BioC2014: RNA-Seq workflow for differential gene expression

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

[1] Department of Biostatistics, Dana Farber Cancer Institute and
Harvard School of Public Health, Boston, US;
[2] European Molecular Biology Laboratory (EMBL), Heidelberg, Germany
[*]michaelisaiahlove (at) gmail.com

July 29, 2014

Abstract

This lab will walk you through an end-to-end RNA-Seq differential expression workflow. We will start from the FASTQ files, align to the reference genome, prepare gene expression values as a count table by counting the sequenced fragments, perform differential gene expression analysis, and visually explore the results.

This lab covers the basic introduction. For a more in-depth explanation of the advanced details, we advise you to proceed to the vignette of the *DESeq2*, *Differential analysis of count data – the* DESeq2 *package*. For a treatment of exon-level differential expression, we refer to the vignette of the *DEXSeq* package, *Analyzing RNA-seq data for differential exon usage with the* DEXSeq *package*.

Contents

1 Input data

As example data for this lab, we will use publicly available data from the article by Felix Haglund et al., *Evidence of a Functional Estrogen Receptor in Parathyroid Adenomas*, J Clin Endocrin Metab, Sep 2012[1].

The purpose of the experiment was to investigate the role of the estrogen receptor in parathyroid tumors. The investigators derived primary cultures of parathyroid adenoma cells from 4 patients. These primary cultures were treated with diarylpropionitrile (DPN), an estrogen receptor β agonist, or with 4-hydroxytamoxifen (OHT). RNA was extracted at 24 hours and 48 hours from cultures under treatment and control. The blocked design of the experiment allows for statistical analysis of the treatment effects while controlling for patient-to-patient variation.

Part of the data from this experiment is provided in the Bioconductor data package *parathyroidSE*.

1.1 Preparing count matrices

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,

[1] http://www.ncbi.nlm.nih.gov/pubmed/23024189

please do not supply other quantities, such as (rounded) normalized counts, or counts of covered base pairs – this will only lead to nonsensical results.

1.2 Aligning reads to a reference

The computational analysis of an RNA-Seq experiment begins earlier: what we get from the sequencing machine is a set of FASTQ files that contain the nucleotide sequence of each read and a quality score at each position. These reads must first be aligned to a reference genome or transcriptome. It is important to know if the sequencing experiment was single-end or paired-end, as the alignment software will require the user to specify both FASTQ files for a paired-end experiment. The output of this alignment step is commonly stored in a file format called BAM.

A number of software programs exist to align reads to the reference genome, and the development is too rapid for this document to provide a current list. We recommend consulting benchmarking papers that discuss the advantages and disadvantages of each software, which include accuracy, ability to align reads over splice junctions, speed, memory footprint, and many other features.

Here we use the TopHat2 spliced alignment software[2] [1] in combination with the Bowtie index available at the Illumina iGenomes page[3]. For full details on this software and on the iGenomes, users should follow the links to the manual and information provided in the links in the footnotes. For example, the paired-end RNA-Seq reads for the *parathyroidSE* package were aligned using TopHat2 with 8 threads, with the call:

```
tophat2 -o file_tophat_out -p 8 genome file_1.fastq file_2.fastq
```

The BAM files for a number of sequencing runs can then be used to generate count matrices, as described in the following section.

1.3 Example BAM files

Besides the main count table, which we will use later, the *parathyroidSE* package also contains a small subset of the raw data from the Haglund et al. experiment, namely three BAM file each with a subset of the reads from three of the samples. We will use these files to demonstrate how a count table can be constructed from BAM files. Afterwards, we will load the full count table corresponding to all samples and all data, which is already provided in the same package, and will continue the analysis with that full table.

We load the data package with the example data

```
library( "parathyroidSE" )
```

The R function `system.file` can be used to find out where on your computer the files from a package have been installed. Here we ask for the full path to the `extdata` directory, which is part of the *parathyroidSE* package:

[2] http://tophat.cbcb.umd.edu/
[3] http://tophat.cbcb.umd.edu/igenomes.html

```
extDataDir <- system.file("extdata", package = "parathyroidSE", mustWork = TRUE)
extDataDir
```

```
## [1] "/usr/local/Cellar/r/3.1.0/R.framework/Versions/3.1/Resources/library/parathyroidSE/
```

In this directory, we find the three BAM files (and some other files):

```
list.files( extDataDir )
```

```
## [1] "conversion.txt"              "GSE37211_series_matrix.txt"
## [3] "SRR479052.bam"               "SRR479053.bam"
## [5] "SRR479054.bam"
```

Typically, we have a table with experimental meta data for our samples. For these three files, it is as follows:

sampleName	fileName	treatment	time
Ctrl_24h_1	SRR479052.bam	Control	24h
Ctrl_48h_1	SRR479053.bam	Control	48h
DPN_24h_1	SRR479054.bam	DPN	24h

To avoid mistakes, it is helpful to store such a sample table explicitly in a text file and load it.

Load it with:

```
sampleTable <- read.csv( "/path/to/your/sampleTable.csv", header=TRUE )
```

This is how the sample table should look like

```
sampleTable
```

```
##   sampleName      fileName treatment time
## 1 Ctrl_24h_1 SRR479052.bam   Control  24h
## 2 Ctrl_48h_1 SRR479053.bam   Control  48h
## 3  DPN_24h_1 SRR479054.bam       DPN  24h
```

Using the fileName column in the table, we construct the full paths to the files we want to perform the counting operation on:

```
bamFiles <- file.path( extDataDir, sampleTable$fileName )
```

We can peek into one of the BAM files to see the naming style of the sequences (chromosomes). Here we use the `BamFile` function from the *Rsamtools* package.

```
library( "Rsamtools" )
seqinfo( BamFile( bamFiles[1] ) )
```

```
## Seqinfo of length 25
## seqnames seqlengths isCircular genome
## 1         249250621       <NA>   <NA>
## 2         243199373       <NA>   <NA>
## 3         198022430       <NA>   <NA>
```

```
## 4         191154276       <NA>   <NA>
## 5         180915260       <NA>   <NA>
## ...             ...        ...    ...
## 21         48129895       <NA>   <NA>
## 22         51304566       <NA>   <NA>
## X         155270560       <NA>   <NA>
## Y          59373566       <NA>   <NA>
## MT            16569       <NA>   <NA>
```

We want to make sure that these sequence names are the same style as that of the gene models we will obtain in the next section.

1.4 Counting reads in genes

To count how many read map to each gene, we need transcript annotation. Download the current GTF file with human gene annotation from Ensembl. (In case the network is too slow for that, use the truncated version of this file, called `Homo_sapiens.GRCh37.75.subset.gtf.gz`, which we have placed on the course server.)

From this file, the function `makeTranscriptDbFromGFF` from the *GenomicFeatures* constructs a database of all annotated transcripts.

```
library( "GenomicFeatures" )
hse <- makeTranscriptDbFromGFF( "/path/to/your/genemodel_file.GTF", format="gtf" )
exonsByGene <- exonsBy( hse, by="gene" )
```

```
## Warning:  Infering Exon Rankings.  If this is not what you expected, then please
be sure that you have provided a valid attribute for exonRankAttributeName
## Warning:  None of the strings in your circ_seqs argument match your seqnames.
```

In the last step, we have used the `exonsBy` function to bring the transcriptome data base into the shape of a list of all genes,

```
exonsByGene
```

```
## GRangesList of length 100:
## $ENSG00000000003
## GRanges with 13 ranges and 2 metadata columns:
##        seqnames               ranges strand   |   exon_id   exon_name
##           <Rle>            <IRanges>  <Rle>   | <integer> <character>
##    [1]        X [99883667, 99884983]      -   |      3038        <NA>
##    [2]        X [99885756, 99885863]      -   |      3039        <NA>
##    [3]        X [99887482, 99887565]      -   |      3040        <NA>
##    [4]        X [99887538, 99887565]      -   |      3041        <NA>
##    [5]        X [99888402, 99888536]      -   |      3042        <NA>
##    ...      ...                  ...    ... ...       ...         ...
```

```
##    [9]        X [99890555, 99890743]      -   |      3046        <NA>
##   [10]        X [99891188, 99891686]      -   |      3047        <NA>
##   [11]        X [99891605, 99891803]      -   |      3048        <NA>
##   [12]        X [99891790, 99892101]      -   |      3049        <NA>
##   [13]        X [99894942, 99894988]      -   |      3050        <NA>
##
## ...
## <99 more elements>
## ---
## seqlengths:
##   7 12  2  6 16  4  3  1 17  8 19  X 11  9 20
##  NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Exercise 1. Note the warnings issued by `makeTranscriptDbFromGFF`. Can we safely ignore them?

Exercise 2. In `exonsByGene`, inspect the genomic intervals given for the exons of the first gene. Note that they are not disjoint (they overlap). Why? Will this influence results in the following step?

After these preparations, the actual counting is easy. The function `summarizeOverlaps` from the *GenomicAlignments* package will do this.

```
library( "GenomicAlignments" )
se <- summarizeOverlaps( exonsByGene, BamFileList( bamFiles ), mode="Union",
        singleEnd=FALSE, ignore.strand=TRUE, fragments=TRUE )
```

We use the counting mode `"Union"`, which indicates that those reads which overlap any portion of exactly one feature are counted. For more information on the different counting modes, see the help page for `summarizeOverlaps`. As this experiment produced paired-end reads, we specify `singleEnd = FALSE`. As protocol was not strand-specific, we specify `ignore.strand = TRUE`. `fragments = TRUE` indicates that we also want to count reads with unmapped pairs. This last argument is only for use with paired-end experiments.

Details on how to read from the BAM files can be specified using the `BamFileList` function. For example, to control the memory, we could have specified that batches of 2000000 reads should be read at a time:

```
BamFileList( bamFiles, yieldSize = 2000000 )
```

Remember that we have only used a small subset of reads from the original experiment: for 3 samples and for 100 genes. Nevertheless, we can still investigate the resulting *SummarizedExperiment* by looking at the counts in the `assay` slot, the phenotypic data about the samples in `colData` slot (in this case an empty *DataFrame*), and the data about the genes in the `rowData` slot. Figure 1 explains the basic structure of the *SummarizedExperiment* class.

```
se
```

```
## class: SummarizedExperiment
## dim: 100 3
## exptData(0):
## assays(1): counts
## rownames(100): ENSG00000000003 ENSG00000000005 ... ENSG00000005469
##   ENSG00000005471
## rowData metadata column names(0):
## colnames(3): SRR479052.bam SRR479053.bam SRR479054.bam
## colData names(0):
```

```
head( assay(se) )
```

```
##                 SRR479052.bam SRR479053.bam SRR479054.bam
## ENSG00000000003             0             0             1
## ENSG00000000005             0             0             0
## ENSG00000000419             0             0             0
## ENSG00000000457             0             1             1
## ENSG00000000460             0             0             0
## ENSG00000000938             0             0             0
```

```
colSums( assay(se) )
```

```
## SRR479052.bam SRR479053.bam SRR479054.bam
##            31            21            30
```

```
colData(se)
```

```
## DataFrame with 3 rows and 0 columns
```

```
rowData(se)
```

```
## GRangesList of length 100:
## $ENSG00000000003
## GRanges with 13 ranges and 2 metadata columns:
##        seqnames               ranges strand   |   exon_id   exon_name
##           <Rle>            <IRanges>  <Rle>   | <integer> <character>
##    [1]        X [99883667, 99884983]      -   |      3038        <NA>
##    [2]        X [99885756, 99885863]      -   |      3039        <NA>
##    [3]        X [99887482, 99887565]      -   |      3040        <NA>
##    [4]        X [99887538, 99887565]      -   |      3041        <NA>
##    [5]        X [99888402, 99888536]      -   |      3042        <NA>
##    ...      ...                  ...    ... ...       ...         ...
##    [9]        X [99890555, 99890743]      -   |      3046        <NA>
##   [10]        X [99891188, 99891686]      -   |      3047        <NA>
##   [11]        X [99891605, 99891803]      -   |      3048        <NA>
##   [12]        X [99891790, 99892101]      -   |      3049        <NA>
##   [13]        X [99894942, 99894988]      -   |      3050        <NA>
```

```
##
## ...
## <99 more elements>
## ---
## seqlengths:
##   7 12  2  6 16  4  3  1 17  8 19  X 11  9 20
##  NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Note that the `rowData` slot is a *GRangesList*, which contains all the information about the exons for each gene, i.e., for each row of the count table.

The `colData` slot, so far empty, should contain all the meta data. We hence assign our sample table to it:

```
colData(se) <- DataFrame( sampleTable )
```

We can extract columns from the `colData` using the `$` operator, and we can omit the `colData` to avoid extra keystrokes.

```
colData(se)$treatment
```

```
## [1] Control Control DPN
## Levels: Control DPN
```

```
se$treatment
```

```
## [1] Control Control DPN
## Levels: Control DPN
```

We can also use the sampleName table to name the columns of our data matrix:

```
colnames(se) <- sampleTable$sampleName
head( assay(se) )
```

```
##                 Ctrl_24h_1 Ctrl_48h_1 DPN_24h_1
## ENSG00000000003          0          0         1
## ENSG00000000005          0          0         0
## ENSG00000000419          0          0         0
## ENSG00000000457          0          1         1
## ENSG00000000460          0          0         0
## ENSG00000000938          0          0         0
```

This *SummarizedExperiment* object `se` is then all we need to start our analysis.

1.5 The DESeqDataSet, column metadata, and the design formula

Bioconductor software packages often have a special class of data object, which contains special slots and requirements. The data object class in *DESeq2* is the *DESeqDataSet*, which is built on top of the

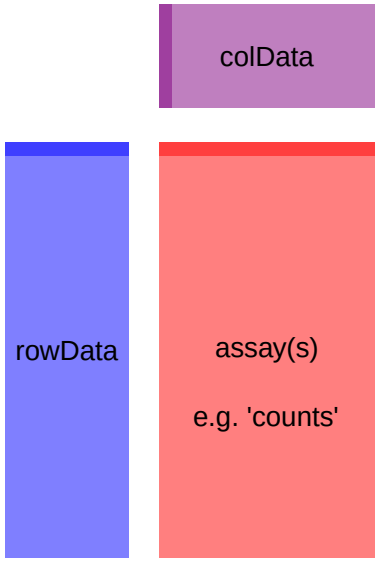

Figure 1: **Diagram of SummarizedExperiment** The component parts of a *SummarizedExperiment* object. The `assay(s)` (red block) contains the matrix (or matrices) of summarized values, the `rowData` (blue block) contains information about the genomic ranges, and the `colData` (purple block) contains information about the samples or experiments. The highlighted line in each block represents the first row (note that the first row of `colData` lines up with the first column of the `assay`.

SummarizedExperiment class. One main differences is that the `assay` slot is instead accessed using the `count` accessor, and the values in this matrix must be non-negative integers.

A second difference is that the *DESeqDataSet* has an associated "design formula". The design is specified at the beginning of the analysis, as it will inform many of the *DESeq2* functions how to treat the samples in the analysis (one exception is the size factor estimation, i. e., the adjustment for differing library sizes, which does not depend on the design formula). The design formula tells which variables in the column metadata table (`colData`) specify the experimental design and how these factors should be used in the analysis.

The simplest design formula for differential expression would be `~ condition`, where `condition` is a column in `colData(dds)` which specifies which of two (or more groups) the samples belong to. For the parathyroid experiment, we will specify `~ patient + treatment`, which means that we want to test for the effect of treatment (the last factor), controlling for the effect of patient (the first factor).

You can use R's formula notation to express any experimental design that can be described within an ANOVA-like framework. Note that *DESeq2* uses the same formula notation as, for instance, the `lm` function of base R. If the question of interest is whether a fold change due to treatment is different across groups, for example across patients, "interaction terms" can be included using models such as

~ patient + treatment + patient:treatment. More complex designs such as these are covered in the other *DESeq2* vignette.

We now use R's data command to load a prepared *SummarizedExperiment* that was generated from the publicly available sequencing data files associated with the Haglund et al. paper, described on page 2. The steps we used to produce this object were equivalent to those you worked through in Section 1.4, except that we used the complete set of samples and all reads.

```
data( "parathyroidGenesSE" )
se <- parathyroidGenesSE
```

A bonus about the workflow we have shown above is that information about the gene models we used is included without extra effort. The rowData for the parathyroid data was obtained with the function makeTranscriptDbFromBiomart, and we can find out all the metadata information about the version of the gene model. The str R function is used to compactly display the structure of the data in the list.

```
str( metadata( rowData(se) ) )
```

```
## List of 1
##  $ genomeInfo:List of 20
##   ..$ Db type                                  : chr "TranscriptDb"
##   ..$ Supporting package                       : chr "GenomicFeatures"
##   ..$ Data source                              : chr "BioMart"
##   ..$ Organism                                 : chr "Homo sapiens"
##   ..$ Resource URL                             : chr "www.biomart.org:80"
##   ..$ BioMart database                         : chr "ensembl"
##   ..$ BioMart database version                 : chr "ENSEMBL GENES 72 (SANGER UK)"
##   ..$ BioMart dataset                          : chr "hsapiens_gene_ensembl"
##   ..$ BioMart dataset description              : chr "Homo sapiens genes (GRCh37.p11)"
##   ..$ BioMart dataset version                  : chr "GRCh37.p11"
##   ..$ Full dataset                             : chr "yes"
##   ..$ miRBase build ID                         : chr NA
##   ..$ transcript_nrow                          : chr "213140"
##   ..$ exon_nrow                                : chr "737783"
##   ..$ cds_nrow                                 : chr "531154"
##   ..$ Db created by                            : chr "GenomicFeatures package from Bioco
##   ..$ Creation time                            : chr "2013-07-30 17:30:25 +0200 (Tue, 30
##   ..$ GenomicFeatures version at creation time: chr "1.13.21"
##   ..$ RSQLite version at creation time         : chr "0.11.4"
##   ..$ DBSCHEMAVERSION                          : chr "1.0"
```

Supposing we have constructed a *SummarizedExperiment* using one of the methods described in the previous section, we now need to make sure that the object contains all the necessary information about the samples, i.e., a table with metadata on the count table's columns stored in the colData slot:

```
colData(se)[1:5,1:4]
```

```
## DataFrame with 5 rows and 4 columns
##           run experiment  patient treatment
##   <character>   <factor> <factor>  <factor>
## 1   SRR479052  SRX140503        1   Control
## 2   SRR479053  SRX140504        1   Control
## 3   SRR479054  SRX140505        1       DPN
## 4   SRR479055  SRX140506        1       DPN
## 5   SRR479056  SRX140507        1       OHT
```

Here we see that this object already contains an informative `colData` slot – because we have already prepared it for you, as described in the *parathyroidSE* vignette. However, when you work with your own data, you will have to add the pertinent sample / phenotypic information for the experiment at this stage. We highly recommend keeping this information in a comma-separated value (CSV) or tab-separated value (TSV) file, which can be exported from an Excel spreadsheet, and the assign this to the `colData` slot, as shown in the previous section.

Make sure that the order of rows in your column data table matches the order of columns in the `assay` data slot.

Once we have our fully annotated *SummarizedExperiment* object, we can construct a *DESeqDataSet* object from it, which will then form the staring point of the actual *DESeq2* package, described in the following sections. Here, we use the *SummarizedExperiment* object we got from the *parathyroidSE* package and augment it by specifying an appropriate design formula.

```
library( "DESeq2" )
ddsFull <- DESeqDataSet( se, design = ~ patient + treatment )
```

Note that there are two alternative functions, `DESeqDataSetFromMatrix` and `DESeqDataSetFromHTSeq`, which allow you to get started in case you have your data not in the form of a *SummarizedExperiment* object, but either as a simple matrix of count values or a s output files from the *htseq-count* script from the *HTSeq* Python package.

1.6 Collapsing technical replicates

There are a number of samples which were sequenced in multiple runs. For example, sample `SRS308873` was sequenced twice. To see, we list the respective columns of the `colData`. (The use of `as.data.frame` forces R to show us the full list, not just the beginning and the end as before.)

```
as.data.frame( colData( ddsFull )[ ,c("sample","patient","treatment","time") ] )
```

```
##       sample patient treatment time
## 1  SRS308865       1   Control  24h
## 2  SRS308866       1   Control  48h
## 3  SRS308867       1       DPN  24h
## 4  SRS308868       1       DPN  48h
## 5  SRS308869       1       OHT  24h
```

```
## 6  SRS308870       1       OHT  48h
## 7  SRS308871       2   Control  24h
## 8  SRS308872       2   Control  48h
## 9  SRS308873       2       DPN  24h
## 10 SRS308873       2       DPN  24h
## 11 SRS308874       2       DPN  48h
## 12 SRS308875       2       OHT  24h
## 13 SRS308875       2       OHT  24h
## 14 SRS308876       2       OHT  48h
## 15 SRS308877       3   Control  24h
## 16 SRS308878       3   Control  48h
## 17 SRS308879       3       DPN  24h
## 18 SRS308880       3       DPN  48h
## 19 SRS308881       3       OHT  24h
## 20 SRS308882       3       OHT  48h
## 21 SRS308883       4   Control  48h
## 22 SRS308884       4       DPN  24h
## 23 SRS308885       4       DPN  48h
## 24 SRS308885       4       DPN  48h
## 25 SRS308886       4       OHT  24h
## 26 SRS308887       4       OHT  48h
## 27 SRS308887       4       OHT  48h
```

We recommend to first add together technical replicates (i.e., libraries derived from the same samples), such that we have one column per sample. We have implemented a convenience function for this, which can take am object, either *SummarizedExperiment* or *DESeqDataSet*, and a grouping factor, in this case the sample name, and return the object with the counts summed up for each unique sample. This will also rename the columns of the object, such that they match the unique names which were used in the grouping factor. Optionally, we can provide a third argument, `run`, which can be used to paste together the names of the runs which were collapsed to create the new object. Note that `dds$variable` is equivalent to `colData(dds)$variable`.

```
ddsCollapsed <- collapseReplicates( ddsFull,
                                    groupby = ddsFull$sample,
                                    run = ddsFull$run )
head( as.data.frame( colData(ddsCollapsed)[ ,c("sample","runsCollapsed") ] ), 12 )
```

```
##              sample       runsCollapsed
## SRS308865 SRS308865           SRR479052
## SRS308866 SRS308866           SRR479053
## SRS308867 SRS308867           SRR479054
## SRS308868 SRS308868           SRR479055
## SRS308869 SRS308869           SRR479056
## SRS308870 SRS308870           SRR479057
## SRS308871 SRS308871           SRR479058
## SRS308872 SRS308872           SRR479059
```

```
## SRS308873 SRS308873 SRR479060,SRR479061
## SRS308874 SRS308874           SRR479062
## SRS308875 SRS308875 SRR479063,SRR479064
## SRS308876 SRS308876           SRR479065
```

We can confirm that the counts for the new object are equal to the summed up counts of the columns that had the same value for the grouping factor:

```
original <- rowSums( counts(ddsFull)[ , ddsFull$sample == "SRS308873" ] )
all( original == counts(ddsCollapsed)[ ,"SRS308873" ] )
```

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

2 Running the DESeq2 pipeline

Here we will analyze a subset of the samples, namely those taken after 48 hours, with either control, DPN or OHT treatment, taking into account the multifactor design.

2.1 Preparing the data object for the analysis of interest

First we subset the relevant columns from the full dataset:

```
dds <- ddsCollapsed[ , ddsCollapsed$time == "48h" ]
```

Sometimes it is necessary to drop levels of the factors, in case that all the samples for one or more levels of a factor in the design have been removed. If time were included in the design formula, the following code could be used to take care of dropped levels in this column.

```
dds$time <- droplevels( dds$time )
```

It will be convenient to make sure that `Control` is the *first* level in the treatment factor, so that the default log2 fold changes are calculated as treatment over control and not the other way around. The function `relevel` achieves this:

```
dds$treatment <- relevel( dds$treatment, "Control" )
```

A quick check whether we now have the right samples:

```
as.data.frame( colData(dds) )
```

```
##                 run experiment patient treatment time submission     study
## SRS308866 SRR479053  SRX140504       1   Control  48h  SRA051611 SRP012167
## SRS308868 SRR479055  SRX140506       1       DPN  48h  SRA051611 SRP012167
## SRS308870 SRR479057  SRX140508       1       OHT  48h  SRA051611 SRP012167
## SRS308872 SRR479059  SRX140510       2   Control  48h  SRA051611 SRP012167
## SRS308874 SRR479062  SRX140512       2       DPN  48h  SRA051611 SRP012167
```

```
## SRS308876 SRR479065  SRX140514       2       OHT  48h  SRA051611 SRP012167
## SRS308878 SRR479067  SRX140516       3   Control  48h  SRA051611 SRP012167
## SRS308880 SRR479069  SRX140518       3       DPN  48h  SRA051611 SRP012167
## SRS308882 SRR479071  SRX140520       3       OHT  48h  SRA051611 SRP012167
## SRS308883 SRR479072  SRX140521       4   Control  48h  SRA051611 SRP012167
## SRS308885 SRR479074  SRX140523       4       DPN  48h  SRA051611 SRP012167
## SRS308887 SRR479077  SRX140525       4       OHT  48h  SRA051611 SRP012167
##               sample       runsCollapsed
## SRS308866 SRS308866           SRR479053
## SRS308868 SRS308868           SRR479055
## SRS308870 SRS308870           SRR479057
## SRS308872 SRS308872           SRR479059
## SRS308874 SRS308874           SRR479062
## SRS308876 SRS308876           SRR479065
## SRS308878 SRS308878           SRR479067
## SRS308880 SRS308880           SRR479069
## SRS308882 SRS308882           SRR479071
## SRS308883 SRS308883           SRR479072
## SRS308885 SRS308885 SRR479074,SRR479075
## SRS308887 SRS308887 SRR479077,SRR479078
```

In order to speed up some annotation steps below, it makes sense to remove genes which have zero counts for all samples. This can be done by simply indexing the `dds` object:

```
dds <- dds[ rowSums( counts(dds) ) > 0 , ]
```

2.2 Running the pipeline

Finally, we are ready to run the differential expression pipeline. Let's recall what design we have specified:

```
design(dds)
```

```
## ~patient + treatment
```

The *DESeq2* analysis modeling counts with patient and treatment effects can now be run with a single call to the function `DESeq`:

```
dds <- DESeq(dds)
```

This function will print out a message for the various steps it performs. These are described in more detail in the manual page for `DESeq`, which can be accessed by typing `?DESeq`. Briefly these are: the estimation of size factors (which control for differences in the library size of the sequencing experiments), the estimation of dispersion for each gene, and fitting a generalized linear model.

A *DESeqDataSet* is returned which contains all the fitted information within it, and the following section describes how to extract out results tables of interest from this object.

2.3 Inspecting the results table

Calling `results` without any arguments will extract the estimated log2 fold changes and p values for the last variable in the design formula. If there are more than 2 levels for this variable – as is the case in this analysis – `results` will extract the results table for a comparison of the last level over the first level. The following section describes how to extract other comparisons.

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

```
## log2 fold change (MAP): treatment OHT vs Control
## Wald test p-value: treatment OHT vs Control
## DataFrame with 32082 rows and 6 columns
##                   baseMean log2FoldChange     lfcSE      stat    pvalue
##                  <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003    613.82       -0.04480    0.0879   -0.5098   0.61017
## ENSG00000000005      0.55       -0.56833    1.0875   -0.5226   0.60127
## ENSG00000000419    304.05        0.11612    0.0962    1.2067   0.22755
## ENSG00000000457    183.52        0.00744    0.1231    0.0604   0.95182
## ENSG00000000460    207.43        0.47084    0.1449    3.2487   0.00116
## ...                   ...            ...       ...       ...       ...
## ENSG00000271699    0.1712        -1.0975     0.910   -1.2065    0.2276
## ENSG00000271704    0.1023        -0.0141     0.716   -0.0198    0.9842
## ENSG00000271707    9.2111        -0.8445     0.483   -1.7477    0.0805
## ENSG00000271709    0.0826        -0.0111     0.678   -0.0163    0.9870
## ENSG00000271711    0.7483        -0.9519     1.066   -0.8934    0.3717
##                       padj
##                  <numeric>
## ENSG00000000003     0.984
## ENSG00000000005        NA
## ENSG00000000419        NA
## ENSG00000000457        NA
## ENSG00000000460        NA
## ...                   ...
## ENSG00000271699        NA
## ENSG00000271704        NA
## ENSG00000271707        NA
## ENSG00000271709        NA
## ENSG00000271711        NA
```

As `res` is a `DataFrame` object, it carries metadata with information on the meaning of the columns:

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

```
## DataFrame with 6 rows and 2 columns
##                        type                                    description
##                 <character>                                    <character>
```

```
## baseMean       intermediate                    the base mean over all rows
## log2FoldChange      results log2 fold change (MAP): treatment OHT vs Control
## lfcSE               results         standard error: treatment OHT vs Control
## stat                results         Wald statistic: treatment OHT vs Control
## pvalue              results      Wald test p-value: treatment OHT vs Control
## padj                results                             BH adjusted p-values
```

The first column, `baseMean`, is a just the average of the normalized count values, dividing by size factors, taken over all samples. The remaining four columns refer to a specific *contrast*, namely the comparison of the levels *DPN* versus *Control* of the factor variable *treatment*. See the help page for `results` (by typing `?results`) for information on how to obtain other contrasts.

The column `log2FoldChange` is the effect size estimate. It tells us how much the gene's expression seems to have changed due to treatment with DPN in comparison to control. This value is reported on a logarithmic scale to base 2: for example, a log2 fold change of 1.5 means that the gene's expression is increased by a multiplicative factor of $2^{1.5} \approx 2.82$.

Of course, this estimate has an uncertainty associated with it, which is available in the column `lfcSE`, the standard error estimate for the log2 fold change estimate. We can also express the uncertainty of a particular effect size estimate as the result of a statistical test. The purpose of a test for differential expression is to test whether the data provides sufficient evidence to conclude that this value is really different from zero. *DESeq2* performs for each gene a *hypothesis test* to see whether evidence is sufficient to decide against the *null hypothesis* that there is no effect of the treatment on the gene and that the observed difference between treatment and control was merely caused by experimental variability (i. e., the type of variability that you can just as well expect between different samples in the same treatment group). As usual in statistics, the result of this test is reported as a p *value*, and it is found in the column `pvalue`. (Remember that a p value indicates the probability that a fold change as strong as the observed one, or even stronger, would be seen under the situation described by the null hypothesis.)

We note that a subset of the p values in `res` are `NA` ("not available"). This is `DESeq`'s way of reporting that all counts for this gene were zero, and hence not test was applied. In addition, p values can be assigned `NA` if the gene was excluded from analysis because it contained an extreme count outlier. For more information, see the outlier detection section of the advanced vignette.

We can examine the counts and normalized counts for the gene with the smallest p value:

```
idx <- which.min(res$pvalue)
counts(dds)[ idx, ]
```

```
## SRS308866 SRS308868 SRS308870 SRS308872 SRS308874 SRS308876 SRS308878 SRS308880
##        95        62        30        24        13        21       276       357
## SRS308882 SRS308883 SRS308885 SRS308887
##       112       388       404       296
```

```
counts(dds, normalized=TRUE)[ idx, ]
```

```
## SRS308866 SRS308868 SRS308870 SRS308872 SRS308874 SRS308876 SRS308878 SRS308880
```

```
##      86.8      61.4      37.4      35.3      16.0      25.9     315.0     201.9
## SRS308882 SRS308883 SRS308885 SRS308887
##     136.9     461.5     269.1     172.0
```

Exercise 3. Use `plot` to examine the normalized counts for the gene with the smallest p-value, i.e., `which.min(res$pvalue)`, over the treatment. Try using `as.integer` on the treatment column to show points instead of boxplots. Try using `as.integer` on the patient column and changing the plotting character (pch) or the color (col) of the points. Try `log="y"` in order to see the differences more clearly. Now turn this into a function which can take any row index as an argument. (Plotting the counts is a useful diagnostic, as such, a helper function `plotCounts` has been added to the next *DESeq2* release in October 2014.)

2.4 Other comparisons

In general, the results for a comparison of any two levels of a variable can be extracted using the `contrast` argument to `results`. The user should specify three values: the name of the variable, the name of the level in the numerator, and the name of the level in the denominator.

First we save the previous results table:

```
resOHT <- res
```

Here we extract results for the log2 of the fold change of DPN / Control.

```
res <- results( dds, contrast = c("treatment", "DPN", "Control") )
res
```

```
## log2 fold change (MAP): treatment DPN vs Control
## Wald test p-value: treatment DPN vs Control
## DataFrame with 32082 rows and 6 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003    613.82        -0.0172    0.0867   -0.1987    0.8425
## ENSG00000000005      0.55        -0.1034    1.0936   -0.0946    0.9246
## ENSG00000000419    304.05        -0.0169    0.0952   -0.1781    0.8587
## ENSG00000000457    183.52        -0.0965    0.1214   -0.7953    0.4264
## ENSG00000000460    207.43         0.3500    0.1438    2.4350    0.0149
## ...                   ...            ...       ...       ...       ...
## ENSG00000271699    0.1712         -1.139     0.925    -1.231     0.218
## ENSG00000271704    0.1023          0.592     0.755     0.785     0.432
## ENSG00000271707    9.2111         -0.510     0.458    -1.114     0.265
## ENSG00000271709    0.0826          0.516     0.711     0.725     0.468
## ENSG00000271711    0.7483         -0.665     1.051    -0.633     0.527
##                      padj
##                 <numeric>
```

```
## ENSG00000000003     0.977
## ENSG00000000005        NA
## ENSG00000000419     0.981
## ENSG00000000457     0.894
## ENSG00000000460     0.276
## ...                   ...
## ENSG00000271699        NA
## ENSG00000271704        NA
## ENSG00000271707        NA
## ENSG00000271709        NA
## ENSG00000271711        NA
```

If results for an interaction term are desired, the `name` argument of `results` should be used. Please see the more advanced vignette for more details.

2.5 Adding gene names

Our result table only uses Ensembl gene IDs, but gene names may be more informative. Bioconductor's annotation packages help with mapping various ID schemes to each other.

We load the annotation package *org.Hs.eg.db*:

```
library( "org.Hs.eg.db" )
```

This is the organism annotation package ("org") for *Homo sapiens* ("Hs"), organized as an *AnnotationDbi* package ("db"), using Entrez Gene IDs ("eg") as primary key.

To get a list of all available key types, use

```
columns(org.Hs.eg.db)
```

```
##  [1] "ENTREZID"     "PFAM"         "IPI"          "PROSITE"      "ACCNUM"
##  [6] "ALIAS"        "CHR"          "CHRLOC"       "CHRLOCEND"    "ENZYME"
## [11] "MAP"          "PATH"         "PMID"         "REFSEQ"       "SYMBOL"
## [16] "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GENENAME"
## [21] "UNIPROT"      "GO"           "EVIDENCE"     "ONTOLOGY"     "GOALL"
## [26] "EVIDENCEALL"  "ONTOLOGYALL"  "OMIM"         "UCSCKG"
```

Converting IDs with the native functions from the *AnnotationDbi* package is currently a bit cumbersome, so we provide the following convenience function (without explaining how exactly it works):

```
convertIDs <- function( ids, fromKey, toKey, db, ifMultiple=c( "putNA", "useFirst" ) ) {
   stopifnot( inherits( db, "AnnotationDb" ) )
   ifMultiple <- match.arg( ifMultiple )
   suppressWarnings( selRes <- AnnotationDbi::select(
      db, keys=ids, keytype=fromKey, columns=c(fromKey,toKey) ) )
   if( ifMultiple == "putNA" ) {
```

```
      duplicatedIds <- selRes[ duplicated( selRes[,1] ), 1 ]
      selRes <- selRes[ ! selRes[,1] %in% duplicatedIds, ] }
   return( selRes[ match( ids, selRes[,1] ), 2 ] )
}
```

This function takes a list of IDs as first argument and their key type as the second argument. The third argument is the key type we want to convert to, the fourth is the *AnnotationDb* object to use. Finally, the last argument specifies what to do if one source ID maps to several target IDs: should the function return an NA or simply the first of the multiple IDs?

To convert the Ensembl IDs in the rownames of `res` to gene symbols and add them as a new column, we use:

```
res$hgnc_symbol <- convertIDs( row.names(res), "ENSEMBL", "SYMBOL", org.Hs.eg.db )
res$entrezid <- convertIDs( row.names(res), "ENSEMBL", "ENTREZID", org.Hs.eg.db )
```

Now the results have the desired external gene ids:

```
head(res, 4)
```

```
## log2 fold change (MAP): treatment DPN vs Control
## Wald test p-value: treatment DPN vs Control
## DataFrame with 4 rows and 8 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000003    613.82        -0.0172    0.0867   -0.1987     0.843
## ENSG00000000005      0.55        -0.1034    1.0936   -0.0946     0.925
## ENSG00000000419    304.05        -0.0169    0.0952   -0.1781     0.859
## ENSG00000000457    183.52        -0.0965    0.1214   -0.7953     0.426
##                      padj hgnc_symbol    entrezid
##                 <numeric> <character> <character>
## ENSG00000000003     0.977      TSPAN6        7105
## ENSG00000000005        NA        TNMD       64102
## ENSG00000000419     0.981        DPM1        8813
## ENSG00000000457     0.894       SCYL3       57147
```

Exercise 4. Go to the Ensembl web site, select the *BioMart* tab, and redo our BioMart query by manually inputting an Ensembl ID and finding the HGNC names and Entrez ids (specifying Filters, Attributes, and then click Results). What other data is available from this mart? Can you modify the code chunk above to add a column `chrom` to the `res` object that tells us for each gene which chromosome it resides on?

3 Further points

3.1 Multiple testing

Novices in high-throughput biology often assume that thresholding these p values at a low value, say 0.01, as is often done in other settings, would be appropriate – but it is not. We briefly explain why:

There are 495 genes with a p value below 0.01 among the 32082 genes, for which the test succeeded in reporting a p value:

```
sum( res$pvalue < 0.01, na.rm=TRUE )
```

```
## [1] 495
```

```
table( is.na(res$pvalue) )
```

```
##
## FALSE
## 32082
```

Now, assume for a moment that the null hypothesis is true for all genes, i.e., no gene is affected by the treatment with DPN. Then, by the definition of p *value*, we expect up to 1% of the genes to have a p value below 0.01. This amounts to 321 genes. If we just considered the list of genes with a p value below 0.01 as differentially expressed, this list should therefore be expected to contain up to $321/495 = 65\%$ false positives!

DESeq2 uses the so-called Benjamini-Hochberg (BH) adjustment; in brief, this method calculates for each gene an *adjusted* p *value* which answers the following question: if one called significant all genes with a p value less than or equal to this gene's p value threshold, what would be the fraction of false positives (the *false discovery rate*, FDR) among them (in the sense of the calculation outlined above)? These values, called the BH-adjusted p values, are given in the column `padj` of the `results` object.

Hence, if we consider a fraction of 10% false positives acceptable, we can consider all genes with an *adjusted* p value below 10%=0.1 as significant. How many such genes are there?

```
sum( res$padj < 0.1, na.rm=TRUE )
```

```
## [1] 249
```

We subset the results table to these genes and then sort it by the log2 fold change estimate to get the significant genes with the strongest down-regulation

```
resSig <- subset(res, res$padj < 0.1 )
head( resSig[ order( resSig$log2FoldChange ), ], 4)
```

```
## log2 fold change (MAP): treatment DPN vs Control
## Wald test p-value: treatment DPN vs Control
## DataFrame with 4 rows and 8 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000163631       233         -0.931     0.284     -3.27  1.06e-03
## ENSG00000119946       152         -0.690     0.157     -4.41  1.04e-05
## ENSG00000041982      1377         -0.686     0.185     -3.72  2.02e-04
```

```
## ENSG00000155111       531         -0.676     0.211     -3.20  1.36e-03
##                      padj hgnc_symbol    entrezid
##                 <numeric> <character> <character>
## ENSG00000163631   0.05817         ALB         213
## ENSG00000119946   0.00246       CNNM1       26507
## ENSG00000041982   0.02045         TNC        3371
## ENSG00000155111   0.06850       CDK19       23097
```

and with the strongest upregulation

```
head( resSig[ order( -resSig$log2FoldChange ), ], 4)
```

```
## log2 fold change (MAP): treatment DPN vs Control
## Wald test p-value: treatment DPN vs Control
## DataFrame with 4 rows and 8 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000092621       559          0.900     0.126      7.16  8.19e-13
## ENSG00000101255       255          0.886     0.169      5.23  1.71e-07
## ENSG00000103257       168          0.826     0.164      5.02  5.06e-07
## ENSG00000156414       143          0.758     0.166      4.56  5.02e-06
##                      padj hgnc_symbol    entrezid
##                 <numeric> <character> <character>
## ENSG00000092621  2.63e-09       PHGDH       26227
## ENSG00000101255  2.35e-04       TRIB3       57761
## ENSG00000103257  3.25e-04      SLC7A5        8140
## ENSG00000156414  1.55e-03       TDRD9      122402
```

3.2 Diagnostic plots

A so-called MA plot provides a useful overview for an experiment with a two-group comparison:

```
plotMA( res, ylim = c(-3, 3) )
```

The plot (Fig. 2) represents each gene with a dot. The x axis is the average expression over all samples, the y axis the log2 fold change between treatment and control. Genes with an adjusted p value below a threshold (here 0.1, the default) are shown in red.

Exercise 5. Use the `identify` function to pick out the results for a single gene in the MA plot. The first argument should be `res$baseMean` and the second argument should be `res$log2FoldChange`. Press *Esc* to finish. R will print out the row number of the gene you selected. Use this to index the `res` object.

This plot demonstrates that only genes with a large average normalized count contain sufficient information to yield a significant call.

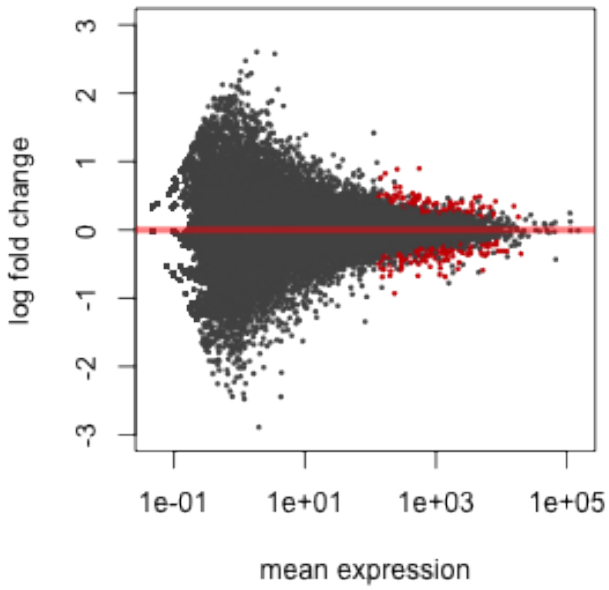

Figure 2: **MA-plot** The MA-plot shows the log2 fold changes from the treatment over the mean of normalized counts, i.e. the average of counts normalized by size factor. The *DESeq2* package incorporates a prior on log2 fold changes, resulting in moderated estimates from genes with low counts and highly variable counts, as can be seen by the narrowing of spread of points on the left side of the plot.

Also note *DESeq2*'s shrinkage estimation of log fold changes (LFCs): When count values are too low to allow an accurate estimate of the LFC, the value is "shrunken" towards zero to avoid that these values, which otherwise would frequently be unrealistically large, dominate the top-ranked log fold changes.

Whether a gene is called significant depends not only on its LFC but also on its within-group variability, which *DESeq2* quantifies as the *dispersion*. For strongly expressed genes, the dispersion can be understood as a squared coefficient of variation: a dispersion value of 0.01 means that the gene's expression tends to differ by typically $\sqrt{0.01} = 10\%$ between samples of the same treatment group. For weak genes, the Poisson noise is an additional source of noise, which is added to the dispersion.

The function `plotDispEsts` visualizes *DESeq2*'s dispersion estimates:

```
plotDispEsts( dds, ylim = c(1e-6, 1e1) )
```

The black points are the dispersion estimates for each gene as obtained by considering the information from each gene separately. Unless one has many samples, these values fluctuate strongly around their true values. Therefore, we fit the red trend line, which shows the dispersions' dependence on the mean, and then shrink each gene's estimate towards the red line to obtain the final estimates (blue points) that are then used in the hypothesis test. The blue circles above the main "cloud" of points are genes which

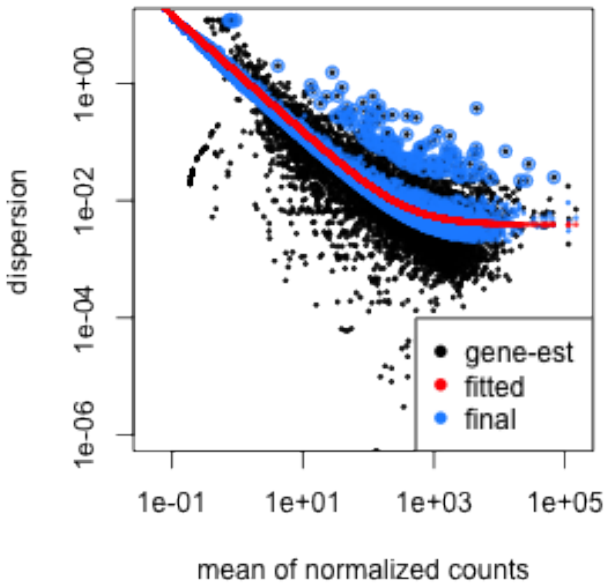

Figure 3: **Plot of dispersion estimates** See text for details

have high gene-wise dispersion estimates which are labelled as dispersion outliers. These estimates are therefore not shrunk toward the fitted trend line.

Another useful diagnostic plot is the histogram of the p values (Fig. 4).

```
hist( res$pvalue, breaks=20, col="grey" )
```

Exercise 6. The p value histogram has some spikes, and one large spike at 1. What happens if you plot the histogram for only those genes with `res$baseMean` greater than 10?

3.3 Independent filtering

The MA plot (Figure 2) highlights an important property of RNA-Seq data. For weakly expressed genes, we have no chance of seeing differential expression, because the low read counts suffer from so high Poisson noise that any biological effect is drowned in the uncertainties from the read counting. We can also show this by examining the ratio of small p values (say, less than, 0.01) for genes binned by mean normalized count:

```
# create bins using the quantile function
qs <- c( 0, quantile( res$baseMean[res$baseMean > 0], 0:7/7 ) )
# "cut" the genes into the bins
bins <- cut( res$baseMean, qs )
```

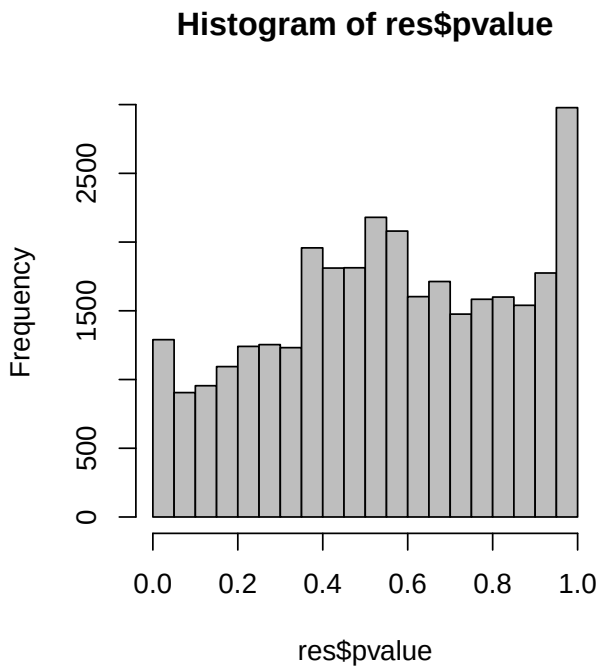

Figure 4: **Histogram** of the p values returned by the test for differential expression

```
# rename the levels of the bins using the middle point
levels(bins) <- paste0("~",round(.5*qs[-1] + .5*qs[-length(qs)]))
# calculate the ratio of p values less than .01 for each bin
ratios <- tapply( res$pvalue, bins, function(p) mean( p < .01, na.rm=TRUE ) )
# plot these ratios
barplot(ratios, xlab="mean normalized count", ylab="ratio of small p values")
```

At first sight, there may seem to be little benefit in filtering out these genes. After all, the test found them to be non-significant anyway. However, these genes have an influence on the multiple testing adjustment, whose performance improves if such genes are removed. By removing the weakly-expressed genes from the input to the FDR procedure, we can find more genes to be significant among those which we keep, and so improved the power of our test. This approach is known as *independent filtering*.

The *DESeq2* software automatically performs independent filtering which maximizes the number of genes which will have adjusted p value less than a critical value (by default, `alpha` is set to 0.1). This automatic independent filtering is performed by, and can be controlled by, the `results` function. We can observe how the number of rejections changes for various cutoffs based on mean normalized count. The following optimal threshold and table of possible values is stored as an *attribute* of the results object.

```
attr(res,"filterThreshold")
```

```
## 70%
```

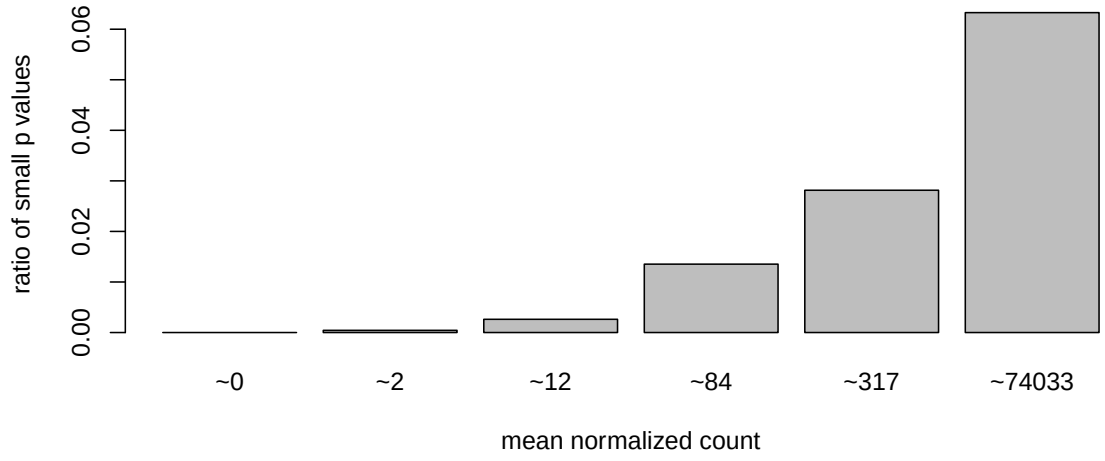

Figure 5: **Ratio of small p values** for groups of genes binned by mean normalized count

```
## 127
plot(attr(res,"filterNumRej"),type="b",
     xlab="quantiles of 'baseMean'",
     ylab="number of rejections")
```

The term *independent* highlights an important caveat. Such filtering is permissible only if the filter criterion is independent of the actual test statistic [3]. Otherwise, the filtering would invalidate the test and consequently the assumptions of the BH procedure. This is why we filtered on the average over *all* samples: this filter is blind to the assignment of samples to the treatment and control group and hence independent.

3.4 Exporting results

Finally, we note that you can easily save the results table in a CSV file, which you can then load with a spreadsheet program such as Excel:

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

3.5 Gene-set enrichment analysis

Do the genes with a strong up- or down-regulation have something in common? We perform next a gene-set enrichment analysis (GSEA) to examine this question.

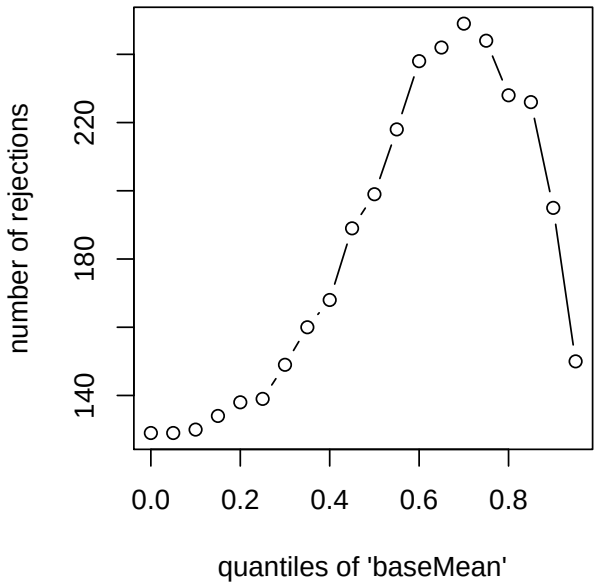

Figure 6: **Independent filtering.** *DESeq2* automatically determines a threshold, filtering on mean normalized count, which maximizes the number of genes which will have an adjusted p value less than a critical value.

As noted in the lecture, gene-set enrichment analysis with RNA-Seq data entails some subtleties. Briefly, a number of different approaches have been proposed that each imply slightly different null hypotheses that are being tested against, and their biologically interpretation differs. This is a topic of ongoing research. We here present a relatively simplistic approach, to demonstrate the basic ideas, but note that a more careful treatment will be needed for more definitive results.

We use the gene sets in the Reactome database

```
library( "reactome.db" )
```

This database works with Entrez IDs, so we will need the `entrezid` column that we added earlier to the `res` object.

First, we subset the results table, `res`, to only those genes for which the Reactome database has data (i.e, whose Entrez ID we find in the respective key column of `reactome.db`) and for which the *DESeq2* test gave an adjusted p value that was not `NA`.

```
res2 <- res[ res$entrezid %in% keys( reactome.db, "ENTREZID" ) &
   !is.na( res$padj ) , ]
head(res2)
```

```
## log2 fold change (MAP): treatment DPN vs Control
## Wald test p-value: treatment DPN vs Control
```

```
## DataFrame with 6 rows and 8 columns
##                  baseMean log2FoldChange     lfcSE      stat    pvalue
##                 <numeric>      <numeric> <numeric> <numeric> <numeric>
## ENSG00000000419       304       -0.01695    0.0952    -0.178     0.859
## ENSG00000001084       312        0.03981    0.1023     0.389     0.697
## ENSG00000001167       399       -0.05600    0.0971    -0.577     0.564
## ENSG00000001630       464        0.07000    0.0968     0.723     0.469
## ENSG00000002822       170       -0.00193    0.1378    -0.014     0.989
## ENSG00000003056       996        0.00772    0.0678     0.114     0.909
##                      padj hgnc_symbol    entrezid
##                 <numeric> <character> <character>
## ENSG00000000419     0.981        DPM1        8813
## ENSG00000001084     0.954        GCLC        2729
## ENSG00000001167     0.936        NFYA        4800
## ENSG00000001630     0.912     CYP51A1        1595
## ENSG00000002822     0.998      MAD1L1        8379
## ENSG00000003056     0.986        M6PR        4074
```

Using `select`, a function from *AnnotationDbi* for querying database objects, we get a table with the mapping from Entrez IDs to Reactome Path IDs

```
reactomeTable <- AnnotationDbi::select( reactome.db,
   keys=as.character(res2$entrezid), keytype="ENTREZID",
   columns=c("ENTREZID","REACTOMEID") )
```

```
## Warning:  'select' and duplicate query keys resulted in 1:many mapping between
## keys and return rows
```

```
head(reactomeTable)
```

```
##   ENTREZID REACTOMEID
## 1     8813     162699
## 2     8813     163125
## 3     8813     392499
## 4     8813     446193
## 5     8813     446203
## 6     8813     446219
```

The next code chunk transforms this table into an *incidence matrix*. This is a Boolean matrix with one row for each Reactome Path and one column for each unique gene in `res2`, which tells us which genes are members of which Reactome Paths. (If you want to understand how this chunk exactly works, read up about the `tapply` function.)

```
incm <- do.call( rbind, with(reactomeTable, tapply(
 ENTREZID, factor(REACTOMEID), function(x) res2$entrezid %in% x ) ))
colnames(incm) <- res2$entrez

str(incm)
```

```
##  logi [1:1458, 1:3400] FALSE FALSE FALSE FALSE FALSE FALSE ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:1458] "1059683" "109581" "109582" "109606" ...
##   ..$ : chr [1:3400] "8813" "2729" "4800" "1595" ...
```

We remove all rows corresponding to Reactome Paths with less than 20 or more than 80 assigned genes.

```
within <- function(x, lower, upper) (x>=lower & x<=upper)
incm <- incm[ within(rowSums(incm), lower=20, upper=80), ]
```

To test whether the genes in a Reactome Path behave in a special way in our experiment, we calculate a number of statistics, including a t-statistic to see whether the average of the genes' log2 fold change values in the gene set is different from zero. To facilitate the computations, we define a little helper function:

```
testCategory <- function( reactomeID ) {
  isMember <- incm[ reactomeID, ]
  data.frame(
     reactomeID  = reactomeID,
     numGenes    = sum( isMember ),
     avgLFC      = mean( res2$log2FoldChange[isMember] ),
     sdLFC       = sd( res2$log2FoldChange[isMember] ),
     zValue      = mean( res2$log2FoldChange[isMember] ) /
                     sd( res2$log2FoldChange[isMember] ),
     strength    = sum( res2$log2FoldChange[isMember] ) / sqrt(sum(isMember)),
     pvalue      = t.test( res2$log2FoldChange[ isMember ] )$p.value,
     reactomeName = reactomePATHID2NAME[[reactomeID]],
     stringsAsFactors = FALSE ) }
```

The function can be called with a Reactome Path ID:

```
testCategory("109606")
```

```
##   reactomeID numGenes  avgLFC  sdLFC zValue strength pvalue
## 1     109606       31 -0.0222 0.0777 -0.285   -0.123  0.123
##                                 reactomeName
## 1 Homo sapiens: Intrinsic Pathway for Apoptosis
```

As you can see the function not only performs the t test and returns the p value but also lists other useful information such as the number of genes in the category, the average log fold change, a "strength" measure (see below) and the name with which Reactome describes the Path.

We call the function for all Paths in our incidence matrix and collect the results in a data frame:

```
reactomeResult <- do.call( rbind, lapply( rownames(incm), testCategory ) )
```

As we performed many tests, we should again use a multiple testing adjustment.

```
reactomeResult$padjust <- p.adjust( reactomeResult$pvalue, "BH" )
```

This is a list of Reactome Paths which are significantly differentially expressed in our comparison of DPN treatment with control, sorted according to sign and strength of the signal:

```
reactomeResultSignif <- reactomeResult[ reactomeResult$padjust < 0.05, ]
head( reactomeResultSignif[ order(-reactomeResultSignif$strength), ] )
```

```
##     reactomeID numGenes  avgLFC  sdLFC zValue strength   pvalue
## 174     191273       21  0.1081 0.0989  1.092    0.495 6.79e-05
## 308     381070       41  0.0706 0.1390  0.508    0.452 2.34e-03
## 60     1638091       30  0.0688 0.1074  0.640    0.377 1.50e-03
## 150    1799339       75  0.0408 0.0779  0.524    0.354 2.15e-05
## 405      72689       64  0.0291 0.0728  0.399    0.232 2.21e-03
## 23     1169091       53 -0.0258 0.0620 -0.416   -0.188 3.80e-03
##                                                              reactomeName
## 174                                    Homo sapiens: Cholesterol biosynthesis
## 308                                Homo sapiens: IRE1alpha activates chaperones
## 60                   Homo sapiens: Heparan sulfate/heparin (HS-GAG) metabolism
## 150 Homo sapiens: SRP-dependent cotranslational protein targeting to membrane
## 405                      Homo sapiens: Formation of a pool of free 40S subunits
## 23                             Homo sapiens: Activation of NF-kappaB in B cells
##     padjust
## 174 0.00784
## 308 0.04160
## 60  0.03297
## 150 0.00332
## 405 0.04113
## 23  0.04887
```

However, as discussed in the lecture, it is highly questionable that a t test is appropriate here. After all, genes in a set are typically correlated, and this violates the assumption of independence that is at the core of a t test. Hence, should we really look at p values from t tests? A p value obtained by sample permutation would solve this issue as sample permutation preserves and so accounts for gene-gene correlation. However, with only four subjects, we do not have enough samples for this.

Hence, it may be more prudent to disregard these questionable p values altogether and instead look at a more primitive statistic, such as simply the average LFC within a path, perhaps divided by the standard deviation. We have stored this above as `zValue`.

```
head( reactomeResult[ order(-reactomeResult$zValue), ] )
```

```
##     reactomeID numGenes avgLFC  sdLFC zValue strength   pvalue
## 174     191273       21 0.1081 0.0989  1.092    0.495 6.79e-05
## 60     1638091       30 0.0688 0.1074  0.640    0.377 1.50e-03
## 229    2426168       26 0.0629 0.1163  0.541    0.321 1.07e-02
## 150    1799339       75 0.0408 0.0779  0.524    0.354 2.15e-05
## 308     381070       41 0.0706 0.1390  0.508    0.452 2.34e-03
## 57      163125       20 0.0356 0.0703  0.507    0.159 3.52e-02
```

```
##                                                                                reactomeName
## 174                                                 Homo sapiens: Cholesterol biosynthesis
## 60                                Homo sapiens: Heparan sulfate/heparin (HS-GAG) metabolism
## 229                             Homo sapiens: Activation of gene expression by SREBF (SREBP)
## 150              Homo sapiens: SRP-dependent cotranslational protein targeting to membrane
## 308                                           Homo sapiens: IRE1alpha activates chaperones
## 57  Homo sapiens: Post-translational modification: synthesis of GPI-anchored proteins
##     padjust
## 174 0.00784
## 60  0.03297
## 229 0.08417
## 150 0.00332
## 308 0.04160
## 57  0.12619
```

If such an analysis is only considered exploratory, we may inspect various such tables and see whether the ranking of Paths helps us make sense of the data. Nevertheless, there is certainly room for improvement here.

4 Exploratory data analysis

4.1 The rlog transform

Many common statistical methods for exploratory analysis of multidimensional data, especially methods for clustering and ordination (e. g., principal-component analysis and the like), work best for (at least approximately) homoskedastic data; this means that the variance of an observable quantity (i.e., here, the expression strength of a gene) does not depend on the mean. In RNA-Seq data, however, variance grows with the mean. For example, if one performs PCA directly on a matrix of normalized read counts, the result typically depends only on the few most strongly expressed genes because they show the largest absolute differences between samples. A simple and often used strategy to avoid this is to take the logarithm of the normalized count values plus a small pseudocount; however, now the genes with low counts tend to dominate the results because, due to the strong Poisson noise inherent to small count values, they show the strongest relative differences between samples.

As a solution, *DESeq2* offers the *regularized-logarithm transformation*, or *rlog* for short. For genes with high counts, the rlog transformation differs not much from an ordinary log2 transformation. For genes with lower counts, however, the values are shrunken towards the genes' averages across all samples. Using an empirical Bayesian prior in the form of a *ridge penalty*, this is done such that the rlog-transformed data are approximately homoskedastic.

Note that the rlog transformation is provided for applications other than differential testing. For differential testing we recommend the `DESeq` function applied to raw counts, as described earlier in this vignette, which also takes into account the dependence of the variance of counts on the mean value during the dispersion estimation step.

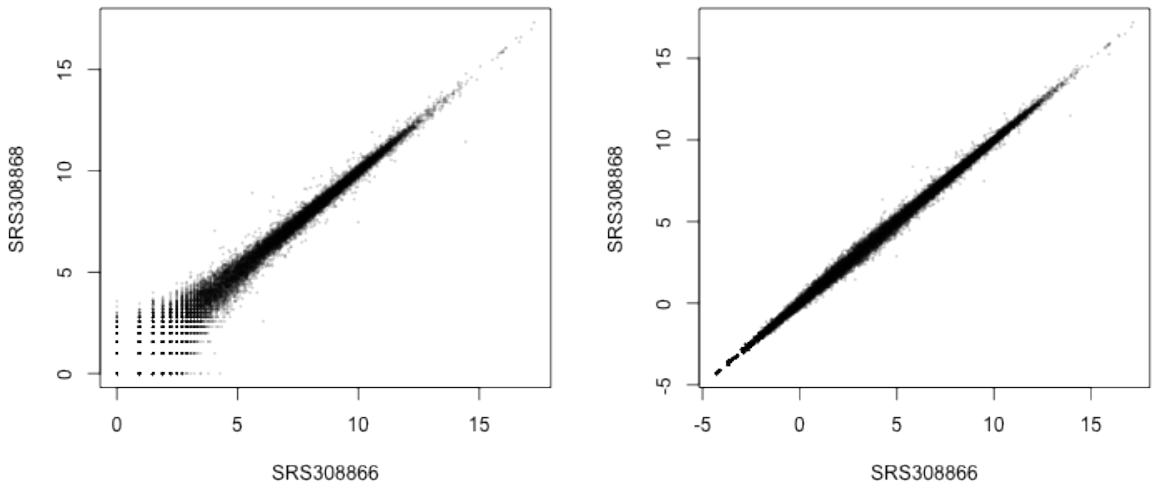

Figure 7: **Scatter plot of sample 2 vs sample 1.** Left: using an ordinary log2 transformation. Right: Using the rlog transformation.

The function `rlog` returns a *SummarizedExperiment* object which contains the rlog-transformed values in its *assay* slot:

```
rld <- rlog( dds )
assay(rld)[ 1:3, 1:3]
```

```
##                 SRS308866 SRS308868 SRS308870
## ENSG00000000003     9.761     9.728     9.865
## ENSG00000000005    -0.981    -0.741    -0.945
## ENSG00000000419     8.065     8.077     8.119
```

To show the effect of the transformation, we plot the first sample against the second, first simply using the `log2` function (after adding 1, to avoid taking the log of zero), and then using the rlog-transformed values.

```
par( mfrow = c( 1, 2 ) )
plot( log2( 1+counts(dds, normalized=TRUE)[, 1:2] ), col="#00000020", pch=20, cex=0.3 )
plot( assay(rld)[, 1:2], col="#00000020", pch=20, cex=0.3 )
```

Note that, in order to make it easier to see where several points are plotted on top of each other, we set the plotting color to a semi-transparent black (encoded as #00000020) and changed the points to solid disks (`pch=20`) with reduced size (`cex=0.3`)[4].

In Figure 7, we can see how genes with low counts seem to be excessively variable on the ordinary logarithmic scale, while the rlog transform compresses differences for genes for which the data cannot provide good information anyway.

[4]The function `heatscatter` from the package *LSD* offers a colorful alternative.

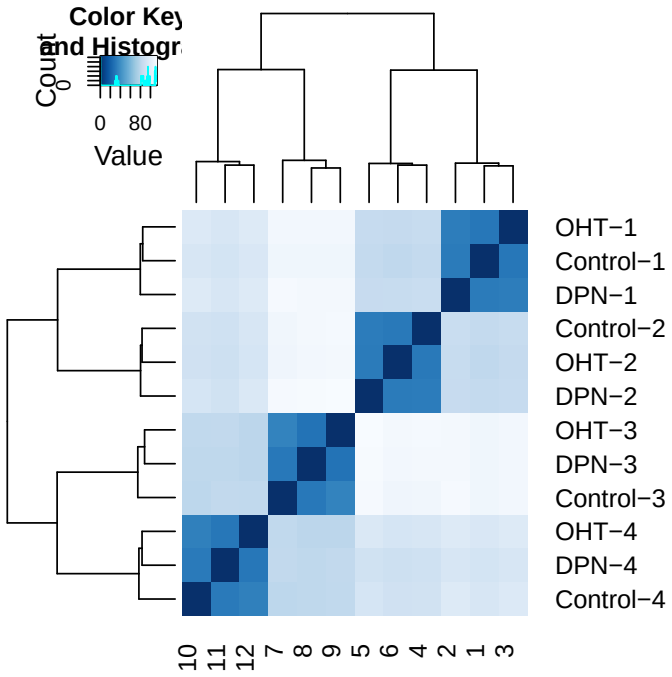

Figure 8: **Heatmap of Euclidean sample distances after rlog transformation.**

4.2 Sample distances

A useful first step in an RNA-Seq analysis is often to assess overall similarity between samples: Which samples are similar to each other, which are different? Does this fit to the expectation from the experiment's design?

We use the R function `dist` to calculate the Euclidean distance between samples. To avoid that the distance measure is dominated by a few highly variable genes, and have a roughly equal contribution from all genes, we use it on the rlog-transformed data:

```
sampleDists <- dist( t( assay(rld) ) )
as.matrix( sampleDists )[ 1:3, 1:3 ]
```

```
##           SRS308866 SRS308868 SRS308870
## SRS308866       0.0      32.9      31.2
## SRS308868      32.9       0.0      33.7
## SRS308870      31.2      33.7       0.0
```

Note the use of the function `t` to transpose the data matrix. We need this because `dist` calculates distances between data *rows* and our samples constitute the columns.

We visualize the distances in a heatmap, using the function `heatmap.2` from the *gplots* package.

```
sampleDistMatrix <- as.matrix( sampleDists )
rownames(sampleDistMatrix) <- paste( rld$treatment,
```

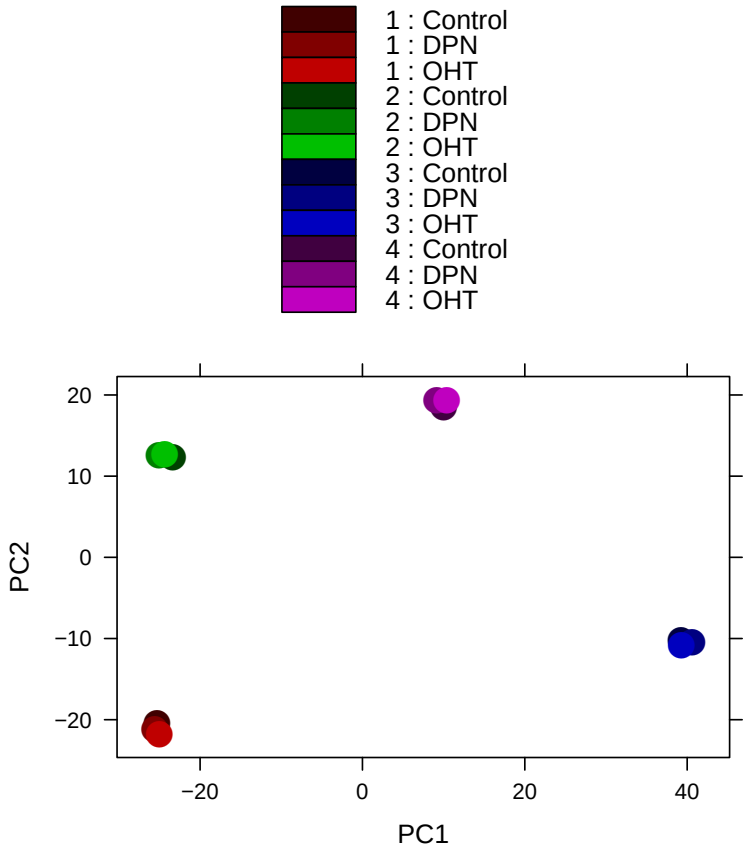

Figure 9: **Principal components analysis (PCA)** of samples after rlog transformation.

```
   rld$patient, sep="-" )
colnames(sampleDistMatrix) <- NULL
library( "gplots" )
library( "RColorBrewer" )
colours = colorRampPalette( rev(brewer.pal(9, "Blues")) )(255)
heatmap.2( sampleDistMatrix, trace="none", col=colours)
```

Note that we have changed the row names of the distance matrix to contain treatment type and patient number instead of sample ID, so that we have all this information in view when looking at the heatmap (Fig. 8).

Another way to visualize sample-to-sample distances is a principal-components analysis (PCA). In this ordination method, the data points (i.e., here, the samples) are projected onto the 2D plane such that they spread out optimally (Fig. 9).

```
colours <- c(rgb(1:3/4,0,0),rgb(0,1:3/4,0),rgb(0,0,1:3/4),rgb(1:3/4,0,1:3/4))
plotPCA( rld, intgroup = c("patient","treatment"), col=colours )
```

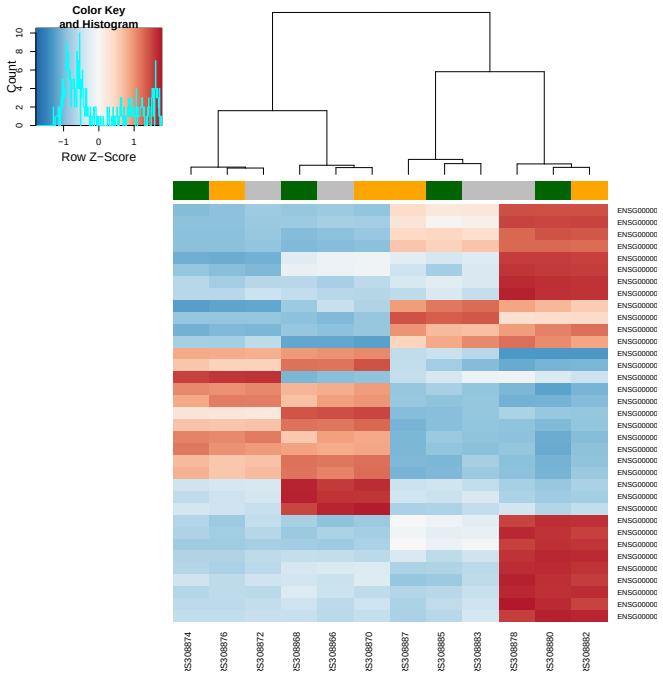

Figure 10: **Heatmap with gene clustering.**

Here, we have used the function `plotPCA` which comes with *DESeq2*. The two terms specified as `intgroup` are column names from our sample data; they tell the function to use them to choose colours.

From both visualizations, we see that the differences between patients is much larger than the difference between treatment and control samples of the same patient. This shows why it was important to account for this paired design ("paired", because each treated sample is paired with one control sample from the *same* patient). We did so by using the design formula `~ patient + treatment` when setting up the data object in the beginning. Had we used an un-paired analysis, by specifying only `~ treatment`, we would not have found many hits, because then, the patient-to-patient differences would have drowned out any treatment effects.

Here, we have performed this sample distance analysis towards the end of our analysis. In practice, however, this is a step suitable to give a first overview on the data. Hence, one will typically carry out this analysis as one of the first steps in an analysis. To this end, you may also find the function `arrayQualityMetrics`, from the package of the same name, useful.

4.3 Gene clustering

In the heatmap of Fig. 8, the dendrogram at the side shows us a hierarchical clustering of the samples. Such a clustering can also be performed for the genes.

Since the clustering is only relevant for genes that actually carry signal, one usually carries it out only for a subset of most highly variable genes. Here, for demonstration, let us select the 35 genes with the highest variance across samples:

```
library( "genefilter" )
topVarGenes <- head( order( rowVars( assay(rld) ), decreasing=TRUE ), 35 )
```

The heatmap becomes more interesting if we do not look at absolute expression strength but rather at the amount by which each gene deviates in a specific sample from the gene's average across all samples. Hence, we center and scale each genes' values across samples, and plot a heatmap.

```
heatmap.2( assay(rld)[ topVarGenes, ], scale="row",
     trace="none", dendrogram="column",
     col = colorRampPalette( rev(brewer.pal(9, "RdBu")) )(255),
     ColSideColors = c( Control="gray", DPN="darkgreen", OHT="orange" )[
        colData(rld)$treatment ] )
```

We can now see (Fig. 10) blocks of genes which covary across patients. Often, such a heatmap is insightful, even though here, seeing these variations across patients is of limited value because we are rather interested in the effects between the treatments from each patient.

Exercise 7. This heatmap shows genes that vary strongest between *patients*. In our differential expression analysis, we found a list of genes that varied significantly between *treatment* and control. Display these genes (or maybe only those showing *up*-regulation upon treatment) in a similar heatmap. Can you confirm the test result from visual inspection of the heatmap?

Exercise 8. In the abstract of the publication for this dataset, 5 parathyroid-related genes are mentioned as differentially expressed due to OHT or DPN treatment in the 48 hour samples compared to control: `c("CASR","VDR","JUN","CALR","ORAI2")`. Using the results table for OHT vs Control and the results table for DPN vs Control, find the rank of these genes in terms of smallest p values using the `match` function.

5 Going further

Exercise 9. Analyze more datasets: use the function defined in the following code chunk to download a processed count matrix from the ReCount website[a].

Suggested datasets:

- *hammer* – spinal nerve ligation of rats at 2 time points (note a misspelling in one of the time points in `colData`). Perform differential expression and describe the most significant gene sets. Do these have something to do with spinal nerve ligation?
- *wang* – human tissue comparison. Make a PCA plot of the samples
- *bottomly* – 2 inbred mouse strains. Make a PCA plot of the samples

Note that the annotation libraries for rat and mouse are *org.Rn.eg.db* and *org.Mm.eg.db*.

[a] http://bowtie-bio.sourceforge.net/recount/

The following function takes a name of the dataset from the ReCount website, e.g. `"hammer"`, and returns a *SummarizedExperiment* object.

```
recount2SE <- function(name) {
  filename <- paste0(name,"_eset.RData")
  if (!file.exists(filename)) download.file(paste0(
    "http://bowtie-bio.sourceforge.net/recount/ExpressionSets/",
    filename),filename)
  load(filename)
  e <- get(paste0(name,".eset"))
  se <- SummarizedExperiment(SimpleList(counts=exprs(e)),
                             colData=DataFrame(pData(e)))
  se
}
```

6 See also

- *DEXSeq* for differential exon usage. See the accompanying vignette, *Analyzing RNA-seq data for differential exon usage with the DEXSeq package*, which is similar to the style of this tutorial.
- Other Bioconductor packages for RNA-Seq differential expression: *edgeR*, *limma*, *DSS*, *BitSeq* (transcript level), *EBSeq*, *cummeRbund* (for importing and visualizing Cufflinks results), *monocle* (single-cell analysis). More at http://bioconductor.org/packages/release/BiocViews.html#___RNASeq
- Methods for gene set testing: `romer` and `roast` in *limma*, permutation based: *safe*
- Packages for normalizing for covariates (e.g., GC content): *cqn*, *EDASeq*
- Packages for detecting batches: *sva*, *RUVSeq*
- Generating HTML results tables with links to outside resources (gene descriptions): *ReportingTools*

References

[1] Daehwan Kim, Geo Pertea, Cole Trapnell, Harold Pimentel, Ryan Kelley, and Steven Salzberg. TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biology*, 14(4):R36+, 2013. URL: http://dx.doi.org/10.1186/gb-2013-14-4-r36, doi:10.1186/gb-2013-14-4-r36.

[2] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009. URL: http://bioinformatics.oxfordjournals.org/content/25/16/2078.abstract, doi:10.1093/bioinformatics/btp352.

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

7 Session Info

As last part of this document, we call the function `sessionInfo`, which reports the version numbers of R and all the packages used in this session. It is good practice to always keep such a record as it will help to trace down what has happened in case that an R script ceases to work because a package has been changed in a newer version. The session information should also always be included in any emails to the Bioconductor mailing list.

- R version 3.1.0 (2014-04-10), `x86_64-apple-darwin12.5.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, utils

- Other packages: AnnotationDbi 1.26.0, Biobase 2.24.0, BiocGenerics 0.10.0, BiocInstaller 1.14.2, Biostrings 2.32.1, BSgenome 1.32.0, DBI 0.2-7, DESeq2 1.4.5, devtools 1.5, genefilter 1.46.1, GenomeInfoDb 1.0.2, GenomicAlignments 1.0.3, GenomicFeatures 1.16.2, GenomicRanges 1.16.3, gplots 2.14.1, IRanges 1.22.9, knitr 1.6, org.Hs.eg.db 2.14.0, parathyroidSE 1.2.0, RColorBrewer 1.0-5, Rcpp 0.11.2, RcppArmadillo 0.4.320.0, reactome.db 1.48.0, Rsamtools 1.16.1, RSQLite 0.11.4, slidify 0.4.5, XVector 0.4.0
- Loaded via a namespace (and not attached): annotate 1.42.1, BatchJobs 1.3, BBmisc 1.7, BiocParallel 0.6.1, BiocStyle 1.2.0, biomaRt 2.20.0, bitops 1.0-6, brew 1.0-6, caTools 1.17, checkmate 1.1, codetools 0.2-8, digest 0.6.4, evaluate 0.5.5, fail 1.2, foreach 1.4.2, formatR 0.10, gdata 2.13.3, geneplotter 1.42.0, grid 3.1.0, gtools 3.4.1, highr 0.3, httr 0.3, iterators 1.0.7, KernSmooth 2.23-12, lattice 0.20-29, locfit 1.5-9.1, markdown 0.7, memoise 0.2.1, RCurl 1.95-4.1, rtracklayer 1.24.2, sendmailR 1.1-2, splines 3.1.0, stats4 3.1.0, stringr 0.6.2, survival 2.37-7, tools 3.1.0, whisker 0.3-2, XML 3.98-1.1, xtable 1.7-3, yaml 2.1.13, zlibbioc 1.10.0