Analyzing RNA-seq data with DESeq2

Michael I. Love, Simon Anders, and Wolfgang Huber

09/28/2023

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, and 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 (http://www.bioconductor.org/help/workflows/rnaseqGene/) on the Bioconductor website covers similar material to this vignette but at a slower pace, including the generation of count matrices from FASTQ files. DESeq2 package version: 1.41.12

Standard workflow

Note: if you use DESeq2 in published research, please cite:

> Love, M.I., Huber, W., Anders, S. (2014) Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2. *Genome Biology*, **15**:550. 10.1186/s13059-014-0550-8 (http://dx.doi.org/10.1186/s13059-014-0550-8)

Other Bioconductor packages with similar aims are edgeR (http://bioconductor.org/packages/edgeR), limma (http://bioconductor.org/packages/limma), DSS (http://bioconductor.org/packages/DSS), EBSeq (http://bioconductor.org/packages/EBSeq), and baySeq (http://bioconductor.org/packages/baySeq).

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`.

```
dds <- DESeqDataSetFromMatrix(countData = cts,
                              colData = coldata,
                              design= ~ batch + condition)
dds <- DESeq(dds)
resultsNames(dds) # lists the coefficients
res <- results(dds, name="condition_trt_vs_untrt")
# or to shrink log fold changes association with condition:
res <- lfcShrink(dds, coef="condition_trt_vs_untrt", type="apeglm")
```

The following starting functions will be explained below:

- If you have performed transcript quantification (with *Salmon*, *kallisto*, *RSEM*, etc.) you could import the data with *tximport*, which produces a list, and then you can use `DESeqDataSetFromTximport()`.
- If you imported quantification data with *tximeta*, which produces a *SummarizedExperiment* with additional metadata, you can then use `DESeqDataSet()`.
- If you have *htseq-count* files, you can use `DESeqDataSetFromHTSeq()`.

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 (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**.

Acknowledgments

Constantin Ahlmann-Eltze has contributed core code for increasing the computational performance of *DESeq2* and building an interface to his *glmGamPoi* package.

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

The Bionconductor Core Team, Alejandro Reyes, Andrzej Oles, Aleksandra Pekowska, Felix Klein, Nikolaos Ignatiadis (IHW), Anqi Zhu (apeglm), Joseph Ibrahim (apeglm), Vince Carey, Owen Solberg, Ruping Sun, Devon Ryan, Steve Lianoglou, Jessica Larson, Christina Chaivorapol, Pan Du, Richard Bourgon, Willem Talloen, Elin Videvall, Hanneke van Deutekom, Todd Burwell, Jesse Rowley, Igor Dolgalev, Stephen Turner, Ryan C Thompson, Tyr Wiesner-Hanks, Konrad Rudolph, David Robinson, Mingxiang Teng, Mathias Lesche, Sonali Arora, Jordan Ramilowski, Ian Dworkin, Bjorn Gruning, Ryan McMinds, Paul Gordon, Leonardo Collado Torres, Enrico Ferrero, Peter Langfelder, Gavin Kelly, Rob Patro, Charlotte Soneson, Koen Van den Berge, Fanny Perraudeau, Davide Risso, Stephan Engelbrecht, Nicolas Alcala, Jeremy Simon, Travis Ptacek, Rory Kirchner, R. Butler, Ben Keith, Dan Liang, Nil Aygün, Rory Nolan, Michael Schubert, Hugo Tavares, Eric Davis, Wancen Mu, Zhang Cheng, Frederik Ziebell, Luca Menestrina, Hendrik Weisse.

Funding

DESeq2 and its developers have been partially supported by funding from the European Union’s 7th Framework Programme via Project RADIANT, NIH NHGRI R01-HG009937, and by a CZI EOSS award.

Input data

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 (http://www.bioconductor.org/help/workflows/rnaseqGene/) 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.

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 (http://bioconductor.org/packages/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

Transcript abundance files and *tximport* / *tximeta*

Our recommended pipeline for *DESeq2* 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 tximport (http://bioconductor.org/packages/tximport) (Soneson, Love, and Robinson 2015). This workflow allows users to import transcript abundance estimates from a variety of external software, including the following methods:

- Salmon (http://combine-lab.github.io/salmon/) (Patro et al. 2017)
- Sailfish (http://www.cs.cmu.edu/~ckingsf/software/sailfish/) (Patro, Mount, and Kingsford 2014)
- kallisto (https://pachterlab.github.io/kallisto/about.html) (Bray et al. 2016)
- RSEM (http://deweylab.github.io/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.

A tutorial on how to use the *Salmon* software for quantifying transcript abundance can be found here (https://combine-lab.github.io/salmon/getting_started/). We recommend using the `--gcBias` flag (http://salmon.readthedocs.io/en/latest/salmon.html#gcbias) which estimates a correction factor for systematic biases commonly present in RNA-seq data (Love, Hogenesch, and Irizarry 2016; Patro et al. 2017), unless you are certain that your data do not contain such bias.

Here, we demonstrate how to import transcript abundances and construct a gene-level *DESeqDataSet* object from *Salmon* `quant.sf` files, which are stored in the tximportData (http://bioconductor.org/packages/tximportData) package. You do not need the `tximportData` package for your analysis, it is only used here for demonstration.

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.

```
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$run, "quant.sf.gz")
names(files) <- samples$run
tx2gene <- read_csv(file.path(dir, "tx2gene.gencode.v27.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 (http://bioconductor.org/packages/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.

Tximeta for import with automatic metadata

Another Bioconductor package, tximeta (https://bioconductor.org/packages/tximeta) (Love et al. 2020), extends *tximport*, offering the same functionality, plus the additional benefit of automatic addition of annotation metadata for commonly used transcriptomes (GENCODE, Ensembl, RefSeq for human and mouse). See the tximeta (https://bioconductor.org/packages/tximeta) package vignette for more details. *tximeta* produces a *SummarizedExperiment* that can be loaded easily into *DESeq2* using the `DESeqDataSet` function, with an example in the *tximeta* package vignette, and below:

```
coldata <- samples
coldata$files <- files
coldata$names <- coldata$run
```

```
library("tximeta")
se <- tximeta(coldata)
ddsTxi <- DESeqDataSet(se, design = ~ condition)
```

The `ddsTxi` object here can then be used as `dds` in the following analysis steps. If *tximeta* recognized the reference transcriptome as one of those with a pre-computed hashed checksum, the `rowRanges` of the `dds` object will be pre-populated. Again, see the *tximeta* vignette for full details.

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 (http://bioconductor.org/packages/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 (http://bioconductor.org/packages/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.

```r
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")]
coldata$condition <- factor(coldata$condition)
coldata$type <- factor(coldata$type)
```

We examine the count matrix and column data to see if they are consistent in terms of sample order.

```r
head(cts,2)
```

```
##             untreated1 untreated2 untreated3 untreated4 treated1 treated2
## FBgn0000003          0          0          0          0        0        0
## FBgn0000008         92        161         76         70      140       88
##             treated3
## FBgn0000003        1
## FBgn0000008       70
```

```
coldata
```

```
##              condition        type
## treated1fb     treated single-read
## treated2fb     treated  paired-end
## treated3fb     treated  paired-end
## untreated1fb untreated single-read
## untreated2fb untreated single-read
## untreated3fb untreated  paired-end
## untreated4fb untreated  paired-end
```

Note that these are not in the same order with respect to samples!

It is absolutely critical that the columns of the count matrix and the rows of the column data (information about samples) are in the same order. DESeq2 will not make guesses as to which column of the count matrix belongs to which row of the column data, these must be provided to DESeq2 already in consistent order.

As they are not in the correct order as given, we need to 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.

```
rownames(coldata) <- sub("fb", "", rownames(coldata))
all(rownames(coldata) %in% colnames(cts))
```

```
## [1] TRUE
```

```
all(rownames(coldata) == colnames(cts))
```

```
## [1] FALSE
```

```
cts <- 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 (http://bioconductor.org/packages/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 *htseq-count* files, see the dedicated input function below.

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

```
library("DESeq2")
dds <- DESeqDataSetFromMatrix(countData = cts,
                              colData = coldata,
                              design = ~ condition)
dds
```

```
## class: DESeqDataSet
## dim: 14599 7
## metadata(1): version
## assays(1): counts
## rownames(14599): FBgn0000003 FBgn0000008 ... FBgn0261574 FBgn0261575
## rowData names(0):
## colnames(7): treated1 treated2 ... untreated3 untreated4
## colData names(2): condition type
```

If you have additional feature data, it can be added to the *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 `dds`.)

```
featureData <- data.frame(gene=rownames(cts))
mcols(dds) <- DataFrame(mcols(dds), featureData)
mcols(dds)
```

```
## DataFrame with 14599 rows and 1 column
##                    gene
##             <character>
## FBgn0000003 FBgn0000003
## FBgn0000008 FBgn0000008
## FBgn0000014 FBgn0000014
## FBgn0000015 FBgn0000015
## FBgn0000017 FBgn0000017
## ...                 ...
## FBgn0261571 FBgn0261571
## FBgn0261572 FBgn0261572
## FBgn0261573 FBgn0261573
## FBgn0261574 FBgn0261574
## FBgn0261575 FBgn0261575
```

htseq-count input

You can use the function *DESeqDataSetFromHTSeqCount* if you have used *htseq-count* from the HTSeq (http://www-huber.embl.de/users/anders/HTSeq) python package (Anders, Pyl, and Huber 2014). For an example of using the python scripts, see the pasilla (http://bioconductor.org/packages/pasilla) data package. First you will want to specify a variable which points to the directory in which the *htseq-count* output files are located.

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

However, for demonstration purposes only, the following line of code points to the directory for the demo *htseq-count* output files packages for the pasilla (http://bioconductor.org/packages/pasilla) package.

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

We specify which files to read in using *list.files*, and select those files which contain the string `"treated"` using *grep*. The *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 *read.table*.

```
sampleFiles <- grep("treated",list.files(directory),value=TRUE)
sampleCondition <- sub("(.*treated).*","\\1",sampleFiles)
sampleTable <- data.frame(sampleName = sampleFiles,
                          fileName = sampleFiles,
                          condition = sampleCondition)
sampleTable$condition <- factor(sampleTable$condition)
```

Then we build the *DESeqDataSet* using the following function:

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

```
## class: DESeqDataSet
## dim: 70463 7
## metadata(1): version
## assays(1): counts
## rownames(70463): FBgn0000003:001 FBgn0000008:001 ... FBgn0261575:001
##   FBgn0261575:002
## rowData names(0):
## colnames(7): treated1fb.txt treated2fb.txt ... untreated3fb.txt
##   untreated4fb.txt
## colData names(1): condition
```

SummarizedExperiment input

If one has already created or obtained a *SummarizedExperiment*, it can be easily input into DESeq2 as follows. First we load the package containing the `airway` dataset.

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

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

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

```
## class: DESeqDataSet
## dim: 63677 8
## metadata(2): '' version
## assays(1): counts
## rownames(63677): ENSG00000000003 ENSG00000000005 ... ENSG00000273492
##   ENSG00000273493
## rowData names(10): gene_id gene_name ... seq_coord_system symbol
## colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
## colData names(9): SampleName cell ... Sample BioSample
```

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 very few reads, we reduce the memory size of the `dds` data object, and we increase the speed of count modeling within DESeq2. It can also improve visualizations, as features with no information for differential expression are not plotted in dispersion plots or MA-plots.

Here we perform pre-filtering to keep only rows that have a count of at least 10 for a minimal number of samples. The count of 10 is a reasonable choice for bulk RNA-seq. A recommendation for the minimal number of samples is to specify the smallest group size, e.g. here there are 3 treated samples. If there are not discrete groups, one can use the minimal number of samples where non-zero counts would be considered interesting. One can also omit this step entirely and just rely on the independent filtering procedures available in `results()`, either *IHW* or *genefilter*. See independent filtering section.

```
smallestGroupSize <- 3
keep <- rowSums(counts(dds) >= 10) >= smallestGroupSize
dds <- dds[keep,]
```

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. In order to see the change of reference levels reflected in the results names, you need to either run `DESeq` or `nbinomWaldTest` / `nbinomLRT` after the re-leveling operation. 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 *DESeqDataSet*, 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 *DESeqDataSet*:

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

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*.

About the pasilla dataset

We continue with the pasilla (http://bioconductor.org/packages/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 (http://bioconductor.org/packages/pasilla) data is provided in the vignette of the data package pasilla (http://bioconductor.org/packages/pasilla).

Differential expression analysis

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

Results tables are generated using the function *results*, which extracts a results table with log2 fold changes, *p* values and adjusted *p* values. With no additional arguments to *results*, the log2 fold change and Wald test *p* value 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 **reference level** (see previous note on factor levels). However, the order of the variables of the design do not matter so long as the user specifies the comparison to build a results table for, using the `name` or `contrast` arguments of *results*.

Details about the comparison are printed to the console, directly above the results table. The text, `condition treated vs untreated`, tells you that the estimates are of the logarithmic fold change log2(treated/untreated).

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

```
## log2 fold change (MLE): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 8148 rows and 6 columns
##               baseMean log2FoldChange     lfcSE       stat    pvalue      padj
##              <numeric>      <numeric> <numeric>  <numeric> <numeric> <numeric>
## FBgn0000008   95.28865     0.00399148  0.225010  0.0177391 0.9858470  0.996699
## FBgn0000017 4359.09632    -0.23842494  0.127094 -1.8759764 0.0606585  0.289604
## FBgn0000018  419.06811    -0.10185506  0.146568 -0.6949338 0.4870968  0.822681
## FBgn0000024    6.41105     0.21429657  0.691557  0.3098756 0.7566555  0.939146
## FBgn0000032  990.79225    -0.08896298  0.146253 -0.6082822 0.5430003  0.848881
## ...                ...            ...       ...        ...       ...       ...
## FBgn0261564   1160.028     -0.0857255  0.108354 -0.7911643 0.4288481  0.789246
## FBgn0261565    620.388     -0.2943294  0.140496 -2.0949303 0.0361772  0.206423
## FBgn0261570   3212.969      0.2971841  0.126742  2.3447877 0.0190379  0.133380
## FBgn0261573   2243.936      0.0146611  0.111365  0.1316493 0.8952617  0.977565
## FBgn0261574   4863.807      0.0179729  0.194137  0.0925784 0.9262385  0.986726
```

Note that we could have specified the coefficient or contrast we want to build a results table for, using either of the following equivalent commands:

```
res <- results(dds, name="condition_treated_vs_untreated")
res <- results(dds, contrast=c("condition","treated","untreated"))
```

One exception to the equivalence of these two commands, is that, using `contrast` will additionally set to 0 the estimated LFC in a comparison of two groups, where all of the counts in the two groups are equal to 0 (while other groups have positive counts). As this may be a desired feature to have the LFC in these cases set to 0, one can use `contrast` to build these results tables. More information about extracting specific coefficients from a fitted *DESeqDataSet* object can be found in the help page `?results` . The use of the `contrast` argument is also further discussed below.

Log fold change shrinkage for visualization and ranking

Shrinkage of effect size (LFC estimates) is useful for visualization and ranking of genes. To shrink the LFC, we pass the `dds` object to the function `lfcShrink` . Below we specify to use the *apeglm* method for effect size shrinkage (Zhu, Ibrahim, and Love 2018), which improves on the previous estimator.

We provide the `dds` object and the name or number of the coefficient we want to shrink, where the number refers to the order of the coefficient as it appears in `resultsNames(dds)` .

```r
resultsNames(dds)
```

```
## [1] "Intercept"                      "condition_treated_vs_untreated"
```

```r
resLFC <- lfcShrink(dds, coef="condition_treated_vs_untreated", type="apeglm")
resLFC
```

```
## log2 fold change (MAP): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 8148 rows and 5 columns
##              baseMean log2FoldChange     lfcSE    pvalue      padj
##             <numeric>      <numeric> <numeric> <numeric> <numeric>
## FBgn0000008   95.28865     0.00195376  0.152654 0.9858470  0.996699
## FBgn0000017 4359.09632    -0.18810628  0.120870 0.0606585  0.289604
## FBgn0000018  419.06811    -0.06893831  0.122805 0.4870968  0.822681
## FBgn0000024    6.41105     0.01786546  0.199499 0.7566555  0.939146
## FBgn0000032  990.79225    -0.06001511  0.121962 0.5430003  0.848881
## ...                ...            ...       ...       ...       ...
## FBgn0261564   1160.028     -0.0669829 0.0976567 0.4288481  0.789246
## FBgn0261565    620.388     -0.2284564 0.1362122 0.0361772  0.206423
## FBgn0261570   3212.969      0.2395981 0.1237304 0.0190379  0.133380
## FBgn0261573   2243.936      0.0115395 0.0981689 0.8952617  0.977565
## FBgn0261574   4863.807      0.0101618 0.1417667 0.9262385  0.986726
```

Shrinkage estimation is discussed more in a later section.

Speed-up and parallelization thoughts

The above steps should take less than 30 seconds for most analyses. For experiments with complex designs and many samples (e.g. dozens of coefficients, ~100s of samples), one may want to have faster computation than provided by the default run of `DESeq`. We have two recommendations:

1. By using the argument `fitType="glmGamPoi"`, one can leverage the faster NB GLM engine written by Constantin Ahlmann-Eltze. Note that glmGamPoi's interface in DESeq2 requires use of `test="LRT"` and specification of a `reduced` design.
2. One can take advantage of parallelized computation. Parallelizing `DESeq`, `results`, and `lfcShrink` can be easily accomplished by loading the BiocParallel package, and then setting the following arguments: `parallel=TRUE` and `BPPARAM=MulticoreParam(4)`, for example, splitting the job over 4 cores. However, some words of advice on parallelization: first, it is recommend to filter genes where all samples have low counts, to avoid sending data unnecessarily to child processes, when those genes have low power and will be independently filtered

anyway; secondly, there is often diminishing returns for adding more cores due to overhead of sending data to child processes, therefore I recommend first starting with small number of additional cores. Note that obtaining `results` for coefficients or contrasts listed in `resultsNames(dds)` is fast and will not need parallelization. As an alternative to `BPPARAM`, one can `register` cores at the beginning of an analysis, and then just specify `parallel=TRUE` to the functions when called.

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

p-values and adjusted p-values

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

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

We can summarize some basic tallies using the *summary* function.

```
summary(res)
```

```
## 
## out of 8148 with nonzero total read count
## adjusted p-value < 0.1
## LFC > 0 (up)       : 533, 6.5%
## LFC < 0 (down)     : 536, 6.6%
## outliers [1]       : 0, 0%
## low counts [2]     : 0, 0%
## (mean count < 5)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```

How many adjusted p-values were less than 0.1?

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

```
## [1] 1069
```

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:

```
res05 <- results(dds, alpha=0.05)
summary(res05)
```

```
##
## out of 8148 with nonzero total read count
## adjusted p-value < 0.05
## LFC > 0 (up)       : 416, 5.1%
## LFC < 0 (down)     : 437, 5.4%
## outliers [1]       : 0, 0%
## low counts [2]     : 0, 0%
## (mean count < 5)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```

```
sum(res05$padj < 0.05, na.rm=TRUE)
```

```
## [1] 853
```

Independent hypothesis weighting

A generalization of the idea of *p* value filtering is to *weight* hypotheses to optimize power. A Bioconductor package, IHW (http://bioconductor.org/packages/IHW), is available that implements the method of *Independent Hypothesis Weighting* (Ignatiadis et al. 2016). Here we show the use of *IHW* for *p* value adjustment of DESeq2 results. For more details, please see the vignette of the IHW (http://bioconductor.org/packages/IHW) package. The *IHW* result object is stored in the metadata.

Note: If the results of independent hypothesis weighting are used in published research, please cite:

> Ignatiadis, N., Klaus, B., Zaugg, J.B., Huber, W. (2016) Data-driven hypothesis weighting increases detection power in genome-scale multiple testing. *Nature Methods*, **13**:7. 10.1038/nmeth.3885 (http://dx.doi.org/10.1038/nmeth.3885)

```
# (unevaluated code chunk)
library("IHW")
resIHW <- results(dds, filterFun=ihw)
summary(resIHW)
sum(resIHW$padj < 0.1, na.rm=TRUE)
metadata(resIHW)$ihwResult
```

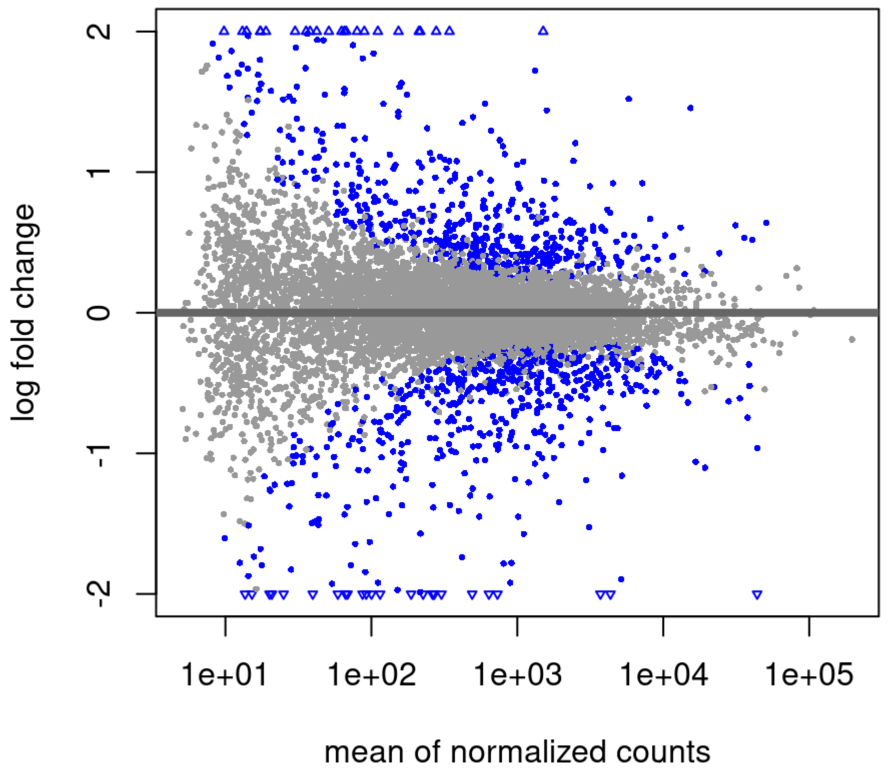
For advanced users, note that all the values calculated by the DESeq2 package are stored in the *DESeqDataSet* object or the *DESeqResults* object, and access to these values is discussed below.

Exploring and exporting results

MA-plot

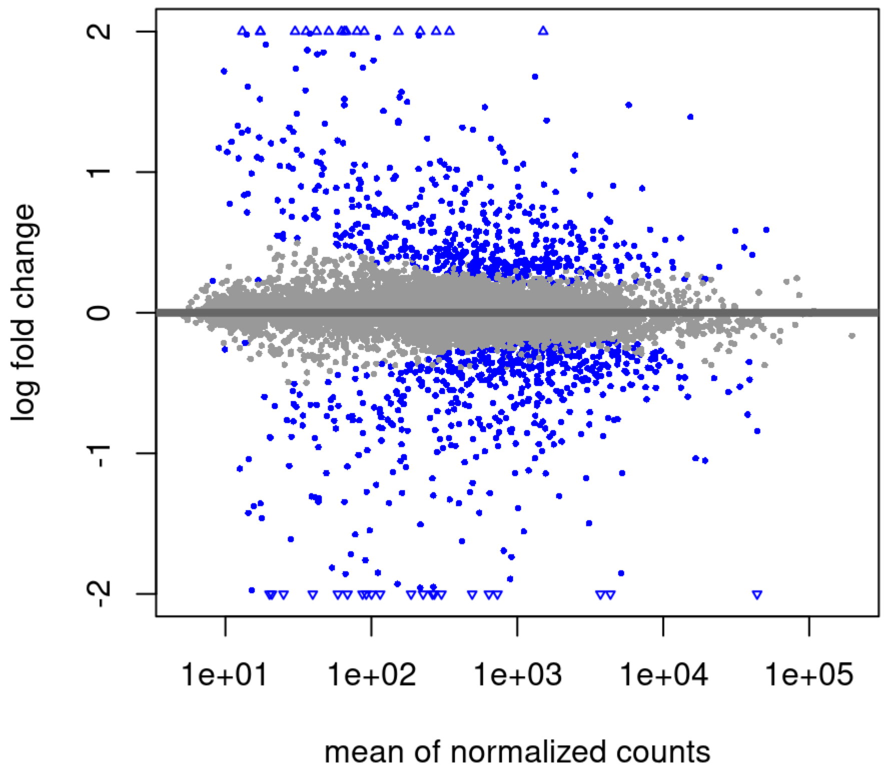
In DESeq2, the function *plotMA* shows the log2 fold changes attributable to a given variable over the mean of normalized counts for all the samples in the *DESeqDataSet*. Points will be colored blue if the adjusted *p* value is less than 0.1. Points which fall out of the window are plotted as open triangles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

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



It is more useful to visualize the MA-plot for the shrunken 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:

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

Alternative shrinkage estimators

The moderated log fold changes proposed by Love, Huber, and Anders (2014) use a normal prior distribution, centered on zero and with a scale that is fit to the data. The shrunken log fold changes are useful for ranking and visualization, without the need for arbitrary filters on low count genes. The normal prior can sometimes produce too strong of shrinkage for certain datasets. In DESeq2 version 1.18, we include two additional adaptive shrinkage estimators, available via the `type` argument of `lfcShrink`. For more details, see `?lfcShrink`

The options for `type` are:

- `apeglm` is the adaptive t prior shrinkage estimator from the apeglm (http://bioconductor.org/packages/apeglm) package (Zhu, Ibrahim, and Love 2018). As of version 1.28.0, it is the default estimator.
- `ashr` is the adaptive shrinkage estimator from the ashr (https://github.com/stephens999/ashr) package (Stephens 2016). Here DESeq2 uses the ashr option to fit a mixture of Normal distributions to form the prior, with `method="shrinkage"` .
- `normal` is the the original DESeq2 shrinkage estimator, an adaptive Normal distribution as prior.

If the shrinkage estimator `apeglm` is used in published research, please cite:

> Zhu, A., Ibrahim, J.G., Love, M.I. (2018) Heavy-tailed prior distributions for sequence count data: removing the noise and preserving large differences. *Bioinformatics*. 10.1093/bioinformatics/bty895 (https://doi.org/10.1093/bioinformatics/bty895)

If the shrinkage estimator `ashr` is used in published research, please cite:

> Stephens, M. (2016) False discovery rates: a new deal. *Biostatistics*, **18**:2. 10.1093/biostatistics/kxw041 (https://doi.org/10.1093/biostatistics/kxw041)

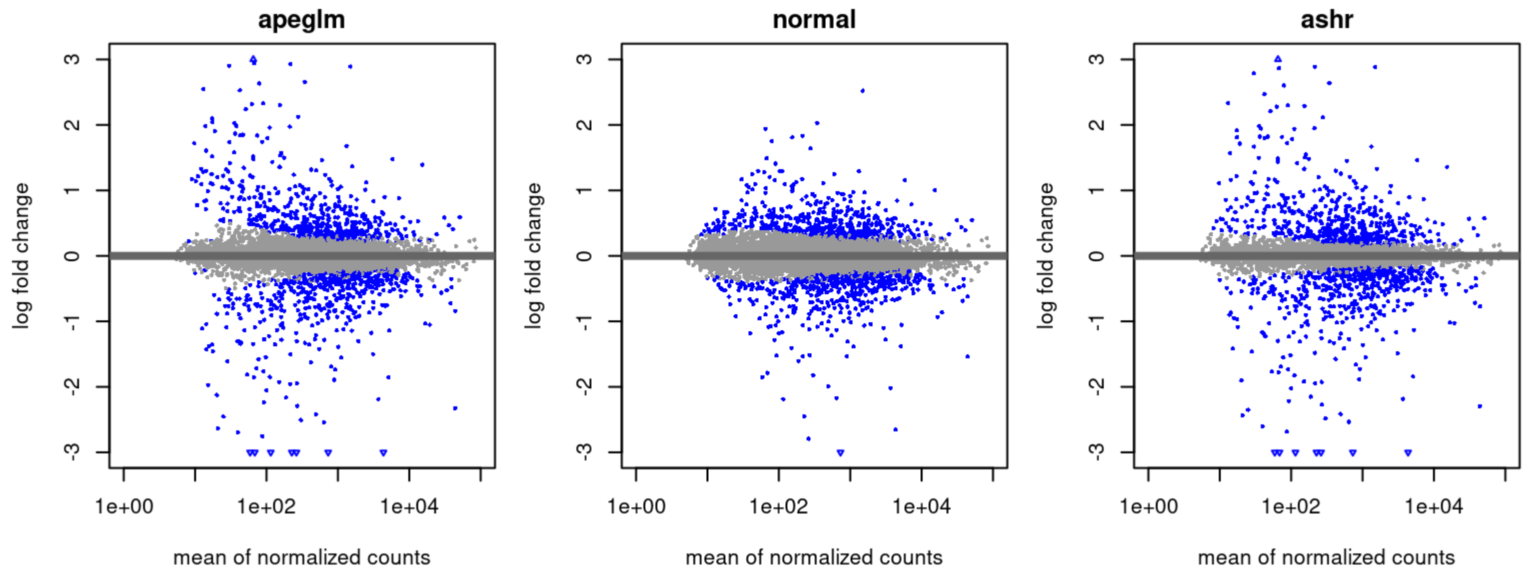
In the LFC shrinkage code above, we specified `coef="condition_treated_vs_untreated"` . We can also just specify the coefficient by the order that it appears in `resultsNames(dds)` , in this case `coef=2` . For more details explaining how the shrinkage estimators differ, and what kinds of designs, contrasts and output is provided by each, see the extended section on shrinkage estimators.

```
resultsNames(dds)
```

```
## [1] "Intercept"                      "condition_treated_vs_untreated"
```

```
# because we are interested in treated vs untreated, we set 'coef=2'
resNorm <- lfcShrink(dds, coef=2, type="normal")
resAsh <- lfcShrink(dds, coef=2, type="ashr")
```

```
par(mfrow=c(1,3), mar=c(4,4,2,1))
xlim <- c(1,1e5); ylim <- c(-3,3)
plotMA(resLFC, xlim=xlim, ylim=ylim, main="apeglm")
plotMA(resNorm, xlim=xlim, ylim=ylim, main="normal")
plotMA(resAsh, xlim=xlim, ylim=ylim, main="ashr")
```



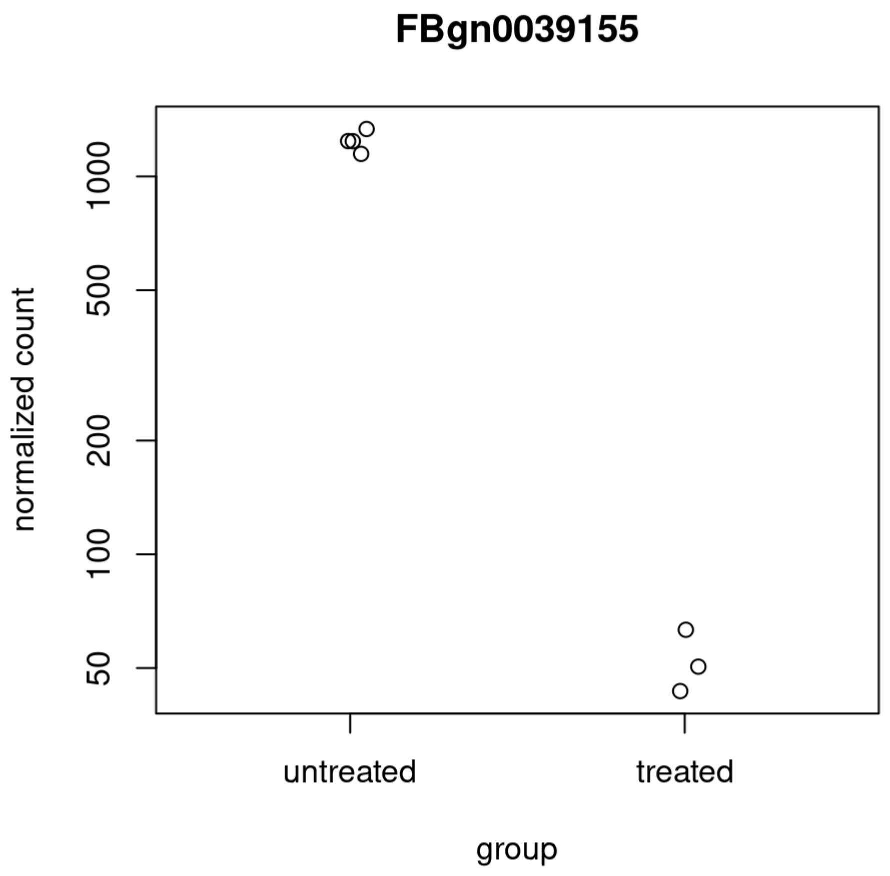
Note: We have sped up the `apeglm` method so it takes roughly about the same amount of time as `normal`, e.g. ~5 seconds for the `pasilla` dataset of ~10,000 genes and 7 samples. If fast shrinkage estimation of LFC is needed, *but the posterior standard deviation is not needed*, setting `apeMethod="nbinomC"` will produce a ~10x speedup, but the `lfcSE` column will be returned with `NA`. A variant of this fast method, `apeMethod="nbinomC*"` includes random starts.

Note: If there is unwanted variation present in the data (e.g. batch effects) it is always recommend to correct for this, which can be accommodated in DESeq2 by including in the design any known batch variables or by using functions/packages such as `svaseq` in sva (http://bioconductor.org/packages/sva) (Leek 2014) or the `RUV` functions in RUVSeq (http://bioconductor.org/packages/RUVSeq) (Risso et al. 2014) to estimate variables that capture the unwanted variation. In addition, the ashr developers have a specific method (https://github.com/dcgerard/vicar) for accounting for unwanted variation in combination with ashr (Gerard and Stephens 2017).

Plot counts

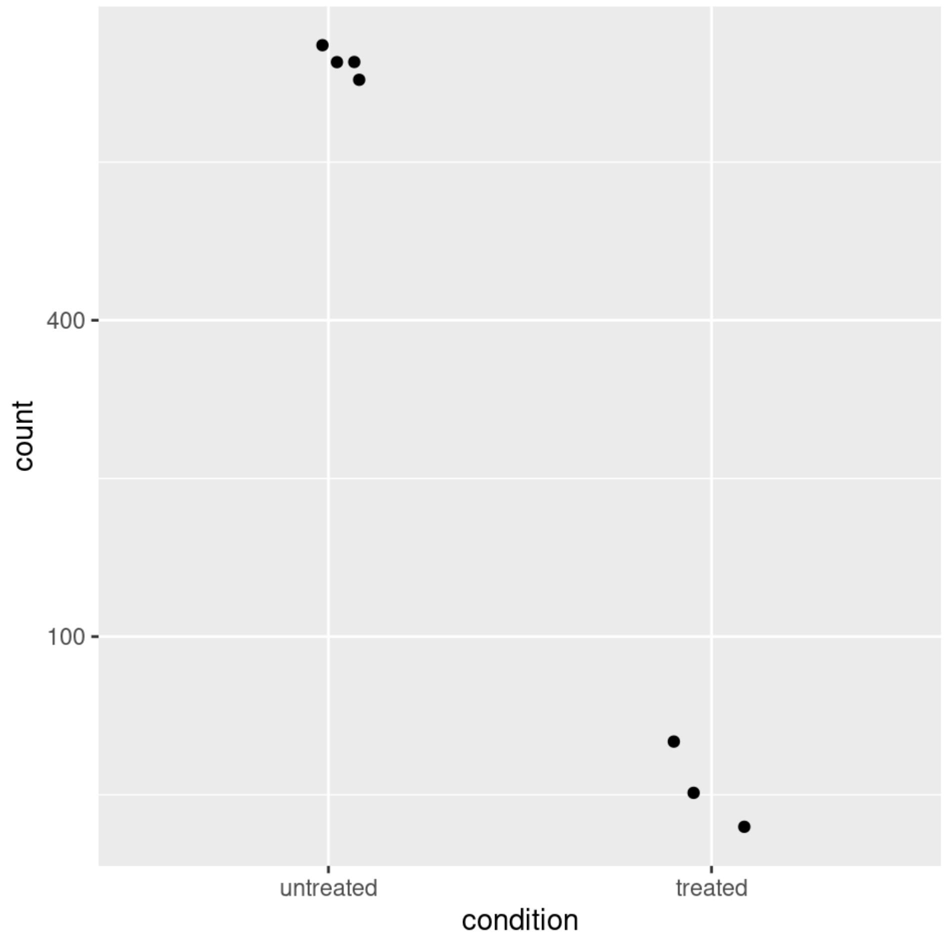
It can also be useful to examine the counts of reads for a single gene across the groups. A simple function for making this plot is *plotCounts*, which normalizes counts by the estimated size factors (or normalization factors if these were used) and adds a pseudocount of 1/2 to allow for log scale plotting. 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.

```
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*.

```
d <- plotCounts(dds, gene=which.min(res$padj), intgroup="condition",
                returnData=TRUE)
library("ggplot2")
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))
```



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.

```
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 `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 `NA`. Description and customization of independent filtering is described below

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 (http://bioconductor.org/packages/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 (http://bioconductor.org/packages/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 (http://bioconductor.org/packages/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 (http://bioconductor.org/packages/regionReport) package.

Glimma Interactive visualization of DESeq2 output, including MA-plots (also called MD-plots) can be generated using the Glimma (http://bioconductor.org/packages/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 (http://bioconductor.org/packages/pcaExplorer) package. See the *Launching the application* section of the package vignette.

iSEE Provides functions for creating an interactive Shiny-based graphical user interface for exploring data stored in SummarizedExperiment objects, including row- and column-level metadata. Particular attention is given to single-cell data in a SingleCellExperiment object with visualization of dimensionality reduction results. iSEE (https://bioconductor.org/packages/iSEE) is on Bioconductor. An example wrapper function for converting a *DESeqDataSet* to a SingleCellExperiment object for use with *iSEE* can be found at the following gist, written by Federico Marini:

- https://gist.github.com/federicomarini/4a543eebc7e7091d9169111f76d59de1 (https://gist.github.com/federicomarini/4a543eebc7e7091d9169111f76d59de1)

The iSEEde (https://bioconductor.org/packages/iSEEde) package provides additional panels that facilitate the interactive visualisation of differential expression results in iSEE applications.

DEvis DEvis is a powerful, integrated solution for the analysis of differential expression data. This package includes an array of tools for manipulating and aggregating data, as well as a wide range of customizable visualizations, and project management functionality that simplify RNA-Seq analysis and provide a variety of ways of exploring and analyzing data. *DEvis* can be found on CRAN (https://cran.r-project.org/package=DEVis) and GitHub (https://github.com/price0416/DEvis).

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.

```
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.

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

```
## log2 fold change (MLE): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 1069 rows and 6 columns
##              baseMean log2FoldChange     lfcSE      stat       pvalue
##             <numeric>      <numeric> <numeric> <numeric>    <numeric>
## FBgn0039155   730.992       -4.61695 0.1667827  -27.6824 1.13645e-168
## FBgn0025111  1504.272        2.90205 0.1261989   22.9958 5.13139e-117
## FBgn0029167  3709.741       -2.19491 0.0957474  -22.9240 2.68013e-116
## FBgn0003360  4344.597       -3.17683 0.1416166  -22.4326 1.89322e-111
## FBgn0035085   638.757       -2.55819 0.1359972  -18.8106  6.18406e-79
## ...               ...            ...       ...       ...          ...
## FBgn0003890 1644.2002      -0.495185  0.199269  -2.48501    0.0129549
## FBgn0010053  178.3259      -0.516894  0.208034  -2.48466    0.0129675
## FBgn0034997 5125.8312       0.250348  0.100801   2.48360    0.0130063
## FBgn0004359   84.1178       0.646359  0.260308   2.48306    0.0130260
## FBgn0027604  283.2465      -0.578675  0.233077  -2.48277    0.0130366
##                     padj
##                <numeric>
## FBgn0039155 9.25983e-165
## FBgn0025111 2.09053e-113
## FBgn0029167 7.27922e-113
## FBgn0003360 3.85649e-108
## FBgn0035085  1.00775e-75
## ...                  ...
## FBgn0003890    0.0991143
## FBgn0010053    0.0991176
## FBgn0034997    0.0993210
## FBgn0004359    0.0993659
## FBgn0027604    0.0993659
```

Multi-factor designs

Experiments with more than one factor influencing the counts can be analyzed using design formulas that include the additional variables. In fact, DESeq2 can analyze any possible experimental design that can be expressed with fixed effects terms (multiple factors, designs with interactions, designs with continuous variables, splines, and so on are all possible).

By adding variables 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).

Experiments with many samples: in experiments with many samples (e.g. 50, 100, etc.) it is highly likely that there will be technical variation affecting the observed counts. Failing to model this additional technical variation will lead to spurious results. Many methods exist that can be used to model technical variation, which can be easily included in the DESeq2 design to control for technical variation while estimating effects of interest. See the RNA-seq workflow (http://www.bioconductor.org/help/workflows/rnaseqGene) for examples of using RUV or SVA in combination with DESeq2. For more details on why it is important to control for technical variation in large sample experiments, see the following thread (https://twitter.com/mikelove/status/1513468597288452097), also archived here (https://htmlpreview.github.io/?https://github.com/frederikziebell/science_tweetorials/blob/master/DESeq2_many_samples.html) by Frederik Ziebell.

The data in the pasilla (http://bioconductor.org/packages/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   1.629707
## treated2   treated   paired-end    0.761162
## treated3   treated   paired-end    0.830312
## untreated1 untreated single-read   1.143904
## untreated2 untreated single-read   1.791281
## untreated3 untreated paired-end    0.645994
## untreated4 untreated paired-end    0.750728
```

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

```
ddsMF <- dds
```

We change the levels of `type` so it only contains letters (numbers, underscore and period are also allowed in design factor levels). Be careful when changing level names to use the same order as the current levels.

```
levels(ddsMF$type)
```

```
## [1] "paired-end"  "single-read"
```

```
levels(ddsMF$type) <- sub("-.*", "", levels(ddsMF$type))
levels(ddsMF$type)
```

```
## [1] "paired" "single"
```

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      padj
##              <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008   95.28865     -0.0390130  0.218997 -0.178144 0.8586100  0.947833
## FBgn0000017 4359.09632     -0.2548984  0.113535 -2.245099 0.0247617  0.131475
## FBgn0000018  419.06811     -0.0625571  0.129956 -0.481372 0.6302523  0.852180
## FBgn0000024    6.41105      0.3097331  0.750231  0.412850 0.6797164  0.877741
## FBgn0000032  990.79225     -0.0465134  0.120215 -0.386918 0.6988171  0.886082
## FBgn0000037   14.11443      0.4541562  0.523436  0.867644 0.3855893  0.691941
```

It is also possible to retrieve the log2 fold changes, *p* values and adjusted *p* values of variables other than the last one in the design. While in this case, `type` is not biologically interesting as it indicates differences across sequencing protocol, for other hypothetical designs, such as `~genotype + condition + genotype:condition`, we may actually be interested in the difference in baseline expression across genotype, which is not the last variable in the design.

In any case, 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

```
resMFType <- results(ddsMF,
                     contrast=c("type", "single", "paired"))
head(resMFType)
```

```
## log2 fold change (MLE): type single vs paired
## Wald test p-value: type single vs paired
## DataFrame with 6 rows and 6 columns
##              baseMean log2FoldChange     lfcSE      stat    pvalue      padj
##             <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008   95.28865      -0.265123  0.217459 -1.219189 0.2227725  0.492729
## FBgn0000017 4359.09632      -0.103203  0.113399 -0.910090 0.3627748  0.642817
## FBgn0000018  419.06811       0.225857  0.128864  1.752669 0.0796589  0.271610
## FBgn0000024    6.41105       0.302083  0.745703  0.405099 0.6854049        NA
## FBgn0000032  990.79225       0.233891  0.119646  1.954855 0.0506002  0.206081
## FBgn0000037   14.11443      -0.053260  0.521939 -0.102043 0.9187228  0.969866
```

If the variable is continuous or an interaction term (see section on interactions) then the results can be extracted using the `name` argument to *results*, where the name is one of elements returned by `resultsNames(dds)`.

Data transformations and visualization

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 parameters equivalent to n_0 above. One makes use of the concept of variance stabilizing transformations (VST) (Tibshirani 1988; Huber et al. 2003; Anders and Huber 2010), and the other is the *regularized logarithm* or *rlog*, which incorporates a prior on the sample differences (Love, Huber, and Anders 2014). Both transformations produce transformed data on the log2 scale which has been normalized with respect to library size or other normalization factors.

The point of these two transformations, the VST and the *rlog*, 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 VST and *rlog* 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.

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).

Blind dispersion estimation

The two functions, *vst* and *rlog* 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.

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, while *vst* takes less than 1 second. The running times are shorter when using `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.

```
vsd <- vst(dds, blind=FALSE)
rld <- rlog(dds, blind=FALSE)
head(assay(vsd), 3)
```

```
##              treated1  treated2  treated3 untreated1 untreated2 untreated3
## FBgn0000008  7.777746  7.984491  7.765448   7.735026   7.807720   7.997378
## FBgn0000017 11.954934 12.035700 12.028610  12.049838  12.295662  12.471723
## FBgn0000018  9.219162  9.089390  9.028791   9.374577   9.173538   9.052143
##             untreated4
## FBgn0000008   7.832458
## FBgn0000017  12.089675
## FBgn0000018   9.142848
```

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 the *vst* function. If a local fit is used (option `fitType="locfit"` to *estimateDispersions*) a numerical integration is used instead. The transformed data should be approximated variance stabilized and also includes correction for size factors or normalization factors. The transformed data is on the log2 scale for large counts.

Regularized log transformation

The function *rlog*, stands for *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 *DESeq* and *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), β_{i0} is an intercept which does not undergo shrinkage, and β_{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 *rlog*.

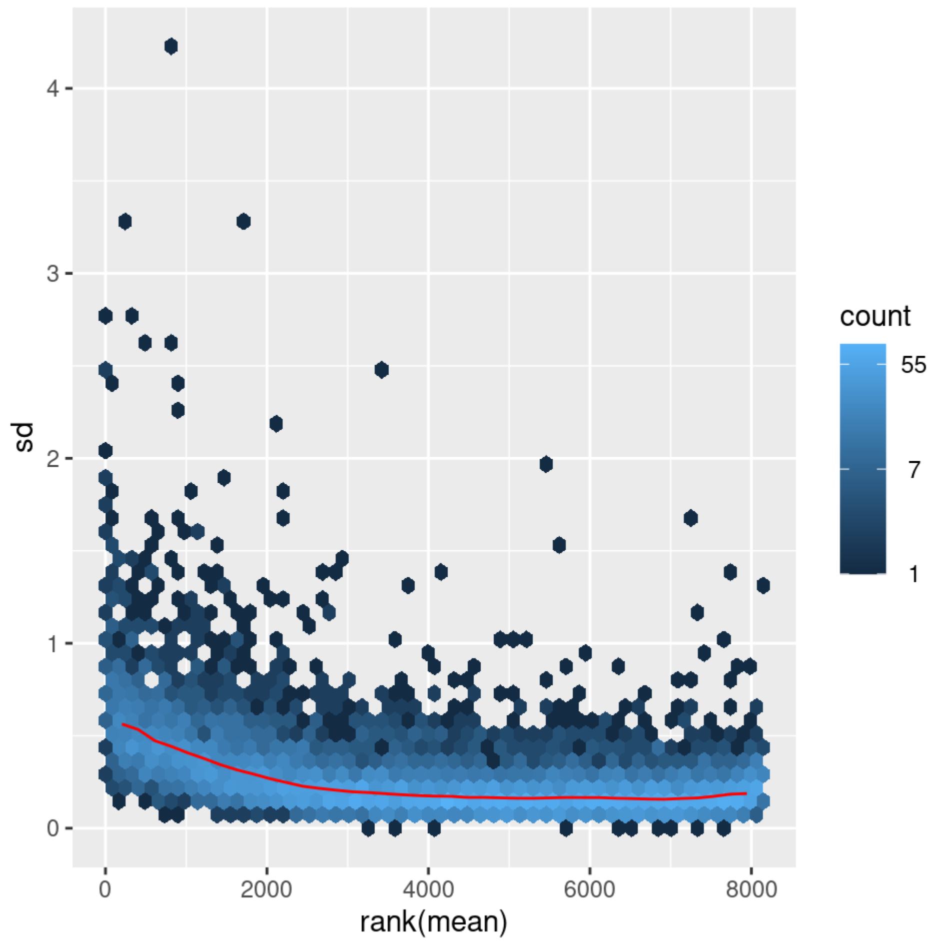
Note that, as q_{ij} represents the part of the mean value μ_{ij} after the size factor s_j has been divided out, it is clear that the rlog transformation inherently accounts for differences in sequencing depth. Without priors, this design matrix would lead to a non-unique solution, however the addition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

Effects of transformations on the variance

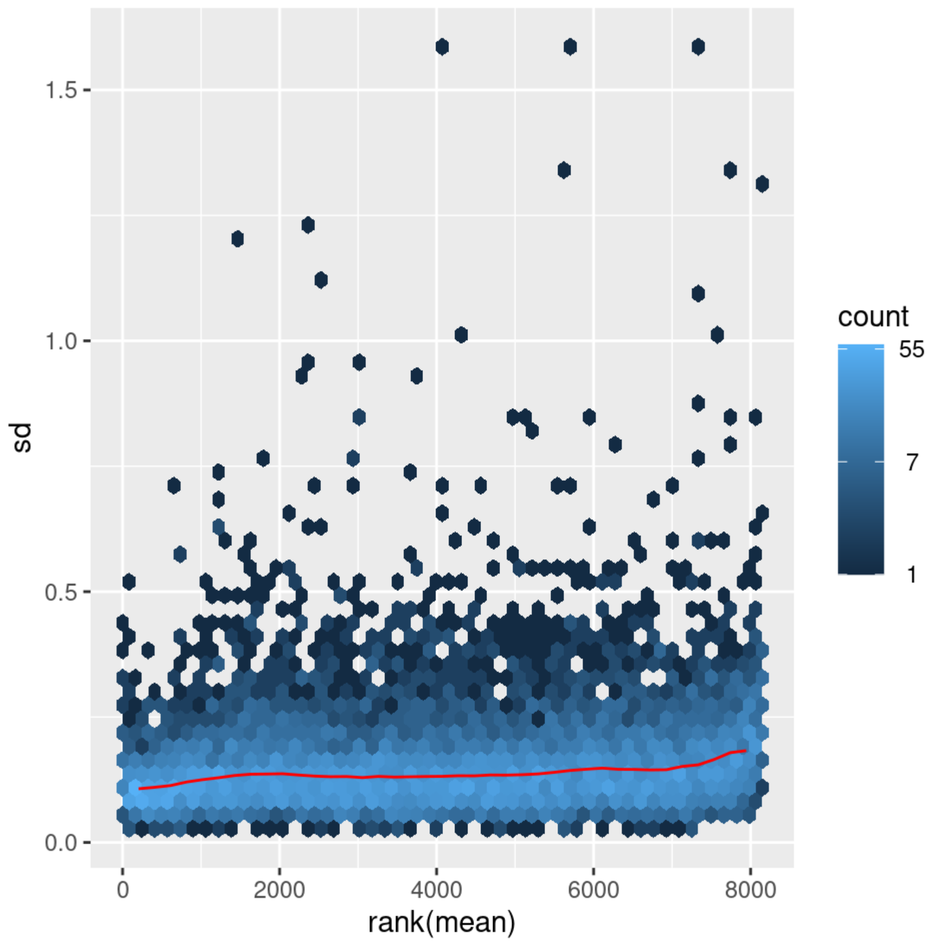
The figure below plots the standard deviation of the transformed data, across samples, against the mean, using the shifted logarithm transformation, the regularized log transformation and the variance stabilizing transformation. The shifted logarithm has elevated standard deviation in the lower count 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.

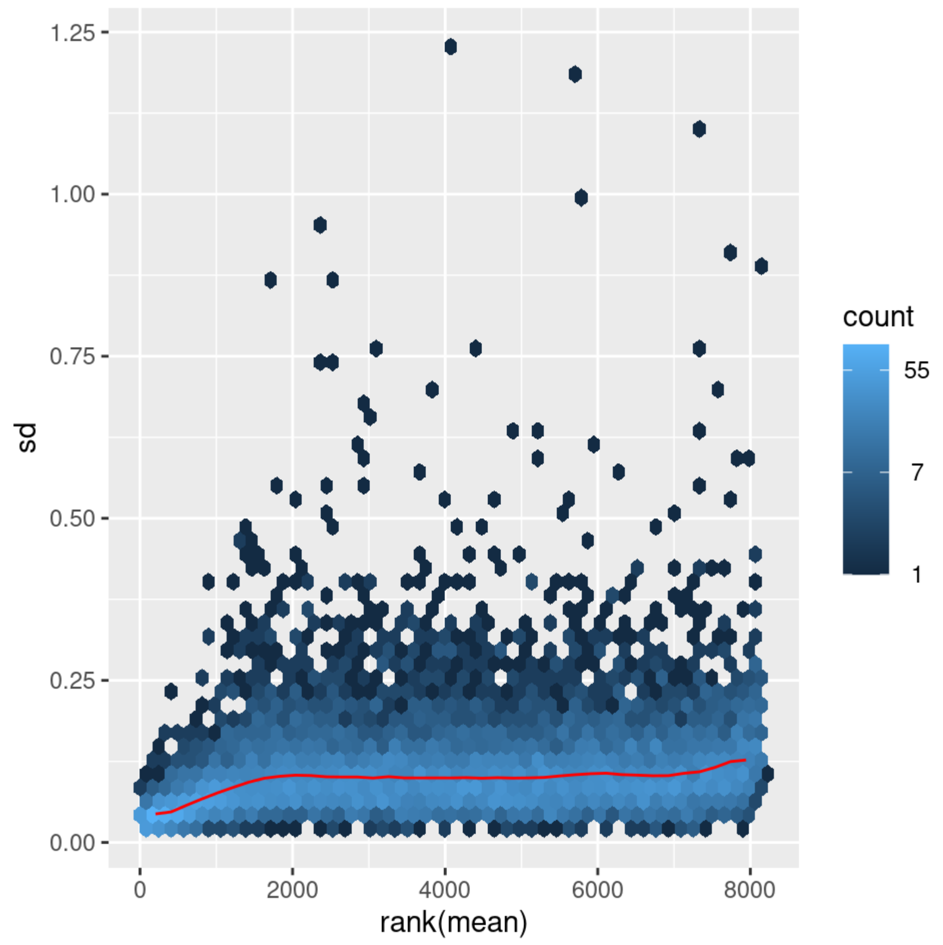
```
# this gives log2(n + 1)
ntd <- normTransform(dds)
library("vsn")
meanSdPlot(assay(ntd))
```



```
meanSdPlot(assay(vsd))
```



```
meanSdPlot(assay(rld))
```



Data quality assessment by sample clustering and visualization

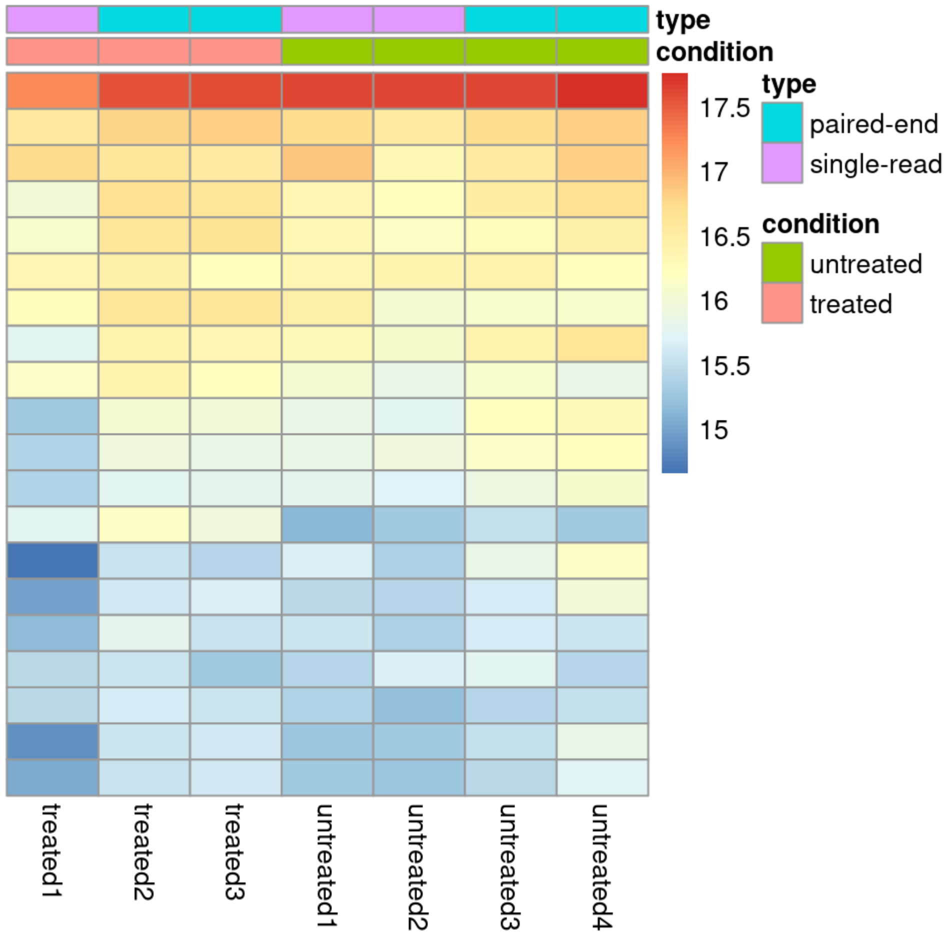
Data quality assessment and quality control (i.e. the removal of insufficiently good data) are essential steps of any data analysis. These steps should typically be performed very early in the analysis of a new data set, preceding or in parallel to the differential expression testing.

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 anormality that renders the data points obtained from these particular samples detrimental to our purpose.

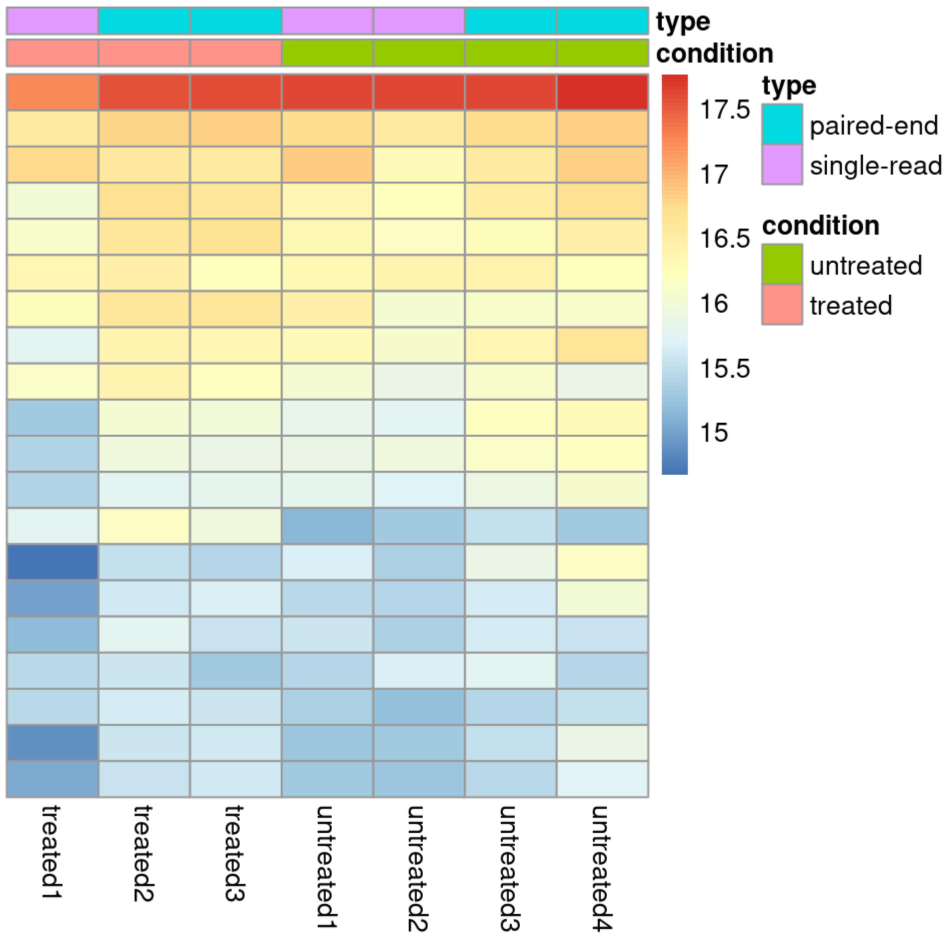
Heatmap of the count matrix

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.

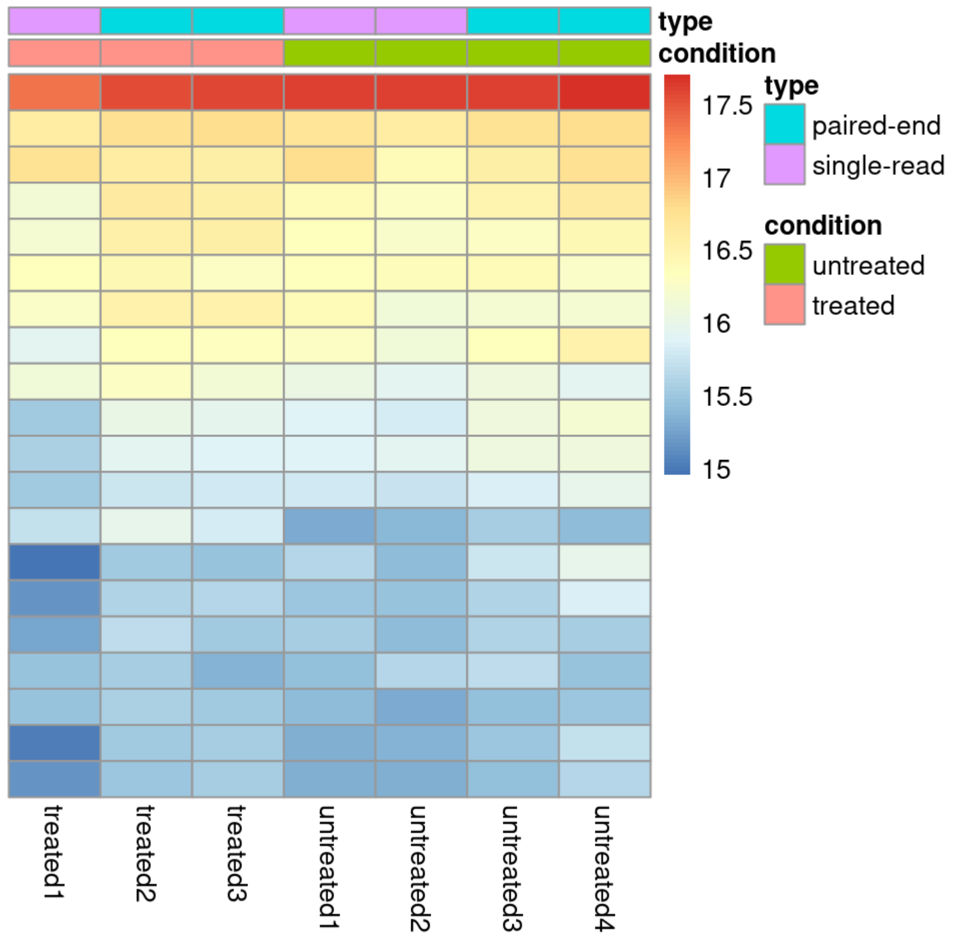
```
library("pheatmap")
select <- order(rowMeans(counts(dds,normalized=TRUE)),
                decreasing=TRUE)[1:20]
df <- as.data.frame(colData(dds)[,c("condition","type")])
pheatmap(assay(ntd)[select,], cluster_rows=FALSE, show_rownames=FALSE,
         cluster_cols=FALSE, annotation_col=df)
```



```
pheatmap(assay(vsd)[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)
```



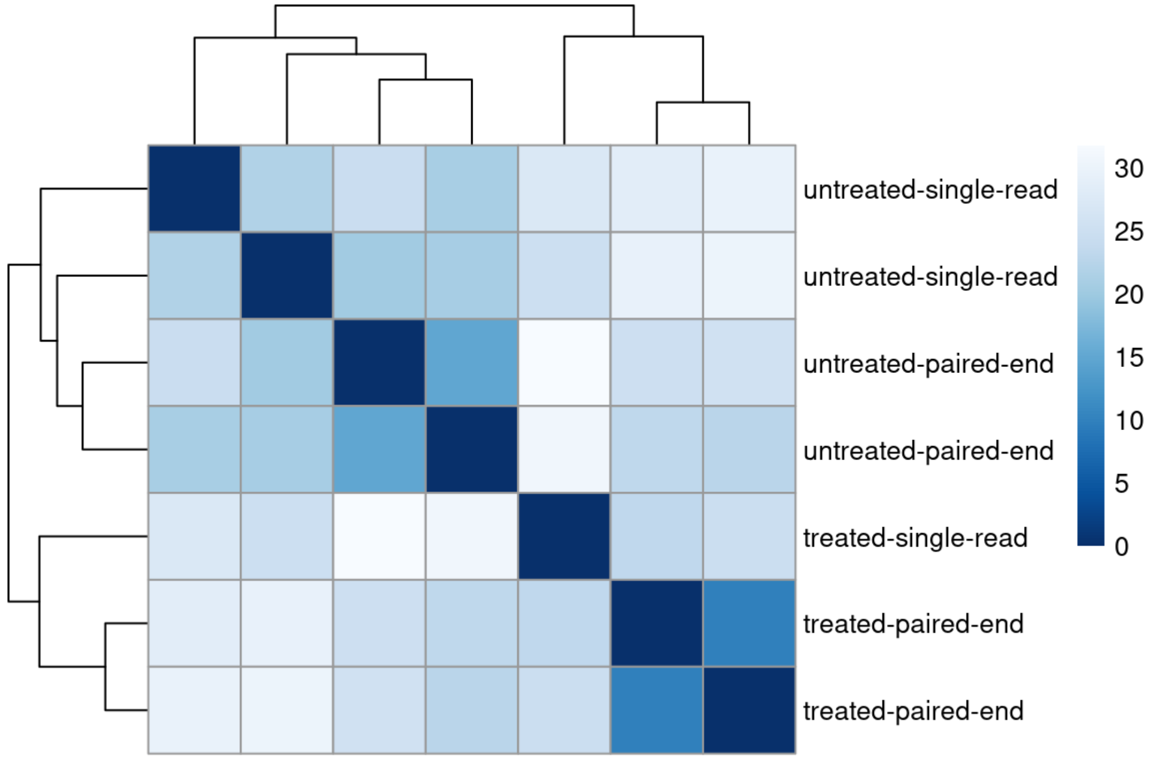
Heatmap of the sample-to-sample distances

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.

```
sampleDists <- dist(t(assay(vsd)))
```

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.

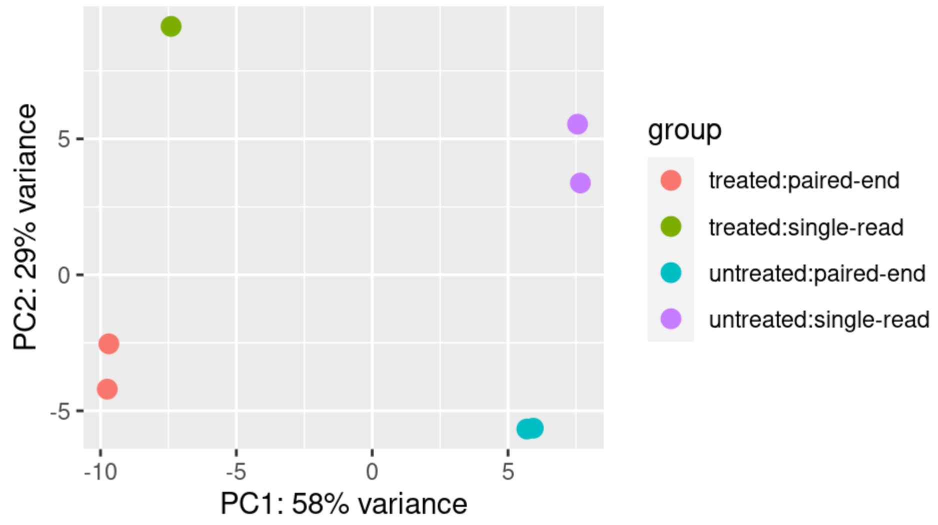
```r
library("RColorBrewer")
sampleDistMatrix <- as.matrix(sampleDists)
rownames(sampleDistMatrix) <- paste(vsd$condition, vsd$type, sep="-")
colnames(sampleDistMatrix) <- NULL
colors <- colorRampPalette( rev(brewer.pal(9, "Blues")) )(255)
pheatmap(sampleDistMatrix,
         clustering_distance_rows=sampleDists,
         clustering_distance_cols=sampleDists,
         col=colors)
```



Principal component plot of the samples

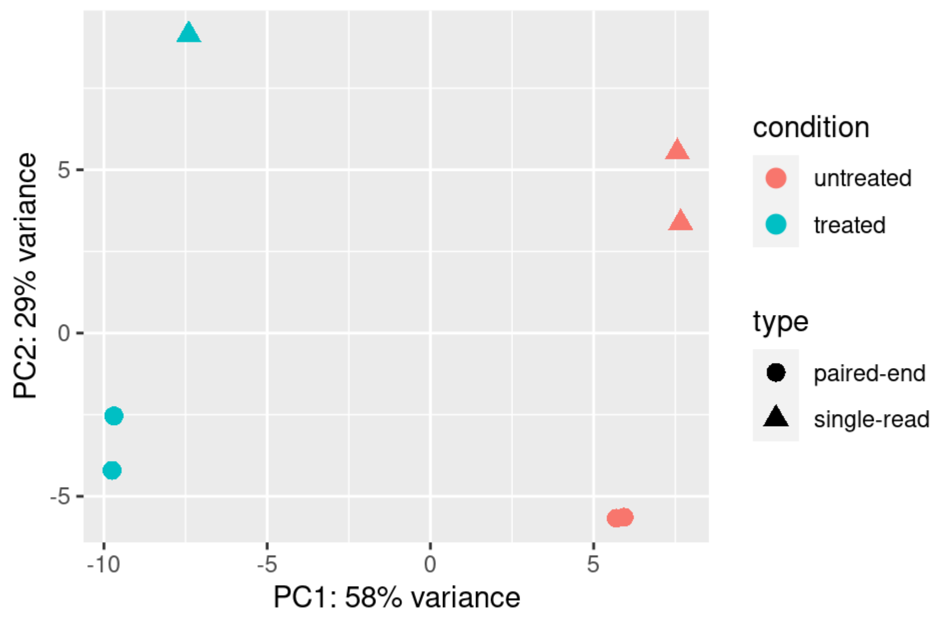
Related to the distance matrix is the PCA plot, which shows the samples in the 2D plane spanned by their first two principal components. This type of plot is useful for visualizing the overall effect of experimental covariates and batch effects.

```r
plotPCA(vsd, intgroup=c("condition", "type"))
```



It is also possible to customize the PCA plot using the *ggplot* function.

```
pcaData <- plotPCA(vsd, 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()
```



Variations to the standard workflow

Wald test individual steps

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

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

Control features for estimating size factors

In some experiments, it may not be appropriate to assume that a minority of features (genes) are affected greatly by the condition, such that the standard median-ratio method for estimating the size factors will not provide correct inference (the log fold changes for features that were truly un-changing will not centered on zero). This is a difficult inference problem for any method, but there is an important feature that can be used: the `controlGenes` argument of `estimateSizeFactors` . If there is any prior information about features (genes) that should not be changing with respect to the condition, providing this set of features to `controlGenes` will ensure that the log fold changes for these features will be centered around 0. The paradigm then becomes:

```
dds <- estimateSizeFactors(dds, controlGenes=ctrlGenes)
dds <- DESeq(dds)
```

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:

```
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. A tutorial (https://github.com/tavareshugo/tutorial_DESeq2_contrasts) describing the use of numeric contrasts for DESeq2 explains a general approach to comparing across groups of samples. Demonstrations of the use of contrasts for various designs can be found in the examples section of the help page `?results` . The mathematical formula that is used to generate the contrasts can be found below.

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.

Initial note: 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

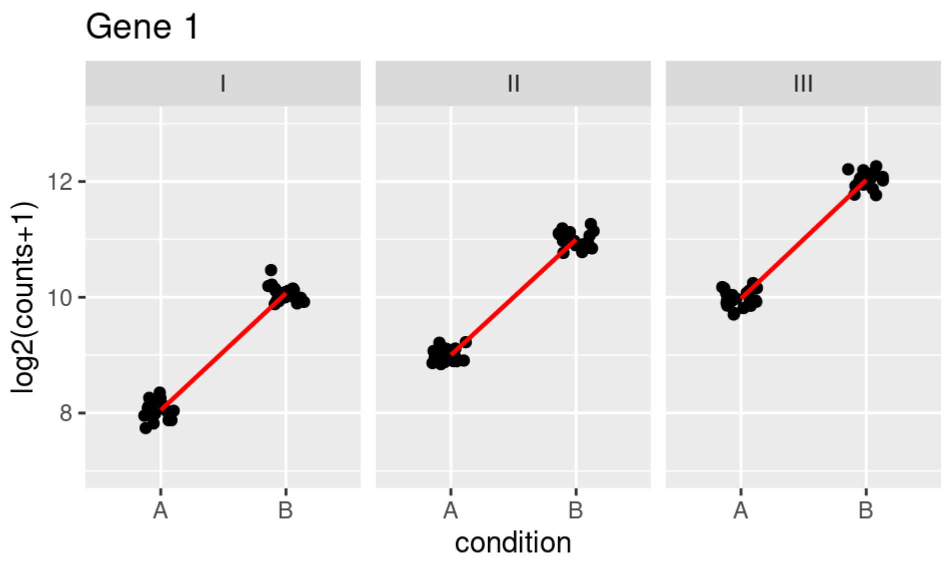
Using this design is similar to adding an interaction term, in that it models multiple condition effects which can be easily extracted with *results*. Suppose we have two factors `genotype` (with values I, II, and III) and `condition` (with values A and B), and we want to extract the condition effect specifically for each genotype. We could use the following approach to obtain, e.g. the condition effect for genotype I:

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

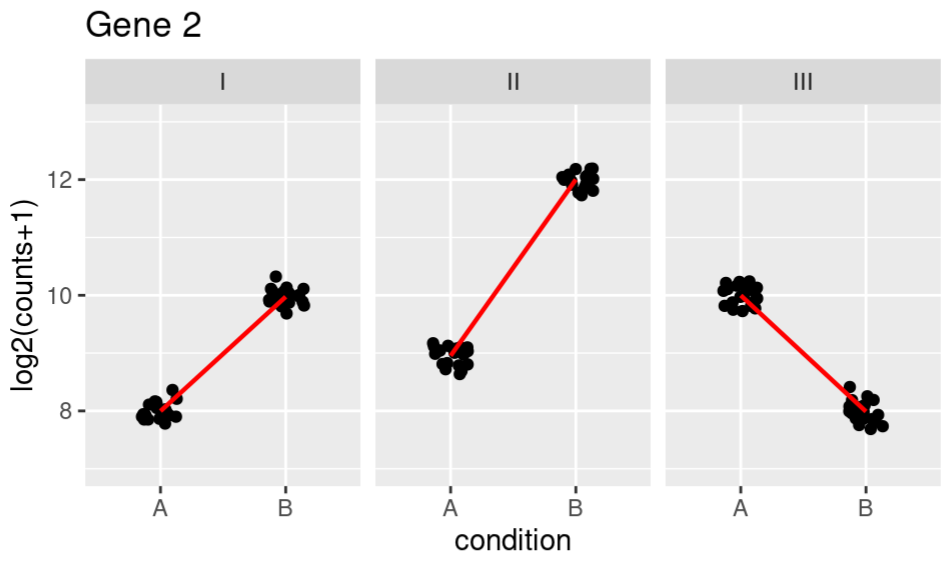
Adding interactions to the design: The following two plots diagram genotype-specific condition effects, which could be modeled with interaction terms by using a design of `~genotype + condition + genotype:condition`.

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.

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, once 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 (http://www.bioconductor.org/help/workflows/rnaseqGene).

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:

```
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:

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

Extended section on shrinkage estimators

Here we extend the discussion of shrinkage estimators. Below is a summary table of differences between methods available in `lfcShrink` via the `type` argument (and for further technical reference on use of arguments please see `?lfcShrink`):

method:	`apeglm` [1]	`ashr` [2]	`normal` [3]
Good for ranking by LFC	✓	✓	✓
Preserves size of large LFC	✓	✓	
Can compute *s-values* (Stephens 2016)	✓	✓	
Allows use of `coef`	✓	✓	✓
Allows use of `lfcThreshold`	✓	✓	✓
Allows use of `contrast`		✓	✓

method:	`apeglm` [1]	`ashr` [2]	`normal` [3]
Can shrink interaction terms	✓	✓	

References: 1. Zhu, Ibrahim, and Love (2018); 2. Stephens (2016); 3. Love, Huber, and Anders (2014)

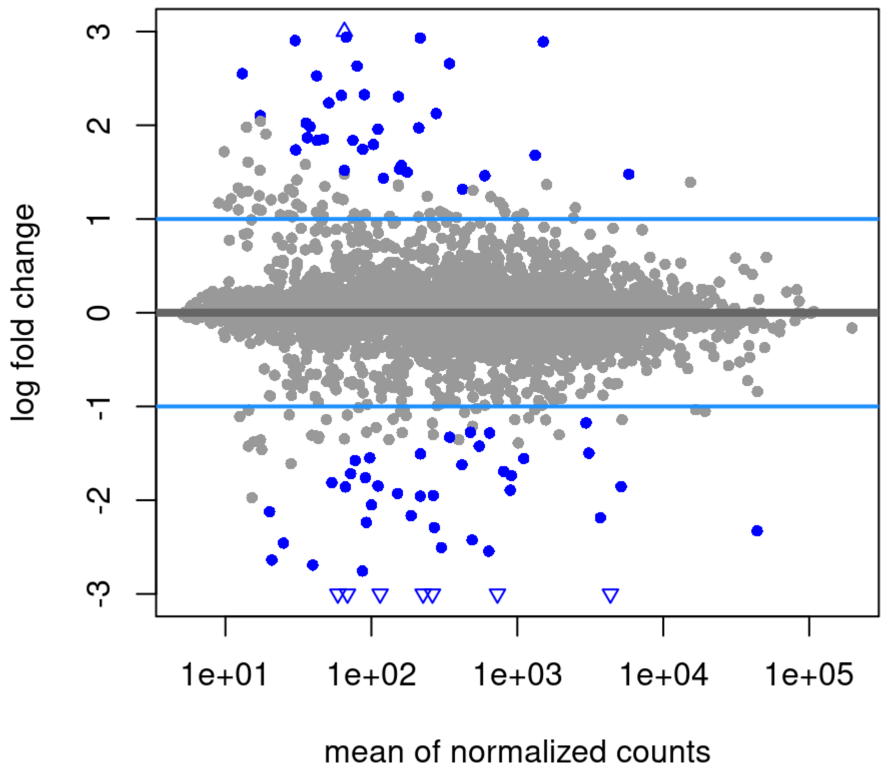
Beginning with the first row, all shrinkage methods provided by DESeq2 are good for ranking genes by “effect size”, that is the log2 fold change (LFC) across groups, or associated with an interaction term. It is useful to contrast ranking by effect size with ranking by a p-value or adjusted p-value associated with a null hypothesis: while increasing the number of samples will tend to decrease the associated p-value for a gene that is differentially expressed, the estimated effect size or LFC becomes more precise. Also, a gene can have a small p-value although the change in expression is not great, as long as the standard error associated with the estimated LFC is small.

The next two rows point out that `apeglm` and `ashr` shrinkage methods help to preserve the size of large LFC, and can be used to compute *s-values*. These properties are related. As noted in the previous section, the original DESeq2 shrinkage estimator used a Normal distribution, with a scale that adapts to the spread of the observed LFCs. Because the tails of the Normal distribution become thin relatively quickly, it was important when we designed the method that the prior scaling is sensitive to the very largest observed LFCs. As you can read in the DESeq2 paper, under the section, “*Empirical prior estimate*”, we used the top 5% of the LFCs by absolute value to set the scale of the Normal prior (we later added weighting the quantile by precision). `ashr` , published in 2016, and `apeglm` use wide-tailed priors to avoid shrinking large LFCs. While a typical RNA-seq experiment may have many LFCs between -1 and 1, we might consider a LFC of >4 to be very large, as they represent 16-fold increases or decreases in expression. `ashr` and `apeglm` can adapt to the scale of the entirety of LFCs, while not over-shrinking the few largest LFCs. The potential for over-shrinking LFC is also why DESeq2’s shrinkage estimator is not recommended for designs with interaction terms.

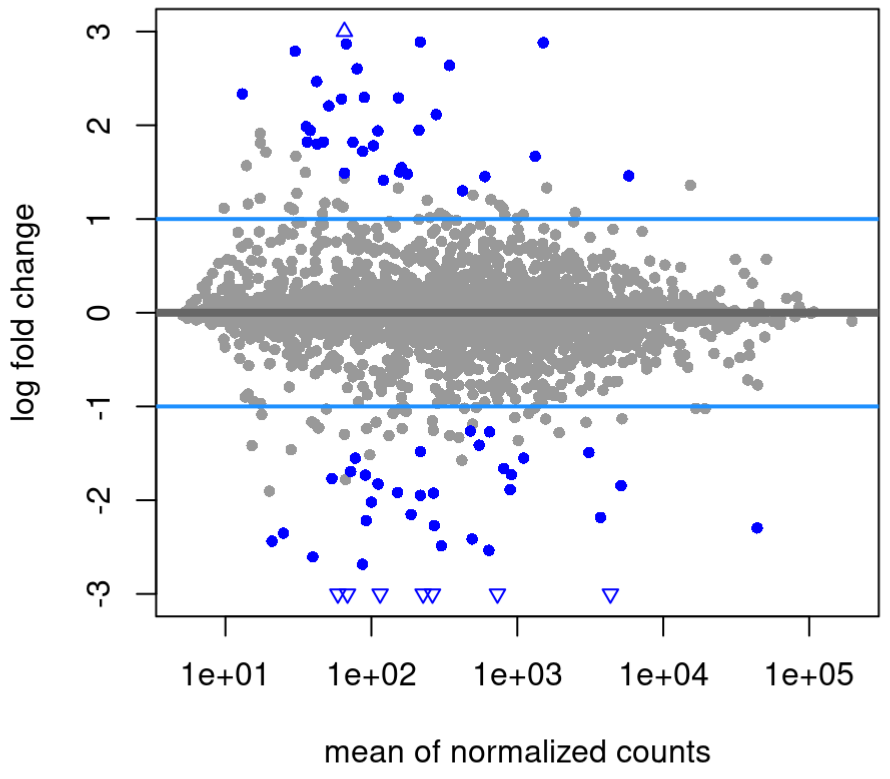
What are *s-values*? This quantity proposed by Stephens (2016) gives the estimated rate of *false sign* among genes with equal or smaller s-value. Stephens (2016) points out they are analogous to the *q*-value of Storey (2003). The s-value has a desirable property relative to the adjusted p-value or *q*-value, in that it does not require supposing there to be a set of null genes with LFC = 0 (the most commonly used null hypothesis). Therefore, it can be benchmarked by comparing estimated LFC and s-value to the “true LFC” in a setting where this can be reasonably defined. For these estimated probabilities to be accurate, the scale of the prior needs to match the scale of the distribution of effect sizes, and so the original DESeq2 shrinkage method is not really compatible with computing s-values.

The last four rows explain differences in whether coefficients or contrasts can have shrinkage applied by the various methods. All three methods can use `coef` with either the name or numeric index from `resultsNames(dds)` to specify which coefficient to shrink. All three methods allow for a positive `lfcThreshold` to be specified, in which case, they will return p-values and adjusted p-values or s-values for the LFC being greater in absolute value than the threshold (see this section for `normal`). For `apeglm` and `ashr` , setting a threshold means that the s-values will give the “false sign or small” rate (FSOS) among genes with equal or small s-value. We found FSOS to be a useful description for when the LFC is either the wrong sign or less than the threshold distance from 0.

```
resApeT <- lfcShrink(dds, coef=2, type="apeglm", lfcThreshold=1)
plotMA(resApeT, ylim=c(-3,3), cex=.8)
abline(h=c(-1,1), col="dodgerblue", lwd=2)
```



```
resAshT <- lfcShrink(dds, coef=2, type="ashr", lfcThreshold=1)
plotMA(resAshT, ylim=c(-3,3), cex=.8)
abline(h=c(-1,1), col="dodgerblue", lwd=2)
```



Finally, `normal` and `ashr` can be used with arbitrary specified `contrast` because `normal` shrinks multiple coefficients simultaneously (`apeglm` does not), and because `ashr` does not estimate a vector of coefficients but models estimated coefficients and their standard errors from upstream methods (here, DESeq2's MLE). Although `apeglm` cannot be used with `contrast` , we note that many designs can be easily rearranged such that what was a contrast becomes its own coefficient. In this case, the dispersion does not have to be estimated again, as the designs are equivalent, up to the meaning of the coefficients. Instead, one need only run `nbinomWaldTest` to re-estimate MLE coefficients – these are necessary for `apeglm` – and then run `lfcShrink` specifying the coefficient of interest in `resultsNames(dds)` .

We give some examples below of producing equivalent designs for use with `coef` . We show how the coefficients change with `model.matrix` , but the user would, for example, either change the levels of `dds$condition` or replace the design using `design(dds)<-` , then run `nbinomWaldTest` followed by `lfcShrink` .

Three groups:

```
condition <- factor(rep(c("A","B","C"),each=2))
model.matrix(~ condition)
```

```
##   (Intercept) conditionB conditionC
## 1           1          0          0
## 2           1          0          0
## 3           1          1          0
## 4           1          1          0
## 5           1          0          1
## 6           1          0          1
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$condition
## [1] "contr.treatment"
```

```
# to compare C vs B, make B the reference level,
# and select the last coefficient
condition <- relevel(condition, "B")
model.matrix(~ condition)
```

```
##   (Intercept) conditionA conditionC
## 1           1          1          0
## 2           1          1          0
## 3           1          0          0
## 4           1          0          0
## 5           1          0          1
## 6           1          0          1
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$condition
## [1] "contr.treatment"
```

Three groups, compare condition effects:

```
grp <- factor(rep(1:3,each=4))
cnd <- factor(rep(rep(c("A","B"),each=2),3))
model.matrix(~ grp + cnd + grp:cnd)
```

```
##    (Intercept) grp2 grp3 cndB grp2:cndB grp3:cndB
## 1            1    0    0    0         0         0
## 2            1    0    0    0         0         0
## 3            1    0    0    1         0         0
## 4            1    0    0    1         0         0
## 5            1    1    0    0         0         0
## 6            1    1    0    0         0         0
## 7            1    1    0    1         1         0
## 8            1    1    0    1         1         0
## 9            1    0    1    0         0         0
## 10           1    0    1    0         0         0
## 11           1    0    1    1         0         1
## 12           1    0    1    1         0         1
## attr(,"assign")
## [1] 0 1 1 2 3 3
## attr(,"contrasts")
## attr(,"contrasts")$grp
## [1] "contr.treatment"
##
## attr(,"contrasts")$cnd
## [1] "contr.treatment"
```

```
# to compare condition effect in group 3 vs 2,
# make group 2 the reference level,
# and select the last coefficient
grp <- relevel(grp, "2")
model.matrix(~ grp + cnd + grp:cnd)
```

```
##    (Intercept) grp1 grp3 cndB grp1:cndB grp3:cndB
## 1            1    1    0    0         0         0
## 2            1    1    0    0         0         0
## 3            1    1    0    1         1         0
## 4            1    1    0    1         1         0
## 5            1    0    0    0         0         0
## 6            1    0    0    0         0         0
## 7            1    0    0    1         0         0
## 8            1    0    0    1         0         0
## 9            1    0    1    0         0         0
## 10           1    0    1    0         0         0
## 11           1    0    1    1         0         1
## 12           1    0    1    1         0         1
## attr(,"assign")
## [1] 0 1 1 2 3 3
## attr(,"contrasts")
## attr(,"contrasts")$grp
## [1] "contr.treatment"
##
## attr(,"contrasts")$cnd
## [1] "contr.treatment"
```

Two groups, two individuals per group, compare within-individual condition effects:

```
grp <- factor(rep(1:2,each=4))
ind <- factor(rep(rep(1:2,each=2),2))
cnd <- factor(rep(c("A","B"),4))
model.matrix(~grp + grp:ind + grp:cnd)
```

```
##   (Intercept) grp2 grp1:ind2 grp2:ind2 grp1:cndB grp2: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
## [1] "contr.treatment"
##
## attr(,"contrasts")$cnd
## [1] "contr.treatment"
```

```
# to compare condition effect across group,
# add a main effect for 'cnd',
# and select the last coefficient
model.matrix(~grp + cnd + grp:ind + grp:cnd)
```

```
##   (Intercept) grp2 cndB grp1:ind2 grp2:ind2 grp2:cndB
## 1           1    0    0         0         0         0
## 2           1    0    1         0         0         0
## 3           1    0    0         1         0         0
## 4           1    0    1         1         0         0
## 5           1    1    0         0         0         0
## 6           1    1    1         0         0         1
## 7           1    1    0         0         1         0
## 8           1    1    1         0         1         1
## attr(,"assign")
## [1] 0 1 2 3 3 4
## attr(,"contrasts")
## attr(,"contrasts")$grp
## [1] "contr.treatment"
##
## attr(,"contrasts")$cnd
## [1] "contr.treatment"
##
## attr(,"contrasts")$ind
## [1] "contr.treatment"
```

Recommendations for single-cell analysis

The DESeq2 developers and collaborating groups have published recommendations for the best use of DESeq2 for single-cell datasets, which have been described first in Van den Berge et al. (2018). Default values for DESeq2 were designed for bulk data and will not be appropriate for single-cell datasets. These settings and additional improvements have also been tested subsequently and published in Zhu, Ibrahim, and Love (2018) and Ahlmann-Eltze and Huber (2020).

- Use `test="LRT"` for significance testing when working with single-cell data, over the Wald test. This has been observed across multiple single-cell benchmarks.
- Set the following `DESeq` arguments to these values: `useT=TRUE` , `minmu=1e-6` , and `minReplicatesForReplace=Inf` . The default setting of `minmu` was benchmarked on bulk RNA-seq and is not appropriate for single cell data when the expected count is often much less than 1.
- The default size factors are not optimal for single cell count matrices, instead consider setting `sizeFactors` from `scran::computeSumFactors` .
- One important concern for single-cell data analysis is the size of the datasets and associated processing time. To address the speed concerns, *DESeq2* provides an interface to glmGamPoi (https://bioconductor.org/packages/glmGamPoi/), which implements faster dispersion and parameter estimation routines for single-cell data (Ahlmann-Eltze and Huber 2020). To use this feature, set `fitType = "glmGamPoi"` . Alternatively, one can use *glmGamPoi* as a standalone package. This provides the additional option to process data on-disk if the full dataset does not fit in memory, a quasi-likelihood framework for differential testing, and the ability to form pseudobulk samples (more details how to use *glmGamPoi* are in its README (https://github.com/const-ae/glmGamPoi)).

Optionally, one can consider using the zinbwave (https://bioconductor.org/packages/zinbwave) package to directly model the zero inflation of the counts, and take account of these in the DESeq2 model. This allows for the DESeq2 inference to apply to the part of the data which is not due to zero inflation. Not all single cell datasets exhibit zero inflation, and instead may just reflect low conditional estimated counts (conditional on cell type or cell state).There is example code for combining *zinbwave* and *DESeq2* package functions in the *zinbwave* vignette. We also have an example of ZINB-WaVE + DESeq2 integration using the splatter (https://bioconductor.org/packages/splatter) package for simulation at the zinbwave-deseq2 (https://github.com/mikelove/zinbwave-deseq2) GitHub repository.

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"]]` .

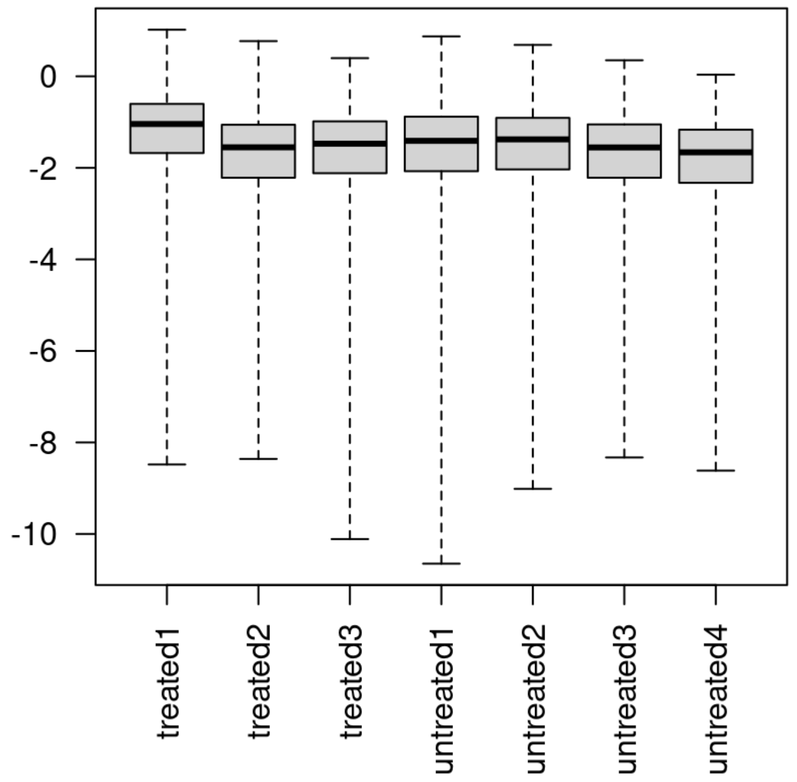
The *results* function automatically flags genes which contain a Cook's distance above a cutoff for samples which have 3 or more replicates. The *p* values and adjusted *p* values for these genes are set to `NA` . At least 3 replicates are required for flagging, as it is difficult to judge which sample 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):

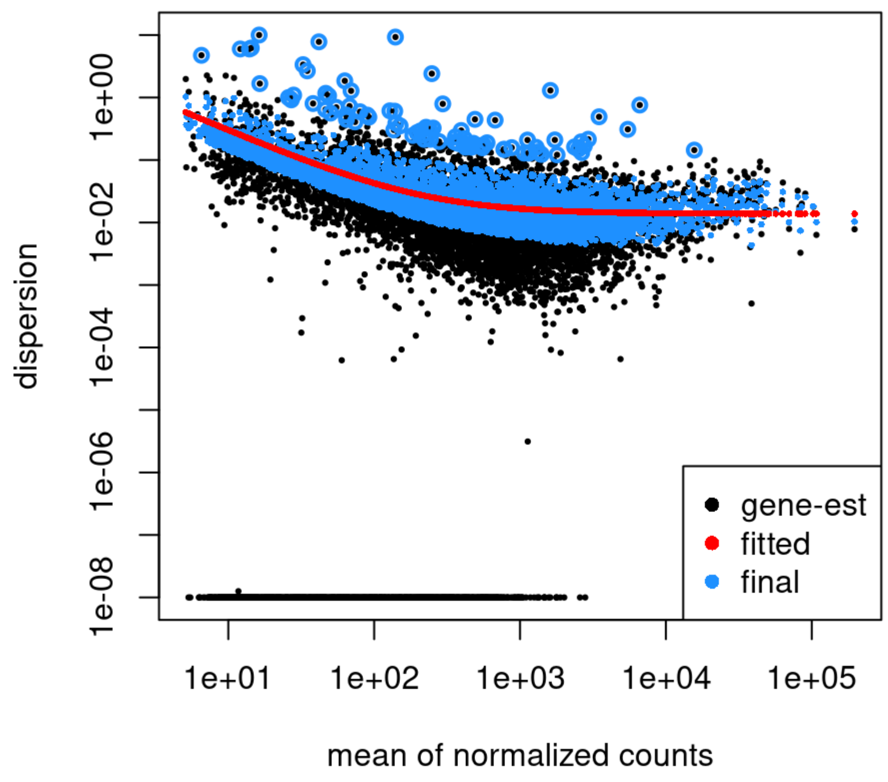
```
par(mar=c(8,5,2,2))
boxplot(log10(assays(dds)[["cooks"]]), range=0, las=2)
```



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.

```
plotDispEsts(dds)
```



Local or mean dispersion fit

A local smoothed dispersion fit is automatically substitited 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 disperion estimates as the fitted value can be specified by providing the argument `fitType="mean"` .

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.

```
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)
```

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 α). 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 `NA`.

The default independent filtering is performed using the *filtered_p* function of the genefilter (http://bioconductor.org/packages/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:

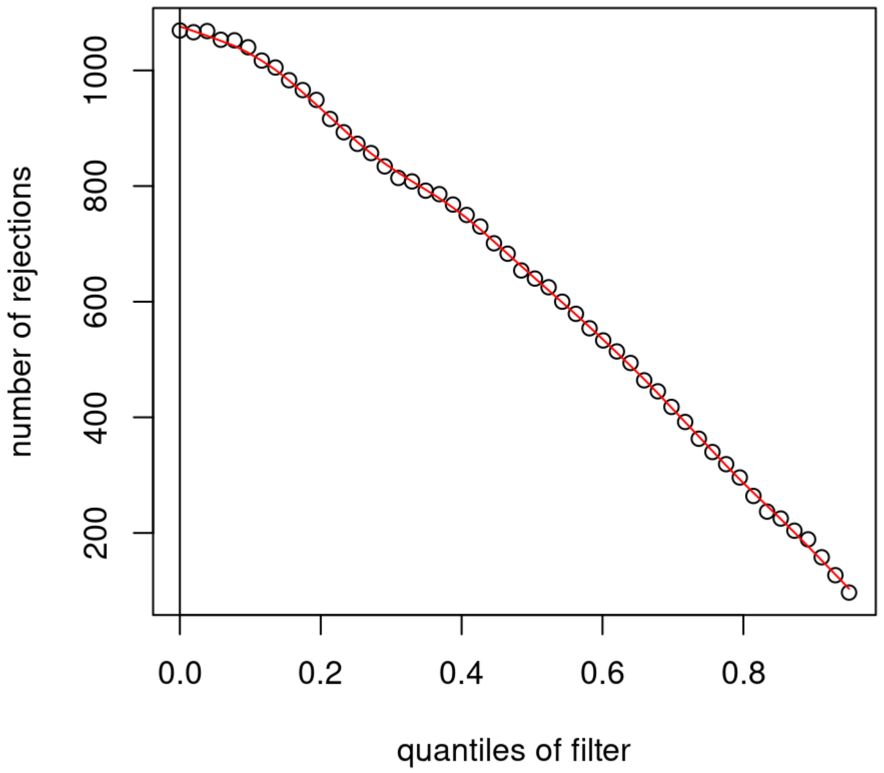
```
metadata(res)$alpha
```

```
## [1] 0.1
```

```
metadata(res)$filterThreshold
```

```
##       0%
## 5.109586
```

```
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` .

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

```
##          noFiltering
## filtering FALSE TRUE  Sum
##     FALSE  7079    0 7079
##     TRUE      0 1069 1069
##     Sum    7079 1069 8148
```

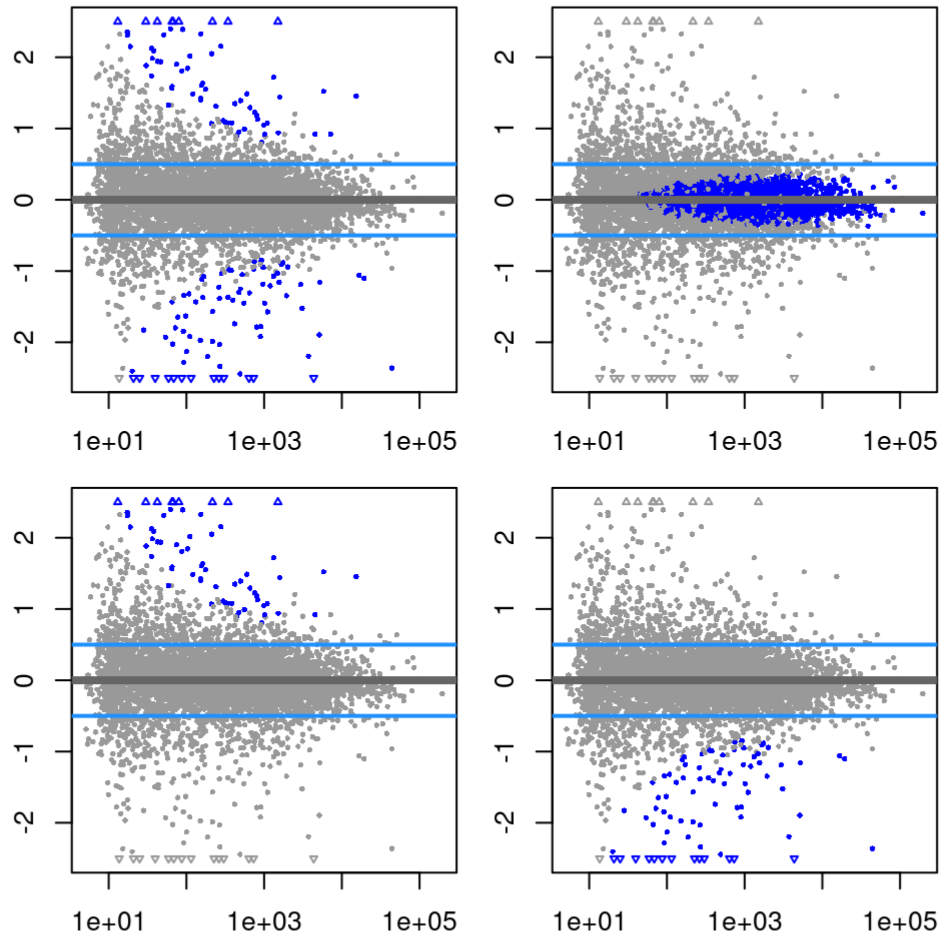
Tests of log2 fold change above or below a threshold

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 β 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 four possible values of `altHypothesis` are demonstrated in the following code and visually by MA-plots in the following figures.

```
par(mfrow=c(2,2),mar=c(2,2,1,1))
ylim <- c(-2.5,2.5)
resGA <- results(dds, lfcThreshold=.5, altHypothesis="greaterAbs")
resLA <- results(dds, lfcThreshold=.5, altHypothesis="lessAbs")
resG <- results(dds, lfcThreshold=.5, altHypothesis="greater")
resL <- results(dds, lfcThreshold=.5, altHypothesis="less")
drawLines <- function() abline(h=c(-.5,.5),col="dodgerblue",lwd=2)
plotMA(resGA, ylim=ylim); drawLines()
plotMA(resLA, ylim=ylim); drawLines()
plotMA(resG, ylim=ylim); drawLines()
plotMA(resL, ylim=ylim); drawLines()
```



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.

```
mcols(dds,use.names=TRUE)[1:4,1:4]
```

```
## DataFrame with 4 rows and 4 columns
##                    gene   baseMean     baseVar   allZero
##             <character>  <numeric>   <numeric> <logical>
## FBgn0000008 FBgn0000008   95.28865 2.29337e+02     FALSE
## FBgn0000017 FBgn0000017 4359.09632 3.71585e+05     FALSE
## FBgn0000018 FBgn0000018  419.06811 2.24013e+03     FALSE
## FBgn0000024 FBgn0000024    6.41105 3.74104e+00     FALSE
```

```
substr(names(mcols(dds)),1,10)
```

```
##  [1] "gene"       "baseMean"   "baseVar"    "allZero"    "dispGeneEs"
##  [6] "dispGeneIt" "dispFit"    "dispersion" "dispIter"   "dispOutlie"
## [11] "dispMAP"    "Intercept"  "condition_" "SE_Interce" "SE_conditi"
## [16] "WaldStatis" "WaldStatis" "WaldPvalue" "WaldPvalue" "betaConv"
## [21] "betaIter"   "deviance"   "maxCooks"
```

```
mcols(mcols(dds), use.names=TRUE)[1:4,]
```

```
## DataFrame with 4 rows and 2 columns
##                  type            description
##           <character>            <character>
## gene
## baseMean intermediate mean of normalized c..
## baseVar  intermediate variance of normaliz..
## allZero  intermediate all counts for a gen..
```

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:

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

```
##               treated1    treated2    treated3 untreated1 untreated2
## FBgn0000008  154.18297   72.011890   78.553988  107.92325  169.00095
## FBgn0000017 6441.09688 3008.344869 3281.645427 5333.50912 8351.93629
## FBgn0000018  657.53926  307.106832  335.006715  495.29423  775.59928
## FBgn0000024   11.43421    5.340404    5.825567    6.91794   10.83305
## FBgn0000032 1559.80292  728.513359  794.696964 1164.47564 1823.49484
## FBgn0000037   27.23930   12.722244   13.878028   13.84125   21.67451
##              untreated3  untreated4
## FBgn0000008   60.947217   70.828452
## FBgn0000017 3011.978798 3500.304091
## FBgn0000018  279.706228  325.054365
## FBgn0000024    3.906750    4.540143
## FBgn0000032  657.611313  764.228344
## FBgn0000037    7.816532    9.083808
```

```
head(assays(dds)[["cooks"]])
```

```
##               treated1    treated2    treated3 untreated1   untreated2
## FBgn0000008 0.08564565 0.297964299 0.077021521 0.10556148 0.0135846198
## FBgn0000017 0.01275474 0.004120273 0.002353591 0.08760807 0.0105562769
## FBgn0000018 0.09813824 0.005595703 0.054059461 0.17277269 0.0021395029
## FBgn0000024 0.06482596 0.129406896 0.030982961 0.26445639 0.0005741831
## FBgn0000032 0.07625620 0.017481712 0.020056352 0.32377428 0.0211262017
## FBgn0000037 0.45819676 0.026766117 0.151000426 0.01530776 0.0859061169
##             untreated3   untreated4
## FBgn0000008 0.19761575 0.0004998745
## FBgn0000017 0.18275047 0.0548693390
## FBgn0000018 0.07090752 0.0104402584
## FBgn0000024 0.02968291 0.0809265573
## FBgn0000032 0.02151283 0.0764805449
## FBgn0000037 0.02257542 0.2489127110
```

The dispersions α_i can be accessed with the *dispersions* function.

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

```
## [1] 0.03082228 0.01305447 0.01518649 0.23983770 0.01654718 0.12963629
```

```
head(mcols(dds)$dispersion)
```

```
## [1] 0.03082228 0.01305447 0.01518649 0.23983770 0.01654718 0.12963629
```

The size factors s_j are accessible via *sizeFactors*:

```
sizeFactors(dds)
```

```
##   treated1   treated2   treated3 untreated1 untreated2 untreated3 untreated4
##  1.6297067  0.7611622  0.8303119  1.1439041  1.7912811  0.6459940  0.7507275
```

For advanced users, we also include a convenience function *coef* for extracting the matrix $[\beta_{ir}]$ 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  6.559896                     0.00399148
## FBgn0000017 12.186903                    -0.23842494
## FBgn0000018  8.758176                    -0.10185506
## FBgn0000024  2.596376                     0.21429657
## FBgn0000032  9.991499                    -0.08896298
## FBgn0000037  3.596936                     0.46606939
```

The beta prior variance σ_r^2 is stored as an attribute of the *DESeqDataSet*:

```
attr(dds, "betaPriorVar")
```

```
## [1] 1e+06 1e+06
```

General information about the prior used for log fold change shrinkage is also stored in a slot of the *DESeqResults* object. This would also contain information about what other packages were used for log2 fold change shrinkage.

```
priorInfo(resLFC)
```

```
## $type
## [1] "apeglm"
##
## $package
## [1] "apeglm"
##
## $version
## [1] '1.23.1'
##
## $prior.control
## $prior.control$no.shrink
## [1] 1
##
## $prior.control$prior.mean
## [1] 0
##
## $prior.control$prior.scale
## [1] 0.2031588
##
## $prior.control$prior.df
## [1] 1
##
## $prior.control$prior.no.shrink.mean
## [1] 0
##
## $prior.control$prior.no.shrink.scale
## [1] 15
##
## $prior.control$prior.var
## [1] 0.0412735
```

```
priorInfo(resNorm)
```

```
## $type
## [1] "normal"
##
## $package
## [1] "DESeq2"
##
## $version
## [1] '1.41.12'
##
## $betaPriorVar
##        Intercept conditiontreated
##     1.000000e+06     1.049206e-01
```

```
priorInfo(resAsh)
```

```
## $type
## [1] "ashr"
##
## $package
## [1] "ashr"
##
## $version
## [1] '2.2.63'
##
## $fitted_g
## $pi
##  [1] 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.128392365
##  [7] 0.000000000 0.452702478 0.057818359 0.000000000 0.176425359 0.000000000
## [13] 0.120967167 0.027145784 0.005428256 0.018807845 0.012312387 0.000000000
## [19] 0.000000000 0.000000000 0.000000000 0.000000000
##
## $mean
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##
## $sd
##  [1] 0.006752731 0.009549803 0.013505461 0.019099607 0.027010923 0.038199213
##  [7] 0.054021845 0.076398426 0.108043690 0.152796852 0.216087381 0.305593704
## [13] 0.432174761 0.611187408 0.864349522 1.222374817 1.728699044 2.444749633
## [19] 3.457398088 4.889499267 6.914796176 9.778998533
##
## attr(,"class")
## [1] "normalmix"
## attr(,"row.names")
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
```

The dispersion prior variance σ_d^2 is stored as an attribute of the dispersion function:

```
dispersionFunction(dds)
```

```
## function (q)
## coefs[1] + coefs[2]/q
## <bytecode: 0x55f1736078a8>
## <environment: 0x55f173607250>
## attr(,"coefficients")
## asymptDisp  extraPois
## 0.01377513 2.88413302
## attr(,"fitType")
## [1] "parametric"
## attr(,"varLogDispEsts")
## [1] 1.025797
## attr(,"dispPriorVar")
## [1] 0.5354389
```

```
attr(dispersionFunction(dds), "dispPriorVar")
```

```
## [1] 0.5354389
```

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] '1.41.12'
```

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 x sample matrix, and *size factors* for a single number per sample. Incorporating normalization factors, the mean parameter μ_{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.

```
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 (http://bioconductor.org/packages/cqn) or EDASeq (http://bioconductor.org/packages/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:

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

“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:

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 β_i in the formula below) is not possible.

```
## DataFrame with 4 rows and 2 columns
##      batch condition
##   <factor>  <factor>
## 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.

```
## DataFrame with 6 rows and 2 columns
##      batch condition
##   <factor>  <factor>
## 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:

```
## DataFrame with 6 rows and 2 columns
##      batch condition
##   <factor>  <factor>
## 1        1         A
## 2        1         B
## 3        1         C
## 4        2         A
## 5        2         B
## 6        2         C
```

Group-specific condition effects, individuals nested within groups

Finally, there is a case where we *can* in fact perform inference, but we may need to re-arrange terms to do so. Consider an experiment with grouped individuals, where we seek to test the group-specific effect of a condition or treatment, while controlling for individual effects. The individuals are nested within the groups: an individual can only be in one of the groups, although each individual has one or more observations across condition.

An example of such an experiment is below:

```
coldata <- DataFrame(grp=factor(rep(c("X","Y"),each=6)),
                     ind=factor(rep(1:6,each=2)),
                     cnd=factor(rep(c("A","B"),6)))
coldata
```

```
## DataFrame with 12 rows and 3 columns
##          grp      ind      cnd
##     <factor> <factor> <factor>
## 1          X        1        A
## 2          X        1        B
## 3          X        2        A
## 4          X        2        B
## 5          X        3        A
## ...      ...      ...      ...
## 8          Y        4        B
## 9          Y        5        A
## 10         Y        5        B
## 11         Y        6        A
## 12         Y        6        B
```

Note that individual (`ind`) is a *factor* not a numeric. This is very important.

To make R display all the rows, we can do:

```
as.data.frame(coldata)
```

```
##    grp ind cnd
## 1    X   1   A
## 2    X   1   B
## 3    X   2   A
## 4    X   2   B
## 5    X   3   A
## 6    X   3   B
## 7    Y   4   A
## 8    Y   4   B
## 9    Y   5   A
## 10   Y   5   B
## 11   Y   6   A
## 12   Y   6   B
```

We have two groups of samples X and Y, each with three distinct individuals (labeled here 1-6). For each individual, we have conditions A and B (for example, this could be control and treated).

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 (http://bioconductor.org/packages/edgeR) user guide, from the section *Comparisons Both Between and Within Subjects*. If we try to analyze with a formula such as, `~ ind + grp*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 simple design, such as `~ ind + cnd`, as long as you remember to replace it before running *DESeq*. Then add a column `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`.

```
coldata$ind.n <- factor(rep(rep(1:3,each=2),2))
as.data.frame(coldata)
```

```
##    grp ind cnd ind.n
## 1    X   1   A     1
## 2    X   1   B     1
## 3    X   2   A     2
## 4    X   2   B     2
## 5    X   3   A     3
## 6    X   3   B     3
## 7    Y   4   A     1
## 8    Y   4   B     1
## 9    Y   5   A     2
## 10   Y   5   B     2
## 11   Y   6   A     3
## 12   Y   6   B     3
```

Now we can reassign our *DESeqDataSet* a design of `~ grp + grp:ind.n + grp:cnd`, before we call *DESeq*. This new design will result in the following model matrix:

```
model.matrix(~ grp + grp:ind.n + grp:cnd, coldata)
```

```
##    (Intercept) grpY grpX:ind.n2 grpY:ind.n2 grpX:ind.n3 grpY:ind.n3 grpX:cndB
## 1            1    0           0           0           0           0         0
## 2            1    0           0           0           0           0         1
## 3            1    0           1           0           0           0         0
## 4            1    0           1           0           0           0         1
## 5            1    0           0           0           1           0         0
## 6            1    0           0           0           1           0         1
## 7            1    1           0           0           0           0         0
## 8            1    1           0           0           0           0         0
## 9            1    1           0           1           0           0         0
## 10           1    1           0           1           0           0         0
## 11           1    1           0           0           0           1         0
## 12           1    1           0           0           0           1         0
##    grpY:cndB
## 1          0
## 2          0
## 3          0
## 4          0
## 5          0
## 6          0
## 7          0
## 8          1
## 9          0
## 10         1
## 11         0
## 12         1
## attr(,"assign")
## [1] 0 1 2 2 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"
```

Note that, if you have unbalanced numbers of individuals in the two groups, you will have zeros for some of the interactions between `grp` and `ind.n`. You can remove these columns manually from the model matrix and pass the corrected model matrix to the `full` argument of the *DESeq* function. See example code in the next section. Note that, in this case, you will not be able to create the *DESeqDataSet* with the design that leads to less than full rank model matrix. You can either use `design=~1` when creating the dataset object, or you can provide the corrected model matrix to the `design` slot of the dataset from the start.

Above, the terms `grpX.cndB` and `grpY.cndB` give the group-specific condition effects, in other words, the condition B vs A effect for group X samples, and likewise for group Y samples. These terms control for all of the six individual effects. These group-specific condition effects can be extracted using *results* with the `name` argument.

Furthermore, `grpX.cndB` and `grpY.cndB` can be contrasted using the `contrast` argument, in order to test if the condition effect is different across group:

```
results(dds, contrast=list("grpY.cndB","grpX.cndB"))
```

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:

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

```
##    group condition
## 1      1         A
## 2      1         A
## 3      1         B
## 4      1         B
## 5      1         C
## 6      1         C
## 7      2         A
## 8      2         A
## 9      2         B
## 10     2         B
## 11     2         C
## 12     2         C
## 13     3         A
## 14     3         A
## 15     3         B
## 16     3         B
```

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.)

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

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

```
unname(m1)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    1    0    0    0    0    0    0    0    0
##  [2,]    1    0    0    0    0    0    0    0    0
##  [3,]    1    1    0    0    0    0    0    0    0
##  [4,]    1    1    0    0    0    0    0    0    0
##  [5,]    1    0    1    0    0    0    0    0    0
##  [6,]    1    0    1    0    0    0    0    0    0
##  [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"
```

```
all.zero <- apply(m1, 2, function(x) all(x==0))
all.zero
```

```
##       (Intercept)        conditionB        conditionC            group2
##             FALSE             FALSE             FALSE             FALSE
##            group3 conditionB:group2 conditionC:group2 conditionB:group3
##             FALSE             FALSE             FALSE             FALSE
## conditionC:group3
##              TRUE
```

We can remove this column like so:

```
idx <- which(all.zero)
m1 <- m1[,-idx]
unname(m1)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
##  [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
##  [7,]    1    0    0    1    0    0    0    0
##  [8,]    1    0    0    1    0    0    0    0
##  [9,]    1    1    0    1    0    1    0    0
## [10,]    1    1    0    1    0    1    0    0
## [11,]    1    0    1    1    0    0    1    0
## [12,]    1    0    1    1    0    0    1    0
## [13,]    1    0    0    0    1    0    0    0
## [14,]    1    0    0    0    1    0    0    0
## [15,]    1    1    0    0    1    0    0    1
## [16,]    1    1    0    0    1    0    0    1
```

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` .

Theory behind DESeq2

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:

$$K_{ij} \sim \mathrm{NB}(\mu_{ij}, \alpha_i)$$

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

$$\log_2(q_{ij}) = x_{j.}\beta_i$$

where counts K_{ij} for gene *i*, sample *j* are modeled using a negative binomial distribution with fitted mean μ_{ij} and a gene-specific dispersion parameter α_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 β_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 α_i defines the relationship between the variance of the observed count and its mean value. In other words, how far do we expected 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.

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

An option in DESeq2 is to provide maximum *a posteriori* estimates of the log2 fold changes in β_i after incorporating a zero-centered Normal prior (`betaPrior`). While previously, these moderated, or shrunken, 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 (http://bioconductor.org/packages/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 α_i by *estimateDispersions*
3. negative binomial GLM fitting for β_i and Wald statistics by *nbinomWaldTest*

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

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 (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.

Methods changes since the 2014 DESeq2 paper

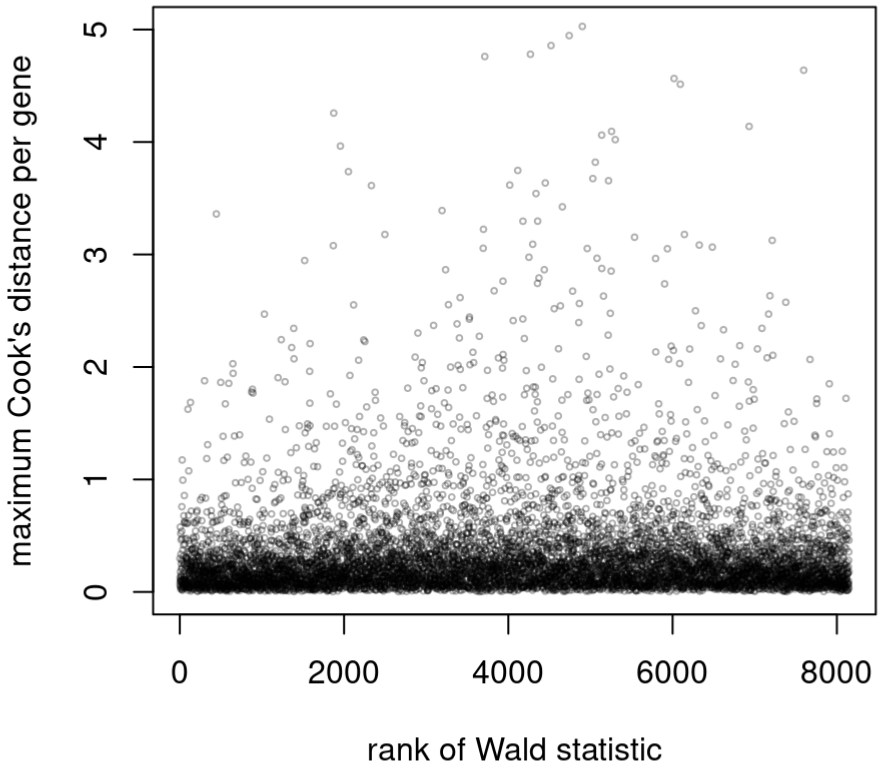
- In version 1.18 (November 2017), we add two alternative shrinkage estimators, which can be used via `lfcShrink` : an estimator using a t prior from the apeglm packages, and an estimator with a fitted mixture of normals prior from the ashr package.
- In version 1.16 (November 2016), the log2 fold change 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.

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.

```
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(0,0,0,.3))
m <- ncol(dds)
p <- 3
abline(h=qf(.99, p, m - p))
```



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 β 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 Σ 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.

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

Expanded model matrices

For the specific combination of `lfcShrink` with the type `normal` and using `contrast` , DESeq2 uses *expanded model matrices* to produce shrunken log2 fold change estimates where the shrinkage is independent of the choice of reference level. In all other cases, DESeq2 uses standard model matrices, as produced by `model.matrix` . 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. This is described in the DESeq2 paper. Using type `normal` with `coef` uses standard model matrices, as does the `apeglm` shrinkage estimator.

Independent filtering and multiple testing

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.

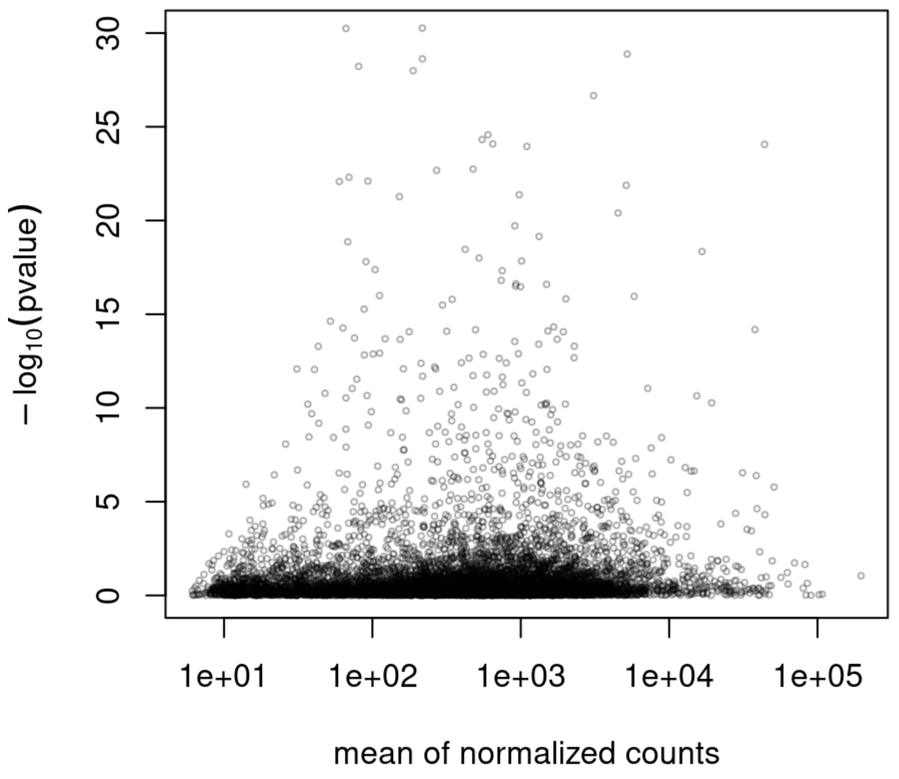
A good choice for a filtering criterion is one that

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:

```
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))
```



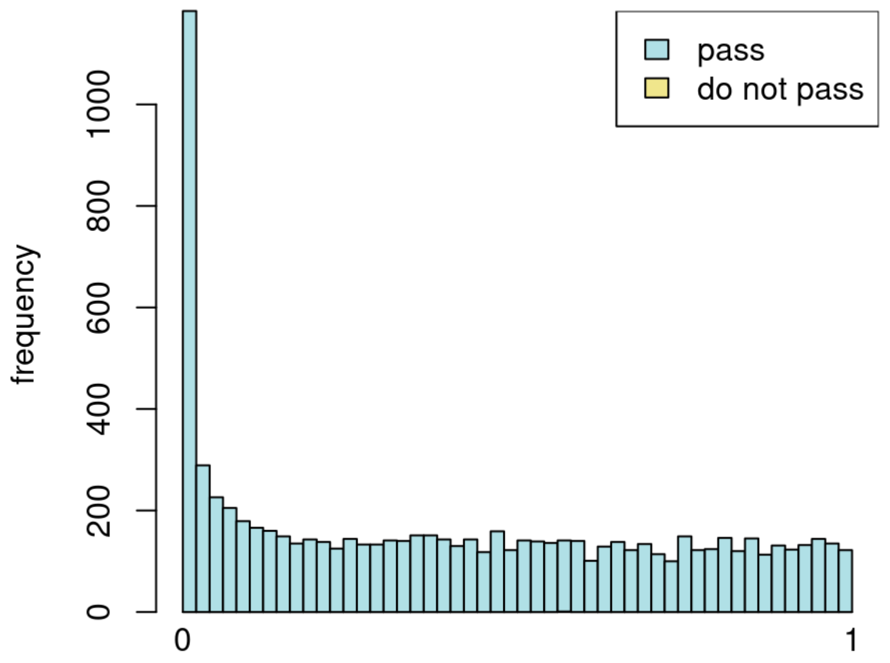
Why does it work?

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:

```r
barplot(height = rbind(h1$counts, h2$counts), beside = FALSE,
        col = colori, space = 0, main = "", ylab="frequency")
text(x = c(0, length(h1$counts)), y = 0, label = paste(c(0,1)),
     adj = c(0.5,1.7), xpd=NA)
legend("topright", fill=rev(colori), legend=rev(names(colori)))
```



Frequently asked questions

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 (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()*.

Why are some *p* values set to NA?

See the details above.

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:

```
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.

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.

Why after VST are there still batches in the PCA plot?

The transformations implemented in *DESeq2*, `vst` and `rlog`, compute a variance stabilizing transformation which is roughly similar to putting the data on the log2 scale, while also dealing with the sampling variability of low counts. It uses the design formula to calculate the within-group variability (if `blind=FALSE`) or the across-all-samples variability (if `blind=TRUE`). It does *not* use the design to remove variation in the data. It therefore does *not* remove variation that can be associated with batch or other covariates (nor does *DESeq2* have a way to specify which covariates are nuisance and which are of interest).

It is possible to visualize the transformed data with batch variation removed, using the `removeBatchEffect` function from *limma*. This simply removes any shifts in the log2-scale expression data that can be explained by batch. The paradigm for this operation for designs with balanced batches would be:

```
mat <- assay(vsd)
mm <- model.matrix(~condition, colData(vsd))
mat <- limma::removeBatchEffect(mat, batch=vsd$batch, design=mm)
assay(vsd) <- mat
plotPCA(vsd)
```

The `design` argument is necessary to avoiding removing variation associated with the treatment conditions. See `?removeBatchEffect` in the *limma* package for details.

Do normalized counts correct for variables in the design?

No. The design variables are not used when estimating the size factors, and `counts(dds, normalized=TRUE)` is providing counts scaled by size or normalization factors. The design is only used when estimating dispersion and log2 fold changes.

The only case in which there is more than size factor scaling on the counts is when either normalization factors have been provided (e.g. from `cqn` or `EDASeq`), or if `tximport` is used and the upstream software corrected for various technical biases (e.g. *Salmon* quantification with GC bias correction). In this case, the average transcript length is taken into account when scaling the counts with `counts(dds, normalized=TRUE)`. For details, see the *tximport* package vignette and citation (Soneson, Love, and Robinson 2015).

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`.

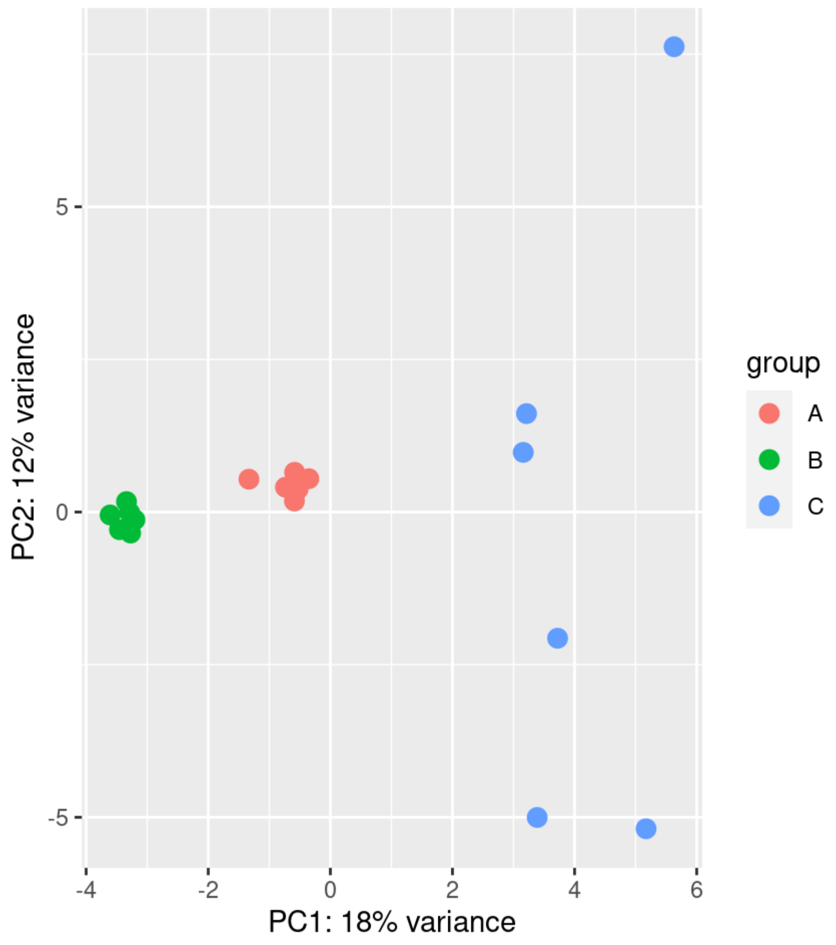
If I have multiple groups, should I run all together or split into pairs of groups?

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:



Can I run DESeq2 to contrast the levels of many groups?

DESeq2 will work with any kind of design specified using the R formula. We enourage users to consider exploratory data analysis such as principal components analysis rather than performing statistical testing of all pairs of many groups of samples. Statistical testing is one of many ways of describing differences between samples.

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, as this will speed up the fitting procedure.

Can I use DESeq2 to analyze a dataset without replicates?

No. This analysis is not possible in *DESeq2*.

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 some cases, more meaningful results may 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*.

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.

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.

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

I want to benchmark DESeq2 comparing to other DE tools.

One aspect which can cause problems for comparison is that, by default, DESeq2 outputs `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 `NA` based on extreme count outlier detection. These `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 `NA` values to 1:

```
res$padj <- ifelse(is.na(res$padj), 1, res$padj)
```

I have trouble installing DESeq2 on Ubuntu/Linux…

"*I try to install DESeq2, but I get an error trying to install the R packages XML and/or RCurl:*"

`ERROR: configuration failed for package XML`

`ERROR: configuration failed for package RCurl`

You need to install the following devel versions of packages using your standard package manager, e.g. `sudo apt-get install` or `sudo apt install`

- libxml2-dev
- libcurl4-openssl-dev

Session info

```
sessionInfo()
```

```
## R version 4.3.1 (2023-06-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.3 LTS
##
## Matrix products: default
## BLAS:   /home/biocbuild/bbs-3.18-bioc/R/lib/libRblas.so
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB              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
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats4    stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] pheatmap_1.0.12             vsn_3.69.0
##  [3] ggplot2_3.4.3               airway_1.21.0
##  [5] pasilla_1.29.0              DEXSeq_1.47.0
##  [7] RColorBrewer_1.1-3          AnnotationDbi_1.63.2
##  [9] BiocParallel_1.35.4         tximeta_1.19.7
## [11] DESeq2_1.41.12              SummarizedExperiment_1.31.1
## [13] Biobase_2.61.0              MatrixGenerics_1.13.1
## [15] matrixStats_1.0.0           GenomicRanges_1.53.1
## [17] GenomeInfoDb_1.37.4         IRanges_2.35.2
## [19] S4Vectors_0.39.2            BiocGenerics_0.47.0
## [21] tximportData_1.29.0         readr_2.1.4
## [23] tximport_1.29.1
##
## loaded via a namespace (and not attached):
##   [1] splines_4.3.1                 later_1.3.1
##   [3] BiocIO_1.11.0                 bitops_1.0-7
##   [5] filelock_1.0.2                tibble_3.2.1
##   [7] preprocessCore_1.63.1         XML_3.99-0.14
##  [9] lifecycle_1.0.3              mixsqp_0.3-48
## [11] lattice_0.21-8               vroom_1.6.3
## [13] ensembldb_2.25.1             MASS_7.3-60
## [15] magrittr_2.0.3               limma_3.57.8
## [17] sass_0.4.7                   rmarkdown_2.25
## [19] jquerylib_0.1.4              yaml_2.3.7
## [21] httpuv_1.6.11                DBI_1.1.3
## [23] abind_1.4-5                  zlibbioc_1.47.0
## [25] AnnotationFilter_1.25.0      RCurl_1.98-1.12
## [27] rappdirs_0.3.3               GenomeInfoDbData_1.2.10
## [29] irlba_2.3.5.1                genefilter_1.83.1
## [31] annotate_1.79.0              codetools_0.2-19
## [33] DelayedArray_0.27.10         xml2_1.3.5
## [35] tidyselect_1.2.0             farver_2.1.1
## [37] BiocFileCache_2.9.1          GenomicAlignments_1.37.0
## [39] jsonlite_1.8.7               ellipsis_0.3.2
## [41] survival_3.5-7               bbmle_1.0.25
## [43] tools_4.3.1                  progress_1.2.2
## [45] Rcpp_1.0.11                  glue_1.6.2
## [47] SparseArray_1.1.12           xfun_0.40
## [49] dplyr_1.1.3                  withr_2.5.1
## [51] numDeriv_2016.8-1.1          BiocManager_1.30.22
## [53] fastmap_1.1.1                fansi_1.0.4
## [55] digest_0.6.33                truncnorm_1.0-9
## [57] R6_2.5.1                     mime_0.12
## [59] colorspace_2.1-0             biomaRt_2.57.1
## [61] RSQLite_2.3.1                utf8_1.2.3
## [63] generics_0.1.3               hexbin_1.28.3
## [65] rtracklayer_1.61.1           prettyunits_1.2.0
## [67] httr_1.4.7                   S4Arrays_1.1.6
## [69] pkgconfig_2.0.3              gtable_0.3.4
## [71] blob_1.2.4                   hwriter_1.3.2.1
## [73] XVector_0.41.1               htmltools_0.5.6
## [75] geneplotter_1.79.1           ProtGenerics_1.33.1
## [77] scales_1.2.1                 png_0.1-8
## [79] ashr_2.2-63                  knitr_1.44
## [81] tzdb_0.4.0                   rjson_0.2.21
## [83] coda_0.19-4                  curl_5.0.2
## [85] bdsmatrix_1.3-6              cachem_1.0.8
## [87] stringr_1.5.0                BiocVersion_3.18.0
## [89] parallel_4.3.1               restfulr_0.0.15
## [91] apeglm_1.23.1                pillar_1.9.0
##  [93] grid_4.3.1                  vctrs_0.6.3
##  [95] promises_1.2.1              dbplyr_2.3.4
##  [97] xtable_1.8-4                archive_1.1.6
##  [99] evaluate_0.21               invgamma_1.1
## [101] GenomicFeatures_1.53.2      mvtnorm_1.2-3
## [103] cli_3.6.1                   locfit_1.5-9.8
## [105] compiler_4.3.1              Rsamtools_2.17.0
## [107] rlang_1.1.1                 crayon_1.5.2
## [109] SQUAREM_2021.1              labeling_0.4.3
## [111] emdbook_1.3.13              affy_1.79.3
## [113] plyr_1.8.8                  stringi_1.7.12
## [115] munsell_0.5.0               Biostrings_2.69.2
## [117] lazyeval_0.2.2              Matrix_1.6-1.1
## [119] hms_1.1.3                   bit64_4.0.5
## [121] KEGGREST_1.41.4             statmod_1.5.0
## [123] shiny_1.7.5                 interactiveDisplayBase_1.39.0
## [125] AnnotationHub_3.9.2         memoise_2.0.1
## [127] affyio_1.71.0               bslib_0.5.1
## [129] bit_4.0.5
```

References

Ahlmann-Eltze, Constantin, and Wolfgang Huber. 2020. “glmGamPoi: Fitting Gamma-Poisson Generalized Linear Models on Single Cell Count Data.” *Bioinformatics*, December. https://doi.org/10.1093/bioinformatics/btaa1009 (https://doi.org/10.1093/bioinformatics/btaa1009).

Anders, Simon, and Wolfgang Huber. 2010. “Differential Expression Analysis for Sequence Count Data.” *Genome Biology* 11: R106. http://genomebiology.com/2010/11/10/R106 (http://genomebiology.com/2010/11/10/R106).

Anders, Simon, Paul Theodor Pyl, and Wolfgang Huber. 2014. “HTSeq – A Python framework to work with high-throughput sequencing data.” *Bioinformatics*. http://dx.doi.org/10.1093/bioinformatics/btu638 (http://dx.doi.org/10.1093/bioinformatics/btu638).

Bourgon, Richard, Robert Gentleman, and Wolfgang Huber. 2010. “Independent Filtering Increases Detection Power for High-Throughput Experiments.” *PNAS* 107 (21): 9546–51. http://www.pnas.org/content/107/21/9546.long (http://www.pnas.org/content/107/21/9546.long).

Bray, Nicolas, Harold Pimentel, Pall Melsted, and Lior Pachter. 2016. “Near-Optimal Probabilistic Rna-Seq Quantification.” *Nature Biotechnology* 34: 525–27. http://dx.doi.org/10.1038/nbt.3519 (http://dx.doi.org/10.1038/nbt.3519).

Brooks, A. N., L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. 2011. “Conservation of an RNA regulatory map between Drosophila and mammals.” *Genome Research*, 193–202. https://doi.org/10.1101/gr.108662.110 (https://doi.org/10.1101/gr.108662.110).

Cook, R. Dennis. 1977. “Detection of Influential Observation in Linear Regression.” *Technometrics*, February.

Cox, D. R., and N. Reid. 1987. “Parameter orthogonality and approximate conditional inference.” *Journal of the Royal Statistical Society, Series B* 49 (1): 1–39. http://www.jstor.org/stable/2345476 (http://www.jstor.org/stable/2345476).

Gerard, David, and Matthew Stephens. 2017. “Empirical Bayes Shrinkage and False Discovery Rate Estimation, Allowing For Unwanted Variation.” *arXiv*. https://arxiv.org/abs/1709.10066 (https://arxiv.org/abs/1709.10066).

Huber, Wolfgang, Anja von Heydebreck, Holger Sültmann, Annemarie Poustka, and Martin Vingron. 2003. “Parameter Estimation for the Calibration and Variance Stabilization of Microarray Data.” *Statistical Applications in Genetics and Molecular Biology* 2 (1): Article 3.

Ignatiadis, Nikolaos, Bernd Klaus, Judith Zaugg, and Wolfgang Huber. 2016. “Data-Driven Hypothesis Weighting Increases Detection Power in Genome-Scale Multiple Testing.” *Nature Methods*. http://dx.doi.org/10.1038/nmeth.3885 (http://dx.doi.org/10.1038/nmeth.3885).

Leek, Jeffrey T. 2014. “svaseq: removing batch effects and other unwanted noise from sequencing data.” *Nucleic Acids Research* 42 (21). http://dx.doi.org/10.1093/nar/gku864 (http://dx.doi.org/10.1093/nar/gku864).

Li, Bo, and Colin N. Dewey. 2011. “RSEM: accurate transcript quantification from RNA-Seq data with or without a reference genome.” *BMC Bioinformatics* 12: 323+. https://doi.org/10.1186/1471-2105-12-3231 (https://doi.org/10.1186/1471-2105-12-3231).

Liao, Y., G. K. Smyth, and W. Shi. 2013. “featureCounts: an efficient general purpose program for assigning sequence reads to genomic features.” *Bioinformatics*, November.

Love, Michael I., John B. Hogenesch, and Rafael A. Irizarry. 2016. “Modeling of Rna-Seq Fragment Sequence Bias Reduces Systematic Errors in Transcript Abundance Estimation.” *Nature Biotechnology* 34 (12): 1287–91. http://dx.doi.org/10.1038/nbt.3682 (http://dx.doi.org/10.1038/nbt.3682).

Love, Michael I., Wolfgang Huber, and Simon Anders. 2014. “Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2.” *Genome Biology* 15 (12): 550. http://dx.doi.org/10.1186/s13059-014-0550-8 (http://dx.doi.org/10.1186/s13059-014-0550-8).

Love, Michael I., Charlotte Soneson, Peter F. Hickey, Lisa K. Johnson, N. Tessa Pierce, Lori Shepherd, Martin Morgan, and Rob Patro. 2020. “Tximeta: Reference sequence checksums for provenance identification in RNA-seq.” *PLOS Computational Biology*. https://doi.org/10.1371/journal.pcbi.1007664 (https://doi.org/10.1371/journal.pcbi.1007664).

Patro, Rob, Geet Duggal, Michael I. Love, Rafael A. Irizarry, and Carl Kingsford. 2017. “Salmon Provides Fast and Bias-Aware Quantification of Transcript Expression.” *Nature Methods*. http://dx.doi.org/10.1038/nmeth.4197 (http://dx.doi.org/10.1038/nmeth.4197).

Patro, Rob, Stephen M. Mount, and Carl Kingsford. 2014. “Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms.” *Nature Biotechnology* 32: 462–64. http://dx.doi.org/10.1038/nbt.2862 (http://dx.doi.org/10.1038/nbt.2862).

Risso, Davide, John Ngai, Terence P Speed, and Sandrine Dudoit. 2014. “Normalization of RNA-seq data using factor analysis of control genes or samples.” *Nature Biotechnology* 32 (9). http://dx.doi.org/10.1038/nbt.2931 (http://dx.doi.org/10.1038/nbt.2931).

Robert, Christelle, and Mick Watson. 2015. “Errors in RNA-Seq quantification affect genes of relevance to human disease.” *Genome Biology*. https://doi.org/10.1186/s13059-015-0734-x (https://doi.org/10.1186/s13059-015-0734-x).

Soneson, Charlotte, Michael I. Love, and Mark Robinson. 2015. “Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences.” *F1000Research* 4 (1521). http://dx.doi.org/10.12688/f1000research.7563.1 (http://dx.doi.org/10.12688/f1000research.7563.1).

Stephens, Matthew. 2016. “False Discovery Rates: A New Deal.” *Biostatistics* 18 (2). https://doi.org/10.1093/biostatistics/kxw041 (https://doi.org/10.1093/biostatistics/kxw041).

Storey, J. 2003. “The positive false discovery rate: A Bayesian interpretation and the q-value.” *The Annals of Statistics* 31 (6): 2013–35.

Tibshirani, Robert. 1988. “Estimating Transformations for Regression via Additivity and Variance Stabilization.” *Journal of the American Statistical Association* 83: 394–405.

Trapnell, Cole, David G Hendrickson, Martin Sauvageau, Loyal Goff, John L Rinn, and Lior Pachter. 2013. “Differential analysis of gene regulation at transcript resolution with RNA-seq.” *Nature Biotechnology*. https://doi.org/10.1038/nbt.2450 (https://doi.org/10.1038/nbt.2450).

Van den Berge, Koen, Fanny Perraudeau, Charlotte Soneson, Michael I Love, Davide Risso, Jean-Philippe Vert, Mark D Robinson, Sandrine Dudoit, and Lieven Clement. 2018. “Observation weights unlock bulk RNA-seq tools for zero inflation and single-cell applications.” *Genome Biology* 19 (24). https://doi.org/10.1186/s13059-018-1406-4 (https://doi.org/10.1186/s13059-018-1406-4).

Wu, Hao, Chi Wang, and Zhijin Wu. 2012. “A new shrinkage estimator for dispersion improves differential expression detection in RNA-seq data.” *Biostatistics*, September. https://doi.org/10.1093/biostatistics/kxs033 (https://doi.org/10.1093/biostatistics/kxs033).

Zhu, Anqi, Joseph G. Ibrahim, and Michael I. Love. 2018. “Heavy-Tailed Prior Distributions for Sequence Count Data: Removing the Noise and Preserving Large Differences.” *Bioinformatics*. https://doi.org/10.1093/bioinformatics/bty895 (https://doi.org/10.1093/bioinformatics/bty895).