Inferring differential exon usage in RNA-Seq data with the DEXSeq package

Alejandro Reyes, Simon Anders, Wolfgang Huber

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany

This vignette describes version 1.18.4 of the *DEXSeq* package.

Last revision of this document: 2016-03-10

Contents

1 Overview

The Bioconductor package *DEXseq* implements a method to test for differential exon usage in comparative RNA-Seq experiments. By *differential exon usage* (DEU), we mean changes in the relative usage of exons caused by the experimental condition. The relative usage of an exon is defined as

$$\frac{\text{number of transcripts from the gene that contain this exon}}{\text{number of all transcripts from the gene}}. \quad (1)$$

The statistical method used by *DEXSeq* was introduced in our paper [**?**]. The basic concept can be summarized as follows. For each exon (or part of an exon) and each sample, we count how many reads map to this exon and how many reads map to any of the other exons of the same gene. We consider the ratio of these two counts, and how it changes across conditions, to infer changes in the relative exon usage (**??**). In the case of an inner exon, a change in relative exon usage is typically due to a change in the rate with which this exon is spliced into transcripts (alternative splicing). Note, however, that DEU is a more general concept than alternative splicing, since it also includes changes in the usage of alternative transcript start sites and polyadenylation sites, which can cause differential usage of exons at the 5' and 3' boundary of transcripts.

Similar as with differential gene expression, we need to make sure that observed differences of values of the ratio (**??**) between conditions are statistically significant, i. e., are sufficiently unlikely to be just due to random fluctuations such as those seen even between samples from the same condition, i. e., between

replicates. To this end, *DEXSeq* assesses the strength of these fluctuations (quantified by the so-called *dispersion*) by comparing replicates before comparing the averages between the sample groups.

The preceding description is somewhat simplified (and perhaps over-simplified), and we recommend that users consult the paper [**?**] for a more complete description, as well as Appendix **??** of this vignette, which describes how the current implementation of *DEXSeq* differs from the original approach described in the paper. Nevertheless, two important aspects should be mentioned already here: First, *DEXSeq* does not actually work on the ratios (**??**), but on the counts in the numerator and denominator, to be able to make use of the information that is contained in the magnitude of count values. (3000 reads versus 1000 reads is the same ratio as 3 reads versus 1 read, but the latter is a far less reliable estimate of the underlying true value, because of statistical sampling.) Second, *DEXSeq* is not limited to simple two-group comparisons; rather, it uses so-called generalized linear models (GLMs) to permit ANOVA-like analysis of potentially complex experimental designs.

2 Preparations

2.1 Example data

To demonstrate the use of *DEXSeq*, we use the *pasilla* dataset, an RNA-Seq dataset generated by Brooks et al. [**?**]. They investigated the effect of siRNA knock-down of the gene *pasilla* on the transcriptome of fly S2-DRSC cells. The RNA-binding protein *pasilla* protein is thought to be involved in the regulation of splicing. (Its mammalian orthologs, NOVA1 and NOVA2, are well-studied examples of splicing factors.) Brooks et al. prepared seven cell cultures, treated three with siRNA to knock down *pasilla* and left four as untreated controls, and performed RNA-Seq on all samples. They deposited the raw sequencing reads with the NCBI Gene Expression Omnibus (GEO) under the accession number GSE18508.[1]

Executability of the code. Usually, Bioconductor vignettes contain automatically executable code, i. e., you can follow the vignette by directly running the code shown, using functionality and data provided with the package. However, it would not be practical to include the voluminous raw data of the pasilla experiment here. Therefore, the code in this section is not automatically executable. You may download the raw data yourself from GEO, as well as the required extra tools, and follow the work flow shown here and in the *pasilla* vignette [**?**]. From Section **??** on, code is directly executable, as usual. Therefore, we recommend that you just read this section, and try following our analysis in R only from the next section onwards. Once you work with your own data, you will want to come back and adapt the work flow shown here to your data.

2.2 Alignment

The first step of the analysis is to align the reads to a reference genome. It is important to align them to the genome, not to the transcriptome, and to use a splice-aware aligner (i. e., a short-read alignment tool

[1] http://www.ncbi.nlm.nih.gov/projects/geo/query/acc.cgi?acc=GSE18508

that can deal with reads that span across introns) such as TopHat2 [**?**], GSNAP [**?**], or STAR [**?**]. The explanation of the analysis work-flow presented here starts with the aligned reads in the SAM format. If you are unfamiliar with the process of aligning reads to obtain SAM files, you can find a summary how we proceeded in preparing the pasilla data in the vignette for the *pasilla* data package [**?**] and a more extensive explanation, using the same data set, in our protocol article on differential expression calling [**?**].

2.3 HTSeq

The initial steps of a *DEXSeq* analysis, described in the following two sections, is typically done outside R, by using two provided Python scripts. You do not need to know how to use Python; however you have to install the Python package *HTSeq*, following the explanations given on the HTSeq web page:

http://www-huber.embl.de/users/anders/HTSeq/doc/install.html

Once you have installed *HTSeq*, you can use the two Python scripts, `dexseq_prepare_annotation.py` (described in Section **??**) and `dexseq_count.py` (Section **??**), that come with the *DEXSeq* package. If you have trouble finding them, start R and ask for the installation directory with

```
pythonScriptsDir = system.file( "python_scripts", package="DEXSeq" )
list.files(pythonScriptsDir)
```

```
## [1] "dexseq_count.py"              "dexseq_prepare_annotation.py"
```

The displayed path should contain the two files. If it does not, try to re-install *DEXSeq* (as usual, with `biocLite`).

An alternative work flow, which replaces the two Python-based steps with R=based code, is also available and is demonstrated in the vignette of the *parathyroidSE* package [**?**].

2.4 Preparing the annotation

The Python scripts expect a GTF file with gene models for your species. We have tested our tools chiefly with GTF files from Ensembl and hence recommend to prefer these, as files from other providers sometimes do not adhere fully to the GTF standard and cause the preprocessing to fail. Ensembl GTF files can be found in the "FTP Download" sections of the Ensembl web sites (i. e., Ensembl, EnsemblPlants, EnsemblFungi, etc.). Make sure that your GTF file uses a coordinate system that matches the reference genome that you have used for aligning your reads. (The safest way to ensure this is to download the reference genome from Ensembl, too.) If you cannot use an Ensembl GTF file, see Appendix **??** for advice on converting GFF files from other sources to make them suitable as input for the `dexseq_prepare_annotation.py` script.

In a GTF file, many exons appear multiple times, once for each transcript that contains them. We need to "collapse" this information to define *exon counting bins*, i. e., a list of intervals, each corresponding to one exon or part of an exon. Counting bins for parts of exons arise when an exonic region appears with different boundaries in different transcripts. See Figure 1 of the DEXSeq paper [**?**] for an illustration.

The Python script `dexseq_prepare_annotation.py` takes an Ensembl GTF file and translates it into a GFF file with collapsed exon counting bins.

Make sure that your current working directory contains the GTF file and call the script (from the command line shell, not from within R) with

```
python /path/to/library/DEXSeq/python_scripts/dexseq_prepare_annotation.py
    Drosophila_melanogaster.BDGP5.72.gtf Dmel.BDGP5.25.62.DEXSeq.chr.gff
```

In this command, which should be entered as a single line, replace `/path/to.../python_scripts` with the correct path to the Python scripts, which you have found with the call to `system.file` shown above. `Drosophila_melanogaster.BDGP5.72.gtf` is the Ensembl GTF file (here the one for fruit fly, already de-compressed) and `Dmel.BDGP5.25.62.DEXSeq.chr.gff` is the name of the output file.

In the process of forming the counting bins, the script might come across overlapping genes. If two genes on the same strand are found with an exon of the first gene overlapping with an exon of the second gene, the script's default behaviour is to combine the genes into a single "aggregate gene" which is subsequently referred to with the IDs of the individual genes, joined by a plus ('+') sign. If you do not like this behaviour, you can disable aggregation with the option "`-r no`". Without aggregation, exons that overlap with other exons from different genes are simply skipped.

2.5 Counting reads

For each SAM file, we next count the number of reads that overlap with each of the exon counting bins defined in the flattened GFF file. This is done with the script `python_count.py`:

```
python /path/to/library/DEXSeq/python_scripts/dexseq_count.py
    Dmel.BDGP5.25.62.DEXSeq.chr.gff untreated1.sam untreated1fb.txt
```

This command (again, to be entered as a single line) expects two files in the current working directory, namely the GFF file produced in the previous step (here `Dmel_flattened.py`) and a SAM file with the aligned reads from a sample (here the file `untreated1.sam` with the aligned reads from the first control sample). The command generates an output file, here called `untreated1fb.txt`, with one line for each exon counting bin defined in the flattened GFF file. The lines contain the exon counting bin IDs (which are composed of gene IDs and exon bin numbers), followed by a integer number which indicates the number of reads that were aligned such that they overlap with the counting bin.

Use the script multiple times to produce a count file from each of your SAM files.

There are a number of crucial points to pay attention to when using the `python_count.py` script:

Paired-end data: If your data is from a paired-end sequencing run, you need to add the option "`-p yes`" to the command to call the script. (As usual, options have to be placed before the file names, surrounded by spaces.) In addition, the SAM file needs to be sorted, either by read name or by position. Most aligners produce sorted SAM files; if your SAM file is not sorted, use `samtools sort -n` to sort by read name (or `samtools sort`) to sort by position. (See e.g. reference [**?**], if you need further explanations on how to sort SAM files.) Use the option "`-r pos`" or "`-r name`" to indicate whether your paired-end

data is sorted by alignment position or by read name.[2]

Strandedness: By default, the counting script assumes your library to be *strand-specific*, i.e., reads are aligned to the same strand as the gene they originate from. If you have used a library preparation protocol that does not preserve strand information (i.e., reads from a given gene can appear equally likely on either strand), you need to inform the script by specifying the option "`-s no`". If your library preparation protocol reverses the strand (i.e., reads appear on the strand opposite to their gene of origin), use "`-s reverse`". In case of paired-end data, the default (`-s yes`) means that the read from the first sequence pass is on the same strand as the gene and the read from the second pass on the opposite strand ("forward-reverse" or "fr" order in the parlance of the Bowtie/TopHat manual) and the options `-s reverse` specifies the opposite case.

SAM and BAM files: By default, the script expects its input to be in plain-text SAM format. However, it can also read BAM files, i.e., files in the the compressed binary variant of the SAM format. If you wish to do so, use the option "`-f bam`". This works only if you have installed the Python package *pysam*, which can be found at https://code.google.com/p/pysam/.

Alignment quality: The scripts takes a further option, `-a` to specify the minimum alignment quality (as given in the fifth column of the SAM file). All reads with a lower quality than specified (with default `-a 10`) are skipped.

Help pages: Calling either script without arguments displays a help page with an overview of all options and arguments.

2.6 Reading the data in to R

The remainder of the analysis is now done in *R*. We will use the output of the python scripts for the *pasilla* experiment, that can be found in the package *pasilla*. Open an *R*session and type:

```
inDir = system.file("extdata", package="pasilla")
countFiles = list.files(inDir, pattern="fb.txt$", full.names=TRUE)
basename(countFiles)
```

```
## [1] "treated1fb.txt"   "treated2fb.txt"   "treated3fb.txt"   "untreated1fb.txt"
## [5] "untreated2fb.txt" "untreated3fb.txt" "untreated4fb.txt"
```

```
flattenedFile = list.files(inDir, pattern="gff$", full.names=TRUE)
basename(flattenedFile)
```

```
## [1] "Dmel.BDGP5.25.62.DEXSeq.chr.gff"
```

Now, we need to prepare a sample table. This table should contain one row for each library, and columns for all relevant information such as name of the file with the read counts, experimental conditions, technical information and further covariates. To keep this vignette simple, we construct the table on the fly.

[2]The possibility to process paired-end data from a file sorted by position is based on recent contributions of Paul-Theodor Pyl to *HTSeq*.

```
sampleTable = data.frame(
   row.names = c( "treated1", "treated2", "treated3",
      "untreated1", "untreated2", "untreated3", "untreated4" ),
   condition = c("knockdown", "knockdown", "knockdown",
      "control", "control", "control", "control" ),
   libType = c( "single-end", "paired-end", "paired-end",
      "single-end", "single-end", "paired-end", "paired-end" ) )
```

While this is a simple way to prepare the table, it may be less error-prone and more prudent to used an existing table that had already been prepared when the experiments were done, save it in CSV format and use the R function `read.csv` to load it.

In any case, it is vital to check the table carefully for correctness.

```
sampleTable
```

```
##            condition    libType
## treated1   knockdown single-end
## treated2   knockdown paired-end
## treated3   knockdown paired-end
## untreated1   control single-end
## untreated2   control single-end
## untreated3   control paired-end
## untreated4   control paired-end
```

Our table contains the sample names as row names and the two covariates that vary between samples: first the experimental condition (factor `condition` with levels `control` and `treatment`) and the library type (factor `libType`), which we included because the samples in this particular experiment were sequenced partly in single-end runs and partly in paired-end runs.

For now, we will ignore this latter piece of information, and postpone the discussion of how to include such additional covariates to Section **??**. If you have only a single covariate and want to perform a simple analysis, the column with this covariate should be named `condition`.

Now, we construct an *DEXSeqDataSet* object from this data. This object holds all the input data and will be passed along the stages of the subsequent analysis. We construct the object with the *DEXSeq* function `DEXSeqDataSetFromHTSeq`, as follows:

```
suppressPackageStartupMessages( library( "DEXSeq" ) )

dxd = DEXSeqDataSetFromHTSeq(
   countFiles,
   sampleData=sampleTable,
   design= ~ sample + exon + condition:exon,
   flattenedfile=flattenedFile )
```

The function takes four arguments. First, a vector with names of count files, i.e., of files that have been generated with the `dexseq_count.py` script. The function will read these files and arrange the

count data in a matrix, which is stored in the *DEXSeqDataSet* object `dxd`. The second argument is our sample table, with one row for each of the files listed in the first argument. This information is simply stored as is in the object's meta-data slot (see below). The third argument is a formula of the form " sample + exon + condition:exon" that specifies the contrast with of a variable from the sample table columns and the 'exon' variable. Using this formula, we are interested in differences in exon usage due to the 'condition' variable changes. Later in this vignette, we will how to add additional variables for complex designs. The fourth argument is a file name, now of the flattened GFF file that was generated with `dexseq_prepare_annotation.py` and used as input to `dexseq_count.py` when creating the count file.

There are other ways to get a *DEXSeq* analysis started. See Appendix **??** and Ref. [**?**] for details.

3 Standard analysis work-flow

3.1 Loading and inspecting the example data

To demonstrate the *DEXSeq* work flow, we will use the `DEXSeqDataSet` constructed in the previous section. However, in order to keep the run-time of this vignette small, we will subset the object to only a few genes.

```
genesForSubset = read.table(
  file.path(inDir, "geneIDsinsubset.txt"),
  stringsAsFactors=FALSE)[[1]]

dxd = dxd[geneIDs( dxd ) %in% genesForSubset,]
```

The *DEXSeqDataSet* class is derived from the *DESeqDataSet*. As such, it contains the usual accessor functions for the column data, row data, and some specific ones. The core data in an *DEXSeqDataSet* object are the counts per exon. Each row of the *DEXSeqDataSet* contains in each column the count data from a given exon ('this') as well as the count data from the sum of the other exons belonging to the same gene ('others'). This annotation, as well as all the information regarding each column of the *DEXSeqDataSet*, is specified in the colData.

```
colData(dxd)
```

```
## DataFrame with 14 rows and 4 columns
##         sample condition    libType     exon
##       <factor>  <factor>   <factor> <factor>
## 1     treated1 knockdown single-end     this
## 2     treated2 knockdown paired-end     this
## 3     treated3 knockdown paired-end     this
## 4   untreated1   control single-end     this
## 5   untreated2   control single-end     this
## ...         ...       ...        ...      ...
## 10    treated3 knockdown paired-end   others
```

```
## 11  untreated1   control single-end   others
## 12  untreated2   control single-end   others
## 13  untreated3   control paired-end   others
## 14  untreated4   control paired-end   others
```

We can access the first 5 rows from the count data by doing,

```
head( counts(dxd), 5 )
```

```
##                   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## FBgn0000256:E001   92   28   43   54  131   51   49 1390  829   923  1115  2495
## FBgn0000256:E002  124   80   91   76  224   82   95 1358  777   875  1093  2402
## FBgn0000256:E003  340  241  262  347  670  260  297 1142  616   704   822  1956
## FBgn0000256:E004  250  189  201  219  507  242  250 1232  668   765   950  2119
## FBgn0000256:E005   96   38   39   71   76   57   62 1386  819   927  1098  2550
##                   [,13] [,14]
## FBgn0000256:E001  1054  1052
## FBgn0000256:E002  1023  1006
## FBgn0000256:E003   845   804
## FBgn0000256:E004   863   851
## FBgn0000256:E005  1048  1039
```

Notice that the number of columns is 14, the first seven (we have seven samples) corresponding to the number of reads mapping to out exonic regions and the last seven correspond to the sum of the counts mapping to the rest of the exons from the same gene on each sample.

```
split( seq_len(ncol(dxd)), colData(dxd)$exon )
```

```
## $this
## [1] 1 2 3 4 5 6 7
##
## $others
## [1]  8  9 10 11 12 13 14
```

We can also access only the first five rows from the count belonging to the exonic regions ('this') (without showing the sum of counts from the rest of the exons from the same gene) by doing,

```
head( featureCounts(dxd), 5 )
```

```
##                   treated1 treated2 treated3 untreated1 untreated2 untreated3
## FBgn0000256:E001       92       28       43         54        131         51
## FBgn0000256:E002      124       80       91         76        224         82
## FBgn0000256:E003      340      241      262        347        670        260
## FBgn0000256:E004      250      189      201        219        507        242
## FBgn0000256:E005       96       38       39         71         76         57
##                   untreated4
## FBgn0000256:E001         49
## FBgn0000256:E002         95
```

```
## FBgn0000256:E003        297
## FBgn0000256:E004        250
## FBgn0000256:E005         62
```

In both cases, the rows are labelled with gene IDs (here Flybase IDs), followed by a colon and the counting bin number. (As a counting bin corresponds to an exon or part of an exon, this ID is called the *feature ID* or *exon ID* within *DEXSeq*.) The table content indicates the number of reads that have been mapped to each counting bin in the respective sample.

To see details on the counting bins, we also print the first 3 lines of the feature data annotation:

```
head( rowRanges(dxd), 3 )
```

```
## GRanges object with 3 ranges and 5 metadata columns:
##                     seqnames             ranges strand |   featureID     groupID
##                        <Rle>          <IRanges>  <Rle> | <character> <character>
##   FBgn0000256:E001    chr2L [3872658, 3872947]      - |        E001 FBgn0000256
##   FBgn0000256:E002    chr2L [3873019, 3873322]      - |        E002 FBgn0000256
##   FBgn0000256:E003    chr2L [3873385, 3874395]      - |        E003 FBgn0000256
##                     exonBaseMean exonBaseVar transcripts
##                        <numeric>   <numeric>      <list>
##   FBgn0000256:E001           64        1251    ########
##   FBgn0000256:E002          110        2770    ########
##   FBgn0000256:E003          345       22148    ########
##   -------
##   seqinfo: 14 sequences from an unspecified genome; no seqlengths
```

So far, this table contains information on the annotation data, such as gene and exon IDs, genomic coordinates of the exon, and the list of transcripts that contain an exon.

The accessor function `annotationData` shows the design table with the sample annotation (which was passed as the second argument to `DEXSeqDataSetFromHTSeq`):

```
sampleAnnotation( dxd )
```

```
## DataFrame with 7 rows and 3 columns
##       sample condition    libType
##     <factor>  <factor>   <factor>
## 1   treated1 knockdown single-end
## 2   treated2 knockdown paired-end
## 3   treated3 knockdown paired-end
## 4 untreated1   control single-end
## 5 untreated2   control single-end
## 6 untreated3   control paired-end
## 7 untreated4   control paired-end
```

In the following sections, we will update the object by calling a number of analysis functions, always using the idiom " `dxd = someFunction( dxd )`", which takes the `dxd` object, fills in the results of the performed computation and writes the returned and updated object back into the variable `dxd`.

3.2 Normalisation

Different samples might be sequenced with different depths. In order to adjust for such coverage biases, we estimate *size factors*, which measure relative sequencing depth. *DEXSeq* uses the same method as *DESeq* and *DESeq2*, which is provided in the function `estimateSizeFactors`.

```
dxd = estimateSizeFactors( dxd )
```

3.3 Dispersion estimation

To test for differential exon usage, we need to estimate the variability of the data. This is necessary to be able to distinguish technical and biological variation (noise) from real effects on exon usage due to the different conditions. The information on the strength of the noise is inferred from the biological replicates in the data set and characterized by the so-called *dispersion*. In RNA-Seq experiments the number of replicates is typically too small to reliably estimate variance or dispersion parameters individually exon by exon, and therefore, variance information is shared across exons and genes, in an intensity-dependent manner.

In this section, we discuss simple one-way designs: In this setting, samples with the same experimental condition, as indicated in the `condition` factor of the design table (see above), are considered as replicates – and therefore, the design table needs to contain a column with the name `condition`. In Section **??**, we discuss how to treat more complicated experimental designs which are not accommodated by a single `condition` factor.

To estimate the dispersion estimates, *DEXSeq* uses the approach of the package *DESeq2*. Internally, the functions from DESeq2 are called, adapting the parameters of the functions for the specific case of the DEXSeq model. Briefly, per-exon dispersions are calculated using a Cox-Reid adjusted profile likelihood estimation, then a dispersion-mean relation is fitted to this individual dispersion values and finally, the fitted values are taken as a prior in order to shrink the per-exon estimates towards the fitted values. See the *DESeq2* paper for the rational behind the shrinkage approach [**?**].

```
dxd = estimateDispersions( dxd )
```

As a shrinkage diagnostic, the *DEXSeqDataSet* inherits the method `plotDispEsts` that allows us to plot the per-exon dispersion estimates versus the mean normalised count, the resulting fitted values and the *a posteriori* (shrinked) dispersion estimates (Figure **??**).

```
plotDispEsts( dxd )
```

3.4 Testing for differential exon usage

Having the dispersion estimates and the size factors, we can now test for differential exon usage. For each gene, *DEXSeq* fits a generalized linear model with the formula

$$\sim \texttt{sample + exon + condition:exon} \tag{2}$$

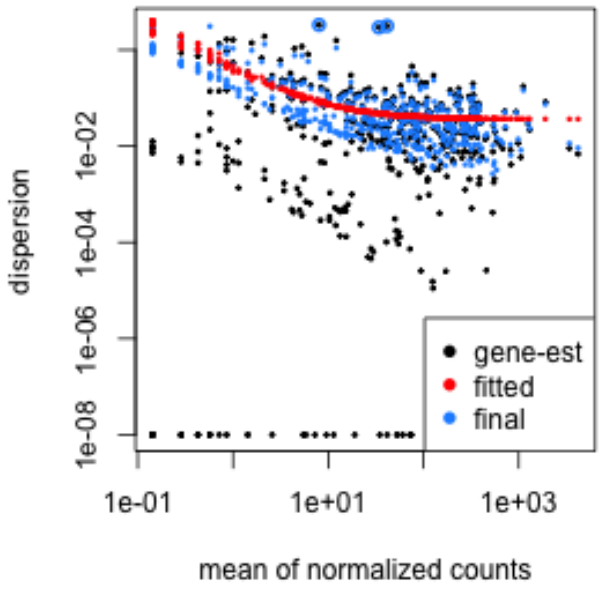

Figure 1: **Fit Diagnostics.** The initial per-exon dispersion estimates (shown by black points), the fitted mean-dispersion values function (red line), and the shrinked values in blue.

and compare it to the smaller model (the null model)

$$\sim \texttt{sample + exon}. \tag{3}$$

In these formulae (which use the standard notation for linear model formulae in *R*; consult a text book on *R* if you are unfamiliar with it), `sample` is a factor with different levels for each sample, `condition` is the factor of experimental conditions that we defined when constructing the *DEXSeqDataSet* object at the beginning of the analysis, and `exon` is a factor with two levels, `this` and `others`, that were specified when we generated our *DEXSeqDataSet* object. The two models described by these formulae are fit for each counting bin, where the data supplied to the fit comprise *two* read count values for each sample, corresponding to the two levels of the `exon` factor: the number of reads mapping to the bin in question (level `this`), and the sum of the read counts from all other bins of the same gene (level `others`). Note that this approach differs from the approach described in the paper [**?**] and used in older versions of *DEXSeq*; see Appendix **??** for further discussion.

Readers familiar with linear model formulae might find one aspect of Equation (**??**) surprising: We have an interaction term `condition:exon`, but denote no main effect for `condition`. Note, however, that all observations from the same sample are also from the same condition, i.e., the `condition` main effects are absorbed in the `sample` main effects, because the `sample` factor is nested within the `condition` factor.

The deviances of both fits are compared using a χ^2-distribution, providing a p value. Based on this p-value, we can decide whether the null model (**??**) is sufficient to explain the data, or whether it

may be rejected in favour of the alternative, model (**??**), which contains an interaction coefficient for `condition:exon`. The latter means that the fraction of the gene's reads that fall onto the exon under the test differs significantly between the experimental conditions.

The function `testForDEU` performs these tests for each exon in each gene.

```
dxd = testForDEU( dxd )
```

The resulting *DEXSeqDataSet* object contains slots with information regarding the test.

For some uses, we may also want to estimate relative exon usage fold changes. To this end, we call `estimateExonFoldChanges`. Exon usage fold changes are calculated based on the coefficients of a GLM fit that uses the formula

$$\texttt{count} \sim \texttt{condition} + \texttt{exon} + \texttt{condition:exon}. \tag{4}$$

Where "condition" can be replaced with any of the column names of `colData` (see man pages for details). The resulting coefficients allow the estimation of changes on the usage of exons across different conditions. Note that the differences on exon usage are distinguished from gene expression differences across conditions. For example, consider the case of a gene that is differentially expressed between conditions and has one exon that is differentially used between conditions. From the coefficients of the fitted model, it is possible to distinguish overall gene expression effects, that alter the counts from all the exons, from exon usage effects, that are captured by the interaction term `condition:exon` and that affect the each exon individually.

```
dxd = estimateExonFoldChanges( dxd, fitExpToVar="condition")
```

So far in the pipeline, the intermediate and final results have been stored in the meta data of a *DEXSeqDataSet* object, they can be accessed using the function `mcol`. In order to summarize the results without showing the values from intermediate steps, we call the function `DEXSeqResults`. The result is a *DEXSeqResults* object, which is a subclass of a *DataFrame* object.

```
dxr1 = DEXSeqResults( dxd )
dxr1
```

```
##
## LRT p-value: full vs reduced
##
## DataFrame with 498 rows and 13 columns
##                        groupID   featureID exonBaseMean dispersion      stat
##                    <character> <character>    <numeric>  <numeric> <numeric>
## FBgn0000256:E001 FBgn0000256        E001           58     0.0172   8.2e-06
## FBgn0000256:E002 FBgn0000256        E002          103     0.0073   1.6e+00
## FBgn0000256:E003 FBgn0000256        E003          326     0.0105   3.6e-02
## FBgn0000256:E004 FBgn0000256        E004          254     0.0110   1.7e-01
## FBgn0000256:E005 FBgn0000256        E005           61     0.0440   2.9e-02
## ...                      ...         ...          ...        ...       ...
```

```
## FBgn0261573:E012 FBgn0261573        E012         23.1      0.022      8.40
## FBgn0261573:E013 FBgn0261573        E013          9.8      0.248      1.16
## FBgn0261573:E014 FBgn0261573        E014         87.5      0.033      1.12
## FBgn0261573:E015 FBgn0261573        E015        268.3      0.012      2.60
## FBgn0261573:E016 FBgn0261573        E016        304.2      0.100      0.15
##                     pvalue      padj   control knockdown
##                  <numeric> <numeric> <numeric> <numeric>
## FBgn0000256:E001      1.00         1        11        11
## FBgn0000256:E002      0.21         1        13        14
## FBgn0000256:E003      0.85         1        19        19
## FBgn0000256:E004      0.68         1        18        17
## FBgn0000256:E005      0.86         1        11        11
## ...                    ...       ...       ...       ...
## FBgn0261573:E012    0.0038     0.098       6.6       8.5
## FBgn0261573:E013    0.2824     1.000       6.0       3.8
## FBgn0261573:E014    0.2902     1.000      13.2      11.9
## FBgn0261573:E015    0.1070     0.966      18.4      17.3
## FBgn0261573:E016    0.7018     1.000      18.9      18.1
##                  log2fold_knockdown_control              genomicData
##                                   <numeric>               <GRanges>
## FBgn0000256:E001                   -0.019  chr2L:3872658-3872947:-
## FBgn0000256:E002                    0.035  chr2L:3873019-3873322:-
## FBgn0000256:E003                   -0.024  chr2L:3873385-3874395:-
## FBgn0000256:E004                   -0.036  chr2L:3874450-3875302:-
## FBgn0000256:E005                    0.014  chr2L:3878895-3879067:-
## ...                                   ...                     ...
## FBgn0261573:E012                    0.357 chrX:19421654-19421867:+
## FBgn0261573:E013                   -0.652 chrX:19422668-19422673:+
## FBgn0261573:E014                   -0.143 chrX:19422674-19422856:+
## FBgn0261573:E015                   -0.094 chrX:19422927-19423634:+
## FBgn0261573:E016                   -0.060 chrX:19423707-19424937:+
##                       countData transcripts
##                        <matrix>      <list>
## FBgn0000256:E001    92 28 43 ...    ########
## FBgn0000256:E002   124 80 91 ...    ########
## FBgn0000256:E003 340 241 262 ...    ########
## FBgn0000256:E004 250 189 201 ...    ########
## FBgn0000256:E005    96 38 39 ...    ########
## ...                        ...         ...
## FBgn0261573:E012    37 23 38 ...    ########
## FBgn0261573:E013       8 3 6 ...    ########
## FBgn0261573:E014    75 66 92 ...    ########
## FBgn0261573:E015 264 234 245 ...    ########
## FBgn0261573:E016 611 187 188 ...    ########
```

The description of each of the column of the object *DEXSeqResults* can be found in the metadata columns.

```
mcols(dxr1)$description
```

```
##  [1] "group/gene identifier"
##  [2] "feature/exon identifier"
##  [3] "mean of the counts across samples in each feature/exon"
##  [4] "exon dispersion estimate"
##  [5] "LRT statistic: full vs reduced"
##  [6] "LRT p-value: full vs reduced"
##  [7] "BH adjusted p-values"
##  [8] "exon usage coefficient"
##  [9] "exon usage coefficient"
## [10] "relative exon usage fold change"
## [11] "GRanges object of the coordinates of the exon/feature"
## [12] "matrix of integer counts, of each column containing a sample"
## [13] "list of transcripts overlapping with the exon"
```

From this object, we can ask how many genes are significant with a false discovery rate of 10%:

```
table ( dxr1$padj < 0.1 )
```

```
##
## FALSE  TRUE
##   426    17
```

We may also ask how many genes are affected

```
table ( tapply( dxr1$padj < 0.1, dxr1$groupID, any ) )
```

```
##
## FALSE  TRUE
##    20     9
```

Remember that our example data set contains only a selection of genes. We have chosen these to contain interesting cases; so the fact that such a large fraction of genes is significantly affected here is not typical.

To see how the power to detect differential exon usage depends on the number of reads that map to an exon, a so-called MA plot is useful, which plots the logarithm of fold change versus average normalized count per exon and marks by red colour the exons which are considered significant; here, the exons with an adjusted p values of less than 0.1 (Figure **??**). There is of course nothing special about the number 0.1, and you can specify other thresholds in the call to `plotMA`.

```
plotMA( dxr1, cex=0.8 )
```

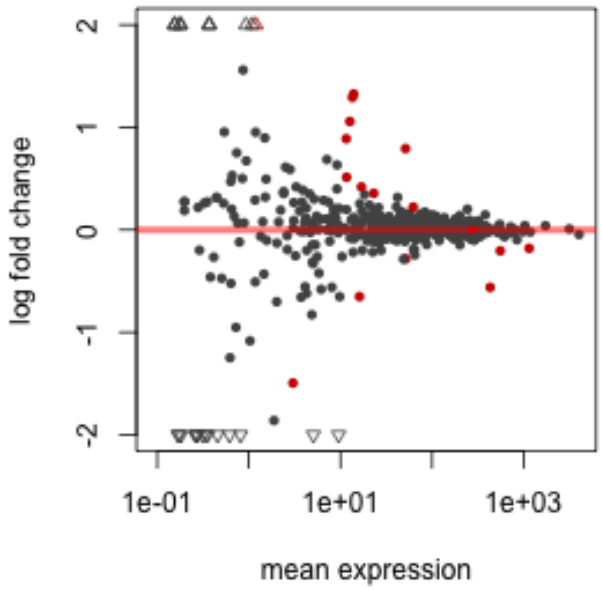

Figure 2: **MA plot.** Mean expression versus $\log_2$ fold change plot. Significant hits (at padj¡0.1) are coloured in red.

4 Additional technical or experimental variables

In the previous section we performed a simple analysis of differential exon usage, in which each sample was assigned to one of two experimental conditions. If your experiment is of this type, you can use the work flow shown above. All you have to make sure is that you indicate which sample belongs to which experimental condition when you construct the *DEXSeqDataSet* object (Section **??**. Do so by means of a column called `condition` in the sample table.

If you have a more complex experimental design, you can provide different or additional columns in the sample table. You then have to indicate the design by providing explicit formulae for the test.

In the *pasilla* dataset, some samples were sequenced in single-end and others in paired-end mode. Possibly, this influenced counts and should hence be accounted for. We therefore use this as an example for a complex design.

When we constructed the *DEXSeqDataSet* object in Section **??**, we provided in the sample table an additional column called `libType`, which has been stored in the object:

```
sampleAnnotation(dxd)
```

```
## DataFrame with 7 rows and 4 columns
##       sample condition    libType sizeFactor
##     <factor>  <factor>   <factor>  <numeric>
```

```
## 1   treated1 knockdown single-end       1.34
## 2   treated2 knockdown paired-end       0.80
## 3   treated3 knockdown paired-end       0.92
## 4 untreated1   control single-end       0.99
## 5 untreated2   control single-end       1.57
## 6 untreated3   control paired-end       0.84
## 7 untreated4   control paired-end       0.83
```

We specify two design formulae, which indicate that the `libType` *factor* should be treated as a blocking factor:

```
formulaFullModel    =  ~ sample + exon + libType:exon + condition:exon
formulaReducedModel =  ~ sample + exon + libType:exon
```

Compare these formulae with the default formulae (**??**, **??**) given in Section **??**. We have added, in both the full model and the reduced model, the term `libType:exon`. Therefore, any dependence of exon usage on library type will be absorbed by this term and accounted for equally in the full and a reduced model, and the likelihood ratio test comparing them will only detect differences in exon usage that can be attributed to `condition`, independent of `type`.

Next, we estimate the dispersions. This time, we need to inform the `estimateDispersions` function about our design by providing the full model's formula, which should be used instead of the default formula (**??**).

```
dxd = estimateDispersions( dxd, formula = formulaFullModel )
```

The test function now needs to be informed about both formulae

```
dxd = testForDEU( dxd,
        reducedModel = formulaReducedModel,
        fullModel = formulaFullModel )
```

Finally, we get a summary table, as before.

```
dxr2 = DEXSeqResults( dxd )
```

How many significant DEU cases have we got this time?

```
table( dxr2$padj < 0.1 )
```

```
##
## FALSE  TRUE
##   393    22
```

We can now compare with the previous result:

```
table( before = dxr1$padj < 0.1, now = dxr2$padj < 0.1 )
```

```
##        now
## before  FALSE TRUE
```

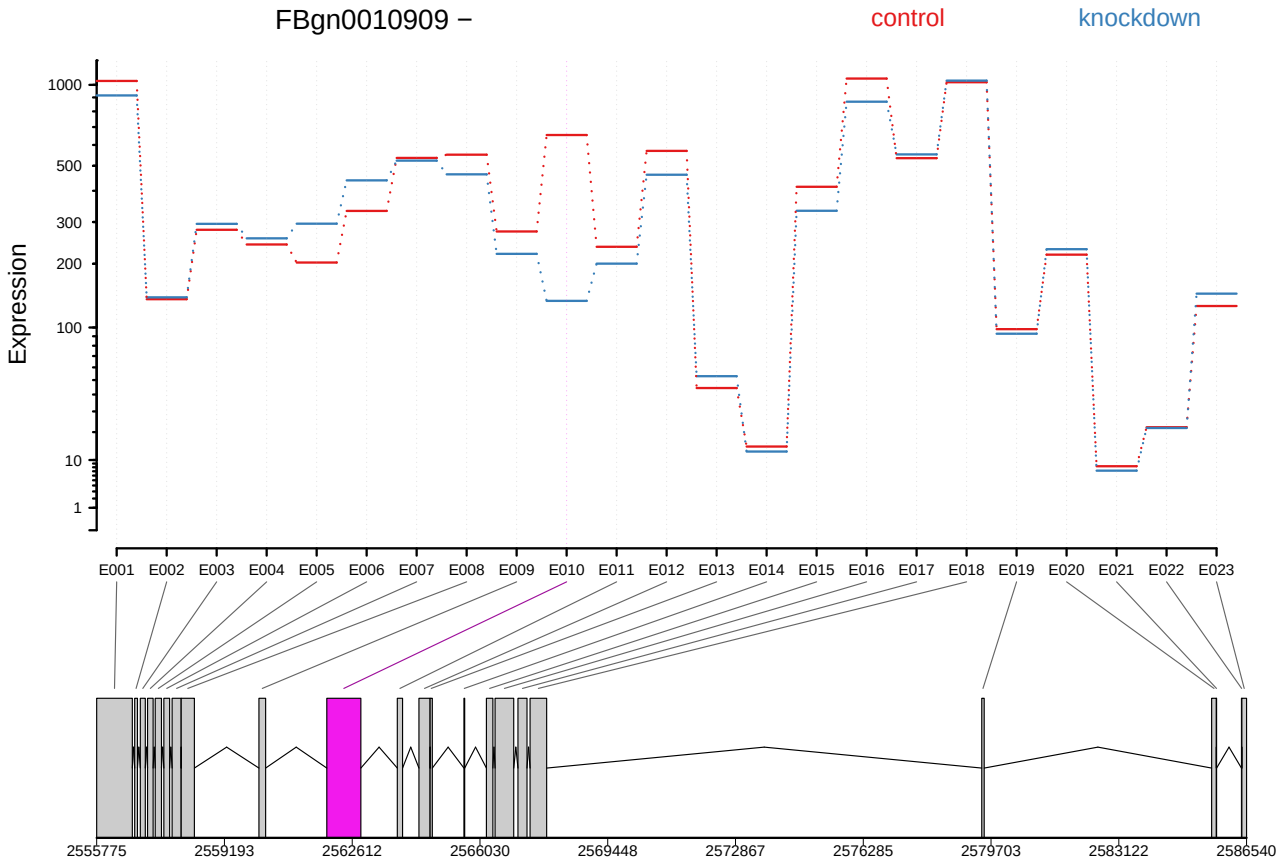

Figure 3: **Fitted expression.** The plot represents the expression estimates from a call to `testForDEU`. Shown in red is the exon that showed significant differential exon usage.

```
##   FALSE   392    6
##   TRUE      1   16
```

Accounting for the library type has allowed us to find six more hits, which confirms that accounting for the covariate improves power.

5 Visualization

The `plotDEXSeq` provides a means to visualize the results of an analysis.

```
plotDEXSeq( dxr2, "FBgn0010909", legend=TRUE, cex.axis=1.2, cex=1.3, lwd=2 )
```

The result is shown in Figure **??**. This plot shows the fitted expression values of each of the exons of gene FBgn0010909, for each of the two conditions, treated and untreated. The function `plotDEXSeq` expects at least two arguments, the *DEXSeqDataSet* object and the gene ID. The option `legend=TRUE` causes a legend to be included. The three remaining arguments in the code chunk above are ordinary

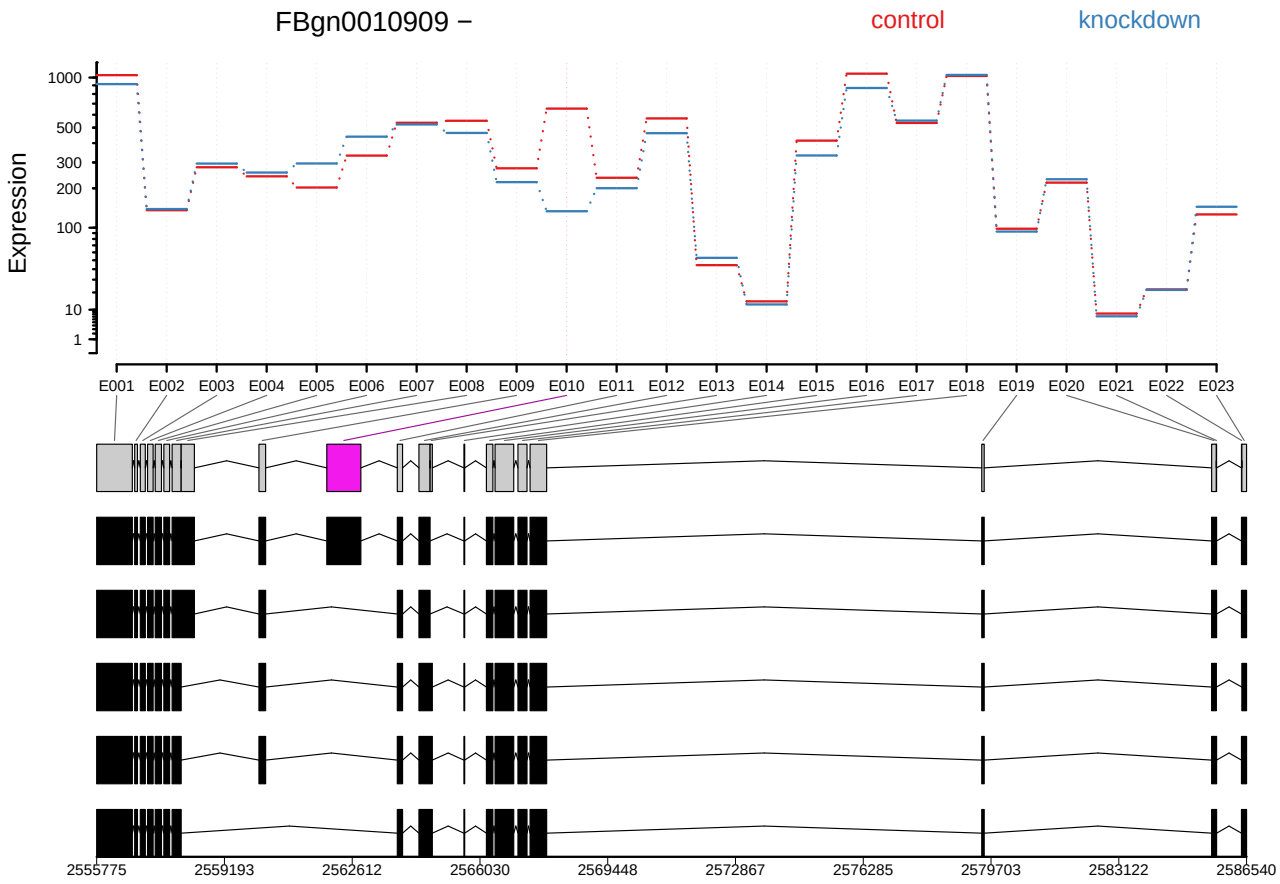

Figure 4: **Transcripts.** As in Figure **??**, but including the annotated transcript models.

plotting parameters which are simply handed over to *R*'s standard plotting functions. They are not strictly needed and included here to improve appearance of the plot. See the help page for `par` for details.

Optionally, one can also visualize the transcript models (Figure **??**), which can be useful for putting differential exon usage results into the context of isoform regulation.

```
plotDEXSeq( dxr2, "FBgn0010909", displayTranscripts=TRUE, legend=TRUE,
   cex.axis=1.2, cex=1.3, lwd=2 )
```

Other useful options are to look at the count values from the individual samples, rather than at the model effect estimates. For this display (option `norCounts=TRUE`), the counts are normalized by dividing them by the size factors (Figure **??**).

```
plotDEXSeq( dxr2, "FBgn0010909", expression=FALSE, norCounts=TRUE,
   legend=TRUE, cex.axis=1.2, cex=1.3, lwd=2 )
```

As explained in Section **??**, *DEXSeq* is designed to find changes in relative exon usage, i. e., changes in the expression of individual exons that are not simply the consequence of overall up- or down-regulation of the gene. To visualize such changes, it is sometimes advantageous to remove overall changes in

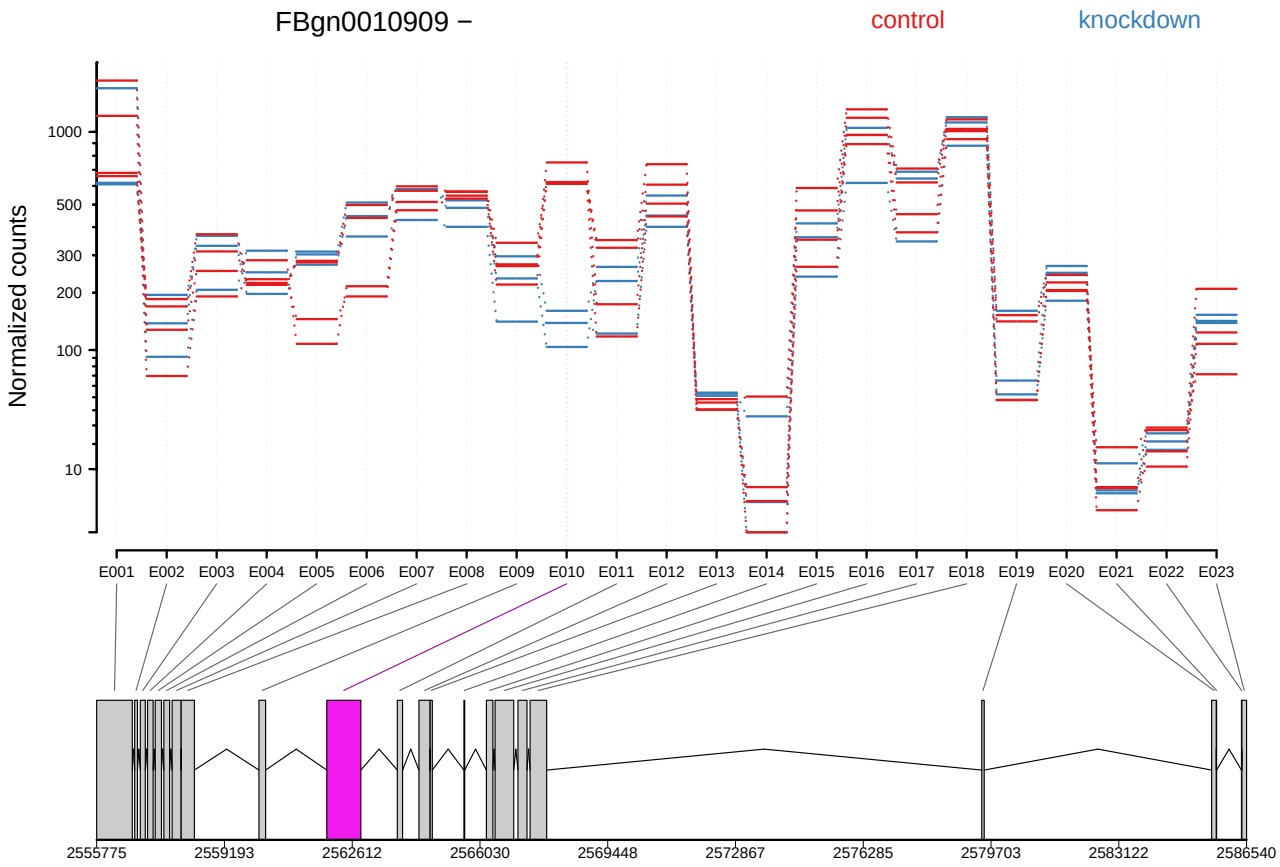

Figure 5: **Normalized counts.** As in Figure **??**, with normalized count values of each exon in each of the samples.

expression from the plots. Use the (somewhat misnamed) option `splicing=TRUE` for this purpose.

```
plotDEXSeq( dxr2, "FBgn0010909", expression=FALSE, splicing=TRUE,
   legend=TRUE, cex.axis=1.2, cex=1.3, lwd=2 )
```

To generate an easily browsable, detailed overview over all analysis results, the package provides an HTML report generator, implemented in the function *DEXSeqHTML*. This function uses the package *hwriter* [**?**] to create a result table with links to plots for the significant results, allowing a more detailed exploration of the results.

```
DEXSeqHTML( dxr2, FDR=0.1, color=c("#FF000080", "#0000FF80") )
```

6 Parallelization and large number of samples

DEXSeq analyses can be computationally heavy, especially with data sets that comprise a large number of samples, or with genomes containing genes with large numbers of exons. While some steps of the analysis work on the whole data set, the computational load can be parallelized for some

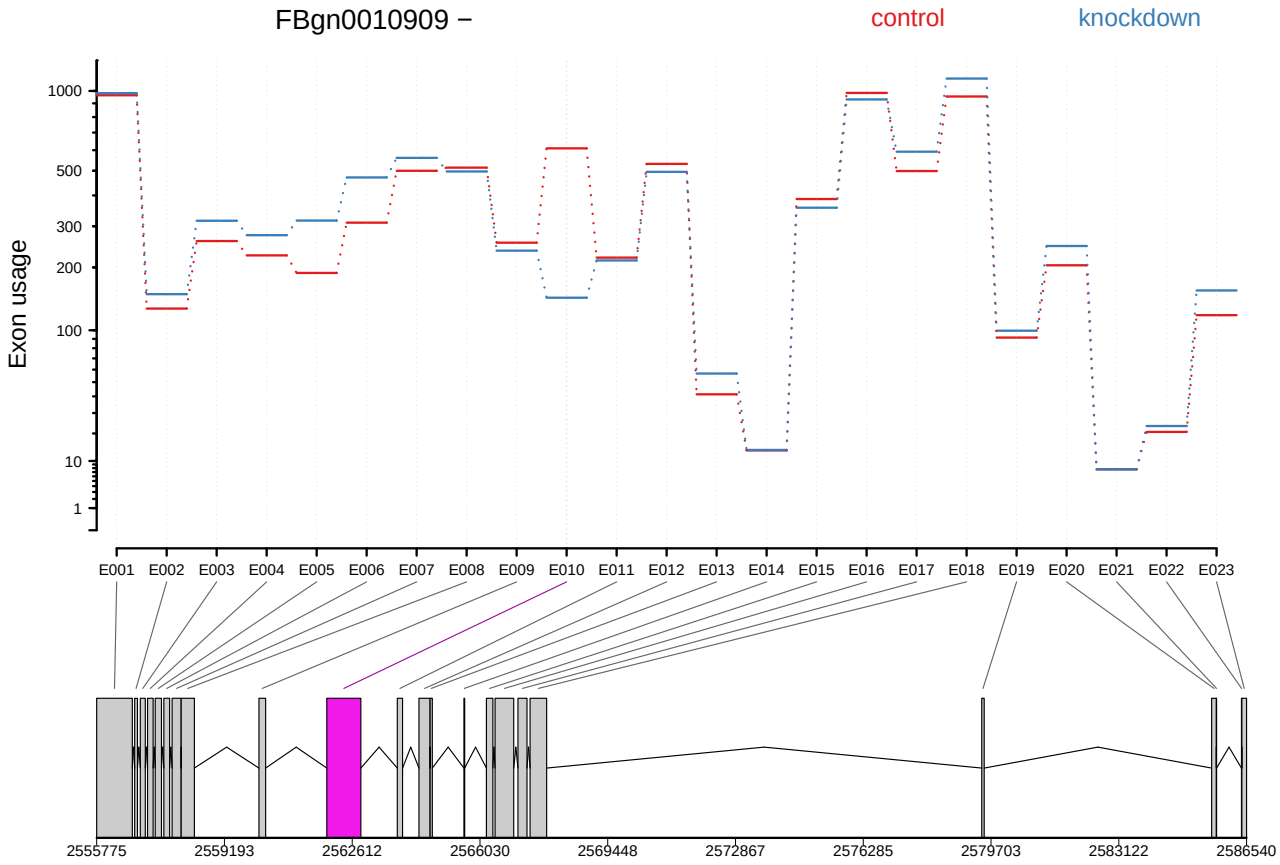

Figure 6: **Fitted splicing.** The plot represents the estimated effects, as in Figure **??**, but after subtraction of overall changes in gene expression.

steps. We use the package *BiocParallel*, and implemented the `BPPARAM` parameter of the functions `estimateDispersions`, `testForDEU` and `estimateExonFoldChanges`:

```
BPPARAM = MultiCoreParam(workers=4)
dxd = estimateSizeFactors( dxd )
dxd = estimateDispersions( dxd, BPPARAM=BPPARAM)
dxd = testForDEU( dxd, BPPARAM=BPPARAM)
dxd = estimateExonFoldChanges(dxd, BPPARAM=BPPARAM)
```

For running analysis with a large number of samples (e.g. more than 100), we recommend configuring *BatchJobsParam* classes to BPPARAM in order to distribute the calculations across a computer cluster and significantly reduce running times. Users might also consider reducing the number of tests by filtering for lowly expressed isoforms or exonic regions with low counts. Appart from reducing running times, this filtering step also leads to more accurate results [**?**].

7 Perform a standard differential exon usage analysis in one command

In the previous sections, we went through the analysis step by step. Once you are sufficiently confident about the work flow for your data, its invocation can be streamlined by the wrapper function `DEXseq`, which runs the analysis shown above through a single function call.

In the simplest case, construct the `DEXSeqDataSet` as shown in Section **??** or in Appendix **??**, then run `DEXSeq` passing the `DEXSeqDataSet` as only argument, this function will output a *DEXSeqResults* object.

```
dxr = DEXSeq(dxd)
class(dxr)
```

```
## [1] "DEXSeqResults"
## attr(,"package")
## [1] "DEXSeq"
```

APPENDIX

A Controlling FDR at the gene level

DEXSeq returns one p-value for each exonic region. Using these p-values, a rejection threshold is established according to a multiple testing correction method. Consider a setting with M exonic regions, where p_k is the p-value for the k exonic region and $\theta \in p_1, ..., p_M \subset [0, 1]$ is a rejection threshold, leading to $V = |\{k : p_k \leq \theta\}|$ rejections (i.e. number of exonic regions being DEU). The Benjamini-Hochberg method establishes that FDR is controlled at the level q^* where

$$q^* = \frac{M\theta}{|\{k : p_k \leq \theta\}|}. \tag{5}$$

Note that θ in the denominator of this expression is the probability of rejecting a single hypothesis if it is true. q^* allows to estimate the number of exons that are differentially used at a specific FDR.

However, sometimes it is informative to know what is the FDR at the gene level, i.e. knowing the numer of genes with at least one differentially used exon while keeping control of FDR. For this scenario, let $p_{i,l}$ be the p-value from the DEU test for exon l in gene i, let n_i the number of exons in gene i, and M the number of genes. A gene i has at least one DEU exon if $\exists l$ such that $p_{i,l} \leq \theta$. Then, FDR is controlled at the level

$$\frac{\sum_{i=1}^{M} 1 - (1 - \theta)^{n_i}}{|\{i : \exists l : p_{il} \leq \theta\}|}. \tag{6}$$

The implementation of the formula above is provided by the function `perGeneQValue`. For the *pasilla* example, the code below calculates the number of genes (at a FDR of 10%) with at least one differentially used exon.

```
numbOfGenes <- sum( perGeneQValue(dxr) < 0.1)
numbOfGenes
```

```
## [1] 9
```

B Preprocessing within R

As an alternative to the approach described in Section **??**, users can also create *DEXSeqDataSeq* objects from other *Bioconductor* data objects. The code for implementationg these functions was kindly contributed by Michael I. Love. For details, see the *parathyroidSE* package vignette [**?**]. The work flow is similar to the one using the *HTSeq* python scripts.

emphNote: The code in this section is not run when the vignette is built, as some of the commands have long run time. Therefore, no output is given.

We use functionality from the following Bioconductor packages

```
library(GenomicRanges)
library(GenomicFeatures)
library(GenomicAlignments)
```

We demonstrate the workflow briefly (for more details, see [**?**]) on the data set of Haglund et al. [**?**], which is provided as example data in the *parathyroidSE* data package.

First, we download the current human gene model annotation from Ensembl via Biomart and create a transcript data base from these. Note that this step takes some time.

```
hse = makeTxDbFromBiomart( biomart="ensembl", dataset="hsapiens_gene_ensembl" )
```

Next, we collapse the gene models into counting bins, analogous to Section **??**.

```
exonicParts = disjointExons( hse, aggregateGenes=FALSE )
```

As before, we have to choose how to handle genes with overlapping exons. The `aggregateGenes` option here plays the same role as the `-r` option to `dexseq_prepare_anotation.py` described at the end of Section **??**. The `exonicParts` object contains a GRanges object with our counting bins. We use it to count the number of read fragments that overlap with the bins by means of the function `summarizeOverlaps`. To demonstrate this, we first determine the paths to the example BAM files in the *parathyroidSE* data package.

```
bamDir = system.file( "extdata", package="parathyroidSE", mustWork=TRUE )
fls = list.files( bamDir, pattern="bam$", full=TRUE )
```

Then, use the following code to count the reads overlapping the bins.

```
bamlst = BamFileList( fls, index=character(), yieldSize=100000, obeyQname=TRUE )
SE = summarizeOverlaps( exonicParts, bamlst, mode="Union", singleEnd=FALSE,
   ignore.strand=TRUE, inter.feature=FALSE, fragments=TRUE )
```

We can now call the function `DEXSeqDataSetFromSE` to build an `DEXSeqDataSet` object. We modify the `colData` slot in order to specify the sample annotation, indicating that the first two BAM files form one experimental condition and the third one the other. Then we create our `DEXSeqDataSet` object.

```
colData(SE)$condition = c("A", "A", "B")
DEXSeqDataSetFromSE( SE, design= ~ sample + exon + condition:exon )
```

B.1 Further accessors

The function `geneIDs` returns the gene ID column of the feature data as a character vector, and the function `exonIDs` return the exon ID column as a *factor*.

```
head( geneIDs(dxd) )
```

```
## [1] "FBgn0000256" "FBgn0000256" "FBgn0000256" "FBgn0000256" "FBgn0000256"
## [6] "FBgn0000256"
```

```
head( exonIDs(dxd) )
```

```
## [1] "E001" "E002" "E003" "E004" "E005" "E006"
```

These functions are useful for subsetting an *DEXSeqDataSet* object.

B.2 Overlap operations

The methods `subsetByOverlaps` and `findOverlaps` have been implemented for the DEXSeqResults object, the `query` argument must be a *DEXSeqResults* object.

```
interestingRegion = GRanges("chr2L", IRanges(start=3872658, end=3875302))
subsetByOverlaps( query=dxr, subject=interestingRegion )
```

```
##
## LRT p-value: full vs reduced
##
## DataFrame with 4 rows and 16 columns
##                       groupID   featureID exonBaseMean dispersion      stat
##                   <character> <character>    <numeric>  <numeric> <numeric>
## FBgn0000256:E001 FBgn0000256        E001           58     0.0172   8.2e-06
## FBgn0000256:E002 FBgn0000256        E002          103     0.0073   1.6e+00
## FBgn0000256:E003 FBgn0000256        E003          326     0.0105   3.6e-02
## FBgn0000256:E004 FBgn0000256        E004          254     0.0110   1.7e-01
##                      pvalue      padj   control knockdown
##                   <numeric> <numeric> <numeric> <numeric>
## FBgn0000256:E001      1.00         1        11        11
## FBgn0000256:E002      0.21         1        13        14
## FBgn0000256:E003      0.85         1        19        19
## FBgn0000256:E004      0.68         1        18        17
##                   log2fold_knockdown_control control.1 knockdown.1
##                                    <numeric> <numeric>   <numeric>
## FBgn0000256:E001                     -0.019        11          11
## FBgn0000256:E002                      0.035        13          14
## FBgn0000256:E003                     -0.024        19          19
## FBgn0000256:E004                     -0.036        18          17
##                   log2fold_knockdown_control.1             genomicData
##                                      <numeric>               <GRanges>
## FBgn0000256:E001                       -0.019 chr2L:3872658-3872947:-
## FBgn0000256:E002                        0.035 chr2L:3873019-3873322:-
## FBgn0000256:E003                       -0.024 chr2L:3873385-3874395:-
```

```
## FBgn0000256:E004                        -0.036 chr2L:3874450-3875302:-
##                         countData transcripts
##                          <matrix>      <list>
## FBgn0000256:E001    92 28 43 ...    ########
## FBgn0000256:E002   124 80 91 ...    ########
## FBgn0000256:E003 340 241 262 ...    ########
## FBgn0000256:E004 250 189 201 ...    ########
```

```
findOverlaps( query=dxr, subject=interestingRegion )
```

```
## Hits object with 4 hits and 0 metadata columns:
##       queryHits subjectHits
##       <integer>   <integer>
##   [1]         1           1
##   [2]         2           1
##   [3]         3           1
##   [4]         4           1
##   -------
##   queryLength: 498 / subjectLength: 1
```

This functions could be useful for further downstream analysis.

C Methodological changes since publication of the paper

In our paper [**?**], we suggested to fit for each exon a model that includes separately the counts for all the gene's exons. However, this turned out to be computationally inefficient for genes with many exons, because the many exons required large model matrices, which are computationally expensive to deal with. We have therefore modified the approach: when fitting a model for an exon, we now sum up the counts from all the other exon and use only the total, rather than the individual counts in the model. Now, computation time per exon is independent of the number of other exons in the gene, which improved *DEXSeq*'s scalability. While the p values returned by the two approaches are not exactly equal, the differences were very minor in the tests that we performed.

Deviating from the paper's notation, we now use the index i to indicate a specific counting bin, with i running through all counting bins of all genes. The samples are indexed with j, as in the paper. We write K_{ij0} for the count or reads mapped to counting bin i in sample j and K_{ij1} for the sum of the read counts from all other counting bins in the same gene. Hence, when we write K_{ijl}, the third index l indicates whether we mean the read count for bin i ($l = 0$) or the sum of counts for all other bins of the same gene ($l = 1$). As before, we fit a GLM of the negative binomial (NB) family

$$K_{ijl} \sim \mathrm{NB}(\text{mean} = s_j \mu_{ijl}, \text{dispersion} = \alpha_i), \tag{7}$$

now with the model specified in Equation (**??**), which we write out as

$$\log_2 \mu_{ijl} = \beta_{ij}^{\mathsf{S}} + l\beta_i^{\mathsf{E}} + \beta_{i\rho_j}^{\mathsf{EC}}. \tag{8}$$

This model is fit separately for each counting bin i. The coefficient β_{ij}^{S} accounts for the sample-specific contribution (factor `sample` in Equation (**??**)), the term β_i^{E} is only included if $l = 1$ and hence estimates the logarithm of the ratio K_{ij1}/K_{ij0} between the counts for all other exons and the counts for the tested exon. As this coefficient is estimated from data from all samples, it can be considered as a measure of "average exon usage". In the R model formula, it is represented by the term `exon` with its two levels `this` ($l = 0$) and `others` ($l = 1$). Finally, the last term, $\beta_{i,\rho_j}^{\mathsf{EC}}$, captures the interaction `condition:exon`, i.e., the change in exon usage if sample j is from experimental condition group $\rho(j)$. Here, the first condition, $\rho = 0$, is absorbed in the sample coefficients, i.e., β_{i0}^{EC} is fixed to zero and does not appear in the model matrix.

For the dispersion estimation, one dispersion value α_i is estimated with Cox-Reid-adjusted maximum likelihood using the full model given above. A mean-variance relation is fitted using the individual dispersion values. Finally, the individual values are shrinked towards the fitted values. For more details about this shrinkage approach look at the *DESeq2* vignette and/or its manuscript [**?**]. For the likelihood ratio test, this full model is fit and compared with the fit of the reduced model, which lacks the interaction term $\beta_{i\rho_j}^{\mathsf{EC}}$. As described in Section **??**, alternative model formulae can be specified.

D Requirements on GTF files

In the initial preprocessing step described in Section **??**, the Python script `dexseq_prepare_annotation.py` is used to convert a GTF file with gene models into a GFF file with collapsed gene models. We recommend to use GTF files downloaded from Ensembl as input for this script, as files from other sources may deviate from the format expected by the script. Hence, if you need to use a GTF or GFF file from another source, you may need to convert it to the expected format. To help with this task, we here give details on how the `dexseq_prepare_annotation.py` script interprets a GFF file.

- The script only looks at `exon` lines, i.e., at lines which contain the term `exon` in the third ("type") column. All other lines are ignored.
- Of the data in these lines, the information about chromosome, start, end, and strand (1st, 4th, 5th, and 7th column) are used, and, from the last column, the attributes `gene_id` and `transcript_id`. The rest is ignored.
- The `gene_id` attribute is used to see which exons belong to the same gene. It must be called `gene_id` (and not `Parent` as in GFF3 files, or `GeneID` as in some older GFF files), and it must give the same identifier to all exons from the same gene, even if they are from different transcripts of this gene. (This last requirement is not met by GTF files generated by the Table Browser function of the UCSC Genome Browser.)
- The `transcript_id` attribute is used to build the `transcripts` attribute in the flattened GFF file, which indicates which transcripts contain the described counting bin. This information is needed only to draw the transcript model at the bottom of the plots when the `displayTranscript` option to `plotDEXSeq` is used.

Therefore, converting a GFF file to make it suitable as input to `dexseq_prepare_annotation.py` amounts to making sure that the exon lines have type `exon` and that the atributes giving gene ID (or gene symbol) and transcript ID are called `gene_id` and `transcript_id`, with this exact spelling.

Remember to also take care that the chromosome names match those in your SAM files, and that the coordinates refer to the reference assembly that you used when aligning your reads.

E Session Information

The session information records the versions of all the packages used in the generation of the present document.

```
sessionInfo()
```

```
## R version 3.3.0 (2016-05-03)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.9.5 (Mavericks)
##
## locale:
## [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats4    parallel  stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] DEXSeq_1.18.4              RColorBrewer_1.1-2
##  [3] AnnotationDbi_1.34.2       DESeq2_1.12.2
##  [5] SummarizedExperiment_1.2.2 GenomicRanges_1.24.0
##  [7] GenomeInfoDb_1.8.2         IRanges_2.6.0
##  [9] S4Vectors_0.10.0           Biobase_2.32.0
## [11] BiocGenerics_0.18.0        BiocParallel_1.6.2
## [13] knitr_1.13
##
## loaded via a namespace (and not attached):
##  [1] genefilter_1.54.2   statmod_1.4.24      locfit_1.5-9.1
##  [4] splines_3.3.0       lattice_0.20-33     colorspace_1.2-6
##  [7] chron_2.3-47        survival_2.39-4     XML_3.98-1.4
## [10] foreign_0.8-66      DBI_0.4-1           plyr_1.8.3
## [13] stringr_1.0.0       zlibbioc_1.18.0     Biostrings_2.40.0
## [16] munsell_0.4.3       gtable_0.2.0        hwriter_1.3.2
## [19] codetools_0.2-14    evaluate_0.9        latticeExtra_0.6-28
## [22] geneplotter_1.50.0  biomaRt_2.28.0      highr_0.6
## [25] Rcpp_0.12.5         acepack_1.3-3.3     xtable_1.8-2
## [28] scales_0.4.0        formatR_1.4         Hmisc_3.17-4
## [31] annotate_1.50.0     XVector_0.12.0      Rsamtools_1.24.0
## [34] gridExtra_2.2.1     digest_0.6.9        BiocStyle_2.0.2
## [37] ggplot2_2.1.0       stringi_1.0-1       grid_3.3.0
```

```
## [40] tools_3.3.0          bitops_1.0-6         magrittr_1.5
## [43] RCurl_1.95-4.8       RSQLite_1.0.0        Formula_1.2-1
## [46] cluster_2.0.4        Matrix_1.2-6         data.table_1.9.6
## [49] rpart_4.1-10         nnet_7.3-12
```

F References