Analysing RNA-Seq data with the DESeq package

Simon Anders

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany

sanders@fs.tum.de

2012-03-16

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 a method 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].

Contents

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

1 Input data and preparations

The *DESeq* package expects count data, as obtained, e.g., from an RNA-Seq or other high-throughput sequencing (HTS) experiment, in the form of a matrix of integer values. Each column corresponds to a sample, e.g., one library preparation or one lane. The rows correspond to the entities for which you want to compare coverage, e. g. to a gene, to a binding region in a ChIP-Seq dataset, a window in CNV-Seq or the like. So, for a typical RNA-Seq experiment, each element in the table tells how many reads have been mapped in a given sample to a given gene.

To obtain such a count table for your own data, you will need to create it from your sequence alignments and suitable annotation. Within Bioconductor, you can use the function `summarizeOverlaps` in the *GenomicRanges* package. See the vignette, Ref. [2], for a worked example. Another possibility (outside of Bioconductor) is the *htseq-count* script distributed with the HT-Seq Python framework [3]. (You do not need to know any Python to use `htseq-count`.) A third possibility might be given by the Bioconductor package *easyrnaseq* (by Nicolas Delhomme; in preparation, available soon; package name may change).

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* package. Even though *HTSeq* is a Python package, you do not need to know any Python to use `htseq-count`. See `http://www-huber.embl.de/users/anders/HTSeq/doc/count.html`. (If you use htseq-count, be sure to remove the extra lines with general counters ("ambiguous" etc.) when importing the data.)

The count values must be raw counts of sequencing reads. This is important for *DESeq*'s statistical model to hold, as only raw reads allow to assess the measurement precision correctly. (Hence, do not supply rounded values of normalized counts, or counts of covered base pairs.)

Furthermore, it is important that each column stems from an independent biological replicate. For purely technical replicates (e. g. when the same library preparation was distributed over multiple lanes of the sequencer in order to increase coverage), 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.

As an example dataset, we use the gene level read counts from the *pasilla* data package. This dataset is from an experiment on *Drosophila melanogaster* cell cultures and investigated the effect of RNAi knock-down of the splicing factor *pasilla* [4].

A table of gene-level count data derived from this dataset is supplied with the package *pasilla* as a text file of tab-separated values. The function `system.file` tells us where this file is stored in the local installation.

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

```
[1] "/Library/Frameworks/R.framework/Versions/2.15/Resources/library/pasilla/extdata/pasilla_gene
```

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
FBgn0000003          0          0          0          0        0        0
FBgn0000008         92        161         76         70      140       88
FBgn0000014          5          1          0          0        4        0
FBgn0000015          0          2          1          2        1        0
FBgn0000017       4664       8714       3564       3150     6205     3072
FBgn0000018        583        761        245        310      722      299
            treated3
FBgn0000003        1
FBgn0000008       70
FBgn0000014        0
FBgn0000015        0
FBgn0000017     3334
FBgn0000018      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 consisting only of integer count values.

We also need a description of the 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
```

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

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

```
> conds
```

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

For your own data, create such a factor simply with

```
> #not run
> conds <- 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, conds )
```

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.

```
> head( counts(cds) )
```

```
            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
```

As first processing step, we need to estimate the effective library size. This information is called the "size factors" vector, as the package only needs to know the relative library sizes. So, if the counts of non-differentially expressed genes in one sample are, on average, twice as high as in another, the size factor for the first sample should be twice as large as the one for the other sample. 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.

[2]In fact, the objects `pasillaGenes` and `cds` from the *pasilla* are also of class *CountDataSet*; here we re-created `cds` from elementary data types, a matrix and a factor, for pedagogic effect.

```
> 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 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, it is prudent to check the dispersion estimates and to make sure that the data quality is as expected.

The function `estimateDispersions` performs three steps. First, it estimates a dispersion value for each gene, then, it fits, for each condition, a curve through the estimates. Finally, it assigns to each gene a dispersion value, using either the estimated or the fitted value. To allow the user to inspect the intermediate steps, a "fit info" object is stored, which contains the empirical dispersion values for each gene, the curve fitted through the dispersions, and the fitted values that will be used in the test.

```
> 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"
```

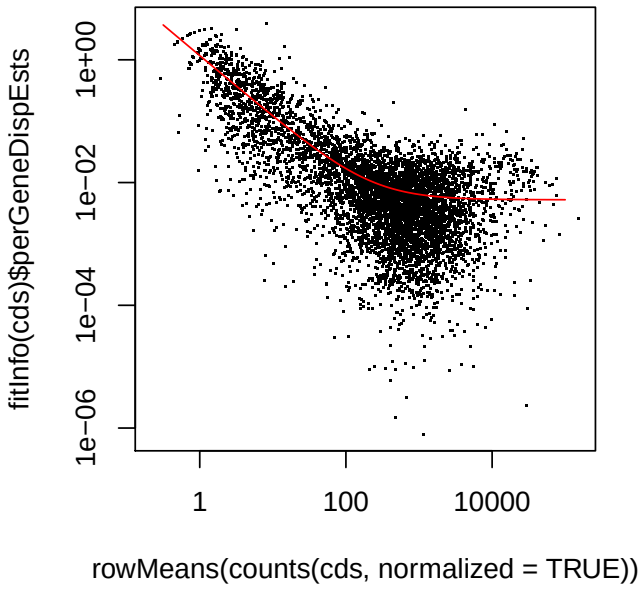

Figure 1: Empirical (black dots) and fitted (red lines) dispersion values plotted against mean expression strength.

```
 $ fittedDispEsts : num [1:14599] 5.21332 0.02055 Inf 1.5008 0.00559 ...
 $ df             : int 2
 $ sharingMode    : chr "maximum"
```

To visualize these, we plot the per-gene estimates against the normalized mean expressions per gene, and then overlay the fitted curve in red. As we will need this again later, we define a function:

```
> plotDispEsts <- function( cds )
+ {
+    plot(
+       rowMeans( counts( cds, normalized=TRUE ) ),
+       fitInfo(cds)$perGeneDispEsts,
+       pch = '.', log="xy" )
+    xg <- 10^seq( -.5, 5, length.out=300 )
+    lines( xg, fitInfo(cds)$dispFun( xg ), col="red" )
+ }
```

Calling the function produces the plot (Fig. 1).

```
> plotDispEsts( cds )
```

The plot in Figure 1 is doubly logarithmic; this may be helpful or misleading, and it is worth experimenting with other plotting styles.

As we estimated the dispersion from only two samples, we should expect the estimates to scatter with quite some sampling variance around their true values. Hence, we *DESeq* should not use the per-gene estimates directly in the test, because using too low dispersion values leads to false positives. Many of the values below the red line are likely to be underestimates of the true dispersions, and hence, it is prudent to instead rather use the fitted value. On the othe hand, not all of he values above the red line are overestimations, and hence, the conservative choice is to keep them instead of replacing them with their fitted values. If you do not like this default behaviour, you can change it with the option `sharingMode` of `estimateDispersions`. Note that *DESeq* orginally (as described in [1]) only used the fitted values (`sharingMode="fit-only"`). The current default (`sharingMode="maximum"`) is more conservative.

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, `estimateDispersion` 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 finally should 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
```

You can verify that `disp_pooled` indeed contains the maximum of the two value vectors we looked at before, namely

```
> 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"
```

Advanced users who want to fiddle with the dispersion estimation can change the values in `fData(cds)` prior to calling the testing function.

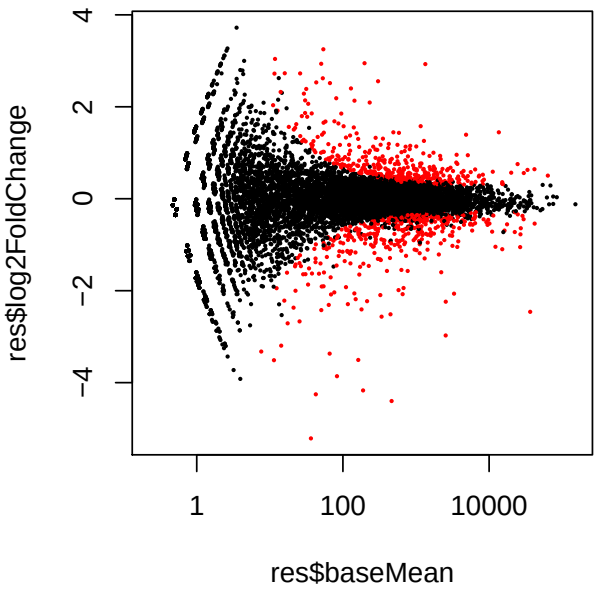

Figure 2: Plot of normalised mean versus log_2 fold change (this plot is sometimes also called the "MA-plot") for the contrast "untreated" versus "treated".

3 Inference: Calling differential expression

3.1 Standard comparison between two experimental conditions

Having estimated the dispersion for each gene, it is now 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 the paper 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
1 FBgn0000003    0.224      0.00     0.449        Inf            Inf 1.000
2 FBgn0000008   76.296     78.16    74.436      0.952        -0.0704 0.835
3 FBgn0000014    0.000      0.00     0.000        NaN            NaN    NA
4 FBgn0000015    0.781      1.56     0.000      0.000           -Inf 0.416
5 FBgn0000017 3298.682   3599.47  2997.890      0.833        -0.2638 0.241
6 FBgn0000018  289.031    293.68   284.385      0.968        -0.0464 0.757
   padj
```

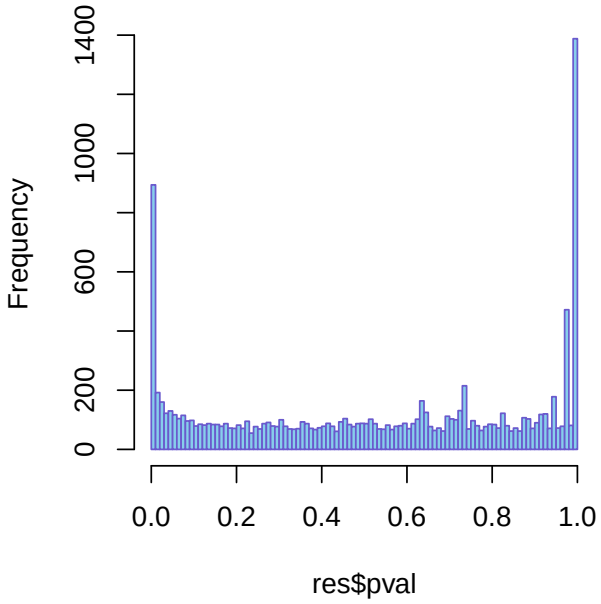

Figure 3: Histogram of p-values from the call to nbinomTest.

```
1 1.000
2 1.000
3    NA
4 1.000
5 0.881
6 1.000
```

For each gene, we get its mean expression level (at the base scale) as a joint estimate from both conditions, and estimated separately for each condition, the fold change from the first to the second condition, the logarithm (to basis 2) of the fold change, and the p value for the statistical significance of this change. The `padj` column contains the p values, 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 base means, colouring in red those genes that are significant at 10% FDR.

```
> plotDE <- function( res )
+    plot(
+       res$baseMean,
+       res$log2FoldChange,
+       log="x", pch=20, cex=.3,
+       col = ifelse( res$padj < .1, "red", "black" ) )
> plotDE( res )
```

See Figure 2 for the plot. As we will use this plot more often, we have stored its code in a function.

It is also instructive to look at the histogram of p values (Figure 3). The enrichment of low po values stems from the differentially expressed genes, while those not differebtially 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 higher bins to the right.)

```
> 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
9831  FBgn0039155      463       885      41.9     0.0474          -4.40
2366  FBgn0025111     1340       311    2369.3     7.6141           2.93
612   FBgn0003360     2544      4514     574.6     0.1273          -2.97
3192  FBgn0029167     2551      4211     891.7     0.2117          -2.24
10305 FBgn0039827      189       357      19.9     0.0556          -4.17
6948  FBgn0035085      447       761     133.3     0.1751          -2.51
          pval      padj
9831  1.64e-124 1.89e-120
2366  3.50e-107 2.01e-103
612    1.55e-99  5.95e-96
3192   4.35e-78  1.25e-74
10305  1.19e-65  2.74e-62
6948   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
14078 FBgn0259236     4.27      8.54      0.00     0.0000           -Inf
13584 FBgn0085359    36.47     71.03      1.92     0.0270          -5.21
9831  FBgn0039155   463.44    884.96     41.91     0.0474          -4.40
2277  FBgn0024288    42.56     80.89      4.24     0.0524          -4.25
10305 FBgn0039827   188.59    357.33     19.86     0.0556          -4.17
6495  FBgn0034434    82.89    155.09     10.68     0.0689          -3.86
          pval      padj
14078  1.31e-03  2.60e-02
13584  2.40e-09  2.19e-07
9831  1.64e-124 1.89e-120
2277   5.57e-20  1.94e-17
10305  1.19e-65  2.74e-62
6495   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
6030  FBgn0033764     53.9     10.26      97.6       9.52           3.25
13079 FBgn0063667     11.8      2.55      21.0       8.23           3.04
7020  FBgn0035189    197.5     45.30     349.6       7.72           2.95
8499  FBgn0037290     50.5     11.66      89.2       7.65           2.94
2366  FBgn0025111   1340.2    311.17    2369.3       7.61           2.93
7264  FBgn0035539     16.0      4.19      27.8       6.63           2.73
          pval      padj
6030   9.95e-18  2.93e-15
13079  1.43e-04  4.22e-03
7020   6.43e-15  1.35e-12
8499   1.04e-14  2.05e-12
2366  3.50e-107 2.01e-103
7264   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.)

```
> #not run
> write.table( res, file="results.txt" )
```

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.

```
> plot( rowMeans(ncu), log2( ncu[,2] / ncu[,1] ), pch=".", log="x" )
```

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:

```
> cdsTTU <- cds[ , 1:3]
> pData( cdsTTU )
```

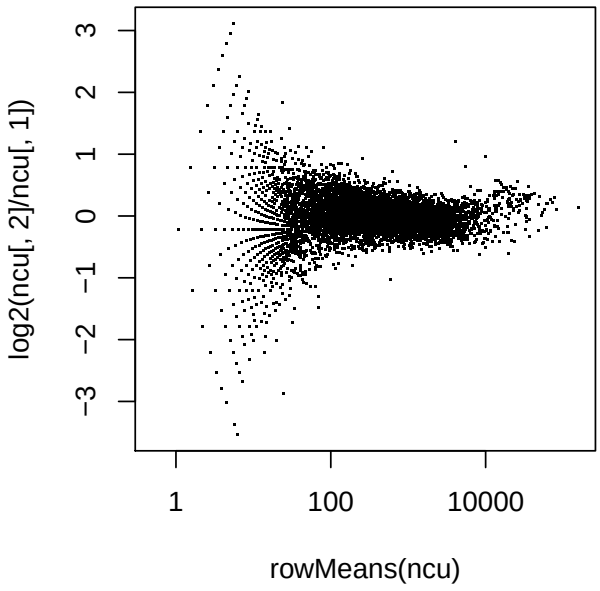

Figure 4: Plot of the log2 fold change between the untreated replicates versus average expression strength.

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

Now, we do the analysis as before.

```
> cdsTTU <- estimateSizeFactors( cdsTTU )
> cdsTTU <- estimateDispersions( cdsTTU )
> resTTU <- nbinomTest( cdsTTU, "untreated", "treated" )
```

We produce the analogous plot as before, again with

```
> plotDE( resTTU )
```

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.

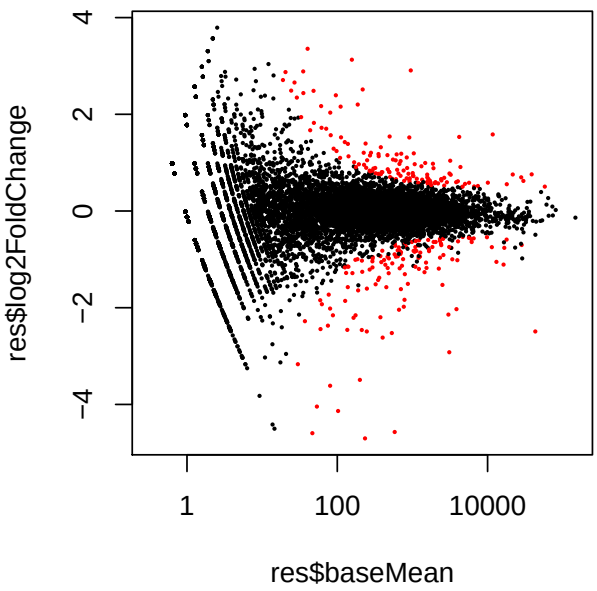

Figure 5: MvA plot for the contrast "treated" vs. "untreated", using two treated and only one untreated sample.

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 any 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 still 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"   ) ]
```

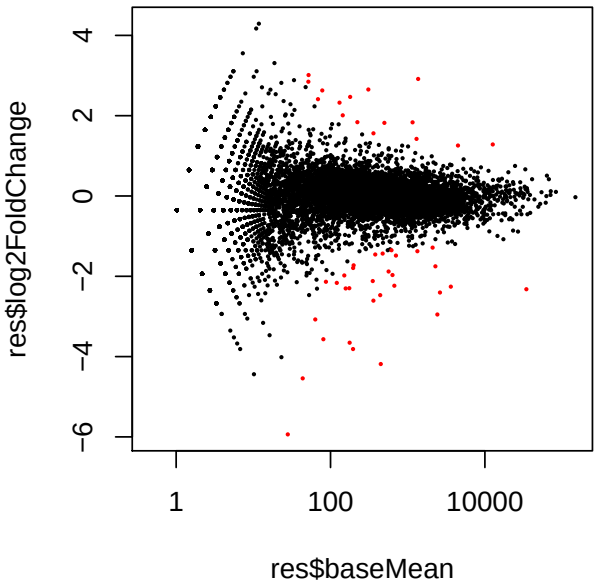

Figure 6: MvA plot, from a test using no replicates.

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 larer than the fitted values. Without replicates, we cannot catch such outliers and so have to disable this function.

Now, we can attempt to find differential expression:

```
> res2 <- nbinomTest( cds2, "untreated", "treated" )
```

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

```
> plotDE( 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 this data:

```
> head( pasillaCountTable )
```

```
            untreated1 untreated2 untreated3 untreated4 treated1 treated2
FBgn0000003          0          0          0          0        0        0
FBgn0000008         92        161         76         70      140       88
FBgn0000014          5          1          0          0        4        0
FBgn0000015          0          2          1          2        1        0
FBgn0000017       4664       8714       3564       3150     6205     3072
FBgn0000018        583        761        245        310      722      299
            treated3
FBgn0000003        1
FBgn0000008       70
FBgn0000014        0
FBgn0000015        0
FBgn0000017     3334
FBgn0000018      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. For the latter, we specify the method "pooled". Then, only one dispersion is computed for each gene, 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.

```
> cdsFull <- estimateSizeFactors( cdsFull )
> cdsFull <- estimateDispersions( cdsFull )
```

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.

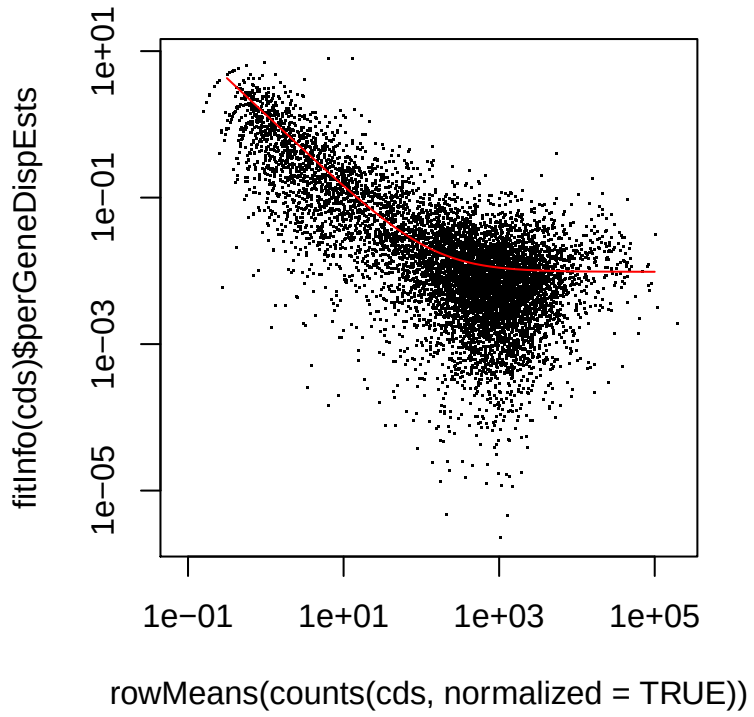

Figure 7: Estimated (black) pooled dispersion values for all seven samples, with regression curve (red).

```
> 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

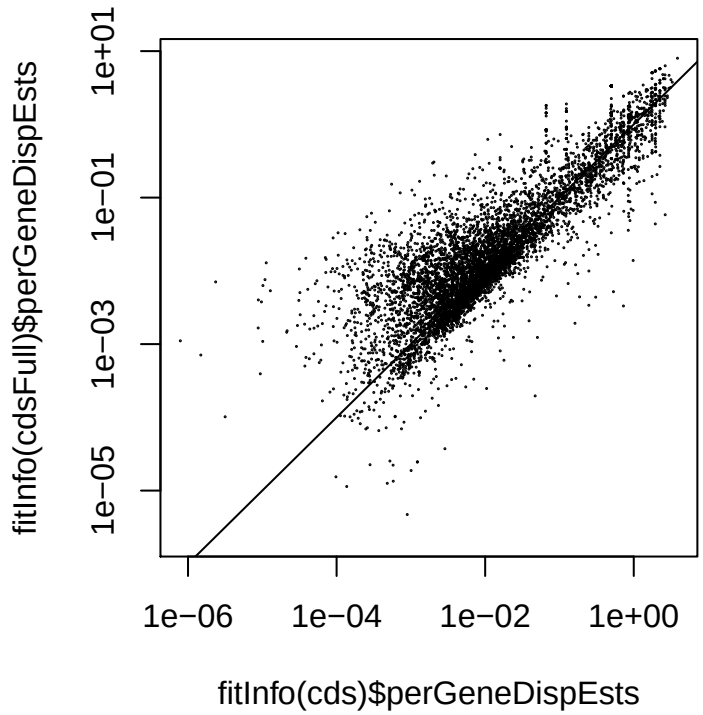

Figure 8: Comparison of per-genes estimates of the dispersion in the analysis using only the four paired-end samples and in the analysis using all seven samples. We can see that the additional samples tend to increase dispersion, which may indicate that the paired-end samples are of higher quality than the single-end ones.

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

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.

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

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

```
                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, despite the fact that we now have more degrees of freedom for the dispersion estimation , i.e., the additional sample increased not only signal but also noise. See Fig. 8, which is produced by

```
> plot(
+    fitInfo(cds)$perGeneDispEsts,
+    fitInfo(cdsFull)$perGeneDispEsts,
+    asp=1, log="xy",  pch=20,cex=.1 )
> abline( a=0, b=1 )
```

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`.)

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" )
```

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

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

5 Independent filtering

The analyses of the previous sections involve the application of statistical tests, one by one, to each row of the dataset, 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 doing the testing. Typically, this results in increased detection power — at the same type I error as measured, e. g., in terms of 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 5.1. Its statistical validity relies on properties 1 and 3, and we refer to [5] for further explanation of this topic. Here, we consider filtering by the overall sum of counts (irrespective of biological condition):

```
> rs <- rowSums ( counts ( cdsFull ))
> use <- (rs > quantile(rs, 0.4))
> table(use)

use
FALSE  TRUE
 5861  8738

> cdsFilt <- cdsFull[ use, ]
```

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
> tab = table( `no filtering`   = padjGLM < .1,
+              `with filtering` = padjFiltForComparison < .1 )
> addmargins(tab)
```

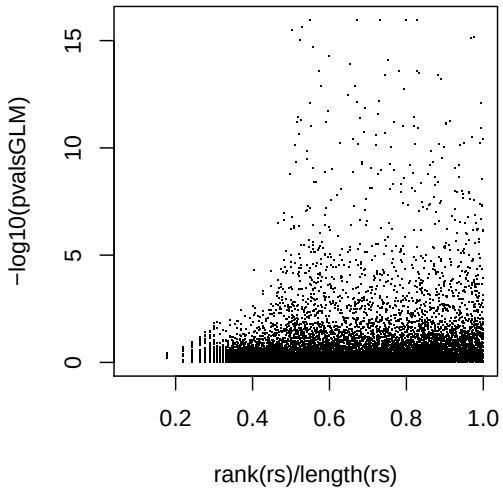

Figure 9: Scatterplot of rank of filter criterion (overall sum of counts `rs`) versus the negative logarithm of the test statistic `pvalsGLM`.

```
             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.

5.1 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 achieve an (unadjusted) p value less than 0.003.

```
> plot(rank(rs)/length(rs), -log10(pvalsGLM), pch=".")
```

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 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")
```

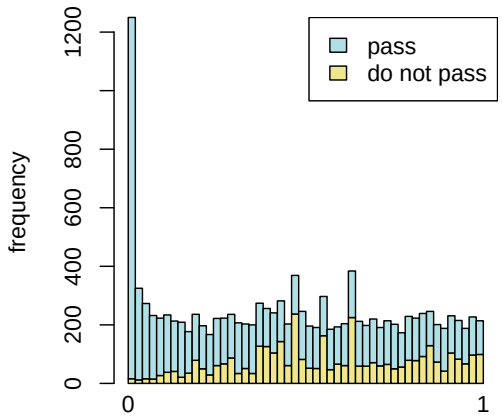

Figure 10: Histogram of p values for all tests (`pvalsGLM`). The area shaded in blue indicates the subset of those that pass the filtering, the area in khaki those that do not pass.

```
> 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)))
```

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 logarithmic transformation. Since count values for a gene can be zero in some conditions (and non-zero in others), some people advocate the use of *pseudocounts*, i. e. transformations of the form

$$y = \log_2(n + n_0), \tag{1}$$

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 is based on error modeling and the concept of variance stabilizing transformations [6, 7, 1]. We estimate an overall mean-dispersion relationship of the data using `estimateDispersions` with the argument `method="blind"` and call the function `getVarianceStabilizedData`.

```
> cdsBlind <- estimateDispersions( cds, method="blind" )
> vsd <- getVarianceStabilizedData( 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

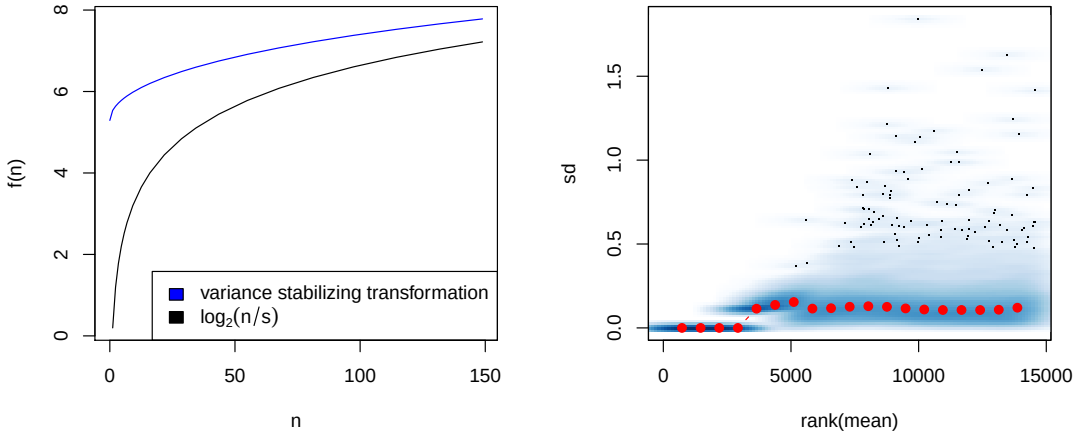

Figure 11: Left: Graphs of the variance stabilizing transformation for sample 1, in blue, and of the transformation $f(n) = log_2(n/s_1)$, in black. n are the counts and s_1 is the size factor for the first sample. Right: density plot of the per-gene standard deviation of the transformed data, across samples, against the rank of the mean.

in the file `vst.pdf`, that is distributed in the 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 the left panel of Figure 11. 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.

The right panel of Figure 11 plots the standard deviation of the transformed data, across samples, against the mean.

```
> library("vsn")
> meanSdPlot(vsd)
```

6.1 Application to moderated fold change estimates

In the beginning of Section 3, we have seen in the dataframe `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 might 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( vsd[, conditions(cds)=="treated", drop=FALSE] ) -
+             rowMeans( vsd[, conditions(cds)=="untreated", drop=FALSE] ))
```

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

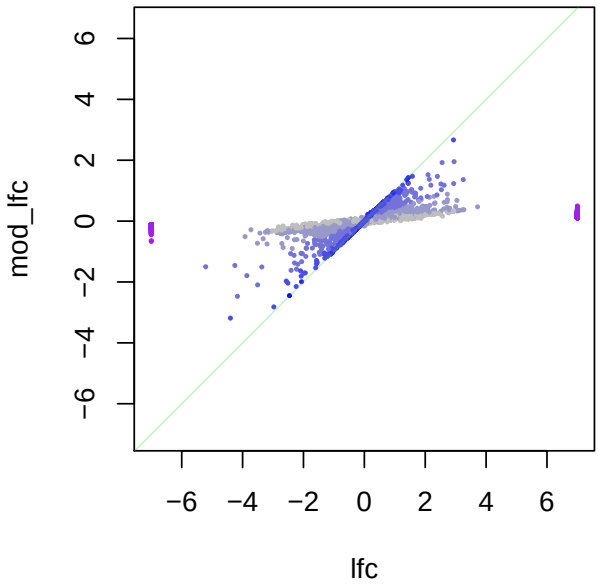

Figure 12: Scatterplot of direct (`lfc`) versus *moderated* log-ratios (`mod_lfc`). The moderation criterion used is variance stabilisation. The points are coloured in a scale from gray to blue, representing weakly to strongly expressed genes. The purple points correspond to values that were infinite in `lfc` and were abitrarily set to ±7 for the purpose of plotting. These values vary in a finite range (as shown in the plot) in `mod_lfc`.

division of 0 by 0).

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

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

For plotting, we replace the infinite values by an arbitrary fixed large number.

```
> largeNumber <- 7
> lfc <- ifelse(finite, lfc, sign(lfc) * largeNumber)
```

We also bin genes by their log10 mean to colour them according to expression strength.

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

The result is shown in Figure 12.

```
> plot( lfc, mod_lfc, pch=20, cex=.4, asp=1,
+       col = ifelse( finite, colors, "purple" ) )
> abline( a=0, b=1, col="#40C04040" )
```

The free parameters of the VST (at least for parametric dispersion fit) are chosen such that for large count values, the difference become asymptoticaly equal to log2 fold changes, as can be seen from the dark blue points in the figure. For lower count values (grayish points), the moderated fold change is smaller than the unmoderated one, reflecting the fact that the same count ratio is then less significant.

6.2 Application to sample clustering and visualisation

The moderated logarithmic fold changes `mod_lfc` are now approximately homoscedastic and hence suitable as input to a distance calculation. This can be useful, e. g., to visualise the expression data of the differentially expressed genes. The result, shown in Figure 13, shows the heatmap for the top 40.

```
> select <- order(res$pval)[1:40]
> colors <- colorRampPalette(c("white","darkblue"))(100)
> heatmap( vsd[select,],
+          col = colors, scale = "none")
```

For comparison, let us also try the same with the untransformed counts.

```
> heatmap( counts(cds)[select,],
+          col = colors, scale = "none")
```

The result is shown in Figure 13.

We note that the `heatmap` function that we have used here is rather basic, and that better options exist. For instance, consider the `heatmap.2` function from the package *gplots* or the manual page for `dendrogramGrob` in the package *latticeExtra*.

Another use of variance stabilized data is sample clustering. Here, we apply the `dist` function to the transpose of the `vsd` matrix to get sample-to-sample distances. We demonstrate this with the full data set with all seven samples.

```
> cdsFullBlind <- estimateDispersions( cdsFull, method = "blind" )
> vsdFull <- getVarianceStabilizedData( cdsFullBlind )
> dists <- dist( t( vsdFull ) )
```

A heatmap of this dstance matrix now shows us, which samples are similar (Figure 14):

```
> heatmap( as.matrix( dists ),
+    symm=TRUE, scale="none", margins=c(10,10),
+    col = colorRampPalette(c("darkblue","white"))(100),
+    labRow = paste( pData(cdsFullBlind)$condition, pData(cdsFullBlind)$libType ) )
```

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 make this conclusion, it was important to re-estimate e dispersion with mode “blind” (in the calculation for `cdsFullBlind` above, as only then, the variance stabilizing transformation is not informed about the design and hence not biased towards a result supporting it.) Such an analysis is useful for quality control, and (even though we finish our vignette with it), it may be useful to this first in any analysis.

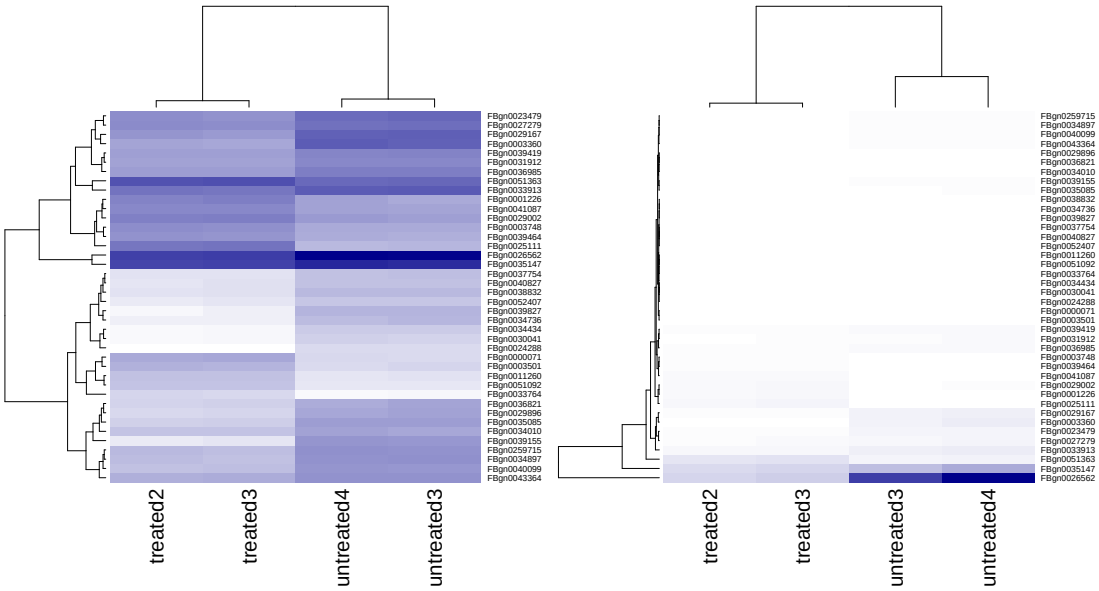

Figure 13: Heatmaps showing the expression data of the top 40 differentially expressed genes. Left, the variance stabilisation transformed data (`vsd`), right, the original count data (`countscds`). The right plot is dominated by a small number of data points with large values.

7 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`.

8 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

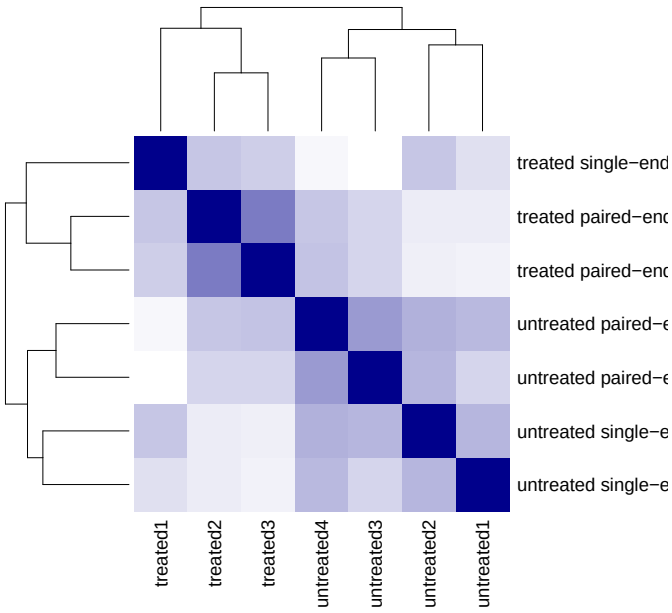

Figure 14: Heatmap showing the Euclidean distances between the samples as calculated from the variance-stabilising transformation of the count data.

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 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*.

9 Session Info

```
> sessionInfo()

R version 2.15.0 (2012-03-30)
Platform: i386-apple-darwin9.8.0/i386 (32-bit)

locale:
[1] C

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

other attached packages:
[1] vsn_3.24.0         DESeq_1.8.3        locfit_1.5-8       Biobase_2.16.0
[5] BiocGenerics_0.2.0

loaded via a namespace (and not attached):
 [1] AnnotationDbi_1.18.1  BiocInstaller_1.4.4   DBI_0.2-5
 [4] IRanges_1.14.3        KernSmooth_2.23-7     RColorBrewer_1.0-5
 [7] RSQLite_0.11.1        affy_1.34.0           affyio_1.24.0
[10] annotate_1.34.0       genefilter_1.38.0     geneplotter_1.34.0
[13] grid_2.15.0           lattice_0.20-6        limma_3.12.0
[16] preprocessCore_1.18.0 splines_2.15.0        stats4_2.15.0
[19] survival_2.36-14      tools_2.15.0          xtable_1.7-0
[22] zlibbioc_1.2.0
```

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] Simon Anders. HTSeq: Analysing high-throughput sequencing data with Python. `http://www-huber.embl.de/users/anders/HTSeq/`, 2011.

[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.

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

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

[7] 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.