Differential analysis of count data – the DESeq2 package

Michael Love[1*], Simon Anders[2], Wolfgang Huber[2]

[1] Max Planck Institute for Molecular Genetics, Berlin, Germany;

[2] European Molecular Biology Laboratory (EMBL), Heidelberg, Germany

[*]michaelisaiahlove (at) gmail.com

July 15, 2013

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 sequence fragments that have been assigned to each gene. Analogous data also arise for other assay types, including comparative ChIP-Seq, HiC, shRNA screening, mass spectrometry. An important analysis question is the quantification and statistical inference of systematic changes between conditions, as compared to within-condition variability. The package *DESeq2* provides methods to test for differential expression by use of negative binomial generalized linear models; the estimates of dispersion and logarithmic fold changes incorporate data-driven prior distributions [1]. This vignette explains the use of the package and demonstrates typical work flows.

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

Contents

1 Quick start

Here we show the most basic steps for a differential expression analysis. These steps imply you have a *SummarizedExperiment* object `se` with a column `condition`.

```
dds <- DESeqDataSet(se = se, design = ~ condition)
dds <- DESeq(dds)
res <- results(dds)
```

2 Input data

2.1 Why raw counts?

As input, the *DESeq2* package expects count data as obtained, e. g., from RNA-Seq or another high-throughput sequencing experiment, in the form of a matrix of integer values. The value in the i-th row and the j-th column of the matrix tells how many reads have been mapped to gene i in sample j. Analogously, for other types of assays, the rows of the matrix might correspond e. g. to binding regions (with ChIP-Seq) or peptide sequences (with quantitative mass spectrometry).

The count values must be raw counts of sequencing reads. This is important for *DESeq2*'s statistical model to hold, as only the actual counts allow assessing the measurement precision correctly. Hence, please do not supply other quantities, such as (rounded) normalized counts, or counts of covered base pairs – this will only lead to nonsensical results.

2.2 SummarizedExperiment input

In the *DESeq2* package, in order to simplify the preparation of a count matrix, we attempt a closer integration with the core Bioconductor package *GenomicRanges*. This should facilitate preparation steps and also downstream exploration of results. For counting aligned reads in genes, the `summarizeOverlaps` function of *GenomicRanges*/*Rsamtools* with `mode="Union"` is encouraged, resulting in a *SummarizedExperiment* object (*easyRNASeq* is another Bioconductor package which can prepare *SummarizedExperiment* objects as input for *DESeq2*). An example of the steps to produce a *SummarizedExperiment* can be found in the data package *parathyroidSE*, which summarizes RNA-Seq data from experiments on 4 human cell cultures [1].

```
library("parathyroidSE")
data("parathyroidGenesSE")
se <- parathyroidGenesSE
colnames(se) <- colData(se)$run
```

The class used by *DESeq2* is *DESeqDataSet*, which differs from *SummarizedExperiment* in having an associated design *formula*. The design formula expresses the variables which will be used in modeling. The formula should be a tilde ($\sim$) followed by the variables with plus signs between them (it will be coerced into an *formula* if it is not already). An intercept is automatically included, representing the base mean of counts. In order to benefit from the default settings of the package, you should put the variable of interest at the end of the formula. The constructor function below shows generation of a *DESeqDataSet* from a *SummarizedExperiment* `se`.

```
library("DESeq2")
ddsGR <- DESeqDataSet(se = se, design = ~ patient + treatment)
colData(ddsGR)$treatment <- factor(colData(ddsGR)$treatment,
                                   levels=c("Control","DPN","OHT"))
ddsGR
```

```
class: DESeqDataSet
dim: 60620 27
exptData(1): MIAME
assays(1): counts
rownames(60620): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
rowData metadata column names(0):
colnames(27): SRR479052 SRR479053 ... SRR479077 SRR479078
colData names(9): fileName run ... study sample
```

2.3 Count matrix input

Alternatively, if you already have prepared a matrix of read counts, you can use the function `DESeqDataSetFromMatrix`. For this function you should provide the counts matrix, the column information as a *DataFrame* or *data.frame* and the design formula.

```
library("pasilla")
data("pasillaGenes")
countData <- counts(pasillaGenes)
colData <- pData(pasillaGenes)[,c("condition","type")]
dds <- DESeqDataSetFromMatrix(countData = countData,
                              colData = colData,
                              design = ~ condition)
colData(dds)$condition <- factor(colData(dds)$condition,
                                 levels=c("untreated","treated"))
dds
```

```
class: DESeqDataSet
dim: 14470 7
```

```
 exptData(0):
 assays(1): counts
 rownames(14470): FBgn0000003 FBgn0000008 ... FBgn0261574 FBgn0261575
 rowData metadata column names(0):
 colnames(7): treated1fb treated2fb ... untreated3fb untreated4fb
 colData names(2): condition type
```

```
detach(package:pasilla)
detach(package:DESeq)
```

2.4 HTSeq input

If you have used the *HTSeq* python scripts, you can use the function `DESeqDataSetFromHTSeqCount`. For an example of using the python scripts, see the *pasilla* or *parathyroid* data package.

```
library("pasilla")
directory <- system.file("extdata", package="pasilla", mustWork=TRUE)
sampleFiles <- grep("treated",list.files(directory),value=TRUE)
sampleCondition <- sub("(.*treated).*","\\1",sampleFiles)
sampleTable <- data.frame(sampleName = sampleFiles,
                          fileName = sampleFiles,
                          condition = sampleCondition)
ddsHTSeq <- DESeqDataSetFromHTSeqCount(sampleTable = sampleTable,
                                       directory = directory,
                                       design= ~ condition)
colData(ddsHTSeq)$condition <- factor(colData(ddsHTSeq)$condition,
                                      levels=c("untreated","treated"))
ddsHTSeq
```

```
 class: DESeqDataSet
 dim: 70467 7
 exptData(0):
 assays(1): counts
 rownames(70467): FBgn0000003:001 FBgn0000008:001 ... _lowaqual
   _notaligned
 rowData metadata column names(0):
 colnames(7): treated1fb.txt treated2fb.txt ... untreated3fb.txt
   untreated4fb.txt
 colData names(1): condition
```

```
detach(package:pasilla)
detach(package:DESeq)
```

2.5 Note on factor levels

In the three examples above, we applied the function `factor` to the column of interest in `colData`, supplying a character vector of levels. It is important to supply levels (otherwise the levels are chosen in alphabetical order) and to put the "control" or "untreated"

level as the first element, so that the $\log_2$ fold changes and results will be most easily interpretable. A helpful R function for changing the base level is `relevel`. The function `model.matrix` is used by the *DESeq2* package to build model matrices, and this function uses the first level as the base level.

2.6 About the pasilla dataset

We continue with the *pasilla* data constructed from the count matrix method above. This data set is from an experiment on *Drosophila melanogaster* cell cultures and investigated the effect of RNAi knock-down of the splicing factor *pasilla* [2]. The detailed transcript of the production of the *pasilla* data is provided in the vignette of the data package *pasilla*.

3 Differential expression analysis

The standard differential expression analysis steps are wrapped into a single function, `DESeq`. The individual functions are still available, described in Section C. The results are accessed using the function `results`, which extracts a results table for a single variable (by default the last variable in the design formula, and if this is a factor, the last level of this variable).

```
dds <- DESeq(dds)
res <- results(dds)
res <- res[order(res$padj),]
head(res)
```

```
DataFrame with 6 rows and 5 columns
             baseMean log2FoldChange     lfcSE    pvalue      padj
            <numeric>      <numeric> <numeric> <numeric> <numeric>
FBgn0039155       453          -4.08    0.1745 5.22e-121 5.96e-117
FBgn0029167      2165          -2.16    0.0965 9.51e-111 5.43e-107
FBgn0035085       367          -2.38    0.1354  4.16e-69  1.58e-65
FBgn0034736       118          -2.97    0.2047  1.48e-47  4.24e-44
FBgn0029896       258          -2.41    0.1679  1.21e-46  2.77e-43
FBgn0040091       611          -1.50    0.1156  1.85e-38  3.53e-35
```

Extracting results of other variables is discussed in section 5. All the values calculated by the *DESeq2* package are stored in the *DESeqDataSet* object, and access to these values is discussed in Section G.

4 Exploring results

4.1 MA-plot

For *DESeq2*, the function `plotMA` shows the $\log_2$ fold changes attributable to a variable over the mean of normalized counts. By default, the last variable in the design formula

is chosen, and points will be colored red if the adjusted p-value is less than 0.1. Points which fall out of the window are plotted as open triangles.

```
plotMA(dds)
```

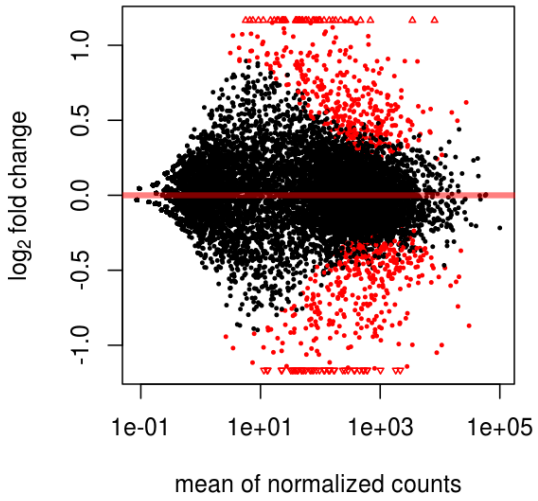

Figure 1: The MA-plot shows the $\log_2$ fold changes from the treatment over the mean of normalized counts, i.e. the average of counts normalized by size factor. The *DESeq2* package incorporates a prior on $\log_2$ fold changes, resulting in moderated estimates from genes with very low counts, as can be seen by the narrowing of spread of points on the left side of the plot.

4.2 More information on results columns

Information about which variables and tests were used can be found by calling the function `mcols` on the results object.

```
mcols(res, use.names=TRUE)
```

```
DataFrame with 5 rows and 2 columns
                     type
                 <character>
baseMean       intermediate
log2FoldChange      results
lfcSE               results
pvalue              results
padj                results
                                                  description
                                                  <character>
baseMean                                the base mean over all rows
```

```
log2FoldChange log2 fold change (MAP): condition treated vs untreated
lfcSE                  standard error: condition treated vs untreated
pvalue                      Wald test: condition treated vs untreated
padj              Wald test, BH adj.: condition treated vs untreated
```

The variable `condition` and the factor level `treated` have been combined into `condition_treated_vs_untreated`. For a particular gene, a $\log_2$ fold change of -1 for `condition_treated_vs_untreated` here means that the treatment induces a change of $2^{-1} = 0.5$ times the counts. If the variable of interest is not a factor, the $\log_2$ fold change can be interpreted as the amount of doubling observed on average for every unit of change.

4.3 Exporting results

The results can be exported using the base R functions `write.csv` or `write.delim`, and a descriptive file name indicating the variable which was tested.

```
write.csv(as.data.frame(res),
          file="condition_treated_results.csv")
```

5 Multi-factor designs

Experiments with more than one factor influencing the counts can be analyzed using model formulae with additional variables. The data in the *pasilla* package have a condition of interest (the column `condition`), as well as the type of sequencing which was performed (the column `type`).

```
colData(dds)
```

```
DataFrame with 7 rows and 3 columns
             condition        type sizeFactor
              <factor>    <factor>  <numeric>
treated1fb     treated single-read      1.512
treated2fb     treated  paired-end      0.784
treated3fb     treated  paired-end      0.896
untreated1fb untreated single-read      1.050
untreated2fb untreated single-read      1.659
untreated3fb untreated  paired-end      0.712
untreated4fb untreated  paired-end      0.784
```

We can account for the different types of sequencing, and get a clearer picture of the differences attributable to the treatment. As `condition` is the variable of interest, we put it at the end of the formula. Here we

```
design(dds) <- formula(~ type + condition)
dds <- DESeq(dds)
```

Again, we access the results using the `results` function.

```
res <- results(dds)
head(res)
```

```
DataFrame with 6 rows and 5 columns
             baseMean log2FoldChange     lfcSE    pvalue      padj
            <numeric>      <numeric> <numeric> <numeric> <numeric>
FBgn0000003     0.159         0.0891     0.117    0.4451     0.840
FBgn0000008    52.226         0.0130     0.252    0.9588     0.988
FBgn0000014     0.390         0.0241     0.145    0.8677     0.973
FBgn0000015     0.905        -0.1229     0.273    0.6523     0.892
FBgn0000017  2358.243        -0.2667     0.122    0.0293     0.204
FBgn0000018   221.242        -0.0663     0.124    0.5921     0.885
```

It is also possible to retrieve the $\log_2$ fold changes, p-values and adjusted p-values of the `type` variable. The function `results` takes an argument `name`, which is a combination of the variable, the level (numeratoFr of the fold change) and the base level (denominator of the fold change). In addition, there might be minor changes made by the `DataFrame` function on column names, e.g. changing - to `..` The function `resultsNames` will tell you the names of all available results.

```
resultsNames(dds)
```

```
[1] "Intercept"                      "type_single.read_vs_paired.end"
[3] "condition_treated_vs_untreated"
```

```
resType <- results(dds, "type_single.read_vs_paired.end")
head(resType)
```

```
DataFrame with 6 rows and 5 columns
             baseMean log2FoldChange     lfcSE    pvalue      padj
            <numeric>      <numeric> <numeric> <numeric> <numeric>
FBgn0000003     0.159        -0.0686     0.106    0.5188     0.831
FBgn0000008    52.226        -0.0808     0.247    0.7439     0.918
FBgn0000014     0.390         0.0147     0.132    0.9113     0.972
FBgn0000015     0.905        -0.2222     0.252    0.3785     0.778
FBgn0000017  2358.243         0.0081     0.122    0.9470     0.984
FBgn0000018   221.242         0.2954     0.122    0.0155     0.117
```

```
mcols(resType)
```

```
DataFrame with 5 rows and 2 columns
          type                                             description
   <character>                                             <character>
1 intermediate                               the base mean over all rows
2      results log2 fold change (MAP): type single-read vs paired-end
3      results         standard error: type single-read vs paired-end
4      results              Wald test: type single-read vs paired-end
5      results     Wald test, BH adj.: type single-read vs paired-end
```

6 Independent filtering and multiple testing

6.1 Filtering by overall count

The analyses of the previous sections involve the application of statistical tests, one by one, to each row of the data set, in order to identify those genes that have evidence for differential expression. The idea of *independent filtering* is to filter out those tests from the procedure that have no, or little chance of showing significant evidence, without even looking at their test statistic. Typically, this results in increased detection power at the same experiment-wide type I error. Here, we measure experiment-wide type I error in terms of the false discovery rate.

A good choice for a filtering criterion is one that

1. is statistically independent from the test statistic under the null hypothesis,
2. is correlated with the test statistic under the alternative, and
3. does not notably change the dependence structure –if there is any– between the 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 6.2. Its statistical validity relies on property 1 – which is simple to formally prove for many combinations of filter criteria with test statistics– and 3, which is less easy to theoretically imply from first principles, but rarely a problem in practice. We refer to [3] for further discussion of this topic.

A simple filtering criterion readily available in the results object is the normalized mean count (irrespective of biological condition). Genes with very low counts are not likely to see significant differences typically due to high dispersion. For example, we can plot the $-\log_{10}$ p-values from all genes over the normalized mean counts, with a red line at the value 10.

```
plot(res$baseMean, pmin(-log10(res$pvalue),50),
     log="x", xlab="mean of normalized counts",
     ylab=expression(-log[10](pvalue)))
abline(v=10,col="red",lwd=1)
use <- res$baseMean >= 10 & !is.na(res$pvalue)
table(use)
```

```
use
FALSE  TRUE
 7178  7292
```

We set aside those genes with normalized mean less than 10. Applying Benjamini-Hochberg adjustment on p-values results in a gain of genes with adjusted p-value below 0.1.

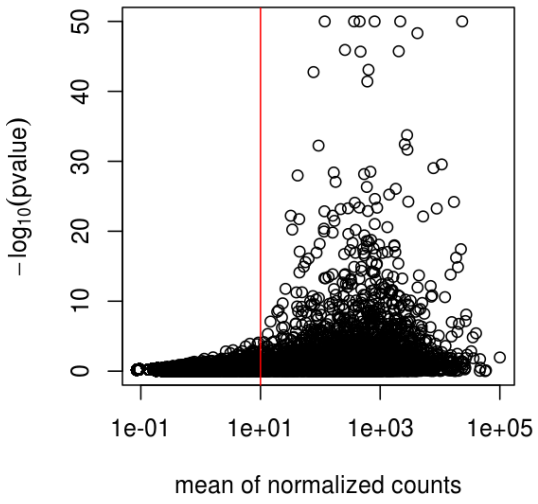

Figure 2: The mean of normalized counts provides an independent statistic for filtering the tests. It is independent because the information about the variables in the design formula is not used. By filtering out genes which fall to the left of the red line, the majority of the low p-values are kept.

```
resFilt <- res[use,]
resFilt$padj <- p.adjust(resFilt$pvalue, method="BH")
sum(res$padj < .1, na.rm=TRUE)
```

```
[1] 1241
```

```
sum(resFilt$padj < .1, na.rm=TRUE)
```

```
[1] 1422
```

6.2 Why does it work?

Consider the p value histogram in Figure 3. 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(res$pvalue[!use], breaks=50, plot=FALSE)
h2 <- hist(res$pvalue[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)))
```

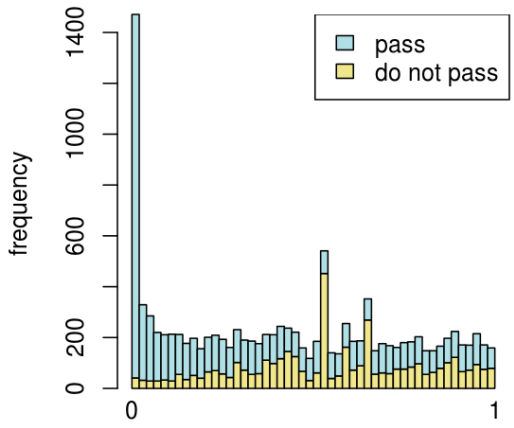

Figure 3: Histogram of p values for all tests (`res$pvalue`). The area shaded in blue indicates the subset of those that pass the filtering, the area in khaki those that do not pass.

6.3 Diagnostic plots for multiple testing

The Benjamini-Hochberg multiple testing adjustment procedure [4] has a simple graphical illustration, which we produce in the following code chunk. Its result is shown in the left panel of Figure 4.

```
orderInPlot <- order(resFilt$pvalue)
showInPlot <- (resFilt$pvalue[orderInPlot] <= 0.08)
alpha <- 0.1


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

Schweder and Spjøtvoll [5] suggested a diagnostic plot of the observed p-values which permits estimation of the fraction of true null hypotheses. For a series of hypothesis tests $H_1, \ldots, H_m$ with p-values p_i, they suggested plotting

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

where $N(p)$ is the number of p-values greater than p. An application of this diagnostic plot to `resFilt$pvalue` is shown in the right panel of Figure 4. When all null hypotheses are true, the p-values are each uniformly distributed in $[0, 1]$, Consequently, the cumulative distribution function of $(p_1, \ldots, p_m)$ is expected to be close to the line $F(t) = t$. By symmetry, the same applies to $(1 - p_1, \ldots, 1 - p_m)$. When (without loss of generality) the first m_0 null hypotheses are true and the other $m - m_0$ are false, the cumulative

distribution function of $(1-p_1,\ldots,1-p_{m_0})$ is again expected to be close to the line $F_0(t)=t$. The cumulative distribution function of $(1-p_{m_0+1},\ldots,1-p_m)$, on the other hand, is expected to be close to a function $F_1(t)$ which stays below F_0 but shows a steep increase towards 1 as t approaches 1. In practice, we do not know which of the null hypotheses are true, so we can only observe a mixture whose cumulative distribution function is expected to be close to

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

Such a situation is shown in the right panel of Figure 4. If $F_1(t)/F_0(t)$ is small for small t, then the mixture fraction $\frac{m_0}{m}$ can be estimated by fitting a line to the left-hand portion of the plot, and then noting its height on the right. Such a fit is shown by the red line in the right panel of Figure 4.

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

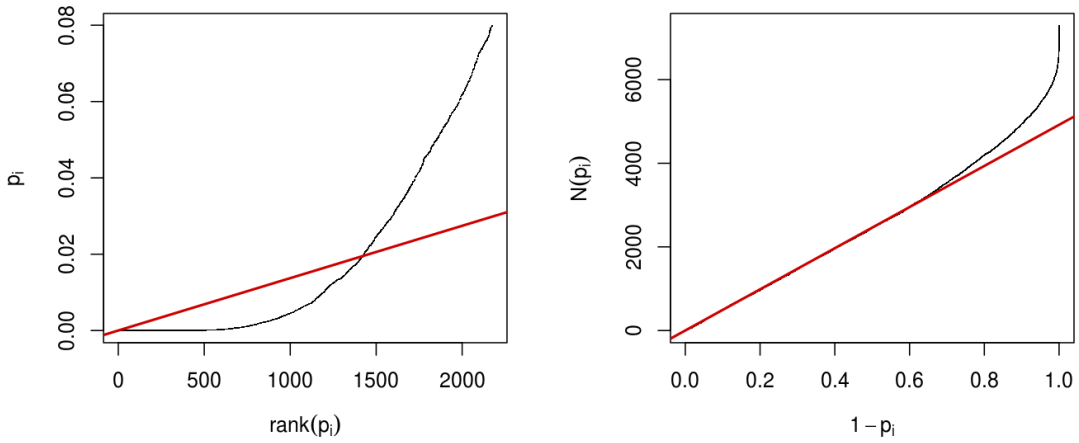

Figure 4: *Left:* illustration of the Benjamini-Hochberg multiple testing adjustment procedure [4]. The black line shows the p-values (y-axis) versus their rank (x-axis), starting with the smallest p-value from the left, then the second smallest, and so on. Only the first 2174 p-values are shown. The red line is a straight line with slope α/n, where $n=7292$ is the number of tests, and $\alpha=0.1$ is a target false discovery rate (FDR). FDR is controlled at the value α if the genes are selected that lie to the left of the rightmost intersection between the red and black lines: here, this results in 1422 genes. *Right:* Schweder and Spjøtvoll plot, as described in the text. For both of these plots, the p-values `resFilt$pvalues` from Section 6.1 were used as a starting point. Analogously, one can produce these types of plots for any set of p-values, for instance those from the previous sections.

7 Count data transformations

For some applications, it is useful to work with transformed versions of the count data. Maybe the most obvious choice is the logarithmic transformation. Since count values for a gene can be zero in some conditions (and non-zero in others), some advocate the use of *pseudocounts*, i. e. transformations of the form

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

where n represents the count values and n_0 is a positive constant.

In this section, we discuss two alternative approaches that offer more theoretical justification and a rational way of choosing the parameter equivalent to n_0 above. One method incorporates priors on the sample differences, and the other uses the concept of variance stabilizing transformations [6–8].

The two functions, `rlogTransformation` and `varianceStabilizingTransformation`, have an argument `blind`, for whether the transformation should be blind to the sample information specified by the design formula. By setting the argument `blind` to `TRUE`, the functions will re-estimate the dispersions using only an intercept (design formula $\tilde{1}$. This setting should be used in order to compare samples in a manner unbiased by the information about experimental groups, for example to perform sample QA (quality assurance) as demonstrated below. By setting `blind` to `FALSE`, the dispersions already estimated will be used to perform transformations, or if not present, they will be estimated using the current design formula. This setting should be used for transforming data for downstream analysis.

```
rld <- rlogTransformation(dds, blind=TRUE)
vsd <- varianceStabilizingTransformation(dds, blind=TRUE)
```

7.1 Regularized log transformation

The function `rlogTransformation`, stands for *regularized log*, transforming the original count data to the $\log_2$ scale by fitting a model with a term for each sample and a prior distribution on the coefficients which is estimated from the data. This is very similar to the regularization used by the `DESeq` and `nbinomWaldTest`, as seen in Figure 1. The resulting data contains elements defined as:

$$\log_2(q_{ij}) = x_{j\cdot}\beta_i$$

where q_{ij} is a parameter proportional to the expected true concentration of fragments for gene i and sample j (see Section B), $x_{j\cdot}$ is the j-th row of the design matrix X, which has a 1 for the intercept and a 1 for the sample-specific beta, and β_i is the vector of coefficients for gene i. Without priors, this design matrix would lead to a non-unique solution, however the addition of a prior on non-intercept betas allows for a unique solution to be found. The regularized log transformation is preferable to the variance stabilizing transformation if the size factors vary widely.

7.2 Variance stabilizing transformation

Above, we used a parametric fit for the dispersion. In this case, the closed-form expression for the variance stabilizing transformation is used by `varianceStabilizingTransformation`, which is derived in the file `vst.pdf`, that is distributed in the package alongside this vignette. If a local fit is used (option `fitType="locfit"` to `estimateDispersions`) a numerical integration is used instead.

The resulting variance stabilizing transformation is shown in Figure 5. The code that produces the figure is hidden from this vignette for the sake of brevity, but can be seen in the `.Rnw` or `.R` source file.

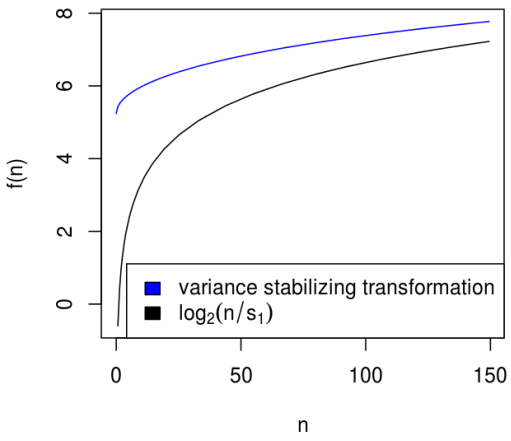

Figure 5: Graphs of the variance stabilizing transformation for sample 1, in blue, and of the transformation $f(n) = \log_2(n/s_1)$, in black. n are the counts and s_1 is the size factor for the first sample.

7.3 Effects of transformations on the variance

Figure 6 plots the standard deviation of the transformed data, across samples, against the mean, using the shifted logarithm transformation (3), the regularized log transformation and the variance stabilizing transformation. The shifted logarithm has elevated standard deviation in the lower count range, and the regularized log to a lesser extent, while for the variance stabilized data the standard deviation is roughly constant along the whole dynamic range.

```
library("vsn")
par(mfrow=c(1,3))
notAllZero <- (rowSums(counts(dds))>0)
meanSdPlot(log2(counts(dds,normalized=TRUE)[notAllZero,] + 1),
           ylim = c(0,2.5))
meanSdPlot(assay(rld[notAllZero,]), ylim = c(0,2.5))
meanSdPlot(assay(vsd[notAllZero,]), ylim = c(0,2.5))
```

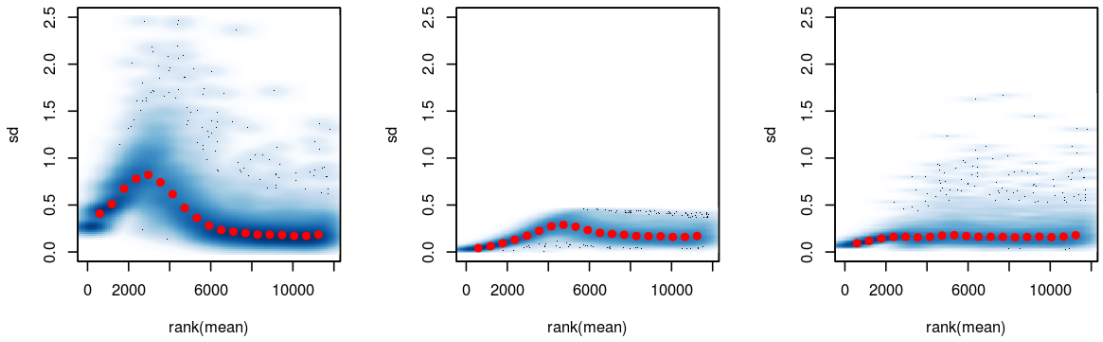

Figure 6: Per-gene standard deviation (taken across samples), against the rank of the mean, for the shifted logarithm $\log_2(n+1)$ (left), the regularized log transformation (center) and the variance stabilizing transformation (right).

8 Data quality assessment by sample clustering and visualization

Data quality assessment and quality control (i. e. the removal of insufficiently good data) are essential steps of any data analysis. Even though we present these steps towards the end of this vignette, they should typically be performed very early in the analysis of a new data set, preceding or in parallel to the differential expression testing.

We define the term *quality* as *fitness for purpose*[2]. Our purpose is the detection of differentially expressed genes, and we are looking in particular for samples whose experimental treatment suffered from an anormality that renders the data points obtained from these particular samples detrimental to our purpose.

8.1 Heatmap of the count table

To explore a count table, it is often instructive to look at it as a heatmap. Below we show how to produce such a heatmap from the raw and transformed data.

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


heatmap.2(counts(dds,normalized=TRUE)[select,], col = hmcol,
          Rowv = FALSE, Colv = FALSE, scale="none",
          dendrogram="none", trace="none", margin=c(10,6))
```

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

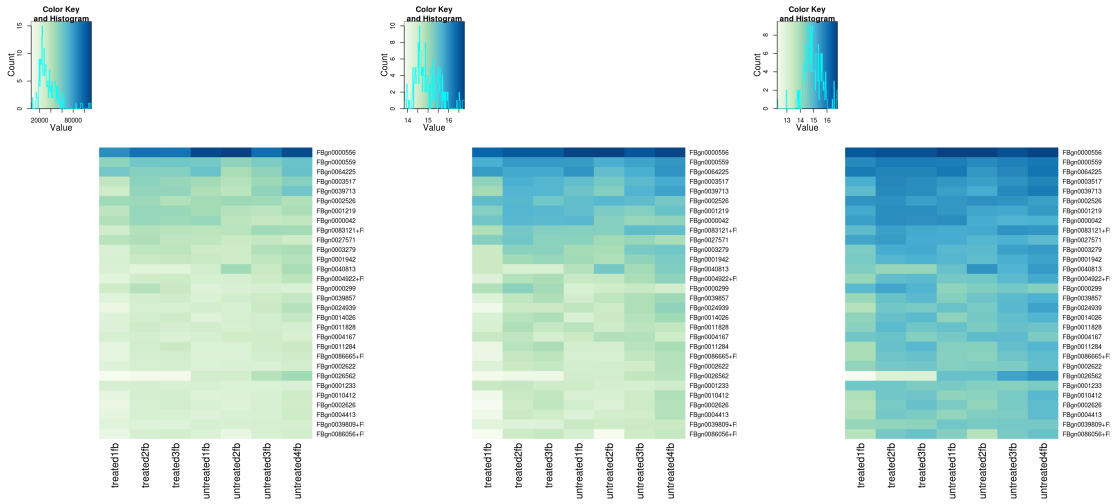

Figure 7: Heatmaps showing the expression data of the 30 most highly expressed genes. The data is of raw counts (left), from regularized log transformation (center) and from variance stabilizing transformation (right).

```
heatmap.2(assay(rld)[select,], col = hmcol,
          Rowv = FALSE, Colv = FALSE, scale="none",
          dendrogram="none", trace="none", margin=c(10, 6))
```

```
heatmap.2(assay(vsd)[select,], col = hmcol,
          Rowv = FALSE, Colv = FALSE, scale="none",
          dendrogram="none", trace="none", margin=c(10, 6))
```

8.2 Heatmap of the sample-to-sample distances

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

```
distsRL <- dist(t(assay(rld)))
```

A heatmap of this distance matrix gives us an overview over similarities and dissimilarities between samples (Figure 8):

```
mat <- as.matrix(distsRL)
rownames(mat) <- colnames(mat) <- with(colData(dds),
                                       paste(condition, type, sep=" : "))
heatmap.2(mat, trace="none", col = rev(hmcol), margin=c(13, 13))
```

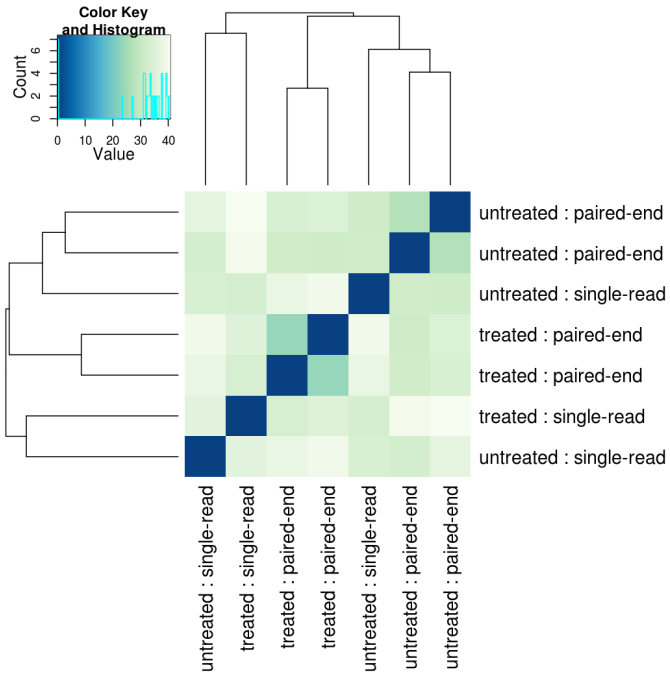

Figure 8: Heatmap showing the Euclidean distances between the samples as calculated from the regularized log transformation.

8.3 Principal component plot of the samples

Related to the distance matrix of Section 8.2 is the PCA plot of the samples, which we obtain as follows (Figure 9).

```
print(plotPCA(rld, intgroup=c("condition", "type")))
```

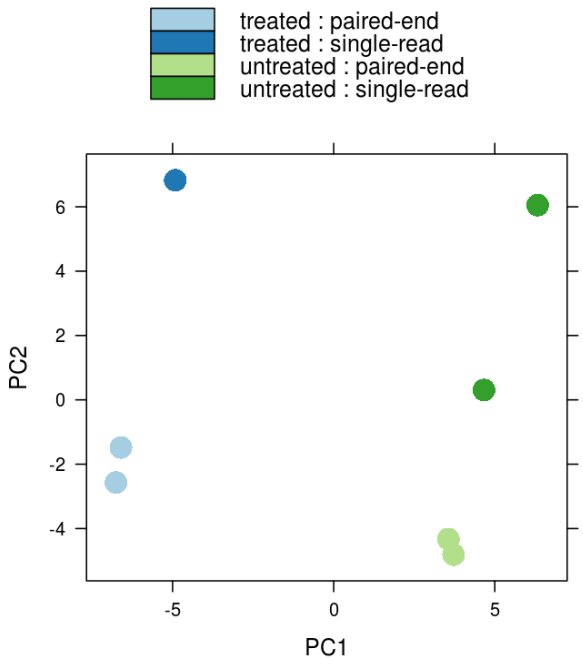

Figure 9: PCA plot. The 7 samples shown in the 2D plane spanned by their first two principal components. This type of plot is useful for visualizing the overall effect of experimental covariates and batch effects.

A Changes compared to the DESeq package

The main changes in the package *DESeq2*, compared to the (older) version *DESeq*, are as follows:

- *SummarizedExperiment* is used as the superclass for storage of input data, intermediate calculations and results.
- Maximum *a posteriori* estimation of GLM coefficients incorporating a zero-mean normal prior with variance estimated from data (equivalent to Tikhonov/ridge regularization). This adjustment has little effect on genes with high counts, yet it helps to moderate the otherwise large spread in $\log_2$ fold changes for genes with low counts (e. g. single digits per condition).
- Maximum *a posteriori* estimation of dispersion replaces the `sharingMode` options `fit-only` or `maximum` of the previous version of the package. [9]
- All estimation and inference is based on the generalized linear model, which includes the two condition case (previously the *exact test* was used).
- The Wald test for significance of GLM coefficients is provided as the default inference method, with the likelihood ratio test of the previous version still available.
- It is possible to provide a matrix of sample-/gene-dependent normalization factors.

B Generalized linear model

The differential expression analysis in *DESeq2* uses a generalized linear model of the form:

$$K_{ij} \sim \mathrm{NB}(\mu_{ij}, \alpha_i)$$
$$\mu_{ij} = s_j q_{ij}$$
$$\log_2(q_{ij}) = x_{j\cdot}\beta_i$$

where counts K_{ij} for gene i, sample j are modeled using a negative binomial distribution with fitted mean μ_{ij} and a gene-specific dispersion parameter α_i. The fitted mean is composed of a sample-specific size factor s_j[3] and a parameter q_{ij} proportional to the expected true concentration of fragments for sample j. The coefficients β_i give the $\log_2$ fold changes for gene i for each column of the model matrix X. Dispersions are estimated using a Cox-Reid adjusted profile likelihood, as first implemented for RNA-Seq data in *edgeR* [10, 11]. For further details on dispersion estimation and inference, please see the manual pages for the functions `DESeq` and `estimateDispersions`. For access to the calculated values see Section G

C Wald test individual steps

The function `DESeq` runs the following functions in order:

```
dds <- estimateSizeFactors(dds)
dds <- estimateDispersions(dds)
dds <- nbinomWaldTest(dds)
```

D Likelihood ratio test

The likelihood ratio test substitutes `nbinomWaldTest` with `nbinomLRT` in the last step above. In this case, the user provides the full formula (the formula stored in `design(dds)`), and a reduced formula, e.g. one which does not contain the variable of interest. The degrees of freedom for the test is obtained from the number of parameters in the two models. The Wald test and the likelihood ratio test share many of the same genes with adjusted p-value $< .1$ for this experiment.

```
ddsLRT <- nbinomLRT(dds, reduced = ~ type)
resLRT <- results(ddsLRT)
tab <- table(Wald=res$padj < .1, LRT=resLRT$padj < .1)
addmargins(tab)
```

[3] The model can be generalised to use sample- **and** gene-dependent normalisation factors, see Appendix I.

```
        LRT
Wald     FALSE  TRUE   Sum
  FALSE 10143     5 10148
  TRUE     11  1230  1241
  Sum   10154  1235 11389
```

E Dispersion plot and fitting alternatives

Plotting the dispersion estimates is a useful diagnostic. The dispersion plot in Figure 10 is typical, with the final estimates shrunk from the gene-wise estimates towards the fitted estimates. Some gene-wise estimates are flagged as outliers and not shrunk towards the fitted value, (this outlier detection is described in the man page for `estimateDispersionsMAP`). The amount of shrinkage can be more or less than seen here, depending on the sample size, the number of coefficients, the row mean and the variability of the gene-wise estimates.

```
plotDispEsts(dds)
```

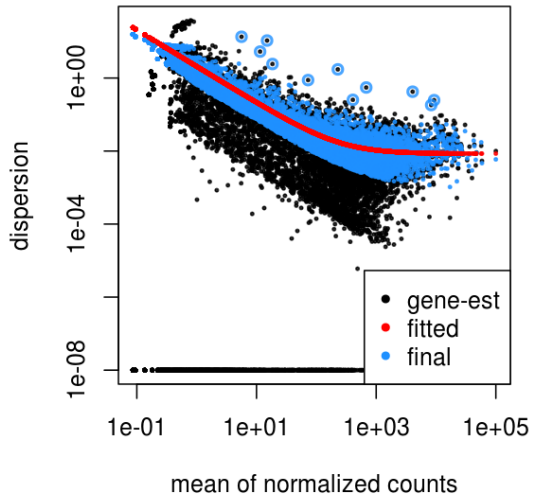

Figure 10: The dispersion estimate plot shows the gene-wise estimates (black), the fitted values (red), and the final maximum *a posteriori* estimates used in testing (blue).

E.1 Local dispersion fit

The local dispersion fit is available in case the parametric fit fails to converge. A warning will be printed that one should use `plotDispEsts` to check the quality of the fit, whether the curve is pulled dramatically by a few outlier points. In this case the two fit types appear to produce similar curves (Figure 11).

```
ddsLocal <- estimateDispersions(dds, fitType="local")
plotDispEsts(ddsLocal)
```

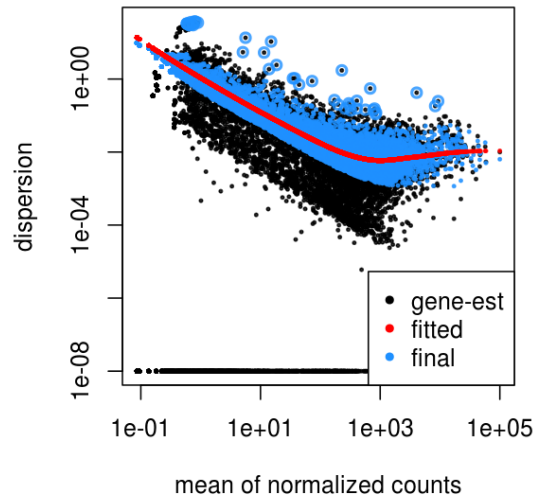

Figure 11: A dispersion estimate plot using a local regression fit is similar to that of Figure 10.

E.2 Mean dispersion

While RNA-Seq data tend to demonstrate a dispersion-mean dependence, this assumption is not appropriate for all assays. An alternative is to use the mean of all gene-wise dispersion estimates to benefit from information sharing across genes (Figure 12).

```
ddsMean <- estimateDispersions(dds, fitType="mean")
plotDispEsts(ddsMean)
```

E.3 Supply a custom dispersion fit

Any fitted values can be provided during dispersion estimation, using the lower-level functions described in the manual page for `estimateDispersionsGeneEst`. In the first line of the code below, the function `estimateDispersionsGeneEst` stores the gene-wise estimates in the metadata column `dispGeneEst`. In the last line, the function `estimateDispersionsMAP`, uses this column and the column `dispFit` to generate maximum *a posteriori* (MAP) estimates of dispersion. The modeling assumption is that the true dispersions are distributed according to a log-normal prior around the fitted values in the column `fitDisp`. The width of this prior is calculated from the data.

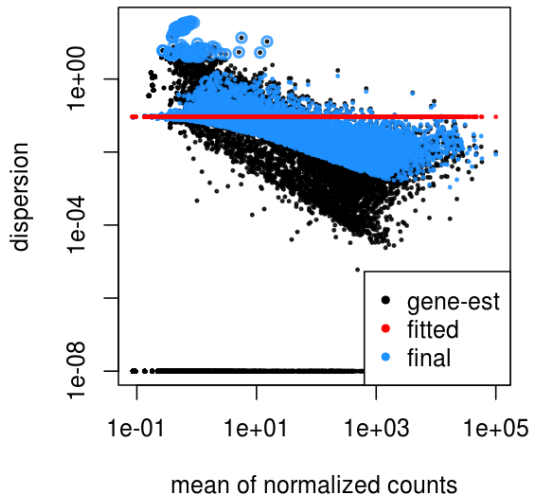

Figure 12: A dispersion estimate plot using the mean, though this would not be recommended for this dataset as the dispersion estimates exhibit a row-mean-dependent trend.

```
ddsMed <- estimateDispersionsGeneEst(dds)
useForMedian <- mcols(ddsMed)$dispGeneEst > 1e-7
medianDisp <- median(mcols(ddsMed)$dispGeneEst[useForMedian],na.rm=TRUE)
mcols(ddsMed)$dispFit <- medianDisp
ddsMed <- estimateDispersionsMAP(ddsMed)
```

F Count outlier detection

DESeq2 relies on the negative binomial distribution to make estimates and perform statistical inference on differences. While the negative binomial is versatile in having a mean and dispersion parameter, extreme counts in individual samples might not fit well to the negative binomial. For this reason, we perform automatic detection of count outliers. We use Cook's distance, which is a measure of how much the fitted coefficients would change if an individual sample were removed. [12] The Cook's distances are stored as a matrix available in `assays(dds)[["cooks"]]`. These values are the same as those produced by the `cooks.distance` function of the *stats* package, except using the fitted dispersion and taking into account the size factors.

By default, if the Cook's distance for a sample is larger than the .75 quantile of the $F(p, m-p)$ distribution (with p the number of parameters including the intercept and m number of samples), then the gene is flagged in `mcols(dds)$cooksOutlier`, and the p-value of the row is set to `NA`. The cutoff can be modified using the `cooksCutoff` argument to `nbinomWaldTest` or `nbinomLRT`. The functionality can be disabled by setting `cooksCutoff` to `Inf` or `FALSE`. If the removal of a sample would mean that a coefficient cannot be fitted (e.g. if there is only one sample for a given group), then the Cook's

distance for this sample is not counted towards the flagging.

```
W <- mcols(dds)$WaldStatistic_condition_treated_vs_untreated
maxCooks <- mcols(dds)$maxCooks
idx <- !is.na(W)
plot(rank(W[idx]), maxCooks[idx], xlab="rank of Wald statistic",
     ylab="maximum Cook's distance per gene",
     ylim=c(0,5), cex=.4, col=rgb(0,0,0,.3))
m <- ncol(dds)
p <- 3
abline(h=qf(.75, p, m - p))
```

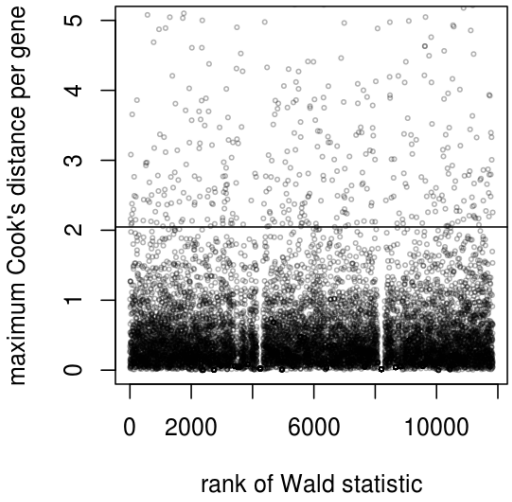

Figure 13: Plot of the maximum Cook's distance per gene over the rank of the Wald statistics for the condition. The two regions with small Cook's distances are genes with a single count in one sample. The horizontal line is the default cutoff used for 7 samples and 3 estimated parameters.

G Access to all calculated values

All row-wise calculated values (intermediate dispersion calculations, coefficients, standard errors, etc.) are stored in the *DESeqDataSet* object, e.g. `dds` in this vignette. These values are accessible by calling `mcols` on `dds`. Descriptions of the columns are accessible by two calls to `mcols`.

```
mcols(dds,use.names=TRUE)[1:4,1:4]
```

```
DataFrame with 4 rows and 4 columns
             baseMean    baseVar    allZero dispGeneEst
            <numeric> <numeric> <logical>    <numeric>
```

```
FBgn0000003     0.159     0.178     FALSE    3.49e-01
FBgn0000008    52.226   154.611     FALSE    5.12e-02
FBgn0000014     0.390     0.444     FALSE    1.44e+01
FBgn0000015     0.905     0.799     FALSE    1.00e-08
```

```
mcols(mcols(dds), use.names=TRUE)[1:4,]
```

```
DataFrame with 4 rows and 2 columns
                    type                         description
             <character>                         <character>
baseMean    intermediate       the base mean over all rows
baseVar     intermediate   the base variance over all rows
allZero     intermediate      all counts in a row are zero
dispGeneEst intermediate gene-wise estimates of dispersion
```

H Multi-level conditions

As mentioned in Section 2.5, it is important to refactor columns which will be used in analysis, providing the levels in the order desired, as the first level will be used as a base level. For a column with 3 levels "Control", "A", and "B", the refactoring would be:

```
colData(x)$condition <- factor(colData(x)$condition,
                               levels=c("Control","A","B"))
```

In this case, there will be two coefficients in the analysis with available results: $\log_2$ fold changes of "A" vs "Control", and $\log_2$ fold changes of "B" vs "Control". It is also possible to set the base level using the R function `relevel`. We are working on an implementation of contrasts, which would allow comparison of the coefficients of "A" against "B".

I Sample-/gene-dependent normalization factors

In some experiments, there might be gene-dependent dependencies which vary across samples. For instance, GC-content bias or length bias might vary across samples coming from different labs or processed at different times. We use the terms "normalization factors" for a gene $\times$ sample matrix, and "size factors" for a single number per sample. Incorporating normalization factors, the mean parameter μ_{ij} from Section B becomes:

$$\mu_{ij} = NF_{ij} q_{ij}$$

with normalization factor matrix NF having the same dimensions as the counts matrix K. This matrix can be incorporated as shown below. We recommend providing a matrix with a mean of 1, which can be accomplished by dividing out the mean of the matrix.

```
normFactors <- normFactors / mean(normFactors)
normalizationFactors(dds) <- normFactors
```

These steps then replace `estimateSizeFactors` in the steps described in Section C. Normalization factors, if present, will always be used in the place of size factors.

The methods provided by the *cqn* or *EDASeq* packages can help correct for GC or length biases. They both describe in their vignettes how to create matrices which can be used by *DESeq2*. From the formula above, we see that normalization factors should be on the scale of the counts, like size factors, and unlike offsets which are typically on the scale of the predictors (i.e. the logarithmic scale for the negative binomial GLM). At the time of writing, the transformation from the matrices provided by these packages should be:

```
cqnOffset <- cqnObject$glm.offset
cqnNormFactors <- exp(cqnOffset)
EDASeqNormFactors <- exp(-1 * EDASeqOffset)
```

J Session Info

- R version 3.0.1 (2013-05-16), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=C`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, grDevices, graphics, grid, methods, parallel, stats, utils
- Other packages: Biobase 2.20.1, BiocGenerics 0.6.0, DESeq2 1.0.18, DEXSeq 1.6.0, GenomicRanges 1.12.4, IRanges 1.18.2, KernSmooth 2.23-10, MASS 7.3-27, RColorBrewer 1.0-5, Rcpp 0.10.4, RcppArmadillo 0.3.900.0, caTools 1.14, gdata 2.13.2, gplots 2.11.3, gtools 3.0.0, lattice 0.20-15, locfit 1.5-9.1, parathyroidSE 0.99.5, vsn 3.28.0
- Loaded via a namespace (and not attached): AnnotationDbi 1.22.6, BiocInstaller 1.10.2, Biostrings 2.28.0, DBI 0.2-7, DESeq 1.12.0, RCurl 1.95-4.1, RSQLite 0.11.4, Rsamtools 1.12.3, XML 3.98-1.1, affy 1.38.1, affyio 1.28.0, annotate 1.38.0, biomaRt 2.16.0, bitops 1.0-5, genefilter 1.42.0, geneplotter 1.38.0, hwriter 1.3, limma 3.16.6, pasilla 0.2.16, preprocessCore 1.22.0, splines 3.0.1, statmod 1.4.17, stats4 3.0.1, stringr 0.6.2, survival 2.37-4, tools 3.0.1, xtable 1.7-1, zlibbioc 1.6.0

References

[1] Felix Haglund, Ran Ma, Mikael Huss, Luqman Sulaiman, Ming Lu, Inga-Lena Nilsson, Anders Höög, Christofer C. Juhlin, Johan Hartman, and Catharina Larsson. Evidence of a Functional Estrogen Receptor in Parathyroid Adenomas. *Journal of Clinical Endocrinology & Metabolism*, September 2012.

[2] 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.

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

[4] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society B*, 57:289–300, 1995.

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

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

[7] Wolfgang Huber, Anja von Heydebreck, Holger Sültmann, Annemarie Poustka, and Martin Vingron. Parameter estimation for the calibration and variance stabilization of microarray data. *Statistical Applications in Genetics and Molecular Biology*, 2(1):Article 3, 2003.

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

[9] Hao Wu, Chi Wang, and Zhijin Wu. A new shrinkage estimator for dispersion improves differential expression detection in RNA-seq data. *Biostatistics*, September 2012.

[10] D. R. Cox and N. Reid. Parameter orthogonality and approximate conditional inference. *Journal of the Royal Statistical Society, Series B*, 49(1):1–39, 1987.

[11] Davis J McCarthy, Yunshun Chen, and Gordon K Smyth. Differential expression analysis of multifactor RNA-Seq experiments with respect to biological variation. *Nucleic Acids Research*, 40:4288–4297, January 2012.

[12] R. Dennis Cook. Detection of Influential Observation in Linear Regression. *Technometrics*, February 1977.