Analysing RNA-Seq data with the DESeq package

Simon Anders

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany

sanders@fs.tum.de

November 21, 2011

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 HTSeq 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]. The data are presented in the object called `pasillaGenes`. For a description how this data object was created from the raw data of Ref. [4], see the vingette included with the *pasilla* package.

The `pasillaGenes` object is of class *CountDataSet*, which is the data container used by *DESeq*. We load the needed packages and the data as follows.

```
> library( "DESeq" )
> library( "pasilla" )
> data( "pasillaGenes" )
```

`pasillaGenes` contains the counts and also metadata about the samples:

```
> head( counts(pasillaGenes) )
```

```
            treated1fb treated2fb treated3fb untreated1fb untreated2fb
FBgn0000003          0          1          1            0            0
```

```
FBgn0000008        118        139         77          89         142
FBgn0000014          0         10          0           1           1
FBgn0000015          0          0          0           0           0
FBgn0000017       4852       4853       3710        4640        7754
FBgn0000018        572        497        322         552         663
            untreated3fb untreated4fb
FBgn0000003            0            0
FBgn0000008           84           76
FBgn0000014            0            0
FBgn0000015            1            2
FBgn0000017         4026         3425
FBgn0000018          272          321
```

```
> pData( pasillaGenes )
```

```
             sizeFactor condition replicate        type
treated1fb           NA   treated         1 single-read
treated2fb           NA   treated         2  paired-end
treated3fb           NA   treated         3  paired-end
untreated1fb         NA untreated         1 single-read
untreated2fb         NA untreated         2 single-read
untreated3fb         NA untreated         3  paired-end
untreated4fb         NA untreated         4  paired-end
```

As you can see, the samples differ by experimental condition (untreated or treated, i. e., with pasilla knocked down) and by library type. To keep things simple, we will only look at the paired-end data for now. In Section 4, we will see how to deal with more than one factor.

For your own analysis, you will start form a count table, so we "unpack" the *countDataSet* object and build a new one "from scratch" to demonstrate how this is done.

```
> pairedSamples <- pData(pasillaGenes)$type == "paired-end"
> countsTable <- counts(pasillaGenes)[ , pairedSamples ]
> conds <- pData(pasillaGenes)$condition[ pairedSamples ]
```

Now, we have a count table, as described above, of integer count data. For your own data, use R's `read.table` or `read.csv` function to read your count data from a text file.

We also need a description of the samples, which is here simply a factor:

```
> conds
```

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

For your own data, create such a factor simply with

```
> #not run
> conds <- factor( c( "treated", "treated", "untreated", "untreated" ) )
```

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

```
> cds <- newCountDataSet( countsTable, 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) )
```

```
           treated2fb treated3fb untreated3fb untreated4fb
FBgn0000003          1          1            0            0
FBgn0000008        139         77           84           76
FBgn0000014         10          0            0            0
FBgn0000015          0          0            1            2
FBgn0000017       4853       3710         4026         3425
FBgn0000018        497        322          272          321
```

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 )
```

```
  treated2fb   treated3fb untreated3fb untreated4fb
       1.297        1.042        0.819        0.911
```

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 ) )
```

```
           treated2fb treated3fb untreated3fb untreated4fb
FBgn0000003      0.771       0.96         0.00          0.0
FBgn0000008    107.176      73.91       102.62         83.4
FBgn0000014      7.710       0.00         0.00          0.0
FBgn0000015      0.000       0.00         1.22          2.2
FBgn0000017   3741.902    3561.30      4918.38       3760.7
FBgn0000018    383.212     309.09       332.29        352.5
```

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 $.20^2 = .04$. Note that the variance seen between counts is the sum of two components: the

[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.

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 [1:14470] -2.2925 0.0327 3.4747 -0.8601 0.0212 ...
 $ dispFunc       :function (q)
  ..- attr(*, "coefficients")= Named num [1:2] 0.00891 1.61534
  .. ..- attr(*, "names")= chr [1:2] "asymptDisp" "extraPois"
  ..- attr(*, "fitType")= chr "parametric"
 $ fittedDispEsts : num [1:14470] 3.7417 0.02651 0.8469 1.89949 0.00931 ...
 $ 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).

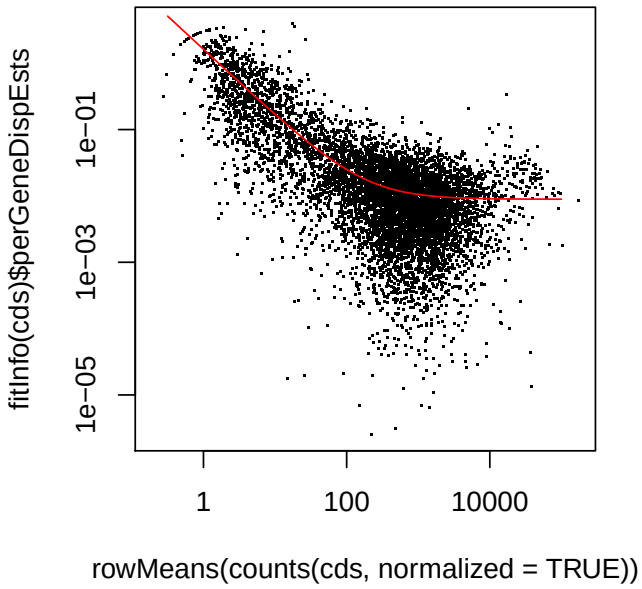

Figure 1: Empirical (black dots) and fitted (red lines) dispersion values plotted against mean expression strength.

```
> 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      3.7417
FBgn0000008      0.0327
FBgn0000014      3.4747
FBgn0000015      1.8995
FBgn0000017      0.0212
FBgn0000018      0.0136
```

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 [1:14470] -2.2925 0.0327 3.4747 -0.8601 0.0212 ...
 $ dispFunc       :function (q)
  ..- attr(*, "coefficients")= Named num [1:2] 0.00891 1.61534
  .. ..- attr(*, "names")= chr [1:2] "asymptDisp" "extraPois"
  ..- attr(*, "fitType")= chr "parametric"
 $ fittedDispEsts : num [1:14470] 3.7417 0.02651 0.8469 1.89949 0.00931 ...
 $ 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.

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    0.433      0.00     0.865        Inf            Inf 0.827
2 FBgn0000008   91.789     93.03    90.545      0.973        -0.0391 1.000
3 FBgn0000014    1.928      0.00     3.855        Inf            Inf 0.378
4 FBgn0000015    0.854      1.71     0.000      0.000           -Inf 0.413
5 FBgn0000017 3995.560   4339.52  3651.603      0.841        -0.2490 0.278
```

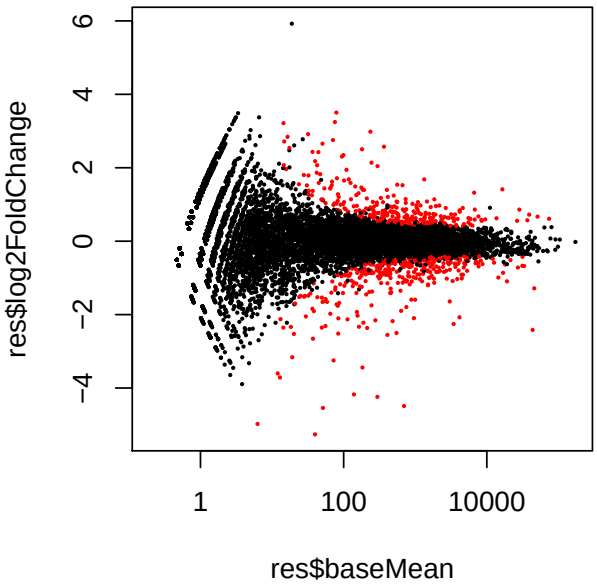

Figure 2: Plot of normalised mean versus log_2 fold change (this plot is sometimes also called the "MA-plot") for the contrast "untreated" versus "treated".

```
6 FBgn0000018  344.264    342.37   346.153      1.011         0.0158 0.891
   padj
1 1.000
2 1.000
3 1.000
4 1.000
5 0.964
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,
```

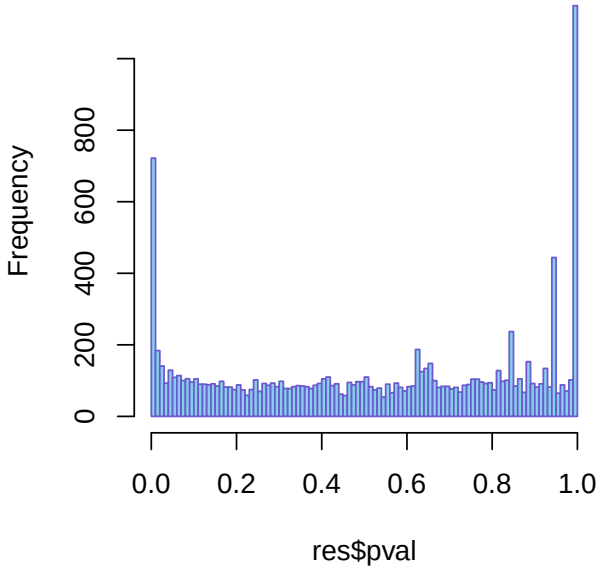

Figure 3: Histogram of p-values from the call to nbinomTest.

```
+       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
9696  FBgn0039155      697      1335      59.4     0.0445          -4.49
10162 FBgn0039827      296       563      29.8     0.0529          -4.24
```

```
3163  FBgn0029167     3435      5680    1190.0     0.2095          -2.25
6408  FBgn0034434      139       264      14.6     0.0554          -4.17
6855  FBgn0035085      547       930     164.3     0.1766          -2.50
6622  FBgn0034736      183       336      30.9     0.0920          -3.44
          pval      padj
9696  5.00e-110 5.75e-106
10162  2.64e-70  1.52e-66
3163   8.16e-50  3.13e-46
6408   1.13e-42  3.25e-39
6855   4.32e-42  9.93e-39
6622   2.10e-39  4.02e-36
```

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
13945 FBgn0259236     4.64      9.28     0.000     0.0000           -Inf
13353 FBgn0085359    39.83     77.63     2.022     0.0260          -5.26
3873  FBgn0030634     6.27     12.15     0.386     0.0317          -4.98
2263  FBgn0024288    51.41     98.59     4.233     0.0429          -4.54
9696  FBgn0039155   697.22   1335.03    59.420     0.0445          -4.49
10162 FBgn0039827   296.27    562.75    29.789     0.0529          -4.24
          pval      padj
13945  1.44e-03  3.59e-02
13353  2.34e-08  2.52e-06
3873   8.47e-04  2.40e-02
2263   3.77e-21  2.55e-18
9696  5.00e-110 5.75e-106
10162  2.64e-70  1.52e-66
```

or at the most strongly up-regulated ones:

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

```
               id baseMean baseMeanA baseMeanB foldChange log2FoldChange
5955  FBgn0033764     79.1     12.82     145.3      11.34           3.50
8385  FBgn0037290     75.5     14.41     136.5       9.48           3.24
12889 FBgn0063667     14.4      2.81      26.1       9.29           3.22
6926  FBgn0035189    236.5     53.11     420.0       7.91           2.98
5505  FBgn0033065     31.8      7.45      56.2       7.55           2.92
9712  FBgn0039178     16.5      4.03      28.9       7.17           2.84
         pval     padj
5955  5.74e-21 3.46e-18
8385  4.56e-16 1.81e-13
12889 8.09e-05 3.17e-03
6926  7.97e-14 2.54e-11
5505  1.49e-07 1.33e-05
9712  4.71e-03 9.18e-02
```

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.)

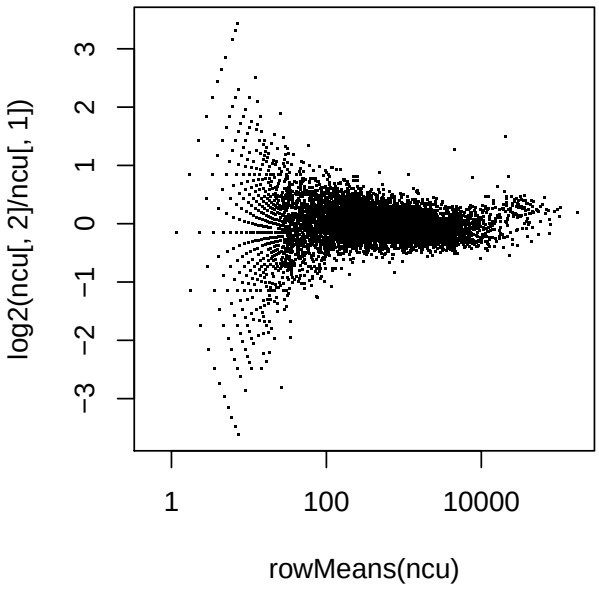

Figure 4: Plot of the log2 fold change between the untreated replicates versus average expression strength.

```
> #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,

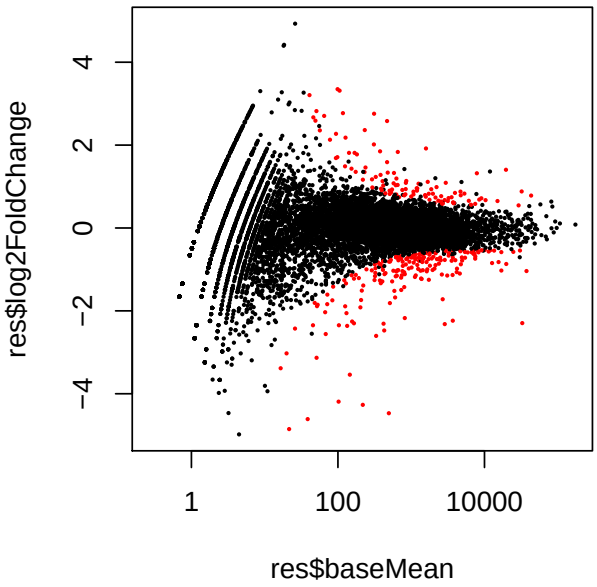

Figure 5: MvA plot for the contrast "treated" vs. "untreated", using two treated and only one untreated sample.

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 )
```

```
             sizeFactor condition
treated2fb        1.297   treated
treated3fb        1.042   treated
untreated3fb      0.819 untreated
```

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.

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(  "untreated3fb", "treated3fb"   ) ]
```

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 10249     2 10251
  TRUE    563    48   611
  Sum   10812    50 10862
```

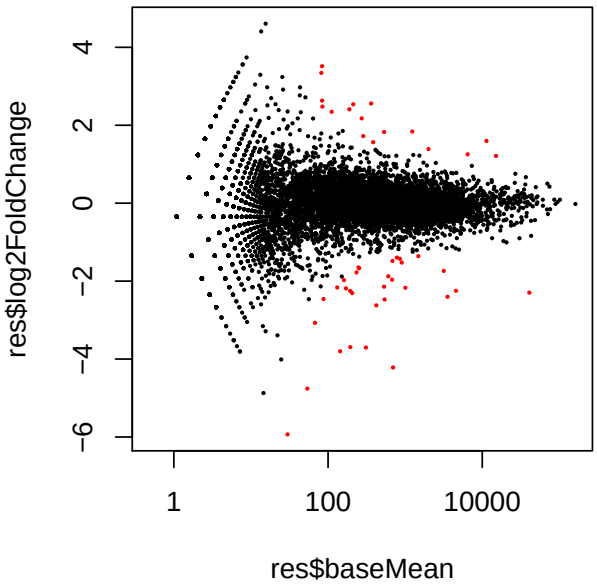

Figure 6: MvA plot, from a test using no replicates.

4 Multi-factor designs

Let us return to the full pasilla data set, which we got as `pasillaGenes` from the *pasilla* package. Due to the usage of both single-end and paired-end libraries, it has a design with two factors, *condition* (or treatment) and library *type*:

```
> design <- pData(pasillaGenes)[ , c("condition","type") ]
> design
```

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

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

```
> fullCountsTable <- counts( pasillaGenes )
> cdsFull <- newCountDataSet( fullCountsTable, design )
```

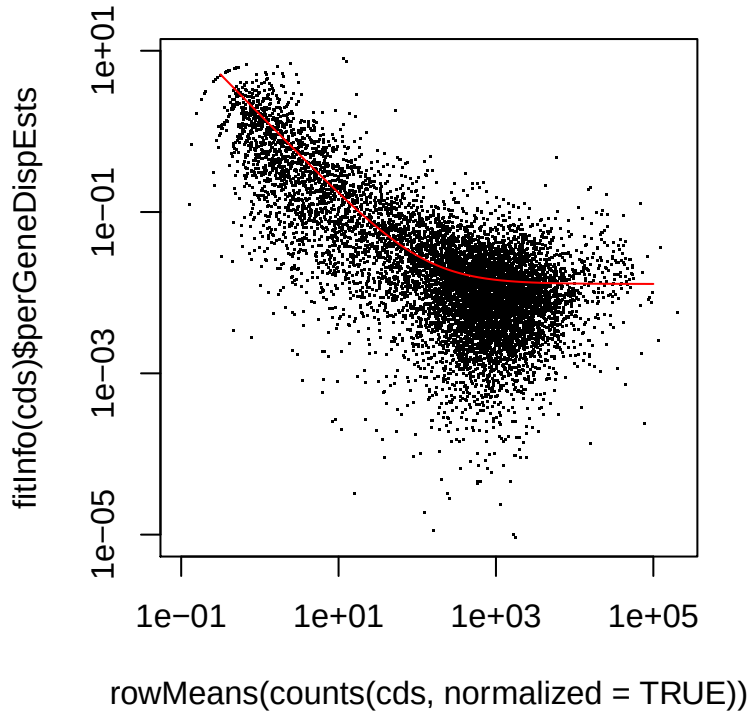

Figure 7: Estimated (black) pooled dispersion values for all seven samples, with regression curve (red).

`cdsFull` is now essentially the same object as `pasillaGenes`, we have only recreated it for demonstration.

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.

```
> fit1 <- fitNbinomGLMs( cdsFull, count ~ type + condition )
> fit0 <- fitNbinomGLMs( cdsFull, count ~ type  )
```

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':        14470 obs. of  5 variables:
 $ (Intercept)       : num  -0.00815 6.74301 1.55925 -33.63515 12.03815 ...
 $ typesingle-read   : num  -34.1079 -0.1665 1.0097 -34.9124 -0.0402 ...
 $ conditionuntreated: num  -34.5327 -0.0336 -3.5819 34.6179 0.2508 ...
 $ deviance          : num  0.00349 2.70161 3.40697 0.04691 2.61974 ...
 $ converged         : logi  TRUE TRUE FALSE TRUE TRUE TRUE ...
 - attr(*, "df.residual")= Named num 4
  ..- attr(*, "names")= chr "FBgn0000003"
```

To perform the test, we call

```
> 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 10564   315 10879
          TRUE     88   523   611
          Sum   10652   838 11490
```

We see that the analyses find 523 genes in common, while 315 were only found in the analysis using all samples and 88 were specific for the *paired end only* analysis. A more informative comparison might be a scatter plot of p values:

```
> bottom = function(x, theta=1e-12) pmax(x, theta)
> plot( bottom(res$pval), bottom(pvalsGLM), log="xy", pch=20, cex=.3 )
> abline(a=0, b=1, col="blue")
```

The result is shown in Fig. 8.

Now, we can 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)
```

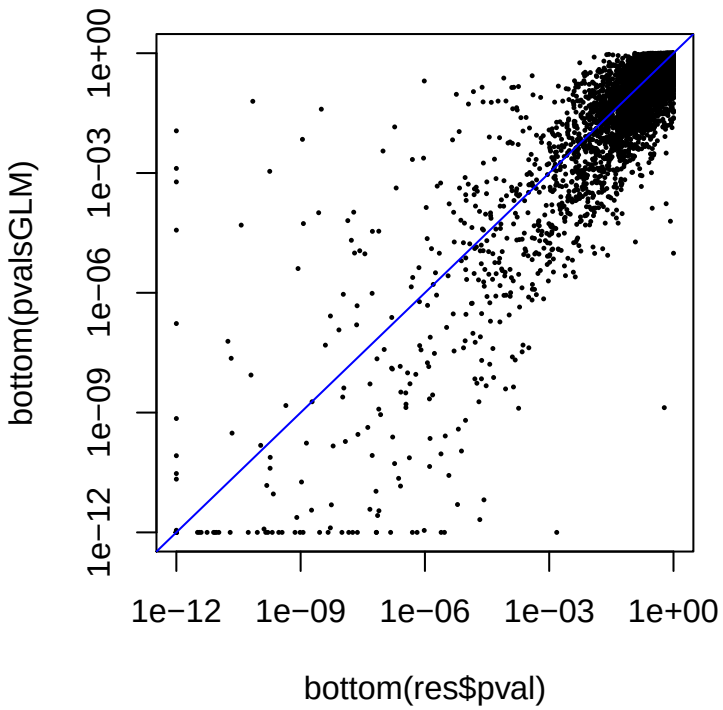

Figure 8: Comparison of p values (unadjusted) from the test using only the four paired-end samples and the test using all seven samples. We can see that the latter tend to be smaller, as expected from the higher power of the test with all samples.

	(Intercept)	typesingle-read	conditionuntreated	deviance	converged
FBgn0000003	-0.00815	-34.1079	-3.45e+01	0.00349	TRUE
FBgn0000008	6.74301	-0.1665	-3.36e-02	2.70161	TRUE
FBgn0000014	1.55925	1.0097	-3.58e+00	3.40697	FALSE
FBgn0000015	-33.63515	-34.9124	3.46e+01	0.04691	TRUE
FBgn0000017	12.03815	-0.0402	2.51e-01	2.61974	TRUE
FBgn0000018	8.63370	0.2949	-3.52e-04	1.97766	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( fullCountsTable, design$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 11358   281 11639
             TRUE     15   557   572
             Sum   11373   838 12211
```

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
 5808  8662
```

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

We perform the testing as before in Section 4.

```
> fitFilt1  <- fitNbinomGLMs( cdsFilt, count ~ type + condition )
> fitFilt0  <- fitNbinomGLMs( cdsFilt, count ~ type  )
> 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)

            with filtering
no filtering FALSE  TRUE   Sum
       FALSE 11282    91 11373
       TRUE      6   832   838
       Sum   11288   923 12211
```

The analysis with filtering found an additional 91 genes, an increase in the detection rate by about 11%, while 6 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 5e-04.

```
> 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")

> 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)))
```

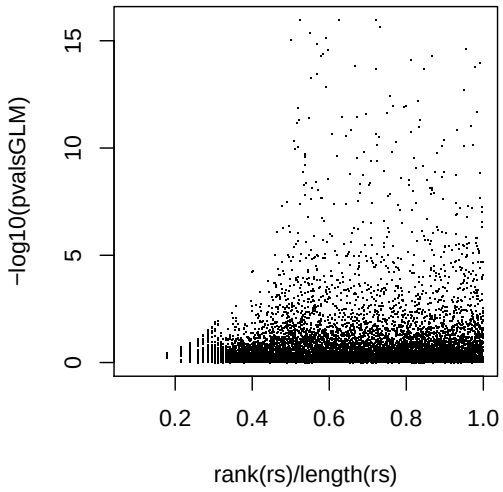

Figure 9: Scatterplot of rank of filter criterion (overall sum of counts `rs`) versus the negative logarithm of the test statistic `pvalsGLM`.

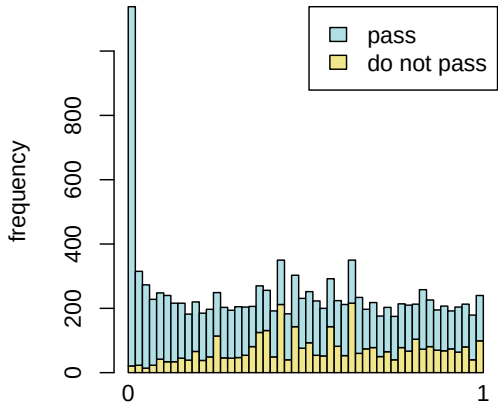

Figure 10: Histogram of p values for all tests (`pvalsGLM`). The area shaded in blue indicates the subset of those that pass the filtering, the area in khaki those that do not pass.

6 Moderated fold change estimates and applications to sample clustering and visualisation

In Section 3 we have seen how to use *DESeq* for calling differentially expressed genes. For each gene, *DESeq* reports a (logarithm base 2) fold change estimate and a p value, as shown for instance in the dataframe `res` in the beginning of that section. When the involved counts are small, the (logarithmic) fold-change estimate can be highly variable, and can even be infinite.

For some purposes, such as the clustering of samples (or genes) according to their overall profiles, or for visualisation of the data, the (logarithmic) fold changes may thus not be useful: the random variability associated with fold changes computed from ratios between low counts might drown informative, systematic signal in other parts of the data. We would like to *moderate* the fold change estimates in some way, so that they are more amenable to plotting or clustering. One approach to do so uses so-called pseudocounts: instead of the log-ratio $\log_2(n_A/n_B)$ between the counts n_A, n_B in two conditions A and B consider $\log_2((n_A+c)/(n_B+c))$, where c is a small positive number, e. g. $c = 0.5$ or $c = 1$. For small values of either n_A or n_B, or both, the value of this term is shifted towards 0 compared to the direct log-ratio $\log_2(n_A/n_B)$. When n_A and n_B are both large, the direct log-ratio and the log-ratio with pseudocounts (asymptotically) agree. This approach is simple and intuitive, but it requires making a choice for what value to use for c, and that may not be obvious.

A variant of this approach is to look for a mathematical function of n_A and n_B that is like $\log_2(n_A/n_B)$ when n_A and n_B are large enough, but still behaves gracefully when they become small. If we interpret *graceful* as having the same variance throughout, then we arrive at variance stabilising transformations (VST) [1]. An advantage is that the parameters of this function are chosen automatically based on the variability of the data, and no *ad hoc* choice of c, as above, is necessary.

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

The data are now on a logarithm-like scale, and we can compute *moderated log fold changes*.

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

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

```
-Inf  Inf  NaN <NA>
 541  640 2980    0
```

For plotting (Figure 11), we replace the infinite values by an arbitrary fixed large number:

```
> largeNumber <- 10
> lfc <- ifelse(finite, lfc, sign(lfc) * largeNumber)
```

```
> plot( lfc, mod_lfc, pch=20, cex=.3,
+       col = ifelse( finite, "#80808040", "red" ) )
> abline( a=0, b=1, col="#40404040" )
```

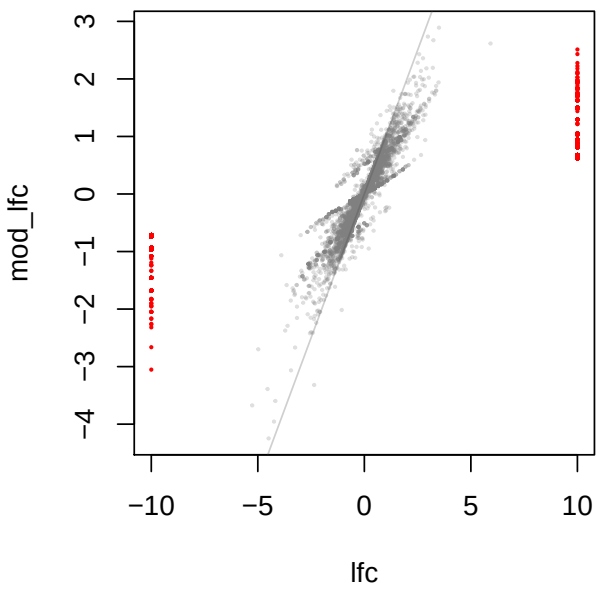

Figure 11: Scatterplot of direct (`lfc`) versus *moderated* log-ratios (`mod_lfc`). The moderation criterion used is variance stabilisation. The red points correspond to values that were infinite in `lfc` and were abitrarily set to ±10 for the purpose of plotting. These values vary in a finite range (as shown in the plot) in `mod_lfc`.

These data 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, say, the top 40 differentially expressed genes.

```
> 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 12.

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.

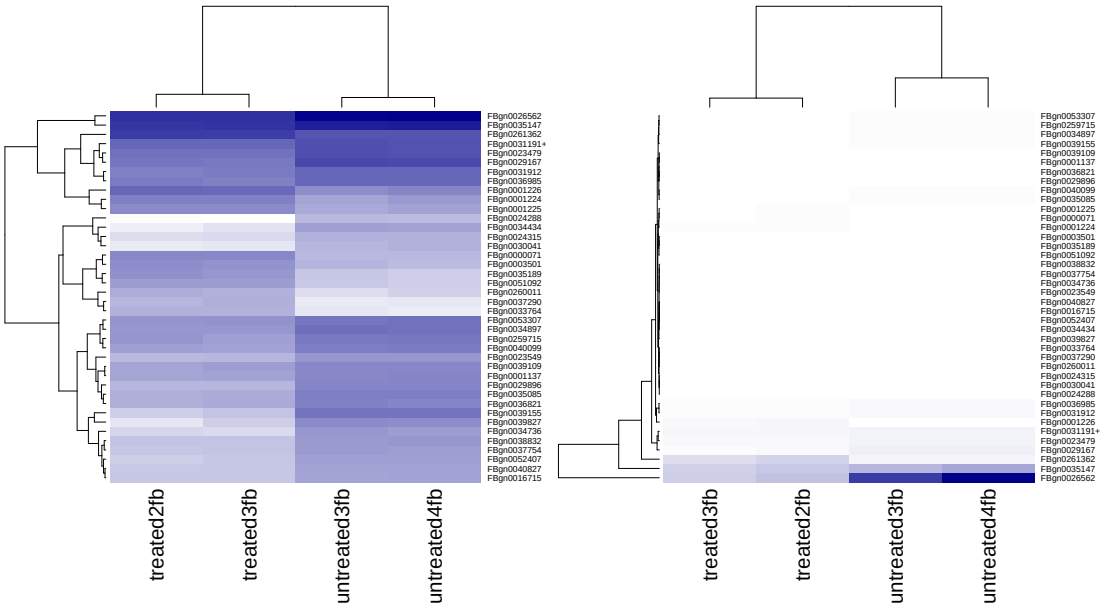

Figure 12: Heatmaps showing the expression data of the top 40 differentially expressed genes. Left, the variance stabilisation transformed data (`vsd`), right, the original count data (`countscds`). The right plot is dominated by a small number of data points with large values.

```
> 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 13):

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

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.

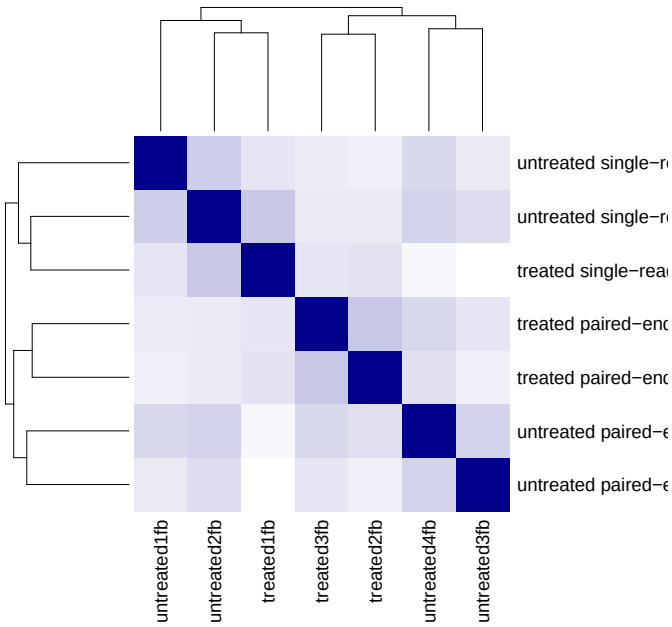

Figure 13: Heatmap showing the Euclidean distances between the samples as calculated from the variance-stabilising transformation of the count data.

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 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.
- 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 2.14.0 (2011-10-31)
Platform: i386-pc-mingw32/i386 (32-bit)

locale:
[1] LC_COLLATE=C
[2] LC_CTYPE=English_United States.1252
[3] LC_MONETARY=English_United States.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

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

other attached packages:
[1] pasilla_0.2.10 DESeq_1.6.1    locfit_1.5-6   lattice_0.20-0 akima_0.5-4
[6] Biobase_2.14.0

loaded via a namespace (and not attached):
 [1] AnnotationDbi_1.16.4 DBI_0.2-5            IRanges_1.12.2
 [4] RColorBrewer_1.0-5   RSQLite_0.10.0       annotate_1.32.0
 [7] genefilter_1.36.0    geneplotter_1.32.1   grid_2.14.0
[10] splines_2.14.0       survival_2.36-10     tools_2.14.0
[13] xtable_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] 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.