Trajectory analysis using Monocle

Ivan Berest and Christian Arnold

Contents

Goals

This vignette will introduce you to Monocle 3, an R package that provides a toolkit for the analysis of single-cell expression and accessibility data. We will use it for constructing single-cell accessibility trajectories and pseudotime analyses, and combine it with the previously calculated TF activity scores to identify TFs that may change their activity in a trajectory-dependent way! An exciting set of analyses and integrations is ahead of us.

References

This vignette is a modified version of this vignette and this vignette as well as this website.

Load libraries

As before, we first load all necessary R packages that are needed for this vignette.

```
suppressPackageStartupMessages({
library(Signac)
library(Seurat)
library(SeuratWrappers)
library(monocle3)
library(tidyverse)
library(patchwork)
library(scales)
library(ComplexHeatmap)
library(circlize)
library(RColorBrewer)
})
set.seed(1990)
```

Load data

We begin by loading our pre-processed `Seurat` object. In most vignettes, we took the object from the introductory QC vignette into R; however here, we will use the final object from the TF analysis vignette because we will need the `chromAVR` derived TF activity scores here as well!

```
# Specify the path to the "outs" folder from cellranger-atac here
# Make sure to have a trailing slash here
outFolder="/mnt/data/cellranger/outs/"
outFolder="/g/scb2/zaugg/zaugg_shared/Courses_and_Teaching/ATAC-Seq_Course/2022/data/testRun_March/coun

# This time, we load a different object at the beginning
seu.s = readRDS(file = paste0(outFolder,"obj.filt.TFAnalysis.rds"))

DefaultAssay(seu.s) <- 'ATAC'

# Let's make sure we are working with correct clusters
Idents(seu.s)  = seu.s$ATAC_snn_res.0.5
```

Running `Monocle`

Before we start, let us recap some general information about the `Monocle 3` package. It has been designed for single-cell RNA-seq data, and its main types of analysis are: - Clustering, classifying, and counting cells. scRNA-Seq experiments allow you to discover new (and possibly rare) subtypes of cells. - Constructing single-cell trajectories. In development, disease, and throughout life, cells transition from one state to another. - Differential expression analysis. Characterizing new cell types and states begins with comparisons to other

In this vignette, we will mainly focus on constructing single-cel accessibility trajectories, which in combination with the `Cicero` package can also be used with scATAC-seq data. The first type of analysis we have already done before for scATAC-seq data, while the last analysis - differential expression analysis - can be replaced by differential accessibility analyses also with the help of `Cicero`.

Convert to `cell_data_set` format

```
traj.cds <- as.cell_data_set(seu.s, graph = "ATAC_snn")
```

```
## Warning: Monocle 3 trajectories require cluster partitions, which Seurat does
## not calculate. Please run 'cluster_cells' on your cell_data_set object
```

```
traj.cds <- cluster_cells(cds = traj.cds, reduction_method = "UMAP", graph = "ATAC_snn")
```

Learn principal graph from the reduced dimension space using reversed graph embedding

`Monocle3` aims to learn how cells transition through a biological program of gene expression changes in an experiment. Each cell can be viewed as a point in a high-dimensional space, where each dimension describes the expression of a different gene. Identifying the program of gene expression changes is equivalent to learning a trajectory that the cells follow through this space. However, the more dimensions there are in the analysis, the harder the trajectory is to learn. Fortunately, many genes typically co-vary with one another, and so the dimensionality of the data can be reduced with a wide variety of different algorithms. `Monocle3` provides two different algorithms for dimensionality reduction via `Monocle3` (UMAP and tSNE). Both take a `Monocle3` object and a number of dimensions allowed for the reduced space. You can also provide a model formula indicating some variables (e.g. batch ID or other technical factors) to "subtract" from the data so it doesn't contribute to the trajectory. The function `Monocle3` is the fourth step in the trajectory building process after `preprocess_cds`, `reduce_dimension`, and `cluster_cells`. After `learn_graph`, `order_cells` is typically called.

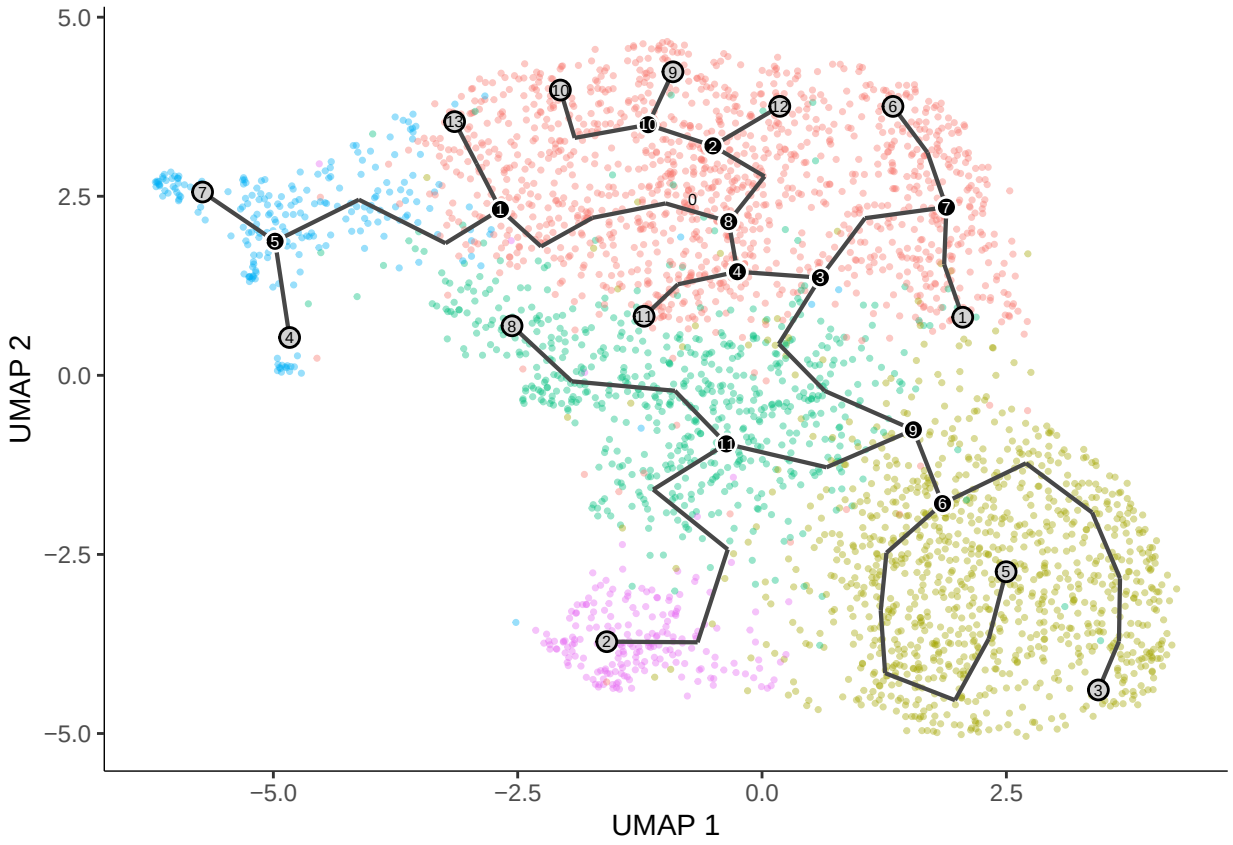
Explore with `minimal_branch_len` 1 To get a better understanding and feeling for the data, we are now exploring the effect of one of the parameters for the `learn_graph` function: The customization that is possible via the `learn_graph_control` parameter, which contains a subparameter called `minimal_branch_len`. First, use the R help to understand what it does!

As you have read by now (hopefully), `minimal_branch_len` denotes the length of the diameter path for a branch to be preserved during graph pruning procedure, with a default is 10. Here, we try a minimal value of 1 first:

After learning the graph, let's first plot the cells along with their trajectories to get a general overview. At this point, we first have to see the general graph before deciding on a potential root node - the origin of the trajectory. Without this guess, we cannot call the `order_cells` function - the last step in the aforementioned `Monocle3` trajectory building process!

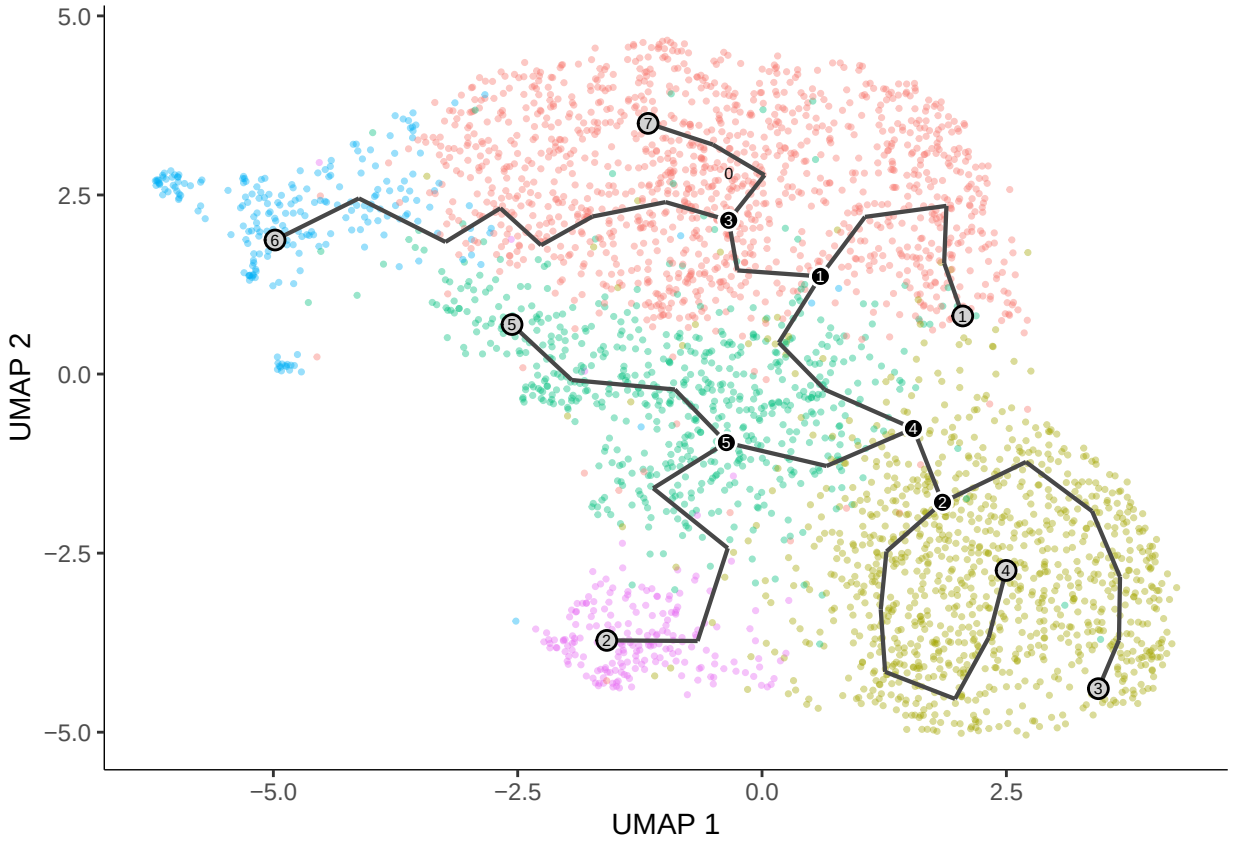
```
traj.cds <- learn_graph(traj.cds, use_partition = TRUE, close_loop = FALSE,
                        learn_graph_control = list(rann.k = 20, minimal_branch_len = 1))

# Plots the cells along with their trajectories.
plot_cells(cds = traj.cds, show_trajectory_graph = TRUE,
           color_cells_by = "ATAC_snn_res.0.5",
           label_cell_groups = TRUE,
           cell_size = 0.75, alpha = .4,
           label_principal_points = FALSE)
```

```
## Warning: `select_()` was deprecated in dplyr 0.7.0.
## Please use `select()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

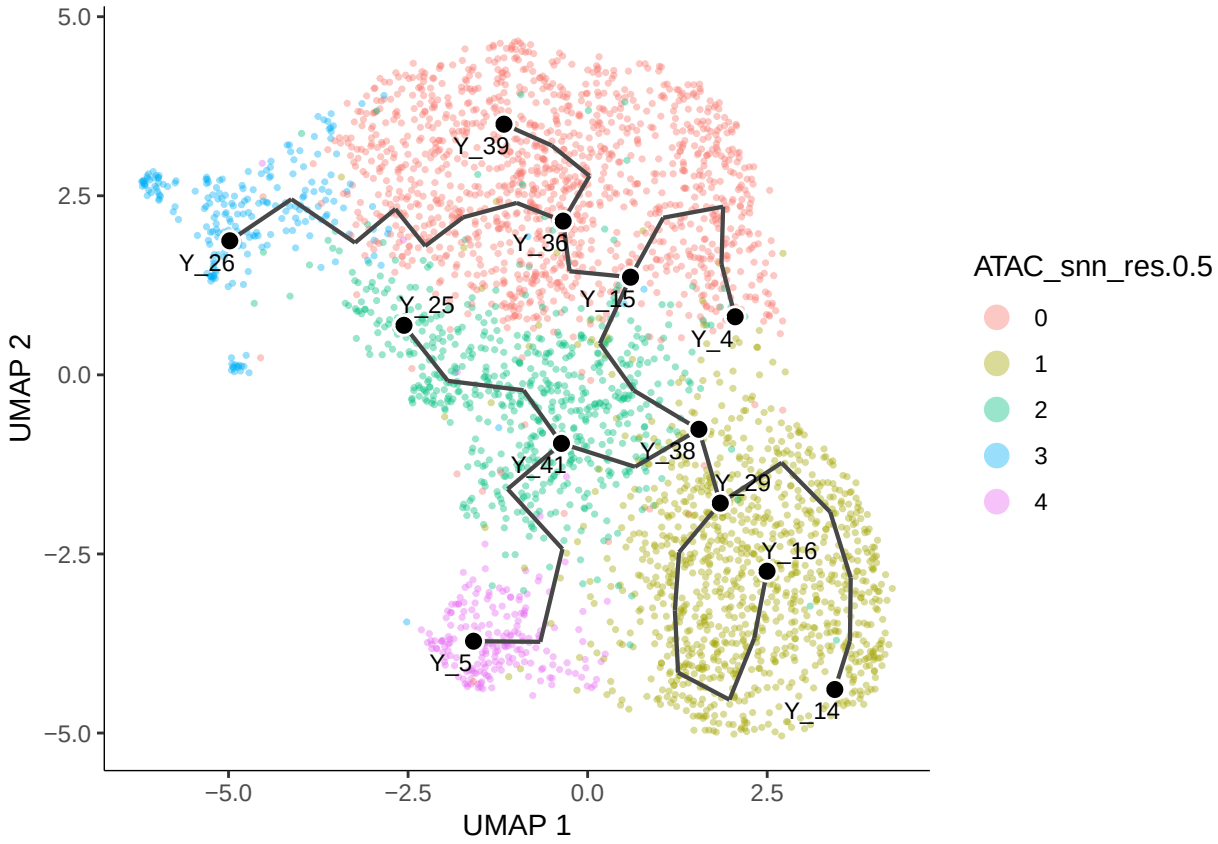


Explore with `minimal_branch_len 3` Let's repeat the same, but with a higher value for `minimal_branch_len`. If you want, create your own function that runs both `learn_graph` and `plot_cells` with `minimal_branch_len` as a paramter to avoid code redundancy!

```
traj.cds <- learn_graph(traj.cds, use_partition = TRUE, close_loop = FALSE,
                        learn_graph_control = list(rann.k = 20, minimal_branch_len = 3))


plot_cells(cds = traj.cds, show_trajectory_graph = TRUE,
           color_cells_by = "ATAC_snn_res.0.5",
           label_cell_groups = TRUE,
           cell_size = 0.75, alpha = .4,
           label_principal_points = FALSE)
```




```
# Learn what the two parameters that we change here do!
plot_cells(cds = traj.cds, show_trajectory_graph = TRUE,
           color_cells_by = "ATAC_snn_res.0.5",
           label_cell_groups = FALSE,
           cell_size = 0.75, alpha = .4,
           label_principal_points = TRUE)
```



The black lines show the structure of the graph. Note that the graph is not fully connected: cells in different partitions are in distinct components of the graph. The circles with numbers in them denote special points within the graph. Each leaf, denoted by light gray circles, corresponds to a different outcome (i.e. cell fate) of the trajectory. Black circles indicate branch nodes, in which cells can travel to one of several outcomes. You can control whether or not these are shown in the plot with the label_leaves and label_branch_points arguments to plot_cells. Please note that numbers within the circles are provided for reference purposes only.

Now that we have a sense of where the early cells fall, we can call `order_cells()`, which will calculate where each cell falls in pseudotime. In order to do so `order_cells()` needs you to specify the root nodes of the trajectory graph. If you don't provide them as an argument, it will launch a graphical user interface for selecting one or more root nodes (see the line that is commented out in the chunk above).

Note that some of the cells are gray. This means they have infinite pseudotime, because they were not reachable from the root nodes that were picked. In general, any cell on a partition that lacks a root node will be assigned an infinite pseudotime. In general, you should choose at least one root per partition.

Cell ordering and pseudotime generation

Once we've learned a graph, we are ready to order the cells according to their progress through the developmental program. `Monocle` measures this progress in *pseudotime*, a measure of how much progress an individual cell has made through a process such as cell differentiation.

Let's dive a bit more into the concept of pseudotime. In many biological processes, cells do not progress in perfect synchrony. In single-cell expression studies of processes such as cell differentiation, captured cells might be widely distributed in terms of progress. That is, in a population of cells captured at exactly the same time, some cells might be far along, while others might not yet even have begun the process. This asynchrony creates major problems when you want to understand the sequence of regulatory changes that occur as cells transition from one state to the next. Tracking the expression across cells captured at the

same time produces a very compressed sense of a gene's kinetics, and the apparent variability of that gene's expression will be very high.

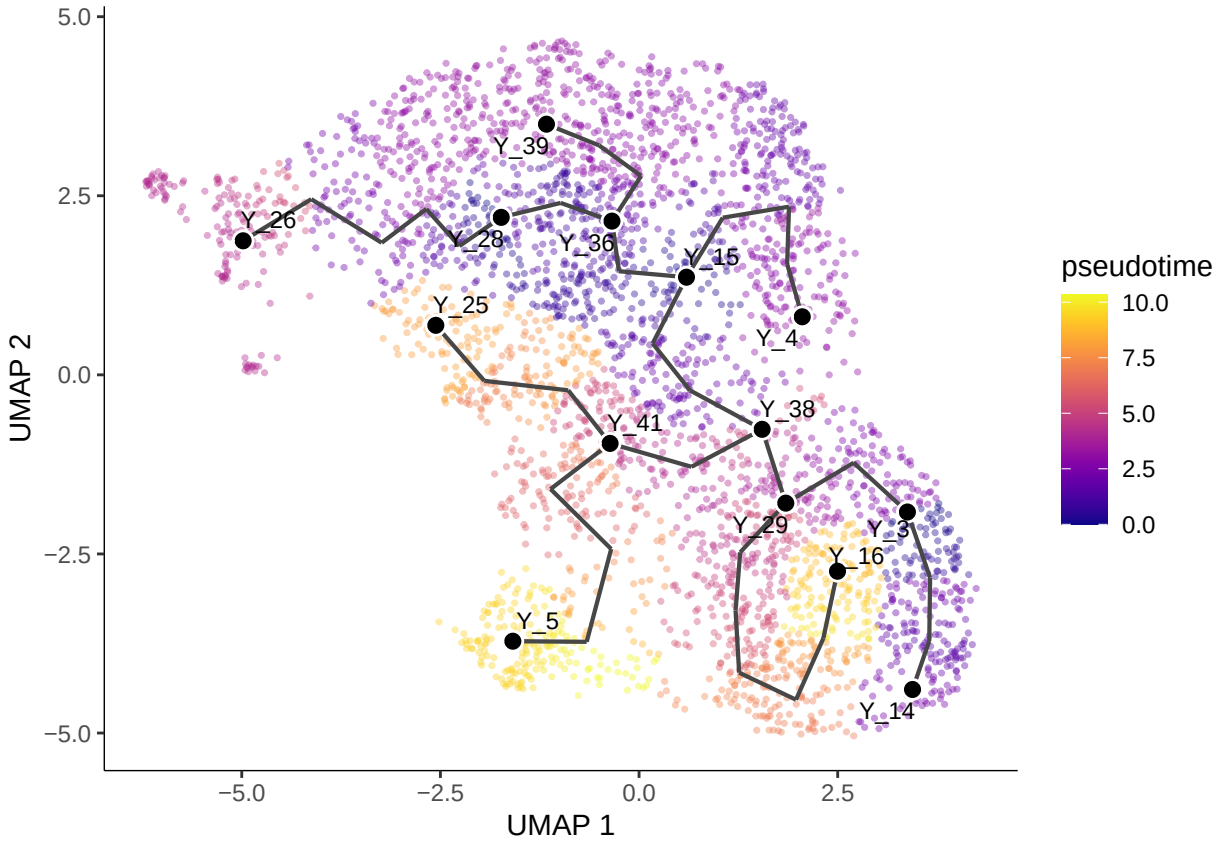
By ordering each cell according to its progress along a learned trajectory, `Monocle` alleviates the problems that arise due to asynchrony. Instead of tracking changes in expression as a function of time, `Monocle` tracks changes as a function of progress along the trajectory, which is termed *pseudotime*. Pseudotime is an abstract unit of progress: it's simply the distance between a cell and the start of the trajectory, measured along the shortest path. The trajectory's total length is defined in terms of the total amount of transcriptional change that a cell undergoes as it moves from the starting state to the end state.

Thus, we can assigns cells a pseudotime value based on their projection on the principal graph learned in the `learn_graph` function and the position of chosen root states. This function takes as input a `cell_data_set` and returns it with pseudotime information stored internally. `order_cells()` optionally takes "root" state(s) in the form of cell or principal graph node IDs, which you can use to specify the start of the trajectory. If you don't provide a root state, an plot will be generated where you can choose the root state(s) interactively. The trajectory will be composed of segments.

How do we identify the starting point of differentiation, which is often denoted the `root node` for trajectory analyses? In order to place the cells in order, we need to tell`Monocle` where the "beginning" of the biological process is. We do so by choosing regions of the graph that we mark as "roots" of the trajectory. In time series experiments, this can usually be accomplished by finding spots in the UMAP space that are occupied by cells from early time points. Here, for our data and the biology, one strategy is to look for pluripotency signatures, which indicates here that cluster `1` may be the origin.

```
# If you dont provide root_pr_nodes, an interactive window opens
# traj.cds <- order_cells(traj.cds, reduction_method = "UMAP")
traj.cds <- order_cells(traj.cds, reduction_method = "UMAP", root_pr_nodes = c("Y_15","Y_28","Y_3"))


# plot trajectories colored by pseudotime
plot_cells(cds = traj.cds, show_trajectory_graph = TRUE,
           color_cells_by = "pseudotime",
           cell_size = 0.75, alpha = .4,
           label_principal_points = TRUE)
```

```
## Cells aren't colored in a way that allows them to be grouped.
```



We just ordered cells and identified different potential trajectories! Let us explore them in more detail! For this, we will use `choose_graph_segments` to choose cells along the path of a principal graph. This allows us to essentially group the cells into different subsets, based on specific trajectories.

In general, it's often desirable to specify the root of the trajectory programmatically, rather than manually picking it as we did above. This can be done by first grouping the cells according to which trajectory graph node they are nearest to. Then, one can calculate what fraction of the cells at each node come from the earliest time point. Then one picks the node that is most heavily occupied by early cells and returns that as the root. For a specific example, take a look at the function `get_earliest_principal_node` here. **A task for those who like being challenged: Can you adapt the function for your data and identify which root node it picks?**

Splitting into different subpopulations based on the infered trajectory

In general, it is often useful to subset cells based on their branch in the trajectory. let's try this out now in practise!

Creating a helper function From the data we gathered, one could argue that cluster 3 has cells differentiating into the mesoderm lineage (vascular, immune related genes/TFs), and a lot of activity-induced Jun/Fos genes. Let's create a hypothetical mesoderm-specific cell subset. Because we will re-use the same code for other cell subsets, we again write a small custom function here which takes the set of starting and ending nodes as input and outputs a subset object as well as some plots.

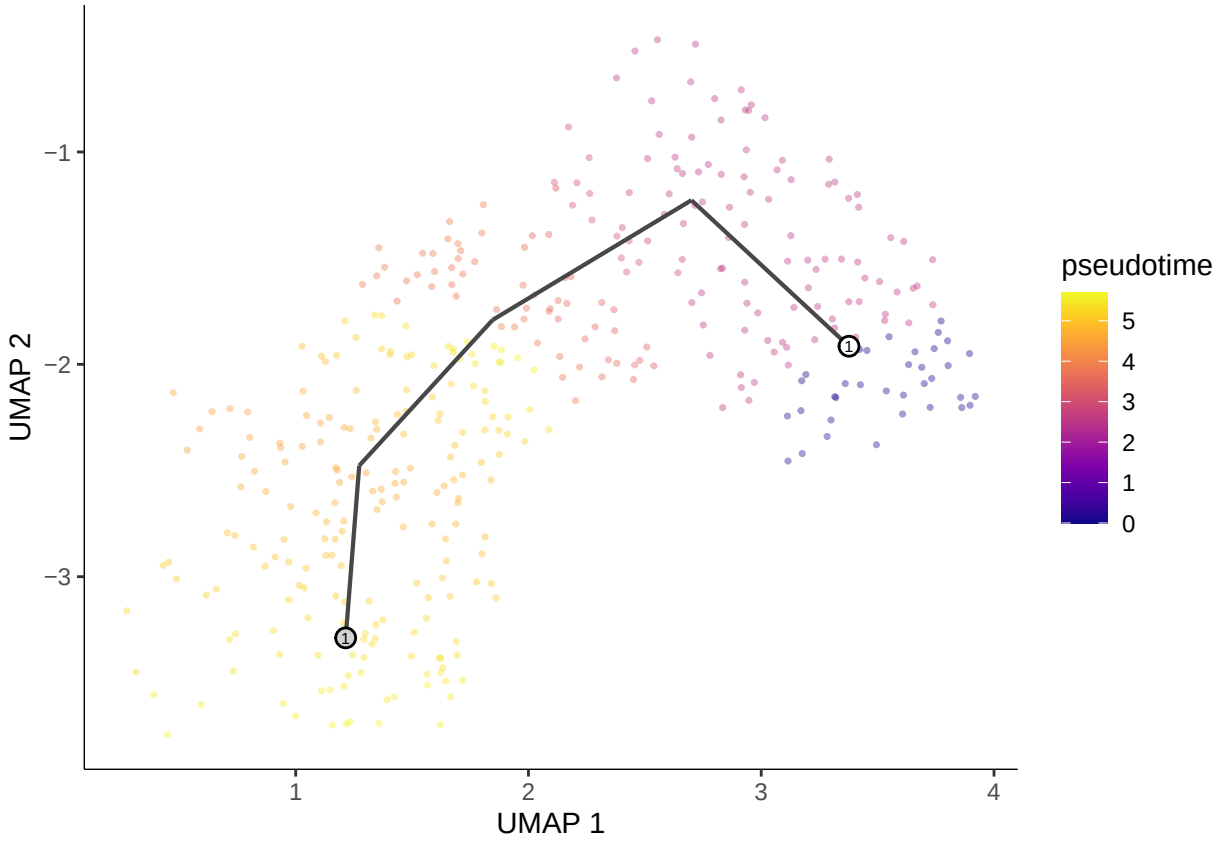
```
subsetAndPlotTrajectory <- function(obj, startingNodes, endNodes) {

    # traj.cds.sub  = choose_graph_segments(traj.cds, clear_cds = F) # interactive
    obj.sub  = choose_graph_segments(obj, clear_cds = FALSE,
                                     starting_pr_node = startingNodes, # only one
```

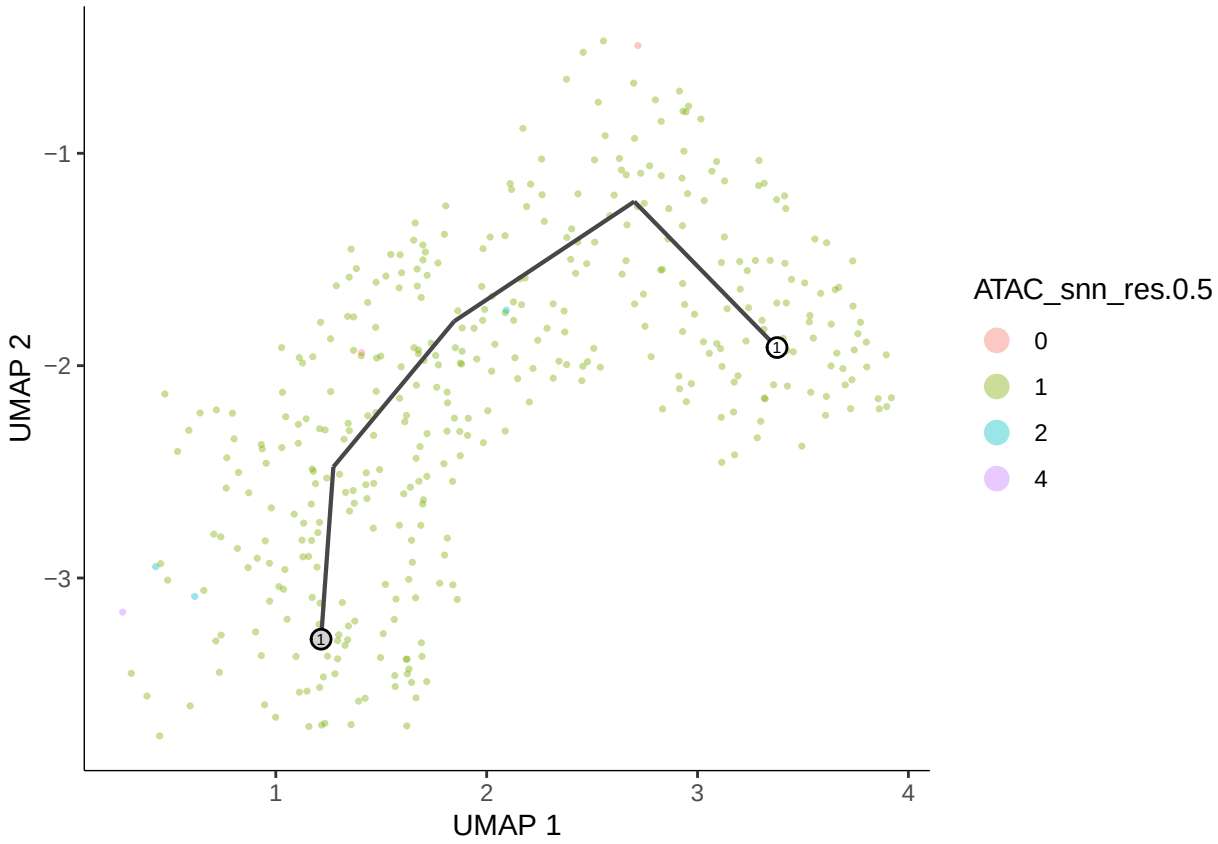
```
                                        ending_pr_nodes = endNodes)

    # We can now order the cells based on the subset
    obj.sub  <- order_cells(obj.sub, reduction_method = "UMAP", root_pr_nodes = startingNodes)

    plot_cells(cds = obj.sub, show_trajectory_graph = TRUE,
               color_cells_by = "pseudotime",
               cell_size = 0.75, alpha = .4,
               label_principal_points = FALSE) %>% plot()

    plot_cells(cds = obj.sub, show_trajectory_graph = TRUE,
               color_cells_by = "ATAC_snn_res.0.5",
               label_cell_groups = FALSE,
               cell_size = 0.75, alpha = .4,
               label_principal_points = FALSE) %>% plot()

    obj.sub

}
```

Hypothetical mesoderm cells We can now try and run our new function:

```
meso.traj.sub = subsetAndPlotTrajectory(traj.cds, startingNodes = c("Y_3"), endNodes = c("Y_23"))
```

```
## Cells aren't colored in a way that allows them to be grouped.
```

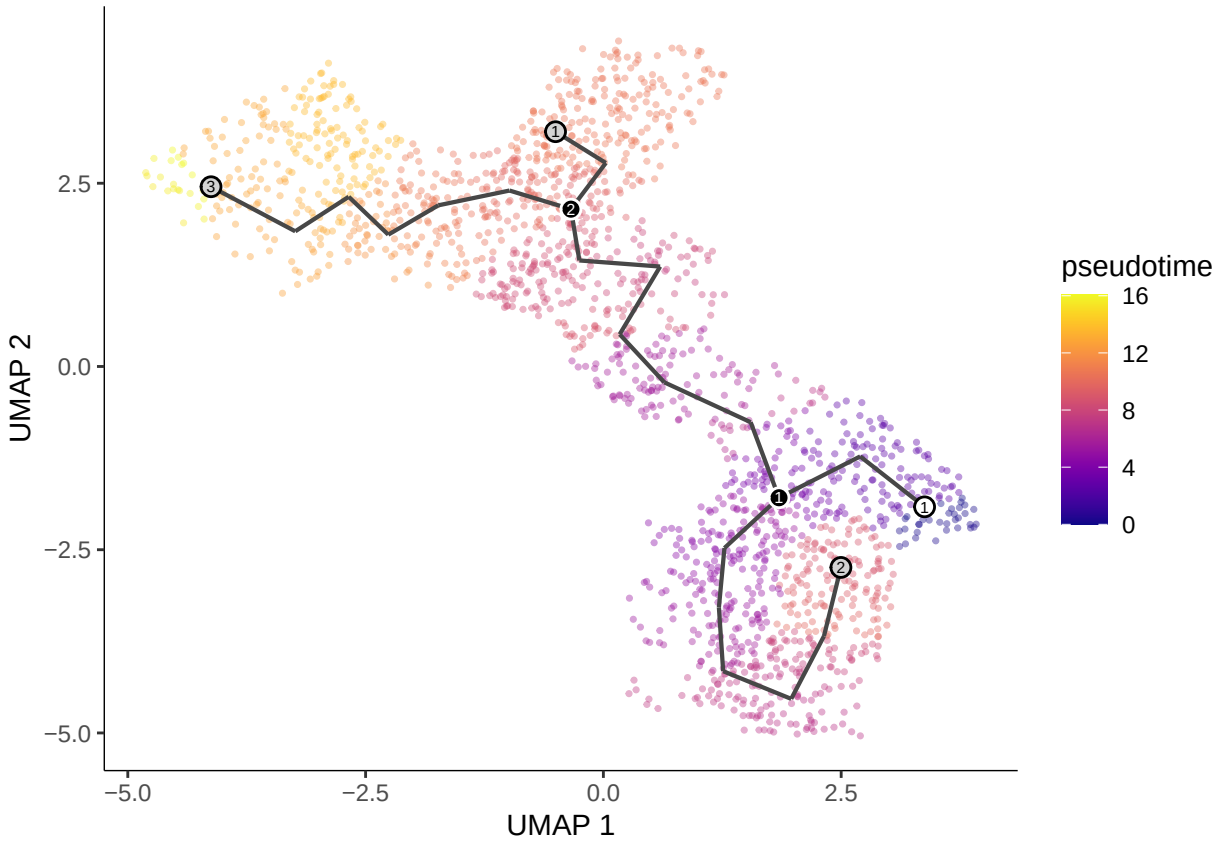


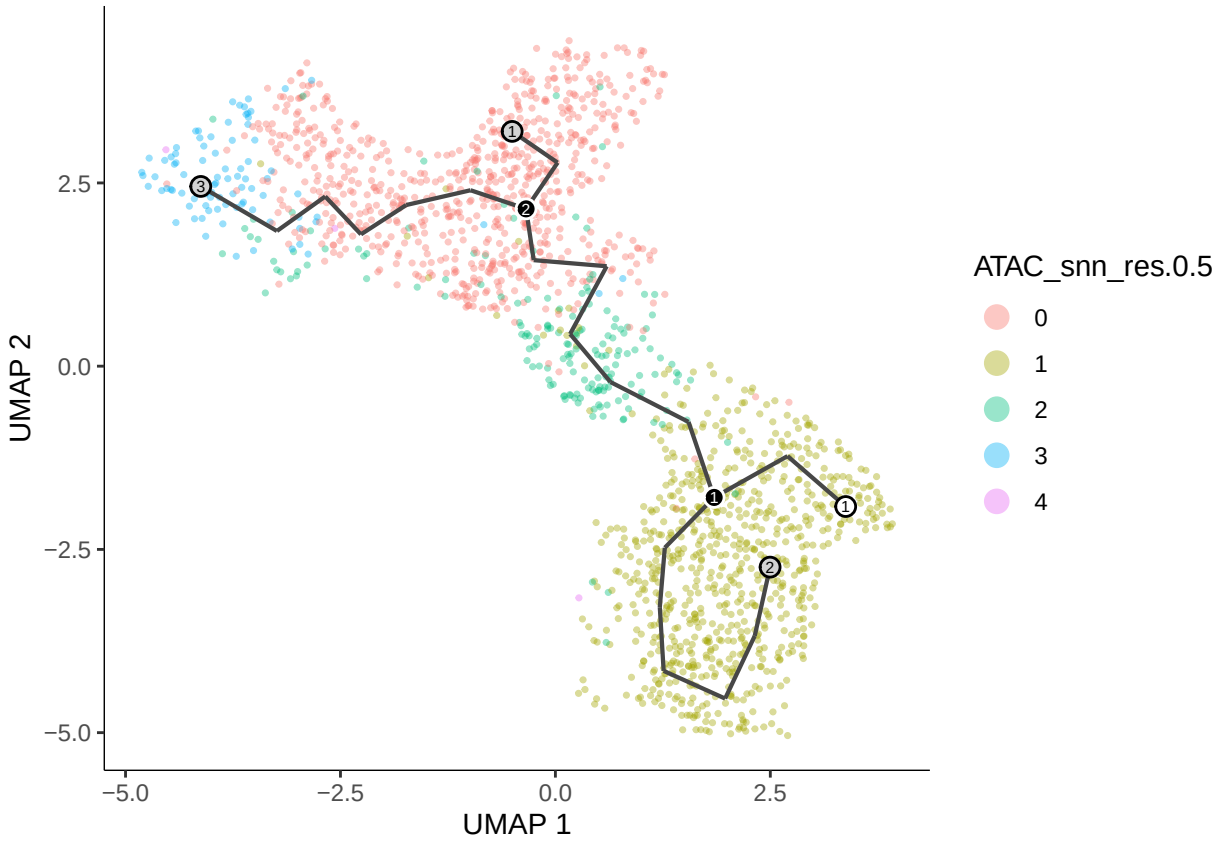


Hypothetical neuronal cells Let's do the same for potentially neuronal cells:

```
neu.traj.sub = subsetAndPlotTrajectory(traj.cds, startingNodes = c("Y_3"), endNodes = c("Y_16","Y_13","
```

```
## Cells aren't colored in a way that allows them to be grouped.
```


pseudotime
16
12
8
4
0
UMAP 2
UMAP 1
2.5
0.0
−2.5
−5.0
−5.0
−2.5
0.0
2.5

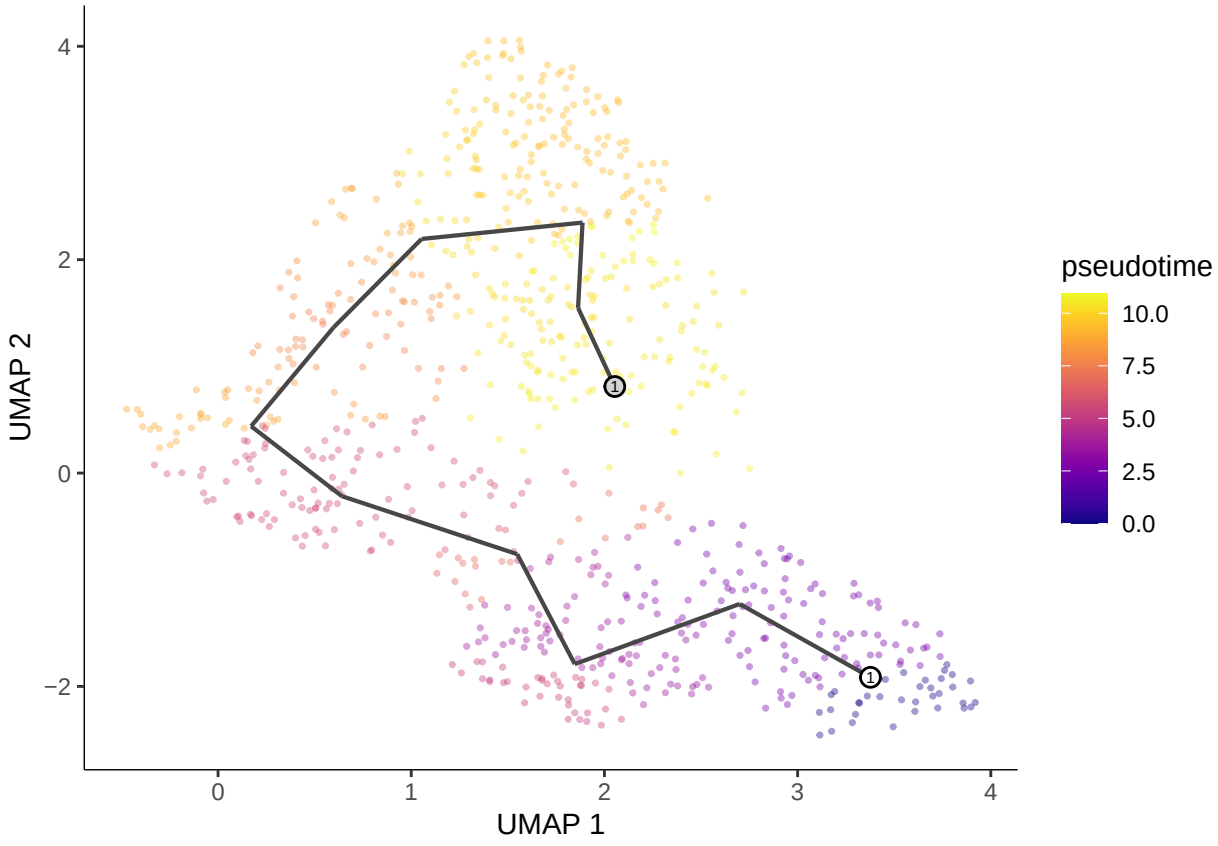

ATAC_snn_res.0.5
0
1
2
3
4
UMAP 2
UMAP 1
2.5
0.0
−2.5
−5.0
−5.0
−2.5
0.0
2.5

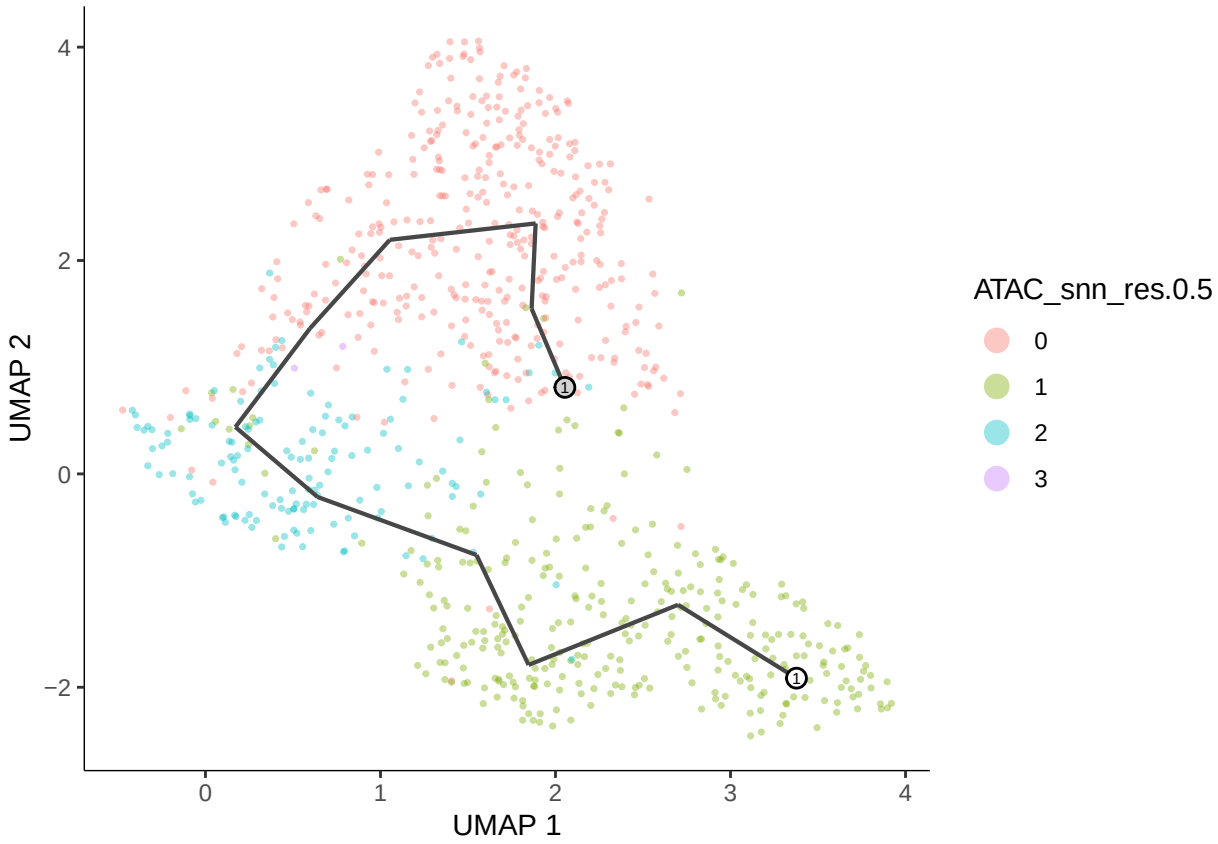
Hypothetical heterogenous cells

Lastly, a group of heterogeneous cells:

```
und.traj.sub = subsetAndPlotTrajectory(traj.cds, startingNodes = c("Y_3"), endNodes = c("Y_4"))
```

```
## Cells aren't colored in a way that allows them to be grouped.
```





Visualize pseudotime per TF

We will now try to combine the pseudotime and trajectory analyses in combination with the `chromVAR` derived TF activity scores to find out whether some TFs are changing their activity with pseudotime.

Add pseudotime to `Seurat` object

Let's first add the pseudotime to our original `Seurat` object in the form of additional metadata! This will allow us to color cells by their pseudotime, subset the object according to it, and so forth. We will create 3 groups here, one for each of our hypothesized cell populations!

```
# Add pseudotime information from each of the 3 sub-populations we identified before

seu.s <- AddMetaData(
  object = seu.s,
  metadata = neu.traj.sub@principal_graph_aux@listData$UMAP$pseudotime,
  col.name = "pseudotime.neu"
)

seu.s <- AddMetaData(
  object = seu.s,
  metadata = meso.traj.sub@principal_graph_aux@listData$UMAP$pseudotime,
  col.name = "pseudotime.meso"
)

seu.s <- AddMetaData(
  object = seu.s,
```

```
  metadata = und.traj.sub@principal_graph_aux@listData$UMAP$pseudotime,
  col.name = "pseudotime.und"
)

# Create a separate Seurat object for each trajectory
seu.sub.neu = subset(seu.s, subset = pseudotime.neu > 0)
seu.sub.meso = subset(seu.s, subset = pseudotime.meso > 0)
seu.sub.und = subset(seu.s, subset = pseudotime.und > 0)
```

TF visualisation

Now, we will try to visualize the TF activity that we derived before in the TF analysis vignette and use the `chromVAR` scores for the accessibility trajectories. In order to minimize code repetition, we will again write a custom function so that we can call it for different TFs and the different cell subsets we created before in an easy and reproducible manner:

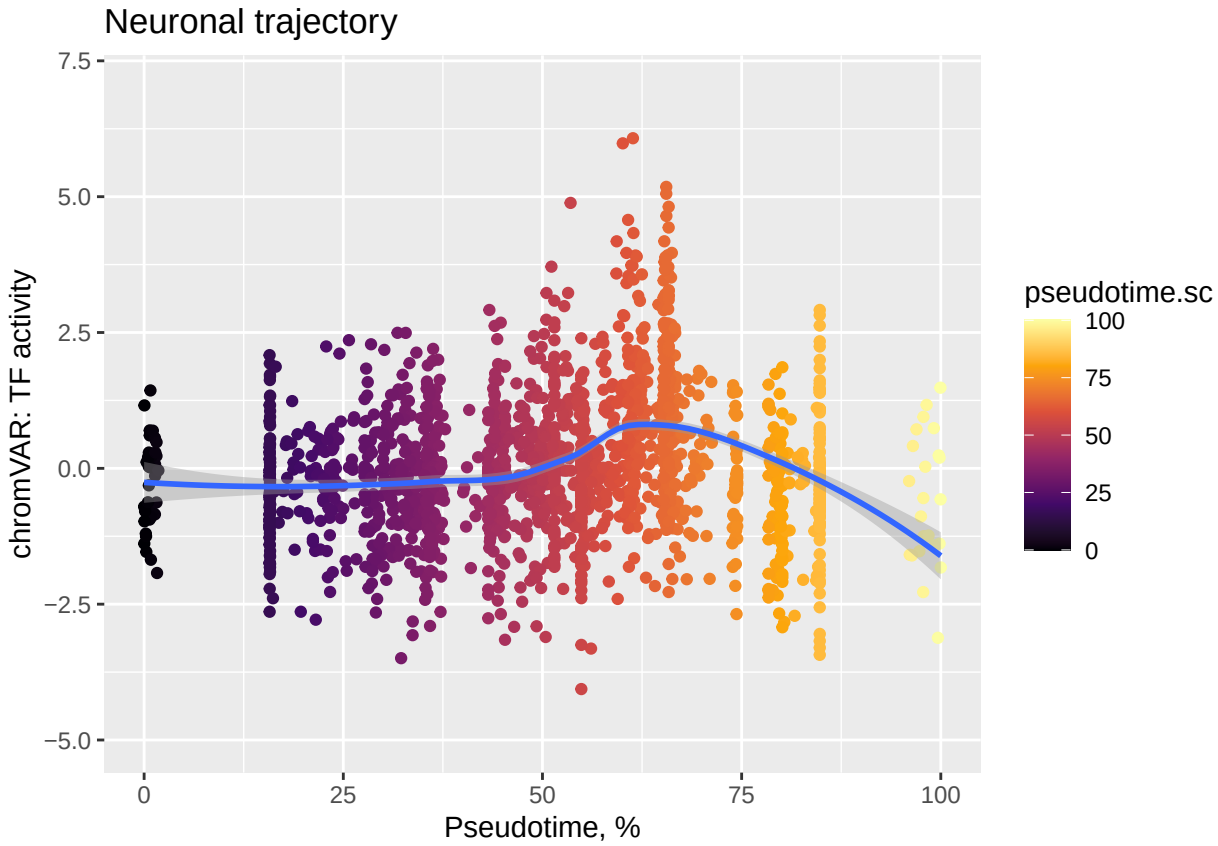
```
visualizeTFTrajectory <- function (TF, obj, pseudotimeColName, trajectoryName, ylim = c(-5,7)) {

    df = tibble(pseudotime = obj@meta.data[[c(pseudotimeColName)]],
                TF         = obj@assays$chromvar@data[TF,]) %>%
         mutate(pseudotime.sc = rescale(pseudotime, to = c(0,100)))

    # ylim = c(floor(range(df$TF)[1]),ceiling(range(df$TF)[2]))

    g = ggplot(df, aes(x = pseudotime.sc, y = TF, color = pseudotime.sc)) +
        geom_point() +
        xlab("Pseudotime, %") + ylab("chromVAR: TF activity") + ylim(ylim) +
        scale_color_viridis_c(option = "B") +
        geom_smooth(method = "loess") +  # Jitter on x-axis a bit
        ggtitle(trajectoryName)

    plot(g)

}
```

Feel free to customize this function even further and to add function parameters for increased flexibility. For example, you may want to add an argument that determines the rescaling in a more flexible manner rather than the hard-coded `c(0,100)` in the current version, or the pseudotime coloring.

Nevertheless, let's try our new function for some TFs. We start with `PITX1-MOUSE.H11MO.0.C`:

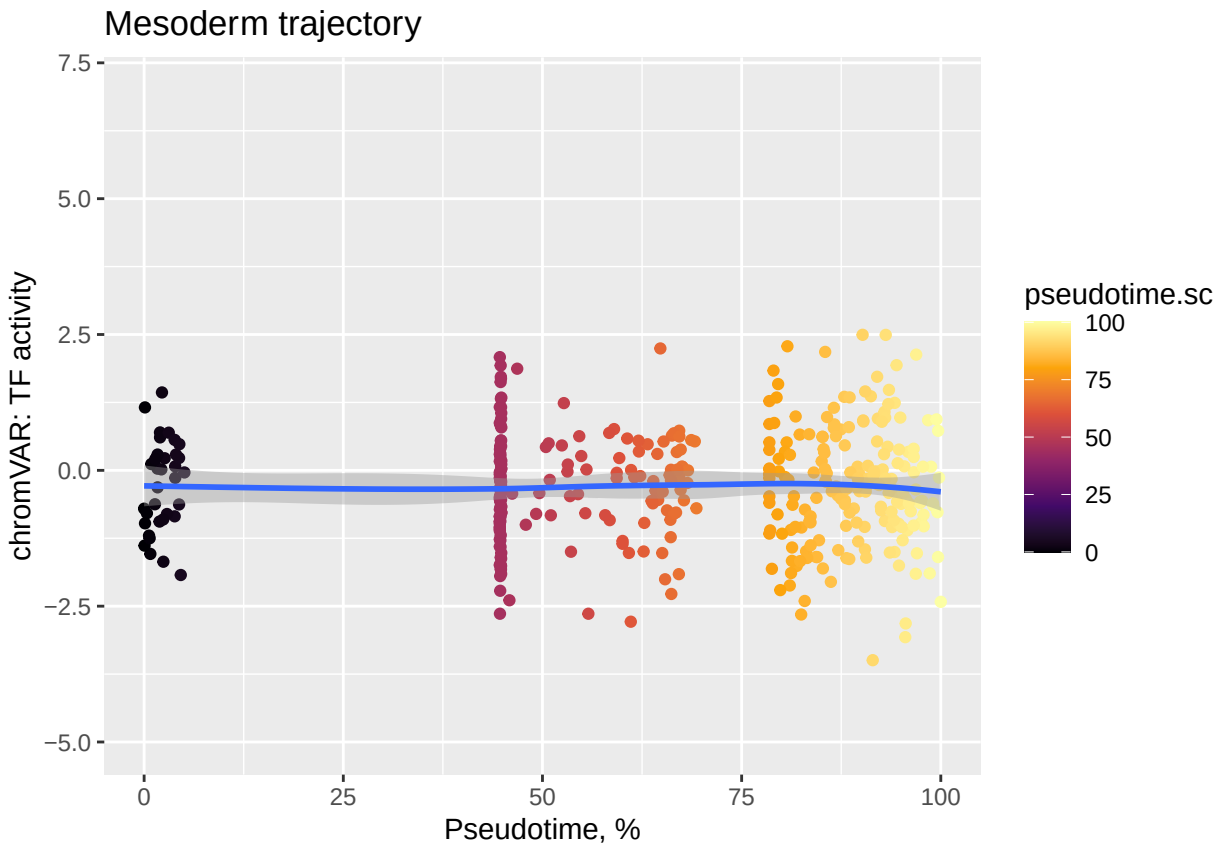
```
TF = "PITX1-MOUSE.H11MO.0.C"

visualizeTFTrajectory(TF = TF,
                      obj = seu.sub.neu, pseudotimeColName = "pseudotime.neu",
                      trajectoryName = "Neuronal trajectory")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



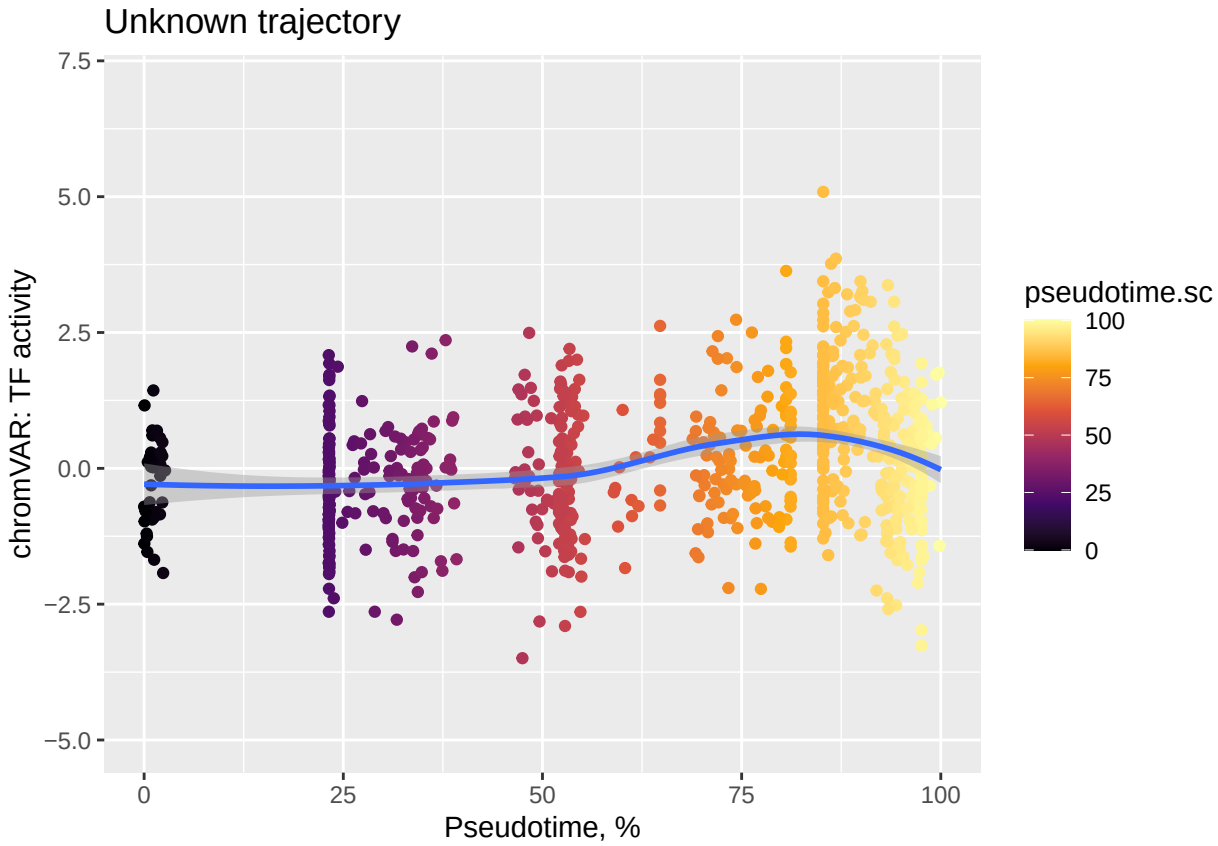

```
visualizeTFTrajectory(TF = TF,
                      obj = seu.sub.meso, pseudotimeColName = "pseudotime.meso",
                      trajectoryName = "Mesoderm trajectory")
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
visualizeTFTrajectory(TF = TF,
                      obj = seu.sub.und, pseudotimeColName = "pseudotime.und",
                      trajectoryName = "Unknown trajectory")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Try this with a few other TFs, which TFs give very different or very clear TF trajectories? Play around and identify candidate TFs that seem to be differential among the three classes we subset here.

Pseudotime TF heatmap based on TF activities

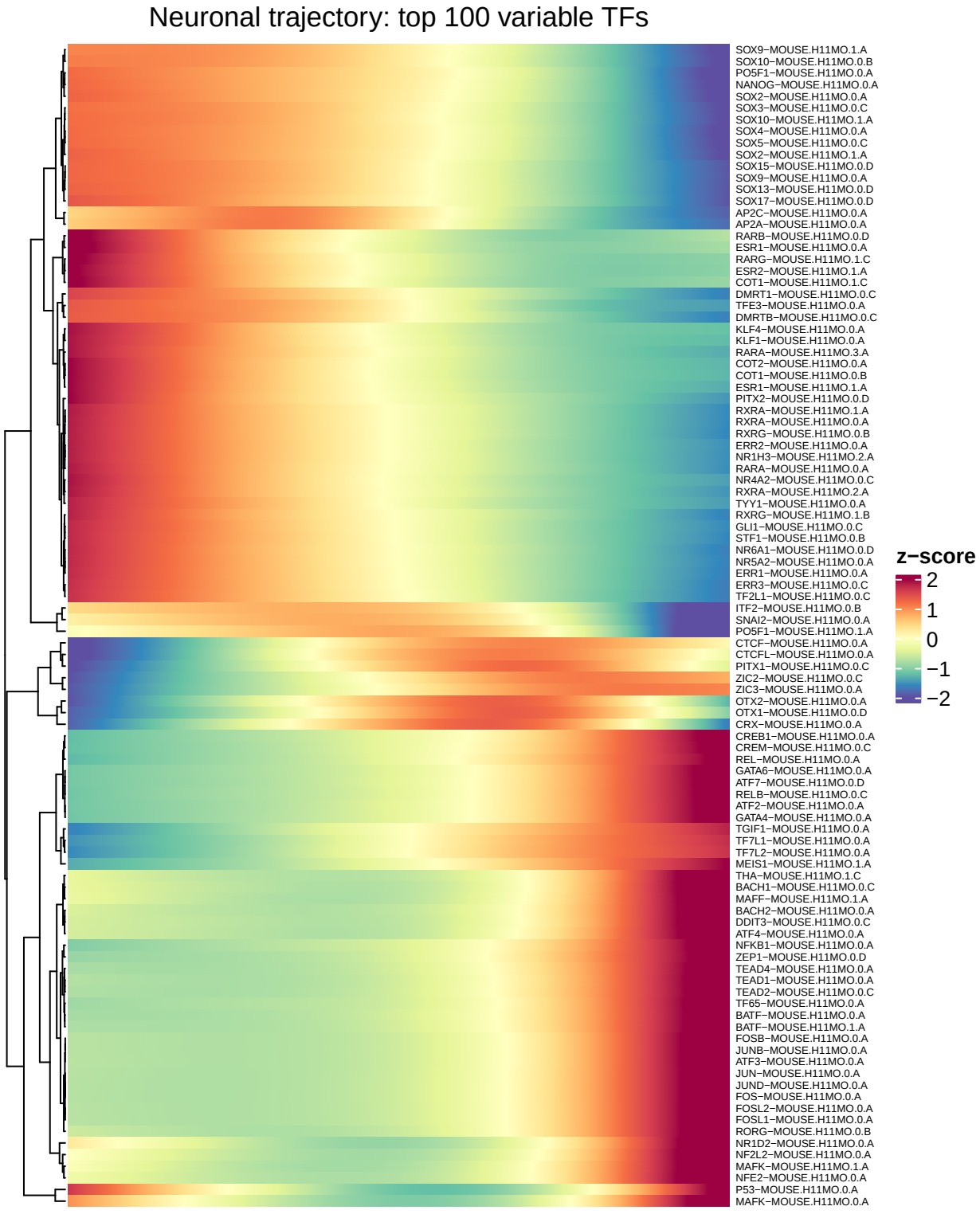
We can also create a pseudotime TF heatmap based on TF activity scores for each sub trajectory. We again first create a custom function that we can call multiple times afterwards:

```
createHeatmap <- function(obj, pseudotimeColName, title = "Trajectory", topTFs = 100) {

  # Code taken and adjusted from here: https://github.com/crickbabs/ZebrafishDevelopingHindbrainAtlas

  TFdata = obj@assays$chromvar@data
  # take X most variable TFs
  TFdata.f = head(TFdata[order(rowSds(TFdata), decreasing = TRUE),], topTFs)

  # TF normalization
  pseudotimeData = obj@meta.data[pseudotimeColName]
  # Fix to the same name here for easier code
  colnames(pseudotimeData) = "pseudotime"

  pseudotimeData = pseudotimeData %>% rownames_to_column("cellID") %>% arrange(pseudotime)
  pt.matrix <- TFdata.f[, pseudotimeData$cellID]
  pt.matrix <- t(apply(pt.matrix,1,function(x){smooth.spline(x,df=3)$y}))
  pt.matrix <- t(apply(pt.matrix,1,function(x){(x-mean(x))/sd(x)}))
```

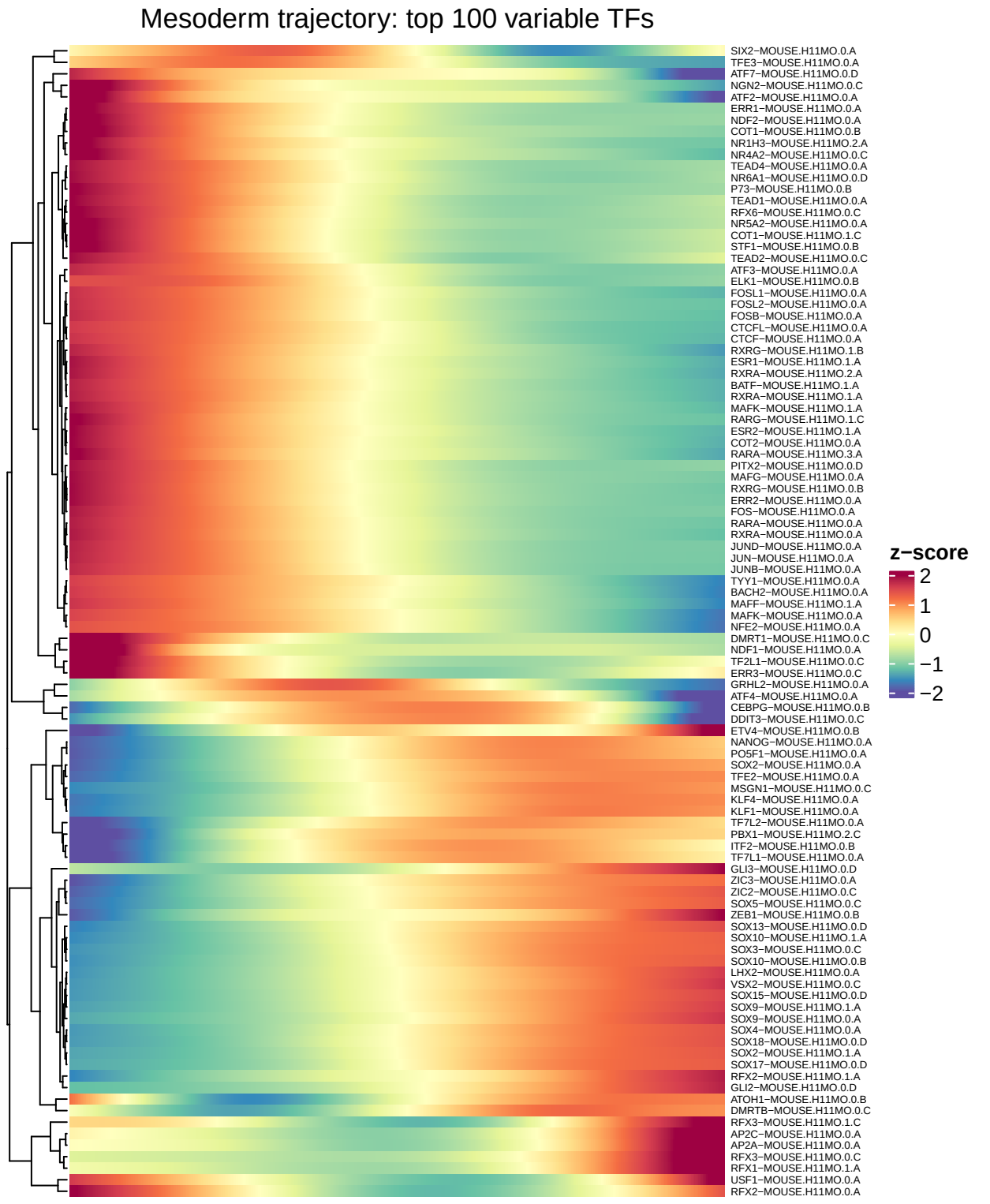
```
  # rownames(pt.matrix) <- genes;

  ht <- Heatmap(
    pt.matrix,
    column_title                = paste0(title, ": top ", topTFs, " variable TFs"),
    name                        = "z-score",
    col                         = colorRamp2(seq(from=-2,to=2,length=11),rev(brewer.pal(11, "Spectral"))
    show_row_names              = TRUE,
    show_column_names           = FALSE,
    row_names_gp                = gpar(fontsize = 5),
    # km = 6, if cluster_rows = TRUE
    row_title_rot               = 0,
    cluster_rows                = TRUE, # TRUE
    cluster_row_slices          = FALSE,
    cluster_columns             = FALSE)

  print(ht)


}
```

Similar to before, we can now plot the pseudotime TF heatmap for each of the three cell clusters:

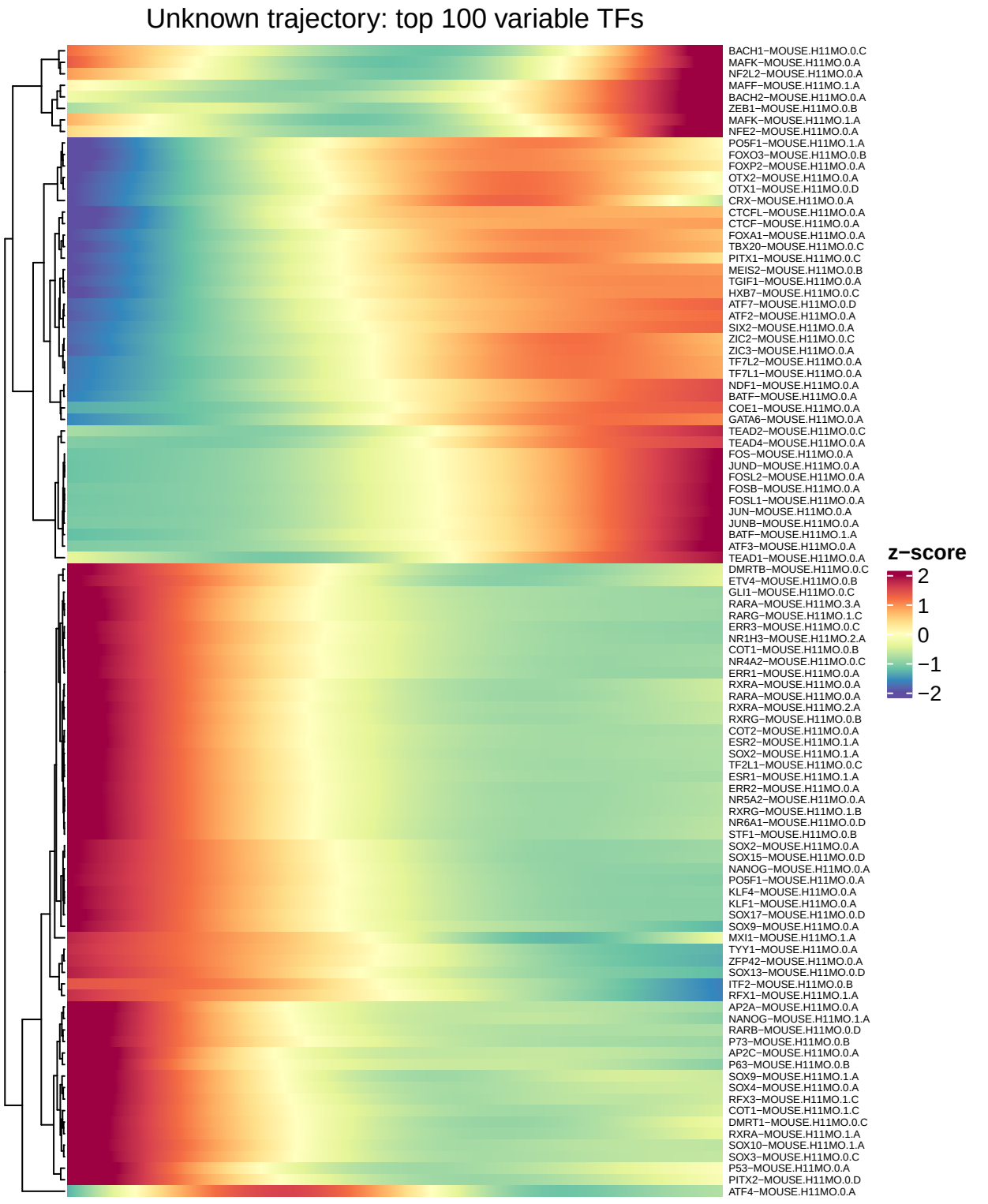
```
createHeatmap(obj = seu.sub.neu, pseudotimeColName = "pseudotime.neu", title = "Neuronal trajectory", t
```




```
createHeatmap(obj = seu.sub.meso, pseudotimeColName = "pseudotime.meso", title = "Mesoderm trajectory",
```




```
createHeatmap(obj = seu.sub.und, pseudotimeColName = "pseudotime.und", title = "Unknown trajectory", to
```



This concludes the vignette, and from here, you can continue exploring the data, connecting different assays and data together, and make sense out of the data and explore new biology!

Save object to disk

We reached the end of the vignette. We again save our updated `Seurat` object to disk with a new name, in analogy to what we did in the other vignettes.

```
saveRDS(seu.s, file = paste0(outFolder,"obj.filt.monocle.rds"))
```

Further reading

Pliner, H.A., Packer, J.S., McFaline-Figueroa, J.L., Cusanovich, D.A., Daza, R.M., Aghamirzaie, D., Srivatsan, S., Qiu, X., Jackson, D., Minkina, A. and Adey, A.C., 2018. Cicero predicts cis-regulatory DNA interactions from single-cell chromatin accessibility data. Molecular cell, 71(5), pp.858-871.

Session info

It is good practice to print the so-called session info at the end of an R script, which prints all loaded libraries, their versions etc. This can be helpful for reproducibility and recapitulating which package versions have been used to produce the results obtained above.

```
sessionInfo()
```

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.4 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] grid      stats4    stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] RColorBrewer_1.1-2          circlize_0.4.13
##  [3] ComplexHeatmap_2.10.0       scales_1.1.1
##  [5] patchwork_1.1.1             forcats_0.5.1
##  [7] stringr_1.4.0               dplyr_1.0.7
##  [9] purrr_0.3.4                 readr_2.1.1
## [11] tidyr_1.1.4                 tibble_3.1.6
## [13] ggplot2_3.3.5               tidyverse_1.3.1
## [15] monocle3_1.0.0              SingleCellExperiment_1.16.0
## [17] SummarizedExperiment_1.24.0 GenomicRanges_1.46.1
## [19] GenomeInfoDb_1.30.0         IRanges_2.28.0
## [21] S4Vectors_0.32.3            MatrixGenerics_1.6.0
## [23] matrixStats_0.61.0          Biobase_2.54.0
## [25] BiocGenerics_0.40.0         SeuratWrappers_0.3.0
## [27] sp_1.4-7                    SeuratObject_4.1.0
```

```
## [29] Seurat_4.1.1               Signac_1.6.0
##
## loaded via a namespace (and not attached):
##   [1] utf8_1.2.2             reticulate_1.24        R.utils_2.11.0
##   [4] tidyselect_1.1.1       htmlwidgets_1.5.4      docopt_0.7.1
##   [7] BiocParallel_1.28.3    Rtsne_0.16             munsell_0.5.0
##  [10] codetools_0.2-18       ica_1.0-2              future_1.25.0
##  [13] miniUI_0.1.1.1         withr_2.4.3            spatstat.random_2.2-0
##  [16] colorspace_2.0-2       progressr_0.10.0       highr_0.9
##  [19] knitr_1.37             rstudioapi_0.13        ROCR_1.0-11
##  [22] tensor_1.5             listenv_0.8.0          labeling_0.4.2
##  [25] slam_0.1-50            GenomeInfoDbData_1.2.7 polyclip_1.10-0
##  [28] farver_2.1.0           parallelly_1.31.1      vctrs_0.4.1
##  [31] generics_0.1.1         xfun_0.29              lsa_0.73.2
##  [34] ggseqlogo_0.1          doParallel_1.0.16      R6_2.5.1
##  [37] clue_0.3-60            rsvd_1.0.5             bitops_1.0-7
##  [40] spatstat.utils_2.3-0   DelayedArray_0.20.0    assertthat_0.2.1
##  [43] promises_1.2.0.1       rgeos_0.5-9            gtable_0.3.0
##  [46] globals_0.14.0         goftest_1.2-3          rlang_1.0.2
##  [49] RcppRoll_0.3.0         GlobalOptions_0.1.2    splines_4.1.2
##  [52] lazyeval_0.2.2         spatstat.geom_2.4-0    broom_0.7.11
##  [55] BiocManager_1.30.16    yaml_2.2.1             reshape2_1.4.4
##  [58] abind_1.4-5            modelr_0.1.8           backports_1.4.1
##  [61] httpuv_1.6.5           tools_4.1.2            ellipsis_0.3.2
##  [64] spatstat.core_2.4-2    proxy_0.4-26           ggridges_0.5.3
##  [67] Rcpp_1.0.8             plyr_1.8.6             zlibbioc_1.40.0
##  [70] RCurl_1.98-1.5         rpart_4.1-15           deldir_1.0-6
##  [73] GetoptLong_1.0.5       pbapply_1.5-0          viridis_0.6.2
##  [76] cowplot_1.1.1          zoo_1.8-10             haven_2.4.3
##  [79] ggrepel_0.9.1          cluster_2.1.2          fs_1.5.2
##  [82] magrittr_2.0.1         magick_2.7.3           data.table_1.14.2
##  [85] scattermore_0.8        reprex_2.0.1           lmtest_0.9-40
##  [88] RANN_2.6.1             SnowballC_0.7.0        fitdistrplus_1.1-8
##  [91] hms_1.1.1              mime_0.12              evaluate_0.14
##  [94] xtable_1.8-4           readxl_1.3.1           sparsesvd_0.2
##  [97] shape_1.4.6            gridExtra_2.3          compiler_4.1.2
## [100] KernSmooth_2.23-20     crayon_1.4.2           R.oo_1.24.0
## [103] htmltools_0.5.2        mgcv_1.8-38            later_1.3.0
## [106] tzdb_0.2.0             lubridate_1.8.0        DBI_1.1.2
## [109] tweenr_1.0.2           dbplyr_2.1.1           MASS_7.3-54
## [112] leidenbase_0.1.9       Matrix_1.4-0           cli_3.3.0
## [115] R.methodsS3_1.8.1      parallel_4.1.2         igraph_1.2.11
## [118] pkgconfig_2.0.3        plotly_4.10.0          spatstat.sparse_2.1-1
## [121] foreach_1.5.1          xml2_1.3.3             XVector_0.34.0
## [124] rvest_1.0.2            digest_0.6.29          sctransform_0.3.3
## [127] RcppAnnoy_0.0.19       spatstat.data_2.2-0    Biostrings_2.62.0
## [130] cellranger_1.1.0       rmarkdown_2.11         leiden_0.3.10
## [133] fastmatch_1.1-3        uwot_0.1.11            shiny_1.7.1
## [136] Rsamtools_2.10.0       rjson_0.2.21           lifecycle_1.0.1
## [139] nlme_3.1-153           jsonlite_1.7.2         viridisLite_0.4.0
## [142] fansi_1.0.0            pillar_1.6.4           lattice_0.20-45
## [145] fastmap_1.1.0          httr_1.4.2             survival_3.2-13
## [148] glue_1.6.0             remotes_2.4.2          qlcMatrix_0.9.7
## [151] iterators_1.0.13       png_0.1-7              ggforce_0.3.3
```

```
## [154] stringi_1.7.6          irlba_2.3.5            future.apply_1.9.0
```