

Supplementary Note

Contents

1 Datasets	4
1.1 <i>dyntoy</i> dataset	4
1.2 Human thymus & PBMC	5
2 Dimensionality reduction	5
3 Pseudotime estimation & random walks simulation	7
3.1 Theoretical concepts	7
3.2 On numerical aspects of the solution	11
4 Topological clustering of random walks	12
4.1 Basic notions of simplicial homology	12
4.2 Triangulation of a point cloud & random walks	17
4.2.1 Data denoising	18
4.3 Cycle representation	19
4.4 Topological hierarchical clustering of random walks	23
5 Connectome	24
6 Graphical user interface (GUI)	24
7 Running the analysis	31
8 Performance evaluation	33
8.1 Sensitivity to hyperparameters	33
8.1.1 Number of nearest neighbors	33
8.1.2 Relaxation parameter for <i>witness complex</i> construction	33

8.2	Running times	34
-----	---------------	----

1 Datasets

In this section we give a brief description of datasets used in this supplementary note.

1.1 *dyntoy* dataset

For the purposes of basic demonstrations we created an artificial dataset using the *dyntoy* R package (?) as follows:

```
devtools::install_github('dynverse/dyntoy')

library(dyntoy)

n_events <- 10000

n_features <- 3896 #randomly chosen value from typical range

set.seed(12345)

d <- dyntoy::generate_dataset(
  id          = 'tviblindi_dyntoy_test',
  model       = 'connected',
  num_features = n_features,
  num_cells    = n_events
)
```

Package *dyntoy* simulates single-cell expression data organized into trajectories. It creates a random milestone network (Figure 1a). The *milestones* (named $M\#$) can be understood as populations of cells. We used the first 25 principal components for downstream analysis. To further highlight some issues that come up in trajectory inference, we modified this dataset by up-sampling a population labeled $M10$, by iterating over all points assigned to this population and adding random convex combinations of their 30 nearest neighbours. This resulted in a new dataset with 35,795 points where 26,532 points lie in

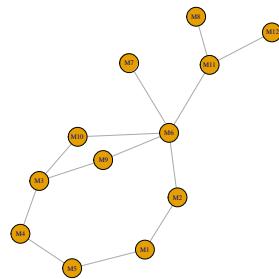
the population $M10$; we will refer to this dataset as the 'up-sampled' *dyntoy* dataset (see Figure 1d,e).

1.2 Human thymus & PBMC

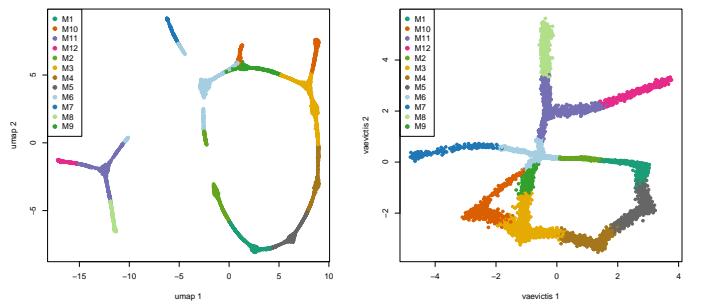
A mass cytometry dataset from human thymus and peripheral mononuclear cells from healthy donors (PBMC) was measured using *Mass panel 1*. (Supplementary Table, Mass panel 1). The analyzed dataset consists of 1,182,802 events and 34 dimensions (markers).

2 Dimensionality reduction

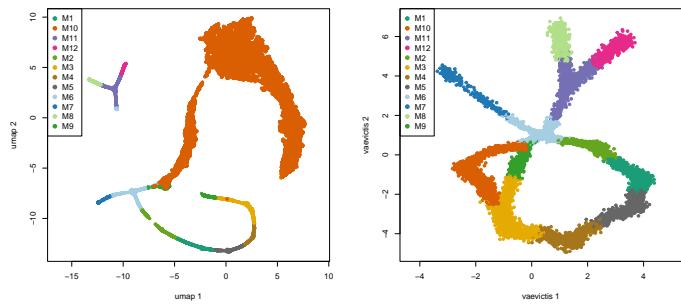
To meet the requirements for data visualization in trajectory inference (TI) problems, i.e. continuity, robustness against the differences in sizes of cell populations and straightforward extrapolation to new datasets, we implemented a variational autoencoder model based on ideas in ??, called *vaevictis*. The objective function is a weighted sum of reconstruction error, variational regularization (divergence from a latent prior distribution) and two regularization terms: the loss function value adapted from *t-SNE* and triplet loss from *ivis*. For additional speedup (which makes the time complexity virtually independent of dataset size) and improved robustness against differences in population size, the dataset is by default first clustered using k -means clustering and a fixed number of representatives are sampled from each cluster. This method is implemented in a separate GitHub repository: <https://github.com/stuchly/vaevictis>.



(a) Topology of *dyntoy* data



(b) 2D UMAP embedding of (c) 2D *vaevictis* embedding of
dyntoy data dyntoy data



(d) 2D UMAP embedding of (e) 2D *vaevictis* embedding of
up-sampled dyntoy data up-sampled dyntoy data

Figure 1: *dyntoy* data structure preservation with dimensionality reduction

Figure 1 illustrates the performance of *vaevictis* in producing a 2-dimensional embedding of the original and the up-sampled *dyntoy* dataset (subsection 1.1), as compared to a corresponding UMAP embedding. While the overall structure of this data (panel *a*) is preserved by *vaevictis* (panel *c*), even for data with numerically unbalanced populations (panel *d*; the population *M10* was enlarged to contain $\sim 74\%$ of all cells, see subsection 1.1), UMAP preserves global structure comparatively worse (panel *b* and *d*).

3 Pseudotime estimation & random walks simulation

3.1 Theoretical concepts

Let $G = (V, E)$ be k nearest neighbor graph (k -NNG) (or any suitable connected graph over the data), where V and E stand for vertices (cells) and edges of the graph respectively. We have the corresponding *distance matrix* S

$$S_{ij} = \begin{cases} d(v_i, v_j) & (v_i, v_j) \in E \\ \infty & \text{otherwise} \end{cases} \quad (1)$$

where d is a distance function (we use Euclidean distance). For modeling diffusion on G we also consider the *transition matrix* T

$$T_{ij} = \begin{cases} f(S_{ij}) & (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where f is a suitable kernel function. Our default choice is the Gaussian kernel

$$f(x) = e^{-x^2/\varepsilon},$$

where ε is determined from the distribution of vertices as follows: for an edge $(v_i, v_j) \in E$ $\varepsilon = \text{median}\{d(v_i, v_k)^2, (v_i, v_k) \in E\}$. For convenience we define the *transition probability matrix* P

$$P = D^{-1}T \tag{3}$$

where D is a diagonal matrix of row-sums of T .

We will calculate the *pseudotime* of each vertex of G in the terms of expected length of random walks until a vertex is reached.

Definition 1 (Random walk). *We define a random walk (of possibly infinite length) on $G = (V, E)$ as a series v^0, v^1, \dots of vertices in G . Each v^{i+1} is chosen from the set $\{v_j : \{v_l, v_j\} \in E, v_l = v^i\}$ with probability $p_{v_j} = P_{lj}$ (P defined in Equation 3).*

If we fix the starting point v_0 of random walks we can define the *pseudotime* as one of the following quantities.

Definition 2 (Pseudotime). *Let $v_0 \in V$ be fixed. We define pseudotime $\tau_{v_0}(v)$ for each $v \in V$ as either **expected hitting time** (the expected length of a random walk sequence starting in v_0 before it reaches v) or **expected hitting distance** (the expected distance traveled by a random walk starting in v_0 before it reaches v). For both definitions it holds that $\tau_{v_0}(v_0) = 0$.*

The *expected hitting time* is calculated (for $v_j \neq v_0$) as follows (e.g. in ?):

$$\tau_{v_0}(v_j) = \sum_{v_i : \{v_i, v_j\} \in E} P_{ij}(\tau_{v_0}(v_i) + 1) \text{ if } v_j \neq v_0, \quad \tau_{v_0}(v_0) = 0$$

The generalization to *expected hitting distance* is straightforward:

$$\tau_{v_0}(v_j) = \sum_{v_i : \{v_i, v_j\} \in E} P_{ij}(\tau_{v_0}(v_i) + S_{ij}) \text{ if } v_j \neq v_0, \quad \tau_{v_0}(v_0) = 0$$

Rewriting this in matrix form (with components corresponding to v_0 dropped), we get

$$\boldsymbol{\tau} = P\boldsymbol{\tau} + (P \odot S)\mathbf{1}$$

where \odot denotes the Hadamard (element-wise) product and $\mathbf{1}$ is a vector of ones. We can solve for $\boldsymbol{\tau}$ the equation

$$(I - P)\boldsymbol{\tau} = (P \odot S)\mathbf{1} \tag{4}$$

where I denotes the identity matrix. Note that if T is symmetric (which is by default enforced by setting $T_{ij} = \max(T_{ij}, T_{ji})$) the Equation 4 can be rewritten as $D^{1/2}(I - P)D^{-1/2}D^{1/2}\boldsymbol{\tau} = D^{1/2}(P \odot S)\mathbf{1}$, where $D^{1/2}(I - P)D^{-1/2}$ is a symmetric matrix. The Equation 4 is a sparse system which can be efficiently solved by iterative methods, such as the (bi)conjugate gradient method, even for graphs with millions of vertices. This allows to keep the resolution of pseudotime on the single-cell level.

Once pseudotime is estimated, we can use it to simulate finite random walks. We do it by creating a directed acyclic graph (DAG) $G' = (V, E')$, where $E' = \{(v_i, v_j) \in E, \tau(v_j) > \tau(v_i)\}$. Random walks on a DAG are necessarily finite, and their final vertices are suitable candidates of developmental endpoints. We simulate large number of walks on G' and record the endpoints for further analysis (e.g. on Figure 2 in gray), where we study the possible trajectories (represented by sets of spatially coherent random walks) by which the endpoints can be reached.

The main motivation for introducing the *expected hitting distance* was our need for robust pseudotime estimation, even for the cases of significant accumulation of cells at some stages of development. The *expected hitting time* tends to rise quickly within compact, numerically large populations, and this would yield undesirable points of attraction when simulating developmental pathways in our downstream analysis. To illustrate this issue, we took the up-sampled *dyntoy* dataset (see subsection 1.1 and Figure 2) and set the point closest to the centroid of population $M4$ as v_0 . If pseudotime is calculated as *expected hitting time*, population $M10$ forms a sink which ‘traps’ any random walk entering it (see the color gradient on Figure Figure 2c). In contrast, pseudotime based on *expected hitting distance* allows random walks to pass through population $M10$ (Figure 2d). Upon inspection of Figure 2b, we see that passing through $M10$ would be equivalent to going back in time when using *expected hitting time* formulation.

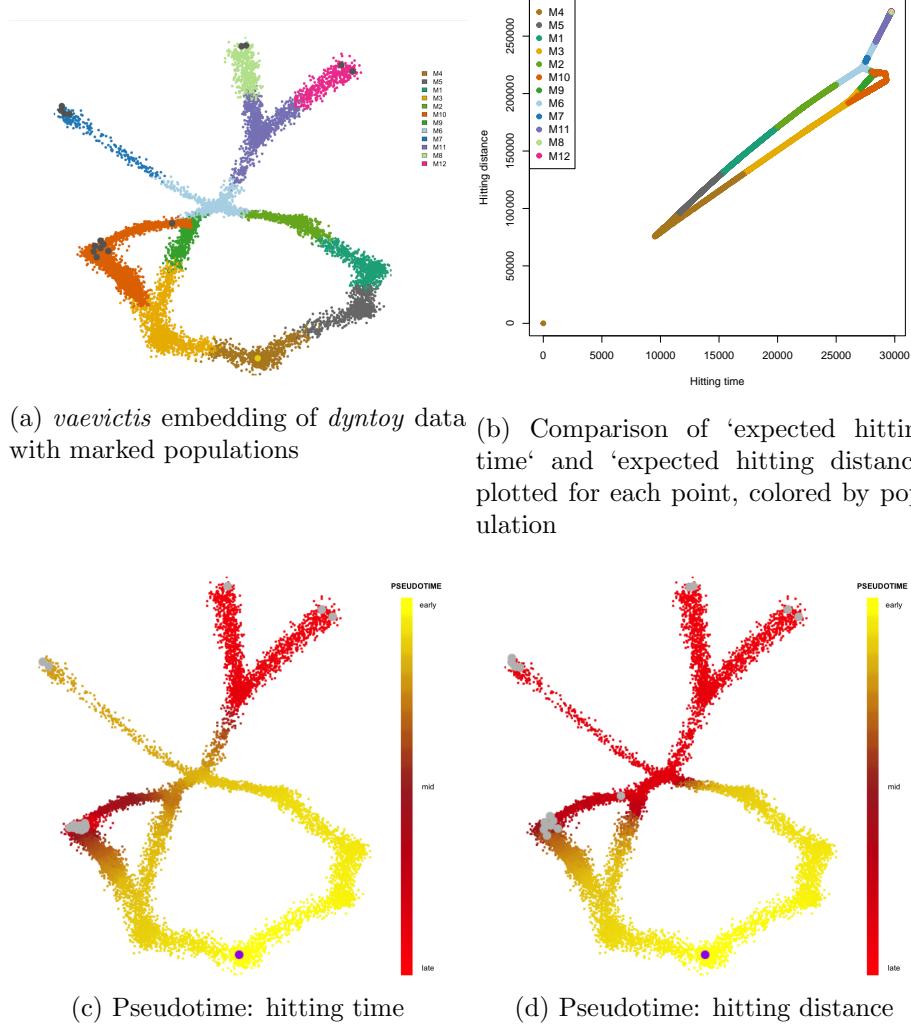


Figure 2: Pseudotime calculation strategies

3.2 On numerical aspects of the solution

One of the crucial advantages of describing *pseudotime* as a solution of a linear system $Ax = b$ (Equation 4) is the possibility to measure the accuracy of our solution as the relative error $\frac{\|Ax-b\|}{\|b\|}$. Although iterative methods can solve the system (provided the

solution exists) to any precision (limited by machine precision of floating point arithmetic), in practice the desired precision might not be possible to reach (e.g. because of an ill-conditioned system).

Although this happened rarely in our experiments, it is an issue worthy of attention. In most cases, non-convergence can be rectified by changing the kernel function (for instance, Gaussian kernel seems to work better than exponential kernel on data from flow cytometry) or adjusting the transformation of the data (an *asinh* transformation is usually a sensible choice).

To summarize, the reported error of the numerical solution for pseudotime values offers an important diagnostic tool in pseudotime estimation.

4 Topological clustering of random walks

Our approach to clustering random walks is based on the idea presented in ?. Detailed descriptions of relevant mathematical concepts and tools (algebraic topology, persistent homology) can be found in ??.

4.1 Basic notions of simplicial homology

For convenience, we define basic terms used throughout this document here.

Definition 3 (p -simplex). *We define a p -simplex as a convex hull of $p+1$ affinely independent points.*

A p -simplex σ is a point for $p = 0$, a line segment for $p = 1$, a triangle for $p = 2$, a tetrahedron for $p = 3$, etc. We use the notation $\sigma = [v_0, v_1, \dots, v_p]$ where v_i are vertices.

*The convex hull of a nonempty subset of vertices is called a **face** of the simplex.*

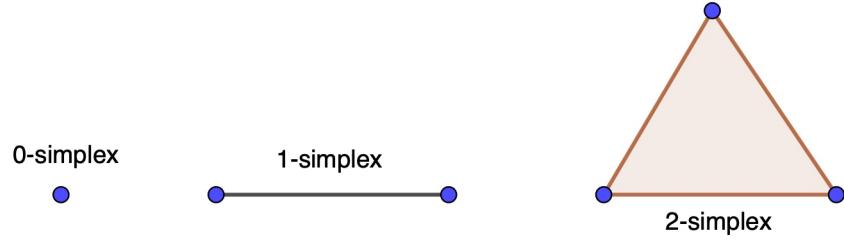


Figure 3: Simplices

Definition 4 (Simplicial complex). *A simplicial complex K is a set of simplices that satisfies the following conditions:*

1. Every face of a simplex from K is also in K .
2. The non-empty intersection of any two simplices $\sigma_1, \sigma_2 \in K$ is a face of both σ_1 and σ_2 .

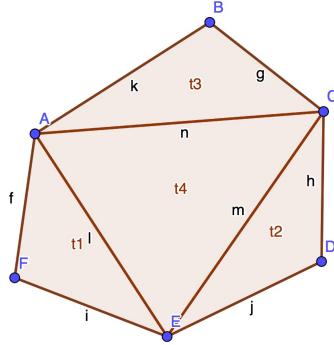


Figure 4: Simplicial complex

Definition 5 (p -chain group). *For a simplicial complex K , the group $C_p(K)$ of p -chains of K is defined as a formal linear combination of p -simplices:*

$$C_p(K) = \left\{ \sum_i m_i \sigma_i \mid m_i \in \mathbb{Z}/2\mathbb{Z} \right\},$$

where σ_i are p -simplices of K .

Note that the coefficients $m_i \in \mathbb{Z}/2\mathbb{Z}$. In general, these coefficients may be from any ring.

This choice simplifies many things (e.g. no issues with orientation arise, as $-1 = 1$), but we lose some descriptive power.

Definition 6 (Boundary). The **boundary homomorphism** $\partial_p : C_p(K) \rightarrow C_{p-1}(K)$ is defined simplex-wise for a simplex $\sigma = [v_0, v_1, \dots, v_p]$ as

$$\partial\sigma = \sum_{j=0}^p [v_0, \dots, v_{j-1}, \hat{v}_j, v_{j+1}, \dots, v_p],$$

where \hat{v}_j indicates deletion of vertex v_j .

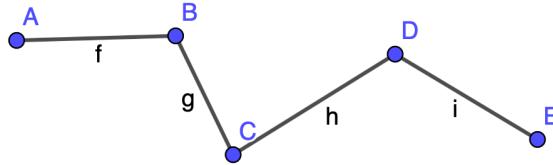


Figure 5: 1-chain $c = f + g + h + i = [A, B] + [B, C] + [C, D] + [D, E]$

$$\begin{aligned}\partial c &= \partial f + \partial g + \partial h + \partial i = \partial[A, B] + \partial[B, C] + \partial[C, D] + \partial[D, E] \\ &= A + B + B + C + C + D + D + E = A + E\end{aligned}$$

(coefficients in $\mathbb{Z}/2\mathbb{Z}$)

The boundary operator can be expressed as a matrix

$$\partial_{i,j} = \begin{cases} 1, & \text{if } \sigma_i \text{ is a co-dimension 1 face of } \sigma_j \\ 0, & \text{otherwise} \end{cases}$$

Definition 7 (Reduced (boundary) matrix). Let R be a matrix with elements in $\mathbb{Z}/2\mathbb{Z}$ and $\text{Low}(j)$ be the row index of the lowest 1 in the column j of the matrix R and $\text{Low}(j)$

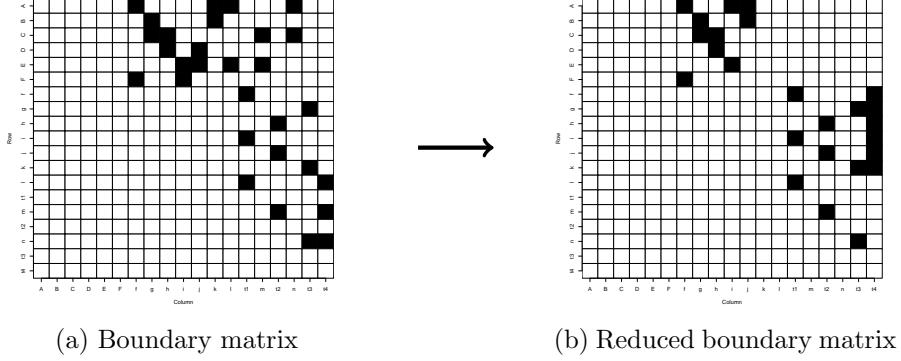


Figure 6: Boundary matrix of simplicial complex from Figure 4, 1s are marked as black squares

*undefined on zero columns. We say that matrix R is **reduced** if for any two non-zero columns i, j , $i \neq j$ $\text{Low}(i) \neq \text{Low}(j)$. See Figure 6.*

Matrix reduction can be performed by efficient algorithms and is essential for calculation of persistence (Definition 11). Matrix reduction corresponds to left-to-right column addition of columns representing boundaries of the same dimension, hence the columns of *reduced boundary matrix* correspond to boundaries of simplicial complexes (sums of boundaries of some p -simplices).

Definition 8 (Boundaries & Cycles). *We call*

p -boundaries: *the group $B_p = \text{Im } \partial_{p+1}$*

p -cycles: *the group $Z_p = \text{Ker } \partial_p$*

p -th homology group¹: $H_p = Z_p / B_p$

Definition 9 (Filtration). **Filtration** is a sequence of simplicial complexes that are ordered by inclusion. Hence, if K_1 and K_2 are complexes where K_1 appears before K_2 in the filtration,

¹Note that by fundamental lemma of homology $\partial_p \partial_{p+1} c = 0$ holds for any $p+1$ -chain c and consequently $B_p \subset Z_p$.

then $K_1 \leq K_2$: K_1 is a sub-complex of K_2 .

By **filtration of a simplicial complex K** we mean a sequence

$$\emptyset = K_0 \subseteq K_1 \subseteq K_2 \subseteq \dots \subseteq K_n = K$$

These inclusions induce a sequence of homology groups connected by homomorphisms

$$f_p^{i,j} : H_p(K_i) \rightarrow H_p(K_j), \quad i \leq j$$

Filtration thus gives a sequence of homology groups

$$0 = H_p(K_0) \rightarrow H_p(K_1) \rightarrow H_p(K_2) \rightarrow \dots \rightarrow H_p(K_n) = H_p(K).$$

Definition 10 (Filtration induced by a monotonic function). *Let K be a simplicial complex and $f : K \rightarrow \mathbb{R}$ a monotonic function on K , that is $f(\sigma) \leq f(\tau)$ whenever σ is a face τ . Let $a_1 < a_2 \dots < a_n$ be the values of function f on the simplices in K and $a_0 = -\infty$, then the sequence $K_i = f^{-1}(-\infty, a_i]$ is a **filtration of K induced by f** . We call a_i **filtration values**.*

Definition 11 (Persistence and essential classes). *Given a filtration, p -th persistent homology groups are defined as $H_p^{i,j} := \text{Im } f_p^{i,j}$, for $0 \leq i \leq j \leq n$. (i.e. $H_p^{i,j} = Z_p(K_i)/(B_p(K_j) \cap Z_p(K_i))$).*

We say that a homology class $\gamma \in H_p(K_i)$ is **born** at K_i if $\gamma \notin H_p^{i-1,i}$. Furthermore we say, that a class γ born at K_i **dies** entering K_j if $f_p^{i,j-1}(\gamma) \notin H_p^{i-1,j-1}$ but $f_p^{i,j}(\gamma) \in H_p^{i-1,j}$.

If the filtration is induced by a monotonic function we define the **persistence** of class γ

which was born at K_i and died at K_j as $\text{pers}(\gamma) = a_j - a_i$. Sometimes it is convenient to consider **index persistence** = $j - i$.

Throughout the text we will refer to the subcomplexes or corresponding filtration values where a homology class is born or where it dies as **birth** and **death** of the class, respectively.

We say that a p -cycle c represents an **essential class** in the simplicial complex K iff there is no $(p + 1)$ -chain $k' \in C_{p+1}(K)$ such that $c = \partial k'$. In other words, essential classes are persistent homology classes which never die.

If we consider a *reduced boundary matrix* with columns and rows ordered by filtration, then every non-zero column j corresponds to a *death* of a homology class. Moreover, this class was *born* at $K_{\text{Low}(j)}$, thus the *index persistence* of this homology class is $j - \text{Low}(j)$.

4.2 Triangulation of a point cloud & random walks

The triangulation of a high-dimensional point cloud is a computationally intensive task. The complexity of building triangulations used in lower-dimensional problems (such as *De-launay triangulation*) grows exponentially with the dimension, while other triangulations, better suited for high-dimensional problems (like the *Vietoris-Rips complex*) can become very large if we need sufficient cover of the space. We decided, therefore, to use the *witness complex* (?), implemented in ?. *Witness complex* is built from a smaller set of landmark points and uses the original point cloud to witness the existence of connections between points. In our implementation we first cluster the points into a large number of clusters (625 by default) by k -means clustering, and use the centroids of the clusters as landmark points. Obviously, the vertices of *witness complex* differ from the vertices in G' where the random walks were simulated. We need to triangulate the random walks as well. This procedure is straightforward: we simply contract the vertices to the corresponding cluster

and remove loops from the new contracted random walks. Thus, after this *triangulation* step, each random walk corresponds to a connected 1-chain in the *witness complex*.

A witness complex can be built up to a preset dimension of simplices. Since we need only 2-simplices to classify 1-chains, we limit the dimension to 2, and effectively create a 2-skeleton. The filtration values of simplices are then calculated the same way as they are in alpha complex filtration (?).

The choice of landmark points (clusters) can limit the resolution of the interactive aggregation of random walks. We did not encounter such an issue in practice and the simplicial complex built with default parameters always offered rich enough structure to achieve the desired level of detail. Potential issues can however be rectified by increasing the number of clusters or by introducing a custom clustering vector. Of note the triangulation of random walks takes place during the interactive session and the filtration can be recalculated independently of all other modules.

4.2.1 Data denoising

Technical noise is always present in single-cell data. The presence of unwanted data points in sparse regions which are used for characterization 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 to help capture the topology correctly, we apply the denoising technique, suggested in ?.

Taking advantage of a previously built k -NNG, we replace the coordinates of each data point with the averaged coordinates of some l of its nearest neighbors. The denoised version of our expression matrix is used to construct the witness complex.

4.3 Cycle representation

In order to classify random walks with respect to homology classes, we need a suitable representation of their dissimilarity. The basic idea of this approach is outlined in ?. We, however, improve on the cited results by allowing for *essential classes*, and consequently the classification of random walks with different endpoints (e.g. developmental branching).

Every random walk can be understood as a connected 1-chain with at least one vertex in its boundary (the cell-of-origin vertex v_0) common to all other random walks. Consider first a set of 1-chains $\{c^0, c^1, \dots, c^N\}$ with a common boundary (i.e. origins and endpoints of all corresponding triangulated random walks are the same). We can now fix a reference 1-chain (e.g. c^0) and consider 1-cycles $\{c^0 + c^1, c^0 + c^2, \dots, c^0 + c^N\}$. If all cycles in this set are boundaries of some 2-chain, they can be uniquely expressed as linear combinations (over $\mathbb{Z}/2\mathbb{Z}$) of columns of the *reduced boundary matrix* (as described in ?). However, if some of these *cycles* represent an *essential class*, we need a more elaborate solution, implemented in Algorithm 1. If we encounter a *cycle* which is not a *boundary* during the calculation of the representation, we simply form an auxiliary *cycle* representing the "essential part" and add it as another column into the *reduced boundary matrix*. The filtration value corresponding to the *death* of this class is ∞ , but for the purpose of visualization on the persistent diagram we assign arbitrary values to all *essential classes*, so that they have the same *death/birth* ratio and the *death* filtration value both higher than any non-essential classes.

If we fix a reference 1-chain c^0 , we can uniquely represent this way any 1-chain c^i with the same boundary by expressing the *cycle* $c^0 + c^i$ by a set of columns of (possibly updated) *reduced boundary matrix*. Note that $c^0 + c^0$ is an empty 1-chain. This representation will be used in subsection 4.4 to hierarchically cluster random walks.

What remains to be discussed is the case of classification of random walks with different endpoints. We implement two options. The first option is to simply choose the endpoint

with highest *pseudotime*, set it as the endpoint of all random walks in question, and connect the other endpoint(s) to it by the shortest path within the simplicial complex. This approach is convenient in the case of closely clustered endpoints that represent the same developmental fate. The second option is to add a 1-simplex between selected endpoints. Since there were no 1-simplices connecting these endpoints (otherwise the random walks in question would end in the endpoint with the higher *pseudotime*), this will effectively create one or more new *essential classes*. Random walks differing by an *essential class* will be split into major branches at the root of the dendrogram (see subsection 4.4).

We close this subsection by a detailed description of the algorithms used and by providing a proof of correctness. Since we use the field \mathbb{Z}_2 as the coefficients ring, we can understand simplicial complexes (and consequently p -chains) as sets of simplices. To simplify further exposition we introduce following notation and arithmetic:

Definition 12 (Notation and arithmetic of p -chains & Low^{-1}). *Given a simplicial complex $K = \{\sigma_1, \sigma_2, \dots, \sigma_N\}$ and a p -chain $c = \{\sigma_{i_1}, \sigma_{i_2}, \dots, \sigma_{i_k}\}$, we define the **index representation** c_I of c as the tuple $c_I = (i_1, i_2, \dots, i_K)$.*

For any two p -chains a_I, b_I we define (in agreement with the arithmetic over \mathbb{Z}_2)

$$a_I + b_I = a_I \Delta b_I \text{ (the symmetric difference of the 2 sets).}$$

*By a slight abuse of notation, we will define the **index representation of a set of p -chains** $C = \{c^1, \dots, c^k\}$ as $C_I := (c^1 + c^2 + \dots + c^k)_I$.*

We will also assume that the indices in the index representation tuple are always ordered with respect to the filtration in descending order, i.e. for any p -chain c , $c_I[1]$ has the highest filtration value among all simplices forming the p -chain.

For the reduced boundary matrix R let us further define the map Low^{-1} :

$\text{Low}^{-1}(i) = j$ if i is the highest row index of a non-zero element in a column j of matrix R
and $\text{Low}^{-1}(i) = -1$ if there is no such column.

Algorithm 1 One cycle representation

Input: c_I cycle, R reduced boundary matrix with corresponding Low^{-1} function

Output: r representation (tuple of column indices) of the cycle c_I as columns of (updated)
boundary matrix R , updated boundary matrix R

```

procedure GET REPRESENTATION_OF_CYCLE( $c_I, R$ )
     $r \leftarrow \emptyset$ 
     $e_I \leftarrow \emptyset$ 
     $b_I \leftarrow \emptyset$ 
     $N \leftarrow$  number of columns of  $R$ 
    while  $c_I \neq \emptyset$  do
         $j \leftarrow \text{Low}^{-1}(c_I[1])$ 
        if  $j > 0$  then
             $r \leftarrow r \cup j$ 
             $c_I \leftarrow c_I + R[, j]_I$ 
             $b_I \leftarrow b_I + R[, j]_I$ 
             $e_I \leftarrow e_I$ 
        else
             $e_I \leftarrow e_I + c_I[1]$ 
             $c_I \leftarrow c_I + c_I[1]$ 
             $r \leftarrow r$ 
        end if
    end while
    if  $e_I \neq \emptyset$  then
         $R \leftarrow [R, e_I]$  (adding a column with 1s at positions  $e_I$  and 0s elsewhere)
         $\text{Low}^{-1} \leftarrow$  update  $\text{Low}^{-1}$  function with respect to the new  $R$ .
         $r \leftarrow r \cup (N + 1)$ 
    end if
    return  $r, R$ 
end procedure

```

Algorithm 2 Cycles representation

Input: $C = \{c_I^1, c_I^2, \dots, c_I^L\}$ set of cycles, R reduced boundary matrix with corresponding Low^{-1} function

Output: Representation $= \{r^1, r^2, \dots, r^L\}$ representations (tuples of column indices of (updated) boundary matrix R) of cycles C

for $i \in 1 \dots L$ **do**

$r^i, R \leftarrow \text{GET_REPRESENTATION_OF_CYCLE}(c_I^i, R)$

end for

return Representation

Proposition 4.1: Correctness of algorithms 1 and 2

- i) Algorithm 1 finds unique representation for any p -cycle c_I in maximum of $\max(c_I)$ steps in the form $c_I = \hat{b}_I + \hat{e}_I$, where \hat{b}_I is a p -boundary and \hat{e}_I is a p -cycle representing essential class(es).
- ii) Matrix R is updated in the i -th step of algorithm 2 iff $\forall j : 0 \leq j < i$ the cycle $c_I^i + c_I^j$ represents an essential class ($c_I^0 := \emptyset$).

Proof. Let us denote the original reduced boundary matrix R_0 and the number of columns of R_0 , N_0 . Notice that for any e_I constructed by algorithm 1 there was no column with lowest non-zero element at $\max(e_I)$ before the update, hence the matrix R is always *reduced* and the columns remain linearly independent after the update(s). Consequently, every element in their span has a unique representation.

- (i) Since $c_I[1] = \max(c_I)$ and if $j = \text{Low}^{-1}(c_I[1]) > 0$ then $\max(R[j], c_I) = \max(c_I)$, the value $\max(c_I)$ is reduced at least by 1 in every iteration.

By construction $\emptyset = c_I + b_I + e_I$, hence e_I constructed by algorithm 1 is a cycle. If $e_I \neq \emptyset$, the matrix R is updated by a column containing a cycle representing an essential class. All cycles not representing an essential class can be expressed as a unique sum of boundaries and hence e_I would be \emptyset .

Consider a representation r corresponding to the cycle c_I . Let us decompose r into

disjoint sets $r = r_b \cup r_e$, such that $\min(r_e) > N_0$, $\max(r_b) \leq N_0$ and define $\hat{e}_I := R[r_e]_I$, $\hat{b}_I := R[r_b]_I$.

(ii) If there exists c_I^j , $0 \leq j < i$ such that the cycle $c_I^i + c_I^j$ does not represent an essential class, then by definition there exists a p -boundary b such that $c_I^i = c_I^j + b$. Since c_I^j has a representation within R and b can be expressed as sum of columns of R_0 , no matrix update is necessary. Conversely, if $\forall 0 \leq j < i$ the cycle $c_I^i + c_I^j$ represents an essential class, then c_I^i cannot be expressed in the terms of the columns of matrix R and e_I constructed by algorithm 1 is non-empty. \square

4.4 Topological hierarchical clustering of random walks

Consider a set of L random walks and corresponding 1-chains $\{c^0, c^1, c^2, \dots, c^L\}$. Let us select c^0 as a reference 1-chain (thus having trivial representation) and the representations obtained by algorithm 2 be $\{\emptyset, r^1, r^2, \dots, r^L\}$. Each r^i corresponds to a set of homological classes each with its own *birth* and *death* filtration values. If we set a threshold on the *death* value and delete all homological classes which *died* before this point from the representation, a natural hierarchical clustering stems from the following idea. Initially, each leaf of the dendrogram contains all random walks with the same representation, then every time a homological class dies as we are increasing the threshold, two leaves/branches of the dendrogram may merge. As soon as all ² homological classes *die*, the dendrogram is complete.

In practice, we proceed differently. We first choose (using an interactive persistence diagram) homology classes we deem significant based on their persistence. Then we order the homology classes present in any representation in decreasing order with respect to

²In practice we set threshold on column indices of corresponding (updated) *reduced boundary matrix*. Thus even *essential classes die* after a finite number of steps

their *death* value. Finally, we split the random walks into two branches³ based on the presence/absence of the homology class with highest *death* value in their representation. The class with the highest *death* value is dropped and we re-iterate the process on each of the two branches.

5 Connectome

First, a multigraph is created from vertices corresponding to clusters. One directed edge is added between any pair of vertices per each traversal by a random walk. Self-loops are deleted. Then, a new graph is plotted where the edges with the same direction between same vertices are merged, setting edge weights to the number of edges before merging (by default, only the majority-direction edge is kept for any pair of vertices). Vertices of the resulting graph are plotted as pie charts of cell populations present in each cluster. The graph layout employs the same 2D representation(s) as *tviblindi* GUI to facilitate the interpretation. Terminal nodes are automatically detected as vertices with fewer outgoing than incoming edges. The widths of edges represent their weights.

Although connectome offers only a low resolution overview of the dataset, the result is informative even for very complex topologies (Figure 7). On this dataset connectome was run as `Connectome(tv1,qq=0.8)` i.e. only arrows with width above the 80% percentile were plotted.

6 Graphical user interface (GUI)

An essential part of *tviblindi* is GUI which enables the on-demand aggregation of random walks and real-time resolution adjustments. In this section we describe the main features

³If one branch is empty, we drop the class and re-iterate

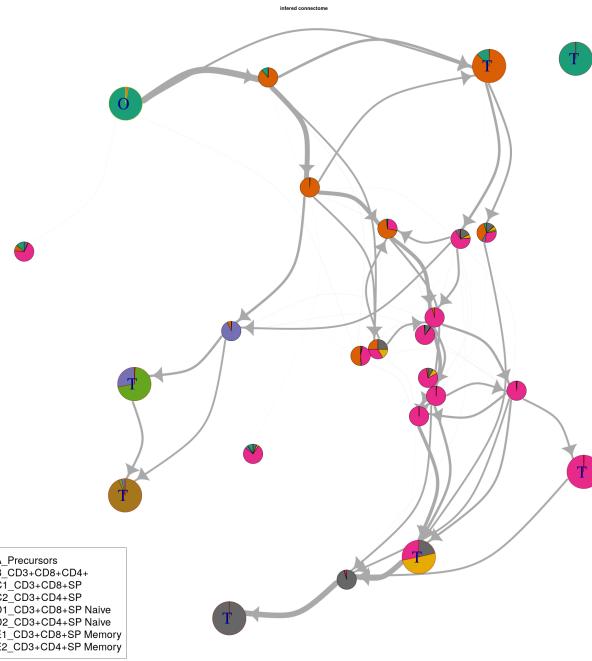


Figure 7: Connectome applied on the human thymus dataset (subsection 1.2)

of this GUI and the analysis process. (See also supplementary videos.) In this text we will refer to the annotated screenshots of the GUI on pages 29 and 30 of this document by numbers in circles, e.g. ①, and squares, e.g. [1], respectively. Please consult section 7 or the documentation of the *twiblindi* package for more details on any function mentioned in the text (function names are typeset as **verbatim**). Note that pop-up tooltips are displayed for button in the GUI when running.

The user usually starts with adjusting visualization and selecting a path model. As every run of the function `DimRed` adds a 2D layout to the *twiblindi* analysis object, we can choose the most appropriate visualization in the drop-down menu ①. Menu ② allows the user to choose the path model (pseudotime and random walks model, for different choice of cell-of-origin, different parameter choices for *pseudotime* calculation etc.). The 2-dimensional layout plot is optimized for large datasets (millions of events), but button

(9) can increase point size when needed. Labels of points (7) serve mainly for visualizing labeled populations on the plot invoked by (8), and for tracking population frequencies along the trajectory (21) second pane).

This first step of the analysis itself is the choice of endpoints (3). The user selects some black dots in the layout via rectangular selection at once, or iteratively by pressing (4) after each selection. The selected endpoints are listed (as row indices in the data matrix) together with the number of random walks which end in a particular endpoint. Pressing (4) moves the endpoints in the current rectangular selection (*Selected terminal nodes* frame) to *Terminal nodes marked for further analysis*.

Once the user is satisfied with the endpoints selection, they can calculate their representation by pressing (5). Checkbox (6) switches between the two options for the multiple endpoints treatment (see subsection 4.3).

At this point the endpoints of interest are selected and the representation of all random walks ending in any of these endpoints are calculated. The user can then switch to the pane *Homology classes by persistence selection* [1] and select homology classes they deem significant, based on their persistence. By default, the y axis of the diagram corresponds to the *death/birth* ratio⁴. This ratio describes the significance of the hole (sparse region) corresponding to the homology class. The larger this value, the sparser the interior of the hole relative to the density of its boundary, and the higher the significance for the random walk classification. The x axis corresponds to the *death* value of each homology class plotted: this value corresponds to the *size* of the corresponding hole, and to the height of the branching on the dendrogram [4]/(13). For the artificial data used on the screenshots, the situation is clear: there are two *essential classes*, one corresponding to the larger hole in

⁴This gives better resolution for homology classes with large differences in size than the more standard *death - birth*. This can be changed using checkbox [3], resulting in a common persistence diagram.

the data and one created by the selection of two disparate developmental fates⁵. Another significant homology class corresponds to the smaller hole in the data.

The selection process is equivalent to the selection of endpoints, and by pressing button [2] the dendrogram [4] is updated.

Using the dendrogram (13), the user can select leaves/branches to aggregate random walks into trajectories. Two trajectories can be built at the same time by selecting a trajectory group A or B (11). The selection process is the same as before: pressing the "+" button (14) adds selected walks into the selection into the active trajectory, the "fire" button resets the trajectory. Note the slider (23), which adjusts the complexity of the dendrogram by setting a threshold on the minimal fraction of random walks for a node to be shown. Nodes below this threshold are merged.

The user can adjust the level of detail by selecting significant homology classes in [1]. In some cases it is more convenient to focus on a smaller subset of random walks. This can be achieved by pressing the button (17): this makes the analysis focus only on random walks in the active trajectory and the user can analyze its topology in detail, e.g. if inspecting finer branching processes. To go back to the entire set of random walks, the user can press the "fire" button (4) and start over with selection of endpoints.

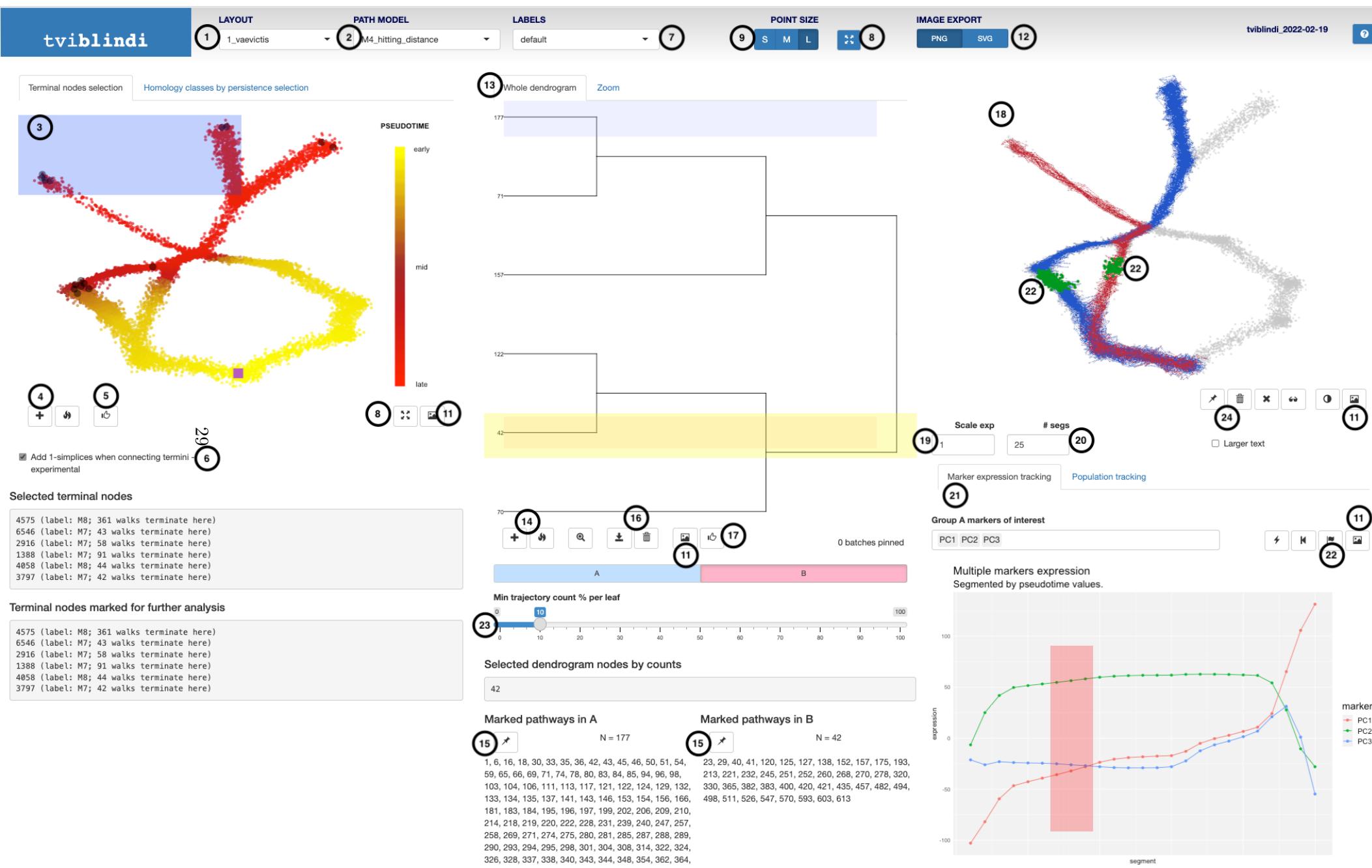
Selected trajectories are visualized in a 2-dimensional embedding (layout) in pane (18). Change in signal from specific markers along a trajectory can be tracked (21). The development of those markers is shown as a line plot connecting the average values of a specific marker (or points on specific random walks, see below [5]) within a pseudotime segment. By default, the pseudotime values are divided into uniform segments (each containing the same number of points) whose number can be set changed in (20). To focus on either early or late stages of the development, adjust the pseudotemporal scale using (19): the

⁵Note that *essential classes* are infinite from the point of view of homology and their *death* values are chosen arbitrarily for visualization purposes

higher the value, the more resolution is put on early stages, and vice-versa. If the user is interested in *where* in the layout a certain part of development is projected, they can select the segments of interest and press (22). If multiple markers are tracked, the points of all trajectories in the selected segments are highlighted. If only one marker is tracked [5], we can see individual random walks and highlight only points from selected walks. If the user wants to manually clean a selected trajectory based on the evolution of the signal levels from a specific marker, they can manually select some random walks and remove them from the trajectory by pressing the "flash" button near [6] "flag".

It is often useful to export the trajectories/selected points for further analysis. Highlighted points can be pinned by pressing the "pin" button (24) (the "thrash" deletes all pinned points). Trajectories can be pinned using (15). When pressing the "pin" button, the user is prompted to name the trajectory or group of points. Pinned results are stored as list slots in the *tviblindi* object or can be exported as an extended FCS file by pressing the "download" button (16) (the "trash bin" button clears pinned trajectories). Once pressed, the user is prompted to name the exported FCS file and the results are stored either in a new FCS file built from the data matrix or added to an existing FCS file (if provided: see section section 7). The following is stored in the FCS file: the active 2D embedding, pseudotime values from the active "PATH MODEL", all sets of labels, all pinned trajectories and all pinned highlighted points.

Button (11) saves the corresponding figure as either a PNG or an SVG file (12).



LAYOUT

1_vaevictis

PATH MODEL

M4_hitting_distance

LABELS

default

POINT SIZE

S

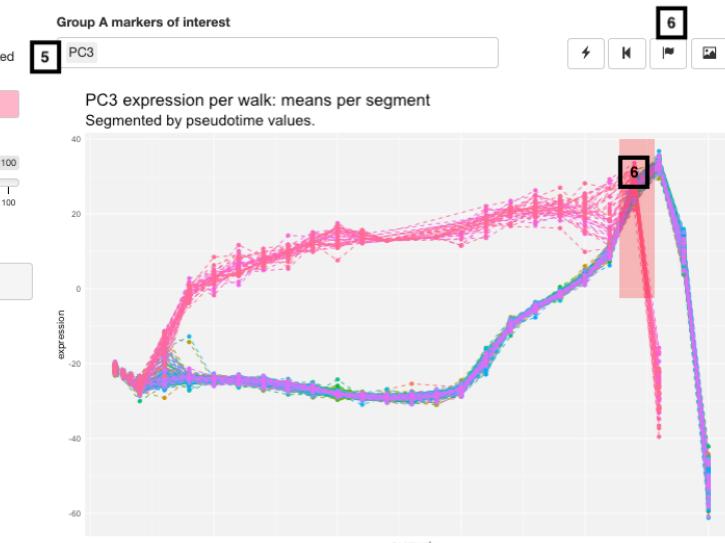
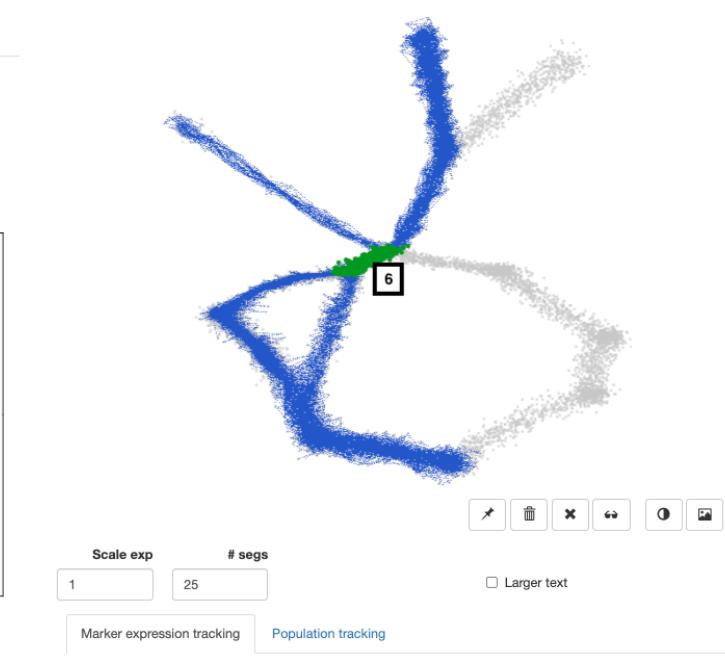
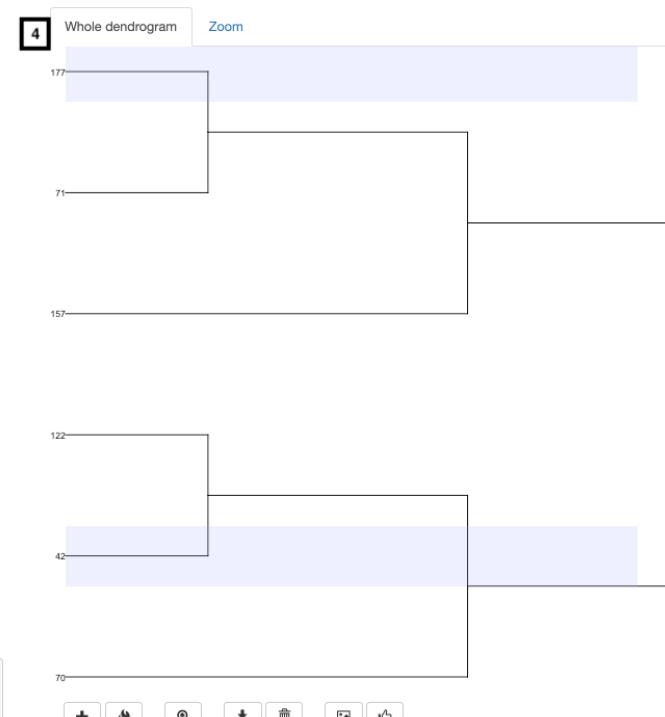
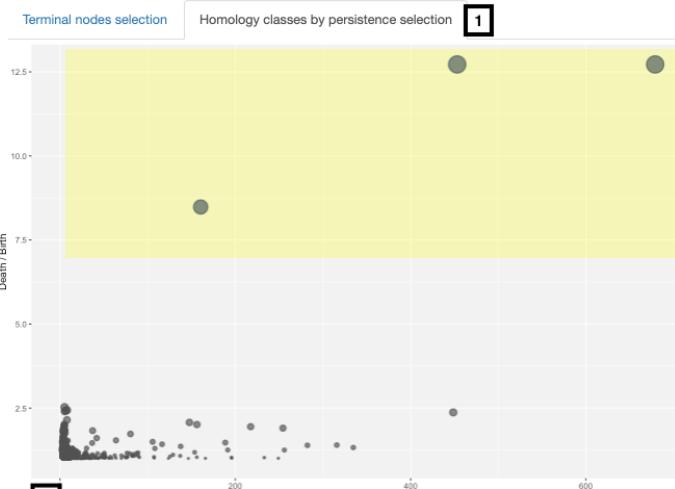
M

L

IMAGE EXPORT

PNG

SVG



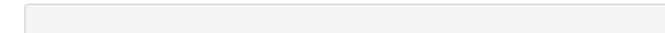
Selected homology classes

Dimension	Birth	Death	BirthSimplex	DeathSimplex
2193	1	18.92563	160.5068	553289
2208	1	53.37344	678.9843	626
2209	1	35.61915	453.1251	1196175
				1528081

Marked homology classes

Dimension	Birth	Death	BirthSimplex	DeathSimplex
2193	1	18.92563	160.5068	553289
2208	1	53.37344	678.9843	626
2209	1	35.61915	453.1251	1196175
				1528081

Selected dendrogram nodes by counts



Marked pathways in A

N = 219

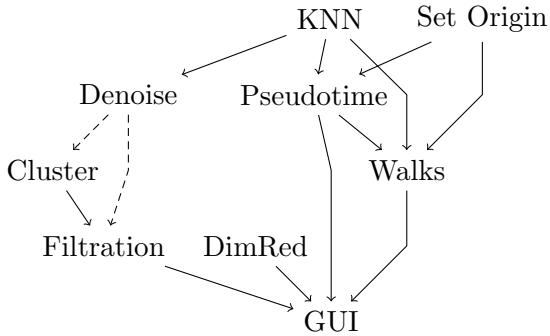
1, 6, 16, 18, 30, 33, 35, 36, 42, 43, 45, 46, 50, 51, 54, 59, 65, 66, 69, 71, 74, 78, 80, 83, 84, 85, 94, 96, 98, 103, 104, 106, 111, 113, 117, 121, 122, 124, 129, 132, 133, 134, 135, 137, 141, 143, 146, 153, 154, 156, 166, 181, 183, 184, 195, 196, 197, 199, 202, 206, 209, 210, 214, 218, 219, 220, 222, 228, 231, 239, 240, 247, 257, 258, 269, 271, 274, 275, 280, 281, 285, 287, 288, 289, 290, 293, 294, 295, 298, 301, 304, 308, 314, 322, 324, 326, 328, 337, 338, 340, 343, 344, 348, 354, 362, 364,

Marked pathways in B

N = 42

7 Running the analysis

The analytical pipeline consists of several steps which are not completely interdependent and can be (re)run in a different order if necessary (see following graph of the dependence structure (a dashed connection shows that the denoising step is optional)).



The *R* code needed to run a generic *tviblindi* analysis is shown below.

```

tv1 <- tviblindi(
  data = d,          # matrix of row-wise data, with named columns
  labels = l,        # vector of string labels per each row of d
  fcs = p,          # path to FCS file if d comes from its expression matrix
  events_sel = i    # indices of rows of FCS file, if only a subset is used
)
Set_origin(
  tv1,
  label = o,        # name of progenitor population
  origin_name = n # path model name (new)
)
KNN(tv1)           # build a k-NN graph
Denoise(tv1)        # reduce noise before witness complex construction
  
```

```

Cluster(tv1) # k-means clustering by default: k = 625 clusters

Filtration(tv1)      # build witness complex

DimRed(tv1)          # dimensionality reduction

Pseudotime(
    tv1,
    origin_name = n # path model name (existing)
)

Walks(
    tv1,
    N = 5000,         # number of finite random walks to simulate
    origin_name = n # path model name (existing)
)

launch_shiny(tv1)   # launch GUI for interactive analysis

```

All functions starting with a capital letter are generic S3 methods and the documentation could be accessed via the class suffix, e.g. `?KNN.tviblindi`. The parameter `origin_name` is optional and serves to distinguish different choices of the cell-of-origin, pseudotime calculation and random walk simulation (different path models; see section 6). The first command deserves clarification. The `fcs`, `events_sel` parameters are optional and connect the `tviblindi` object to an existing FCS file to be ‘enhanced’ as a part of the analysis. If omitted, the GUI will create a new FCS file out of the data matrix if the user chooses to export an FCS file. Otherwise if these parameters are provided, analysis results will be added to an existing FCS file as additional columns. If only a subset of rows from the FCS file is used for the analysis, the parameter `events_sel` records the indices of these rows as an integer vector.

8 Performance evaluation

8.1 Sensitivity to hyperparameters

8.1.1 Number of nearest neighbors

The pseudotime calculation is robust with respect to the number of nearest neighbors considered. The *Spearman correlation coefficient* between pseudotime estimated using 10 nearest neighbors and 50 nearest neighbors was 0.9986367 for the data from subsection 1.2.

The choice of the number of nearest neighbors k used for *random walks* simulation has significant effect on the number of detected endpoints. Lower k leads to higher number of endpoints, while higher k tends to smooth the topology of the graph leading to more global view. Since the *random walks* simulation does not have any effect on the results of other modules (section 7), the user can experiment with this setting very efficiently. But even when larger k causes spurious connection between unrelated population of cells, the topological clustering of *random walks* allows for the separation of different trajectories.

8.1.2 Relaxation parameter for *witness complex* construction

The relaxation parameter `alpha2` (function `Filtration`) controls the size of the simplicial complex. Lower values of `alpha2` lead to smaller simplicial complex which might not give sufficient resolution for the downstream analysis. On the other hand, smaller simplicial complexes lead to significant speedup.

The default setting is very conservative and creates very large simplicial complex (roughly 1-2 orders of magnitude larger than necessary, according to our tests) while maintaining feasible running times (subsection 8.2).

8.2 Running times

tviblindi outperforms tested TI methods on medium-sized and large datasets (Table 1) in terms of running time. All methods were run in a dedicated Docker container with 20 CPUs and RAM limited to 200 GB.

Table 1: Running times (minutes). Evaluated on the dataset from subsection 1.2.

n cells	tviblindi	VIA (coarse)	VIA (fine)	Palantir	Monocle 3	STREAM	PAGA
50000	0.9	0.7	2.1	3.2	4.9	10.7	0.8
100000	1.7	1.4	2.8	6.5	20.4	20.8	2
200000	4.1	3.1	6.1	14	74	32.7	6
500000	9.7	11.25	19.7	33.4	X	62	20.4
All (1,182,80 2)	22	28.1	51.8	87.9	X	136.7	46.1

The current implementation of *tviblindi* has effectively linear complexity with respect to the number of events for datasets with less than 10^7 events Figure 8. There is no clear theoretical reason for the complexity to be super-linear and we are currently focusing on the implementation to pinpoint possible overhead.

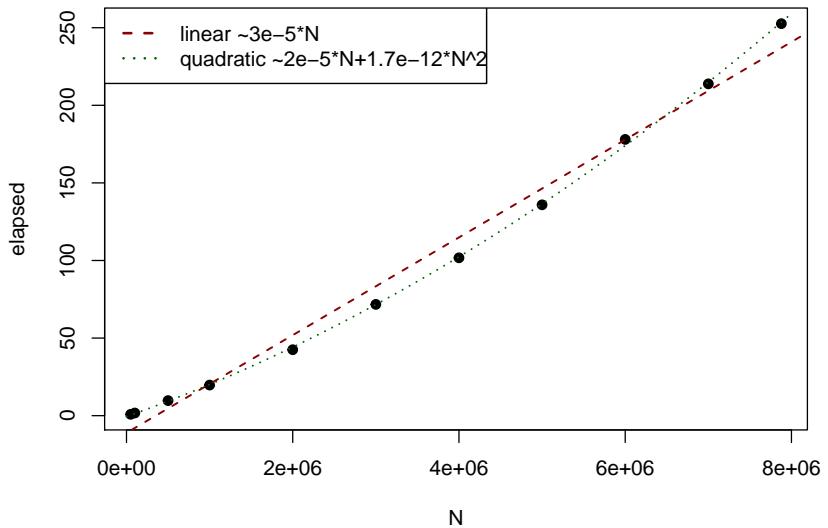


Figure 8: Running times of *tviblindi* with respect to the number of events (minutes).