Differential expression of RNA-Seq data at the gene level – the DESeq package

Simon Anders[1], Wolfgang Huber

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany
[1]sanders@fs.tum.de

DESeq version 1.12.1 (Last revision 2013-02-24)

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 reads that have been assigned to a gene. Analogous analyses also arise for other assay types, such as comparative ChIP-Seq. The package *DESeq* provides methods to test for differential expression by use of the negative binonial distribution and a shrinkage estimator for the distribution's variance[1]. This vignette explains the use of the package. For an exposition of the statistical method, please see our paper [1] and the additional information in Section 9.

Contents

1 Input data and preparations **2**
1.1 The count table . . . 2
1.2 The metadata . . . 2
1.3 Normalisation . . . 4

2 Variance estimation **4**

3 Inference: Calling differential expression **6**
3.1 Standard comparison between two experimental conditions . . . 6
3.2 Working partially without replicates . . . 9
3.3 Working without any replicates . . . 9

4 Multi-factor designs **10**

5 Independent filtering and multiple testing **14**
5.1 Filtering by overall count . . . 14
5.2 Why does it work? . . . 15
5.3 How to choose the filter statistic and the cutoff? . . . 16
5.4 Diagnostic plots for multiple testing . . . 16

6 Variance stabilizing transformation **16**
6.1 Application to moderated fold change estimates . . . 17

7 Data quality assessment by sample clustering and visualisation **18**
7.1 Heatmap of the count table . . . 19
7.2 Heatmap of the sample-to-sample distances . . . 19
7.3 Principal component plot of the samples . . . 20
7.4 arrayQualityMetrics . . . 20

8 Further reading **23**

9 Changes since publication of the paper **23**

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

1 Input data and preparations

1.1 The count table

As input, the *DESeq* package expects count data as obtained, e. g., from RNA-Seq or another high-throughput sequencing experiment, in the form of a rectangular table of integer values. The table cell in the i-th row and the j-th column of the table tells how many reads have been mapped to gene i in sample j. Analogously, for other types of assays, the rows of the table 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 *DESeq*'s statistical model to hold, as only the actual counts allow assessing the measurement precision correctly. Hence, please do do not supply other quantities, such as (rounded) normalized counts, or counts of covered base pairs – this will only lead to nonsensical results.

Furthermore, it is important that each column stems from an independent biological replicate. For technical replicates (e. g. when the same library preparation was distributed over multiple lanes of the sequencer), please sum up their counts to get a single column, corresponding to a unique biological replicate. This is needed in order to allow *DESeq* to estimate variability in the experiment correctly.

To obtain such a count table for your own data, you will need to create it from your sequence alignments and suitable annotation. In Bioconductor, you can use the function `summarizeOverlaps` in the *GenomicRanges* package. See the vignette, reference [2], for a worked example. Another possibility is provided by the Bioconductor package *easyRNASeq*.

Another easy way to produce such a table from the output of the aligner is to use the `htseq-count` script distributed with the *HTSeq* Python package. Even though *HTSeq* is written in Python, you do not need to know any Python to use `htseq-count`. See `http://www-huber.embl.de/users/anders/HTSeq/doc/count.html`. HTSeq-count produces one count file for each sample. *DESeq* offers the function `newCountDataSetFromHTSeqCount`, described below, to get an analysis started from these files.

In this vignette, we will work with the gene level read counts from the *pasilla* data package. 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 how we produced the *pasilla* count table from second generation sequencing (Illumina) FASTQ files is provided in the vignette of the data package *pasilla*. The table is supplied by the *pasilla* package as a text file of tab-separated values. The function `system.file` tells us where this file is stored on your computer.

```
> datafile = system.file( "extdata/pasilla_gene_counts.tsv", package="pasilla" )
> datafile
```

```
[1] "/home/biocbuild/bbs-2.12-bioc/R/library/pasilla/extdata/pasilla_gene_counts.tsv"
```

Have a look at the file with a text editor to see how it is formatted. To read this file with R, we use the function `read.table`.

```
> pasillaCountTable = read.table( datafile, header=TRUE, row.names=1 )
> head( pasillaCountTable )
```

```
            untreated1 untreated2 untreated3 untreated4 treated1 treated2 treated3
FBgn0000003          0          0          0          0        0        0        1
FBgn0000008         92        161         76         70      140       88       70
FBgn0000014          5          1          0          0        4        0        0
FBgn0000015          0          2          1          2        1        0        0
FBgn0000017       4664       8714       3564       3150     6205     3072     3334
FBgn0000018        583        761        245        310      722      299      308
```

Here, `header=TRUE` indicates that the first line contains column names and `row.names=1` means that the first column should be used as row names. This leaves us with a *data.frame* containing integer count values.

1.2 The metadata

The best data are useless without metadata. *DESeq* uses familiar idioms in Bioconductor to manage the metadata that go with the count table. The metadata can be divided into three groups: information about the samples (table colums), about the features (table rows), and about the overall experiment.

First, we need a description of the samples, which we keep in a *data.frame* whose columns correspond to different types of information, and whose rows correspond to the 7 samples:

```
> pasillaDesign = data.frame(
+    row.names = colnames( pasillaCountTable ),
+    condition = c( "untreated", "untreated", "untreated",
+       "untreated", "treated", "treated", "treated" ),
+    libType = c( "single-end", "single-end", "paired-end",
+       "paired-end", "single-end", "paired-end", "paired-end" ) )
> pasillaDesign
```

```
           condition    libType
untreated1 untreated single-end
untreated2 untreated single-end
untreated3 untreated paired-end
untreated4 untreated paired-end
treated1     treated single-end
treated2     treated paired-end
treated3     treated paired-end
```

Here, simply use a chunk of R code to set up this define. More often, you will read this into R from a spreadsheet table, using the R functions `read.table` or `read.csv`; or perhaps even from a relational database table using R's database access facilities.

To analyse these samples, we will have to account for the fact that we have both single-end and paired-end method. To keep things simple at the start, we defer the discussion of this to Section 4 and first demonstrate a simple analysis by using only the paired-end samples.

```
> pairedSamples = pasillaDesign$libType == "paired-end"
> countTable = pasillaCountTable[ , pairedSamples ]
> condition = pasillaDesign$condition[ pairedSamples ]
```

Now, we have data input as follows.

```
> head(countTable)
```

```
            untreated3 untreated4 treated2 treated3
FBgn0000003          0          0        0        1
FBgn0000008         76         70       88       70
FBgn0000014          0          0        0        0
FBgn0000015          1          2        0        0
FBgn0000017       3564       3150     3072     3334
FBgn0000018        245        310      299      308
```

```
> condition
```

```
[1] untreated untreated treated   treated
Levels: treated untreated
```

For your own data, create such a factor simply with

```
> #not run
> condition = factor( c( "untreated", "untreated", "treated", "treated" ) )
```

We can now instantiate a *CountDataSet*, which is the central data structure in the *DESeq* package:

```
> library( "DESeq" )
> cds = newCountDataSet( countTable, condition )
```

If you have used *htseq-count* (see above) to obtain your read counts, you can use the funtion `newCountDataSetFromHTSeqCount` to obtain a CountDataSe directly from the count files produced by *htseq-count*. To this end, produce a data frame, similar to *pasillaDesign* constructed above, with the sample names in the first column, the file names in the scond column, and the condition in the third column (or, if you have more than one factor in your design matrix, all these factors in the third and folling columns), and pass this data frame to the function. See the help page ("`?newCountDataSetFromHTSeqCount`") for further details.

The *CountDataSet* class is derived from *Biobase*'s *eSet* class and so shares all features of this standard Bioconductor class. Furthermore, accessors are provided for its data slots[2]. For example, the counts can be accessed with the `counts` function, as we will see in the next section.

1.3 Normalisation

As a first processing step, we need to estimate the effective library size. This step is sometimes also called *normalisation*, even though there is no relation to normality or a normal distribution. The effective library size information is called the *size factors* vector, since the package only needs to know the relative library sizes. If the counts of non-differentially expressed genes in one sample are, on average, twice as high as in another (because the library was sequenced twice as deeply), the size factor for the first sample should be twice that of the other sample [1, 4]. The function `estimateSizeFactors` estimates the size factors from the count data. (See the man page of `estimateSizeFactorsForMatrix` for technical details on the calculation.)

```
> cds = estimateSizeFactors( cds )
> sizeFactors( cds )

untreated3 untreated4   treated2   treated3 
     0.873      1.011      1.022      1.115 
```

If we divide each column of the count table by the size factor for this column, the count values are brought to a common scale, making them comparable. When called with `normalized=TRUE`, the `counts` accessor function performs this calculation. This is useful, e.g., for visualization.

```
> head( counts( cds, normalized=TRUE ) )

            untreated3 untreated4 treated2 treated3
FBgn0000003       0.00       0.00      0.0    0.897
FBgn0000008      87.05      69.27     86.1   62.803
FBgn0000014       0.00       0.00      0.0    0.000
FBgn0000015       1.15       1.98      0.0    0.000
FBgn0000017    4082.02    3116.93   3004.5 2991.238
FBgn0000018     280.61     306.75    292.4  276.335
```

2 Variance estimation

The inference in *DESeq* relies on an estimation of the typical relationship between the data's variance and their mean, or, equivalently, between the data's dispersion and their mean.

The *dispersion* can be understood as the square of the coefficient of biological variation. So, if a gene's expression typically differs from replicate to replicate sample by 20%, this gene's dispersion is $0.2^2 = .04$. Note that the variance seen between counts is the sum of two components: the sample-to-sample variation just mentioned, and the uncertainty in measuring a concentration by counting reads. The latter, known as shot noise or Poisson noise, is the dominating noise source for lowly expressed genes. The former dominates for highly expressed genes. The sum of both, shot noise and dispersion, is considered in the differential expression inference.

Hence, the variance v of count values is modelled as

$$v = s\mu + \alpha s^2 \mu^2,$$

where μ is the expected normalized count value (estimated by the average normalized count value), s is the size factor for the sample under consideration, and α is the dispersion value for the gene under consideration.

To estimate the dispersions, use this command.

```
> cds = estimateDispersions( cds )
```

We could now proceed straight to the testing for differetial expression in Section 3. However, let us first spend a little more time with looking at the results of `estimateDispersions`, in order to better understand the subsequent inference. Also, verification of the dispersion plot Figure 1, explained below, is an aspect of data quality control: how well do the data accord to the expectations of our analytic approach? Quality assessment is a crucial step of the data analysis, and we will see further quality diagnostics in Section 7.

[2] In fact, the object `pasillaGenes` from the *pasilla* package is of class *CountDataSet*, and we could do the subsequent analysis on the basis of this object. Here we re-create `cds` from elementary data types, a *data.frame* and a *factor*, for pedagogic effect.

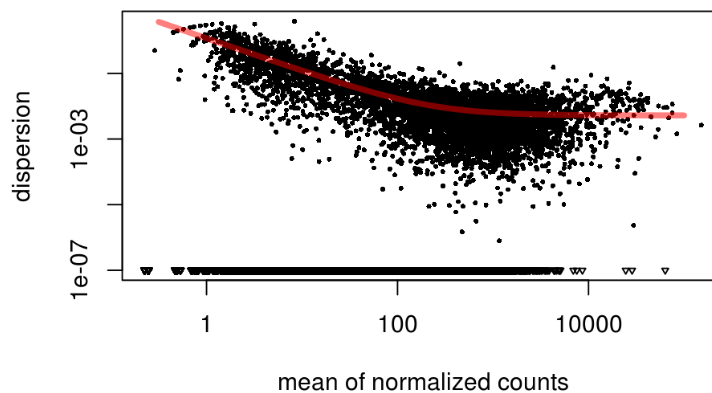

Figure 1: Empirical (black dots) and fitted (red lines) dispersion values plotted against the mean of the normalised counts.

The function `estimateDispersions` performs three steps. First, it estimates a dispersion value for each gene, then it fits a curve through the estimates. Finally, it assigns to each gene a dispersion value, using a choice between the per-gene estimate and the fitted value. To allow the user to inspect the intermediate steps, a `fitInfo` object is stored, which contains the per-gene estimate, the fitted curve and the values that will subsequently be used for inference.

```
> str( fitInfo(cds) )
```

```
List of 5
 $ perGeneDispEsts: num [1:14599] -0.4696 0.0237 NaN -0.9987 0.0211 ...
 $ dispFunc       :function (q)
  ..- attr(*, "coefficients")= Named num [1:2] 0.00524 1.16816
  .. ..- attr(*, "names")= chr [1:2] "asymptDisp" "extraPois"
  ..- attr(*, "fitType")= chr "parametric"
 $ fittedDispEsts : num [1:14599] 5.21332 0.02055 Inf 1.5008 0.00559 ...
 $ df             : int 2
 $ sharingMode    : chr "maximum"
```

It is useful to inspect the results of these steps visually. We can plot the per-gene estimates against the mean normalized counts per gene and overlay the fitted curve by using the function `plotDispEsts` (Fig. 1).

```
> plotDispEsts( cds )
```

As we estimated the dispersions from a small number of replicates, the estimates scatter with quite some sampling variance around their true values. An initial assumption that one could make is that the regression line shown in Figure 1 models the true underlying dispersions, and that the variation of the point estimates around simply reflects sampling variance. This is the assumption that we put forward in the first paper on *DESeq* [1]. However, subsequent experience with larger data sets indicates that not all of the variability of the points around the regression line seen in Figure 1 is sampling variance: some of it reflects differences of the true, underlying variance between different genes. Hence, the default behaviour of *DESeq* now uses a more prudent or conservative approach: if a per-gene estimates lies below the regression line, we assume that this might indeed be sampling variance, and shift the estimate upwards to the value predicted by the regression line. If, however, the per-gene estimate lies above the line, we do not shift it downwards to the line, but rather keep it as is.

The option `sharingMode` of the function `estimateDispersions` can be used to control this behaviour. The value `sharingMode="maximum"` corresponds to the default. If you have many replicates (where *many* starts at around 7), the choice `sharingMode="gene-est-only"` will typically be more adequate. If you would like to use the behaviour described in [1], this is achieved by specifying `sharingMode="fit-only"`.

Another difference of the current *DESeq* version to the original method described in the paper is the way how the mean-dispersion relation is fitted. By default, `estimateDispersions` now performs a parametric fit: Using a gamma-family GLM, two coefficients α_0, α_1 are found to parametrize the fit as $\alpha = \alpha_0 + \alpha_1/\mu$. (The values of the

two coefficients can be found in the `fitInfo` object, as attribute `coefficients` to `dispFunc`.) For some data sets, the parametric fit may give bad results, in which case one should try a local fit (the method described in the paper), which is available via the option `fitType="local"` to `estimateDispersions`.

In any case, the dispersion values which will be used by the subsequent testing are stored in the feature data slot of `cds`:

```
> head( fData(cds) )
```

```
            disp_pooled
FBgn0000003     5.21332
FBgn0000008     0.02368
FBgn0000014         Inf
FBgn0000015     1.50080
FBgn0000017     0.02110
FBgn0000018     0.00928
```

Fixme: Why is the value for FBgn00000014 Inf and not 0 (given that all counts are 0)? You can verify that these values are indeed the maxima from the two value vectors in `fitInfo(cds)`, which we saw on page 5. Advanced users who want to fiddle with the dispersion estimation can change the values in `fData(cds)` prior to calling the testing function.

3 Inference: Calling differential expression

3.1 Standard comparison between two experimental conditions

Having estimated the dispersion for each gene, it is straight-forward to look for differentially expressed genes. To contrast two conditions, e.g., to see whether there is differential expression between conditions "untreated" and "treated", we simply call the function `nbinomTest`. It performs the tests as described in [1] and returns a data frame with the p values and other useful information.

```
> res = nbinomTest( cds, "untreated", "treated" )
```

```
> head(res)
```

```
           id baseMean baseMeanA baseMeanB foldChange log2FoldChange  pval  padj
1 FBgn0000003    0.224      0.00     0.449        Inf            Inf 1.000 1.000
2 FBgn0000008   76.296     78.16    74.436      0.952        -0.0704 0.835 1.000
3 FBgn0000014    0.000      0.00     0.000        NaN            NaN    NA    NA
4 FBgn0000015    0.781      1.56     0.000      0.000           -Inf 0.416 1.000
5 FBgn0000017 3298.682   3599.47  2997.890      0.833        -0.2638 0.241 0.881
6 FBgn0000018  289.031    293.68   284.385      0.968        -0.0464 0.757 1.000
```

The interpretation of the columns of *data.frame* is as follows.

`id`	feature identifier
`baseMean`	mean normalised counts, averaged over all samples from both conditions
`baseMeanA`	mean normalised counts from condition A
`baseMeanB`	mean normalised counts from condition B
`foldChange`	fold change from condition A to B
`log2FoldChange`	the logarithm (to basis 2) of the fold change
`pval`	p value for the statistical significance of this change
`padj`	p value adjusted for multiple testing with the Benjamini-Hochberg procedure (see the R function `p.adjust`), which controls false discovery rate (FDR)

Let us first plot the $\log_2$ fold changes against the mean normalised counts, colouring in red those genes that are significant at 10% FDR (Figure 2).

```
> plotMA(res)
```

It is also instructive to look at the histogram of p values (Figure 3). The enrichment of low p values stems from the differentially expressed genes, while those not differentially expressed are spread uniformly over the range from zero to one (except for the p values from genes with very low counts, which take discrete values and so give rise to high counts for some bins at the right.)

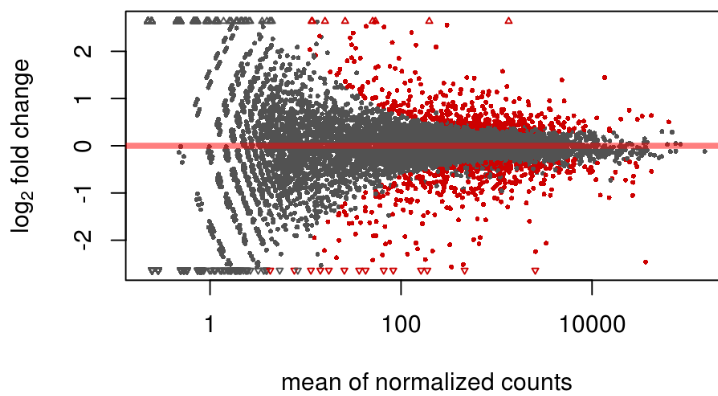

Figure 2: Plot of normalised mean versus log_2 fold change for the contrast *untreated* versus *treated*.

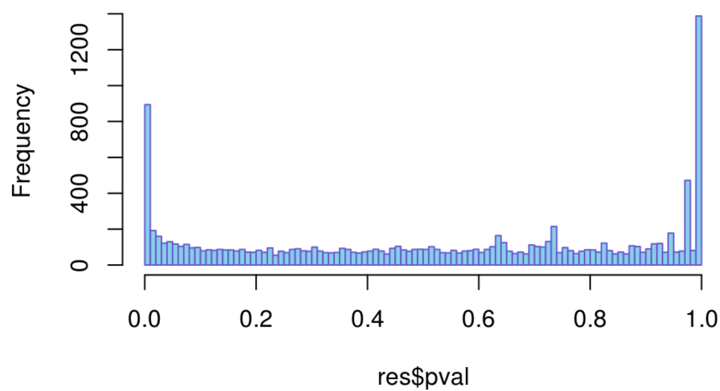

Figure 3: Histogram of p-values from the call to nbinomTest.

```
> hist(res$pval, breaks=100, col="skyblue", border="slateblue", main="")
```

We can filter for significant genes, according to some chosen threshold for the false dicovery rate (FDR),

```
> resSig = res[ res$padj < 0.1, ]
```

and list, e. g., the most significantly differentially expressed genes:

```
> head( resSig[ order(resSig$pval), ] )
```

```
               id baseMean baseMeanA baseMeanB foldChange log2FoldChange      pval      padj
9831  FBgn0039155      463       885      41.9     0.0474          -4.40 1.64e-124 1.89e-120
2366  FBgn0025111     1340       311    2369.3     7.6141           2.93 3.50e-107 2.01e-103
612   FBgn0003360     2544      4514     574.6     0.1273          -2.97  1.55e-99  5.95e-96
3192  FBgn0029167     2551      4211     891.7     0.2117          -2.24  4.35e-78  1.25e-74
10305 FBgn0039827      189       357      19.9     0.0556          -4.17  1.19e-65  2.74e-62
6948  FBgn0035085      447       761     133.3     0.1751          -2.51  3.15e-56  6.03e-53
```

We may also want to look at the most strongly down-regulated of the significant genes,

```
> head( resSig[ order( resSig$foldChange, -resSig$baseMean ), ] )
```

```
             id baseMean baseMeanA baseMeanB foldChange log2FoldChange      pval      padj
14078 FBgn0259236     4.27      8.54      0.00     0.0000           -Inf  1.31e-03  2.60e-02
13584 FBgn0085359    36.47     71.03      1.92     0.0270          -5.21  2.40e-09  2.19e-07
9831  FBgn0039155   463.44    884.96     41.91     0.0474          -4.40 1.64e-124 1.89e-120
2277  FBgn0024288    42.56     80.89      4.24     0.0524          -4.25  5.57e-20  1.94e-17
10305 FBgn0039827   188.59    357.33     19.86     0.0556          -4.17  1.19e-65  2.74e-62
6495  FBgn0034434    82.89    155.09     10.68     0.0689          -3.86  5.94e-32  5.25e-29
```

or at the most strongly up-regulated ones:

```
> head( resSig[ order( -resSig$foldChange, -resSig$baseMean ), ] )
```

```
             id baseMean baseMeanA baseMeanB foldChange log2FoldChange      pval      padj
6030  FBgn0033764     53.9     10.26      97.6       9.52           3.25  9.95e-18  2.93e-15
13079 FBgn0063667     11.8      2.55      21.0       8.23           3.04  1.43e-04  4.22e-03
7020  FBgn0035189    197.5     45.30     349.6       7.72           2.95  6.43e-15  1.35e-12
8499  FBgn0037290     50.5     11.66      89.2       7.65           2.94  1.04e-14  2.05e-12
2366  FBgn0025111   1340.2    311.17    2369.3       7.61           2.93 3.50e-107 2.01e-103
7264  FBgn0035539     16.0      4.19      27.8       6.63           2.73  6.76e-05  2.14e-03
```

To save the output to a file, use the R functions `write.table` and `write.csv`. (The latter is useful if you want to load the table in a spreadsheet program such as Excel.)

```
> write.csv( res, file="My Pasilla Analysis Result Table.csv" )
```

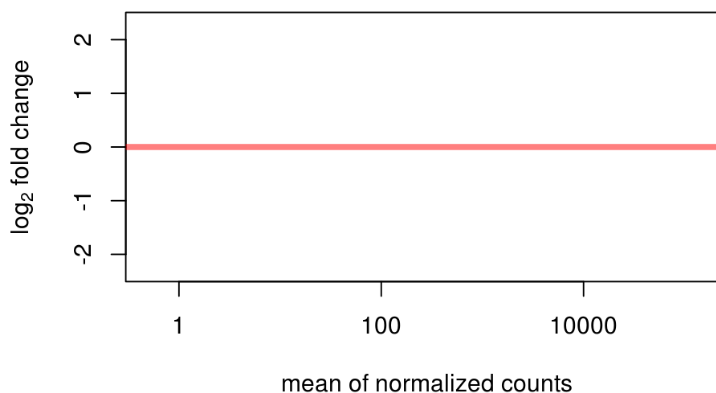

Figure 4: Plot of the log2 fold change between the untreated replicates versus average expression strength.

Note in Fig. 2 how the power to detect significant differential expression depends on the expression strength. For weakly expressed genes, stronger changes are required for the gene to be called significantly expressed. To understand the reason for this let, us compare the normalized counts between two replicate samples, here taking the two untreated samples as an example:

```
> ncu = counts( cds, normalized=TRUE )[ , conditions(cds)=="untreated" ]
```

`ncu` is now a matrix with two columns.

```
> plotMA(data.frame(baseMean = rowMeans(ncu),
+                   log2FoldChange = log2( ncu[,2] / ncu[,1] )),
+        col = "black")
```

As one can see in Figure 4, the log fold changes between replicates are stronger for lowly expressed genes than for highly expressed ones. We ought to conclude that a gene's expression is influenced by the treatment only if the change between treated and untreated samples is stronger than what we see between replicates, and hence, the dividing line between red and black in Figure 2 mimics the shape seen in Figure 4.

3.2 Working partially without replicates

If you have replicates for one condition but not for the other, you can still proceed as before. In such cases only the conditions with replicates will be used to estimate the dispersion. Of course, this is only valid if you have good reason to believe that the unreplicated condition does not have larger variation than the replicated one.

To demonstrate, we subset our data object to only three samples:

```
> cdsUUT = cds[ , 1:3]
> pData( cdsUUT )

           sizeFactor condition
untreated3      0.873 untreated
untreated4      1.011 untreated
treated2        1.022   treated
```

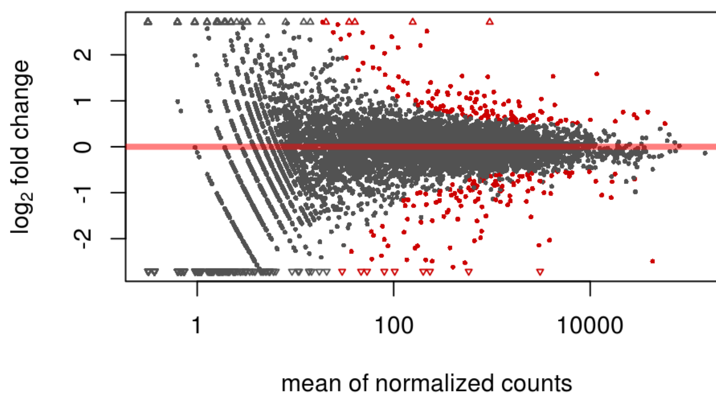

Figure 5: MvA plot for the contrast "treated" vs. "untreated", using two treated and only one untreated sample.

Now, we do the analysis as before.

```
> cdsUUT = estimateSizeFactors( cdsUUT )
> cdsUUT = estimateDispersions( cdsUUT )
> resUUT = nbinomTest( cdsUUT, "untreated", "treated" )
```

We produce the analogous plot as before, again with

```
> plotMA(resUUT)
```

Figure 5 shows the same symmetry in up- and down-regulation as in Fig. 2, but a certain asymmetry in the boundary line for significance. This has an easy explanation: low counts suffer from proportionally stronger shot noise than high counts, and since there is only one "untreated" sample versus two "treated" ones, a stronger downward fold-change is required to be called significant than for the upward direction.

3.3 Working without any replicates

Proper replicates are essential to interpret a biological experiment. After all, if one compares two conditions and finds a difference, how else can one know that this difference is due to the different conditions and would not have arisen between replicates, as well, just due to experimental or biological noise? Hence, any attempt to work without replicates will lead to conclusions of very limited reliability.

Nevertheless, such experiments are sometimes undertaken, and the *DESeq* package can deal with them, even though the soundness of the results may depend much on the circumstances.

Our primary assumption is now that the mean is a good predictor for the dispersion. Once we accept this assumption, we may argue as follows: Given two samples from different conditions and a number of genes with comparable expression levels, of which we expect only a minority to be influenced by the condition, we may take the

dispersion estimated from comparing their counts *across* conditions as ersatz for a proper estimate of the variance across replicates. After all, we assume most genes to behave the same within replicates as across conditions, and hence, the estimated variance should not be affected too much by the influence of the hopefully few differentially expressed genes. Furthermore, the differentially expressed genes will only cause the dispersion estimate to be too high, so that the test will err to the side of being too conservative.

We shall now see how well this works for our example data. We reduce our count data set to just two columns, one "untreated" and one "treated" sample:

```
> cds2 = cds[ ,c(  "untreated3", "treated3"   ) ]
```

Now, without any replicates at all, the `estimateDispersions` function will refuse to proceed unless we instruct it to ignore the condition labels and estimate the variance by treating all samples as if they were replicates of the same condition:

```
> cds2 = estimateDispersions( cds2, method="blind", sharingMode="fit-only" )
```

Note the option `sharingMode="fit-only"`. Remember that the default, `sharingMode="maximum"`, takes care of outliers, i.e., genes with dispersion much larger than the fitted values. Without replicates, we cannot catch such outliers and so have to disable this functionality.

Now, we can attempt to find differential expression:

```
> res2 = nbinomTest( cds2, "untreated", "treated" )
```

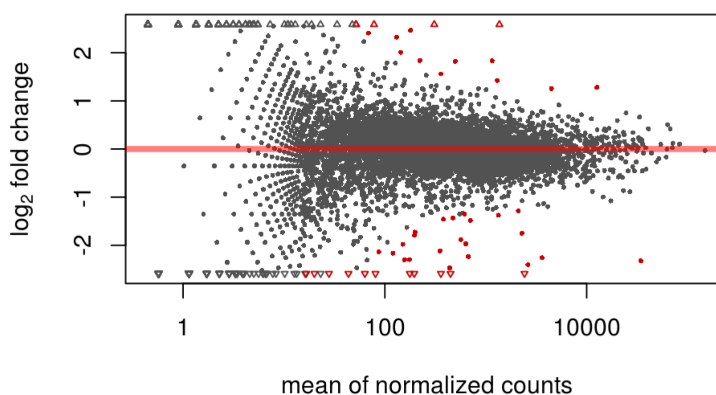

Figure 6: MvA plot, from a test using no replicates.

Unsurprisingly, we find much fewer hits, as can be seen from the plot (Fig. 6)

```
> plotMA(res2)
```

and from this table, tallying the number of significant hits in our previous and our new, restricted analysis:

```
> addmargins( table( res_sig = res$padj < .1, res2_sig = res2$padj < .1 ) )

       res2_sig
res_sig FALSE  TRUE   Sum
  FALSE 10084     1 10085
  TRUE    773    48   821
  Sum   10857    49 10906
```

4 Multi-factor designs

Let us return to the full pasilla data set. Remember that we started of with these data:

```
> head( pasillaCountTable )
```

```
           untreated1 untreated2 untreated3 untreated4 treated1 treated2 treated3
FBgn0000003          0          0          0          0        0        0        1
FBgn0000008         92        161         76         70      140       88       70
FBgn0000014          5          1          0          0        4        0        0
FBgn0000015          0          2          1          2        1        0        0
FBgn0000017       4664       8714       3564       3150     6205     3072     3334
FBgn0000018        583        761        245        310      722      299      308
```

```
> pasillaDesign
```

```
           condition    libType
untreated1 untreated single-end
untreated2 untreated single-end
untreated3 untreated paired-end
untreated4 untreated paired-end
treated1     treated single-end
treated2     treated paired-end
treated3     treated paired-end
```

When creating a count data set with multiple factors, just pass a data frame instead of the condition factor:

```
> cdsFull = newCountDataSet( pasillaCountTable, pasillaDesign )
```

As before, we estimate the size factors and then the dispersions. Here, the default method (`method=■pooled■` to `estimateDispersion`), means that, as before, one dispersion is computed for each gene, which is now an average over all cells (weighted by the number of samples for each cells), where the term *cell* denotes any of the four combinations of factor levels of the design. An alternative is the option `method=■pooled-CR■`, which uses a Cox-Reid-adjusted maximum likelihood estimator [5, 6]. The latter is slower but useful when the cell concept is not applicable (e.g. in paired designs).

```
> cdsFull = estimateSizeFactors( cdsFull )
> cdsFull = estimateDispersions( cdsFull )
```

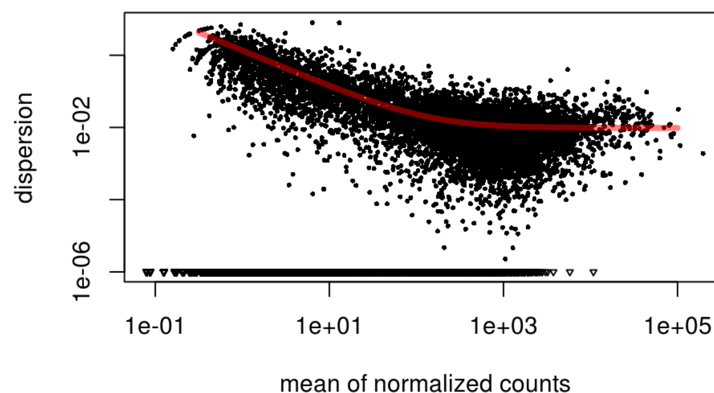

Figure 7: Estimated (black) pooled dispersion values for all seven samples, with regression curve (red).

We check the fit (Fig. 7):

```
> plotDispEsts( cdsFull )
```

For inference, we now specify two *models* by formulas. The *full model* regresses the genes' expression on both the library type and the treatment condition, the *reduced model* regresses them only on the library type. For each gene, we fit generalized linear models (GLMs) according to the two models, and then compare them in order to infer whether the additional specification of the treatment improves the fit and hence, whether the treatment has significant effect.

```
> fit1 = fitNbinomGLMs( cdsFull, count ~ libType + condition )
> fit0 = fitNbinomGLMs( cdsFull, count ~ libType  )
```

These commands take a while to execute. Also, they may produce a few warnings, informing you that the GLM fit failed to converge (and the results from these genes should be interpreted with care). The "fit" objects are data frames with three columns:

```
> str(fit1)

'data.frame':        14599 obs. of  5 variables:
 $ (Intercept)       : num  -0.722 6.654 -30.278 -1.591 11.978 ...
 $ libTypesingle-end : num  -31.082 -0.261 31.568 -0.462 -0.1 ...
 $ conditionuntreated: num  -30.8575 0.0405 -0.0306 2.1028 0.2562 ...
 $ deviance          : num  0.452 2.275 1.098 2.759 2.557 ...
 $ converged         : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
 - attr(*, "df.residual")= num 4
```

To perform the test, we call

```
> pvalsGLM = nbinomGLMTest( fit1, fit0 )
> padjGLM = p.adjust( pvalsGLM, method="BH" )
```

Fixme: Can we add a paragraph on what to do if we only want to make specific comparisons, e.g. let `fac` be a factor with three levels A, B and C, and we want to test pairwise for differences between A and B (without C), between A and C (without B) etc.

The function `nbinomTestGLM` returned simply a vector of p values which we handed to the standard R function `p.adjust` to adjust for multiple testing using the Benjamini-Hochberg (BH) method.

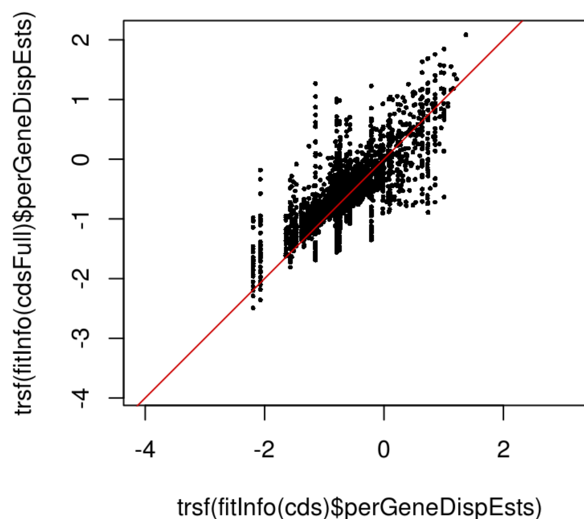

Figure 8: Comparison of per-gene estimates of the dispersion in the analysis using only the four paired-end samples (x-axis) and in the analysis using all seven samples (y-axis). For visualisation, the values are on a logarithm-like scale, defined by the function `trsf`, which is of the form $f(x) = \log\left((x + \sqrt{x^2+1})/2\right)$. For $x > 2$, the values of $f(x)$ and $\log(x)$ are approximately the same, whereas for values of x around 0, $f(x) \approx x$ is approximated by a straight line of slope 1.

Let's compare with the result from the four-samples test:

```
> tab1 = table( "paired-end only" = res$padj < .1, "all samples" = padjGLM < .1 )
> addmargins( tab1 )

               all samples
paired-end only FALSE  TRUE   Sum
```

```
           FALSE 10392    288 10680
           TRUE     143    678   821
           Sum    10535    966 11501
```

We see that the analyses find 678 genes in common, while 288 were only found in the analysis using all samples and 143 were specific for the *paired-end only* analysis.

In this case, the improvement in power that we gained from the additional samples is rather modest. This might indicate that the single-end samples (which are presumably older than the paired-end one) are of lower quality. This is supported by the observation that the dispersion estimates in the full data set tend to be larger than in the paired-end only one. So, despite the fact that with 7 samples we gain power from having more degrees of freedom in the estimation of the treatment effect, the additional samples also increase the dispersion (or equivalently, the variance), which loses power.

```
> table(sign(fitInfo(cds)$perGeneDispEsts - fitInfo(cdsFull)$perGeneDispEsts))

  -1    1
8175 3326
```

See also Figure 8, which is produced by

```
> trsf = function(x) log( (x + sqrt(x*x+1))/2 )
> plot( trsf(fitInfo(cds)$perGeneDispEsts),
+       trsf(fitInfo(cdsFull)$perGeneDispEsts), pch=16, cex=0.45, asp=1)
> abline(a=0, b=1, col="red3")
```

Continuing with the analysis, we can now extract the significant genes from the vector `padjGLM` as before. To see the corresponding fold changes, we have a closer look at the object `fit1`.

```
> head(fit1)

            (Intercept) libTypesingle-end conditionuntreated deviance converged
FBgn0000003      -0.722           -31.082           -30.8575    0.452      TRUE
FBgn0000008       6.654            -0.261             0.0405    2.275      TRUE
FBgn0000014     -30.278            31.568            -0.0306    1.098      TRUE
FBgn0000015      -1.591            -0.462             2.1028    2.759      TRUE
FBgn0000017      11.978            -0.100             0.2562    2.557      TRUE
FBgn0000018       8.568             0.230             0.0647    1.585      TRUE
```

The first three columns show the fitted coefficients, converted to a logarithm base 2 scale. The log2 fold change due to the condition is shown in the third column. As indicated by the column name, it is the effect of "untreated", i.e., the log ratio of "untreated" versus "treated". (This is unfortunately the other way round as before, due to the peculiarities of contrast coding.) Note that the library type also had noticeable influence on the expression, and hence would have increased the dispersion estimates (and so reduced power) if we had not fitted an effect for it.

The column *deviance* is the deviance of the fit. (Comparing the deviances with a χ^2 likelihood ratio test is how `nbinomGLMTest` calculates the p values.) The last column, *converged*, indicates whether the calculation of coefficients and deviance has fully converged. (If it is false too often, you can try to change the GLM control parameters, as explained in the help page to `fitNbinomGLMs`.) *Fixme: This is a bit vague, 'too often' should be specified and ideally the remedy be automated.*

Finally, we show that taking the library type into account was important to have good detection power by doing the analysis again using the standard workflow, as outlined earlier, and without informing *DESeq* of the library types:

```
> cdsFullB = newCountDataSet( pasillaCountTable, pasillaDesign$condition )
> cdsFullB = estimateSizeFactors( cdsFullB )
> cdsFullB = estimateDispersions( cdsFullB )
> resFullB = nbinomTest( cdsFullB, "untreated", "treated" )

> tab2 = table(
+    `all samples simple` = resFullB$padj < 0.1,
+    `all samples GLM`    = padjGLM < 0.1 )
> addmargins(tab2)

                   all samples GLM
all samples simple FALSE  TRUE   Sum
             FALSE 11387   389 11776
             TRUE      6   577   583
             Sum   11393   966 12359
```

We see that the two analyses find 577 genes in common, while in addition 389 genes were found by the analysis that took library type into account. The small number, namely 6, of genes that only made the FDR cutoff in the simple analysis is likely due to sampling variation.

5 Independent filtering and multiple testing

5.1 Filtering by overall count

The analyses of the previous sections involve the application of statistical tests, one by one, to each row of the data set, in order to identify those genes that have evidence for differential expression. The idea 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 test statistics of nulls and alternatives.

The benefit from filtering relies on property 2, and we will explore it further in Section 5.2. Its statistical validity relies on properties 1 and 3. We refer to [7] for further discussion on the mathematical and conceptual background.

```
> rs = rowSums ( counts ( cdsFull ))
> theta = 0.4
> use = (rs > quantile(rs, probs=theta))
> table(use)

use
FALSE  TRUE
 5861  8738

> cdsFilt = cdsFull[ use, ]
```

Above, we consider as a filter criterion `rs`, the overall sum of counts (irrespective of biological condition), and remove the genes in the lowest 40% quantile (as indicated by the parameter `theta`). We perform the testing as before in Section 4.

```
> fitFilt1  = fitNbinomGLMs( cdsFilt, count ~ libType + condition )
> fitFilt0  = fitNbinomGLMs( cdsFilt, count ~ libType  )
> pvalsFilt = nbinomGLMTest( fitFilt1, fitFilt0 )
> padjFilt  = p.adjust(pvalsFilt, method="BH" )
```

Let us compare the number of genes found at an FDR of 0.1 by this analysis with that from the previous one (`padjGLM`).

```
> padjFiltForComparison = rep(+Inf, length(padjGLM))
> padjFiltForComparison[use] = padjFilt
> tab3 = table( `no filtering`   = padjGLM < .1,
+              `with filtering` = padjFiltForComparison < .1 )
> addmargins(tab3)

            with filtering
no filtering FALSE  TRUE   Sum
       FALSE 11287   106 11393
       TRUE      3   963   966
       Sum   11290  1069 12359
```

The analysis with filtering found an additional 106 genes, an increase in the detection rate by about 11%, while 3 genes were only found by the previous analysis.

5.2 Why does it work?

First, consider Figure 9, which shows that among the 40–45% of genes with lowest overall counts, `rs`, there are essentially none that achieved an (unadjusted) p value less than 0.003 (this corresponds to about 2.5 on the $-\log_{10}$-scale).

```
> plot(rank(rs)/length(rs), -log10(pvalsGLM), pch=16, cex=0.45)
```

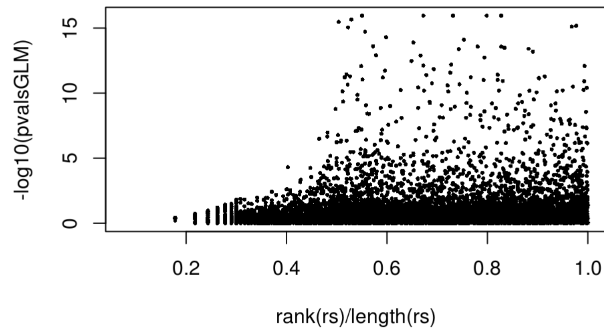

Figure 9: Scatterplot of rank of filter criterion (overall sum of counts `rs`) versus the negative logarithm of the test statistic `pvalsGLM`.

This means that by dropping the 40% genes with lowest `rs`, we do not loose anything substantial from our subsequent results. Second, consider the p value histogram in Figure 10. 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 $[0, 1]$.

```
> h1 = hist(pvalsGLM[!use], breaks=50, plot=FALSE)
> h2 = hist(pvalsGLM[use], breaks=50, plot=FALSE)
> 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)))
```

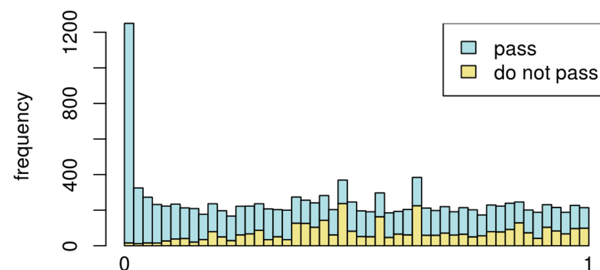

Figure 10: Histogram of p values for all tests (`pvalsGLM`). The area shaded in blue indicates the subset of those that pass the filtering, the area in khaki those that do not pass.

5.3 How to choose the filter statistic and the cutoff?

Please refer to the vignette *Diagnostic plots for independent filtering* in the *genefilter* package for a discussion on

- best choice of filter criterion and
- the choice of filter cutoff.

5.4 Diagnostic plots for multiple testing

The Benjamini-Hochberg multiple testing adjustment procedure [8] has a simple graphical illustration, which we produce in the following code chunk. Its result is shown in the left panel of Figure 11.

```
> orderInPlot = order(pvalsFilt)
> showInPlot = (pvalsFilt[orderInPlot] <= 0.08)
> alpha = 0.1

> plot(seq(along=which(showInPlot)), pvalsFilt[orderInPlot][showInPlot],
+      pch=".", xlab = expression(rank(p[i])), ylab=expression(p[i]))
> abline(a=0, b=alpha/length(pvalsFilt), col="red3", lwd=2)
```

Schweder and Spjøtvoll [9] 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

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

where $N(p)$ is the number of p-values greater than p. An application of this diagnostic plot to `pvalsFilt` is shown in the right panel of Figure 11. 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

$$F(t) = \frac{m_0}{m} F_0(t) + \frac{m - m_0}{m} F_1(t). \tag{2}$$

Such a situation is shown in the right panel of Figure 11. 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 11. Here we used a value for `slope` of 6865, the computation can be seen in the `.Rnw` source code of this vignette.

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

6 Variance stabilizing transformation

For some applications, it is useful to work with transformed versions of the count data. Maybe the most obvious choice is the logarithmic transformation. 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

$$y = \log_2(n + 1) \quad \text{or more generally,} \quad y = \log_2(n + n_0), \tag{3}$$

where n represents the count values and n_0 is a somehow chosen positive constant. In this section, we discuss a related, alternative approach that offers more theoretical justification and a rational way of choosing the parameter equivalent to n_0 above. It is based on error modeling and the concept of variance stabilizing transformations [1,10,11]. We estimate an overall mean-dispersion relationship of the data using `estimateDispersions` with the argument `method="blind"` and call the function `getVarianceStabilizedData`.

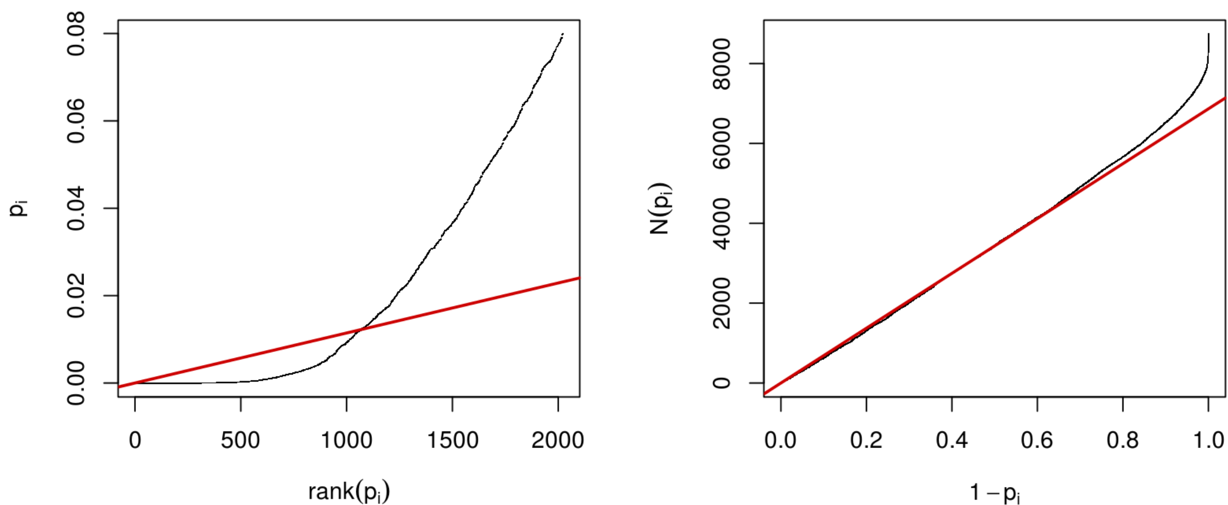

Figure 11: *Left:* illustration of the Benjamini-Hochberg multiple testing adjustment procedure [8]. The black line shows the p-values (y-axis) versus their rank (x-axis), starting with the smallest p-value from the left, then the second smallest, and so on. Only the first 2021 p-values are shown. The red line is a straight line with slope α/n, where $n = 8738$ 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 1069 genes. *Right:* Schweder and Spjøtvoll plot, as described in the text. For both of these plots, the p-values `pvalsFilt` from Section 5.1 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.

```
> cdsBlind = estimateDispersions( cds, method="blind" )
> vsd = varianceStabilizingTransformation( cdsBlind )
```

Here, we have used a parametric fit for the dispersion. In this case, the a closed-form expression for the variance stabilizing transformation is used by `getVarianceStabilizedData`, which is derived in the document `vst.pdf` (you find it in the *DESeq* package alongside this vignette). If a local fit is used (option `fittype="local"` to `estimateDispersion`) a numerical integration is used instead. The resulting transformation is shown in Figure 12. 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.

Figure 13 plots the standard deviation of the transformed data, across samples, against the mean, first using the shifted logarithm transformation (3), then using *DESeq*'s variance stabilising transformation. While for the latter, the standard deviation is roughly constant along the whole dynamic range, the shifted logarithm results in a highly elevated standard deviation in the lower count range.

```
> library("vsn")
> par(mfrow=c(1,2))
> notAllZero = (rowSums(counts(cds))>0)
> meanSdPlot(log2(counts(cds)[notAllZero, ] + 1), ylim = c(0,2.5))
> meanSdPlot(vsd[notAllZero, ], ylim = c(0,2.5))
```

6.1 Application to moderated fold change estimates

In the beginning of Section 3, we have seen in the *data.frame* `res` the (logarithm base 2) fold change estimate computed from the size-factor adjusted counts. When the involved counts are small, these (logarithmic) fold-change estimates can be highly variable, and can even be infinite. For some purposes, such as the clustering of samples or genes according to their expression profiles, or for visualisation of the data, this high variability from ratios between low counts tends to drown informative, systematic signal in other parts of the data. The variance stabilizing transformation offers one way to *moderate* the fold change estimates, so that they are more amenable to plotting or clustering.

```
> mod_lfc = (rowMeans( exprs(vsd)[, conditions(cds)=="treated", drop=FALSE] ) -
+            rowMeans( exprs(vsd)[, conditions(cds)=="untreated", drop=FALSE] ))
```

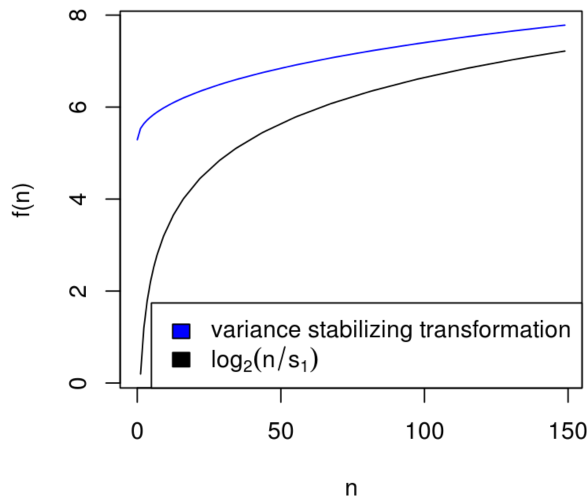

Figure 12: Graphs of the variance stabilizing transformation for sample 1, in blue, and of the transformation $f(n) = log_2(n/s_1)$, in black. n are the counts and s_1 is the size factor for the first sample.

Let us compare these to the original ($\log_2$) fold changes. First we find that many of the latter are infinite (resulting from division of a finite value by 0) or *not a number* (NaN, resulting from division of 0 by 0).

```
> lfc = res$log2FoldChange
> table(lfc[!is.finite(lfc)], useNA="always")

-Inf  Inf  NaN <NA>
 568  608 3098    0
```

For plotting, let us bin genes by their $\log_{10}$ mean to colour them according to expression strength,

```
> logdecade = 1 + round( log10( 1+rowMeans(counts(cdsBlind, normalized=TRUE)) ) )
> lfccol = colorRampPalette( c( "gray", "blue" ) )(6)[logdecade]
```

and then compare their ordinary log-ratios (`lfc`) against the *moderated* log-ratios (`mod_lfc`) in a scatterplot.

```
> ymax = 4.5
> plot( pmax(-ymax, pmin(ymax, lfc)), mod_lfc,
+       xlab = "ordinary log-ratio", ylab = "moderated log-ratio",
+       cex=0.45, asp=1, col = lfccol,
+       pch = ifelse(lfc<(-ymax), 60, ifelse(lfc>ymax, 62, 16)))
> abline( a=0, b=1, col="red3")
```

The result is shown in Figure 14.

7 Data quality assessment by sample clustering and visualisation

Data quality assessment and quality control (i. e. the removal of insufficiently good data) are essential steps of any data analysis. Even though we present these steps towards the end of this vignette, they 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.

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

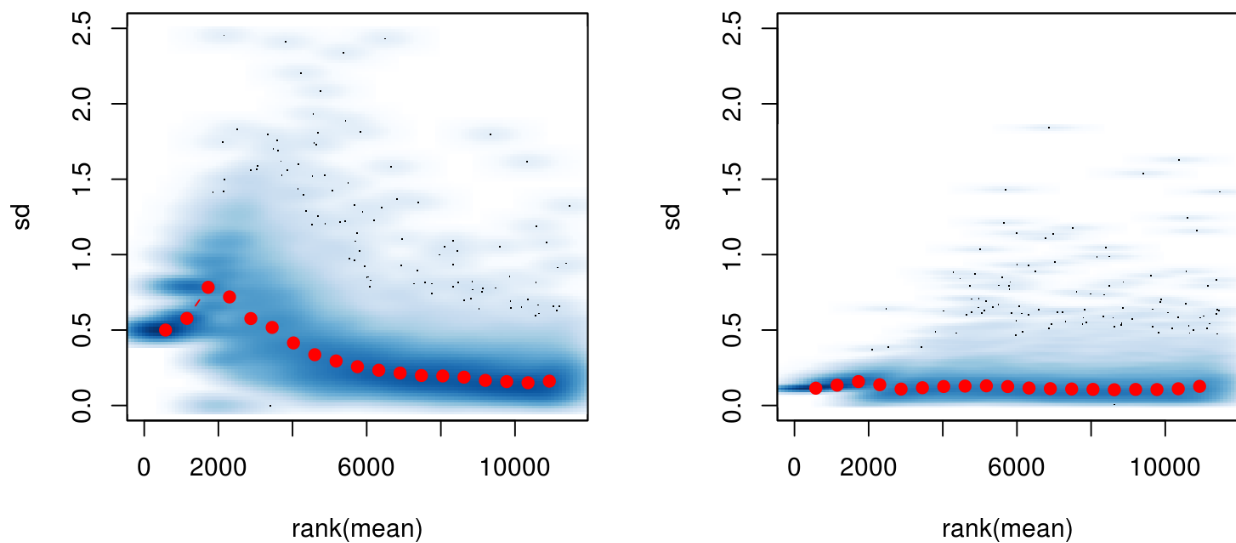

Figure 13: Per-gene standard deviation (taken across samples), against the rank of the mean, for the shifted logarithm $\log_2(n+1)$ (left) and *DESeq*'s variance stabilising transformation (right).

7.1 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 variance stabilisation transformed data for all 7 samples.

```
> cdsFullBlind = estimateDispersions( cdsFull, method = "blind" )
> vsdFull = varianceStabilizingTransformation( cdsFullBlind )
```

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

```
> heatmap.2(exprs(vsdFull)[select,], col = hmcol, trace="none", margin=c(10, 6))
```

The left panel of Figure 15 shows the resulting heatmap for the 30 most highly expressed genes. For comparison, let us also do the same with the untransformed counts.

```
> heatmap.2(counts(cdsFull)[select,], col = hmcol, trace="none", margin=c(10,6))
```

7.2 Heatmap of the sample-to-sample distances

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

```
> dists = dist( t( exprs(vsdFull) ) )
```

A heatmap of this distance matrix gives us an overview over similarities and dissimilarities between samples (Figure 16):

```
> mat = as.matrix( dists )
> rownames(mat) = colnames(mat) = with(pData(cdsFullBlind), paste(condition, libType, sep=" : "))
> heatmap.2(mat, trace="none", col = rev(hmcol), margin=c(13, 13))
```

The clustering correctly reflects our experimental design, i.e., samples are more similar when they have the same treatment or the same library type. (To avoid potential circularities in this conclusion, it was important to re-estimate the dispersions with `method="blind"` in the calculation for `cdsFullBlind` above, as only then, the variance stabilizing transformation is not informed about the design, and we can be sure that it is not biased towards a result supporting the design.)

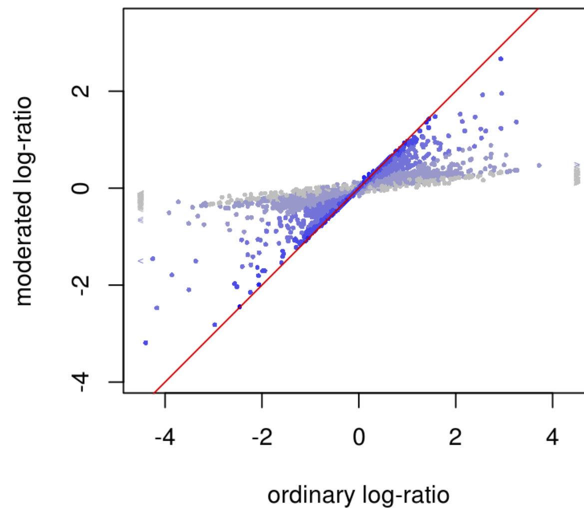

Figure 14: Scatterplot of ordinary (`lfc`) versus *moderated* log-ratios (`mod_lfc`). The points are coloured in a scale from grey to blue, representing weakly to strongly expressed genes. For the highly expressed genes (blue), `lfc` and `mod_lfc` agree. Differences arise when the involved counts are small (grey points): in this case, the moderated log-ratio is shrunk towards 0, compared to the ordinary log-ratio. The $>$ and $<$ symbols correspond to values of `lfc` whose absolute value was larger than 4.5 (including those that were infinite). All values of `mod_lfc` vary in a finite range.

7.3 Principal component plot of the samples

Related to the distance matrix of Section 7.2 is the PCA plot of the samples, which we obtain as follows (Figure 17).

```
> print(plotPCA(vsdFull, intgroup=c("condition", "libType")))
```

7.4 arrayQualityMetrics

Heatmaps and PCA plots similar to Figures 16 and 17 as well as some further diagnostics are also provided by the package *arrayQualityMetrics*. The package may be particularly useful for larger data set with dozens of samples. You can call its main function, which is also called `arrayQualityMetrics`, with an *ExpressionSet* such as `vsdFull`, which you will have produced from your *CountDataSet* with calls to `estimateDispersions` and *varianceStabilizingTransformation* as in the beginning of Section 7.1.

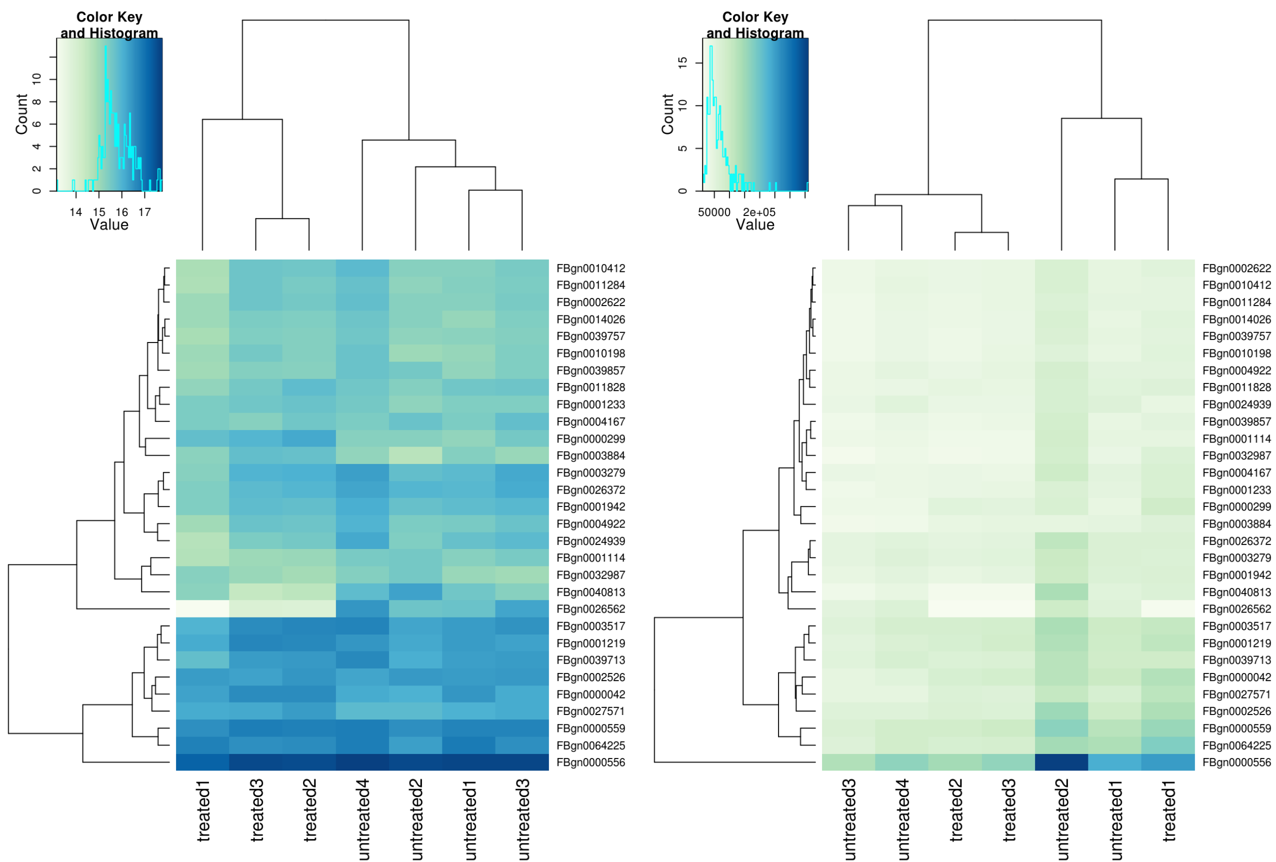

Figure 15: Heatmaps showing the expression data of the 30 most highly expressed genes. Left, the variance stabilisation transformed data (`vsdFull`) are shown, right, the original count data (`cdsFull`). In the left panel, the sample clustering aligns with the experimental factor (*treated* / *untreated*). The clustering and the colour scale in the right plot is dominated by a small number of data points with large values.

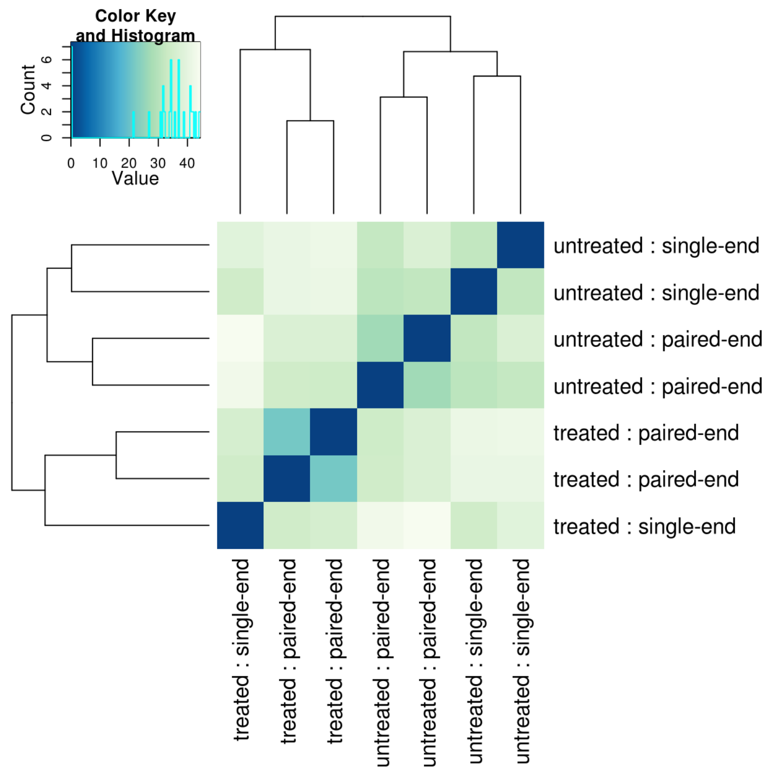

Figure 16: Heatmap showing the Euclidean distances between the samples as calculated from the variance stabilising transformation of the count data.

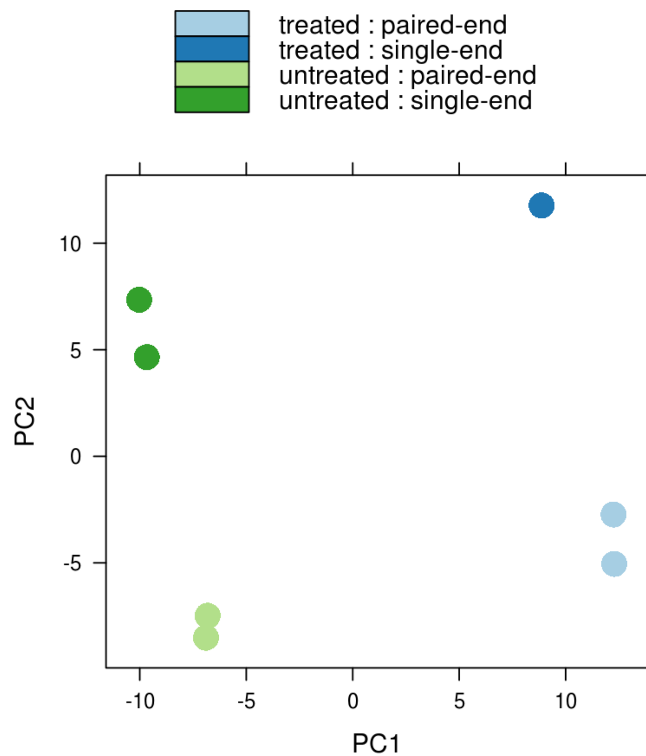

Figure 17: PCA plot. The 7 samples shown in the 2D plane spanned by their first two principal components. This type of plot is useful for visualizing the overall effect of experimental covariates and batch effects. For this data set, no batch effects besides the known effects of *condition* and *libType* are discernible.

8 Further reading

For more information on the statistical method, see our paper [1]. For more information on how to customize the *DESeq* work flow, see the package help pages, especially the help page for `estimateDispersions`.

9 Changes since publication of the paper

Since our paper on *DESeq* was published in Genome Biology in Oct 2010, we have made a number of changes to algorithm and implementation, which are listed here.

- `nbinomTest` calculates a p value by summing up the probabilities of all per-group count sums a and b that sum up to the observed count sum k_{iS} and are more extreme than the observed count sums k_{iA} and k_{iB}. Equation (11) of the paper defined *more extreme* as those pairs of values (a, b) that had smaller probability than the observed pair. This caused problems in cases where the dispersion exceeded 1. Hence, we now sum instead the probabilities of all values pairs that are *further out* in the sense that they cause a more extreme fold change $(a/s_A)/(b/s_B)$, where s_A and s_B are the sums of the size factors of the samples in conditions A and B, respectively. We do this in a one-tailed manner and double the result. Furthermore, we no longer approximate the sum, but always calculate it exactly.
- We added the possibility to fit GLMs of the negative binomial family with log link. This new functionality is described in this vignette. p values are calculated by a χ^2 likelihood ratio test. The logarithms of the size factors are provided as offsets to the GLM fitting function.
- The option `sharingMode='maximum'` was added to `estimateDispersion` and made default. This change makes DESeq robust against variance outliers and was not yet discussed in the paper.
- By default, DESeq now estimates one pooled dispersion estimate across all (replicated) conditions. In the original version, we estimated a separate dispersion-mean relation for each condition. The "maximum" sharing mode achieves its goal of making DESeq robust against outliers only with pooled dispersion estimate, and hence, this is now the default. The option `method='per-condition'` to `estimateDispersions` allows the user to go back to the old method.
- In the paper, the mean-dispersion relation is fitted by local regression. Now, DESeq also offers a parametric regression, as described in this vignette. The option `fitType` to `estimateDispersions` allows the user to choose between these. If a parametric regression is used, the variance stabilizing transformation is calculated using the closed-form expression given in the vignette supplement file `vst.pdf`.
- Finally, instead of the term *raw squared coefficient of variance* used in the paper we now prefer the more standard term *dispersion*.

10 Session Info

```
> sessionInfo()

R version 3.0.1 (2013-05-16)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=C               LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=C                 LC_NAME=C                  LC_ADDRESS=C
[10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] grid      parallel  stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] gplots_2.11.3      MASS_7.3-28        KernSmooth_2.23-10 caTools_1.14       gdata_2.13.2
 [6] gtools_3.0.0       RColorBrewer_1.0-5 vsn_3.28.0         DESeq_1.12.1       lattice_0.20-23
[11] locfit_1.5-9.1     Biobase_2.20.1     BiocGenerics_0.6.0
```

```
loaded via a namespace (and not attached):
 [1] AnnotationDbi_1.22.6  BiocInstaller_1.10.3  DBI_0.2-7             IRanges_1.18.3
 [5] RSQLite_0.11.4        XML_3.98-1.1          affy_1.38.1           affyio_1.28.0
 [9] annotate_1.38.0       bitops_1.0-6          genefilter_1.42.0     geneplotter_1.38.0
[13] limma_3.16.7          preprocessCore_1.22.0 splines_3.0.1         stats4_3.0.1
[17] survival_2.37-4       tools_3.0.1           xtable_1.7-1          zlibbioc_1.6.0
```

References

[1] Simon Anders and Wolfgang Huber. Differential expression analysis for sequence count data. *Genome Biology*, 11:R106, 2010.

[2] Valerie Obenchain. Counting with `summarizeOverlaps`. Vignette, distributed as part of the Bioconductor package *GenomicRanges*, as file *summarizeOverlaps.pdf*, 2011.

[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.

[4] Simon Anders, Alejandro Reyes, and Wolfgang Huber. Detecting differential usage of exons from RNA-seq data. *Genome Research*, 2012.

[5] 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.

[6] 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.

[7] Richard Bourgon, Robert Gentleman, and Wolfgang Huber. Independent filtering increases detection power for high-throughput experiments. *PNAS*, 107(21):9546–9551, 2010.

[8] 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.

[9] T. Schweder and E. Spjotvoll. Plots of P-values to evaluate many tests simultaneously. *Biometrika*, 69:493–502, 1982.

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

[11] 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.