Link：<https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd>

# RNA-Seq workflow: gene-level exploratory analysis and differential expression

Michael Love [1], Simon Anders [2], Wolfgang Huber [2]

[1] Department of Biostatistics, Dana-Farber Cancer Institute and Harvard School of Public Health, Boston, US;

[2] European Molecular Biology Laboratory (EMBL), Heidelberg, Germany.

Minor modifications by Martin Morgan, 26 October 2014.

set.seed(123L)

options(bitmapType="cairo")

library("BiocStyle")

BiocStyle::markdown()

options(width=100)

knitr::opts\_chunk$set(message = FALSE, error = FALSE)

## Contents

* [Counting reads](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#count)
* [Building a DESeqDataSet](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#construct)
* [Visual exploration](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#eda)
* [Differential expression](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#de)
* [Diagnostic plots](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#diagnostic)
* [Annotation](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#annotate)
* [Accounting for unknown batches](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#batch)
* [Time series experiments](https://github.com/Bioconductor/LearnBioconductor/blob/master/vignettes/B02.1.1_RNASeqLab.Rmd#time)

## Introduction

This lab will walk you through an end-to-end RNA-Seq differential expression workflow, using r Biocpkg("DESeq2") along with other Bioconductor packages. We will start from the FASTQ files, show how these were aligned to the reference genome, prepare gene expression values as a count matrix by counting the sequenced fragments, perform exploratory data analysis (EDA), perform differential gene expression analysis with DESeq2, and visually explore the results.

We note that a number of other Bioconductor packages can also be used for statistical inference of differential expression at the gene level including r Biocpkg("edgeR"), r Biocpkg("BaySeq"), r Biocpkg("DSS") and r Biocpkg("limma").

## Experimental data

The data used in this workflow is an RNA-Seq experiment of airway smooth muscle cells treated with dexamethasone, a synthetic glucocorticoid steroid with anti-inflammatory effects. Glucocorticoids are used, for example, in asthma patients to prevent or reduce inflammation of the airways. In the experiment, four primary human airway smooth muscle cell lines were treated with 1 micromolar dexamethasone for 18 hours. For each of the four cell lines, we have a treated and an untreated sample. The reference for the experiment is:

Himes BE, Jiang X, Wagner P, Hu R, Wang Q, Klanderman B, Whitaker RM, Duan Q, Lasky-Su J, Nikolos C, Jester W, Johnson M, Panettieri R Jr, Tantisira KG, Weiss ST, Lu Q. "RNA-Seq Transcriptome Profiling Identifies CRISPLD2 as a Glucocorticoid Responsive Gene that Modulates Cytokine Function in Airway Smooth Muscle Cells." PLoS One. 2014 Jun 13;9(6):e99625. PMID: [24926665](http://www.ncbi.nlm.nih.gov/pubmed/24926665). GEO: [GSE52778](http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE52778).

## Preparing count matrices

As input, the DESeq2 package expects count data as obtained, e.g., from RNA-Seq or another high-throughput sequencing experiment, in the form of a matrix of integer values. The value in the i-th row and the j-th column of the matrix tells how many reads have been mapped to gene i in sample j. Analogously, for other types of assays, the rows of the matrix might correspond e.g., to binding regions (with ChIP-Seq) or peptide sequences (with quantitative mass spectrometry).

The count values must be raw counts of sequencing reads. This is important for DESeq2's statistical model to hold, as only the actual counts allow assessing the measurement precision correctly. Hence, please do not supply other quantities, such as (rounded) normalized counts, or counts of covered base pairs -- this will only lead to nonsensical results.

## Aligning reads to a reference

The computational analysis of an RNA-Seq experiment begins earlier: what we get from the sequencing machine is a set of FASTQ files that contain the nucleotide sequence of each read and a quality score at each position. These reads must first be aligned to a reference genome or transcriptome. It is important to know if the sequencing experiment was single-end or paired-end, as the alignment software will require the user to specify both FASTQ files for a paired-end experiment. The output of this alignment step is commonly stored in a file format called [SAM/BAM](http://samtools.github.io/hts-specs).

A number of software programs exist to align reads to the reference genome, and the development is too rapid for this document to provide an up-to-date list. We recommend consulting benchmarking papers that discuss the advantages and disadvantages of each software, which include accuracy, ability to align reads over splice junctions, speed, memory footprint, and many other features.

The reads for this experiment were aligned to the Ensembl release 75 human reference genome using the [STAR read aligner](https://code.google.com/p/rna-star/):

for f in `cat files`; do STAR --genomeDir ../STAR/ENSEMBL.homo\_sapiens.release-75 \

--readFilesIn fastq/$f\\_1.fastq fastq/$f\\_2.fastq \

--runThreadN 12 --outFileNamePrefix aligned/$f.; done

[SAMtools](http://samtools.sourceforge.net/) was used to generate BAM files.

cat files | parallel -j 7 samtools view -bS aligned/{}.Aligned.out.sam -o aligned/{}.bam

The BAM files for a number of sequencing runs can then be used to generate count matrices, as described in the following section.

## Counting reads

Besides the main count matrix, which we will use later, the r Biocexptpkg("airway") package also contains a small subset of the raw data, namely eight BAM file each with a subset of the reads. We will use these files to demonstrate how a count matrix can be constructed from BAM files. Afterwards, we will load the full count matrix corresponding to all samples and all data, which is already provided in the same package, and will continue the analysis with that full table.

We load the data package with the example data:

library("airway")

The R function system.file can be used to find out where on your computer the files from a package have been installed. Here we ask for the full path to the extdata directory, which is part of the r Biocexptpkg("airway") package.

dir <- system.file("extdata", package="airway", mustWork=TRUE)

In this directory, we find the eight BAM files (and some other files):

list.files(dir)

Typically, we have a table with experimental metadata for our samples. For your own project, you might create such a comma-separated value (CSV) file using a text editor or spreadsheet software such as Excel.

We load this file with read.csv. The parentheses around the last line are used to print the result in addition to storing it to the sampleTable object.

csvfile <- file.path(dir,"sample\_table.csv")

(sampleTable <- read.csv(csvfile,row.names=1))

Once the reads have been aligned, there are a number of tools which can be used to count the number of reads which can be unambiguously assigned to genomic features for each sample. These often take as input SAM/BAM alignment files and a file specifiying the genomic features, e.g. a GFF3 or GTF file specifying the gene models.

The following tools can be used generate count matrices:

| **function** | **package** | **framework** | **output** | **DESeq2 input function** |
| --- | --- | --- | --- | --- |
| summarizeOverlaps | r Biocpkg("GenomicAlignments") | R/Bioconductor | SummarizedExperiment | DESeqDataSet |
| featureCounts | r Biocpkg("Rsubread") | R/Bioconductor | matrix | DESeqDataSetFromMatrix |
| htseq-count | [HTSeq](http://www-huber.embl.de/users/anders/HTSeq) | Python | files | DESeqDataSetFromHTSeq |

Using the Run column in the sample table, we construct the full paths to the files we want to perform the counting operation on:

filenames <- file.path(dir, paste0(sampleTable$Run, "\_subset.bam"))

We indicate in Bioconductor that these files are BAM files using the BamFileList function. Here we also specify details about how the BAM files should be treated, e.g., only process 2000000 reads at a time.

library("Rsamtools")

bamfiles <- BamFileList(filenames, yieldSize=2000000)

**Note:** make sure that the chromosome names of the genomic features in the annotation you use are consistent with the chromosome names of the reference used for read alignment. Otherwise, the scripts might fail to count any reads to features due to the mismatching names. We can check the chromosome names in the alignment files like so:

seqinfo(bamfiles[1])

Next, we need to read in the gene model which will be used for counting reads. We will read the gene model from a [GTF file](http://www.ensembl.org/info/website/upload/gff.html), using makeTranscriptDbFromGFF from the r Biocpkg("GenomicFeatures") package. GTF files can be downloaded from Ensembl's FTP site or other gene model repositories. A TranscriptDb object is a database which can be used to generate a variety of range-based objects, such as exons, transcripts, and genes. We will want to make a list of exons grouped by gene.

There are other options for constructing a TranscriptDB. For the known genes track from the UCSC Genome Browser, one can use the pre-built Transcript DataBase: r Biocannopkg("TxDb.Hsapiens.UCSC.hg19.knownGene"). The makeTranscriptDbFromBiomart function can be used to automatically pull a gene model from Biomart.

library("GenomicFeatures")

gtffile <- file.path(dir,"Homo\_sapiens.GRCh37.75\_subset.gtf")

(txdb <- makeTranscriptDbFromGFF(gtffile, format="gtf"))

The following line produces a GRangesList of all the exons grouped by gene.

(genes <- exonsBy(txdb, by="gene"))

After these preparations, the actual counting is easy. The function summarizeOverlaps from the r Biocpkg("GenomicAlignments") package will do this. This produces a SummarizedExperiment object, which contains a variety of information about an experiment, and will be described in more detail below.

**Note:** If it is desired to perform counting using multiple cores, one can use the register and MulticoreParam functions from the r Biocpkg("BiocParallel") package before the counting call below.

library("GenomicAlignments")

se <- summarizeOverlaps(features=genes, reads=bamfiles,

mode="Union",

singleEnd=FALSE,

ignore.strand=TRUE,

fragments=TRUE )

We specify a number of arguments besides the features and the reads. The mode argument describes what kind of read overlaps will be counted as a hit. These modes are shown in Figure 1 of the "Counting reads with summarizeOverlaps" vignette for the r Biocpkg("GenomicAlignments") package. Setting singleEnd to FALSE indicates that the experiment produced paired-end reads, and we want to count a pair of reads only once toward the read count for a gene. In order to produce correct counts, it is important to know if the RNA-Seq experiment was strand-specific or not. This experiment was not strand-specific so we set ignore.strand to FALSE. The fragments argument can be used when singleEnd=FALSE to specify if unpaired reads should be counted (yes if fragments=TRUE).

par(mar=c(0,0,0,0))

plot(1,1,xlim=c(0,100),ylim=c(0,100),bty="n",

type="n",xlab="",ylab="",xaxt="n",yaxt="n")

polygon(c(45,80,80,45),c(10,10,70,70),col=rgb(1,0,0,.5),border=NA)

polygon(c(45,80,80,45),c(68,68,70,70),col=rgb(1,0,0,.5),border=NA)

text(62.5,40,"assay(s)")

text(62.5,30,"e.g. 'counts'")

polygon(c(20,40,40,20),c(10,10,70,70),col=rgb(0,0,1,.5),border=NA)

polygon(c(20,40,40,20),c(68,68,70,70),col=rgb(0,0,1,.5),border=NA)

text(30,40,"rowData")

polygon(c(45,80,80,45),c(75,75,90,90),col=rgb(.5,0,.5,.5),border=NA)

polygon(c(45,47,47,45),c(75,75,90,90),col=rgb(.5,0,.5,.5),border=NA)

text(62.5,82.5,"colData")

Here we show the component parts of a SummarizedExperiment object, and also its subclasses, such as the DESeqDataSet which is explained in the next section. The assay(s) (red block) contains the matrix (or matrices) of summarized values, the rowData (blue block) contains information about the genomic ranges, and the colData (purple block) contains information about the samples or experiments. The highlighted line in each block represents the first row (note that the first row of colData lines up with the first column of the assay.

This example code above actually only counts a small subset of reads from the original experiment. Nevertheless, we can still investigate the resulting SummarizedExperiment by looking at the counts in the assay slot, the phenotypic data about the samples in colData slot (in this case an empty DataFrame), and the data about the genes in the rowData slot.

se

head(assay(se))

colSums(assay(se))

colData(se)

rowData(se)

Note that the rowData slot is a GRangesList, which contains all the information about the exons for each gene, i.e., for each row of the count matrix. It also contains metadata about the construction of the gene model in the metadata slot.

str(metadata(rowData(se)))

The colData slot, so far empty, should contain all the metadata. We hence assign our sample table to it:

(colData(se) <- DataFrame(sampleTable))

At this point, we have counted the reads which overlap the genes in the gene model we specified. This is a branching point where we could use a variety of Bioconductor packages for exploration and differential expression of the counts, including r Biocpkg("edgeR"), r Biocpkg("BaySeq"), r Biocpkg("DSS") and r Biocpkg("limma"). We will continue, using r Biocpkg("DESeq2"). The SummarizedExperiment object is all we need to start our analysis. In the following section we will show how to use it to create the data object used by r Biocpkg("DESeq2").

## The DESeqDataSet, column metadata, and the design formula

Bioconductor software packages often define and use a custom class for their data object, which makes sure that all the needed data slots are consistently provided and fulfill the requirements. In addition, Bioconductor has general data classes (such as the SummarizedExperiment) that can be used to move data between packages. In r Biocpkg("DESeq2"), the custom class is called DESeqDataSet. It is built on top of the SummarizedExperiment class (in technical term, it is a subclass), and it is easy to convert SummarizedExperiment instances into DESeqDataSet and vice versa. One of the main differences is that the assay slot is instead accessed using the count accessor, and the class enforces that the values in this matrix are non-negative integers.

A second difference is that the DESeqDataSet has an associated design formula. The experimental design is specified at the beginning of the analysis, as it will inform many of the DESeq2 functions how to treat the samples in the analysis (one exception is the size factor estimation, i.e., the adjustment for differing library sizes, which does not depend on the design formula). The design formula tells which variables in the column metadata table (colData) specify the experimental design and how these factors should be used in the analysis.

The simplest design formula for differential expression would be ~ condition, where condition is a column in colData(dds) which specifies which of two (or more groups) the samples belong to. For the airway experiment, we will specify ~ cell + dex, which means that we want to test for the effect of dexamethasone (the last factor), controlling for the effect of different cell line (the first factor).

You can use R's formula notation to express any experimental design that can be described within an ANOVA-like framework. Note that DESeq2 uses the same formula notation as, for instance, the lm function of base R. If the question of interest is whether a fold change due to treatment is different across groups, interaction terms can be included using models such as ~ group + treatment + group:treatment. See the manual page for ?results for examples of extracting contrasts from more complex designs such as these.

In the following sections, we will demonstrate the construction of the DESeqDataSet from two starting points:

* from a SummarizedExperiment object created by, e.g., summarizeOverlaps in the above example
* more generally, from a count matrix and a column metadata table which have been loaded into R

For a full example of using the HTSeq Python package for read counting, please see the r Biocexptpkg("pasilla") vignette. For an example of generating the DESeqDataSet from files produced by htseq-count, please see the r Biocpkg("DESeq2") vignette.

### Starting from SummarizedExperiment

We now use R's data command to load a prepared SummarizedExperiment that was generated from the publicly available sequencing data files associated with the Himes et al. paper, described above. The steps we used to produce this object were equivalent to those you worked through in the previous sections, except that we used all the reads and all the genes. For more details on the exact steps used to create this object type browseVignettes("airway") into your R session.

data("airway")

se <- airway

We can quickly check the millions of fragments which uniquely aligned to the genes (the second argument of round tells how many decimal points to keep).

round( colSums(assay(se)) / 1e6, 1 )

Supposing we have constructed a SummarizedExperiment using one of the methods described in the previous section, we now need to make sure that the object contains all the necessary information about the samples, i.e., a table with metadata on the count matrix's columns stored in the colData slot:

colData(se)

Here we see that this object already contains an informative colData slot -- because we have already prepared it for you, as described in the r Biocexptpkg("airway") vignette. However, when you work with your own data, you will have to add the pertinent sample / phenotypic information for the experiment at this stage. We highly recommend keeping this information in a comma-separated value (CSV) or tab-separated value (TSV) file, which can be exported from an Excel spreadsheet, and the assign this to the colData slot, making sure that the rows correspond to the columns of the SummarizedExperiment. We made sure of this correspondence by specifying the BAM files using a column of the sample table.

Once we have our fully annotated SummarizedExperiment object, we can construct a DESeqDataSet object from it, which will then form the starting point of the actual DESeq2 package, described in the following sections. We add an appropriate design for the analysis.

library("DESeq2")

dds <- DESeqDataSet(se, design = ~ cell + dex)

Note that there are two alternative functions, DESeqDataSetFromMatrix and DESeqDataSetFromHTSeq, which allow you to get started in case you have your data not in the form of a SummarizedExperiment object, but either as a simple matrix of count values or as output files from the htseq-count script from the HTSeq Python package.

Below we demonstrate using DESeqDataSetFromMatrix.

### Starting from count matrices

In this section, we will show how to build an DESeqDataSet supposing we only have a count matrix and a table of sample information.

**Note:** if you have prepared a SummarizedExperiment you should skip this section. While the previous section would be used to contruct a DESeqDataSet from a SummarizedExperiment, here we first extract the individual object (count matrix and sample info) from the SummarizedExperiment in order to build it back up into a new object -- only for demonstration purposes. In practice, the count matrix would either be read in from a file or perhaps generated by an R function like featureCounts from the r Biocpkg("Rsubread") package.

The information in a SummarizedExperiment object can be accessed with accessor functions. For example, to see the actual data, i.e., here, the read counts, we use the assay function. (The head function restricts the output to the first few lines.)

countdata <- assay(se)

head(countdata)

In this count matrix, each row represents an Ensembl gene, each column a sequenced RNA library, and the values give the raw numbers of sequencing reads that were mapped to the respective gene in each library. We also have metadata on each of the samples (the columns of the count matrix). If you've counted reads with some other software, you need to check at this step that the columns of the count matrix correspond to the rows of the column metadata.

coldata <- colData(se)

We now have all the ingredients to prepare our data object in a form that is suitable for analysis, namely:

* countMatrix: a table with the read counts
* coldata: a table with metadata on the count matrix's columns

To now construct the data object from the matrix of counts and the metadata table, we use:

(ddsMat <- DESeqDataSetFromMatrix(countData = countdata,

colData = coldata,

design = ~ cell + dex))

We will continue with the object generated from the SummarizedExperiment section.

## Visually exploring the dataset

### The rlog transformation

Many common statistical methods for exploratory analysis of multidimensional data, especially methods for clustering and ordination (e.g., principal-component analysis and the like), work best for (at least approximately) homoskedastic data; this means that the variance of an observed quantity (here, the expression strength of a gene) does not depend on the mean. In RNA-Seq data, however, variance grows with the mean. For example, if one performs PCA (principal components analysis) directly on a matrix of normalized read counts, the result typically depends only on the few most strongly expressed genes because they show the largest absolute differences between samples. A simple and often used strategy to avoid this is to take the logarithm of the normalized count values plus a small pseudocount; however, now the genes with low counts tend to dominate the results because, due to the strong Poisson noise inherent to small count values, they show the strongest relative differences between samples.

As a solution, DESeq2 offers the regularized-logarithm transformation, or rlog for short. For genes with high counts, the rlog transformation differs not much from an ordinary log2 transformation. For genes with lower counts, however, the values are shrunken towards the genes' averages across all samples. Using an empirical Bayesian prior on inter-sample differences in the form of a ridge penalty, this is done such that the rlog-transformed data are approximately homoskedastic. See the help for ?rlog for more information and options. Another transformation, the variance stabilizing transformation, is discussed alongside the rlog in the DESeq2 vignette.

**Note:** the rlog transformation is provided for applications other than differential testing. For differential testing we recommend the DESeq function applied to raw counts, as described later in this workflow, which also takes into account the dependence of the variance of counts on the mean value during the dispersion estimation step.

The function rlog returns a SummarizedExperiment object which contains the rlog-transformed values in its assay slot:

rld <- rlog(dds)

head(assay(rld))

To show the effect of the transformation, we plot the first sample against the second, first simply using the log2 function (after adding 1, to avoid taking the log of zero), and then using the rlog-transformed values. For the log2 method, we need estimate size factors to account for sequencing depth (this is done automatically for the rlog method).

par( mfrow = c( 1, 2 ) )

dds <- estimateSizeFactors(dds)

plot( log2( 1 + counts(dds, normalized=TRUE)[ , 1:2] ),

col=rgb(0,0,0,.2), pch=16, cex=0.3 )

plot( assay(rld)[ , 1:2],

col=rgb(0,0,0,.2), pch=16, cex=0.3 )

Note that, in order to make it easier to see where several points are plotted on top of each other, we set the plotting color to a semi-transparent black and changed the points to solid circles (pch=16) with reduced size (cex=0.3).

We can see how genes with low counts seem to be excessively variable on the ordinary logarithmic scale, while the rlog transform compresses differences for genes for which the data cannot provide good information anyway.

### Sample distances

A useful first step in an RNA-Seq analysis is often to assess overall similarity between samples: Which samples are similar to each other, which are different? Does this fit to the expectation from the experiment's design?

We use the R function dist to calculate the Euclidean distance between samples. To avoid that the distance measure is dominated by a few highly variable genes, and have a roughly equal contribution from all genes, we use it on the rlog-transformed data:

sampleDists <- dist( t( assay(rld) ) )

sampleDists

Note the use of the function t to transpose the data matrix. We need this because dist calculates distances between data rows and our samples constitute the columns.

We visualize the distances in a heatmap, using the function heatmap.2 from the r CRANpkg("gplots") package.

library("gplots")

library("RColorBrewer")

We have to provide a hierarchical clustering hc to the heatmap.2 function based on the sample distances, or else the heatmap.2 function would calculate a clustering based on the distances between the rows/columns of the distance matrix.

sampleDistMatrix <- as.matrix( sampleDists )

rownames(sampleDistMatrix) <- paste( rld$dex, rld$cell, sep="-" )

colors <- colorRampPalette( rev(brewer.pal(9, "Blues")) )(255)

hc <- hclust(sampleDists)

heatmap.2( sampleDistMatrix, Rowv=as.dendrogram(hc),

symm=TRUE, trace="none", col=colors,

margins=c(2,10), labCol=FALSE )

Note that we have changed the row names of the distance matrix to contain treatment type and patient number instead of sample ID, so that we have all this information in view when looking at the heatmap.

Another option for calculating sample distances is to use the Poisson Distance, implemented in the CRAN package r CRANpkg("PoiClaClu"). Similar to the transformations offered in DESeq2, this measure of dissimilarity also takes the variance structure of counts into consideration when calculating the distances between samples. The PoissonDistance function takes the original count matrix (not normalized) with samples as rows instead of columns, so we need to tranpose the counts in dds.

library("PoiClaClu")

poisd <- PoissonDistance(t(counts(dds)))

We can plot the heatmap as before:

samplePoisDistMatrix <- as.matrix( poisd$dd )

rownames(samplePoisDistMatrix) <- paste( dds$dex, dds$cell, sep="-" )

colors <- colorRampPalette( rev(brewer.pal(9, "Blues")) )(255)

hc <- hclust(poisd$dd)

heatmap.2( samplePoisDistMatrix, Rowv=as.dendrogram(hc),

symm=TRUE, trace="none", col=colors,

margins=c(2,10), labCol=FALSE )

### PCA plot

Another way to visualize sample-to-sample distances is a principal-components analysis (PCA). In this ordination method, the data points (i.e., here, the samples) are projected onto the 2D plane such that they spread out in the two directions which explain most of the differences in the data. The x-axis is the direction (or principal component) which separates the data points the most. The amount of the total variance which is contained in the direction is printed in the axis label.

plotPCA(rld, intgroup = c("dex", "cell"))

Here, we have used the function plotPCA which comes with DESeq2. The two terms specified by intgroup are the interesting groups for labelling the samples; they tell the function to use them to choose colors. We can also build the PCA plot from scratch using r CRANpkg("ggplot2"). This is done by asking the plotPCA function to return the data used for plotting rather than building the plot. See the ggplot2 [documentation](http://docs.ggplot2.org/current/) for more details on using ggplot.

(data <- plotPCA(rld, intgroup = c( "dex", "cell"), returnData=TRUE))

percentVar <- round(100 \* attr(data, "percentVar"))

We can then use this data to build up the plot, specifying that the color of the points should reflect dexamethasone treatment and the shape should reflect the cell line.

library("ggplot2")

qplot(PC1, PC2, color=dex, shape=cell, data=data) +

xlab(paste0("PC1: ",percentVar[1],"% variance")) +

ylab(paste0("PC2: ",percentVar[2],"% variance"))

From both visualizations, we see that the differences between cells are considerable, though not stronger than the differences due to treatment with dexamethasone. This shows why it will be important to account for this in differential testing by using a paired design ("paired", because each dex treated sample is paired with one untreated sample from the same cell line). We are already set up for this by using the design formula ~ cell + dex when setting up the data object in the beginning.

## MDS plot

Another plot, very similar to the PCA plot, can be made using the multidimensional scaling (MDS) function in base R. This is useful when we don't have the original data, but only a matrix of distances. Here we have the MDS plot for the distances calculated from the rlog transformed counts:

mds <- data.frame(cmdscale(sampleDistMatrix))

mds <- cbind(mds, colData(rld))

qplot(X1,X2,color=dex,shape=cell,data=mds)

And here from the PoissonDistance:

mds <- data.frame(cmdscale(samplePoisDistMatrix))

mds <- cbind(mds, colData(dds))

qplot(X1,X2,color=dex,shape=cell,data=mds)

## Differential expression analysis

It will be convenient to make sure that untrt is the first level in the dex factor, so that the default log2 fold changes are calculated as treated over untreated (by default R will chose the first alphabetical level, remember: computers don't know what to do unless you tell them). The function relevel achieves this:

dds$dex <- relevel(dds$dex, "untrt")

In addition, if you have at any point subset the columns of the DESeqDataSet you should similarly call droplevels on the factors if the subsetting has resulted in some levels having 0 samples.

### Running the pipeline

Finally, we are ready to run the differential expression pipeline. With the data object prepared, the DESeq2 analysis can now be run with a single call to the function DESeq:

dds <- DESeq(dds)

This function will print out a message for the various steps it performs. These are described in more detail in the manual page for DESeq, which can be accessed by typing ?DESeq. Briefly these are: the estimation of size factors (which control for differences in the library size of the sequencing experiments), the estimation of dispersion for each gene, and fitting a generalized linear model.

A DESeqDataSet is returned which contains all the fitted information within it, and the following section describes how to extract out results tables of interest from this object.

### Building the results table

Calling results without any arguments will extract the estimated log2 fold changes and p values for the last variable in the design formula. If there are more than 2 levels for this variable, results will extract the results table for a comparison of the last level over the first level.

(res <- results(dds))

As res is a DataFrame object, it carries metadata with information on the meaning of the columns:

mcols(res, use.names=TRUE)

The first column, baseMean, is a just the average of the normalized count values, dividing by size factors, taken over all samples. The remaining four columns refer to a specific contrast, namely the comparison of the trt level over the untrt level for the factor variable dex. See the help page for results (by typing ?results) for information on how to obtain other contrasts.

The column log2FoldChange is the effect size estimate. It tells us how much the gene's expression seems to have changed due to treatment with dexamethasone in comparison to untreated samples. This value is reported on a logarithmic scale to base 2: for example, a log2 fold change of 1.5 means that the gene's expression is increased by a multiplicative factor of $2^{1.5} \approx 2.82$.

Of course, this estimate has an uncertainty associated with it, which is available in the column lfcSE, the standard error estimate for the log2 fold change estimate. We can also express the uncertainty of a particular effect size estimate as the result of a statistical test. The purpose of a test for differential expression is to test whether the data provides sufficient evidence to conclude that this value is really different from zero. DESeq2 performs for each gene a hypothesis test to see whether evidence is sufficient to decide against the null hypothesis that there is no effect of the treatment on the gene and that the observed difference between treatment and control was merely caused by experimental variability (i.e., the type of variability that you can just as well expect between different samples in the same treatment group). As usual in statistics, the result of this test is reported as a p value, and it is found in the column pvalue. (Remember that a p value indicates the probability that a fold change as strong as the observed one, or even stronger, would be seen under the situation described by the null hypothesis.)

We can also summarize the results with the following line of code, which reports some additional information, which will be covered in later sections.

summary(res)

Note that there are many genes with differential expression due to dexamethasone treatment at the FDR level of 10%. This makes sense, as the smooth muscle cells of the airway are known to react to glucocorticoid steroids. However, there are two ways to be more strict about which set of genes are considered significant:

* lower the false discovery rate threshold (the threshold on padj in the results table)
* raise the log2 fold change threshold from 0 using the lfcThreshold argument of results. See the DESeq2 vignette for a demonstration of the use of this argument.

Sometimes a subset of the p values in res will be NA ("not available"). This is DESeq's way of reporting that all counts for this gene were zero, and hence not test was applied. In addition, p values can be assigned NA if the gene was excluded from analysis because it contained an extreme count outlier. For more information, see the outlier detection section of the vignette.

### Other comparisons

In general, the results for a comparison of any two levels of a variable can be extracted using the contrast argument to results. The user should specify three values: the name of the variable, the name of the level in the numerator, and the name of the level in the denominator. Here we extract results for the log2 of the fold change of one cell line over another:

results(dds, contrast=c("cell", "N061011", "N61311"))

If results for an interaction term are desired, the name argument of results should be used. Please see the help for the results function for more details.

### Multiple testing

Novices in high-throughput biology often assume that thresholding these p values at a low value, say 0.05, as is often done in other settings, would be appropriate -- but it is not. We briefly explain why:

There are r sum(res$pvalue < .05, na.rm=TRUE) genes with a p value below 0.05 among the r sum(!is.na(res$pvalue)) genes, for which the test succeeded in reporting a p value:

sum(res$pvalue < 0.05, na.rm=TRUE)

sum(!is.na(res$pvalue))

Now, assume for a moment that the null hypothesis is true for all genes, i.e., no gene is affected by the treatment with dexamethasone. Then, by the definition of p value, we expect up to 5% of the genes to have a p value below 0.05. This amounts to r round(sum(!is.na(res$pvalue)) \* .05 ) genes. If we just considered the list of genes with a p value below 0.05 as differentially expressed, this list should therefore be expected to contain up to r round(sum(!is.na(res$pvalue)) \* .05) / r sum(res$pvalue < .05, na.rm=TRUE) = r round(sum(!is.na(res$pvalue))\*.05 / sum(res$pvalue < .05, na.rm=TRUE) \* 100)% false positives.

DESeq2 uses the Benjamini-Hochberg (BH) adjustment as described in the base R p.adjust function; in brief, this method calculates for each gene an adjusted p value which answers the following question: if one called significant all genes with a p value less than or equal to this gene's p value threshold, what would be the fraction of false positives (the false discovery rate, FDR) among them (in the sense of the calculation outlined above)? These values, called the BH-adjusted p values, are given in the column padj of the res object.

Hence, if we consider a fraction of 10% false positives acceptable, we can consider all genes with an adjusted p value below $10% = 0.1$ as significant. How many such genes are there?

sum(res$padj < 0.1, na.rm=TRUE)

We subset the results table to these genes and then sort it by the log2 fold change estimate to get the significant genes with the strongest down-regulation.

resSig <- subset(res, padj < 0.1)

head(resSig[ order( resSig$log2FoldChange ), ])

...and with the strongest upregulation. The order function gives the indices in increasing order, so a simple way to ask for decreasing order is to add a - sign. Alternatively, you can use the argument decreasing=TRUE.

head(resSig[ order( -resSig$log2FoldChange ), ])

## Diagnostic plots

A quick way to visualize the counts for a particular gene is to use the plotCounts function, which takes as arguments the DESeqDataSet, a gene name, and the group over which to plot the counts.

topGene <- rownames(res)[which.min(res$padj)]

plotCounts(dds, gene=topGene, intgroup=c("dex"))

We can also make more customizable plots using the ggplot function from the r CRANpkg("ggplot2") package:

data <- plotCounts(dds, gene=topGene, intgroup=c("dex","cell"), returnData=TRUE)

ggplot(data, aes(x=dex, y=count, color=cell)) +

scale\_y\_log10() +

geom\_point(position=position\_jitter(width=.1,height=0))

Here we use a more structural arrangement instead of random jitter, and color by the treatment.

ggplot(data, aes(x=dex, y=count, fill=dex)) +

scale\_y\_log10() +

geom\_dotplot(binaxis="y", stackdir="center")

Note that the DESeq test actually takes into account the cell line effect, so a more detailed plot would also show the cell lines.

ggplot(data, aes(x=dex, y=count, color=cell, group=cell)) +

scale\_y\_log10() +

geom\_point() + geom\_line()

An "MA-plot" provides a useful overview for an experiment with a two-group comparison. The log2 fold change for a particular comparison is plotted on the y-axis and the average of the counts normalized by size factor is shown on the x-axis ("M" for minus, because a log ratio is equal to log minus log, and "A" for average).

plotMA(res, ylim=c(-5,5))

Each gene is represented with a dot. Genes with an adjusted $p$ value below a threshold (here 0.1, the default) are shown in red. The DESeq2 package incorporates a prior on log2 fold changes, resulting in moderated log2 fold changes from genes with low counts and highly variable counts, as can be seen by the narrowing of spread of points on the left side of the plot. This plot demonstrates that only genes with a large average normalized count contain sufficient information to yield a significant call.

We can label individual points on the MA plot as well. Here we use the with R function to plot a circle and text for a selected row of the results object. Within the with function, only the baseMean and log2FoldChange values for the selected rows of res are used.

plotMA(res, ylim=c(-5,5))

with(res[topGene, ], {

points(baseMean, log2FoldChange, col="dodgerblue", cex=2, lwd=2)

text(baseMean, log2FoldChange, topGene, pos=2, col="dodgerblue")

})

Whether a gene is called significant depends not only on its LFC but also on its within-group variability, which DESeq2 quantifies as the dispersion. For strongly expressed genes, the dispersion can be understood as a squared coefficient of variation: a dispersion value of 0.01 means that the gene's expression tends to differ by typically $\sqrt{0.01} = 10\%$ between samples of the same treatment group. For weak genes, the Poisson noise is an additional source of noise.

The function plotDispEsts visualizes DESeq2's dispersion estimates:

plotDispEsts(dds)

The black points are the dispersion estimates for each gene as obtained by considering the information from each gene separately. Unless one has many samples, these values fluctuate strongly around their true values. Therefore, we fit the red trend line, which shows the dispersions' dependence on the mean, and then shrink each gene's estimate towards the red line to obtain the final estimates (blue points) that are then used in the hypothesis test. The blue circles above the main "cloud" of points are genes which have high gene-wise dispersion estimates which are labelled as dispersion outliers. These estimates are therefore not shrunk toward the fitted trend line.

Another useful diagnostic plot is the histogram of the p values.

hist(res$pvalue, breaks=20, col="grey50", border="white")

This plot becomes a bit smoother by excluding genes with very small counts:

hist(res$pvalue[res$baseMean > 1], breaks=20, col="grey50", border="white")

## Gene clustering

In the sample distance heatmap made previously, the dendrogram at the side shows us a hierarchical clustering of the samples. Such a clustering can also be performed for the genes. Since the clustering is only relevant for genes that actually carry signal, one usually carries it out only for a subset of most highly variable genes. Here, for demonstration, let us select the 35 genes with the highest variance across samples. We will work with the rlog transformed counts:

library("genefilter")

topVarGenes <- head(order(-rowVars(assay(rld))),35)

The heatmap becomes more interesting if we do not look at absolute expression strength but rather at the amount by which each gene deviates in a specific sample from the gene's average across all samples. Hence, we center each genes' values across samples, and plot a heatmap. We provide the column side colors to help identify the treated samples (in blue) from the untreated samples (in grey).

colors <- colorRampPalette( rev(brewer.pal(9, "PuOr")) )(255)

sidecols <- c("grey","dodgerblue")[ rld$dex ]

mat <- assay(rld)[ topVarGenes, ]

mat <- mat - rowMeans(mat)

colnames(mat) <- paste0(rld$dex,"-",rld$cell)

heatmap.2(mat, trace="none", col=colors, ColSideColors=sidecols,

labRow=FALSE, mar=c(10,2), scale="row")

We can now see blocks of genes which covary across patients. Note that a set of genes at the top of the heatmap are separating the N061011 cell line from the others. At the bottom of the heatmap, we see a set of genes for which the treated samples have higher gene expression.

## Independent filtering

The MA plot highlights an important property of RNA-Seq data. For weakly expressed genes, we have no chance of seeing differential expression, because the low read counts suffer from so high Poisson noise that any biological effect is drowned in the uncertainties from the read counting. We can also show this by examining the ratio of small p values (say, less than, 0.01) for genes binned by mean normalized count:

# create bins using the quantile function

qs <- c(0, quantile(res$baseMean[res$baseMean > 0], 0:7/7))

# cut the genes into the bins

bins <- cut(res$baseMean, qs)

# rename the levels of the bins using the middle point

levels(bins) <- paste0("~",round(.5\*qs[-1] + .5\*qs[-length(qs)]))

# calculate the ratio of $p$ values less than .01 for each bin

ratios <- tapply(res$pvalue, bins, function(p) mean(p < .01, na.rm=TRUE))

# plot these ratios

barplot(ratios, xlab="mean normalized count", ylab="ratio of small p values")

At first sight, there may seem to be little benefit in filtering out these genes. After all, the test found them to be non-significant anyway. However, these genes have an influence on the multiple testing adjustment, whose performance improves if such genes are removed. By removing the weakly-expressed genes from the input to the FDR procedure, we can find more genes to be significant among those which we keep, and so improved the power of our test. This approach is known as independent filtering.

The DESeq2 software automatically performs independent filtering which maximizes the number of genes which will have adjusted p value less than a critical value (by default, alpha is set to 0.1). This automatic independent filtering is performed by, and can be controlled by, the results function. We can observe how the number of rejections changes for various cutoffs based on mean normalized count. The following optimal threshold and table of possible values is stored as an attribute of the results object.

attr(res,"filterThreshold")

plot(attr(res,"filterNumRej"),type="b",

xlab="quantiles of 'baseMean'",

ylab="number of rejections")

The term independent highlights an important caveat. Such filtering is permissible only if the filter criterion is independent of the actual test statistic. Otherwise, the filtering would invalidate the test and consequently the assumptions of the BH procedure. This is why we filtered on the average over all samples: this filter is blind to the assignment of samples to the treatment and control group and hence independent. The independent filtering software used inside DESeq2 comes from the r Biocpkg("genefilter") package, which contains a reference to a paper describing the statistical foundation for independent filtering.

## Annotation: adding gene names

Our result table only uses Ensembl gene IDs, but gene names may be more informative. Bioconductor's annotation packages help with mapping various ID schemes to each other.

We load the r Biocpkg("AnnotationDbi") package and the annotation package r Biocannopkg("org.Hs.eg.db"):

library("AnnotationDbi")

library("org.Hs.eg.db")

This is the organism annotation package ("org") for Homo sapiens ("Hs"), organized as an AnnotationDbi database package ("db"), using Entrez Gene IDs ("eg") as primary key. To get a list of all available key types, use:

columns(org.Hs.eg.db)

Converting IDs with the native functions from the AnnotationDbi package is a bit cumbersome, so we provide the following convenience function (without explaining how exactly it works):

convertIDs <- function( ids, from, to, db, ifMultiple=c("putNA", "useFirst")) {

stopifnot( inherits( db, "AnnotationDb" ) )

ifMultiple <- match.arg( ifMultiple )

suppressWarnings( selRes <- AnnotationDbi::select(

db, keys=ids, keytype=from, columns=c(from,to) ) )

if ( ifMultiple == "putNA" ) {

duplicatedIds <- selRes[ duplicated( selRes[,1] ), 1 ]

selRes <- selRes[ ! selRes[,1] %in% duplicatedIds, ]

}

return( selRes[ match( ids, selRes[,1] ), 2 ] )

}

This function takes a list of IDs as first argument and their key type as the second argument. The third argument is the key type we want to convert to, the fourth is the AnnotationDb object to use. Finally, the last argument specifies what to do if one source ID maps to several target IDs: should the function return an NA or simply the first of the multiple IDs? To convert the Ensembl IDs in the rownames of res to gene symbols and add them as a new column, we use:

res$hgnc\_symbol <- convertIDs(row.names(res), "ENSEMBL", "SYMBOL", org.Hs.eg.db)

res$entrezgene <- convertIDs(row.names(res), "ENSEMBL", "ENTREZID", org.Hs.eg.db)

Now the results have the desired external gene ids:

resOrdered <- res[order(res$pvalue),]

head(resOrdered)

## Exporting results

You can easily save the results table in a CSV file, which you can then load with a spreadsheet program such as Excel. The call to as.data.frame is necessary to convert the DataFrame object (r Biocpkg("IRanges") package) to a data.frame object which can be processed by write.csv.

write.csv(as.data.frame(resOrdered), file="results.csv")

## Plotting fold changes in genomic space

If we have used the summarizeOverlaps function to count the reads, then our DESeqDataSet object is built on top of ready-to-use Bioconductor objects specifying the genomic location of the genes. We can therefore easily plot our differential expression results in genomic space. While the results function by default outputs a DataFrame, using the format argument, we can ask for GRanges or GRangesList output.

(resGR <- results(dds, format="GRanges"))

resGR$symbol <- convertIDs(names(resGR), "ENSEMBL", "SYMBOL", org.Hs.eg.db)

We will use the r Biocpkg("Gviz") package for plotting the GRanges and associated metadata: the log fold changes due to dexamethasone treatment.

library("Gviz")

The following code chunk specifies a window of 1 million base pairs upstream and downstream from the gene with the smallest p value. We create a subset of our full results, for genes within the window which have a fold change (exclude genes with no counts). We add the gene symbol as a name, if the symbol exists or is not duplicated in our subset.

window <- resGR[topGene] + 1e6

strand(window) <- "\*"

hasLFC <- !is.na(resGR$log2FoldChange)

resGRsub <- resGR[resGR %over% window & hasLFC]

naOrDup <- is.na(resGRsub$symbol) | duplicated(resGRsub$symbol)

resGRsub$group <- ifelse(naOrDup, names(resGRsub), resGRsub$symbol)

We create a vector specifying if the genes in this subset had a low false discovery rate.

sig <- factor(ifelse(is.na(resGRsub$padj) | resGRsub$padj > .1,"notsig","sig"))

We can then plot the results using r Biocpkg("Gviz") functions. We create an axis track specifying our location in the genome, a track which will show the genes and their names, colored by significance, and a data track which will draw vertical bars showing the moderated log fold change produced by DESeq2, which we know are only large when the effect is well supported by the information in the counts.

options(ucscChromosomeNames=FALSE)

g <- GenomeAxisTrack()

a <- AnnotationTrack(resGRsub, name="gene ranges", feature=sig)

d <- DataTrack(resGRsub, data="log2FoldChange", baseline=0,

type="h", name="log2 fold change", strand="+")

plotTracks(list(g,d,a), groupAnnotation="group", notsig="lightblue", sig="pink")

## Removing hidden batch effects

Suppose we did not know that there were different cell lines involved in the experiment, only that there was treatment with dexamethasone. The cell line effect on the counts then would represent some hidden and unwanted variation which might be affecting many or all of the genes in the dataset. We can use statistical methods from the r Biocpkg("sva") package to detect such groupings of the samples, and then we can add these to the DESeqDataSet's design, in order to account for them. The SVA software uses the term surrogate variables for the estimated variables which we want to account for in our analysis.

library("sva")

Below we get a matrix of counts for which the average count across samples is larger than 1. As we described above, we are trying to recover any hidden batch effects, supposing that we do not know the cell line information. So we use a full model matrix with the \Robject{dex} variable, and a reduced, or null, model matrix with only an intercept term. Finally we specify that we want to estimate 2 surrogate variables. For more information read the help at ?svaseq.

idx <- rowMeans(counts(dds)) > 1

dat <- counts(dds)[idx,]

mod <- model.matrix(~ dex, colData(dds))

mod0 <- model.matrix(~ 1, colData(dds))

svseq <- svaseq(dat, mod, mod0, n.sv=2)

svseq$sv

Because we actually do know the cell lines, we can see how well the SVA method did at recovering these variables:

par(mfrow=c(2,1),mar=c(5,5,1,1))

stripchart(svseq$sv[,1] ~ dds$cell,vertical=TRUE)

abline(h=0)

stripchart(svseq$sv[,2] ~ dds$cell,vertical=TRUE)

abline(h=0)

Finally, in order to use SVA to remove any effect on the counts from our surrogate variables, we simply add these two as columns to the DESeqDataSet and add them to the design.

ddssva <- dds

ddssva$SV1 <- svseq$sv[,1]

ddssva$SV2 <- svseq$sv[,2]

design(ddssva) <- ~ SV1 + SV2 + dex

ddssva <- DESeq(ddssva)

head(results(ddssva), 4)

## Time series experiments

DESeq2 can be used to analyze time series experiments, for example to find those genes which react in a condition specific manner over time. Here we demontrate a basic time series analysis with the r Biocexptpkg("fission") data package, which contains gene counts for an RNA-Seq time course of fission yeast. The yeast were exposed to oxidative stress, and half of the samples contain a deletion of the gene atf21. We use a design which models the strain difference at time 0, the difference over time, and any strain-specific differences over time (the interaction term strain:minute).

library("fission")

data("fission")

ddsTC <- DESeqDataSet(fission, ~ strain + minute + strain:minute)

The following chunk performs a likelihood ratio test, where we remove the strain-specific differences over time. Genes with small p values from this test are those which, at one or more time points after time 0 showed a strain-specific effect. Note therefore that this will not give small p values to genes which moved up or down over time in the same way in both strains.

ddsTC <- DESeq(ddsTC, test="LRT", reduced = ~ strain + minute)

resTC <- results(ddsTC)

resTC$symbol <- mcols(ddsTC)$symbol

head(resTC[order(resTC$pvalue),],4)

This is just one of the tests which can be applied to time series data. Another option would be to model the counts as a smooth function of time, and to include an interaction term of the condition with the smooth function. It is possible to build such a model using spline basis functions within R.

We can plot the counts for the groups over time using r CRANpkg("ggplot2"), for the gene with the smallest p value, testing for condition-dependent time profile and accounting for differences at time 0. Keep in mind that the interaction terms are the difference between the two groups at a given time after accounting for the difference at time 0.

data <- plotCounts(ddsTC, which.min(resTC$pvalue),

intgroup=c("minute","strain"), returnData=TRUE)

ggplot(data, aes(x=minute, y=count, color=strain, group=strain)) +

geom\_point() + stat\_smooth(se=FALSE,method="loess") + scale\_y\_log10()

Wald tests for the log2 fold changes at individual time points can be investigated using the test argument to results:

resultsNames(ddsTC)

res30 <- results(ddsTC, name="strainmut.minute30", test="Wald")

res30[which.min(resTC$pvalue),]

We can further more cluster significant genes by their profiles.

betas <- coef(ddsTC)

colnames(betas)

mat <- betas[,-c(1,2)]

mat[mat < -5] <- -5

mat[mat > 5] <- 5

topGenes <- head(order(resTC$pvalue),40)

colors <- colorRampPalette( rev(brewer.pal(9, "PuOr")) )(255)

heatmap.2(mat[ topGenes, ], trace="none", dendrogram="row",

Colv=FALSE, col=colors, mar=c(12,8))

## Session information

As last part of this document, we call the function sessionInfo, which reports the version numbers of R and all the packages used in this session. It is good practice to always keep such a record as it will help to trace down what has happened in case that an R script ceases to work because the functions have been changed in a newer version of a package. The session information should also **always** be included in any emails to the [Bioconductor support site](https://support.bioconductor.org) along with all code used in the analysis.

sessionInfo()