CellTrails: Reconstruction, Visualization, and Analysis of Branching Trajectories from Single-cell Expression Data

Daniel C. Ellwanger // dcellwanger.dev[at]gmail.com

Package: 0.99.12 // Manual: 2018-06-16

Contents

Preface

This manual describes the practical use of the *CellTrails* implementation, an unsupervised algorithm for the *de novo* chronological ordering, visualization and analysis of single-cell expression data. *CellTrails* makes use of a geometrically motivated concept of lower-dimensional manifold learning, which exhibits a multitude of virtues that counteract intrinsic noise of single cell data caused by drop-outs, technical variance, and redundancy of predictive variables. *CellTrails* enables the reconstruction of branching trajectories and provides an intuitive graphical representation of expression patterns along all branches simultaneously. It allows the user to define and infer the expression dynamics of individual and multiple pathways towards distinct phenotypes.

We are pleased that you consider using *CellTrails* in your research. A detailed theoretical description of the algorithm and its application to biological uses has been published in:

Ellwanger, DC, M Scheibinger, RA Dumont, PG Barr-Gillespie, and S Heller. 2018. “Transcriptional Dynamics of Hair-Bundle Morphogenesis Revealed with Celltrails.” *Cell Reports* 23 (10): 2901–14.

Chapter 1

Prerequisites

1.1 Terminology

In the following, we use a general termonology to describe the biological data of interest. We analyze quantitative *expression values* (e.g., RT-qPCR Log2Ex, RNA-Seq log2 counts, usf.) of *features* (e.g., genes, transcripts, spike-in controls, usf.), which were obtained from individual *samples* (e.g., single cells).

1.2 Load Library

Before ready to use, the *CellTrails* libraries must be loaded into the *R* environment:

```r
library(CellTrails)
```

1.3 Third-party Software: yEd

We strongly recommend to download and install the graph visualization software *yEd* (http://www.yworks.com/products/yed). It provides great capabilities to perform planar embedding, visualization, and analysis of a trajectory graph produced by *CellTrails*.

1.4 Input: SingleCellExperiment

CellTrails organizes its data in an object of Bioconductor's *SingleCellExperiment* (Lun and Risso, 2017) class. It provides all attributes required for smooth and user-friendly data processing and analysis of single cell data and enables interoperability between packages. Please, refer to the *SingleCellExperiment* vignette for details.

1.4.1 Shape of Expression Data

CellTrails expects the expression data to be normalized and log-transformed; it is not required that features were filtered at this point. The expression data is expected to be available from the `logcounts` assay entry. If this entry is empty (check for its existence with function `assays`), the function `logcounts<-` can be used to store the log-normalized data in a *SingleCellExperiment* object.

If your expression data is not stored in an object of class *SingleCellExperiment*, we suggest to initiate an object from a numerical matrix composed of the log-normalized expression values; features should be listed in rows, and samples in columns (see *Example Datasets*).

1.4.2 Spike-in Controls

There is no need to remove spike-in controls from your *SingleCellExperiment* object. *CellTrails* automatically ignores spike-in controls for its analysis, if they were properly annotated in the object using the function `isSpike` (see #exdat).

1.5 Example Datasets

exSim

In this vignette, simulated data (with log-transformed Negative Binomial distributed expression counts) and real expression data are used to illustrate the functionality of the *CellTrails* package.

The first dataset, `exSim`, is composed of expression values of 15,000 features measured in 100 samples; 80 spike-in transcripts were added.

```r
# Create example expression data
# with 15,000 features and 100 samples
set.seed(1101)
emat <- simulate_exprs(n_features=15000, n_samples=100)

# Create SingleCellExperiment object
exSim <- SingleCellExperiment(assays=list(logcounts=emat))

# Annotate ERCC spike-ins
isSpike(exSim, "ERCC") <- 1:80
show(exSim)
```

```
## class: SingleCellExperiment
## dim: 15000 100
## metadata(0):
## assays(1): logcounts
## rownames(15000): feature_1 feature_2 ... feature_14999
##   feature_15000
## rowData names(0):
## colnames(100): sample_1 sample_2 ... sample_99 sample_100
## colData names(0):
## reducedDimNames(0):
## spikeNames(1): ERCC
```

exBundle

The second dataset, `exBundle`, contains transcript expression profiles of 183 genes expressed during sensory hair cell bundle maturation and function, which were quantified in the chicken utricle sensory epithelium at embryonic day 15 using multiplex RT-qPCR. Experimental metadata was generated during tissue preparation (cell origin) and cell sorting (uptake of FM1-43 dye indicating cell maturity). This data set is the foundation used for the development of *CellTrails*. If you use this dataset for your research, please cite Ellwanger et al. (2018).

```r
# Load bundle data
exBundle <- readRDS(system.file("exdata", "bundle.rds", package="CellTrails"))
```

Chapter 2

Overview

The following illustrates the typical sequence of steps performed during a *CellTrails* analysis and lists the available functions, respectively.

- Selection of trajectory features
  - `filterTrajFeaturesByDL`
  - `filterTrajFeaturesByCOV`
  - `filterTrajFeaturesByFF`
- Lower-dimensional manifold learning
  - `embedSamples`
  - `findSpectrum`
  - `latentSpace`
  - `plotManifold`
- Clustering
  - `findStates`
  - `states`
  - `plotStateSize`
  - `plotStateExpression`
- Determination of the trajectory topology
  - `connectStates`
  - `showTrajInfo`
  - `trajComponents`
  - `selectTrajectory`
  - `plotStateTrajectory`
- Chronologically ordering of samples
  - `fitTrajectory`
  - `plotTrajectoryFit`
- Trajectory visualization
  - `write.ygraphml`
  - `read.ygraphml`
  - `trajLayout`
  - `plotMap`
- Identification of paths (trails) on the trajectory
  - `landmarks`
  - `userLandmarks`
  - `addTrail`, `removeTrail`
  - `trailNames`
  - `trails`
  - `plotTrail`

- Inference of expression dynamics of trails
  - `fitDynamic`
  - `plotDynamic`
- Intra- and inter-trail expression dynamic comparison
  - `contrastTrailExpr`

By calling the function `showTrajInfo`, an informative overview of the data relevant for, or stored by *CellTrails* is printed. We suggest to use this function multiple times during a *CellTrails* analysis, as it provides useful insights into the analysis' progress.

```r
showTrajInfo(exBundle)
```

```
## [[ CellTrails ]]
## logcounts: 183 features, 1008 samples
## Pheno data:
##   sampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   phenoNames: "fm143" "origin" (2)
## Feature data:
##   featureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   latentSpace: none
##   states: none
## Trajectories: none
##   trajSampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   trajResiduals: MSE=NA
##   landmarks: none
##   trajLayout: none
## Trail data:
##   trailNames: none
```

The entries *logcounts* and *Feature data* correspond to the expression matrix and feature information provided and annotated by the user, respectively. *Pheno data* contains metainformation for each sample stored by the user and by *CellTrails*. The entries *Trajectory data*, *Trajectories* and *Trail data* denote *CellTrails* specific information and will be described in detail in the respective section of this handbook.

Chapter 3

Selection of Trajectory Features

An expression matrix from an high-throughput experiment contains hundreds of features that bear no or little information about a sample's progress through the trajectory of interest. These non-relevant features not only increase the computational runtime of the trajectory reconstruction, but they also impair the accuracy of downstream analysis results. *CellTrails* assumes that key features show high expression variability during a sample's progression along the biological process under study. Therefore, *CellTrails* enables to unbiasedly filter the informative features.

Please note that the following functions only indicates features used for dimensionality reduction, state detection and trajectory reconstruction; it does *not* remove features from the assay stored in the *SingleCellExperiment* object. Thus, all features are still available for *CellTrails* donwstream analyses capabilities, such as cluster analysis and inference of expression dynamics, as well as, for functions available from complementary analysis packages.

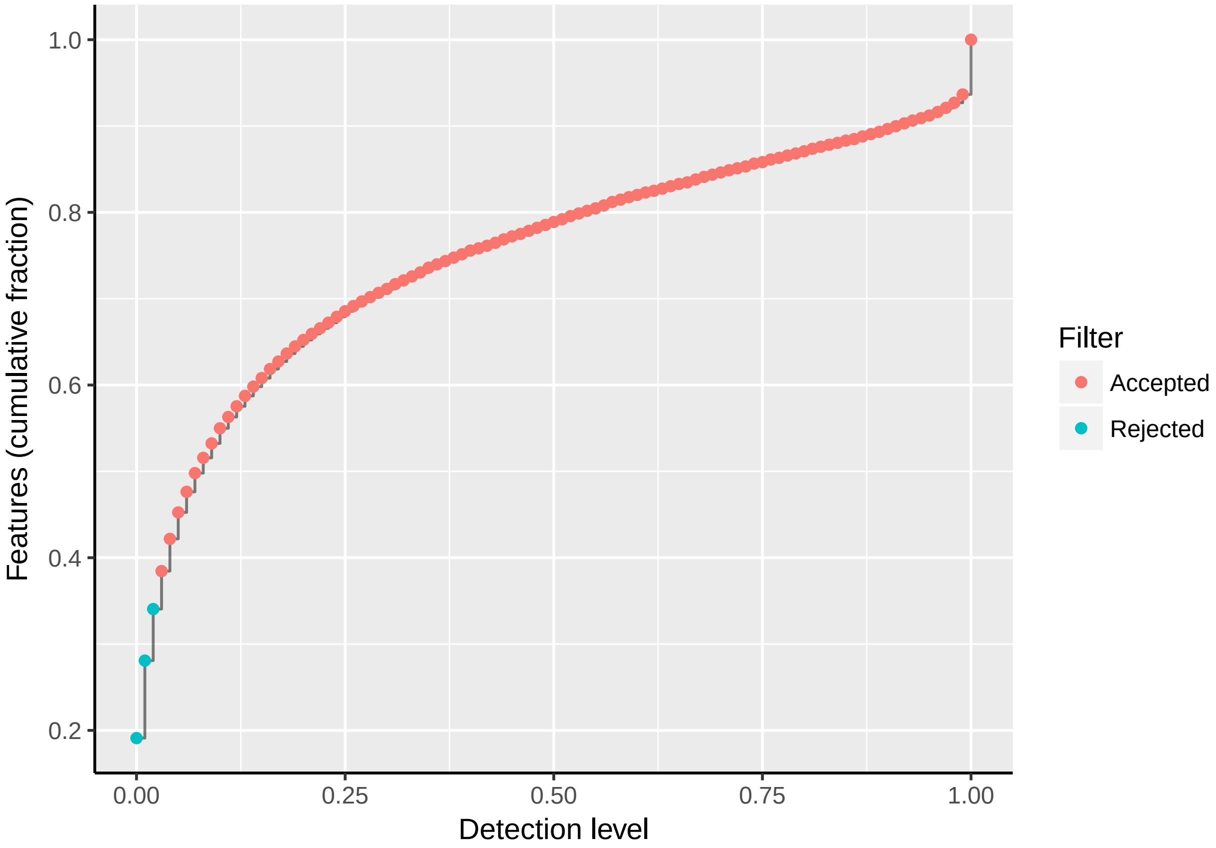
The names of trajectory features can be received from and set to a *SingleCellExperiment* object using the function `trajFeatureNames`. The `showTrajInfo` function, as well, provides an overview of the selected trajectory features.

Please note that the following filter functions can be used incrementally, since statistical characteristics are only analyzed for already designated trajectory features. In other words, using the filters multiple times or combining the filters will result in a more stringent selection of trajectory features. Initially, all features in a *SingleCellExperiment* (excluding spike-in controls), are assumed to be trajectory features.

3.1 Filter by Detection Level

This filter determines trajectory features that are present in a disproportionate small number of samples. It removes features that are not expressed or that do not sufficiently reach the technological limit of detection. The detection level is defined as the fraction of samples in which a feature is detected, i.e., the relative number of samples having a feature's expression value greater than 0. If a threshold >= 1 is selected, its value is automatically converted to a relative fraction of the total sample count. The empirical cumulative distribution function of all samples and the fraction of removed features is shown.

```
# Filter features expressed in at least 3 samples
tfeat <- filterTrajFeaturesByDL(exSim, threshold=2)
```



```
tfeat[1:5]
```

```
## [1] "feature_84" "feature_88" "feature_92" "feature_94" "feature_96"
```

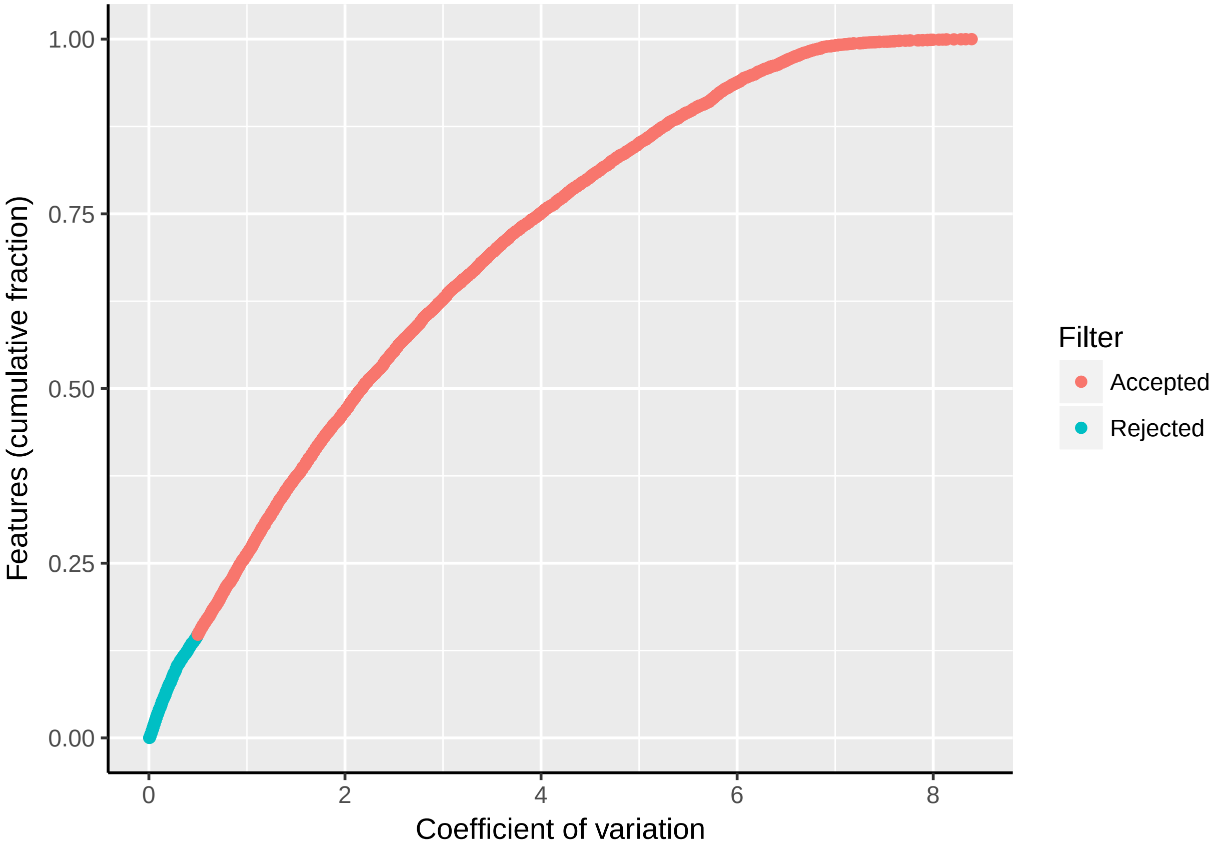
```
# Setting trajectory features to the object
trajFeatureNames(exSim) <- tfeat
showTrajInfo(exSim)
```

```
## [[ CellTrails ]]
## logcounts: 15000 features, 100 samples
## Pheno data:
##   sampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   phenoNames: none
## Feature data:
##   featureNames: "feature_1" "feature_2" ... "feature_15000" (15000)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "feature_84" "feature_88" ... "feature_15000" (9839)
##   latentSpace: none
##   states: none
## Trajectories: none
##   trajSampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   trajResiduals: MSE=NA
##   landmarks: none
##   trajLayout: none
## Trail data:
##   trailNames: none
```

3.2 Filter by Coefficient of Variation

This filter determines trajectory features with a narrow standard deviation (sd) with respect to its average expression (mean). This filter removes features with high expression and low variance, such as housekeeping genes. The coefficient of variation is computed by CoV(x) = sd(x)/mean(x). Features with a CoV(x) greater than a given threshold remain labeled as trajectory feature in the *SingleCellExperiment* object. The empirical cumulative distribution function of all samples and the fraction of removed features is shown.

```
# Filter features with COV > 0.5
tfeat <- filterTrajFeaturesByCOV(exSim, threshold=0.5)
```



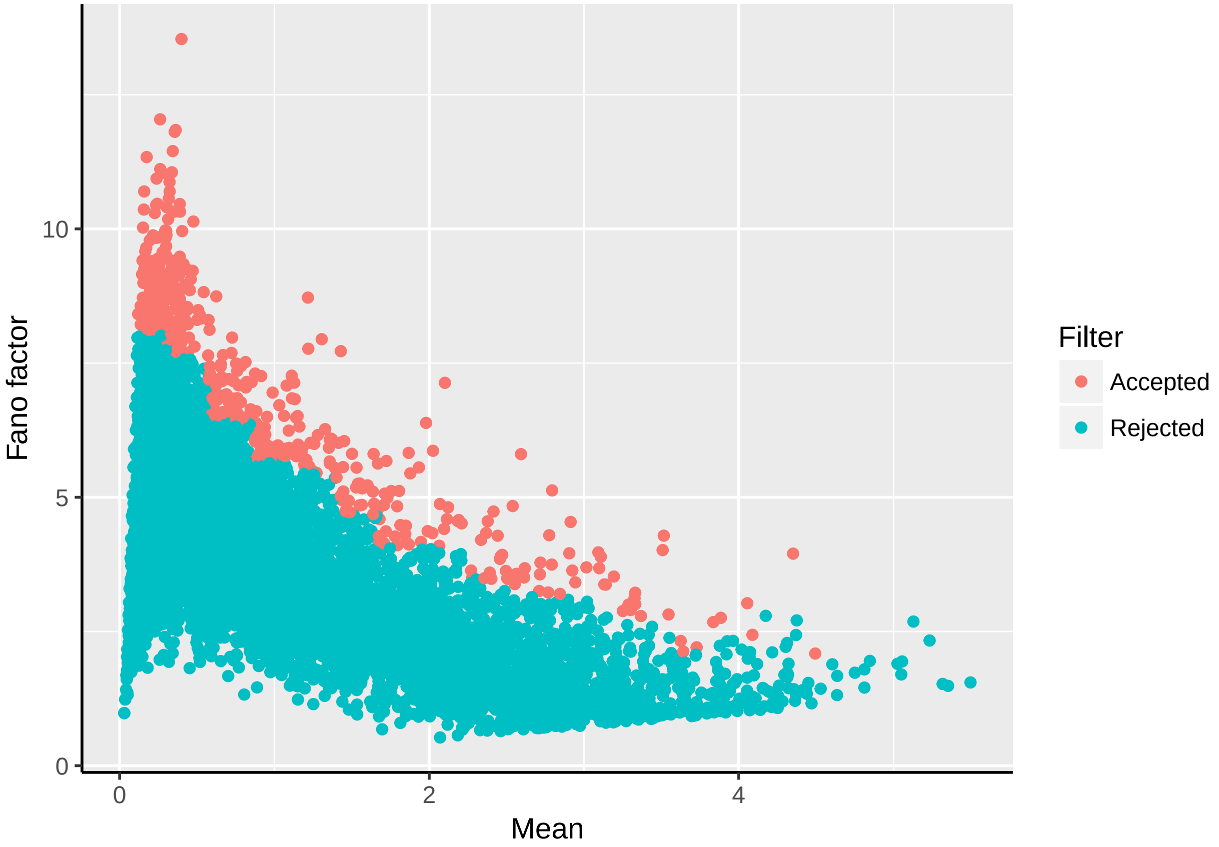
```
# Setting trajectory features to the object
trajFeatureNames(exSim) <- tfeat
showTrajInfo(exSim)
```

```
## [[ CellTrails ]]
## logcounts: 15000 features, 100 samples
## Pheno data:
##   sampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   phenoNames: none
## Feature data:
##   featureNames: "feature_1" "feature_2" ... "feature_15000" (15000)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "feature_84" "feature_88" ... "feature_14998" (8386)
##   latentSpace: none
##   states: none
## Trajectories: none
##   trajSampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   trajResiduals: MSE=NA
##   landmarks: none
##   trajLayout: none
## Trail data:
##   trailNames: none
```

3.3 Filter by Fano Factor

This filter identifies the most variable features while considering their average expression level. Features are placed into 20 bins based on their mean expression. For each bin, the distribution of Fano factors, which is a windowed version of the index of dispersion (IOD = variance / mean), is computed and standardized (*Z*-score(x) = x/sd(x) - mean(x)/sd(x)). Highly variable features with a *Z*-score greater than a given threshold remain labeled as trajectory feature in the *SingleCellExperiment* object. The parameter `min_expr` defines the minimum average expression level of a feature to be considered for this filter (default: 0). The Fano factor and the average expression is shown for each feature; filtered features are highlighted.

```
# Filter features with Z > 1.7
tfeat <- filterTrajFeaturesByFF(exSim, threshold=1.7)
```



```
# Setting trajectory features to the object
trajFeatureNames(exSim) <- tfeat
showTrajInfo(exSim)
```

```
## [[ CellTrails ]]
## logcounts: 15000 features, 100 samples
## Pheno data:
```

```
##   sampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   phenoNames: none
## Feature data:
##   featureNames: "feature_1" "feature_2" ... "feature_15000" (15000)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "feature_118" "feature_183" ... "feature_14998" (485)
##   latentSpace: none
##   states: none
## Trajectories: none
##   trajSampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   trajResiduals: MSE=NA
##   landmarks: none
##   trajLayout: none
## Trail data:
##   trailNames: none
```

3.4 Blocking Uniformative Substructures

The functions `filterTrajFeaturesByCOV` and `filterTrajFeaturesByFF` allow to define a design matrix to account for systematic bias in the expression data (e.g., batch, gender or cell cycle). It should list the nuisance factors that should be blocked and their values per sample. It is suggested to construct the design matrix with the `model.matrix` function. Please see *Manifold Learning/Blocking Uniformative Substructures* of this vignette for an example on how to create a proper design matrix.

3.5 Using Alternative Methods

The function `trajFeatureNames` allows to designate any set of features as trajectory features. The only requirement is that the names in the set match the stored feature names in the *SingleCellExperiment* object (check with function `featureNames`). For example, one could also use an abundance-dependent variance trend fit, as implemented in the *scran* package (Lun et al., 2016), to indicate trajectory features, as shown below. *Please note* that the (Bioconductor) package *scran* is not part of *CellTrails* and may be needed to be installed first.

```
## Not run:
##library(scran)
## End(Not run)

# Filter using scran
var_fit <- scran::trendVar(x=exSim, use.spikes=FALSE)
var_out <- scran::decomposeVar(x=exSim, fit=var_fit)
tfeat <- featureNames(exSim)[which(var_out$FDR < 0.01)]

# Setting trajectory features to the object
trajFeatureNames(exSim) <- tfeat
showTrajInfo(exSim)
```

```
## [[ CellTrails ]]
## logcounts: 15000 features, 100 samples
## Pheno data:
##   sampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   phenoNames: none
## Feature data:
##   featureNames: "feature_1" "feature_2" ... "feature_15000" (15000)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "feature_118" "feature_183" ... "feature_14987" (1058)
##   latentSpace: none
##   states: none
## Trajectories: none
##   trajSampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   trajResiduals: MSE=NA
##   landmarks: none
##   trajLayout: none
## Trail data:
##   trailNames: none
```

Please note that filters from different packages can also be combined by subsetting the *SingleCellExperiment* object with the trajectory features.

```
# Reset: all features are trajectory features
trajFeatureNames(exSim) <- featureNames(exSim)

# Use CellTrails filter
trajFeatureNames(exSim) <- filterTrajFeaturesByDL(exSim, threshold=2,
                                                  show_plot=FALSE)
trajFeatureNames(exSim) <- filterTrajFeaturesByCOV(exSim, threshold=0.5,
                                                   show_plot=FALSE)

exSim_sub <- exSim[trajFeatureNames(exSim), ]

# Filter using scran
var_fit <- scran::trendVar(x=exSim_sub, use.spikes=FALSE)
var_out <- scran::decomposeVar(x=exSim_sub, fit=var_fit)
tfeat <- featureNames(exSim_sub)[which(var_out$FDR < 0.01)]

# Setting trajectory features to the object
trajFeatureNames(exSim) <- tfeat
showTrajInfo(exSim)
```

```
## [[ CellTrails ]]
## logcounts: 15000 features, 100 samples
## Pheno data:
##   sampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   phenoNames: none
## Feature data:
##   featureNames: "feature_1" "feature_2" ... "feature_15000" (15000)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "feature_118" "feature_183" ... "feature_14987" (466)
##   latentSpace: none
##   states: none
## Trajectories: none
##   trajSampleNames: "sample_1" "sample_2" ... "sample_100" (100)
##   trajResiduals: MSE=NA
##   landmarks: none
##   trajLayout: none
## Trail data:
##   trailNames: none
```

Chapter 4

Manifold Learning

The samples' expression profiles are shaped by many factors, such as developmental age, tissue region of origin, cell cycle stage, as well as extrinsic sources such as status of signaling receptors, and environmental stressors, but also technical noise. In other words, a single dimension, despite just containing feature expression information, represents an underlying combination of multiple dependent and independent, relevant and non-relevant factors, whereat each factor's individual contribution is non-uniform. To obtain a better resolution and to extract underlying information, *CellTrails* aims to find a meaningful low-dimensional structure - a manifold - that represents samples mainly by their latent temporal relation.

4.1 Spectral Embedding

CellTrails aims to decipher the temporal relation between samples by computing a novel data representation which amplifies trajectory information in its first n dimensions. For this purpose, *CellTrails* employs spectral graph theory. Due to their locality-preserving character, spectral embedding techniques are advantageous because these consider the data's manifold structure, are insensitive to outliers and noise, are not susceptible to short-circuiting, and emphasize naturally occurring clusters in the data (Belkin and Niyogi, 2003; Sussman et al., 2012). In a nutshell, *CellTrails* assumes that two samples that have a high statistical dependency are represented in close proximity along a trajectory. *CellTrails* captures the intrinsic data geometry as a weighted graph (nodes = samples, edges = statistical dependencies between pairs of samples) by means of fuzzy mutual information and uses spectral graph decomposition to unfold the manifold revealing the hidden trajectory information.

The spectral embedding is performed using the function `embedSamples` and results in a list with the eigenspace representation of the original expression data.

```
# Spectral Embedding
se <- embedSamples(exBundle)
```

```
## Computing adjacency matrix ...

## Computing spectral embedding ...
```

```
names(se)
```

```
## [1] "components"  "eigenvalues"
```

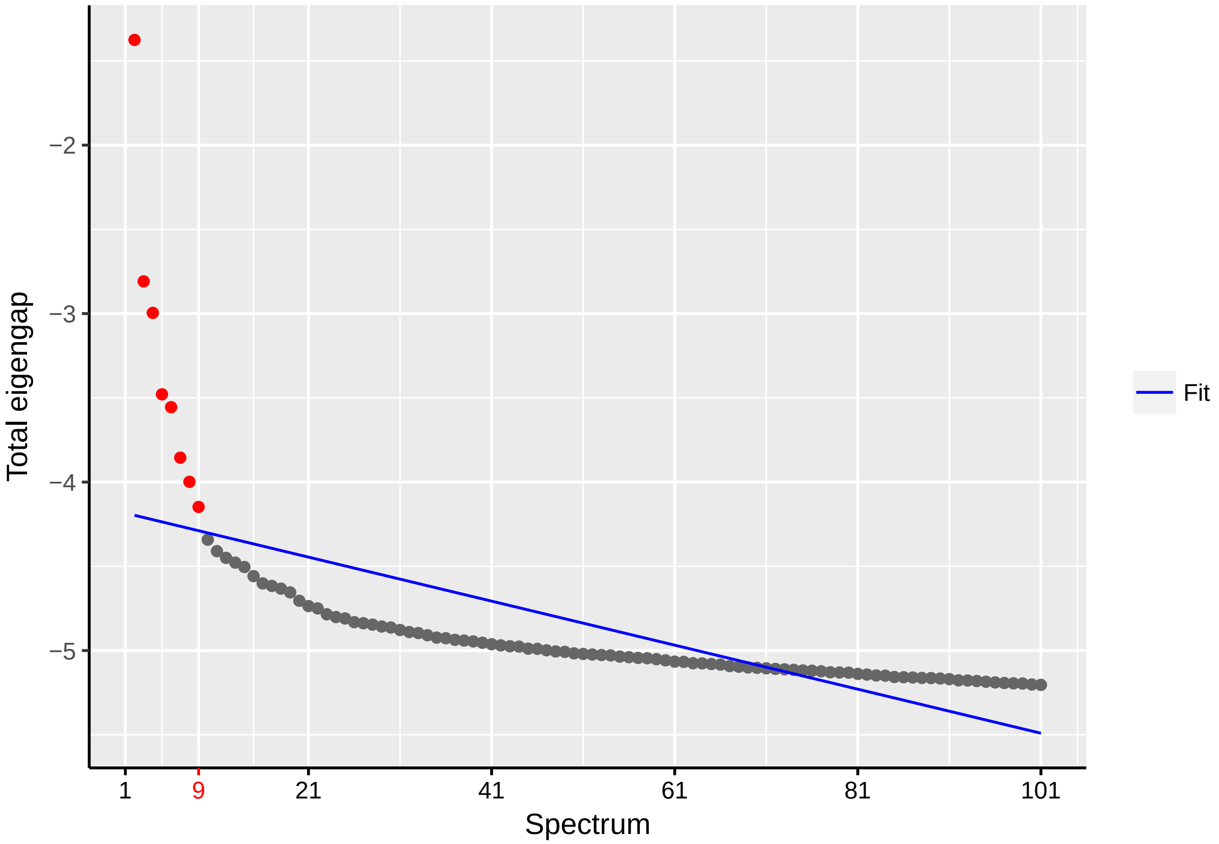
Please note that this function can also be applied to any numerical matrix of interest.

4.2 Dimensionality Reduction

CellTrails assumes that the expression vectors are lying on or near a manifold with a low dimensionality that is embedded in the higher-dimensional space. The number of dimensions can be reduced, which lowers noise (i.e., truncates non-relevant dimensions), while the geometry of the trajectory is emphasized.

The function `findSpectrum` helps to identify the intrisic dimensionality of the data. It determines the most informative dimensions based on the eigenvalues (spectrum) of the eigenspace. Components of the latent space are ranked by their information content. In the following example, *CellTrails* identifies relevant components by using a linear fit on the eigengaps of the first 100 eigenvalues.

```
# Identify relevant components
d <- findSpectrum(se$eigenvalues, frac=100)
```



```
d
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

We suggest to assess the resulting Scree plot (eigengaps versus spectrum size) for whether the estimation of the unknown intrinsic dimensionality was reasonable. Otherwise, we recommend to adjust the parameter `frac` accordingly.

Please note that considering too few components of the latent space may result in loss of information, while selecting lower ranked components could increase noise.

Next, we set the identified latent space to our *SingleCellExperiment* object:

```
latentSpace(exBundle) <- se$components[, d]
```

```
## Calculating approximation of CellTrails manifold for 2D visualization...
## Used tSNE perplexity: 30
```

4.3 Blocking Uninformative Substructures

Single-cell measurements are susceptible to the influence of confounders, such as batch, gender or cell cycle effects. Blocking these nuisance factors during manifold learning may be necessary to significantly improve the result of downstream data analyses, such as reconstruction of the temporal trajectory. Therefore, the function `embedSamples` can account for confounding effects via the parameter `design`, as will be demonstrated on the example of single-cell RNA-Seq data of murine T helper 2 cell ($\mathrm{T_h}2$) differentiation (Mahata et al., 2014). In a nutshell, Buettner *et al.* (Buettner et al., 2015) identified cell cycle effects as major confounder in this dataset and applied a single-cell latent variable model (scLVM) approach to account for this factor. They unbiasedly identified then two cell populations, namely a group of partially and a group of fully differentiated cells. The normalized, log transformed and filtered scRNA-Seq data can be obtained from the supplementary materials of their article (Table S5 and S7); further, a curated list of $\mathrm{T_h}2$ marker genes, the scLVM-corrected expression matrix, and a binary cluster assignment for each cell can be downloaded.

For your convenience, the numeric expression matrix (here called `th2`) and the list of marker genes were already organized in a *SingleCellExperiment* container. Here, the expression matrix consists of 7,063 selected genes (116 of which are marker genes) which have been detected in more than 3 of all 81 cells.

```
th2 <- readRDS(system.file("exdata", "th2.rds", package="CellTrails"))
th2
```

```
## class: SingleCellExperiment
## dim: 7063 81
## metadata(0):
## assays(1): logcounts
## rownames(7063): Gnai3 Cdc45 ... ENSMUSG00000097906 X4933404O12Rik
## rowData names(2): isMarker ENSEMBL
## colnames(81): Cell 1 Cell 2 ... Cell 80 Cell 81
## colData names(0):
## reducedDimNames(0):
## spikeNames(0):
```

```
# Number of markers
nMarkers <- sum(rowData(th2)$isMarker)
nMarkers
```

```
## [1] 116
```

```
# Number of total genes
nGenes <- nrow(th2)
nGenes
```

```
## [1] 7063
```

First, we have a quick look into the unprocessed dataset. If the latent temporal factor is a major source of variance, two clusters, which separate fully from partially differentiated cells, should be detectable; if those clusters are not identifiable, the data is affected by uniformative substructures. We assume that $\mathrm{T_h}2$ marker genes should be enriched in the group of genes differentially expressed between clusters, i.e. the enrichment odds ratio should be >1 and the enrichment P-value should be significant if cells were clustered by maturity.

```
# Clustering in the original space
D <- dist(t(logcounts(th2)))
dendro <- hclust(D, method="ward.D2")
cluster <- cutree(dendro, k=2)
```

```
# Differential expression
pvals <- apply(logcounts(th2), 1, function(x) {
  wilcox.test(x[cluster == 1],
              x[cluster == 2],
              exact=FALSE)$p.value})
fdr <- p.adjust(pvals, method="fdr")

# Number of differentially expressed markers for FDR < 0.05
de <- names(fdr[fdr < 0.05]) #differentially expressed genes
deGenes <- length(de) #number of genes
deMarkers <- sum(rowData(th2[de, ])$isMarker) #number of markers

# Enrichment statistic
enrichment.test(deMarkers, nMarkers, deGenes, nGenes)
```

```
## $p.value
## [1] 0.989218
##
## $odds.ratio
## odds ratio
##  0.6526187
##
## $conf.int
## [1] 0.4687965       Inf
## attr(,"conf.level")
## [1] 0.95
##
## $method
## [1] "Fisher's exact test for enrichment"
```

Since the enrichment is not significant (with an odds ratio < 1), we argue that cells were not properly separated by maturity in the original space.

To block the cell cycle effects, *CellTrails* expects a design matrix modeling the cell cycle stage as the explanatory factor for each cell. As the cell-cycle stage of each cell is not known in this data set, we need to predict cell cycle phases. In this example, we use the classifier `cyclon` from the *scran* package (Lun et al., 2016). To be able to run the algrithm properly, gene symbols were translated to Ensembl identifiers using Bioconductors' annotation database interface package *AnnotationDbi* (Pagès et al., 2017) and the mouse annotation data package *org.Mm.eg.db* (Carlson, 2017).

Please note that these packages are not part of *CellTrails* and may be needed to be installed first.

```
## Not run:
##library(scran)
## End(Not run)

# Run cyclone
mcm <- readRDS(system.file("exdata", "mouse_cycle_markers.rds",
                           package="scran"))
set.seed(1101)
cellCycle <- scran::cyclone(x=logcounts(th2),
                            pairs=mcm,
                            gene.names=rowData(th2)$ENSEMBL)

# Number of predicted phases
table(cellCycle$phases)
```

```
##
##  G1 G2M   S
##  59  14   8
```

Let's create the respective design matrix using the `cyclon` classification scores.

```
# Design matrix
cc_design <- model.matrix(~ cellCycle$scores$G1 + cellCycle$scores$G2M)
head(cc_design)
```

```
##   (Intercept) cellCycle$scores$G1 cellCycle$scores$G2M
## 1           1               0.108                0.701
## 2           1               1.000                0.002
## 3           1               0.971                0.009
## 4           1               1.000                0.000
## 5           1               1.000                0.000
## 6           1               0.671                0.338
```

Next, we reduce the dimensionality using *CellTrails*. Passing the design matrix to `embedSamples` ensures that *CellTrails* properly regresses out the effects of the explanatory variables before learning the manifold. Then, we cluster the cells in the derived lower-dimensional space.

```
# Perform Dimensionality Reduction with Design Matrix
se <- embedSamples(th2, design=cc_design)
```

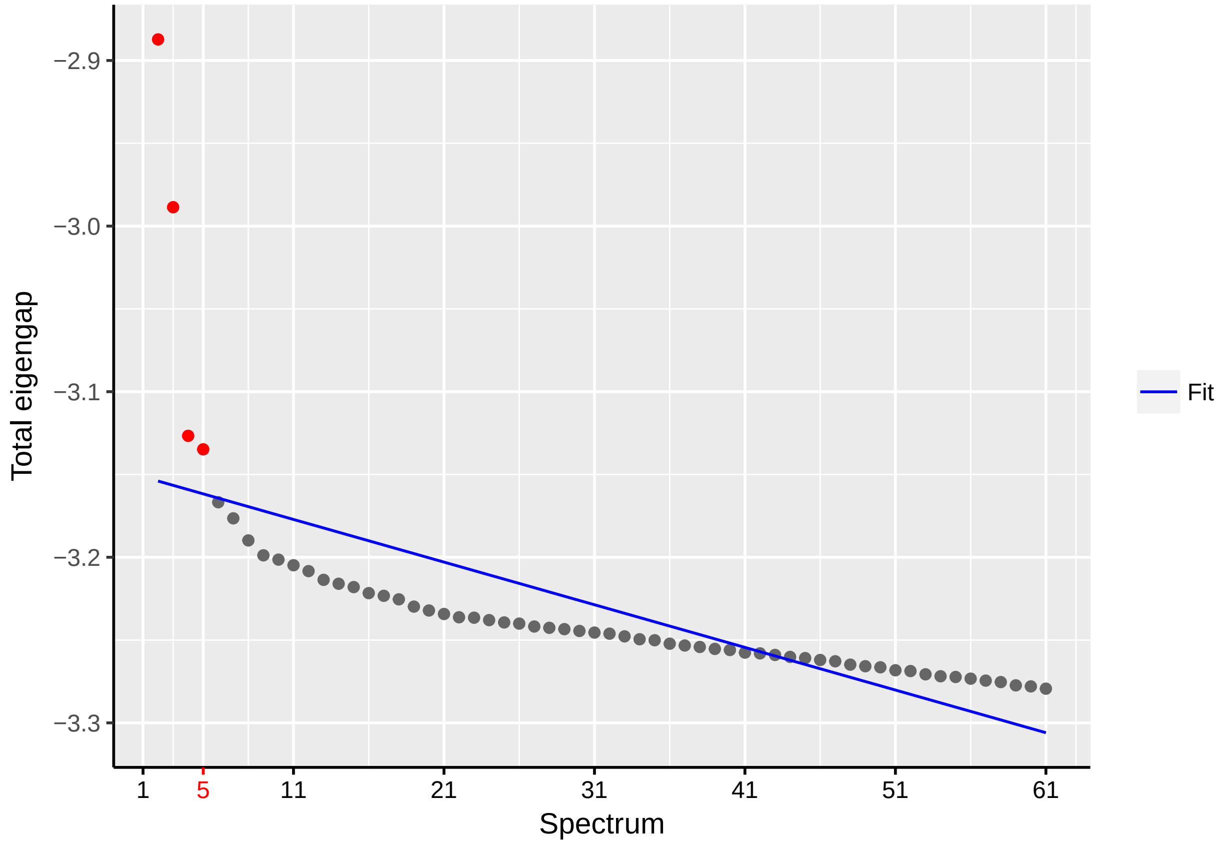
```
## Warning in .embedSamples_def(x = M, design = design): Please note that
## trajectory features weren't selected. Thus, spectral embedding will be
## performed on all features, which may result in lower accuracy and longer
## computation time.
```

```
## Blocking nuisance factors ...
```

```
## Computing adjacency matrix ...
```

```
## Computing spectral embedding ...
```

```
d <- findSpectrum(se$eigenvalues, frac=60)
```



```
latentSpace(th2) <- se$components[, d]
```

```
## Calculating approximation of CellTrails manifold for 2D visualization...

## Used tSNE perplexity: 16
```

```
# Clustering in Latent Space
D <- dist(latentSpace(th2))
dendro <- hclust(D, method="ward.D2")
cluster <- cutree(dendro, k=2)
```

We test the quality of clustering by quantifying the enrichment of marker genes in the set of differentially expressed genes.

```
# Differential expression
pvals <- apply(logcounts(th2), 1, function(x) {
  wilcox.test(x[cluster == 1],
              x[cluster == 2],
              exact=FALSE)$p.value})
fdr <- p.adjust(pvals, method="fdr")

# Number of differentially expressed markers for FDR < 0.05
de <- names(fdr[fdr < 0.05]) #differentially expressed genes
deGenes <- length(de) #number of genes
deMarkers <- sum(rowData(th2[de, ])$isMarker) #number of markers

# Enrichment statistic
enrichment.test(deMarkers, nMarkers, deGenes, nGenes)
```

```
## $p.value
## [1] 6.144029e-07
##
## $odds.ratio
## odds ratio
##   5.971408
##
## $conf.int
## [1] 3.431202      Inf
## attr(,"conf.level")
## [1] 0.95
##
## $method
## [1] "Fisher's exact test for enrichment"
```

The marker gene enrichment is significant (P-value $< 10^{-6}$) and the odds ratio is remarkably increased to ~6, indicating that the cells are now properly separated by maturity. In comparison, an enrichment odds ratio of 2.4 was achieved using the cell-cycle 'corrected' data and the clustering provided in the original scLVM study (Buettner et al., 2015).

Please note that the differential gene expression analysis using the *CellTrails* derived clusters was performed on the actual expression matrix and not the cell-cycle 'corrected' expression values. In contrast to scLVM, *CellTrails* blocks the nuisance variables for manifold learning only and keeps the original expression values for downstream analysis. This is due to the fact that the manipulated expression matrix does not represent the actual transcript levels measured in each cell, nor does it account for the uncertainty of estimation of the blocking factor terms. By this means, *CellTrails* protects against confounding effects without discarding information.

Besides cell cycle, technical confounders may also be relevant to be accounted for. Those can occur, for example, if samples were processed on different plates or if samples were pooled from multiple sequencing runs. In this case, a design matrix with the respective explanatory variables can be constructed and passed to `embedSamples`.

4.4 Using Alternative Methods

If the user prefers to use an alternative approach for dimensionality reduction, any latent space can be set to a *SingleCellExperiment* object. The latent space has to be a numerical matrix; rows represent samples and columns the components of the latent space. *CellTrails* uses by default spectral embedding, but the framework also operates well with any other spectral dimensionality reduction method, such as PCA (e.g., available in *CellTrails* via function `pca`) and diffusion maps (e.g., available via the *destiny* package (Angerer et al., 2015); please note this package is not part of *CellTrails* and may be needed to be installed first):

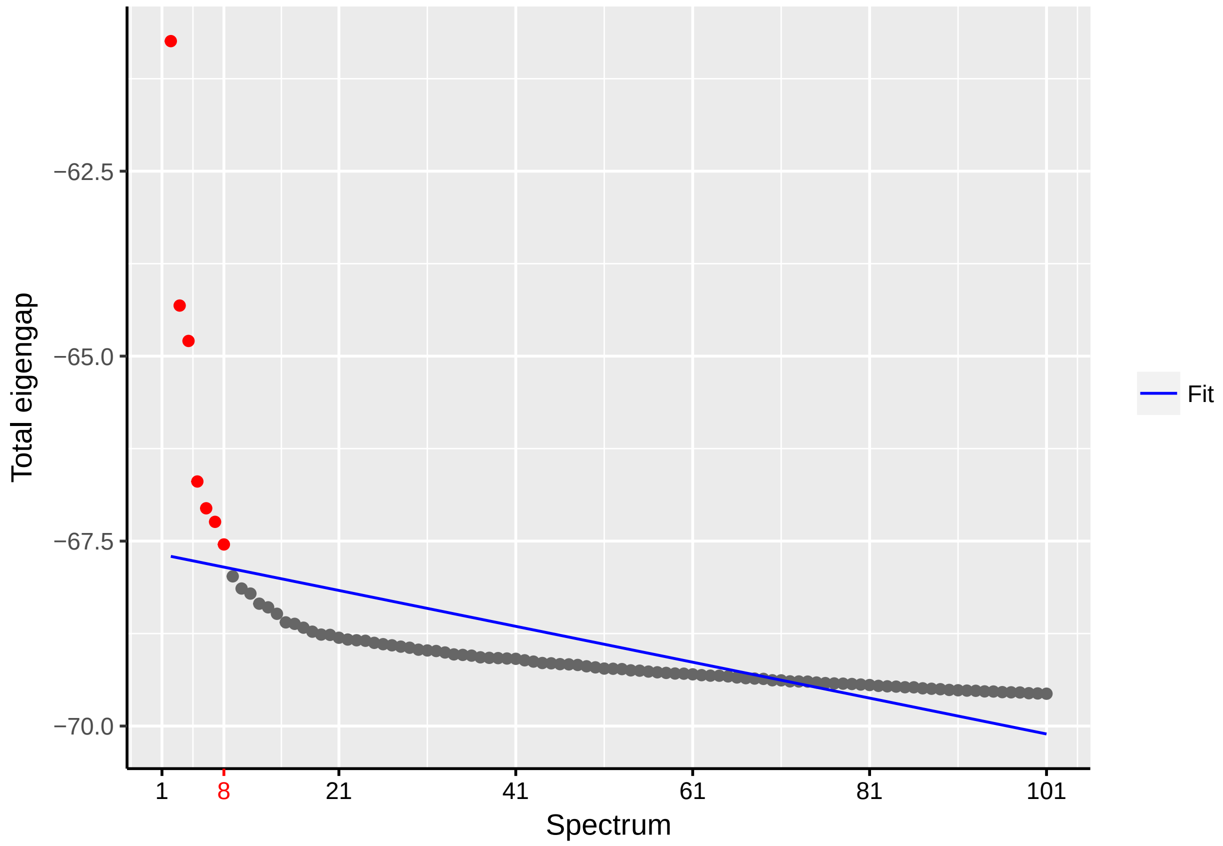
```
# Make copy of example data
exAlt <- exBundle

# PCA
pca_result <- pca(exAlt)
```

```
## Warning in .pca_def(M = M, do_scaling = do_scaling, design = design): 1
## feature(s) do(es) not encode valuable information (i.e., has/have constant
## expression over all samples) and was/were therefore neglected.
```

```
## Performing PCA ...
```

```
d <- findSpectrum(pca_result$eigenvalues, frac=100)
```



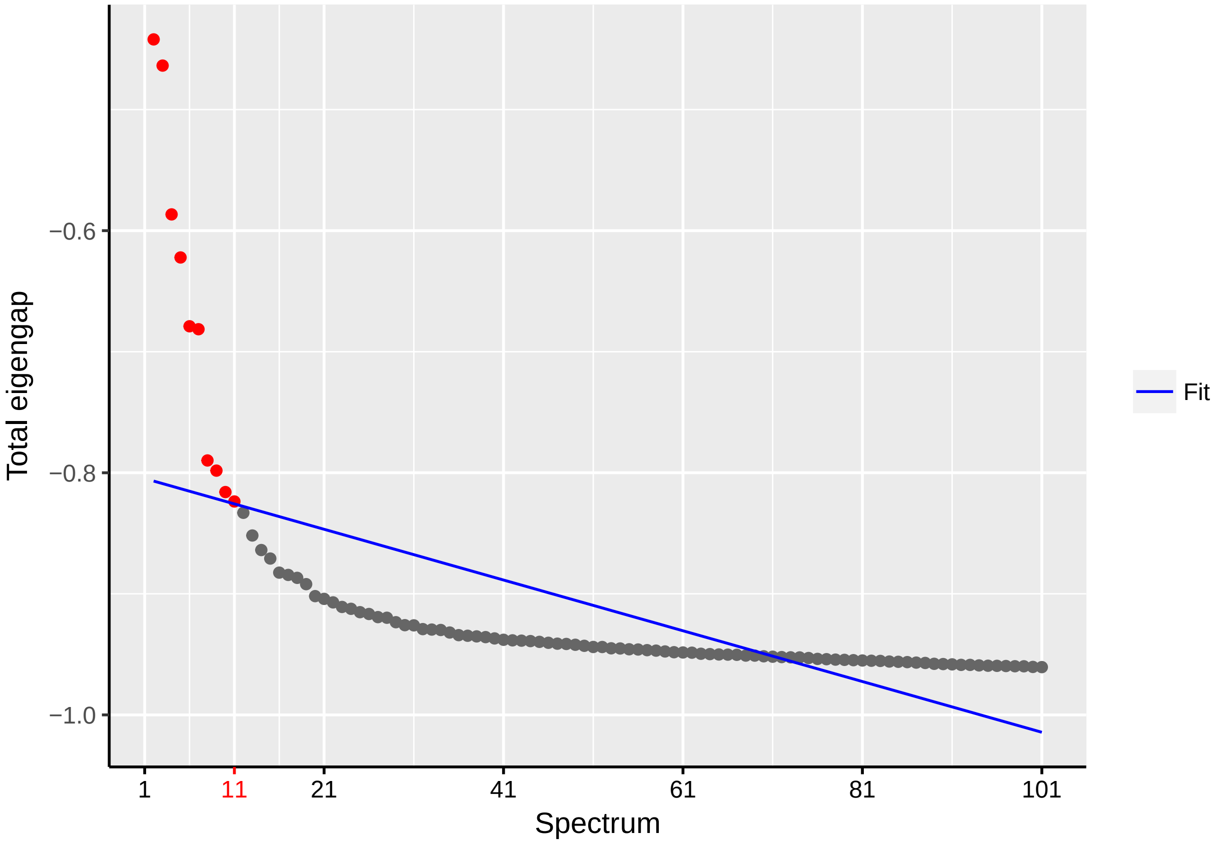
```
latentSpace(exAlt) <- pca_result$components[, d]
```

```
## Calculating approximation of CellTrails manifold for 2D visualization...
```

```
## Used tSNE perplexity: 30
```

```
# Diffusion maps
## Not run:
##library(destiny)
## End(Not run)

lcounts <- t(logcounts(exAlt))
dmaps_result <- destiny::DiffusionMap(lcounts, n_eigs = 101)
d <- findSpectrum(destiny:::eigenvalues(dmaps_result), frac=100)
```



```
latentSpace(exAlt) <- destiny:::eigenvectors(dmaps_result)[, d]
```

```
## Calculating approximation of CellTrails manifold for 2D visualization...
## Used tSNE perplexity: 30
```

Please note that the function `latentSpace<-` accepts any numerical matrix. Therefore, any latent space with an already reduced number of dimensions can be assigned to a *CellTrailsSet* object with this function; eigenvalues are only used to determine the intrinsic dimensionality of the data set.

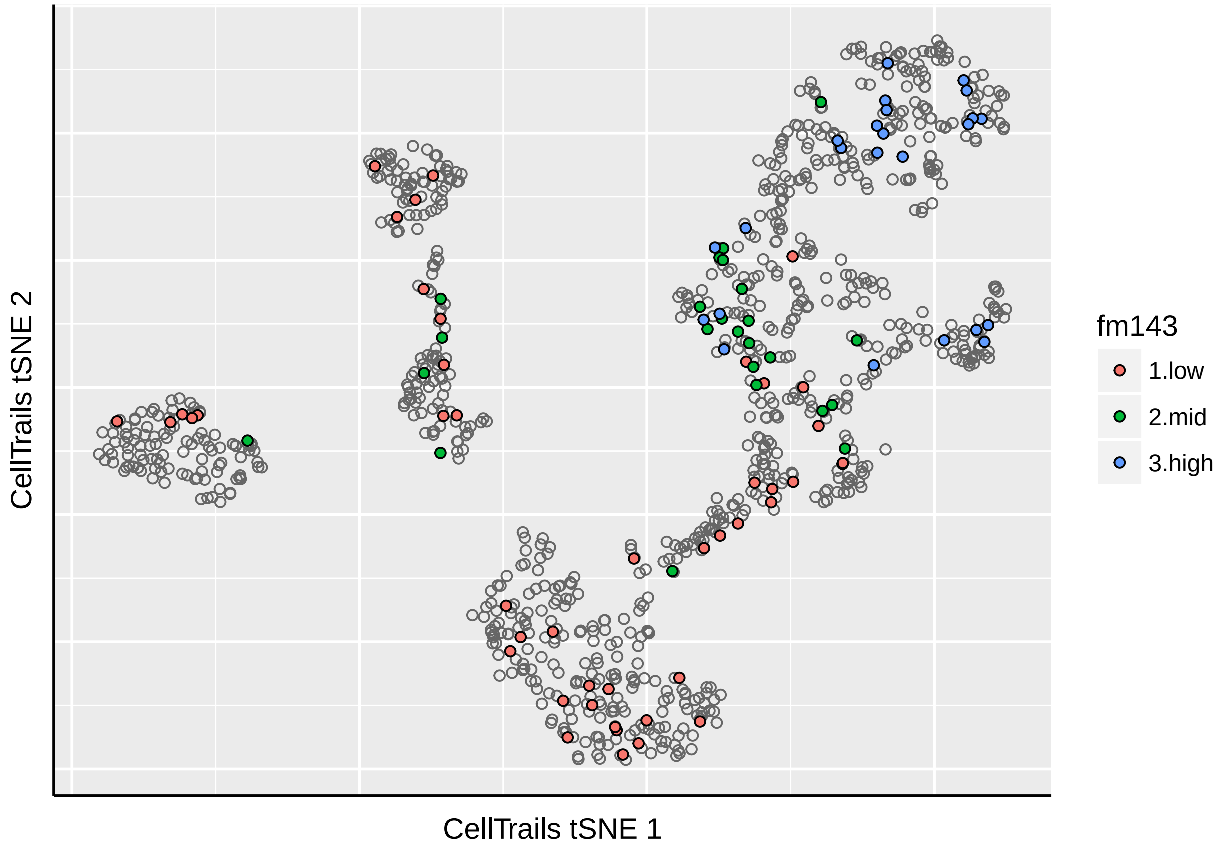
4.5 Visualization

CellTrails allows us to visualize an approximation of the learned lower-dimensional manifold in two dimensions. *CellTrails'* plot function `plotManifold` uses t-distributed stochastic neighbor embedding (tSNE) (van der Maaten and Hinton, 2008) to illustrate the arrangement of the samples in the latent space in a two-dimensional plot. Points denote individual samples, the colorization indicates either a metadata label or expression of a single feature. Empty points denote a missing label or missing expression value (non-detects). Available phenotype lables can be listed with the function `phenoNames`, available features with `featureNames`, respectively.

```
# Show available phenotype labels
phenoNames(exBundle)
```

```
## [1] "fm143"  "origin"
```

```
# Show sample metainformation 'fm143 dye uptake'
plotManifold(exBundle, color_by="phenoName", name="fm143")
```

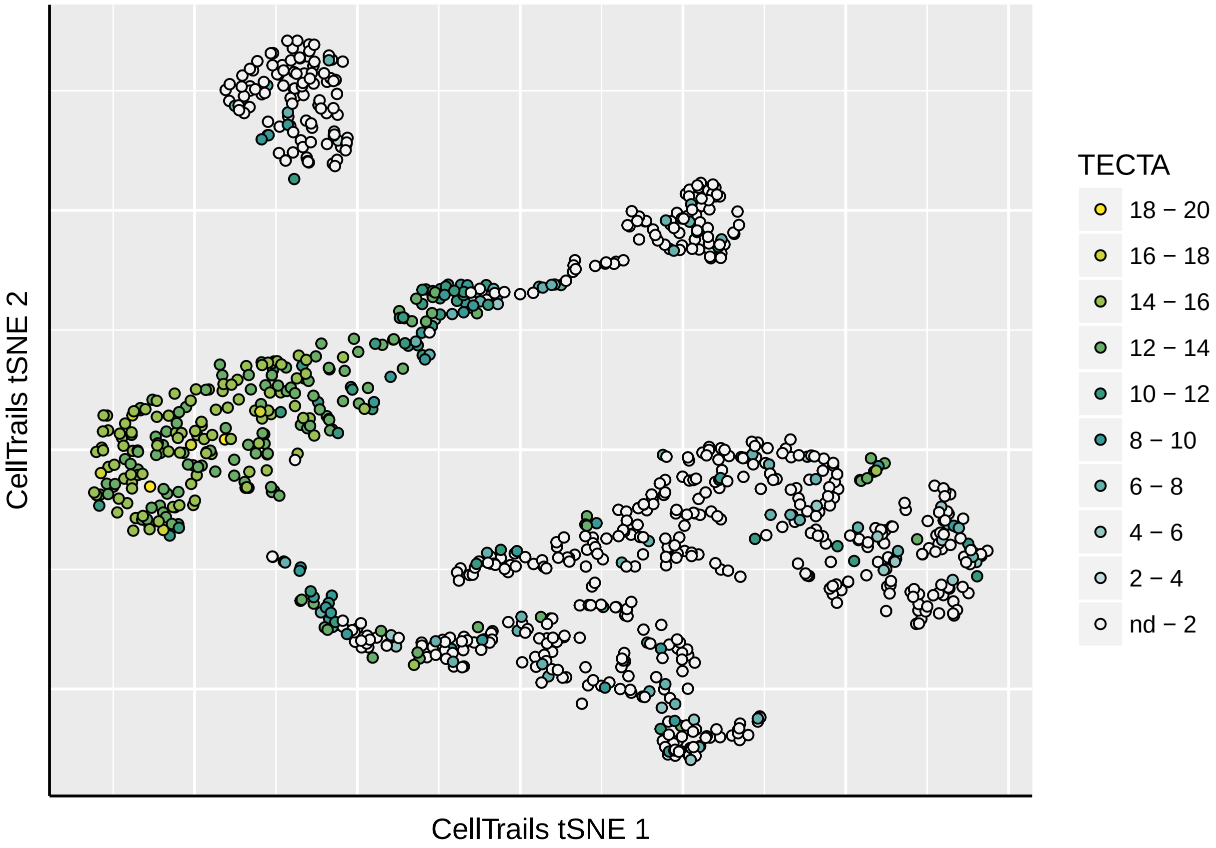


The function `plotManifold` returns a `ggplot` object (Wickham, 2009) from the *ggplot2* package, which can be adapted by the user's needs and preferences (for details, please refer to the *ggplot2* manual; to save the plot see function `ggsave`). The 2D representation of the latent manifold is by default already stored the the *SingleCellExperiment* object (also accessible via `reducedDims`). However, the `plotManifold` function provides the parameter `recalculate`. For example, if we want to change the `perplexity` parameter of the tSNE calculation, then we set `recalculate=TRUE`. The new tSNE result needs to be set to the *SingleCellExperiment* object using the `manifold2D` function, respectively.

```r
# Show feature expression (e.g., gene TECTA)
gp <- plotManifold(exBundle, color_by="featureName", name="TECTA", recalculate=TRUE)
```

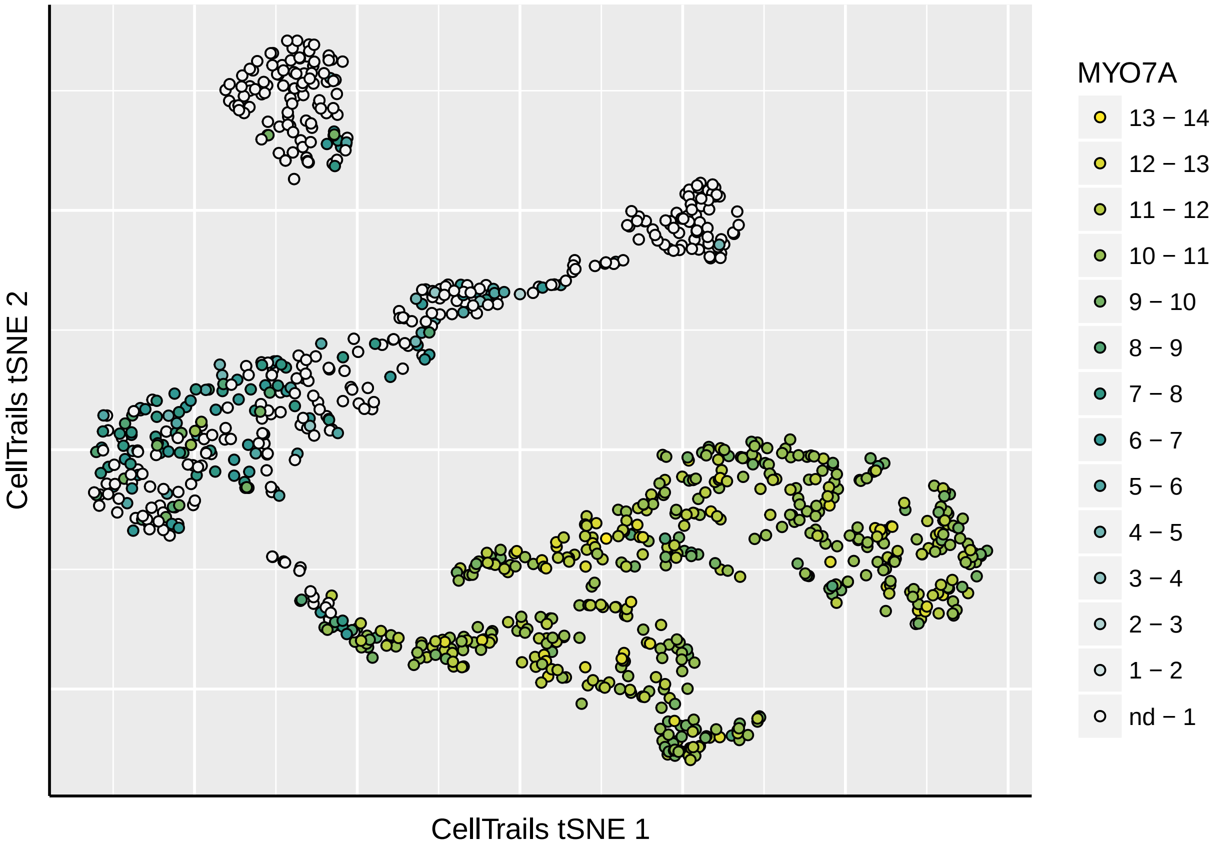
```
## Calculating 2D approximation of CellTrails manifold...
```

```r
gp
```



```
# Store tSNE result
manifold2D(exBundle) <- gp

# Show feature expression (e.g., genes MYO7A)
plotManifold(exBundle, color_by="featureName", name="MYO7A")
```


CellTrails tSNE 2
CellTrails tSNE 1
MYO7A
13 − 14
12 − 13
11 − 12
10 − 11
9 − 10
8 − 9
7 − 8
6 − 7
5 − 6
4 − 5
3 − 4
2 − 3
1 − 2
nd − 1

Chapter 5

Clustering

5.1 Hierarchical Spectral Clustering

To identify cellular subpopulations, *CellTrails* performs hierarchical clustering via minimization of a square error criterion (and, 1963) in the lower-dimensional space. To determine the number of clusters, *CellTrails* conducts an unsupervised *post-hoc* analysis. Here, it is assumed that differential expression of assayed features determines distinct cellular stages. Hierarchical clustering in the latent space generates a cluster dendrogram. *CellTrails* makes use of this information and identifies the maximal fragmentation of the data space, i.e. the lowest cutting height in the clustering dendrogram that ensures that the resulting clusters contain at least a certain fraction of samples. Then, processing from this height towards the root, *CellTrails* iteratively joins siblings if they do not have at least a certain number of differentially expressed features. Statistical significance is tested by means of a two-sample non-parametric linear rank test accounting for censored values (Peto and Peto, 1972). The null hypothesis is rejected using the Benjamini-Hochberg (Benjamini and Hochberg, 1995) procedure for a given significance level. The number of clusters can impact the outcome of the trajectory reconstruction and therefore, this step might require some parameter tuning depending on the input data (for more information on the parameters call `?findStates`).

```
cl <- findStates(exBundle, min_size=0.01, min_feat=5, max_pval=1e-4, min_fc=2)
```

```
## Initialized 25 clusters with a minimum size of 10 samples each.
```

```
## Performing post-hoc test ...
```

```
## Found 11 states.
```

```
head(cl)
```

```
## [1] S7  S1  S4  S11 S9  S8
## Levels: S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11
```

The clusters identified by *CellTrails* are referred to as states along the trajectory. The function `states` can be used to set the clusters to the *SingleCellExperiment* object.

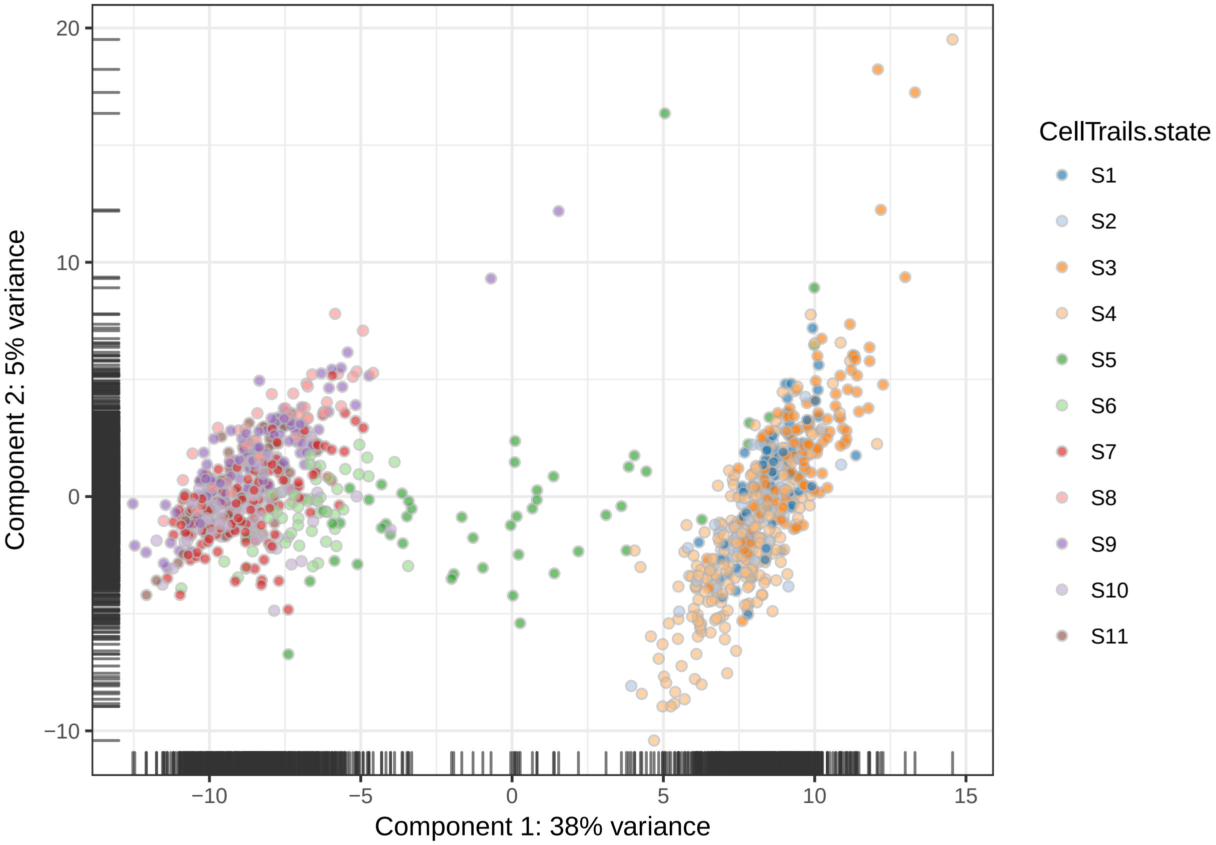
```
# Set clusters
states(exBundle) <- cl
```

State assignments are stored as sample metainformation and can be either recieved via `colData` or `states`. Since `CellTrails` operates on a *SingleCellExperiment* object, its results can be easily used by other packages. For example, visualizing a principal component analysis with *scater* (McCarthy et al., 2017):

```
## Not run:
##library(scater)
```

```
## End(Not run)

# Plot scater PCA with CellTrails cluster information
scater::plotPCA(exBundle, colour_by="CellTrails.state")
```



Please note that the (Bioconductor) package *scater* is not part of *CellTrails* and may be needed to be installed first.

5.2 Using Alternative Methods

Technically, the function `states<-` allows to set any clustering result to a *SingleCellExperiment* object. Any numeric, character or factor vector containing the cluster assignments for each sample is accepted.

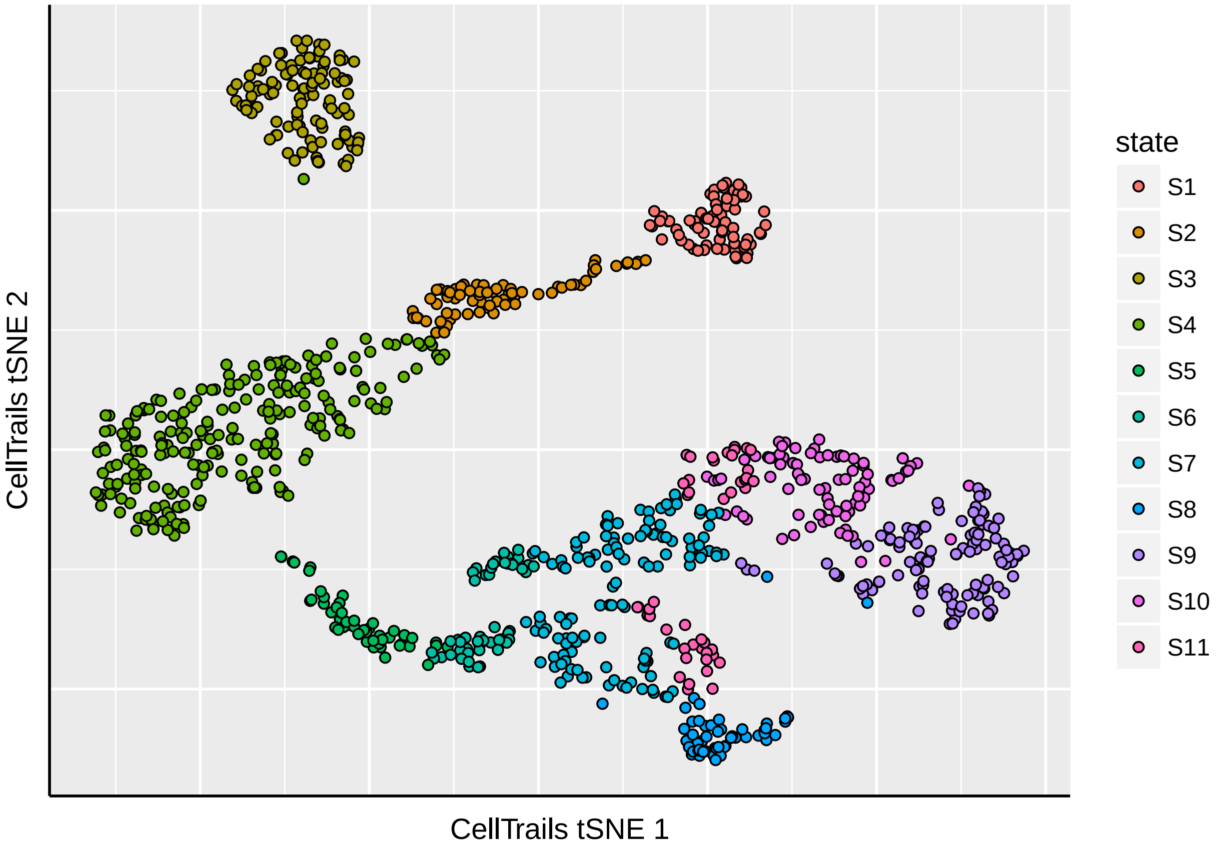
5.3 Visualization

As before, we can visualize the approximated lower-dimensional manifold and colorize each sample by its assigned state.

```
# States are now listed as phenotype
phenoNames(exBundle)
```

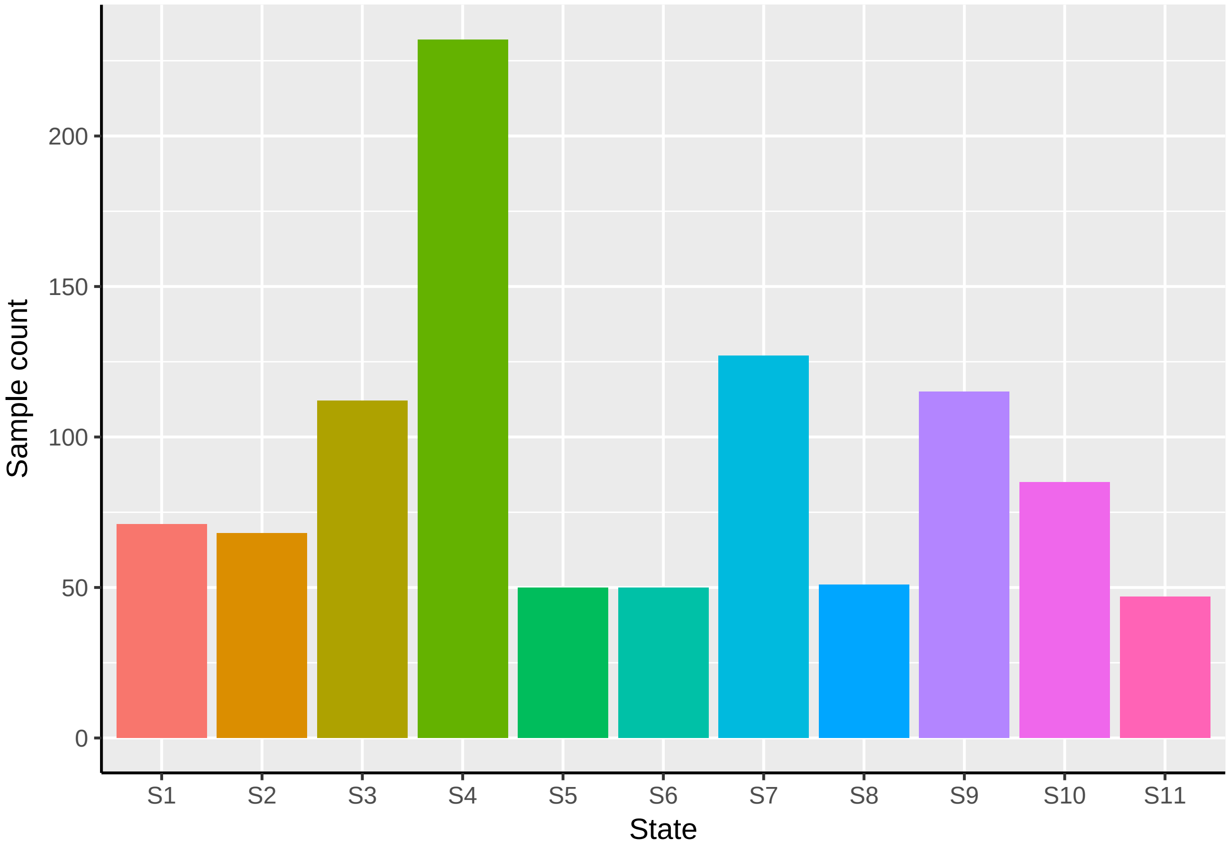
```
## [1] "fm143"  "origin" "state"
```

```
# Show manifold
plotManifold(exBundle, color_by="phenoName", name="state")
```



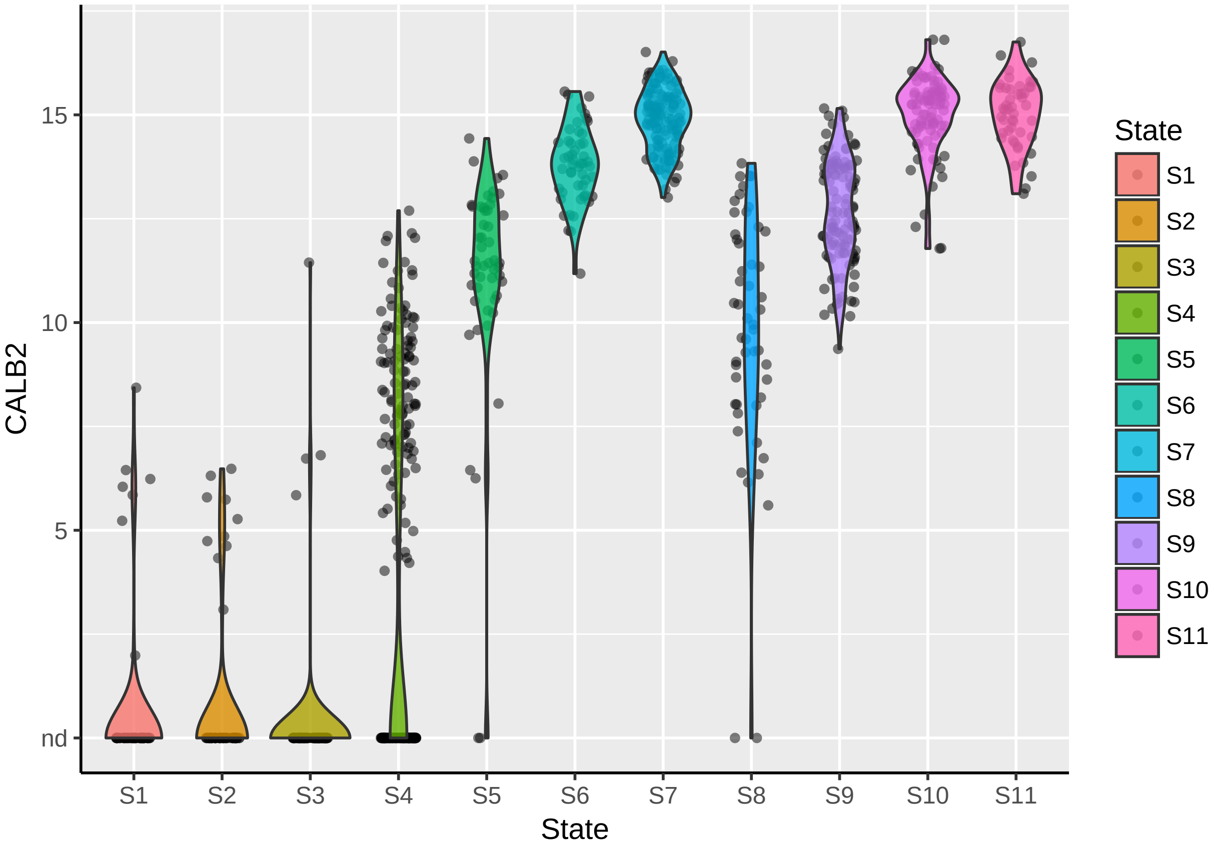
The function `plotStateSize` generates a barplot showing the absolute sizes of each state.

```
plotStateSize(exBundle)
```



Further, violin plots can be produced showing the expression distribution of a feature per state. Each point displays the feature's expression value in a single sample. A violine represents a vertically mirrored density plot on each side.

```r
plotStateExpression(exBundle, feature_name="CALB2")
```


CALB2
15
10
5
nd
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
State
State
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11

Chapter 6

Sample Ordering

CellTrails assumes that the arrangement of samples in the computed latent space constitutes a trajectory. *CellTrails* aims to place single samples along a maximum parsimony tree, which resembles a branching developmental continuum. Distances between samples in the latent space are computed using the Euclidean metric.

6.1 State Trajectory Graph

To avoid overfitting and to facilitate the accurate identification of bifurcations, *CellTrails* simplifies the problem. Analogous to the idea of a 'broken-stick regression', *CellTrails* groups the data and performs linear fits to separate trajectory segments, which are determined by the branching chronology of states. This leaves the optimization problem of finding the minimum number of associations between states that maximize the total parsimony. In theory, this problem can be solved by any minimum spanning tree algorithm. *CellTrails* adapts this concept by assuming that adjacent states should be located nearby and, therefore, share a relative high number of neighboring samples.

Each state defines a submatrix of samples that is composed of a distinct set of data vectors, i.e. each state is a distinct set of samples represented in the lower-dimensional space. For each state, *CellTrails* identifies the l-nearest neighbors to each state's data vector and takes note of their state memberships and distances. This results in two vectors of length l times the state size. Subsequently, *CellTrails* removes spurious neighbors (outliers/false-positive neighbors), which are statistically too distal. For each state *CellTrails* calculates the relative frequency on how often a state occurs in the neighborhood of a given state, which is referred to as the interface cardinality scores.

CellTrails implements a greedy algorithm to find the tree maximizing the total interface cardinality score, similar to Kruskal's minimum spanning tree algorithm (Kruskal, 1956). The graph construction has a relaxed requirement (number of edges < number of nodes) compared to a tree (number of edges = number of nodes - 1), which may result in a graph having multiple tree components (= forest) indicating potentially independent trajectories or isolated nodes.

Please note that the function `connectStates` can be adjusted, such that the resulting number of components may be lower or higher by increasing or decreasing the parameter `l`, respectively.

```
# State trajectory graph computation
exBundle <- connectStates(exBundle, l=10)
```

```
## Calculating layout of state trajectory graph component 1...
```

```
## Calculating layout of state trajectory graph component 2...
```

In our example dataset, we identified two components, as indicated by the *Trajectories* entity of the `showTrajInfo` function: component 1 is a tree with 10 states connected by 9 edges, and component 2 is an isolated state (one state, zero edges).

```
# Show trajectory information
showTrajInfo(exBundle)
```

```
## [[ CellTrails ]]
## logcounts: 183 features, 1008 samples
## Pheno data:
##   sampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   phenoNames: "fm143" "origin" "state" (3)
## Feature data:
##   featureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   latentSpace: 1008 samples, 9 dimensions
##   states: "S1" "S2" ... "S11" (11)
## Trajectories: [Component(#Vertices,Edges)]: 1(10,9) 2(1,0)
##   trajSampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   trajResiduals: MSE=NA
##   landmarks: none
##   trajLayout: none
## Trail data:
##   trailNames: none
```

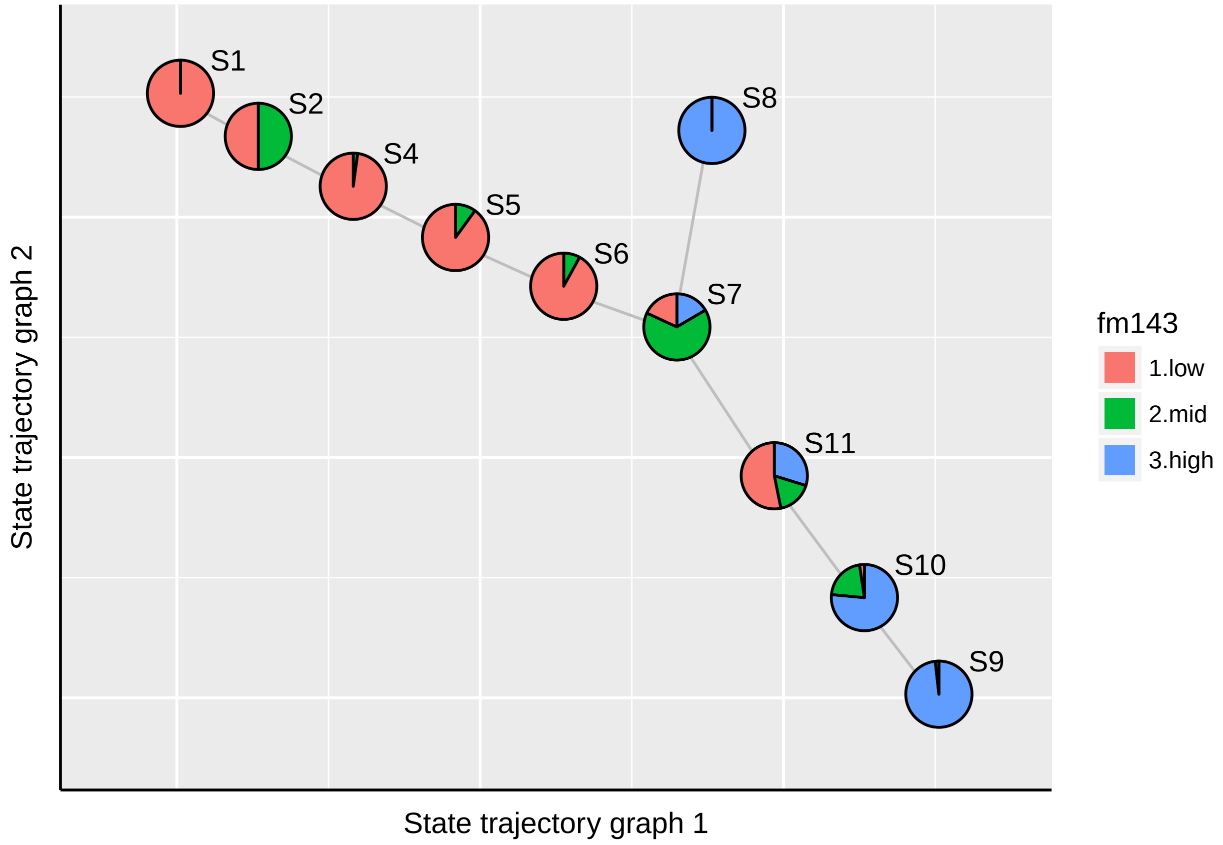
The function `trajComponents` provides us information about the states contained in each component.

```
# Show trajectory information
trajComponents(exBundle)
```

```
## [[1]]
##  [1] "S1"  "S2"  "S4"  "S5"  "S6"  "S7"  "S8"  "S9"  "S10" "S11"
##
## [[2]]
## [1] "S3"
```

Further, the inferred state trajectory graph can be visualized using `plotStateTrajectory`. If the graph is a forest, the parameter `component` can be used to define which tree should be shown. The optional parameters `point_size` and `label_offset` are useful to adjust the graph layout, the size of the points and the relative position of the point labels, respectively. Let's have a look at the FM1-43 uptake and the *CALB2* expression in component 1:

```
# FM1-43 uptake
plotStateTrajectory(exBundle, color_by="phenoName", name="fm143",
                    component=1, point_size=1.5, label_offset=4)
```

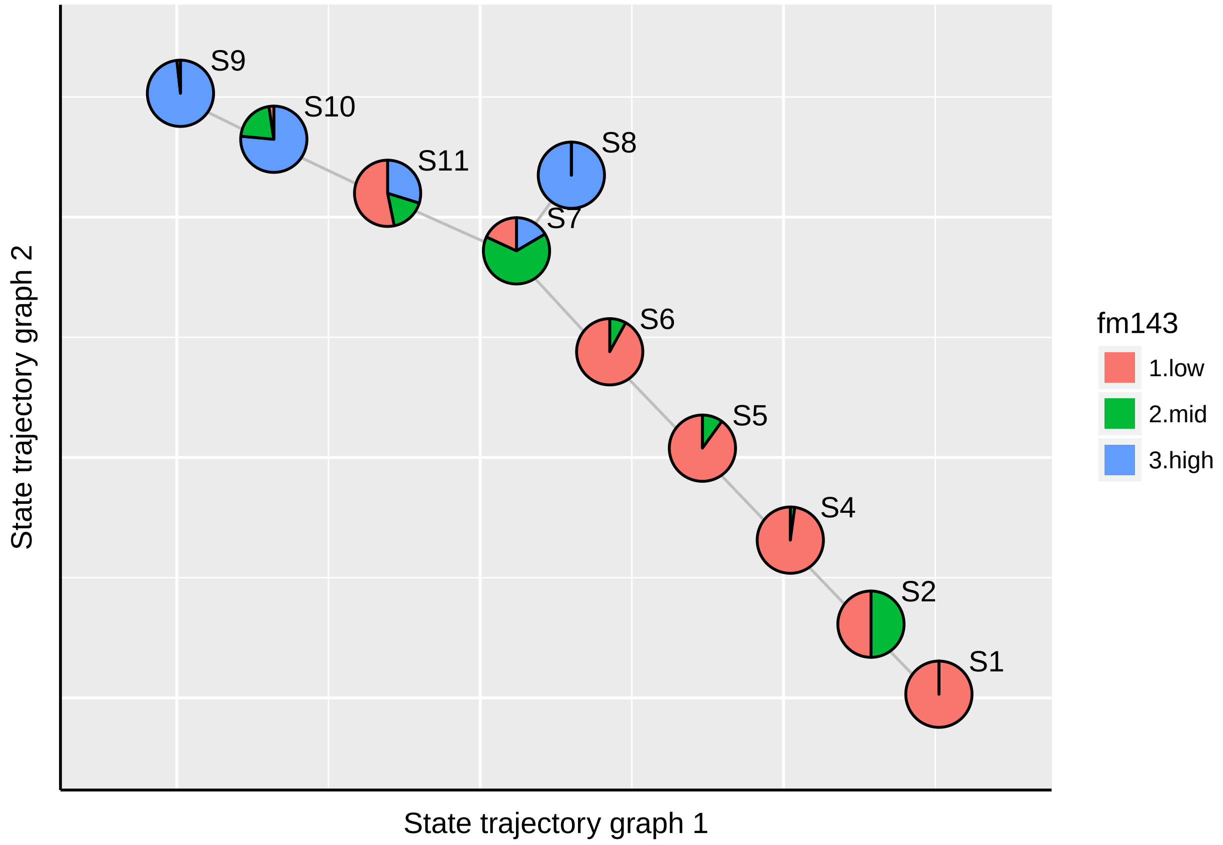


The `plotStateTrajectory` function uses the Fruchterman-Reingold graph layout algorithm (Fruchterman and Reingold, 1991) for visualization. If the user needs to re-compute the layout, it can be achieved by setting the parameter `recalculate=TRUE`. The new layout should then be stored to the *SingleCellExperiment* object.

```
# FM1-43 uptake
gp <- plotStateTrajectory(exBundle, color_by="phenoName", name="fm143",
                          component=1, point_size=1.5, label_offset=4,
                          recalculate=TRUE)
```

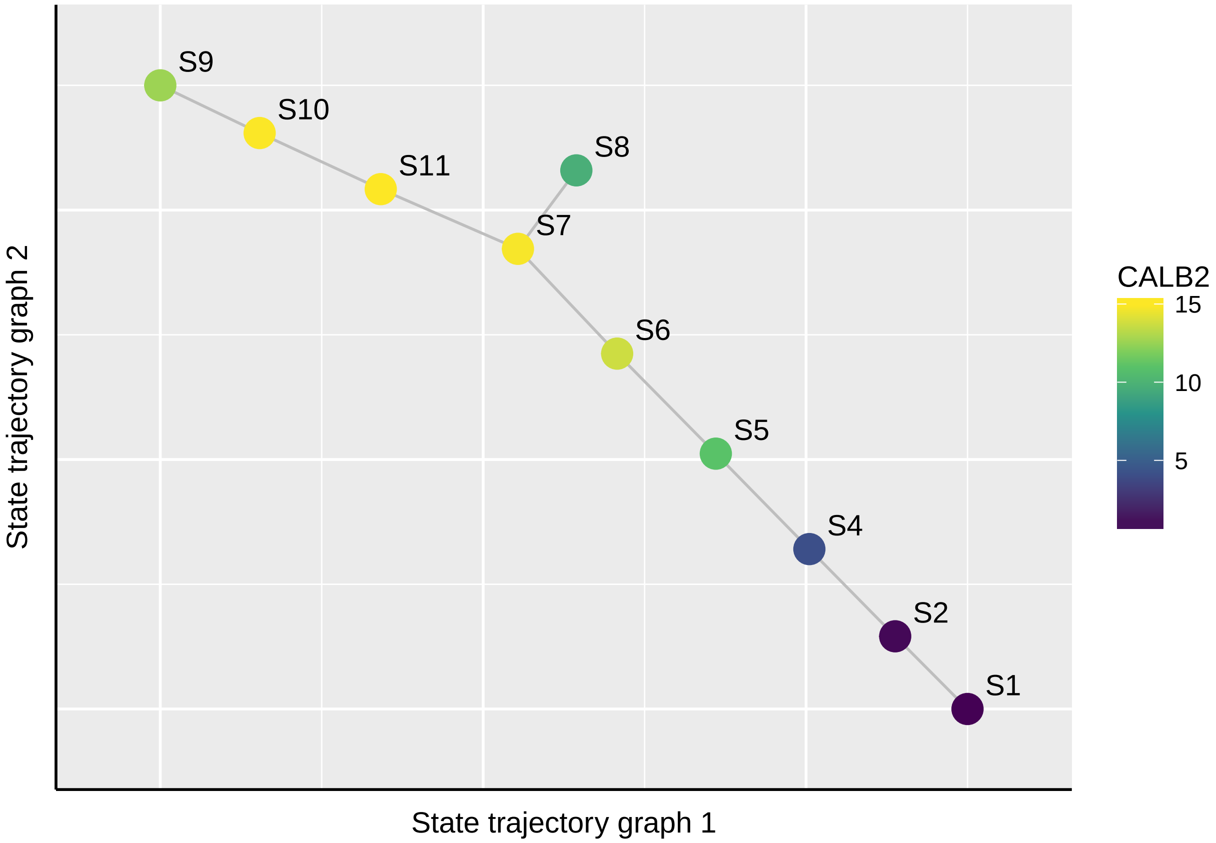
```
## Calculating layout of state trajectory graph ...
```

```
gp
```



```
# Store layout
stateTrajLayout(exBundle) <- gp

# CALB2 exoression
plotStateTrajectory(exBundle, color_by="featureName", name="CALB2",
                    component=1, point_size=5)
```



6.2 Aligning Samples Onto the Trajectory

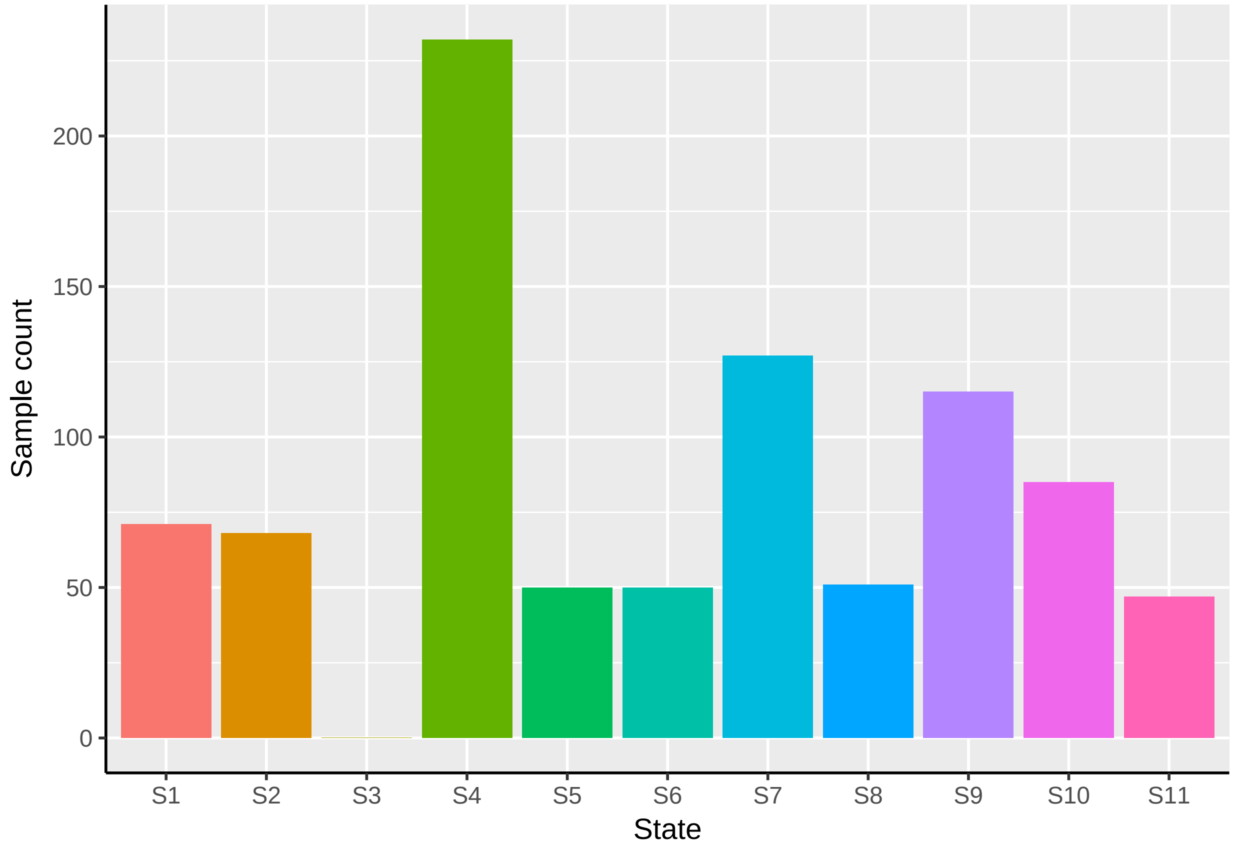
For the sake of simplicity and performance, it makes sense to conduct subsequent steps for each component individually. In this case, we select the tree formed by graph component 1 with 896 samples for our example data analysis.

```
# Select trajectory
exBundle <- selectTrajectory(exBundle, component=1)
```

The function `trajSampleNames` returns the names of the 896 samples which were selected for trajectory reconstruction. If further details or analyses are of interest for this set of samples exclusively, the *SingleCellExperiment* object can be subset.

```
# Subset SingleCellExperiment object by
# trajectory sample names
exBundle_subset <- exBundle[, trajSampleNames(exBundle)]

# Plot state sizes
plotStateSize(exBundle_subset)
```



As expected, the isolated state S3 (trajectory graph component 2) is not contained in this subset.

The selected graph component defines the trajectory backbone. The function `fitTrajectory` embeds the trajectory structure in the latent space by computing straight lines passing through the mediancentres (Bedall and Zimmermann, 1979) of adjacent states. Then, a fitting function is learned. Each sample is projected to its most proximal straight line passing through the mediancentre of its assigned state. Here, whenever possible, orthogonal projections onto line segments between two mediancentres are preferred to line segments which are only incident to a single median centre. Fitting deviations are given by the Euclidean distance between the sample's location and the straight line, and are indicated by an aggregated statistic (Mean Squared Error, MSE) shown by `showTrajInfo` and can be directly accessed via `trajResiduals`. Finally, a weighted acyclic trajectory graph can be constructed based on each sample's position along its straight line. Nodes in this graph are samples; edges are constructed between neighboring samples. Each edge is weighted by the distance between its nodes along the straight line.

```r
# Align samples onto trajectory
exBundle <- fitTrajectory(exBundle)
showTrajInfo(exBundle)
```

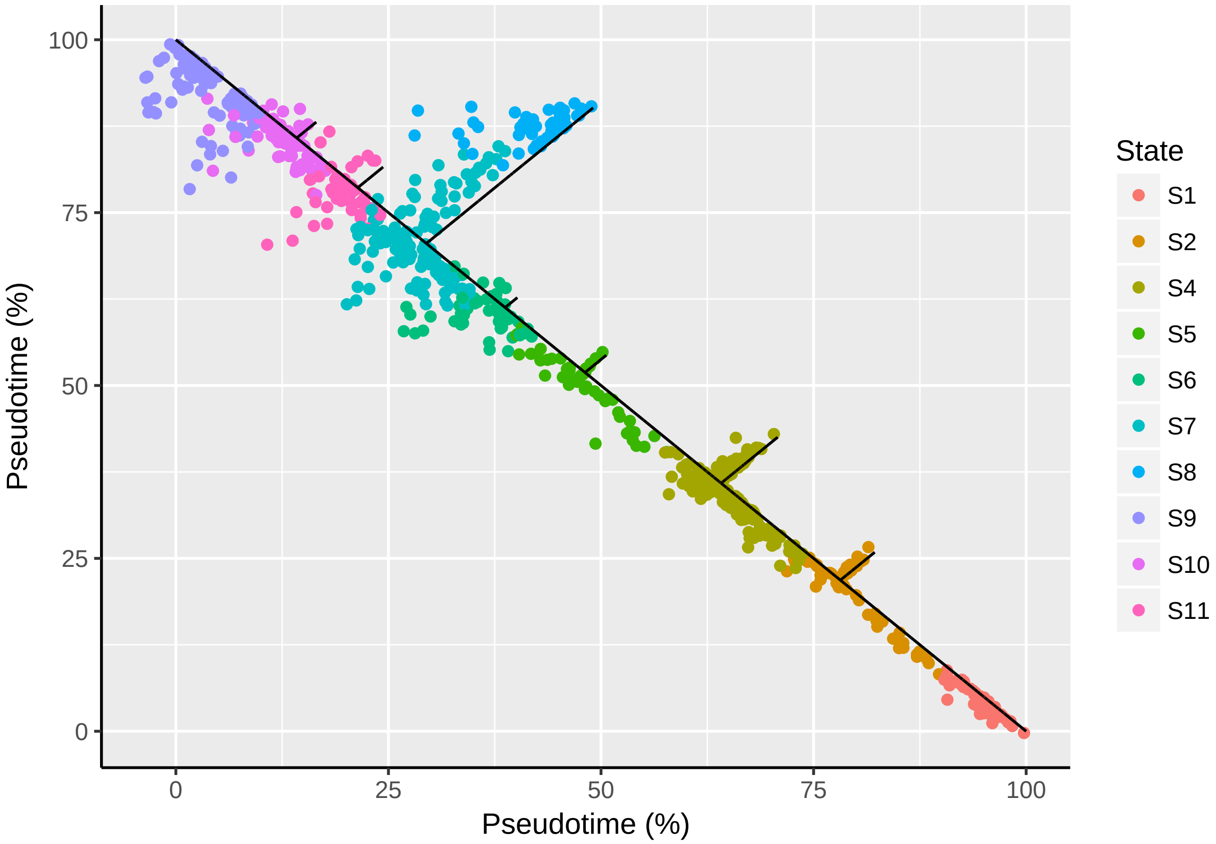
```
## [[ CellTrails ]]
## logcounts: 183 features, 1008 samples
## Pheno data:
##   sampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   phenoNames: "fm143" "origin" ... "landmark" (4)
## Feature data:
##   featureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   latentSpace: 1008 samples, 9 dimensions
##   states: "S1" "S2" ... "S11" (11)
## Trajectories: [Component(#Vertices,Edges)]: 1(10,9)
##   trajSampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (896)
##   trajResiduals: MSE=9.9e-03
##   landmarks:  #Branches=7 #Terminals=9 #User=0
##   trajLayout: none
## Trail data:
##   trailNames: none
```

```
trajResiduals(exBundle)[1:5]
```

```
## [1] 0.008616594 0.008064917 0.004659595 0.039835749 0.017354939
```

Of note, the fitting function implies potential side branches in the trajectory graph; those could be caused due to technical variance or encompass samples that were statistically indistinguishable from the main trajectory given the selected features used for trajectory reconstruction. The `plotTrajectoryFit` function shows the trajectory backbone (longest shortest path between two samples) and the fitting deviations of each sample indicated by the perpendicular jitter.

```
plotTrajectoryFit(exBundle)
```



Chapter 7

CellTrails Maps

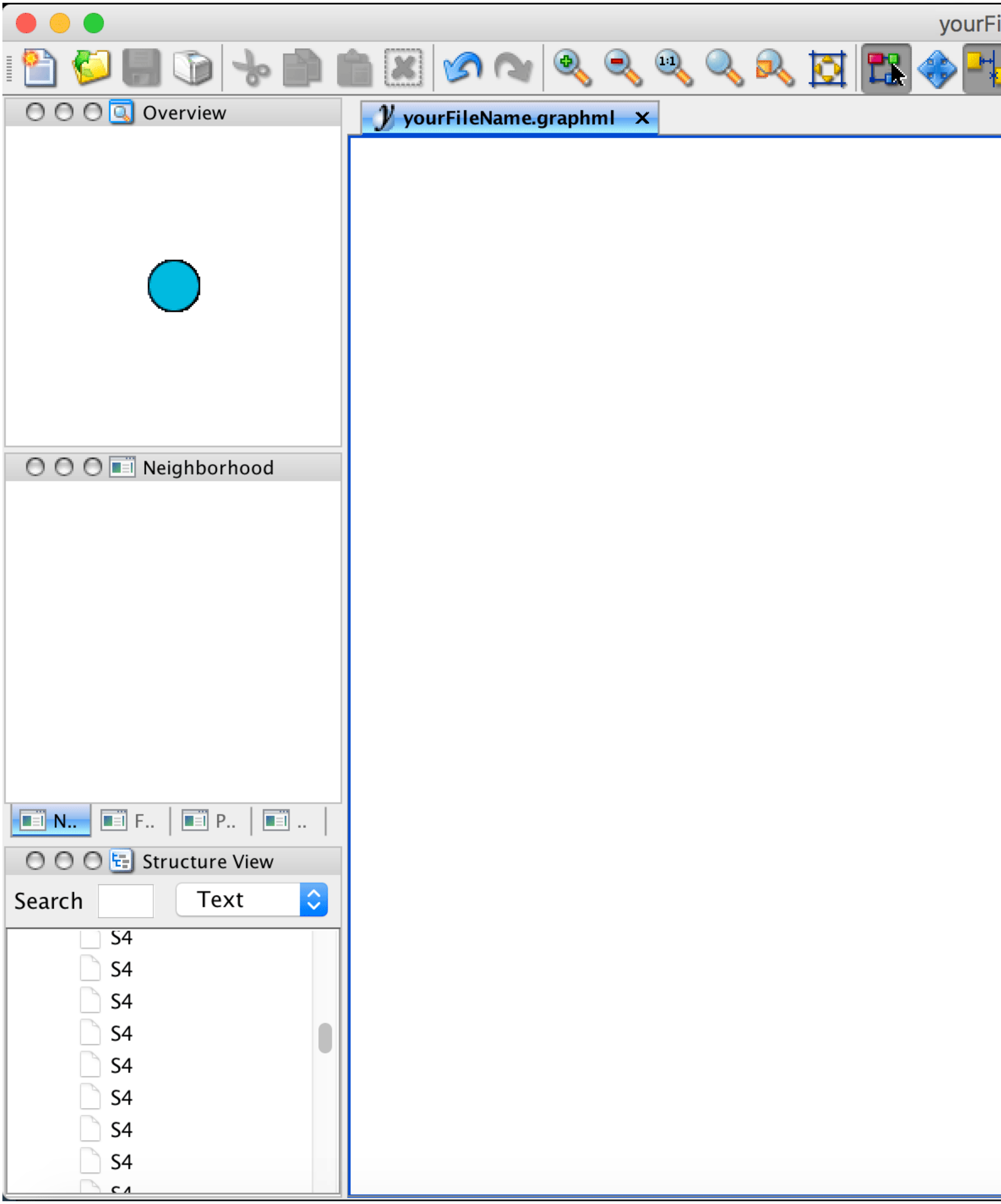
7.1 Graph Layout

CellTrails portrays a computed trajectory as collection of trails that can be found in a landscape shaped by individual expression dynamics. To generate such a topographic trail map - the *CellTrails* map - a two-dimensional spatiotemporal ordination of the expression matrix has to be computed. This can be done by any graph layout algorithm using the structural information from the trajectory graph, which is composed of nodes (=samples) and edges (=chronological relation between samples). We found that the freely available graph visualization software *yEd* (http://www.yworks.com/products/yed) has great capabilities to visualize and analyze a trajectory graph. An optimal layout is planar, i.e. it exhibits no crossing edges or overlapping nodes.

Therefore, *CellTrails* enables to export and import the trajectory graph structure using the common *graphml* file format. This file format can be interpreted by most third-party graph analysis applications, allowing the user to subject the trajectory graph to a wide range of (tree) layout algorithms. In particular, the exported file has additional ygraph attributes best suited to be used with *yEd*, which is freely available for all major platforms (Windows, Mac OS, and Linux).

```
write.ygraphml(exBundle, file="yourFileName.graphml")
```

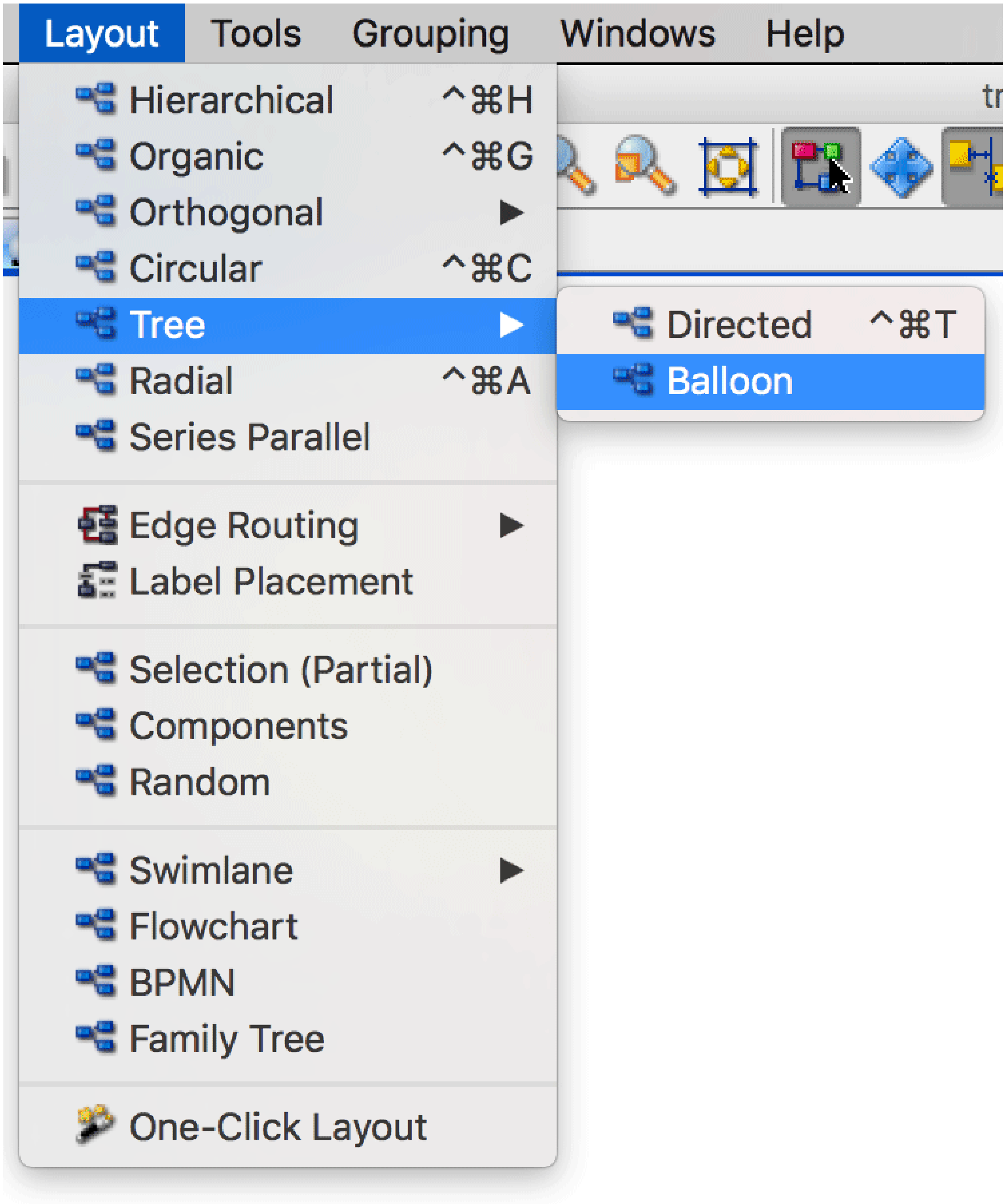
Let's open the exported graphml file in *yEd*:

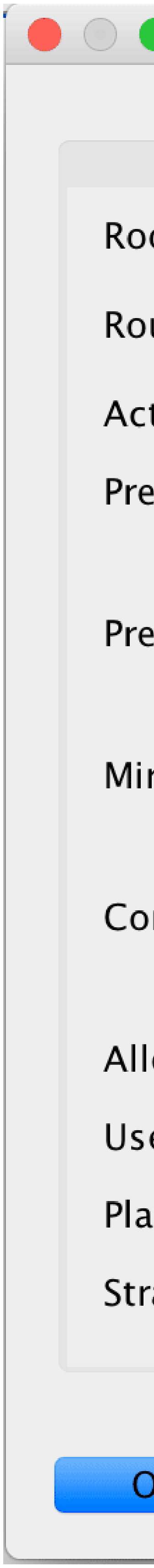

yourFi
Overview
yourFileName.graphml
Neighborhood
N..
F..
P..
..
Structure View
Search
Text
S4
S4
S4
S4
S4
S4
S4
S4

If a layout has already been defined for a *SingleCellExperiment* object, the samples' coordinates will be listed in the exported graphml file and will be directly interpreted by *yEd*. In this example, no layout was defined yet and therefore, all samples (nodes) are overlapping.

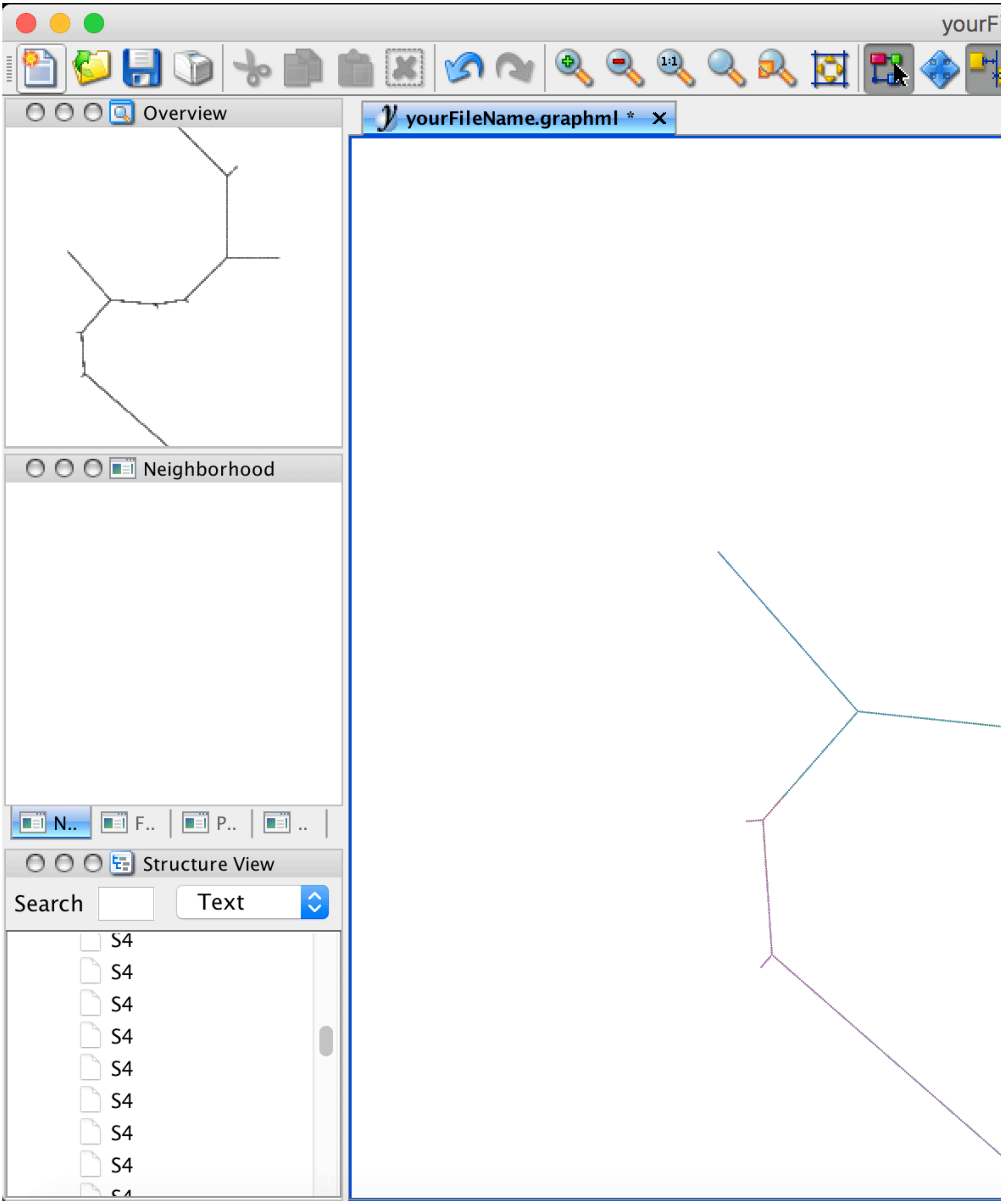
Please note that the export function `write.ygraphml` colorizes automatically nodes by their assigned state. However, it is possible to colorize and label nodes by any phenotype or feature expression data stored in the *SingleCellExperiment* object. For example, we could colorize the nodes by the recorded FM1-43 dye intensity to get an idea where the trajectory might start and end (a high FM1-43 dye indicates mature cells) and label the nodes by their determined state by setting the parameters `color_by="phenoName"`, `name="fm143"` and `label="state"`. Nodes with a missing phenotype information are not colorized and remain transparent.

Next, we layout the graph. Since the trajectory resembles a tree structre, we use a tree algorithm. We found that routing the trajectory graph in a quasi-radial style, called balloon style, works very well for our case. The layouter can be selected via *Layout* → *Tree* → *Balloon*. The following parameter setting was used in the original *CellTrails* publication:

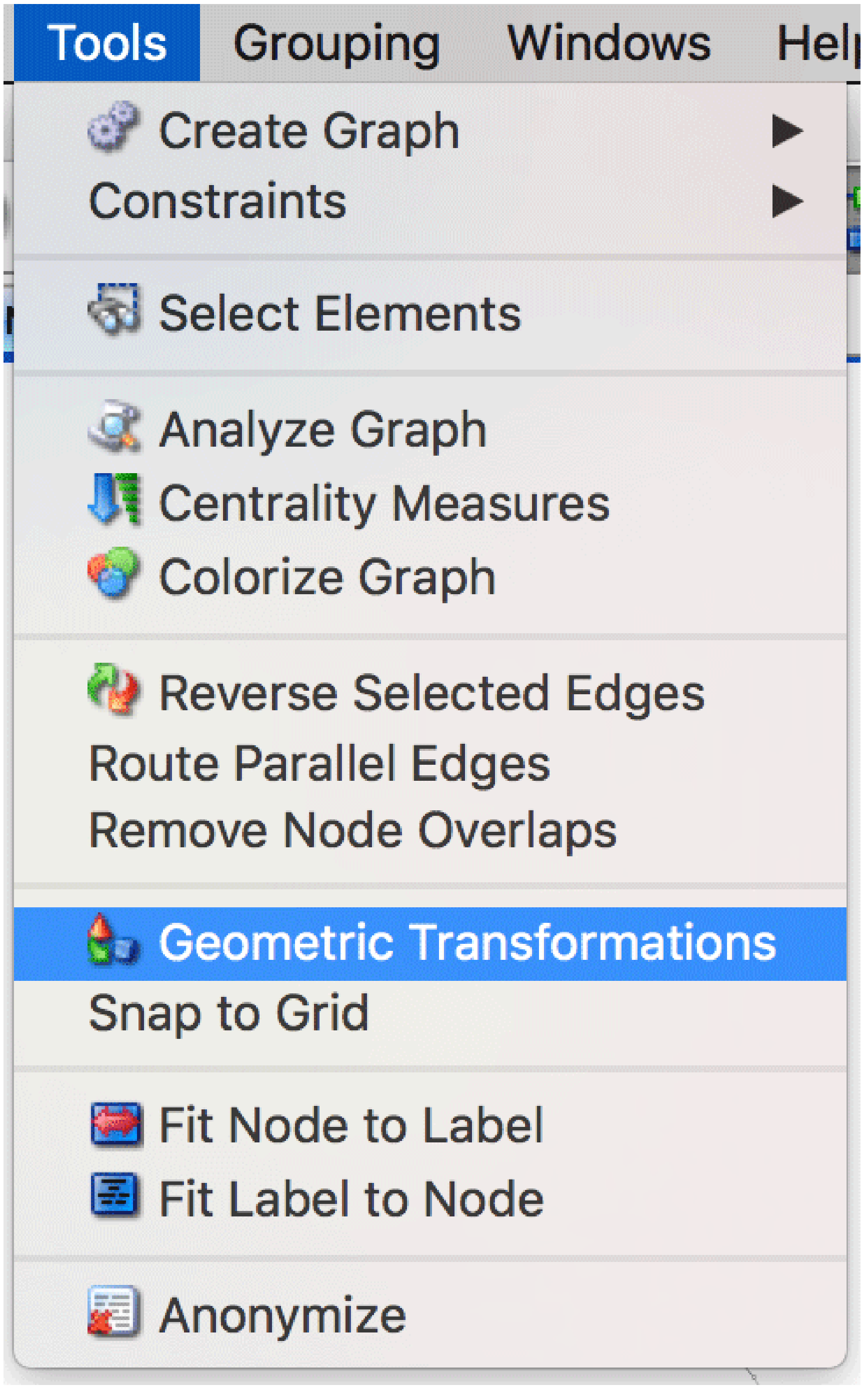

Layout
Tools
Grouping
Windows
Help
Hierarchical ^⌘H
Organic ^⌘G
Orthogonal ▶
Circular ^⌘C
Tree ▶
Radial ^⌘A
Series Parallel
Edge Routing ▶
Label Placement
Selection (Partial)
Components
Random
Swimlane ▶
Flowchart
BPMN
Family Tree
One-Click Layout
Directed ^⌘T
Balloon
tr

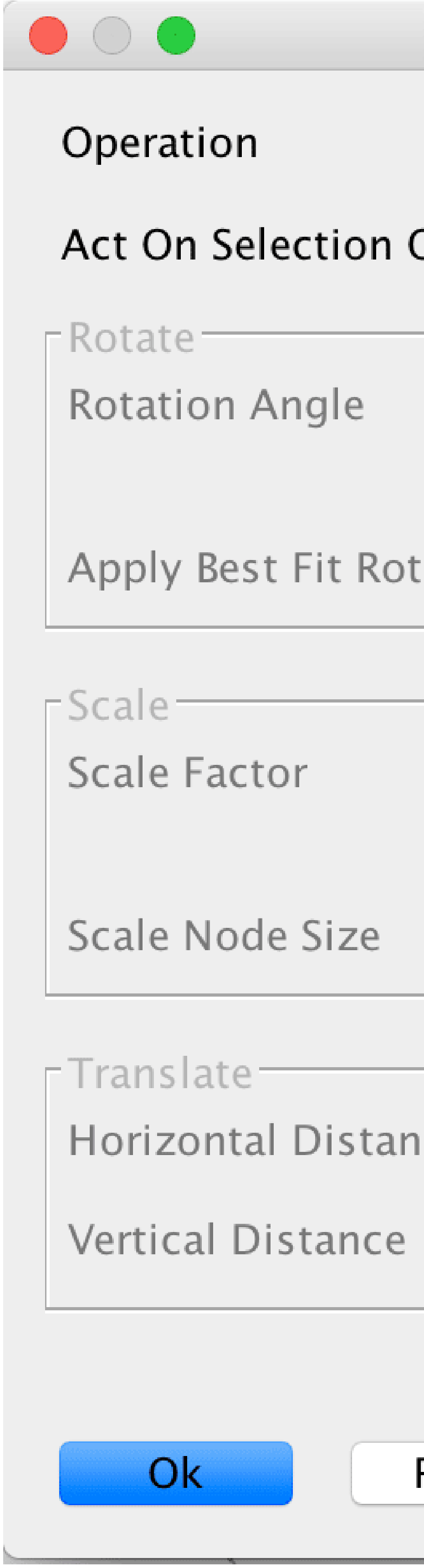

Roo
Rou
Act
Pre
Pre
Mir
Co
All
Use
Pla
Str
O

Let’s run the algorithm to compute the layout:


yourF
Overview
yourFileName.graphml *
Neighborhood
N..
F..
P..
..
Structure View
Search
Text
S4
S4
S4
S4
S4
S4
S4
S4

Please note that edge crossings (i.e., two or more edges overlap) are not useful and if they occur we suggest to re-run the layouter with different parameters before saving the layout. Using either your mouse or the *View* → *Zoom In* option allows to have a closer look. If we want to have the trajectory progressing from bottom left to bottom right (based on a specific feature expression or phenotype label), we need to transform the graph. This can be done via *Tools* → *Geometric transformations*. Here, we select *Mirror on Y axis* and *Mirror on X axis*.


Tools
Grouping
Windows
Help
Create Graph ▶
Constraints ▶
Select Elements
Analyze Graph
Centrality Measures
Colorize Graph
Reverse Selected Edges
Route Parallel Edges
Remove Node Overlaps
Geometric Transformations
Snap to Grid
Fit Node to Label
Fit Label to Node
Anonymize


Operation
Act On Selection C
Rotate
Rotation Angle
Apply Best Fit Rot
Scale
Scale Factor
Scale Node Size
Translate
Horizontal Distan
Vertical Distance
Ok
R

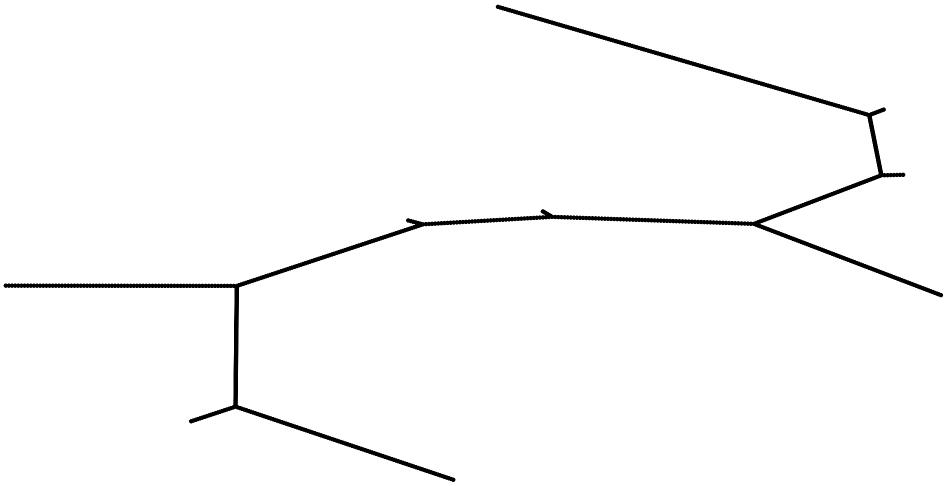
Finally, the file can be saved via *File* → *Save* and reimported to R.

```
tl <- read.ygraphml("yourFileName.graphml")
```

For illustration purposes, the computed trajectory layout for the example dataset is available within this package.

```
# Import layout
tl <- read.ygraphml(system.file("exdata", "bundle.graphml",
                                package="CellTrails"))

# Plot layout
plot(tl[,1:2], axes=FALSE, xlab="", ylab="", pch=20, cex=.25)
```



Finally, we set the trajectory layout to the *SingleCellExperiment* object using the `trajLayout` function. Here, the parameter `adjust` indicates if edge lengths should be adjusted, such that they correspond to the inferred pseudotime.

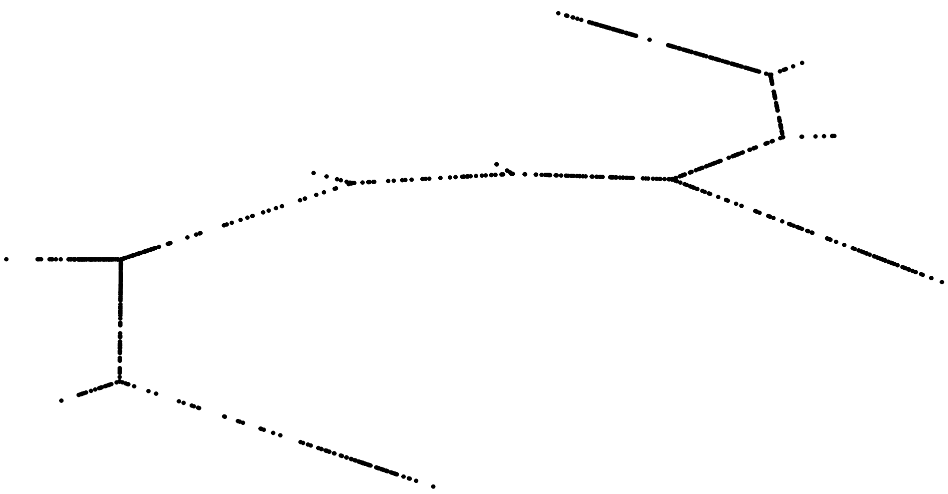
```
# Adjust layout and store to object
trajLayout(exBundle, adjust=TRUE) <- tl

showTrajInfo(exBundle)
```

```
## [[ CellTrails ]]
## logcounts: 183 features, 1008 samples
## Pheno data:
##   sampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   phenoNames: "fm143" "origin" ... "landmark" (4)
## Feature data:
##   featureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   latentSpace: 1008 samples, 9 dimensions
##   states: "S1" "S2" ... "S11" (11)
## Trajectories: [Component(#Vertices,Edges)]: 1(10,9)
##   trajSampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (896)
##   trajResiduals: MSE=9.9e-03
##   landmarks:  #Branches=7 #Terminals=9 #User=0
##   trajLayout: available
## Trail data:
```

```
##   trailNames: none
```

```
# Plot adjusted layout
plot(trajLayout(exBundle), axes=FALSE, xlab="", ylab="", pch=20, cex=.25)
```

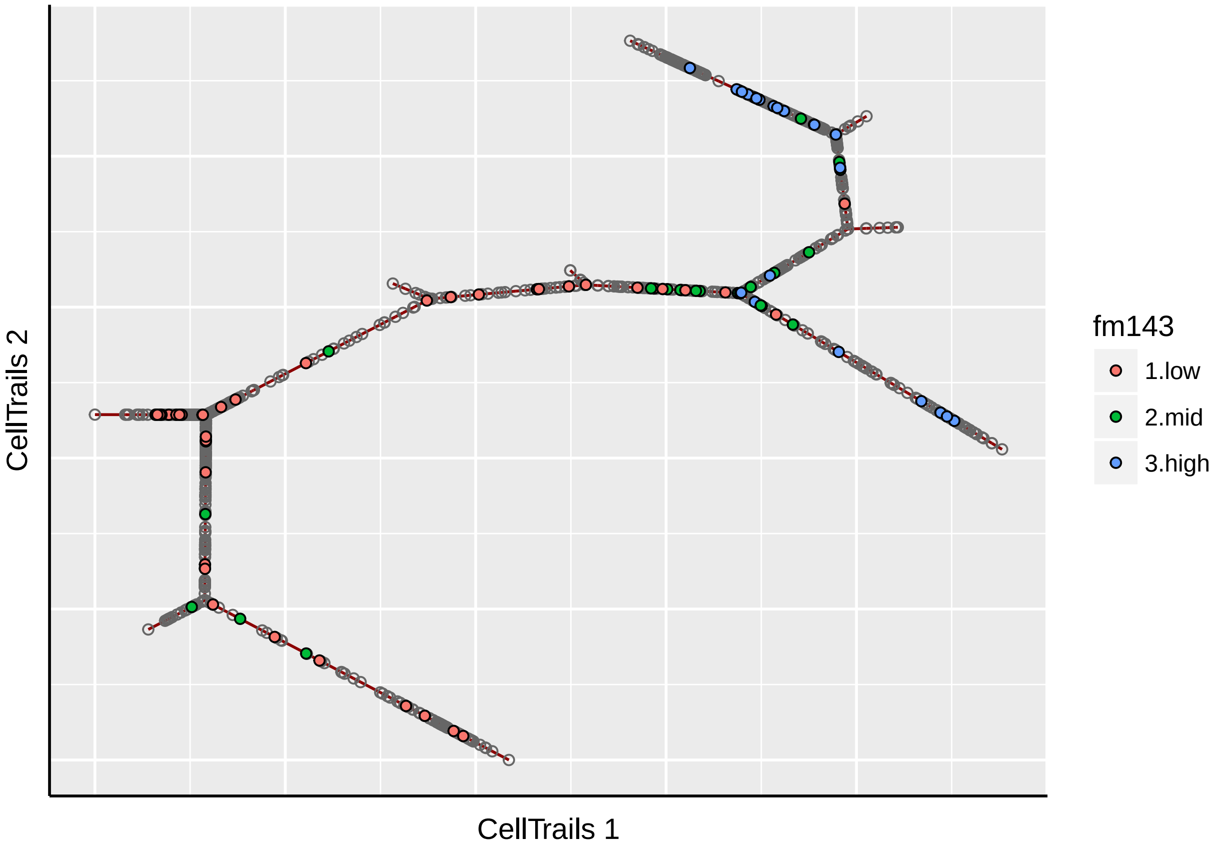


Sidenote. It is not a requirement to use yEd. The trajectory layout can also be defined by the user otherwise. The minimum requirement is, that the coordinates of each sample are stored in a `data.frame` whose row names correspond to the trajectory sample names. The sample names can be pulled from the *SingleCellExperiment* object using the function `trajSampleNames`. The layout can then be set using the accessor function `trajLayout` accordingly.

7.2 Plot Maps

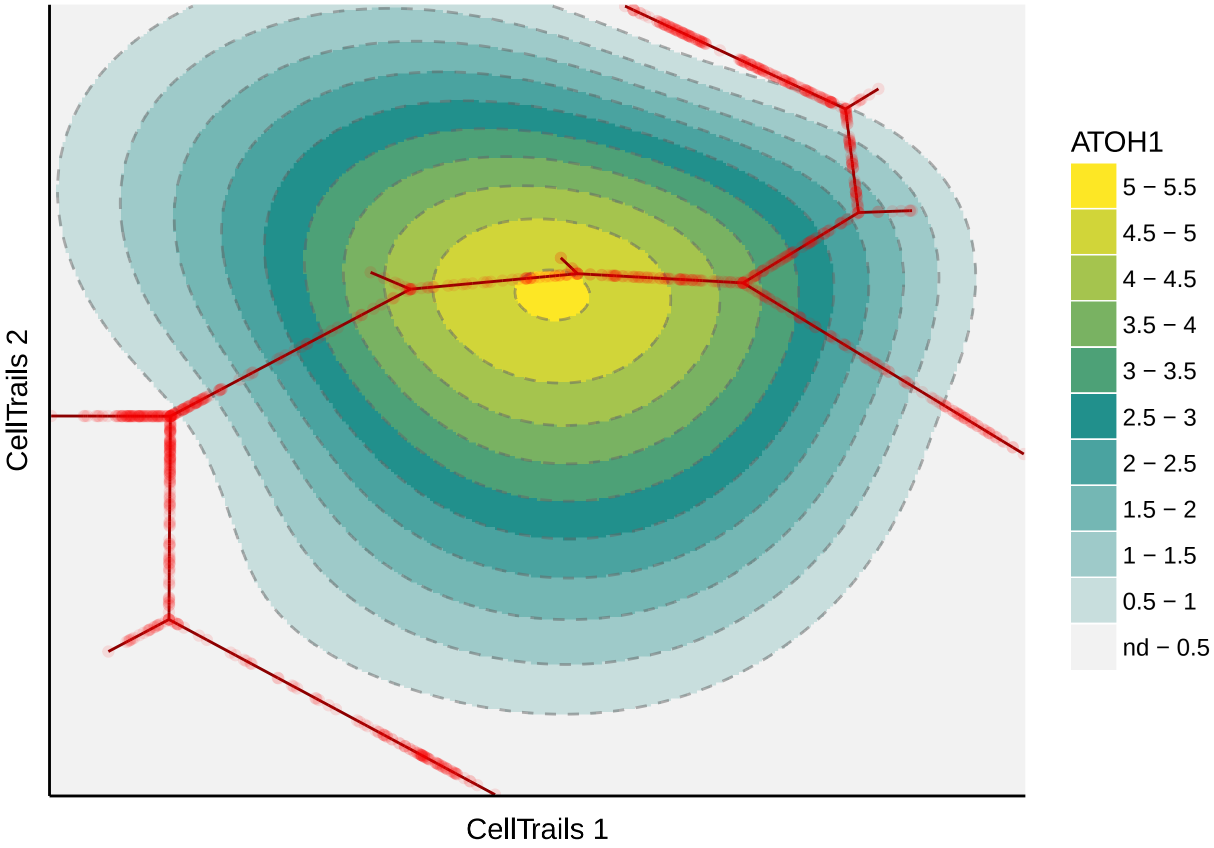
After generating the layout, a two-dimensional visualization of the trajectory can be drawn. Here, the line represents the trajectory and individual points represent the samples. This plot type either colorizes individual samples by a metadata label (as available via `phenoNames`) or it shows the topography of a given feature's expression landscape (as available via `featureNames`). When metadata are being visualized, the grey line represents the trajectory and the individual points represent samples. Samples that do not have a specific metadata label or a missing value are not shown. Let's visualize how the cellular FM1-43 uptake distributes along the trajectory:

```
plotMap(exBundle, color_by="phenoName", name="fm143")
```



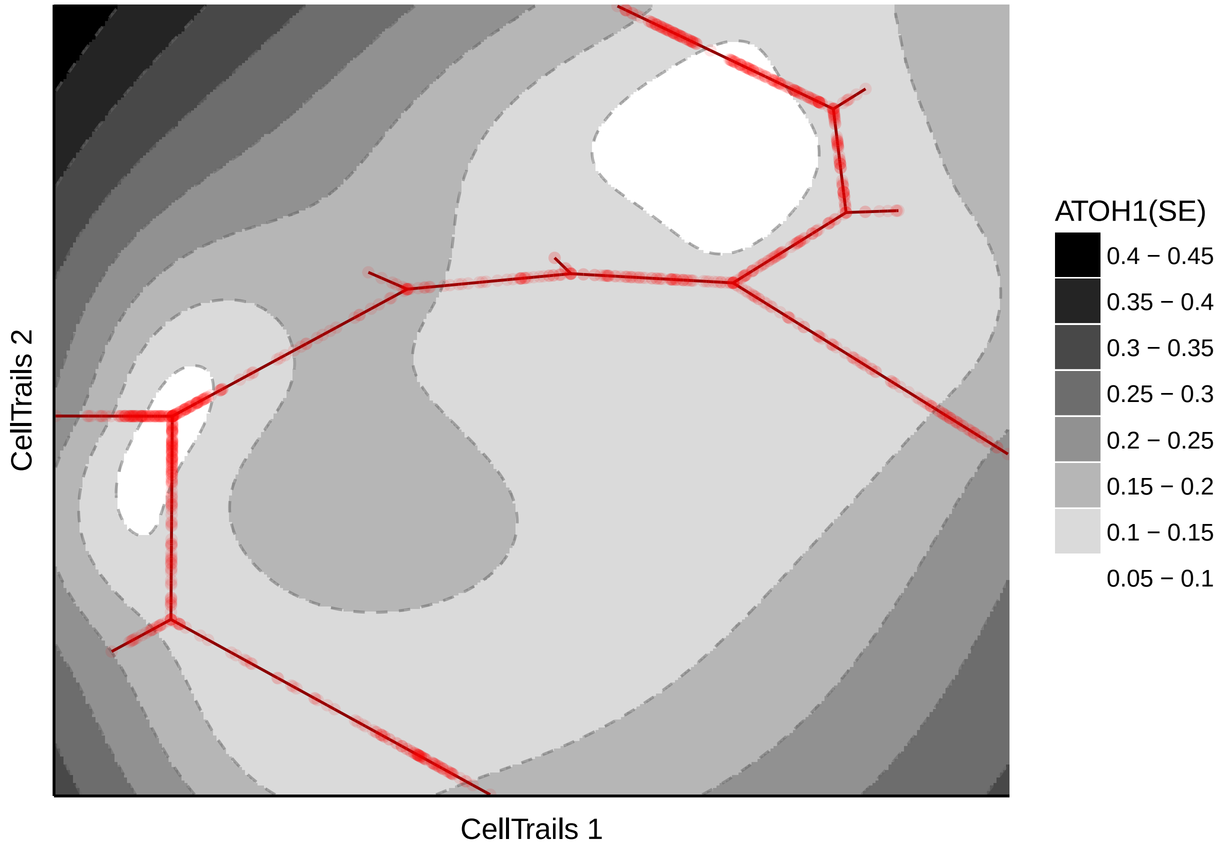
In the topographical plot, a smooth surface is fitted and values are predicted for a regular grid resulting in the shown map topography; the red line signifies the trajectory. Let's take a view into the *ATOH1* expression landscape, an early key transcription factor during sensory hair cell development:

```r
plotMap(exBundle, color_by="featureName", name="ATOH1", type="surface.fit")
```



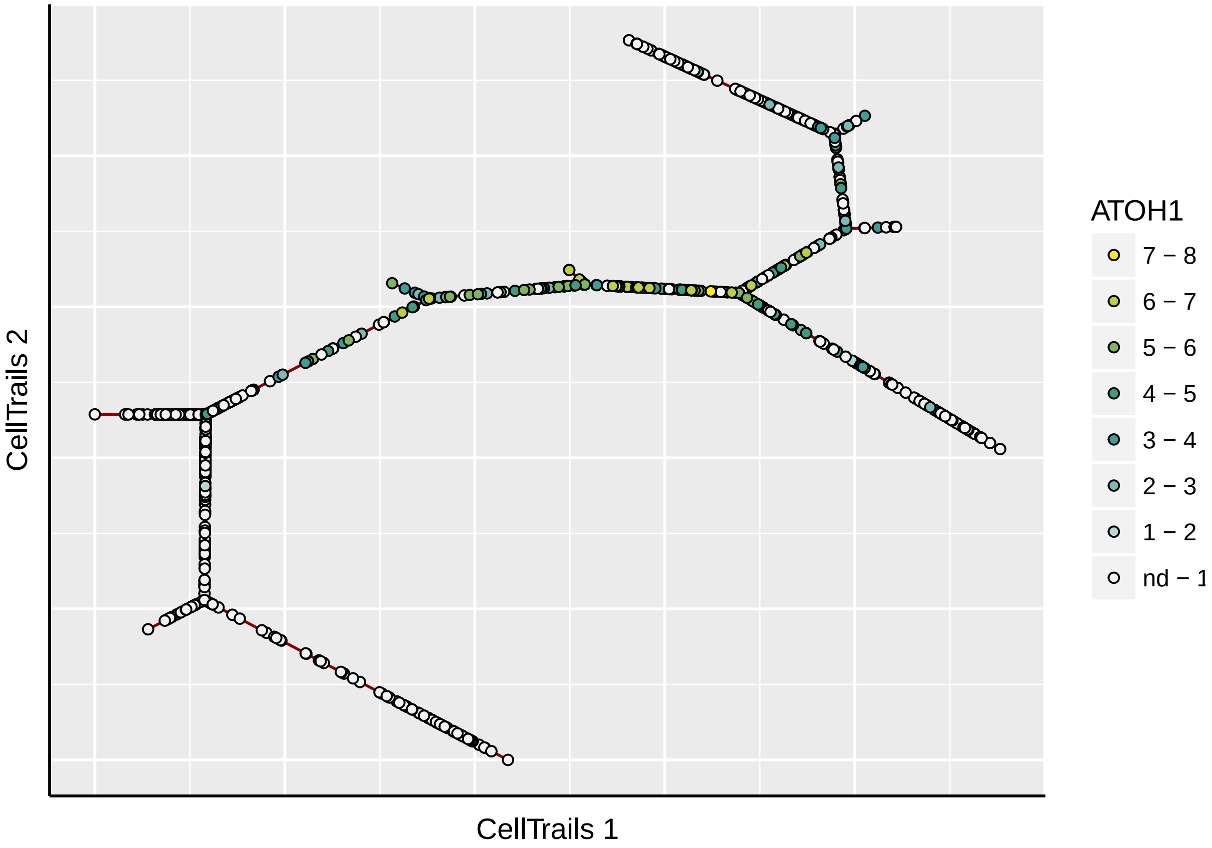
Let's have a look into the standard error of the predicted expression surface by setting the parameter `type`.

```
plotMap(exBundle, color_by="featureName", name="ATOH1", type="surface.se")
```

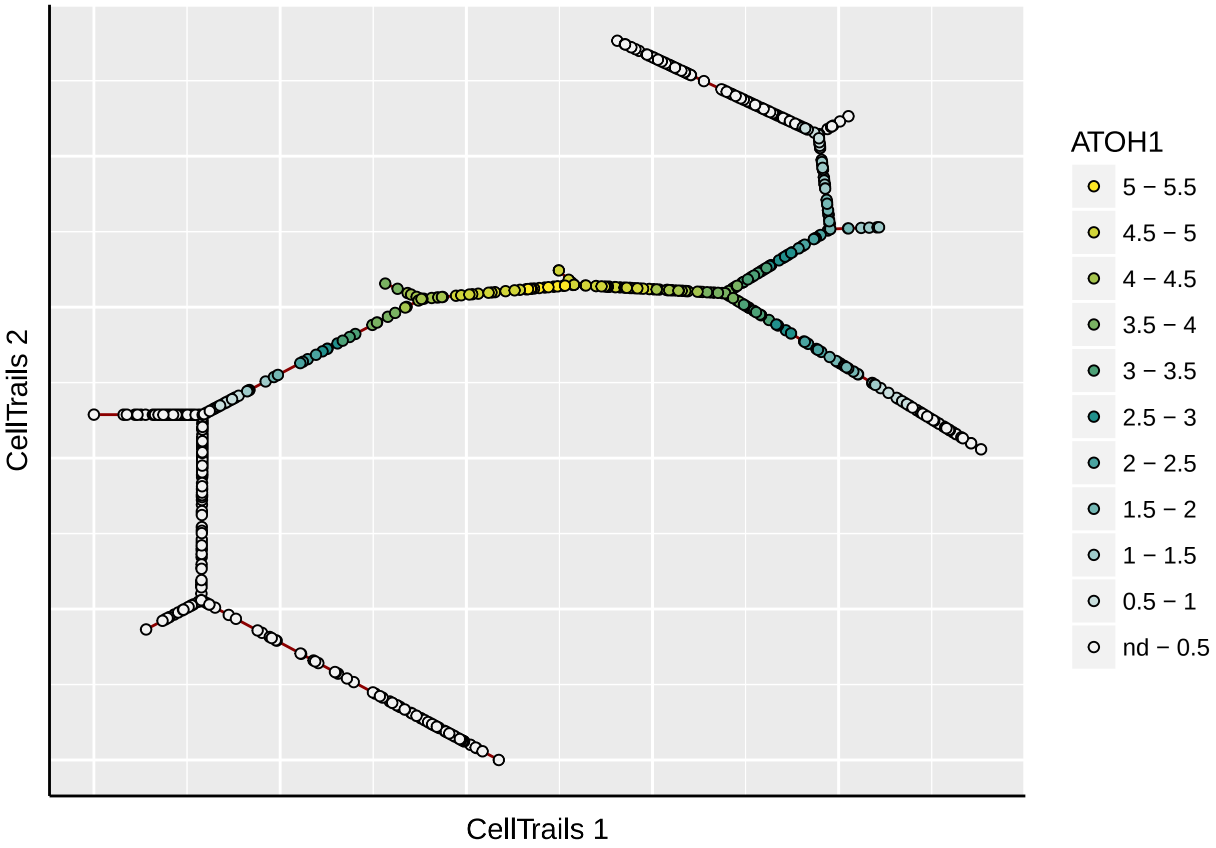


Alternatively, *CellTrails* enables to show the samples only, instead of the whole fitted expression surface. Here, we have two options: either the raw expression or the smoothed values.

```
# Raw
plotMap(exBundle, color_by="featureName", name="ATOH1", type="raw")
```



```r
# Smoothed
plotMap(exBundle, color_by="featureName", name="ATOH1", type="surface.fit",
        samples_only=TRUE)
```


CellTrails 2
CellTrails 1
ATOH1
5 − 5.5
4.5 − 5
4 − 4.5
3.5 − 4
3 − 3.5
2.5 − 3
2 − 2.5
1.5 − 2
1 − 1.5
0.5 − 1
nd − 0.5

Chapter 8

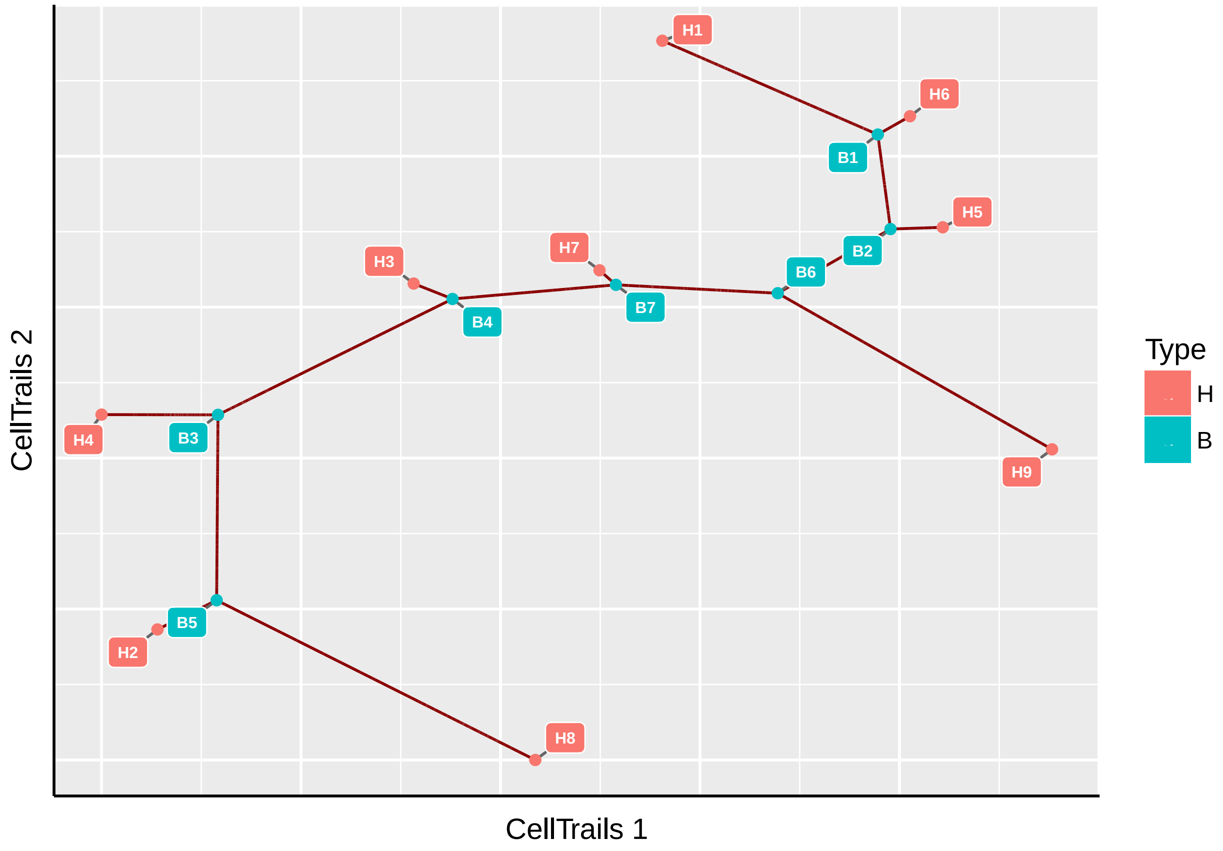
Expression Dynamics

During the trajectory fitting process, landmarks are automatically identified on the trajectory: trail heads (leafs), H, and branching points, B. The assigned landmark IDs can be obtained via `landmarks`. We use this information to define individual trails along the trajectory.

8.1 Trail Definition

Trails are usful to infer expression dynamics of features along subsections of the trajectory. A trail denotes a path between two landmarks. To be able to properly define a trail, we display the available landmark points on the trajectory map.

```
plotMap(exBundle, color_by="phenoName", name="landmark")
```



Based on the experimental metainformation and the expression pattern of marker features, we identified in the original *CellTrails* publication path *B3* to *H9* as developmental trail toward a striolar sensory hair bundle morphology, and *B3* to *H1* as developmental trail toward an extrastriolar bundle morphology. Let's mark those trails on the map using the function addTrail.

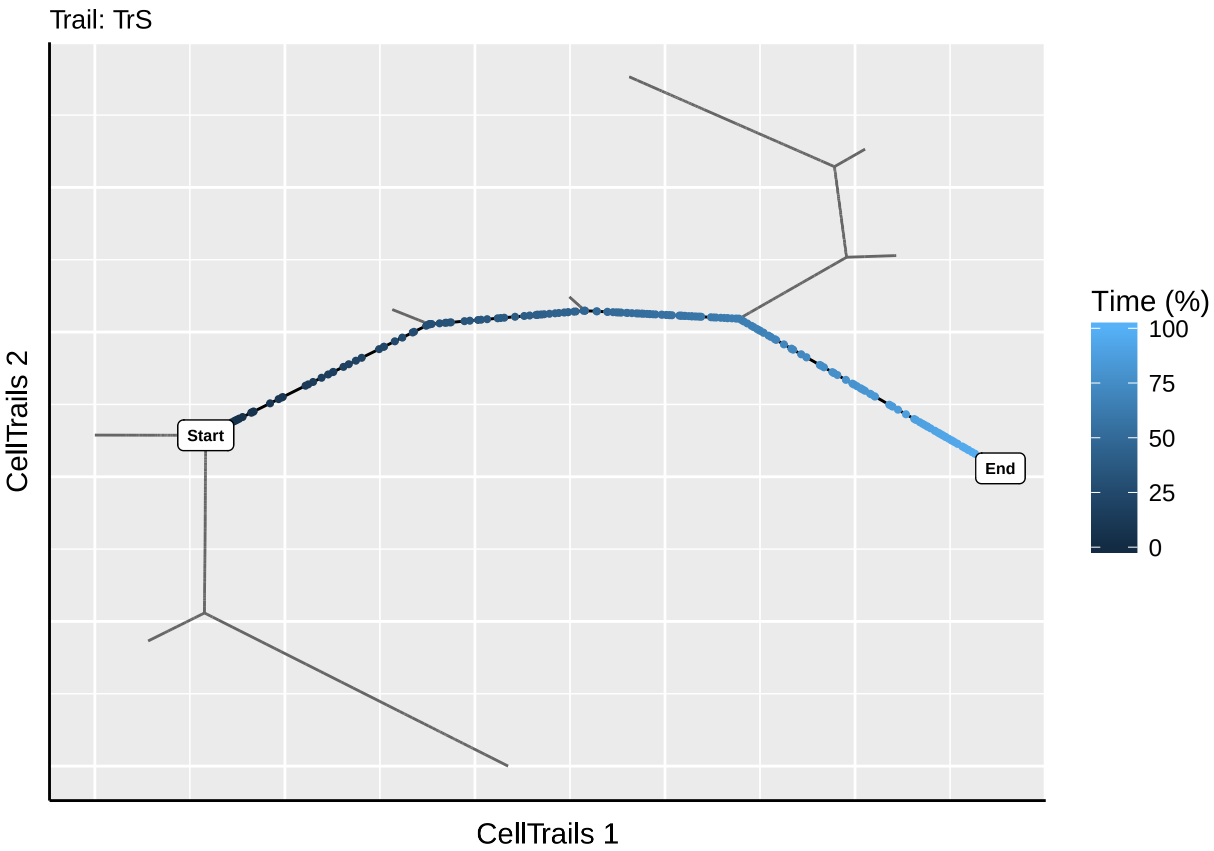
```
# Define trails
exBundle <- addTrail(exBundle, from="B3", to="H9", name="TrS")
exBundle <- addTrail(exBundle, from="B3", to="H1", name="TrES")

showTrajInfo(exBundle)
```

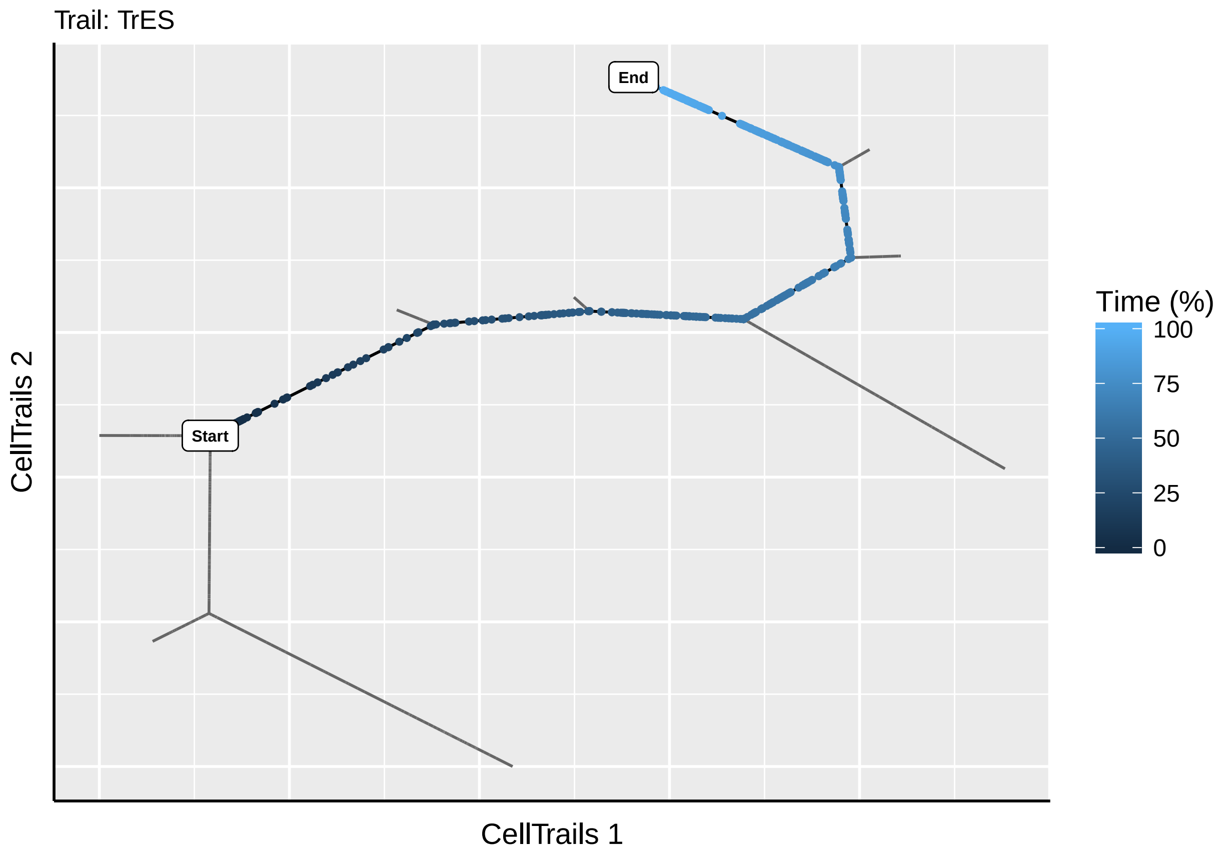
```
## [[ CellTrails ]]
## logcounts: 183 features, 1008 samples
## Pheno data:
##   sampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   phenoNames: "fm143" "origin" ... "landmark" (6)
## Feature data:
##   featureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   latentSpace: 1008 samples, 9 dimensions
##   states: "S1" "S2" ... "S11" (11)
## Trajectories: [Component(#Vertices,Edges)]: 1(10,9)
##   trajSampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (896)
##   trajResiduals: MSE=9.9e-03
##   landmarks:  #Branches=7 #Terminals=9 #User=0
##   trajLayout: available
## Trail data:
##   trailNames: "TrS" "TrES" (2)
```

Next, we want to make sure that the intended trails were extracted by showing the trajectory map and highlight the defined trails along with its corresponding pseudotime.

```
plotTrail(exBundle, name="TrS")
```



```
plotTrail(exBundle, name="TrES")
```



The function `addTrail` automatically extracts the samples and their pseudotime along the trail by computing the shortest path between the trail start and end.

```
# Get trail names
trailNames(exBundle)
```

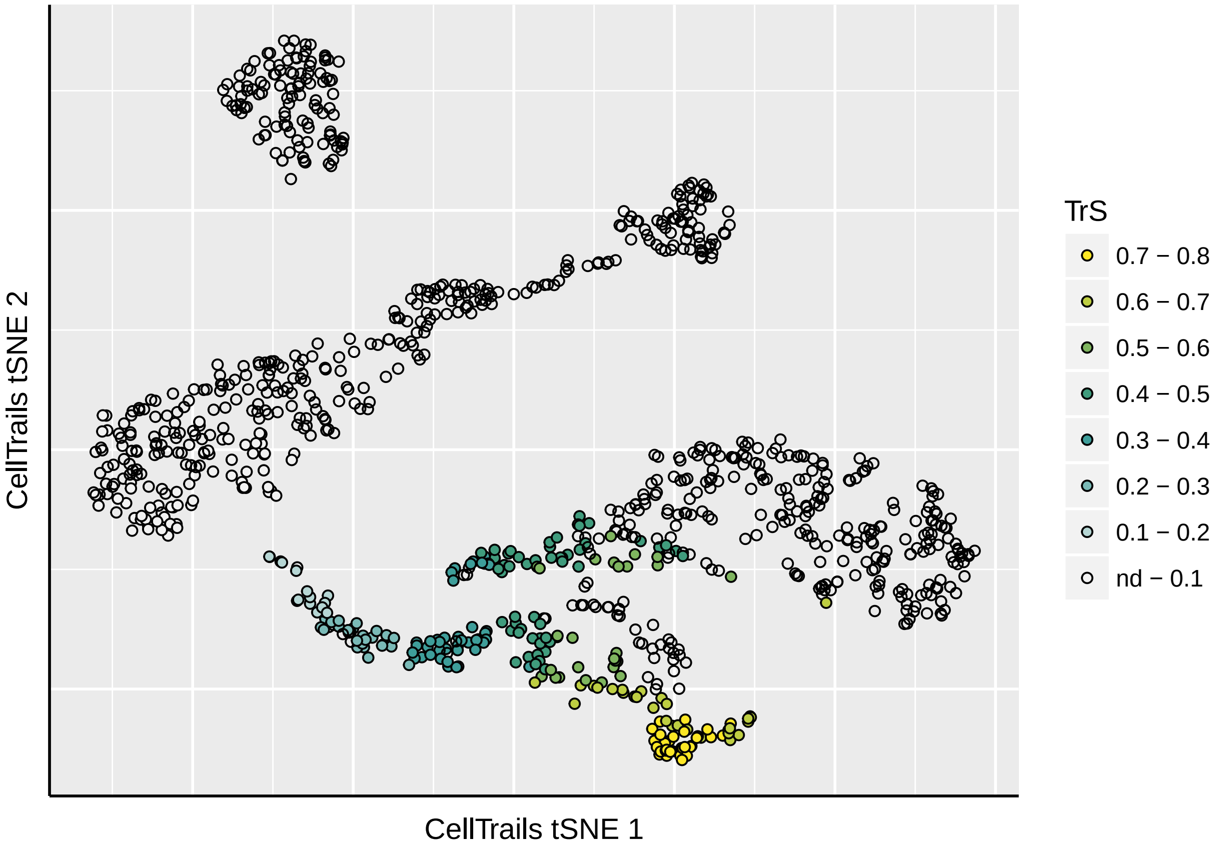
```
## [1] "TrS"  "TrES"
```

```
# Get trail pseudotime
trails(exBundle)[1:5, ]
```

```
## DataFrame with 5 rows and 2 columns
##                 TrS      TrES
##           <numeric> <numeric>
## Cell-1-1 0.5776028        NA
## Cell-1-2        NA        NA
## Cell-1-3        NA        NA
## Cell-1-4        NA        NA
## Cell-1-5        NA 0.8723274
```

The pseudotime information is automatically stored as sample metadata (see `phenoNames`). For example, we could plot it on the lower-dimensional manifold.

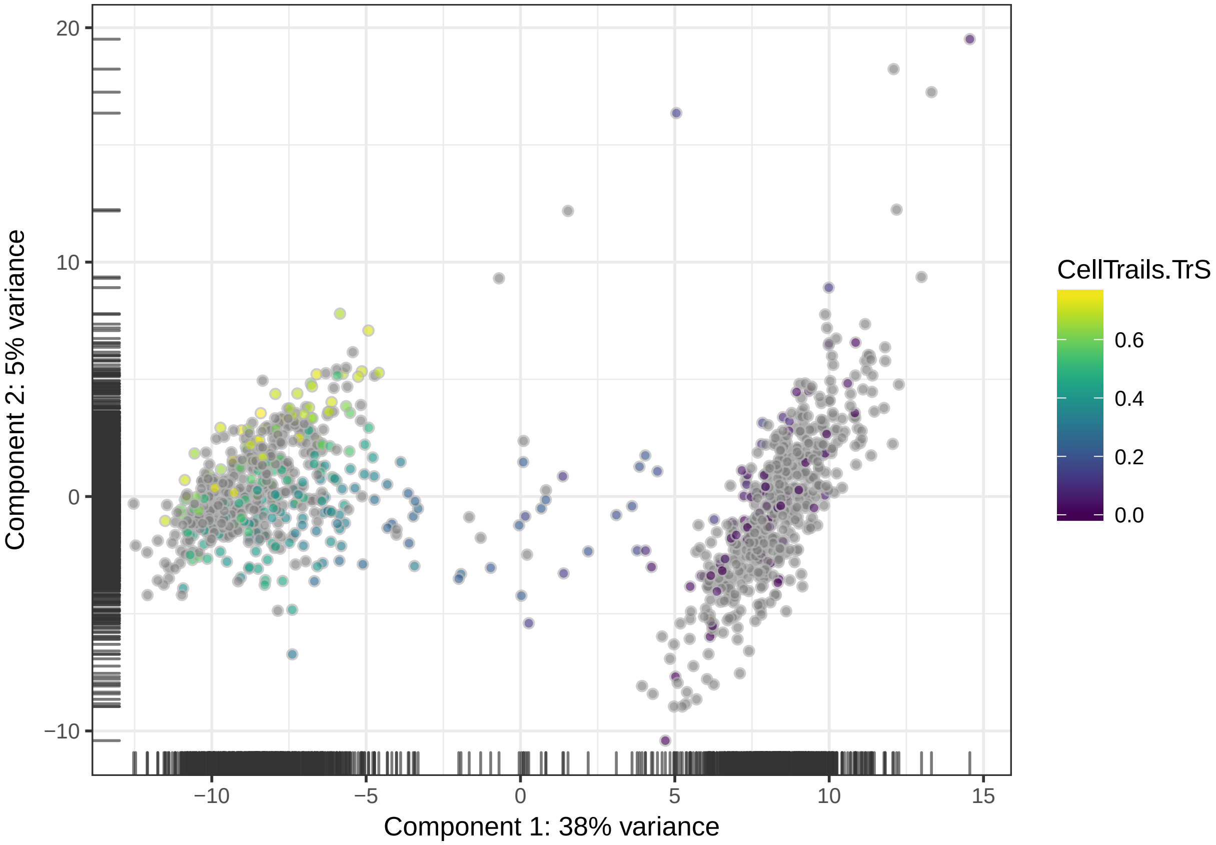
```
# Get trail names
plotManifold(exBundle, color_by="phenoName", name="TrS")
```



Also, the data is accessible via `colData` of the *SingleCellExperiment* object and can therefore be analyzed using alternative packages. For example, colorizing the trail TrS in a principal component analysis using the *scater* package (McCarthy et al., 2017).

```
## Not run:
##library(scater)
## End(Not run)

# Plot scater PCA with CellTrails pseudotime information
scater::plotPCA(exBundle, colour_by="CellTrails.TrS")
```



Please note that trails can be renamed with `trailNames<-` and removed with `removeTrail`, respectively. Adding another trail with the same name, will show a warning message and override the existing definition.

8.2 Defining Subtrails

It might be needed to define subtrails if trails overlap. This is neccessary if the dynamics of one trail are subdynamics of another trail. Because pseudotime mirrors the location of each datapoint in the latent space, a significant gap in pseudotime could indicate separate sample populations. However, these populations have only subtle feature expression profile differences and were linearly aligned in the latent space. Since pseudotime can also be interpreted as a function of transcriptional change, one can argue that these populations undergo the same expression program (for the selected features), with the small but distinct difference that samples ordered at the terminal end of the longer trail up- or down-regulate additional features late during their maturation. Thus, trails can overlap, while one trail is a subtrail of the longer trail.

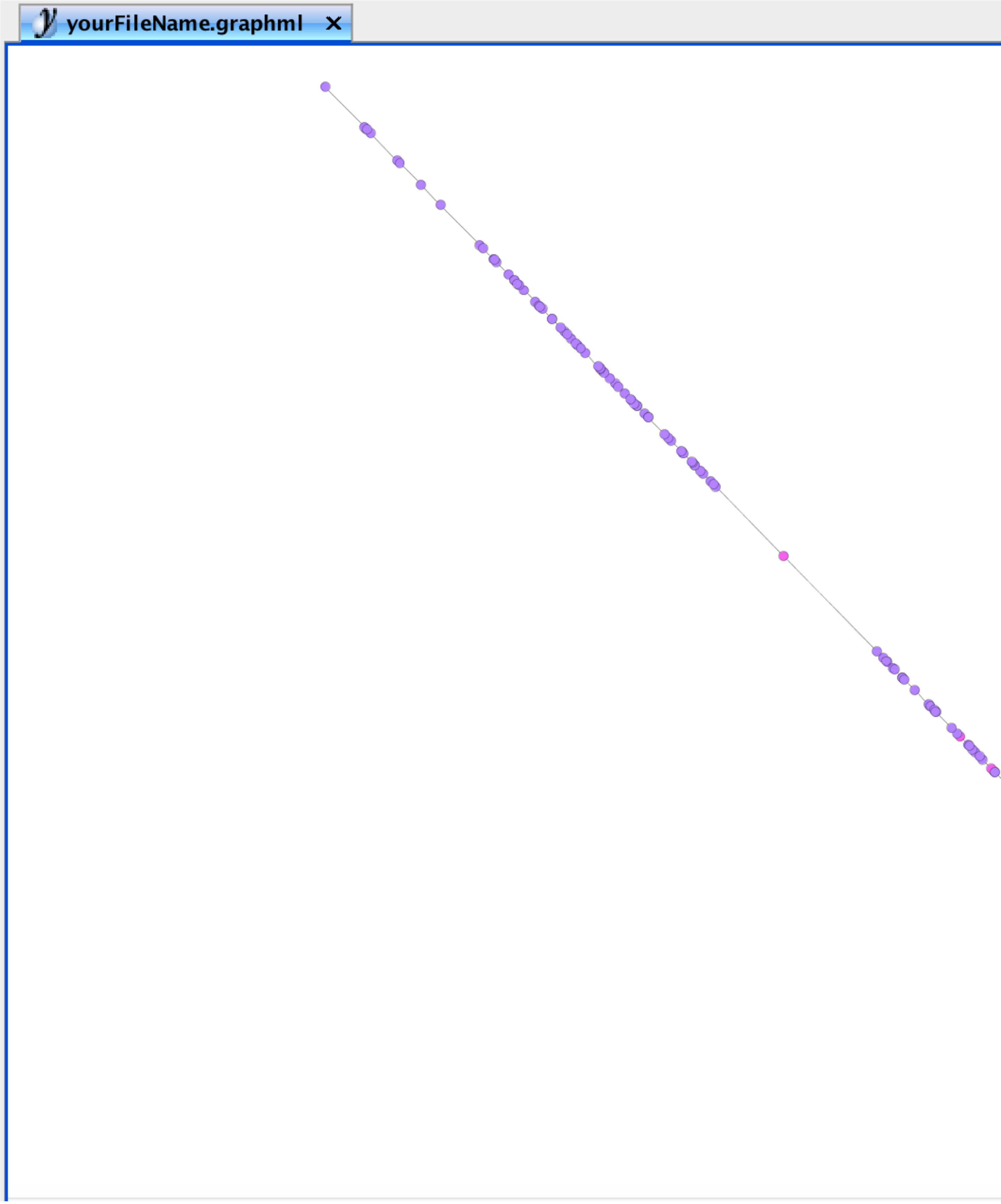
8.2.1 Using yEd

U landmarks that are needed to define a subtrail can be determined by the user, as demonstrated in the following.

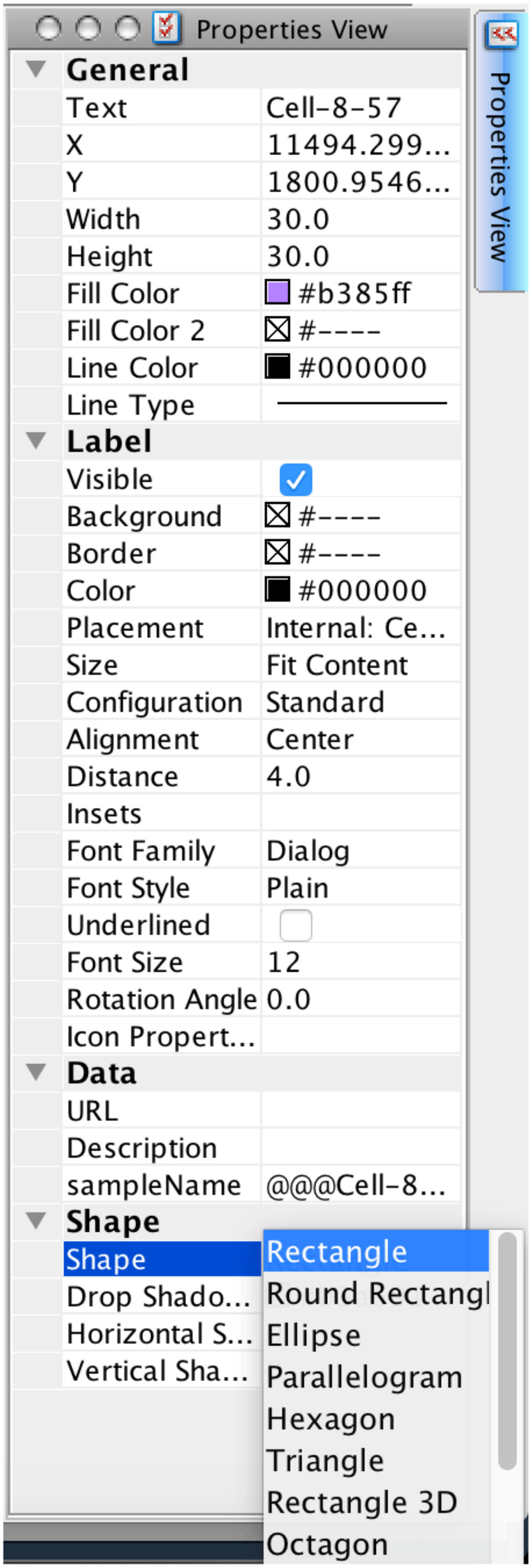
First, we want to give a rational for selecting a specific node. As described in the original *CellTrails* article, we found a gap in pseudotime near the terminal end of trail TrES, which might indicate that the terminal state can be split, and two trails are actually overlapping. This gap becomes already quite obvious visually when we utilize *yEd* to have a closer look into the trajectory graph. First, we export the graph. By default, nodes are colorized by state (see *CellTrails Maps / Graph Layout*).

```
write.ygraphml(exBundle, file='yourFileName.graphml')
```

Then we open the graphml file in *yEd*. The gap in the purple colored population is obvious:


yourFileName.graphml

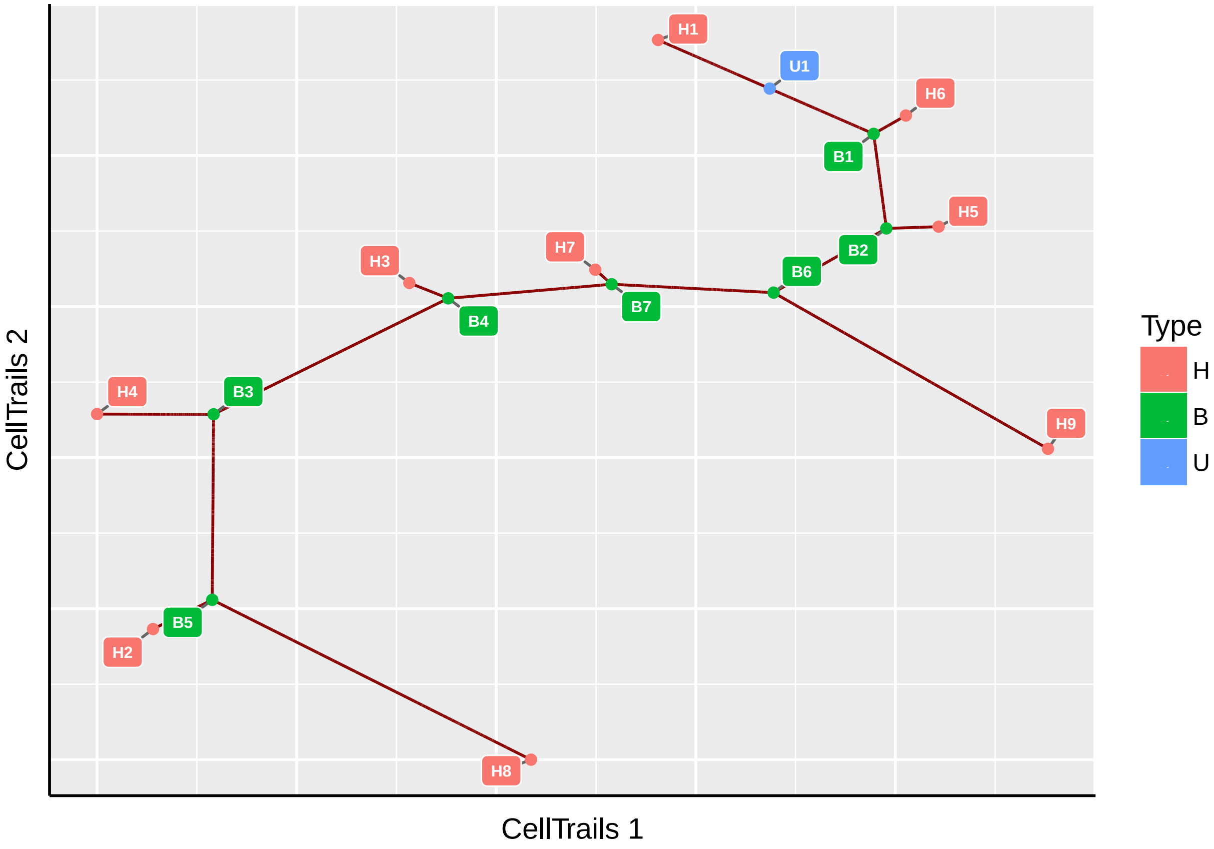
To indicate this sample as landmark, we simply change the shape of this node. This can be any shape, but not *ellipse*, which is used as default for other nodes. The shape can be changed using the *Properties View* panel on the right border of the *yEd* application.



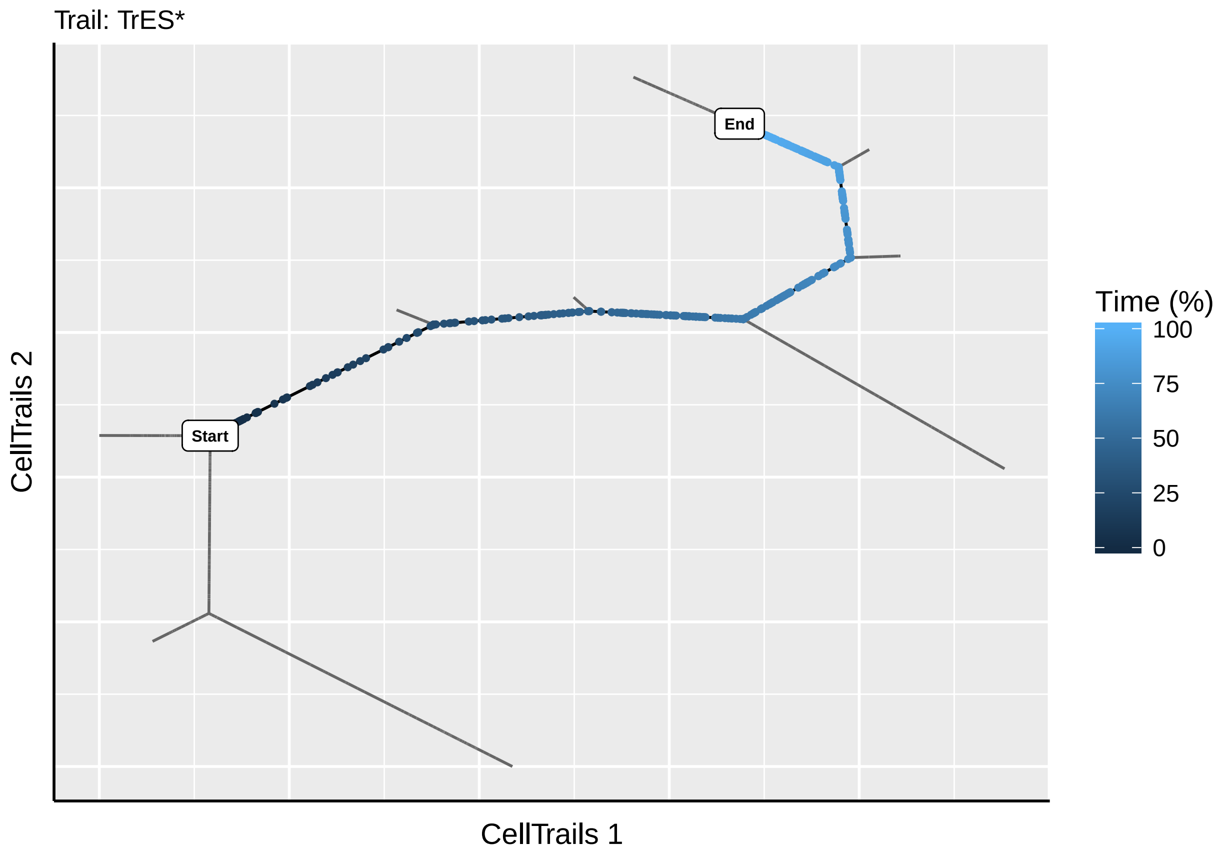
General	
Text	Cell–8–57
X	11494.299...
Y	1800.9546...
Width	30.0
Height	30.0
Fill Color	▨ #b385ff
Fill Color 2	☒ #----
Line Color	■ #000000
Line Type	———
Label	
Visible	☑
Background	☒ #----
Border	☒ #----
Color	■ #000000
Placement	Internal: Ce...
Size	Fit Content
Configuration	Standard
Alignment	Center
Distance	4.0
Insets	
Font Family	Dialog
Font Style	Plain
Underlined	□
Font Size	12
Rotation Angle	0.0
Icon Propert...	
Data	
URL	
Description	
sampleName	@@@Cell–8...
Shape	
Shape	Rectangle
Drop Shado...	
Horizontal S...	
Vertical Sha...	

After saving the layout, it can be reimported to *CellTrails* and the landmark can be used to define the subtrail:

```
# Trail Identification
plotMap(exBundle, color_by="phenoName", name="landmark")
```



```
exBundle <- addTrail(exBundle, from="B3", to="U1", name="TrES*")
plotTrail(exBundle, name="TrES*")
```

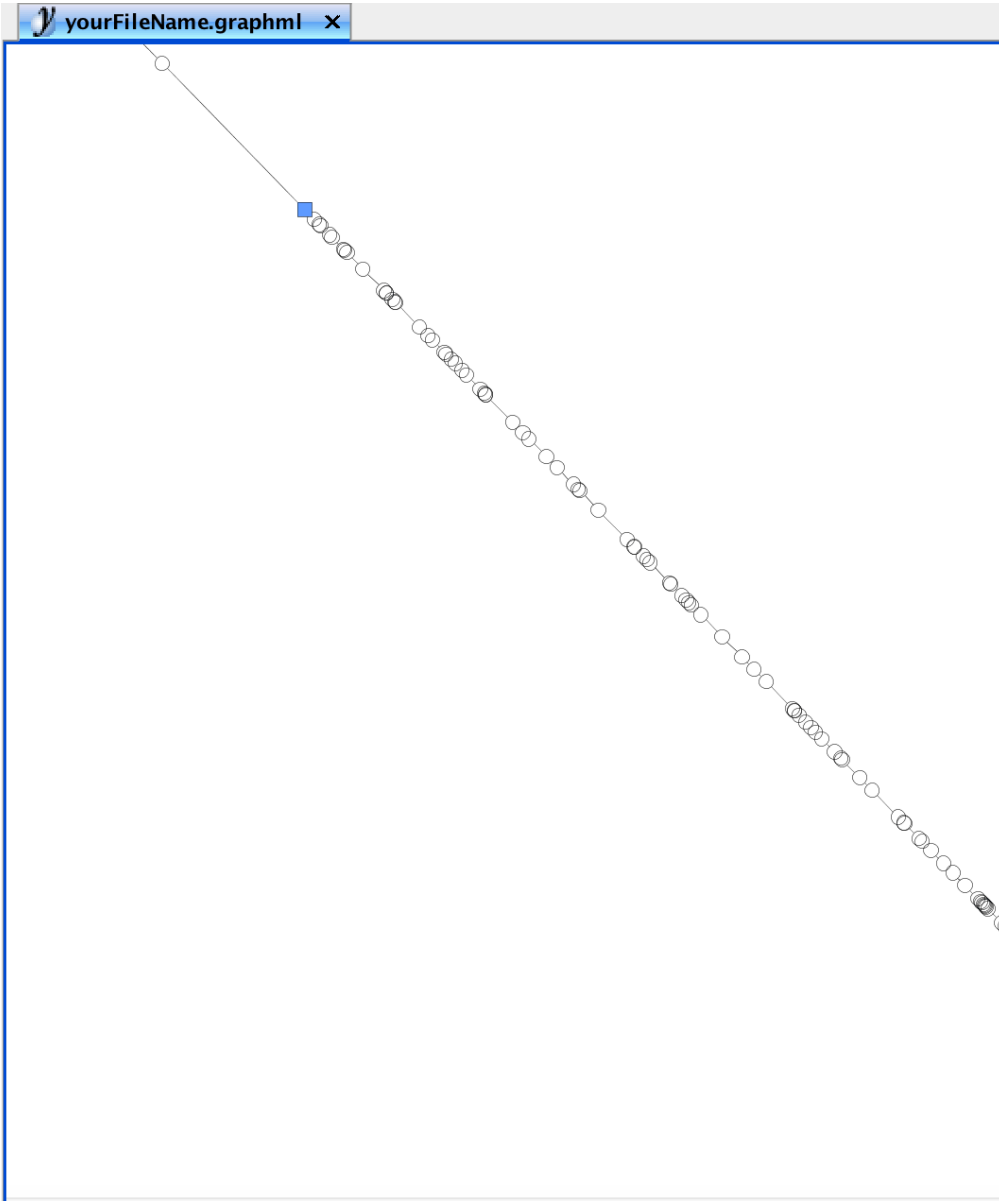


```
showTrajInfo(exBundle)
```

```
## [[ CellTrails ]]
## logcounts: 183 features, 1008 samples
## Pheno data:
##   sampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   phenoNames: "fm143" "origin" ... "landmark" (7)
## Feature data:
##   featureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   latentSpace: 1008 samples, 9 dimensions
##   states: "S1" "S2" ... "S11" (11)
## Trajectories: [Component(#Vertices,Edges)]: 1(10,9)
##   trajSampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (896)
##   trajResiduals: MSE=9.9e-03
##   landmarks:  #Branches=7 #Terminals=9 #User=1
##   trajLayout: available
## Trail data:
##   trailNames: "TrS" "TrES" "TrES*" (3)
```

Please note that the trajectory graph can also be exported having all landmarks highlighted. This is particulary helpful if user-defined landmarks need to be changed.

```
# Export Trajectory Graph Layout with sample names
write.ygraphml(exBundle, file='yourFileName.graphml',
               color_by="phenoName", name="landmark",
               node_label="landmark")
```


yourFileName.graphml ×

Here, blue nodes denote user-defined landmarks, green nodes are branching points and red nodes are leafs. Landmark IDs, as listed by `landmarks`, are indicated as node names, respectively.

8.2.2 Using R

A visual and empiric identification of user-defined landmarks can be helpful, but scientifically more appropriate is a statistical approach. For this purpose we analyze the distribution of all lagged differences along trail TrES. Here, we make use of the pseudotime information of each trail, respectively.

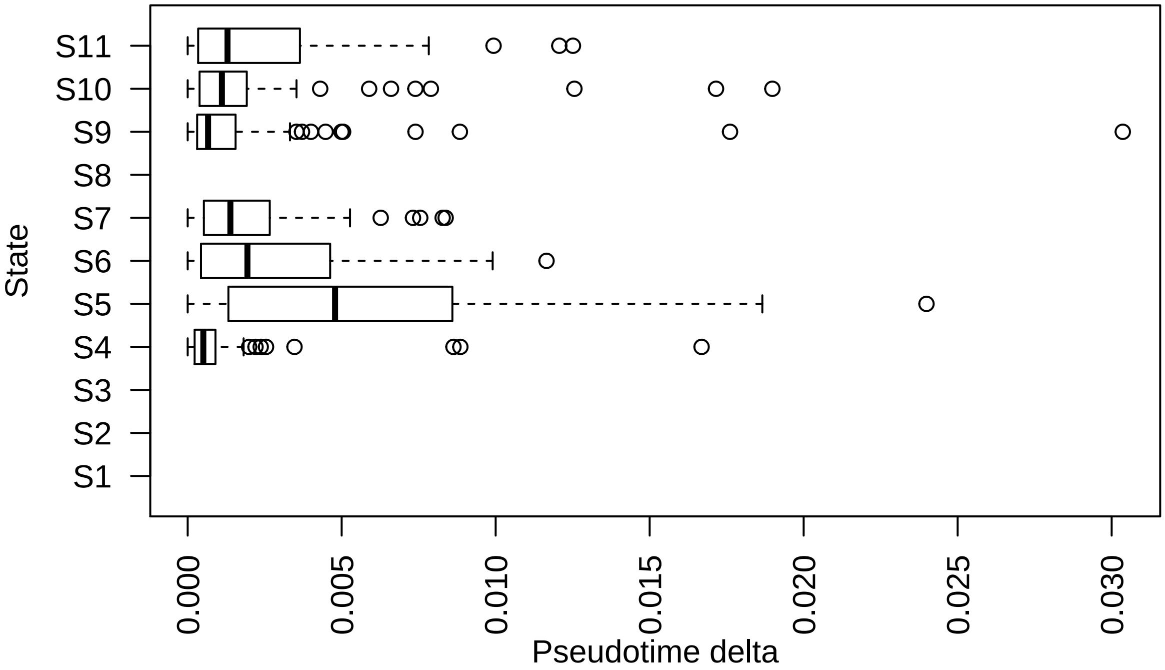
```
# Extract pseudotime of TrES
ptime <- trails(exBundle)[, "TrES"]

# Subset SingleCellExperiment set
# to samples which are part of trail TrES
trES <- exBundle[, !is.na(ptime)]

# Order samples by pseudotime
o <- order(trails(trES)[, "TrES"])
trES <- trES[, o]
ptime <- trails(trES)[, "TrES"]
names(ptime) <- colnames(trES)

# Lagged pseudotime values per state
ptime_states <- split(ptime, states(trES))
lptime <- lapply(ptime_states,
                 function(x){y <- diff(sort(x)); y[-length(y)]})

bp <- boxplot(lptime, horizontal=TRUE,
              ylab="State", xlab="Pseudotime delta", las=2)
```



The boxplot statistics indicate that there is a strong outlier in state S9 (which is termed state i in the original *CellTrails* article). Let's extract the sample right before the leap.

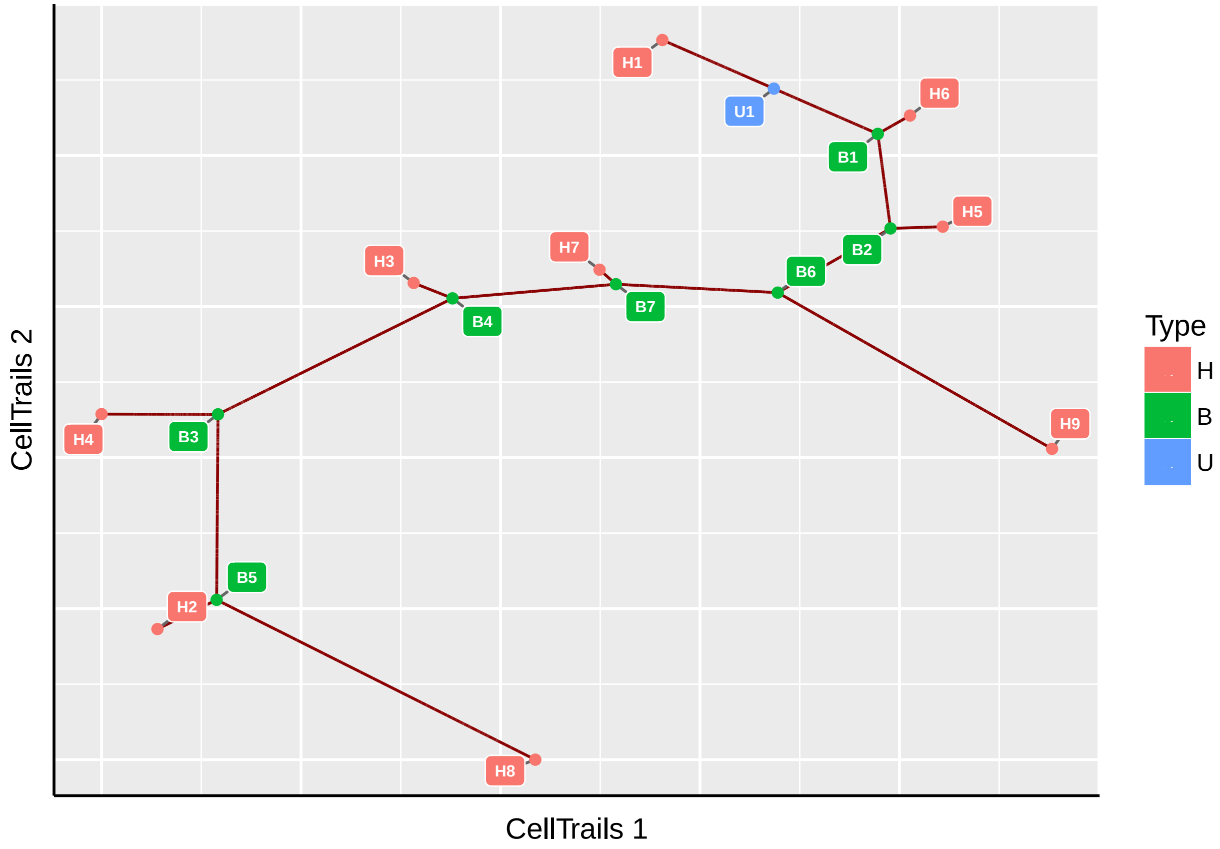
```
leap <- lptime$S9[which.max(lptime$S9) - 1]
names(leap)
```

```
## [1] "Cell-8-57"
```

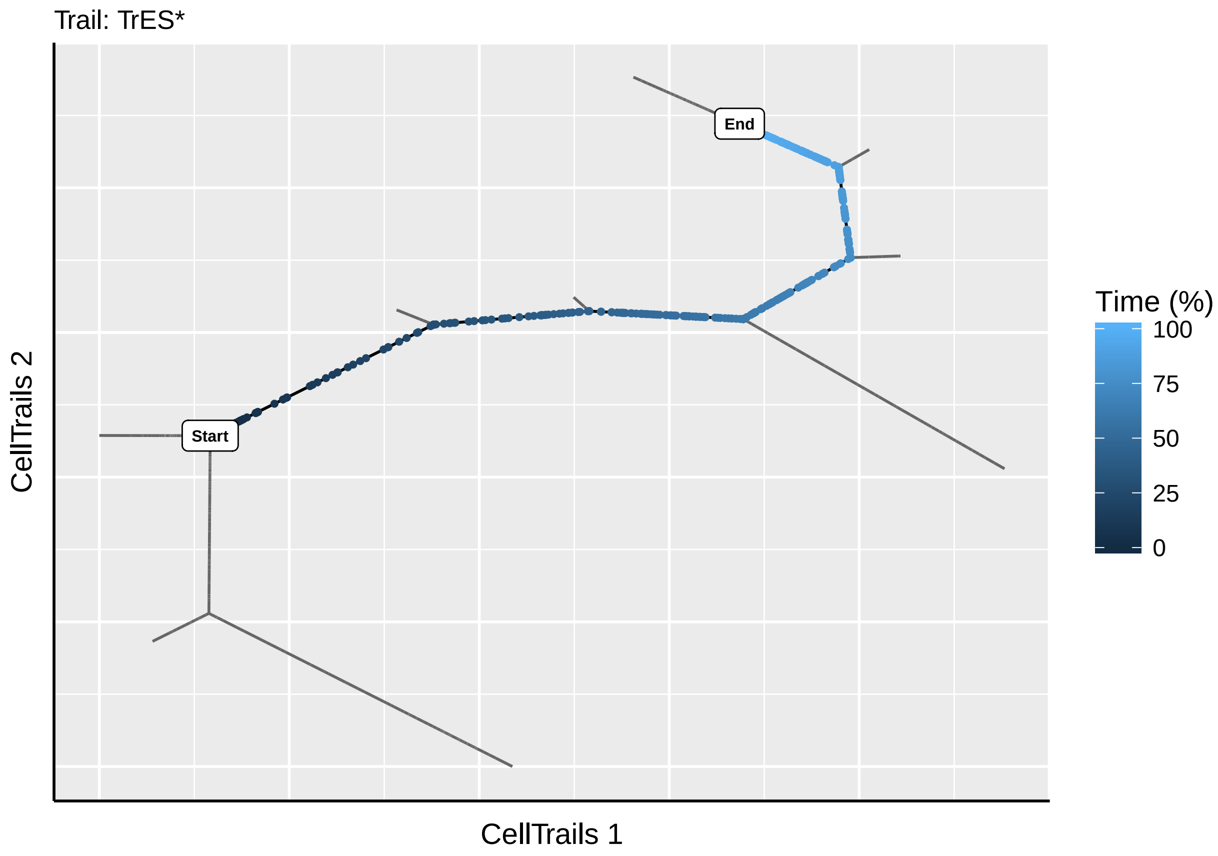
The function `userLandmarks<-` enables us to (re-)define the set of user landmarks.

```
userLandmarks(exBundle) <- names(leap)

# Trail Identification
plotMap(exBundle, color_by="phenoName", name="landmark")
```



```
exBundle <- addTrail(exBundle, from="B3", to="U1", name="TrES*")
plotTrail(exBundle, name="TrES*")
```



```
showTrajInfo(exBundle)
```

```
## [[ CellTrails ]]
## logcounts: 183 features, 1008 samples
## Pheno data:
##   sampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (1008)
##   phenoNames: "fm143" "origin" ... "landmark" (7)
## Feature data:
##   featureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   rowData: none
## Trajectory data:
##   trajFeatureNames: "ABCA5" "ARF1" ... "USH2A" (183)
##   latentSpace: 1008 samples, 9 dimensions
##   states: "S1" "S2" ... "S11" (11)
## Trajectories: [Component(#Vertices,Edges)]: 1(10,9)
##   trajSampleNames: "Cell-1-1" "Cell-1-2" ... "Cell-11-82" (896)
##   trajResiduals: MSE=9.9e-03
##   landmarks:  #Branches=7 #Terminals=9 #User=1
##   trajLayout: available
## Trail data:
##   trailNames: "TrS" "TrES" "TrES*" (3)
```

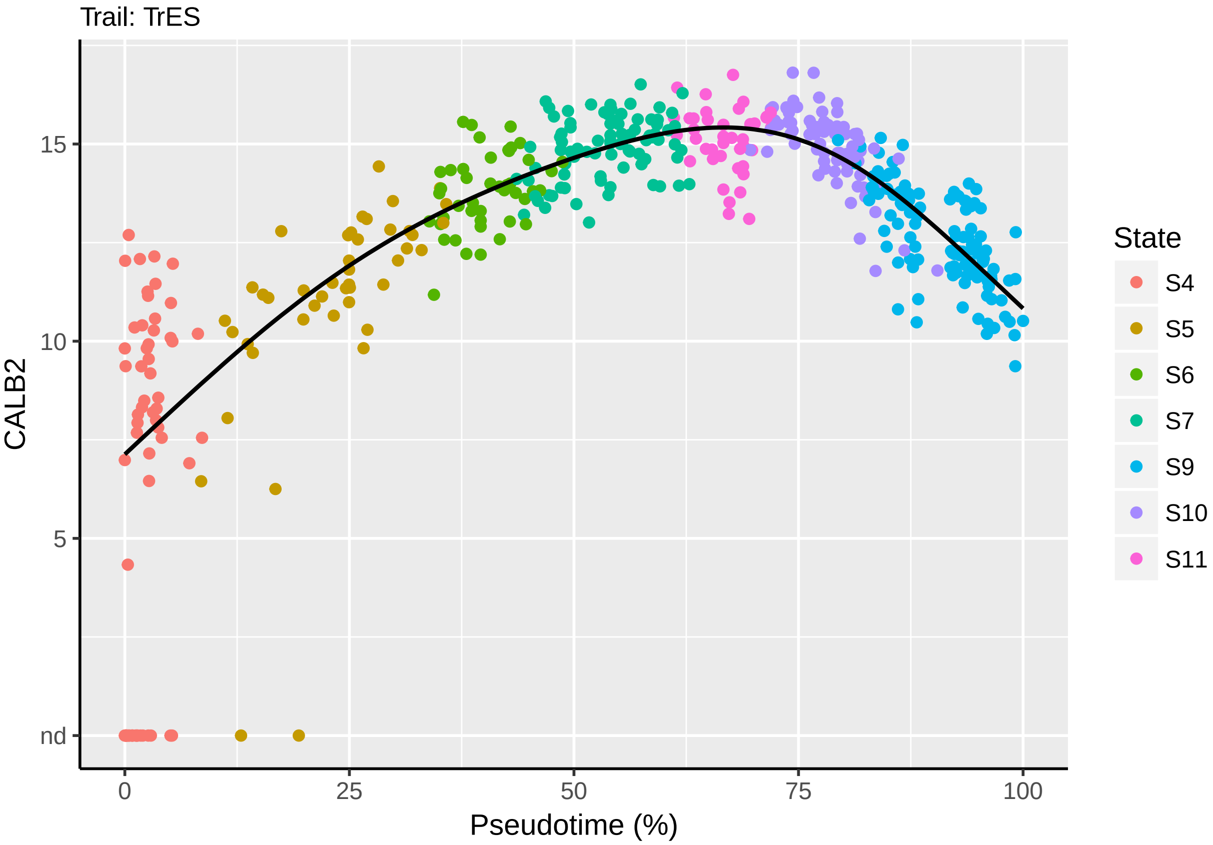
Please note that all user-defined landmarks can be removed using `userLandmarks(exBundle) <- NULL`.

8.3 Inference of Dynamics

CellTrails defines pseudotime as the geodesic distance of each node of the trail from the start node. To learn the expression level of a feature as a function of pseudotime, *CellTrails* used generalized additive models (GAM) with a single smoothing term with five basis dimensions. Here, for each feature, *CellTrails* introduces prior weights for each observation to lower the confounding effect of missing data to the maximum-likelihood-based fitting process.

Feature expression as a function of pseudotime along an individual trail can be plotted with the `plotDynamic` function. This results in the fitted dynamic function (= black line) and the individual expression per sample (= points represent samples colored by their state membership). For example, the expression of the calcium buffer *CALB2* during extrastriolar hair cell development can be displayed as follows:

```
plotDynamic(exBundle, feature_name="CALB2", trail_name="TrES")
```



Please note that the fitting function automatically scales pseudotime between 0 and 100% for each trail.

The fit information can be extracted via function `fitDynamic` and used for further downstream analyses:

```
fit <- fitDynamic(exBundle, trail_name="TrES", feature_name="CALB2")

summary(fit)
```

```
##            Length Class  Mode
## pseudotime 470    -none- numeric
## expression 470    -none- numeric
## gam         52    gam    list
```

```
range(fit$pseudotime)
```

```
## [1] 0 1
```

```
range(fit$expression)
```

```
## [1]  7.131096 15.419633
```

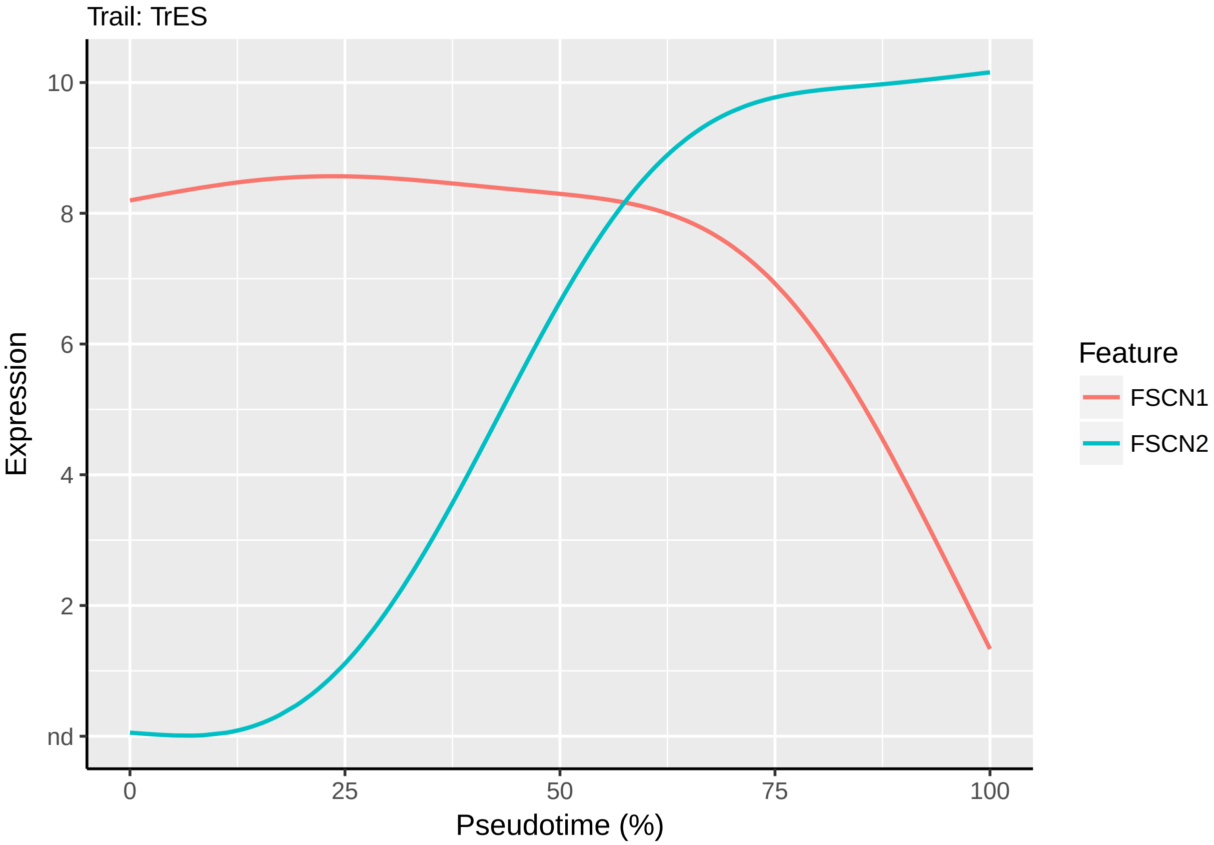
```
# Predict expression at 0%, 25%, 50%, 75% and 100% of pseudotime
timepoints <- data.frame(x=c("0%"=0, "25%"=.25, "50%"=.5, "75%"=.75, "100%"=1))
predict(fit$gam, newdata=timepoints)
```

```
##        0%       25%       50%       75%      100%
##  7.131096 11.917652 14.649184 15.117750 10.831007
```

8.4 Dynamic Comparison: Within Trails

CellTrails allows the analysis and comparison of the expression of multiple features along a single trail. For example, the expression dynamics of the acting crosslinkers *FSCN1* and *FSCN2* can be displayed in a single plot as follows:

```
plotDynamic(exBundle, feature_name=c("FSCN1", "FSCN2"), trail_name="TrES")
```



By using the fitting function `fitDynamic`, the similarity/correspondence between curves can be quantified. This allows a quantitative assessment of the observed anticorrelation seen in the plot above between *FSCN1* and *FSCN2*:

```
fscn1_fit <- fitDynamic(exBundle, trail_name="TrES", feature_name="FSCN1")
fscn2_fit <- fitDynamic(exBundle, trail_name="TrES", feature_name="FSCN2")

# Correlation
cor(fscn1_fit$expression, fscn2_fit$expression)
```

```
## [1] -0.6432156
```

8.5 Dynamic Comparison: Between Trails

Genes have non-uniform expression rates and each trail has a distinct set of upregulated features, but also contains unequal numbers of samples. Since pseudotime is computed based on expression differences between individual samples, the pseudotime axis may be distorted, leading to stretched or compressed sections of longitudinal expression data that make comparisons of such trails challenging. To align different trails, despite these differences, *CellTrails* employs a strategy that has long been known in speech recognition, called dynamic time warping (Sakoe and Chiba, 1978). Feature expression dynamics are modeled analogous to how dynamic time warping is used to align phonetic dynamics present in speech. Innate non-linear variation in the length of individual phonemes (i.e., states) is appropriately modeled, which results in stretching and shrinking of word (i.e., trail) segments. This allows the computation of inter-trail alignment warps of individual expression time series that are similar but locally out of phase. The overall dissimilarity between two expression time series can be estimated by the root-mean-square deviation (*RMSD*), the total deviation (*TD*), the area between curves (*ABC*), or Pearson's correlation coefficient (*COR*) over all aligned elements. The warp and the corresponding quantitative score can be computed using the function `contrastTrailExpr`.

```
# Compare ATOH1 dynamic
# Root-mean-square deviation
contrastTrailExpr(exBundle, feature_names=c("ATOH1"),
                  trail_names=c("TrS", "TrES"), score="RMSD")
```

```
##     ATOH1
## 0.0321192
```

```
# Total deviation
contrastTrailExpr(exBundle, feature_names=c("ATOH1"),
                  trail_names=c("TrS", "TrES"), score="TD")
```

```
##    ATOH1
## 6.364661
```

```
# Area between curves
contrastTrailExpr(exBundle, feature_names=c("ATOH1"),
                  trail_names=c("TrS", "TrES"), score="ABC")
```

```
##      ATOH1
## 0.02300188
```

```
# Pearson's correlation coefficient
contrastTrailExpr(exBundle, feature_names=c("ATOH1"),
                  trail_names=c("TrS", "TrES"), score="COR")
```

```
##     ATOH1
## 0.9999033
```

In this example, *ATOH1* is expected to have a highly similar dynamic between both trails. Therefore, *RMSD*, *TD*, *ABC* values should be low and *COR* values high relative to other assayed features.

To identify features that differ between two trails, we can compute the divergence for all features and analyze the Z-score distribution as derived by `scale`:

```
# Compare TrS and TrES dynamics
# Root-mean-square deviation
all_rmsd <- contrastTrailExpr(exBundle,
                              trail_names=c("TrS", "TrES"), score="RMSD")

# Identify highly differing features
all_rmsd <- all_rmsd[all_rmsd > 0]
zscores <- scale(log(all_rmsd))
sort(all_rmsd[zscores > 1.65])
```

```
##     SYN3    AKAP5      OCM   SLC8A1   ATP2B2    MYO1H   CAB39L   MCOLN3
## 1.494187 1.529849 1.601328 1.903307 1.953127 1.981343 2.304601 2.417525
##  CHRNA10    MYO3A     CIB2    TNNC2    SKOR2   LOXHD1     TMC2
## 2.637487 2.671320 2.790146 3.051273 3.717097 4.129766 4.799793
```

8.6 Parallelization

In the case one wants to compare a large number of features (e.g. from an RNA-Seq experiment), the computation can be significantly sped up by parallel computing. In this example, we use the package *doSNOW*, but any other package may also be used for this purpose.

```
library(doSNOW)
# Register parallel backend
cpu.cl <- makeCluster(parallel::detectCores() * 2)
registerDoSNOW(cpu.cl)

# Compute warps
fnames <- featureNames(exBundle)
all_rmsd <- foreach(i=seq_along(fnames), .combine=rbind)  %dopar%  {
  g <- fnames[i]
  CellTrails::contrastTrailExpr(exBundle,
                                feature_name=g,
                                trail_names=c("TrES", "TrS"),
                                score="RMSD")
}
stopCluster(cpu.cl)
all_rmsd <- all_rmsd[, 1]
names(all_rmsd) <- fnames
```

```
# Identify highly differing features
all_rmsd <- all_rmsd[all_rmsd > 0]
zscores <- scale(log(all_rmsd))
sort(all_rmsd[zscores > 1.65])
```

```
##     SYN3    AKAP5      OCM   SLC8A1   ATP2B2    MYO1H   CAB39L   MCOLN3
## 1.494187 1.529849 1.601328 1.903307 1.953127 1.981343 2.304601 2.417525
##  CHRNA10    MYO3A     CIB2    TNNC2    SKOR2   LOXHD1     TMC2
## 2.637487 2.671320 2.790146 3.051273 3.717097 4.129766 4.799793
```

Please note that the advantage in computation time increases with the number of features; for a small number of features parallel computing may be slower than the sequential approach due to its overhead.

Chapter 9

Appendix

9.1 Protocols

The following protocols describe how this package can be used to perform the data analysis shown in the original *CellTrails* article.

9.1.1 Chicken E15 Utricle Data

```
# Load expression data
bundle <- readRDS(system.file("exdata", "bundle.rds", package="CellTrails"))

# Manifold Learning
se <- embedSamples(bundle)
d <- findSpectrum(se$eigenvalues, frac=100) #Similar to Figure 1E
latentSpace(bundle) <- se$components[, d]

# Clustering
states(bundle) <- findStates(bundle, min_size=0.01,
                             min_feat=5, max_pval=1e-04, min_fc=2)

# Sample Ordering
bundle <- connectStates(bundle, l=10)
showTrajInfo(bundle)

bundle <- selectTrajectory(bundle, component=1)
bundle <- fitTrajectory(bundle)

# CellTrails maps
# Please note: For illustration purposes, the layout was
# computed in yEd using functions write.ygraphml and
# read.ygraphml, and is part of the CellTrails package
tl <- read.ygraphml(system.file("exdata", "bundle.graphml",
                                package="CellTrails"))
trajLayout(bundle, adjust=TRUE) <- tl

# Define subtrail by adding a user-defined landmark
```

```r
userLandmarks(bundle) <- "Cell-8-57"

# Analysis of Expression Dynamics
bundle <- addTrail(bundle, from="B3", to="H9", name="TrS")
bundle <- addTrail(bundle, from="B3", to="H1", name="TrES")
bundle <- addTrail(bundle, from="B3", to="U1", name="TrES*")

# Inter-trail comparison (similar to Figure 5B)
rmsd_all <- contrastTrailExpr(bundle, trail_names=c("TrS", "TrES"))
rmsd_all <- rmsd_all[rmsd_all > 0]
sort(rmsd_all[scale(log(rmsd_all)) > 1.65])

# ------------------------------
# Visualizations
# ------------------------------
# Plot size of clusters (similar to Figure 2E)
plotStateSize(bundle)

# Plot expression distribution
plotStateExpression(bundle, feature_name="OTOA")
plotStateExpression(bundle, feature_name="ATOH1")
plotStateExpression(bundle, feature_name="CALB2")
plotStateExpression(bundle, feature_name="ATP2B2")

# Plot manifold (similar to Figure S4F)
set.seed(1101)
gp <- plotManifold(bundle, color_by="phenoName", name="state")
manifold2D(bundle) <- gp
plotManifold(bundle, color_by="phenoName", name="fm143")
plotManifold(bundle, color_by="featureName", name="OTOA")
plotManifold(bundle, color_by="featureName", name="CALB2")

# Plot state trajectory graph (similar to Figure 1G)
plotStateTrajectory(bundle, color_by="phenoName",
                    name="fm143", point_size=1.5,
                    label_offset=4, component=1)
plotStateTrajectory(bundle, color_by="phenoName",
                    name="origin", point_size=1.5,
                    label_offset=4, component=1)
plotStateTrajectory(bundle, color_by="featureName",
                    name="OTOA", point_size=5,
                    label_offset=4, component=1)
plotStateTrajectory(bundle, color_by="featureName",
                    name="ATOH1", point_size=5,
                    label_offset=4, component=1)
plotStateTrajectory(bundle, color_by="featureName",
                    name="CALB2", point_size=5,
                    label_offset=4, component=1)

# Plot trajectory fit (similar to Figure 2E)
plotTrajectoryFit(bundle)

# Plot CellTrails maps (similar to Figure 3 and Table S2)
```

```
plotMap(bundle, color_by="phenoName", name="fm143")
plotMap(bundle, color_by="featureName",
        name="ATOH1", type="raw")
plotMap(bundle, color_by="featureName",
        name="ATOH1", type="surface.fit")
plotMap(bundle, color_by="featureName",
        name="ATOH1", type="surface.se")
plotMap(bundle, color_by="featureName",
        name="ATOH1", type="surface.fit", samples_only=TRUE)

# Plot landmarks
plotMap(bundle, color_by="phenoName", name="landmark")

# Plot trails (similar to Figure 4K)
plotTrail(bundle, name="TrS")
plotTrail(bundle, name="TrES")
plotTrail(bundle, name="TrES*")

# Plot single dynamics (similar to Figure 4B,G and Table S2)
plotDynamic(bundle, feature_name="CALB2", trail_name="TrES")
plotDynamic(bundle, feature_name="ATP2B2", trail_name="TrES")

# Compare dynamics (similar to Figure 6A)
plotDynamic(bundle,
            feature_name=c("TECTA", "OTOA", "ATOH1", "POU4F3",
                           "MYO7A", "CALB2", "SYN3", "SKOR2",
                           "ATP2B2", "LOXHD1", "MYO3A", "TMC2",
                           "TNNC2"), trail_name="TrES")

plotDynamic(bundle,
            feature_name=c("TECTA", "OTOA", "ATOH1", "POU4F3",
                           "MYO7A", "CALB2", "SYN3", "SKOR2",
                           "ATP2B2", "LOXHD1", "MYO3A", "TMC2",
                           "TNNC2"), trail_name="TrS")
```

9.1.2 Mouse P1 Utricle Data

In the following, we provide a protocol to analyze the publicly-available dataset containing single-cell RNASeq measurements of 14,313 genes in 120 cells from P1 mouse utricles (*GEO* accession code: GSE71982). Experimental metadata was generated during cell sorting (GFP and tdTomato fluorescence indicating major cell types). The processed data (`mmu_p1_utricle.rda`) can be downloaded as *SingleCellExperiment* object from here. The trajectory layout (`mmu_p1_utricle.graphml`) can be downloaded from here.

If you use this dataset for your research, please cite the original work by Burns et al. (2015).

```
# Load expression data
p1utricle <- readRDS("mmu_p1_utricle.rds")

# Feature Selection
trajFeatureNames(p1utricle) <- filterTrajFeaturesByDL(p1utricle, threshold=2)
trajFeatureNames(p1utricle) <- filterTrajFeaturesByCOV(p1utricle, threshold=.5)
trajFeatureNames(p1utricle) <- filterTrajFeaturesByFF(p1utricle)
```

```
showTrajInfo(p1utricle)

# Manifold Learning
se <- embedSamples(p1utricle)
d <- findSpectrum(se$eigenvalues)
latentSpace(p1utricle) <- se$components[, d]

# Clustering (parameters account for low sample size)
states(p1utricle) <- findStates(p1utricle, max_pval=1e-3, min_feat=2)

# Sample Ordering
p1utricle <- connectStates(p1utricle)
p1utricle <- fitTrajectory(p1utricle)
showTrajInfo(p1utricle)

# CellTrails maps
tl <- read.ygraphml("mmu_p1_utricle.graphml")
trajLayout(p1utricle) <- tl

# Analysis of Expression Dynamics
p1utricle <- addTrail(p1utricle, from="H1", to="H3", name="Tr1")
p1utricle <- addTrail(p1utricle, from="H1", to="H2", name="Tr2")

# Inter-trail comparison
rmsd_all <- contrastTrailExpr(p1utricle, trail_names=c("Tr1", "Tr2"))
rmsd_all <- rmsd_all[rmsd_all > 0]
sort(rmsd_all[scale(log(rmsd_all)) > 1.65])

# Alternative: using parallel computing using doSNOW
library(doSNOW)
cpu.cl <- makeCluster(parallel::detectCores() * 2)
registerDoSNOW(cpu.cl)

fnames <- featureNames(p1utricle)
all_rmsd <- foreach(i=seq_along(fnames), .combine=rbind)  %dopar%  {
  g <- fnames[i]
  CellTrails::contrastTrailExpr(p1utricle, feature_name=g,
                                trail_names=c("Tr1", "Tr2"), score="RMSD")
}
stopCluster(cpu.cl)
all_rmsd <- all_rmsd[, 1]
names(all_rmsd) <- fnames
all_rmsd <- all_rmsd[all_rmsd > 0]
zscores <- scale(log(all_rmsd))
sort(all_rmsd[zscores > 1.65])

# -------------------------------
# Visualizations
# -------------------------------
# Plot size of clusters (similar to Figure 7A)
plotStateSize(p1utricle)

# Plot expression distribution
```

```r
plotStateExpression(p1utricle, feature_name="Otoa")
plotStateExpression(p1utricle, feature_name="Atoh1")
plotStateExpression(p1utricle, feature_name="Sox2")

# Plot manifold
set.seed(1101)
gp <- plotManifold(p1utricle, color_by="phenoName", name="state")
manifold2D(p1utricle) <- gp
plotManifold(p1utricle, color_by="phenoName", name="gate")
plotManifold(p1utricle, color_by="featureName", name="Otoa")
plotManifold(p1utricle, color_by="featureName", name="Fscn2")

# Plot state trajectory graph (similar to Figure 1G)
plotStateTrajectory(p1utricle, color_by="phenoName",
                    name="gate", point_size=1.5,
                    label_offset=4, component=1)
plotStateTrajectory(p1utricle, color_by="featureName",
                    name="Otoa", point_size=5,
                    label_offset=4, component=1)

# Plot trajectory fit (similar to Figure 7A)
plotTrajectoryFit(p1utricle)

# Plot CellTrails maps (similar to Figure 7C-E)
plotMap(p1utricle, color_by="phenoName", name="gate")
plotMap(p1utricle, color_by="featureName",
        name="Atoh1", type="raw")
plotMap(p1utricle, color_by="featureName",
        name="Atoh1", type="surface.fit")
plotMap(p1utricle, color_by="featureName",
        name="Atoh1", type="surface.se")
plotMap(p1utricle, color_by="featureName",
        name="Atoh1", type="surface.fit",
        samples_only=TRUE)

# Plot landmarks
plotMap(p1utricle, color_by="phenoName", name="landmark")

# Plot trails (similar to Figure 7F)
plotTrail(p1utricle, name="Tr1")
plotTrail(p1utricle, name="Tr2")

# Plot single dynamics (similar to Figure 7I)
plotDynamic(p1utricle, feature_name="Fgf21", trail_name="Tr2")
plotDynamic(p1utricle, feature_name="Fgf21", trail_name="Tr1")

# Compare dynamics
plotDynamic(p1utricle, feature_name=c("Fscn1", "Fscn2"), trail_name="Tr2")
```

9.2 Runtime

In this section, we illustrate that a *CellTrails* analysis can be performed in a reasonable period of time. The elapsed computational runtime of each function was measured on a MacBook Pro (Early 2015) with a 3.1 GHz Intel Core i7 processor, 16 GB 1867 MHz DDR3 RAM, and an Intel Iris Graphics 6100 1536 MB graphics card.

9.2.1 Protocol: Chicken E15 Utricle Data

This dataset consists of 183 features and 1,008 samples. The computation time of the whole protocol as listed above took less then two minutes. Let's assume that computing the layout takes about two minutes (starting *yEd*, running the layouter, saving the file), then the total runtime is up to five minutes.

9.2.2 Protocol: Mouse P1 Utricle Data

This dataset consists of 14,313 features and 120 samples. The computation time of the whole protocol as listed above (with parallelization of the inter-trail dynamics comparison) took less then seven minutes. Let's assume that computing the layout takes about two minutes (starting *yEd*, running the layouter, saving the file), then the total runtime is up to 10 minutes.

9.3 Session Info

We used *yEd* version 3.14.4.
The *R* session and the system used to compile this document is listed below.

```r
sessionInfo()
```

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS Sierra 10.12.4
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel  stats4    stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] CellTrails_0.99.12         SingleCellExperiment_1.0.0
##  [3] SummarizedExperiment_1.8.0 DelayedArray_0.4.1
##  [5] matrixStats_0.52.2         Biobase_2.38.0
##  [7] GenomicRanges_1.30.0       GenomeInfoDb_1.14.0
##  [9] IRanges_2.12.0             S4Vectors_0.16.0
## [11] BiocGenerics_0.24.0        BiocStyle_2.6.1
##
## loaded via a namespace (and not attached):
```

```
##   [1] backports_1.1.2         Hmisc_4.0-3
##   [3] RcppEigen_0.3.3.3.1     plyr_1.8.4
##   [5] igraph_1.1.2            lazyeval_0.2.1
##   [7] sp_1.2-5                shinydashboard_0.6.1
##   [9] splines_3.4.3           BiocParallel_1.12.0
##  [11] ggplot2_2.2.1           scater_1.6.1
##  [13] digest_0.6.12           htmltools_0.3.6
##  [15] viridis_0.4.0           magrittr_1.5
##  [17] checkmate_1.8.5         memoise_1.1.0
##  [19] cluster_2.0.6           limma_3.34.3
##  [21] xts_0.10-0              prettyunits_1.0.2
##  [23] colorspace_1.3-2        blob_1.1.0
##  [25] ggrepel_0.7.0           xfun_0.1
##  [27] dplyr_0.7.4             RCurl_1.95-4.8
##  [29] tximport_1.6.0          EnvStats_2.3.0
##  [31] lme4_1.1-14             bindr_0.1
##  [33] survival_2.41-3         zoo_1.8-0
##  [35] glue_1.2.0              gtable_0.2.0
##  [37] zlibbioc_1.24.0         XVector_0.18.0
##  [39] MatrixModels_0.4-1      cba_0.2-19
##  [41] car_2.1-6               kernlab_0.9-25
##  [43] prabclus_2.2-6          DEoptimR_1.0-8
##  [45] SparseM_1.77            VIM_4.7.0
##  [47] scales_0.5.0            mvtnorm_1.0-6
##  [49] DBI_0.7                 edgeR_3.20.1
##  [51] Rcpp_0.12.14            dtw_1.18-1
##  [53] laeken_0.4.6            viridisLite_0.2.0
##  [55] xtable_1.8-2            progress_1.1.2
##  [57] htmlTable_1.11.0        foreign_0.8-69
##  [59] bit_1.1-12              proxy_0.4-20
##  [61] mclust_5.4              Formula_1.2-2
##  [63] DT_0.2                  vcd_1.4-4
##  [65] htmlwidgets_0.9         FNN_1.1
##  [67] RColorBrewer_1.1-2      fpc_2.1-10
##  [69] acepack_1.4.1           modeltools_0.2-21
##  [71] pkgconfig_2.0.1         XML_3.98-1.9
##  [73] flexmix_2.3-14          nnet_7.3-12
##  [75] locfit_1.5-9.1          dynamicTreeCut_1.63-1
##  [77] labeling_0.3            rlang_0.1.4
##  [79] reshape2_1.4.3          AnnotationDbi_1.40.0
##  [81] munsell_0.4.3           tools_3.4.3
##  [83] RSQLite_2.0             evaluate_0.10.1
##  [85] stringr_1.2.0           maptree_1.4-7
##  [87] yaml_2.1.16             knitr_1.20
##  [89] bit64_0.9-7             robustbase_0.92-8
##  [91] purrr_0.2.4             dendextend_1.6.0
##  [93] bindrcpp_0.2            nlme_3.1-131
##  [95] quantreg_5.34           whisker_0.3-2
##  [97] mime_0.5                scran_1.6.6
##  [99] biomaRt_2.34.0          pbkrtest_0.4-7
## [101] compiler_3.4.3          rstudioapi_0.7
## [103] png_0.1-7               curl_3.1
## [105] beeswarm_0.2.3          e1071_1.6-8
## [107] smoother_1.1            tibble_1.3.4
## [109] statmod_1.4.30         stringi_1.1.6
## [111] lattice_0.20-35        trimcluster_0.1-2
## [113] Matrix_1.2-12          nloptr_1.0.4
## [115] lmtest_0.9-35          data.table_1.10.4-3
## [117] bitops_1.0-6           httpuv_1.3.5
## [119] R6_2.2.2               latticeExtra_0.6-28
## [121] bookdown_0.7           gridExtra_2.3
## [123] vipor_0.4.5            boot_1.3-20
## [125] MASS_7.3-47            assertthat_0.2.0
## [127] destiny_2.6.1          rhdf5_2.22.0
## [129] rprojroot_1.2          rjson_0.2.15
## [131] GenomeInfoDbData_0.99.1 diptest_0.75-7
## [133] mgcv_1.8-22            grid_3.4.3
## [135] rpart_4.1-11           minqa_1.2.4
## [137] tidyr_0.7.2            class_7.3-14
## [139] rmarkdown_1.8          Rtsne_0.13
## [141] TTR_0.23-2             shiny_1.0.5
## [143] base64enc_0.1-3        ggbeeswarm_0.6.0
```

Bibliography

and, J. W. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244.

Angerer, P., Haghverdi, L., Buettner, M., Theis, F., Marr, C., and Buettner, F. (2015). destiny: diffusion maps for large-scale single-cell data in r. *Bioinformatics*, 32:1241–1243.

Bedall, F. and Zimmermann, H. (1979). Algorithm as143. the mediancentre. *Appl Statist*, 28:325–328.

Belkin, M. and Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15:1373–1396.

Benjamini, Y. and Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society*, Series B 57:289–300.

Buettner, F., Natarajan, K., Casale, F., Proserpio, V., Scialdone, A., Theis, F., Teichmann, S., Marioni, J., and Stegle, O. (2015). Computational analysis of cell-to-cell heterogeneity in single-cell rna-sequencing data reveals hidden subpopulations of cells. *Nature Biotechnology*, 33(2):155–160.

Burns, J., Kelly, M., Hoa, M., Morell, R., and Kelley, M. (2015). Single-cell rna-seq resolves cellular complexity in sensory organs from the neonatal inner ear. *Nature Communications*, 15(6):8557. doi: 10.1038/ncomms9557.

Carlson, M. (2017). *org.Mm.eg.db: genome wide annotation for mouse.* R package version 3.5.0.

Ellwanger, D., Scheibinger, M., Dumont, R., Barr-Gillespie, P., and Heller, S. (2018). Transcriptional dynamics of hair-bundle morphogenesis revealed with celltrails. *Cell Reports*, 23(10):2901–2914. doi: 10.1016/j.celrep.2018.05.002e.

Fruchterman, T. and Reingold, E. (1991). Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164.

Kruskal, J. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc Amer Math Soc*, 7:48–50.

Lun, A., McCarthy, D., and Marioni, J. (2016). A step-by-step workflow for low-level analysis of single-cell rna-seq data with bioconductor. *F1000Res.*, 5:2122.

Lun, A. and Risso, D. (2017). *SingleCellExperiment: S4 Classes for Single Cell Data.* R package version 1.0.0.

Mahata, B., Zhang, X., Kolodziejczyk, A., Proserpio, V., Haim-Vilmovsky, L., Taylor, A., Hebenstreit, D., Dingler, F., Moignard, V., Goettgens, B., Arlt, W., McKenzie, A., and Teichmann, S. (2014). Single-cell rna sequencing reveals t helper cells synthesizing steroids de novo to contribute to immune homeostasis. *Cell Reports*, 7(4):1130–42.

McCarthy, D., Campbell, K., Lun, A., and Wills, Q. (2017). Scater: pre-processing, quality control, normalisation and visualisation of single-cell rna-seq data in r. *Bioinformatics*, 14:1179–1186.

Pagès, H., Carlson, M., Falcon, S., and Li, N. (2017). *AnnotationDbi: Annotation Database Interface*. R package version 1.40.0.

Peto, R. and Peto, J. (1972). Asymptotically efficient bank invariant test procedures (with discussion). *Journal of the Royal Statistical Society*, Series A 135:185–206.

Sakoe, H. and Chiba, S. (1978). Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signaling Processing*, 26:43–49.

Sussman, D., Tang, M., Fishkind, D., and Priebe, C. (2012). A consistent adjacency spectral embedding for stochastic blockmodel graphs. *J Am Stat Assoc*, 107:1119–1128.

van der Maaten, L. and Hinton, G. (2008). Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.