RNA-Seq - differential expression using DESeq2

Denis Puthier

16 December 2014

Contents

The Pasilla dataset

This dataset is available from the Pasilla Bioconductor library and is derived from the work from Brooks et al. (Conservation of an RNA regulatory map between Drosophila and mammals. Genome Research, 2010).

Alternative splicing is generally controlled by proteins that bind directly to regulatory sequence elements and either activate or repress splicing of adjacent splice sites in a target pre-mRNA. Here, the authors have combined RNAi and mRNA-seq to identify exons that are regulated by Pasilla (PS), the Drosophila melanogaster ortholog of the mammalian RNA-binding proteins NOVA1 and NOVA2.

Loading the dataset

To get the dataset, you need to install the pasilla library from Bioconductor then load this library.

```
## Install the library if needed then load it
if(!require("pasilla")){
  source("http://bioconductor.org/biocLite.R")
  biocLite("pasilla")
}
```

```
## Loading required package: pasilla
```

```
library("pasilla")
```

To get the path to the tabulated file containing count table use the command below. Here, the **system.file()** function is simply used to get the path to the directory containing the count table. The "pasilla_gene_counts.tsv" file contains counts for each gene (row) in each sample (column).

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

Load the file using the read.table function.

```
## Read the data table
count.table<-  read.table( datafile, header=TRUE, row.names=1, quote="", comment.char="" )
head(count.table)
```

```
##             untreated1 untreated2 untreated3 untreated4 treated1 treated2
## FBgn0000003          0          0          0          0        0        0
## FBgn0000008         92        161         76         70      140       88
## FBgn0000014          5          1          0          0        4        0
## FBgn0000015          0          2          1          2        1        0
## FBgn0000017       4664       8714       3564       3150     6205     3072
## FBgn0000018        583        761        245        310      722      299
##             treated3
## FBgn0000003        1
## FBgn0000008       70
## FBgn0000014        0
## FBgn0000015        0
## FBgn0000017     3334
## FBgn0000018      308
```

As some genes were not detected in any of the samples, we will discard them before further analysis.

```
## Some genes were not detected at all in these samples. We will discard them.
count.table <- count.table[rowSums(count.table) > 0,]
```

Phenotypic data

The dataset contains RNA-Seq count data for RNAi treated or S2-DRSC untreated cells (late embryonic stage). Some results were obtained through single-end sequencing whereas others were obtained using paired-end sequencing. We will store these informations in two vectors (cond.type and lib.type).

```
cond.type <-  c( "untreated", "untreated", "untreated","untreated", "treated", "treated", "treated" )
lib.type   <-  c( "single-end", "single-end", "paired-end", "paired-end", "single-end", "paired-end", "
```

Next, we will extract a subset of the data containing only paired-end samples.

```
## Select only Paired-end datasets
isPaired <-  lib.type == "paired-end"
show(isPaired) # show is an R alternative to print
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
```

```
count.table <-  count.table[ , isPaired ] ## Select only the paired samples
head(count.table)
```

```
##             untreated3 untreated4 treated2 treated3
## FBgn0000003          0          0        0        1
## FBgn0000008         76         70       88       70
## FBgn0000014          0          0        0        0
## FBgn0000015          1          2        0        0
## FBgn0000017       3564       3150     3072     3334
## FBgn0000018        245        310      299      308
```

```
cond.type <-  cond.type[isPaired]
```

Phenotypic data need to be stored in a data.frame object for further analysis.

```
## We need to get phenotypic data into a data.frame for
## subsequent analysis
conditions <- data.frame(cond.type=factor(cond.type))
print(conditions)
```

```
##   cond.type
## 1 untreated
## 2 untreated
## 3   treated
## 4   treated
```

Descriptive statistics

Before going further in the analysis, we will compute some descriptive statistics on the dataset.

```
## Dimensions
ncol(count.table)
```

```
## [1] 4
```

```
nrow(count.table)
```

```
## [1] 12359
```

```
dim(count.table)
```

```
## [1] 12359     4
```

```
## Min, Max, median...
summary(count.table)
```

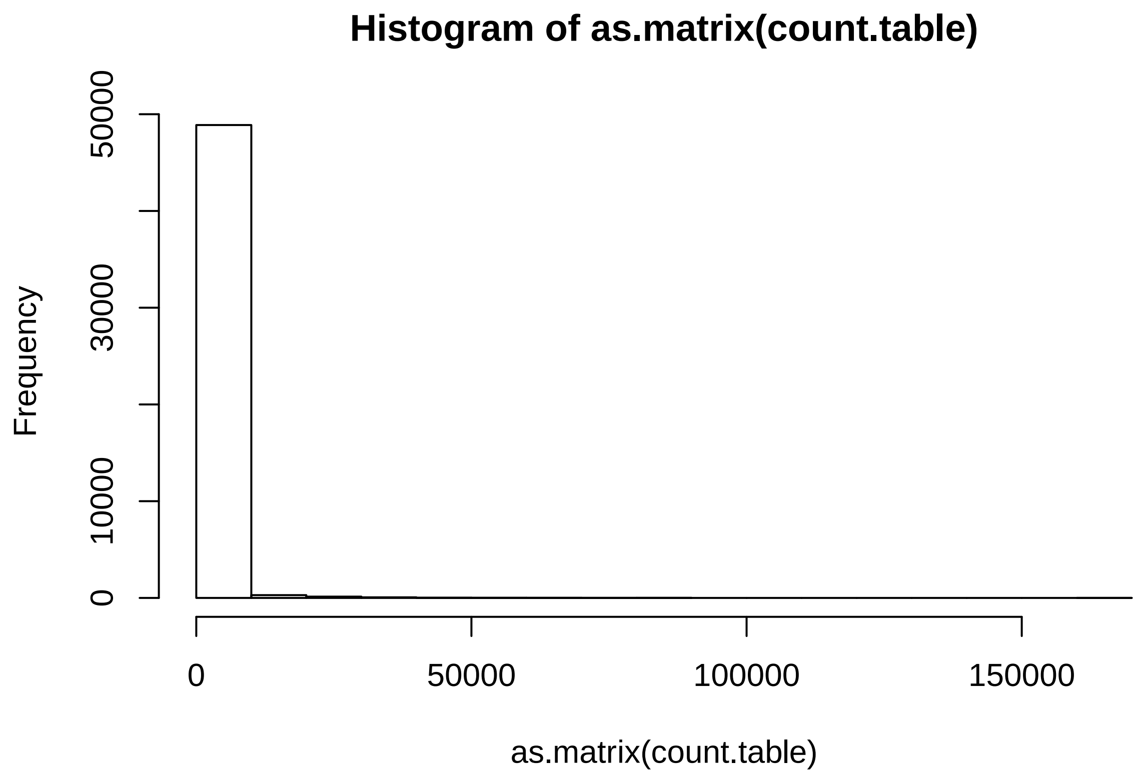
```
##    untreated3         untreated4          treated2           treated3
##  Min.   :     0.0   Min.   :     0.0   Min.   :     0.0   Min.   :     0
##  1st Qu.:     2.0   1st Qu.:     2.0   1st Qu.:     2.0   1st Qu.:     2
##  Median :    67.0   Median :    80.0   Median :    82.0   Median :    91
##  Mean   :   676.3   Mean   :   796.3   Mean   :   774.5   Mean   :   837
##  3rd Qu.:   466.0   3rd Qu.:   546.0   3rd Qu.:   552.0   3rd Qu.:   599
##  Max.   :131242.0   Max.   :167116.0   Max.   :146390.0   Max.   :164148
```

The summary only displays a few milestone values (mean, median, quartiles). In order to get a better intuition of the data, we can draw an histogram of all values.

```
## Define colors for subsequent plots
col <- c("green","orange")[as.numeric(conditions$cond.type)]
show(col)
```

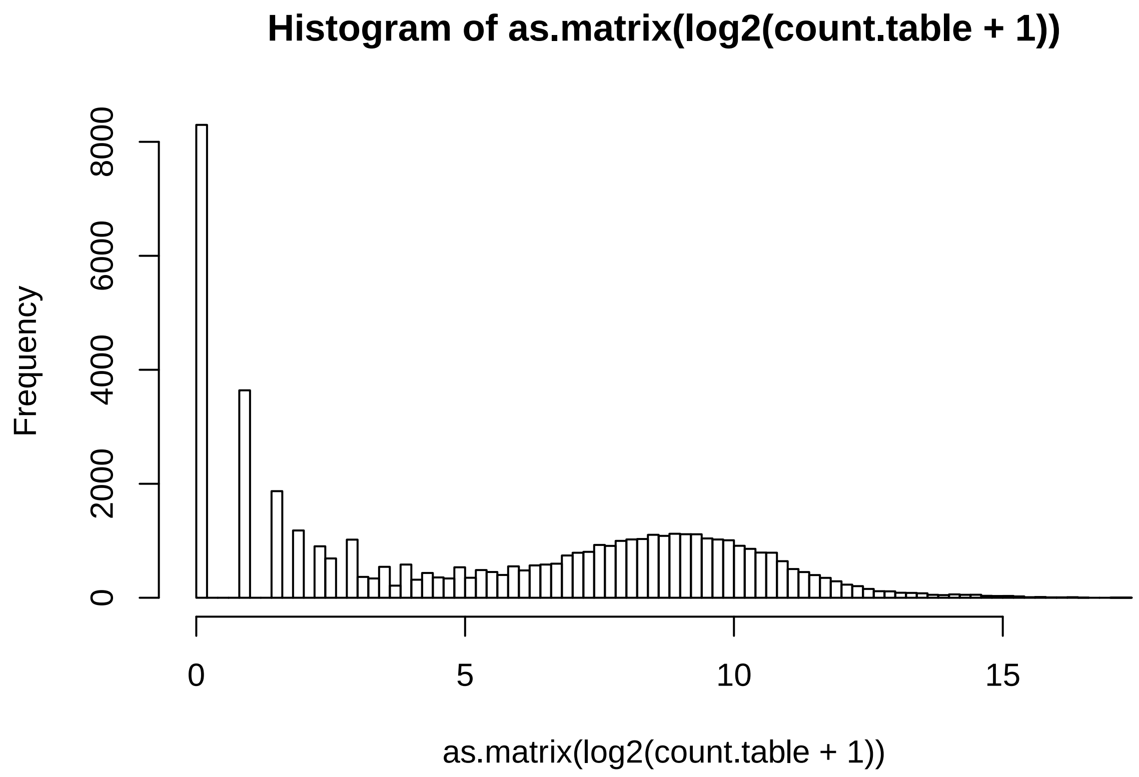
```
## [1] "orange" "orange" "green"  "green"
```

```
## Data distribution
hist(as.matrix(count.table))
```

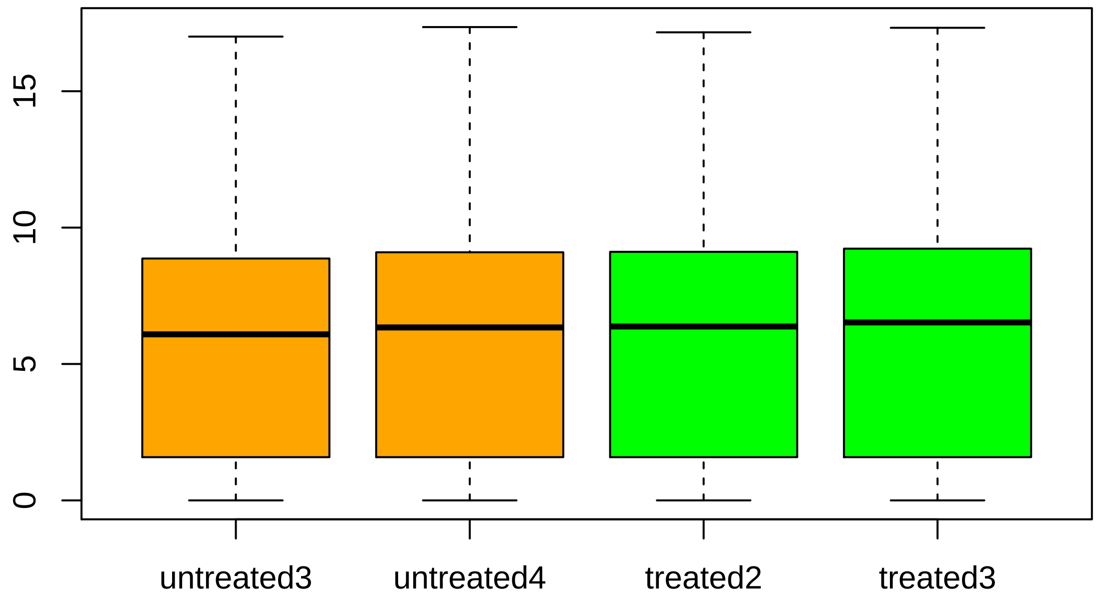


The histogram is not very informative so far, apparently due to the presence of a few very high count values, that impose a very large scale on the X axis. We can use a logarithmic transformation to improve the readability. Note that we will add pseudo count to avoid problems with “zero” counts observed for some genes in some samples.

```
## Data distribution in log scale.
## Logarithmic transformation
hist(as.matrix(log2(count.table + 1)),  breaks=100)
```



```
## Boxplot
boxplot(log2(count.table + 1), col=col)
```

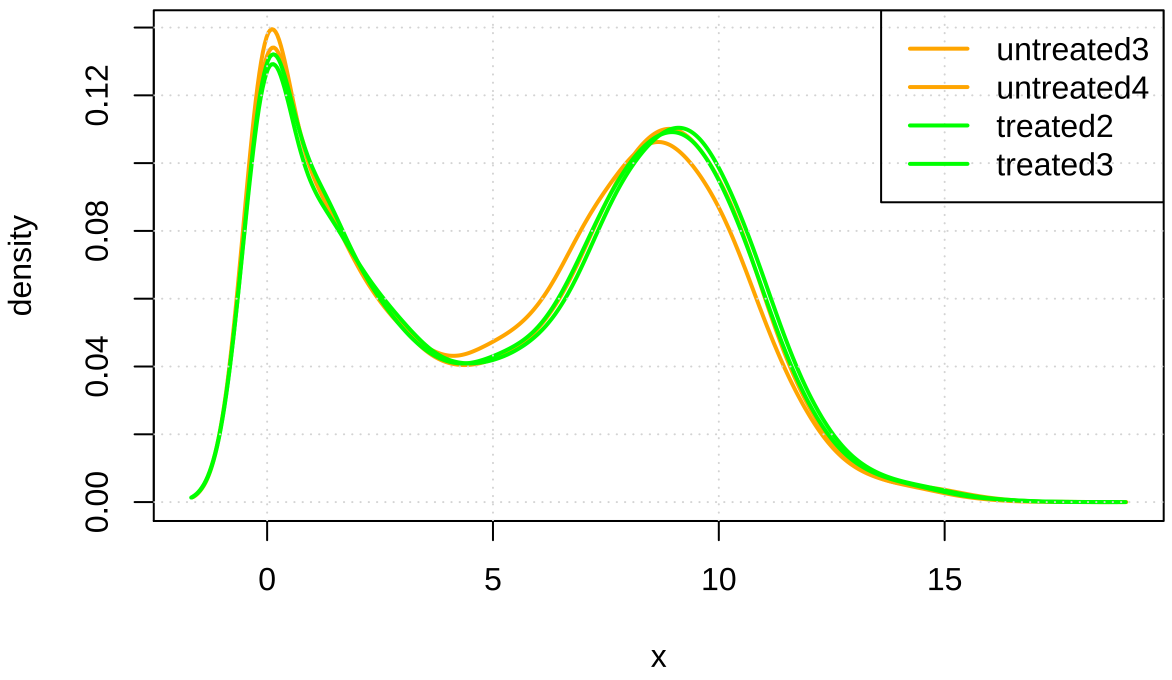


Anothre way to get an intuition of the value distributions is to use the *plotDensity()* function, which draws one density curve for each sample.

```
## Density
## We will require one function from the affy package
if(!require("affy")){
  source("http://bioconductor.org/biocLite.R")
  biocLite("affy")
}
```

```
## Loading required package: affy
## Loading required package: BiocGenerics
## Loading required package: parallel
##
## Attaching package: 'BiocGenerics'
##
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
##
## The following object is masked from 'package:stats':
##
##     xtabs
##
## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, as.vector, cbind,
##     colnames, duplicated, eval, evalq, Filter, Find, get,
##     intersect, is.unsorted, lapply, Map, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,
##     rbind, Reduce, rep.int, rownames, sapply, setdiff, sort,
##     table, tapply, union, unique, unlist
##
## Loading required package: Biobase
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
```

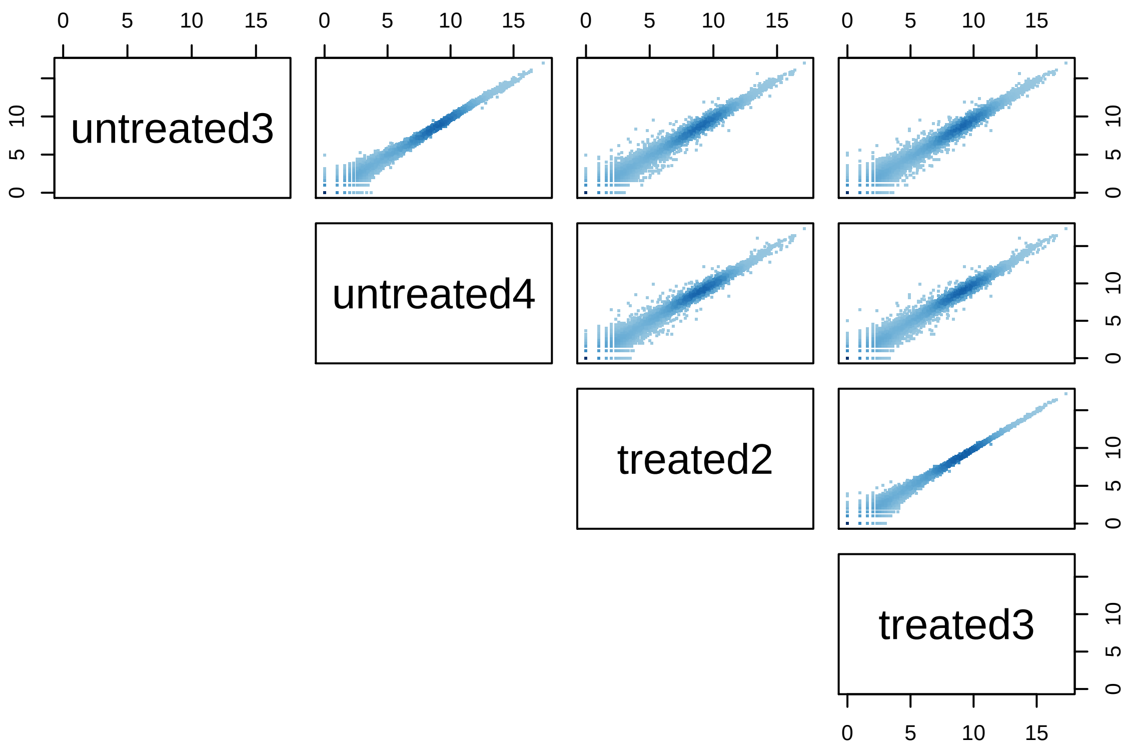
```
library(affy)
plotDensity(log2(count.table + 1), lty=1, col=col, lwd=2)
grid()
legend("topright", legend=names(count.table), col=col, lwd=2)
```



Beware: the R function *plotDensity()* does not display the actual distribution of your values, but a polynomial fit. The representation thus generally looks smoother than the actual data. It is important to realize that, in some particular cases, the fit can lead to extrapolated values which can be misleading.

Let's have a look at the scatter plots using the pairs() function.

```
plotFun <- function(x,y){ dns <- densCols(x,y); points(x,y, col=dns, pch=".") }
pairs(log2(count.table + 1), panel=plotFun, lower.panel = NULL)
```



The command *pairs()* draws a scatter plot for each pair of columns of the input dataset. The plot shows a fairly good reproducibility between samples of the same type (treated or untreated, respectively): all points are aligned along te diagonal, with a relatively wider dispersion at the bottom, corresponding to small number fluctuations.

In contrast, on all the plots comparing an untreated and a treated sample, we can see some points (genes) discarding from the diagonal, on the side of either treated (up-regulated) or untreated (down-regulated) sample.

Creating a DESeqDataSet dataset

Now that we have all the required material, we will create a **DESeqDataSet** object (named dds) that will be used by DESeq2 to perform differential expression call.

Install and load the DESeq2 library.

Create a DESeqDataSet using the **DESeqDataSetFromMatrix()** function and get some help about this object. Have a look at some important accessor methods: **counts**, **conditions**, **estimateSizeFactors**, **sizeFactors**, **estimateDispersions** and **nbinomTest**.

```
## Install the library if needed then load it
if(!require("DESeq2")){
  source("http://bioconductor.org/biocLite.R")
  biocLite("DESeq2")
}
```

```
## Loading required package: DESeq2
## Loading required package: GenomicRanges
## Loading required package: IRanges
## Loading required package: XVector
## Loading required package: Rcpp
## Loading required package: RcppArmadillo
```

```
library("DESeq2")

## Use the DESeqDataSetFromMatrix to create a DESeqDataSet object
dds <-  DESeqDataSetFromMatrix(count.table, conditions, formula(~ cond.type))
show(dds)
```

```
## class: DESeqDataSet
## dim: 12359 4
## exptData(0):
## assays(1): counts
## rownames(12359): FBgn0000003 FBgn0000008 ... FBgn0261574
##   FBgn0261575
## rowData metadata column names(0):
## colnames(4): 1 2 3 4
## colData names(1): cond.type
## What kind of object is it ?
is(dds)
```

```
## [1] "DESeqDataSet"         "SummarizedExperiment"
```

```
isS4(dds)
```

```
## [1] TRUE
```

```
## What does it contain ?
# The list of slot names
slotNames(dds)
```

```
## [1] "design"             "dispersionFunction" "exptData"
## [4] "rowData"            "colData"            "assays"
```

```
## Get some help about the "CountDataSet" class.
## NOT RUN
#?"DESeqDataSet-class"
```

Normalization

The normalization procedure (RLE) is implemented through the **estimateSizeFactors** function.

How is it computed ?

From DESeq help files: Given a matrix or data frame of count data, this function estimates the size factors as follows: Each column is divided by the **geometric means** of the rows. The **median** (or, if requested, another location estimator) **of these ratios** (skipping the genes with a geometric mean of zero) is used as the size factor for this column.

```
### Let's implement such a function
### cds is a countDataset
estimSf <- function (cds){
    # Get the count matrix
    cts <- counts(cds)

    # Compute the geometric mean
    geomMean <- function(x) prod(x)^(1/length(x))

    # Compute the geometric mean over the line
    gm.mean  <-  apply(cts, 1, geomMean)

    # Zero values are set to NA (avoid subsequentcdsdivision by 0)
    gm.mean[gm.mean == 0] <- NA

    # Divide each line by its corresponding geometric mean
    # sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)
    # MARGIN: 1 or 2 (line or columns)
    # STATS: a vector of length nrow(x) or ncol(x), depending on MARGIN
    # FUN: the function to be applied
    cts <- sweep(cts, 1, gm.mean, FUN="/")

    # Compute the median over the columns
    med <- apply(cts, 2, median, na.rm=TRUE)

    # Return the scaling factor
    return(med)
}
```

Now, check that the results obtained with our function are the same as those produced by DESeq. The method associated with normalization for the "CountDataSet" class is **estimateSizeFactors()**.

```
## Normalizing using the method for an object of class"CountDataSet"
dds.norm <-  estimateSizeFactors(dds)
sizeFactors(dds.norm)
```

```
##         1         2         3         4
## 0.8730966 1.0106112 1.0224517 1.1145888
```

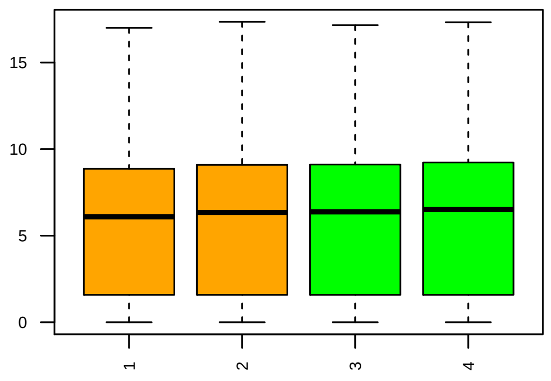
```
## Now get the scaling factor with our homemade function.cds.norm
estimSf(dds)
```

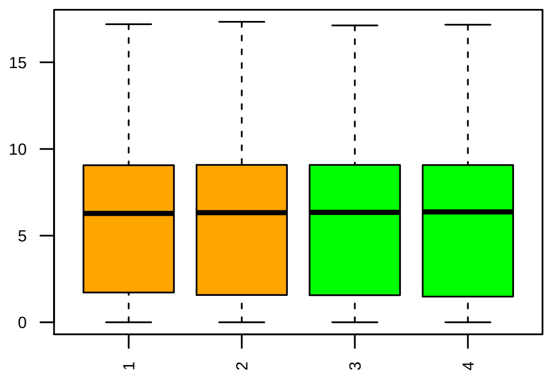
```
##         1         2         3         4
## 0.8730966 1.0106112 1.0224517 1.1145888
```

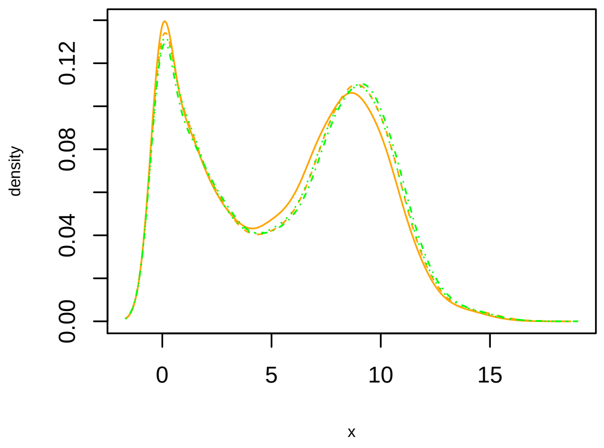
```
round(estimSf(dds),6) == round(sizeFactors(dds.norm), 6)
```

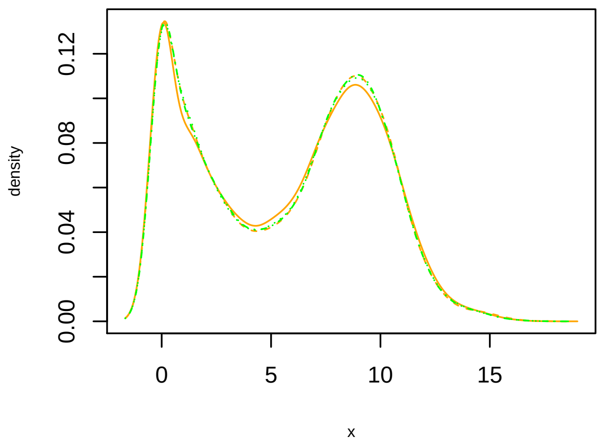
```
##    1    2    3    4
## TRUE TRUE TRUE TRUE
```

```
## Checking the normalization
par(mfrow=c(2,2),cex.lab=0.7)
boxplot(log2(counts(dds.norm)+1),  col=col, cex.axis=0.7, las=2)
boxplot(log2(counts(dds.norm, normalized=TRUE)+1),  col=col,, cex.axis=0.7, las=2)
plotDensity(log2(counts(dds.norm)+1),  col=col, cex.lab=0.7)
plotDensity(log2(counts(dds.norm, normalized=TRUE)+1),  col=col, cex.lab=0.7)
```









Differential expression

In this section we will search for genes whose expression is affected by the si-RNA treatment.

Some intuition about the problem

Let us imagine that we would produce a lot of RNA-Seq experiments from the same samples (technical replicates). For each gene g the measured read counts would be expected to vary rather slighlty around the expected mean and would be probably well modeled using a Poisson distribution. However, when working with biological replicates more variations are intrinsically expected. Indeed, the measured expression values for each genes are expected to fluctuate more importantly, due to the combination of biological and technical factors: inter-individual variations in gene regulation, sample purity, cell-synchronization issues or reponses to environment (e.g. heat-shock).

The Poisson distribution has only one parameter indicating its expected mean : λ. The variance of the distribution equals its mean λ. Thus in most cases, the Poisson distribution is not expected to fit very well with the count distribution in biological replicates, since we expect some over-dispersion (greater variability) due to biological noise.

As a consequence, when working with RNA-Seq data, many of the current approaches for differential expression call rely on an alternative distribution: the *negative binomial* (note that this holds true also for other -Seq approaches, e.g. ChIP-Seq with replicates).

What is the negative binomial ?

The negative binomial distribution is a discrete distribution that can be used to model over-dispersed data (in this case this overdispersion is relative to the poisson model). There are two ways to parametrize the negative binomial distribution.

The probability of x failures before n success First, given a Bernouilli trial with a probability p of success, the **negative binomial** distribution describes the probability of observing x failures before a target number of successes n is reached. In this case the parameters of the distribution will thus be p, n (in **dnbinom()** function of R, n and p are denoted by arguments **size** and **prob** respectively).

$$P_{NegBin}(x;n,p) = \binom{x+n-1}{x} \cdot p^n \cdot (1-p)^x = C^x_{x+n-1} \cdot p^n \cdot (1-p)^x$$

In this formula, p^n denotes the probability to observe n successes, $(1-p)^x$ the probability of x failures, and the binomial coefficient C^x_{x+n-1} indicates the number of possible ways to dispose x failures among the $x+n-1$ trials that precede the last one (the problem statement imposes for the last trial to be a success).

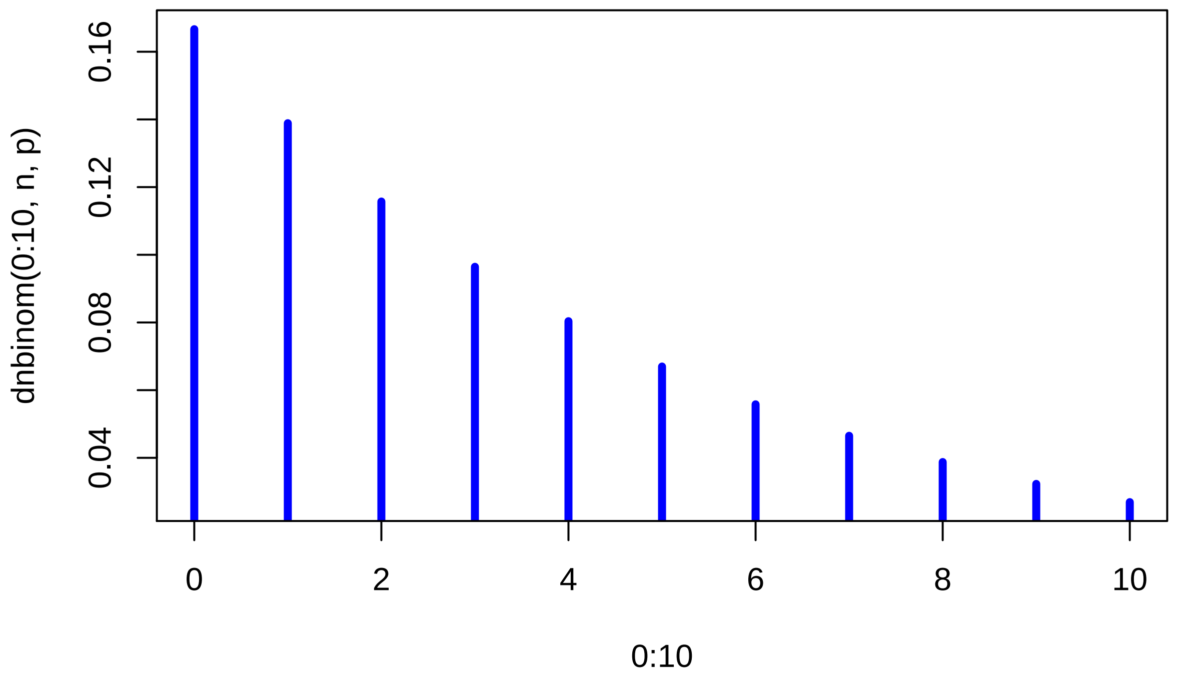
The negative binomial distribution has expected value $n\frac{q}{p}$ and variance $n\frac{q}{p^2}$. Some examples of using this distribution in R are provided below.

Particular case: when $n = 1$ the negative binomial corresponds to the the **geometric distribution**, which models the probability distribution to observe the first success after x failures: $P_{NegBin}(x;1,p) = P_{geom}(x;p) = p \cdot (1-p)^x$.

```
par(mfrow=c(1,1))

## Some intuition about the negative binomiale parametrized using n and p.
## The simple case, one success (see geometric distribution)
# Let's have a look at the density
p <- 1/6 # the probability of success
n <- 1   # target for number of successful trials

# The density function
plot(0:10, dnbinom(0:10, n, p), type="h", col="blue", lwd=4)
```



```r
# the probability of zero failure before one success.
# i.e the probability of success
dnbinom(0, n , p)
```

```
## [1] 0.1666667
```

```r
## i.e the probability of at most 5 failure before one success.
sum(dnbinom(0:5, n , p)) # == pnbinom(5, 1, p)
```

```
## [1] 0.665102
```

```r
## The probability of at most 10 failures before one sucess
sum(dnbinom(0:10, n , p)) # == pnbinom(10, 1, p)
```

```
## [1] 0.865412
```

```r
## The probability to have more than 10 failures before one sucess
1-sum(dnbinom(0:10, n , p)) # == 1 - pnbinom(10, 1, p)
```

```
## [1] 0.134588
```

```r
## With two successes
## The probability of x failure before two success (e.g. two six)
n <- 2
plot(0:30, dnbinom(0:30, n, p), type="h", col="blue", lwd=2)

# Expected value
q <- 1-p
(ev <- n*q/p)
```

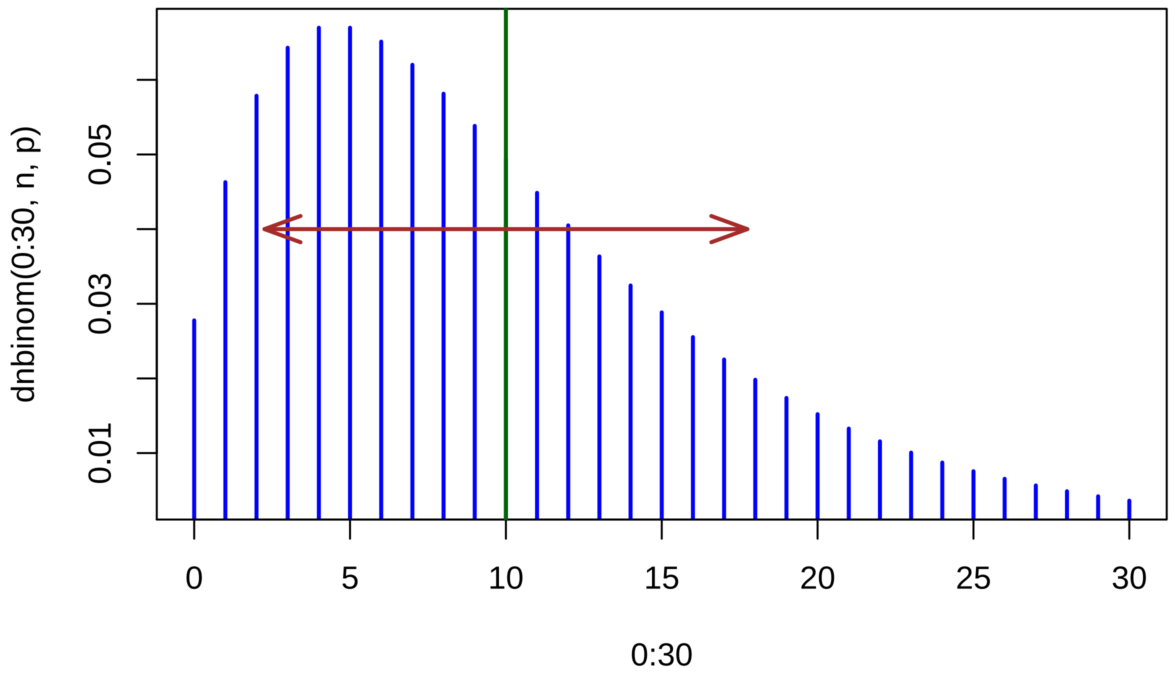
```
## [1] 10
```

```
abline(v=ev, col="darkgreen", lwd=2)

# Variance
(v <- n*q/p^2)
```

```
## [1] 60
```

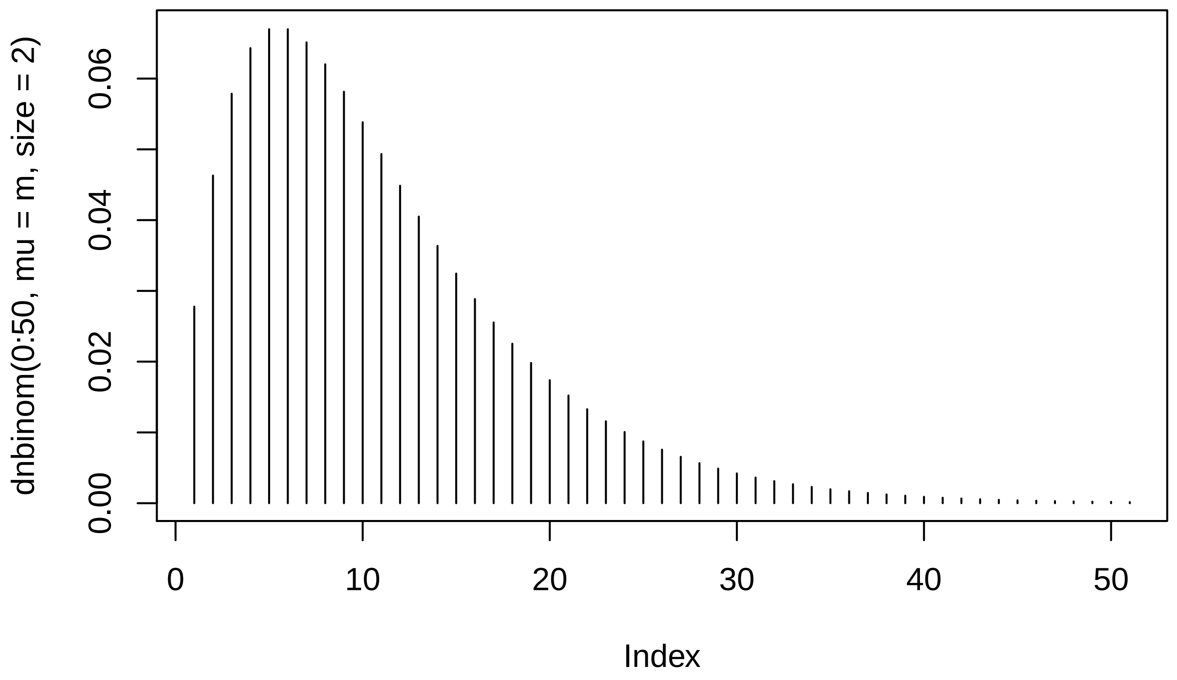
```
arrows(x0=ev-sqrt(v), y0 = 0.04, x1=ev+sqrt(v), y1=0.04, col="brown",lwd=2, code=3, , length=0.2, angle=
```



Using mean and dispersion The second way of parametrizing the distribution is using the mean value m and the dispersion parameter r (in **dnbinom()** function of R, m and r are denoted by arguments **mu** and **size** respectively). The variance of the distribution can then be computed as $m + m^2/r$.

This second parameterization is used in DESeq as it will use it in an attempt to model the variance for each gene given a particular mean.

```
## The density function with m=10 and r=2
m <- 10
r <- 2
plot(dnbinom(0:50, mu=m, size=2), type="h")
```



```
## Note that the same results can be obtained using both parametrizations
n <- r <- 10
p <- 1/6
q <- 1-p
ev <- n*q/p

all(dnbinom(0:50, mu=ev, size=n) == dnbinom(0:50, size=n, prob=p))
```

```
## [1] TRUE
```

Modelling read counts through a negative binomial

To perform diffential expression call DESeq2 will assume that, for each gene, the read counts are generated by a negative binomial distribution. One problem here will be to estimate, for each gene, the two parameters for the corresponding negative binomial distribution: its mean and dispersion.

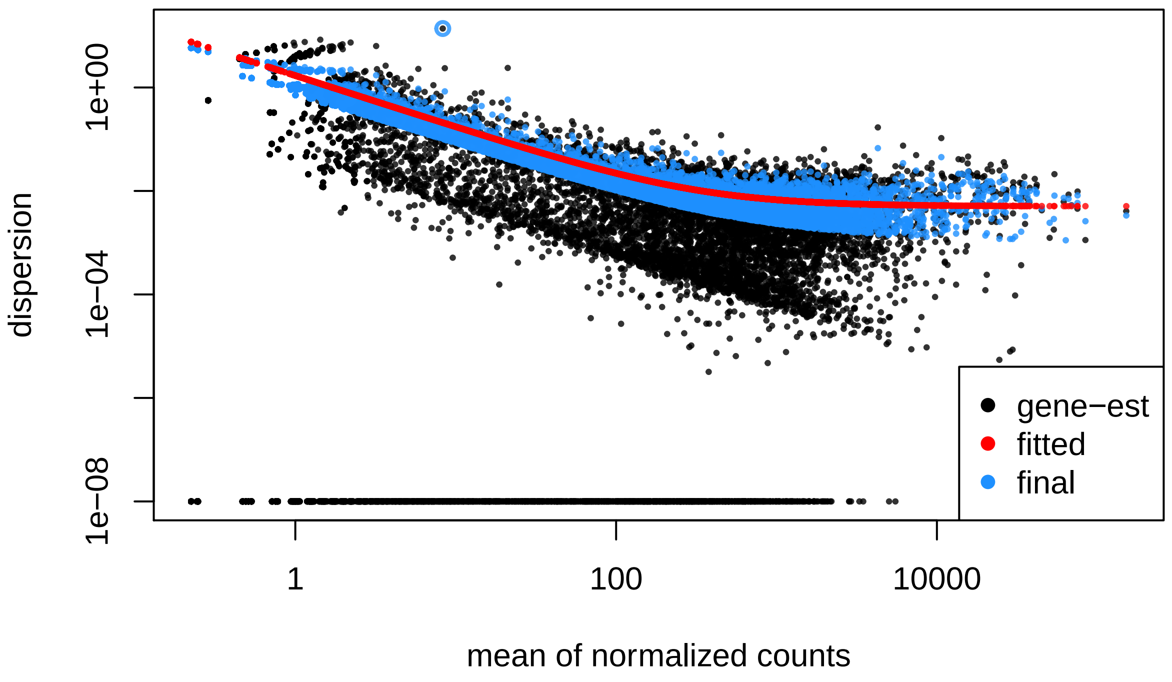
The first step will be to estimate a gene-wise dispersion. When only a few samples are available to compute accurately the variance of each gene, DESeq2 will perform some schrinkage, i.e. assume that genes of similar expression level (counts) have similar variance (note that this is a strong assumption). DESeq2 will regress the gene-wise dispersion onto the means of the normalized counts to obtain an estimate of the dispersion that will be subsequently used to compute build the binomial model for each gene.

To be continued

```
## Performing normalization and negBin parameters in one step
dds <- DESeq(dds)
```

```
## estimating size factors
## estimating dispersions
## gene-wise dispersion estimates
## mean-dispersion relationship
## final dispersion estimates
## fitting model and testing
```

```
## A diagnostic plot which
## shows the mean of normalized counts (x axis)
## and dispersion estimate for each genes
plotDispEsts(dds)
```


dispersion
1e+00
1e−04
1e−08
mean of normalized counts
gene−est
fitted
final