Monocle: Cell counting, differential expression, and trajectory analysis for single-cell RNA-Seq experiments

Cole Trapnell
University of Washington,
Seattle, Washington, USA
coletrap@uw.edu

Davide Cacchiarelli
Harvard University,
Cambridge, Massachussetts, USA
davide@broadinstitute.org

Xiaojie Qiu
University of Washington,
Seattle, Washington, USA
xqiu@uw.edu

March 6, 2017

Abstract

Single cell gene expression studies enable profiling of transcriptional regulation during complex biological processes and within highly hetergeneous cell populations. These studies allow discovery of genes that identify certain subtypes of cells, or that mark a particular intermediate states during a biological process. In many single cell studies, individual cells are executing through a gene expression program in an unsynchronized manner. In effect, each cell is a snapshot of the transcriptional program under study. The package *monocle* provides tools for analyzing single-cell expression experiments. Monocle introduced the strategy of ordering single cells in *pseudotime*, placing them along a trajectory corresponding to a biological process such as cell differentiation. Monocle learns this trajectory directly from the data, in either a fully unsupervised or a semi-supervised manner. It also performs differential gene expression and clustering to identify important genes and cell states. It is designed for RNA-Seq studies, but can be used with other assays. For more information on the algorithm at the core of *monocle*, or to learn more about how to use single cell RNA-Seq to study a complex biological process, see Trapnell and Cacchiarelli *et al* [1]

Contents

1 Introduction

The *monocle* package provides a toolkit for analyzing single cell gene expression experiments. This vignette provides an overview of a single cell RNA-Seq analysis workflow with Monocle. Monocle was originally developed to analyze dynamic biological processes such as cell differentiation, although it also supports simpler experimental settings.

Monocle 2 includes new, improved algorithms for classifying and counting cells, performing differential expression analysis between subpopulations of cells, and reconstructing cellular trajectories. Monocle 2 has also been re-engineered to work well with very large single-cell RNA-Seq experiments containing tens of thousands of cells or even more. Monocle can help you perform three main types of analysis:

1. **Classifying and counting cells.** Single-cell RNA-Seq experiments allow you to discover new (and possibly rare) subtypes of cells. Monocle helps you identify them.
2. **Constructing single-cell trajectories.** In development, disease, and throughout life, cells transition from one state to another. Monocle helps you discover these transitions.
3. **Differential expression analysis.** Characterizing new cell types and states begins with comparing them to other, better understood cells. Monocle includes a sophisticated but easy to use system for differential expression.

Before we look at Monocle's functions for each of these common analysis tasks, let's see how to load up single-cell datasets in Monocle.

2 Getting started with Monocle

The *monocle* package takes a matrix of gene expression values as calculated by Cufflinks [2] or another gene expression estimation program. Monocle can work with relative expression values (e.g. FPKM or TPM units) or absolute transcript counts (e.g. from UMI experiments). Although Monocle can be used with raw read counts, these are not directly proportional to expression values unless you normalize them by length, so some Monocle functions could produce

nonsense results. If you don't have UMI counts, We recommend you load up FPKM or TPM values instead of raw read counts.

2.1 The CellDataSet class

monocle holds single cell expression data in objects of the *CellDataSet* class. The class is derived from the Bioconductor *ExpressionSet* class, which provides a common interface familiar to those who have analyzed microarray experiments with Bioconductor. The class requires three input files:

1. `exprs`, a numeric matrix of expression values, where rows are genes, and columns are cells
2. `phenoData`, an *AnnotatedDataFrame* object, where rows are cells, and columns are cell attributes (such as cell type, culture condition, day captured, etc.)
3. `featureData`, an *AnnotatedDataFrame* object, where rows are features (e.g. genes), and columns are gene attributes, such as biotype, gc content, etc.

The expression value matrix *must* have the same number of columns as the `phenoData` has rows, and it must have the same number of rows as the `featureData` data frame has rows. Row names of the `phenoData` object should match the column names of the expression matrix. Row names of the `featureData` object should match row names of the expression matrix.

You can create a new *CellDataSet* object as follows:

```
#not run
HSMM_expr_matrix <- read.table("fpkm_matrix.txt")
HSMM_sample_sheet <- read.delim("cell_sample_sheet.txt")
HSMM_gene_annotation <- read.delim("gene_annotations.txt")
```

Once these tables are loaded, you can create the CellDataSet object like this:

```
pd <- new("AnnotatedDataFrame", data = HSMM_sample_sheet)
fd <- new("AnnotatedDataFrame", data = HSMM_gene_annotation)
HSMM <- newCellDataSet(as.matrix(HSMM_expr_matrix), phenoData = pd, featureData = fd)
```

This will create a CellDataSet object with expression values measured in FPKM, a measure of relative expression reported by Cufflinks. By default, Monocle assumes that your expression data is log-normally distrubuted and uses a Tobit model to test for differential expression in downstream steps. However, if you're not using TPM or FPKM data, see below for how to tell Monocle how to model it in downstream steps

2.2 Choosing a distribution for expression data

Monocle works well with both relative expression data and count-based measures (e.g. UMIs). In general, it works best with transcript count data, especially UMI data. Whatever your data type, it is *critical* that specify the appropriate distribution for it. FPKM/TPM values are generally log-normally distributed, while UMIs or read counts are better modeled with the negative binomial. To work with count data, specify the negative binomial distribution as the `expressionFamily` argument to `newCellDataSet`:

```
# Not run
HSMM <- newCellDataSet(count_matrix,
                       phenoData = pd,
                       featureData = fd,
                       expressionFamily=negbinomial())
```

There are several allowed values for `expressionFamily`, which expects a "family function" from the *VGAM* package:

Family function	Data type	Notes
`tobit()` (default)	FPKM, TPM	Tobits are truncated normal distributions. Using `tobit()` will tell Monocle to log-transform your data where appropriate. Do not transform it yourself.
`negbinomial()`	UMIs, Transcript counts from experiments with spike-ins or `relative2abs`, raw read counts	Using `negbinomial()` can be slow for very large datasets. In these cases, consider `negbinomial.size()`.
`negbinomial.size()`	UMIs, Transcript counts from experiments with spike-ins, raw read counts	Slightly less accurate for differential expression than `negbinomial()`, but much, much faster.
`gaussianff()`	log-transformed FPKM/TPMs, Ct values from single-cell qPCR	If you want to use Monocle on data you have already transformed to be normally distributed, you can use this function, though some Monocle features may not work well.

Using the wrong expressionFamily for your data will lead to bad results, errors from Monocle, or both. However, if you have FPKM/TPM data, you can still use negative binomial if you first convert your relative expression values to transcript counts using `relative2abs`. This often leads to much more accurate results than using `tobit()`. See 2.4 for details.

2.3 Working with large data sets

Some single-cell RNA-Seq experiments report measurements from tens of thousands of cells or more. As instrumentation improves and costs drop, experiments will become ever larger and more complex, with many conditions, controls, and replicates. A matrix of expression data with 50,000 cells and a measurement for each of the 25,000+ genes in the human genome can take up a lot of memory. However, because current protocols typically don't capture all or even most of the mRNA molecules in each cell, many of the entries of expression matrices are zero. Using *sparse matrices* can help you work with huge datasets on a typical computer.

To work with your data in a sparse format, simply provide it to Monocle as a sparse matrix from the *Matrix* package:

```
HSMM <- newCellDataSet(as(as.matrix(rpc_matrix), "sparseMatrix"),
                       phenoData = pd,
                       featureData = fd,
                       lowerDetectionLimit=1,
                       expressionFamily=negbinomial.size())
```

Other sparse matrix packages, such as *slam* or *SparseM* are not supported.

2.4 Converting relative expression values into mRNA counts

If you performed your single-cell RNA-Seq experiment using *spike-in* standards, you can convert these measurements into mRNAs per cell (RPC). RPC values are often easier to analyze than FPKM or TPM values, because have better statistical tools to model them. In fact, it's possible to convert FPKM or TPM values to RPC values even if there were no spike-in standards included in the experiment. Monocle 2 includes an algorithm called *Census* which performs this conversion (Qiu *et al*, submitted). You can convert to RPC values before creating your CellDataSet object using the `relative2abs` function, as follows:

```
pd <- new("AnnotatedDataFrame", data = HSMM_sample_sheet)
fd <- new("AnnotatedDataFrame", data = HSMM_gene_annotation)

# First create a CellDataSet from the relative expression levels
HSMM <- newCellDataSet(as.matrix(HSMM_expr_matrix),
                       phenoData = pd,
                       featureData = fd)

# Next, use it to estimate RNA counts
rpc_matrix <- relative2abs(HSMM)
```

```
# Now, make a new CellDataSet using the RNA counts
HSMM <- newCellDataSet(as(as.matrix(rpc_matrix), "sparseMatrix"),
                       phenoData = pd,
                       featureData = fd,
                       lowerDetectionLimit=1,
                       expressionFamily=negbinomial.size())
```

Note that since we are using RPC values, we have changed the value of `lowerDetectionLimit` to reflect the new scale of expression. Importantly, we have also set the `expressionFamily` to `negbinomial()`, which tells Monocle to use the negative binomial distribution in certain downstream statistical tests. Failing to change these two options can create problems later on, so make sure not to forget them when using RPC values.

Finally, we'll also call two functions that pre-calculate some information about the data. Size factors help us normalize for differences in mRNA recovered across cells, and "dispersion" values will help us perform differential expression analysis later.

```
HSMM <- estimateSizeFactors(HSMM)
HSMM <- estimateDispersions(HSMM)

## Removing 100 outliers
```

We're now ready to start using the `HSMM` object in our analysis.

2.5 Filtering low-quality cells

The first step in any single-cell RNA-Seq analysis is identifying poor-quality libraries from further analysis. Most single-cell workflows will include at least some libraries made from dead cells or empty wells in a plate. It's also crucial to remove doublets: libraries that were made from two or more cells accidentally. These cells can disrupt downstream steps like pseudotime ordering or clustering. This section walks through typical quality control steps that should be performed as part of all analyses with Monocle.

It is often convenient to know how many express a particular gene, or how many genes are expressed by a given cell. Monocle provides a simple function to compute those statistics:

```
HSMM <- detectGenes(HSMM, min_expr = 0.1)
print(head(fData(HSMM)))

##                     gene_short_name        biotype num_cells_expressed
## ENSG00000000003.10          TSPAN6 protein_coding                 184
## ENSG00000000005.5             TNMD protein_coding                   0
## ENSG00000000419.8             DPM1 protein_coding                 211
## ENSG00000000457.8            SCYL3 protein_coding                  18
## ENSG00000000460.12        C1orf112 protein_coding                  47
## ENSG00000000938.8              FGR protein_coding                   0
##                     use_for_ordering
## ENSG00000000003.10             FALSE
## ENSG00000000005.5              FALSE
## ENSG00000000419.8              FALSE
## ENSG00000000457.8              FALSE
## ENSG00000000460.12              TRUE
## ENSG00000000938.8              FALSE

expressed_genes <- row.names(subset(fData(HSMM), num_cells_expressed >= 10))
```

The vector `expressed_genes` now holds the identifiers for genes expressed in at least 50 cells of the data set. We will use this list later when we put the cells in order of biological progress. It is also sometimes convenient to exclude genes expressed in few if any cells from the *CellDataSet* object so as not to waste CPU time analyzing them for differential expression.

Let's start trying to remove unwanted, poor quality libraries from the CellDataSet. Your single cell RNA-Seq protocol may have given you the opportunity to image individual cells after capture but prior to lysis. This image data allows you to score your cells, confirming that none of your libraries were made from empty wells or wells with excess cell debris. With some protocols and instruments, you may get more than one cell captured instead just a single cell.

You should exclude libraries that you believe did not come from a single cell, if possible. Empty well or debris well libraries can be especially problematic for Monocle. It's also a good idea to check that each cell's RNA-seq library was sequenced to an acceptible degree. While there is no widely accepted minimum level for what constitutes seequencing "deeply enough", use your judgement: a cell sequenced with only a few thousand reads is unlikely to yield meaningful measurements.

CellDataSet objects provide a convenient place to store per-cell scoring data: the `phenoData` slot. Simply include scoring attributes as columns in the data frome you used to create your *CellDataSet* container. You can then easily filter out cells that don't pass quality control. You might also filter cells based on metrics from high throughput sequencing quality assessment packages such as FastQC. Such tools can often identify RNA-Seq libraries made from heavily degraded RNA, or where the library contains an abnormally large amount of ribosomal, mitochondrial, or other RNA type that you might not be interested in.

The HSMM dataset included with this package has scoring columns built in:

```
print(head(pData(HSMM)))
```

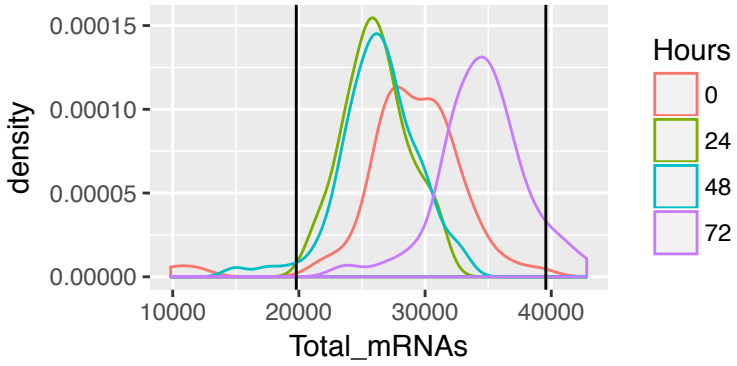
```
##                Library Well Hours Media Mapped.Fragments Pseudotime State
## T0_CT_A01 SCC10013_A01  A01     0    GM          1958074  23.916673     1
## T0_CT_A03 SCC10013_A03  A03     0    GM          1930722   9.022265     1
## T0_CT_A05 SCC10013_A05  A05     0    GM          1452623   7.546608     1
## T0_CT_A06 SCC10013_A06  A06     0    GM          2566325  21.463948     1
## T0_CT_A07 SCC10013_A07  A07     0    GM          2383438  11.299806     1
## T0_CT_A08 SCC10013_A08  A08     0    GM          1472238  67.436042     2
##           Size_Factor num_genes_expressed
## T0_CT_A01    1.392811                6850
## T0_CT_A03    1.311607                6947
## T0_CT_A05    1.218922                7019
## T0_CT_A06    1.013981                5560
## T0_CT_A07    1.085580                5998
## T0_CT_A08    1.099878                6055
```

This dataset has already been filtered using the following commands:

```
valid_cells <- row.names(subset(pData(HSMM), Cells.in.Well == 1 & Control == FALSE & Clump == FALSE & De
HSMM <- HSMM[,valid_cells]
```

If you are using RPC values to measure expresion, as we are in this vignette, it's also good to look at the distribution of mRNA totals across the cells:

```
pData(HSMM)$Total_mRNAs <- Matrix::colSums(exprs(HSMM))


HSMM <- HSMM[,pData(HSMM)$Total_mRNAs < 1e6]
upper_bound <- 10^(mean(log10(pData(HSMM)$Total_mRNAs)) + 2*sd(log10(pData(HSMM)$Total_mRNAs)))
lower_bound <- 10^(mean(log10(pData(HSMM)$Total_mRNAs)) - 2*sd(log10(pData(HSMM)$Total_mRNAs)))
qplot(Total_mRNAs, data=pData(HSMM), color=Hours, geom="density") +
  geom_vline(xintercept=lower_bound) +
  geom_vline(xintercept=upper_bound)
```

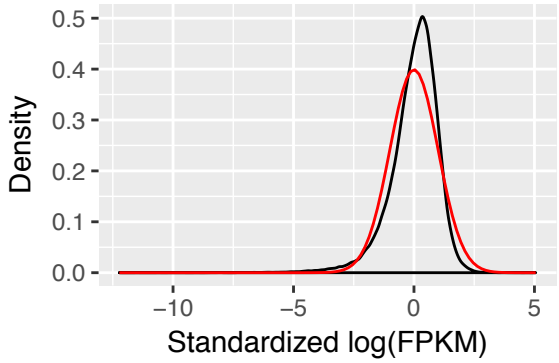


```
HSMM <- HSMM[,pData(HSMM)$Total_mRNAs > lower_bound &
             pData(HSMM)$Total_mRNAs < upper_bound]
HSMM <- detectGenes(HSMM, min_expr = 0.1)
```

We've gone ahead and removed the few cells with either very low mRNA recovery or far more mRNA that the typical cell. Often, doublets or triplets have roughly twice the mRNA recovered as true single cells, so the latter filter is another means of excluding all but single cells from the analysis. Such filtering is handy if your protocol doesn't allow directly visualization of cell after they've been captured. Note that these thresholds of 10,000 and 40,000 mRNAs are specific to this dataset. You may need to adjust filter thresholds based on your experimental protocol.

Once you've excluded cells that do not pass your quality control filters, you should verify that the expression values stored in your *CellDataSet* follow a distribution that is roughly lognormal:

```
# Log-transform each value in the expression matrix.
L <- log(exprs(HSMM[expressed_genes,]))

# Standardize each gene, so that they are all on the same scale,
# Then melt the data with plyr so we can plot it easily"
melted_dens_df <- melt(Matrix::t(scale(Matrix::t(L))))

# Plot the distribution of the standardized gene expression values.
qplot(value, geom="density", data=melted_dens_df) +  stat_function(fun = dnorm, size=0.5, color='red') +
  xlab("Standardized log(FPKM)") +
  ylab("Density")
```



3 Classifying and counting cells of different types

Single cell experiments are often performed on complex mixtures of multiple cell types. Dissociated tissue samples might contain two, three, or even many different cells types. In such cases, it's often nice to classify cells based on type using known markers. In the myoblast experiment, the culture contains fibroblasts that came from the original muscle biopsy used to establish the primary cell culture. Myoblasts express some key genes that fibroblasts don't. Selecting only the genes that express, for example, sufficiently high levels of *MYF5* excludes the fibroblasts. Likewise, fibroblasts express high levels of *ANPEP* (CD13), while myoblasts tend to express few if any transcripts of this gene.

3.1 Classifying cells with CellTypeHierarchy

Monocle provides a simple system for tagging cells based on the expression of marker genes of your choosing. You simply provide a set of functions that Monocle can use to annotate each cell. For example, you could provide a function for each of several cell types. These functions accept as input the expression data for each cell, and return TRUE to tell Monocle that a cell meets the criteria defined by the function. So you could have one function that returns TRUE for cells that express myoblast-specific genes, another function for fibroblast-specific genes, etc. Here's an example of such a set of "gating" functions:

```
MYF5_id <- row.names(subset(fData(HSMM), gene_short_name == "MYF5"))
ANPEP_id <- row.names(subset(fData(HSMM), gene_short_name == "ANPEP"))
```

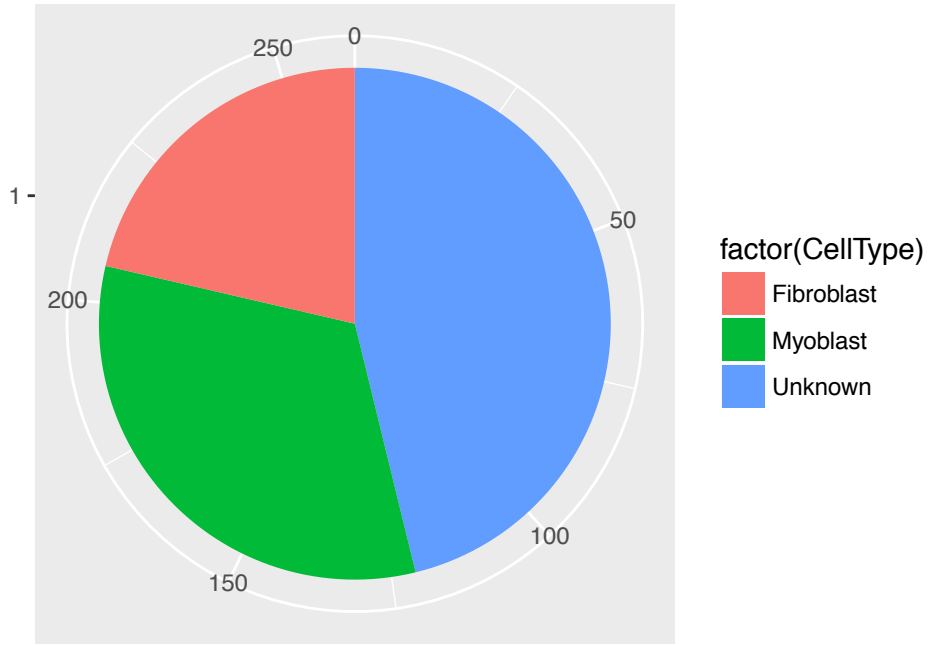
```
cth <- newCellTypeHierarchy()
cth <- addCellType(cth, "Myoblast", classify_func=function(x) {x[MYF5_id,] >= 1})
cth <- addCellType(cth, "Fibroblast", classify_func=function(x)
       {x[MYF5_id,] < 1 & x[ANPEP_id,] > 1})
```

The functions are organized into a small data structure called a `CellTypeHierarchy`, that Monocle uses to classify the cells. You first initialize a new CellTypeHierarchy object, then register your gating functions within it. Once the data strucure is set up, you can use it to classify all the cells in the experiment:

```
HSMM <- classifyCells(HSMM, cth, 0.1)
```

The function `classifyCells` applies each gating function to each cell, classifies the cells according to the gating functions, and returns the `CellDataSet` with a new column, `CellType` in its `pData` table. We can now count how many cells of each type there are in the experiment.

```
table(pData(HSMM)$CellType)

##
## Fibroblast   Myoblast    Unknown
##         56         85        121

pie <- ggplot(pData(HSMM), aes(x = factor(1), fill = factor(CellType))) +
  geom_bar(width = 1)
pie + coord_polar(theta = "y") +
  theme(axis.title.x=element_blank(), axis.title.y=element_blank())
```



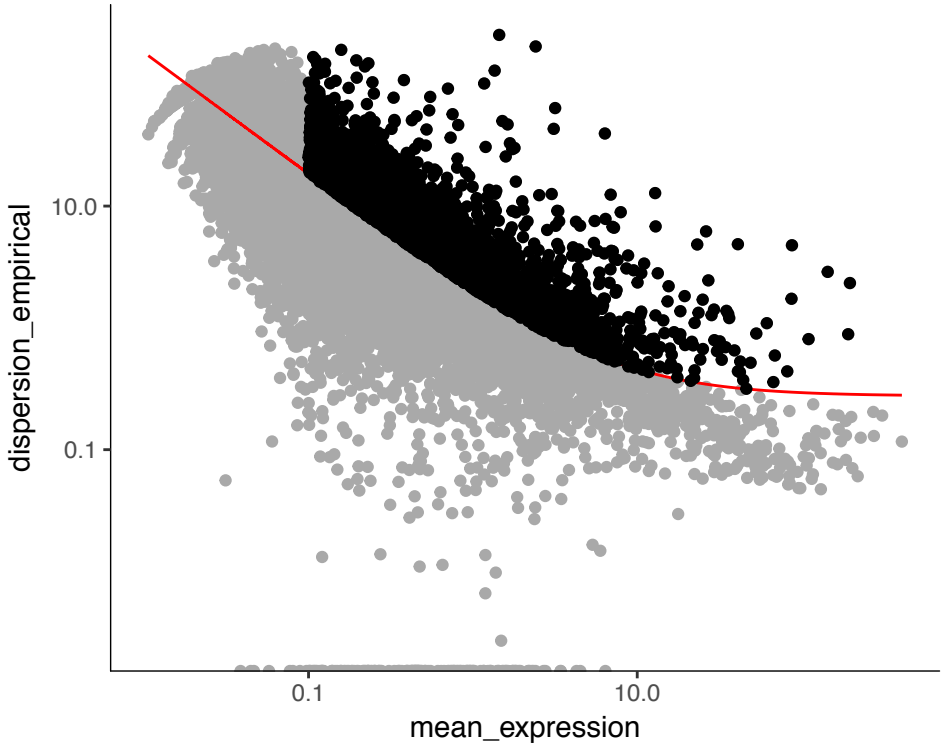
Note that many cells are marked "Unknown". This is common, largely because of the low rate of mRNA capture in most single-cell RNA-Seq experiments. A cell might express a few *MYF5* mRNAs, but we weren't lucky enough to capture one of them. When a cell doesn't meet any of the criteria specified in your classification functions, it's marked "Unknown". If it meets multiple functions' criteria, it's marked "Ambiguous". You could just exclude such cells, but you'd be throwing out a lot of your data. In this case, we'd lose more than half of the cells!

3.2 Unsupervised cell clustering

Fortunately, Monocle provides an algorithm you can use to impute the types of the "Unknown" cells. This algorithms, implemented in the function `clusterCells`, groups cells together according to global expression profile. That way, if your cell expresses lots of genes specific to myoblasts, but just happens to lack *MYF5*, we can still recognize it as a myoblast. `clusterCells` can be used in an unsupervised manner, as well as in a "semi-supervised" mode, which allows to assist the algorithm with some expert knowledge. Let's look at the unsupervised mode first.

The first step is to decide which genes to use in clustering the cells. We could use all genes, but we'd be including a lot of genes that are not expressed at a high enough level to provide meaningful signal. Including them would just add noise to the system. We can filter genes based on average expression level, and we can additionally select genes that are unusually variable across cells. These genes tend to be highly informative about cell state.

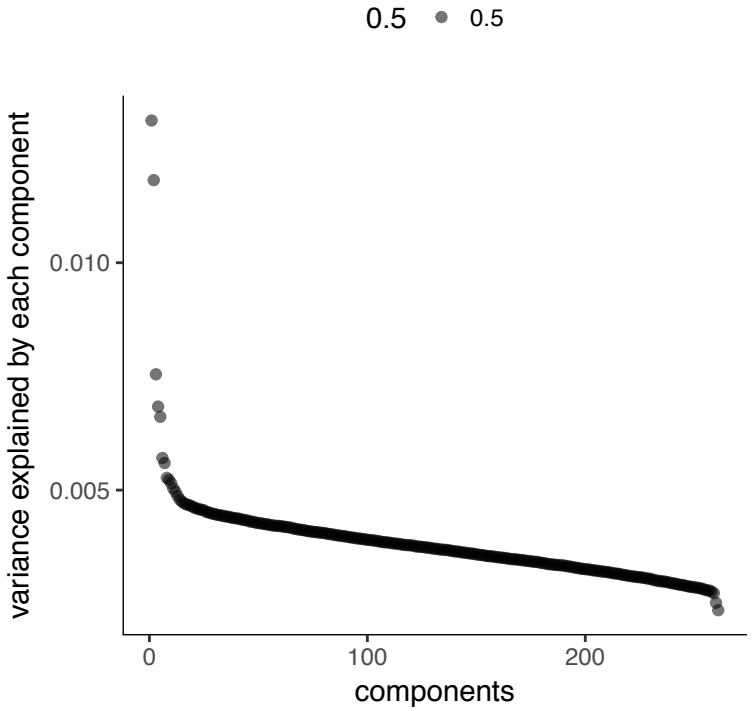
```
disp_table <- dispersionTable(HSMM)
unsup_clustering_genes <- subset(disp_table, mean_expression >= 0.1
                                 & dispersion_empirical >= 1 * dispersion_fit)
HSMM <- setOrderingFilter(HSMM, unsup_clustering_genes$gene_id)
plot_ordering_genes(HSMM)
```



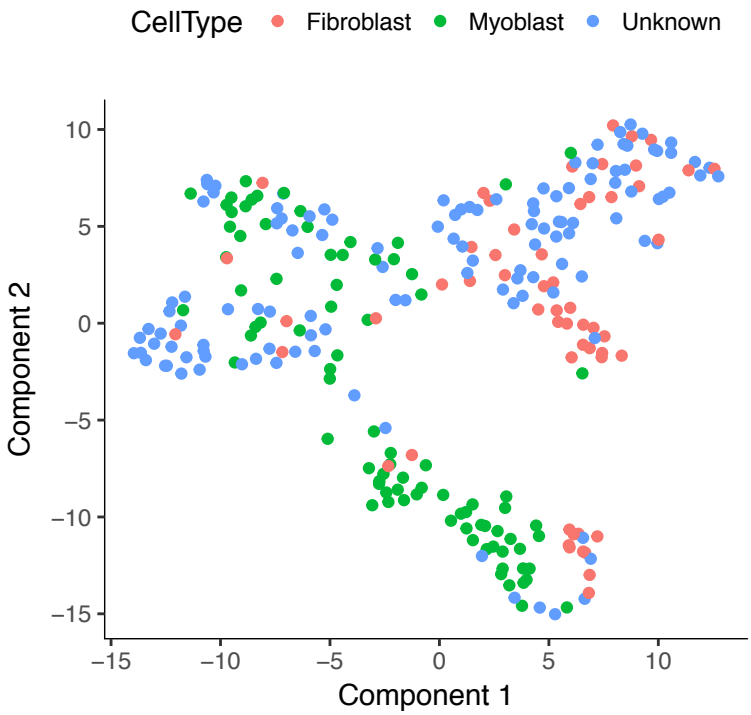
The `setOrderingFilter` function marks genes that will be used for clustering in subsequent calls to `clusterCells`, although we will be able to provide other lists of genes if we want. `plot_ordering_genes` shows how variability (dispersion) in a gene's expression depends on the average expression across cells. The red line shows Monocle's expectation of the dispersion based on this relationship. The genes we marked for use in clustering are shown as black dots, while the others are shown as grey dots.

Now we're ready to try clustering the cells:

```
# HSMM@auxClusteringData[["tSNE"]]$variance_explained <- NULL
plot_pc_variance_explained(HSMM, return_all = F) # norm_method = 'log',
```



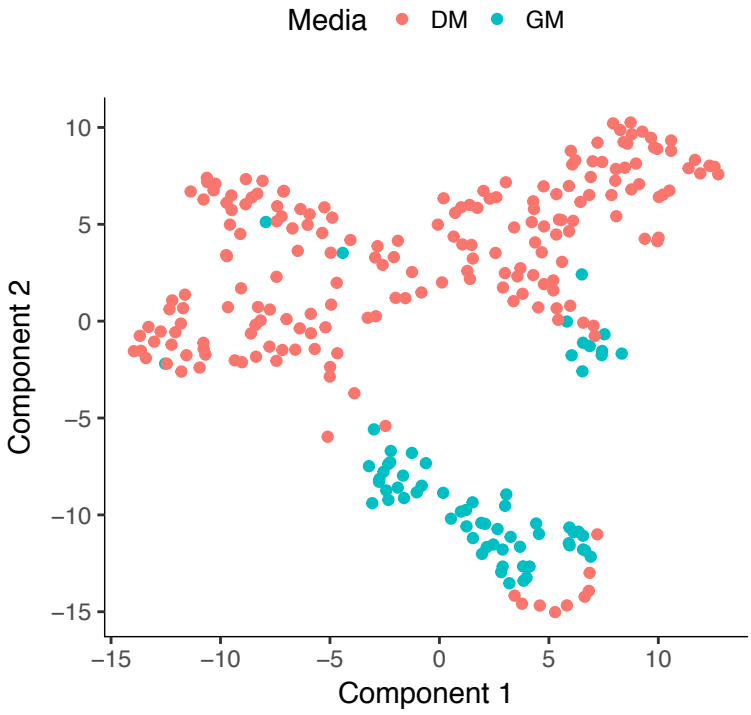

```
HSMM <- reduceDimension(HSMM, max_components=2, num_dim = 5,
                        reduction_method = 'tSNE', verbose = T)

## Remove noise by PCA ...
## Reduce dimension by tSNE ...

HSMM <- clusterCells(HSMM,
 number_clusters=2,
 clustering_genes=unsup_clustering_genes$gene_id)

## Distance cutoff calculated to 0.9256021

## the length of the distance:  34191

plot_cell_clusters(HSMM, 1, 2, color="CellType")
```



As you can see, the cells fall into two different clusters. The cells tagged as myoblasts by our gating functions are marked in green, while the fibroblasts are tagged in red. The cells that don't express either marker are blue. Unfortunately, the cells don't cluster by type. This isn't all that surprising, because myoblasts and contaminating interstitial fibroblasts express many of the same genes in these culture conditions.

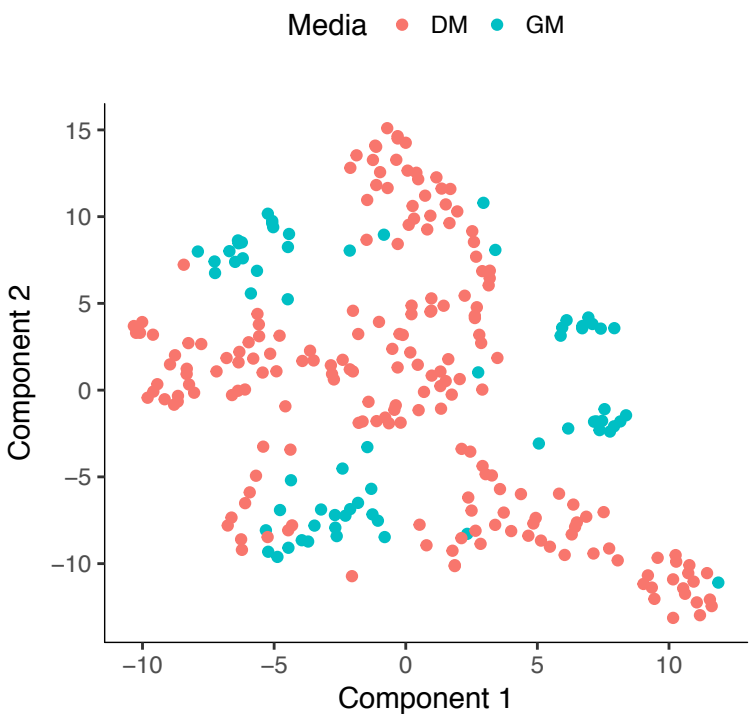
Moroever, there are other sources of variation in the experiment that might be driving the clustering. One source of variation in the experiment stems from the experimental design. To initiate myoblast differentiation, we switch media from a high-mitogen growth medium (GM) to a low-mitogen differentiation medium (DM). Perhaps the cells are clustering based on the media they're cultured in?

```
plot_cell_clusters(HSMM, 1, 2, color="Media")
```



Fortunately, Monocle allows us to subtract the effects of "uninteresting" sources of variation so they don't impact the clustering. You can do this with the `residualModelFormulaStr` argument to `clusterCells` and several other Monocle functions. This argument accepts an R model formula string specifying the effects you want to subtract prior to clustering.

```
HSMM <- reduceDimension(HSMM, max_components=2, num_dim = 5, reduction_method = 'tSNE',
                        residualModelFormulaStr="~Media", verbose = T)

## Removing batch effects
## Remove noise by PCA ...
## Reduce dimension by tSNE ...

HSMM <- clusterCells(HSMM,
                                  number_clusters=2,
                                  clustering_genes=unsup_clustering_genes$gene_id)

## Distance cutoff calculated to 1.00222

## the length of the distance:  34191

# HSMM <- clusterCells(HSMM, residualModelFormulaStr="~Media", num_clusters=2)
#
plot_cell_clusters(HSMM, 1, 2, color="Media")
```



Now the culture media appears to play less of a role in the clustering. Another common *technical* source of variation is the efficiency of the library construction reaction (particularly the cDNA synthesis steps). When library prep goes particularly well for a cell, we'll recover more mRNA and detect more genes from it. We can control for this effect and the media effect at the same time by expanding the residual model:

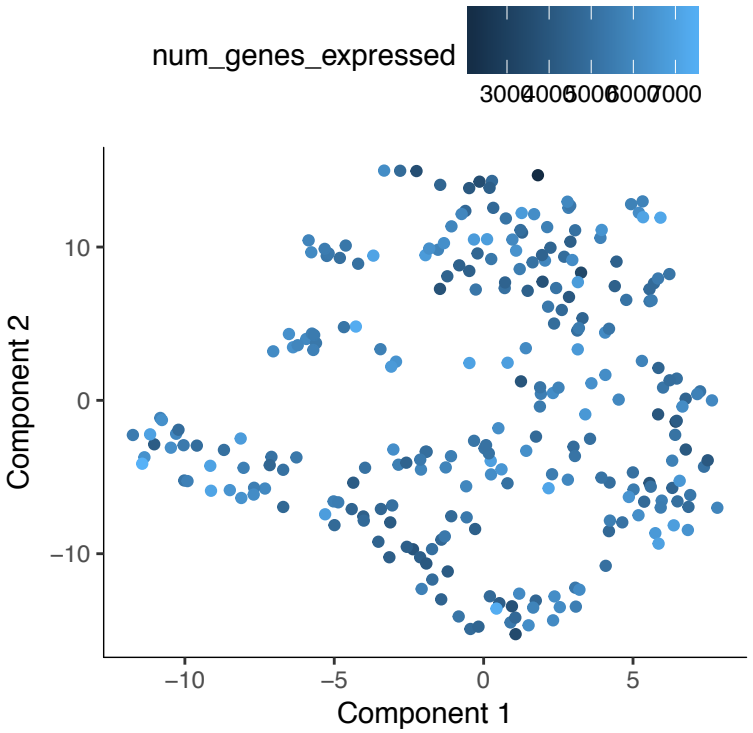
```
HSMM <- reduceDimension(HSMM, max_components=2, num_dim = 5, reduction_method = 'tSNE',
                        residualModelFormulaStr="~Media + num_genes_expressed", verbose = T)

## Removing batch effects
## Remove noise by PCA ...
## Reduce dimension by tSNE ...

HSMM <- clusterCells(HSMM,
```

```
                                  number_clusters=2,
                                  clustering_genes=unsup_clustering_genes$gene_id)
```

```
## Distance cutoff calculated to 0.9853297
```

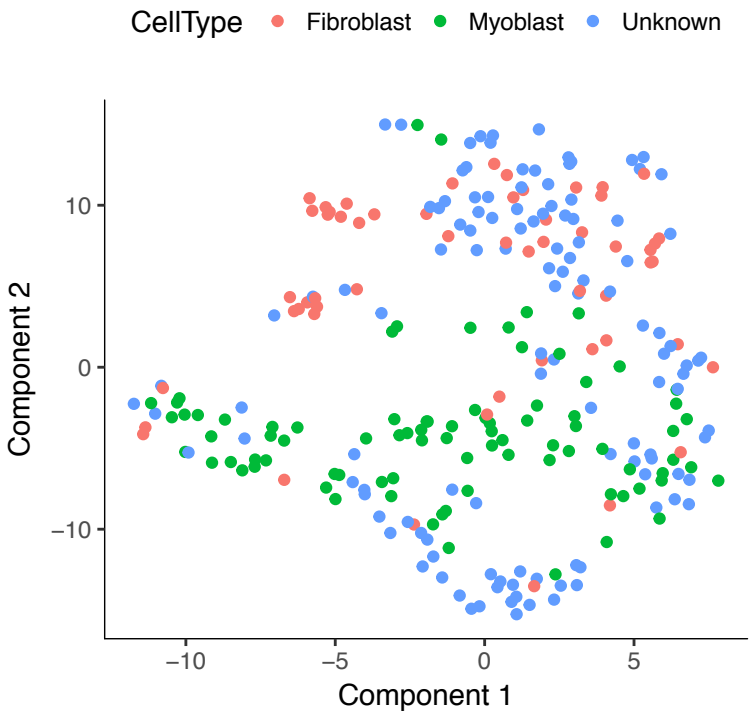
```
## the length of the distance:  34191
```

```
plot_cell_clusters(HSMM, 1, 2, color="num_genes_expressed")
```



Now that we've accounted for some unwanted sources of variation, we're ready to take another crack at classifying the cells by unsupervised clustering:

```
plot_cell_clusters(HSMM, 1, 2, color="CellType")
```



Now, most of the myoblasts are in one cluster, most of the fibroblasts are in the other, and the unknowns are spread across both. However, we still see some cells of both types in each cluster. This could be due to lack of specificity in our marker genes and our CellTypeHierarchy functions, but it could also be due to suboptimal clustering. To help rule out the latter, let's try running `clusterCells` in its semi-supervised mode.

3.3 Semi-supervised cell clustering with known marker genes

First, we'll select a different set of genes to use for clustering the cells. Before we just picked genes that were highly expressed and highly variable. Now, we'll pick genes that *co-vary* with our markers. In a sense, we'll be building a large list of genes to use as markers, so that even if a cell doesn't have *MYF5*, it might be recognizable as a myoblast based on other genes.

```
marker_diff <- markerDiffTable(HSMM[expressed_genes,],
                               cth,
                               residualModelFormulaStr="~Media",
                               cores=1)
```

The function `markerDiffTable` takes a CellDataSet and a CellTypeHierarchy and classifies all the cells into types according to your provided functions. It then removes all the "Unknown" and "Ambiguous" functions before identifying genes that are differentially expressed between the types. Again, you can provide a residual model of effects to exclude from this test. The function then returns a data frame of test results, and you can use this to pick the genes you want to use for clustering. Often it's best to pick the top 10 or 20 genes that are most specific for each cell type. This ensures that the clustering genes aren't dominated by markers for one cell type. You generally want a balanced panel of markers for each type if possible. Monocle provides a handy function for ranking genes by how restricted their expression is for each type.

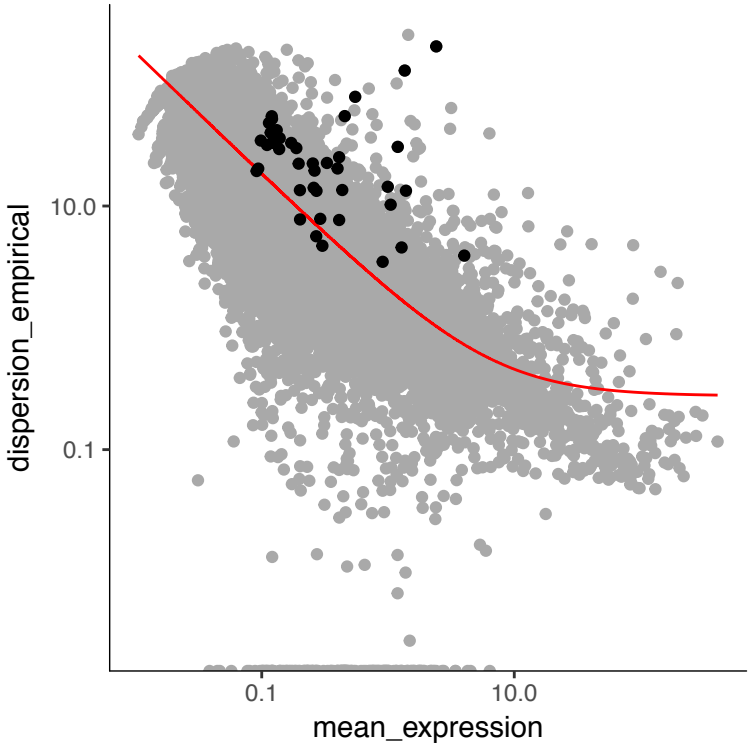
```
candidate_clustering_genes <- row.names(subset(marker_diff, qval < 0.05))
marker_spec <- calculateMarkerSpecificity(HSMM[candidate_clustering_genes,], cth)
head(selectTopMarkers(marker_spec, 3))
```

```
##              gene_id   CellType specificity
## 1  ENSG00000124875.5 Fibroblast   1.0000000
## 2 ENSG00000128340.10 Fibroblast   0.9999611
## 3  ENSG00000259121.2 Fibroblast   1.0000000
## 4  ENSG00000156298.8   Myoblast   0.9884418
## 5  ENSG00000233494.1   Myoblast   1.0000000
## 6  ENSG00000270123.1   Myoblast   1.0000000
```

The last line above shows the top three marker genes for myoblasts and fibroblasts. The "specificity" score is calculated using the metric described in Cabili et al [**?**] and can range from zero to one. The closer it is to one, the more restricted it is to the cell type in question. You can use this feature to define new markers for known cell types, or pick out genes you can use in purifying newly discovered cell types. This can be highly valuable for downstream follow up experiments.

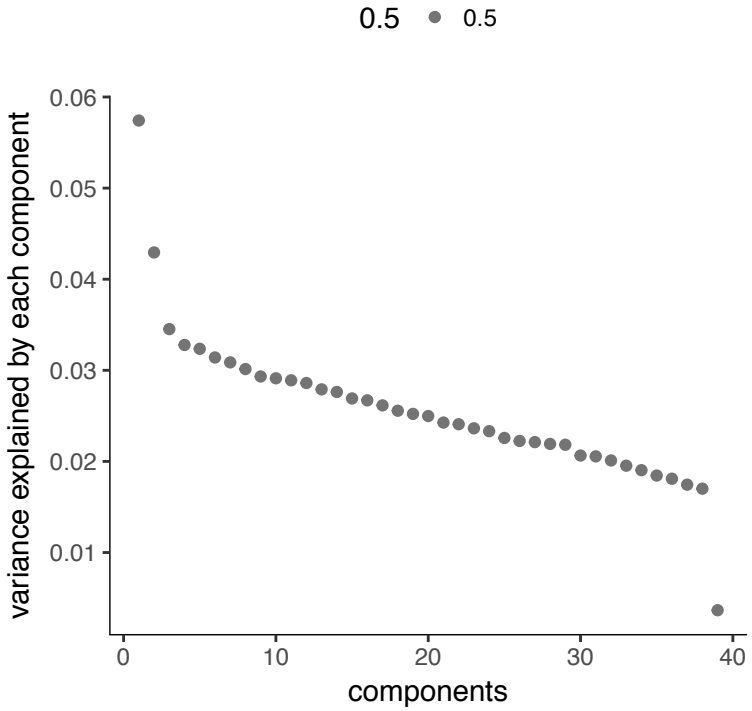
To cluster the cells, we'll choose the top ten markers for each of these cell types:

```
semisup_clustering_genes <- unique(selectTopMarkers(marker_spec, 20)$gene_id)
HSMM <- setOrderingFilter(HSMM, semisup_clustering_genes)
plot_ordering_genes(HSMM)
```

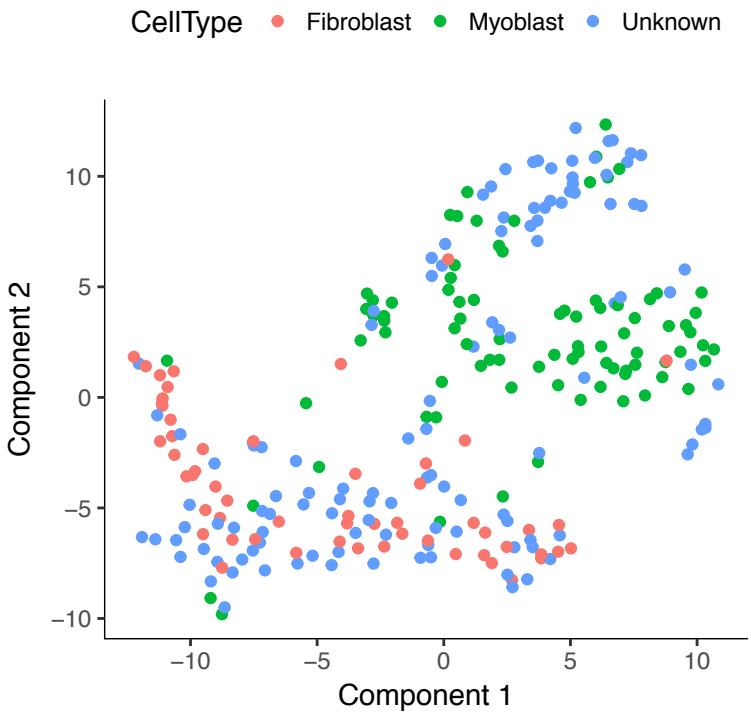


Note that we've got a smaller set of genes, and some of them are not especially highly expressed or variable across the experiment. However, they are great for distinguishing cells that express *MYF5* from those that have *ANPEP*. We've already marked them for use in clustering, but even if we hadn't, we could still use them by providing them directly to `clusterCells`.

```
HSMM@auxClusteringData[["tSNE"]]$variance_explained <- NULL
plot_pc_variance_explained(HSMM, return_all = F) # norm_method = 'log',
```



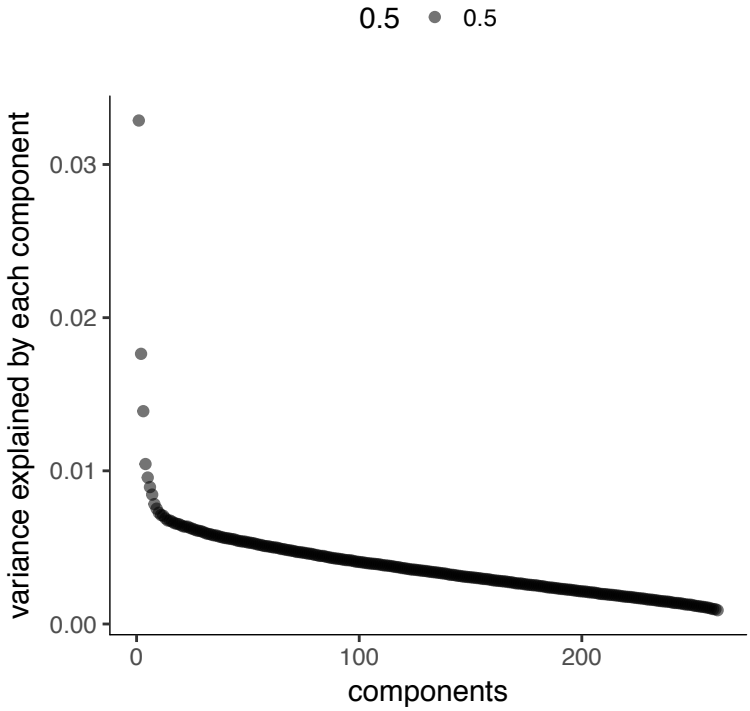

```
HSMM <- reduceDimension(HSMM, max_components=2, num_dim = 4, reduction_method = 'tSNE',
                        residualModelFormulaStr="~Media + num_genes_expressed", verbose = T)

## Removing batch effects
## Remove noise by PCA ...
## Reduce dimension by tSNE ...

HSMM <- clusterCells(HSMM,
 number_clusters=2,
 clustering_genes=semisup_clustering_genes) #

## Distance cutoff calculated to 0.8972102

## the length of the distance:  34191

plot_cell_clusters(HSMM, 1, 2, color="CellType")
```



3.4 Imputing cell type

Note that we've reduce the number of "contaminating" fibroblasts in the myoblast cluster, and vice versa. But what about the "Unknown" cells? If you provide `clusterCells` with a the CellTypeHierarcy, Monocle will use it classify *whole clusters*, rather than just individual cells. Essentially, cluserCells works exactly as before, except after the clusters are built, it counts the frequency of each cell type in each cluster. When a cluster is composed of more than a certain percentage (in this case, 10%) of a certain type, all the cells in the cluster are set to that type. If a cluster is composed of more than one cell type, the whole thing is marked "Ambiguous". If there's no cell type thats above the threshold, the cluster is marked "Unknown". Thus, Monocle helps you impute the type of each cell even in the presence of missing marker data.

```
HSMM <- setOrderingFilter(HSMM, row.names(subset(marker_diff, qval < 0.05)))

HSMM@auxClusteringData[["tSNE"]]$variance_explained <- NULL
plot_pc_variance_explained(HSMM, return_all = F) # norm_method = 'log',
```




```
HSMM <- reduceDimension(HSMM, max_components=2, num_dim = 4, reduction_method = 'tSNE', residualModelFor
```

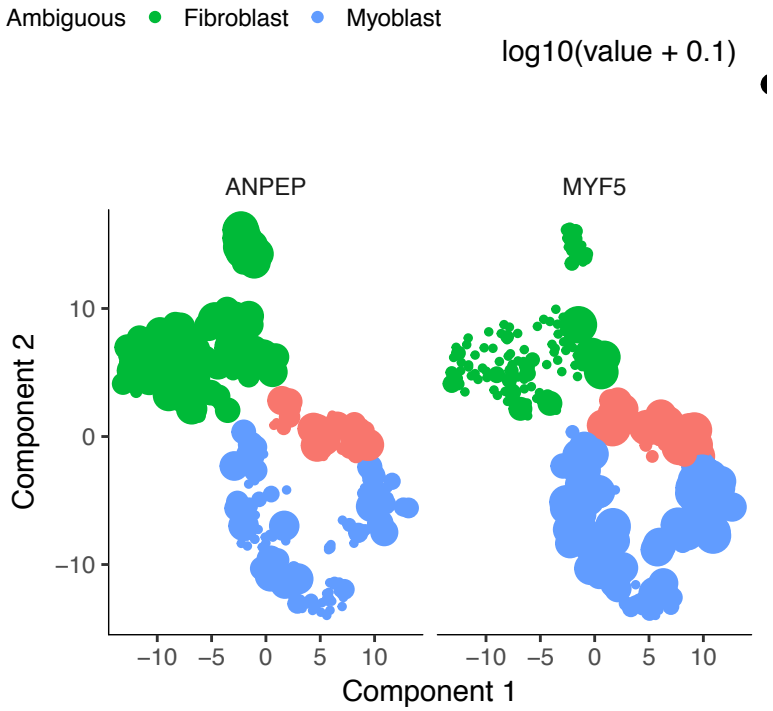
```
## Removing batch effects
## Remove noise by PCA ...
## Reduce dimension by tSNE ...
```

```
HSMM <- clusterCells(HSMM,
                     num_clusters=2,
                     frequency_thresh=0.1,
                     cell_type_hierarchy=cth,
                     clustering_genes=row.names(subset(marker_diff, qval < 0.05)),
                     residualModelFormulaStr="~Media + num_genes_expressed")
```

```
## Distance cutoff calculated to 1.01013
```

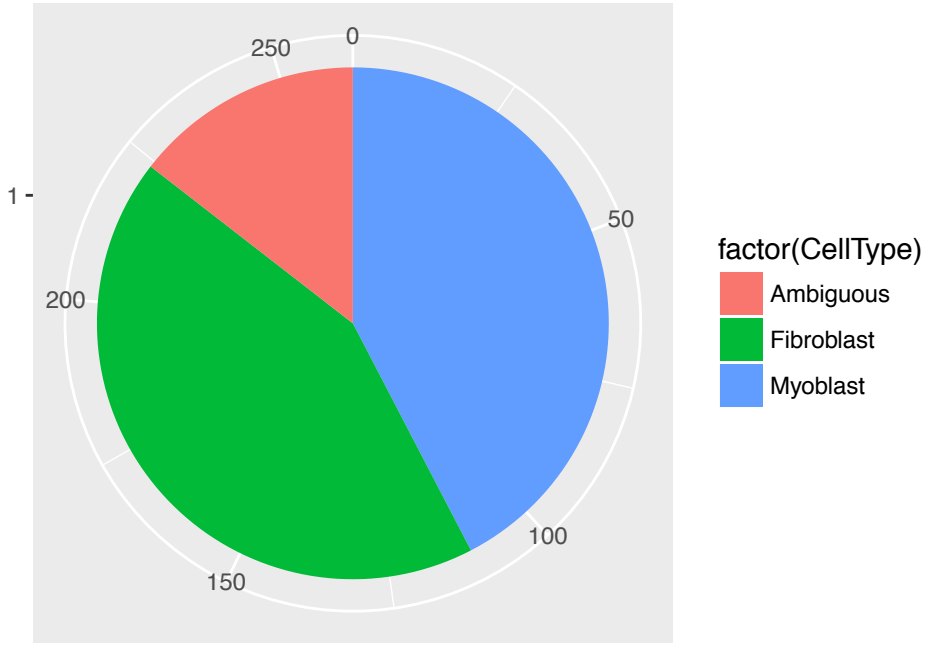
```
## the length of the distance:  34191
```

```
plot_cell_clusters(HSMM, 1, 2, color="CellType", markers = c("MYF5", "ANPEP"))
```



As you can see, the clusters are highly pure in terms of *MYF5* expression. There are some cells expressing *ANPEP* in both clusters, but those in the myoblast cluster also express *MYF5*. This is not surprising, as ANPEP isn't a very specific marker of fibroblasts. Overall, we've successfully classified all the cells:

```
pie <- ggplot(pData(HSMM), aes(x = factor(1), fill = factor(CellType))) +
  geom_bar(width = 1)
pie + coord_polar(theta = "y") +
  theme(axis.title.x=element_blank(), axis.title.y=element_blank())
```



Finally, we subset the CellDataSet object to create $HSMM_myo$, which includes only myoblasts.

```
HSMM_myo <- HSMM[,pData(HSMM)$CellType == "Myoblast"]
HSMM_myo <- estimateDispersions(HSMM_myo)

## Removing 102 outliers
```

4 Constructing single cell trajectories

During development, in response to stimuli, and througout life, cells transition from one functional "state" to another. Cells in different states express different sets of genes, producing a dynamic repetoire of proteins and metabolites that carry out their work. As cells move between states, undergo a process of transcriptional re-configuration, with some genes being silenced and others newly activated. These transient states are often hard to characterize because purifying cells in between more stable endpoint states can be difficult or impossible. Single-cell RNA-Seq can enable you to see these states without the need for purification. However, to do so, we must determine where each cell is the range of possible states.

Monocle introduced the strategy of using RNA-Seq for *single cell trajectory analysis*. Rather than purifying cells into discrete states experimentally, Monocle uses an algorithm to learn the sequence of gene expression changes each cell must go through as part of a dynamic biological process. Once it has learned the overall "trajectory" of gene expression changes, Monocle can place each cell at its proper position in the trajectory. You can then use Monocle's differential analysis toolkit to find genes regulated over the course of the trajectory, as described in section 5.3. If there are multiple outcome for the process, Monocle will reconstruct a "branched" trajectory. These branches correspond to cellular "decisions", and Monocle provides powerful tools for identifying the genes affected by them and involved in making them. You can see how to analyze branches in section 8.9. Monocle relies on a machine learning technique called *manifold learning* to construct single-cell trajectories. You can read more about the theoretical foundations of Monocle's approach in section 8, or consult the references shown below in section **??**.

4.1 "Pseudotime": a measure of progress through a biological process

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 we term "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. For further details, see section 8.

4.2 The ordering algorithm

4.2.1 Choosing genes for ordering

Inferring a single-cell trajectory is a hard machine learning problem. The first step is to select the genes Monocle will use as input for its machine learning approach. This is called *feature selection*, and it has a major impact in the shape of the trajectory. In single-cell RNA-Seq, genes expressed at low levels are often very noisy, but some may contain important information regarding the state of the cell. If we simply provide all the input data, Monocle might get confused by the noise, or fix on a feature of the data that isn't biologically meaningful, such as batch effects arising from collecting data on different days. Monocle provides you with a variety of tools to select genes that will yield a robust, accurate, and biologically meaningful trajectory. You can use these tools to either perform a completely "unsupervised" analysis, in which Monocle has no forehand knowledge of which gene you consider important. Alternatively, you can make use of expert knowledge in the form of genes that are already known to define biolgical progress to shape Monocle's trajectory. We consider this mode "semi-supervised", because Monocle will augment the markers you provide with other, related genes. Monocle then uses these genes to produce trajectories consistent with known biology but that often reveal new regulatory structure. We return to the muscle data to illustrate both of these modes.

4.2.2 Reducing the dimensionality of the data

Once we have selected the genes we will use to order the cells, Monocle applies a *dimensionality reduction* to the data, which will drastically improves the quality of the trajectory. Monocle reduces the dimensionality of the data with the `reduceDimension` function. This function has a number of options, so you should familiarize yourself with them by consulting the its manual page. You can choose from two algorithms in `reduceDimension`. The first, termed Independent Component Analysis, is a classic linear technique for decomposing data that powered the original version of Monocle. The second, called DDRTree, is a much more powerful nonlinear technique that is the default for Monocle 2. For more on how these both work, see section 8.

4.2.3 Ordering the cells in pseudotime

With the expression data projected into a lower dimensional space, Monocle is ready to learn the trajectory that describes how cells transition from one state into another. Monocle assumes that the trajectory has a tree structure, with one end of it the "root", and the others the "leaves". A cell at the beginning of the biological process starts at the root and progresses along the trunk until it reaches the first branch, if there is one. It then chooses a branch, and moves further and further along the tree until it reaches a leaf. These mathematical assumptions translate into some important biological ones. First, that the data includes all the major stages of the biological process. If your experiment failed to capture any cells at a key developmental transition, Monocle won't know its there. Second, that gene expression changes are smooth as a cell moves from one stage to the next. This assumption is realistic: major discontinuities in the trajectory would amount to a cell almost instantaneously turning over its transcriptome, which probably doesn't happen in most biolgical processes.

To order your cells, Monocle uses the `orderCells` function. This routine has one important argument, which allows you to set the root of the tree, and thus the beginning of the process. See the manual page for `orderCells` for more details.

4.3 Unsupervised ordering

In this section, we discuss ordering cells in a completely unsupervised fashion. First, we must decide which genes we will use to define a cell's progress through myogenesis. Monocle orders cells by examining the pattern of expression of these genes across the cell population. Monocle looks for genes that vary in "interesting" (i.e. not just noisy) ways, and uses these to structure the data. We ultimately want a set of genes that increase (or decrease) in expression as a function of progress through the process we're studying.

Ideally, we'd like to use as little prior knowledge of the biology of the system under study as possible. We'd like to discover the important ordering genes from the data, rather than relying on literature and textbooks, because that might introduce bias in the ordering. One effective way to isolate a set of ordering genes is to simply compare the cells collected at the beginning of the process to those at the end and find the differentially expressed genes, as described above. The command below will find all genes that are differentially expressed in response to the switch from growth medium to differentiation medium:

```
#not run
diff_test_res <- differentialGeneTest(HSMM_myo[expressed_genes,],
                                      fullModelFormulaStr="~Media")
ordering_genes <- row.names (subset(diff_test_res, qval < 0.01))
```

Choosing genes based on differential analysis of time points is often highly effective, but what if we don't have time series data? If the cells are asynchronously moving through our biological process (as is usually the case), Monocle can often reconstruct their trajectory from a single population captured all at the same time. Below are two methods to select genes that require no knowledge of the design of the experiment at all.

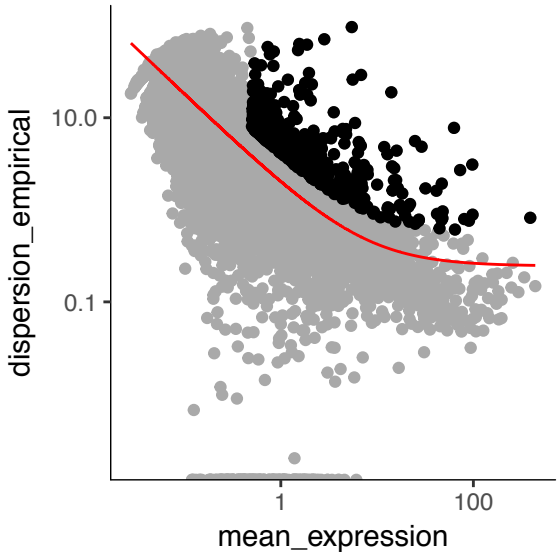
4.3.1 Selecting genes with high dispersion across cells

Genes that vary a lot are often highly informative for identifying cell subpopulations or ordering cells along a trajectory. In RNA-Seq, a gene's variance typically depends on its mean, so we have to be a bit careful about how we select genes based on their variance.

```
disp_table <- dispersionTable(HSMM_myo)
ordering_genes <- subset(disp_table,
                         mean_expression >= 0.5 &
                         dispersion_empirical >= 2 * dispersion_fit)$gene_id
```

Once we have a list of gene ids to be used for ordering, we need to set them in the `HSMM` object, because the next several functions will depend on them.

```
HSMM_myo <- setOrderingFilter(HSMM_myo, ordering_genes)
plot_ordering_genes(HSMM_myo)
```



The genes we've chosen to use for ordering define the *state space* of the cells in our data set. Each cell is a point in this space, which has dimensionality equal to the number of genes we've chosen. So if there are 500 genes used for

ordering, each cell is a point in a 500-dimensional space. For a number of reasons, Monocle works better if we can *reduce* the dimensionality of that space before we try to put the cells in order. In this case, we will reduce the space down to one with two dimensions, which we will be able to easily visualize and interpret while Monocle is ordering the cells.

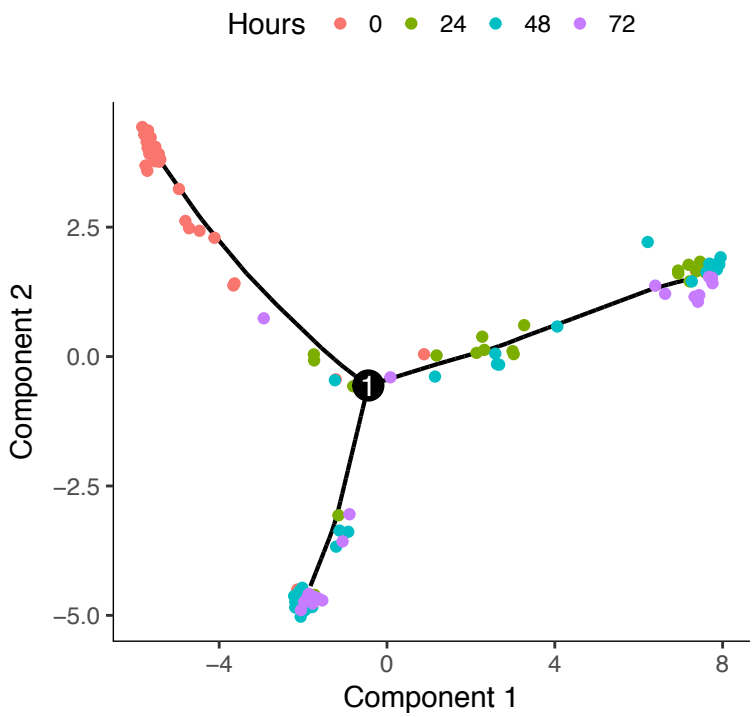
```
HSMM_myo <- reduceDimension(HSMM_myo, max_components=2)
```

Now that the space is reduced, it's time to order the cells using the `orderCells` function as shown below. The second argument is the `reverse` flag. Monocle won't be able to tell without some help which cells are at the beginning of the process and which are at the end. The `reverse` flag tells Monocle to reverse the orientation of the entire process as it's being discovered from the data, so that the cells that would have been assigned to the end are instead assigned to the beginning, and so on. Setting `reverse` to true will reverse the ordering of the cells in pseudotime, which also has a significant impact on the orientation of branches in the trajectory associated with cell fate decisions.

```
HSMM_myo <- orderCells(HSMM_myo, reverse=FALSE)
```

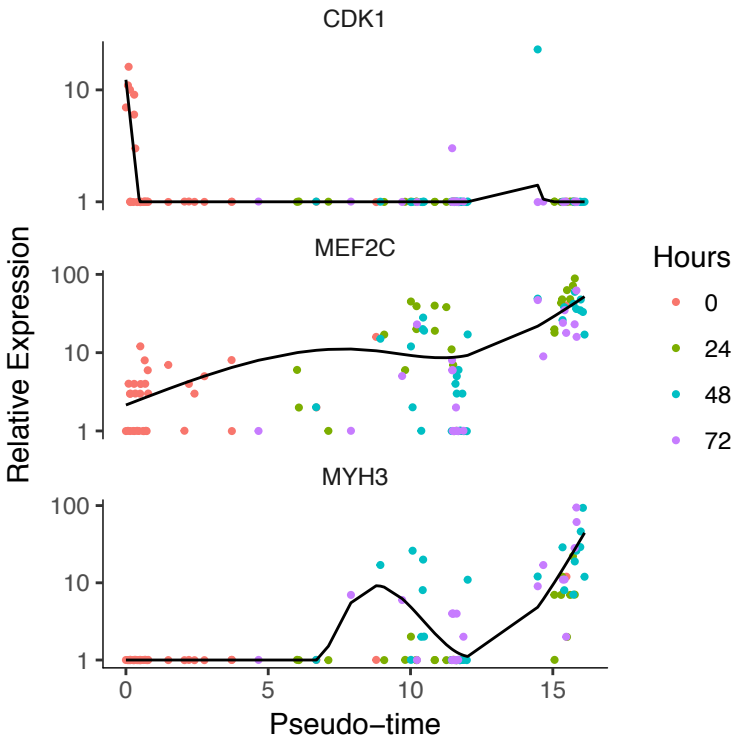
Once the cells are ordered, we can visualize the trajectory in the reducted dimesional space.

```
plot_cell_trajectory(HSMM_myo, color_by="Hours")
```



To confirm that the ordering is correct, and to verify that we don't need to flip around the ordering using the `reverse` flag in `orderCells`, we can select a couple of markers of myogenic progress. Plotting these genes demonstrates that ordering looks good:

```
HSMM_expressed_genes <-  row.names(subset(fData(HSMM_myo), num_cells_expressed >= 10))
HSMM_filtered <- HSMM_myo[HSMM_expressed_genes,]

my_genes <- row.names(subset(fData(HSMM_filtered),
                             gene_short_name %in% c("CDK1", "MEF2C", "MYH3")))

cds_subset <- HSMM_filtered[my_genes,]
plot_genes_in_pseudotime(cds_subset, color_by="Hours")
```



4.3.2 Selecting genes based on PCA loading

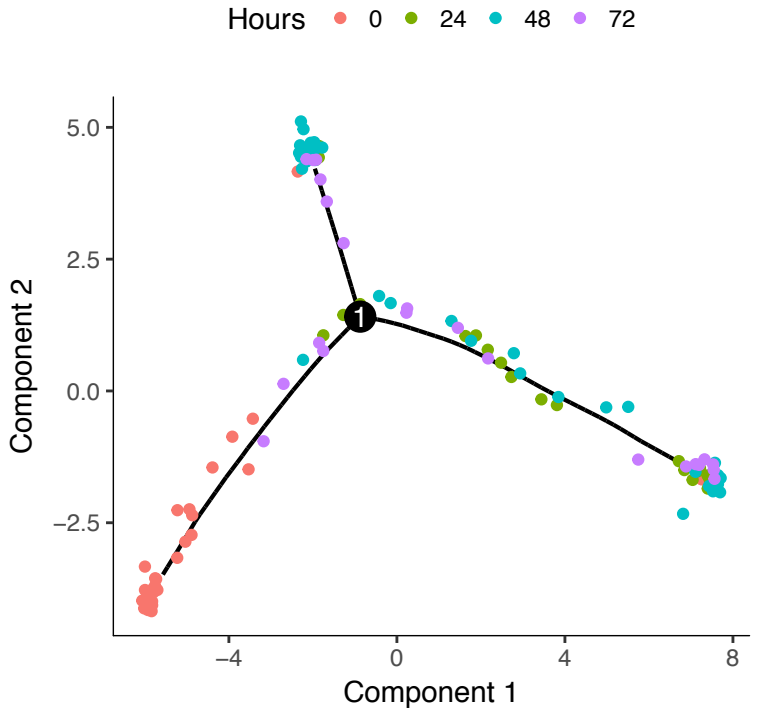
A number of single-cell clustering studies have found that principal component analysis (PCA) is an effective way of finding genes that vary widely across cells. You can use PCA to pick out a good set of ordering genes in a completely unsupervised manner. The procedure below first normalizes the expression data by adding a pseudocount, log-transforming it, and then standardizing it so that all genes have roughly the same dyanmic range. Then, it uses the *irlba* package to perform the PCA. The code below is a little more complicated than just calling `prcomp()`, but it has the advantage of performing the entire computation with sparse matrix operations. This is important for large, sparse CellDataSet objects like the ones common in droplet-based single-cell RNA-Seq.

```
exprs_filtered <- t(t(exprs(HSMM_filtered)/pData(HSMM_filtered)$Size_Factor))
nz_genes <- which(exprs_filtered != 0)
exprs_filtered[nz_genes] <- log(exprs_filtered[nz_genes] + 1)

# Calculate the variance across genes without converting to a dense
# matrix:
expression_means <- Matrix::rowMeans(exprs_filtered)
expression_vars <- Matrix::rowMeans((exprs_filtered - expression_means)^2)

# Filter out genes that are constant across all cells:
genes_to_keep <- expression_vars > 0
exprs_filtered <- exprs_filtered[genes_to_keep,]
expression_means <- expression_means[genes_to_keep]
expression_vars <- expression_vars[genes_to_keep]

# Here's how to take the top PCA loading genes, but using
# sparseMatrix operations the whole time, using irlba.
irlba_pca_res <- irlba(t(exprs_filtered),
                       nu=0,
                       center=expression_means,
                       scale=sqrt(expression_vars),
                       right_only=TRUE)$v

row.names(irlba_pca_res) <- row.names(exprs_filtered)
```

```
# Take the top 100 genes from components 2 and 3. Component
# 1 usually is driven by technical noise.
PC2_genes <- names(sort(abs(irlba_pca_res[, 2]), decreasing = T))[1:100]
PC3_genes <- names(sort(abs(irlba_pca_res[, 3]), decreasing = T))[1:100]

ordering_genes <- union(PC2_genes, PC3_genes)
```

Using these to order the cells as above yields the following trajectory:

```
HSMM_myo <- setOrderingFilter(HSMM_myo, ordering_genes)
HSMM_myo <- reduceDimension(HSMM_myo, max_components=2)
HSMM_myo <- orderCells(HSMM_myo, reverse=FALSE)
plot_cell_trajectory(HSMM_myo, color_by="Hours")
```



4.4 Unsupervised feature selection based on density peak clustering

During our reanalysis of many sc RNA-seq data, we find that genes informative for biological processes often form block structure among clusters of cells involved in the process. We thus developed a unsupervised algorithm, DPfeature, to select those feature genes for reconstructing the trajectory.

DPfeature works as following: Firstly, we need to set the superset of feature genes as genes expressed at least 5% among all the cells.

```
fData(HSMM_myo)$use_for_ordering <- fData(HSMM_myo)$num_cells_expressed > 0.05 * ncol(HSMM_myo)
```

Then we will perform a PCA analysis to identify the variance explained by each PC (principal component). We can look at scree plot and determine how many pca dimensions you want based on whether or not there is a significant gap between that component and the component after it. By selecting only the high loading PCs, we effectively only focus on the more interesting biological variations.

```
HSMM_myo@auxClusteringData[["tSNE"]]$variance_explained <- NULL
plot_pc_variance_explained(HSMM_myo, return_all = F) #look at the plot and decide how many di-
mensions you need. It is determined by a huge drop of variance at that dimension. pass that num-
ber to num_dim in the next function. I used 10 there.

## Error in irlba(FM, nv = min(dim(FM)) - 1, nu = 0, center = expression_means, :  starting vector
near the null space
```

We will then run reduceDimension with t-SNE as the reduction method on those top PCs and project them further down to two dimension.

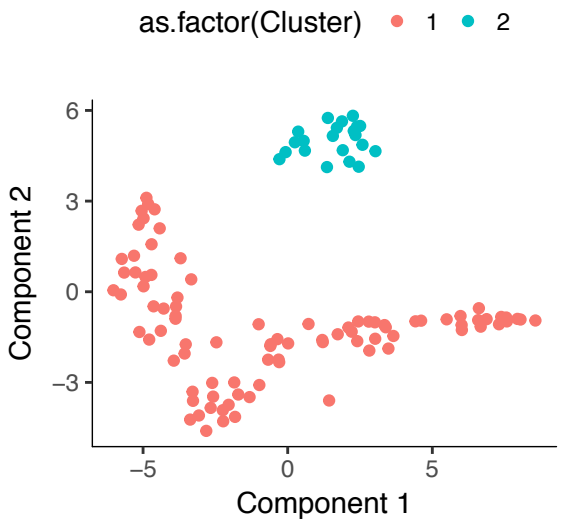
```
  HSMM_myo <- reduceDimension(HSMM_myo, max_components=2, norm_method = 'log', num_dim = 4, reduction_me

## Remove noise by PCA ...
## Reduce dimension by tSNE ...
```

Then we can run density peak clustering to identify the clusters on the 2-D t-SNE space. Density peak algorithm clusters cells based on each cell's local density (ρ) and the nearest distance δ of a cell to another cell with higher distance. We can set a threshold for the ρ, δ and define any cells with higher local density and distance than the thresholds as the density peaks. Those peaks are then used to define the clusters for all cells. By default, `clusterCells` choose 95% of ρ and δ to define the thresholds. We can also set a number of clusters (n) we want to cluster. In this setting, we will find the top n cells with high δ with ρ among the top 50% range. The default setting often gives good clustering.

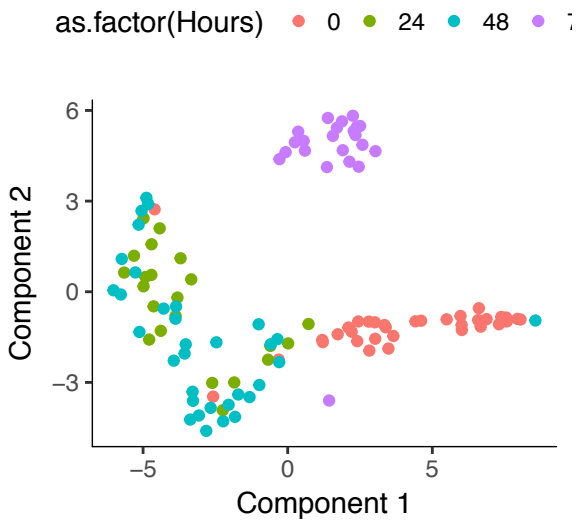
```
HSMM_myo <- clusterCells(HSMM_myo, verbose = F)

## Distance cutoff calculated to 0.5121023

## the length of the distance:  6105
```

After the clustering, we can check the clustering results.

```
plot_cell_clusters(HSMM_myo, color_by = 'as.factor(Cluster)', show_density = F)
```

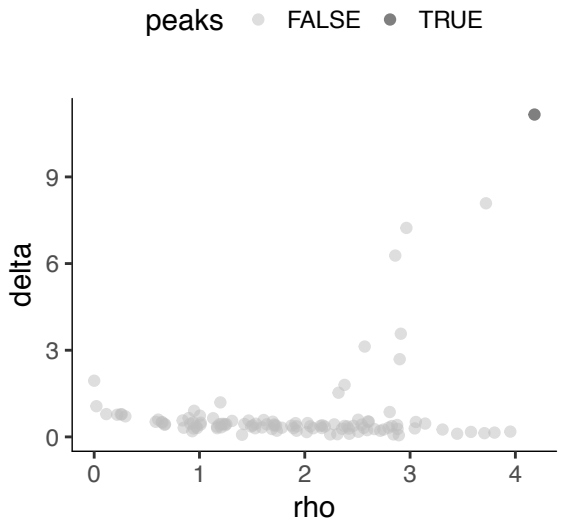



```
plot_cell_clusters(HSMM_myo, color_by = 'as.factor(Hours)', show_density = F)
```



We also provide the decision plot for users to check the ρ, δ for each cell and decide the threshold for defining the cell clusters.

```
plot_rho_delta(HSMM_myo, rho_threshold = 2, delta_threshold = 10 )
```

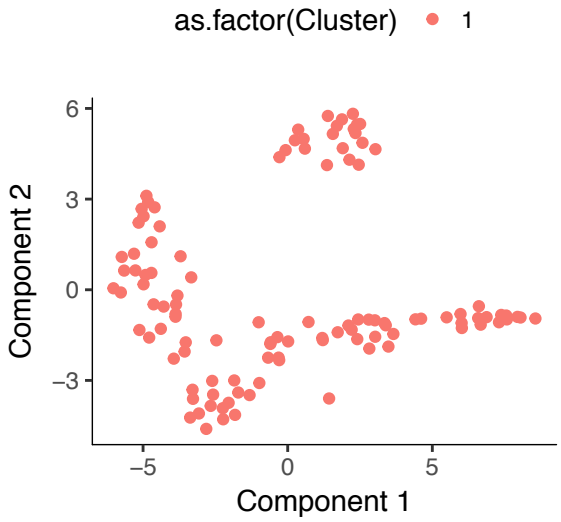


We could then re-run clustering based on the user defined threshold. To facilitate the computation, we can set (skip_rho_sigma = T) which enable us skipping the calculation of the ρ, σ.

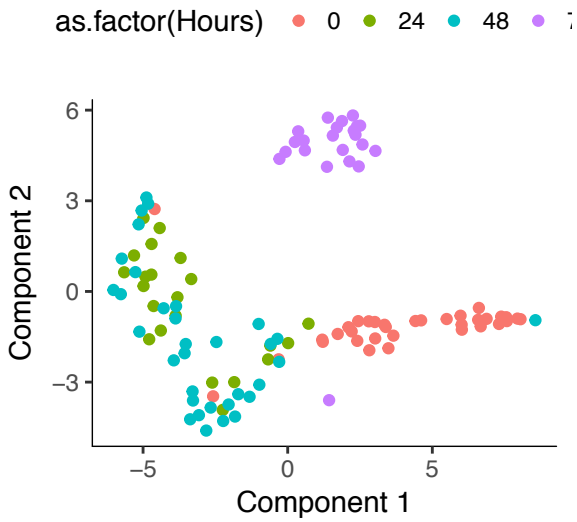
```
HSMM_myo <- clusterCells(HSMM_myo,  rho_threshold = 2, delta_threshold = 10, skip_rho_sigma = T, verbose
```

We can check the final clustering results as following:

```
plot_cell_clusters(HSMM_myo, color_by = 'as.factor(Cluster)', show_density = F)
```




```
plot_cell_clusters(HSMM_myo, color_by = 'as.factor(Hours)', show_density = F)
```

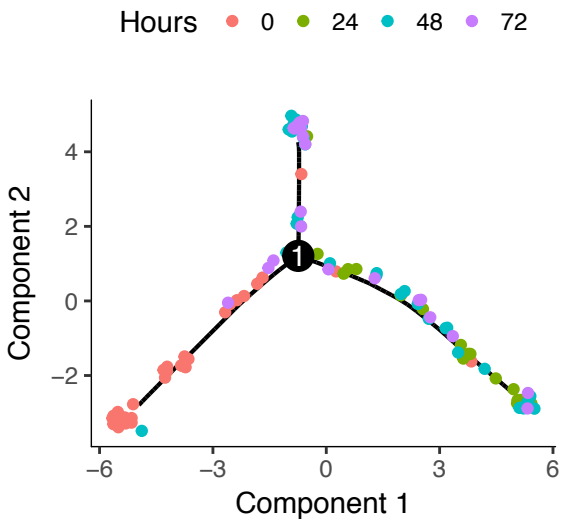


After we confirm the clustering makes sense, we can then perform differential gene expression test as a value to identify those block genes.

```
clustering_DEG_genes <- differentialGeneTest(HSMM_myo, fullModelFormulaStr = '~Cluster', cores = detectC
clustering_DEG_genes_subset <- clustering_DEG_genes[fData(HSMM_myo)$num_cells_expressed > 0.05 * ncol(HS
```

Note that the DEG test is a heuristic approach to identify the genes with block structure, in future we may provide a more principled approach to directly identify those genes with block expression. We will then select the top 1000 significant genes as the ordering genes.

```
HSMM_ordering_genes <- row.names(clustering_DEG_genes_subset)[order(clustering_DEG_genes_subset$qval)][1

HSMM_myo <- setOrderingFilter(HSMM_myo, ordering_genes = HSMM_ordering_genes)
HSMM_myo <- reduceDimension(HSMM_myo)
HSMM_myo <- orderCells(HSMM_myo)

plot_cell_trajectory(HSMM_myo, color_by="Hours")
```



4.5 Semi-supervised ordering with known marker genes

Unsupervised ordering is desirable because it avoids introducing bias into the analysis. However, unsupervised machine learning will often fix on a strong feature of the data that's not the focus of your experiment. For example, where each cell is in the cell cycle has a major impact on the shape of the trajectory when you use unsupervised learning. But what if you wish to focus on cycle-independent effects in your biological process? Monocle's "semi-supervised" ordering mode can help you focus on the aspects of the process you're interested in.

Ordering your cells in a semi-supervised manner is very simple. You first define genes that mark progress using the `CellTypeHierchy` system, very similar to how we used it for cell type classification. Then, you use it to select ordering genes that co-vary with these markers. Finally, you order the cell based on these genes just as we do in unsupervised ordering. So the only difference between unsupervised and semi-supervised ordering is in which genes we use for ordering.

As we saw before, myoblasts begin differnentation by exiting the cell cycle and then proceed through a sequence of regulatory events that leads to expression of some key muscle-specific proteins needed for contraction. We can mark cycling cells with cyclin B2 (*CCNB2*) and recognize myotubes as those cells expressed high levels of myosin heavy chain 3 (*MYH3*).

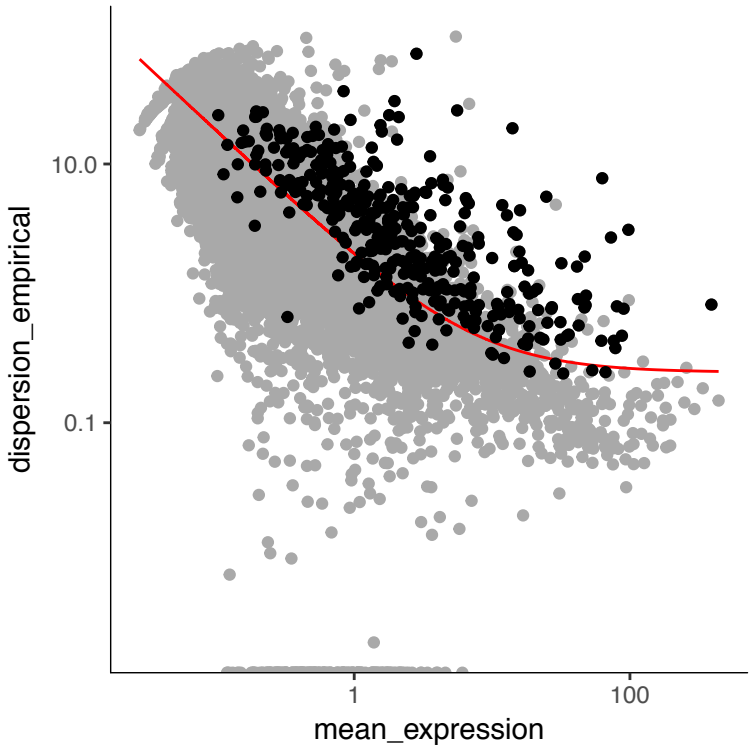
```
CCNB2_id <- row.names(subset(fData(HSMM_myo), gene_short_name == "CCNB2"))
MYH3_id <- row.names(subset(fData(HSMM_myo), gene_short_name == "MYH3"))

cth <- newCellTypeHierarchy()
cth <- addCellType(cth, "Cycling myoblast", classify_func=function(x) {x[CCNB2_id,] >= 1})
cth <- addCellType(cth, "Myotube", classify_func=function(x) {x[MYH3_id,] >=1})
```

Now we select the set of genes that co-vary (in either direction) with these two "bellweather" genes:

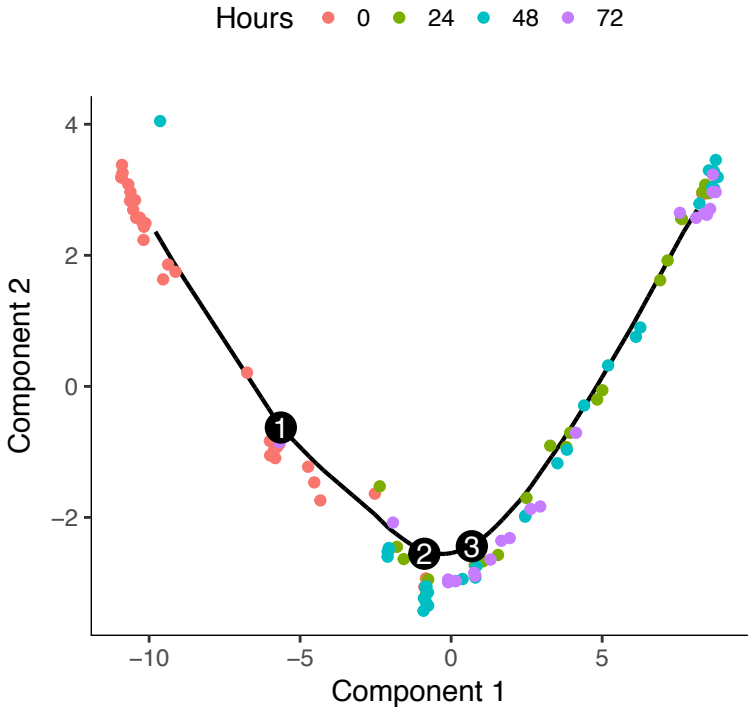
```
marker_diff <- markerDiffTable(HSMM_myo[expressed_genes,],
                               cth,
                               cores=1)
semisup_clustering_genes <- row.names(subset(marker_diff, qval < 0.01))
length(semisup_clustering_genes)

## [1] 373
```

Using the 705 genes for ordering produces a trajectory that's highly similar to the one we obtained with unsupervised methods, but it's a little "cleaner": the small branches are gone.

```
HSMM_myo <- setOrderingFilter(HSMM_myo, semisup_clustering_genes)
plot_ordering_genes(HSMM_myo)
```



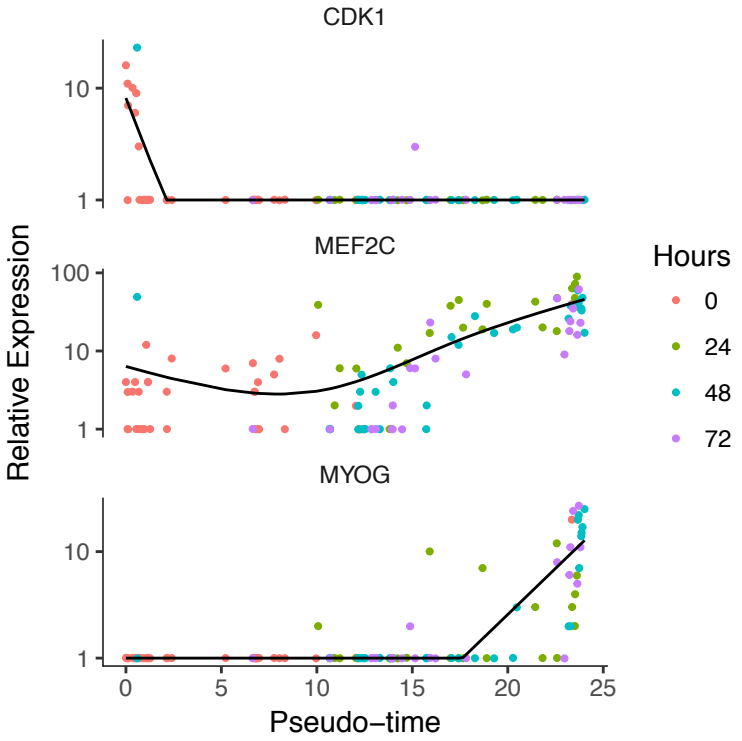

```
HSMM_myo <- reduceDimension(HSMM_myo, max_components=2)
HSMM_myo <- orderCells(HSMM_myo, reverse=TRUE)
plot_cell_trajectory(HSMM_myo, color_by="Hours")
```



And we can see that kinetics of notable genes we examined before are largely the same.

```
HSMM_filtered <- HSMM_myo[expressed_genes,]

my_genes <- row.names(subset(fData(HSMM_filtered),
                             gene_short_name %in% c("CDK1", "MEF2C", "MYOG")))
```

```
cds_subset <- HSMM_filtered[my_genes,]
plot_genes_in_pseudotime(cds_subset, color_by="Hours")
```

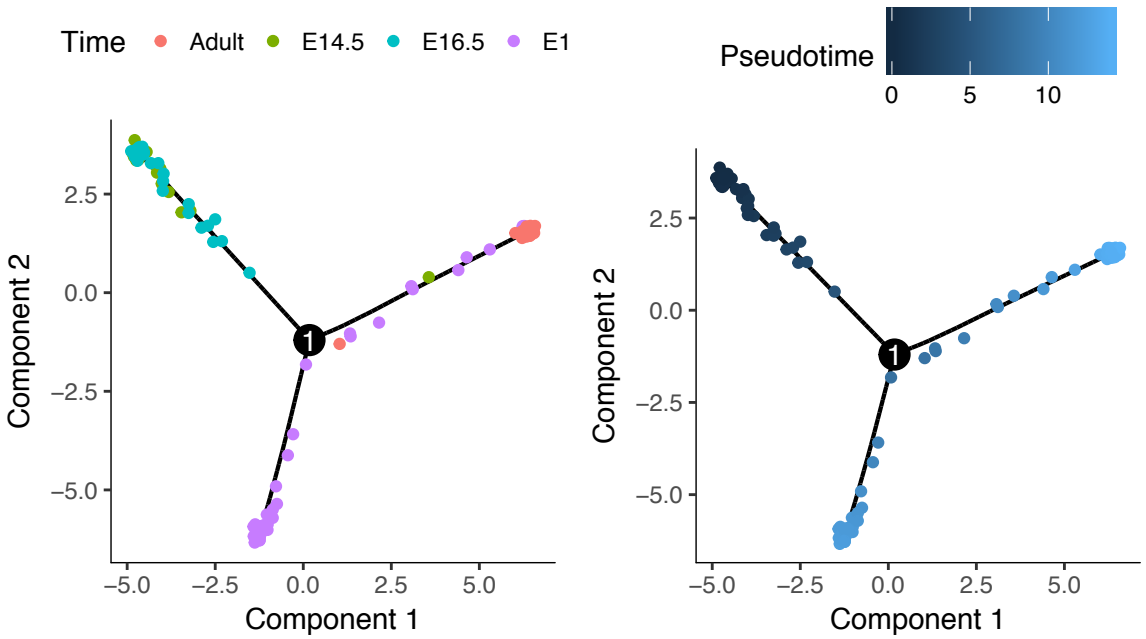


4.6 Reconstructing branched trajectories

Monocle reconstructs a linear trajectory because muscle differentiation is a relatively simple process with just one outcome for the cells. It can't tell a-priori which side of the trajectory is the start and which is the end, but that's easily solved with the `reverse` flag. However, if the trajectory includes multiple outcomes, Monocle might need a little more help from you. Consider the experiment performed by Steve Quake's lab by Barbara Treutlein and colleages, who captured cells from the developing mouse lung. They captured cells early in development, later when the lung contains both major types of epithelial cells (AT1 and AT2), and cells right about to make the decision to become either AT1 or AT2. Monocle can reconstruct this process as a *branched* trajectory, allow you to analyze the decision point in great detail. We'll learn more about branch analysis in section 8.9.

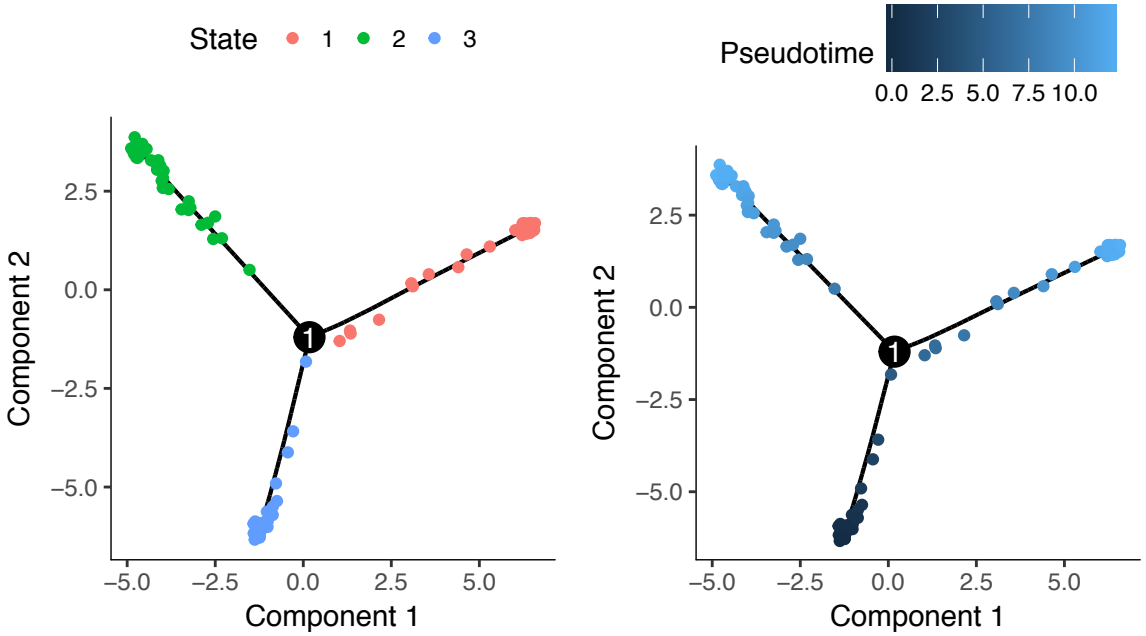
Monocle reconstructs branched trajectories, but it doesn't necessarily know which part of of the trajectory is the beginning. It's important to tell Monocle where the trajectory starts so that the Pseudotime axis runs in the right direction and reflects the underlying biological process. In the figure below, you can see that Monocle reconstructs a tree with two outcomes from the Truetlein data. The "root" of the trajectory happens to the branch where most of the cells collected early in the experiment (E14.5 and E16.5) can be found. So Monocle happened to start the trajectory at the right spot in this case.

```
lung <- load_lung()

## Removing 4 outliers

plot_cell_trajectory(lung, color_by="Time")
plot_cell_trajectory(lung, color_by="Pseudotime")
```



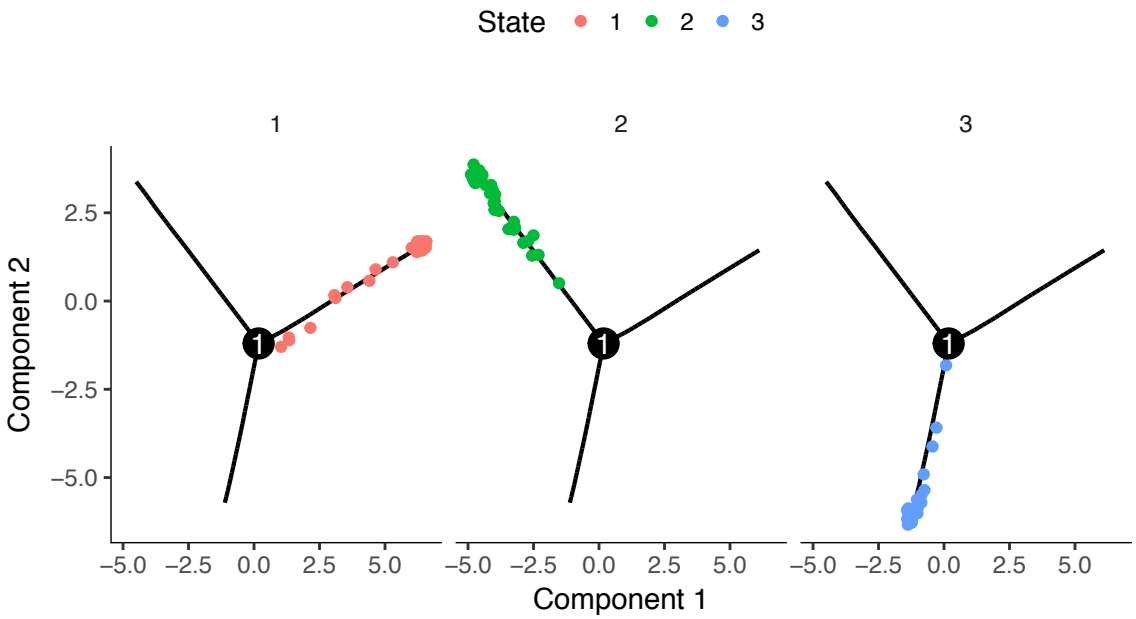
If however, we want to change the root of the tree, we first plot the trajectory coloring the cells by "State" so we can see our options for where we can set the root. "State" is just Monocle's jargon for the different segments of the tree. This tree has 3 States, but more complex trees could have quite a few more. We then call `order_cells()` *again*, passing this state as the `root_state` argument:

```
plot_cell_trajectory(lung, color_by="State")
lung <- orderCells(lung, root_state=3)
plot_cell_trajectory(lung, color_by="Pseudotime")
```



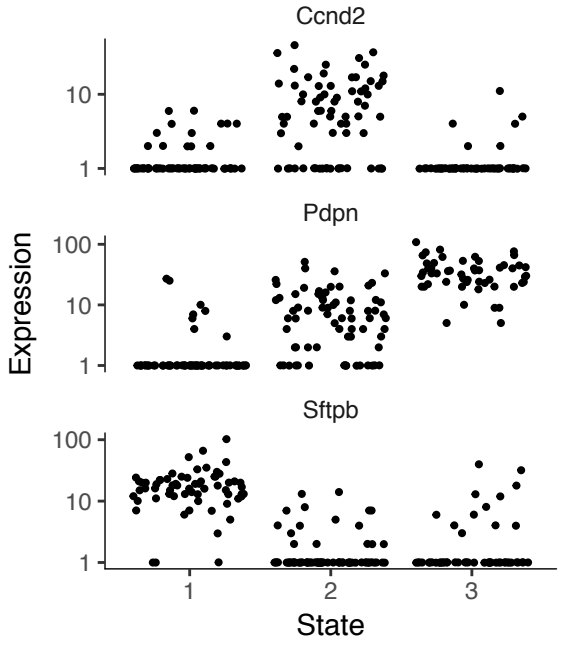
If there are a ton of states in your tree, it can be a little hard to make out where each one falls on the tree. Sometimes it can be handy to "facet" the trajectory plot so it's easier to see where each of the states are located:

```
plot_cell_trajectory(lung, color_by="State") + facet_wrap(~State)
```



And if you don't have a timeseries, you might need to set the root based on where certain marker genes are expressed, using your biological knowledge of the system. For example, in this experiment, a highly proliferative population of progenitor cells are generating two types of post-mitotic cells. So the root should have cells that express high levels of proliferation markers. We can use the jitter plot to pick figure out which state corresponds to rapid proliferation:

```
lung_genes <- row.names(subset(fData(lung), gene_short_name %in% c("Ccnd2", "Sftpb", "Pdpn")))
plot_genes_jitter(lung[lung_genes,], grouping="State")
```



Here we can see that State 1 is the proliferative state on the basis high expression of Cyclin B2.

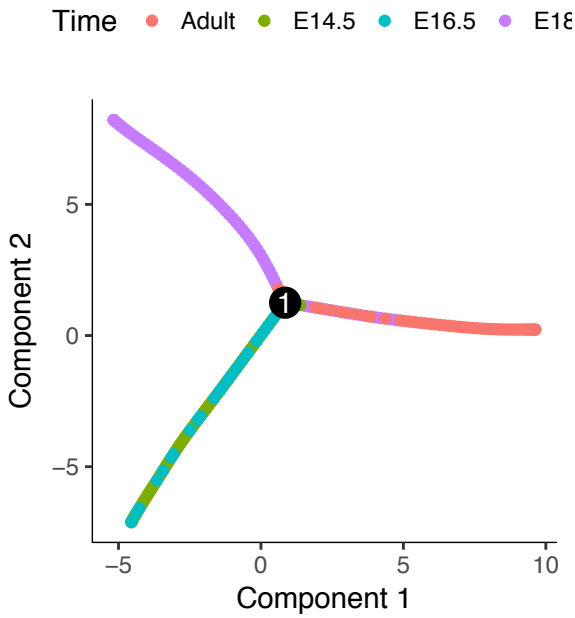
4.7 Applying other reversed graph embedding algorithms for trajectory reconstruction

Monocle integrates several reversed graph embedding algorithms for reconstructing the developmental trajectory. By default, monocle uses DDRTree for trajectory reconstruction which learns the principal graph and performs dimension reduction simulatenously. On the other hand, users may be interested in performing a dimension reduction first and then apply Monocle to learn the principal graph on the reduced dimension. For example, users of the destiny package (another nice method for pseudotime estimation and branch identification ([])), can first apply diffusion map and

then learn the prinicipal graph on the low dimension. By default, simplePPT, SGL-tree and $\mathcal{L}_1$-graph use PCA as initial dimension reduction. It then learns the principal graph on the low dimensional PCA space. The following scripts show how can we run simplePPT and SGL-tree easily. Because $\mathcal{L}_1$-graph tries to learn a general graph which is a very difficult task, the graph learned from $\mathcal{L}_1$-graph captures the main structure of the data, the disconnected component from the graph often prevents us to have an easy solution to assign pseudotime for each cell. On the other hand, since L1-graph can learn arbitrary structure, it is promising to learn loop structure or other disconnected graph structure. Monocle users should expect more powerful methods based on L1-graph will be released soon.

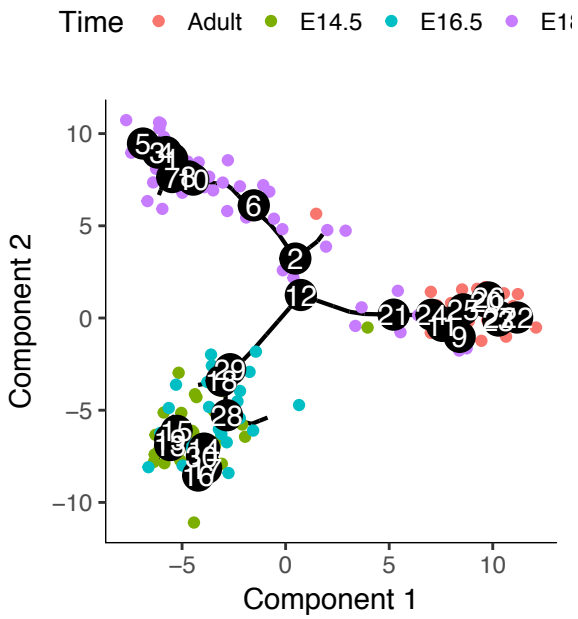
Result after applying SimplePPT:

```
#run SimplePPT
lung <- reduceDimension(lung, norm_method="log", pseudo_expr = 1, reduction_method = 'SimplePPT') #
lung <- orderCells(lung)
plot_cell_trajectory(lung, color_by="Time")
```



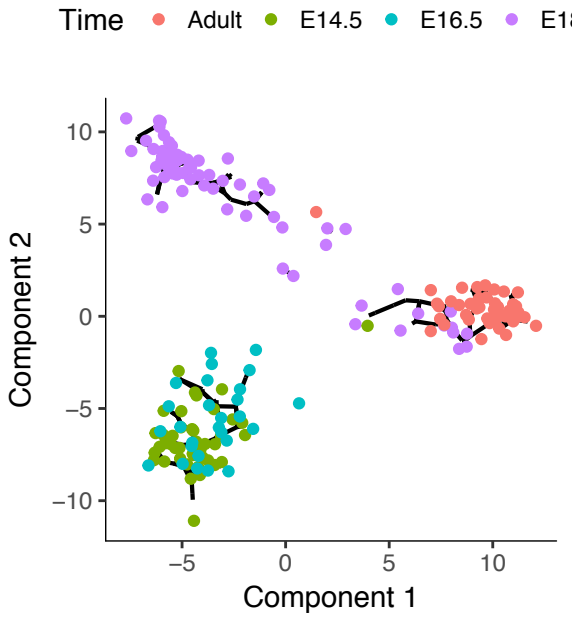
Result after applying SGL-tree:

```
#run SGL-tree
lung <- reduceDimension(lung, norm_method="log", pseudo_expr = 1, reduction_method = 'SGL-tree') #
lung <- orderCells(lung)
plot_cell_trajectory(lung, color_by="Time")
```



Result after applying L1-graph:

```
#run L1 graph
lung <- reduceDimension(lung, norm_method="log", pseudo_expr = 1, reduction_method = 'L1-graph') #
plot_cell_trajectory(lung, color_by="Time", show_branch_points = F)
```



4.8 Initialize RGE with other non-linear dimension reduction method

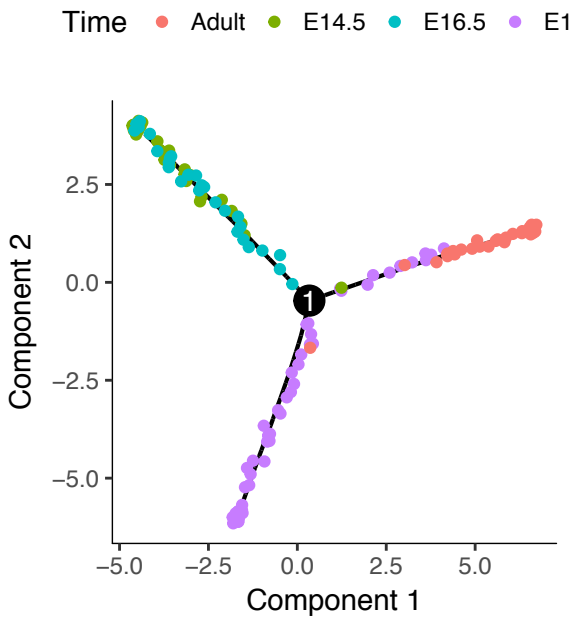
Recently, there are a few groups applied some non-linear dimension reduction methods, including diffusion maps, LLE (local linear embedding), etc., to facilitate the trajectory reconstruction. Monocle 2 provides a very convinient way for users who are interested in them to seamlessly integrate RGE with those methods by simplying passing a dimension reduction method to the argument `initial`$_{m}ethodin$`reduceDimension`.$For example, we can use the nice package$`destiny`$from Fab$

```
#run L1 graph
library(destiny)
run_dpt <- function(data, branching = T, norm_method = 'log', root = NULL, verbose = F){
  if(verbose)
```

```
    message('root should be the id to the cell not the cell name ....')

  data <- data[!duplicated(data), ]
  dm <- DiffusionMap(as.matrix(data))
  return(dm@eigenvectors)
}
```

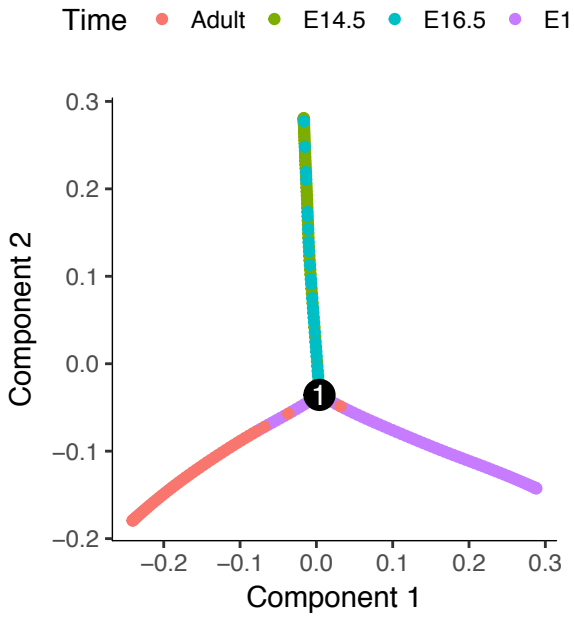
Result after applying DDRTree:

```
#run DDRTree
lung <- reduceDimension(lung, norm_method="log", pseudo_expr = 1, reduction_method = 'DDRTree', initial_
lung <- orderCells(lung)
plot_cell_trajectory(lung, color_by="Time")
```



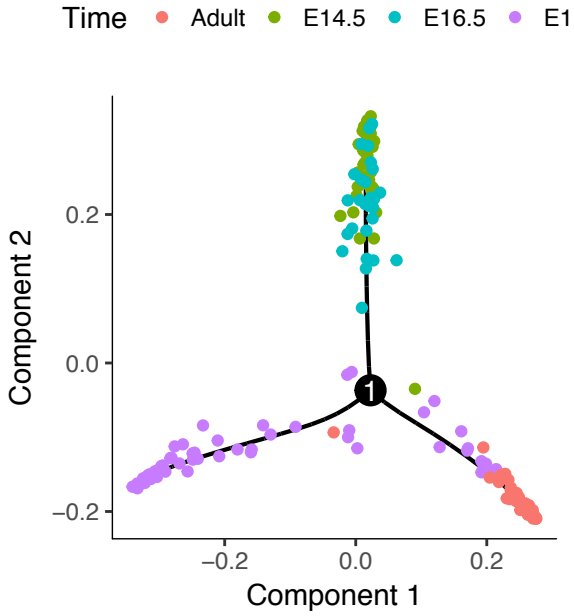
Result after applying SimplePPT:

```
#run SimplePPT
lung <- reduceDimension(lung, norm_method="log", pseudo_expr = 1, reduction_method = 'SimplePPT', initia
lung <- orderCells(lung)
plot_cell_trajectory(lung, color_by="Time")
```



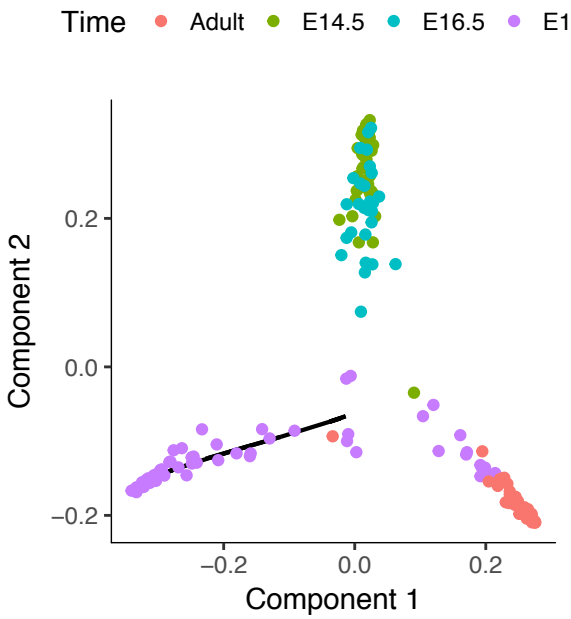
Result after applying SGL-tree:

```
#run SGL-tree
lung <- reduceDimension(lung, norm_method="log", pseudo_expr = 1, reduction_method = 'SGL-tree', initial
lung <- orderCells(lung)
plot_cell_trajectory(lung, color_by="Time")
```



Result after applying L1-graph:

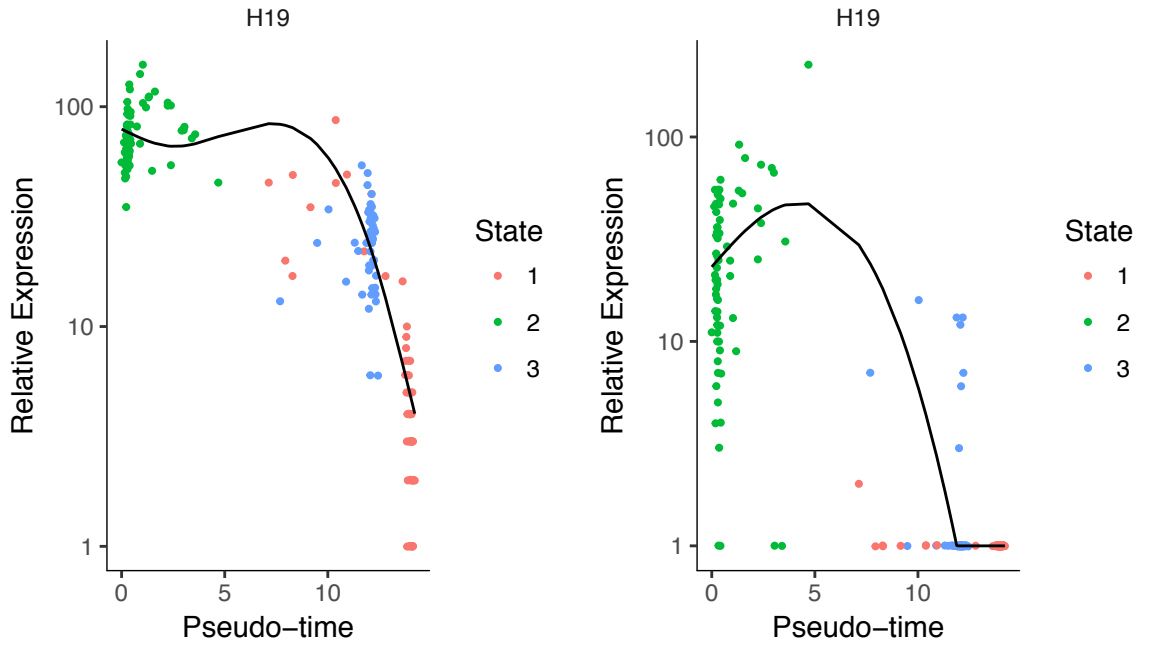
```
#run L1 graph
lung <- reduceDimension(lung, norm_method="log", pseudo_expr = 1, reduction_method = 'L1-graph', initial
plot_cell_trajectory(lung, color_by="Time", show_branch_points = F)
```



4.9 Reverse embed the low dimension data back to high dimension data

DDRTree learns cell positions in the intrinsic low dimension space and a principal graph for those positions as well as a mapping function from low dimension back to the high dimension. We can apply this mapping function to those cell positions to recover the "de-noised" high dimension data point. Then we will scale up the de-noised data point to match up the expression in the original data. This can be directly achieved by using reverseEnbeddingCDS.

```
#run reverseEnbedingCDS
lung <- load_lung() # rerun lung data with DDRTree

## Removing 4 outliers

RGE_cds <- reverseEnbeddingCDS(lung)
plot_genes_in_pseudotime(RGE_cds[1, ])
plot_genes_in_pseudotime(lung[1, ])
```



One observation we can immediately make is that reverse embedded data are smoother and the drop out in the original data are largely alleviated. Whether or not we can apply the denoised data for downstream analysis or for data imputation for dropout may be interesting future directions.

5 Differential expression analysis

Differential gene expression analysis is a common task in RNA-Seq experiments. Monocle can help you find genes that are differentially expressed between groups of cells and assesses the statistical signficance of those changes. These comparisons require that you have a way to collect your cells into two or more groups. These groups are defined by columns in the phenoData table of each CellDataSet Monocle will assess the signficance of each gene's expression level across the different groups of cells.

5.1 Basic differential analysis

Performing differential expression analysis on all genes in the human genome can take a substantial amount of time. For a dataset as large as the myoblast data from [1], which contains several hundred cells, the analysis can take several hours on a single CPU. Let's select a small set of genes that we know are important in myogenesis to demonstrate Monocle's capabilities:

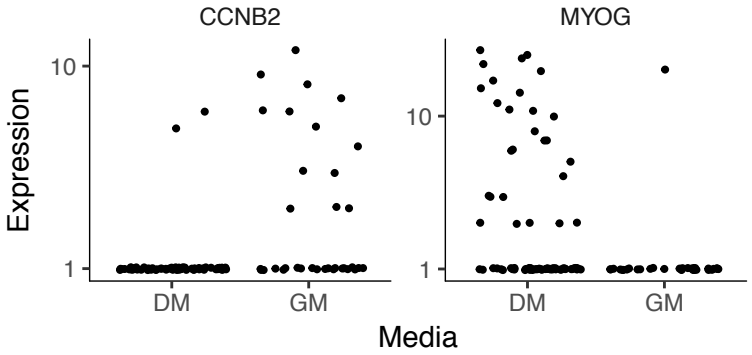
```
marker_genes <- row.names(subset(fData(HSMM_myo),
                                 gene_short_name %in% c("MEF2C", "MEF2D", "MYF5",
                                                        "ANPEP", "PDGFRA","MYOG",
                                                        "TPM1",  "TPM2",  "MYH2",
                                                        "MYH3",  "NCAM1", "TNNT1",
                                                        "TNNT2", "TNNC1", "CDK1",
                                                        "CDK2",  "CCNB1", "CCNB2",
                                                        "CCND1", "CCNA1", "ID1")))
```

In the myoblast data, the cells collected at the outset of the experiment were cultured in ■growth medium■ (GM) to prevent them from differentiating. After they were harvested, the rest of the cells were switched over to ■differentiation medium■ (DM) to promote differentiation. Let's have monocle find which of the genes above are affected by this switch:

```
diff_test_res <- differentialGeneTest(HSMM_myo[marker_genes,],
                                      fullModelFormulaStr="~Media")

# Select genes that are significant at an FDR < 10%
sig_genes <- subset(diff_test_res, qval < 0.1)

sig_genes[,c("gene_short_name", "pval", "qval")]

##                     gene_short_name         pval         qval
## ENSG00000081189.9            MEF2C 1.786873e-20 9.381081e-20
## ENSG00000105048.12           TNNT1 4.095823e-10 1.228747e-09
## ENSG00000109063.9             MYH3 2.287928e-33 1.601549e-32
## ENSG00000111049.3             MYF5 9.223719e-35 9.684905e-34
## ENSG00000114854.3            TNNC1 2.033274e-16 7.116458e-16
## ENSG00000118194.14           TNNT2 3.800019e-38 7.980040e-37
## ENSG00000122180.4             MYOG 1.143403e-09 2.667941e-09
## ENSG00000125414.13            MYH2 7.462178e-07 1.305881e-06
## ENSG00000125968.7               ID1 3.090564e-03 4.326790e-03
## ENSG00000134057.10           CCNB1 6.390120e-08 1.219932e-07
## ENSG00000140416.15            TPM1 5.686648e-08 1.194196e-07
## ENSG00000149294.12           NCAM1 5.076913e-17 2.132303e-16
## ENSG00000157456.3            CCNB2 6.226063e-10 1.634342e-09
## ENSG00000166825.9            ANPEP 6.291204e-02 8.257206e-02
## ENSG00000170312.11            CDK1 2.971693e-06 4.800427e-06
## ENSG00000198467.8             TPM2 1.230467e-04 1.845700e-04
```

So 16 of the 22 genes are significant at a 10% false discovery rate! This isn't surprising, as most of the above genes are highly relevant in myogenesis. Monocle also provides some easy ways to plot the expression of a small set of genes grouped by the factors you use during differential analysis. This helps you visualize the differences revealed by the tests above. One type of plot is a ■jitter■ plot.

```
MYOG_ID1 <- HSMM_myo[row.names(subset(fData(HSMM_myo),
                                      gene_short_name %in% c("MYOG", "CCNB2"))),]
plot_genes_jitter(MYOG_ID1, grouping="Media", ncol=2)
```



Note that we can control how to layout the genes in the plot by specifying the number of rows and columns. See the man page on plot_genes_jitter for more details on controlling its layout. Most if not all of Monocle's plotting routines return a plot object from the *ggplot2*. This package uses a grammar of graphics to control various aspects of a plot, and makes it easy to customize how your data is presented. See the *ggplot2* book [?] for more details.

In this section, we'll explore how to use Monocle to find genes that are differentially expressed according to several different criteria. First, we'll look at how to use our previous classification of the cells by type to find genes that distinguish fibroblasts and myoblasts. Second, we'll look at how to find genes that are differentially expressed as a function of pseudotime, such as those that become activated or repressed during differentiation. Finally, you'll see how to perform multi-factorial differential analysis, which can help subtract the effects of confounding variables in your experiment.

To keep the vignette simple and fast, we'll be working with small sets of genes. Rest assured, however, that Monocle can analyze many thousands of genes even in large experiments, making it useful for discovering dynamically regulated genes during the biological process you're studying.

5.2 Finding genes that distinguish cell type or state

During a dynamic biological process such as differentiation, cells might assume distinct intermediate or final states. Recall that earlier we distinguished myoblasts from contaminating fibroblasts on the basis of several key markers. Let's look at several other genes that should distinguish between fibroblasts and myoblasts.

```
to_be_tested <- row.names(subset(fData(HSMM),
                                 gene_short_name %in% c("UBC", "NCAM1", "ANPEP")))
cds_subset <- HSMM[to_be_tested,]
```

To test the effects of CellType on gene expression, we simply call differentialGeneTest on the genes we've selected. However, we have to specify a *model formula* in the call to tell Monocle that we care about genes with expression levels that depends on *CellType*. Monocle's differential expression analysis works essentially by fitting two models to the expression values for each gene, working through each gene independently. The first of the two models is called the *full* model. This model is essentially a way of predicting the expression value of each gene in a given cell knowing only whether that cell is a fibroblast or a myoblast. The second model, called the *reduced* model, does the same thing, but it doesn't know the CellType for each cell. It has to come up with a reasonable prediction of the expression value for the gene that will be used for *all* the cells. Because the full model has more information about each cell, it will do a better job of predicting the expression of the gene in each cell. The question Monocle must answer for each

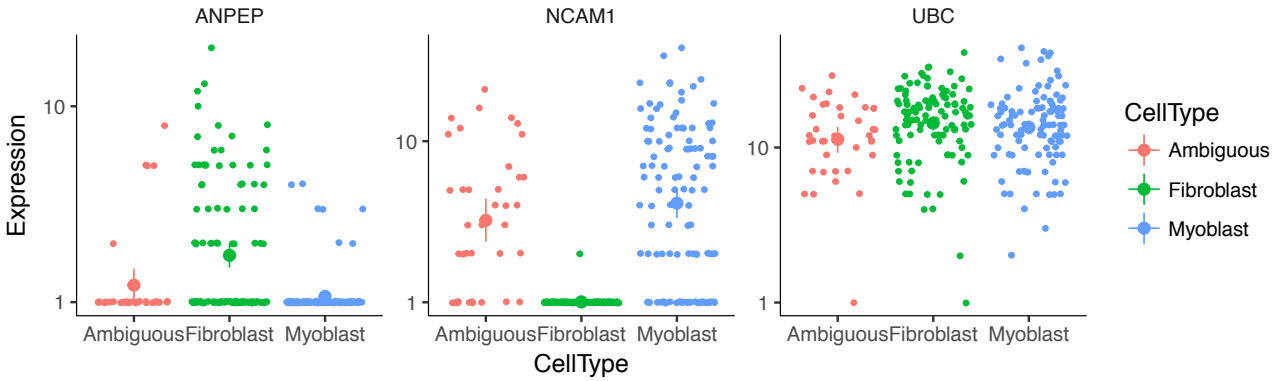
gene is *how much better* the full model's prediction is than the reduced model's. The greater the improvement that comes from knowing the CellType of each cell, the more significant the differential expression result. This is a common strategy in differential analysis, and we leave a detailed statistical exposition of such methods to others.

To set up the test based on CellType, we simply call differentialGeneTest with a string specifying fullModelFormulaStr. We don't have to specify the reduced model in this case, because the default of 1 is what we want here.

```
diff_test_res <- differentialGeneTest(cds_subset, fullModelFormulaStr="~CellType")
diff_test_res[,c("gene_short_name", "pval", "qval")]

##                     gene_short_name         pval         qval
## ENSG00000149294.12           NCAM1 5.886066e-93 1.765820e-92
## ENSG00000150991.10             UBC 2.173875e-01 2.173875e-01
## ENSG00000166825.9            ANPEP 1.803623e-15 2.705434e-15
```

Note that all the genes are significantly differentially expressed as a function of CellType except the housekeeping gene TBP, which we're using a negative control. However, we don't know which genes correspond to myoblast-specific genes (those more highly expressed in myoblasts versus fibroblast specific genes. We can again plot them with a jitter plot to see:

```
plot_genes_jitter(cds_subset, grouping="CellType", color_by="CellType",
                  nrow=1, ncol=NULL, plot_trend=TRUE)

## geom_path: Each group consists of only one observation. Do you need to
## adjust the group aesthetic?
## geom_path: Each group consists of only one observation. Do you need to
## adjust the group aesthetic?
## geom_path: Each group consists of only one observation. Do you need to
## adjust the group aesthetic?
```



Note that we could also simply compute summary statistics such as mean or median expression level on a per-CellType basis to see this, which might be handy if we are looking at more than a handful of genes. Of course, we could test for genes that change as a function of Hours to find time-varying genes, or Media to identify genes that are responsive to the serum switch. In general, model formulae can contain terms in the pData table of the CellDataSet.

The differentialGeneTest function is actually quite simple ■under the hood■. The call above is equivalent to:

```
full_model_fits <- fitModel(cds_subset,  modelFormulaStr="~CellType")
reduced_model_fits <- fitModel(cds_subset, modelFormulaStr="~1")
diff_test_res <- compareModels(full_model_fits, reduced_model_fits)
diff_test_res
```

Occassionally, as we'll see later, it's useful to be able to call fitModel directly.

5.3 Finding genes that change as a function of pseudotime

Monocle's main job is to put cells in order of progress through a biological process (such as cell differentiation) without knowing which genes to look at ahead of time. Once it's done so, you can analyze the cells to find genes that changes as the cells make progress. For example, you can find genes that are significantly upregulated as the cells ■mature■. Let's look at a panel of genes important for myogenesis:

```
to_be_tested <- row.names(subset(fData(HSMM),
                                 gene_short_name %in% c("MYH3", "MEF2C", "CCNB2", "TNNT1")))
cds_subset <- HSMM_myo[to_be_tested,]
```

Again, we'll need to specify the model we want to use for differential analysis. This model will be a bit more complicated than the one we used to look at the differences between CellType. Monocle assigns each cell a ■pseudotime■ value, which records its progress through the process in the experiment. The model can test against changes as a function of this value. Monocle uses the *VGAM* package to model a gene's expression level as a smooth, nonlinear function of pseudotime:

```
diff_test_res <- differentialGeneTest(cds_subset,  fullModelFormulaStr="~sm.ns(Pseudotime)")
```

The sm.ns function states that Monocle should fit a natural spline through the expression values to help it describe the changes in expression as a function of progress. We'll see what this trend looks like in just a moment. Other smoothing functions are available.

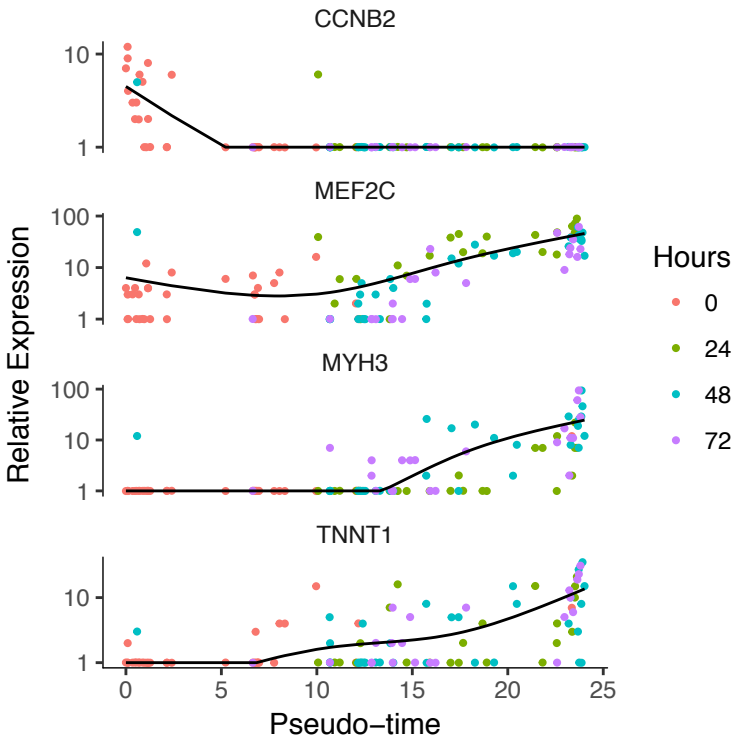
Once again, let's add in the gene annotations so it's easy to see which genes are significant.

```
diff_test_res[,c("gene_short_name", "pval", "qval")]
```

```
##                     gene_short_name         pval         qval
## ENSG00000081189.9             MEF2C 6.800087e-40 1.360017e-39
## ENSG00000105048.12            TNNT1 4.680095e-29 6.240127e-29
## ENSG00000109063.9              MYH3 1.037228e-65 4.148911e-65
## ENSG00000157456.3             CCNB2 3.278366e-19 3.278366e-19
```

We can plot the expression levels of these genes, all of which show significant changes as a function of differentiation, using the function plot_genes_in_pseudotime. This function has a number of cosmetic options you can use to control the layout and appearance of your plot.

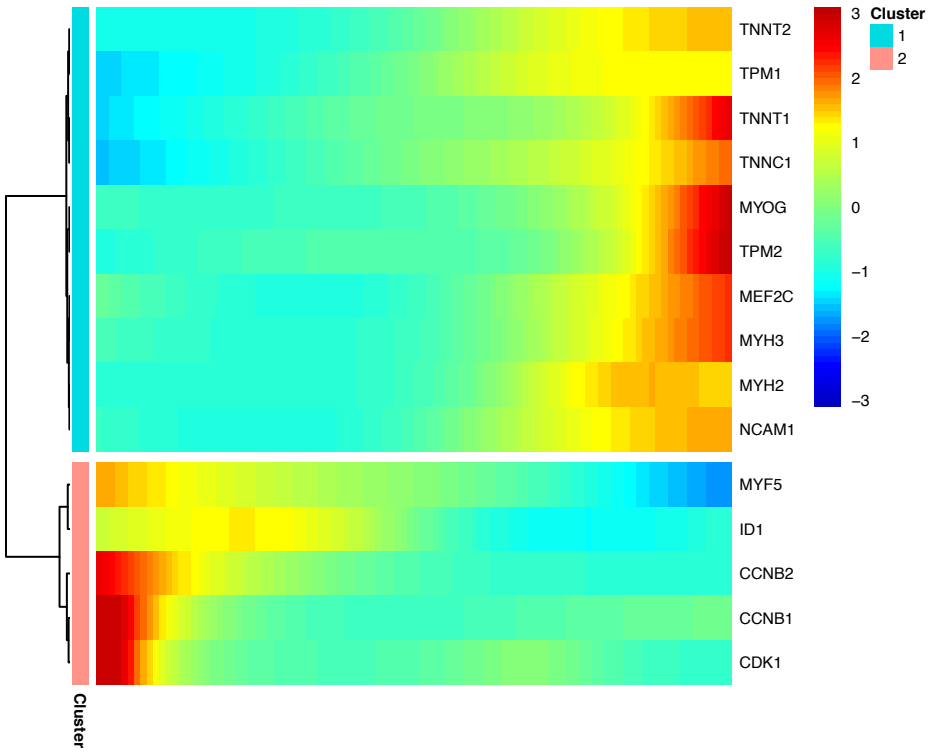
```
plot_genes_in_pseudotime(cds_subset, color_by="Hours")
```



5.4 Clustering genes by pseudotemporal expression pattern

A common question that arises when studying time-series gene expression studies is: ■which genes follow similar kinetic trends■? Monocle can help you answer this question by grouping genes that have similar trends, so you can analyze these groups to see what they have in common. Monocle provides a convenient way to visualize all pseudotime-dependent genes. The function plot_pseudotime_hea takes a CellDataSet object (usually containing a only subset of significant genes) and generates smooth expression curves much like plot_genes_in_pseudotime. Then, it clusters these genes and plots them using the *pheatmap* package. This allows you to visualize modules of genes that co-vary across pseudotime.

```
diff_test_res <- differentialGeneTest(HSMM_myo[marker_genes,],
                                      fullModelFormulaStr="~sm.ns(Pseudotime)")

sig_gene_names <- row.names(subset(diff_test_res, qval < 0.1))


plot_pseudotime_heatmap(HSMM_myo[sig_gene_names,],
                        num_clusters = 2,
                        cores = 1,
                        show_rownames = T)
```

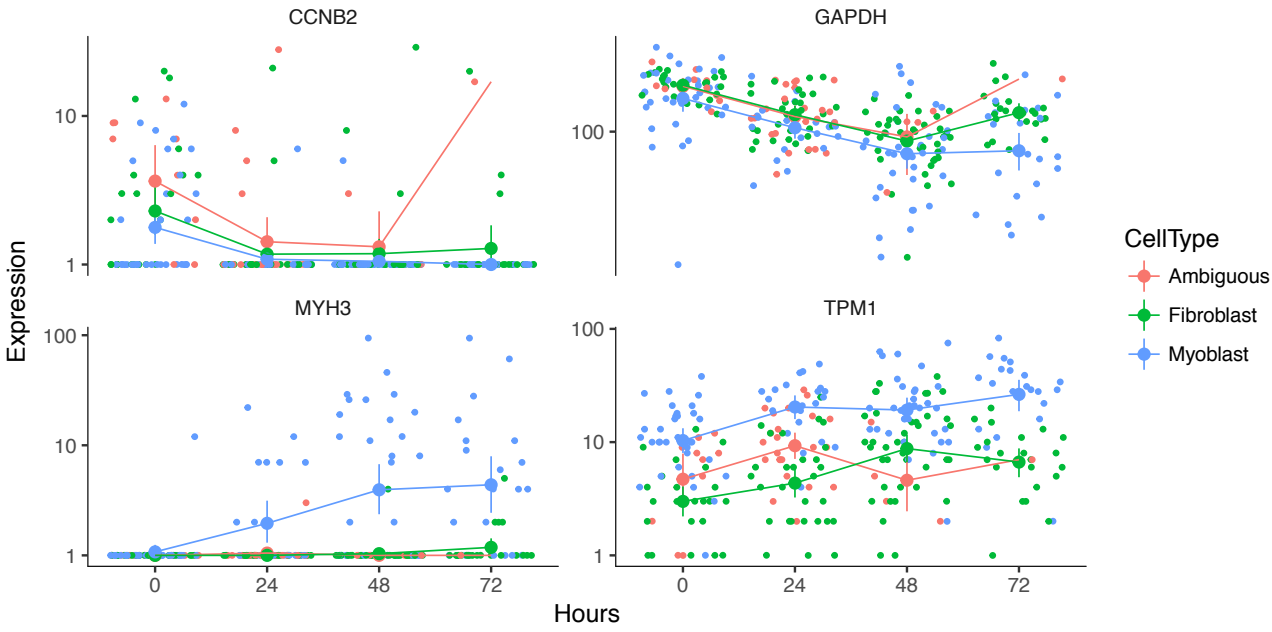


5.5 Multi-factorial differential expression analysis

Monocle can perform differential analysis in the presence of multiple factors, which can help you subtract some factors to see the effects of others. In the simple example below, Monocle tests three genes for differential expression between myoblasts and fibroblasts, while subtracting the effect of Hours, which encodes the day on which each cell was collected. To do this, we must specify both the full model and the reduced model. The full model captures the effects of both CellType and Hours, while the reduced model only knows about Hours.

When we plot the expression levels of these genes, we can modify the resulting object returned by plot_genes_jitter to allow them to have independent y-axis ranges, to better highlight the difference between cell states.

```
to_be_tested <- row.names(subset(fData(HSMM),
                                 gene_short_name %in% c("TPM1", "MYH3", "CCNB2", "GAPDH")))
cds_subset <- HSMM[to_be_tested,]

diff_test_res <- differentialGeneTest(cds_subset,
                                      fullModelFormulaStr="~CellType + Hours",
                                      reducedModelFormulaStr="~Hours")
diff_test_res[,c("gene_short_name", "pval", "qval")]

##                     gene_short_name         pval         qval
## ENSG00000109063.9              MYH3 2.445319e-64 9.781277e-64
## ENSG00000111640.10            GAPDH 1.525177e-01 1.525177e-01
## ENSG00000140416.15             TPM1 6.591956e-30 1.318391e-29
## ENSG00000157456.3             CCNB2 1.887679e-11 2.516905e-11

plot_genes_jitter(cds_subset,
                  grouping="Hours", color_by="CellType", plot_trend=TRUE) +
  facet_wrap( ~ feature_label, scales="free_y")
```

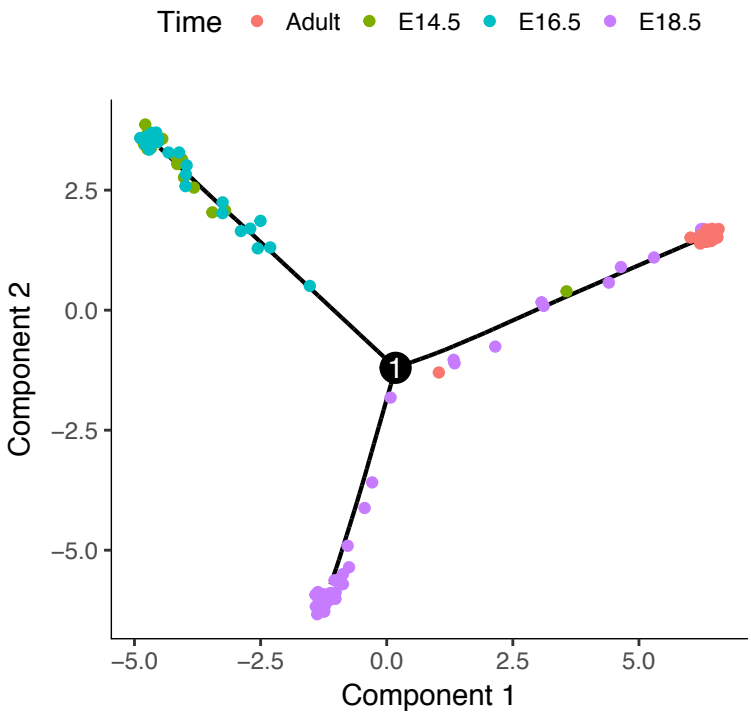


6 Analyzing branches in single-cell trajectories

Often, single-cell trajectories include branches. The branches occur because cells execute alternative gene expression programs. Branches appear in trajectories during development, when cells make fate choices: one developmental lineage proceeds down one path, while the other lineage produces a second path. Monocle contains extensive functionality for analyzing these branching events.

Truetlein and colleagues perfomed a single cell analysis of lung epithelial cell development that reveals a key cell fate decision. The figure below shows the trajectory Monocle reconstructs using some of their data. There is a single branch, labeled "1". What genes change as cells pass from the early developmental stage the top left of the tree through the branch? What genes are differentially expressed between the branches? To answer this question, Monocle provides you with a special statistical test: branched expression analysis modeling, or BEAM.

```
lung <- load_lung()

## Removing 4 outliers

plot_cell_trajectory(lung, color_by="Time")
```

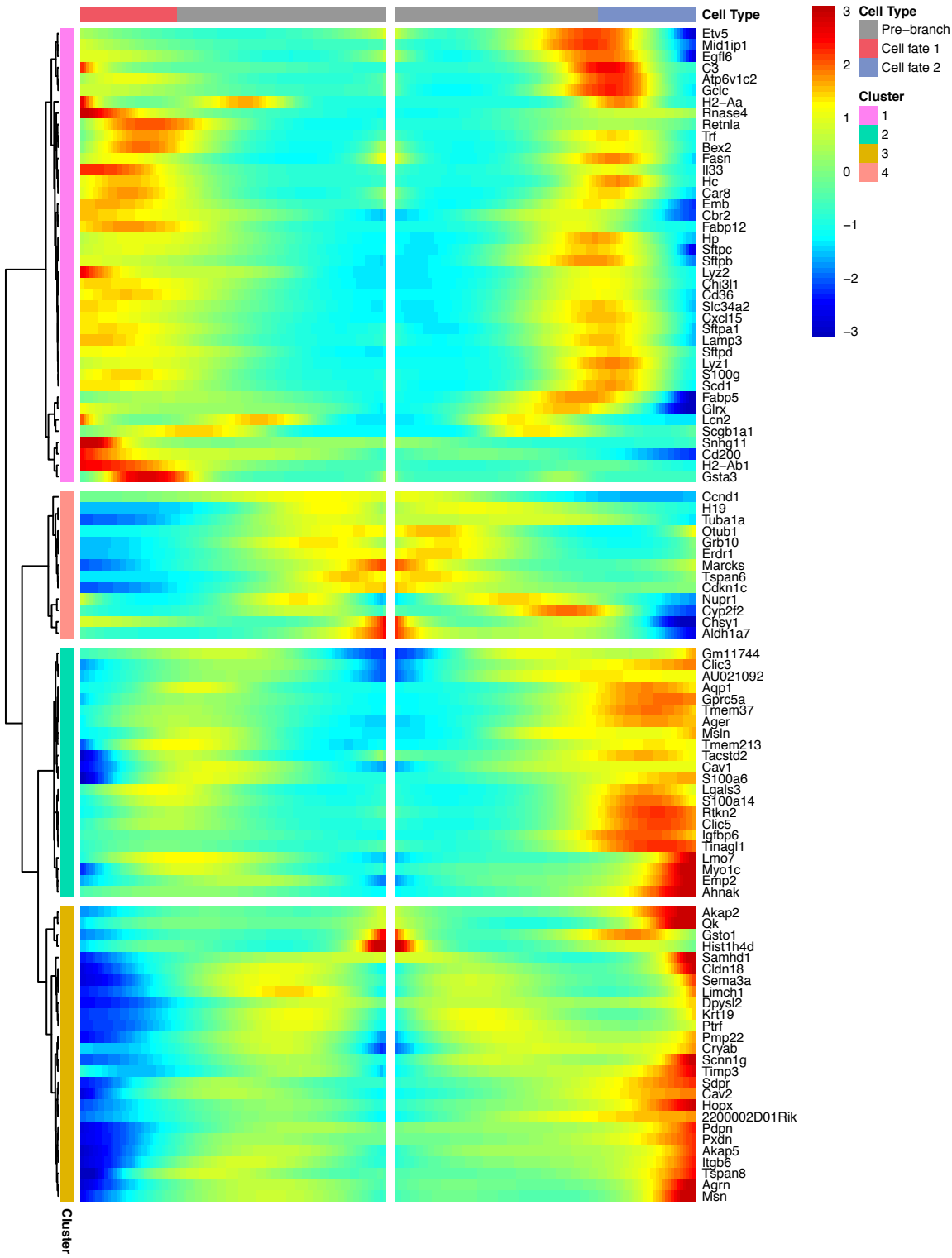


BEAM takes as input a CellDataSet that's been ordered with orderCells and the name of a branch point in the trajectory. It returns a table of significance scores for each gene. Genes that score significant are said to be *branch-dependent* in their expression.

```
BEAM_res <- BEAM(lung, branch_point=1, cores = 1)
BEAM_res <- BEAM_res[order(BEAM_res$qval),]
BEAM_res <- BEAM_res[,c("gene_short_name", "pval", "qval")]
```

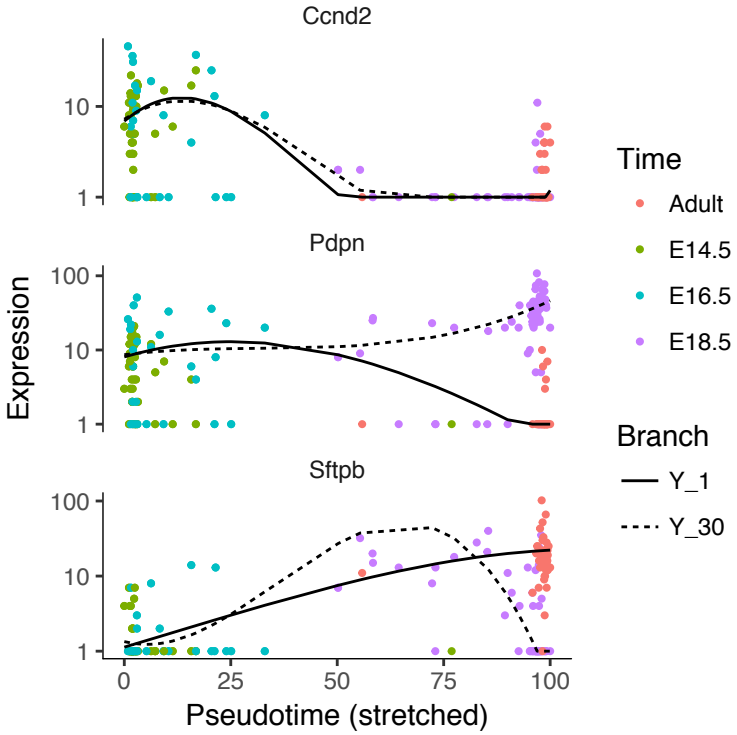
You can visualize changes for all the genes that are significantly branch dependent using a special type of heatmap. This heatmap shows changes in both lineages at the same time. It also requires that you choose a branch point to inspect. Columns are points in pseudotime, rows are genes, and the beginning of pseudotime is in the middle of the heatmap. As you read from the middle of the heatmap to the right, you are following one lineage through pseudotime. As you read left, the other. The genes are clustered hierarchically, so you can visualize modules of genes that have similar lineage-dependent expression patterns.

```
plot_genes_branched_heatmap(lung[row.names(subset(BEAM_res, qval < 1e-4)),],
                            branch_point = 1,
                            num_clusters = 4,
                            cores = 1,
                            use_gene_short_name = T,
                            show_rownames = T)
```



We can plot a couple of these genes, such as *Pdpn* and *Sftpb* (both known markers of fate in this system), using the plot_genes_branched_pseudotime function, which works a lot like the plot_genes_i function, except it shows two kinetic trends, one for each lineage, instead of one. We also show *Ccnd2*, a cell cycle gene, which is downregulated in both branches and is not signficant by the BEAM test.

```
lung_genes <- row.names(subset(fData(lung), gene_short_name %in% c("Ccnd2", "Sftpb", "Pdpn")))
plot_genes_branched_pseudotime(lung[lung_genes,],
                               branch_point=1,
                               color_by="Time",
                               ncol=1)
```



The plot_clusters function returns a ggplot2 object showing the shapes of the expression patterns followed by the 100 genes we've picked out. The topographic lines highlight the distributions of the kinetic patterns relative to the overall trend lines, shown in red.

7 Major changes between Monocle version 1 and version 2

Monocle 2 is a near-complete re-write of Monocle 1. Monocle 2 is geared towards larger, more complex single-cell RNA-Seq experiments than those possible at the time Monocle 1 was written. It's also redesigned to support analysis of mRNA counts, which were hard to estimate experimentally in early versions of single-cell RNA-Seq. Now, with spike controls or UMIs, gene expression can be measured in mRNA counts. Analysis of these counts is typically easier and more accurate than relative expression values, and we encourage all users to adopt an mRNA-count centered workflow. Numerous Monocle functions have been re-written to take advantage of the nicer statistical properties of mRNA counts. For example, we adopt the dispersion modeling and variance-stabilization techniques introduced by DESeq [3] during differential analysis, dimensionality reduction, and other steps.

Trajectory reconstruction in Monocle 2 is vastly more robust, faster, and more powerful than in Monocle 1. Monocle 2 uses an advanced nonlinear reconstruction algorithm called DDRTree [4], described below in section 8. This algorithm can expose branches that are hard to see with the less powerful linear technique used in Monocle 1. The algorithm is also far less sensitive to outliers, so careful QC and selection of high quality cells is less critical. Finally, DDRTree is much more robust in that it reports qualititatively similar trajectories more consistently when you vary the number of cells in the experiment. Although which genes are included in the ordering still greatly impact the trajectory, varying them also produces more qualititatively consistent trajectories than the previous linear technique.

Because Monocle 2 is so much better at finding branches, it also includes some additional tools to help you interpret them. Branch expression analysis modeling (BEAM) is a new test for analyzing specific branch points [5]. BEAM reports branch-dependent genes, and Monocle 2 includes some new visualization functions to help you inspect these genes. Overall, we find that branching is pervasive in diverse biological processes, and thus we expect BEAM will be very useful to those analyzing single-cell RNA-Seq data in many settings.

A manuscript describing Monocle 2 and the general stragegy of using reversed graph embedding for single-cell trajectory analysis will be forthcoming (Qiu et al, in preparation). Until it appears, please continue to cite [1] when you use Monocle or discuss single-cell pseudotemporal ordering in your work.

8 Theory behind Monocle

8.1 DPfeature: feature selection by detecting DEGs across cluster of cells

An interesting observation we found when analyzing multiple datasets associated with development is that genes informative to differentiation trajectory often form clusters across cell states (Figure 1 of [?], Figure 2-4 of [?] or Figure 2 of [?]) . That is, there are clusters of genes which have relative high expression in the progenitor cell states but low expression in the terminal cell state, etc. Inspired by this observation, we developed a simple procedure, termed dpFeature, to automatically select those clusters of genes that have the block expression patterns.

First, dpFeature filters genes that only expressed in a very small percentage of cells (by default, 5%). Second, dpFeature performs PCA on the expressed genes and then users can decide the number of principal components (PCs) used for downstream analysis based on whether or not there is a significan drop in the variance explained at the selected component. These top PCs will be further used to initialize t-SNE which projects the cells into two-dimension t-SNE space. Third, dpFeature uses a recently developed density based clustering algorithm, called density peak clustering [?] to cluster the cells in the two-dimensional t-SNE space. The density peak clustering algorithm calculates each cell's local density (ρ) and distance (δ) of a cell to another cell with higher density. The ρ and δ values for each cell can be plotted in a so-called decision plot. Cells with high local density that are far away from other cells with high local density correspond to the density peaks. These density peaks nucleate clusters: all other cells will be associated with the nearest density peak cell to form clusters. Finally, we identify genes that differ between the clusters by performing a likelihood ratio test between a negative-binomial generalized linear model that knows the cluster to which each cell is assigned and a model that doesn't. We then select (by default) the top 1,000 significantly differentially expressed genes, after Benjamini-Hochberg correction, as the ordering genes for the trajectory reconstruction.

8.2 Reversed graph embedding

Single-cell expression datasets are some of the largest and most complex encountered in genomics. Even the smallest single-cell RNA-Seq experiments sample hundreds of cells, measuring the expression level of the more than 20,000 genes in each cell. Visualizing these datasets, identifying cells of different types, and comparing them to one another all pose major bioinformatics challenges.

Manifold learning is a common strategy for dealing with complex, high-dimensional data. The premise of this approach is simple: the data may reside in a very high-dimensional space, but the intrinsic structure of the dataset is much simpler. Moreover, the data are not random - they are generated by a process that can be understood by inspecting the global structure of the dataset. For example, a single single-cell RNA-Seq experiment may reside in 20,000 dimensions, but the cells might all lie on or ■near■ a curve *embedded* within a much lower dimensional space. For example, we might expect that cells in different phases of the cell cycle be distributed along a closed loop. Indeed a recent large-scale single-cell RNA-Seq study found exactly that [6].

Manifold learning often involves *dimensionality reduction* techniques as a first step. Conventional, dimensionality reduction approaches (for example, PCA, ICA, Isomap, LLE, etc.) are limited in their ability to explictly recover the intrinisic structure from the data.

Monocle 2 uses a newly developed technique, called reversed graph embedding, to simultaneously:

1. Reduce high dimensional expression data into a lower dimension space.
2. Learn an explicit, smooth manifold that generates the data.
3. Assign each cell to its position on that manifold

Together, these tasks allow Monocle 2 to order cells in pseudotime in an entirely unsupervised, data-driven way. Importantly, Monocle 2 learns manifolds that are trees without needing any *a priori* information about the structure of the tree. Users do not need to provide Monocle 2 with constraints on the number of branches, etc. These are learned from the data. This allows Monocle 2 to to discriminate between linear and branched trajectories automatically. To our knowledge, Monocle 2 is the first trajectory reconstruction algorithm to learn smooth tree-like manifolds without needing to know its high-level structure ahead of time.

8.3 Reversed graph embedding

Monocle 2 uses a technique called *reversed graph embedding* [7] to learn the structure of the manifold that describes a single-cell experiment.

Reversed graph embedding simultaneously learns a *principal graph* that approximates the manifold, as well as a function that maps points on the graph (which is embedded in low dimensions) back to the original high dimensional space. Reversed graph embedding aims to learn both a set of *latent points* $\mathcal{Z} = \{\mathbf{z}_1, ..., \mathbf{z}_M\}$ corresponding to the input data that reside in the low-dimensional space along with a graph $\mathcal{G}$ that connects them. This graph approximates the manifold. In order to map points on the manifold back to the original high-dimensional input space, we also need to learn a function $f_\mathcal{G}$.

Learning a good reversed graph embedding can be described as an optimization problem that joint captures the positions of the latent points $\mathbf{z}$, the graph $\mathcal{G}$, and the function $f_\mathcal{G}$.

To learn the positions of the latent points $\mathbf{z}$, we must optimize:

$$\min_{f_g \in \mathcal{F}} \min_{\{\mathbf{z}_1,...,\mathbf{z}_M\}} \sum_{i=1}^{N} ||\mathbf{x}_i - f_g(\mathbf{z}_i)||^2 \tag{1}$$

Given a set of latent point coordinates, the optimization of graph inference can be represented as:

$$\min_{f_\mathcal{G} \in \mathcal{F}} \min_{\{\mathbf{z}_1,...,\mathbf{z}_M\}} \sum_{(V_i,V_j)\in\mathcal{E}} b_{i,j} ||f_g(\mathbf{z}_i) - f_g(\mathbf{z}_j)||^2 \tag{2}$$

where $\mathcal{X} = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$ are the original single-cell expression profiles. The V_i are the the vertices of the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The weights for the edge in $\mathcal{E}$ are encoded as b_{ij}.

The first optimization problem aims to position the latent points such that their image under $f_\mathcal{G}$ (that is, their corresponding positions in the high-dimensional space) will be "close" to the input data. The second optimization aims to keep latent points that are close to one another in the low dimensional space close to one another in the high dimensional space as well. These two goals must be balanced against one another. Reversed graph embedding achieves this through the parameter λ

$$\min_{\mathcal{G}\in\hat{\mathcal{G}}_b} \min_{f_g \in \mathcal{F}} \min_{\{\mathbf{z}_1,...,\mathbf{z}_M\}} \sum_{i=1}^{N} ||\mathbf{x}_i - f_g(\mathbf{z}_i)||^2 + \frac{\lambda}{2} \sum_{(V_i,V_j)\in\mathcal{E}} b_{i,j} ||f_g(\mathbf{z}_i) - f_g(\mathbf{z}_j)||^2 \tag{3}$$

Reversed graph embedding requires a feasible set $\hat{\mathcal{G}}_b$ of graph and a mapping function $f_\mathcal{G}$. In practice, implementing reversed graph embedding requires that we place some constraints on $\hat{\mathcal{G}}_b$ and $f_\mathcal{G}$. As work on reversed graph embedding continues, we anticipate that more general schemes that consider a wider range of feasible graphs and mapping functions will become available. Monocle users should expect more general reversed graph embedding schemes in future versions.

Mao *et al* initially described two specific ways to implement the general framework of reversed graph embedding. Both are briefly summarized below. See the original paper on DDRTree for more details. Monocle 2 uses the second scheme, but can easily be run in a mode that corresponds to the first.

8.4 DRTree: Dimensionality reduction via learning a tree

The first scheme, called ■DRTree■ described by Mao *et al* learns principal graphs that are undirected trees, with one node per input data point, along with a linear function $f_\mathcal{G}$. Because the algorithm restricts the feasible set to trees, optimization problem 2 is solved by simply finding the minimum spanning tree. This can be solved quickly via Kruskal's algoritm. DRTree uses a linear projection model $f_\mathcal{G}(\mathbf{z}) = \mathbf{W}\mathbf{z}$ as the mapping function. The scheme optimizes:

$$\min_{\mathbf{W},\mathbf{Z},\mathbf{B}} \sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{W}\mathbf{z}_i||^2 + \frac{\lambda}{2} \sum_{i,j} b_{i,j} ||\mathbf{W}\mathbf{z}_i - \mathbf{W}\mathbf{z}_j||^2 \tag{4}$$

where $\mathbf{W} = [\mathbf{w}_1, ..., \mathbf{w}_d] \in \mathcal{R}^{D\times d}$ is an orthogonal set of d linear basis vectors. Because several key steps of the above optimization can be solved analytically, and because the two terms can be minimized in an alternating fashion, solving it is generally very fast. However, for large input datasets, the graph can become complex, and so DRTree can run into scalability problems.

8.5 DDRTree: Discriminative dimensionality reduction via learning a tree

To overcome problems posed by large complex input datasets, Mao *et al* proposed a second scheme, "DDRTree". that Monocle 2 uses instead of DRTree. Recall that the graph in DRTree contains one node per input data point. To avoid long computations and large memory footprints that come with DRTree, the second scheme learns a graph on a second, smaller set of latent points $\{\mathbf{y}_k\}_{k=1}^K$. These points are treated by the algorithm as the centers of $\{\mathbf{z}_i\}_{i=1}^N$. The number of these points is controlled through the ncenter argument to reduceDimension() in Monocle 2. Using this algorithm drastically speeds up the computations in reduceDimension(), and also serves to regularize the manifold, often producing cleaner, more accurate single-cell trajectories.

The DDRTree scheme works via the following optimization:

$$\min_{\mathbf{W},\mathbf{Z},\mathbf{B},\mathbf{Y},\mathbf{R}} \sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{W}\mathbf{z}_i||^2 + \frac{\lambda}{2}\sum_{k,k'} b_{k,k'}||\mathbf{W}\mathbf{y}_k - \mathbf{W}\mathbf{y}'_k||^2 + \gamma\Big[\sum_{k=1}^{K}\sum_{i=1}^{N} r_{i,k}||\mathbf{z}_i - \mathbf{y}_k||^2 + \sigma\Omega(\mathbf{R})\Big] \quad (5)$$

The optimization is constrained such that $\mathbf{W}^T\mathbf{W} = \mathbf{I}, \sum_{k=1}^{K} r_{i,k} = 1, r_{i,k} \leq 0, \forall i, \forall k$. The matrix $\mathbf{R} \in \mathcal{R}^{N\times N}$ is used to regularize the graph, through the *negative entropy regularization* $\Omega(\mathbf{R}) = \sum_{i=1}^{N}\sum_{k=1}^{k} r_{i,k}\log\ r_{i,k}$. In effect, DDRTree uses the latent points $\{\mathbf{y}_k\}_{k=1}^K$ as the centers of K clusters. That is, the algorithm acts as soft K-means clustering on the points $\{\mathbf{z}_i\}_{i=1}^N$, and jointly learns a graph on the K cluster centers. The matrix $\mathbf{R}$ transforms the hard assignments used in K-means into soft assignments with $\sigma > 0$ as a regularization parameter.

Problem 5 again contains a number of analytical steps, and can be solved by alternating between the terms. Moreover, because some of the more expensive numerical operations involve matrices that are K dimensional (instead of N dimensional), they have complexity that is invariant of the size of the input data.

Our accessory package *DDRTree* implements the DDRTree algorithm using a number of key performance optimizations. Monocle 2 calls DDRTree to learn the core manifold describing a CellDataSet, and then computes pseudotime coordinates and branch assignments using this manifold. Monocle also uses the manifold in downstream analysis steps such as BEAM. A manuscript describing the general stragegy of using reversed graph embedding for single-cell trajectory analysis has been posted on biorxiv ([]).

8.6 SimplePPT: A simple principal tree algorithm.

SimplePPT is the first RGE technique proposed by Mao et al for learning a tree structure to describe a set of observed data points. The tree can be learned in the original space or in some lower dimension retrieved by some dimensionality reduction method such as PCA [?]. SimplePPT makes some choices that simplify the optimization problem. Notably, $f_G(z_i)$ is optimized as one single variable instead of two separate sets of variables. Moreover, the loss function in the reversed graph embedding is replaced by the empirical quantization error, which serves as the measurement between the $f_G(z_i)$ and its corresponding observed points x_i. The joint optimization of $f_G(z_i)$ is efficient from the perspective of optimization with respect to $\{b_{ij}\}$, which is solved by simply finding the minimum spanning tree.

8.7 The principal $\mathcal{L}_1$ graph algorithm.

Mao et al later proposed an extension of SimplePPT that can learn arbitrary graphs, rather than just trees, which describe large datasets embedded in the same space as the input [?]. An $\mathcal{L}_1$ graph is a sparse graph which is based on the assumption that each data point (or cell) has a small number of neighborhoods in which the minimum number of points that span a low-dimensional affine subspace [?] passing through that point. In addition, there may exist noise in certain elements of z_i and a natural idea is to estimate the edge weights by tolerating these errors. In general, a sparse solution is more robust and facilitates the consequent identification of test sample (or sequenced single-cell samples). Unlike SimplePPT, this method learns the graph by formulating the optimization as a linear programming problem.

In the same work [?], they also proposed a generalization of SimplePPT, which we term as SGL-tree (Principal Graph and Structure Learning for tree), to learn tree structure for large dataset by similarly considering clustering of data points as in DDRTree. Principal $\mathcal{L}_1$ graph and SGL-tree are all treated as SGL in this study.

8.8 Census

In [5], we introduced Census, a normalization method to convert of single-cell mRNA transcript to relative transcript counts. Using relative transcript counts (or spike-in derived counts or UMI counts if available) with the negative binomial distribution can dramatically improve the differenti expression test compared to using the negative binomial with read counts or the Tobit with TPM/FPKM values.

Census aims to convert relative abundances X_{ij} into lysate transcript counts Y_{ij}. Without loss of generality, we consider relative abundances is on the TPM scale, and assume that a gene's TPM value is proportional to the relative frequencies of its mRNA within the total pool of mRNA in a given cell's lysate, i.e., $TPM_{ij} \propto \frac{Y_{ij}}{\sum_{j=1} Y_{ij}}$. The generative model discussed in Qiu *et al.* predicts that when only a minority of the transcripts in a cell is captured in the library, signal from most detectably expressed genes will originate from a single mRNA. Because the number of sequencing reads per transcript is proportionate to molecular frequency after normalizing for length (i.e. TPM or FPKM), all such genes in a given cell should have similar TPM values.

Census works by first identifying the (log-transformed) TPM value in each cell i, written as x_i^*, that corresponds to genes from which signal originates from a single transcript. Because our generative model predicts that these most detectable genes should fall into this category, we simply estimate x_i^* as the mode of the log-transformed TPM distribution for cell i. This mode is obtained by log-transforming the TPM values, performing a Gaussian kernel density estimation and then identifying the peak of the distribution. Given the TPM value for a single transcript in cell i, it is straightforw to convert all relative abundances to their lysate transcript counts. The total number of mRNAs captured for cell i can be estimated as:

$$M_i = \frac{1}{\theta} \cdot \frac{n_i}{\frac{1}{T}\int_{\epsilon}^{X_i^*} X_{i,j} dX} \tag{6}$$

where ϵ is a TPM value below which no mRNA is believed to be present (by default, 0.1), n_i is the number of genes with TPM values in the interval (ϵ, x_i^*) and T is the sum of TPM values of all expressed genes in a single cell. That is, we could estimate the total mRNA counts as the total number of single-mRNA genes divided by their combined expression relative to all genes, as illustrated in Figure 1A of [5]. However, cDNA and PCR amplification steps during library prep can lead to superlinear growth in the relative abundance of a transcript as a function of copy number. In practice, the above formula often over-estimates total mRNAs in the lysate. To alleviate this issue, we used an alternative formula in Census:

$$M_i = \frac{1}{\theta} \cdot \frac{n_i}{F_{X_i}(X_i^*) - F_{X_i}(\epsilon)} = \frac{1}{\theta} \cdot \frac{n_i}{F_{X_i}(X_i^*)} \qquad (x \geq \epsilon) \tag{7}$$

where F_{X_i} represents the cumulative distribution function for the TPM values of genes expressed above ϵ for cell i. Effectively, this simple approach estimates the pre-amplification cDNA count as the number of genes expressed above ϵ. Although this is necessarily an underestimate, in practice it is typically close to the true total (as shown in Figure 1c in [5]), since a large fraction of genes are expressed at 1 cDNA copy. Note that we also scale the cDNA count by $\frac{1}{\theta}$ to yield an estimate for the number of mRNAs that were in the cell's lysate, including those that were not actually captured. This scaling step is performed mainly to facilitate comparison with spike-in derived estimates. While we do not know the capture rate θ a priori, it is a highly protocol-dependent quantity that appears to have little dependence on cell type or state. In the original study [5], we assume a value of 0.25, which is close to the lung and neuron experiments of Truetlein et al.

With an estimate of the total lysate mRNAs M_i in cell i, we simply rescale its TPM values into mRNA counts for each gene:

$$\hat{Y}_{ij} = X_{ij} \cdot \frac{M_i}{10^6} \tag{8}$$

For more details about Census, including a generative model of the single-cell RNA-seq process, and some discussion of Census's limitations, please see the original study [5]. Importantly, Census cannot control for non-linear amplification, and should therefore be considered as a simple but effective way to normalize relative expression levels so that they work better with the negative binomial distribution. Census counts should *not* be treated as absolute transcript counts.

8.9 BEAM

Monocle 2 assigns each cell a pseudotime value and a "State" encoding the segment of the trajectory it resides upon based on a trajectory learning algorithm (See below). In Monocle 2, we develop BEAM to test for branch-dependent gene expression by formulating the problem as a contrast between two negative binomial GLMs.

The null model

$$expression \sim sm.ns(Pseudotime) \tag{9}$$

for the test assumes the gene being tested is not a branch specific gene, whereas the alternative model:

$$expression \sim sm.ns(Pseudotime) + Branch + sm.ns(Pseudotime) : Branch \tag{10}$$

assumes that the gene is a branch specific gene where : represents an interaction term between branch and transformed pseudotime. Each model includes a natural spline (here with three degrees of freedom) describing smooth changes in mean expression as a function of pseudotime. The null model fits only a single curve, whereas the alternative will fit a distinct curve for each branch. Our current implementation of Monocle 2 relies on VGAM's "smart" spline fitting functionality, hence the use of the sm.ns() function instead of the more widely used ns() function from the splines package in R. Likelihood ratio testing was performed with the VGAM lrtest() function, similar to Monocle's other differential expression tests. A significant branch-dependent genes means that the gene has distinct expression dynamics along each branch, with smoothed curves that have different shapes.

To fit the full model, each cell must be assigned to the appropriate branch, which is coded through the factor "Branch" in the above model formula. Monocle's function for testing branch dependence accepts an argument specifying which branches are to be compared. These arguments are specified using the 'State' attribute assigned by Monocle during trajectory reconstructions. For example, in our analysis of the Truetlein et al data, Monocle 2 reconstructed a trajectory with two branches (L_{AT1}, L_{AT2} for AT1 and AT2 lineages, respectively), and three states (S_{BP}, S_{AT1}, S_{AT2} for progenitor, AT1, or AT2 cells). The user specifies that he or she wants to compare L_{AT1} and L_{AT2} by providing S_{AT1} and S_{AT2} as arguments to the function. Alternatively, the user can specify a branch point leading to the two states. Monocle then assigns all the cells with state S_{AT1} to branch L_{AT1} and similarly for the $AT2$ cells. However, the cells with S_{BP} must be members of both branches, because they are on the path from each branch back to the root of the tree. In order to ensure the independence of data points required for the LRT as well as the robustness and stability of our algorithm, we implemented a strategy to partition the progenitor cells into two groups, with each branch receiving a group. The groups are computed by simply ranking the progenitor cells by pseudotime and assigning the odd-numbered cells to one group and the even numbered cells to the other. We assign the first progenitor to both branches to ensure they start at the same time which is required for spline fitting involved in the test.

In order to facilitate downstream branch kinetic curve clusterin as well as branch time point detection. In the current implmentation of monocle 2, we duplicate the progenitor cells and assign it to both lineage before spline fitting. The branch plots in section *Analyzing branches in single-cell trajectories* use this method.

8.10 Branch time point detection

The branching time point for each gene can be quantified by fitting a separate spline curves for each branch from all the progenitor to each cell fate. To robustly detect the pseudotime point (t^i_β) when a gene i with a branching expression pattern starts to diverge between two cell fates L_1, L_2, we developed the branch time point detection algorithm. The algorithm starts from the end of stretched pseudotime (pseudotime $t = 10$, see the *supplementary note 1* for details) to calculate the divergence ($D_i(t = 100) = x_{L_1}(t = 100) - x_{L_2}(t = 100)$) of gene i's expression ($x_{L_1}(t = 100), x_{L_2}(t = 100)$) between two cell fates, L_1, L_2 , (for a branching gene, the divergence at this moment should be large if not the largest across pseudotime). It then moves backwards to find the latest intersection point between two fitted spline curves, which corresponds to the time when the gene starts to diverge between two branches. To add further flexibility, the algorithm moves forward to find the time point when the gene expression diverges up to a user controllable threshold (ϵ), or $D_i(t) \geq \epsilon(t)$, and defines this time point as the branch time point, t^i_β , for that particular gene i.

9 Citation

?? If you use Monocle to analyze your experiments, please cite:

```
citation("monocle")

##
##   Cole Trapnell and Davide Cacchiarelli et al (2014): The dynamics
##   and regulators of cell fate decisions are revealed by
##   pseudo-temporal ordering of single cells. Nature Biotechnology
##
## A BibTeX entry for LaTeX users is
##
##   @Article{,
##     title = {The dynamics and regulators of cell fate decisions are revealed by pseudo-temporal order
##     author = {Cole Trapnell and Davide Cacchiarelli and Jonna Grimsby and Prapti Pokharel and Shuqian
##     year = {2014},
##     journal = {Nature Biotechnology},
##   }
```

10 Acknowledgements

Monocle was originally built by Cole Trapnell and Davide Cacchiarelli, with substantial design input John Rinn and Tarjei Mikkelsen. We are grateful to Sharif Bordbar, Chris Zhu, Amy Wagers and the Broad RNAi platform for technical assistance, and Magali Soumillon for helpful discussions. Cole Trapnell is a Damon Runyon Postdoctoral Fellow. Davide Cacchiarelli is a Human Frontier Science Program Fellow. Cacchiarelli and Mikkelsen were supported by the Harvard Stem Cell Institute. John Rinn is the Alvin and Esta Star Associate Professor. This work was supported by NIH grants 1DP2OD00667, P01GM099117, and P50HG006193-01. This work was also supported in part by the Single Cell Genomics initiative, a collaboration between the Broad Institute and Fluidigm Inc.

Monocle 2 was developed by Cole Trapnell's lab. Significant portions were written by Xiaojie Qiu. The work was supported by NIH grant 1DP2HD088158 as well as an Alfred P. Sloan Foundation Research Fellowship.

This vignette was created from Wolfgang Huber's Bioconductor vignette style document, and patterned after the vignette for *DESeq*, by Simon Anders and Wolfgang Huber.

11 Session Info

```
sessionInfo()

## R version 3.3.2 (2016-10-31)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X Yosemite 10.10.5
##
## 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] splines   stats4    parallel  stats     graphics  grDevices utils
##  [8] datasets  methods   base
##
## other attached packages:
##  [1] destiny_2.0.3          monocle_2.2.0          L1Graph_0.1.0
##  [4] lpSolveAPI_5.5.2.0-17  simplePPT_0.1.0        igraph_1.1.0
##  [7] DDRTree_0.1.5          irlba_2.1.2            VGAM_1.0-3
## [10] Matrix_1.2-8           HSMMSingleCell_0.108.0 ggplot2_2.2.1
## [13] reshape2_1.4.2         Biobase_2.34.0         BiocGenerics_0.20.0
```

```
## [16] knitr_1.15.1
##
## loaded via a namespace (and not attached):
##  [1] Formula_1.2-1        assertthat_0.1       TTR_0.23-1
##  [4] sp_1.2-4             highr_0.6            latticeExtra_0.6-28
##  [7] robustbase_0.92-7    slam_0.1-40          backports_1.0.5
## [10] VIM_4.6.0            lattice_0.20-34      quantreg_5.29
## [13] limma_3.30.8         densityClust_0.2.1   RcppEigen_0.3.2.9.0
## [16] digest_0.6.11        RColorBrewer_1.1-2   checkmate_1.8.2
## [19] minqa_1.2.4          colorspace_1.3-2     fastICA_1.2-0
## [22] htmltools_0.3.5      plyr_1.8.4           pkgconfig_2.0.0
## [25] pheatmap_1.0.8       qlcMatrix_0.9.5      SparseM_1.74
## [28] scales_0.4.1         Rtsne_0.11           MatrixModels_0.4-1
## [31] lme4_1.1-12          htmlTable_1.8        tibble_1.2
## [34] proxy_0.4-16         combinat_0.0-8       mgcv_1.8-16
## [37] car_2.1-4            nnet_7.3-12          lazyeval_0.2.0
## [40] pbkrtest_0.4-6       survival_2.40-1      magrittr_1.5
## [43] evaluate_0.10        laeken_0.4.6         nlme_3.1-130
## [46] MASS_7.3-45          xts_0.9-7            foreign_0.8-67
## [49] class_7.3-14         FNN_1.1              tools_3.3.2
## [52] data.table_1.10.4    smoother_1.1         matrixStats_0.51.0
## [55] stringr_1.1.0        munsell_0.4.3        cluster_2.0.5
## [58] vcd_1.4-3            e1071_1.6-7          nloptr_1.0.4
## [61] grid_3.3.2           base64enc_0.1-3      labeling_0.3
## [64] boot_1.3-18          gtable_0.2.0         DBI_0.5-1
## [67] R6_2.2.0             zoo_1.7-14           gridExtra_2.2.1
## [70] dplyr_0.5.0          Hmisc_4.0-2          stringi_1.1.2
## [73] Rcpp_0.12.9          rpart_4.1-10         acepack_1.4.1
## [76] scatterplot3d_0.3-38 lmtest_0.9-34        DEoptimR_1.0-8
```

References

[1] Cole Trapnell, Davide Cacchiarelli, Jonna Grimsby, Prapti Pokharel, Shuqiang Li, Michael Morse, Niall J. Lennon, Kenneth J. Livak, Tarjei S. Mikkelsen, and John L. Rinn. The dynamics and regulators of cell fate decisions are revealed by pseudotemporal ordering of single cells. *Nature Biotechnology*, 2014.

[2] Cole Trapnell, Adam Roberts, Loyal Goff, Geo Pertea, Daehwan Kim, David R Kelley, Harold Pimentel, Steven L Salzberg, John L Rinn, and Lior Pachter. Differential gene and transcript expression analysis of RNA-seq experiments with TopHat and Cufflinks. *Nature Protocols*, 7(3):562-578, March 2012.

[3] Simon Anders and Wolfgang Huber. Differential expression analysis for sequence count data. *Genome Biol.*, 11(10):R106, 2010.

[4] Qi Mao, Li Wang, Steve Goodison, and Yijun Sun. Dimensionality reduction via graph structure learning. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 765-774. ACM, 2015.

[5] X Qiu, A Hill, J Packer, D Lin, YA Ma, and C Trapnell. Single-cell mrna quantification and differential analysis with census. *Nature methods*, 2017.

[6] Evan Z Macosko, Anindita Basu, Rahul Satija, James Nemesh, Karthik Shekhar, Melissa Goldman, Itay Tirosh, Allison R Bialas, Nolan Kamitaki, Emily M Martersteck, John J Trombetta, David A Weitz, Joshua R Sanes, Alex K Shalek, Aviv Regev, and Steven A McCarroll. Highly parallel genome-wide expression profiling of individual cells using nanoliter droplets. *Cell*, 161(5):1202-1214, 2015.

[7] Qi Mao, Le Yang, Li Wang, Steve Goodison, and Yijun Sun. SimplePPT: A simple principal tree algorithm. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 792-800.