# Supplementary Materials

David Novák[1], Naděžda Brdičková[1], Jan Stuchlý[1], and Tomáš Kalina[1]

[1,2,3]Department of Paediatric Haematology and Oncology,

2nd Faculty of Medicine, Charles University. Prague, Czech Republic.

May 2, 2021

# 1 Theoretical background

`tviblindi` uses computational tools for analysis of high-dimensional data to describe tentative developmental pathways. Our analysis of cytometry, scRNA-seq or CITE-seq ADT datasets can be formulated using a handful of concepts from graph theory and topological data analysis (TDA). We will use *ad hoc* definitions of useful concepts throughout, allowing us to explain the inner workings of `tviblindi` without being too elaborate.

## 1.1 *k*-nearest-neighbour graph construction

Our main input is a coordinate matrix of data points corresponding to single cells; together they form a point cloud. Consider this point cloud $X$ composed of $n$ points. We construct a $k$-NN graph where $X$ is taken as the vertex set $V$.

**Definition 1** (Graph)**.** *We define a graph as an ordered pair $G = (V, E)$ where $V$ is a set of vertices (nodes) $v_1, v_2, ..., v_N$ and $E \subseteq \{\{x, y\} | (x, y) \in V^2 \wedge x \neq y\}$ a set of edges, i.e. pairs of vertices.*

**Definition 2** ($k$-th nearest neighbour)**.** *Let $V$ be a set of $m$ vertices. Let $D$ be a matrix of distances: $\forall i, j \in [1, m] : D_{ij} = d(v_i, v_j)$ for some distance function $d$. For any $v_i, v_j \in V$ and a positive integer $k$, we set $\boldsymbol{d} := D_{i\_}$ and $\boldsymbol{d_i} := \infty$. We say that $v_j$ is the $k$-th nearest neighbour of $v_i$ iff $\boldsymbol{d_j}$ is the $k$-th smallest value in $\boldsymbol{d}$.*

**Definition 3** (Weighted graph)**.** *For a weighted graph $G_w = (V, E, \omega)$, we define a weight function $\omega : E \to \mathbb{R}_0^+$ which maps edges to their weights. Weighted graph is a type of graph as defined earlier.*

**Definition 4** ($k$-nearest neighbour graph)**.** *Let $G = (V, E, \omega)$ be a weighted graph. For a positive integer $k$, $E$ are edges connecting each $v \in V$ to its $k$ nearest neighbours, using a distance metric of choice. Then, we call $G$ a $k$-nearest neighbour ($k$-NN) graph.*

Typically, we set the lower bound for $k$ so as to make our graph connected. Alternatively, we can work with the single connected component that contains a preset cell-of-origin (the one in which tentative developmental trajectories will begin). We call our $k$-NN graph $G_{kNN}$.

## 1.2 Transition model

We set edge weights in $G_{kNN}$ to be the distances between each pair of connected vertices. The next step is to construct a transition graph $G_t$, representing what we call a *transition model*: a model of probabilities of transitioning between vertices that are connected in $G_{kNN}$. The weight of any edge $\omega(\{v_i, v_j\})$ will be indirectly proportional to the distance between them.

In the construction of $G_t$, `tviblindi` uses matrices to represent edge weights. Immediately, we get a *distance matrix* $S$ from $G_{kNN}$, written as

$$S_{ij} = \begin{cases} d(v_i, vj) & v_i \text{ and } v_j \text{ connected by an edge} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $d$ is the Euclidean distance function.

We symmetrise $S$ by setting each entry $S_{ij} := max\{S_{ij}, S_{ji}\}$ (the transition model will ultimately only allow for transitions in a single direction anyway). We calculate a matrix $T$ of edge weights of $G_T$ as

$$T_{ij} = \begin{cases} f(S_{ij}) & v_i \text{ and } v_j \text{ connected by an edge} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $f$ is a function approximating probability density for transition from any vertex to one of its $k$ nearest neighbours.

For default use, we define an exponential kernel

$$f(x) = e^{-x/\varepsilon} \tag{3}$$

where $\varepsilon = 2 \times median\{\boldsymbol{d}\}^2$, with $\boldsymbol{d}$ containing all non-zero elements of $S$. This method of computing $\varepsilon$ is a heuristic adopted from the `R` package `diffusionMap` (Richards and Cannoodt, 2020).

Normalising each row of $T$ to sum to 1 would cause any given row to contain an estimate of a discrete probability distribution of transitions (this will be used later). $T$ now defines our $G_t$, whereby each $i$-th row corresponds to $v_i$, each $j$-th column to $v_j$ and the entries give $\omega(\{v_i, v_j\})$.

## 1.3 Pseudotime inference

Edge weights in $G_t$ now determine the likelihood of jumping from any node $v_i$ to a particular neighbouring node $v_j$. Our goal is to use the graph to represent some developmental series which flows strictly forward in developmental time. We describe a single simulated graph traversal as a *random walk*.

**Definition 5** (Random walk)**.** *Let $G = (V, E, \omega)$ be a weighted graph. We define a random walk of length $n$ as a series $v_0, v_1, ..., v_n$ of nodes in $G$. Each $v_j$ is chosen from $\{v_i : \{v_{i-1}, v_i\} \in E\}$ with probability $P_{v_i} = \frac{\omega(\{v_{i-1}, v_i\})}{\sum_{z:\{v_{i-1},z\}\in E} \omega(\{v_{i-1},z\})}$.*

To orient these walks, we introduce an artificial time quantity *pseudotime* to associate each node with a value that shows how far along a developmental series each node is located.

**Definition 6** (Pseudotime)**.** *Let $G$ be a graph. We define a pseudotime function $\tau_{v_0} : V \to \mathbb{R}_0^+$ for each node in vertex set $V$ of $G$. Pseudotime (value) $\tau_{v_0}(w)$ is computed as the expected length of a random walk from $v_0$ until $w$ is reached. If we suppose that $v_0$ is fixed, we simply write $\tau$ to mean $\tau_{v_0}$.*

We will also refer to $v_0$ as the *cell-of-origin* (and require it as a known prior). Intuitively, $\tau_{v_0}(v_0) = 0$. Now, suppose that pseudotime for all neighbours of some $y$ was computed already. Then, the pseudotime of $y$ is calculated as the weighted average of its neighbors' pseudotime values increased by their distances. Formally, we write this 'one extra step' rule as

$$\tau_{v_0}(y) = \sum_{x:\{x,y\}\in E} p_{x,y}(\tau_{v_0}(x) + S_{xy}) \text{ if } x \neq v_0 \tag{4}$$

Here, we take probabilities $p_{x,y}$ of arriving at $y$ from each vertex $x$ connected to it and multiply each by the pseudotime value up to that point increased by distance between $x$ and $y$.

Next, we will apply our 'one extra step' rule to devise an equation that lets us obtain a vector of pseudotime values $\boldsymbol{\tau}$. To capture both the connectedness and the distances between vertices in a graph, we will use the following working definition of a degree matrix.

**Definition 7** (Degree matrix)**.** *Let $G = (V, E, \omega)$ be a weighted graph with $m$ vertices and $T$ a transition matrix associated with it. We define the $n$-by-$n$ degree matrix $D$ of $G$ as*

$$D_{ij} = \begin{cases} \sum_{l=1}^{m} T_{il} & i = j \\ 0 & \textit{otherwise} \end{cases}$$

Note that $D$ is a matrix with row sums of $T$ on the diagonal and zeroes elsewhere. We now define the *graph Laplacian matrix* calculated using the information we have already obtained. (Graph Laplacian is a matrix representation of a graph, and constructing it will allow us to write down the 'one extra step' rule using matrices and solve for $\boldsymbol{p}$.)

**Definition 8** (Laplacian matrix)**.** *Let $G = (V, E, \omega)$ be a weighted graph, $T$ a transition matrix that is given by $T_{i,j} = \omega(\{v_i, v_j\})$ and $D$ the degree matrix of $G$. We define a simple graph Laplacian matrix of $G$ as $L_{sim} = D - T$ (symmetric by symmetricity of $T$). We define a symmetric normalised graph Laplacian matrix of $G$ as $L_{sym} = D^{-\frac{1}{2}} L_{sim} D^{-\frac{1}{2}}$. For short, we write 'Laplacian' to mean 'graph Laplacian*

*matrix'.*

We can think of the simple Laplacian as a matrix governing diffusion between nodes, based on the transition model. Positive values on the diagonal translate to inflow of some quantity to a given node, and the remaining values in the corresponding rows determine the distribution of sources of this flow from the neighbours. The symmetric normalised Laplacian is a symmetrised and normalised version such that all diagonal elements are 1.

It remains to re-write the 'one extra step' rule using matrix algebra. With our definition of random walk in mind, we write down the probabilities $p_{x,y}$ of arriving at any vertex $y$ from vertex $x$ using a matrix $P$ as

$$P = D^{-1}T \tag{5}$$

where the right side constitutes a normalised transition matrix, such that each row sums up to one (since each row vector is divided by the row sum). It follows that $D^{-1}T\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ is a column vector of ones. (Multiplying a matrix by $\mathbf{1}$ then yields a vector of its row sums.)

We now re-write the sum in Equation 4 using matrix multiplication. We do this by writing

$$\boldsymbol{\tau} = P\boldsymbol{\tau} + (P \odot S)\mathbf{1} = D^{-1}T\boldsymbol{\tau} + (D^{-1}T \odot S)\mathbf{1} \tag{6}$$

where $\odot$ denotes the Hadamard (element-wise) product. We subtract $D^{-1}T\boldsymbol{\tau}$ from

both sides and get

$$\boldsymbol{\tau} - D^{-1}T\boldsymbol{\tau} = (D^{-1}T \odot S)\mathbf{1} \tag{7}$$

$$(I - D^{-1}T)\boldsymbol{\tau} = (D^{-1}T \odot S)\mathbf{1} \tag{8}$$

where $I$ denotes an identity matrix. Finally this gives us

$$D^{-1}(D - T)\boldsymbol{\tau} = (D^{-1}T \odot S)\mathbf{1} \tag{9}$$

Note that $(D - T)$ is the graph Laplacian $L_{sim}$ defined earlier. For computational efficiency, we use $L_{sym} = D^{-\frac{1}{2}}L_{sim}D^{-\frac{1}{2}}$, as defined before, instead. We obtain

$$L_{sym}\boldsymbol{\tau'} = (D^{-\frac{1}{2}}T \odot S)\mathbf{1} \tag{10}$$

where we solve for $\boldsymbol{\tau'}$ and then compute the pseudotime values vector $\boldsymbol{\tau} = D^{-\frac{1}{2}}\boldsymbol{\tau'}$.

The linear system in Equation 10 is solved numerically using the conjugate gradient method (suitable for large sparse systems) or analytically for small number of vertices. Then, for each pair of connected vertices $v_i$ and $v_j$, edge $\{v_i, v_j\}$ only is kept if $p_{v_0}(v_i) < p_{v_0}(v_j)$, otherwise only $\{v_j, v_i\}$ is kept. (After this, any newly simulated random walks through $G_t$ will be acyclical.)

## 1.4 Simulating finite random walks

Next, we simulate independent random walks again, this time using the edges that were left in $G_t$ and such that each transition is to a vertex with a higher pseudotime value. Clearly, there are going to exist vertices from which we cannot transition elsewhere in this manner. We call each such vertex a *terminal node* (*terminus*).

Each random walk can now be simulated such that it terminates in one of these nodes. At the same time, the existence of a terminal node does not guarantee that any random walks will actually terminate there. Since simulations of random walks are stochastic, we expect the series of vertices which define each walk to differ between simulations.

## 1.5 Capturing the shape of a point cloud

The next task is to extract features of walks which allow us to classify subsets of walks based on differences between them. Ultimately, we will be able to provide a hierarchical clustering method for finite random walks. To separate two walks which start in the origin node and end in the same terminal node, we need to quantify the differences in *how* they get from the origin to the terminus. We will refer to distinguishing between walks based on this criterion (different ways to get from origin to terminus) as 'classification'. In contrast, we will refer to distinguishing between walks based on their termini as 'categorisation'.x

The following subsections on algebraic topology and persistent homology reveal the apparatus for *classification*.

## 1.6 Simplices and boundaries

First, we introduce *simplices*.

**Definition 9** ($n$-simplex)**.** *We define an $n$-simplex as a convex hull of $n+1$ affinely independent points. An $n$-simplex is a point for $n = 0$, a line segment for $n = 1$, a triangle for $n = 2$ and a tetrahedron for $n = 3$... We use the notation $\sigma = [v_1, v_2, ..., v_{n+1}]$ where $v$ are vertices.*

Clearly, the data points in our point cloud $X$ can be seen as 0-simplices and any connecting edges as 1-simplices. If any three distinct edges (1-simplices) between points in $X$ together form a triangle, this can be seen as a 2-simplex. In discussing relationships of $n$-simplices, we will make use of the *face* and *coface* relation.

**Definition 10** ($m$-face, coface)**.** *Let $\sigma$ be an $n$-simplex and $m \in [0, n]$. The convex hull of $m + 1$ vertices of $\sigma$ is an $m$-simplex that is an $m$-face of $\sigma$. We call 0-faces vertices, 1-faces edges and any $(n-1)$-faces are called facets. We call each $(n+k)$-simplex for which $\sigma_n$ is an $n$-face a codimension-$k$ coface.*

The concept of faces leads us to the useful definition of *boundaries*. (We limit ourselves to computation of boundaries over $\mathbb{Z}_2$ and use definitions that are useful for this purpose.)

**Definition 11** (Boundary operator, boundary)**.** *Let $\sigma_n$ be an $n$-simplex. The boundary operator $\partial$ sends $\sigma_n$ to a formal sum of its $(n-1)$-faces over $\mathbb{Z}_2$. We then refer to the result as the boundary of $\sigma_n$.*

Taking the formal sum over $\mathbb{Z}_2$ means that elements with even parity cancel out. Now, 0-simplices have no faces, therefore $\partial\sigma_0 = 0$. Additionally, any $\partial\partial\sigma_n = 0$

(boundary of a boundary)[1]. To conceptualise this, consider the boundary of a 2-simplex: three adjacent 1-simplices. Since each pair of these 1-simplices shares a 0-face, applying the boundary operator to these 1-simplices will get us three distinct 0-simplices, each with multiplicity 2. These cancel out. For a visual representation of this principle, see Figure 1.

It remains to formalise the combining of simplices into more complicated shapes. To capture the structure of a point cloud, we use *simplicial complexes* and describe their chaining.

**Definition 12** (Simplicial complex)**.** *We define a simplicial complex $C$ as a collection of simplices which meets the condition that for each simplex $\sigma \in C$ all the faces of $\sigma$ are also included in $C$. (Wasserman, 2018) Additionally, the intersection of any two simplices in $C$ will also be included in $C$.*

Now, we want to formally describe some subsets of $k$-simplices in a complex. (The reason to do this is to be able to eventually describe trajectories as combinations of 1-simplices.)

**Definition 13** ($k$-chain)**.** *For a positive integer $k$, we define a $k$-chain as the formal sum of some $k$-simplices over $\mathbb{Z}_2$. To describe a chain $\mathbf{c}$, we can use a set of indexed simplices $\{\sigma_1, \sigma_2, ..., \sigma_n\}$ and write down a vector*

$$c_i = \begin{cases} 1 & \sigma_i \textit{ included in the chain} \\ 0 & \textit{otherwise.} \end{cases}$$

[1]This is referred to as the fundamental lemma of homology and proven in *Chapter IV: Homology* of Edelsbrunner and Harer (2010). Some implications are also discussed in Hatcher (2001).

*Chains can then be chained (added together over $\mathbb{Z}_2$) further. This means that if we add some chains $\boldsymbol{c}^1$ and $\boldsymbol{c}^2$, the resulting chain will be*

$$c_i^{add} = \begin{cases} 1 & (c_i^1 = 1 \wedge c_i^2 = 0) \vee (c_i^1 = 0 \wedge c_i^2 = 1) \\ 0 & \textit{otherwise.} \end{cases}$$

*(In other words, we compute the formal sum of simplices in the same manner that we did for the simplices resulting as boundaries of a simplex in the previous example.)*

A special type of chain which will interest us is a *cycle*. The reason for their importance is again related to trajectories: chaining together two trajectories (1-chains) which share the same starting point and end point forms a *cycle*.

**Definition 14** (Cycle)**.** *We define a cycle as a $k$-chain $c$ such that its boundary $\partial c = \emptyset$. We will also use the term $k$-cycle.*

Figure 1 shows some simplices and chain of simplices and illustrates how the boundary operator works. It also provides some visual intuition for the fact that the boundary of a boundary is an empty set.

The boundary operator for a simplicial complex can be identified with a matrix.

**Definition 15** (Boundary matrix)**.** *Boundary matrix $B_p$ of a simplicial complex $C$ is defined for simplex dimension $p$ as*

$$B_{p\,ij}(C) = \begin{cases} 1 & \textit{$i$-th $(p-1)$-simplex is face of $j$-th $p$-simplex in } C \\ 0 & \textit{otherwise.} \end{cases}$$

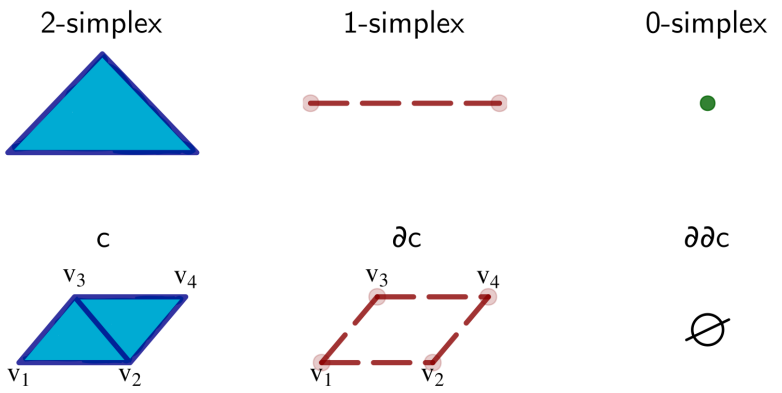


Figure 1: The top row shows graphical representations of a 2-, 1- and 0-simplices. Different colours are used to distinguish between simplices of different dimensions. The bottom row shows a 2-chain $\boldsymbol{c}$ on the left. In the middle, its boundary $\partial \boldsymbol{c}$ is shown. Note that the 1-simplex $[v_2, v_3]$ is not included in $\partial \boldsymbol{c}$, because the $[v_1, v_2, v_3]$ and $[v_2, v_2, v_4]$ share it as a facet. The chain $\partial \boldsymbol{c}$ also happens to be a cycle. Because of this, $[v_1]$, $[v_2]$, $[v_3]$ and $[v_4]$ each have even parity when we add the boundaries of all the 1-simplices in $\partial \boldsymbol{c}$. Therefore, the picture on the bottom right shows that $\partial\partial \boldsymbol{c}$ is empty (since they cancelled out).

*(Edelsbrunner and Harer, 2008)*

We can obtain boundaries $B$ of $p$-simplices in complex $C$ by computing $B_p(C)$. Additionally, we can concatenate boundary matrices for multiple values of $p$ by columns to obtain, for instance, a boundary matrix $D_{0,1,2}$ that records the boundaries of 0-, 1- and 2-simplices.

It remains to describe a specific way in which to connect points in our point cloud $X$.

## 1.7 Filtration

In this subsection we describe two types of simplicial complexes and explain the *process of filtration*, which lets us associate the constituent simplices with numeric values. This will help us describe shapes in different parts of our point cloud.

**Definition 16** (Filtration, process of filtration)**.** *Filtration is an indexed family of simplices and the process of filtration is a sequence of steps by which a filtration is obtained.*

*Let $X$ be a vertex set. We define the process of filtration as sequential insertion of some $n$ simplices into a simplicial complex $C$, with points in $X$ as its 0-simplices. The 'time' at which any simplex was added is referred to as its filtration value.*

*Simplices in $C$ are indexed, such that $C = \{\sigma_1, \sigma_2, ..., \sigma_n\}$, by non-decreasing filtration values. Before any simplex $\sigma$ is added, its boundary faces must have been added.*

*Formally, filtration is written as $\mathcal{F} = (C, \phi)$. Here, $\phi : \sigma \to \mathbb{R}_0^+$ is a function mapping simplices to their non-negative filtration values.*

We first introduce the useful concept of *Delaunay triangulation*. Then, we can define *alpha-complex filtration*, which is based on it.

**Definition 17** (Delaunay triangulation)**.** *Delaunay triangulation $DT(P)$ of a set of points $P$ is a simplicial complex such that no point in $P$ can be found in the circumsphere (circum-hypersphere) of any triangle in $DT(P)$.*[2]

[2]It turns out that the circumcenters (centers of circumspheres) in $DT$ of a plane are the vertices of its Voronoi tessellation.

**Definition 18** (Voronoi tessellation)**.** *Voronoi tessellation of a plane refers to partitioning of the*

**Definition 19** (Alpha complex, alpha-complex filtration)**.** *The $\alpha$ complex (alpha complex) is a sub-complex of the Delaunay triangulation with a filtration value function. For each n-simplex $\sigma_n$, we check whether its circumsphere contains any other vertices of the triangulation. If it does not, we say the simplex is 'Gabriel' and its filtration value becomes the square of the radius of the circumsphere. If it does, we identify all the codimension-1 cofaces which make it not Gabriel. The minimum of their filtration values becomes the filtration value of $\sigma_n$. We then take the minimum of their filtration values. If a value $\alpha$ is provided, simplices with filtration value greater than $\alpha^2$ are removed (Rouvreau, 2019).*

Intuitively, we can see that simplices which span a large part of the Euclidean space will generally be associated with higher filtration values than those which are small. This is a desirable property of alpha-complex filtration. Unfortunately, construction of an alpha complex is computationally demanding. To explain why, we first define the *n-skeleton of a simplicial complex.*

**Definition 20** ($n$-skeleton of a simplicial complex)**.** *For some non-negative integer n, we define the n-skeleton $C_n$ of a simplicial complex $C$ as the complex comprising the $0, ..., n$-simplices of $C$.*

In fact, we will only need to keep a 2-skeleton of our filtration, since we seek to describe pathways in data that we believe lies on a 2-manifold.

In order to obtain a 2-skeleton of an alpha complex, we first need to construct the entire alpha complex (we cannot simply limit ourselves to inserting 0-, 1- and 2-

*plane into Voronoi cells: convex polygons, each containing a single seed. The seed of each cell is a point such that all the points in that cell are closer to it than to the seed of any other Voronoi cell.*

simplices during construction) and then re-compute filtration values. This is largely computationally unfeasible for data of high dimensionality.

To tackle this problem, we need to define the *witness complex*. This time, computation of filtration values is not addressed in the definition and will be solved separately.

**Definition 21** (Witness complex)**.** *Let $L$ be a vertex set of $m$ 'landmark points' and $V$ be a vertex set of $k$ 'witness points'. We construct a witness complex $W$ in the following way. Any simplex $\sigma \subset L$ is witnessed by $v \in V$ if $\forall l \in \sigma, \forall l' \in L \setminus \sigma$ : $d(v,l)^2 \leq d(v,l')^2 + \alpha^2$ where $d$ is the Euclidean distance function. We bound the initial dimension of $\sigma$ to 1. (Additionally, if $\alpha \neq 0$, we call $W$ a relaxed witness complex.) Each 1-simplex that is witnessed is added to $W$ and, inductively, any $p$-simplex is added when all its faces have already been added. (See lazy witness complex in Kachanovich (2019).)*

The principle of 'witnessing', essential for witness-complex construction, is illustrated in Figure 2.

In `tviblindi`, the set of data points are partitioned into landmarks and witnesses by the means of clustering using the $k$-means algorithm or via `FlowSOM` (Van Gassen et al., 2015). Specifically, for some choice of $k''$ we find the $k''$-th nearest cluster centers to each point in our point cloud $X$ and compute the distances. The original points in $X$ then act as witnesses and cluster centers in their neighbourhoods act as landmark points.

In contrast to the alpha complex, witness-complex construction *can* be limited to inserting 0-, 1- and 2-simplices to obtain the 2-skeleton. In `tviblindi`, filtration

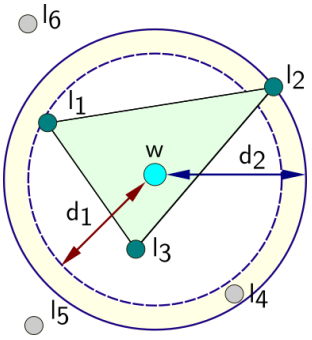


Figure 2: Landmarks $l_1, l_2, ..., l_6$, witness $w$ and two concentric circles in 2-dimensional space are shown. Simplex $\sigma = l_1, l_2, l_3$ is witnessed by $w$ in a relaxed witness complex with given $\alpha$ where $d_1 = \sqrt{d(w, l_1)^2}$ and $d_2 = \sqrt{d(w, l_1)^2 + \alpha^2}$.

values of the witness complex are subsequently calculated as alpha-complex filtration values. In that regard, alpha-complex construction is mimicked but filtration can be applied directly to high-dimensional data, without prior dimension reduction.

## 1.8 Persistent homology

In this sub-section, we make use of the concept of filtration and introduce concepts from persistent homology, which will then allow us to introduce a method for hierarchical clustering of walks using classification.

For the purposes of this work, persistent homology serves as a tool for describing relatively sparse regions in high-dimensional data. Central to the `tviblindi` pipeline is an examination of cycles, which we explain below.

To start, we assume that a filtration $\mathcal{F} = (C, \phi)$ of our point cloud $X$ is available. We take simplices in $C$ associated with up to some filtration value $f$, so as to obtain

a sub-complex $C_f \subset C$.

Next, we note the following: all the boundaries which can be obtained in $C_f$ are a subset of all cycles in $C_f$. Conversely, there exist some cycles which *cannot* be obtained as boundaries. This leads us to the definition of *equivalence class* and *homology class*.

**Definition 22** (Equivalence, equivalence class)**.** *Equivalence is a binary relation* $\sim$ *on some set* $S$ *which satisfies reflexivity* ($\forall x \in S : x \sim x$)*, symmetry* ($\forall x, y \in S : x \sim y \implies y \sim x$) *and transitivity* ($\forall x, y, z \in S : x \sim y \wedge y \sim z \implies x \sim z$)*. The equivalence class* $[s]$ *of an element* $s \in S$ *for an equivalence relation* $\sim$ *is then defined as* $\{x \in S | x \sim s\}$.

**Definition 23** (Homology class)**.** *We define a homology class of a cycle* $k$ *in a simplicial complex* $H$ *as a set of cycles such that their difference with* $k$ *over* $\mathbb{Z}_2$ *is a boundary cycle. This is a type of equivalence class, denoted* $[k]$*. If we fix a dimension* $n$ *and denote all* $n$*-dimesional cyctes in* $H$ *as* $K_n$ *and all* $n$*-dimensional boundaries in* $H$ *as* $B_n$*, we define the homology class of* $k$ *as* $[k] = \{l \in K_n | k + l \in B_n\}$.

Both concepts are adopted from Edelsbrunner and Harer (2008).

By finding all non-trivial homology classes, we identify topological cavities. This translates roughly to the following: identifying cycles in $C_f$ can help us describe holes that are present at some point during the process of filtration (before they are filled with simplices). These holes can help us to describe the shape of our point cloud $X$.

On the other hand, not all cycles in $C_f$ will represent an 'empty hole' in $X$. For instance, cycles which are filled with a large amount of data points probably do not

represent any cavity in the shape of $X$. Conversely, if there are few data point inside the cycle (or none), the cycle is likely representative of a 'significant empty hole'. Persistent homology gives us a way to distinguish between these cases in a robust manner.

Suppose a simplicial complex is obtained via alpha-complex filtration. This process can be seen as the sequential addition of simplices until the entire shape, defined by our data points, is filled. Large simplices (those which cover large segments of the shape) are added *later* than small simplices (those which connect vertices close to each other). This amounts to interpreting *filtration values* as a time parameter (simplices with lower filtration values are added *earlier*). As we proceed with adding simplices, cycles are formed, and they are at first empty holes: they are not *yet* filled with simplices. The empty holes, as noted earlier, correspond to homology classes. As the process of filtration continues, homology classes are created (they are *'born'*). Once a simplex is inserted which fills the empty hole, the given homology class ceases to exist (it *'dies'*). Now, homology classes differ in terms of their persistence, *i.e.* difference between 'time of death' and 'time of birth'.

Since the 'time' parameter corresponds to filtration values, the persistence of a homology class is in fact *the difference between filtration value of a simplex which caused its death (filled an existing cycle) and the filtration value of a simplex which caused its birth (closed that cycle).* The more persistent the homology class, the more significant a hole in the overall shape it represents, since it is evidence of a sparse region surrounded by a denser region.

It is possible to generalise this example: homology classes are not limited to 1-

cycles. In this work, however, we are concerned with 1-homology, *i.e.* we consider 1-cycles and disregard the rest.

To capture relationships between simplices in our simplicial complex $C$, we construct a boundary matrix $D_{0,1,2}$ (recall Definition 15). Boundary matrices acquire more useful properties when manipulated via elementary column operations over $\mathbb{Z}_2$. The algorithm for this, taken verbatim from Edelsbrunner and Harer (2008), proceeds as follows. *'Let* `low(j)` *be the row number of the lowest non-zero entry in column $j$, where we set* `low(j)` $= 0$ *if the entire column is zero. We call $D$ reduced if the restriction of* `low` *to its non-zero columns is injective, that is, each row has at most one entry that is the lowest* 1 *for a column. To reduce $D$ we proceed from left to right and expand the reduced submatrix one column at a time.*

```
for j = 1 to n do
   while ∀j′ < j with low(j′) = low(j) ≠ 0 do
      add column j′ to column j
   end while
end for
```

*Adding column $j'$ decreases* `low(j)` *which implies that the algorithm terminates after at most $n^2$ column operation[s].'* Despite this, actual running time in `tviblindi` analyses is rarely prohibitive. Implementation of algorithms for working with boundary matrices are laid out in Bauer et al. (2016).

In a boundary matrix $D$, rows and columns correspond to single simplices in our complex. In a reduced boundary matrix $R$, columns are generated as sums of columns of $D$ and they correspond to chains: each column specifies the boundary of

a chain of simplices.

Consider that the algorithm adds some column $j'$ to $j$. The lowest entry $i$ of column $j$ corresponds to an $(n-1)$-simplex $\sigma_i$ that is the $(n-1)$-simplex whose addition closed a cycle and caused a homology class to be born. Since this cycle is the boundary of the simplicial complex in column $j$ (created by column operations), the addition of $\sigma_j$ kills the class (filling the cycle).

With the reduced boundary matrix we are able to pinpoint the birth and death of each homology class. (We will use $D_{0,1,2}$ and $R_{0,1,2}$ in the rest of the section.) An example of the boundary matrix and reduced boundary matrix corresponding to a simplicial complex in 2 dimensions is shown in Figure 3.

A common way to visualise persistences of homology classes is the *persistence diagram*. Its horizontal axis shows filtration values corresponding to birth of a homology class. Its vertical axis shows filtration values corresponding to death of a homology class. Each homology class is represented by a point on this diagram. The further the point is from the diagonal (it can be above or to the left), the more persistent a homology class it represents.

For instance, when we work with 0-, 1- and 2-simplices (points, line segments and triangles), we can look for 0-homology classes (connected components) and 1-homology classes (to which we referred above). As stated earlier, our analysis focuses on 1-homology. In Figure 4 we show an example of a persistence diagram.

The persistence diagram (rotated) is used as an interactive element in `tviblindi` for selecting homology classes used for classification of trajectories.

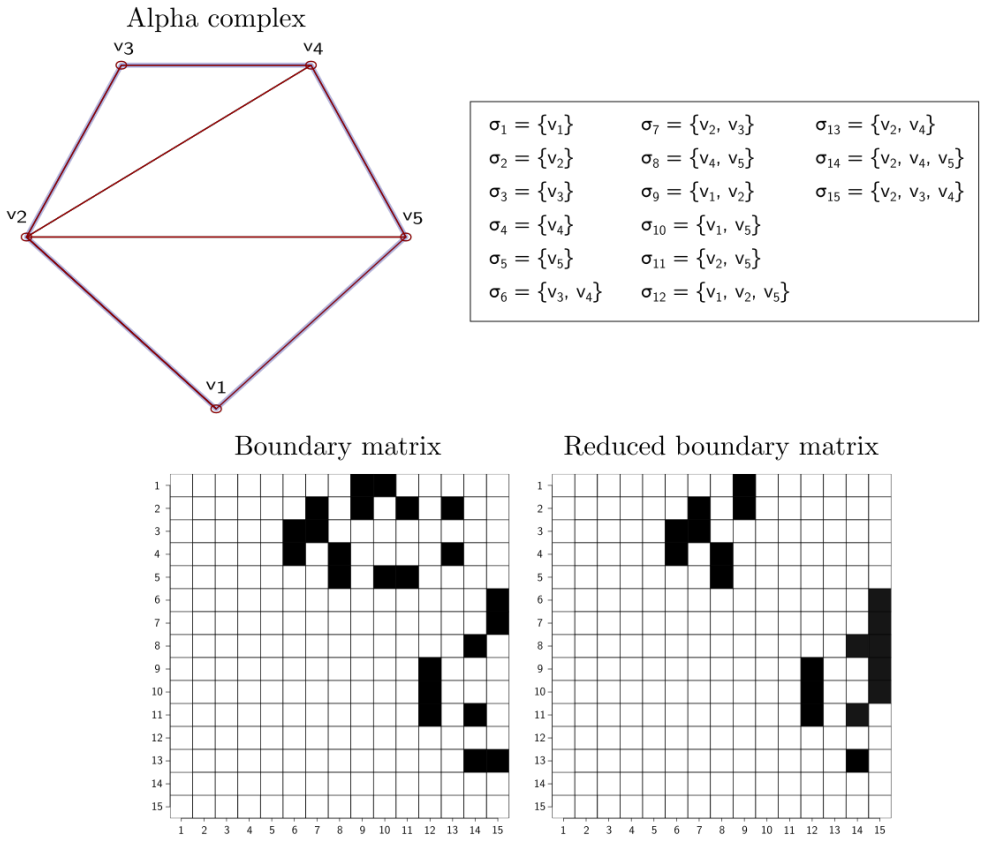


Figure 3: The simplicial complex created by alpha-complex filtration of a pentagon in $\mathbb{R}^2$ is shown on the top-left, with definitions of all simplices (by non-decreasing filtration value) beside it. Below, the boundary matrix and reduced boundary matrix are shown, with black squares corresponding to entries of 1 and white squares to 0.

## 1.9 Triangulation and classification of walks

Previously, we simulated random walks through a graph $G_t$ using a transition model $T$. Furthermore, we identified terminal nodes of $G_t$. Intuitively, the initial step of categorisation involved diving the walks up by terminal nodes. (By definition, the

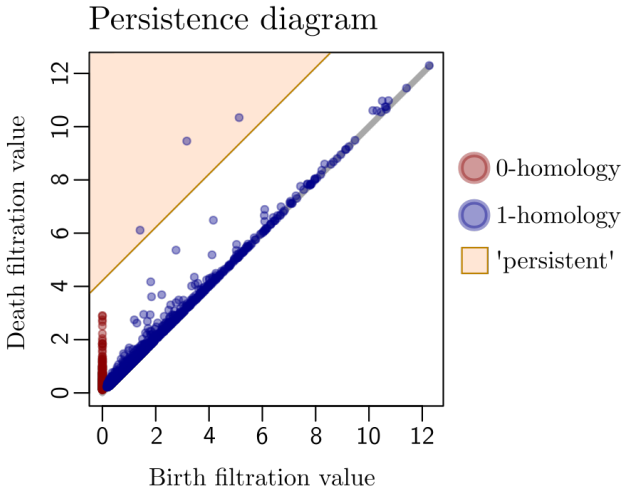


Figure 4: A sample persistence diagram is shown. Red dots correspond to 0-homology classes. Blue dots correspond to 1-homology classes. Note that the filtration value of birth of all 0-homology classes is 0. This is since a filtration value of 0 is assigned to all 0-simplices and the addition of a 0-simplex causes the class to be born. The filtration values of death of 0-homology classes are the filtration values of 1-simplices which connected the classes. The beige triangle shows an arbitrary delimitation of the 'most persistent' homology classes which the user conducting analysis can define for herself. These most persistent homology classes can then be used for classification of walks using their triangulated counterparts.

first node is common to all trajectories: it is the cell-of-origin.)

The simulated finite random walks cannot be classified based on persistence of homology classes directly, since $G_t$ and the simplicial complex $C$ were built independently. We need to 'triangulate' the walks first: this refers to their reconstruction using only the 1-simplices in $C$. We take the 1-skeleton of $C$ and convert it to a graph $G_c$. Then, each edge $\{v_i, v_j\}$ in a walk is approximated independently using the shortest path from $v_i$ to $v_j$ in $G_c$. Subsequently, any cycles which were created by this triangulation are removed. In the rest of this subsection, the term 'trajectory'

will be used to mean 'triangulated random walk'.

If we chain two trajectories in the same category, the result is a 1-cycle. (This is because the first and last node are common to both trajectories.) Now, this 1-cycle can be represented as an addition (over $\mathbb{Z}_2$) of chains of simplices representative of homology classes. In other words, we take the columns of our reduced boundary matrix $R_2$ and find the unique combination of some of them such that their addition produces the encoding for our 1-cycle. (The algorithm for finding the relevant combinations of columns of $R_2$ for our cycles is laid out in Pokorny et al. (2014).)

We are now clearly able to trace each pair of trajectories to a set of homology classes. By extension, say we randomly pick a single 'base' trajectory in a category. Then, we iteratively form cycles with each of the other trajectories in its category. We can associate each trajectory with a set of homology classes that corresponds to the 'difference versus base' (since each of them forms a cycle when connected with base). This way, we obtain coordinates of each trajectory with respect to 'base'. We call the set of homology classes associated with a trajectory the *trajectory representation*. By definition, we set the trajectory representation of the base trajectory to $\emptyset$. It does not matter which trajectory was chosen as base.

Ultimately, we can classify trajectories based on the following rule. If the representations of two trajectories differ by some homology class which the user deemed as significant, they must be classified differently.

This principle of trajectory classification is then further extended to *hierarchical clustering by persistence*, where a classification tree is built. To build this tree, the user first selects homology classes from an interactive persistence diagram. Height in

the tree corresponds to filtration values of simplices which killed the given homology classes, and each split corresponds to a difference between groups of walks in terms of presence of the homology class.

# 2 Implementation details

## 2.1 *k*-nearest-neighbours graph construction

The initial building of a $k$-NN graph is computationally demanding. `tviblindi` offers an exact and a faster approximate option for this.

For the exact option, we use a vantage-point tree data structure (Uhlmann, 1991) and use `OpenMP` for parallelisation (`C++` code from a multicore `t-SNE` implementation in Ulyanov (2016) was adopted for this).

For the approximate option, `tviblindi` includes an `R` wrapper for the `Annoy` algorithm, provided in (Bernhardsson, 2020), making use of some code from `RcppAnnoy` (Eddelbuettel, 2020).

## 2.2 Data denoising

Technical noise is always present in single-cell data. The presence of unwanted data points in sparse regions which are used for characterisation of the shape of our point cloud can make the use of persistent homology less effective at describing our data. In order to reduce this noise and help capture the topology correctly, we apply the following denoising technique, described in Carlsson et al. (2008).

Taking advantage of having built a $k$-NN graph previously, we replace the coordinates of each data point with the averaged coordinates of some $l$ of its nearest neighbours. This can be repeated iteratively.

The denoised version of our expression matrix is used to make the construction of our witness complex (described in Subsection 1.7 and below in Section 1.7) sim-

pler, with fewer outliers. (Additionally, the denoised expression matrix is used as input to the dimensionality reduction algorithm if present.) For the simulation of random walks through our point cloud, we use the original expression matrix without denoising.

## 2.3 Dimensionality reduction

For visualisation of our input dataset, we construct a 2-dimensional projection of the original high-dimensional data. By reducing dimensionality of $X$ we obtain an $n$-by-2 coordinate matrix which can be used for plotting points coloured according to their labels or pseudotime values. While `tviblindi` enables the use of principal component analysis, `t-SNE` (Van Der Maaten and Hinton, 2008) and `UMAP` (McInnes et al., 2018), we recommend the use of `vaevictis`: a custom dimension-reduction tool provided with `tviblindi`.

## 2.4 Pseudotime computation

As described in Subsection 1.3 of the previous section, pseudotime computation results in a vector of values from $\mathbb{R}_0^+$ per each row of $X$. This allows to orient edges of the graph representing our transition model, *i.e.* the allowed paths for traversal of the point cloud. This is done by solving a system of linear equations. Such a system can be represented in matrix form $A\boldsymbol{x} = \boldsymbol{b}$ (solving for $\boldsymbol{x}$). In addition, we know that $A$ is a symmetric matrix in this case.

Now, for a small number of vertices the system can be solved exactly: $x = A^{-1}\boldsymbol{b}$. However, the worst-case complexity of $O(n^3)$ might be prohibitive. For this reason,

we solve numerically for more than $1,000$ vertices by default. For our numerical solution, we use the conjugate gradient method, as implemented in `RcppEigen`, integrating the `C++ Eigen` library for linear algebra for use with `R` code (Guennebaud and Benoît, 2010; Bates and Eddebuettel, 2014). Crucially, the computation error when using this method is reported. This is an important sanity check, indicating whether pseudotime values converged and can be expected reasonably to estimate the maturity of cells along a developmental continuum.

An additional parameter in the implementation of pseudotime computation is the number $k'$ of nearest neighbours for the purposes of building a transition model. Here, the previously constructed $k$-NN graph is replaced by a $k'$-NN graph ($k' \leq k$). The choice of $k'$ can ultimately influence the number and location of terminal nodes of simulated walks. The same approach of only considering a limited number $k'$ of neighbours can be applied in the simulation of random walks after the computation of pseudotime.

## 2.5 Filtration

The process of filtration is essential for correct description of the shape of $X$. The theory behind filtration was explained in Subsection 1.7.

Since construction of an alpha complex is usually computationally prohibitive without prior dimensionality reduction, a witness complex is used.

## 2.6 Clustering data for witness-complex construction

A major performance boost to the `tviblindi` pipeline has been implemented by applying the $k$-means or Kohonen self-organising maps (SOM) (Kohonen, 1990) clustering algorithm, using the `stats` package of `R` or `FlowSOM` (Van Gassen et al., 2015), respectively. The explanation of $k$-means clustering is omitted here, since it is ubiquitous in the clustering literature. The Kohonen model is a simple neural network with $d$ (equal to number of columns in $X$) input neurons and $m$ output neurons (in the 'Kohonen layer'). In `FlowSOM`, the Kohonen layer is built as a two-dimensional grid with pre-set dimesions $x$-by-$y$, $m = x \times y$. Each output neuron is associated with $d$ synaptic weights $w_i, 1 \leq i \leq d$, initialised randomly. In training a Kohonen model, weight vectors $w_{ij}$ for $i$ from $[1, d]$ and $j$ from $[1, m]$ are then iteratively adjusted in the following manner.

At time $t$, a training pattern $\boldsymbol{x}$ (a row of $X$) is presented. Distance between training pattern and weight vectors is computed as

$$\delta_j = \sqrt{\sum_{i=1}^{d} (x_i(t) - w_{ij}(t))^2}$$

Now, a single output neuron with minimum distance from the input is selected and its index is saved as $k$. Weights are updated using the rule

$$w_{ij}(t+1) = w_{ij}(t) + \alpha(t)\phi(k, j)(x_i(t) - w_{ij}(t))$$

where $\alpha(t)$ is a vigilance coefficient between 0 and 1 which decreases over time. $\phi(k, j)$ is a lateral inhibition function which governs how neighbouring nodes of $k$-th

node are affected. In `FlowSOM`,

$$\phi(k,j) = \begin{cases} 1 & \sqrt{(\sum_{i=1}^{d}(w_{ik} - w_{ij})^2)} < \epsilon(t) \\ 0 & \text{otherwise} \end{cases}$$

where $\epsilon$ is a distance boundary that decreases over time. (Euclidean distances were used in this example. `FlowSOM` allows for use of other distance metrics.)

It can be shown that this algorithm converges. The weight vectors (vectors of synapse weights leading to each of the $m$ output neurons) are taken as cluster centers and each data point is assigned to the cluster with the nearest center. In cluster analyses with `FlowSOM`, we would often proceed to form *metaclusters* by means of hierarchical clustering of $m$ clusters. However, in the `tviblindi` pipeline we use the original high-resolution cluster centers, as they tend to reflect the overall shape of our point cloud.

We propose that `FlowSOM` clusters can capture the basic shape of our data. This is useful in the context of using persistent homology for classification of simulated walks. To enable the usage of SOM clustering for our method of trajectory inference, we implement a 'contraction' of our simulated random walks to walks through the clusters, with additional removal of any cycles which can possibly be induced by this. Classification of walks based on persistent homology is done using the clustered data. However, the mapping from original random walks to the contracted triangulated trajectories is kept, so that classified original random walks can be exported.

As defined in Subsection 1.7, the witness complex is a type of simplicial complex. For its construction, we need to provide two sets of points: witnesses and landmarks.

The previously produced SOM clusters are especially useful for this purpose. For some choice of $k''$, we find the $k''$-th nearest SOM cluster centers to each point in our point cloud $X$ and compute the distances. The original points in $X$ then act as witnesses and cluster centers in their neighbourhoods act as landmark points.

A relaxed witness complex is constructed using these inputs and a specified $\alpha$ parameter. Subsequently, filtration values are computed for each simplex in the complex as filtration values of an alpha-complex 2-skeleton.

## 2.7 `Shiny` interface

An essential part of `tviblindi` is a user interface written in `R Shiny`. This is a web application which can be launched after filtration and the simulation of walks through $X$. The app contains a manual for first-time users and is made to be accessible even to users with little to no knowledge of the theoretical background described in this work.

The application layout is divided into three panes: left, middle and right. The left pane is dedicated to selection of developmental trajectories by terminal nodes, plotted on a 2-dimensional layout of data, and selection of homology classes by persistence, plotted using a modified persistence diagram. (Selection of homology classes is equivalent to identifying relatively sparse regions in data for the use in characterisation of differences between developmental trajectories.) The middle pane uses the new approach of *hierarchical clustering by persistence* (HCP) to allow the user to select walks on an interactive dendrogram and add them to either or two groups: A or B. It also includes control elements for appending marked trajectories

to enhanced FCS files and exporting them. The right pane uses raster graphics for rapid plotting of trajectories and contains tools for tracking the expression of markers of interest or composition of segmennts in terms of labelled populations. It also allows for highlighting points in marked simulated walks which lie in specified pseudotime segments on the 2-dimensional layout. This serves for better understanding of the visual representation. Moreover, the user is able to remove ('zap') select walks based on tracked marker expression.

In the construction of a transition model, terminal nodes are identified in the data. If the model captures developmental dynamics correctly, these termini correspond to the most differentiated cells in development. However, in practice we encounter multiple identified terminal nodes which are close to each other; this is apparent also from their closeness in the 2-dimensional layout created via dimensionality reduction.

Analysis in the GUI begins with the user selecting all the terminal nodes which are considered to correspond to a single cell fate. Selection can be done iteratively, by drawing rectangular selections and adding the selected terminal nodes to those marked previously. In case of mistakes, the set of marked nodes can be cleared. (Throughout the GUI, this distinction between 'selection' and subsequent 'marking' is made.) When the desired terminal nodes are marked, the terminal node of each of the relevant simulated walks is set to the marked node with highest pseudotime and edges connecting all other termini are added. By ensuring a common terminal node for the walks, we meet the condition that by concatenation of any two triangulated trajectories we obtain a cycle (the reason for this was explained in Subsection 1.8).

After marking desired terminal nodes, the user can mark homology classes of

interest by their persistence. This corresponds to choosing which relatively sparse regions in the shape of our point cloud are to be considered for hierarchical clustering of simulated walks. In practice, the representation of each triangulated trajectory is filtered to only include those marked homology classes.

For a visualisation of homology classes, the persistence digram for our filtration is generated and rotated by 45° clockwise. (The horizontal axis of a regular pesistence digram shows filtration values of birth: we can call them $B$. The vertical axis of a regular persistence diagram shows filtration values of death: we can call them $D$. In the modified persistence diagram used here, the vertical axis shows $\frac{D-B}{2}$ and the horizontal axis shows $\frac{D+B}{2}$. This way, space under the diagonal, which is empty by definition, can be omitted.)

This modification of the persistence diagram lets the user apply a two-part heuristic when marking points on the diagram, if desired. First, to pick the most persistent homology classes the user marks those points which are 'on the top', omitting the bulk of the points on the bottom: this selection corresponds to separation of dense regions by the sparse ones (surrounded by dense boundaries). Second, to pick the homology classes corresponding to relatively large empty holes (those which were born late and died late) the user marks those points 'on the right': this selection corresponds to low probability paths going through sparse regions.

Once a set of simulated walks is chosen by termini and homology classes for use in classification are marked, triangulated trajectories can be clustered hierarchically. (Since each triangulated trajectory corresponds to a simulated random walk, we can also say that we obtain the hierarchical clustering of walks. In fact, at this

point we do not need to distinguish between walks and trajectories.) A custom method of clustering based on persistent homology is implemented: we use the term *hierachical clustering by persistence*, or the abbrevatiation HCP. The result of HCP is a horizontal dendrogram with root on the right end.

Horizontal position in the dendrogram corresponds to filtration value of simplices which kill, *i.e.* connect homology classes, with increased values to the right. Leaves of the dendrogram group together trajectories with identical filtered representations (*i.e.* representations using only homology classes which were chosen prior to this). Each merging of branches occurs 'at' the filtration value of death of some homology class $h$: the merged branches here are those whose representations differ *only* in the presence or absence of $h$. For high enough filtration values all the homology classes have died, therefore all chosen trajectories are clustered together at the root of the dendrogram.[3]

Construction of the tree (dendrogram) is implemented as recursive building of a binary tree structure. The displayed dendrogram is interactive, allowing the user to select leaves of the tree and mark the corresponding trajectories to add them to one of two groups: A or B. The user can choose a part of the dendrogram and zoom in to view it in more detail. This is helpful for trees with many branches.

A plot of walks in both groups drawn over the 2-dimensional projection is updated with each change in the content of group A or group B.

For each group of walks the user can view how expression of markers of interest

[3]In contrast to the Delaunay triangulation, it is not guaranteed that the witness complex does not contain holes, *i.e.* undying classes. This can be easily identified and rectified by augmenting the relaxation parameter.

changes with increasing pseudotime (*i.e.* along the developmental pathway). This is done by dividing the relevant data into a chosen number of pseudotime segments. The average measure of expression in all walks per segment is displayed. If a single marker of interest is selected, average values per segment for each walk are displayed instead, providing greater resolution.

Now, it is entirely possible that the ordering of events based on pseudotime values is correct, yet the values themselves are inaccurate in that some stages of development are estimated as being relatively shorter or longer than we believe them to be. For this reason, the user might prefer the values at which pseudotime is cut into segments to be distributed unevenly, thereby compensating for this distortion.

This can be achieved in the following way. To divide pseudotime values up into $n$ segments, an increasing sequence of $n+1$ evenly spaced values from $[0, 1]$ is generated at first. Then, we exponentiate each value in this sequence to the $s$-th power where $s$ is called a user-defined *scaling exponent*. The values in this sequence are then used to delimit the span of each of the $n$ pseudotime segments. For a scaling exponent less than 1, pseudotime in earlier stages in development is comparatively 'stretched out', whereas for a scaling exponent larger than 1, the later stages are.

In addition to being able to view the expression of markers of interest, the user can remove ('zap') select walks based on the expression of a single marker. This leads to easy 'purification' of groups of walks based on prior knowledge, in case HCP does not provide sufficient resolution. Furthermore, a range of pseudotime segments can be highlighted on the 2-dimensional projection, which proves useful in understanding the mapping of different phenotypes onto the projection.

In much the same way that the user can track the expression of markers of interest, the counts of data points with any manually entered label of interest across pseudotime segments can be viewed. In case large spikes in population sizes are observed, $\log_2$ of these counts can be viewed instead.

Once the user is satisfied with her selection of walks, the marked nodes in either group can be exported, along with pseudotime values for each node. This is done by appending artificial 'channels' to the original input FCS file, allowing for further analysis of data with standard software for analyses of cytometric data. Multiple batches of pseudotime values for events included in marked simulated walks can be added iteratively.

For each batch, a corresponding `which_event` and `local_pseudotime` channel are added. In each appended channel, a placeholder value of $-100$ is assigned to any event not included in the marked walks (or not included in the `tviblindi` analysis if only a specified subset of events recorded in an FCS file was used in analysis). For the included events, pseudotime values re-computed as ordering (integers from 1 to number of included events) are entered in the `local_pseudotime` channels and values of $1,000$ are entered in the `which_event` channels. The 2-dimensional layout of data is also appended to the exported FCS file (channels `dimension_reduction_1` and `dimension_reduction_2`). This allows for easy manual gating on pseudotime values or coordinates in the 2-dimensional layout.

# Bibliography

Bates, D., and D. Eddebuettel, 2014: Fast and Elegant Numerical Linear Algebra Using the RcppEigen Package. *Journal of Statistical Software*, **52 (5)**, 1–24, doi: 10.18637/jss.v052.i05, URL http://www.jstatsoft.org/v52/i05/.

Bauer, U., M. Kerber, J. Reininghaus, and H. Wagner, 2016: PHAT – Persistent Homology Algorithms Toolbox. *Journal of Symbolic Computation*, **78 (1-2)**, 76–90, doi:10.1016/j.jsc.2016.03.008.

Bernhardsson, E., 2020: Annoy (Approximate Nearest Neighbors Oh Yeah). GitHub, URL https://github.com/spotify/annoy.

Carlsson, G., T. Ishkhanov, V. De Silva, and A. Zomorodian, 2008: On the local behavior of spaces of natural images. *International Journal of Computer Vision*, **76 (1)**, 1–12, doi:10.1007/s11263-007-0056-x.

Eddelbuettel, D., 2020: RcppAnnoy: 'Rcpp' Bindings for 'Annoy', a Library for Approximate Nearest Neighbors. CRAN, URL https://cran.r-project.org/package=RcppAnnoy.

Edelsbrunner, H., and J. Harer, 2008: Persistent homology — a survey. *Contemporary Mathematics*, **453 (January 2015)**, doi:10.1090/conm/453/08802.

Edelsbrunner, H., and J. Harer, 2010: *Computational Topology - An Introduction*.

Guennebaud, G., and J. Benoît, 2010: Eigen v3. URL http://eigen.tuxfamily.org.

Hatcher, A., 2001: Simplicial Homology. *Algebraic Topology*, Cambridge University Press, chap. Simplicial, 104–107, URL http://pi.math.cornell.edu/~hatcher/AT/AT.pdf.

Kachanovich, S., 2019: Witness complex. URL http://gudhi.gforge.inria.fr/doc/latest/group__witness__complex.html.

Kohonen, T., 1990: The Self-Organizing Map. *Proceedings of the IEEE*, **78 (9)**, 1464–1480, doi:10.1109/5.58325.

McInnes, L., J. Healy, and J. Melville, 2018: UMAP : Uniform Manifold Approximation and Projection for Dimension Reduction. arXiv:1802.03426v2.

Pokorny, F. T., M. Hawasly, and S. Ramamoorthy, 2014: Multiscale Topological Trajectory Classification with Persistent Homology. *Robotics: Science and Systems*, doi:10.15607/rss.2014.x.054.

Richards, J., and R. Cannoodt, 2020: diffusionMap. GitHub, URL https://github.com/rcannood/diffusionMap.

Rouvreau, V., 2019: Alpha-complex. URL http://gudhi.gforge.inria.fr/doc/latest/group__alpha__complex.html, 2019 pp.

Uhlmann, J. K., 1991: Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, **40 (11)**, 175–179, doi:10.1016/0020-0190(91)90074-R.

Ulyanov, D., 2016: Multicore-TSNE. GitHub, URL https://github.com/DmitryUlyanov/Multicore-TSNE.

Van Der Maaten, L. J. P., and G. E. Hinton, 2008: Visualizing Data using t-SNE. *Journal of Machine Learning Research*, **9**, 2579–2605, doi:10.1007/s10479-011-0841-3, 1307.1662.

Van Gassen, S., B. Callebaut, M. J. Van Helden, B. N. Lambrecht, P. Demeester, T. Dhaene, and Y. Saeys, 2015: FlowSOM: Using Self-Organizing Maps for Visualization and Interpretation of Cytometry Data. *Cytometry Part A*, **87 (7)**, 636–645, doi:10.1002/cyto.a.22625.

Wasserman, L., 2018: Topological Data Analysis. *Annual Reviews of Statistics and Its Application*, **2018 (5)**, 501–532, doi:10.1146/annurev-statistics-031017-100045.