Differential analysis of count data – the DESeq2 package

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

[1] Department of Biostatistics, Dana Farber Cancer Institute and
Harvard School of Public Health, Boston, US;
[2] European Molecular Biology Laboratory (EMBL), Heidelberg, Germany
[*]michaelisaiahlove (at) gmail.com

May 12, 2014

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 work flows. Another vignette, "Beginner's guide to using the DESeq2 package", covers similar material but at a slower pace, including the generation of count tables from FASTQ files.

DESeq2 version: 1.4.5

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. bioRxiv (2014). doi:10.1101/002832 [1]

[1]Other *Bioconductor* packages with similar aims are *edgeR*, *baySeq* and *DSS*.

Contents

1 Standard workflow

1.1 Quick start

Here we show the most basic steps for a differential expression analysis. These steps imply you have a *SummarizedExperiment* object `se` with a column `condition` in `colData(se)`.

```
dds <- DESeqDataSet(se = se, design = ~ condition)
dds <- DESeq(dds)
res <- results(dds)
```

1.2 Input data

1.2.1 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).

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

1.2.2 SummarizedExperiment input

The class used by the *DESeq2* package to store the read counts is *DESeqDataSet* which extends the *SummarizedExperiment* class of the *GenomicRanges* 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 *SummarizedExperiment* object (*easyRNASeq* is another *Bioconductor* package which can prepare *SummarizedExperiment* objects as input for *DESeq2*). An example of the steps to produce a *SummarizedExperiment* can be found in the data package *parathyroidSE*, which summarizes RNA-Seq data from experiments on 4 human cell cultures [2]. In this last line of the following code chunk, note that `dds$variable` is equivalent to `colData(dds)$variable`.

```
library("parathyroidSE")
data("parathyroidGenesSE")
se <- parathyroidGenesSE
colnames(se) <- se$run
```

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 ($\sim$) 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 *SummarizedExperiment* `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")
ddsPara <- DESeqDataSet(se = se, design = ~ patient + treatment)
ddsPara$treatment <- factor(ddsPara$treatment,
                            levels=c("Control","DPN","OHT"))
ddsPara
```

```
## class: DESeqDataSet
## dim: 63193 27
## exptData(1): MIAME
## assays(1): counts
## rownames(63193): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
## rowData metadata column names(0):
## colnames(27): SRR479052 SRR479053 ... SRR479077 SRR479078
```

```
## colData names(8): run experiment ... study sample
```

1.2.3 Count matrix input

Alternatively, if you already have prepared a matrix of read counts, you can use the function `DESeqDataSetFromMatr`
For this function you should provide the counts matrix, the column information as a *DataFrame* or *data.frame* and the design formula.

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

Now that we have a matrix of counts and the column information, we can construct a *DESeqDataSet*:

```
dds <- DESeqDataSetFromMatrix(countData = countData,
                              colData = colData,
                              design = ~ condition)
dds$condition <- factor(dds$condition,
                        levels=c("untreated","treated"))
dds
```

```
## class: DESeqDataSet
## dim: 14470 7
## exptData(0):
## assays(1): counts
## rownames(14470): FBgn0000003 FBgn0000008 ... FBgn0261574 FBgn0261575
## rowData metadata column names(0):
## colnames(7): treated1fb treated2fb ... untreated3fb untreated4fb
## colData names(2): condition type
```

1.2.4 HTSeq input

If you have `htseq-count` from the *HTSeq* python package[2] you can use the function `DESeqDataSetFromHTSeqCoun`
For an example of using the python scripts, see the *pasilla* or *parathyroid* 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.

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

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

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

1.2.5 Note on factor levels

In the three examples above, we applied the function `factor` to the column of interest in `colData`, supplying a character vector of levels. It is important to supply levels (otherwise the levels are chosen in alphabetical order) and to put the "control" or "untreated" level as the first element ("base level"), so that the log2 fold changes produced by default will be the expected comparison against the base level. An *R* function for easily changing the base level is `relevel`. An example of setting the base level of a factor with `relevel` is:

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

In addition, when subsetting 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 are 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)
```

1.2.6 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* [4]. The detailed transcript of the production of the *pasilla* data is provided in the vignette of the data package *pasilla*.

1.3 Differential expression analysis

The standard differential expression analysis steps are wrapped into a single function, `DESeq`. The steps of this function are described in Section 4.1 and in the manual page for `?DESeq`. 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.

```
dds <- DESeq(dds)
res <- results(dds)
resOrdered <- res[order(res$padj),]
head(resOrdered)
```

```
## 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>
## FBgn0039155       453          -4.28     0.192     -22.3 3.58e-110 2.76e-106
## FBgn0029167      2165          -2.18     0.108     -20.2  1.02e-90  3.93e-87
## FBgn0035085       367          -2.44     0.151     -16.2  6.05e-59  1.56e-55
## FBgn0029896       258          -2.51     0.182     -13.8  3.88e-43  7.50e-40
## FBgn0034736       118          -3.17     0.238     -13.3  1.56e-40  2.42e-37
## FBgn0040091       611          -1.53     0.128     -11.9  7.46e-33  9.61e-30
```

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 counts for each gene, optimizing the number of genes which will have an adjusted p value below a given threshold. This will be discussed further in Section 3.8.

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*

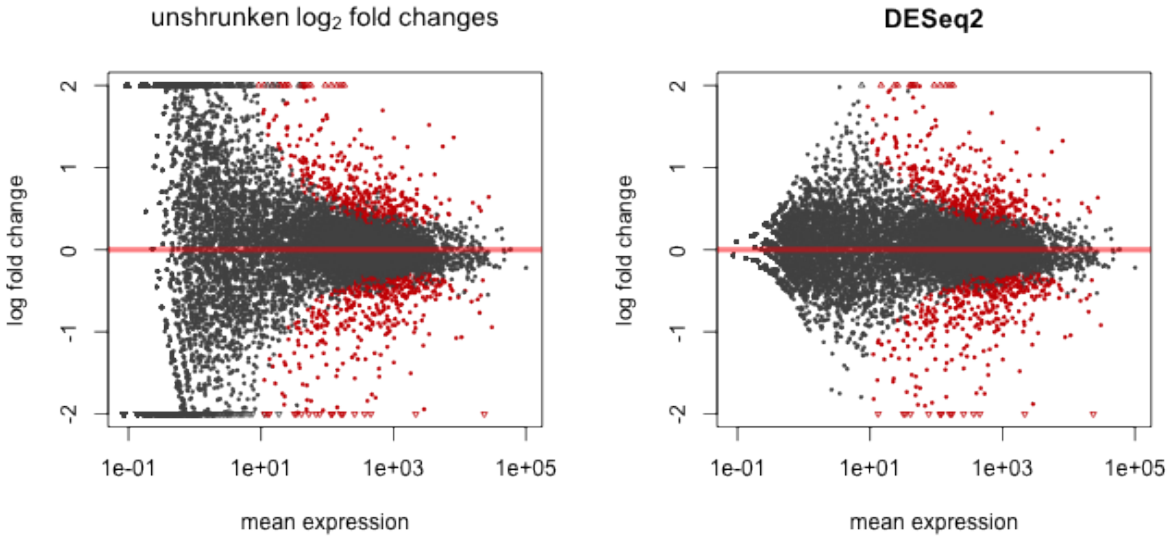

Figure 1: **MA-plot.** These plots show the log2 fold changes from the treatment over the mean of normalized counts, i.e. the average of counts normalized by size factors. The left plot shows the "unshrunken" log2 fold changes, while the right plot, produced by the code above, shows the shrinkage of 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.

returned by `DESeq`. Multi-factor designs are discussed further in Section 1.5, and the use of the `contrast` argument is dicussed in Section 3.2.

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.

1.4 Exploring and exporting results

1.4.1 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))
```

The `plotMA` function can also take the `DESeqDataSet` as its first argument, in which case `results` will be called internally.

1.4.2 More information on results columns

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

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

```
## [1] "the base mean over all rows"
## [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 change in observed 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 that some values in the results table can be set to `NA`, for either 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 are 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 3.5,
3. If a row is filtered by automatic independent filtering, based on low mean normalized count, then only the adjusted p value is set to `NA`. Description and customization of independent filtering is described in Section 3.8.

1.4.3 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")
```

1.5 Multi-factor designs

Experiments with more than one factor influencing the counts can be easily analyzed using model formulae including the additional variables. 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>
## FBgn0000003     0.159         0.1462     0.228    0.6421    0.5208        NA
## FBgn0000008    52.226         0.0127     0.292    0.0434    0.9653     0.980
## FBgn0000014     0.390         0.0531     0.327    0.1621    0.8712        NA
## FBgn0000015     0.905        -0.1768     0.473   -0.3734    0.7088        NA
## FBgn0000017  2358.243        -0.2716     0.119   -2.2781    0.0227     0.129
## FBgn0000018   221.242        -0.0686     0.160   -0.4281    0.6686     0.863
```

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. Contrasts are described in more detail 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>
## FBgn0000003     0.159       -0.10999     0.210   -0.5231    0.6009        NA
## FBgn0000008    52.226       -0.08898     0.288   -0.3087    0.7576     0.877
## FBgn0000014     0.390        0.03197     0.304    0.1051    0.9163        NA
## FBgn0000015     0.905       -0.30627     0.444   -0.6893    0.4906        NA
## FBgn0000017  2358.243        0.00752     0.119    0.0632    0.9496     0.982
## FBgn0000018   221.242        0.30050     0.159    1.8941    0.0582     0.212
```

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 1.3. 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. One method incorporates priors on the sample differences [1], and the other uses the concept of variance stabilizing transformations [5, 6, 7].

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 explanable 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 shrinks the tranformed 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 is used in the transformation. So setting `blind` to `FALSE` is still mostly unbiased by the information about the samples.

2.1.2 Extracting transformed values

The two functions return *SummarizedExperiment* objects, as the data are no longer counts. The `assay` function is used to extract the matrix of normalized values.

```
rld <- rlog(dds)
vsd <- varianceStabilizingTransformation(dds)
rlogMat <- assay(rld)
vstMat <- assay(vsd)
```

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.

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 2. 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.

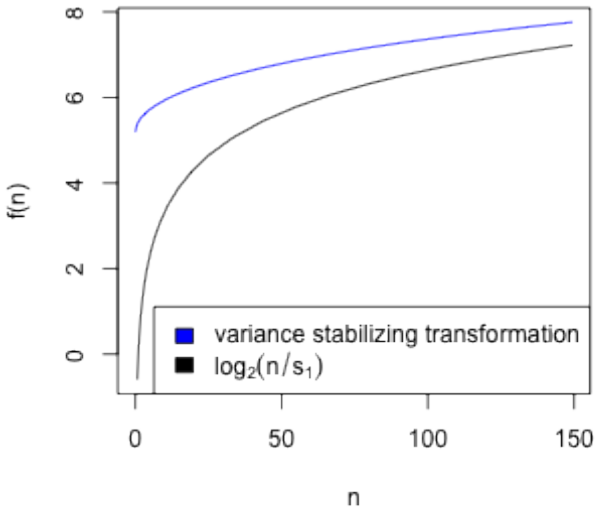

Figure 2: **VST and log2.** Graphs of the variance stabilizing transformation for sample 1, in blue, and of the transformation $f(n) = \log_2(n/s_1)$, in black. n are the counts and s_1 is the size factor for the first sample.

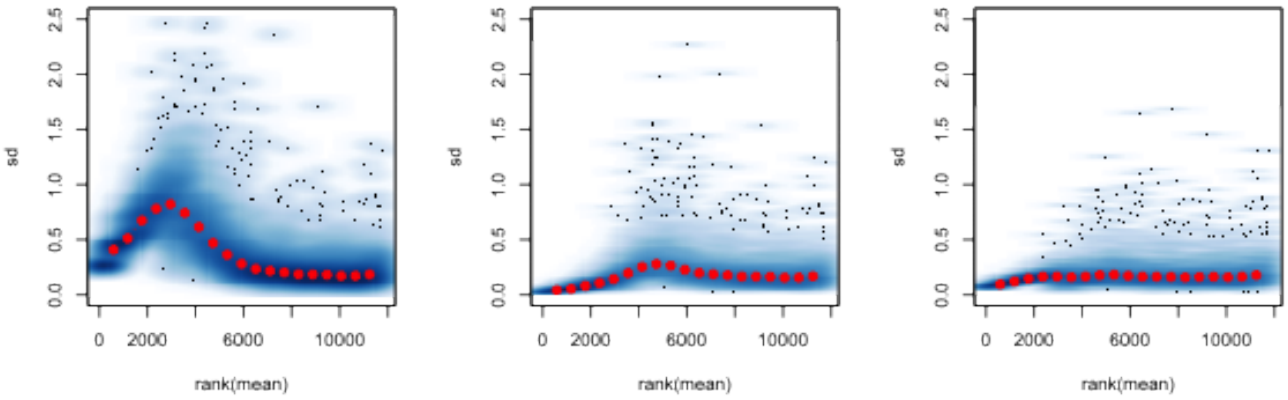

Figure 3: Per-gene standard deviation (taken across samples), against the rank of the mean, for the shifted logarithm $\log_2(n+1)$ (left), the regularized log transformation (center) and the variance stabilizing transformation (right).

2.1.5 Effects of transformations on the variance

Figure 3 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")
par(mfrow=c(1,3))
notAllZero <- (rowSums(counts(dds))>0)
meanSdPlot(log2(counts(dds,normalized=TRUE)[notAllZero,] + 1),
           ylim = c(0,2.5))
meanSdPlot(assay(rld[notAllZero,]), ylim = c(0,2.5))
meanSdPlot(assay(vsd[notAllZero,]), ylim = c(0,2.5))
```

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*[3]. 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 table

To explore a count table, it is often instructive to look at it as a heatmap. Below we show how to produce such a heatmap from the raw and transformed data.

```
library("RColorBrewer")
library("gplots")
select <- order(rowMeans(counts(dds,normalized=TRUE)),decreasing=TRUE)[1:30]
hmcol <- colorRampPalette(brewer.pal(9, "GnBu"))(100)
```

```
heatmap.2(counts(dds,normalized=TRUE)[select,], col = hmcol,
          Rowv = FALSE, Colv = FALSE, scale="none",
          dendrogram="none", trace="none", margin=c(10,6))
```

```
heatmap.2(assay(rld)[select,], col = hmcol,
          Rowv = FALSE, Colv = FALSE, scale="none",
          dendrogram="none", trace="none", margin=c(10, 6))
```

```
heatmap.2(assay(vsd)[select,], col = hmcol,
          Rowv = FALSE, Colv = FALSE, scale="none",
          dendrogram="none", trace="none", margin=c(10, 6))
```

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

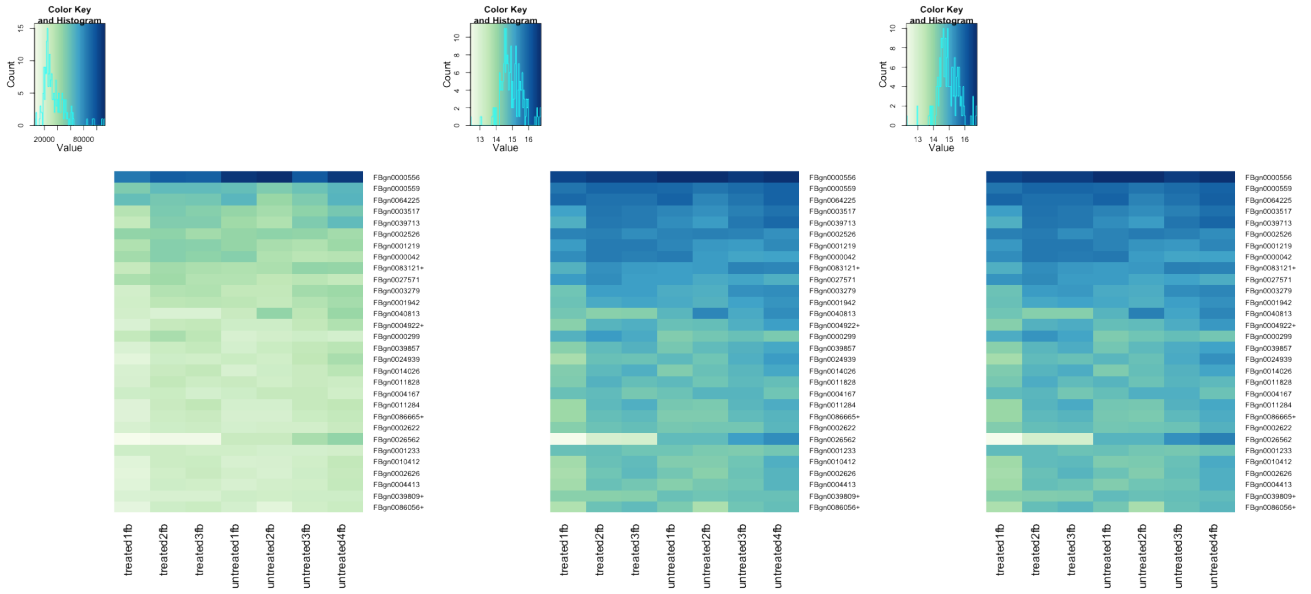

Figure 4: Heatmaps showing the expression data of the 30 most highly expressed genes. The data is of raw counts (left), from regularized log transformation (center) and from variance stabilizing transformation (right).

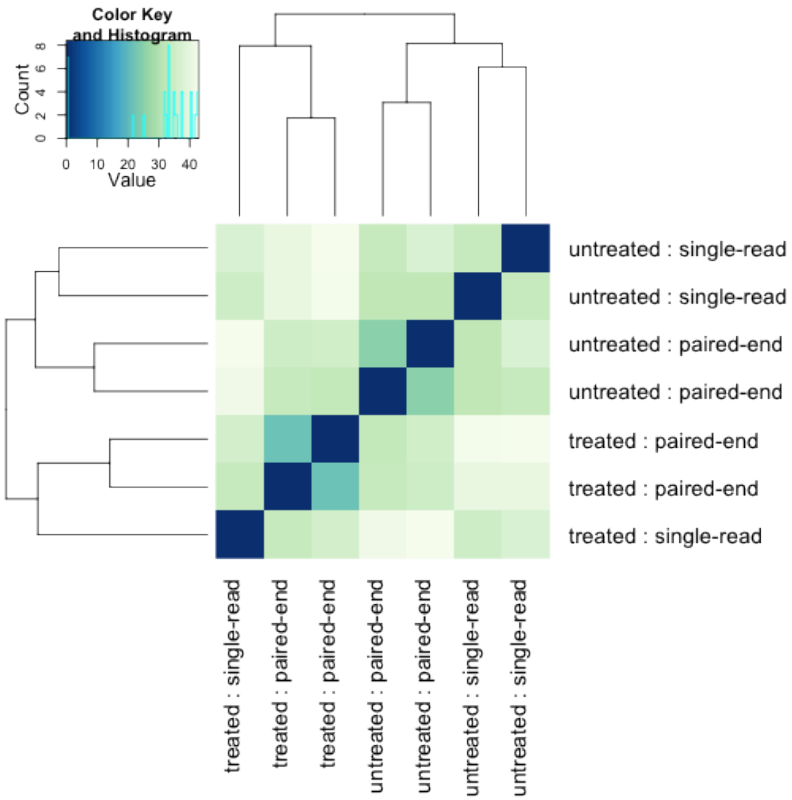

Figure 5: **Sample-to-sample distances.** Heatmap showing the Euclidean distances between the samples as calculated from the regularized log transformation.

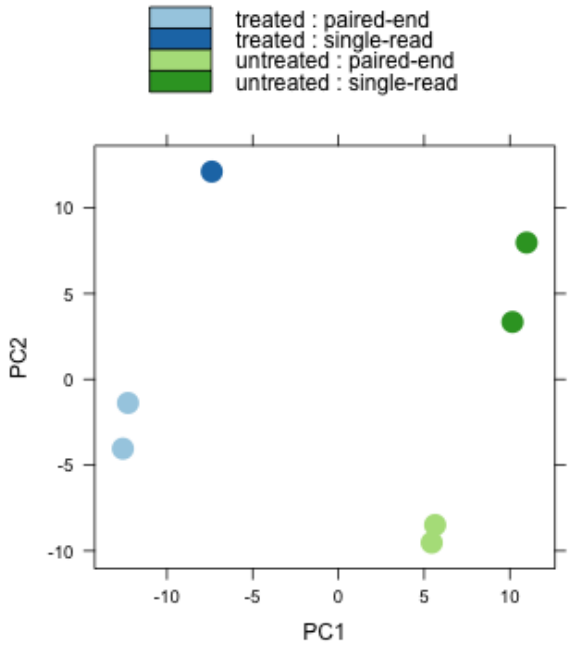

Figure 6: **PCA plot.** PCA plot. The 7 samples shown in the 2D plane spanned by their first two principal components. This type of plot is useful for visualizing the overall effect of experimental covariates and batch effects.

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.

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

A heatmap of this distance matrix gives us an overview over similarities and dissimilarities between samples (Figure 5):

```
mat <- as.matrix(distsRL)
rownames(mat) <- colnames(mat) <- with(colData(dds),
                                       paste(condition, type, sep=" : "))
heatmap.2(mat, trace="none", col = rev(hmcol), margin=c(13, 13))
```

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 6).

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

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 factor level means, which can be used to test if differences between groups are actually 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 other three possible pairs will simply have $-1\times$ the log2 fold changes of these three).

In order to fit models with "shrunken" log2 fold changes in a manner which is independent to the choice of base level, *DESeq2* uses "expanded model matrices", described further in Section 4.5. The expanded model matrices include a coefficient for each level of the factors in addition to an intercept. The `contrast` argument of `results` function is again used to extract test results of log2 fold changes of interest.

Here we show how to perform contrasts using the parathyroid dataset which was built in Section 1.2.2. The three levels of the factor `treatment` are: Control, DPN and OHT. The samples are also split according to the patient from which the cell cultures were derived, so we include this in the design formula.

```
ddsCtrst <- ddsPara[, ddsPara$time == "48h"]
as.data.frame(colData(ddsCtrst)[,c("patient","treatment")])
```

```
##           patient treatment
## SRR479053       1   Control
## SRR479055       1       DPN
## SRR479057       1       OHT
## SRR479059       2   Control
## SRR479062       2       DPN
## SRR479065       2       OHT
## SRR479067       3   Control
## SRR479069       3       DPN
## SRR479071       3       OHT
## SRR479072       4   Control
## SRR479074       4       DPN
## SRR479075       4       DPN
```

```
## SRR479077       4       OHT
## SRR479078       4       OHT
```

```
design(ddsCtrst) <- ~ patient + treatment
```

First we run DESeq:

```
ddsCtrst <- DESeq(ddsCtrst)
```

Using the `contrast` argument of the `results` function, we can specify a test of OHT vs DPN. The contrast argument takes a character vector of length three, containing the name of the factor, the name of the numerator level, and the name of the denominator level, where we test the log2 fold change of numerator vs denominator. Here we extract the results for the log2 fold change of OHT vs DPN for the treatment factor.

```
resCtrst <- results(ddsCtrst, contrast=c("treatment","OHT","DPN"))
head(resCtrst,2)
```

```
## log2 fold change (MAP): treatment OHT vs DPN
## Wald test p-value: treatment OHT vs DPN
## DataFrame with 2 rows and 6 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003   526.077        -0.0552    0.0565    -0.976     0.329
## ENSG00000000005     0.674         0.0317    0.1506     0.210     0.833
##                      padj
##                 <numeric>
## ENSG00000000003     0.756
## ENSG00000000005        NA
```

Additionally, `contrast` can take a list of length two, where the first element is a character vector of effects for the numerator of the contrast and the second element is a character vector of effects for the denominator of the contrast. The names in the list should be elements of `resultsNames(dds)`. The list contruction allows for multiple effects to be added together in either the numerator or the denominator, e.g., main effects and interaction effects. Here we produce the same results table by specifying a list to `contrast`.

```
resultsNames(ddsCtrst)
```

```
## [1] "Intercept"        "patient1"         "patient2"         "patient3"
## [5] "patient4"         "treatmentControl" "treatmentDPN"     "treatmentOHT"
```

```
resCtrst <- results(ddsCtrst, contrast=list("treatmentOHT","treatmentDPN"))
head(resCtrst,2)
```

```
## log2 fold change (MAP): treatmentOHT vs treatmentDPN
## Wald test p-value: treatmentOHT vs treatmentDPN
## DataFrame with 2 rows and 6 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
```

```
##                  <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003   526.077        -0.0552    0.0565    -0.976     0.329
## ENSG00000000005     0.674         0.0317    0.1506     0.210     0.833
##                      padj
##                 <numeric>
## ENSG00000000003     0.756
## ENSG00000000005        NA
```

For advanced users, a numeric contrast vector can also be provided with one element for each element provided by `resultsNames`, i.e. columns of the model matrix. Note that the following contrast is the same as specified by the character vector and the list in the previous two code chunks.

```
resultsNames(ddsCtrst)
```

```
## [1] "Intercept"        "patient1"         "patient2"         "patient3"
## [5] "patient4"         "treatmentControl" "treatmentDPN"     "treatmentOHT"
```

```
resCtrst <- results(ddsCtrst, contrast=c(0,0,0,0,0,0,-1,1))
head(resCtrst,2)
```

```
## log2 fold change (MAP): 0,0,0,0,0,0,-1,+1
## Wald test p-value: 0,0,0,0,0,0,-1,+1
## DataFrame with 2 rows and 6 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003   526.077        -0.0552    0.0565    -0.976     0.329
## ENSG00000000005     0.674         0.0317    0.1506     0.210     0.833
##                      padj
##                 <numeric>
## ENSG00000000003     0.756
## ENSG00000000005        NA
```

The formula that is used to generate the contrasts can be found in Section 4.4.

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 for different groups of samples. There are a variety of models involving interaction terms, but here we will show an example using the parathyroid dataset introduced earlier.

For demonstration purposes, we will disregard the `time` variable and consider these samples across time as replicates. Note that this is not recommended for an analysis of this or similar datasets.

```
ddsX <- ddsPara
design(ddsX) <- ~ patient + treatment + patient:treatment
```

We can then run the standard analysis and examine the names of the results columns. The model includes an effect for each patient, an effect for each treatment, and additionally interactions for each combination of treatment and patient.

```
ddsX <- DESeq(ddsX)
resultsNames(ddsX)
```

```
##  [1] "Intercept"                 "patient1"
##  [3] "patient2"                  "patient3"
##  [5] "patient4"                  "treatmentControl"
##  [7] "treatmentDPN"              "treatmentOHT"
##  [9] "patient1.treatmentControl" "patient2.treatmentControl"
## [11] "patient3.treatmentControl" "patient4.treatmentControl"
## [13] "patient1.treatmentDPN"     "patient2.treatmentDPN"
## [15] "patient3.treatmentDPN"     "patient4.treatmentDPN"
## [17] "patient1.treatmentOHT"     "patient2.treatmentOHT"
## [19] "patient3.treatmentOHT"     "patient4.treatmentOHT"
```

In order to test if the log2 fold change for OHT over the control sample is different for patient 4, one would use the following call to `results`:

```
resX <- results(ddsX, contrast =
                list("patient4.treatmentOHT",
                     "patient4.treatmentControl"))
head(resX, 2)
```

```
## log2 fold change (MAP): patient4.treatmentOHT vs patient4.treatmentControl
## Wald test p-value: patient4.treatmentOHT vs patient4.treatmentControl
## DataFrame with 2 rows and 6 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003   526.077      -0.139813    0.1287   -1.0861     0.277
## ENSG00000000005     0.674      -0.000821    0.0255   -0.0322     0.974
##                      padj
##                 <numeric>
## ENSG00000000003         1
## ENSG00000000005         1
```

Note that the log2 fold change for treatment of OHT over control for patient 4 is the interaction effect above in addition to the main effect of treatment OHT over control.

If the factors in the model have only two levels, the handling of interaction is slightly different to simplify the analysis. Here only a single interaction term is used in the model, and a test for the interaction effect is extracted using the `name` argument to `results`. Here we drop all but two levels of treatment and patient variables.

```
ddsXsub <- ddsX[,ddsX$treatment %in% c("Control","OHT") &
                ddsX$patient %in% c("1","2")]
```

```
ddsXsub$treatment <- droplevels(ddsXsub$treatment)
ddsXsub$patient <- droplevels(ddsXsub$patient)
```

We can then rerun DESeq, and note that instead of terms for each level of the factors (expanded model matrices), we have the standard comparisons over the base level, and a single interaction term which can be extracted by name.

```
ddsXsub <- DESeq(ddsXsub)
resultsNames(ddsXsub)
```

```
## [1] "Intercept"                "patient_2_vs_1"
## [3] "treatment_OHT_vs_Control" "patient2.treatmentOHT"
```

```
resXsub <- results(ddsXsub, name="patient2.treatmentOHT")
head(resXsub, 2)
```

```
## log2 fold change (MAP): patient2.treatmentOHT
## Wald test p-value: patient2.treatmentOHT
## DataFrame with 2 rows and 6 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003    639.72        -0.0174     0.179   -0.0968     0.923
## ENSG00000000005      1.31        -0.1129     0.103   -1.0984     0.272
##                      padj
##                 <numeric>
## ENSG00000000003         1
## ENSG00000000005         1
```

3.4 Time-series experiments

As with models containing interaction terms, 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 3.6, 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 term will test whether the condition induces a change in gene expression at any time point after the base-level time point (time 0).

Effects at individual time points can also be investigated using interactions, however we note that testing all combinations of time points and conditions is just one approach to exploring time course data. We also suggest that users consider applying transformations which stabilize the variance of the count data, as described in Section 2. This transformed data can then be used for exploratory data analysis, by selecting a subset of genes which has the highest variance and then using other packages to perform gene clustering.

3.5 Dealing with 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 (and `nbinomWaldTest`/`nbinomLRT` functions) 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.

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

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. The gene flagging functionality can be disabled by setting `cooksCutoff` to `FALSE` or `Inf`. The automatic outlier replacement performed by `DESeq` can be disabled by setting the `minReplicatesForReplace` argument to `Inf`.

`DESeq` replaces outliers by calling the `replaceOutliers` function, which has more arguments for controlling the replacement behavior. `DESeq` preserves the original counts in `counts(dds)` saving the replacement counts as `replaceCounts` in `assays(dds)`. The function `replaceOutliers` replaces the counts in `counts(dds)`, saving the original counts as `originalCounts` in `assays(dds)`.

Here we demonstrate the `replaceOutliers` function on the `pasilla` dataset, using the version of `dds` which was analyzed with only `condition` in the design formula. This is only for demonstration purposes, as we suggest 7 or more replicates in order to consider replacement for large Cook's distances. The single factor design is necessary for the demonstration, such that there are 3 or more replicates for every sample.

```
ddsReplace <- replaceOutliers(dds, minReplicates=3)
```

Finally we re-run all the steps of DESeq.

```
ddsReplace <- DESeq(ddsReplace)
tab <- table(initial = results(dds)$padj < .1,
             replace = results(ddsReplace)$padj < .1)
addmargins(tab)
```

```
##        replace
## initial FALSE TRUE  Sum
##   FALSE  6918    1 6919
##   TRUE      0  810  810
##   Sum    6918  811 7729
```

3.6 Likelihood ratio test

One reason to use the likelihood ratio test is in order to test the null hypothesis that log2 fold changes for multiple levels of a factor, or for multiple variables, such as all interactions between two variables, are equal to zero. The likelihood ratio test can also be specified using the `test` argument to DESeq, which substitutes `nbinomWaldTest` with `nbinomLRT`. In this case, the user provides the full formula (the formula stored in `design(dds)`), and a reduced formula, e.g. one which does not contain the variable of interest. The degrees of freedom for the test is obtained from the number of parameters in the two models. The Wald test and the likelihood ratio test share many of the same genes with adjusted p value < 0.1 for this experiment.

As we already have an object `dds` with dispersions calculated for the design formula `~ condition`, we only need to run the function `nbinomLRT`, with a reduced formula including only the intercept, in order to test the log2 fold change attributable to the condition.

```
ddsLRT <- nbinomLRT(dds, reduced = ~ 1)
resLRT <- results(ddsLRT)
head(resLRT,2)
```

```
## log2 fold change: condition treated vs untreated
## LRT p-value: '~ condition' vs '~ 1'
## DataFrame with 2 rows and 6 columns
##              baseMean log2FoldChange     lfcSE      stat    pvalue      padj
##             <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000003     0.159        15.0447   195.974     0.791     0.374        NA
## FBgn0000008    52.226         0.0281     0.298     0.010     0.920     0.971
```

```
tab <- table(Wald=res$padj < .1, LRT=resLRT$padj < .1)
addmargins(tab)
```

```
##        LRT
## Wald    FALSE TRUE  Sum
```

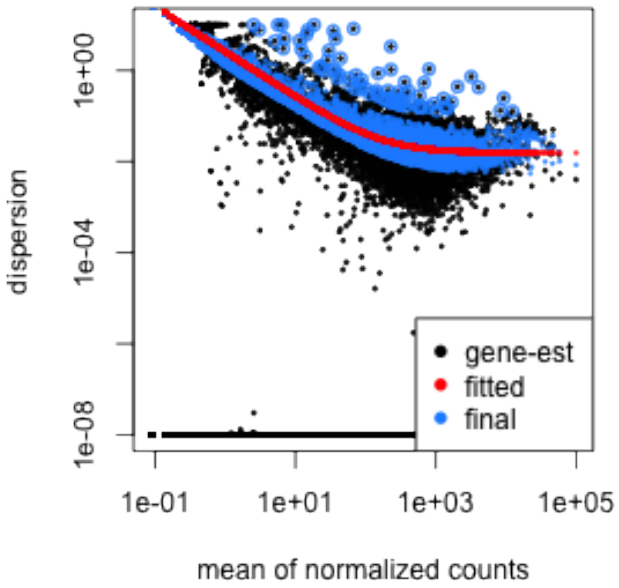

Figure 7: **Dispersion plot.** The dispersion estimate plot shows the gene-wise estimates (black), the fitted values (red), and the final maximum *a posteriori* estimates used in testing (blue).

```
##    FALSE  6894    0 6894
##    TRUE     15  794  809
##    Sum    6909  794 7703
```

If the full design formula had multiple factors, ~ `type + condition`, then the reduced formula would be ~ `type`, i.e., accounting for only the type of sequencing.

3.7 Dispersion plot and fitting alternatives

Plotting the dispersion estimates is a useful diagnostic. The dispersion plot in Figure 7 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 dispersion fit

The local dispersion fit is available in case the parametric fit fails to converge. A warning will be printed that one should use `plotDispEsts` to check the quality of the fit, whether the curve is pulled dramatically by a few outlier points.

```
ddsLocal <- estimateDispersions(dds, fitType="local")
```

3.7.2 Mean dispersion

While RNA-Seq data tend to demonstrate a dispersion-mean dependence, this assumption is not appropriate for all assays. An alternative is to use the mean of all gene-wise dispersion estimates.

```
ddsMean <- estimateDispersions(dds, fitType="mean")
```

3.7.3 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 first line of the code below, the function `estimateDispersionsGeneEst` stores the gene-wise estimates in the metadata column `dispGeneEst`. In the last line, the function `estimateDispersionsMAP`, uses this column and the column `dispFit` to generate maximum *a posteriori* (MAP) estimates of dispersion. The modeling assumption is that the true dispersions are distributed according to a log-normal prior around the fitted values in the column `fitDisp`. The width of this prior is calculated from the data.

```
ddsMed <- estimateDispersionsGeneEst(dds)
useForMedian <- mcols(ddsMed)$dispGeneEst > 1e-7
medianDisp <- median(mcols(ddsMed)$dispGeneEst[useForMedian],na.rm=TRUE)
mcols(ddsMed)$dispFit <- medianDisp
ddsMed <- estimateDispersionsMAP(ddsMed)
```

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 4.6. 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 attributes of the

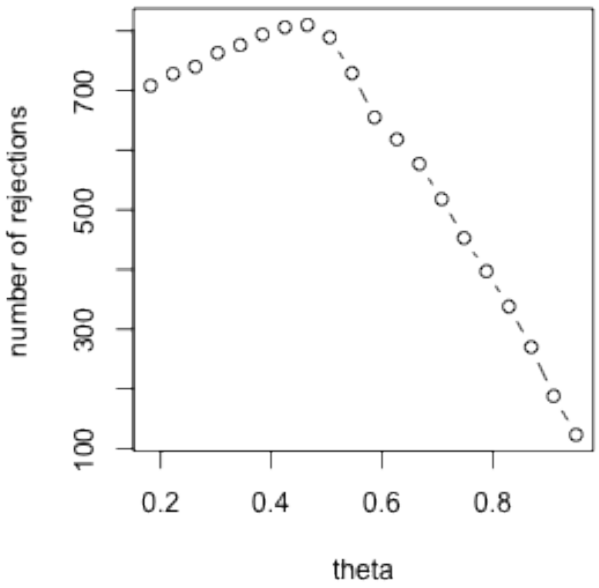

Figure 8: **Independent filtering.** The `results` function maximizes the number of rejections (adjusted p value less than a significance level), over theta, the quantiles of a filtering statistic (in this case, the mean of normalized counts).

object returned by `results`. For example, we can easily visualize the optimization by plotting the `filterNumRej` attribute of the results object, as seen in Figure 8.

```
attr(res,"filterThreshold")
```

```
## 46.5%
##   9.2
```

```
plot(attr(res,"filterNumRej"),type="b",
     ylab="number of rejections")
```

Independent filtering can be turned off by setting `independentFiltering` to `FALSE`. Alternative filtering statistics can be easily provided as an argument to the `results` function.

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

```
##          noFiltering
## filtering FALSE TRUE
##     FALSE  6919    0
##     TRUE    113  697
```

```
library(genefilter)
```

```
rv <- rowVars(counts(dds,normalized=TRUE))
resFiltByVar <- results(dds, filter=rv)
table(rowMean=(res$padj < .1), rowVar=(resFiltByVar$padj < .1))
```

```
##         rowVar
## rowMean FALSE TRUE
##   FALSE  6303   19
##   TRUE      0  807
```

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.

The four possible values of `altHypothesis` are demonstrated in the following code and visually by MA-plots in Figure 9. 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")
```

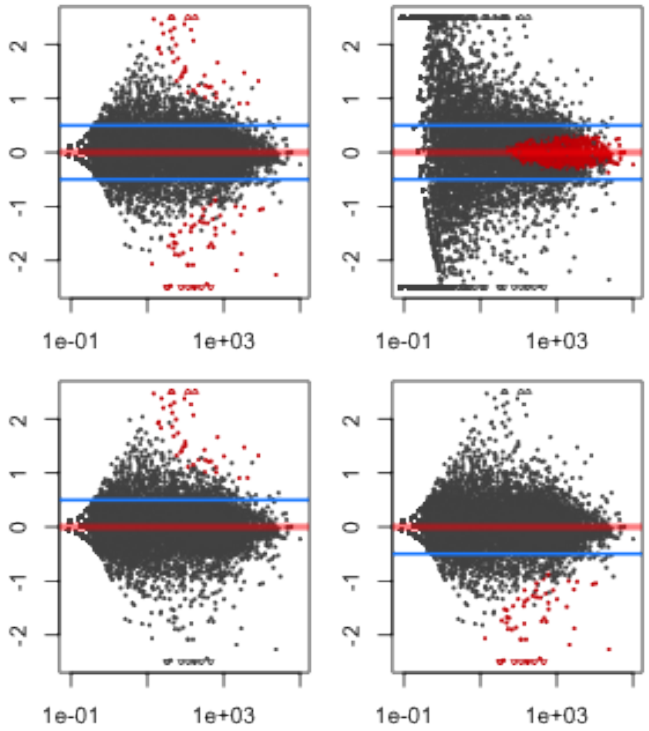

Figure 9: **MA-plots of tests of log2 fold change with respect to a threshold value.** Going left to right across rows, the tests are for `altHypothesis = "greaterAbs"`, `"lessAbs"`, `"greater"`, and `"less"`.

```
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
##              baseMean   baseVar   allZero dispGeneEst
##             <numeric> <numeric> <logical>   <numeric>
## FBgn0000003     0.159     0.178     FALSE    1.00e-08
## FBgn0000008    52.226   154.611     FALSE    3.67e-02
## FBgn0000014     0.390     0.444     FALSE    1.00e-08
## FBgn0000015     0.905     0.799     FALSE    1.00e-08
```

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

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

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

```
## DataFrame with 4 rows and 2 columns
##                     type                        description
##              <character>                        <character>
## baseMean    intermediate       the base mean over all rows
## baseVar     intermediate   the base variance over all rows
## allZero     intermediate      all counts in a row are zero
## dispGeneEst intermediate gene-wise estimates of dispersion
```

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))
```

```
## DataFrame with 6 rows and 3 columns
##             Intercept conditionuntreated conditiontreated
##             <numeric>          <numeric>        <numeric>
## FBgn0000003    -2.742            -0.0789           0.0789
## FBgn0000008     5.704            -0.0131           0.0131
## FBgn0000014    -1.358            -0.0328           0.0328
## FBgn0000015    -0.233             0.1032          -0.1032
## FBgn0000017    11.178             0.1361          -0.1361
## FBgn0000018     7.786             0.0584          -0.0584
```

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 a mean of 1, which can be accomplished by dividing out the mean of the matrix.

```
normFactors <- normFactors / mean(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)
```

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 pre-print [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[4] 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 priori* 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* [8, 9]. 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:

- *SummarizedExperiment* is used as the superclass for storage of input data, intermediate calculations and results.
- Maximum *a posteriori* estimation of GLM coefficients incorporating a zero-mean 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 spread in log2 fold changes for genes with low counts (e. g. single digits per condition).
- Maximum *a posteriori* estimation of dispersion replaces the `sharingMode` options `fit-only` or `maximum` of the previous version of the package [10].

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

- All estimation and inference is based on the generalized linear model, which includes the two condition case (previously the *exact test* was used).
- The Wald test for significance of GLM coefficients is provided as the default inference method, with the likelihood ratio test of the previous version still available.
- It is possible to provide a matrix of sample-/gene-dependent normalization factors.

4.3 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 [11]. For more on the implementation of Cook's distance see Section 3.5 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))
```

4.4 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.

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

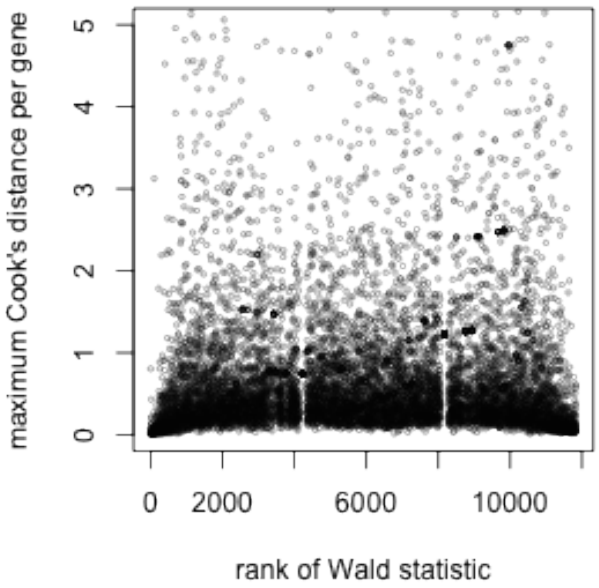

Figure 10: **Cook's distance.** Plot of the maximum Cook's distance per gene over the rank of the Wald statistics for the condition. The two regions with small Cook's distances are genes with a single count in one sample. The horizontal line is the default cutoff used for 7 samples and 3 estimated parameters.

4.5 Expanded model matrices

As mentioned in Section 3.2, *DESeq2* uses "expanded model matrices" with the log2 fold change prior, in order to produce log2 fold change estimates and test results which are independent of the choice of base level. These 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. Expanded model matrices are not used without the log2 fold change prior or in the case of designs with 2 level factors and an interaction term.

These matrices are therefore 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 base level independent. The `contrast` argument of the `results` function is again used in order to generate comparisons of interest.

4.6 Independent filtering and multiple testing

4.6.1 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 4.6.2. 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 [12] 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 11). 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))
```

4.6.2 Why does it work?

Consider the p value histogram in Figure 12. 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 > attr(res,"filterThreshold")
table(use)
```

```
## use
## FALSE  TRUE
##  6728  7742
```

```
h1 <- hist(res$pvalue[!use], breaks=0:50/50, plot=FALSE)
h2 <- hist(res$pvalue[use], breaks=0:50/50, plot=FALSE)
colori <- c(`do not pass`="khaki", `pass`="powderblue")
```

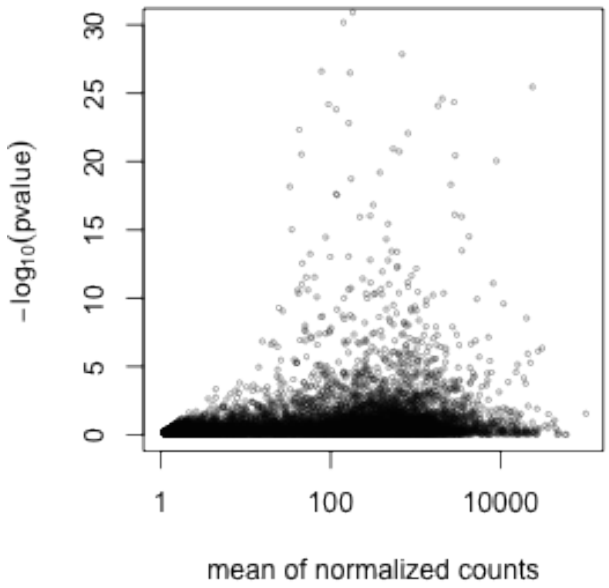

Figure 11: **Mean counts as a filter statistic.** The mean of normalized counts provides an independent statistic for filtering the tests. It is independent because the information about the variables in the design formula is not used. By filtering out genes which fall on the left side of the plot, the majority of the low p values are kept.

```
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)))
```

4.6.3 Diagnostic plots for multiple testing

The Benjamini-Hochberg multiple testing adjustment procedure [13] has a simple graphical illustration, which we produce in the following code chunk. Its result is shown in the left panel of Figure 13.

```
resFilt <- res[use & !is.na(res$pvalue),]
orderInPlot <- order(resFilt$pvalue)
showInPlot <- (resFilt$pvalue[orderInPlot] <= 0.08)
alpha <- 0.1
```

```
plot(seq(along=which(showInPlot)), resFilt$pvalue[orderInPlot][showInPlot],
     pch=".", xlab = expression(rank(p[i])), ylab=expression(p[i]))
```

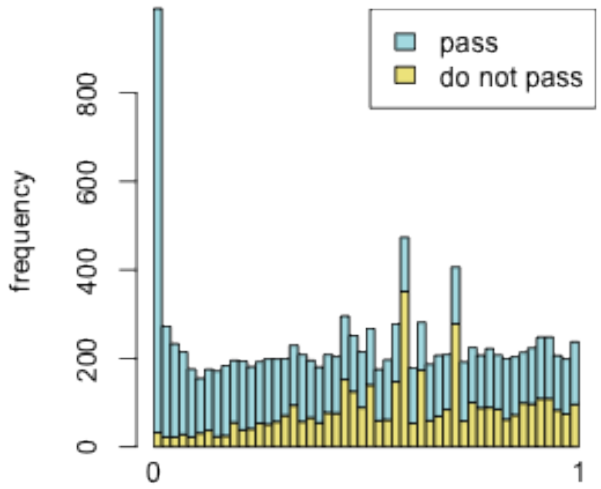

Figure 12: **Histogram of p values** for all tests (`res$pvalue`). The area shaded in blue indicates the subset of those that pass the filtering, the area in khaki those that do not pass.

```
abline(a=0, b=alpha/length(resFilt$pvalue), col="red3", lwd=2)
```

Schweder and Spjøtvoll [14] suggested a diagnostic plot of the observed p-values which permits estimation of the fraction of true null hypotheses. For a series of hypothesis tests $H_1, \ldots, H_m$ with p-values p_i, they suggested plotting

$$(1 - p_i, N(p_i)) \text{ for } i \in 1, \ldots, m, \tag{2}$$

where $N(p)$ is the number of p-values greater than p. An application of this diagnostic plot to `resFilt$pvalue` is shown in the right panel of Figure 13. When all null hypotheses are true, the p-values are each uniformly distributed in $[0, 1]$, Consequently, the cumulative distribution function of $(p_1, \ldots, p_m)$ is expected to be close to the line $F(t) = t$. By symmetry, the same applies to $(1 - p_1, \ldots, 1 - p_m)$. When (without loss of generality) the first m_0 null hypotheses are true and the other $m - m_0$ are false, the cumulative distribution function of $(1 - p_1, \ldots, 1 - p_{m_0})$ is again expected to be close to the line $F_0(t) = t$. The cumulative distribution function of $(1 - p_{m_0+1}, \ldots, 1 - p_m)$, on the other hand, is expected to be close to a function $F_1(t)$ which stays below F_0 but shows a steep increase towards 1 as t approaches 1. In practice, we do not know which of the null hypotheses are true, so we can only observe a mixture whose cumulative distribution function is expected to be close to

$$F(t) = \frac{m_0}{m} F_0(t) + \frac{m - m_0}{m} F_1(t). \tag{3}$$

Such a situation is shown in the right panel of Figure 13. If $F_1(t)/F_0(t)$ is small for small t, then the mixture fraction $\frac{m_0}{m}$ can be estimated by fitting a line to the left-hand portion of the plot, and then

noting its height on the right. Such a fit is shown by the red line in the right panel of Figure 13.

```
plot(1-resFilt$pvalue[orderInPlot],
     (length(resFilt$pvalue)-1):0, pch=".",
     xlab=expression(1-p[i]), ylab=expression(N(p[i])))
abline(a=0, slope, col="red3", lwd=2)
```

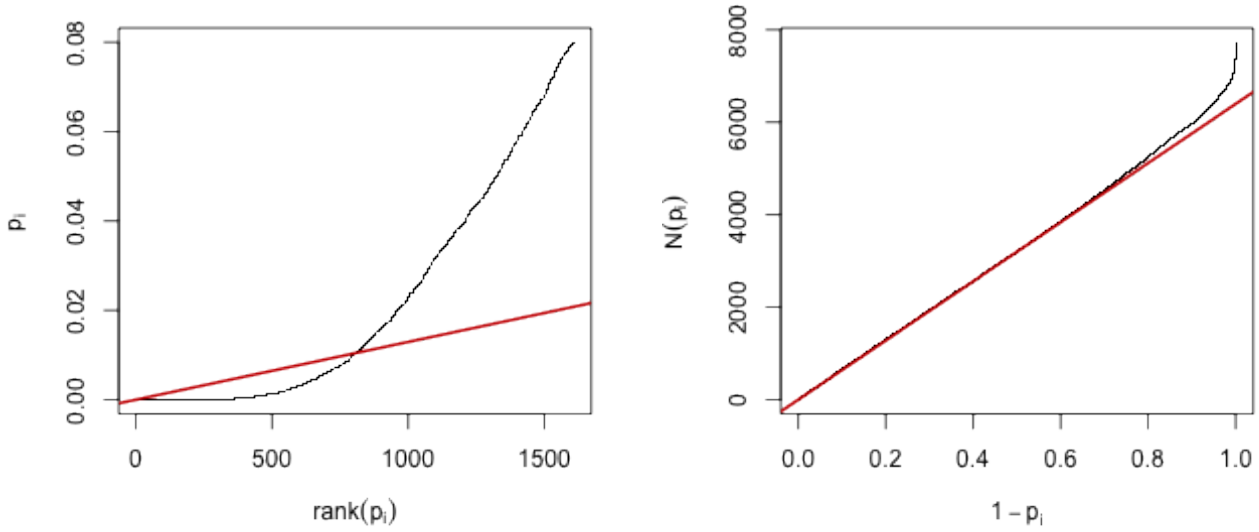

Figure 13: *Left:* illustration of the Benjamini-Hochberg multiple testing adjustment procedure [13]. The black line shows the p-values (y-axis) versus their rank (x-axis), starting with the smallest p-value from the left, then the second smallest, and so on. Only the first 1608 p-values are shown. The red line is a straight line with slope α/n, where $n = 7729$ is the number of tests, and $\alpha = 0.1$ is a target false discovery rate (FDR). FDR is controlled at the value α if the genes are selected that lie to the left of the rightmost intersection between the red and black lines: here, this results in 810 genes. *Right:* Schweder and Spjøtvoll plot, as described in the text. For both of these plots, the p-values `resFilt$pvalues` from Section 4.6.1 were used as a starting point. Analogously, one can produce these types of plots for any set of p-values, for instance those from the previous sections.

5 Frequently asked questions

5.1 How should I email a question?

We welcome emails with 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 emails and follow-up questions should take place over the Bioconductor list, which serves as a repository of information and helps saves the developers' time in responding to similar questions.

The subject line should contain "DESeq2" and a few words describing the problem.
- first search the Bioconductor list, http://bioconductor.org/help/mailing-list/, 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 1.4.2.

5.3 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 1.3.

5.4 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 1.5.

5.5 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`.

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

Continuous covariates can be easily 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`.

5.7 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.

6 Session Info

- R version 3.1.0 (2014-04-10), `x86_64-apple-darwin10.8.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, utils
- Other packages: AnnotationDbi 1.26.0, Biobase 2.24.0, BiocGenerics 0.10.0, biomaRt 2.20.0, Biostrings 2.32.0, BSgenome 1.32.0, DESeq2 1.4.5, genefilter 1.46.1, GenomeInfoDb 1.0.2, GenomicAlignments 1.0.1, GenomicFeatures 1.16.0, GenomicRanges 1.16.3, gplots 2.13.0, IRanges 1.22.6, knitr 1.5, parathyroidSE 1.2.0, pasilla 0.4.0, RColorBrewer 1.0-5, Rcpp 0.11.1, RcppArmadillo 0.4.300.0, Rsamtools 1.16.0, vsn 3.32.0, XVector 0.4.0
- Loaded via a namespace (and not attached): affy 1.42.2, affyio 1.32.0, annotate 1.42.0, BatchJobs 1.2, BBmisc 1.6, BiocInstaller 1.14.2, BiocParallel 0.6.0, BiocStyle 1.2.0, bitops 1.0-6, brew 1.0-6, caTools 1.17, codetools 0.2-8, DBI 0.2-7, DESeq 1.16.0, digest 0.6.4, evaluate 0.5.5, fail 1.2, foreach 1.4.2, formatR 0.10, gdata 2.13.3, geneplotter 1.42.0, grid 3.1.0, gtools 3.4.0, highr 0.3, iterators 1.0.7, KernSmooth 2.23-12, lattice 0.20-29, limma 3.20.1, locfit 1.5-9.1, plyr 1.8.1, preprocessCore 1.26.1, RCurl 1.95-4.1, RSQLite 0.11.4, rtracklayer 1.24.0, sendmailR 1.1-2, splines 3.1.0, stats4 3.1.0, stringr 0.6.2, survival 2.37-7, tools 3.1.0, XML 3.98-1.1, xtable 1.7-3, zlibbioc 1.10.0

References

[1] Wolfgang Huber Michael I Love and Simon Anders. Moderated estimation of fold change and dispersion for RNA-Seq data with DESeq2. *bioRxiv preprint*, 2014. URL: http://dx.doi.org/10.1101/002832.

[2] Felix Haglund, Ran Ma, Mikael Huss, Luqman Sulaiman, Ming Lu, Inga-Lena Nilsson, Anders Höög, Christofer C. Juhlin, Johan Hartman, and Catharina Larsson. Evidence of a Functional Estrogen Receptor in Parathyroid Adenomas. *Journal of Clinical Endocrinology & Metabolism*, September 2012. URL: http://dx.doi.org/10.1210/jc.2012-2484, doi:10.1210/jc.2012-2484.

[3] Paul Theodor Pyl Simon Anders and Wolfgang Huber. HTSeq - A Python framework to work with high-throughput sequencing data. *bioRxiv preprint*, 2014. URL: http://dx.doi.org/10.1101/002824.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society B*, 57:289–300, 1995.

[14] T. Schweder and E. Spjotvoll. Plots of P-values to evaluate many tests simultaneously. *Biometrika*, 69:493–502, 1982. doi:10.1093/biomet/69.3.493.