Differential analysis of count data – the DESeq2 package

Michael I. Love[1], Simon Anders[2,3], Wolfgang Huber[3]

[1] Department of Biostatistics, Dana-Farber Cancer Institute and
Harvard TH Chan School of Public Health, Boston, US;
[2] Institute for Molecular Medicine Finland (FIMM), Helsinki, Finland;
[3] European Molecular Biology Laboratory (EMBL), Heidelberg, Germany

December 21, 2015

Abstract

A basic task in the analysis of count data from RNA-seq is the detection of differentially expressed genes. The count data are presented as a table which reports, for each sample, the number of sequence fragments that have been assigned to each gene. Analogous data also arise for other assay types, including comparative ChIP-Seq, HiC, shRNA screening, mass spectrometry. An important analysis question is the quantification and statistical inference of systematic changes between conditions, as compared to within-condition variability. The package *DESeq2* provides methods to test for differential expression by use of negative binomial generalized linear models; the estimates of dispersion and logarithmic fold changes incorporate data-driven prior distributions[1]. This vignette explains the use of the package and demonstrates typical workflows. An RNA-seq workflow[2] 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 version: 1.10.1

If you use *DESeq2* in published research, please cite:

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

[1] Other *Bioconductor* packages with similar aims are *edgeR*, *limma*, *DSS*, *EBSeq* and *baySeq*.
[2] http://www.bioconductor.org/help/workflows/rnaseqGene/

Contents

1 Standard workflow

1.1 Quick start

Here we show the most basic steps for a differential expression analysis. These steps require you have a *RangedSummarizedExperiment* object `se` which contains the counts and information about samples. The `design` indicates that we want to measure the effect of condition, controlling for batch differences. The two factor variables `batch` and `condition` should be columns of `colData(se)`.

```
dds <- DESeqDataSet(se, design = ~ batch + condition)
dds <- DESeq(dds)
res <- results(dds, contrast=c("condition","trt","con"))
```

If you have a count matrix and sample information table, the first line would use `DESeqDataSetFromMatrix` instead of `DESeqDataSet`, as shown in Section 1.3.3.

1.2 How to get help

All *DESeq2* questions should be posted to the Bioconductor support site: https://support.bioconductor.org, which serves as a repository of questions and answers. See the first question in the list of Frequently Asked Questions (Section 5) for more information about how to construct an informative post.

1.3 Input data

1.3.1 Why raw 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 have been mapped to gene i in sample j. Analogously, for other types of assays, the rows of the matrix might correspond e. g. to binding regions (with ChIP-Seq) or peptide sequences (with quantitative mass spectrometry). We will list method for obtaining count tables in a section below.

The count values must be raw counts of sequencing reads (for single-end RNA-seq) or fragments (for paired-end RNA-seq). The RNA-seq workflow describes multiple techniques for preparing such count matrices. It is important to provide count matrices as input for *DESeq2*'s statistical model [1] to hold, as only the count values allow assessing the measurement precision correctly. The *DESeq2* model internally corrects for library size, so transformed values such as counts scaled by library size should never be used as input.

1.3.2 SummarizedExperiment input

The class used by the *DESeq2* package to store the read counts is *DESeqDataSet* which extends the *RangedSummarizedExperiment* class of the *SummarizedExperiment* package. This facilitates preparation steps and also downstream exploration of results. For counting aligned reads in genes, the `summarizeOverlaps` function of *GenomicAlignments* with `mode="Union"` is encouraged, resulting in a *RangedSummarizedExperiment* object. Other methods for obtaining count matrices are described in the next section.

An example of the steps to produce a *RangedSummarizedExperiment* can be found in an RNA-seq workflow on the Bioconductor website: http://www.bioconductor.org/help/workflows/rnaseqGene/ and in the vignette for the data package *airway*. Here we load the *RangedSummarizedExperiment* from that package in order to build a *DESeqDataSet*.

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

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). An intercept is included, representing the base mean of counts. 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. The constructor function below shows the generation of a *DESeqDataSet* from a *RangedSummarizedExperiment* `se`.

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.

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

```
## class: DESeqDataSet
## dim: 64102 8
## metadata(1): ''
## assays(1): counts
## rownames(64102): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
## rowRanges metadata column names(0):
## colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
## colData names(9): SampleName cell ... Sample BioSample
```

1.3.3 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 in the *Rsubread* package. To use `DESeqDataSetFromMatrix`, the user should provide the counts matrix, the information about the samples (the columns of the count matrix) as a *DataFrame* or *data.frame*, and the design formula.

To demonstate the use of `DESeqDataSetFromMatrix`, we will first load the `pasillaGenes` data object, pull out the count matrix which we will name `countData`, and sample information, `colData`. Below we describe how to extract these objects from, e.g. `featureCounts` output.

```
library("pasilla")
library("Biobase")
data("pasillaGenes")
countData <- counts(pasillaGenes)
colData <- pData(pasillaGenes)[,c("condition","type")]
```

The count matrix and column data:

```
head(countData)
```

```
##             treated1fb treated2fb treated3fb untreated1fb untreated2fb
## FBgn0000003          0          0          1            0            0
## FBgn0000008         78         46         43           47           89
## FBgn0000014          2          0          0            0            0
## FBgn0000015          1          0          1            0            1
## FBgn0000017       3187       1672       1859         2445         4615
## FBgn0000018        369        150        176          288          383
##             untreated3fb untreated4fb
## FBgn0000003            0            0
## FBgn0000008           53           27
## FBgn0000014            1            0
## FBgn0000015            1            2
## FBgn0000017         2063         1711
## FBgn0000018          135          174
```

```
head(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
```

If you have used the `featureCounts` function in the *Rsubread* package, the matrix of read counts can be directly provided from the `"counts"` element in the list output. The count matrix and column data can also 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, `countData`, and the sample information, `colData`, we can construct a *DESeqDataSet*:

```
dds <- DESeqDataSetFromMatrix(countData = countData,
                              colData = colData,
                              design = ~ condition)
dds
```

```
## class: DESeqDataSet
## dim: 14470 7
## metadata(0):
## assays(1): counts
## rownames(14470): FBgn0000003 FBgn0000008 ... FBgn0261574 FBgn0261575
## rowRanges metadata column names(0):
## colnames(7): treated1fb treated2fb ... untreated3fb untreated4fb
## 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(pasillaGenes))
(mcols(dds) <- DataFrame(mcols(dds), featureData))
```

```
## DataFrame with 14470 rows and 1 column
##              gene
##          <factor>
## 1     FBgn0000003
## 2     FBgn0000008
## 3     FBgn0000014
## 4     FBgn0000015
## 5     FBgn0000017
## ...           ...
## 14466 FBgn0261571
## 14467 FBgn0261572
## 14468 FBgn0261573
## 14469 FBgn0261574
## 14470 FBgn0261575
```

1.3.4 HTSeq input

You can use the function DESeqDataSetFromHTSeqCount if you have htseq-count from the *HTSeq* python package[3]. For an example of using the python scripts, see the *pasilla* data package. First you will want to specify a variable which points to the directory in which the *HTSeq* 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* output files packages for the *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)
ddsHTSeq <- DESeqDataSetFromHTSeqCount(sampleTable = sampleTable,
                                       directory = directory,
                                       design= ~ condition)
ddsHTSeq
```

```
## class: DESeqDataSet
## dim: 70463 7
```

[3] available from http://www-huber.embl.de/users/anders/HTSeq, described in [2]

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

1.3.5 Pre-filtering

While it is not necessary to pre-filter low count genes before running the *DESeq2* functions, there are two reasons which make pre-filtering useful: by removing rows in which there are no reads or nearly no reads, we reduce the memory size of the `dds` data object and we increase the speed of the transformation and testing functions within *DESeq2*. Here we perform a minimal pre-filtering to remove rows that have only 0 or 1 read. Note that more strict filtering to increase power is *automatically* applied via independent filtering on the mean of normalized counts within the `results` function, which will be discussed in Section 3.8.

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

1.3.6 Note on factor levels

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

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

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

```
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*:

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

1.3.7 Collapsing technical replicates

DESeq2 provides a function `collapseReplicates` which can assist in combining the counts from technical replicates into single columns. See the manual page for an example of the use of `collapseReplicates`.

1.3.8 About the pasilla dataset

We continue with the *pasilla* data constructed from the count matrix method above. This data set is from an experiment on *Drosophila melanogaster* cell cultures and investigated the effect of RNAi knock-down of the splicing factor *pasilla* [3]. The detailed transcript of the production of the *pasilla* data is provided in the vignette of the data package *pasilla*.

1.4 Differential expression analysis

The standard differential expression analysis steps are wrapped into a single function, `DESeq`. The estimation steps performed by this function are described in Section 4.1, in the manual page for `?DESeq` and in the Methods section of the *DESeq2* publication [1]. The individual sub-functions which are called by `DESeq` are still available, described in Section 3.1.

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 arguments to `results`, the results will be for the last variable in the design formula, and if this is a factor, the comparison will be the last level of this variable over the first level. Details about the comparison are printed to the console. The text, `condition treated vs untreated`, tells you that the estimates are of the logarithmic fold change $\log_2(\text{treated}/\text{untreated})$.

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

```
## log2 fold change (MAP): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 11041 rows and 6 columns
##              baseMean log2FoldChange     lfcSE      stat    pvalue      padj
##             <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008    52.226         0.0197    0.2092    0.0942     0.925     0.989
## FBgn0000014     0.390         0.0118    0.0622    0.1901     0.849        NA
## FBgn0000015     0.905        -0.0429    0.1053   -0.4076     0.684        NA
## FBgn0000017  2358.243        -0.2554    0.1198   -2.1317     0.033     0.244
## FBgn0000018   221.242        -0.1038    0.1558   -0.6666     0.505     0.873
## ...               ...            ...       ...       ...       ...       ...
## FBgn0261570   1793.06        0.26011     0.125    2.0879    0.0368     0.261
## FBgn0261572      3.24       -0.20257     0.171   -1.1846    0.2362        NA
## FBgn0261573   1401.11       -0.00229     0.133   -0.0172    0.9863     0.997
## FBgn0261574   2719.02        0.02538     0.167    0.1521    0.8791     0.984
## FBgn0261575      3.36        0.08990     0.163    0.5510    0.5816        NA
```

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

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

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

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

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

```
summary(res)
```

```
##
## out of 11041 with nonzero total read count
## adjusted p-value < 0.1
## LFC > 0 (up)     : 393, 3.6%
## LFC < 0 (down)   : 404, 3.7%
## outliers [1]     : 15, 0.14%
## low counts [2]   : 2565, 23%
## (mean count < 3)
## [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] 797
```

The `results` function contains a number of arguments to customize the results table which is generated. 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 in Section 3.8. 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 11041 with nonzero total read count
## adjusted p-value < 0.05
## LFC > 0 (up)     : 306, 2.8%
## LFC < 0 (down)   : 325, 2.9%
## outliers [1]     : 15, 0.14%
## low counts [2]   : 2565, 23%
## (mean count < 3)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```

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

```
## [1] 631
```

If a multi-factor design is used, or if the variable in the design formula has more than two levels, the `contrast` argument of `results` can be used to extract different comparisons from the *DESeqDataSet* returned by `DESeq`. Multi-factor designs are discussed further in Section 1.6, and the use of the `contrast` argument is dicussed in Section 3.2.

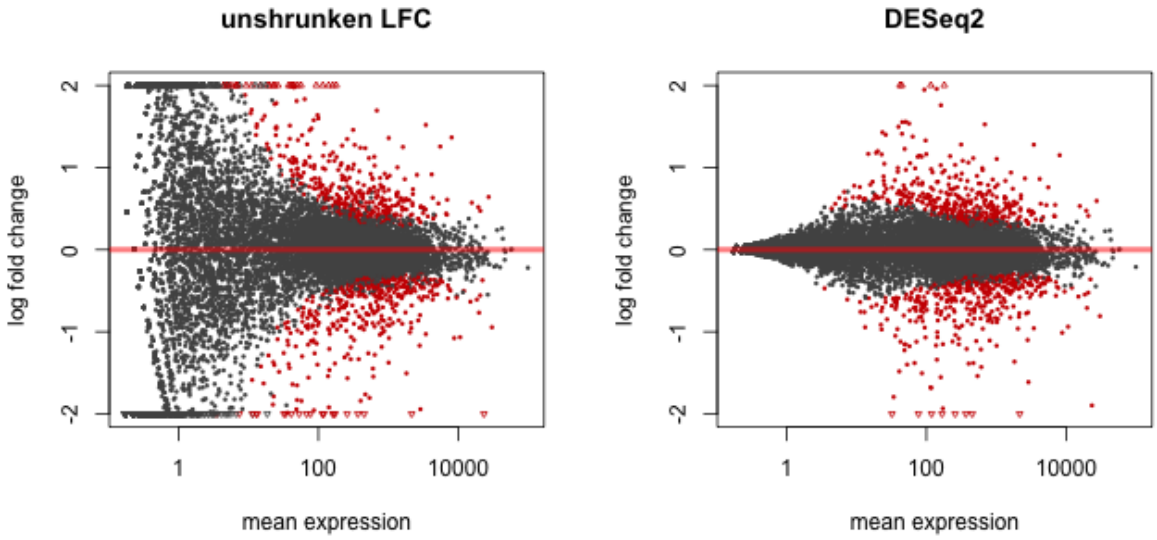

Figure 1: **MA-plot.** These plots show the log2 fold changes from the treatment over the mean of normalized counts, i.e. the average of counts normalized by size factors. The left plot shows the "unshrunken" log2 fold changes, while the right plot, produced by the code above, shows the shrinkage of log2 fold changes resulting from the incorporation of zero-centered normal prior. The shrinkage is greater for the log2 fold change estimates from genes with low counts and high dispersion, as can be seen by the narrowing of spread of leftmost points in the right plot.

For advanced users, note that all the values calculated by the *DESeq2* package are stored in the *DESeqDataSet* object, and access to these values is discussed in Section 3.10.

1.5 Exploring and exporting results

1.5.1 MA-plot

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

```
plotMA(res, main="DESeq2", 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]
```

The MA-plot of log2 fold changes returned by *DESeq2* allows us to see how the shrinkage of fold changes works for genes with low counts. You can still obtain results tables which include the "unshrunken" log2 fold changes (for a simple comparison, the ratio of the mean normalized counts in the two groups). A column `lfcMLE` with the unshrunken maximum likelihood estimate (MLE) for the log2 fold change will be added with an additional argument to `results`:

```
resMLE <- results(dds, addMLE=TRUE)
head(resMLE, 4)
```

```
## log2 fold change (MAP): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 4 rows and 7 columns
##              baseMean log2FoldChange    lfcMLE     lfcSE      stat    pvalue
##             <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008    52.226         0.0197    0.0281    0.2092    0.0942     0.925
## FBgn0000014     0.390         0.0118    0.6239    0.0622    0.1901     0.849
## FBgn0000015     0.905        -0.0429   -0.8135    0.1053   -0.4076     0.684
## FBgn0000017  2358.243        -0.2554   -0.2758    0.1198   -2.1317     0.033
##                  padj
##             <numeric>
## FBgn0000008     0.989
## FBgn0000014        NA
## FBgn0000015        NA
## FBgn0000017     0.244
```

One can make an MA-plot of the unshrunken estimates like so:

```
plotMA(resMLE, MLE=TRUE, main="unshrunken LFC", ylim=c(-2,2))
```

1.5.2 Plot counts

It can also be useful to examine the counts of reads for a single gene across the groups. A simple function for making this plot is `plotCounts`, which normalizes counts by sequencing depth and adds a pseudocount of $\frac{1}{2}$ 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))
```

1.5.3 More information on results columns

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

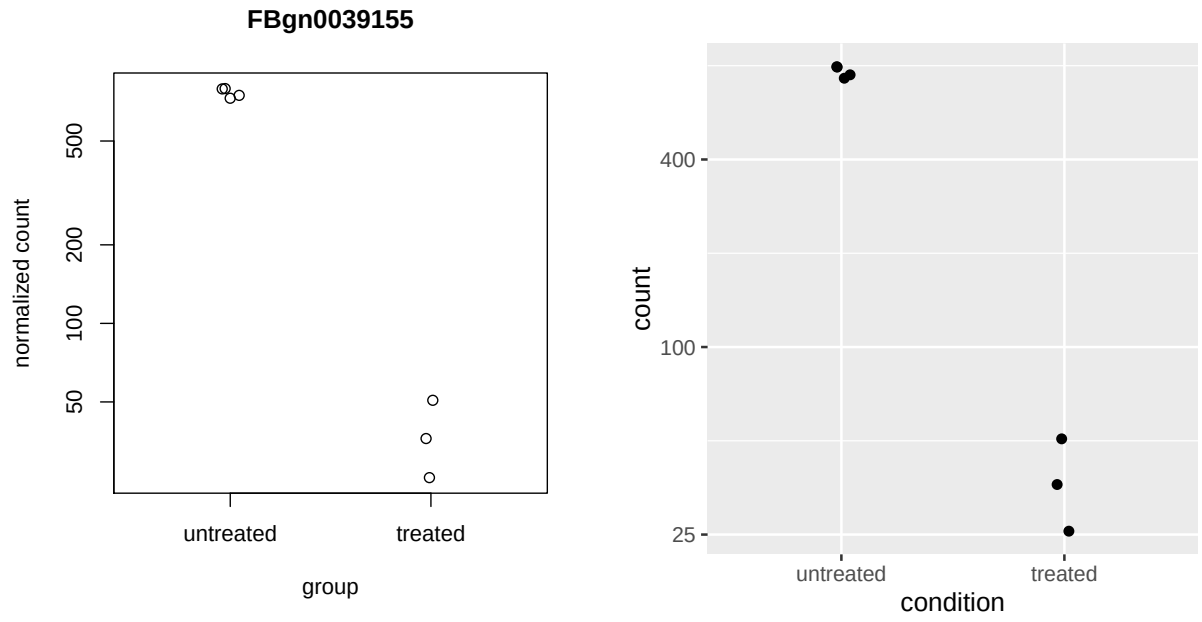

Figure 2: **Plot of counts for one gene.** The plot of normalized counts (plus a pseudocount of $\frac{1}{2}$) either made using the `plotCounts` function (left) or using another plotting library (right, using *ggplot2*).

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

```
## [1] "mean of normalized counts for all samples"
## [2] "log2 fold change (MAP): 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:

1. 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`.
2. 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 in Section 3.6,
3. 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 in Section 3.8.

1.5.4 Exporting results to HTML or CSV files

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

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 (MAP): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 797 rows and 6 columns
##              baseMean log2FoldChange     lfcSE      stat    pvalue      padj
##             <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0039155       453          -3.71     0.160     -23.2 4.01e-119 3.39e-115
## FBgn0029167      2165          -2.08     0.104     -20.1  6.68e-90  2.83e-86
## FBgn0035085       367          -2.23     0.137     -16.3  1.89e-59  5.33e-56
## FBgn0029896       258          -2.21     0.159     -13.9  5.85e-44  1.24e-40
## FBgn0034736       118          -2.56     0.185     -13.9  8.06e-44  1.36e-40
## ...               ...            ...       ...       ...       ...       ...
## FBgn0029504     589.6         -0.393     0.151      -2.6   0.00929    0.0991
## FBgn0002567      29.3          0.574     0.221       2.6   0.00932    0.0992
## FBgn0050359      28.4          0.594     0.229       2.6   0.00931    0.0992
## FBgn0026150     495.6          0.334     0.128       2.6   0.00939    0.0999
## FBgn0028428      62.0          0.510     0.196       2.6   0.00942    0.1000
```

1.6 Multi-factor designs

Experiments with more than one factor influencing the counts can be analyzed using design formula that include the additional variables. By adding these to the design, one can control for additional variation in the counts. For example, if the condition samples are balanced across experimental batches, by including the `batch` factor to the design, one can increase the sensitivity for finding differences due to `condition`. There are multiple ways to analyze experiments when the additional variables are of interest and not just controlling factors (see Section 3.3 on interactions).

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

```
colData(dds)
```

```
## DataFrame with 7 rows and 3 columns
```

```
##              condition        type sizeFactor
##               <factor>    <factor>  <numeric>
## treated1fb     treated single-read      1.512
## treated2fb     treated  paired-end      0.784
## treated3fb     treated  paired-end      0.896
## untreated1fb untreated single-read      1.050
## untreated2fb untreated single-read      1.659
## untreated3fb untreated  paired-end      0.712
## untreated4fb untreated  paired-end      0.784
```

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

```
ddsMF <- dds
```

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

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

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

```
resMF <- results(ddsMF)
head(resMF)
```

```
## log2 fold change (MAP): 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    52.226        0.01216    0.2072    0.0587    0.9532     0.989
## FBgn0000014     0.390        0.00962    0.0558    0.1724    0.8631        NA
## FBgn0000015     0.905       -0.03536    0.0924   -0.3828    0.7019        NA
## FBgn0000017  2358.243       -0.25659    0.1100   -2.3331    0.0196     0.141
## FBgn0000018   221.242       -0.06663    0.1416   -0.4706    0.6379     0.892
## FBgn0000024     3.115        0.09471    0.1499    0.6320    0.5274        NA
```

It is also possible to retrieve the log2 fold changes, p values and adjusted p values of the `type` variable. The `contrast` argument of the function `results` takes a character vector of length three: the name of the variable, the name of the factor level for the numerator of the log2 ratio, and the name of the factor level for the denominator. The `contrast` argument can also take other forms, as described in the help page for `results` and in Section 3.2.

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

```
## log2 fold change (MAP): type single-read vs paired-end
## Wald test p-value: type single-read vs paired-end
## DataFrame with 6 rows and 6 columns
```

```
##              baseMean log2FoldChange     lfcSE      stat    pvalue      padj
##             <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008    52.226        -0.0623    0.1973   -0.3159    0.7521     0.892
## FBgn0000014     0.390         0.0057    0.0490    0.1163    0.9074        NA
## FBgn0000015     0.905        -0.0558    0.0815   -0.6845    0.4937        NA
## FBgn0000017  2358.243         0.0094    0.1088    0.0865    0.9311     0.974
## FBgn0000018   221.242         0.2686    0.1383    1.9421    0.0521     0.219
## FBgn0000024     3.115         0.0463    0.1333    0.3473    0.7284        NA
```

If the variable is continuous or an interaction term (see Section 3.3) then the results can be extracted using the `name` argument to `results`, where the name is one of elements returned by `resultsNames(dds)`.

2 Data transformations and visualization

2.1 Count data transformations

In order to test for differential expression, we operate on raw counts and use discrete distributions as described in the previous Section 1.4. 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

$$y = \log_2(n+1) \quad \text{or more generally,} \quad y = \log_2(n+n_0), \tag{1}$$

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

In this section, we discuss two alternative approaches that offer more theoretical justification and a rational way of choosing the parameter equivalent to n_0 above. The *regularized logarithm* or *rlog* incorporates a prior on the sample differences [1], and the other uses the concept of variance stabilizing transformations (VST) [4, 5, 6]. Both transformations produce transformed data on the $\log_2$ scale which has been normalized with respect to library size.

The point of these two transformations, the *rlog* and the VST, is to remove the dependence of the variance on the mean, particularly the high variance of the logarithm of count data when the mean is low. Both *rlog* and VST use the experiment-wide trend of variance over mean, in order to transform the data to remove the experiment-wide trend. Note that we do not require or desire that all the genes have *exactly* the same variance after transformation. Indeed, in Figure 4 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 the variance stabilizing transformation might be a better 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 [1].

2.1.1 Blind dispersion estimation

The two functions, `rlog` and `varianceStabilizingTransformation`, 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 (design formula ~ 1). 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 tranform 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 unbiased by the information about which samples were in which experimental group.

2.1.2 Extracting transformed values

The two functions return an object of class *DESeqTransform* which is a subclass of *RangedSummarizedExperiment*. The `assay` function is used to extract the matrix of normalized values:

```
rld <- rlog(dds)
vsd <- varianceStabilizingTransformation(dds)
head(assay(rld), 3)
```

```
##             treated1fb treated2fb treated3fb untreated1fb untreated2fb
## FBgn0000008      5.690      5.746      5.660        5.630        5.709
## FBgn0000014     -1.349     -1.371     -1.372       -1.372       -1.373
## FBgn0000015     -0.202     -0.211     -0.194       -0.213       -0.203
##             untreated3fb untreated4fb
## FBgn0000008         5.86        5.541
## FBgn0000014        -1.35       -1.371
## FBgn0000015        -0.19       -0.173
```

2.1.3 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 the `DESeq` and `nbinomWaldTest`, as seen in Figure 1. 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 Section 4.1), 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. The regularized log transformation is preferable to the variance stabilizing transformation if the size factors vary widely.

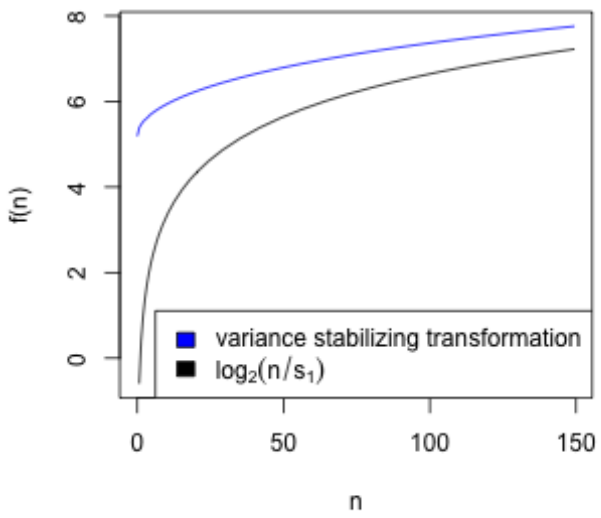

Figure 3: **VST and log2.** Graphs of the variance stabilizing transformation for sample 1, in blue, and of the transformation $f(n) = \log_2(n/s_1)$, in black. n are the counts and s_1 is the size factor for the first sample.

2.1.4 Variance stabilizing transformation

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

The resulting variance stabilizing transformation is shown in Figure 3. The code that produces the figure is hidden from this vignette for the sake of brevity, but can be seen in the `.Rnw` or `.R` source file. 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.

2.1.5 Effects of transformations on the variance

Figure 4 plots the standard deviation of the transformed data, across samples, against the mean, using the shifted logarithm transformation (1), 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.

```
library("vsn")
notAllZero <- (rowSums(counts(dds))>0)
meanSdPlot(log2(counts(dds,normalized=TRUE)[notAllZero,] + 1))
```

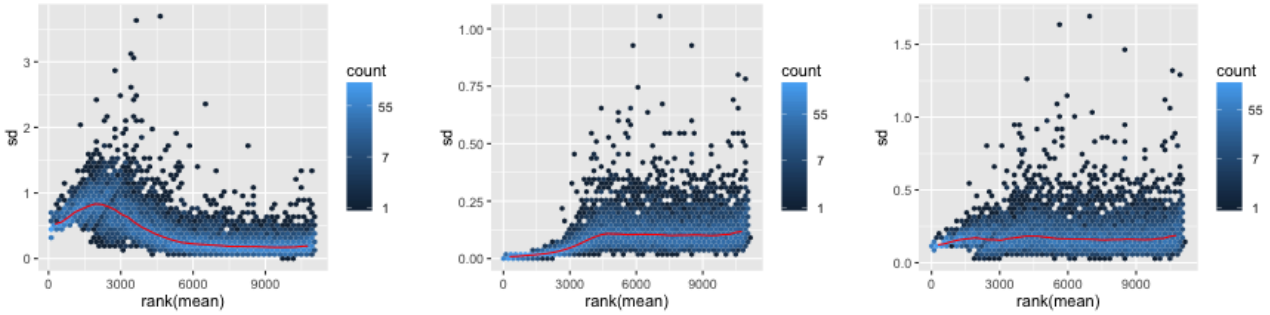

Figure 4: Per-gene standard deviation (taken across samples), against the rank of the mean, for the shifted logarithm $\log_2(n+1)$ (left), the regularized log transformation (center) and the variance stabilizing transformation (right).

```
meanSdPlot(assay(rld[notAllZero,]))
```

```
meanSdPlot(assay(vsd[notAllZero,]))
```

2.2 Data quality assessment by sample clustering and visualization

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

We define the term *quality* as *fitness for purpose*[4]. 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.

2.2.1 Heatmap of the count matrix

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

```
library("pheatmap")
select <- order(rowMeans(counts(dds,normalized=TRUE)),decreasing=TRUE)[1:20]
```

```
nt <- normTransform(dds) # defaults to log2(x+1)
log2.norm.counts <- assay(nt)[select,]
df <- as.data.frame(colData(dds)[,c("condition","type")])
pheatmap(log2.norm.counts, 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)
```

[4] http://en.wikipedia.org/wiki/Quality_%28business%29

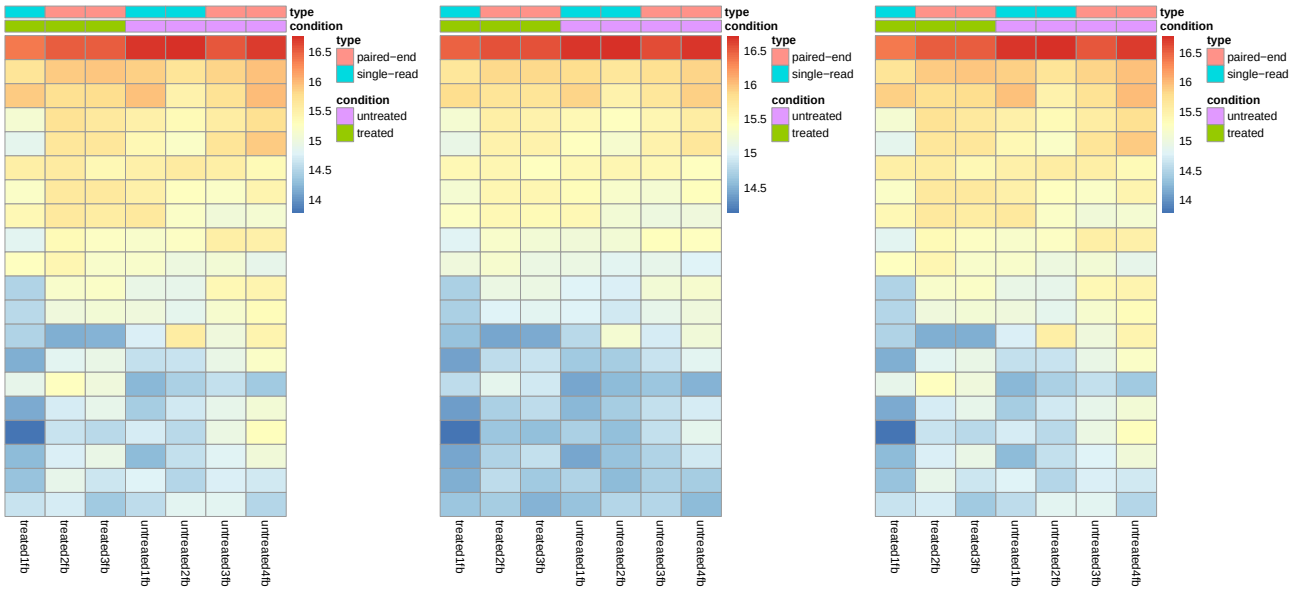

Figure 5: Heatmaps showing the expression data of the 20 most highly expressed genes. The data is of log2 normalized counts (left), from regularized log transformation (center) and from variance stabilizing transformation (right).

```
pheatmap(assay(vsd)[select,], cluster_rows=FALSE, show_rownames=FALSE,
         cluster_cols=FALSE, annotation_col=df)
```

2.2.2 Heatmap of the sample-to-sample distances

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

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

A heatmap of this distance matrix gives us an overview over similarities and dissimilarities between samples (Figure 6): 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.

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

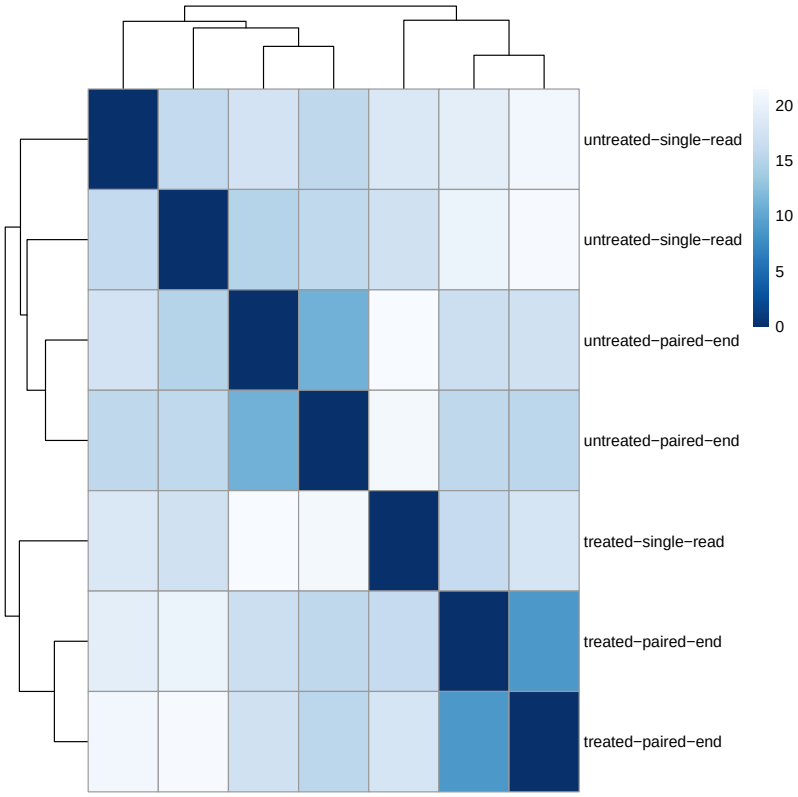

Figure 6: **Sample-to-sample distances.** Heatmap showing the Euclidean distances between the samples as calculated from the regularized log transformation.

2.2.3 Principal component plot of the samples

Related to the distance matrix of Section 2.2.2 is the PCA plot of the samples, which we obtain as follows (Figure 7).

```
plotPCA(rld, intgroup=c("condition", "type"))
```

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

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

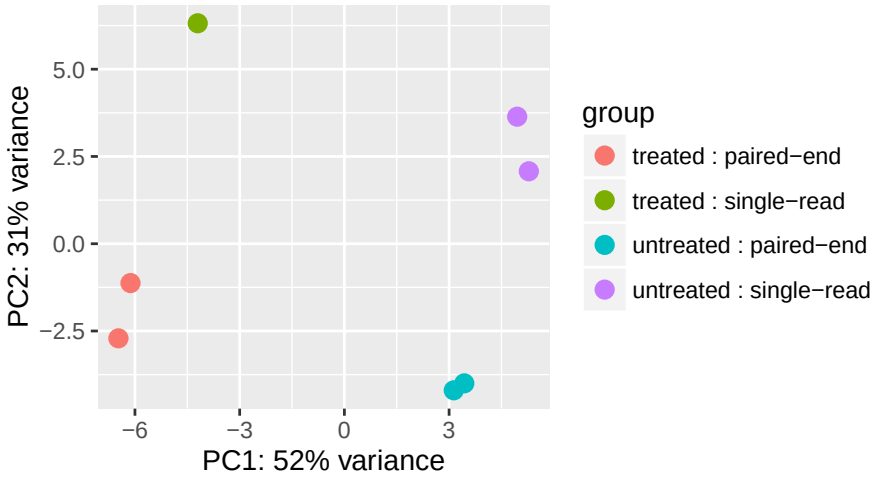

Figure 7: **PCA plot.** PCA plot. The 7 samples shown in the 2D plane spanned by their first two principal components. This type of plot is useful for visualizing the overall effect of experimental covariates and batch effects.

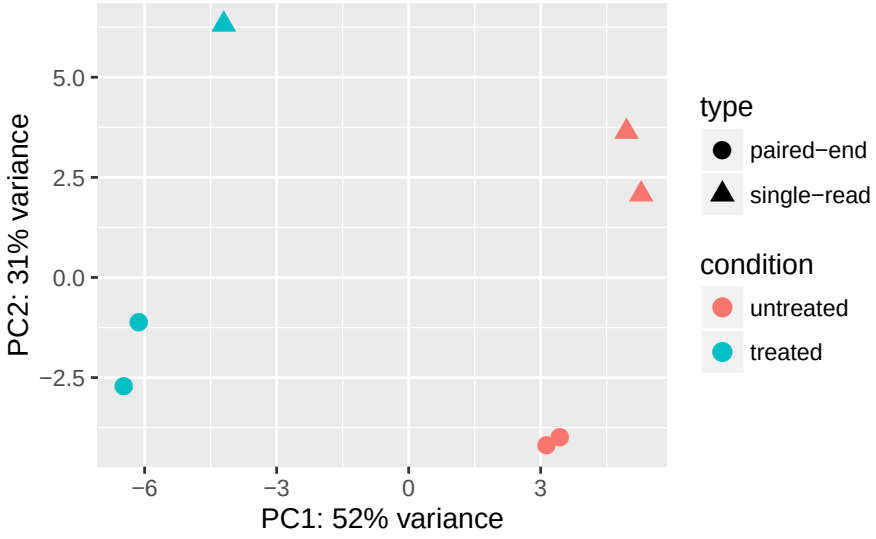

Figure 8: **PCA plot.** PCA plot customized using the *ggplot2* library.

3 Variations to the standard workflow

3.1 Wald test individual steps

The function DESeq runs the following functions in order:

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

3.2 Contrasts

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

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

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

Alternatively, a numeric vector of the length of `resultsNames(dds)` can be provided, for manually specifying the linear combination of terms. Demonstrations of the use of contrasts for various designs can be found in the examples section of the help page for the `results` function. The mathematical formula that is used to generate the contrasts can be found in Section 4.5.

3.3 Interactions

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

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

1. combine the factors of interest into a single factor with all combinations of the original factors
2. change the design to include just this factor, e.g. $\sim$ `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"))
```

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). For a diagram of how interactions might look across genotypes please refer to Figure 9.

The key point to remember about designs with interaction terms is that, unlike for a design $\sim$ `genotype + condition`, where the condition effect represents the *overall* effect controlling for differences due to genotype,

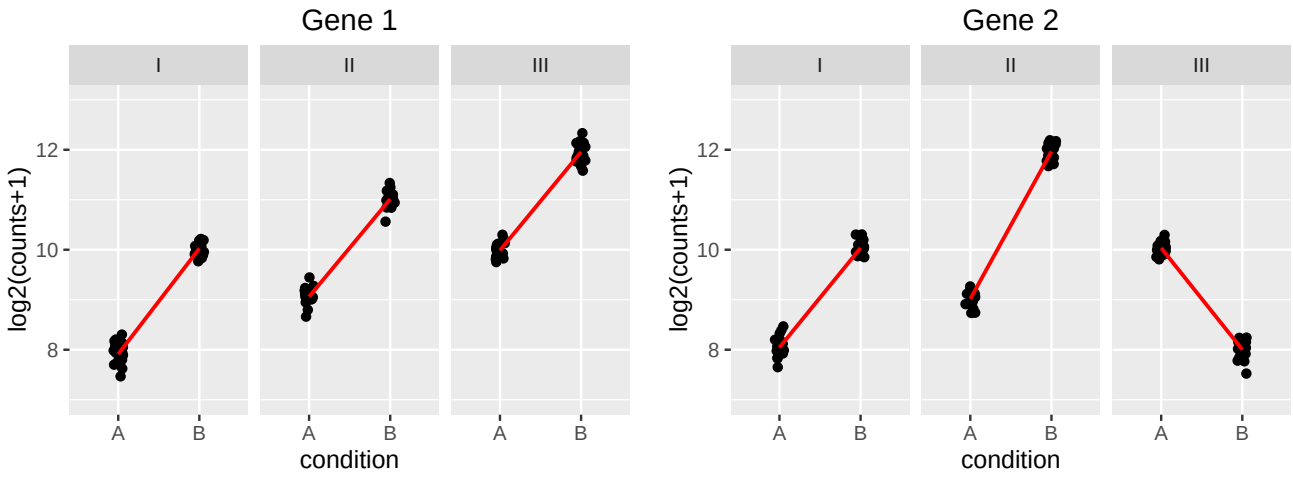

Figure 9: **Genotype-specific condition effects.** Here, the y-axis represents $\log_2(\text{counts}+1)$, and each group has 20 samples (black dots). A red line connects the mean of the groups within each genotype. On the left side (Gene 1), note that the condition effect is consistent across genotypes. 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. On the right side (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 (Section 1.5.2).

by adding `genotype:condition`, the main condition effect only represents the effect of condition for the *reference level* of genotype (I, or whichever level was defined by the user as the reference level). The interaction terms `genotypeII.conditionB` and `genotypeIII.conditionB` give the *difference* between the condition effect for a given genotype and the condition effect for the reference genotype.

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

3.4 Time-series experiments

There are a number of ways to analyze time-series experiments, depending on the biological question of interest. In order to test for any differences over multiple time points, once can use a design including the time factor, and then test using the likelihood ratio test as described in Section 3.5, 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 an RNA-seq workflow on the Bioconductor website: http://www.bioconductor.org/help/workflows/rnaseqGene/.

3.5 Likelihood ratio test

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

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

The likelihood ratio test can be specified using the `test` argument to `DESeq`, which substitutes `nbinomWaldTest` with `nbinomLRT`. In this case, the user needs to provide a reduced 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.

3.6 Approach to count outliers

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

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

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

With many degrees of freedom – i. e., many more samples than number of parameters to be estimated – it is undesirable to remove entire genes from the analysis just because their data include a single count outlier. When there are 7 or more replicates for a given sample, the `DESeq` function will automatically replace counts with large Cook's distance with the trimmed mean over all samples, scaled up by the size factor or normalization factor for that sample. This approach is conservative, it will not lead to false positives, as it replaces the outlier

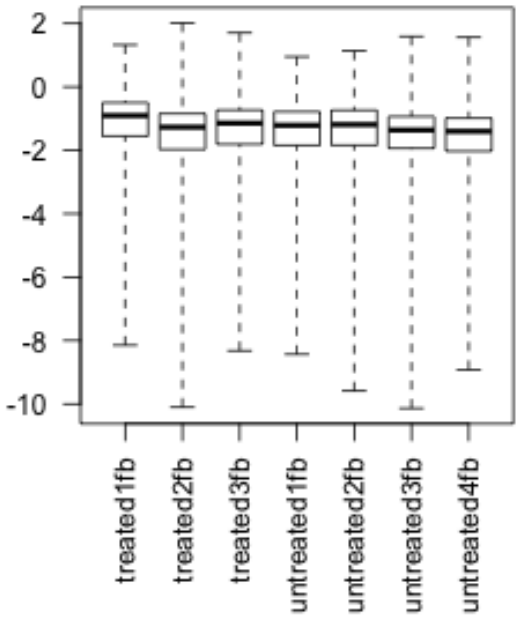

Figure 10: **Boxplot of Cook's distances.** Here we can look to see if one sample has much higher Cook's distances than the other samples. In this case, the samples all have comparable range of Cook's distances.

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 using the code example in Section 2.2.3 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:

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

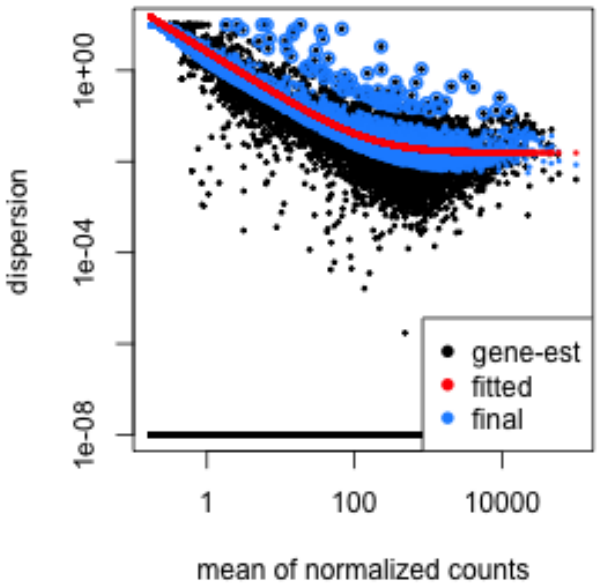

Figure 11: **Dispersion plot.** The dispersion estimate plot shows the gene-wise estimates (black), the fitted values (red), and the final maximum *a posteriori* estimates used in testing (blue).

3.7 Dispersion plot and fitting alternatives

Plotting the dispersion estimates is a useful diagnostic. The dispersion plot in Figure 11 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 man 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)
```

3.7.1 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"`.

3.7.2 Supply a custom dispersion fit

Any fitted values can be provided during dispersion estimation, using the lower-level functions described in the manual page for `estimateDispersionsGeneEst`. In the code chunk below, we store the gene-wise estimates

which were already calculated and saved in the metadata column `dispGeneEst`. Then we calculate the median value of the dispersion estimates above a threshold, and save these values as the fitted dispersions, using the replacement function for `dispersionFunction`. In the last line, the function `estimateDispersionsMAP`, uses the fitted dispersions to generate maximum *a posteriori* (MAP) estimates of dispersion.

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

3.8 Independent filtering of results

The `results` function of the *DESeq2* package performs independent filtering by default using the mean of normalized counts as a filter statistic. A threshold on the filter statistic is found which optimizes the number of adjusted p values lower than a significance level `alpha` (we use the standard variable name for significance level, though it is unrelated to the dispersion parameter α). The theory behind independent filtering is discussed in greater detail in Section 4.7. The adjusted p values for the genes which do not pass the filter threshold are set to `NA`.

The independent filtering is performed using the `filtered_p` function of the *genefilter* package, and all of the arguments of `filtered_p` can be passed to the `results` function. The filter threshold value and the number of rejections at each quantile of the filter statistic are available as metadata of the object returned by `results`. For example, we can visualize the optimization by plotting the `filterNumRej` attribute of the results object, as seen in Figure 12.

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

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

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

```
## 23.3%
##  3.47
```

```
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  7664    0 7664
##     TRUE     75  722  797
##     Sum    7739  722 8461
```

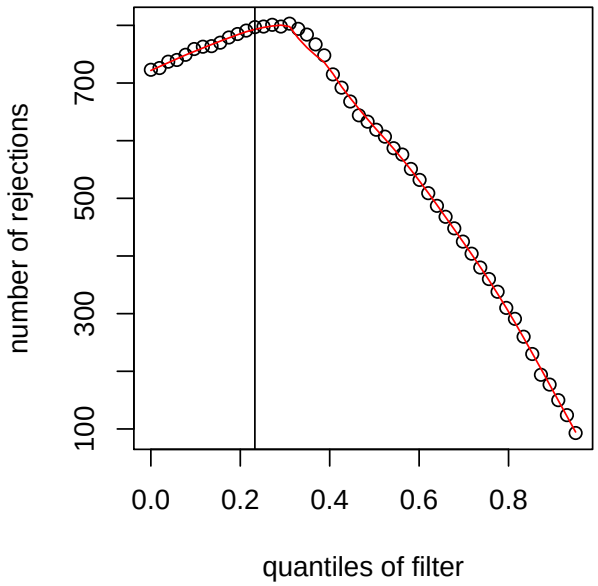

Figure 12: **Independent filtering.** The `results` function maximizes the number of rejections (adjusted p value less than a significance level), over the quantiles of a filter statistic (the mean of normalized counts). The threshold chosen (vertical line) is the lowest quantile of the filter for which 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.

3.9 Tests of log2 fold change above or below a threshold

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

- `greaterAbs` - $|\beta| > \text{lfcThreshold}$ - tests are two-tailed
- `lessAbs` - $|\beta| < \text{lfcThreshold}$ - p values are the maximum of the upper and lower tests
- `greater` - $\beta > \text{lfcThreshold}$
- `less` - $\beta < -\text{lfcThreshold}$

The test `altHypothesis="lessAbs"` requires that the user have run `DESeq` with the argument `betaPrior=FALSE`. To understand the reason for this requirement, consider that during hypothesis testing, the null hypothesis is favored unless the data provide strong evidence to reject the null. For this test, including a zero-centered prior on log fold change would favor the alternative hypothesis, shrinking log fold changes toward zero. Removing the prior on log fold changes for tests of small log fold change allows for detection of only those genes where the data alone provides evidence against the null.

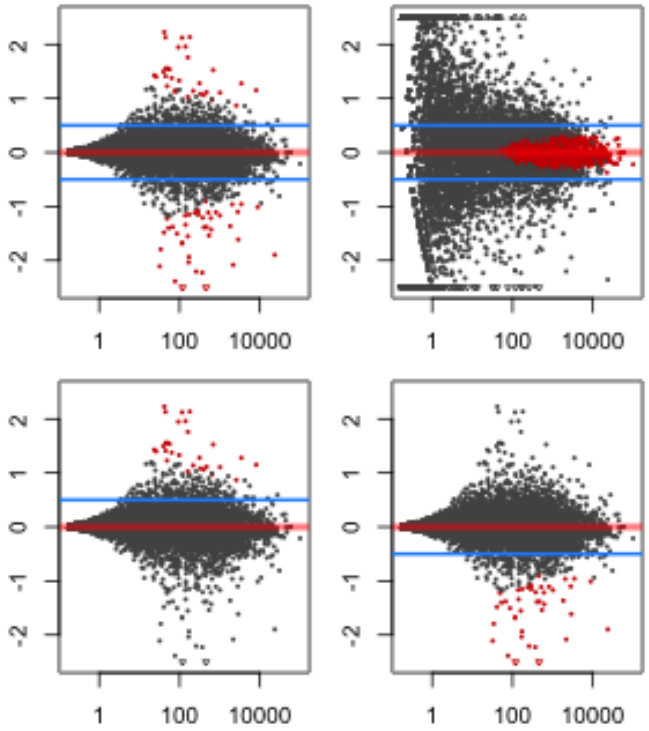

Figure 13: **MA-plots of tests of log2 fold change with respect to a threshold value.** Going left to right across rows, the tests are for `altHypothesis = "greaterAbs"`, `"lessAbs"`, `"greater"`, and `"less"`.

The four possible values of `altHypothesis` are demonstrated in the following code and visually by MA-plots in Figure 13. First we run `DESeq` and specify `betaPrior=FALSE` in order to demonstrate `altHypothesis="lessAbs"`.

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

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

```
par(mfrow=c(2,2),mar=c(2,2,1,1))
yl <- c(-2.5,2.5)

resGA <- results(dds, lfcThreshold=.5, altHypothesis="greaterAbs")
resLA <- results(ddsNoPrior, lfcThreshold=.5, altHypothesis="lessAbs")
resG <- results(dds, lfcThreshold=.5, altHypothesis="greater")
resL <- results(dds, lfcThreshold=.5, altHypothesis="less")

plotMA(resGA, ylim=yl)
abline(h=c(-.5,.5),col="dodgerblue",lwd=2)
plotMA(resLA, ylim=yl)
abline(h=c(-.5,.5),col="dodgerblue",lwd=2)
plotMA(resG, ylim=yl)
abline(h=.5,col="dodgerblue",lwd=2)
plotMA(resL, ylim=yl)
abline(h=-.5,col="dodgerblue",lwd=2)
```

3.10 Access to all calculated values

All row-wise calculated values (intermediate dispersion calculations, coefficients, standard errors, etc.) are stored in the *DESeqDataSet* object, e.g. `dds` in this vignette. These values are accessible by calling `mcols` on `dds`. Descriptions of the columns are accessible by two calls to `mcols`.

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

```
## DataFrame with 4 rows and 4 columns
##                    gene  baseMean   baseVar   allZero
##                <factor> <numeric> <numeric> <logical>
## FBgn0000008 FBgn0000008    52.226  1.55e+02     FALSE
## FBgn0000014 FBgn0000014     0.390  4.44e-01     FALSE
## FBgn0000015 FBgn0000015     0.905  7.99e-01     FALSE
## FBgn0000017 FBgn0000017  2358.243  1.16e+05     FALSE
```

```
# here using substr() only for display purposes
substr(names(mcols(dds)),1,10)
```

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

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

```
## DataFrame with 4 rows and 2 columns
##                  type                                     description
##           <character>                                     <character>
## gene            input
## baseMean  intermediate     mean of normalized counts for all samples
## baseVar   intermediate variance of normalized counts for all samples
## allZero   intermediate                all counts for a gene are zero
```

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"]])
```

```
##             treated1fb treated2fb treated3fb untreated1fb untreated2fb
## FBgn0000008     79.292     41.141     46.989       54.328       85.815
## FBgn0000014      0.593      0.308      0.352        0.409        0.646
## FBgn0000015      1.287      0.668      0.762        0.921        1.454
## FBgn0000017   3208.119   1664.555   1901.138     2659.780     4201.343
## FBgn0000018    322.080    167.114    190.865      240.407      379.743
## FBgn0000024      4.962      2.574      2.940        3.200        5.055
##             untreated3fb untreated4fb
## FBgn0000008       36.828       40.552
## FBgn0000014        0.277        0.305
## FBgn0000015        0.624        0.687
```

```
## FBgn0000017     1803.024     1985.332
## FBgn0000018      162.968      179.446
## FBgn0000024        2.169        2.389
```

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

```
##             treated1fb treated2fb treated3fb untreated1fb untreated2fb
## FBgn0000008   0.001619   0.052975   0.030610       0.0522     0.005116
## FBgn0000014   1.462285   0.100402   0.122996       0.0931     0.181793
## FBgn0000015   0.026910   0.168946   0.020971       0.1773     0.031421
## FBgn0000017   0.000413   0.000182   0.004514       0.0361     0.054716
## FBgn0000018   0.209917   0.078048   0.048382       0.2022     0.000436
## FBgn0000024   0.065175   0.627158   0.000355       0.0382     0.170217
##             untreated3fb untreated4fb
## FBgn0000008       0.4135      0.25905
## FBgn0000014       0.3425      0.05883
## FBgn0000015       0.0362      0.42199
## FBgn0000017       0.1121      0.10367
## FBgn0000018       0.1291      0.00422
## FBgn0000024       0.0974      0.39125
```

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

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

```
## [1] 0.0538 6.3982 1.7320 0.0133 0.0220 0.5951
```

```
# which is the same as
head(mcols(dds)$dispersion)
```

```
## [1] 0.0538 6.3982 1.7320 0.0133 0.0220 0.5951
```

The size factors s_j are accessible via `sizeFactors`:

```
sizeFactors(dds)
```

```
##   treated1fb   treated2fb   treated3fb untreated1fb untreated2fb untreated3fb
##        1.512        0.784        0.896        1.050        1.659        0.712
## untreated4fb
##        0.784
```

For advanced users, we also include a convenience function `coef` for extracting the matrix of coefficients $[\beta_{ir}]$ for all genes i and parameters r, as in the formula in Section 4.1. 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 conditionuntreated conditiontreated
## FBgn0000008     5.703           -0.00986          0.00986
## FBgn0000014    -1.355           -0.00591          0.00591
## FBgn0000015    -0.211            0.02147         -0.02147
## FBgn0000017    11.179            0.12768         -0.12768
```

```
## FBgn0000018     7.787            0.05192         -0.05192
## FBgn0000024     1.661           -0.05340          0.05340
```

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

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

```
##          Intercept conditionuntreated   conditiontreated
##           1.00e+06           1.05e-01           1.05e-01
```

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
## <environment: 0x7fdd479f0ed0>
## attr(,"coefficients")
## asymptDisp  extraPois
##     0.0154     2.5652
## attr(,"fitType")
## [1] "parametric"
## attr(,"varLogDispEsts")
## [1] 0.961
## attr(,"dispPriorVar")
## [1] 0.47
```

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

```
## [1] 0.47
```

3.11 Sample-/gene-dependent normalization factors

In some experiments, there might be gene-dependent dependencies which vary across samples. For instance, GC-content bias or length bias might vary across samples coming from different labs or processed at different times. We use the terms "normalization factors" for a gene $\times$ sample matrix, and "size factors" for a single number per sample. Incorporating normalization factors, the mean parameter μ_{ij} from Section 4.1 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` in the steps described in Section 3.1. Normalization factors, if present, will always be used in the place of size factors.

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

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

3.12 "Model matrix not full rank"

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

3.12.1 Linear combinations

The simplest case is the linear combination, or linear dependency problem, when two variables contain exactly the same information, such as in the following sample table. The software cannot fit an effect for `batch` and `condition`, because they produce identical columns in the model matrix. This is also referred to as "perfect confounding". A unique solution of coefficients (the β_i in the formula in Section 4.1) is not possible.

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

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

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

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

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

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

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

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

This design can be analyzed by *DESeq2* but requires a bit of refactoring in order to fit the model terms. Here we will use a trick described in the *edgeR* user guide, from the section "Comparisons Both Between and Within Subjects". If we try to analyze with a formula such as, $\sim$ `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, use a dummy design, such as $\sim$ `1`. 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(1:2,each=2),2))
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   Y   3   A     1
## 6   Y   3   B     1
## 7   Y   4   A     2
## 8   Y   4   B     2
```

Now we can reassign our *DESeqDataSet* a design of $\sim$ `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:cndB grpY:cndB
## 1           1    0           0           0         0         0
## 2           1    0           0           0         1         0
## 3           1    0           1           0         0         0
## 4           1    0           1           0         1         0
## 5           1    1           0           0         0         0
## 6           1    1           0           0         0         1
## 7           1    1           0           1         0         0
## 8           1    1           0           1         0         1
## attr(,"assign")
## [1] 0 1 2 2 3 3
## attr(,"contrasts")
## attr(,"contrasts")$grp
## [1] "contr.treatment"
##
## attr(,"contrasts")$ind.n
## [1] "contr.treatment"
##
## attr(,"contrasts")$cnd
## [1] "contr.treatment"
```

where the terms `grpX.cndB` and `grpY.cndB` give the group-specific condition effects. These 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"))
```

3.12.2 Levels without samples

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

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

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

```
##    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"
```

We can remove this column like so:

```
m1 <- m1[,-9]
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 $\sim$ `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`.

4 Theory behind DESeq2

4.1 The DESeq2 model

The *DESeq2* model and all the steps taken in the software are described in detail in our publication [1], 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:

$$K_{ij} \sim \mathrm{NB}(\mu_{ij}, \alpha_i)$$
$$\mu_{ij} = s_j q_{ij}$$
$$\log_2(q_{ij}) = x_{j\cdot}\beta_i$$

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[5] 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.

By default these log2 fold changes are the maximum *a posteriori* estimates after incorporating a zero-centered Normal prior – in the software referrred to as a β-prior – hence *DESeq2* provides "moderated" log2 fold change estimates. 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* [7, 8]. The steps performed by the `DESeq` function are documented in its manual page; 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 Section 3.10

4.2 Changes compared to the DESeq package

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.
- 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.
- 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 [9].
- 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.

[5] The model can be generalized to use sample- **and** gene-dependent normalization factors, see Appendix 3.11.

- It is possible to provide a matrix of sample-/gene-dependent normalization factors (Section 3.11).
- Automatic independent filtering on the mean of normalized counts (Section 4.7).
- Automatic outlier detection and handling (Section 4.4).

4.3 Methods changes since the 2014 DESeq2 paper

- 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 man 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 designs with interaction terms, the solution described in the paper is no longer used (log fold change shrinkage only applied to interaction terms). Instead, *DESeq2* now turns off log fold change shrinkage for all terms if an interaction term is present (`betaPrior=FALSE`). While the inference on interaction terms was correct with `betaPrior=TRUE`, the interpretation of the individual terms and the extraction of contrasts was too confusing. This change was introduced in version 1.10 (October 2015).
- 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 a list of all changes since version 1.0.0, see the NEWS file included in the package.

4.4 Count outlier detection

DESeq2 relies on the negative binomial distribution to make estimates and perform statistical inference on differences. While the negative binomial is versatile in having a mean and dispersion parameter, extreme counts in individual samples might not fit well to the negative binomial. For this reason, we perform automatic detection of count outliers. We use Cook's distance, which is a measure of how much the fitted coefficients would change if an individual sample were removed [10]. For more on the implementation of Cook's distance see Section 3.6 and 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))
```

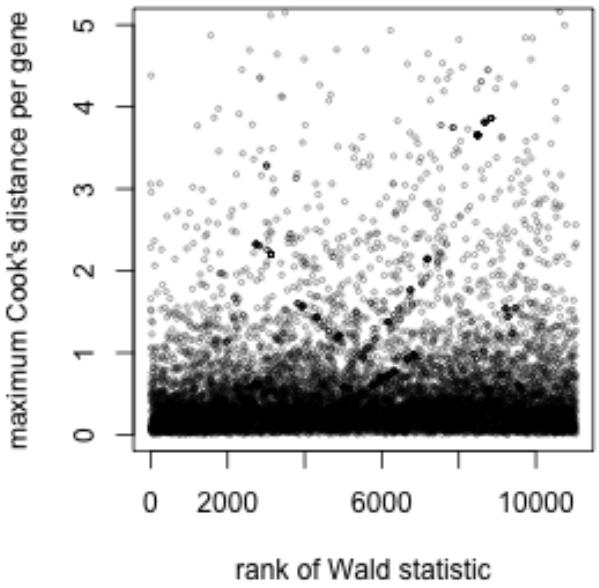

Figure 14: **Cook's distance.** Plot of the maximum Cook's distance per gene over the rank of the Wald statistics for the condition. The two regions with small Cook's distances are genes with a single count in one sample. The horizontal line is the default cutoff used for 7 samples and 3 estimated parameters.

4.5 Contrasts

Contrasts can be calculated for a *DESeqDataSet* object for which the GLM coefficients have already been fit using the Wald test steps (`DESeq` with `test="Wald"` or using `nbinomWaldTest`). The vector of coefficients 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.

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

4.6 Expanded model matrices

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

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

4.7 Independent filtering and multiple testing

4.7.1 Filtering criteria

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

A good choice for a filtering criterion is one that

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

The benefit from filtering relies on property 2, and we will explore it further in Section 4.7.2. 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 [11] 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 (Figure 15), 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))
```

4.7.2 Why does it work?

Consider the p value histogram in Figure 16. 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 $[0, 1]$.

```
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)
```

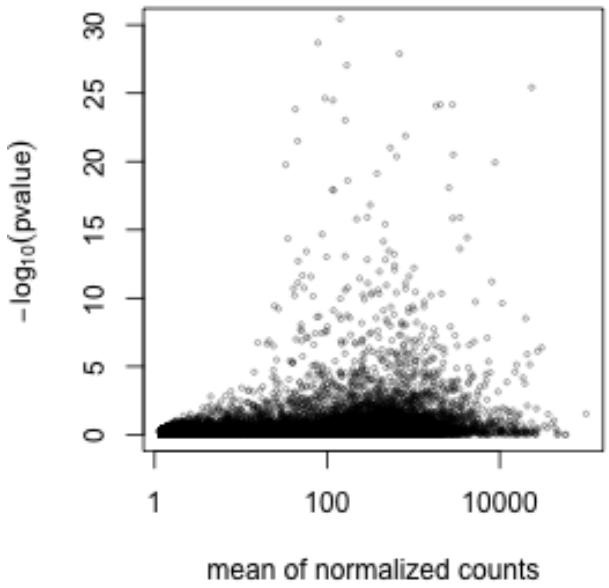

Figure 15: **Mean counts as a filter statistic.** The mean of normalized counts provides an independent statistic for filtering the tests. It is independent because the information about the variables in the design formula is not used. By filtering out genes which fall on the left side of the plot, the majority of the low p values are kept.

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

```
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)))
```

5 Frequently asked questions

5.1 How can I get support for DESeq2?

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

- all questions should take place on the Bioconductor support site: https://support.bioconductor.org, which serves as a repository of questions and answers. This helps to save the developers' time in responding to similar questions. Make sure to tag your post with "deseq2". It is often very helpful in addition to describe the aim of your experiment.

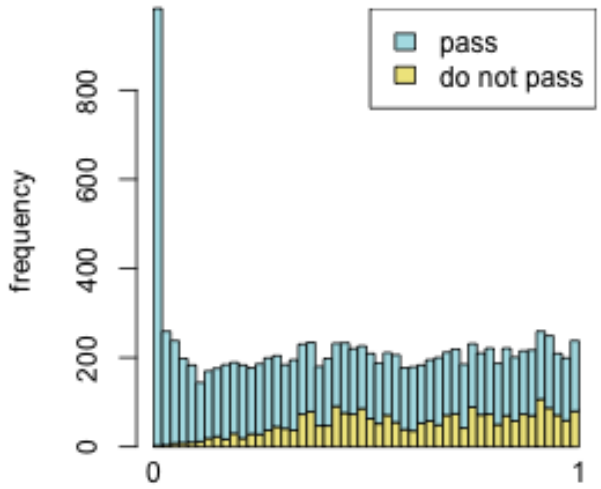

Figure 16: **Histogram of p values for all tests.** The area shaded in blue indicates the subset of those that pass the filtering, the area in khaki those that do not pass.

- before posting, first search the Bioconductor support site mentioned above for past threads which might have answered your question.
- if you have a question about the behavior of a function, read the sections of the manual page for this function by typing a question mark and the function name, e.g. `?results`. We spend a lot of time documenting individual functions and the exact steps that the software is performing.
- include all of your R code, especially the creation of the *DESeqDataSet* and the design formula. Include complete warning or error messages, and conclude your message with the full output of `sessionInfo()`.
- if possible, include the output of `as.data.frame(colData(dds))`, so that we can have a sense of the experimental setup. If this contains confidential information, you can replace the levels of those factors using `levels()`.

5.2 Why are some p values set to NA?

See the details in Section 1.5.3.

5.3 How can I get unfiltered DESeq 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.

5.4 How do I use the variance stabilized or rlog transformed 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 in Section 1.4.

5.5 Can I use DESeq2 to analyze paired samples?

Yes, you should use a multi-factor design which includes the sample information as a term in the design formula. This will account for differences between the samples while estimating the effect due to the condition. The condition of interest should go at the end of the design formula. See Section 1.6.

5.6 Can I run DESeq2 to contrast the levels of 100 groups?

DESeq2 will work with any kind of design specified using the R formula. We enourage users to consider exploratory data analysis such as principal components analysis as described in Section 2.2.3, rather than performing statistical testing of all combinations of dozens of groups.

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

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

If a *DESeqDataSet* is provided with an experimental design without replicates, a message is printed, that the samples are treated as replicates for estimation of dispersion. More details can be found in the manual page for ?DESeq.

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

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

5.9 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.

5.10 Is there an official Galaxy tool for DESeq2?

Yes. The repository for the *DESeq2* tool is https://github.com/galaxyproject/tools-iuc/tree/master/tools/deseq2 and a link to its location in the Tool Shed is https://toolshed.g2.bx.psu.edu/view/iuc/deseq2/d983d19fbbab.

6 Acknowledgments

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 Oleś, Aleksandra Pekowska, Felix Klein, Vince Carey, 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, Björn Grüning, Ryan McMinds.

7 Session Info

- R version 3.2.3 (2015-12-10), `x86_64-apple-darwin13.4.0`
- Locale: `C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: Biobase 2.30.0, BiocGenerics 0.16.1, DESeq2 1.10.1, GenomeInfoDb 1.6.1, GenomicRanges 1.22.2, IRanges 2.4.6, RColorBrewer 1.1-2, Rcpp 0.12.2, RcppArmadillo 0.6.400.2.2, S4Vectors 0.8.5, SummarizedExperiment 1.0.1, airway 0.104.0, ggplot2 2.0.0, knitr 1.11, pasilla 0.10.0, pheatmap 1.0.8, vsn 3.38.0
- Loaded via a namespace (and not attached): AnnotationDbi 1.32.2, BiocInstaller 1.20.1, BiocParallel 1.4.3, BiocStyle 1.8.0, DBI 0.3.1, DESeq 1.22.0, Formula 1.2-1, Hmisc 3.17-1, RSQLite 1.0.0, XML 3.98-1.3, XVector 0.10.0, acepack 1.3-3.3, affy 1.48.0, affyio 1.40.0, annotate 1.48.0, cluster 2.0.3, codetools 0.2-14, colorspace 1.2-6, digest 0.6.8, evaluate 0.8, foreign 0.8-66, formatR 1.2.1, futile.logger 1.4.1, futile.options 1.0.0, genefilter 1.52.0, geneplotter 1.48.0, grid 3.2.3, gridExtra 2.0.0, gtable 0.1.2, hexbin 1.27.1, highr 0.5.1, labeling 0.3, lambda.r 1.1.7, lattice 0.20-33, latticeExtra 0.6-26, limma 3.26.3, locfit 1.5-9.1, magrittr 1.5, munsell 0.4.2, nnet 7.3-11, plyr 1.8.3, preprocessCore 1.32.0, rpart 4.1-10, scales 0.3.0, splines 3.2.3, stringi 1.0-1, stringr 1.0.0, survival 2.38-3, tools 3.2.3, xtable 1.8-0, zlibbioc 1.16.0

References

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

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

[3] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*,

pages 193–202, 2011. URL: http://genome.cshlp.org/cgi/doi/10.1101/gr.108662.110, doi:10.1101/gr.108662.110.

[4] Robert Tibshirani. Estimating transformations for regression via additivity and variance stabilization. *Journal of the American Statistical Association*, 83:394–405, 1988.

[5] Wolfgang Huber, Anja von Heydebreck, Holger Sültmann, Annemarie Poustka, and Martin Vingron. Parameter estimation for the calibration and variance stabilization of microarray data. *Statistical Applications in Genetics and Molecular Biology*, 2(1):Article 3, 2003.

[6] Simon Anders and Wolfgang Huber. Differential expression analysis for sequence count data. *Genome Biology*, 11:R106, 2010. URL: http://genomebiology.com/2010/11/10/R106.

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

[8] Davis J McCarthy, Yunshun Chen, and Gordon K Smyth. Differential expression analysis of multifactor RNA-Seq experiments with respect to biological variation. *Nucleic Acids Research*, 40:4288–4297, January 2012. URL: http://www.ncbi.nlm.nih.gov/pubmed/22287627, doi:10.1093/nar/gks042.

[9] Hao Wu, Chi Wang, and Zhijin Wu. A new shrinkage estimator for dispersion improves differential expression detection in RNA-seq data. *Biostatistics*, September 2012. URL: http://dx.doi.org/10.1093/biostatistics/kxs033, doi:10.1093/biostatistics/kxs033.

[10] R. Dennis Cook. Detection of Influential Observation in Linear Regression. *Technometrics*, February 1977.

[11] Richard Bourgon, Robert Gentleman, and Wolfgang Huber. Independent filtering increases detection power for high-throughput experiments. *PNAS*, 107(21):9546–9551, 2010. URL: http://www.pnas.org/content/107/21/9546.long.