

topologyR: User Manual

Topological Connectivity Analysis via Graph-Induced Topologies

José Mauricio Gómez Julián

2026-04-04

Contents

1	THEORETICAL FOUNDATIONS	4
1.1	Introduction to Topological Spaces	4
1.1.1	Formal Definition	4
1.1.2	Fundamental Properties	4
1.2	Bases and Subbases	5
1.2.1	Definition and Properties of Bases	5
1.2.2	Subbases and their Relationship with Bases	5
1.3	Connectivity in Topological Spaces	6
1.3.1	Topological Definition of Connectivity	6
1.3.2	Connected Components	7
1.3.3	Connectivity in Graphs and its Relation to Topological Connectivity	7
1.4	Intuitive Explanation of Theoretical Foundations	7
2	GRAPH-THEORETIC TOPOLOGY CONSTRUCTION	8
2.1	The Nada et al. (2018) Framework	8
2.1.1	The Nada et al. (2018) Algorithm	9
2.1.2	The Role of Closed Neighborhoods	9
2.2	Visibility Graphs as the Graph Source	10
2.2.1	Horizontal Visibility Graph (HVG)	10
2.2.2	Natural Visibility Graph (NVG)	10
2.2.3	HVG vs. NVG: Comparison	11
2.2.4	Directed Visibility Graphs (v0.2.0)	11
2.2.5	Why Not Threshold-Based Graphs?	12
2.3	From Subbase to Base: Fixed-Point Closure Under Finite Intersections	12
2.4	From Base to Topology: Fixed-Point Closure Under Arbitrary Unions	13
2.5	Safety Limits and Informative Termination	13

3	CONNECTIVITY DETERMINATION	14
3.1	The Exact Topological Definition	14
3.2	Why the Three Legacy Heuristics Fail	14
3.2.1	<code>is_topology_connected()</code> — Undirected Graph DFS	14
3.2.2	<code>is_topology_connected2()</code> — Directed Graph DFS	14
3.2.3	<code>is_topology_connected_manual()</code> — Coverage Check	14
3.3	The Specialization Preorder on Finite Spaces	14
3.3.1	Theoretical Foundation: Finite Topological Spaces as Alexandrov Spaces	14
3.3.2	Definition: $S(x)$ and the Specialization Preorder	15
3.3.3	The Comparability Graph	15
3.3.4	Why the Base-Graph is Insufficient	15
3.3.5	Implementation	16
3.4	Why This Works Without Topology Enumeration	16
3.5	The Complement-Based Standalone Check	16
3.6	Legacy Heuristics Retained for Comparison	16
4	DIRECTED TOPOLOGIES, ALEXANDROV TOPOLOGY, AND BITOPOLOGICAL SPACES (v0.2.0)	17
4.1	Motivation: Temporal Irreversibility	17
4.2	Forward and Backward Nada Topologies	17
4.2.1	Forward Topology τ^+	17
4.2.2	Backward Topology τ^-	17
4.2.3	No New C++ Code Required	17
4.3	The Alexandrov Topology	17
4.3.1	Definition	17
4.3.2	Algorithm	18
4.3.3	The Identity Theorem for Upsets	18
4.4	Resolution Hierarchy	18
4.5	Bitopological Spaces (Kelly, 1963)	19
4.5.1	Definition	19
4.5.2	Irreversibility Indices	19
4.5.3	Pairwise Connectedness (Kelly, 1963)	19
5	IMPLEMENTATION ARCHITECTURE	20
5.1	Design Philosophy: Base as the Primary Object	20
5.2	C++ Backend via Rcpp	20
5.2.1	Multi-Word Bitset Representation	20
5.2.2	Template Dispatch for Compile-Time Optimization	21

5.2.3	Core Data Structures	21
5.2.4	Shared Bitset Infrastructure (<code>bitset_ops.h</code>)	22
5.2.5	Wavefront Propagation Pattern	22
5.3	OpenMP Parallelization	22
5.4	Algorithmic Complexities	23
5.5	CRAN Compliance	23
6	COMPLETE FUNCTION REFERENCE	24
6.1	Graph Construction Functions	24
6.1.1	<code>horizontal_visibility_graph(series, directed = FALSE)</code>	24
6.1.2	<code>natural_visibility_graph(series, directed = FALSE)</code>	25
6.2	Topology Generation	25
6.2.1	<code>generate_topology(adjacency, n_elements, max_open_sets, max_base_sets, check_connected, verify_axioms)</code>	25
6.2.2	<code>generate_alexandrov_topology(out_adjacency, n_elements, max_open_sets, check_connected, verify_axioms)</code>	27
6.2.3	<code>generate_bitotology(series, graph_type, max_open_sets, max_base_sets, alexandrov)</code>	28
6.2.4	<code>bitotology_invariants(forward, backward, n_elements, undirected, alexandrov)</code>	30
6.3	Connectivity Functions	31
6.3.1	<code>is_topology_connected_exact(topology, n_elements)</code>	31
6.3.2	<code>is_topology_connected(topology)</code> — Legacy	32
6.3.3	<code>is_topology_connected2(topology)</code> — Legacy	32
6.3.4	<code>is_topology_connected_manual(topology)</code> — Legacy	32
6.4	Threshold-Based Functions (Metric Approximation)	32
6.4.1	<code>calculate_thresholds(data)</code>	32
6.4.2	<code>calculate_topology(data, threshold)</code>	33
6.4.3	<code>visualize_topology_thresholds(data, plot = TRUE)</code>	33
6.4.4	<code>analyze_topology_factors(data, factors = NULL, plot = TRUE)</code>	34
6.5	Degenerate and Special Topologies	34
6.5.1	<code>simplest_topology(data)</code>	34
6.5.2	<code>complete_topology(data, verify_axioms = FALSE)</code>	35
7	WORKFLOW AND USAGE PATTERNS	35
7.1	Standard Workflow	35
7.2	Choosing Between HVG and NVG	36
7.3	Handling Large n	36
7.4	Comparing Graph-Theoretic vs. Threshold-Based Results	37

7.5	Bitopological Analysis Workflow (v0.2.0)	37
7.5.1	Step-by-Step Bitopological Analysis	38
7.5.2	Pairwise Connectedness (Small Series Only)	38
7.5.3	Comparing HVG and NVG Bitopologies	39
8	REFERENCES	39

1 THEORETICAL FOUNDATIONS

1.1 Introduction to Topological Spaces

1.1.1 Formal Definition

To understand the algorithm implementation, fundamental topological concepts must first be comprehended, specifically those of base, subbase, and their relationship with an object's topology.

Let X be an arbitrary set. A topology in X is any system τ of subsets G of X that verifies two conditions (Kolmogórov & Fomin, 1978, p. 90):

1. The set X itself and the empty set ($\emptyset$) belong to τ .
2. The union $\bigcup_{\alpha} G_{\alpha}$ of any number (finite or infinite) and the intersection $\bigcap_k G_k$ of a finite number of sets in τ belong to τ . Another way to define it, following Kelley (1955, p. 50), is that the intersection of any two members of τ is a member of τ (τ is closed under intersections) and that the union of the members of any family of τ is part of X (the topology space of τ is closed under arbitrary unions).

Thus, the set X is called the topology space of τ . τ is called the topology of X , and the pair (X, τ) is called a topological space, that is, $T = (X, \tau)$.

The sets belonging to the system of sets are called open (Kolmogórov & Fomin, 1978, p. 90).

As Kolmogórov & Fomin (1978, p. 90) point out, a metric space consists of a set of points and a metric introduced in that set; similarly, a topological space consists of a set of points and a topology introduced in it.

Consequently, defining a topological space means defining a set X and a topology on it, that is, indicating which subsets are considered open in X (Kolmogórov & Fomin, 1978, p. 90).

It is clear that different topologies can be introduced in the same set X , thus transforming it into different topological spaces (Kolmogórov & Fomin, 1978, p. 90).

However, the topological space, that is, the pair (X, τ) , will be denoted by T , and the elements of the topological space will be called "points".

1.1.2 Fundamental Properties

The sets $T \setminus G$, that is, the elements of T that do not belong to G , which are complementary to the open sets, are called closed sets of the topological space T (Kolmogórov & Fomin, 1978, p. 90).

The principle of duality consists of the fact that from any theorem concerning a system of subsets of a fixed set S , another dual theorem can be automatically deduced by replacing the original sets with their complements, the sum of sets with their intersection, and the intersection with the sum (Kolmogórov & Fomin, 1978, p. 16).

By virtue of the duality relations, from axioms 1 (the complement of the sum is equal to the intersection of the complements) and 2 (the complement of the intersection is equal to the sum of the complements), it follows that (Kolmogórov & Fomin, 1978, p. 90):

1. The empty set and the entire space T are closed.
2. The intersection of any number (finite or infinite) and the union of a finite number of closed sets are closed.

In every topological space, the concepts of neighborhood, adherence points, set adherence, etc., are naturally introduced from these definitions.

1.2 Bases and Subbases

1.2.1 Definition and Properties of Bases

As Kolmogórov & Fomin (1978, p. 91) point out, a neighborhood of the point $x \in T$ is any open set $G \subset T$ that contains the point x ; a point $x \in T$ is called an adherence point of the set $M \subset T$ when every neighborhood of point x contains at least one point of M ; a point x is called an accumulation point of the set M , when every neighborhood of the point x contains an infinite number of points of M . The totality of adherence points of the set M is called the adherence of the set M and is denoted by the symbol $[M]$ (Kolmogórov & Fomin, 1978, p. 91).

As Kolmogórov & Fomin (1978, p. 93) note, a collection ζ of open subsets is called a base of the topological space T , when every open subset of T can be represented as a certain number of sets from ζ .

For example, the collection of all open balls (of all possible radii and centers) constitutes a base in a metric space. In particular, the system of all intervals is a base on the line. From the above, it follows that the topology of the space T is defined if a base ζ is indicated in this space. (Kolmogórov & Fomin, 1978, p. 93)

As Kolmogórov & Fomin (1978, p. 93) point out, for this method of introducing topology to have practical value, it is necessary to indicate those conditions that a system ζ of subsets of the given set T must satisfy so that the collection of all possible sums of sets from ζ can be considered as the collection of open sets in T (that is, so that these sums verify axioms 1 and 2 of topological space). Such conditions are given by the following theorem:

Theorem. Suppose that in a set T , a system ζ of subsets G_α has been chosen that verifies the following conditions:

- a. Every point $x \in T$ is contained in at least one subset $G_\alpha \in \zeta$.
- b. If $x \in G_\alpha$ and $x \in G_\beta$, there exists a $G_\gamma \in \zeta$ such that $x \in G_\gamma \subset G_\alpha \cap G_\beta$.

Thus, if we declare open in T the empty set and all sets that can be represented as the sum of certain $G_\alpha \in \zeta$, the set T will result in a topological space (that is, these sums will verify axioms 1 and 2) and the system ζ will be a base of it (Kolmogórov & Fomin, 1978, p. 93).

1.2.2 Subbases and their Relationship with Bases

To prove whether a given collection of open sets is a base or not, the following criterion is often useful (Kolmogórov & Fomin, 1978, p. 94):

Theorem. For a system G_α of open sets to be a base of the topological space T , it is necessary and sufficient that for every open set G and every $x \in G$, there exists a set G_α from this system such that $x \in G_\alpha \subset G$.

Finally, Kelley (1955, p. 61) notes that a family S of sets is a sub-base of a topology if the family of all finite intersections of members of S is a base of τ (or equivalently: every member of τ is a union of finite intersections of members of S).

A more intuitive way to see it is that a subbase is a collection S of subsets of a topological space that is contained in a base of the topology and can be completed to form a base by adding all finite intersections of subsets of the set S .

1.3 Connectivity in Topological Spaces

1.3.1 Topological Definition of Connectivity

Kelley (1955, p. 67) notes that a topological space $T = (X, \tau)$ is connected if X is not the union of two separated non-empty subsets.

In other words, it is a subset of a topological space that cannot be described as a disjoint union of two non-empty open sets of the topological space in question.

The above means that it cannot be simultaneously true that, on one hand, if we separate the topological space into two sets, that is, the endpoints of each new set (resulting from the separation) are not included in these two sets, upon reuniting them the result is equivalent to the original set and, on the other hand, that this intersection is empty (that they have no elements in common).

This does not occur simultaneously because if they are separated into two subsets, it is possible to achieve that upon intersecting them (reuniting) the result is an empty set, but only at the cost of omitting some point from the original sets; on the other hand, it is possible to return to the original set, but only at the cost of not omitting any of the original points, in which case their intersection would not be an empty set.

The above can also be formally expressed by saying that a connected set is a subset $C \subset X$ of a topological space (X, τ) that cannot be described as a disjoint union of two non-empty open sets of the topology, where τ is the collection of open sets of the topological space; this subset, as its own definition reveals, may be equivalent to the complete topological space.

In other words, it is formed by a single piece and is not divisible because all its parts are connected in some way, so analyzing parts in isolation will affect the global properties of the set.

By contrast, in a non-connected or disconnected set, the situation is different.

A non-connected set can be separated into two or more disjoint open subsets, each of which is called a connected component.

This means that in disconnected sets, it is possible to analyze a part (i.e., a connected component) of the set independently, and in many cases, this will not affect the global properties of the set.

However, there are important nuances.

First, local properties can be completely analyzed in a connected component without affecting the others.

Second, some global properties will be maintained when analyzing a single component, while others will require considering all components; for example, the compactness of a component does not imply the compactness of the total set.

Some topological (i.e., essential) properties that can be analyzed locally (by components) are the connectivity of each component, the compactness of each component, or other more general local topological properties (for example, being locally Euclidean).

In contrast, global properties that will require considering all components, even when the set is disconnected, include the total number of connected components, the compactness of the total set (which requires all components to be compact and that there be a finite number of them), different metric or measure properties (to see if they are generalizable to the entire space), among others.

Thus, although it is possible to analyze components separately, it is crucial to remember that some global properties can only be understood by considering all components together.

The main difference between connected and non-connected sets in terms of local properties is that in a connected set, local properties can “propagate” or have global implications in ways that do not occur in non-connected sets.

For example, consider a continuous function f on a closed interval $[a, b]$ predefined as connected.

It is possible to analyze local properties such as differentiability at each point; however, the Intermediate Value Theorem, which is a global property, is derived from local continuity and the connectivity of the interval: consequently, what is not possible in a connected set is to completely isolate a part of the set from the rest, in topological terms, since there will always be some form of “connection” between the parts.

1.3.2 Connected Components

A connected component of a topological space (X, τ) is a maximal connected subset of X . Every topological space decomposes uniquely into its connected components: these are pairwise disjoint, and their union is X . A space is connected if and only if it has exactly one connected component.

Equivalently, a topological space is disconnected if and only if there exists a proper non-empty subset $U \subset X$ that is simultaneously open and closed (a *clopen* set). If such a clopen set U exists, then its complement $X \setminus U$ is also clopen, and $X = U \sqcup (X \setminus U)$ is a non-trivial partition into two open sets.

This clopen characterization is the basis for one of the connectivity algorithms implemented in this package (see Section 3.5).

1.3.3 Connectivity in Graphs and its Relation to Topological Connectivity

Once the topology (complete or approximated by optimization) is revealed, its connectivity can be determined by constructing a directed graph or an undirected graph.

A graph is a set of objects called vertices or nodes connected by links called edges or arcs, which allow representing binary relations between elements of a set (Trudeau, 1993, pp. 19-20).

An undirected graph is justified when it is valid to assume that the connection relationship between the topology elements is symmetric.

In other words, if element A is connected to element B, then B is also connected to A.

However, it is essential to distinguish between *graph connectivity* and *topological connectivity*. A graph being connected (every pair of vertices linked by a path of edges) is necessary but not sufficient for the induced topological space to be connected. The exact relationship between these two concepts is established through the specialization preorder, which is the subject of Section 3.

1.4 Intuitive Explanation of Theoretical Foundations

Imagine we have a set X representing the Cartesian plane.

Within this plane, we draw a circle with its corresponding circumference.

The circumference is the line that delimits the circle, while the circle is the area enclosed by the circumference.

Now, let’s introduce a topology in this set X . The topology is a collection of subsets of X that meets certain conditions; the Cartesian plane would be the underlying space from which the topology is constructed, while the plane plus the topology would be the topological space.

In our example, we can consider that the open sets in this topology are the circle (without including the circumference) and the area outside the circle (excluding the circumference).

These open sets satisfy the mentioned conditions: the set X (entire plane) and the empty set belong to τ , and arbitrary unions and finite intersections of open sets are also open.

On the other hand, the closed sets in this topology would be the circumference and the complete set X (including the circle and its exterior).

These closed sets satisfy the dual conditions: arbitrary intersections and finite unions of closed sets are also closed.

Now, imagine a point inside the circle. A neighborhood of x would be any open set that contains x , that is, any area within the circle that includes x .

A point y on the circumference would be an adherence point of the circle, since any neighborhood of y (any open area containing y) will always include at least one point of the circle.

The adherence of the circle would be the circle along with its circumference, as it includes all adherence points.

A base ζ of the topology would be a collection of open sets such that any open set in τ can be expressed as a union of elements of ζ . In our example, a base could be the collection of all open disks within the circle.

Finally, a sub-base S of the topology would be a collection of sets such that all finite intersections of elements of S form a base of τ . In our example, a sub-base could be the collection of all open regions that include the circle's center and extend to the circumference, along with all open regions outside the circle that extend to the circumference.

Any region of the plane (i.e., any set of points in space) for which if separated into two subsets it is possible to achieve that upon intersecting them (reuniting) the result is an empty set, but only at the cost of omitting some point from the original sets or returning to the original set, but only at the cost of not omitting any of the original points, so that their intersection would not be an empty set, is a connected set, a single piece, an indivisible totality.

2 GRAPH-THEORETIC TOPOLOGY CONSTRUCTION

2.1 The Nada et al. (2018) Framework

The central theoretical contribution implemented in `topologyR` is the framework of Nada et al. (2018), which establishes a principled method for inducing a topology from a graph. The procedure is as follows:

1. **Graph.** Start with a graph $G = (V, E)$, where V is a finite set of vertices and E is the set of edges.
2. **Closed neighborhoods.** For each vertex $v \in V$, define the closed neighborhood $N[v] = \{v\} \cup \{u \in V : (v, u) \in E\}$.
3. **Subbase.** The collection of all closed neighborhoods $\mathcal{S} = \{N[v] : v \in V\}$ forms the subbase of the topology.
4. **Base.** The base $\mathcal{B}$ is obtained by closing the subbase under all finite intersections: $\mathcal{B} = \left\{ \bigcap_{i=1}^k S_i : S_i \in \mathcal{S}, k \geq 1 \right\}$.
5. **Topology.** The topology τ is obtained by closing the base under arbitrary unions: $\tau = \left\{ \bigcup_{\alpha} B_{\alpha} : B_{\alpha} \in \mathcal{B} \right\} \cup \{\emptyset\}$.

The pair (V, τ) is then a finite topological space. Connectivity, connected components, and other topological properties can be determined from this space.

2.1.1 The Nada et al. (2018) Algorithm

The following pseudocode formalizes the procedure described above (Nada et al., 2018):

Input: A number of vertices of a graph G , Adjacency matrix of $V(G)$.

Output: A topology of G (TG).

1. Insert the number of vertices of G .
2. for $i \in n$ Enter the name of the vertices of a graph G . end
3. for $i \in n$ for $j \in n$ Enter the adjacency matrix of $V(G)$. end end
4. for $i \in n$ for $j \in n$ Calculate the degree of $V(G)$. end end
5. for $i \in n$ for $j \in n$ if (degree $\neq 0$) class(i, j)= $x(j)$. $R = (\text{degree}(x(i))x(t), \text{degree}(x(j))x(f))$. end end end
6. for $i \in n$ for $j \in n$ if (class(i, j) $\neq 0$) subbase class(j). end end end
7. for $v \in n$ for $vj \in n$ base subbase + intersection (v, vj). end end
8. for $vi \in n$ for $vj \in n$ if (union (v, vj) $\neq (\emptyset)$) $vi, vj \in$ union. union(vi, vj) $\in$ union. end topology base + union(vi, vj). end end

Important limitations of this pseudocode. Steps 7 and 8 as stated perform only *pairwise* intersections and unions. This is insufficient for a correct implementation: the closure under intersections (step 7) must reach a fixed point, not merely compute one round of pairwise intersections, because intersections of three or more subbase elements can produce sets not obtainable from any pair. Likewise, the closure under unions (step 8) must iterate to a fixed point. The **topologyR** implementation addresses this via wavefront propagation (see Sections 2.4 and 2.5).

2.1.2 The Role of Closed Neighborhoods

A critical detail in the Nada et al. (2018) framework is the use of **closed neighborhoods** $N[v] = \{v\} \cup N(v)$, not open neighborhoods $N(v) = \{u \in V : (v, u) \in E\}$. The distinction is consequential:

- With $N(v)$ (open neighborhood), vertex v does not appear in its own subbase element. The only way v appears in a base element is if some *other* vertex's neighborhood includes v . In sparse graphs, this produces extreme fragmentation: many vertices end up with incomparable neighborhood profiles, leading to a topology that is much finer (closer to discrete) than the theoretical framework predicts.
- With $N[v]$ (closed neighborhood), vertex v is guaranteed to appear in at least one subbase element — its own. This creates systematic overlap between the neighborhood profiles of adjacent vertices, producing a topology that correctly reflects the graph's connectivity structure.

The standard convention in the literature on graph-induced topologies (El Atik et al., 2020; Nada et al., 2018) is $N[v]$. Using $N(v)$ is not a design choice but an error that produces a different (and spuriously fragmented) topological space.

Concrete example. Consider the complete graph K_6 on 6 vertices. With $N[v]$, every closed neighborhood is $N[v] = V$ (all 6 vertices), so the subbase consists of a single set V , the base is $\{V\}$, and the topology is $\{\emptyset, V\}$ — the indiscrete topology, which is connected. This is correct: the complete graph should produce a single connected component. With $N(v) = V \setminus \{v\}$, the neighborhoods are all distinct (each one misses a different vertex), pairwise intersections produce singletons, and the resulting topology is the discrete topology 2^V with $2^6 = 64$ open sets — maximally disconnected. This is incorrect for the framework.

2.2 Visibility Graphs as the Graph Source

The Nada et al. (2018) framework requires a graph as input. For time series analysis, the question is: which graph captures the relevant structure of the series without introducing arbitrary parameters?

The `topologyR` package implements two parameter-free graph construction methods from the time series literature: the Horizontal Visibility Graph (HVG) and the Natural Visibility Graph (NVG). Both derive the graph structure entirely from the geometry of the series, with no thresholds, bandwidths, or tuning parameters.

2.2.1 Horizontal Visibility Graph (HVG)

The Horizontal Visibility Graph was introduced by Luque et al. (2009). Given a time series $\{y_1, y_2, \dots, y_n\}$, two observations (t_a, y_a) and (t_b, y_b) with $t_a < t_b$ are connected by an edge if and only if every intermediate observation (t_c, y_c) with $t_a < t_c < t_b$ satisfies:

$$y_c < \min(y_a, y_b)$$

Intuitively, two data points can “see” each other horizontally if no intermediate point is as tall as either of them.

Key properties of the HVG:

- **Always connected as a graph.** Every pair of temporally adjacent observations (y_i, y_{i+1}) is connected (there are no intermediate points to obstruct), so the HVG is always connected as a graph with at least $n - 1$ edges.
- **Parameter-free.** The graph depends only on the relative ordering of the values, with no thresholds or parameters.
- **$O(n)$ complexity.** The implementation uses a monotone stack algorithm where each element is pushed and popped at most once.
- **Local structure.** The HVG primarily captures local dynamics: edges tend to be short-range, connecting observations to nearby neighbors. Long-range edges only exist when a local maximum dominates a wide interval.

Algorithm. The stack-based HVG algorithm works as follows. A stack maintains a sequence of indices whose corresponding values form a non-increasing sequence. For each new observation y_i :

1. Pop all elements from the stack whose value is strictly less than y_i , adding an edge between each popped element and i .
2. If the stack is non-empty after popping, add an edge between the current stack top and i (the first element that is $\geq y_i$).
3. Push i onto the stack.

Equal values require special handling: when the stack top has the same value as y_i , the algorithm sets a flag to connect y_i to the element *below* the stack top (if it exists), since equal-height barriers do block horizontal visibility in the strict sense.

2.2.2 Natural Visibility Graph (NVG)

The Natural Visibility Graph was introduced by Lacasa et al. (2008). Two observations (t_a, y_a) and (t_b, y_b) with $t_a < t_b$ are connected if and only if every intermediate observation (t_c, y_c) with $t_a < t_c < t_b$ satisfies:

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

This is the geometric visibility condition: the point (t_c, y_c) must lie below the straight line connecting (t_a, y_a) and (t_b, y_b) . Two points can “see” each other if the line of sight between them is not obstructed by any intermediate point.

Key properties of the NVG:

- **Superset of HVG.** Every HVG edge is also an NVG edge. If two points can see each other horizontally, they can certainly see each other along the line connecting them. Formally, $E_{\text{HVG}} \subseteq E_{\text{NVG}}$.
- **Richer structure.** The NVG captures both local and medium-range dynamics. Peaks can “see over” intermediate valleys to connect with distant points if the slope condition is satisfied.
- **$O(n \log n)$ expected complexity.** The divide-and-conquer algorithm partitions on the global maximum, which blocks all cross-visibility between the two halves. Edges from the pivot are found by a linear slope scan in each direction. Expected time is $O(n \log n)$; worst case (monotone sequences) is $O(n^2)$.
- **Dynamical sensitivity.** The NVG preserves dynamical properties of the original series, including periodicity, chaoticity, and long-range correlations (Lacasa et al., 2008).

Algorithm. The divide-and-conquer NVG algorithm:

1. Find the global maximum y_p at position p in the range $[l, r]$.
2. From p , scan left: track the maximum slope $\sigma = (y_p - y_j)/(p - j)$ seen so far. For each $j < p$, if the slope from p to j exceeds all intermediate slopes, add edge (j, p) .
3. From p , scan right: analogous slope scan.
4. Recurse on $[l, p - 1]$ and $[p + 1, r]$.

2.2.3 HVG vs. NVG: Comparison

Property	HVG	NVG
Visibility condition	Horizontal ($y_c < \min(y_a, y_b)$)	Geometric (below line of sight)
Edge set	Subset of NVG	Superset of HVG
Density	Lower	Higher
Range of edges	Primarily local	Local to medium-range
Complexity	$O(n)$	$O(n \log n)$ expected
Topological components	More (finer fragmentation)	Fewer (coarser structure)
Best for	Local dynamics, regime detection	Global structure, connectivity

In practice, the NVG produces a denser graph with more overlap between neighborhoods, leading to fewer topological connected components. The HVG, being sparser, produces finer topological fragmentation. Both are valid; the choice depends on whether the analyst prioritizes local or global dynamical structure.

2.2.4 Directed Visibility Graphs (v0.2.0)

Time series are inherently directional: time flows from past to future. The undirected visibility graph discards this information. Version 0.2.0 introduces **directed visibility graphs** via the `directed = TRUE` parameter in both `horizontal_visibility_graph()` and `natural_visibility_graph()`.

When `directed = TRUE`, the functions return two additional adjacency lists:

- **out_adjacency** (forward neighbors): For each vertex v , the set of vertices w such that there is an edge from v to w and $w > v$ (later in time). These form the **forward neighborhood** $N^+(v) = \{w : v \rightarrow w, w > v\}$.
- **in_adjacency** (backward neighbors): For each vertex v , the set of vertices w such that there is an edge from w to v and $w < v$ (earlier in time). These form the **backward neighborhood** $N^-(v) = \{w : w \rightarrow v, w < v\}$.

By construction, all directed edges satisfy from $<$ to (earlier time to later time), so the directed graph is a **directed acyclic graph (DAG)** with the time index serving as topological order.

Key property: The union of forward and backward neighbors recovers the undirected neighbors: $N^+(v) \cup N^-(v) = N(v)$ for all v .

Purpose of directed adjacency: The forward and backward adjacency lists serve as input for three distinct topological constructions:

1. **Forward Nada topology** τ^+ : Pass `out_adjacency` to `generate_topology()`. The engine constructs closed forward neighborhoods $N^+[v] = \{v\} \cup N^+(v)$.
2. **Backward Nada topology** τ^- : Pass `in_adjacency` to `generate_topology()`. The engine constructs closed backward neighborhoods $N^-[v] = \{v\} \cup N^-(v)$.
3. **Alexandrov topology** τ_A : Pass `out_adjacency` to `generate_alexandrov_topology()`, which computes reachability upsets.

These constructions are detailed in Section 2.

2.2.5 Why Not Threshold-Based Graphs?

The `topologyR` package also retains legacy threshold-based functions (see Section 5.4) for historical comparison. In threshold-based construction, two points x_i and x_j are connected if $|x_i - x_j| \leq \epsilon$ for some threshold ϵ . This approach has a fundamental problem: the resulting topology depends critically on the arbitrary choice of ϵ . Different threshold methods (mean difference, median difference, standard deviation, IQR, DBSCAN-derived) can produce base sizes varying by a factor of 7 or more on the same data (e.g., from 119 to 836 base elements on a 129-observation series).

Visibility graphs eliminate this dependency entirely. The graph is determined uniquely by the series, and the resulting topology is a well-defined mathematical object.

2.3 From Subbase to Base: Fixed-Point Closure Under Finite Intersections

Given the subbase $\mathcal{S} = \{N[v] : v \in V\}$, the base is obtained by closing under all finite intersections. This is not simply pairwise intersections — the closure must include intersections of three, four, or more subbase elements if they produce new sets.

The `topologyR` implementation computes this closure using a **wavefront propagation** algorithm:

1. **Initialization.** Seed the base with all subbase elements: $\mathcal{B}_0 = \mathcal{S}$.
2. **Wavefront iteration.** At each iteration, compute the intersection of every element in the current frontier with every element in the accumulated base. Any new set (not already in $\mathcal{B}$) is added to both $\mathcal{B}$ and the next frontier.
3. **Fixed point.** When the frontier is empty (no new sets are produced), the closure is complete: $\mathcal{B}$ is the base.

The empty set is excluded from the base (it will be added to the topology directly).

Safety limit. The base can grow combinatorially. The `max_base_sets` parameter (default: 100,000) halts the closure if the base exceeds this size. The return field `base_complete` indicates whether the closure finished within the limit (`TRUE`) or was truncated (`FALSE`). Connectivity results computed from a truncated base remain valid but may be approximate — an incomplete base can only underestimate the overlap between neighborhood profiles, potentially over-counting components.

2.4 From Base to Topology: Fixed-Point Closure Under Arbitrary Unions

The topology is obtained by closing the base under arbitrary unions:

$$\tau = \left\{ \bigcup_{\alpha} B_{\alpha} : B_{\alpha} \in \mathcal{B} \right\} \cup \{\emptyset, V\}$$

The same wavefront propagation pattern is used:

1. **Initialization.** Seed the topology with $\emptyset$, V , and all base elements.
2. **Wavefront iteration.** At each step, compute the union of every frontier element with every accumulated topology element. New sets enter the frontier.
3. **Fixed point.** When no new unions are produced, the topology is complete.

Safety limit. The `max_open_sets` parameter (default: 1,000,000) halts enumeration if the topology exceeds this size. The return field `complete` indicates whether the enumeration finished (`TRUE`) or was truncated (`FALSE`).

Important: Topology enumeration is **optional** for connectivity analysis. The specialization preorder (Section 3.3) determines exact connected components from the base alone, without requiring the full topology. Setting `max_open_sets = 0` skips enumeration entirely, which is recommended when only connectivity is needed, especially for large n where the topology can be astronomically large.

2.5 Safety Limits and Informative Termination

The package provides two safety limits to prevent unbounded computation:

Parameter	Default	Controls	Indicator
<code>max_base_sets</code>	100,000	Intersection closure (subbase $\rightarrow$ base)	<code>base_complete</code>
<code>max_open_sets</code>	1,000,000	Union closure (base $\rightarrow$ topology)	<code>complete</code>

When a limit is reached, the computation terminates gracefully:

- Results computed *before* the limit are valid and returned.
- The corresponding indicator flag is set to `FALSE`.
- Connectivity (from the specialization preorder) is always computed from whatever base is available, regardless of whether enumeration completed.

Additionally, if topology enumeration is requested on a space that has already been determined to be disconnected (via the preorder), the engine emits a warning suggesting `max_open_sets = 0` to skip the potentially expensive enumeration.

3 CONNECTIVITY DETERMINATION

3.1 The Exact Topological Definition

A topological space (X, τ) is connected if and only if the only subsets of X that are simultaneously open and closed (clopen) are $\emptyset$ and X itself (Kelley, 1955, p. 67). Equivalently, (X, τ) is disconnected if and only if there exists a proper non-empty open set $U \in \tau$ whose complement $X \setminus U$ is also in τ .

This is the gold standard: any algorithm that correctly determines connectivity must, directly or indirectly, verify this condition.

3.2 Why the Three Legacy Heuristics Fail

The original `topologyR` package included three connectivity functions, all of which are necessary but not sufficient conditions for topological connectivity. They are retained for backward compatibility and comparison, but should not be used for definitive connectivity determination.

3.2.1 `is_topology_connected()` — Undirected Graph DFS

This function constructs an undirected graph where two elements share an edge if they appear together in any open set, then checks graph connectivity via depth-first search. **Why it fails:** Two elements appearing in the same open set does not imply they are in the same connected component of the topological space. The undirected graph can be connected while the topology is disconnected (the topology $\{\emptyset, \{1, 2\}, \{3, 4\}, \{1, 2, 3, 4\}\}$ has all four elements appearing together in the full set, making the graph connected, but $\{1, 2\}$ and $\{3, 4\}$ are clopens, so the space is disconnected).

3.2.2 `is_topology_connected2()` — Directed Graph DFS

This function creates directed edges between consecutive sorted elements within each open set, then checks reachability from the minimum element. **Why it fails:** The sequential ordering within an open set has no topological meaning. Reachability in this directed graph is an even weaker condition than undirected graph connectivity.

3.2.3 `is_topology_connected_manual()` — Coverage Check

This function checks whether every element from 1 to $\max(V)$ appears in at least one open set. **Why it fails:** This verifies $\bigcup \tau = X$ (coverage), which is always true for any topology (since $X \in \tau$). Coverage says nothing about connectivity. The discrete topology has full coverage and is maximally disconnected.

3.3 The Specialization Preorder on Finite Spaces

The `topologyR` package uses the **specialization preorder** to determine exact topological connected components directly from the base, without requiring the full topology. This is the central algorithmic innovation.

3.3.1 Theoretical Foundation: Finite Topological Spaces as Alexandrov Spaces

Every finite topological space is an Alexandrov space: arbitrary intersections of open sets are open (not just finite intersections), because in a finite space every collection is finite. The structure theory of finite topological spaces was established by McCord (1966) and Stong (1966):

Theorem (McCord, 1966; Stong, 1966). *Let (X, τ) be a finite topological space. Define the specialization preorder $\leq$ on X by $x \leq y$ if and only if every open set containing x also contains y (equivalently, $x \in \overline{\{y\}}$, the closure of $\{y\}$). Then the connected components of (X, τ) coincide with the connected components of the comparability graph of $\leq$.*

The comparability graph has vertex set X and an edge between x and y whenever $x \leq y$ or $y \leq x$ (i.e., whenever x and y are comparable in the preorder).

3.3.2 Definition: $S(x)$ and the Specialization Preorder

For practical computation, the specialization preorder is expressed in terms of the base. Define, for each element $x \in V$:

$$S(x) = \{B \in \mathcal{B} : x \in B\}$$

That is, $S(x)$ is the set of all base elements that contain x . Then:

$$x \leq y \iff S(x) \subseteq S(y)$$

The intuition is: $x \leq y$ means that y is “at least as distinguishable” as x in the topology — every open set that can “see” x can also “see” y .

3.3.3 The Comparability Graph

Two elements $x, y \in V$ are **comparable** if $S(x) \subseteq S(y)$ or $S(y) \subseteq S(x)$. The comparability graph G_{comp} has:

- **Vertices:** $V = \{1, 2, \dots, n\}$.
- **Edges:** (x, y) if x and y are comparable.

Theorem. The connected components of G_{comp} are exactly the topological connected components of (V, τ) .

This is an exact result, not an approximation.

3.3.4 Why the Base-Graph is Insufficient

A simpler approach might be to build a graph where two elements are connected if they share any base element, then check graph connectivity. This fails. Consider $V = \{1, 2, 3, 4\}$ with base $\mathcal{B} = \{\{1, 2\}, \{2, 3\}, \{3, 4\}\}$:

- The “sharing” graph is a path $1 - 2 - 3 - 4$, which is connected.
- But $\{1, 2\}$ and $\{3, 4\}$ are clopens (each is a union of base elements, and the complement of each is also a union of base elements), so the space is disconnected into $\{1, 2\}$ and $\{3, 4\}$.
- The specialization preorder correctly identifies this: $S(1) = \{\{1, 2\}\}$, $S(2) = \{\{1, 2\}, \{2, 3\}\}$, $S(3) = \{\{2, 3\}, \{3, 4\}\}$, $S(4) = \{\{3, 4\}\}$. No pair between $\{1, 2\}$ and $\{3, 4\}$ is comparable: $S(1) \not\subseteq S(3)$, $S(3) \not\subseteq S(1)$, etc. The comparability graph has two connected components: $\{1, 2\}$ and $\{3, 4\}$.

3.3.5 Implementation

The algorithm is implemented in C++ and works as follows:

1. **Encode $S(x)$ as bitsets.** Let $B = |\mathcal{B}|$ (the number of base elements). For each $x \in V$, encode $S(x)$ as a bitset of $W_{\text{base}} = \lceil B/64 \rceil$ 64-bit words, where bit b is set if $x \in \mathcal{B}_b$.
2. **BFS on the comparability graph.** For each unvisited element x , start a BFS. For each neighbor candidate y , check if $S(x) \subseteq S(y)$ or $S(y) \subseteq S(x)$ using word-by-word bitwise operations. If comparable, y is in the same component.
3. **Output.** The connected components partition V into groups. The number of groups and their membership are the exact topological connected components.

The subset check $S(x) \subseteq S(y)$ is computed as: for every word w , verify that $S(x)[w] \& S(y)[w] = S(x)[w]$. This is a constant-time operation per word.

3.4 Why This Works Without Topology Enumeration

The key insight is that the base carries all the information needed for connectivity. The full topology τ is obtained from the base by taking arbitrary unions, but the specialization preorder depends only on which base elements contain each point. Since every open set is a union of base elements, the set of open sets containing x is completely determined by $S(x)$: an open set U contains x if and only if some base element in $S(x)$ is a subset of U .

This means the specialization preorder can be computed in polynomial time from the base, even when the topology itself has exponentially many open sets. For a base of size B and n elements, the comparability graph construction is $O(n^2 \cdot W_{\text{base}})$ where $W_{\text{base}} = \lceil B/64 \rceil$. This is vastly more efficient than enumerating the full topology, which can have $O(2^B)$ elements.

3.5 The Complement-Based Standalone Check

The function `is_topology_connected_exact()` provides an alternative exact connectivity check based directly on the clopen definition: it iterates over all open sets in the topology and checks whether any proper non-empty open set has its complement also in the topology (using hash lookup).

This method is exact but requires the **full topology** to be enumerated. It is only applicable when the topology fits within the `max_open_sets` limit. When the topology is too large to enumerate (which is common for $n > 20$), the specialization preorder is the only feasible exact method.

When to use which:

Method	Requires	Complexity	Result
Specialization preorder	Base only	$O(n^2 \cdot \lceil B/64 \rceil)$	Exact components
Complement check	Full topology	$O(\tau \cdot W)$	Connected / first clopen pair

3.6 Legacy Heuristics Retained for Comparison

The three legacy functions (`is_topology_connected`, `is_topology_connected2`, `is_topology_connected_manual`) are retained in the package for backward compatibility and for pedagogical comparison. Their documentation explicitly notes that they are approximations, not exact connectivity checks. Users are directed to use `generate_topology()` with `check_connected = TRUE` for exact results.

4 DIRECTED TOPOLOGIES, ALEXANDROV TOPOLOGY, AND BITOPOLOGICAL SPACES (v0.2.0)

4.1 Motivation: Temporal Irreversibility

In time series analysis, many dynamical processes exhibit **temporal irreversibility**: the statistical properties of the series read forward in time differ from those read backward. Asymmetric business cycles (gradual expansions, abrupt recessions) are a canonical example. The undirected topology τ from Nada et al. (2018) treats all visibility edges symmetrically, discarding the temporal direction. Version 0.2.0 introduces a **bitopological** framework that explicitly captures this asymmetry.

4.2 Forward and Backward Nada Topologies

Given a directed visibility graph (DAG), we define two directed topologies using the same Nada et al. (2018) procedure applied to directed neighborhoods:

4.2.1 Forward Topology τ^+

The **forward Nada topology** is generated from the forward closed neighborhoods:

$$N^+[v] = \{v\} \cup \{w : v \rightarrow w\}$$

This subbase is passed to `generate_topology()`, which computes the intersection/union closure as usual. The forward topology captures the topological structure of “looking ahead” in time: how the series connects from each point to its future-visible neighbors.

4.2.2 Backward Topology τ^-

The **backward Nada topology** is generated from the backward closed neighborhoods:

$$N^-[v] = \{v\} \cup \{w : w \rightarrow v\}$$

This captures the structure of “looking back” in time. In a reversible process, $\tau^+ \cong \tau^-$; in an irreversible process, they diverge.

4.2.3 No New C++ Code Required

A key design insight: the existing Nada topology engine already supports directed neighborhoods. The engine takes an adjacency list and constructs closed neighborhoods $N[v] = \{v\} \cup \text{adj}[v]$. When passed `out_adjacency`, it produces $N^+[v]$; when passed `in_adjacency`, it produces $N^-[v]$. No modifications to `topology_engine.cpp` were needed.

4.3 The Alexandrov Topology

4.3.1 Definition

The **Alexandrov topology** on a finite set equipped with a preorder is the canonical topology whose open sets are exactly the **upsets** (upper sets) of the preorder. A subset $U \subseteq V$ is an upset if $v \in U$ and $v \leq w$ imply $w \in U$.

For a directed visibility graph (DAG), the natural preorder is the **reachability relation**: $v \leq w$ if there exists a directed path from v to w . The minimal open set for vertex v is then the set of all vertices reachable from v :

$$U_v = \{w \in V : v \rightsquigarrow w\} \cup \{v\}$$

4.3.2 Algorithm

The reachability sets are computed by reverse-order bitset propagation:

1. Initialize $\text{reach}[v] = \{v\}$ for all v (as bitsets).
2. Process vertices in reverse order ($v = n, n-1, \dots, 1$).
3. For each forward neighbor w of v : $\text{reach}[v] \mid= \text{reach}[w]$.

After propagation, $\text{reach}[v]$ is exactly the set of vertices reachable from v , and the collection $\{U_v = \text{reach}[v] : v \in V\}$ forms the base of the Alexandrov topology.

Complexity: $O(n \cdot m/64)$ where m is the number of directed edges. The implementation uses the same template dispatch as the Nada engine (compile-time $W = 1, 2, 3$ plus runtime fallback).

Key property: For the Alexandrov topology, the base is already closed under intersections (upsets of a preorder are closed under intersection). This means **base_complete** is always **TRUE** and no wavefront iteration is needed.

4.3.3 The Identity Theorem for Upsets

A non-obvious but crucial mathematical result underpins the correctness of the implementation:

Theorem. *The upsets of a relation R and the upsets of its transitive closure R^* are identical.*

Proof sketch. Any upset of R^* is trivially an upset of R (since $R \subseteq R^*$). Conversely, let U be an upset of R : if $a \in U$ and aR^*b (meaning there exists a chain $a = z_0 R z_1 R \dots R z_k = b$), then by induction on the chain length, $z_1 \in U$ (since $a \in U$ and aRz_1), then $z_2 \in U$ (since $z_1 \in U$ and z_1Rz_2), and so on, yielding $b \in U$. Hence U is also an upset of R^* . $\square$

Consequence: The Alexandrov topology from direct visibility edges is *identical* to the Alexandrov topology from the full reachability relation. We do not need to compute the transitive closure explicitly — the reverse-order propagation produces the correct reachability sets directly.

4.4 Resolution Hierarchy

The Alexandrov base is always a subset of the Nada base:

$$\tau_A \subseteq \tau_{\text{Nada}}^+$$

This is because the Nada intersection closure generates sets that are **not upsets** of the reachability relation. The additional sets arise from intersections of neighborhoods.

Concrete example: Consider the path DAG $1 \rightarrow 2 \rightarrow 3$.

- **Nada base:** $N^+[1] = \{1, 2\}$, $N^+[2] = \{2, 3\}$, $N^+[3] = \{3\}$. Intersection closure adds $\{1, 2\} \cap \{2, 3\} = \{2\}$. Base = $\{\{1, 2\}, \{2, 3\}, \{3\}, \{2\}\}$ (4 elements).

- **Alexandrov base:** $\text{reach}[1] = \{1, 2, 3\}$, $\text{reach}[2] = \{2, 3\}$, $\text{reach}[3] = \{3\}$. $\text{Base} = \{\{1, 2, 3\}, \{2, 3\}, \{3\}\}$ (3 elements).

The set $\{2\}$ is in the Nada base but is **not** an upset (since $2 \rightarrow 3$ but $3 \notin \{2\}$). The difference $|\mathcal{B}_{\text{Nada}}| - |\mathcal{B}_A|$ quantifies how much additional topological structure the Nada closure captures beyond the pure order structure.

The `bitopology_invariants()` function reports this difference as `nada_forward_base_gain` and `nada_backward_base_gain`, along with relative gains.

4.5 Bitopological Spaces (Kelly, 1963)

4.5.1 Definition

A **bitopological space** is a triple (X, τ_1, τ_2) where τ_1 and τ_2 are two topologies on the same set X (Kelly, 1963). In our setting, $\tau_1 = \tau^+$ (forward Nada topology) and $\tau_2 = \tau^-$ (backward Nada topology).

4.5.2 Irreversibility Indices

The divergence between τ^+ and τ^- quantifies temporal irreversibility through two normalized indices:

Component irreversibility:

$$I_C = \frac{|C^+ - C^-|}{\max(C^+, C^-)} \in [0, 1]$$

where C^+ and C^- are the number of connected components in τ^+ and τ^- respectively. $I_C = 0$ means equal component counts (necessary for reversibility). $I_C = 1$ means maximally asymmetric (one topology is connected, the other has the maximum number of components).

Base irreversibility:

$$I_B = \frac{|B^+ - B^-|}{\max(B^+, B^-)} \in [0, 1]$$

where B^+ and B^- are the base sizes. This measures asymmetry in topological fineness.

Asymmetry direction:

$$D = C^- - C^+$$

Positive D means the forward topology is more connected (fewer components) than the backward topology. For time series with gradual expansions and abrupt contractions (e.g., asymmetric business cycles), the prediction is $D > 0$: forward visibility during gradual rises encounters fewer obstructions than backward visibility after abrupt drops.

4.5.3 Pairwise Connectedness (Kelly, 1963)

The bitopological space (X, τ^+, τ^-) is **pairwise disconnected** if there exists a proper non-empty subset $A \subset X$ that is simultaneously τ^+ -open and τ^- -closed (i.e., its complement $X \setminus A$ is τ^- -open). This is stronger than either topology being individually disconnected.

The `bitopology_invariants()` function checks this by iterating over all τ^+ open sets and testing whether their complement is τ^- -open (via hash lookup). This requires both topologies to be fully enumerated (`max_open_sets > 0` and `complete = TRUE`).

5 IMPLEMENTATION ARCHITECTURE

5.1 Design Philosophy: Base as the Primary Object

The architecture of `topologyR` is organized around a key insight: **the base is the primary mathematical object**, not the topology.

In a finite topological space:

- The **topology** can be exponentially large (up to $2^{|B|}$ open sets), making full enumeration infeasible for even moderate-sized problems.
- The **base** is typically much smaller and carries all the information needed for computing connectivity (via the specialization preorder), verifying topological properties, and understanding the structure of the space.
- **Topology enumeration is optional** and is only needed when the user wants to inspect individual open sets or run the complement-based connectivity check.

This design principle — compute the base first, derive connectivity from it, and enumerate the topology only if requested and feasible — avoids the combinatorial explosion that makes naive topology construction impractical for real-world data.

5.2 C++ Backend via Rcpp

All computationally intensive operations are implemented in C++ and exposed to R via Rcpp. The C++ backend consists of four source files:

File	Purpose
<code>topology_engine.cpp</code>	Nada topology: subbase $\rightarrow$ base $\rightarrow$ connectivity $\rightarrow$ topology
<code>alexandrov_engine.cpp</code>	Alexandrov topology: reverse-order bitset reachability propagation
<code>bitset_ops.h</code>	Shared bitset infrastructure (templates, hash sets, flat vectors)
<code>visibility_graph.cpp</code>	HVG and NVG construction

The backend handles:

- Subbase encoding from adjacency lists (undirected, forward, or backward).
- Wavefront propagation for intersection closure (subbase $\rightarrow$ base).
- Specialization preorder computation and BFS.
- Wavefront propagation for union closure (base $\rightarrow$ topology).
- Reverse-order bitset propagation for Alexandrov reachability.
- Axiom verification.
- The complement-based standalone connectivity check.

5.2.1 Multi-Word Bitset Representation

Each subset of $V = \{1, 2, \dots, n\}$ is represented as a bitset of $W = \lceil n/64 \rceil$ unsigned 64-bit integers. Bit i of the bitset is 1 if element $i + 1$ is in the set. All set operations (union, intersection, complement, subset test, equality, hashing) are implemented as word-by-word bitwise operations:

Operation	Implementation	Cost per operation
Union ($A \cup B$)	<code>dst[w] = a[w] b[w]</code> for each word w	W OR instructions
Intersection ($A \cap B$)	<code>dst[w] = a[w] & b[w]</code> for each word w	W AND instructions
Complement ($V \setminus A$)	<code>dst[w] = ~a[w] & mask[w]</code> for each word w	W AND-NOT instructions
Subset ($A \subseteq B$)	<code>(a[w] & b[w]) == a[w]</code> for each word w	W AND + CMP instructions
Equality ($A = B$)	<code>a[w] == b[w]</code> for each word w	W CMP instructions
Hash	Splitmix64 per word, combined via boost-style mixing	W multiply-shift operations

This representation supports arbitrary n with no upper limit.

5.2.2 Template Dispatch for Compile-Time Optimization

The engine uses C++ templates parameterized by W (the number of 64-bit words per bitset). At compile time, three specializations are instantiated:

Template	Range of n	Optimization
<code>engine_impl<1></code>	$n \leq 64$	All loops have $W = 1$ iterations, compiler eliminates loop overhead entirely. Each set operation compiles to a single machine instruction.
<code>engine_impl<2></code>	$65 \leq n \leq 128$	Loops unrolled to pairs of instructions.
<code>engine_impl<3></code>	$129 \leq n \leq 192$	Loops unrolled to triples of instructions.
<code>engine_runtime</code>	$n > 192$	Runtime loop with W passed as a parameter. Still fast (word-by-word operations), but with loop overhead.

The dispatch is a compile-time `switch` on W :

```
switch (W) {
  case 1: return engine_impl<1>(...);
  case 2: return engine_impl<2>(...);
  case 3: return engine_impl<3>(...);
  default: return engine_runtime(...);
}
```

For most practical applications ($n \leq 192$), all bitwise operations are fully unrolled at compile time with zero loop overhead.

5.2.3 Core Data Structures

FlatVec<W> — A contiguous vector of W -word bitsets. Internally, a single `std::vector<uint64_t>` of size $\text{count} \times W$, with index arithmetic to access element i at offset $i \times W$. This provides cache-friendly sequential access and avoids pointer indirection.

FlatHash<W> — An open-addressing hash set for W -word bitsets. Uses linear probing with a load factor limit of 0.7. The hash function is splitmix64 applied to each word and combined via boost-style mixing. This provides $O(1)$ amortized insertion and lookup for deduplication during wavefront propagation.

Both data structures have runtime- W counterparts (**RtVec**, **RtHash**) used in the fallback path for $W > 3$.

5.2.4 Shared Bitset Infrastructure (**bitset_ops.h**)

In v0.2.0, all bitset operations, data structures, and helper functions were extracted from **topology_engine.cpp** into a shared header **bitset_ops.h**. This header provides:

- **BitOps<W>**: Templated struct with static methods for OR, AND, NOT, equality, zero check, subset test, hash, copy, and zero fill. When W is a compile-time constant, all loops are eliminated by the compiler.
- **bs_test_bit()** / **bs_set_bit()**: Inline functions for single-bit operations.
- **FlatVec<W>** and **FlatHash<W>**: Templated containers (described above).
- **decode_bitset()**: Converts a bitset to an R integer vector (1-based).
- **compute_full_set()**: Constructs the full-set bitset (all n bits set).
- **rtops namespace**: Runtime- W bitset operations (same interface as **BitOps<W>** but with W as a parameter).
- **RtVec** and **RtHash**: Runtime- W containers.

Both **topology_engine.cpp** and **alexandrov_engine.cpp** include this header, eliminating code duplication while maintaining full compile-time optimization for $W \leq 3$.

5.2.5 Wavefront Propagation Pattern

Both closure operations (intersection for the base, union for the topology) use the same algorithmic pattern:

```
frontier = initial_elements
accumulated = initial_elements
repeat:
    new_frontier = {}
    for f in frontier:
        for a in accumulated:
            result = operation(f, a) // intersection or union
            if result is new (not in hash set):
                add result to accumulated
                add result to new_frontier
    frontier = new_frontier
until frontier is empty OR safety limit reached
```

This is equivalent to a breadth-first exploration of the closure, where each “level” represents one round of new combinations. The hash set ensures each set is processed exactly once. The algorithm terminates because the universe of possible subsets is finite.

5.3 OpenMP Parallelization

The package is compiled with OpenMP support (**-fopenmp** flag). The OpenMP thread count can be controlled via:

```
Sys.setenv(OMP_NUM_THREADS = "8")
```

The OpenMP flags are configured in `Makevars` / `Makevars.win` following CRAN conventions:

```
PKG_CXXFLAGS = $(SHLIB_OPENMP_CXXFLAGS)
PKG_LIBS = $(SHLIB_OPENMP_CXXFLAGS)
CXX_STD = CXX17
```

On systems without OpenMP support, the package compiles and runs correctly in single-threaded mode (the OpenMP pragmas are no-ops).

5.4 Algorithmic Complexities

Operation	Complexity	Notes
HVG construction	$O(n)$	Stack-based, each element pushed/popped once
NVG construction	$O(n \log n)$ expected	Divide-and-conquer; $O(n^2)$ worst case
Subbase encoding	$O(n \cdot d_{\max} \cdot W)$	$d_{\max}$ = max degree
Base closure (intersections)	$O(\mathcal{B} ^2 \cdot W)$ per iteration	Wavefront; typically few iterations
Specialization preorder	$O(n^2 \cdot W_{\text{base}})$	BFS on comparability graph
Topology enumeration (unions)	$O(\tau ^2 \cdot W)$ per iteration	Can be very large; optional
Axiom verification	$O(\tau ^2 \cdot W)$	Pairwise intersection check
Alexandrov reachability	$O(n \cdot m/64)$	Reverse-order bitset propagation; m = edges
Bitopological analysis	$3 \times \text{Nada} + 1 \times \text{Alexandrov}$	Four topology constructions
Pairwise connectedness check	$O(\tau^+ \cdot W)$	Hash lookup for complements in τ^-
Complement connectivity check	$O(\tau \cdot W)$	Hash lookup for each complement

5.5 CRAN Compliance

The package follows CRAN policies:

- No `print()` or `cat()` calls in function bodies; diagnostic output uses `Rcpp::warning()`.
- English parameter names and documentation.
- Proper DESCRIPTION metadata with `NeedsCompilation: yes`, `LinkingTo: Rcpp`.
- `useDynLib(topologyR, .registration = TRUE)` in `NAMESPACE`.
- `#ifdef _OPENMP` guards for all OpenMP includes.
- C++17 standard specified in `Makevars`.
- Input validation on all exported R functions.
- Examples in all `.Rd` files.
- MIT license.

6 COMPLETE FUNCTION REFERENCE

6.1 Graph Construction Functions

6.1.1 `horizontal_visibility_graph(series, directed = FALSE)`

Constructs the Horizontal Visibility Graph from a time series.

Parameters:

- **series:** Numeric vector representing the time series. Must not contain NA, Inf, or NaN values.
- **directed:** Logical (default FALSE). If TRUE, additionally returns directed adjacency lists for bitopological analysis.

Returns: A list with:

- **edges:** A `data.frame` with columns `from` and `to` (integer, 1-based) representing undirected edges.
- **n:** Integer. Number of observations.
- **n_edges:** Integer. Number of edges.
- **adjacency:** A list of integer vectors, where element `i` contains the 1-based indices of all neighbors of node `i`. This is the neighborhood structure used as input to `generate_topology()`.
- **out_adjacency:** (*Only if `directed = TRUE`.*) A list of integer vectors, where element `i` contains the indices of all forward (later-in-time) neighbors of node `i`. Used as input for the forward Nada topology τ^+ or the Alexandrov topology.
- **in_adjacency:** (*Only if `directed = TRUE`.*) A list of integer vectors, where element `i` contains the indices of all backward (earlier-in-time) neighbors of node `i`. Used as input for the backward Nada topology τ^- .

Example:

```
library(topologyR)
series <- c(3, 1, 4, 1, 5, 9, 2, 6)
g <- horizontal_visibility_graph(series)
g$n_edges
# [1] 10
head(g$edges)
#   from to
# 1     1 2
# 2     2 3
# ...

# Directed mode for bitopological analysis
gd <- horizontal_visibility_graph(series, directed = TRUE)
gd$out_adjacency[[1]] # forward neighbors of observation 1
gd$in_adjacency[[5]]  # backward neighbors of observation 5
```

Input validation:

- Non-numeric input: "'series' must be a numeric vector."
- Contains NA: "'series' must not contain NA values."
- Contains Inf/NaN: "'series' must not contain Inf or NaN values."
- Length < 2: Returns a valid but empty graph (0 edges). If `directed = TRUE`, also returns empty `out_adjacency` and `in_adjacency`.

6.1.2 natural_visibility_graph(series, directed = FALSE)

Constructs the Natural Visibility Graph from a time series.

Parameters:

- **series:** Numeric vector representing the time series. Same validation rules as `horizontal_visibility_graph()`.
- **directed:** Logical (default FALSE). Same behavior as in `horizontal_visibility_graph()`.

Returns: Same structure as `horizontal_visibility_graph()` (including `out_adjacency`/`in_adjacency` when `directed = TRUE`).

Example:

```
g_nvng <- natural_visibility_graph(series)
g_nvng$n_edges >= g$n_edges
# [1] TRUE (NVG always has at least as many edges as HVG)
```

Relationship to HVG: Every HVG edge is also an NVG edge ($E_{\text{HVG}} \subseteq E_{\text{NVG}}$). This can be verified:

```
hvg <- horizontal_visibility_graph(series)
nvng <- natural_visibility_graph(series)
for (k in seq_len(nrow(hvg$edges))) {
  stopifnot(hvg$edges$to[k] %in% nvng$adjacency[[hvg$edges$from[k]]])
}
```

6.2 Topology Generation

6.2.1 generate_topology(adjacency, n_elements, max_open_sets, max_base_sets, check_connected, verify_axioms)

The central function of the package. Generates the topology induced by a graph following Nada et al. (2018).

Parameters:

Parameter	Type	Default	Description
adjacency	list of integer vectors	(required)	Adjacency list. Element <code>i</code> contains 1-based indices of neighbors of node <code>i</code> . Typically from <code>horizontal_visibility_graph()</code> \$adjacency or <code>natural_visibility_graph()</code> \$adjacency.
n_elements	integer	(required)	Total number of elements in V . Must equal <code>length(adjacency)</code> .
max_open_sets	integer	1,000,000	Maximum open sets to enumerate. Set to 0 to skip enumeration.
max_base_sets	integer	100,000	Maximum base elements during intersection closure.
check_connected	logical	TRUE	Whether to compute exact connectivity via specialization preorder.

Parameter	Type	Default	Description
<code>verify_axioms</code>	logical	<code>FALSE</code>	Whether to verify topology axioms after enumeration. Only meaningful when <code>complete = TRUE</code> .

Returns: A list with:

Field	Type	Description
<code>subbase</code>	list of integer vectors	The closed neighborhoods $N[v]$ used as subbase.
<code>base</code>	list of integer vectors	The base (closure of subbase under finite intersections).
<code>base_complete</code>	logical	<code>TRUE</code> if intersection closure finished within <code>max_base_sets</code> .
<code>connected</code>	logical	<code>TRUE</code> if the space is topologically connected. <code>NA</code> if <code>check_connected = FALSE</code> . This is an exact result based on the specialization preorder.
<code>components</code>	list of integer vectors	The exact topological connected components. Each vector contains 1-based element indices.
<code>topology</code>	list of integer vectors or <code>NULL</code>	The open sets, if enumeration was requested and feasible.
<code>n_open_sets</code>	integer or <code>NA</code>	Number of open sets if enumerated.
<code>complete</code>	logical	<code>TRUE</code> if enumeration finished within <code>max_open_sets</code> .
<code>iterations_base</code>	integer	Number of wavefront iterations for base closure.
<code>iterations_topo</code>	integer	Number of wavefront iterations for union closure.
<code>axioms_ok</code>	logical	Only present if <code>verify_axioms = TRUE</code> and <code>complete = TRUE</code> .
<code>axiom_failure</code>	character	Only present if axiom verification fails. Diagnostic message.

Example — standard workflow:

```
series <- c(3, 1, 4, 1, 5, 9, 2, 6)
nvg <- natural_visibility_graph(series)
topo <- generate_topology(
  adjacency      = nvg$adjacency,
  n_elements     = nvg$n,
  max_open_sets  = 0L,           # skip enumeration, connectivity only
  check_connected = TRUE
)
topo$connected
# [1] TRUE
length(topo$components)
# [1] 1
length(topo$base)
# [1] 15
```

Example — full enumeration with axiom verification:

```

topo_full <- generate_topology(
  adjacency      = nv$adjacency,
  n_elements     = nv$n,
  verify_axioms = TRUE
)
topo_full$complete
# [1] TRUE
topo_full$n_open_sets
# [1] 42
topo_full$axioms_ok
# [1] TRUE

```

Example — large n , connectivity only:

```

# For n = 129 (real economic data), skip enumeration
hvg <- horizontal_visibility_graph(GDP_change)
topo <- generate_topology(
  adjacency      = hvg$adjacency,
  n_elements     = hvg$n,
  max_open_sets  = 0L,
  check_connected = TRUE
)
cat("Components:", length(topo$components), "\n")
# Components: 36

```

Input validation:

- adjacency not a list: "'adjacency' must be a list of integer vectors."
- n_elements < 1: "'n_elements' must be >= 1."
- length(adjacency) != n_elements: "Length of 'adjacency' must equal 'n_elements'."

Note on $N[v]$: The engine automatically converts the open neighborhoods in `adjacency` to closed neighborhoods by adding each vertex to its own neighborhood. The user provides the standard adjacency list (without self-loops); the conversion is internal.

6.2.2 generate_alexandrov_topology(out_adjacency, n_elements, max_open_sets, check_connected, verify_axioms)

Computes the Alexandrov topology induced by a directed acyclic graph (DAG) via bitset-based reachability propagation. The open sets are the upsets of the reachability relation.

Parameters:

Parameter	Type	Default	Description
out_adjacency	list of integer vectors	(required)	Forward adjacency list from a directed visibility graph. Element i contains 1-based indices of forward neighbors of node i . Must represent a DAG with all edges from lower to higher index.

Parameter	Type	Default	Description
<code>n_elements</code>	integer	(required)	Total number of elements. Must equal <code>length(out_adjacency)</code> .
<code>max_open_sets</code>	integer	0	Maximum open sets to enumerate. 0 = skip enumeration.
<code>check_connected</code>	logical	TRUE	Whether to compute exact connectivity.
<code>verify_axioms</code>	logical	FALSE	Whether to verify topology axioms after enumeration.

Returns: A list with the same structure as `generate_topology()`:

Field	Type	Description
<code>subbase</code>	list of integer vectors	The reachability upsets (= base for Alexandrov).
<code>base</code>	list of integer vectors	Same as subbase (upsets are already closed under intersection).
<code>base_complete</code>	logical	Always TRUE (no iteration needed).
<code>connected</code>	logical	Exact topological connectivity.
<code>components</code>	list of integer vectors	Exact connected components.
<code>topology</code>	list of integer vectors or NULL	Open sets if enumerated.
<code>n_open_sets</code>	integer or NA	Number of open sets if enumerated.
<code>complete</code>	logical	Whether enumeration finished.

Example:

```
series <- c(3, 1, 4, 1, 5)
g <- horizontal_visibility_graph(series, directed = TRUE)
alex <- generate_alexandrov_topology(g$out_adjacency, g$n)
alex$connected
# [1] TRUE
length(alex$base)
# [1] 5

# Compare with Nada topology
nada <- generate_topology(g$out_adjacency, g$n, max_open_sets = 0L)
cat("Nada base:", length(nada$base), " Alexandrov base:", length(alex$base), "\n")
# Nada base always >= Alexandrov base
```

Input validation:

- `out_adjacency` not a list: "'out_adjacency' must be a list of integer vectors."
- `n_elements < 1`: "'n_elements' must be >= 1."
- `length(out_adjacency) != n_elements`: "Length of 'out_adjacency' must equal 'n_elements'."

6.2.3 `generate_bitopology(series, graph_type, max_open_sets, max_base_sets, alexandrov)`

Performs a complete bitopological analysis of a time series. This is the high-level pipeline function that builds a directed visibility graph and computes four topological spaces: undirected Nada τ , forward Nada τ^+ , backward Nada τ^- , and Alexandrov τ_A .

Parameters:

Parameter	Type	Default	Description
<code>series</code>	numeric vector	(required)	Time series. Must not contain NA, Inf, or NaN. Must have ≥ 2 observations.
<code>graph_type</code>	character	"hvg"	Either "hvg" (Horizontal Visibility Graph) or "nvg" (Natural Visibility Graph).
<code>max_open_sets</code>	integer	0	Maximum open sets to enumerate per topology. Required for pairwise connectedness check.
<code>max_base_sets</code>	integer	100,000	Maximum base elements during Nada intersection closure.
<code>alexandrov</code>	logical	TRUE	Whether to also compute the Alexandrov topology for resolution comparison.

Returns: A list with:

Field	Type	Description
<code>graph</code>	list	The directed visibility graph (with <code>adjacency</code> , <code>out_adjacency</code> , <code>in_adjacency</code>).
<code>undirected</code>	list	Undirected Nada topology τ (output of <code>generate_topology()</code>).
<code>forward</code>	list	Forward Nada topology τ^+ .
<code>backward</code>	list	Backward Nada topology τ^- .
<code>alexandrov</code>	list or NULL	Alexandrov topology τ_A (if <code>alexandrov = TRUE</code>).
<code>invariants</code>	list	Bitopological invariants (output of <code>bitopology_invariants()</code>).

Example — standard bitopological analysis:

```
series <- c(3, 1, 4, 1, 5, 9, 2, 6)
bt <- generate_bitopology(series, graph_type = "hvg")

# Irreversibility diagnostics
inv <- bt$invariants
cat("Forward components:", inv$forward_components, "\n")
cat("Backward components:", inv$backward_components, "\n")
cat("Irreversibility (components):", inv$irreversibility_components, "\n")
cat("Asymmetry direction:", inv$asymmetry_direction, "\n")

# Resolution hierarchy
cat("Alexandrov base:", inv$resolution$alexandrov_base_size, "\n")
cat("Nada fwd base gain:", inv$resolution$nada_forward_base_gain, "\n")
```

Example — with pairwise connectedness (small series):

```
bt_enum <- generate_bitopology(c(3, 1, 4), max_open_sets = 1000L)
pw <- bt_enum$invariants$pairwise
if (!is.na(pw$pairwise_connected)) {
  cat("Pairwise connected:", pw$pairwise_connected, "\n")
}
```

Input validation:

- Non-numeric series: "'series' must be a numeric vector."
- Contains NA: "'series' must not contain NA values."
- Contains Inf/NaN: "'series' must not contain Inf or NaN values."
- Length < 2: "'series' must have at least 2 observations."

6.2.4 bitopology_invariants(forward, backward, n_elements, undirected, alexandrov)

Computes bitopological invariants from pre-computed forward and backward topologies.

Parameters:

Parameter	Type	Default	Description
forward	list	(required)	Forward Nada topology τ^+ (output of <code>generate_topology()</code>).
backward	list	(required)	Backward Nada topology τ^- (output of <code>generate_topology()</code>).
n_elements	integer	(required)	Number of elements in V .
undirected	list or NULL	NULL	Undirected topology (optional, for comparison).
alexandrov	list or NULL	NULL	Alexandrov topology (optional, for resolution comparison).

Returns: A list with:

Field	Type	Description
forward_components	integer	Number of components in τ^+ .
backward_components	integer	Number of components in τ^- .
forward_connected	logical	Whether τ^+ is connected.
backward_connected	logical	Whether τ^- is connected.
forward_base_size	integer	Base elements in τ^+ .
backward_base_size	integer	Base elements in τ^- .
irreversibility_components	numeric	$ C^+ - C^- / \max(C^+, C^-)$, in $[0, 1]$.
irreversibility_base	numeric	$ B^+ - B^- / \max(B^+, B^-)$, in $[0, 1]$.
asymmetry_direction	integer	$C^- - C^+$. Positive = forward more connected.
pairwise	list	<code>pairwise_connected</code> (logical or NA), <code>clopen_forward</code> , <code>clopen_backward</code> .
resolution	list or NULL	Alexandrov vs Nada comparison (if <code>alexandrov</code> provided). Contains <code>alexandrov_base_size</code> , <code>alexandrov_components</code> , <code>nada_forward_base_gain</code> , <code>nada_backward_base_gain</code> , and relative gains.

Field	Type	Description
undirected_info	list or NULL	Undirected topology summary (if <code>undirected</code> provided).

Example:

```
g <- horizontal_visibility_graph(c(3, 1, 4, 1, 5, 9, 2, 6), directed = TRUE)
tf <- generate_topology(g$out_adjacency, g$n, max_open_sets = 0L)
tb <- generate_topology(g$in_adjacency, g$n, max_open_sets = 0L)
inv <- bitopology_invariants(tf, tb, g$n)
cat("Asymmetry direction:", inv$asymmetry_direction, "\n")
# Positive = forward more connected (consistent with gradual expansion dynamics)
```

6.3 Connectivity Functions

6.3.1 `is_topology_connected_exact(topology, n_elements)`

Checks topological connectivity using the exact clopen definition: searches for a proper non-empty open set whose complement is also open.

Parameters:

- `topology`: A list of integer vectors representing the open sets.
- `n_elements`: Integer. Total number of elements in V .

Returns: A list with:

- `connected`: Logical. TRUE if connected.
- `component1`: Integer vector (only if disconnected). One side of the clopen partition.
- `component2`: Integer vector (only if disconnected). The other side.

Requires: The full topology must be provided. If the topology was not fully enumerated (i.e., `generate_topology()` returned `complete = FALSE`), this function cannot be used — use the preorder-based result from `generate_topology()` instead.

Example:

```
# Connected topology
tau <- list(integer(0), 1:3, c(1L, 2L), c(2L, 3L))
is_topology_connected_exact(tau, 3L)
# $connected
# [1] TRUE

# Disconnected topology
tau2 <- list(integer(0), 1:4, c(1L, 2L), c(3L, 4L))
result <- is_topology_connected_exact(tau2, 4L)
result$connected
# [1] FALSE
result$component1
# [1] 1 2
result$component2
# [1] 3 4
```

6.3.2 `is_topology_connected(topology)` — Legacy

Undirected graph DFS heuristic. **Approximation only.** Constructs an undirected graph where elements sharing an open set are connected, then checks graph connectivity. Returns `TRUE` if the graph is connected.

This is a necessary but not sufficient condition for topological connectivity.

6.3.3 `is_topology_connected2(topology)` — Legacy

Directed graph DFS heuristic. **Approximation only.** Creates directed edges between consecutive sorted elements in each open set, checks reachability from the minimum element.

This is a necessary but not sufficient condition for topological connectivity.

6.3.4 `is_topology_connected_manual(topology)` — Legacy

Coverage check heuristic. **Approximation only.** Checks whether all elements from 1 to `max(V)` appear in at least one open set. This verifies $\bigcup \tau = X$, which is always true for any topology and says nothing about connectivity.

6.4 Threshold-Based Functions (Metric Approximation)

These functions construct topological neighborhoods using metric proximity ($|x_i - x_j| \leq \epsilon$) rather than graph-theoretic visibility. They are retained for comparison with the Nada et al. (2018) approach but are **not** the recommended path for topology construction.

The fundamental limitation of threshold-based methods is that the resulting topology depends on the arbitrary choice of ϵ . Different threshold methods can produce dramatically different results on the same data.

6.4.1 `calculate_thresholds(data)`

Computes five different threshold methods for defining neighborhoods.

Parameters:

- **data:** Numeric vector with at least 2 elements, no NA values.

Returns: A named list with:

Method	Formula	Description
<code>mean_diff</code>	Mean of $ y_{(i+1)} - y_{(i)} $	Mean gap between sorted adjacent values
<code>median_diff</code>	Median of $ y_{(i+1)} - y_{(i)} $	Median gap between sorted adjacent values
<code>sd</code>	$\text{sd}(y)$	Standard deviation of the data
<code>iqr</code>	$\text{IQR}(y)/4$	Interquartile range divided by 4
<code>dbscan</code>	k -th nearest neighbor distance	$k = \lceil \log(n) \rceil$

Example:

```
th <- calculate_thresholds(rnorm(100))
th$sd
# [1] 0.982 (approximately)
```

6.4.2 calculate_topology(data, threshold)

Computes the size of the topological base for a given threshold.

Parameters:

- **data:** Numeric vector with at least 2 elements.
- **threshold:** Non-negative numeric scalar.

Returns: Integer. The number of sets in the base.

The base is constructed by: (1) forming neighborhoods $N_\epsilon(x_i) = \{j : |x_i - x_j| \leq \epsilon\}$, (2) computing all pairwise intersections, (3) adding $\emptyset$ and V .

Example:

```
calculate_topology(rnorm(30), threshold = 0.5)
# [1] 147 (varies with random seed)
```

6.4.3 visualize_topology_thresholds(data, plot = TRUE)

Computes all five threshold methods and their corresponding base sizes, with optional visualization.

Parameters:

- **data:** Numeric vector.
- **plot:** Logical (default TRUE). Whether to generate **ggplot2** plots.

Returns: A **data.frame** with columns **method**, **threshold**, **base_size**. If **plot = TRUE**, carries an attribute "plots" containing a list of three **ggplot** objects:

- **threshold:** Bar chart comparing threshold values by method.
- **base_size:** Bar chart comparing base sizes by method.
- **scatter:** Scatter plot of threshold vs. base size.

Example:

```
results <- visualize_topology_thresholds(rnorm(50))
print(results)
#      method threshold base_size
# 1 mean_diff  0.042530      312
# 2 median_diff 0.019284      187
# ...

# Access plots:
plots <- attr(results, "plots")
print(plots$scatter)
```

6.4.4 `analyze_topology_factors(data, factors = NULL, plot = TRUE)`

Analyzes how different IQR divisor factors affect topology characteristics.

Parameters:

- **data:** Numeric vector with at least 2 elements, no NA values.
- **factors:** Numeric vector of factors to test (default: `c(1, 2, 4, 8, 16)`). The threshold used is $\text{IQR}(y)/f$ for each factor f .
- **plot:** Logical (default `TRUE`). Whether to generate a `ggplot2` plot.

Returns: A `data.frame` with columns:

- **factor:** The IQR divisor factor.
- **threshold:** The calculated threshold (IQR/f).
- **base_size:** Number of sets in the base.
- **max_set_size:** Size of the largest set in the base.
- **min_set_size:** Size of the smallest set in the base.

If `plot = TRUE`, carries an attribute "plot" with a `ggplot` object showing the factor-base size relationship on a log scale.

Example:

```
af <- analyze_topology_factors(rnorm(50), factors = c(1, 2, 4, 8, 16, 32))
print(af)
#   factor threshold base_size max_set_size min_set_size
# 1      1  1.3425000      102          50           0
# 2      2  0.6712500      287          50           0
# ...
```

6.5 Degenerate and Special Topologies

6.5.1 `simplest_topology(data)`

Generates the discrete topology on n elements. In the discrete topology, every subset of V is open. The base consists of all singletons $\{\{1\}, \{2\}, \dots, \{n\}\}$.

Parameters:

- **data:** Numeric vector with at least 1 element.

Returns: A list with:

- **subbase:** List of singleton sets.
- **base:** Same as subbase (singletons are already a base for the discrete topology).
- **topology_type:** Character: "discrete".
- **n:** Integer. Number of elements.
- **connected:** Logical. Always `FALSE` for $n \geq 2$ (the discrete topology is maximally disconnected: every singleton is clopen).

Example:

```
st <- simplest_topology(c(10, 20, 30, 40, 50))
st$topology_type
# [1] "discrete"
st$connected
# [1] FALSE
st$n
# [1] 5
```

The discrete topology is the finest possible topology: it has the maximum number of open sets (2^n) and the maximum number of connected components (n). It represents the case where every point is topologically isolated from every other point.

6.5.2 complete_topology(data, verify_axioms = FALSE)

Constructs the topology induced by the complete graph K_n on the data points. In the complete graph, every vertex is a neighbor of every other vertex, so each closed neighborhood is $N[v] = V$ (the entire vertex set). This produces the **indiscrete topology** $\tau = \{\emptyset, V\}$, which is the coarsest possible topology and is always connected.

Parameters:

- **data**: Numeric vector with at least 2 elements, no NA values.
- **verify_axioms**: Logical (default FALSE). Whether to verify topology axioms.

Returns: Same structure as `generate_topology()`.

Example:

```
ct <- complete_topology(c(1, 2, 3, 4, 5))
ct$n_open_sets
# [1] 2      (just empty set and V)
ct$connected
# [1] TRUE
```

Note: With closed neighborhoods $N[v] = V$, the complete graph produces a single subbase element (since all neighborhoods are identical), a trivial base, and the indiscrete topology. This is the theoretical minimum: 2 open sets, 1 connected component.

7 WORKFLOW AND USAGE PATTERNS

7.1 Standard Workflow

The recommended workflow for topological analysis of a time series is:

```
library(topologyR)

# 1. Construct a visibility graph
hvg <- horizontal_visibility_graph(series)
# or: nvg <- natural_visibility_graph(series)

# 2. Generate the topology (connectivity only, no enumeration)
```

```

topo <- generate_topology(
  adjacency      = hvg$adjacency,
  n_elements     = hvg$n,
  max_open_sets  = OL,
  check_connected = TRUE
)

# 3. Inspect results
cat("Connected:", topo$connected, "\n")
cat("Components:", length(topo$components), "\n")
cat("Base size:", length(topo$base), "\n")
cat("Base complete:", topo$base_complete, "\n")

# 4. Examine component structure
for (ci in seq_along(topo$components)) {
  idx <- sort(topo$components[[ci]])
  cat("Component", ci, "(n =", length(idx), "):",
      paste(range(idx), collapse = "-"), "\n")
}

```

7.2 Choosing Between HVG and NVG

Criterion	Use HVG	Use NVG
Speed	$O(n)$ — always fast	$O(n \log n)$ — fast for most data
Resolution	Finer (more components)	Coarser (fewer components)
Edge range	Local only	Local + medium-range
Sensitivity	Local regime changes	Global dynamical structure
Theory	Luque et al. (2009)	Lacasa et al. (2008)

Recommendation. Run both and compare. If HVG gives many small components but NVG gives a few large ones, the difference reveals which connections are local-only vs. which extend over longer time ranges.

7.3 Handling Large n

For large time series ($n > 50$), the full topology is typically too large to enumerate. The recommended approach is:

```

topo <- generate_topology(
  adjacency      = graph$adjacency,
  n_elements     = graph$n,
  max_open_sets  = OL,          # skip enumeration entirely
  check_connected = TRUE
)

```

This computes exact connectivity from the base alone (via the specialization preorder) without attempting to enumerate the exponentially large topology.

If axiom verification is desired for validation purposes, it can be run on a small subset:

```

# Verify axioms on first 10 observations
graph_small <- horizontal_visibility_graph(series[1:10])
topo_small <- generate_topology(
  adjacency = graph_small$adjacency,
  n_elements = graph_small$n,
  verify_axioms = TRUE
)
stopifnot(topo_small$axioms_ok)

```

7.4 Comparing Graph-Theoretic vs. Threshold-Based Results

The package supports both the Nada et al. (2018) graph-theoretic approach and legacy threshold-based methods, allowing direct comparison:

```

# Graph-theoretic path (parameter-free)
nvg <- natural_visibility_graph(series)
topo_graph <- generate_topology(nvg$adjacency, nvg$n, max_open_sets = 0L)
cat("Graph-theoretic: connected =", topo_graph$connected,
    "| components =", length(topo_graph$components), "\n")

# Threshold-based path (parameter-dependent)
thresholds <- calculate_thresholds(series)
for (nm in names(thresholds)) {
  bs <- calculate_topology(series, threshold = thresholds[[nm]])
  cat(sprintf(" %-12s threshold = %.4f -> base size = %d\n",
              nm, thresholds[[nm]], bs))
}

```

The graph-theoretic result is unique and reproducible. The threshold-based results vary with the choice of method, illustrating precisely why the parameter-free approach is preferred.

7.5 Bitopological Analysis Workflow (v0.2.0)

The bitopological workflow extends the standard workflow by adding directed topology construction and irreversibility analysis:

```

library(topologyR)

# --- One-call pipeline ---
bt <- generate_bitopology(series, graph_type = "hvg")

# --- Inspect irreversibility ---
inv <- bt$invariants
cat("Forward components (+):", inv$forward_components, "\n")
cat("Backward components (-):", inv$backward_components, "\n")
cat("Irreversibility (components):", inv$irreversibility_components, "\n")
cat("Irreversibility (base):", inv$irreversibility_base, "\n")
cat("Asymmetry direction:", inv$asymmetry_direction, "\n")
# Positive asymmetry_direction = + more connected (fewer components)

# --- Interpretation ---

```

```

if (inv$asymmetry_direction > 0) {
  cat("Forward visibility less obstructed → consistent with gradual expansions\n")
} else if (inv$asymmetry_direction < 0) {
  cat("Backward visibility less obstructed → consistent with gradual contractions\n")
} else {
  cat("Symmetric dynamics (equal component counts)\n")
}

# --- Resolution hierarchy (Nada vs Alexandrov) ---
if (!is.null(inv$resolution)) {
  res <- inv$resolution
  cat("Alexandrov base:", res$alexandrov_base_size, "\n")
  cat("Nada forward base gain:", res$nada_forward_base_gain,
      "(+", round(res$nada_forward_relative_gain * 100), "%)\n")
  cat("Nada backward base gain:", res$nada_backward_base_gain,
      "(+", round(res$nada_backward_relative_gain * 100), "%)\n")
}

# --- Compare undirected vs directed ---
cat("\nUndirected topology:", inv$undirected_info$undirected_components, "components\n")
cat("Forward topology:", inv$forward_components, "components\n")
cat("Backward topology:", inv$backward_components, "components\n")

```

7.5.1 Step-by-Step Bitopological Analysis

For finer control, the analysis can be performed step by step:

```

# 1. Build directed visibility graph
g <- horizontal_visibility_graph(series, directed = TRUE)

# 2. Compute forward Nada topology +
tau_fwd <- generate_topology(g$out_adjacency, g$n, max_open_sets = 0L)

# 3. Compute backward Nada topology -
tau_bwd <- generate_topology(g$in_adjacency, g$n, max_open_sets = 0L)

# 4. Compute Alexandrov topology _A
tau_alex <- generate_alexandrov_topology(g$out_adjacency, g$n)

# 5. Compute undirected topology (for comparison)
tau_und <- generate_topology(g$adjacency, g$n, max_open_sets = 0L)

# 6. Compute invariants
inv <- bitopology_invariants(tau_fwd, tau_bwd, g$n, tau_und, tau_alex)

```

7.5.2 Pairwise Connectedness (Small Series Only)

Pairwise connectedness (Kelly, 1963) requires full enumeration of both τ^+ and τ^- . This is only feasible for small n (typically $n \leq 20$):

```

# For a small subseries, enumerate all open sets
bt_small <- generate_bitopology(series[1:15], max_open_sets = 100000L)
pw <- bt_small$invariants$pairwise

if (!is.na(pw$pairwise_connected)) {
  if (pw$pairwise_connected) {
    cat("Pairwise connected: no single set is simultaneously +-open and --closed\n")
  } else {
    cat("Pairwise disconnected!\n")
    cat("  Clopen set ( +-open, --closed):", pw$clopen_forward, "\n")
    cat("  Its complement ( --open):", pw$clopen_backward, "\n")
  }
} else {
  cat("Pairwise connectedness: could not compute (enumeration incomplete)\n")
}

```

7.5.3 Comparing HVG and NVG Bitopologies

Since the NVG is a superset of the HVG, the two graph types produce different bitopological structures. Comparing them reveals whether irreversibility patterns are robust:

```

bt_hvg <- generate_bitopology(series, graph_type = "hvg")
bt_nvg <- generate_bitopology(series, graph_type = "nvg")

cat("HVG irreversibility:", bt_hvg$invariants$irreversibility_components, "\n")
cat("NVG irreversibility:", bt_nvg$invariants$irreversibility_components, "\n")
cat("HVG asymmetry direction:", bt_hvg$invariants$asymmetry_direction, "\n")
cat("NVG asymmetry direction:", bt_nvg$invariants$asymmetry_direction, "\n")
# If both agree on the sign, the irreversibility finding is robust

```

8 REFERENCES

- Alexandrov, P. (1937). Diskrete Räume. *Matematicheskii Sbornik*, 2(44), 501–519.
- El Atik, A. E. F., Nasef, A. A., & Azzam, A. A. (2020). Rough approximation of a topological space using graphs. *International Journal of Biomathematics*, 13(3), 2050025.
- Kelly, J. C. (1963). Bitopological spaces. *Proceedings of the London Mathematical Society*, 3(1), 71–89.
- Kelley, J. L. (1955). *Topología General*. Editorial Universitaria de Buenos Aires.
- Kolmogórov, A. N., & Fomin, S. V. (1978). *Elementos de la Teoría de Funciones y del Análisis Funcional*. Editorial MIR.
- Lacasa, L., & Toral, R. (2010). Description of stochastic and chaotic series using visibility graphs. *Physical Review E*, 82(3), 036120.
- Lacasa, L., Luque, B., Ballesteros, F., Luque, J., & Nuno, J. C. (2008). From time series to complex networks: The visibility graph. *Proceedings of the National Academy of Sciences*, 105(13), 4972–4975.
- Luque, B., Lacasa, L., Ballesteros, F., & Luque, J. (2009). Horizontal visibility graphs: Exact results for random time series. *Physical Review E*, 80(4), 046103.
- McCord, M. C. (1966). Singular homology groups and homotopy groups of finite topological spaces. *Duke Mathematical Journal*, 33(3), 465–474.

- Nada, S., El Atik, A. E. F., & Atef, M. (2018). New types of topological structures via graphs. *Mathematical Methods in the Applied Sciences*, 41(15), 5801-5810.
- Stong, R. E. (1966). Finite topological spaces. *Transactions of the American Mathematical Society*, 123(2), 325-340.
- Trudeau, R. J. (1993). *Introduction to Graph Theory*. Dover Publications.