Analysing RNA-Seq data with the "DESeq" package

Simon Anders

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany

`sanders@fs.tum.de`

2010-01-19

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 Quick start

This first section just shows the commands necessary for an analysis at a glance. For a more gentle introduction, 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.

Important: Each column must stem from an independent experiment or sample. If you spread sample material from one experiment over several "lanes" of the sequencer in order to get better coverage, you must sum up the counts from the lanes to get a single column. Failing to do so will result in incorrect variance estimation and overly optimistic p values

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
```

---

[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". This paper is currently under review. A preprint can be obtained from Nature Preceedings: `http://dx.doi.org/10.1038/npre.2010.4282.1`

```
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  T2 N  N
Levels: N T T2
```

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 1.18159527 1.04376733
2 0.5377309 0.9155028 0.70164236 0.64357934
3 0.3591742 0.8604077 1.79618706 0.00000000
4 0.2395555 0.7574043 0.06300499 0.48377564
5 0.7037074 0.9741942 1.88356930 0.42935025
6 0.5776440 0.9260629 0.19754700 0.01728059
```

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. 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/Rinst1401872080/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:

```
> 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", "T2", "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 ("T2").

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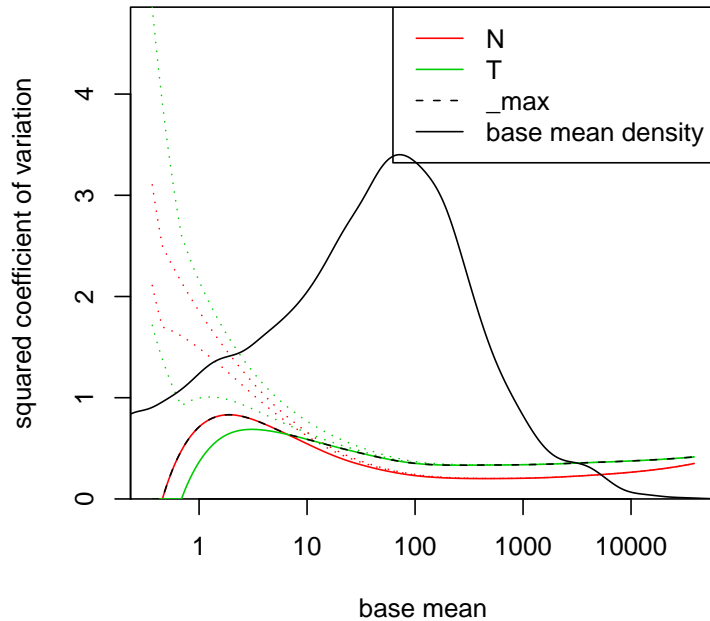
Figure 1: Plot to show the estimated variances (as squared coefficients of variation (SCV), i.e., variance over squared mean), produced with the function `scvPlot`.

```
> 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
```

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 )
```

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.

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.

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.

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" )
```
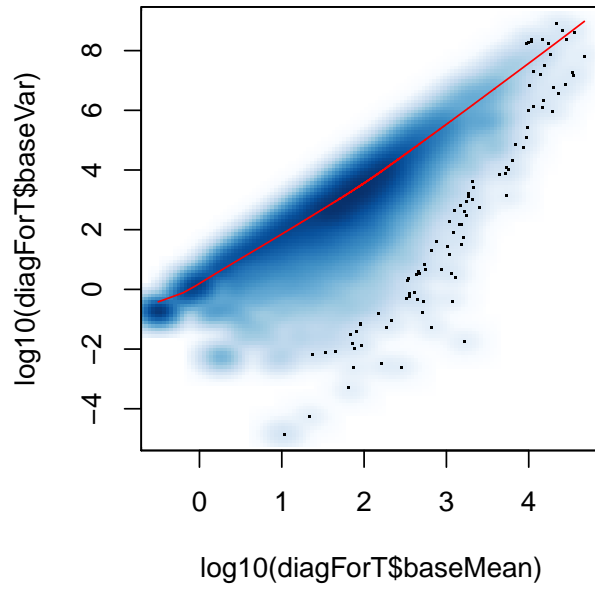
6

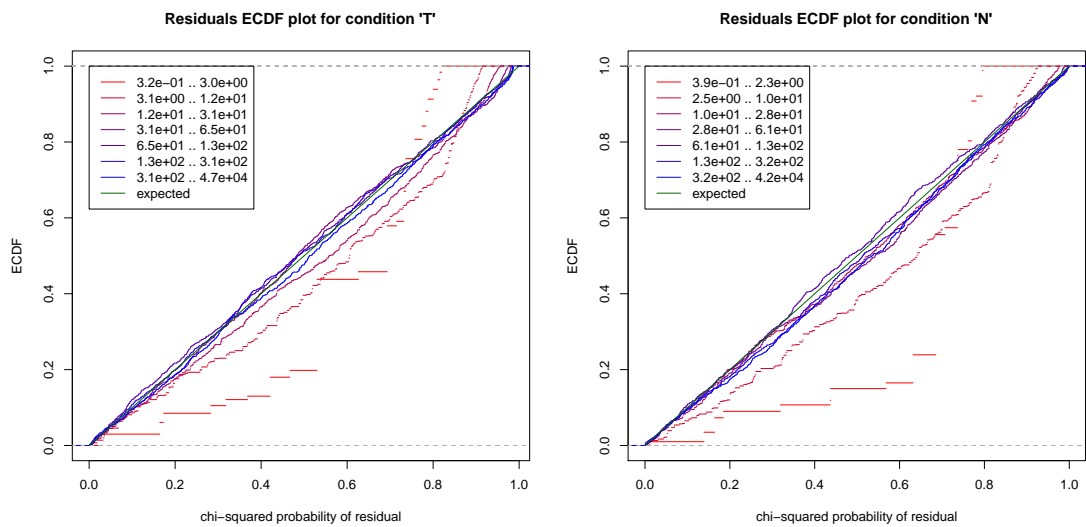Figure 2: Diagnostic plot to check the fit of the variance function.

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

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.

For the condition "T2", we cannot estimate a variance function as we have no replicates. When a variance estimate is needed for "T2", 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    T2
  "N"   "T" "_max"
```

4 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 0.93439523 1.2639751173
2 0.4507755 0.9303373 0.78135345 0.1453474989
3 0.2477559 0.8158866 0.00000000 1.2639751230
4 0.4553225 0.9303373 0.39952892 0.0001412121
5 0.6033328 0.9671820 0.46451459 3.7715992031
6 0.5199256 0.9352163 0.01724698 0.1105204434
```

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".

Let us first plot the $\log_2$ fold changes against the base means, colouring in red those genes that are significant at 10% FDR.
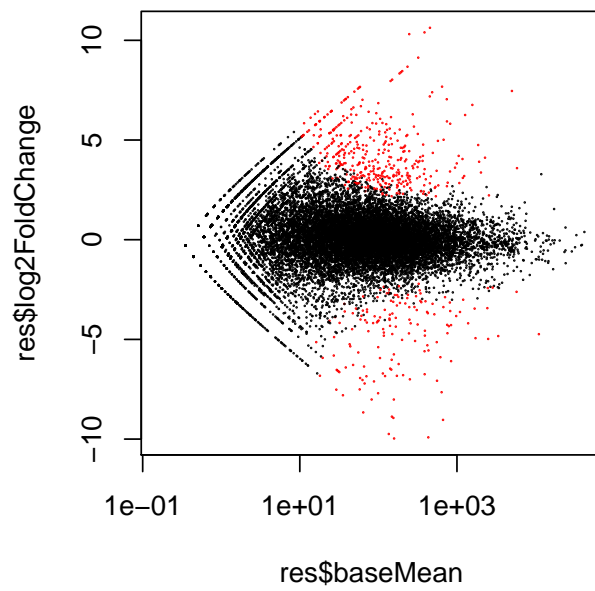
```
> plot(
+     res$baseMean,
```

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

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

See Fig. 4 for the plot.

Finally, we may 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  0.000000 2628.5538       Inf            Inf
8420  Gene_08420  520.4418  0.000000 1040.8835       Inf            Inf
10387 Gene_10387  844.3113  0.000000 1688.6226       Inf            Inf
1769  Gene_01769 1270.5469 32.845739 2508.2481   76.36449       6.254830
17550 Gene_17550  450.9093  6.149391  895.6692  145.65169       7.186379
12942 Gene_12942  646.1424  6.287872 1285.9969  204.52022       7.676100
            pval          padj      resVarA    resVarB
12236 2.930102e-21 5.389044e-17 0.000000e+00 23.368365
8420  2.344410e-20 2.155919e-16 0.000000e+00 23.583001
10387 4.953891e-20 3.037066e-16 0.000000e+00 23.573305
1769  8.893126e-20 4.089060e-16 1.267509e-03 20.768651
17550 2.051616e-19 7.546663e-16 2.468740e-05  1.132551
12942 2.914633e-19 8.934322e-16 9.390478e-04 21.398063
```

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 6.042413e-11 2.849540e-08 19.04949091       0
16153 8.930491e-10 3.220580e-07 11.39214536       0
14803 8.021482e-09 2.201957e-06  8.72355050       0
3664  2.063105e-09 6.431292e-07  9.99408382       0
6705  1.753820e-08 4.300833e-06 34.55699339       0
429   1.031462e-08 2.789802e-06  0.05660231       0
```

or at the most strongly up-regulated ones:

```
> head( resSig[ order( -resSig$foldChange, -resSig$baseMean ), ] )
```
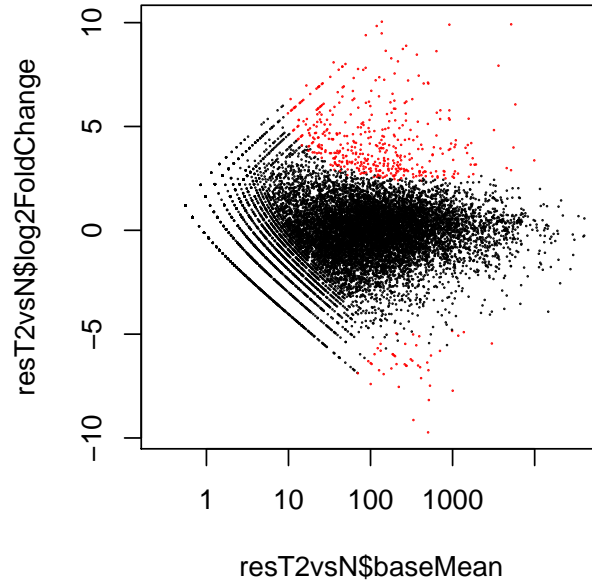
Figure 5: MvA plot for the contrast "T2" vs. "N".

	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	2.930102e-21	5.389044e-17	0	23.36836
10387	4.953891e-20	3.037066e-16	0	23.57331
3806	4.081541e-19	1.072396e-15	0	23.60416
8420	2.344410e-20	2.155919e-16	0	23.58300
11756	7.275628e-16	8.363334e-13	0	23.04616
4189	1.171298e-17	2.154252e-14	0	23.46816

5   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 "T2" sample against the two "N" samples.

11

```
> resT2vsN <- nbinomTest( cds, "N", "T2" )
```

We produce the same plot as before, again with

```
> plot(
+    resT2vsN$baseMean,
+    resT2vsN$log2FoldChange,
+    log="x", pch=20, cex=.1,
+    col = ifelse( resT2vsN$padj < .1, "red", "black" ) )
```

The result (Fig. 5) 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 "T2" 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.

6 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.

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 pooled over all samples:

```
> cds2 <- estimateVarianceFunctions( cds2, pool=TRUE )
```

Now, we can attempt to find differential expression:

```
> res2 <- nbinomTest( cds2, "N", "T" )
```
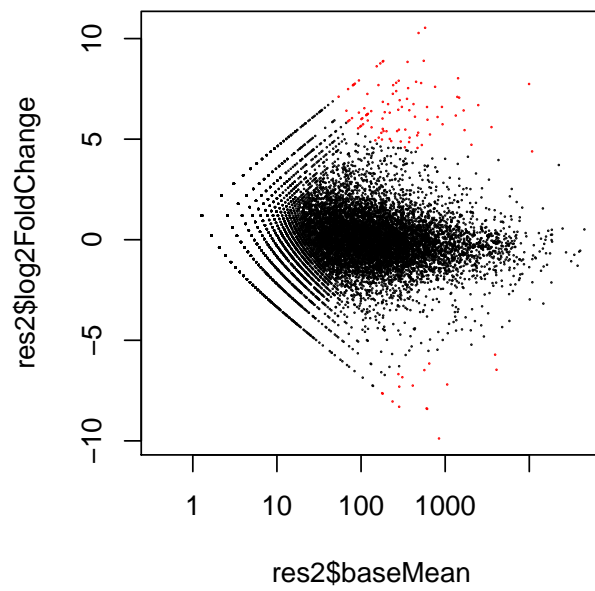
12

Figure 6: MvA plot for the contrast "T" vs. "N", from a test using no replicates.

Unsurprisingly, we find much fewer hits, as can be seen from the plot (Fig. reffigDE2)

```
> 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:

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

        res2_sig
res_sig FALSE  TRUE
  FALSE 15534    68
  TRUE    413   203
```

As can be seen, we have still found about 1/4 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 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, "_pooled" )( baseMeansNT ),
+     rawVarFunc( cds2, "_pooled" )( baseMeansNT ) )
```

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

```
> padj2b <- p.adjust( pvals2b, method="BH" )
> table( res_sig = res$padj < .1, res2b_sig = padj2b < .1 )

       res2b_sig
res_sig FALSE  TRUE
  FALSE  4335 13379
  TRUE    143   535
```

In conclusion, the worse variance estimates costs very roughly half of the hits, and the reduces sample size then another half, leaving us with a quarter of the original hits.

7 Sample clustering

Another feature of the *DESeq* package is the variance stabilising transformation (VST). This is a monotonous function, which is calculated for each sample from the applicable variance function, that transforms the count data such that its variance becomes independent of the mean, i.e., it produces a homoscedastic version of the data.

To demonstrate a potential use for this, we instantiate another CountDataSet and do not remove sample T1a this time:

```
> cds3 <- newCountDataSet( countsTable, conds )
> cds3 <- estimateSizeFactors( cds3 )
> cds3 <- estimateVarianceFunctions( cds3 )
```

Now, we call the function `getVarianceStabilizedData`,

```
> vsd <- getVarianceStabilizedData( cds3 )
```

which returns a matrix of non-integer data:

```
> head( vsd )
```

	T1a	T1b	T2	T3	N1	N2
Gene_00001	0.01150873	0.01146355	2.084739	0.01181223	0.01189542	1.57173044
Gene_00002	4.18267091	3.22195921	3.509000	2.73580879	3.98184460	4.41989938
Gene_00003	2.32432765	0.01146355	2.084739	0.01181223	0.01189542	0.01168536
Gene_00004	5.92630074	6.19054084	7.192879	6.65564740	7.47863203	7.62273474
Gene_00005	3.41710310	3.97581593	2.581017	0.01181223	2.56170425	3.37836210
Gene_00006	6.72231079	6.79510415	8.140204	7.25465684	7.31420225	6.79708707

This data is now approximately homoscedastic and hence suitable as input to various statistical procedures, e.g., a simple distance calculation:

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

We visualise the distance matrix as a heatmap:

```
> heatmap( as.matrix( dists ), symm=TRUE )
```

From the output (Fig. 7), we can see, e.g., that the two samples T1a and T1b that were derived from the same patient are indeed quite similar, and that the sample T3 with the atypical histopathological phenotype is not that different from sample T2.

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` to hold the vector of size factors and the factor of conditions. The user may add further columns for annotation.
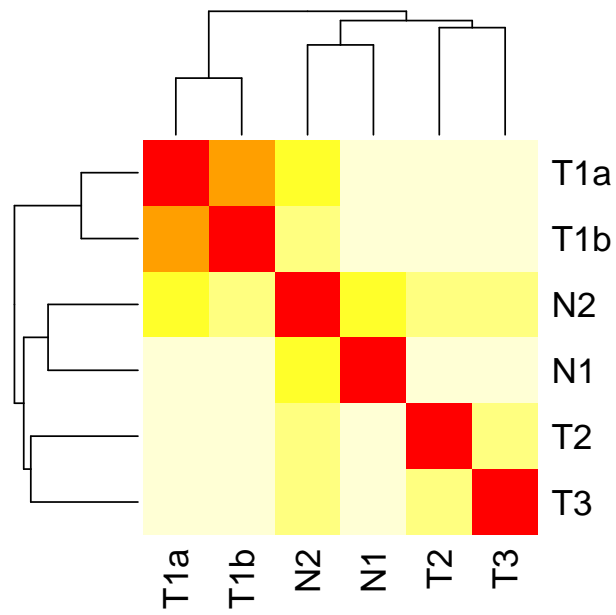
Figure 7: A heatmap showing the distances between the samples as calculated from the variance-stabilising transformation of the count data.

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 `"_pooled"` for the single pooled estimate that `estimateVarianceFunction` produces when called with `pool=TRUE`. Finally, the slot `rawVarFuncTable` contains a named character vector that 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 2.11.1 (2010-05-31)
x86_64-unknown-linux-gnu

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

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

other attached packages:
[1] DESeq_1.0.6    locfit_1.5-6    lattice_0.19-11 akima_0.5-4
[5] Biobase_2.8.0

loaded via a namespace (and not attached):
 [1] annotate_1.26.1     AnnotationDbi_1.10.2 DBI_0.2-5
 [4] genefilter_1.30.0   geneplotter_1.26.0  grid_2.11.1
 [7] KernSmooth_2.23-3   RColorBrewer_1.0-2  RSQLite_0.9-2
[10] splines_2.11.1      survival_2.35-8     xtable_1.5-6
```