Analysing RNA-Seq data with the "DESeq" package

Simon Anders

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany

sanders@fs.tum.de

April 20, 2011

Contents

Abstract

In RNA-Seq and related assay types (including comparative ChIP-Seq etc.), one works with tables of count data, which report, for each sample, the number of reads that have been assigned to a gene (or other types of entities). The package "DESeq" provides a powerful tool to estimate the variance in such data and test for differential expression[1]. The present vignette explains the use of the package; for an exposition of the statistical method employed, see our paper[2].

[1] Other Bioconductor packages for this use case (but employing different methods) are *edgeR* and *baySeq*.

[2] The companion paper for DESeq is: S. Anders, W. Huber: *Differential expression analysis for sequence count data*, Genome Biology, 11:R106, 2010.

1 Quick start

This first section just shows the commands necessary for an analysis at a glance. For a more gentle introduction, skip this section on first reading and start reading at Section 2.

The *DESeq* package expects count data, as obtained, e.g., from an RNA-Seq or other high-throughput sequencing (HTS) experiment, in form of a matrix of integer values. Each column corresponds to a sample, i.e., typically one run on the sequencer. Each row corresponds to entity for which you count hits, e.g., a gene, an exon, a binding region in ChIP-Seq, a window in CNV-Seq, or the like.

Each column must stem from an independent biological replicate. For purely technical replicates (e. g. when the sample 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 important in order to allow *DESEq* to estimate variability in the experiment correctly.

Let's say you have the counts in a matrix or data frame `countTable`, and you further have a factor `conds` with as many element as there are columns in `countTable` that indicates treatment groups, i.e., data in the following form:

```
> head(countsTable)

           T1a T1b  T2  T3  N1  N2
Gene_00001   0   0   2   0   0   1
Gene_00002  20   8  12   5  19  26
Gene_00003   3   0   2   0   0   0
Gene_00004  75  84 241 149 271 257
Gene_00005  10  16   4   0   4  10
Gene_00006 129 126 451 223 243 149

> conds

[1] T  T  T  Tb N  N
Levels: N T Tb
```

Then, the minimal set of commands to run a full analysis is:

```
> cds <- newCountDataSet( countsTable, conds )
> cds <- estimateSizeFactors( cds )
> cds <- estimateVarianceFunctions( cds )
> res <- nbinomTest( cds, "T", "N")
```

The last command tests for differential expression between the conditions labelled `"T"` and `"N"`. It returns a data frame with p values (raw and adjusted), mean values, fold changes, and other useful information, which looks as follows:

```
> head(res)

          id    baseMean   baseMeanA  baseMeanB foldChange log2FoldChange
1 Gene_00001   0.4509631   0.3938651   0.536610  1.3624208      0.4461724
2 Gene_00002  17.9472488  16.0027575  20.863986  1.3037744      0.3826943
3 Gene_00003   1.0629635   1.7716058   0.000000  0.0000000           -Inf
4 Gene_00004 171.8057235 128.6778649 236.497511  1.8379036      0.8780611
5 Gene_00005  11.3021880  14.2894570   6.821284  0.4773648     -1.0668358
```

```
6 Gene_00006 198.2748364 218.2198341 168.357340  0.7715034     -0.3742556
       pval      padj    resVarA    resVarB
1 1.0000000 1.0000000 0.32470113 0.58677009
2 0.5247345 0.9327543 0.55266977 0.58059131
3 0.3620748 0.8776299 0.62957835 0.00000000
4 0.2451842 0.7679753 0.06033205 0.46985350
5 0.6770847 0.9861480 1.27442334 0.39877633
6 0.5825584 0.9587618 0.18697871 0.01700303
```

2 Preparations

As example data, we use Tag-Seq data from an experiment studying certain human tissue culture samples, which P. Bertone kindly permitted us to use. As these data are not yet published, we have obscured annotation data and will, for now, remain vague concerning their biological properties. We will amend this once Bertone and coworkers have published their paper.

They extracted mRNA from the cultures and sequenced only the 3' end of the transcripts (Tag-Seq) with an Illumina GenomeAnalyzer, one lane per sample. They got from 6.8 to 13.6 mio reads from each lane, which they assigned to genes. They were able to assign 30% to 50% of the tags unambiguously to annotated genes and produced a table that gives these counts.[3] A version of this table is distributed with the *DESeq* package as example data in a file called "TagSeqExample.tab". The `system.file` function allows to see where R has stored the file when the package was installed:

```
> library( DESeq )
> exampleFile = system.file( "extra/TagSeqExample.tab", package="DESeq" )
> exampleFile
```

```
[1] "/tmp/RtmpnnDcjl/Rinst12d08f69/DESeq/extra/TagSeqExample.tab"
```

It is a tab-delimited file with column headers in the first line. We read it in with

```
> countsTable <- read.delim( exampleFile, header=TRUE, stringsAsFactors=TRUE )
> head( countsTable )
```

```
        gene T1a T1b  T2  T3  N1  N2
1 Gene_00001   0   0   2   0   0   1
2 Gene_00002  20   8  12   5  19  26
3 Gene_00003   3   0   2   0   0   0
4 Gene_00004  75  84 241 149 271 257
5 Gene_00005  10  16   4   0   4  10
6 Gene_00006 129 126 451 223 243 149
```

To obtain such a table for your own data, you will need other software; this is out of the scope of DESeq. In the course materials from the Workshops section of the Bioconductor web page, you might find further information how to do this with the *ShortRead* and *IRanges* packages.

The first column is the gene ID. (We have shuffled the table rows, removed the RefSeq IDs and replaced them with dummy identifiers of the form "Gene_*NNNNN*".) We use the gene IDs for the row names and remove the gene ID column:

[3] An 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`.

```
> rownames( countsTable ) <- countsTable$gene
> countsTable <- countsTable[ , -1 ]
```

We are now left with six columns, referring to the six samples. The first four (labelled "T1a", "T1b", "T2", and "T3") are from cancerous tissue, the last two (labelled "N1", "N2") are from healthy tissue and served as control.

We code this information in the following vector, which assigns each sample a "condition":

```
> conds <- c( "T", "T", "T", "Tb", "N", "N" )
```

where "T" stands for a sample derived from a certain tumour type and "N" for a sample derived from non-pathological tissue. The first three samples had a very similar histopathological phenotype, while the fourth sample was atypical, and hence, we assign it another condition ("Tb").

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 the *eSet* class and so shares all features of this standard Bioconductor class. Furthermore, accessors are provided for its data slots. For example, the counts can be accessed with the `counts` function.

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

```
           T1a T1b  T2  T3  N1  N2
Gene_00001   0   0   2   0   0   1
Gene_00002  20   8  12   5  19  26
Gene_00003   3   0   2   0   0   0
Gene_00004  75  84 241 149 271 257
Gene_00005  10  16   4   0   4  10
Gene_00006 129 126 451 223 243 149
```

One feature derived from the *eSet* class is the possibility to subset. We can remove the first sample (i.e., the first column) as follows

```
> cds <- cds[ ,-1 ]
```

We remove it because samples T1a and T1b were derived from the same individuum and are hence more similar than the others. In order to keep the present example simple we continue without sample T1a.

As first processing step, we have to estimate the effective library size. This information is called the "size factors" vector, as the package only needs to now the relative library sizes. So, if a non-differentially expressed gene produces twice as many counts in one sample than in another, the size factor for this sample should be twice as large as the one for the other sample. You could simply use the actual total numbers of reads and assign them to the `cds` object:

```
> libsizes <- c( T1a=6843583, T1b=7604834, T2=13625570, T3=12291910,
+    N1=12872125, N2=10502656 )
> sizeFactors(cds) <- libsizes[-1]
```

However, one seems to get better results by estimating the size factors from the count data. The function `estimateSizeFactors` does that for you. (See the man page of `estimateSizeFactorsForMatrix` for technical details on the calculation.)

```
> cds <- estimateSizeFactors( cds )
> sizeFactors( cds )
```

```
      T1b        T2        T3        N1        N2
0.5587394 1.5823096 1.1270425 1.2869337 0.8746998
```

3 Variance estimation

As explained in detail in the paper, the core assumption of this method it that the mean is a good predictor of the variance, i.e., that genes with a similar expression level also have similar variance across replicates. Hence, we need to estimate for each condition a function that allows to predict the variance from the mean. This estimation is done by calculating, for each gene, the sample mean and variance within replicates and then fitting a curve to this data.

This computation is performed by the following command.

```
> cds <- estimateVarianceFunctions( cds )
```

In order to use the package, you do not need to know what precisely these raw variance functions estimate. For the interested reader, a few extra details are given here:

The point of the variance functions is to predict how much variance one should expect for counts at a certain level. For example, let us assume that we have found 123 tags for a certain gene in the "T1b" sample. We may now calculate the expected "raw variance" as follows. First we get the "base level", by which we mean this count value divided by the size factor. This makes the values from different columns comparable. Then, we insert this into the raw variance function to get the estimated "raw variance" which needs to be scaled up to the count level by multiplying with the size factor (squared, because this is a variance). Once we add the expected shot- noise variance (i.e., the variance due to the Poisson counting process), which is equal to the count value, we get the full variance. The square root of this full variance is then the estimated standard deviation for count values at the given level (provided, of course, that our fundamental assumption is right that the mean allows to get a reasonable prediction for the variance).

```
> countValue <- 123
> baseLevel <- countValue / sizeFactors(cds)["T1b"]
> rawVarFuncForGB <- rawVarFunc( cds, "T" )
> rawVariance <- rawVarFuncForGB( baseLevel )
> fullVariance <- countValue + rawVariance * sizeFactors(cds)["T1b"]^2
> sqrt( fullVariance )
```

```
     T1b
71.89861
attr(,"size")
[1] 2
```

Of course, you do not have to do the calculation just outlined yourself, the package does this automatically.

If you are confident that the package did a good job in estimating the variance functions, you may now skip directly to the Section 4. If you, however, would like to check whether the fit was good, the rest of this sections explains how to inspect and verify the variance function estimates.

The function 'scvPlot' shows all the base variance functions in one plot:

```
> scvPlot( cds, ylim=c(0,2) )
```

In the produced plot (Fig. 1), the x axis is the base mean, the y axis the squared coefficient of variation (SCV), i.e., the ratio of the variance at base level to the square of the base mean. The solid lines are the SCV for the raw variances, i.e., the noise due to biological replication. There is one coloured solid line per condition, and, in case there are non-replicated conditions, a dashed black line for the maximum of the raw variances, which is used for these.

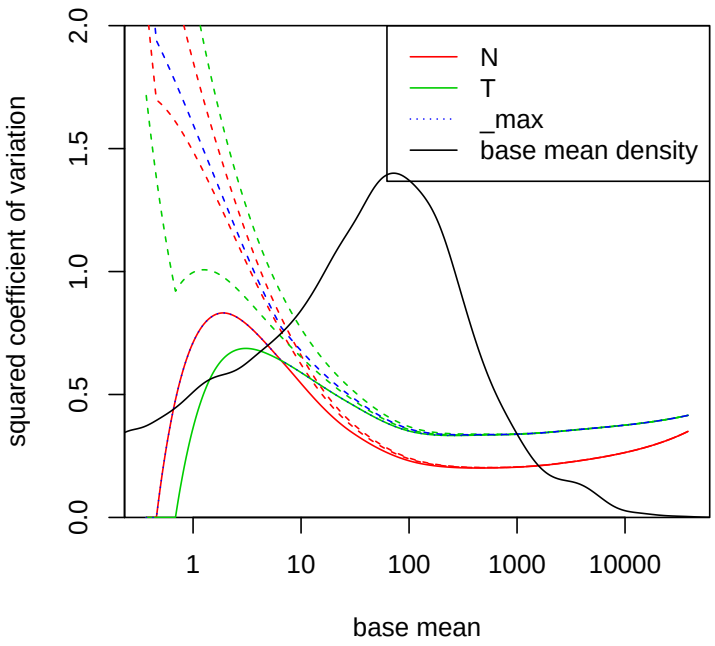

Figure 1: Plot to show the estimated variances (as squared coefficients of variation (SCV), i.e., variance over squared mean), produced with the function `scvPlot`.

On top of the variance, there is shot noise, i.e., the Poissonean variance inherent to the process of counting reads. The amount of shot noise depends on the size factor, and hence, for each sample, a dotted line in the colour of its condition is plotted above the solid line. The dotted line is the base variance, i.e., the full variance, scaled down to base level by the size factors. The vertical distance between solid and dotted lines is the shot noise.

The solid black line is a density estimate of the base means: Only were there is an appreciable number of base mean values, the variance estimates can be expected to be accurate.

For the condition "Tb", we cannot estimate a variance function as we have no replicates. When a variance estimate is needed for "Tb", the package will use the maximum of the variances estimated for all the other conditions. To see the assignment of conditions to variance functions, use the `rawVarFuncTable` accessor function:

```
> rawVarFuncTable( cds )

     N      T     Tb
   "N"    "T" "_max"
```

It is instructive to observe at which count level the biological noise starts to dominate the shot noise. At low counts, where shot noise dominates, higher sequencing depth (larger library size) will improve the signal-to-noise ratio while for high counts, where the biological noise dominates, only additional biological replicates will help.

One should check whether the base variance functions seem to follow the empirical variance well. To this end, two diagnostic functions are provided. The function `varianceFitDiagnostics` returns, for a specified condition, a data frame with four columns: the mean base level for each gene, the base variance as estimated from the count values of this gene only, and the fitted base variance, i.e., the predicted value from the local fit through the base variance estimates from all genes. As one typically has few replicates, the single-gene estimate of the base variance can deviate wildly from the fitted value. To see whether this might be too wild, the cumulative probability for this ratio of single-gene estimate to fitted value is calculated from the χ^2 distribution, as explained in the paper. These values are the fourth column.

```
> diagForT <- varianceFitDiagnostics( cds, "T" )
> head( diagForT )

              baseMean      baseVar fittedRawVar fittedBaseVar     pchisq
Gene_00001   0.6319876    0.7988166 6.319876e-09  7.652518e-01 0.69307480
Gene_00002  10.9508978   22.6740118 6.932106e+01  8.258113e+01 0.39971516
Gene_00003   0.6319876    0.7988166 6.319876e-09  7.652518e-01 0.69307480
Gene_00004 151.3237122    1.9415961 7.733836e+03  7.917068e+03 0.01249452
Gene_00005  15.5819200  340.8122533 1.297212e+02  1.485888e+02 0.87009670
Gene_00006 255.2670119 1771.2413710 2.171358e+04  2.202268e+04 0.22328186
```

We may now plot the per-gene estimates of the base variance against the base levels and draw a line with the fit from the local regression:

```
> smoothScatter( log10(diagForT$baseMean), log10(diagForT$baseVar) )
> lines( log10(fittedBaseVar) ~ log10(baseMean),
+    diagForT[ order(diagForT$baseMean), ], col="red" )
```

As one can see (Fig. 2), the fit (red line) follows the single-gene estimates well, even though the spread of the latter is considerable, as one should expect, given that each variance value is estimated from just three values.

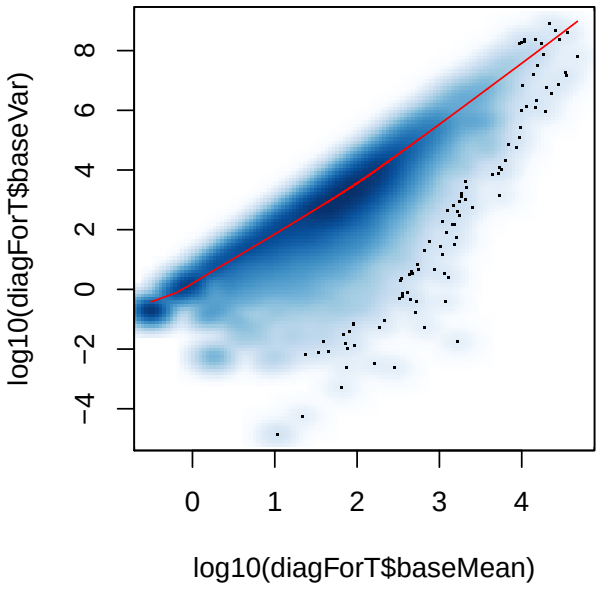

Figure 2: Diagnostic plot to check the fit of the variance function.

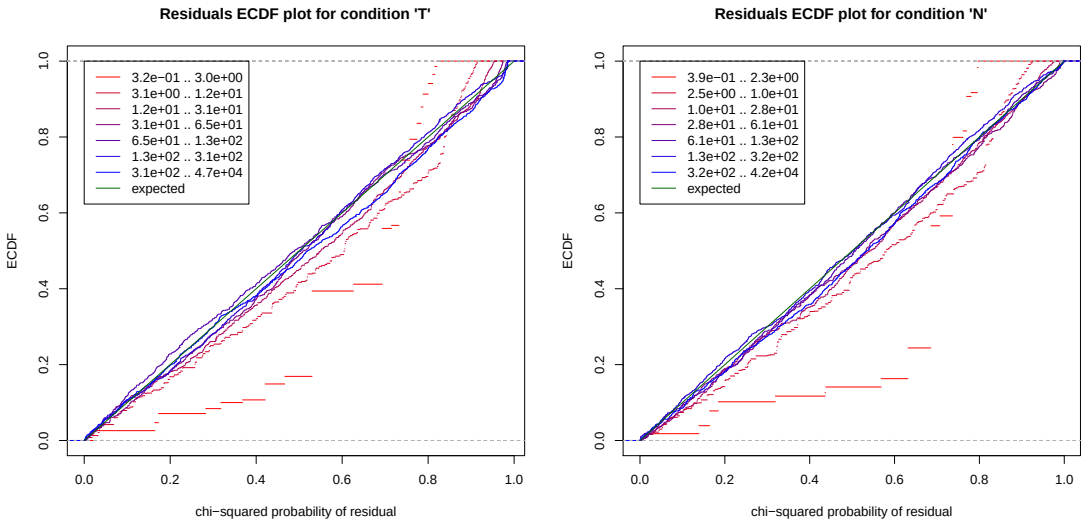

Figure 3: Another diagnostic plot to check the fit of the variance functions. This one is produced with the function `residualsEcdfPlot`.

Another way to study the diagnostic data is to check whether the probabilities in the fourth column of the diagnostics data frame are uniform, as they should be. One may simply look at the histogram of `diagForGB$pchisq` but a more convenient way is the function `residualsEcdfPlot`, which show empirical cumulative density functions (ECDF) stratified by base level. We look at them for the conditions "T" and "N":

```
> par( mfrow=c(1,2 ) )
> residualsEcdfPlot( cds, "T" )
> residualsEcdfPlot( cds, "N" )
```

Fig. 3 shows the output. In both cases, the ECDF curves follow the diagonal well, i.e., the fit is good. Only for very low counts (below 10), the deviations become stronger, but as at these levels, shot noise dominates, this is no reason for concern.

If in your data the residuals ECDF plot indicates problems with the fit, you may want to manually adjust the variance estimates. If the ECDF curves are below the green line, variance is underestimated, and if you test for differential expression (see next section) you will get too low p values (and hence, too many false positives). If the ECDF curves are above the green line, variance is overestimated, which leads to too high p values (and hence, an overestimation of the false discovery rate).

The first case (curves below the green line) may indicate a serious problem that might compromise your results. However, this seems to rarely happen (and I'd appreciate if you could sent me a mail if you observe it with real data so I can investigate). The second case (curves above the green line) is usually nothing to worry about; it only causes DESeq to be conservative with the tests.

4 Inference: Calling differential expression

Having estimated and verified the variance–mean dependence, 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 "N" and "T", we simply call the function `nbinomTest`. It performs the tests as described in the paper and returns a data frame with the p value and other useful data.

```
> res <- nbinomTest( cds, "N", "T" )
> head(res)
```

```
          id    baseMean   baseMeanA   baseMeanB foldChange log2FoldChange
1 Gene_00001   0.6018061   0.5716247   0.6319876  1.1055988      0.1448279
2 Gene_00002  16.5975136  22.2441294  10.9508978  0.4923051     -1.0223755
3 Gene_00003   0.3159938   0.0000000   0.6319876        Inf            Inf
4 Gene_00004 201.7601420 252.1965718 151.3237122  0.6000229     -0.7369106
5 Gene_00005  11.4261243   7.2703285  15.5819200  2.1432209      1.0997806
6 Gene_00006 217.4247729 179.5825340 255.2670119  1.4214468      0.5073601
       pval      padj    resVarA      resVarB
1 1.0000000 1.0000000 0.53327970 0.5219305089
2 0.4401904 0.9289183 0.68364266 0.1321576847
3 0.3302722 0.9025172 0.00000000 0.5219305099
4 0.4601667 0.9289183 0.38911331 0.0001390365
5 0.5816774 0.9642673 0.42627612 3.0290090161
6 0.5190449 0.9366699 0.01696371 0.1080731022
```

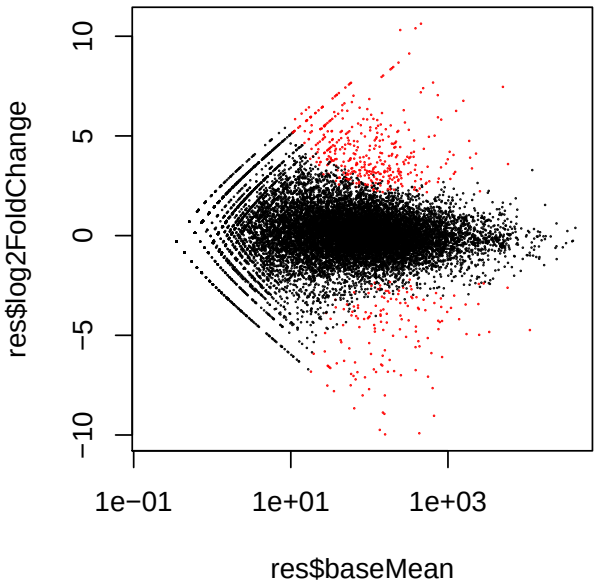

Figure 4: MvA plot for the contrast "T" vs. "N".

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 standard R function `p.adjust`), which controls false discovery rate (FDR). The last two columns show the ratio of the single gene estimates for the base variance to the fitted value. This may help to notice false hits due to "variance outliers". Any hit that has a very large value in one these two columns should be checked carefully.

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=.1,
+       col = ifelse( res$padj < .1, "red", "black" ) )
> plotDE( res )
```

See Fig. 4 for the plot. As we will use this plot more often, we have stored its code in a function.

We can filter for the significant genes,

```
> resSig <- res[ res$padj < .1, ]
```

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

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

```
              id  baseMean baseMeanA baseMeanB foldChange log2FoldChange
12236 Gene_12236 1314.2769 0.0000000 2628.5538        Inf            Inf
8420  Gene_08420  520.4418 0.0000000 1040.8835        Inf            Inf
10387 Gene_10387  844.3113 0.0000000 1688.6226        Inf            Inf
3806  Gene_03806  637.5960 0.0000000 1275.1920        Inf            Inf
4189  Gene_04189  261.0109 0.0000000  522.0217        Inf            Inf
17263 Gene_17263  453.0908 0.5716247  905.6100   1584.274       10.62961
              pval         padj      resVarA  resVarB
12236 2.940849e-21 5.408809e-17 0.000000e+00 23.22154
8420  2.352988e-20 2.163808e-16 0.000000e+00 23.20920
10387 4.964117e-20 3.043335e-16 0.000000e+00 23.34167
3806  4.092018e-19 1.881510e-15 0.000000e+00 23.29760
4189  1.174467e-17 4.320159e-14 0.000000e+00 22.74140
17263 6.374589e-17 1.954024e-13 1.584562e-05 23.11462
```

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
12457 Gene_12457 243.2076  486.4152         0          0           -Inf
16153 Gene_16153 230.3059  460.6119         0          0           -Inf
14803 Gene_14803 140.1458  280.2916         0          0           -Inf
3664  Gene_03664 138.2713  276.5426         0          0           -Inf
6705  Gene_06705 136.3325  272.6650         0          0           -Inf
429   Gene_00429 113.0759  226.1519         0          0           -Inf
              pval         padj     resVarA resVarB
12457 2.308952e-10 1.179618e-07 18.39362665       0
16153 2.534270e-09 9.322057e-07 10.98091651       0
14803 1.880220e-08 5.239545e-06  8.25267222       0
3664  4.937565e-09 1.539181e-06  9.44914573       0
6705  4.174475e-08 1.037526e-05 32.65275547       0
429   3.086259e-08 7.994714e-06  0.05303476       0
```

or at the most strongly up-regulated ones:

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

```
              id  baseMean baseMeanA baseMeanB foldChange log2FoldChange
12236 Gene_12236 1314.2769         0 2628.5538        Inf            Inf
10387 Gene_10387  844.3113         0 1688.6226        Inf            Inf
3806  Gene_03806  637.5960         0 1275.1920        Inf            Inf
8420  Gene_08420  520.4418         0 1040.8835        Inf            Inf
11756 Gene_11756  269.7254         0  539.4509        Inf            Inf
4189  Gene_04189  261.0109         0  522.0217        Inf            Inf
              pval         padj resVarA  resVarB
12236 2.940849e-21 5.408809e-17       0 23.22154
```

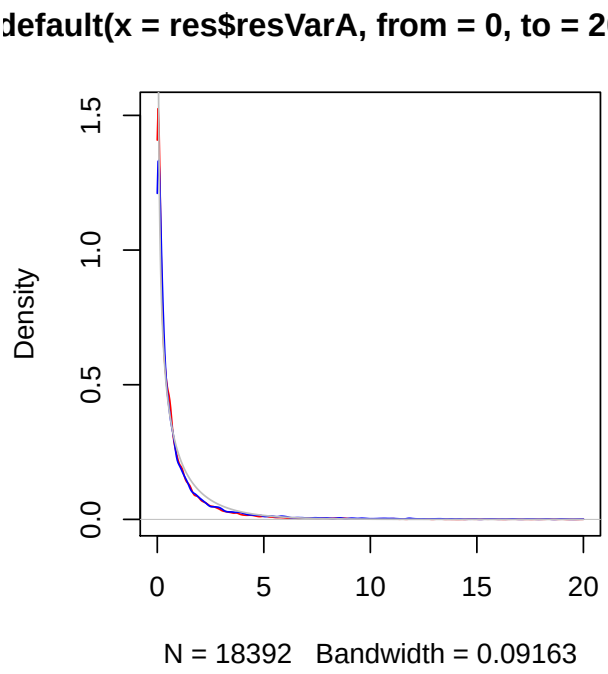

Figure 5: Density of residual variance ratios.

```
10387 4.964117e-20 3.043335e-16        0 23.34167
3806  4.092018e-19 1.881510e-15        0 23.29760
8420  2.352988e-20 2.163808e-16        0 23.20920
11756 7.271243e-16 1.485919e-12        0 22.35437
4189  1.174467e-17 4.320159e-14        0 22.74140
```

The test is based on the assumption that the fitted variance, i.e., the variance as deduced from the mean vie the raw variance functions, is a good estimate for a gene's true variance. We have tested the appropriateness of this approach above with the plot produced by `residualsEcdfPlot` and concluded that it seems to hold well for most genes. The `res` object gives us two columns to have a closer look at this, namely `resVarA` and `resVarB`. These contain the residual variance quotients, i.e. the ratio of the variance as calculated only from the counts for the gene under consideration to the fitted variance.

We can plot the density of these ratios (Fig. 5):

```
> plot( density( res$resVarA, na.rm=TRUE, from=0, to=20 ), col="red" )
> lines( density( res$resVarB, na.rm=TRUE, from=0, to=20 ), col="blue" )
> xg <- seq( 0, 20, length.out=1000 ); lines( xg, dchisq( xg, df=1 ), col="grey" )
```

The first two lines estimate the density of the quotients for conditions A and B and plot them in red and blue. If the model holds, these should agree with a χ^2 distribution with 1 degree of freedom (we have two replicates for each condition, and the number of degrees of freedom is one less than the number of replicates). The third line adds the theoretical density function

in grey. The fact that the curves agree well is not surprising; we have seen this already in the residual ECDF plots (which show the same information, but in a way that makes it easier to see deviations).

We can also see that hardly any genes have a ratio exceeding, say, 20. In fact, there are, however, a few such genes, but we cannot see them in a density plot:

```
> table( res$resVarA > 15 | res$resVarB > 15)

FALSE  TRUE
18186   206
```

From the chi^2 distribution, we expect such high ratios to only occur for maybe two genes:

```
> ( 1 - pchisq( 15, df=1) ) * nrow(counts(cds))

[1] 2.01691
```

Hence, these genes seem to be "variance outliers", and it may be prudent to exclude them from the list of significant hits. (Of course, the threshold of 15 was chosen ad hoc here and other thresholds of the same order of magnitude would be defensible as well.)

4.1 Working partially without replicates

If you have replicates for one condition but not for the other, you can still proceed as before. As already stated above, the testing function will simply take the maximum of all estimated variance function for conditions without replicates. If we consider this acceptable, we can contrast the single "Tb" sample against the two "N" samples.

```
> resTbvsN <- nbinomTest( cds, "N", "Tb" )
```

We produce the same plot as before, again with

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

The result (Fig. 6) shows the same symmetry in up- and down-regulation as in Fig. 4 but a striking asymmetry in the boundary line for significance. This has an easy explanation: low counts suffer from proportionally stronger shot noise than high counts, and this is more pronounced in the "Tb" data than in the "N" data due to the lack of replicates. Hence a stronger signal is required to call a down-regulation significant than for an up-regulation.

4.2 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 would one know that this difference is due to the different conditions and would not have arisen between replicates, as well, just due to noise? Hence, any attempt to work without any replicates will lead to conclusions of very limited reliability.

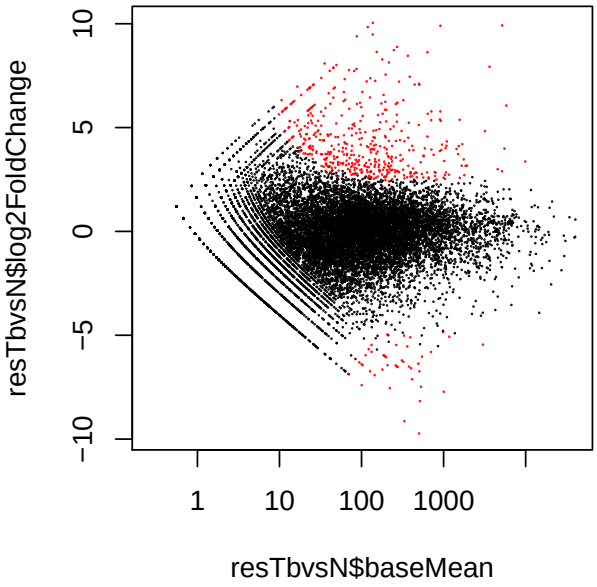

Figure 6: MvA plot for the contrast “Tb” vs. “N”.

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

Our primary assumption is still that the mean is a good predictor for the variance. Hence, if a number of genes with similar expression level are compared between replicates, we expect that their variation is of comparable magnitude. 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 variance estimated from comparing their count rates *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 change too much due to the influence of the hopefully few differentially expressed genes. Furthermore, the differentially expressed genes will only cause the variance estimate to be too high, so that the test will err to the side of being too conservative, i.e., we only lose power.

We shall now see how well this works for our example data, even though it has rather many differentially expressed genes.

We reduce our count data set to just two columns, one "T" and one "N" sample:

```
> cds2 <- cds[ ,c( "T1b", "N1" ) ]
```

Now, without any replicates at all, the `estimateVarianceFunctions` 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 <- estimateVarianceFunctions( cds2, method="blind" )
```

Now, we can attempt to find differential expression:

```
> res2 <- nbinomTest( cds2, "N", "T" )
```

Unsurprisingly, we find much fewer hits, as can be seen from the plot (Fig. 7)

```
> plot(
+    res2$baseMean,
+    res2$log2FoldChange,
+    log="x", pch=20, cex=.1,
+    col = ifelse( res2$padj < .1, "red", "black" ) )
```

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 15533    70 15603
  TRUE    414   201   615
  Sum   15947   271 16218
```

As can be seen, we have still found about 1/5 of the hits, and only a reassuringly small number of new (and potentially false) hits.

One may finally ask whether the reduction of discoveries to a quarter is due to the higher variance estimate, or due to the lower confidence in the base mean estimates, which is due to

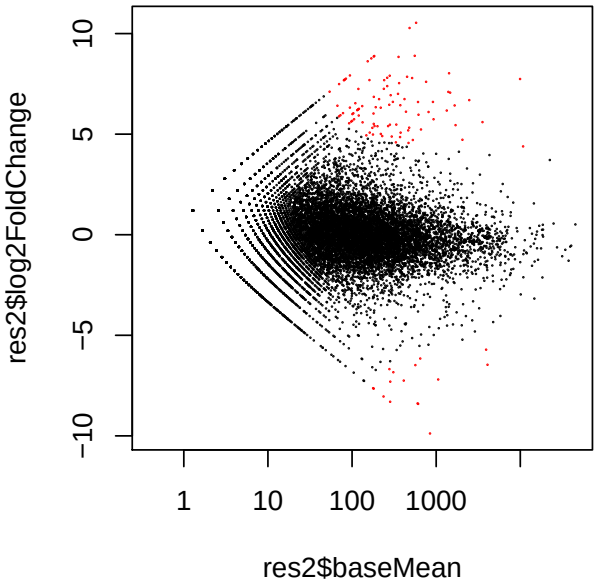

Figure 7: MvA plot for the contrast “T” vs. “N”, from a test using no replicates.

the reduced sample size. To see this, we run the original analysis again, but now using the new, worse variance function. As this analysis is outside the standard work-flow, we have to use a lower-level function of the package, which does not recognise the *CountDataSet* S4 object but instead expects all data to be specified separately. (This is inconvenient normally but enables us here to substitute another variance function for the one that the `nbinomTest` function would use.)

```
> colsN <- conditions(cds) == "N"
> colsT <- conditions(cds) == "T"
> baseMeansNT <- getBaseMeansAndVariances(
+    counts(cds)[ , colsN|colsT ],
+    sizeFactors(cds)[ colsN|colsT ] )$baseMean
> pvals2b <- nbinomTestForMatrices(
+    counts(cds)[ ,colsN ],
+    counts(cds)[ ,colsT ],
+    sizeFactors(cds)[ colsN ],
+    sizeFactors(cds)[ colsT ],
+    rawVarFunc( cds2, "_blind", TRUE )( baseMeansNT ),
+    rawVarFunc( cds2, "_blind", TRUE )( baseMeansNT ) )
```

We adjust the p values and then compare the hits with those found originally:

```
> padj2b <- p.adjust( pvals2b, method="BH" )
> notNAinRes2 <- !is.na( res2$padj )
> addmargins( table(
+    res_sig = res$padj[notNAinRes2] < .1,
+    res2b_sig = padj2b[notNAinRes2] < .1 ) )
```

```
       res2b_sig
res_sig FALSE  TRUE   Sum
  FALSE  2419 13184 15603
  TRUE     99   516   615
  Sum    2518 13700 16218
```

(There are a few genes with NA as p value in `res2` because all counts in the restricted data set were zero. We have excluded these to make the tables comparable.)

In conclusion, the worse variance estimates costs less in power than the reduction in sample size. For another data set, this may well be quite different.

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

In Section 4 we have seen how to use *DESeq* for calling differentially expressed genes. For each gene, *DESeq* reports a (log) 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 (log) 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 (log) 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 $\log_2\left((n_A+c)/(n_B+c)\right)$, 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 between-replicate variability of the data, and no *ad hoc* choice of c, as above, is necessary.

```
> vsd <- getVarianceStabilizedData( cds )
```

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

```
> mod_lfc = (rowMeans( vsd[, conditions(cds)=="T"] ) -
+            rowMeans( vsd[, conditions(cds)=="N"] ))
```

Now let us compare these to the original (log) 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>
 977 1480  368    0
```

For plotting (Figure 8), 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=16,
+       col = ifelse(finite, "#80808040", "red"))
> abline(a=0, b=1, col="#40404040")
```

These data are now approximately homoscedastic and hence suitable as input to a sample to sample distance calculation,

```
> dists <- dist( t( vsd ) )
```

which we can visualize as a heatmap in Figure 9.

```
> heatmap(as.matrix( dists ),
+         symm=TRUE,
+         scale="none",
+         col = colorRampPalette(c("darkblue","white"))(100))
```

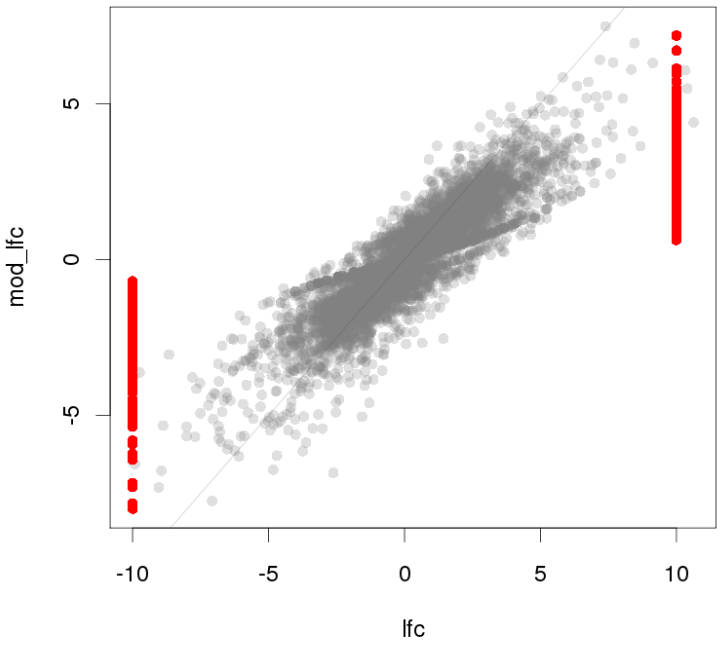

Figure 8: Scatterplot of direct (`lfc`) versus *moderated* log-ratios (`moderated_lfc`). The moderation criterion used is variance stabilisation. The red points correspond to values that were infinite in `lfc` and were abitrarily set to 10 for the purpose of plotting. These values vary in a finite range (as shown in the plot) in `moderated_lfc`.

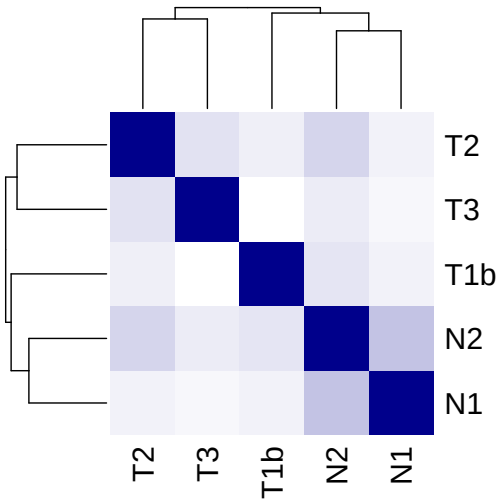

Figure 9: Heatmap showing the Euclidean distances between the samples as calculated from the variance-stabilising transformation of the count data.

We can also visualise the expression data of, say, the top 100 differentially expressed genes.

```
> select = order(res$pval)[1:100]
> 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 10.

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

A Reference overview

This appendix gives a terse overview of the class and all the functions defined in the package. The description assumed that the reader is familiar with the `eSet` class.

The package defines one S4 class, *CountDataSet*, which is derived from *eSet*. To instantiate an object, use the function `newCountDataSet`. Do not call `new` directly.

The class's `assayData` is a locked environment containing a single object, namely the matrix `counts` with the count data. The `featureData` is not used internally, but the user may wish to store annotation there. The `phenoData` contains two columns, `_sizeFactors` and `_conditions`

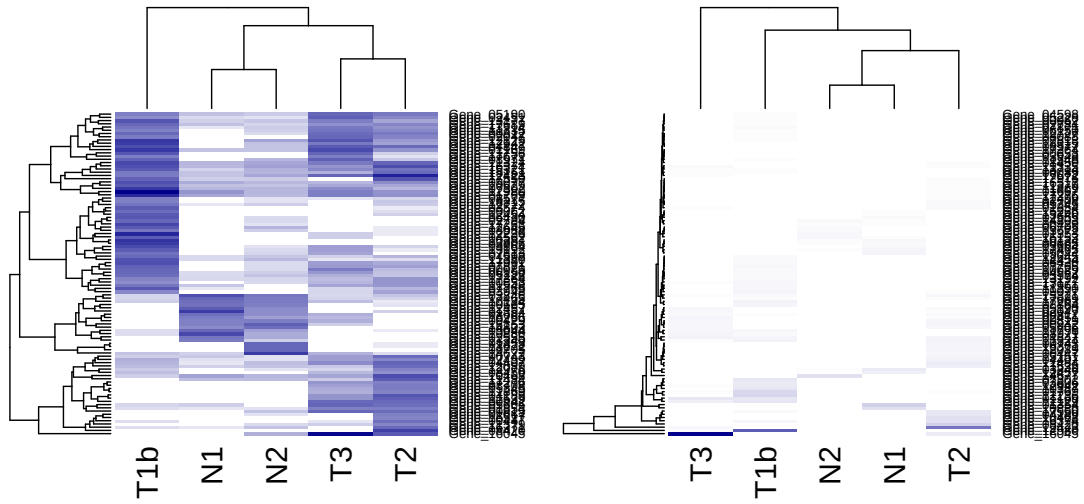

Figure 10: Heatmaps showing the expression data of the top 100 differentially expressed genes. Left: on a colour scale proportional to the variance stabilisation transformed scale (`vsd`), right: colour scale proportional to the original count scale (`countscds`). The right plot is dominated by a small number of data points with large values.

to hold the vector of size factors and the factor of conditions. The user may add further columns for annotation.

Furthermore, there is a slot `rawVarFuncs` of type environment that holds the raw variance functions. Each of these functions has a name to access it in the environment, which is either the name of a condition, or `"_max"` for conditions without replicates in normal mode, or `"_blind"` or `"_pooled"` for the single estimate that `estimateVarianceFunction` produces when called with `method="blind"` or or `method="pooled"`. Finally, the slot `rawVarFuncTable` contains a charcter vector which serves as a look-up table. The names are the conditions and the values are the function names, i.e., the hash keys for the `rawVarFuncs` environment.

All these properties are checked by the validity method.

The following slot accessors are provided: `counts`, `sizeFactors`, `conditions`, `rawVarFunc`, `rawVarFuncTable`. To avoid accidental invalidation, a setter is provided only for `sizeFactors`. (The other slots may be change via the "`@`" syntax, but only at the user's risk.)

All functions that perform actual calculations are offered in two variants: a "core" one that takes base types as explicit arguments, and a wrapper that takes a *CountDataObject* and finds the data there itself. As these function pairs have rather different argument lists, they are not made as generic functions, but rather have two different names, as follows:

Purpose	Wrapper function	Core function
estimate size factors	`estimateSizeFactors`	`estimateSizeFactorsForMatrix`
estimate variance functions	`estimateVarianceFunctions`	`estimateVarianceFunctionForMatrix`
calculate base means and variances	N/A	`getBaseMeansAndVariances`
get diagnostics for variance fit	`varianceFitDiagnostics`	`varianceFitDiagnosticsForMatrix`
produce ECDF plot for variance residuals	`residualsEcdfPlot`	`residualsEcdfPlotFromDiagnostics`
perform test	`nbinomTest`	`nbinomTestForMatrices`

B Session Info

```
> sessionInfo()

R version 2.13.0 (2011-04-13)
Platform: x86_64-unknown-linux-gnu (64-bit)

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

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

other attached packages:
[1] DESeq_1.4.1     locfit_1.5-6    lattice_0.19-23 akima_0.5-4
[5] Biobase_2.12.1

loaded via a namespace (and not attached):
 [1] AnnotationDbi_1.14.1 DBI_0.2-5           KernSmooth_2.23-4
 [4] RColorBrewer_1.0-2   RSQLite_0.9-4       annotate_1.30.0
 [7] genefilter_1.34.0    geneplotter_1.30.0  grid_2.13.0
[10] splines_2.13.0       survival_2.36-8     tools_2.13.0
[13] xtable_1.5-6
```

References

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