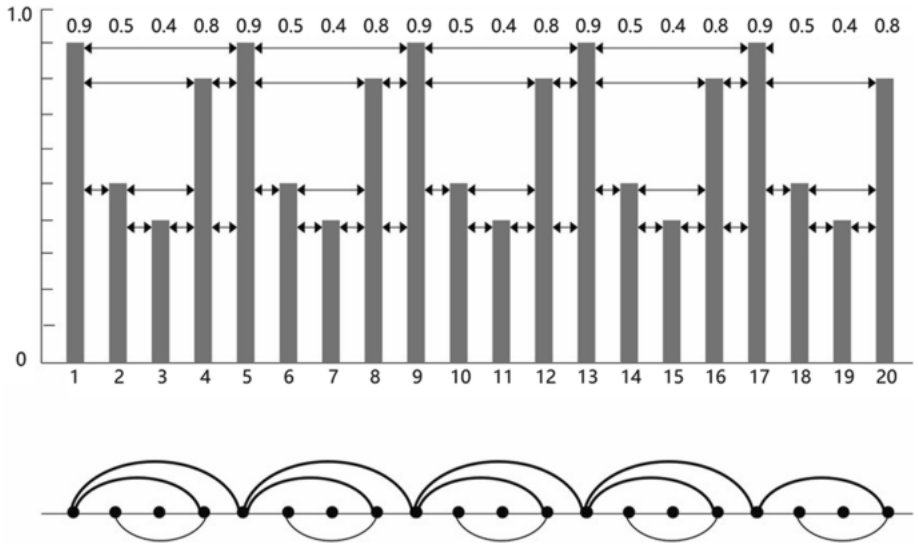


Figure 20: Here we depict the horizontal visibility of bars in a time series and its associated Horizontal Visibility Graph below (Luque et al. 19).

Considering only two are points horizontally visible if a horizontal line can be drawn without intersection with the bars significantly reduces the number of edges, and similar analysis can be done at greater speed. This is additionally sped up by the aid of closed-form formulas for various metrics such as the clustering coefficients, mean path lengths, mean degrees, and degree distributions. (Luque et al. 19). We posit an additional simplification that greatly reduces not just the number of edges, but primarily, the number of nodes. This is accomplished by our concept of *U-shaped segments* and so metrics may be computed not at every data point but instead on each U-shaped segment or subinterval. We provide characterizations of horizontal visibility and later, our own tunnelability, with both the per data point and per U-shaped segment construction.

Definition 2.43 (Horizontal Visibility)**.** Let $X \in \mathcal{E}^{\text{ex}}$ be a time series with distinct, finite extrema and n-many U-shaped segments lying on subintervals $U = \{I_1, ..., I_n\}$.

Consider the horizontal visibility function $\mathcal{HV} : (X, I_a, I_b) \to \{0, 1\}$ with $a, b \in \{1, ..., n\}$.

$$\mathcal{HV}(X, I_a, I_b) = \begin{cases} 1 & \text{if } \exists\, p_1 \in I_a, p_2 \in I_b \text{ s.t. } X(p_1) = X(p_2) \text{ and the horizontal} \\ & \text{line } x = X(p_1) \text{ has no intersection with } X \text{ on } (p_1, p_2) \text{ or} \\ & (p_2, p_1) \text{ if } p_1 > p_2 \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

Two points X_{t_1}, X_{t_2} with $t_1 \in I_a, t_2 \in I_b$ are *horizontally visible* if $\mathcal{HV}(X_t, I_a, I_b) = 1$ with the choice of $p_1 = X_{t_1}, p_2 = X_{t_2}$. They are termed *distinctly horizontally visible* if additionally $I_a \neq I_b$.

Then, the horizontal visibility of a particular U-shaped segment I_c is

$$\mathsf{HV}(I_c) = \sum_{\substack{I_i \in U \\ I_i \neq I_c}} \mathcal{HV}(X, I_c, I_i) \tag{25}$$

Definition 2.44 (Horizontal Visibility Graph)**.** The Horizontal Visibility Graph on Data Points is constructed by assigning each data point a vertex and connecting edges where data points are horizontally visible. The Horizontal Visibility Graph on U-Shaped Segments is constructed by assigning each U-shaped segment a node and connecting edges where U-shaped segments are horizontally visible.

2.9 Duality of Time Series Merge Trees and Horizon Visibility

As an overture to our duality results, we briefly summarize Colin Stephen's earlier work (Stephen 28) establishing the duality of Time Series Merge Trees and *Horizon Visibility Graphs*.

Definition 2.45 (Horizon Visibility Graph (Stephen 28))**.** Given a time series $\tau = (x_1, ..., x_n)$ its horizon visibility graph $\mathsf{HVG}_\infty(\tau)$ is defined to be the horizontal visibility graph of $\tau_i nfty = (\infty, x_1, ..., x_n, \infty)$.

The horizon visibility graph, thus, is simply the horizontal visibility graph with two additional vertices representing the infinite past and the infinite future.

Theorem 2.2 (Horizon Visibility Graphs are Dual to Merge Trees (Stephen 28))**.** *Given a time series* $\tau = (x_1, ..., x_N)$ *its horizon visibility graph* $\mathsf{HVG}_\infty(\tau)$ *is exactly the dual of its merge tree:* $\mathsf{HVG}_\infty(\tau) = T^*_{\check{\tau}}$.

Just as horizontal visibility graphs can be analyzed by the metrics listed above and can later be used to estimate dynamical parameters such as reversibility, the Lyapuvnov exponent, and Hurst exponent amongst many others, so too can this be established with horizon visibility graphs (Stephen 28). There are additional benefits however, other than just the theoretical benefit of being able to work with either HVGs or time series merge trees. The addition of information about the past and future allows horizon visibility to detect the difference between leading and trailing trends, are more sensitive to monotonic sub-sequences, and can similarly be reconstructed from degree sequences (Stephen 28).

3 Horizontal Tunnelability of a Time Series

We now introduce our main contribution to this area, the notion of *horizontal tunnelability*. As discussed previously, the main advantages are the reduction in the size of the graphs— U-shaped segments reduce the number of vertices and tunnelability itself is shown to be a sub-relation of visibility. We give exact details of the construction of these objects and discuss their applicability.

3.1 Horizontal Tunnelability

Horizontal tunnelability is extremely similar to horizontal visibility, the main difference being that two points are tunnelable if and only if they are visible and they pass through a *mountain*, that is, a portion of the linear interpolation of the time series that lies above the two points. As such, U-shaped segments dictate the *valleys* between the mountains with the sole exceptions of those segments corresponding to the implied unenclosed valleys at the beginnging and end of the excursion.

We define this term here along with the subsequent horizontal tunnelability graph. This graph construction, we will later show, bears a unique relevance to level-set trees. As such, we obtain interchangeability results to and from time series merge trees via Horton pruning.

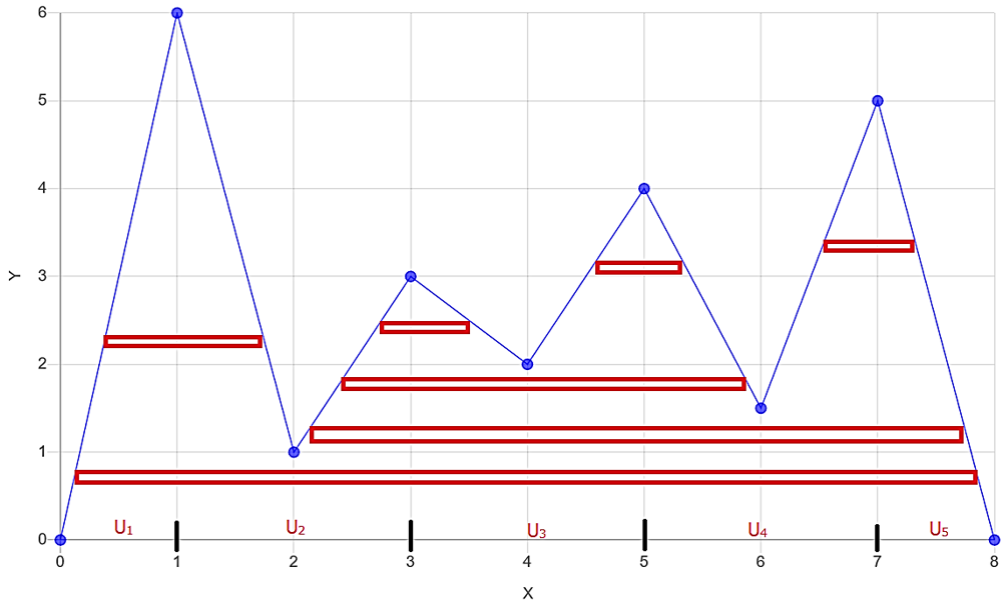


Figure 21: We graphically depict the U-shaped segments (via the labeled subintervals) of the time series in blue and label horizontally tunnelable points via red tunnels.

Definition 3.1. (Horizontal Tunnelability) Let X be a time series as before with the same U-shaped segments. Horizontal tunnelability is essentially horizontal visibility

with the added condition that the time series always be *above* the two visible points. More formally, consider the horizontal tunnelability function $\mathcal{HT} : (X, I_a, I_b) \to 0, 1$ with $a, b \in 1, ..., n$.

$$\mathcal{HT}(X, I_a, I_b) = \begin{cases} 1 & \text{if } \mathcal{HV}(X, I_a, I_b) = 1 \text{ with some choice of } p_1, p_2 \text{ s.t.} \\ & X > X(p_1) \text{ on } (p_1, p_2). \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

Two points X_{t_1}, X_{t_2} with $t_1 \in I_a, t_2 \in I_b$ are termed *horizontally tunnelable* if $\mathcal{HT}(X_t, I_a, I_b) = 1$ with the choice of $p_1 = X_{t_1}, p_2 = X_{t_2}$. They are termed *distinctly horizontally tunnelable* if additionally $I_a \neq I_b$.

Then, the horizontal tunnelability of a particular U-shaped segment I_c is

$$\mathsf{HT}(I_c) = \sum_{\substack{I_i \in U \\ I_i \neq I_c}} \mathcal{HT}(X, I_c, I_i) \tag{27}$$

Remark. There are some time series for which horizontal visibility on U-shaped segments is the same as its horizontal tunnelability. These are those time series who do not possess "valleys"- that is, a U-shaped segment that is smaller in maximal height than the U-shaped segments adjacent to it.

3.2 Horizontal Tunnelability Graphs

One can sufficiently invert U-shaped segments to obtain a time series for whose horizontal tunnelabilities is precisely its horizontal visibilities. Clearly, if something is horizontally tunnelable it is also horizontally visible.

Definition 3.2 (Horizontal Tunnelability Graph)**.** Denoted HTG, it is the graph with vertices associated to U-shaped segments possessing edges connecting them wherever

the segments are horizontally tunnelable. Similarly, HTG for data points is defined as the graph with vertices per data point and edges wherever two points are horizontally tunnelable.

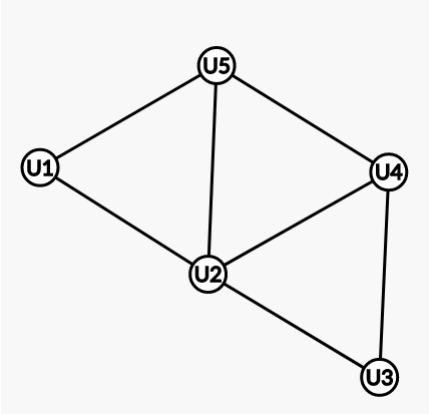


Figure 22: We compute the horizontal tunnelability graph constructed from the prior time series. Observe that the edges correspond to the tunnelability of each of the U-shaped segments of the time series (ex: U_1 is tunnelable, connected, to U_2 and U_5.

Lemma 3.1. *The horizontal tunnelability graph is a subgraph of the horizontal visibility graph of the time series.*

Proof. This immediately follows from the fact that any two points and any two U-shaped segments are horizontally tunnelable if and only if they are also horizontally visible. We attain a *proper* subgraph for those time series that do not possess *valleys* as per the prior remark. □

We conclude this section by emphasizing the increased utility of horizontal tunnelability. The smaller graph we aim to show retains many of the same important capabilities as others. We would like to additionally highlight that our construction is the first to make use of the additional geometric information imparted by the linear interpolations of time series. The horizontal tunnelability concept is one that is more suited to fields in which convexity or concavity are important measures.

4 Dual of a Level-Set Tree

Our discovery of horizontal tunnelability as a concept begins first with the study of duals of level-set trees. In doing so, we have given a characterization of the duals of rooted, binary trees and how to construct them that is not found in existing literature. It is this characterization that later helps motivate our algorithm to generate these duals.

4.1 Duals of Rooted Binary Trees

The general characterization of dual of rooted binary trees follows from the general construction of rooted, binary trees. We will first begin with the minimal rooted, binary tree with two leaves. We additional embed all of our trees and duals in the disk for clarity. To construct the dual, we similarly begin from the minimal triangle dual.

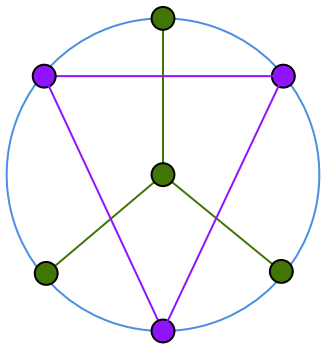


Figure 23: We begin with the minimal rooted binary tree (in green) and the minimal dual (in purple).

Next, to iterate another rooted, binary tree instance, we arbitrarily append to a leaf a *v-branch*. Similarly, for the dual we add a new point onto the circle and connect two edges to its neighbor points keeping all prior edges. This new point is arbitrary but for sake of clarity we choose those points that correspond to our tree construction in our figures. We iterate this process for both the tree and the dual.

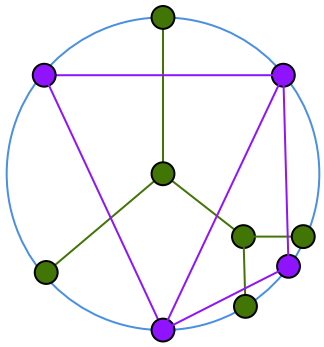


Figure 24: The construction iterates by adding v-shaped branches in the green tree and associated *triangles* in the purple dual.

By iterating this process, we may generate any rooted, binary tree and any dual of a rooted, binary tree. This works because each face-vertex of the dual corresponds precisely to a v-branch and each face-vertex necessarily connects to its two neighbor edges (for binary trees). No other edges are possible in the dual as faces are only formable by correspondent v-branches.

4.2 The Dual of a Level-Set Tree is the Horizontal Tunnelability Graph on U-shaped Segments

We now continue to establish the main result of our thesis— that is, that level set trees are dual to our horizontal tunnelability graphs. We will begin with a few helpful lemmas that will make this correspondence clear.

Lemma 4.1. *Nonzero distinctly horizontally tunnelable points on a time series, $X_t \in \mathcal{E}^{ex}$ with distinct, finite extrema and no consecutively repeated values, either share an edge in its level set tree $T = \mathsf{Level}(X_t)$ or have the same value as some local minima in the time series.*

Proof. Suppose two points, $X(t_1) = X(t_2) \neq 0$ are distinctly horizontally tunnelable with $t_1 \in I_a, t_2 \in I_b, t_2 > t_1$ and $I_a \neq I_b$ being subintervals of associated U-shaped

segments. Since the two points are horizontally tunnelable, for all $t \in (t_1, t_2)$, X_t is always above $X_{t_1} = X_{t_2}$. If this were not the case, then by the Intermediate Value Theorem (since all piecewise linear interpolations are continuous), there would be some $t_3 \in (t_1, t_2)$ such that $X_{t_3} = X_{t_1} = X_{t_2}$. But this would violate the horizontal tunnelability (and visibility) of X_{t_1} and X_{t_2}.

In the case that the time series is always above the horizontally tunnelable points, the infimum on $[X_{t_1}, X_{t_2}]$ is exactly $X_{t_1} = X_{t_2}$. Hence, by the metric d_X on the level set tree T, $X_{t_1} \sim X_{t_2}$ because

$$d_X(t_1, t_2) = X_{t_1} - X_{t_1} + X_{t_1} - X_{t_1} = 0. \tag{28}$$

A two point equivalence class bijectively corresponds to edge portions meaning that $u_T(t_1)$ and $u_T(t_2)$ simply map to the left and right (resp.) portions of the same edge in the level set tree. Alternatively, the two horizontally tunnelable points could instead belong to a 3+ point equivalence class in which case $X_{t_1} = X_{t_2}$ is a local minima value obtained somewhere along the time series. □

Remark. It is important for our theorem to require horizontal tunnelability as opposed to horizontal visibility because otherwise there would no longer be a guaranteed equivalence class to which the two points would belong and so they could lie on very different edges in the tree.

Lemma 4.2. *Suppose that there exists a pair of nonzero distinctly horizontally tunnelable points (X_{t_1}, X_{t_2}) on a time series, $X_t \in \mathcal{E}^{ex}$ with distinct, finite extrema and no consecutively repeated values. Let this pair correspond to the 3 point equivalence class of local minima and further let $t_1 \in I_a, t_2 \in I_b$ where I_a, I_b are support subintervals of distinct U-shaped segments. Then, there exists another pair of points (X_{t_3}, X_{t_4}) with $t_3 \in I_a, t_4 \in I_b$ that are also nonzero distinctly horizontally tunnelable but instead*

correspond to the 2 point equivalence class of a shared edge.

Proof. Since the time series has distinct, finite extrema, the set of local minimal y-values has measure 0. Thus, on the U-shaped segment containing X_{t_1}, one can choose a point slightly below that y-value that does not have the same height as some local minima. The same can be done for the U-shape segment containing X_{t_2} such that the newly chosen points (X_{t_3}, X_{t_4}) are nonzero distinctly horizontally tunnelable. This can only not be done in the case that one of the points X_{t_1} or X_{t_2} is itself a local minima, in which case choose (X_{t_3}, X_{t_4}) to be slightly above. There is always space to choose points above and below so long as our original pair is nonzero and because real intervals are dense and have nonzero measure. The original pair of points cannot both be local minima because the time series has distinct extrema. □

Lemma 4.3. *U-shaped segments in the time series X correspond to the borders of the "faces" formed by the embedding of $T = \mathsf{Level}(X)$ into D^2, or alternatively, to the vertices of the dual graph.*

Proof. Let there be n U-shaped segments with support on subintervals $U = \{I_1, ..., I_n\}$ and denote the local maxima of X in order left-to-right as occurring at $t_1, ..., t_n$. Consider the images of the tree traversal function u_T on each of these subintervals. $u_T(I_1) = u_T([0, t_1])$ so this is the path of leftmost edges from the root to the leftmost leaf on the level set tree (since each leaf corresponds to a local maxima). For $I_2, ..., I_{n-1}$, the images of the tree traversal function on these subintervals is the paths of edges between adjacent leaves in depth-first search order. Similarly, for I_n the appropriate image is just the path of edges from the rightmost leaf down to the root. Thus all the images are depth-first search paths between either the root to the leftmost leaf, between a leaf to the right adjacent leaf, or from the rightmost leaf to the root. When the level set tree is embedded into D^2, we place the root and leaves onto S^1. Between any two of these points on S^1, there is exactly one, unique portion of

the circle connecting them. This portion, along with the tree traversal image of the given subinterval, forms the border of the "faces" and hence the nodes of the dual graph. □

Theorem 4.4. *Suppose we have a time series $X_t \in \mathcal{E}^{ex}$ with distinct, finite extrema and no consecutively repeated values. The horizontal tunnelability of its U-shaped segments is given precisely by the degree sequence of the dual of its level set tree. Moreover, the horizontal tunnelability graph itself is precisely the dual of its level set tree.*

Proof. According to Lemma 4.3, each U-shaped segment corresponds to a vertex in the dual graph. Moreover, by the definition of the dual, the number of edges forming the borders of each vertex-face not lying on S^1 is exactly the degree of that vertex. By Lemmas 4.1 and 4.2, the horizontal tunnelability of each U-shaped segment is the number of edges shared on the level set tree by the tree traversal image of that U-shaped segment. Hence, the U-shaped segments' horizontal tunnelabilities is precisely given by the corresponding degrees of the dual graph's vertices. □

Remark. Suppose that the level set tree be drawn with the root at the bottom and the dual be drawn such that its nodes are located centrally amongst the "faces". Then, the U-shaped segment's horizontal tunnelabilities from left to right is given by the degrees on the dual in clockwise order starting from the appropriate vertex (which is always located near the root).

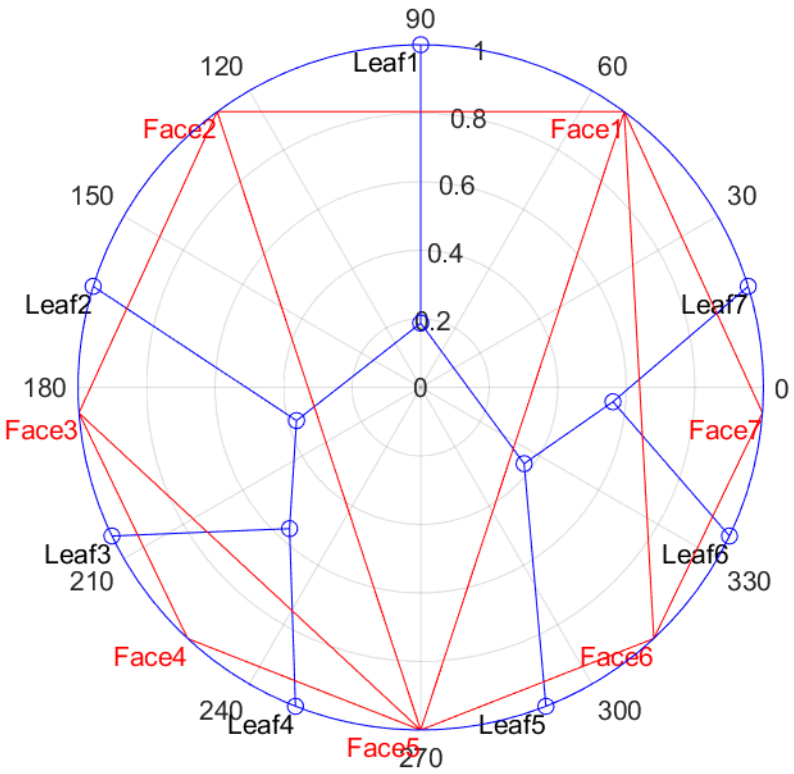


Figure 25: In Matlab, we compute the dual (in red) of a disk embedded level set tree (in blue). Note that the blue disk is not part of the tree.

There are some interesting properties of this dual graph. We summarize one such property here and leave the rest for empirical assay.

Lemma 4.5. *The degree function f of the dual graph of a binary, planted tree $T \in \mathcal{T}^{|}$ has a fixed point $f(2) = 2$. Further, $f(0) = f(1) = 0$ and $f(k_{max}) = 1$ or $f(k_{max}) = 2$ where k_{max} is the maximal degree of the tree.*

Proof. Since T is a binary, planted tree, the two edges going to the left and right from the planted edge demarcates two faces, ergo, two vertices in the dual graph. This is because the left edge will lead to the leftmost vertex of the tree which will necessarily be on S^1 upon embedding the tree into D^2. The same is true for the rightmost vertex. □

5 Empirical Studies

We have written Matlab code (see Appendix) to produce the duals of our level set trees, the horizontal tunnelability graphs. The code is written with some help from tinevez's `@tree` package (Tinevez 29) and the `level_set_tree.m` function from Zoe Haskell's Ph.D thesis (Haskell 11) code. This code is available here https://tinevez.github.io/matlab-tree/ and here https://zenodo.org/record/4302471 respectively. The totality of our code is hosted here https://github.com/pikhan/Research.

In similar fashion to prior literature, we compute the following metrics:

1. Local and Global Clustering Coefficients
2. Local and Global Mean Path Lengths
3. Local and Global Mean Degrees
4. Degree Distributions and Degree Sequences

All of our experiments were conducted by generating level set trees from simulated time series of 1000 data points. We additionally note that our results remained stable with very small differences in numbers or image quality compared to the large ones we see between different distributions. In general, we believe the results of our experimentation validates the success of this method in being able to perform the same functions as its predecessors at a fraction of the computational cost.

Our actual code is, however, somewhat slow, but not due to the horizontal tunnelability object structure, rather it is primarily due to the computation of some of our metrics. For example, we use the Floyd-Warshall algorithm (Ingerman 12) which has complexity $O(V^3)$ for computing the average path length for a graph with V-many vertices, which we run recursively to compute the mean path length as a function of n

(Rosen 24). Implementing Dijkstra's would be much faster and an improvement that, along with many other speedups, would improve the versatility of the code. Despite, this we were able to run each of the simulations in seconds on an old Lenovo laptop. We tested the following time series:

1. Uniform Random
2. Brownian (Gaussian) Random Walk
3. Fractional Brownian Motion with Hurst exponent 0.7
4. Time Series with Linear Trend (slope=0.1) and Noise
5. Lorenz Attractor with $\sigma = 10, \beta = \frac{8}{3}, \rho = 28, x_0 = 0.1, y_0 = 0, z_0 = 20$

5.1 Single-Value Metrics

We summarize some of our results in the following table.

Time Series	Mean Clustering Coeff.	Mean Path Length	Mean Degree
Uniform Random	0.9827	2.7150	94.3473
Brownian	0.9829	1.9644	123.0996
Frac. Brownian	0.9149	3.0454	21.5869
Linear Trend	0.3442	2.1068	2.9091
Lorenz	0.9138	1.4744	6.6154

Table 3: Here is a small summary of our single-valued experimental results.

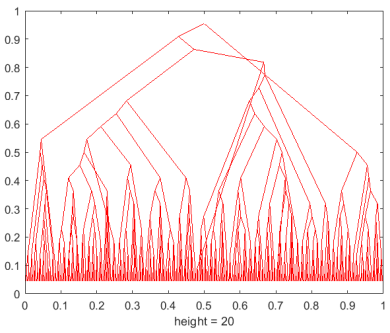
We easily observe a sharp difference in the mean clustering coefficient between our time series with linear trend and the other, more chaotic series. The mean path length and mean degree in particular, are much more sensitive measure metrics. Time Series with linear trends, in particular, are easily detectable by all three single-value metrics. Although, we did not compute any closed form formulas for these metrics or the others discussed below, we believe a similar approach as before (Luque et al.

19) could be undertaken and one may similarly also estimate dynamical parameters such as the Hurst exponent.

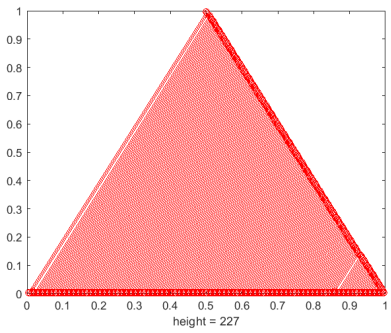
In particular, observe the fractional Brownian motion results. With Hurst exponent 0.7 we achieve the results as in Table 3. Although, not listed we also tested with many other Hurst exponents. We observe strong changes in behavior of our degree distributions and mean path lengths graphs between the regime of Hurst exponent being close to 0 versus 1. Typically, we see behavior such as the degree distribution exponentially rising and the mean path length plateaus after rising (in low Hurst exponents, opposite in high). We hint at a method for Hurst exponent estimation via a known relationship between fractional brownian noise and our plotted graph metrics. However, our single-value metrics are not enough to perform Hurst exponent estimation as empirically we do not observe any good correlations between Hurst exponent and our single-value metrics. However, we do observe consistency between distributions and by our results in general. The stability of our results implies a strong applicability for distributional discrimination and identification of time series data, especially by our plotted metrics.

5.2 Visual Discrimination by Duals and Trees

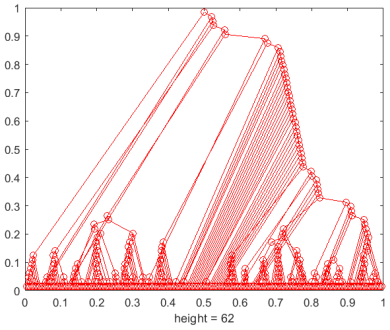
We now compare some of our duals and trees. There are remarkable, immediately striking differences between the trees and duals of these distributions.



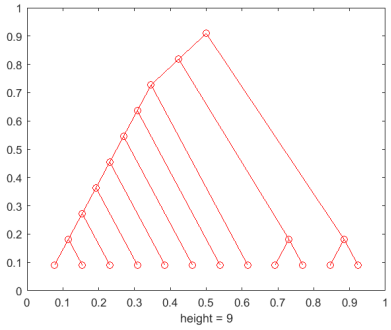
(a) Uniform Random Level Set Tree



(b) Linear Trend Level Set Tree

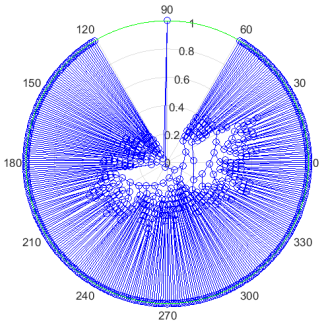


(c) Frac. Brownian Level Set Tree

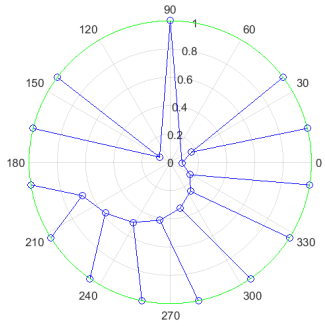


(d) Lorenz Level Set Tree

Figure 26: Here we provide a few plots of some of our computed level-set trees.



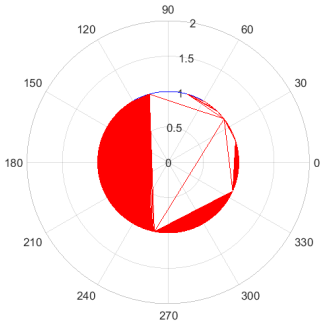
(a) Uniform Random Level Set Tree embedded in the disk



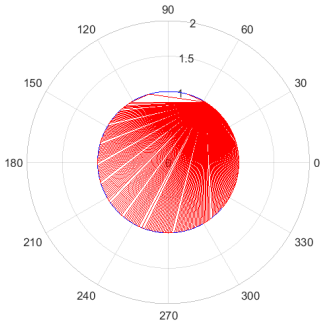
(b) Lorenz Level Set Tree embedded in the disk

Figure 27: Here are plots of some of our level-set trees embedded in the disk.

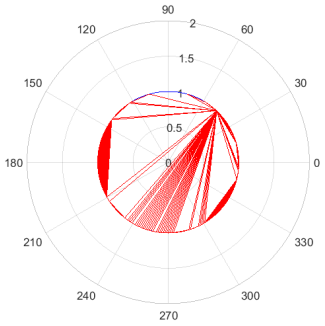
We now summarize the plots of our horizontal tunnelability graphs, which bear similar stark visual differences.



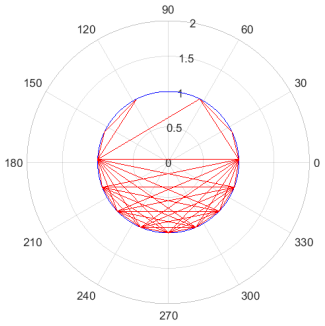
(a) Uniform Random HTG



(b) Linear Trend HTG



(c) Frac. Brownian HTG

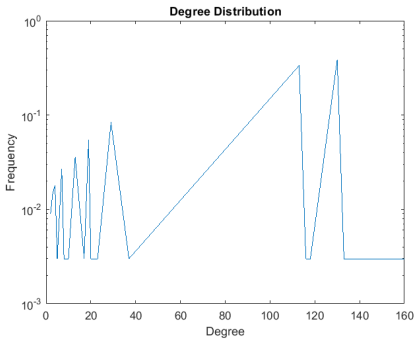


(d) Lorenz

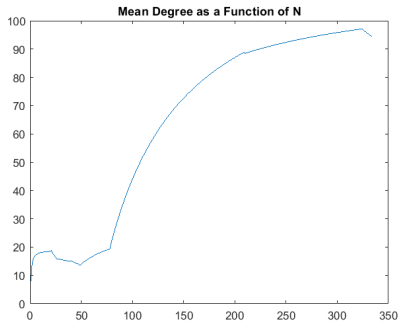
Figure 28: Finally, we plot in Matlab (in polar coordinates) some of our HTGs.

5.3 Plot Metrics

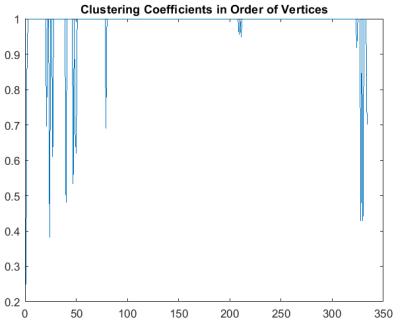
By far, however, our most informative metrics are the degree distributions, local clustering coefficients, mean path lengths, and mean degrees (as functions of n). Although, not computed, clustering *distributions* may also be computed (instead of just raw local clustering coefficients), although we did not find the clustering coefficient metric to be particularly informative in our experiments. We summarize our results below.



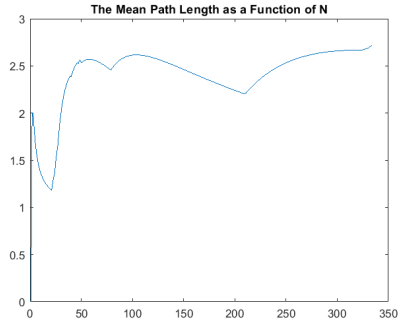
(a) Uniform Random HTG Degree Distribution



(b) Uniform Random HTG Mean Degree



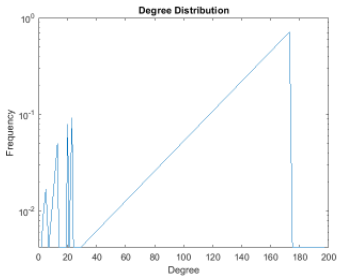
(c) Uniform Random HTG Local Clustering Coeff.



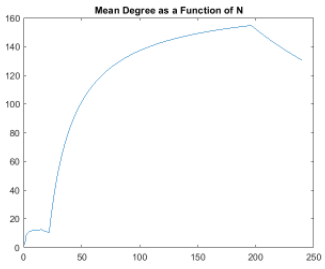
(d) Uniform Random HTG Mean Path Length

Figure 29: We compute four graph metrics for Uniform Random HTGs. These include their degree distributions in a semi-log scale for the probability (y-axis), the mean degree as a function of vertices included, the depth-first search ordered (from the level set tree) local clustering coefficients, and the mean path length as a function of vertices included in the analyzed subgraph.

As seen in Figure 29 we observe our random time series to have somewhat fat-tailed and somewhat exponential rising degree distributions. The mean degrees tend to increase and mean path lengths tend to a middle. On the surface, we see behavior similar to that of our Brownian HTGs in Figure 30 which are random (but normally distributed), but crucially observe that our Brownian HTG mean path lengths plateau much lower than 2.5 and exhibit stronger small-world property. Fractional brownian noise, linear trend, and lorenz time series show much more contrasting distributions, mean path lengths, and mean degrees.

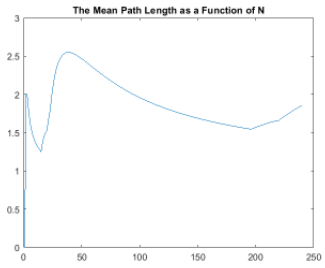


(a) Brownian HTG Degree Distribution



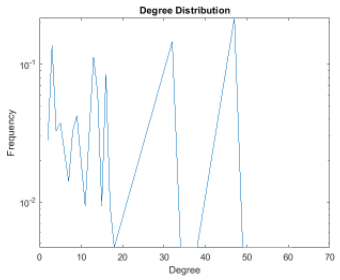
(b) Brownian HTG Mean Degree

(c) Brownian HTG Local Clustering Coeff.

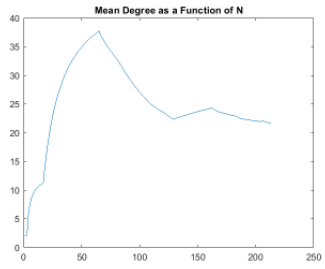


(d) Brownian HTG Mean Path Length

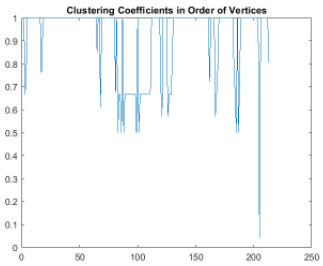
Figure 30: Here we compute the same metrics as before for Brownian HTGs. Notice, that similar to uniform random we exhibit similar fat-tailed behavior in our degree distributions and increase in our mean degree. This is what we expect, is similar to previous work with horizontal visibility, and is due to the existence of large outliers in the data.



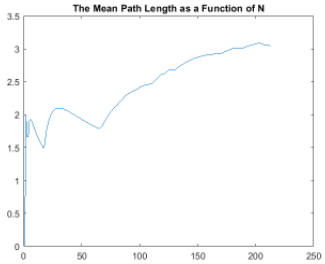
(a) Frac. Brown HTG Degree Distribution



(b) Frac. Brown HTG Mean Degree

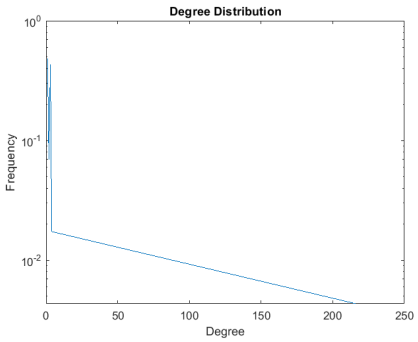


(c) Frac. Brown HTG Local Clustering Coeff.

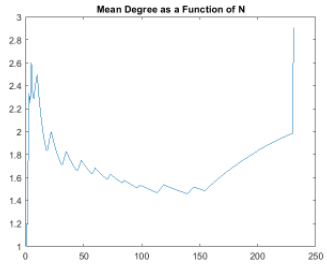


(d) Frac. Brown HTG Mean Path Length

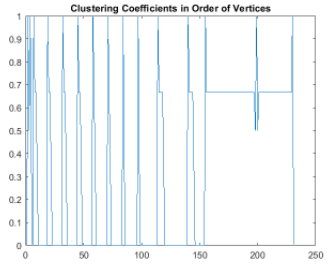
Figure 31: Observe that for Fractional Brownian motion, we observe strikingly different behavior. In particular, in Matlab we can see that our degree distribution can be well approximated by a power-law distribution (recall that the y-axis is semilog). Also observe that our mean path lengths tend to increase.



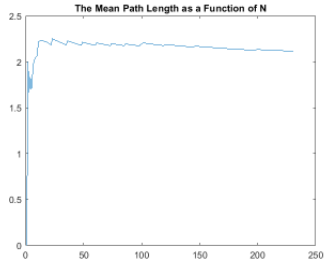
(a) Linear Trend HTG Degree Distribution



(b) Linear Trend HTG Mean Degree

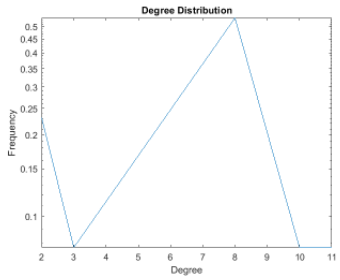


(c) Linear Trend HTG Local Clustering Coeff.

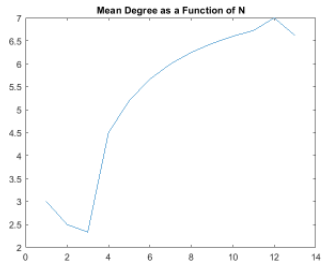


(d) Linear Trend HTG Mean Path Length

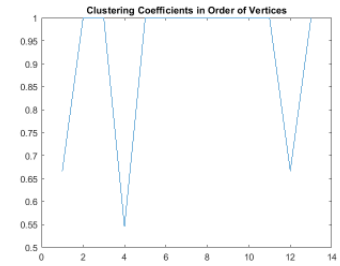
Figure 32: Almost no matter what method we use to assess linear trends, they are easy to pick out. Even here we see flat-lining mean path lengths and linearly decreasing degree distributions (along with U-shaped mean degree). This is precisely what we expect for a linearly trending time series and replicates similar results with horizontal visibility.



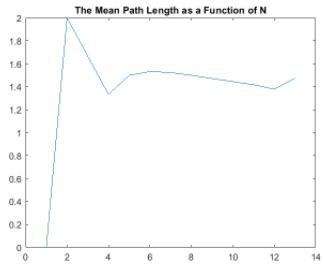
(a) Lorenz HTG Degree Distribution



(b) Lorenz HTG Mean Degree



(c) Lorenz HTG Local Clustering Coeff.



(d) Lorenz HTG Mean Path Length

Figure 33: Lorenz Attractor based time series produced interesting results, with a noticeable single major peak in the degree distribution.

6 Discussion

The development of our concept of horizontal tunnelability marks a noteworthy advancement in time series analysis, bridging the gap between disparate geometric approaches. By unifying methodologies like tree methods, visibility algorithms, and persistence-based barcodes, we have not only provided a novel perspective but have also succeeded in illuminating connections between previously unrelated work. The duality between the level set tree, obtained as the Harris path of a time series, and the time series' horizontal tunnelability graph, a subgraph of the horizontal visibility graph, is a particularly significant finding.

A critical aspect of our study is the exploration of computational algorithms for such geometric time series analysis. Algorithms such as the Floyd-Warshall algorithm brought computational challenges, affecting the overall performance of the code. Po-

tential improvements like implementing Dijkstra's algorithm (Dijkstra 8) could offer a more efficient computation balance between accuracy and speed. Visual discrimination between trees and duals of various distributions offers insight into the structural nuances of the data, providing opportunities for classification or understanding the underlying process generating the time series.

The sharp difference in metrics between various time series, such as the mean path lengths, degree distributions, and mean degrees between, for example, linear trend time series and other chaotic series, provides intriguing insights. These differences may be connected to the underlying characteristics of the time series, that may persist in scale-free resolutions.

One of the key strengths of our approach lies in its computational efficiency over prior methods, offering promising applications across various domains, including economics, climatology, and other fields requiring robust, geometric time series analysis. Reflecting on overall stability and consistency, our results validate the method while identifying limitations, such as specific cases where performance may be compromised. Future work could explore closed-form formulas, improving computational efficiency, or applying the method to different types or larger datasets. Particularly, we would like to test our methods on real-world data. Despite potential challenges in implementation, the real-world applicability of this research could lead to problem-solving in various disciplines.

In conclusion, the provision of empirical code bridges the gap between theory and practice, making our innovative approach accessible to practitioners and researchers alike. Further exploration and validation, especially in real-world applications, may yield deeper insights and refine our understanding of time series behavior. This thesis presents our main methods and some proof-of-concept simulations but also hints at far-reaching implications and future directions, such as a possible method for Hurst

exponent estimation.

Appendices

A Conjectures

Conjectures on Level-Set Trees

1. We know that $Level(f)$ produces a level-set tree corresponding to the function f and that moreover, local maxima (finite patterns of length 2 on $\{1, -1\}^n$: $(+1, -1)$) corresponds to its leaves. Local maxima are traditionally found by setting the derivative equal to 0, which correspond to our search for patterns of type $(+1, -1)$. This is because the intermediate value theorem states that going from a positive to negative derivative/slope implies that the zero lies between there. The same is true for the situation of (-1,1) patterns but this gives us minima and in the symbols used in Haskell's thesis the full state space is described by $\mathcal{B}_2$.

   Idea: Assess $Level(f')$ (the derivative) trees. Just like in the situation of $Level(f)$ trees, local maxima correspond to leaves but now, local maxima of f' mean inflection points. Can we transform the $Level(f)$ to $Level(f')$ trees with an appropriate operation? Can we identify concavity or inflection points from Level(f) trees alone? Although not Horton pruning, there may be some pruning operation that corresponds to taking the derivative. That is, Level(f') = PLevel(f) where P is some pruning operation. This is because, the number of local maxima decreases when taking the derivative along the same interval.

   Idea: Consider a degree n polynomial. Differentiating it yields a degree $n - 1$

polynomial and hence there are at max $\lceil \frac{n-1}{2} \rceil$ local maxima.Similarly, differentiating again will yield at max $\lceil \frac{n-2}{2} \rceil$ inflection points. So the Level(f') tree should have ~ 1 of the leaves pruned. Never will the Level(f') tree have more leaves than its Level(f) counterpart, so the appropriate tree-level differentiation operation should do some kind of pruning.

2. Assign Horton-Strahler orders to level-set trees. How are these orders preserved and what patterns may be observed?

Conjectures on Patterns of Length n

1. Only patterns of length 2 have representation as tree vertex. This much is clear from the construction of level set trees. What are other "identifiable features" of level set trees? We already know the correspondence between internal vertices, degree 1 vertices, and edges. Whats more is that these are still *level*-set trees, the level information can only be gotten by looking at maxima and minima. So patterns of greater than length 2 dont give us any information more. Even-length patterns just provide the same information over a longer interval, odd-length patterns dont provide enough information.

2. Instead of looking at patterns of length n and their occurrence translated directly into the level set tree, can we instead look at the statistics of their occurrences on both the time series and level set trees? For example, can we say something about heteroskedasticity of a time series from the number of local maxima observed? Can we say something about heteroskedasticity by tracking the degrees of vertices? What do measures of graph density/sparseness tell us about the time series?

Miscellaneous

1. What continuous functions correspond to general Critical Tokunaga processes?

2. What do HTGs and HVGs look like for Galton-Watson trees?

B Code

The code is hosted on github at https://github.com/pikhan/Research. To run simulations one can download the repository, unzip the folder, and run the `mySimulations.m` MatLab program (MatLab R2019B or above should work). It is important to keep the downloaded folder's structure and naming due to file dependencies. Should one wish to compute duals alone, `calc_circular_layout.m` and `calc_dual_layout.m` are the functions of concern. Helpful comments are maintained in the github repository but for brevity we only provide the code itself here.

B.1 `tree.m`

The code used in this thesis is primarily written in `Matlab` and utilizes a modified version of the `@tree` package by tinevez (Tinevez 29). The modification is in the `tree.m` file as below:

```
1     classdef tree
2     properties (SetAccess = private)
3         Node = { [] };
4         Parent = [ 0 ]; %#ok<NBRAK>
5     end
6     methods
7         function [obj, root_ID] = tree(content, val)
8             if nargin < 1
9                 root_ID = 1;
```

```
10                  return
11              end
12              if isa(content, 'tree')
13                  obj.Parent = content.Parent;
14                  if nargin > 1
15                      if strcmpi(val, 'clear')
16                          obj.Node = cell(numel(obj.Parent), 1)
                                ;
17                      else
18                          cellval = cell(numel(obj.Parent), 1);
19                          for i = 1 : numel(obj.Parent)
20                              cellval{i} = val;
21                          end
22                          obj.Node = cellval;
23                      end
24                  else
25                      obj.Node = content.Node;
26                  end
27              elseif isa(content, 'double')
28                  obj.Parent = transpose(content);
29                  obj.Node = transpose(cellstr(string(1:length(
                        content))));
30                  root_ID = find(content == 0);
31              else
32                  obj.Node = { content };
33                  root_ID = 1;
```

```
34              end
35          end
36          function [obj, ID] = addnode(obj, parent, data)
37              if parent < 0 || parent > numel(obj.Parent)
38                  error('MATLAB:tree:addnode', ...
39                      'Cannot add to unknown parent with index
                            %d.\n', parent)
40              end
41              if parent == 0
42                  obj.Node = { data };
43                  obj.Parent = 0;
44                  ID = 1;
45                  return
46              end
47              obj.Node{ end + 1, 1 } = data;
48              obj.Parent = [
49                  obj.Parent
50                  parent ];
51              ID = numel(obj.Node);
52          end
53          function flag = isleaf(obj, ID)
54             if ID < 1 || ID > numel(obj.Parent)
55                  error('MATLAB:tree:isleaf', ...
56                      'No node with ID %d.', ID)
57             end
58             parent = obj.Parent;
```

```
59              flag = ~any( parent == ID );
60          end
61          function IDs = findleaves(obj)
62              parents = obj.Parent;
63              IDs = (1 : numel(parents)); % All IDs
64              IDs = setdiff(IDs, parents); % Remove those which
                    are marked as parent
65          end
66          function content = get(obj, ID)
67              content = obj.Node{ID};
68          end
69          function obj = set(obj, ID, content)
70              obj.Node{ID} = content;
71          end
72          function IDs = getchildren(obj, ID)
73              parent = obj.Parent;
74              IDs = find( parent == ID );
75              IDs = IDs';
76          end
77          function ID = getparent(obj, ID)
78              if ID < 1 || ID > numel(obj.Parent)
79                  error('MATLAB:tree:getparent', ...
80                      'No node with ID %d.', ID)
81              end
82              ID = obj.Parent(ID);
83          end
```

```
84          function IDs = getsiblings(obj, ID)
85              if ID < 1 || ID > numel(obj.Parent)
86                  error('MATLAB:tree:getsiblings', ...
87                      'No node with ID %d.', ID)
88              end
89              if ID == 1 % Special case: the root
90                  IDs = 1;
91                  return
92              end
93              parent = obj.Parent(ID);
94              IDs = obj.getchildren(parent);
95          end
96          function n = nnodes(obj)
97              n = numel(obj.Parent);
98          end
99      end
100     methods (Static)
101         hl = decorateplots(ha)
102         function [lineage, duration] = example
103             lineage_AB = tree('AB');
104             [lineage_AB, id_ABa] = lineage_AB.addnode(1, 'AB.
                    a');
105             [lineage_AB, id_ABp] = lineage_AB.addnode(1, 'AB.
                    p');
106             [lineage_AB, id_ABal] = lineage_AB.addnode(id_ABa
                    , 'AB.al');
```

```
107             [lineage_AB, id_ABar] = lineage_AB.addnode(id_ABa
                    , 'AB.ar');
108             [lineage_AB, id_ABala] = lineage_AB.addnode(
                    id_ABal, 'AB.ala');
109             [lineage_AB, id_ABalp] = lineage_AB.addnode(
                    id_ABal, 'AB.alp');
110             [lineage_AB, id_ABara] = lineage_AB.addnode(
                    id_ABar, 'AB.ara');
111             [lineage_AB, id_ABarp] = lineage_AB.addnode(
                    id_ABar, 'AB.arp');
112             [lineage_AB, id_ABpl] = lineage_AB.addnode(id_ABp
                    , 'AB.pl');
113             [lineage_AB, id_ABpr] = lineage_AB.addnode(id_ABp
                    , 'AB.pr');
114             [lineage_AB, id_ABpla] = lineage_AB.addnode(
                    id_ABpl, 'AB.pla');
115             [lineage_AB, id_ABplp] = lineage_AB.addnode(
                    id_ABpl, 'AB.plp');
116             [lineage_AB, id_ABpra] = lineage_AB.addnode(
                    id_ABpr, 'AB.pra');
117             [lineage_AB, id_ABprp] = lineage_AB.addnode(
                    id_ABpr, 'AB.prp');
118             lineage_P1 = tree('P1');
119             [lineage_P1, id_P2] = lineage_P1.addnode(1, 'P2')
                    ;
```

```
120 [lineage_P1, id_EMS] = lineage_P1.addnode(1, 'EMS');
121 [lineage_P1, id_P3] = lineage_P1.addnode(id_P2, 'P3');
122 [lineage_P1, id_C] = lineage_P1.addnode(id_P2, 'C');
123 [lineage_P1, id_Ca] = lineage_P1.addnode(id_C, 'C.a');
124 [lineage_P1, id_Caa] = lineage_P1.addnode(id_Ca, 'C.aa');
125 [lineage_P1, id_Cap] = lineage_P1.addnode(id_Ca, 'C.ap');
126 [lineage_P1, id_Cp] = lineage_P1.addnode(id_C, 'C.p');
127 [lineage_P1, id_Cpa] = lineage_P1.addnode(id_Cp, 'C.pa');
128 [lineage_P1, id_Cpp] = lineage_P1.addnode(id_Cp, 'C.pp');
129 [lineage_P1, id_MS] = lineage_P1.addnode(id_EMS, 'MS');
130 [lineage_P1, id_MSa] = lineage_P1.addnode(id_MS, 'MS.a');
131 [lineage_P1, id_MSp] = lineage_P1.addnode(id_MS, 'MS.p');
132 [lineage_P1, id_E] = lineage_P1.addnode(id_EMS, 'E');
```

```
133             [lineage_P1, id_Ea] = lineage_P1.addnode(id_E, 'E
                    .a');
134             [lineage_P1, id_Eal] = lineage_P1.addnode(id_Ea,
                    'E.al'); %#ok<*NASGU>
135             [lineage_P1, id_Ear] = lineage_P1.addnode(id_Ea,
                    'E.ar');
136             [lineage_P1, id_Ep] = lineage_P1.addnode(id_E, 'E
                    .p');
137             [lineage_P1, id_Epl] = lineage_P1.addnode(id_Ep,
                    'E.pl');
138             [lineage_P1, id_Epr] = lineage_P1.addnode(id_Ep,
                    'E.pr');
139             [lineage_P1, id_P4] = lineage_P1.addnode(id_P3, '
                    P4');
140             [lineage_P1, id_Z2] = lineage_P1.addnode(id_P4, '
                    Z2');
141             [lineage_P1, id_Z3] = lineage_P1.addnode(id_P4, '
                    Z3');
142             [lineage_P1, id_D] = lineage_P1.addnode(id_P3, 'D
                    ');
143             lineage = tree('Zygote');
144             lineage = lineage.graft(1, lineage_AB);
145             lineage = lineage.graft(1, lineage_P1);
146             duration = tree(lineage, 'clear');
147             iterator = duration.depthfirstiterator;
148             for i = iterator
```

```
149                 duration = duration.set(i, round(20*rand));
150             end
151         end
152     end
153 end
```

From there, we define a number of custom functions which are given as follows.

B.2 calc_circular_layout.m

```
1  function layout = calc_circular_layout(tree,root_id,
       start_angle,total_angle,scale,plotParam)
2      angles = containers.Map('KeyType','double','ValueType','
           any');
3      angleSelect = get_angle(tree,root_id,start_angle,
           total_angle);
4      distances = containers.Map('KeyType','double','ValueType'
           ,'any');
5      queue = {root_id};
6      leaf_iterator = 0;
7      angles(root_id)=90;
8      distances(root_id)=1;
9      while ~isempty(queue)
10         node = queue{1};
11         queue(1) = [];
12         children = tree.getchildren(node);
13         for i = 1:length(children)
14             child_id = children(i);
```

```
15              angle=0;
16              distance=0;
17              if tree.isleaf(child_id)
18                  distance = 1;
19                  angle=angleSelect(child_id);
20              elseif child_id==1
21                  distance = get_distance_from_root(tree,
                        child_id,root_id,scale);
22                  angle = angleSelect(child_id);
23              elseif isempty(child_id)
24                  disp('leaf');
25              else
26                  distance = get_distance_from_root(tree,
                        child_id,root_id,scale);
27                  angle = angleSelect(child_id);
28              end
29              angles(child_id) = angle;
30              distances(child_id)=distance;
31              queue{end+1} = child_id;
32          end
33      end
34  anglevals=angles.values;
35  anglevalues=vertcat(anglevals{:});
36  distancevals=distances.values;
37  distancevalues=vertcat(distancevals{:});
38  layout={angles,distances};
```

```
39 if plotParam==1
40     figure();
41     polarplot(deg2rad(anglevalues), distancevalues, 'o', '
          Color','blue');
42     hold on;
43     for i=1:tree.nnodes
44         for j=1:tree.nnodes
45             if i==tree.getparent(j) || j==tree.getparent(i)
46                 line([deg2rad(angles(i)), deg2rad(angles(j))
                       ], [distances(i), distances(j)],'Color','
                       blue');
47             end
48         end
49     end
50     k = 1;
51     theta = linspace(0,2*pi);
52     rho = linspace(k,k);
53     polarplot(theta,rho,'green');
54     hold off;
55 end
```

B.3 calc_dual_layout.m

```
1 function layout = calc_dual_layout(tree,treelayout,
     graph_param)
2 angles=treelayout{1};
3 distances=treelayout{2};
```

```
4  anglevals=angles.values;
5  anglevalues=vertcat(anglevals{:});
6  distancevals=distances.values;
7  distancevalues=vertcat(distancevals{:});
8  figure();
9  polarplot(deg2rad(anglevalues), distancevalues, 'o', 'Color',
       'blue');
10 selectedKeys = keys(distances);
11 selectedKeys = selectedKeys(cell2mat(cellfun(@(x) x == 1,
       values(distances), 'UniformOutput', false)));
12 subsetMap = containers.Map(selectedKeys, values(angles,
       selectedKeys));
13 leafAngles = cell2mat(subsetMap.values);
14 leafAngles_copy=leafAngles;
15 leafAngles(leafAngles>360)=leafAngles(leafAngles>360) - 360;
16 centered_angles = leafAngles - 90;
17 wrapped_angles = centered_angles;
18 wrapped_angles(wrapped_angles < 0) = wrapped_angles(
       wrapped_angles < 0) + 360;
19 sorted_wrapped_angles = sort(wrapped_angles);
20 sorted_angles = mod(sorted_wrapped_angles + 90, 360);
21 sorted_angles_og=sorted_angles;
22 sorted_angles_og(sorted_angles_og<90)=sorted_angles_og(
       sorted_angles_og<90)+360;
23 sorted_keys = zeros(size(sorted_angles_og));
24 for i = 1:length(sorted_angles_og)
```

```
25      for j = 1:length(angles)
26          if sorted_angles_og(i) == angles(j)
27              sorted_keys(i) = j;
28              break;
29          end
30      end
31  end
32  faceEdgeList = populateFaces(tree,sorted_keys); %populate our
        face edge list
33  connections = {};
34  for i = 1:length(faceEdgeList)-1
35      currentElement = faceEdgeList{i};
36      temp={}; %temp array to store the j for which there are
            connnections
37      for j = i+1:length(faceEdgeList)
38          nextElement = faceEdgeList{j};
39          sharedElements = intersect(currentElement,
                nextElement);
40          if isempty(sharedElements)==0
41              temp=[temp j];
42          end
43      end
44      connections{i} = temp;
45  end
46  layout=connections;
47  for i = 1:length(leafAngles)
```

```
48          disp(i);
49          myIndex= find(sorted_angles==leafAngles(i));
50          iterator_select=myIndex;
51      text(deg2rad(leafAngles(i)), 1, "Leaf"+string(
            iterator_select),'HorizontalAlignment', 'right', '
            VerticalAlignment', 'top', 'Margin', 25);
52  end
53  hold on;
54  for i=1:tree.nnodes
55      for j=1:tree.nnodes
56          if i==tree.getparent(j) || j==tree.getparent(i)
57              line([deg2rad(angles(i)), deg2rad(angles(j))], [
                    distances(i), distances(j)],'Color','blue');
58          end
59      end
60  end
61  midangles={};
62  for i=1:length(sorted_angles_og)
63      mid_angle=0;
64      if i~=length(sorted_angles_og)
65          mid_angle=(sorted_angles_og(i)+sorted_angles_og(i+1))
                /2;
66      else
67          mid_angle=(sorted_angles_og(i)+450)/2;
68      end
69      midangles{i}=mid_angle;
```

```
70 end
71 for i=1:length(midangles)-1
72     text(deg2rad(midangles{i}), 1, "Face"+string(i+1),'
          HorizontalAlignment', 'right', 'VerticalAlignment', '
          top', 'Margin', 25,'Color','red');
73 end
74 text(deg2rad(midangles{length(midangles)}), 1, "Face"+string
    (1),'HorizontalAlignment', 'right', 'VerticalAlignment', '
    top', 'Margin', 25,'Color','red');
75 midangles_reorder=midangles(1:end-1);
76 last_element=midangles{end};
77 midangles_reorder=[last_element midangles_reorder];
78 for i=1:length(connections)
79     for j=1:length(connections{i})
80         line([deg2rad(midangles_reorder{connections{i}{j}}),
              deg2rad(midangles_reorder{i})],[1,1],'Color','red'
              );
81     end
82 end
83 k = 1;
84 theta = linspace(0,2*pi);
85 rho = linspace(k,k);
86 polarplot(theta,rho,'blue');
87 hold off;
88 if graph_param==1
89     figure();
```

```
90      k = 1;
91      theta = linspace(0,2*pi);
92      rho = linspace(k,k);
93      polarplot(theta,rho,'blue');
94      for i=1:length(connections)
95          for j=1:length(connections{i})
96              line([deg2rad(midangles_reorder{connections{i}{j
                    }}),deg2rad(midangles_reorder{i})],[1,1],'
                    Color','red');
97          end
98      end
99  end
```

B.4 draw_tree.m

```
1   function draw_tree(root, layout)
2       figure;
3       hold on;
4       for i = 1:length(layout)
5           node = layout.keys{i};
6           pos = layout(node);
7           x = pos(2)*cosd(pos(1));
8           y = pos(2)*sind(pos(1));
9           parent = get_parent(node);
10          if ~isempty(parent)
11              parent_pos = layout(parent);
12              parent_x = parent_pos(2)*cosd(parent_pos(1));
```

```
13              parent_y = parent_pos(2)*sind(parent_pos(1));
14              line([parent_x, x], [parent_y, y], 'LineWidth',
                    2);
15          end
16          scatter(x, y, 'filled');
17          text(x, y, node);
18      end
19      axis equal;
20      hold off;
21  end
```

B.5 get_angle.m

```
1       function angles = get_angle(tree,root_id,start_angle,
            total_angle)
2   angles = containers.Map('KeyType','double','ValueType','any')
        ;
3   for i = 1:nnodes(tree)
4       angles(i) = NaN;
5   end
6   angle_increment = total_angle/(numel(tree.findleaves)+1);
7   leaf_iterator = 0;
8   it = traverse_left_to_right(tree,root_id);
9   for i=it
10      if tree.isleaf(i)
11          leaf_iterator=leaf_iterator+1;
12          angles(i) = start_angle+leaf_iterator*angle_increment
```

```
            ;
    elseif i==root_id | i==tree.getchildren(root_id)
        angles(i) = 90;
    end
end
while sum(~isnan(cell2mat(values(angles))))~=nnodes(tree)
    queue = [root_id];
    while ~isempty(queue)
        node = queue(1);
        queue(1) = [];
        children = tree.getchildren(node);
        truth=0;
        for i=1:length(children)
            truth=truth+~isnan(angles(children(i)));
        end
        if truth==length(children) && length(children)~=0
            average=0;
            for i=1:length(children)
                average=average+angles(children(i));
            end
            average=average/length(children);
            angles(node)=average;
        end
        queue = [children, queue];
    end
end
```

```
38 end
```

B.6 get_distance_from_root.m

```
1      function distance = get_distance_from_root(tree,child_id,
          root_id,scale)
2      depth=0;
3      node_id=child_id;
4      while node_id ~= root_id
5          node_id=tree.getparent(node_id);
6          depth=depth+1;
7          disp(node_id);
8      end
9      distance=scale*((depth)/tree.depth);
10 end
```

B.7 intersections.m

This is a modified version of the typical intersections function gotten from Schwarz's Matlab package (Schwarz 25).

```
1      function [x0,y0,iout,jout] = intersections(x1,y1,x2,y2,
          robust)
2  if verLessThan('matlab','7.13')
3          error(nargchk(2,5,nargin)) %#ok<NCHKN>
4  else
5          narginchk(2,5)
6  end
```

```
7  switch nargin
8          case 2
9                  robust = true;
10                 x2 = x1;
11                 y2 = y1;
12                 self_intersect = true;
13         case 3
14                 robust = x2;
15                 x2 = x1;
16                 y2 = y1;
17                 self_intersect = true;
18         case 4
19                 robust = true;
20                 self_intersect = false;
21         case 5
22                 self_intersect = false;
23 end
24 if sum(size(x1) > 1) ~= 1 || sum(size(y1) > 1) ~= 1 || ...
25                 length(x1) ~= length(y1)
26         error('X1 and Y1 must be equal-length vectors of at
              least 2 points.')
27 end
28 if sum(size(x2) > 1) ~= 1 || sum(size(y2) > 1) ~= 1 || ...
29                 length(x2) ~= length(y2)
30         error('X2 and Y2 must be equal-length vectors of at
              least 2 points.')
```

```
31 end
32 x1 = x1(:);
33 y1 = y1(:);
34 x2 = x2(:);
35 y2 = y2(:);
36 n1 = length(x1) - 1;
37 n2 = length(x2) - 1;
38 xy1 = [x1 y1];
39 xy2 = [x2 y2];
40 dxy1 = diff(xy1);
41 dxy2 = diff(xy2);
42 if n1 > 1000 || n2 > 1000 || verLessThan('matlab','7.4')
43         if n1 >= n2
44                 ijc = cell(1,n2);
45                 min_x1 = mvmin(x1);
46                 max_x1 = mvmax(x1);
47                 min_y1 = mvmin(y1);
48                 max_y1 = mvmax(y1);
49                 for k = 1:n2
50                         k1 = k + 1;
51                         ijc{k} = find( ...
52                                 min_x1 <= max(x2(k),x2(k1)) &
                                        max_x1 >= min(x2(k),x2(k1
                                    )) & ...
53                                 min_y1 <= max(y2(k),y2(k1)) &
                                        max_y1 >= min(y2(k),y2(k1
```

```
                                        ))));
54                          ijc{k}(:,2) = k;
55                  end
56                  ij = vertcat(ijc{:});
57                  i = ij(:,1);
58                  j = ij(:,2);
59          else
60                  ijc = cell(1,n1);
61                  min_x2 = mvmin(x2);
62                  max_x2 = mvmax(x2);
63                  min_y2 = mvmin(y2);
64                  max_y2 = mvmax(y2);
65                  for k = 1:n1
66                          k1 = k + 1;
67                          ijc{k}(:,2) = find( ...
68                                  min_x2 <= max(x1(k),x1(k1)) &
                                        max_x2 >= min(x1(k),x1(k1
                                    )) & ...
69                                  min_y2 <= max(y1(k),y1(k1)) &
                                        max_y2 >= min(y1(k),y1(k1
                                    ))));
70                          ijc{k}(:,1) = k;
71                  end
72                  ij = vertcat(ijc{:});
73                  i = ij(:,1);
74                  j = ij(:,2);
```

```
75          end
76  elseif verLessThan('matlab','9.1')
77          [i,j] = find( ...
78                  bsxfun(@le,mvmin(x1),mvmax(x2).') & ...
79                  bsxfun(@ge,mvmax(x1),mvmin(x2).') & ...
80                  bsxfun(@le,mvmin(y1),mvmax(y2).') & ...
81                  bsxfun(@ge,mvmax(y1),mvmin(y2).'));
82  else
83          [i,j] = find( ...
84                  mvmin(x1) <= mvmax(x2).' & mvmax(x1) >= mvmin
                        (x2).' & ...
85                  mvmin(y1) <= mvmax(y2).' & mvmax(y1) >= mvmin
                        (y2).');
86  end
87  if self_intersect
88          remove = isnan(sum(dxy1(i,:) + dxy2(j,:),2)) | j <= i
                + 1;
89  else
90          remove = isnan(sum(dxy1(i,:) + dxy2(j,:),2));
91  end
92  i(remove) = [];
93  j(remove) = [];
94  n = length(i);
95  T = zeros(4,n);
96  AA = zeros(4,4,n);
97  AA([1 2],3,:) = -1;
```

```
98  AA([3  4],4,:) = -1;
99  AA([1  3],1,:) = dxy1(i,:).';
100 AA([2  4],2,:) = dxy2(j,:).';
101 B = -[x1(i) x2(j) y1(i) y2(j)].';
102 if robust
103         overlap = false(n,1);
104         warning_state = warning('off','MATLAB:singularMatrix'
                );
105         try
106                 lastwarn('')
107                 for k = 1:n
108                         T(:,k) = AA(:,:,k)\B(:,k);
109                         [unused,last_warn] = lastwarn; %#ok<
                            ASGLU>
110                         lastwarn('')
111                         if strcmp(last_warn,'MATLAB:
                            singularMatrix')
112                                 T(1,k) = NaN;
113                                 overlap(k) = rcond([dxy1(i(k)
                                    ,:);xy2(j(k),:) - xy1(i(k)
                                    ,:)]) < eps;
114                         end
115                 end
116                 warning(warning_state)
117         catch err
118                 warning(warning_state)
```

```
119             rethrow(err)
120         end
121         in_range = (T(1,:) >= 0 & T(2,:) >= 0 & T(1,:) <= 1 &
                T(2,:) <= 1).';
122         if any(overlap)
123             ia = i(overlap);
124             ja = j(overlap);
125             T(3,overlap) = (max(min(x1(ia),x1(ia+1)),min(
                    x2(ja),x2(ja+1))) + ...
126                     min(max(x1(ia),x1(ia+1)),max(x2(ja),
                            x2(ja+1)))).'/2;
127             T(4,overlap) = (max(min(y1(ia),y1(ia+1)),min(
                    y2(ja),y2(ja+1))) + ...
128                     min(max(y1(ia),y1(ia+1)),max(y2(ja),
                            y2(ja+1)))).'/2;
129             selected = in_range | overlap;
130         else
131             selected = in_range;
132         end
133         xy0 = T(3:4,selected).';
134         [xy0,index] = unique(xy0,'rows');
135         x0 = xy0(:,1);
136         y0 = xy0(:,2);
137         if nargout > 2
138             sel_index = find(selected);
139             sel = sel_index(index);
```

```
140                 iout = i(sel) + T(1,sel).';
141                 jout = j(sel) + T(2,sel).';
142         end
143 else % non-robust option
144         for k = 1:n
145                 [L,U] = lu(AA(:,:,k));
146                 T(:,k) = U\(L\B(:,k));
147         end
148         in_range = (T(1,:) >= 0 & T(2,:) >= 0 & T(1,:) < 1 &
              T(2,:) < 1).';
149         x0 = T(3,in_range).';
150         y0 = T(4,in_range).';
151         if nargout > 2
152                 iout = i(in_range) + T(1,in_range).';
153                 jout = j(in_range) + T(2,in_range).';
154         end
155 end
156 function y = mvmin(x)
157 y = min(x(1:end-1),x(2:end));
158 function y = mvmax(x)
159 y = max(x(1:end-1),x(2:end));
```

B.8 populateFaces.m

```
1     function faceEdgeList = populateFaces(tree,leafkeys)
2 faceEdgeList = cell(length(leafkeys),1);
3 j=1;
```

```
while j~=length(leafkeys)+1
    faceNum=0;
    leaf1=0;
    leaf2=0;
    if j~=length(leafkeys)
        faceNum=j+1;
        leaf1 = leafkeys(j+1);
        leaf2 = leafkeys(j);
    end
    if j==length(leafkeys) %condition for the last leaf
       connecting to the root
        leaf1=leafkeys(j);
        leaf2=leafkeys(1);
        faceNum=1;
    end
    visited1 = {leaf1};
    if leaf1~=1
        parent1 = tree.getparent(leaf1);
        while ~isequal(parent1, 1)
            visited1 = [visited1, parent1];
            parent1 = tree.getparent(parent1);
        end
        visited1 = [visited1, 1]; % add the root node to the
            list
    end
    visited2 = {leaf2};
```

```
28      if leaf2~=1
29          parent2 = tree.getparent(leaf2);
30          while ~isequal(parent2, 1)
31              visited2 = [visited2, parent2];
32              parent2 = tree.getparent(parent2);
33          end
34          visited2 = [visited2, 1];  % add the root node to the
                list
35      end
36      visited1=cell2mat(visited1);
37      visited2=cell2mat(visited2);
38      disp(visited1);
39      disp(visited2);
40      common_node = max(intersect(visited1, visited2));
41      disp(common_node);
42      index1 = find(visited1 == common_node);
43      disp(index1);
44      visited1 = visited1(1:index1-1); %get rid of common node
           as we dont actually traverse that node's correspondent
            edge.
45      disp(visited1);
46      index2 = find(visited2 == common_node);
47      visited1 = [visited1, visited2(index2-1:-1:1)];
48      path = visited1;
49      faceEdgeList{faceNum,1}=path;
50      j=j+1;
```

```
51 end
```

B.9 traverse_left_to_right.m

```
1      function output = traverse_left_to_right(tree, id)
2      output = [];
3      if ~(tree.isleaf(id))
4          children = tree.getchildren(id);
5          for i = 1:length(children)
6              output = [output, traverse_left_to_right(tree,
                   children(i))];
7          end
8      end
9      output = [output, id];
10     fprintf('%d ', id);
11 end
```

B.10 layout_to_adj.m

```
1  function adj = layout_to_adj(layout)
2      n = length(layout) + 1;
3      adj = inf(n); % Initialize adjacency matrix with inf
4      for i = 1:n-1
5          for j = 1:length(layout{i})
6              adj(i,layout{i}{j}) = 1; % Set edge between i
                   and layout{i}{j}
```

```
7               adj(layout{i}{j},i) = 1;  % Set edge between
                    layout{i}{j} and i
8           end
9       end
10      adj(1,n) = 1;
11      adj(n,1) = 1;
12      for i = 1:n
13          adj(i,i) = 0;
14      end
15  end
```

B.11 floyd_warshall.m

```
1   function [dist, apl] = floyd_warshall(adj)
2       n = size(adj,1);
3       dist = adj;
4       for k = 1:n
5           for i = 1:n
6               for j = 1:n
7                   if dist(i,k) + dist(k,j) < dist(i,j)
8                       dist(i,j) = dist(i,k) + dist(k,j);
9                   end
10              end
11          end
12      end
13  apl = sum(sum(dist))/(n*(n-1));
14  end
```

B.12 compute_mean_path_length.m

```
function mean_path_length_n = compute_mean_path_length_n(dist)
    N = length(dist);
    mean_path_length_n = zeros(1, N);
    for n = 1:N
        subgraph_dist = dist(1:n, 1:n);
        valid_paths = subgraph_dist(subgraph_dist ~= inf & subgraph_dist ~= 0);
        mean_path_length_n(n) = sum(valid_paths) / length(valid_paths);
    end
end
```

B.13 compute_clustering_coefficients.m

```
function [clustering_coefficients, mean_clustering_coefficient] = compute_clustering_coefficient(adj_matrix)
N = length(adj_matrix);
clustering_coefficients = zeros(1, N);
for i = 1:N
    adj_matrix(isinf(adj_matrix)|isnan(adj_matrix)) = 0;
    neighbors = find(adj_matrix(i, :)); % get the indices of the neighbors
    N_i = length(neighbors); % number of neighbors
```

```
8       if N_i > 1 % only consider nodes with at least two
            neighbors
9           E_i = sum(sum(adj_matrix(neighbors, neighbors))) / 2;
                % number of edges between neighbors
10          clustering_coefficients(i) = 2 * E_i / (N_i * (N_i -
                1));
11      end
12  end
13  mean_clustering_coefficient = mean(clustering_coefficients);
```

B.14 reorder_parent_pointer.m

```
1       function pp_new = reorder_parent_pointer(pp_old,
            root_index)
2       pp_new = zeros(size(pp_old));
3       shift_amount = root_index - 1;
4       for i = 1:length(pp_old)
5           new_index = mod(i - shift_amount - 1, length(pp_old))
                + 1;
6           if pp_old(i) == 0  % If this is the root
7               pp_new(new_index) = 0;
8           else
9               new_parent_index = mod(pp_old(i) - shift_amount -
                    1, length(pp_old)) + 1;
10              pp_new(new_index) = new_parent_index;
11          end
12      end
```

```
13 end
```

B.15 compute_degrees.m

```
1  function [degree_sequence, unique_degrees,
       degree_distribution, mean_deg, mean_deg_n] =
       compute_degrees(layout)
2      degrees = zeros(1, length(layout) + 1);
3      for i = 1:length(layout)
4          degrees(i) = length(layout{i});
5      end
6      for i = 1:length(layout)
7          for j = 1:length(layout{i})
8              degrees(layout{i}{j}) = degrees(layout{i}{j}) +
                   1;
9          end
10     end
11 degree_sequence = sort(degrees, 'descend');
12 [unique_degrees, ~, degree_indices] = unique(degree_sequence)
       ;
13 degree_counts = accumarray(degree_indices, 1);
14 degree_distribution = degree_counts / sum(degree_counts);
15 figure();
16 semilogy(unique_degrees, degree_distribution);
17 xlabel('Degree');
18 ylabel('Frequency');
19 title('Degree Distribution');
```

```
20 mean_deg = mean(degrees);
21 mean_deg_n = zeros(1, length(layout) + 1);
22 cum_sum_degrees = cumsum(degrees);
23     for i = 1:length(layout)
24         mean_deg_n(i) = cum_sum_degrees(i) / i;
25     end
26     mean_deg_n(end) = mean_deg;  % The mean degree of all
       vertices is the mean degree we computed earlier
27 end
```

B.16 mySimulations.m

```
1 function sim = mySimulations()
2 levelSetTree = level_set_tree(time_series,[1:1000],0); %our
    level set tree, code from Zoe Haskell's Ph.D thesis
3 root_index = find(levelSetTree == 0);
4 lvlTreeParentPointer = reorder_parent_pointer(levelSetTree,
    root_index);
5 lvlTree = tree(lvlTreeParentPointer);
6 treeplot(levelSetTree);
7 figure();
8 treelayout = calc_circular_layout(lvlTree,1,120,300,0.75,1);
9 layout = calc_dual_layout(lvlTree,treelayout,1);
10 [degree_sequence, unique_degrees, degree_distribution,
    mean_deg, mean_deg_n] = compute_degrees(layout);
11 disp("The degree sequence: ");
12 disp("The unique degrees: ");
```

```
13 disp("The degree dist");
14 disp(degree_distribution);
15 disp("the mean degree");
16 disp(mean_deg);
17 disp("the mean deg n");
18 figure();
19 plot(mean_deg_n);
20 title('Mean Degree as a Function of N')
21 adj = layout_to_adj(layout);
22 [dist, apl] = floyd_warshall(adj);
23 disp("the apl:");
24 disp(apl);
25 mean_path_length_n = compute_mean_path_length_n(dist);
26 disp("the mean path length n");
27 mean_path_length_n(isnan(mean_path_length_n))=0;
28 figure();
29 plot(mean_path_length_n);
30 title('The Mean Path Length as a Function of N');
31 [clustering_coefficients, mean_clustering_coefficient] =
      compute_clustering_coefficient(adj);
32 disp("the clustering coefficients:");
33 figure();
34 plot(clustering_coefficients);
35 title('Clustering Coefficients in Order of Vertices');
36 disp("the mean clustering coefficient");
37 disp(mean_clustering_coefficient);
```

```
38 sim = 1;
```

C Circle Packing Code

This code was unused in the final version of the thesis. It can be used to get approximate, but not exact, duals.

```
1      function [C, E] = circle_packing(R, C)
2 n = size(C, 1); % Number of circles
3 E = []; % List of edges
4 D = sqrt(sum((reshape(C, [n,1,2]) - reshape(C, [1,n,2])).^2,
      3));% Compute minimum and maximum radii
5 rmin = min(R);
6 rmax = max(R);
7 Rsum = bsxfun(@plus, R, R.');
8 Ravg = Rsum / 2;
9 disp("tester");
10 disp(size(D));
11 delta = D-(R+R')/2;
12 theta = zeros(n);
13 for i = 1:n
14     for j = i+1:n
15         if delta(i,j) < 0
16             if D(i,j) < (R(i) + R(j))
17                 theta(i,j) = pi;
18             else
```

```
19                    a = acos((R(i)^2 + D(i,j)^2 - R(j)^2) / (2 *
                          R(i) * D(i,j)));
20                    b = acos((R(j)^2 + D(i,j)^2 - R(i)^2) / (2 *
                          R(j) * D(i,j)));
21                    theta(i,j) = a + b;
22                end
23                theta(j,i) = theta(i,j);  % Symmetric
24                E = [E; i, j];  % Add edge to list
25            end
26        end
27    end
28    for i = 1:n
29        t = sum(theta(i,:));
30        if t == 0
31            C(i,:) = [0, 0];
32        else
33            C(i,:) = sum(C .* (theta(i,:)' * ones(1,2)), 1) / t;
34        end
35    end
36    end
```

D Potential Alternative Procedure For Producing Duals

We briefly give here a potential alternative procedure for producing duals inspired by a 1988 paper by Thurston and others on triangulations and trees (Sleator, Tarjan,

and Thurston 26).

1. Start with a binary, rooted tree with m nodes
2. Order nodes with inorder traversal for binary trees
3. Identify the index of the root node
4. Draw a convex $(m+2) - gon$
5. Label the non-root vertices of the polygon counterclockwise with the root occupying the top left and top right vertices.
6. Draw the *root triangle* by connecting the root vertices of the polygon with the vertex with the same index as the tree's root
7. Recursively triangulate the *diagonals* or the root triangle partitions of the polygon. Triangulate the left and right sides. The diagonal's two vertices are now the *new* polygon roots corresponding to a subtree with its own root.
8. Obtain the structure of the constructed diagonals
9. Obtain the dual by removing the 2 edges between the polygons original two roots and the constructed diagonal of the rightmost original root. Leave all sides of the formative dual (the sides will form a smaller polygon) in tact. Connect any loose points to the leftmost original root of the polygon.

References

[1] Mehmet E. Aktas, Esra Akbas, and Ahmed El Fatmaoui. "Persistence homology of networks: methods and applications". In: *Applied Network Science* 4(61) (2019). URL: https://doi.org/10.1007/s41109-019-0179-3.

[2] W. Asif et al. "On the Complexity of Average Path length for Biological Networks and Patterns". In: *International Journal of Biomathematics* 7(4) (2014). URL: https://doi.org/10.1142/S1793524514500387.

[3] Yuliy Baryshnikov. "Time Series, Persistent Homology and Chirality". In: *Preprint* (2019). URL: https://arxiv.org/pdf/1909.09846.pdf.

[4] G.E. Box et al. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[5] Julian Brüggeman. "On merge trees and discrete Morse functions on paths and trees". In: *Journal of Applied and Computational Topology* 7 (2023), pp. 103–138. URL: https://doi.org/10.1007/s41468-022-00101-w.

[6] B. Cazalles et al. "Wavelet analysis of ecological time series". In: *Oecologia* 156(2) (2008), pp. 287–304.

[7] Justin Curry. "The Fiber of the Persistence Map for Functions on the Interval". In: *Journal of Applied and Computational Topology* 2 (2018), pp. 301–321. URL: https://doi.org/10.1007/s41468-019-00024-z.

[8] E.W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[9] Boris Goldfarb. "Singular Persistent Homology with Geometrically Parallelizable Computation". In: *Topology Proceedings* 55 (Feb. 2020), pp. 273–294. URL: https://doi.org/10.48550/arXiv.1607.01257.

[10] Ted Harris. "The theory of branching processes". In: *RAND Corp.* MR1991122 (1964). URL: https://www.rand.org/pubs/reports/R381.html.

[11] Zoe Haskell. "Mapping between tree metric spaces and time series: New constructions with applications to limit theorems for branch counts in partial

Galton-Watson trees and signal processing". PhD thesis. University of Nevada Reno, 2020. URL: http://hdl.handle.net/11714/7701.

[12] Peter Z. Ingerman. "Algorithm 141: Path Matrix". In: *Communications of the ACM* 5(11) (1962), p. 556.

[13] Jussi Klemelä. "Level Set Tree Methods". In: *WIREs Computational Statistics*. Advanced Review 10(5) (2018).

[14] Yevgeniy Kovchegov and Ilya Zaliapin. "Random self-similar trees: A mathematical theory of Horton laws". In: *Probability Surveys* 17 (2020), pp. 1–213. URL: https://doi.org/10.1214/19-PS331.

[15] Yevgeniy Kovchegov, Ilya Zaliapin, and Efi Foufoula-Georgiou. "Random Self-similar Trees: Emergence of Scaling Laws". In: *Survey in Geophysics* (2022). URL: https://doi.org/10.1007/s10712-021-09682-0.

[16] L. Lacasa et al. "From time series to complex networks: The visibility graph". In: *PNAS* 105(13) (2008), pp. 4972–4975. URL: https://doi.org/10.1073/pnas.0709247105.

[17] Steven Lalley. "BRANCHING PROCESSES". In: *Stochastic Processes Notes* (). URL: https://galton.uchicago.edu/~lalley/Courses/312/Branching.pdf.

[18] L. Ljung. *Signal analysis and prediction*. Vol. System identification. Birkhauser, 1999.

[19] B. Luque et al. "Horizontal visibility graphs: exact results for random time series". In: *Phys. Rev. E* 80:046103 (2009). URL: https://doi.org/10.48550/arXiv.1002.4526.

[20] Ian McKenna. "The Galton Watson Process – Part I". In: *Core Computations* (2011). URL: https://corecomputations.wordpress.com/2011/07/26/the-galton-watson-process-part-i/.

[21] H. Mehlhorn and F. Schreiber. "Small-World Property". In: *Encyclopedia of Systems Biology* (2013). URL: https://doi.org/10.1007/978-1-4419-9863-7_2.

[22] J. Pitman. *Combinatorial stochastic processes*. Vol. 1875. Lecture Notes in Mathematics. Lectures from the 32nd Summer School on Probability Theory held in Saint-Flour, July 7–24, 2002, With a foreword by Jean Picard. Berlin: Springer-Verlag, 2006, pp. x+256. ISBN: 978-3-540-30990-1. DOI: 10.1007/b11601500. URL: http://bibserver.berkeley.edu/csp/april05/bookcsp.pdf.

[23] D. Revuz and M. Yor. *Continuous martingales and Brownian motion*. Springer Science & Business Media, 2005.

[24] Kenneth H. Rosen. *Discrete Mathematics and Its Applications 5 ed.* Addison Wesley, 2003. ISBN: 978-0-07-119881-3.

[25] Douglas M. Schwarz. *intersections*. https://www.mathworks.com/matlabcentral/fileexchange/11837-fast-and-robust-curve-intersections. 2017.

[26] D. Sleator, R. Tarjan, and W. Thurston. "Rotation Distance, Triangulations, and Hyperbolic Geometry". In: *Journal of the American Mathematical Society* (1988).

[27] C. Song, S. Havlin, and H. Makse. "Self-similarity of complex networks". In: *Nature* 433(7024) (2005). URL: https://doi.org/10.1038/nature03248.

[28] Colin Stephen. "Horizon Visibility Graphs and Time Series Merge Trees are Dual". In: *Preprint* (June 2019). URL: http://arxiv.org/abs/1906.08825.

[29] Jean-Yves Tinevez. *@tree: A MATLAB class to represent the tree data structure.* https://github.com/tinevez/matlab-tree. 2015.

[30] S. Unkel et al. "Statistical methods for the prospective detection of infectious disease outbreaks: a review". In: *Journal of the Royal Statistical Society.* A: Statistics in Society 171(1) (2012), pp. 49–82.

[31] H. W. Watson and Francis Galton. "On the Probability of the Extinction of Families". In: *The Journal of the Anthropological Institute of Great Britain and Ireland* 4 (1875), pp. 138–144. URL: https://www.jstor.org/stable/pdf/2841222.pdf.

[32] D.J. Watts and Steven Strogatz. "Collective dynamics of 'small-world' networks". In: *Nature* 393(6684) (1998). URL: https://doi.org/10.1038/30918.