Analyzing RNA-seq data with DESeq2

Michael I. Love, Simon Anders, and Wolfgang Huber

12 January 2017

Abstract
A basic task in the analysis of count data from RNA-seq is the detection of differentially expressed genes. The count data are presented as a table which reports, for each sample, the number of sequence fragments that have been assigned to each gene. Analogous data also arise for other assay types, including comparative ChIP-Seq, HiC, shRNA screening, mass spectrometry. An important analysis question is the quantification and statistical inference of systematic changes between conditions, as compared to within-condition variability. The package DESeq2 provides methods to test for differential expression by use of negative binomial generalized linear models; the estimates of dispersion and logarithmic fold changes incorporate data-driven prior distributions. This vignette explains the use of the package and demonstrates typical workflows. An RNA-seq workflow on the Bioconductor website covers similar material to this vignette but at a slower pace, including the generation of count matrices from FASTQ files.

Package
DESeq2 1.15.28

Contents

1 Standard workflow ................................................. 4
  1.1 Quick start .................................................. 4
  1.2 How to get help for DESeq2. ............................. 4
  1.3 Input data .................................................... 5
  1.4 Differential expression analysis .......................... 12
  1.5 Exploring and exporting results ......................... 15
  1.6 Multi-factor designs ....................................... 20

2 Data transformations and visualization ....................... 22
  2.1 Count data transformations ............................... 22
  2.2 Data quality assessment by sample clustering and visualization 27

3 Variations to the standard workflow ........................... 33
  3.1 Wald test individual steps ............................... 33
  3.2 Contrasts .................................................. 33
  3.3 Interactions ................................................. 33
# Analyzing RNA-seq data with DESeq2

- **3.4 Time-series experiments**
- **3.5 Likelihood ratio test**
- **3.6 Approach to count outliers**
- **3.7 Dispersion plot and fitting alternatives**
- **3.8 Independent filtering of results**
- **3.9 Tests of log2 fold change above or below a threshold**
- **3.10 Access to all calculated values**
- **3.11 Sample-/gene-dependent normalization factors**
- **3.12 ”Model matrix not full rank”**

## Theory behind DESeq2

- **4.1 The DESeq2 model**
- **4.2 Changes compared to DESeq**
- **4.3 Methods changes since the 2014 DESeq2 paper**
- **4.4 Count outlier detection**
- **4.5 Contrasts**
- **4.6 Expanded model matrices**
- **4.7 Independent filtering and multiple testing**

## Frequently asked questions

- **5.1 How can I get support for DESeq2?**
- **5.2 Why are some p values set to NA?**
- **5.3 How can I get unfiltered DESeq2 results?**
- **5.4 How do I use VST or rlog data for differential testing?**
- **5.5 Can I use DESeq2 to analyze paired samples?**
- **5.6 If I have multiple groups, should I run all together or split into pairs of groups?**
- **5.7 Can I run DESeq2 to contrast the levels of 100 groups?**
- **5.8 Can I use DESeq2 to analyze a dataset without replicates?**
- **5.9 How can I include a continuous covariate in the design formula?**
- **5.10 I ran a likelihood ratio test, but results() only gives me one comparison.**
- **5.11 What are the exact steps performed by DESeq()?**
- **5.12 Is there an official Galaxy tool for DESeq2?**
- **5.13 I want to benchmark DESeq2 comparing to other DE tools.**
- **5.14 I have trouble installing DESeq2 on Ubuntu/Linux.**

## Acknowledgments

**6 Acknowledgments**

**7 Session info**
1 Standard workflow

If you use DESeq2 in published research, please cite:


Other Bioconductor packages with similar aims are edgeR, limma, DSS, EBSeq, and baySeq.

1.1 Quick start

Here we show the most basic steps for a differential expression analysis. There are a variety of steps upstream of DESeq2 that result in the generation of counts or estimated counts for each sample, which we will discuss in the sections below. This code chunk assumes that you have a count matrix called `cts` and a table of sample information called `coldata`. The `design` indicates how to model the samples, here, that we want to measure the effect of the condition, controlling for batch differences. The two factor variables `batch` and `condition` should be columns of `coldata`.

```r
dds <- DESeqDataSetFromMatrix(countData = cts, colData = coldata, design= ~ batch + condition)
dds <- DESeq(dds)
res <- results(dds, contrast=c("condition","treated","control"))
```

The following starting functions will be explained below:

- If you have transcript quantification files, as produced by Salmon, Sailfish, or kallisto, you would use `DESeqDataSetFromTximport`.
- If you have htseq-count files, the first line would use `DESeqDataSetFromHTSeq`.
- If you have a `RangedSummarizedExperiment`, the first line would use `DESeqDataSet`.

1.2 How to get help for DESeq2

Any and all DESeq2 questions should be posted to the Bioconductor support site, which serves as a searchable knowledge base of questions and answers:

https://support.bioconductor.org

Posting a question and tagging with “DESeq2” will automatically send an alert to the package authors to respond on the support site. See the first question in the list of Frequently Asked Questions (FAQ) for information about how to construct an informative post.

You should not email your question to the package authors, as we will just reply that the question should be posted to the Bioconductor support site.
1.3 Input data

1.3.1 Why un-normalized counts?

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 can be assigned 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). We will list method for obtaining count matrices in sections below.

The values in the matrix should be un-normalized counts or estimated counts of sequencing reads (for single-end RNA-seq) or fragments (for paired-end RNA-seq). The RNA-seq workflow describes multiple techniques for preparing such count matrices. It is important to provide count matrices as input for DESeq2’s statistical model (Love, Huber, and Anders 2014) to hold, as only the count values allow assessing the measurement precision correctly. The DESeq2 model internally corrects for library size, so transformed or normalized values such as counts scaled by library size should not be used as input.

1.3.2 The DESeqDataSet

The object class used by the DESeq2 package to store the read counts and the intermediate estimated quantities during statistical analysis is the DESeqDataSet, which will usually be represented in the code here as an object dds.

A technical detail is that the DESeqDataSet class extends the RangedSummarizedExperiment class of the SummarizedExperiment package. The “Ranged” part refers to the fact that the rows of the assay data (here, the counts) can be associated with genomic ranges (the exons of genes). This association facilitates downstream exploration of results, making use of other Bioconductor packages’ range-based functionality (e.g. find the closest ChIP-seq peaks to the differentially expressed genes).

A DESeqDataSet object must have an associated design formula. The design formula expresses the variables which will be used in modeling. The formula should be a tilde (~) followed by the variables with plus signs between them (it will be coerced into an formula if it is not already). The design can be changed later, however then all differential analysis steps should be repeated, as the design formula is used to estimate the dispersions and to estimate the log2 fold changes of the model.

Note: In order to benefit from the default settings of the package, you should put the variable of interest at the end of the formula and make sure the control level is the first level.

We will now show 4 ways of constructing a DESeqDataSet, depending on what pipeline was used upstream of DESeq2 to generated counts or estimated counts:

1) From transcript abundance files and tximport
2) From a count matrix
3) From htseq-count files
4) From a SummarizedExperiment object
1.3.3 Transcript abundance files and tximport input

A newer and recommended pipeline is to use fast transcript abundance quantifiers upstream of DESeq2, and then to create gene-level count matrices for use with DESeq2 by importing the quantification data using the tximport package. This workflow allows users to import transcript abundance estimates from a variety of external software, including the following methods:

- **Salmon** (Patro et al. 2016)
- **Sailfish** (Patro, Mount, and Kingsford 2014)
- **kallisto** (Bray et al. 2016)
- **RSEM** (Li and Dewey 2011)

Some advantages of using the above methods for transcript abundance estimation are: (i) this approach corrects for potential changes in gene length across samples (e.g. from differential isoform usage) (Trapnell et al. 2013), (ii) some of these methods (**Salmon**, **Sailfish**, **kallisto**) are substantially faster and require less memory and disk usage compared to alignment-based methods that require creation and storage of BAM files, and (iii) it is possible to avoid discarding those fragments that can align to multiple genes with homologous sequence, thus increasing sensitivity (Robert and Watson 2015).

Full details on the motivation and methods for importing transcript level abundance and count estimates, summarizing to gene-level count matrices and producing an offset which corrects for potential changes in average transcript length across samples are described in (Soneson, Love, and Robinson 2015). Note that the tximport-to-DESeq2 approach uses estimated gene counts from the transcript abundance quantifiers, but not normalized counts.

Here, we demonstrate how to import transcript abundances and construct of a gene-level DESeqDataSet object from Salmon quant.sf files, which are stored in the tximportData package. Note that, instead of locating dir using system.file, a user would typically just provide a path, e.g. /path/to/quant/files. For a typical use, the condition information should already be present as a column of the sample table samples, while here we construct artificial condition labels for demonstration.

```r
library("tximport")
library("readr")
library("tximportData")
dir <- system.file("extdata", package="tximportData")
samples <- read.table(file.path(dir,"samples.txt"), header=TRUE)
samples$condition <- factor(rep(c("A","B"),each=3))
rownames(samples) <- samples$run
samples[,c("pop","center","run","condition")]
## pop center run condition
## ERR188297 TSI UNIGE ERR188297 A
## ERR188088 TSI UNIGE ERR188088 A
## ERR188329 TSI UNIGE ERR188329 A
## ERR188288 TSI UNIGE ERR188288 B
## ERR188021 TSI UNIGE ERR188021 B
## ERR188356 TSI UNIGE ERR188356 B
```

Next we specify the path to the files using the appropriate columns of samples, and we read in a table that links transcripts to genes for this dataset.
files <- file.path(dir,"salmon", samples$run, "quant.sf")
names(files) <- samples$run
Tx2gene <- read.csv(file.path(dir, "tx2gene.csv"))

We import the necessary quantification data for DESeq2 using the tximport function. For further details on use of tximport, including the construction of the tx2gene table for linking transcripts to genes in your dataset, please refer to the tximport package vignette.

txi <- tximport(files, type="salmon", tx2gene=tx2gene)

Finally, we can construct a DESeqDataSet from the txi object and sample information in samples.

library("DESeq2")
ddsTxi <- DESeqDataSetFromTximport(txi,
  colData = samples,
  design = ~ condition)

The ddsTxi object here can then be used as dds in the following analysis steps.

1.3.4 Count matrix input

Alternatively, the function DESeqDataSetFromMatrix can be used if you already have a matrix of read counts prepared from another source. Another method for quickly producing count matrices from alignment files is the featureCounts function (Liao, Smyth, and Shi 2013) in the Rsubread package. To use DESeqDataSetFromMatrix, the user should provide the counts matrix, the information about the samples (the columns of the count matrix) as a DataFrame or data.frame, and the design formula.

To demonstrate the use of DESeqDataSetFromMatrix, we will read in count data from the pasilla package. We read in a count matrix, which we will name cts, and the sample information table, which we will name coldata. Further below we describe how to extract these objects from, e.g. featureCounts output.

library("pasilla")
pasCts <- system.file("extdata", "pasilla_gene_counts.tsv", package="pasilla", mustWork=TRUE)
pasAnno <- system.file("extdata", "pasilla_sample_annotation.csv", package="pasilla", mustWork=TRUE)
cts <- as.matrix(read.csv(pasCts,sep="\t",row.names="gene_id"))
coldata <- read.csv(pasAnno, row.names=1)
coldata <- coldata[,c("condition","type")]

We examine the count matrix and column data to see if they are consistent:

head(cts)
##     FBgn0000003     FBgn0000008     FBgn0000014
untreated1  untreated2  untreated3  untreated4  treated1  treated2
untreated1  0         0         0         0         0         0
untreated2  92        161       76        70        140        88
untreated3  5         1         0         0         4         0
Note that these are not in the same order with respect to samples!

It is critical that the columns of the count matrix and the rows of the column data (information about samples) are in the same order. We should re-arrange one or the other so that they are consistent in terms of sample order (if we do not, later functions would produce an error). We additionally need to chop off the "fb" of the row names of coldata, so the naming is consistent.

```r
rownames(coldata) <- sub("fb", ",", rownames(coldata))
all(rownames(coldata) %in% colnames(cts))
## [1] TRUE
cnts <- cts[, rownames(coldata)]
all(rownames(coldata) == colnames(cts))
## [1] TRUE
```

If you have used the featureCounts function (Liao, Smyth, and Shi 2013) in the Rsubread package, the matrix of read counts can be directly provided from the "counts" element in the list output. The count matrix and column data can typically be read into R from flat files using base R functions such as read.csv or read.delim. For htsq-cout files, see the dedicated input function below.

With the count matrix, cts, and the sample information, coldata, we can construct a DESeqDataSet:

```r
library("DESeq2")
dds <- DESeqDataSetFromMatrix(countData = cts,
                               colData = coldata,
                               design = ~ condition)
dds
## class: DESeqDataSet
## dim: 14599 7
## metadata(1): version
## assays(1): counts
If you have additional feature data, it can be added to the \textit{DESeqDataSet} by adding to the metadata columns of a newly constructed object. (Here we add redundant data just for demonstration, as the gene names are already the rownames of the \texttt{dds}.)

```r
featureData <- data.frame(gene=rownames(cts))
mcols(dds) <- DataFrame(mcols(dds), featureData)
mcols(dds)
## DataFrame with 14599 rows and 1 column
## gene
## <factor>
## 1 FBgn0000003
## 2 FBgn0000008
## 3 FBgn0000014
## 4 FBgn0000015
## 5 FBgn0000017
## ... ...
## 14595 FBgn0261571
## 14596 FBgn0261572
## 14597 FBgn0261573
## 14598 FBgn0261574
## 14599 FBgn0261575
```

1.3.5 \textit{htseq-count} input

You can use the function \texttt{DESeqDataSetFromHTSeqCount} if you have used \textit{htseq-count} from the \texttt{HTSeq} python package (Anders, Pyl, and Huber 2014). For an example of using the python scripts, see the \texttt{pasilla} data package. First you will want to specify a variable which points to the directory in which the \textit{htseq-count} output files are located.

```r
directory <- "/path/to/your/files/
```

However, for demonstration purposes only, the following line of code points to the directory for the demo \textit{htseq-count} output files packages for the \texttt{pasilla} package.

```r
directory <- system.file("extdata", package="pasilla",
               mustWork=TRUE)
```

We specify which files to read in using \texttt{list.files}, and select those files which contain the string "treated" using \texttt{grep}. The \texttt{sub} function is used to chop up the sample filename to obtain the condition status, or you might alternatively read in a phenotypic table using \texttt{read.table}.

```r
sampleFiles <- grep("treated",list.files(directory),value=TRUE)
sampleCondition <- sub("(.+treated).*","\1",sampleFiles)
sampleTable <- data.frame(sampleName = sampleFiles,
                           fileName = sampleFiles,
                           condition = sampleCondition)
```
Then we build the `DESeqDataSet` using the following function:

```r
library("DESeq2")
ddsHTSeq <- DESeqDataSetFromHTSeqCount(sampleTable = sampleTable,
                                        directory = directory,
                                        design = ~ condition)
ddsHTSeq
```

1.3.6 **SummarizedExperiment input**

An example of the steps to produce a `RangedSummarizedExperiment` can be found in the RNA-seq workflow and in the vignette for the data package `airway`. Here we load the `RangedSummarizedExperiment` from that package in order to build a `DESeqDataSet`.

```r
library("airway")
data("airway")
se <- airway
```

The constructor function below shows the generation of a `DESeqDataSet` from a `RangedSummarizedExperiment` `se`.

```r
library("DESeq2")
ddsSE <- DESeqDataSet(se, design = ~ cell + dex)
ddsSE
```

1.3.7 **Pre-filtering**

While it is not necessary to pre-filter low count genes before running the DESeq2 functions, there are two reasons which make pre-filtering useful: by removing rows in which there are no reads or nearly no reads, we reduce the memory size of the `dds` data object and we increase the speed of the transformation and testing functions within DESeq2. Here we perform a
minimal pre-filtering to remove rows that have only 0 or 1 read. Note that more strict filtering
to increase power is automatically applied via independent filtering on the mean of normalized
counts within the results function.

```r
dds <- dds[ rowSums(counts(dds)) > 1, ]
```

1.3.8 Note on factor levels

By default, R will choose a reference level for factors based on alphabetical order. Then, if
you never tell the DESeq2 functions which level you want to compare against (e.g. which
level represents the control group), the comparisons will be based on the alphabetical order of
the levels. There are two solutions: you can either explicitly tell results which comparison to
make using the contrast argument (this will be shown later), or you can explicitly set the
factors levels. Setting the factor levels can be done in two ways, either using factor:

```r
dds$condition <- factor(dds$condition, levels=c("untreated","treated"))
```

...or using relevel, just specifying the reference level:

```r
dds$condition <- relevel(dds$condition, ref="untreated")
```

If you need to subset the columns of a DESeqData, i.e., when removing certain samples
from the analysis, it is possible that all the samples for one or more levels of a variable in
the design formula would be removed. In this case, the droplevels function can be used to
remove those levels which do not have samples in the current DESeqData:

```r
dds$condition <- droplevels(dds$condition)
```

1.3.9 Collapsing technical replicates

DESeq2 provides a function collapseReplicates which can assist in combining the counts
from technical replicates into single columns of the count matrix. The term technical
replicate implies multiple sequencing runs of the same library. You should not collapse
biological replicates using this function. See the manual page for an example of the use of
collapseReplicates.

1.3.10 About the pasilla dataset

We continue with the pasilla data constructed from the count matrix method above. This
data set is from an experiment on Drosophila melanogaster cell cultures and investigated the
effect of RNAi knock-down of the splicing factor pasilla (Brooks et al. 2011). The detailed
transcript of the production of the pasilla data is provided in the vignette of the data package
pasilla.
1.4 Differential expression analysis

The standard differential expression analysis steps are wrapped into a single function, \textit{DESeq}. The estimation steps performed by this function are described below, in the manual page for \texttt{DESeq} and in the Methods section of the DESeq2 publication (Love, Huber, and Anders 2014).

Results tables are generated using the function \textit{results}, which extracts a results table with log2 fold changes, \textit{p} values and adjusted \textit{p} values. With no arguments to \textit{results}, the results will be for the last variable in the design formula, and if this is a factor, the comparison will be the last level of this variable over the first level. Details about the comparison are printed to the console. The text, \textit{condition treated vs untreated}, tells you that the estimates are of the logarithmic fold change log2(treated/untreated).

```r
dds <- DESeq(dds)
res <- results(dds)
res
```

## log2 fold change (MLE): condition treated vs untreated
## Wald test \textit{p}-value: condition treated vs untreated
## Dataframe with 11638 rows and 6 columns

<table>
<thead>
<tr>
<th>baseMean</th>
<th>log2FoldChange</th>
<th>lfcSE</th>
<th>stat</th>
<th>\textit{p}value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBgn0000008</td>
<td>95.1446790</td>
<td>0.002151683</td>
<td>0.2238867</td>
<td>0.99233197</td>
</tr>
<tr>
<td>FBgn0000014</td>
<td>1.0565722</td>
<td>-0.496689957</td>
<td>2.1597256</td>
<td>-0.229978272</td>
</tr>
<tr>
<td>FBgn0000015</td>
<td>0.8467233</td>
<td>-1.882756713</td>
<td>2.1063362</td>
<td>-0.893853836</td>
</tr>
<tr>
<td>FBgn0000017</td>
<td>4352.5928988</td>
<td>-0.240025055</td>
<td>0.1260345</td>
<td>-1.904439437</td>
</tr>
<tr>
<td>FBgn0000018</td>
<td>418.6149305</td>
<td>-0.104798934</td>
<td>0.1482908</td>
<td>-0.706712077</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FBgn0261570</td>
<td>3208.384460</td>
<td>0.29543213</td>
<td>0.1270246</td>
<td>2.32578599</td>
</tr>
<tr>
<td>FBgn0261572</td>
<td>6.197137</td>
<td>-0.95912781</td>
<td>0.7769982</td>
<td>-1.23440151</td>
</tr>
<tr>
<td>FBgn0261573</td>
<td>2240.983986</td>
<td>0.01261611</td>
<td>0.1127225</td>
<td>0.01192186</td>
</tr>
<tr>
<td>FBgn0261574</td>
<td>4857.742672</td>
<td>0.01525741</td>
<td>0.1931199</td>
<td>0.07900487</td>
</tr>
<tr>
<td>FBgn0261575</td>
<td>10.683554</td>
<td>0.16355063</td>
<td>0.9386206</td>
<td>0.99233197</td>
</tr>
</tbody>
</table>

In previous versions of DESeq2, the \textit{DESeq} function by default would produce moderated, or shrunken, log2 fold changes through the use of the \texttt{betaPrior} argument. In version 1.16 and higher, we have split the moderation of log2 fold changes into a separate function, \texttt{lfcShrink}, for reasons described in the \textit{changes section} below.
Here we provide the dds object and the number of the coefficient we want to moderate. It is also possible to specify a contrast, instead of coef, which works the same as the contrast argument of the results function. If a results object is provided, the log2FoldChange column will be swapped out, otherwise lfcShrink returns a vector of shrunken log2 fold changes.

```r
resultsNames(dds)
## [1] "Intercept" "condition_treated_vs_untreated"
resLFC <- lfcShrink(dds, coef=2, res=res)
resLFC
## log2 fold change (MAP): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 11638 rows and 5 columns
## baseMean log2FoldChange stat pvalue padj
## <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008 95.1440790 0.001476959 0.009610592 0.99233197 0.9970815
## FBgn0000014 1.0565722 -0.011952307 -0.229978272 0.81810865 NA
## FBgn0000015 0.8467233 -0.046559241 -0.893853836 0.37140010 NA
## FBgn0000017 4352.5928988 -0.209784559 -1.904439437 0.05685298 0.2862230
## FBgn0000018 418.6149305 -0.087416357 -0.706712077 0.47974542 0.8282460
## ... ... ... ... ... ...
## FBgn0261570 3208.384460 0.25779079 2.325795899 0.02002997 0.1428209
## FBgn0261572 6.197137 -0.14722257 -1.234401515 0.21705333 0.6097343
## FBgn0261573 2240.983986 0.01131286 0.11192186 0.91088536 0.9824950
## FBgn0261574 4857.742672 0.01140563 0.0790487 0.93702875 0.9888664
## FBgn0261575 10.683554 0.01883364 0.17424573 0.86167235 0.9688434
```

The above steps should take less than 30 seconds for most analyses. For experiments with many samples (e.g. 100 samples), one can take advantage of parallelized computation. Both of the above functions have an argument parallel which if set to TRUE can be used to distribute computation across cores specified by the register function of BiocParallel. For example, the following chunk (not evaluated here), would register 4 cores, and then the two functions above, with parallel=TRUE, would split computation over these cores.

```r
library("BiocParallel")
register(MulticoreParam(4))
```

We can order our results table by the smallest adjusted p value:

```r
resOrdered <- res[order(res$padj),]
```

We can summarize some basic tallies using the summary function.

```r
summary(res)
##
## out of 11638 with nonzero total read count
## adjusted p-value < 0.1
## LFC > 0 (up) : 515, 4.4%
## LFC < 0 (down) : 537, 4.6%
## outliers [1] : 1, 0.0086%
## low counts [2] : 3159, 27%
## (mean count < 6)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```
How many adjusted p-values were less than 0.1?

```r
sum(res$padj < 0.1, na.rm=TRUE)
## [1] 1052
```

The `results` function contains a number of arguments to customize the results table which is generated. You can read about these arguments by looking up `?results`. Note that the `results` function automatically performs independent filtering based on the mean of normalized counts for each gene, optimizing the number of genes which will have an adjusted p value below a given FDR cutoff, `alpha`. Independent filtering is further discussed below. By default the argument `alpha` is set to 0.1. If the adjusted p value cutoff will be a value other than 0.1, `alpha` should be set to that value:

```r
res05 <- results(dds, alpha=0.05)
summary(res05)
##
## out of 11638 with nonzero total read count
## adjusted p-value < 0.05
## LFC > 0 (up) : 408, 3.5%
## LFC < 0 (down) : 433, 3.7%
## outliers [1] : 1, 0.0086%
## low counts [2] : 3159, 27%
## (mean count < 6)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
sum(res05$padj < 0.05, na.rm=TRUE)
## [1] 841
```

A generalization of the idea of p value filtering is to weight hypotheses to optimize power. A Bioconductor package, `IHW`, is available that implements the method of Independent Hypothesis Weighting (Ignatiadis et al. 2015). Here we show the use of `IHW` for p value adjustment of DESeq2 results. For more details, please see the vignette of the `IHW` package. Note that the `IHW` result object is stored in the metadata.

```r
library("IHW")
resIHW <- results(dds, filterFun=ihw)
summary(resIHW)
##
## out of 11638 with nonzero total read count
## adjusted p-value < 0.1
## LFC > 0 (up) : 515, 4.4%
## LFC < 0 (down) : 549, 4.7%
## outliers [1] : 1, 0.0086%
## [1] see 'cooksCutoff' argument of ?results
## [2] see metadata(res)$ihwResult on hypothesis weighting
sum(resIHW$padj < 0.1, na.rm=TRUE)
## [1] 1064
metadata(resIHW)$ihwResult
## ihwResult object with 11638 hypothesis tests
## Nominal FDR control level: 0.1
## Split into 7 bins, based on an ordinal covariate
```
If a multi-factor design is used, or if the variable in the design formula has more than two levels, the `contrast` argument of `results` can be used to extract different comparisons from the `DESeqDataSet` returned by `DESeq`. The use of the `contrast` argument is further discussed below.

For advanced users, note that all the values calculated by the DESeq2 package are stored in the `DESeqDataSet` object, and access to these values is discussed below.

### 1.5 Exploring and exporting results

#### 1.5.1 MA-plot

In DESeq2, the function `plotMA` shows the log2 fold changes attributable to a given variable over the mean of normalized counts for all the samples in the `DESeqDataSet`. Points will be colored red if the adjusted \( p \) value is less than 0.1. Points which fall out of the window are plotted as open triangles pointing either up or down.

```
plotMA(res, ylim=c(-2,2))
```

It is also useful to visualize the MA-plot for the shrunk log2 fold changes, which remove the noise associated with log2 fold changes from low count genes without requiring arbitrary filtering thresholds.

```
plotMA(resLFC, ylim=c(-2,2))
```
After calling `plotMA`, one can use the function `identify` to interactively detect the row number of individual genes by clicking on the plot. One can then recover the gene identifiers by saving the resulting indices:

```r
idx <- identify(res$baseMean, res$log2FoldChange)
rownames(res)[idx]
```

### 1.5.2 Plot counts

It can also be useful to examine the counts of reads for a single gene across the groups. A simple function for making this plot is `plotCounts`, which normalizes counts by sequencing depth and adds a pseudocount of 1/2 to allow for log scale plotting. The counts are grouped by the variables in `intgroup`, where more than one variable can be specified. Here we specify the gene which had the smallest $p$ value from the results table created above. You can select the gene to plot by rowname or by numeric index.

```r
plotCounts(dds, gene=which.min(res$padj), intgroup="condition")
```
For customized plotting, an argument `returnData` specifies that the function should only return a `data.frame` for plotting with `ggplot`.

```r
library("ggplot2")
d <- plotCounts(dds, gene=which.min(res$padj), intgroup="condition", returnData=TRUE)
ggplot(d, aes(x=condition, y=count)) +
geom_point(position=position_jitter(w=0.1,h=0)) +
scale_y_log10(breaks=c(25,100,400))
```
1.5.3 More information on results columns

Information about which variables and tests were used can be found by calling the function `mcols` on the results object.

```r
mcols(res)$description
```

## 
## [1] "mean of normalized counts for all samples"
## [2] "log2 fold change (MLE): condition treated vs untreated"
## [3] "standard error: condition treated vs untreated"
## [4] "Wald statistic: condition treated vs untreated"
## [5] "Wald test p-value: condition treated vs untreated"
## [6] "BH adjusted p-values"

For a particular gene, a log2 fold change of -1 for `condition treated vs untreated` means that the treatment induces a multiplicative change in observed gene expression level of $2^{-1} = 0.5$ compared to the untreated condition. If the variable of interest is continuous-valued, then the reported log2 fold change is per unit of change of that variable.

**Note on p-values set to NA**: some values in the results table can be set to `NA` for one of the following reasons:

- If within a row, all samples have zero counts, the `baseMean` column will be zero, and the log2 fold change estimates, `p` value and adjusted `p` value will all be set to `NA`. 
If a row contains a sample with an extreme count outlier then the \( p \) value and adjusted \( p \) value will be set to \( \text{NA} \). These outlier counts are detected by Cook’s distance. Customization of this outlier filtering and description of functionality for replacement of outlier counts and refitting is described below.

If a row is filtered by automatic independent filtering, for having a low mean normalized count, then only the adjusted \( p \) value will be set to \( \text{NA} \). Description and customization of independent filtering is described below.

### 1.5.4 Rich visualization and reporting of results

**ReportingTools.** An HTML report of the results with plots and sortable/filterable columns can be generated using the ReportingTools package on a DESeqDataSet that has been processed by the DESeq function. For a code example, see the RNA-seq differential expression vignette at the ReportingTools page, or the manual page for the publish method for the DESeqDataSet class.

**regionReport.** An HTML and PDF summary of the results with plots can also be generated using the regionReport package. The DESeq2Report function should be run on a DESeqDataSet that has been processed by the DESeq function. For more details see the manual page for DESeq2Report and an example vignette in the regionReport package.

**Glimma.** Interactive visualization of DESeq2 output, including MA-plots (also called MD-plot) can be generated using the Glimma package. See the manual page for glMDPlot.DESeqResults.

**pcaExplorer.** Interactive visualization of DESeq2 output, including PCA plots, boxplots of counts and other useful summaries can be generated using the pcaExplorer package. See the Launching the application section of the package vignette.

### 1.5.5 Exporting results to CSV files

A plain-text file of the results can be exported using the base R functions write.csv or write.delim. We suggest using a descriptive file name indicating the variable and levels which were tested.

```r
write.csv(as.data.frame(resOrdered),
          file="condition_treated_results.csv")
```

Exporting only the results which pass an adjusted \( p \) value threshold can be accomplished with the subset function, followed by the write.csv function.

```r
resSig <- subset(resOrdered, padj < 0.1)
resSig
```

| FBgn0039155 | 730.5958 | -4.619006 | 0.16872512 | -27.37593 | 5.307306e-165 |
| FBgn0025111 | 1501.4105 | 2.899863 | 0.12693550 | 22.84517 | 1.632133e-115 |
| FBgn0029167 | 3706.1165 | -2.197001 | 0.09701773 | -22.64535 | 1.550285e-113 |
| FBgn0003360 | 4343.0354 | -3.179672 | 0.14352683 | -22.15385 | 9.577104e-109 |
| FBgn0035085 | 638.2326 | -2.560409 | 0.13731558 | -18.64517 | 1.356647e-77 |
1.6 Multi-factor designs

Experiments with more than one factor influencing the counts can be analyzed using design formula that include the additional variables. By adding these to the design, one can control for additional variation in the counts. For example, if the condition samples are balanced across experimental batches, by including the batch factor to the design, one can increase the sensitivity for finding differences due to condition. There are multiple ways to analyze experiments when the additional variables are of interest and not just controlling factors (see section on interactions).

The data in the pasilla package have a condition of interest (the column condition), as well as information on the type of sequencing which was performed (the column type), as we can see below:

colData(dds)
## DataFrame with 7 rows and 3 columns
## condition type sizeFactor
## <factor> <factor> <numeric>
## treated1 treated single-read 1.6355751
## treated2 treated paired-end 0.7612698
## treated3 treated paired-end 0.8326526
## untreated1 untreated single-read 1.1382630
## untreated2 untreated single-read 1.7930004
## untreated3 untreated paired-end 0.6495470
## untreated4 untreated paired-end 0.7516892

We create a copy of the DESeqDataSet, so that we can rerun the analysis using a multi-factor design.
ddsMF <- dds

We can account for the different types of sequencing, and get a clearer picture of the differences attributable to the treatment. As `condition` is the variable of interest, we put it at the end of the formula. Thus the `results` function will by default pull the `condition` results unless `contrast` or name arguments are specified. Then we can re-run DESeq:

design(ddsMF) <- formula(~ type + condition)
ddsMF <- DESeq(ddsMF)

Again, we access the results using the `results` function.

resMF <- results(ddsMF)
head(resMF)
## log2 fold change (MLE): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 6 rows and 6 columns
## baseMean log2FoldChange lfcSE stat pvalue
## <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008 95.1440790 -0.04067393 0.2222916 -0.18297560 0.85481716
## FBgn0000014 1.0565722 -0.08498351 2.1115371 -0.04024722 0.96789603
## FBgn0000015 0.8467233 -1.86105812 2.2635706 -0.82217807 0.41097556
## FBgn0000017 4352.5928988 -0.25612969 0.1118570 -2.28979575 0.02203316
## FBgn0000018 418.6149305 -0.06468996 0.1317230 -0.49110616 0.62335136
## FBgn0000024 6.4062892 0.31109845 0.7658820 0.40619635 0.68459834
## padj
## <numeric>
## FBgn0000008 0.9504077
## FBgn0000014 NA
## FBgn0000015 NA
## FBgn0000017 0.1303866
## FBgn0000018 0.8640563
## FBgn0000024 0.8919545

It is also possible to retrieve the log2 fold changes, p values and adjusted p values of the `type` variable. The `contrast` argument of the function `results` takes a character vector of length three: the name of the variable, the name of the factor level for the numerator of the log2 ratio, and the name of the factor level for the denominator. The `contrast` argument can also take other forms, as described in the help page for `results` and below.

resMFTtype <- results(ddsMF,
contrast=c("type", "single-read", "paired-end"))
head(resMFTtype)
## log2 fold change (MLE): type single-read vs paired-end
## Wald test p-value: type single.read vs paired.end
## DataFrame with 6 rows and 6 columns
## baseMean log2FoldChange lfcSE stat pvalue
## <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008 95.1440790 -0.26225891 0.2207626 -1.1879680 0.2348460
## FBgn0000014 1.0565722 3.29057851 2.0869706 1.5767249 0.1148588
## FBgn0000015 0.8467233 -0.58154078 2.1821934 -0.2664937 0.7889590
## FBgn0000017 4352.5928988 -0.09976491 0.1117182 -0.8930049 0.3718545
## FBgn0000018 418.6149305 -0.06468996 0.1317230 -0.49110616 0.62335136
## FBgn0000024 6.4062892 0.31109845 0.7658820 0.40619635 0.68459834
## padj
## <numeric>
## FBgn0000008 0.9504077
## FBgn0000014 NA
## FBgn0000015 NA
## FBgn0000017 0.1303866
## FBgn0000018 0.8640563
## FBgn0000024 0.8919545
## Data transformations and visualization

### 2.1 Count data transformations

In order to test for differential expression, we operate on raw counts and use discrete distributions as described in the previous section on differential expression. However for other downstream analyses – e.g. for visualization or clustering – it might be useful to work with transformed versions of the count data.

Maybe the most obvious choice of transformation is the logarithm. Since count values for a gene can be zero in some conditions (and non-zero in others), some advocate the use of pseudocounts, i.e. transformations of the form:

\[ y = \log_2(n + n_0) \]

where \( n \) represents the count values and \( n_0 \) is a positive constant.

In this section, we discuss two alternative approaches that offer more theoretical justification and a rational way of choosing the parameter equivalent to \( n_0 \) above. The regularized logarithm or \textit{rlog} incorporates a prior on the sample differences (Love, Huber, and Anders 2014), and the other uses the concept of variance stabilizing transformations (VST) (Tibshirani 1988; Huber et al. 2003; Anders and Huber 2010). Both transformations produce transformed data on the log2 scale which has been normalized with respect to library size.

The point of these two transformations, the \textit{rlog} and the VST, is to remove the dependence of the variance on the mean, particularly the high variance of the logarithm of count data when the mean is low. Both \textit{rlog} and VST use the experiment-wide trend of variance over mean, in order to transform the data to remove the experiment-wide trend. Note that we do not require or desire that all the genes have exactly the same variance after transformation. Indeed, in a figure below, you will see that after the transformations the genes with the same mean do not have exactly the same standard deviations, but that the experiment-wide trend has flattened. It is those genes with row variance above the trend which will allow us to cluster samples into interesting groups.
Analyzing RNA-seq data with DESeq2 (PDF)

**Note on running time:** if you have many samples (e.g. 100s), the rlog function might take too long, and so the vst function will be a faster choice. The rlog and VST have similar properties, but the rlog requires fitting a shrinkage term for each sample and each gene which takes time. See the DESeq2 paper for more discussion on the differences (Love, Huber, and Anders 2014).

### 2.1.1 Blind dispersion estimation

The two functions, rlog and vst have an argument `blind`, for whether the transformation should be blind to the sample information specified by the design formula. When `blind` equals `TRUE` (the default), the functions will re-estimate the dispersions using only an intercept. This setting should be used in order to compare samples in a manner wholly unbiased by the information about experimental groups, for example to perform sample QA (quality assurance) as demonstrated below.

However, blind dispersion estimation is not the appropriate choice if one expects that many or the majority of genes (rows) will have large differences in counts which are explainable by the experimental design, and one wishes to transform the data for downstream analysis. In this case, using blind dispersion estimation will lead to large estimates of dispersion, as it attributes differences due to experimental design as unwanted noise, and will result in overly shrinking the transformed values towards each other. By setting `blind` to `FALSE`, the dispersions already estimated will be used to perform transformations, or if not present, they will be estimated using the current design formula. Note that only the fitted dispersion estimates from mean-dispersion trend line are used in the transformation (the global dependence of dispersion on mean for the entire experiment). So setting `blind` to `FALSE` is still for the most part not using the information about which samples were in which experimental group in applying the transformation.

### 2.1.2 Extracting transformed values

These transformation functions return an object of class `DESeqTransform` which is a subclass of `RangedSummarizedExperiment`. For ~20 samples, running on a newly created `DESeqDataSet`, `rlog` may take 30 seconds, `varianceStabilizingTransformation` may take 5 seconds, and `vst` less than 1 second (by subsetting to 1000 genes for calculating the global dispersion trend). However, the running times are shorter and more similar with `blind=FALSE` and if the function `DESeq` has already been run, because then it is not necessary to re-estimate the dispersion values. The `assay` function is used to extract the matrix of normalized values.

```r
rld <- rlog(dds, blind=FALSE)
vsd <- varianceStabilizingTransformation(dds, blind=FALSE)
vsd.fast <- vst(dds, blind=FALSE)
head(assay(rld), 3)
## treated1 treated2 treated3 untreated1 untreated2
## FBgn0000014 0.1781321 0.1489758 0.1486405 0.1997286 0.1537222
## FBgn0000015 -0.2871970 -0.2937085 -0.2939834 -0.2948680 -0.2801519
## untreated3 untreated4
## FBgn0000014 0.1495966 0.1490241
## FBgn0000015 0.1537222
```
2.1.3 Regularized log transformation

The function \texttt{rlog}, stands for \textit{regularized log}, transforming the original count data to the log2 scale by fitting a model with a term for each sample and a prior distribution on the coefficients which is estimated from the data. This is the same kind of shrinkage (sometimes referred to as regularization, or moderation) of log fold changes used by the \textit{DESeq} and \textit{nbinomWaldTest}. The resulting data contains elements defined as:

\[
\log_2(q_{ij}) = \beta_{i0} + \beta_{ij}
\]

where \(q_{ij}\) is a parameter proportional to the expected true concentration of fragments for gene \(i\) and sample \(j\) (see formula below), \(\beta_{i0}\) is an intercept which does not undergo shrinkage, and \(\beta_{ij}\) is the sample-specific effect which is shrunk toward zero based on the dispersion-mean trend over the entire dataset. The trend typically captures high dispersions for low counts, and therefore these genes exhibit higher shrinkage from the \texttt{rlog}.

Note that, as \(q_{ij}\) represents the part of the mean value \(\mu_{ij}\) after the size factor \(s_j\) has been divided out, it is clear that the \texttt{rlog} transformation inherently accounts for differences in sequencing depth. Without priors, this design matrix would lead to a non-unique solution, however the addition of a prior on non-intercept betas allows for a unique solution to be found.

2.1.4 Variance stabilizing transformation

Above, we used a parametric fit for the dispersion. In this case, the closed-form expression for the variance stabilizing transformation is used by \texttt{varianceStabilizingTransformation}, which is derived in the file \texttt{vst.pdf}, that is distributed in the package alongside this vignette. If a local fit is used (option \texttt{fitType="locfit"} to \texttt{estimateDispersions}) a numerical integration is used instead.

2.1.5 Effects of transformations on the variance

The figure below plots the standard deviation of the transformed data, across samples, against the mean, using the shifted logarithm transformation, the regularized log transformation and the variance stabilizing transformation. The shifted logarithm has elevated standard deviation in the lower count range, and the regularized log to a lesser extent, while for the variance stabilized data the standard deviation is roughly constant along the whole dynamic range.

Note that the vertical axis in such plots is the square root of the variance over all samples, so including the variance due to the experimental conditions. While a flat curve of the square root of variance over the mean may seem like the goal of such transformations, this may be unreasonable in the case of datasets with many true differences due to the experimental conditions.

```r
# this gives log2(n + 1)
ntd <- normTransform(dds)
library("vsn")
notAllZero <- (rowSums(counts(dds))>0)
meanSdPlot(assay(ntd)[notAllZero,])
```
meanSdPlot(assay(rld[notAllZero,]))
meanSdPlot(assay(vsd[notAllZero,]))
2.2 Data quality assessment by sample clustering and visualization

Data quality assessment and quality control (i.e. the removal of insufficiently good data) are essential steps of any data analysis. These steps should typically be performed very early in the analysis of a new data set, preceding or in parallel to the differential expression testing.

We define the term **quality as fitness for purpose**. Our purpose is the detection of differentially expressed genes, and we are looking in particular for samples whose experimental treatment suffered from an abnormality that renders the data points obtained from these particular samples detrimental to our purpose.

2.2.1 Heatmap of the count matrix

To explore a count matrix, it is often instructive to look at it as a heatmap. Below we show how to produce such a heatmap for various transformations of the data.

```r
library("pheatmap")
select <- order(rowMeans(counts(dds,normalized=TRUE)), decreasing=TRUE)[1:20]
```
df <- as.data.frame(colData(dds)[,c("condition","type")])
pheatmap(assay(ntd)[select,], cluster_rows=FALSE, show_rownames=FALSE, cluster_cols=FALSE, annotation_col=df)

pheatmap(assay(rld)[select,], cluster_rows=FALSE, show_rownames=FALSE, cluster_cols=FALSE, annotation_col=df)
pheatmap(assay(vsd)[select,], cluster_rows=FALSE, show_row_names=FALSE,
cluster_cols=FALSE, annotation_col=df)
2.2.2 Heatmap of the sample-to-sample distances

Another use of the transformed data is sample clustering. Here, we apply the dist function to the transpose of the transformed count matrix to get sample-to-sample distances. We could alternatively use the variance stabilized transformation here.

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

A heatmap of this distance matrix gives us an overview over similarities and dissimilarities between samples. We have to provide a hierarchical clustering `hc` to the heatmap function based on the sample distances, or else the heatmap function would calculate a clustering based on the distances between the rows/columns of the distance matrix.

```r
library("RColorBrewer")
sampleDistMatrix <- as.matrix(sampleDists)
rownames(sampleDistMatrix) <- paste(rld$condition, rld$type, sep="-")
colnames(sampleDistMatrix) <- NULL
colors <- colorRampPalette( rev(brewer.pal(9, "Blues")) )(255)
heatmap(sampleDistMatrix,
       clustering_distance_rows=sampleDists,
       clustering_distance_cols=sampleDists,
       col=colors)
```
2.2.3 Principal component plot of the samples

Related to the distance matrix is the PCA plot, which shows the samples in the 2D plane spanned by their first two principal components. This type of plot is useful for visualizing the overall effect of experimental covariates and batch effects.

```r
plotPCA(rld, intgroup=c("condition", "type"))
```
It is also possible to customize the PCA plot using the `ggplot` function.

```r
cpyaData <- plotPCA(rld, intgroup=c("condition", "type"), returnData=TRUE)
percentVar <- round(100 * attr(pcaData, "percentVar"))
ggplot(pcaData, aes(PC1, PC2, color=condition, shape=type)) +
  geom_point(size=3) +
  xlab(paste0("PC1: ", percentVar[1], "% variance")) +
  ylab(paste0("PC2: ", percentVar[2], "% variance")) +
  coord_fixed()
```
3 Variations to the standard workflow

3.1 Wald test individual steps

The function ***DESeq*** runs the following functions in order:

```r
dds <- estimateSizeFactors(dds)
dds <- estimateDispersions(dds)
dds <- nbinomWaldTest(dds)
```

3.2 Contrasts

A contrast is a linear combination of estimated log2 fold changes, which can be used to test if differences between groups are equal to zero. The simplest use case for contrasts is an experimental design containing a factor with three levels, say A, B and C. Contrasts enable the user to generate results for all 3 possible differences: log2 fold change of B vs A, of C vs A, and of C vs B. The ***contrast*** argument of ***results*** function is used to extract test results of log2 fold changes of interest, for example:

```r
results(dds, contrast=c("condition","C","B"))
```

Log2 fold changes can also be added and subtracted by providing a list to the ***contrast*** argument which has two elements: the names of the log2 fold changes to add, and the names of the log2 fold changes to subtract. The names used in the list should come from ***resultsNames(dds)***.

Alternatively, a numeric vector of the length of ***resultsNames(dds)*** can be provided, for manually specifying the linear combination of terms. Demonstrations of the use of contrasts for various designs can be found in the examples section of the help page for the ***results*** function. The mathematical formula that is used to generate the contrasts can be found below.

3.3 Interactions

Interaction terms can be added to the design formula, in order to test, for example, if the log2 fold change attributable to a given condition is different based on another factor, for example if the condition effect differs across genotype.

Many users begin to add interaction terms to the design formula, when in fact a much simpler approach would give all the results tables that are desired. We will explain this approach first, because it is much simpler to perform. If the comparisons of interest are, for example, the effect of a condition for different sets of samples, a simpler approach than adding interaction terms explicitly to the design formula is to perform the following steps:

- combine the factors of interest into a single factor with all combinations of the original factors
- change the design to include just this factor, e.g. `~ group`
Using this design is similar to adding an interaction term, in that it models multiple condition effects which can be easily extracted with `results`. Suppose we have two factors `genotype` (with values I, II, and III) and `condition` (with values A and B), and we want to extract the condition effect specifically for each genotype. We could use the following approach to obtain, e.g. the condition effect for genotype I:

```r
dds$group <- factor(paste0(dds$genotype, dds$condition))
design(dds) <- ~ group
dds <- DESeq(dds)
resultsNames(dds)
results(dds, contrast=c("group", "IB", "IA"))
```

The following two plots diagram hypothetical genotype-specific condition effects, which could be modeled with interaction terms by using a design of `~genotype + condition + genotype:condition`.

In the first plot (Gene 1), note that the condition effect is consistent across genotypes. Although condition A has a different baseline for I, II, and III, the condition effect is a log2 fold change of about 2 for each genotype. Using a model with an interaction term `genotype:condition`, the interaction terms for genotype II and genotype III will be nearly 0.

Here, the y-axis represents log2(n+1), and each group has 20 samples (black dots). A red line connects the mean of the groups within each genotype.
In the second plot (Gene 2), we can see that the condition effect is not consistent across genotype. Here the main condition effect (the effect for the reference genotype I) is again 2. However, this time the interaction terms will be around 1 for genotype II and -4 for genotype III. This is because the condition effect is higher by 1 for genotype II compared to genotype I, and lower by 4 for genotype III compared to genotype I. The condition effect for genotype II (or III) is obtained by adding the main condition effect and the interaction term for that genotype. Such a plot can be made using the `plotCounts` function as shown above.

Now we will continue to explain the use of interactions in order to test for differences in condition effects. We continue with the example of condition effects across three genotypes (I, II, and III).

The key point to remember about designs with interaction terms is that, unlike for a design ~genotype + condition, where the condition effect represents the overall effect controlling for differences due to genotype, by adding `genotype:condition`, the main condition effect only represents the effect of condition for the reference level of genotype (I, or whichever level was defined by the user as the reference level). The interaction terms `genotypeII.conditionB` and `genotypeIII.conditionB` give the difference between the condition effect for a given genotype and the condition effect for the reference genotype.

This genotype-condition interaction example is examined in further detail in Example 3 in the help page for `results`, which can be found by typing `?results`. In particular, we show how to test for differences in the condition effect across genotype, and we show how to obtain the condition effect for non-reference genotypes.
Note that for DESeq2 versions higher than 1.10, the `DESeq` function will turn off log fold change shrinkage (setting `betaPrior=FALSE`), for designs which contain an interaction term. Turning off the log fold change shrinkage allows the software to use standard model matrices (as would be produced by `model.matrix`), where the interaction coefficients are easier to interpret.

### 3.4 Time-series experiments

There are a number of ways to analyze time-series experiments, depending on the biological question of interest. In order to test for any differences over multiple time points, one can use a design including the time factor, and then test using the likelihood ratio test as described in the following section, where the time factor is removed in the reduced formula. For a control and treatment time series, one can use a design formula containing the condition factor, the time factor, and the interaction of the two. In this case, using the likelihood ratio test with a reduced model which does not contain the interaction terms will test whether the condition induces a change in gene expression at any time point after the reference level time point (time 0). An example of the later analysis is provided in our RNA-seq workflow.

### 3.5 Likelihood ratio test

DESeq2 offers two kinds of hypothesis tests: the Wald test, where we use the estimated standard error of a log2 fold change to test if it is equal to zero, and the likelihood ratio test (LRT). The LRT examines two models for the counts, a full model with a certain number of terms and a reduced model, in which some of the terms of the full model are removed. The test determines if the increased likelihood of the data using the extra terms in the full model is more than expected if those extra terms are truly zero.

The LRT is therefore useful for testing multiple terms at once, for example testing 3 or more levels of a factor at once, or all interactions between two variables. The LRT for count data is conceptually similar to an analysis of variance (ANOVA) calculation in linear regression, except that in the case of the Negative Binomial GLM, we use an analysis of deviance (ANODEV), where the deviance captures the difference in likelihood between a full and a reduced model.

The likelihood ratio test can be performed by specifying `test="LRT"` when using the `DESeq` function, and providing a reduced design formula, e.g. one in which a number of terms from `design(dds)` are removed. The degrees of freedom for the test is obtained from the difference between the number of parameters in the two models. A simple likelihood ratio test, if the full design was `~condition` would look like:

```r
dds <- DESeq(dds, test="LRT", reduced=~1)
res <- results(dds)
```

If the full design contained other variables, such as a batch variable, e.g. `~batch + condition` then the likelihood ratio test would look like:

```r
dds <- DESeq(dds, test="LRT", reduced=~batch)
res <- results(dds)
```
3.6 Approach to count outliers

RNA-seq data sometimes contain isolated instances of very large counts that are apparently unrelated to the experimental or study design, and which may be considered outliers. There are many reasons why outliers can arise, including rare technical or experimental artifacts, read mapping problems in the case of genetically differing samples, and genuine, but rare biological events. In many cases, users appear primarily interested in genes that show a consistent behavior, and this is the reason why by default, genes that are affected by such outliers are set aside by DESeq2, or if there are sufficient samples, outlier counts are replaced for model fitting. These two behaviors are described below.

The DESeq function calculates, for every gene and for every sample, a diagnostic test for outliers called Cook’s distance. Cook’s distance is a measure of how much a single sample is influencing the fitted coefficients for a gene, and a large value of Cook’s distance is intended to indicate an outlier count. The Cook’s distances are stored as a matrix available in assays(dds)["cooks"].

The results function automatically flags genes which contain a Cook’s distance above a cutoff for samples which have 3 or more replicates. The p values and adjusted p values for these genes are set to NA. At least 3 replicates are required for flagging, as it is difficult to judge which sample might be an outlier with only 2 replicates. This filtering can be turned off with results(dds, cooksCutoff=FALSE).

With many degrees of freedom – i.e., many more samples than number of parameters to be estimated – it is undesirable to remove entire genes from the analysis just because their data include a single count outlier. When there are 7 or more replicates for a given sample, the DESeq function will automatically replace counts with large Cook’s distance with the trimmed mean over all samples, scaled up by the size factor or normalization factor for that sample. This approach is conservative, it will not lead to false positives, as it replaces the outlier value with the value predicted by the null hypothesis. This outlier replacement only occurs when there are 7 or more replicates, and can be turned off with DESeq(dds, minRepliCatesForReplace=Inf).

The default Cook’s distance cutoff for the two behaviors described above depends on the sample size and number of parameters to be estimated. The default is to use the 99% quantile of the F(p,m-p) distribution (with p the number of parameters including the intercept and m number of samples). The default for gene flagging can be modified using the cooksCutoff argument to the results function. For outlier replacement, DESeq preserves the original counts in counts(dds) saving the replacement counts as a matrix named replaceCounts in assays(dds). Note that with continuous variables in the design, outlier detection and replacement is not automatically performed, as our current methods involve a robust estimation of within-group variance which does not extend easily to continuous covariates. However, users can examine the Cook’s distances in assays(dds)["cooks"], in order to perform manual visualization and filtering if necessary.

Note on many outliers: if there are very many outliers (e.g. many hundreds or thousands) reported by summary(res), one might consider further exploration to see if a single sample or a few samples should be removed due to low quality. The automatic outlier filtering/replacement is most useful in situations which the number of outliers is limited. When there are thousands of reported outliers, it might make more sense to turn off the outlier filtering/replacement (DESeq with minRepliCatesForReplace=Inf and results with cooksCutoff=FALSE) and perform manual inspection: First it would be advantageous to make a PCA plot as described above to spot individual sample outliers; Second, one can make a boxplot of the Cook’s distances to see if one sample is consistently higher than others (here this is not the case):
3.7 Dispersion plot and fitting alternatives

Plotting the dispersion estimates is a useful diagnostic. The dispersion plot below is typical, with the final estimates shrunk from the gene-wise estimates towards the fitted estimates. Some gene-wise estimates are flagged as outliers and not shrunk towards the fitted value, (this outlier detection is described in the manual page for `estimateDispersionsMAP`). The amount of shrinkage can be more or less than seen here, depending on the sample size, the number of coefficients, the row mean and the variability of the gene-wise estimates.

```R
plotDispEsts(dds)
```
3.7.1 Local or mean dispersion fit

A local smoothed dispersion fit is automatically substituted in the case that the parametric curve doesn’t fit the observed dispersion mean relationship. This can be prespecified by providing the argument `fitType="local"` to either `DESeq` or `estimateDispersions`. Additionally, using the mean of gene-wise dispersion estimates as the fitted value can be specified by providing the argument `fitType="mean"`.

3.7.2 Supply a custom dispersion fit

Any fitted values can be provided during dispersion estimation, using the lower-level functions described in the manual page for `estimateDispersionsGeneEst`. In the code chunk below, we store the gene-wise estimates which were already calculated and saved in the metadata column `dispGeneEst`. Then we calculate the median value of the dispersion estimates above a threshold, and save these values as the fitted dispersions, using the replacement function for `dispersionFunction`. In the last line, the function `estimateDispersionsMAP`, uses the fitted dispersions to generate maximum a posteriori (MAP) estimates of dispersion.

```r
ddsCustom <- dds
useForMedian <- mcols(ddsCustom)$dispGeneEst > 1e-7
medianDisp <- median(mcols(ddsCustom)$dispGeneEst[useForMedian], na.rm=TRUE)
dispersionFunction(ddsCustom) <- function(mu) medianDisp
ddsCustom <- estimateDispersionsMAP(ddsCustom)
```
3.8 Independent filtering of results

The `results` function of the DESeq2 package performs independent filtering by default using the mean of normalized counts as a filter statistic. A threshold on the filter statistic is found which optimizes the number of adjusted $p$ values lower than a significance level $\alpha$ (we use the standard variable name for significance level, though it is unrelated to the dispersion parameter $\alpha$). The theory behind independent filtering is discussed in greater detail below. The adjusted $p$ values for the genes which do not pass the filter threshold are set to $\text{NA}$.

The default independent filtering is performed using the `filtered_p` function of the `genefilter` package, and all of the arguments of `filtered_p` can be passed to the `results` function. The filter threshold value and the number of rejections at each quantile of the filter statistic are available as metadata of the object returned by `results`.

For example, we can visualize the optimization by plotting the `filterNumRej` attribute of the `results` object. The `results` function maximizes the number of rejections (adjusted $p$ value less than a significance level), over the quantiles of a filter statistic (the mean of normalized counts). The threshold chosen (vertical line) is the lowest quantile of the filter for which the number of rejections is within 1 residual standard deviation to the peak of a curve fit to the number of rejections over the filter quantiles:

```r
metadata(res)$alpha
## [1] 0.1
metadata(res)$filterThreshold
## 27.14286%
## 5.561712
plot(metadata(res)$filterNumRej,
     type="b", ylab="number of rejections",
     xlab="quantiles of filter")
lines(metadata(res)$lo.fit, col="red")
abline(v=metadata(res)$filterTheta)
```
Independent filtering can be turned off by setting `independentFiltering` to `FALSE`.

```r
resNoFilt <- results(dds, independentFiltering=FALSE)
addmargins(table(filtering=(res$padj < .1),
              noFiltering=(resNoFilt$padj < .1)))
```

### 3.9 Tests of log2 fold change above or below a threshold

It is also possible to provide thresholds for constructing Wald tests of significance. Two arguments to the `results` function allow for threshold-based Wald tests: `lfcThreshold`, which takes a numeric of a non-negative threshold value, and `altHypothesis`, which specifies the kind of test. Note that the `alternative hypothesis` is specified by the user, i.e., those genes which the user is interested in finding, and the test provides `p` values for the null hypothesis, the complement of the set defined by the alternative. The `altHypothesis` argument can take one of the following four values, where \( \beta \) is the log2 fold change specified by the `name` argument, and \( x \) is the `lfcThreshold`.

- `greaterAbs - |\beta| > x` - tests are two-tailed
- `lessAbs - |\beta| < x` - `p` values are the maximum of the upper and lower tests
- `greater - \beta > x`
- `less - \beta < -x`
The test `altHypothesis="lessAbs"` requires that the user have run `DESeq` with the argument `betaPrior=FALSE`. To understand the reason for this requirement, consider that during hypothesis testing, the null hypothesis is favored unless the data provide strong evidence to reject the null. For this test, including a zero-centered prior on log fold change would favor the alternative hypothesis, shrinking log fold changes toward zero. Removing the prior on log fold changes for tests of small log fold change allows for detection of only those genes where the data alone provides evidence against the null.

The four possible values of `altHypothesis` are demonstrated in the following code and visually by MA-plots in the following figures. First we run `DESeq` and specify `betaPrior=FALSE` in order to demonstrate `altHypothesis="lessAbs"`.

```r
ddsNoPrior <- DESeq(dds, betaPrior=FALSE)
```

In order to produce results tables for the following tests, the same arguments (except `ylim`) would be provided to the `results` function.

```r
par(mfrow=c(2,2),mar=c(2,2,1,1))
yl <- c(-2.5,2.5)
resGA <- results(dds, lfcThreshold=.5, altHypothesis="greaterAbs")
resLA <- results(ddsNoPrior, lfcThreshold=.5, altHypothesis="lessAbs")
resG <- results(dds, lfcThreshold=.5, altHypothesis="greater")
resL <- results(dds, lfcThreshold=.5, altHypothesis="less")
plotMA(resGA, ylim=yl)
abline(h=c(-.5,.5),col="dodgerblue",lwd=2)
plotMA(resLA, ylim=yl)
abline(h=c(-.5,.5),col="dodgerblue",lwd=2)
plotMA(resG, ylim=yl)
abline(h=.5,col="dodgerblue",lwd=2)
plotMA(resL, ylim=yl)
abline(h=-.5,col="dodgerblue",lwd=2)
```
3.10 Access to all calculated values

All row-wise calculated values (intermediate dispersion calculations, coefficients, standard errors, etc.) are stored in the `DESeqDataSet` object, e.g. `dds` in this vignette. These values are accessible by calling `mcols` on `dds`. Descriptions of the columns are accessible by two calls to `mcols`. Note that the call to `substr` below is only for display purposes.

```r
mcols(dds, use.names=TRUE)[1:4,1:4]
## DataFrame with 4 rows and 4 columns
## gene    baseMean   baseVar  allZero
## <factor> <numeric> <numeric> <logical>
## FBgn0000008 FBgn0000008 95.1440790 2.246236e+02 FALSE
## FBgn0000014 FBgn0000014 1.0565722 2.962193e+00 FALSE
## FBgn0000015 FBgn0000015 0.8467233 1.008136e+00 FALSE
## FBgn0000017 FBgn0000017 4352.5928988 3.616417e+05 FALSE
substr(names(mcols(dds)),1,10)
## [1] "gene" "baseMean" "baseVar" "allZero" "dispGeneEs"
## [6] "dispFit" "dispersion" "dispIter" "dispOutlie" "dispMAP"
## [11] "Intercept" "condition" "SE_Interce" "SE_condition" "WaldStatis"
## [16] "WaldStatis" "WaldPvalue" "WaldPvalue" "betaConv" "betaIter"
## [21] "deviance" "maxCooks"
```
The mean values $\mu_{ij} = s_j q_{ij}$ and the Cook’s distances for each gene and sample are stored as matrices in the assays slot:

```r
head(assays(dds)[["mu"]])
```

```r
  ## treated1 treated2 treated3 untreated1 untreated2
  ## FBgn0000008 154.3987025 71.8640585 78.6026192 107.292174 169.007435
  ## FBgn0000014 1.4994497 0.6979109 0.7633527 1.472389 2.319318
  ## FBgn0000015 0.5665956 0.2637189 0.2884474 1.454158 2.290600
  ## FBgn0000017 6450.3164679 3002.2656444 3283.7825771 5301.611319 8351.137808
  ## FBgn0000018 658.3598354 306.4300993 335.1634866 492.700578 776.105635
  ## FBgn0000024 11.4502130 5.3294410 5.8291729 6.805336 10.845833
    ## untreat3 untreat4
    ## FBgn0000008 61.2260212 70.8539000
    ## FBgn0000014 0.8402153 0.9723404
    ## FBgn0000015 0.8298117 0.9603008
    ## FBgn0000017 3025.3517513 3501.0926097
    ## FBgn0000018 281.1583999 325.3709575
    ## FBgn0000024 3.9291004 4.5469570
head(assays(dds)[["cooks"]])
```

```r
  ## treated1 treated2 treated3 untreated1 untreated2
  ## FBgn0000008 0.08830715 0.303802564 0.077771958 0.09822247 0.0136367583
  ## FBgn0000014 1.88689792 0.218390054 0.251831283 1.883137649
  ## FBgn0000015 0.00929777 0.072800933 0.077912313 0.12891986 0.0028137530
  ## FBgn0000017 0.01373552 0.004963502 0.002162421 0.08041054 0.016072921
  ## FBgn0000018 0.09518037 0.004725965 0.054717320 0.18464143 0.002810730
  ## FBgn0000024 0.06630034 0.131135115 0.031232734 0.27067340 0.0004894252
    ## untreat3 untreat4
    ## FBgn0000008 0.18898455 0.0005301336
    ## FBgn0000014 0.15308104 0.189019982
    ## FBgn0000015 0.00324020 0.184780154
    ## FBgn0000017 0.17264780 0.0550852696
    ## FBgn0000018 0.07648755 0.0108795118
    ## FBgn0000024 0.03105357 0.0814894716
```

The dispersions $\alpha_i$ can be accessed with the `dispersions` function.

```r
head(dispersions(dds))
```

```r
  ## [1] 0.03040956 2.86301787 2.20957889 0.01283362 0.01560434 0.23856732
head(mcols(dds)$dispersion)
```

```r
  ## [1] 0.03040956 2.86301787 2.20957889 0.01283362 0.01560434 0.23856732
```

The size factors $s_j$ are accessible via `sizeFactors`:
Analyzing RNA-seq data with DESeq2 (PDF)

```
sizeFactors(dds)
##   treated1   treated2   treated3   untreated1   untreated2   untreated3   untreated4
## 1.6355751 0.7612698 0.8326526  1.1382630  1.7930004  0.6495470  0.7516892
```

For advanced users, we also include a convenience function `coef` for extracting the matrix $[\beta_i]$, for all genes $i$ and model coefficients $r$. This function can also return a matrix of standard errors, see `?coef`. The columns of this matrix correspond to the effects returned by `resultsNames`. Note that the `results` function is best for building results tables with $p$ values and adjusted $p$ values.

```
head(coef(dds))
##   Intercept condition_treated_vs_untreated
## FBgn0000008  6.5585671 0.002151683
## FBgn0000014  0.3713251 -0.496689957
## FBgn0000015  0.3533500 -1.882756713
## FBgn0000017 12.1853813 -0.240025055
## FBgn0000018  8.7577334 -0.104798934
## FBgn0000024  2.5966931  0.210811388
```

The beta prior variance $\sigma_r^2$ is stored as an attribute of the `DESeqDataSet`:

```
attr(dds, "betaPriorVar")
## [1] 1e+06 1e+06
```

The dispersion prior variance $\sigma_d^2$ is stored as an attribute of the dispersion function:

```
dispersionFunction(dds)
## function (q)
## coefs[1] + coefs[2]/q
## <bytecode: 0x10676b08>
## <environment: 0x17cff520>
## attr(,"coefficients")
## asymptDisp extraPois
## 0.01396112 2.72102337
## attr(,"fitType")
## [1] "parametric"
## attr(,"varLogDispEsts")
## [1] 0.9891644
## attr(,"dispPriorVar")
## [1] 0.4988066
attr(dispersionFunction(dds), "dispPriorVar")
## [1] 0.4988066
```

The version of DESeq2 which was used to construct the `DESeqDataSet` object, or the version used when `DESeq` was run, is stored here:

```
metadata(dds)[["version"]]
## [1] '1.15.28'
```
3.11 Sample-/gene-dependent normalization factors

In some experiments, there might be gene-dependent dependencies which vary across samples. For instance, GC-content bias or length bias might vary across samples coming from different labs or processed at different times. We use the terms normalization factors for a gene \times sample matrix, and size factors for a single number per sample. Incorporating normalization factors, the mean parameter $\mu_{ij}$ becomes:

$$\mu_{ij} = NF_{ij}q_{ij}$$

with normalization factor matrix $NF$ having the same dimensions as the counts matrix $K$. This matrix can be incorporated as shown below. We recommend providing a matrix with row-wise geometric means of 1, so that the mean of normalized counts for a gene is close to the mean of the unnormalized counts. This can be accomplished by dividing out the current row geometric means.

```r
normFactors <- normFactors / exp(rowMeans(log(normFactors)))
normalizationFactors(dds) <- normFactors
```

These steps then replace `estimateSizeFactors` which occurs within the `DESeq` function. The `DESeq` function will look for pre-existing normalization factors and use these in the place of size factors (and a message will be printed confirming this).

The methods provided by the `cqn` or `EDASeq` packages can help correct for GC or length biases. They both describe in their vignettes how to create matrices which can be used by DESeq2. From the formula above, we see that normalization factors should be on the scale of the counts, like size factors, and unlike offsets which are typically on the scale of the predictors (i.e. the logarithmic scale for the negative binomial GLM). At the time of writing, the transformation from the matrices provided by these packages should be:

```r
cqnOffset <- cqnObject$glm.offset
cqnNormFactors <- exp(cqnOffset)
EDASeqNormFactors <- exp(-1 * EDASeqOffset)
```

3.12 “Model matrix not full rank”

While most experimental designs run easily using design formula, some design formulas can cause problems and result in the `DESeq` function returning an error with the text: “the model matrix is not full rank, so the model cannot be fit as specified.” There are two main reasons for this problem: either one or more columns in the model matrix are linear combinations of other columns, or there are levels of factors or combinations of levels of multiple factors which are missing samples. We address these two problems below and discuss possible solutions:

3.12.1 Linear combinations

The simplest case is the linear combination, or linear dependency problem, when two variables contain exactly the same information, such as in the following sample table. The software cannot fit an effect for `batch` and `condition`, because they produce identical columns in the model matrix. This is also referred to as perfect confounding. A unique solution of coefficients (the $\beta_i$ in the formula below) is not possible.
Analyzing RNA-seq data with DESeq2 (PDF)

## batch condition
## 1 1 A
## 2 1 A
## 3 2 B
## 4 2 B

Another situation which will cause problems is when the variables are not identical, but one variable can be formed by the combination of other factor levels. In the following example, the effect of batch 2 vs 1 cannot be fit because it is identical to a column in the model matrix which represents the condition C vs A effect.

## batch condition
## 1 1 A
## 2 1 A
## 3 1 B
## 4 1 B
## 5 2 C
## 6 2 C

In both of these cases above, the batch effect cannot be fit and must be removed from the model formula. There is just no way to tell apart the condition effects and the batch effects. The options are either to assume there is no batch effect (which we know is highly unlikely given the literature on batch effects in sequencing datasets) or to repeat the experiment and properly balance the conditions across batches. A balanced design would look like:

## batch condition
## 1 1 A
## 2 1 B
## 3 1 C
## 4 2 A
## 5 2 B
## 6 2 C

Finally, there is a case where we can in fact perform inference. Consider an experiment with grouped individuals, where we seek to test the group-specific effect of a treatment, while controlling for individual effects. A simple example of such a design is:

```r
(coldata <- data.frame(grp=factor(rep(c("X","Y"),each=4)),
                       ind=factor(rep(1:4,each=2)),
                       cnd=factor(rep(c("A","B"),4))))
```

## grp ind cnd
## 1 X 1 A
## 2 X 1 B
## 3 X 2 A
## 4 X 2 B
## 5 Y 3 A
## 6 Y 3 B
## 7 Y 4 A
## 8 Y 4 B
This design can be analyzed by DESeq2 but requires a bit of refactoring in order to fit the model terms. Here we will use a trick described in the edgeR user guide, from the section Comparisons Both Between and Within Subjects. If we try to analyze with a formula such as, \(~\text{ind} + \text{grp}\times\text{cnd}\), we will obtain an error, because the effect for group is a linear combination of the individuals.

However, the following steps allow for an analysis of group-specific condition effects, while controlling for differences in individual. For object construction, you can use a blank design, such as \(~\text{1}\), as long as you remember to replace it before running DESeq. Then add a column \text{ind.n} which distinguishes the individuals nested within a group. Here, we add this column to coldata, but in practice you would add this column to dds.

```r
coldata$ind.n <- factor(rep(rep(1:2,each=2),2))
coldata
#>  ind cnd ind.n
#> 1  1 A   1
#> 2  1 B   1
#> 3  2 A   2
#> 4  2 B   2
#> 5  3 A   1
#> 6  3 B   1
#> 7  4 A   2
#> 8  4 B   2
```

Now we can reassign our DESeqDataSet a design of \(~\text{grp} + \text{grp}\cdot\text{ind.n} + \text{grp}\cdot\text{cnd}\), before we call DESeq. This new design will result in the following model matrix:

```r
model.matrix(~ grp + grp:ind.n + grp:cnd, coldata)
#> (Intercept) grpY grpX:ind.n2 grpY:ind.n2 grpX:cndB grpY:cndB
#> 1 1 0 0 0 0 0
#> 2 1 0 0 0 1 0
#> 3 1 0 1 0 0 0
#> 4 1 0 1 0 1 0
#> 5 1 1 0 0 0 0
#> 6 1 1 0 0 0 1
#> 7 1 1 0 1 0 0
#> 8 1 1 0 1 0 1
#> attr("assign")
#>[1] 0 1 2 2 3 3
#> attr("contrasts")
#> attr("contrasts")$grp
#>[1] "contr.treatment"
#> attr("contrasts")$ind.n
#>[1] "contr.treatment"
#> attr("contrasts")$cnd
#>[1] "contr.treatment"
```

where the terms \text{grpX.cndB} and \text{grpY.cndB} give the group-specific condition effects. These can be extracted using results with the name argument. Furthermore, \text{grpX.cndB} and \text{grpY.cndB} can be contrasted using the contrast argument, in order to test if the condition effect is different across group:
3.12.2 Levels without samples

The base R function for creating model matrices will produce a column of zeros if a level is missing from a factor or a combination of levels is missing from an interaction of factors. The solution to the first case is to call `droplevels` on the column, which will remove levels without samples. This was shown in the beginning of this vignette.

The second case is also solvable, by manually editing the model matrix, and then providing this to `DESeq`. Here we construct an example dataset to illustrate:

```r
group <- factor(rep(1:3,each=6))
condition <- factor(rep(rep(c("A","B","C"),each=2),3))
d <- data.frame(group, condition)[-c(17,18),]
d
```

Note that if we try to estimate all interaction terms, we introduce a column with all zeros, as there are no condition C samples for group 3. (Here, `unname` is used to display the matrix concisely.)

```r
m1 <- model.matrix(~ condition*group, d)
colnames(m1)
```

```
## [1] "(Intercept)"   "conditionB"   "conditionC"
## [4] "group2"       "group3"       "conditionB:group2"
## [7] "conditionC:group2" "conditionB:group3" "conditionC:group3"
```

```r
unname(m1)
```

```
## [1,] 1  0  0  0  0  0  0  0
## [2,] 1  0  0  0  0  0  0  0
## [3,] 1  1  0  0  0  0  0  0
## [4,] 1  1  0  0  0  0  0  0
## [5,] 1  0  1  0  0  0  0  0
## [6,] 1  0  1  0  0  0  0  0
```
We can remove this column like so:

```r
m1 <- m1[, -9]
unname(m1)
```

Now this matrix `m1` can be provided to the `full` argument of `DESeq`. For a likelihood ratio test of interactions, a model matrix using a reduced design such as `~ condition + group` can be given to the `reduced` argument. Wald tests can also be generated instead of the likelihood ratio test, but for user-supplied model matrices, the argument `betaPrior` must be set to `FALSE`. 

```r
# [7,] 1 0 0 1 0 0 0 0 0
# [8,] 1 0 0 1 0 0 0 0 0
# [9,] 1 1 0 1 0 1 0 0 0
# [10,] 1 1 0 1 0 1 0 0 0
# [11,] 1 0 1 1 0 0 1 0 0
# [12,] 1 0 1 1 0 0 1 0 0
# [13,] 1 0 0 0 1 0 0 0 0
# [14,] 1 0 0 0 1 0 0 0 0
# [15,] 1 1 0 0 1 0 0 1 0
# [16,] 1 1 0 0 1 0 0 1 0
## attr("assign")
## [1] 0 1 1 2 2 3 3 3 3
## attr("contrasts")
## attr("contrasts")$condition
## [1] "contr.treatment"
## attr("contrasts")$group
## [1] "contr.treatment"
```
4 Theory behind DESeq2

4.1 The DESeq2 model

The DESeq2 model and all the steps taken in the software are described in detail in our publication (Love, Huber, and Anders 2014), and we include the formula and descriptions in this section as well. The differential expression analysis in DESeq2 uses a generalized linear model of the form:

\[ K_{ij} \sim NB(\mu_{ij}, \alpha_i) \]

\[ \mu_{ij} = s_j q_{ij} \]

\[ \log_2(q_{ij}) = x_j \beta_i \]

where counts \( K_{ij} \) for gene \( i \), sample \( j \) are modeled using a negative binomial distribution with fitted mean \( \mu_{ij} \) and a gene-specific dispersion parameter \( \alpha_i \). The fitted mean is composed of a sample-specific size factor \( s_j \) and a parameter \( q_{ij} \) proportional to the expected true concentration of fragments for sample \( j \). The coefficients \( \beta_i \) give the log2 fold changes for gene \( i \) for each column of the model matrix \( X \). Note that the model can be generalized to use sample- and gene-dependent normalization factors \( s_{ij} \).

The dispersion parameter \( \alpha_i \) defines the relationship between the variance of the observed count and its mean value. In other words, how far do we expect the observed count will be from the mean value, which depends both on the size factor \( s_j \) and the covariate-dependent part \( q_{ij} \) as defined above.

\[ \text{Var}(K_{ij}) = E[(K_{ij} - \mu_{ij})^2] = \mu_{ij} + \alpha_i \mu_{ij}^2 \]

An option in DESeq2 is to provide maximum a posteriori estimates of the log2 fold changes in \( \beta_i \) after incorporating a zero-centered Normal prior (betaPrior). While previously, these moderated, or shrunk, estimates were generated by DESeq or nbinomWaldTest functions, they are now produced by the lfcShrink function. Dispersions are estimated using expected mean values from the maximum likelihood estimate of log2 fold changes, and optimizing the Cox-Reid adjusted profile likelihood, as first implemented for RNA-seq data in edgeR (Cox and Reid 1987, edgeR_GLM). The steps performed by the DESeq function are documented in its manual page ?DESeq; briefly, they are:

1) estimation of size factors \( s_j \) by estimateSizeFactors
2) estimation of dispersion \( \alpha_i \) by estimateDispersions
3) negative binomial GLM fitting for \( \beta_i \) and Wald statistics by nbinomWaldTest

For access to all the values calculated during these steps, see the section above.

4.2 Changes compared to DESeq

The main changes in the package DESeq2, compared to the (older) version DESeq, are as follows:
RangedSummarizedExperiment is used as the superclass for storage of input data, intermediate calculations and results.

Optional, maximum a posteriori estimation of GLM coefficients incorporating a zero-centered Normal prior with variance estimated from data (equivalent to Tikhonov/ridge regularization). This adjustment has little effect on genes with high counts, yet it helps to moderate the otherwise large variance in log2 fold change estimates for genes with low counts or highly variable counts. These estimates are now provided by the lfcShrink function.

Maximum a posteriori estimation of dispersion replaces the sharingMode options fit-only or maximum of the previous version of the package. This is similar to the dispersion estimation methods of DSS (H. Wu, Wang, and Wu 2012).

All estimation and inference is based on the generalized linear model, which includes the two condition case (previously the exact test was used).

The Wald test for significance of GLM coefficients is provided as the default inference method, with the likelihood ratio test of the previous version still available.

It is possible to provide a matrix of sample-/gene-dependent normalization factors.

Automatic independent filtering on the mean of normalized counts.

Automatic outlier detection and handling.

4.3 Methods changes since the 2014 DESeq2 paper

In version 1.16 (November 2016), the log2 fold change shrinkage is no longer default for the DESeq and nbinomWaldTest functions, by setting the defaults of these to betaPrior=FALSE, and by introducing a separate function lfcShrink, which performs log2 fold change shrinkage for visualization and ranking of genes. While for the majority of bulk RNA-seq experiments, the LFC shrinkage did not affect statistical testing, DESeq2 has become used as an inference engine by a wider community, and certain sequencing datasets show better performance with the testing separated from the use of the LFC prior. Also, the separation of LFC shrinkage to a separate function lfcShrink allows for easier methods development of alternative effect size estimators.

A small change to the independent filtering routine: instead of taking the quantile of the filter (the mean of normalized counts) which directly maximizes the number of rejections, the threshold chosen is the lowest quantile of the filter for which the number of rejections is close to the peak of a curve fit to the number of rejections over the filter quantiles. “Close to” is defined as within 1 residual standard deviation. This change was introduced in version 1.10 (October 2015).

For the calculation of the beta prior variance, instead of matching the empirical quantile to the quantile of a Normal distribution, DESeq2 now uses the weighted quantile function of the Hmisc package. The weighting is described in the manual page for nbinomWaldTest. The weights are the inverse of the expected variance of log counts (as used in the diagonals of the matrix W in the GLM). The effect of the change is that the estimated prior variance is robust against noisy estimates of log fold change from genes with very small counts. This change was introduced in version 1.6 (October 2014).

For a list of all changes since version 1.0.0, see the NEWS file included in the package.
4.4 Count outlier detection

DESeq2 relies on the negative binomial distribution to make estimates and perform statistical inference on differences. While the negative binomial is versatile in having a mean and dispersion parameter, extreme counts in individual samples might not fit well to the negative binomial. For this reason, we perform automatic detection of count outliers. We use Cook’s distance, which is a measure of how much the fitted coefficients would change if an individual sample were removed (Cook 1977). For more on the implementation of Cook’s distance see the manual page for the `results` function. Below we plot the maximum value of Cook’s distance for each row over the rank of the test statistic to justify its use as a filtering criterion.

```r
W <- res$stat
maxCooks <- apply(assays(dds)["cooks"], 1, max)
idx <- !is.na(W)
plot(rank(W[idx]), maxCooks[idx], xlab="rank of Wald statistic", ylab="maximum Cook's distance per gene", ylim=c(0,5), cex=.4, col=rgb(0,0,.3))
m <- ncol(dds)
p <- 3
abline(h=qf(.99, p, m - p))
```

4.5 Contrasts

Contrasts can be calculated for a `DESeqDataSet` object for which the GLM coefficients have already been fit using the Wald test steps (`DESeq` with `test="Wald"` or using `nbinomWaldTest`). The vector of coefficients \( \beta \) is left multiplied by the contrast vector \( c \) to form the numerator
of the test statistic. The denominator is formed by multiplying the covariance matrix $\Sigma$ for the coefficients on either side by the contrast vector $c$. The square root of this product is an estimate of the standard error for the contrast. The contrast statistic is then compared to a normal distribution as are the Wald statistics for the DESeq2 package.

$$W = \frac{c^t \beta}{\sqrt{c^t \Sigma c}}$$

### 4.6 Expanded model matrices

DESeq2 uses expanded model matrices in conjunction with the log2 fold change prior, in order to produce shrunken log2 fold change estimates and test results which are independent of the choice of reference level. Another way of saying this is that the shrinkage is symmetric with respect to all the levels of the factors in the design. The expanded model matrices differ from the standard model matrices, in that they have an indicator column (and therefore a coefficient) for each level of factors in the design formula in addition to an intercept. Note that in version 1.10 and onward, standard model matrices are used for designs with interaction terms, as the shrinkage of log2 fold changes is not recommended for these designs.

The expanded model matrices are not full rank, but a coefficient vector $\beta_i$ can still be found due to the zero-centered prior on non-intercept coefficients. The prior variance for the log2 fold changes is calculated by first generating maximum likelihood estimates for a standard model matrix. The prior variance for each level of a factor is then set as the average of the mean squared maximum likelihood estimates for each level and every possible contrast, such that that this prior value will be reference-level-independent. The contrast argument of the results function is used in order to generate comparisons of interest.

### 4.7 Independent filtering and multiple testing

#### 4.7.1 Filtering criteria

The goal of independent filtering is to filter out those tests from the procedure that have no, or little chance of showing significant evidence, without even looking at their test statistic. Typically, this results in increased detection power at the same experiment-wide type I error. Here, we measure experiment-wide type I error in terms of the false discovery rate.

A good choice for a filtering criterion is one that

1) is statistically independent from the test statistic under the null hypothesis,
2) is correlated with the test statistic under the alternative, and
3) does not notably change the dependence structure – if there is any – between the tests that pass the filter, compared to the dependence structure between the tests before filtering.

The benefit from filtering relies on property (2), and we will explore it further below. Its statistical validity relies on property (1) – which is simple to formally prove for many combinations of filter criteria with test statistics – and (3), which is less easy to theoretically imply from first principles, but rarely a problem in practice. We refer to (Bourgon, Gentleman, and Huber 2010) for further discussion of this topic.
A simple filtering criterion readily available in the results object is the mean of normalized counts irrespective of biological condition, and so this is the criterion which is used automatically by the `results` function to perform independent filtering. Genes with very low counts are not likely to see significant differences typically due to high dispersion. For example, we can plot the $-\log_{10} p$ values from all genes over the normalized mean counts:

```r
plot(res$baseMean+1, -log10(res$pvalue),
     log="x", xlab="mean of normalized counts",
     ylab=expression(-log[10](pvalue)),
     ylim=c(0,30),
     cex=.4, col=rgb(0,0,0,.3))
```

### 4.7.2 Why does it work?

Consider the $p$ value histogram below. It shows how the filtering ameliorates the multiple testing problem – and thus the severity of a multiple testing adjustment – by removing a background set of hypotheses whose $p$ values are distributed more or less uniformly in $[0,1]$.

```r
use <- res$baseMean > metadata(res)$filterThreshold
h1 <- hist(res$pvalue[!use], breaks=0:50/50, plot=FALSE)

h2 <- hist(res$pvalue[use], breaks=0:50/50, plot=FALSE)

colori <- c('do not pass'="khaki", 'pass'="powderblue")
```

Histogram of $p$ values for all tests. The area shaded in blue indicates the subset of those that pass the filtering, the area in khaki those that do not pass:
5 Frequently asked questions

5.1 How can I get support for DESeq2?

We welcome questions about our software, and want to ensure that we eliminate issues if and when they appear. We have a few requests to optimize the process:

- all questions should take place on the Bioconductor support site: https://support.bioconductor.org, which serves as a repository of questions and answers. This helps to save the developers' time in responding to similar questions. Make sure to tag your post with deSeq2. It is often very helpful in addition to describe the aim of your experiment.
- before posting, first search the Bioconductor support site mentioned above for past threads which might have answered your question.
- if you have a question about the behavior of a function, read the sections of the manual page for this function by typing a question mark and the function name, e.g. ?results. We spend a lot of time documenting individual functions and the exact steps that the software is performing.
- include all of your R code, especially the creation of the DESeqDataSet and the design formula. Include complete warning or error messages, and conclude your message with the full output of sessionInfo().
• if possible, include the output of `as.data.frame(colData(dds))`, so that we can have a sense of the experimental setup. If this contains confidential information, you can replace the levels of those factors using `levels()`.

5.2 Why are some $p$ values set to NA?

See the details above.

5.3 How can I get unfiltered DESeq2 results?

Users can obtain unfiltered GLM results, i.e. without outlier removal or independent filtering with the following call:

```r
dds <- DESeq(dds, minReplicatesForReplace=Inf)
res <- results(dds, cooksCutoff=FALSE, independentFiltering=FALSE)
```

In this case, the only $p$ values set to NA are those from genes with all counts equal to zero.

5.4 How do I use VST or rlog data for differential testing?

The variance stabilizing and rlog transformations are provided for applications other than differential testing, for example clustering of samples or other machine learning applications. For differential testing we recommend the `DESeq` function applied to raw counts as outlined above.

5.5 Can I use DESeq2 to analyze paired samples?

Yes, you should use a multi-factor design which includes the sample information as a term in the design formula. This will account for differences between the samples while estimating the effect due to the condition. The condition of interest should go at the end of the design formula, e.g. `~ subject + condition`.

5.6 If I have multiple groups, should I run all together or split into pairs of groups?

Typically, we recommend users to run samples from all groups together, and then use the `contrast` argument of the `results` function to extract comparisons of interest after fitting the model using `DESeq`.

The model fit by `DESeq` estimates a single dispersion parameter for each gene, which defines how far we expect the observed count for a sample will be from the mean value from the model given its size factor and its condition group. See the section above and the DESeq2 paper for full details. Having a single dispersion parameter for each gene is usually sufficient for analyzing multi-group data, as the final dispersion value will incorporate the within-group variability across all groups.
However, for some datasets, exploratory data analysis (EDA) plots could reveal that one or more groups has much higher within-group variability than the others. A simulated example of such a set of samples is shown below. This is case where, by comparing groups A and B separately – subsetting a DESeqDataSet to only samples from those two groups and then running DESeq on this subset – will be more sensitive than a model including all samples together. It should be noted that such an extreme range of within-group variability is not common, although it could arise if certain treatments produce an extreme reaction (e.g. cell death). Again, this can be easily detected from the EDA plots such as PCA described in this vignette.

Here we diagram an extreme range of within-group variability with a simulated dataset. Typically, it is recommended to run DESeq across samples from all groups, for datasets with multiple groups. However, this simulated dataset shows a case where it would be preferable to compare groups A and B by creating a smaller dataset without the C samples. Group C has much higher within-group variability, which would inflate the per-gene dispersion estimate for groups A and B as well:
5.7 Can I run DESeq2 to contrast the levels of 100 groups?

DESeq2 will work with any kind of design specified using the R formula. We encourage users to consider exploratory data analysis such as principal components analysis rather than performing statistical testing of all pairs of many groups of samples. Furthermore, if one is contrasting all pairs of many levels, and then only presenting the results of a subset of those pairs, one needs to perform multiple testing across contrasts as well as genes to control for this additional form of multiple testing. This can be done by using the p.adjust function across a long vector of p values from all pairs of contrasts, then re-assigning these adjusted p values to the appropriate results table.

As a speed concern with fitting very large models, note that each additional level of a factor in the design formula adds another parameter to the GLM which is fit by DESeq2. Users might consider first removing genes with very few reads, e.g. genes with row sum of 1, as this will speed up the fitting procedure.

5.8 Can I use DESeq2 to analyze a dataset without replicates?

If a DESeqDataSet is provided with an experimental design without replicates, a warning is printed, that the samples are treated as replicates for estimation of dispersion. This kind of analysis is only useful for exploring the data, but will not provide the kind of proper statistical inference on differences between groups. Without biological replicates, it is not possible to estimate the biological variability of each gene. More details can be found in the manual page for ?DESeq.

5.9 How can I include a continuous covariate in the design formula?

Continuous covariates can be included in the design formula in exactly the same manner as factorial covariates, and then results for the continuous covariate can be extracted by specifying name. Continuous covariates might make sense in certain experiments, where a constant fold change might be expected for each unit of the covariate. However, in many cases, more meaningful results can be obtained by cutting continuous covariates into a factor defined over a small number of bins (e.g. 3-5). In this way, the average effect of each group is controlled for, regardless of the trend over the continuous covariates. In R, numeric vectors can be converted into factors using the function cut.

5.10 I ran a likelihood ratio test, but results() only gives me one comparison.

"... How do I get the p values for all of the variables/levels that were removed in the reduced design?"

This is explained in the help page for ?results in the section about likelihood ratio test p-values, but we will restate the answer here. When one performs a likelihood ratio test, the p values and the test statistic (the stat column) are values for the test that removes all of
the variables which are present in the full design and not in the reduced design. This tests
the null hypothesis that all the coefficients from these variables and levels of these factors are
equal to zero.

The likelihood ratio test \( p \) values therefore represent a test of *all the variables and all the
levels of factors* which are among these variables. However, the results table only has space
for one column of log fold change, so a single variable and a single comparison is shown
(among the potentially multiple log fold changes which were tested in the likelihood ratio
test). This is indicated at the top of the results table with the text, e.g., log2 fold change
(MLE): condition C vs A, followed by, LRT \( p \)-value: `~ batch + condition` vs `~ batch`. This
indicates that the \( p \) value is for the likelihood ratio test of *all the variables and all the levels*,
while the log fold change is a single comparison from among those variables and levels. See
the help page for *results* for more details.

5.11 What are the exact steps performed by DESeq()?

See the manual page for *DESeq*, which links to the subfunctions which are called in order,
where complete details are listed. Also you can read the three steps listed in the *DESeq2
model* in this document.

5.12 Is there an official Galaxy tool for DESeq2?

Yes. The repository for the DESeq2 tool is

https://github.com/galaxyproject/tools-iuc/tree/master/tools/deseq2

and a link to its location in the Tool Shed is

https://toolshed.g2.bx.psu.edu/view/iuc/deseq2/d983d19fbbab.

5.13 I want to benchmark DESeq2 comparing to other DE tools.

One aspect which can cause problems for comparison is that, by default, DESeq2 outputs \( \text{NA} \)
values for adjusted \( p \) values based on independent filtering of genes which have low counts.
This is a way for the DESeq2 to give extra information on why the adjusted \( p \) value for this
gene is not small. Additionally, \( p \) values can be set to \( \text{NA} \) based on extreme count outlier
detection. These \( \text{NA} \) values should be considered *negatives* for purposes of estimating
sensitivity and specificity. The easiest way to work with the adjusted \( p \) values in a benchmarking context
is probably to convert these \( \text{NA} \) values to 1:

\[
\text{res$padj} <- \text{ifelse(is.na(res$padj), 1, res$padj)}
\]

5.14 I have trouble installing DESeq2 on Ubuntu/Linux...

"I try to install DESeq2 using biocLite(), but I get an error trying to install the R packages
XML and/or RCurl!"

ERROR: configuration failed for package XML
You need to install the following dev versions of packages using your standard package manager, e.g. `sudo apt-get install` or `sudo apt install`:

- libxml2-dev
- libcurl4-openssl-dev

### Acknowledgments

We have benefited in the development of DESeq2 from the help and feedback of many individuals, including but not limited to:


### Session info

```r
sessionInfo()
## R Under development (unstable) (2016-12-02 r71716)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.1 LTS
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8   LC_NUMERIC=C
## [3] LC_TIME=en_US.UTF-8    LC_COLLATE=C
## [5] LC_MONETARY=en_US.UTF-8 LC_MESSAGES=en_US.UTF-8
## [7] LC_PAPER=en_US.UTF-8   LC_NAME=C
## [9] LC_ADDRESS=C           LC_TELEPHONE=C
##
## attached base packages:
## [1] parallel  stats4  stats graphics grDevices utils datasets
## [8] methods  base
##
## other attached packages:
## [1] BiocStyle_2.3.28        RColorBrewer_1.1-2
## [3] pheatmap_1.0.8           hexbin_1.27.1
## [5] vsn_3.43.7                ggplot2_2.2.1
## [7] IHW_1.3.0                 airway_0.109.0
## [9] pasilla_1.3.0             DESeq2_1.15.28
## [11] SummarizedExperiment_1.5.3 Biobase_2.35.0
## [13] GenomicRanges_1.27.19    GenomeInfoDb_1.11.6
```
References


