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 1.8.0 of the *DEXSeq* package.
Last revision of this document: 2013-10-08

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

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

The statistical method used by *DEXSeq* was introduced in our paper [1]. 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 (1). 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 (1) 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 [1] for a more complete description, as well as Appendix C 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 (1), 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. [2]. 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 [3]. From Section 3 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 that can deal with reads that span across introns) such as TopHat2 [4], GSNAP [5], or STAR [6]. 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 [3] and a more extensive explanation, using the same data set, in our protocol article on differential expression calling [7].

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

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 2.4) and `dexseq_count.py` (Section 2.5), that come with the *DEXSeq* package. If you have trouble finding them, start R and ask for the installation directory with

```
> system.file( "python_scripts", package="DEXSeq" )
```

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 [8].

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 D 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 [1] 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_flattened.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_flattenend.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_flattenend.gff untreated1.sam untreated1.counts
```

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 `untreated1.counts`, 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 [7], 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, tehs cript 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

[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*.

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 process is now done in *R*. Start *R* and load *DEXSeq*

```
> library( "DEXSeq" )
```

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.

```
> sampleTable <- data.frame(
+    row.names = c( "untreated1", "untreated2", "untreated3",
+       "untreated4", "treated1", "treated2", "treated3" ),
+    countFile = c( "untreated1.counts", "untreated2.counts",
+       "untreated3.counts", "untreated4.counts",
+       "treated1.counts", "treated2.counts","treated3.counts" ),
+    condition = c( "control", "control", "control",
+       "control", "knockdown", "knockdown", "knockdown" ),
+    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

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

Our table contains the sample names as row names, the names of the count files that we prepared in the previous step 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, however, ignore this latter piece of information, and postpone the discussion of how to include such additional covariates to Section 4. 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 *ExonCountSet* 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 `read.HTSeqCounts`, as follows:

```
> ecs <- read.HTSeqCounts(
+    sampleTable$countFile,
+    sampleTable,
+    "Dmel_flattenend.gff" )
```

The function takes three 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 *ExonCountSet* object `ecs`. 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 file name again, 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 A and Ref. [8] for details.

3 Standard analysis work-flow

3.1 Loading and inspecting the example data

To demonstrate the *DEXSeq* work flow, we have prepared, in the *pasilla* data package, an *ExonCountSet* similar to the one constructed in the previous section. However, in order to keep the run-time of this vignette small, we have subset the object to only a few genes.

To start, we load the package *DEXSeq* and the example data from the *pasilla* package.

```
> library( DEXSeq )
> data( "pasillaExons", package="pasilla" )
```

The object `pasillaExons` is an *ExonCountSet*, which has been constructed with `read.HTSeqCounts` as described above. For consistency, we rename it to `ecs` as above, and then inspect it.

```
> ecs <- pasillaExons
> ecs

ExonCountSet (storageMode: environment)
assayData: 498 features, 7 samples
```

```
  element names: counts
protocolData: none
phenoData
  sampleNames: treated1fb treated2fb ... untreated4fb (7 total)
  varLabels: sizeFactor condition type countfiles
  varMetadata: labelDescription
featureData
  featureNames: FBgn0000256:E001 FBgn0000256:E002 ... FBgn0261573:E016
    (498 total)
  fvarLabels: geneID exonID ... transcripts (13 total)
  fvarMetadata: labelDescription
experimentData: use experimentData(object)
  pubMedIds: 20921232
Annotation:
```

The *ExonCountSet* class is derived from *eSet*, Bioconductor's standard container class for experimental data [9]. As such, it contains the usual accessor functions for sample, feature and assay data (including `pData`, `fData`, `experimentData`), and some specific ones. The core data in an *ExonCountSet* object are the counts per exon. Let's have a look at the first 7 rows.

```
> head( counts(ecs), 7 )
```

```
                 treated1fb treated2fb treated3fb untreated1fb untreated2fb
FBgn0000256:E001         92         28         43           54          131
FBgn0000256:E002        124         80         91           76          224
FBgn0000256:E003        340        241        262          347          670
FBgn0000256:E004        250        189        201          219          507
FBgn0000256:E005         96         38         39           71           76
FBgn0000256:E006          1          0          1            0            2
FBgn0000256:E007        149         70         71          130          281
                 untreated3fb untreated4fb
FBgn0000256:E001           51           49
FBgn0000256:E002           82           95
FBgn0000256:E003          260          297
FBgn0000256:E004          242          250
FBgn0000256:E005           57           62
FBgn0000256:E006            0            2
FBgn0000256:E007          115           94
```

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 *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( fData(pasillaExons), 3 )
```

```
                        geneID exonID testable dispBeforeSharing dispFitted
FBgn0000256:E001 FBgn0000256   E001       NA                NA         NA
FBgn0000256:E002 FBgn0000256   E002       NA                NA         NA
FBgn0000256:E003 FBgn0000256   E003       NA                NA         NA
                 dispersion pvalue padjust   chr   start     end strand
FBgn0000256:E001         NA     NA      NA chr2L 3872658 3872947      -
FBgn0000256:E002         NA     NA      NA chr2L 3873019 3873322      -
FBgn0000256:E003         NA     NA      NA chr2L 3873385 3874395      -

FBgn0000256:E001 FBtr0077511;FBtr0077513;FBtr0077512;FBtr0290077;FBtr0290079;FBtr0290078;FBtr0290082;F
FBgn0000256:E002 FBtr0077511;FBtr0077513;FBtr0077512;FBtr0290077;FBtr0290079;FBtr0290078;FBtr0290082;F
FBgn0000256:E003 FBtr0077511;FBtr0077513;FBtr0077512;FBtr0290077;FBtr0290079;FBtr0290078;FBtr0290082;F
```

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. Further columns for intermediate and final results are already present (and filled with `NA`) and will be filled with results in subsequent steps of the analysis.

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

```
> design(ecs)

             condition        type
treated1fb     treated single-read
treated2fb     treated  paired-end
treated3fb     treated  paired-end
untreated1fb untreated single-read
untreated2fb untreated single-read
untreated3fb untreated  paired-end
untreated4fb untreated  paired-end
```

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

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`.

```
> ecs <- estimateSizeFactors( ecs )
```

You may want to inspect the estimated size factors

```
> sizeFactors(ecs)
```

```
  treated1fb   treated2fb   treated3fb untreated1fb untreated2fb untreated3fb
        1.34         0.80         0.92         0.99         1.57         0.84
untreated4fb
        0.83
```

3.3 Dispersion estimation

To test for differential expression, 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 expression 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 4, we discuss how to treat more complicated experimental designs which are not accommodated by a single `condition` factor.

The first step is to get a dispersion estimate for each exon. This task is performed by the function `estimateDispersions`, using Cox-Reid (CR) likelihood estimation (our approach here follows that of the package *edgeR* [10]). Before starting estimating the CR dispersion estimates, `estimateDispersions` first defines the "testable" counting bins. In this step, all bins are excluded with less that `minCount` reads. By default, `minCount=10`, i.e., the bin must have at least 10 reads, *summed up* across all samples, but other values can be specified when calling `estimateDispersion`. If a gene has only one testable counting bin, then this counting bin is then considered as not testable, either, because it's usage cannot be compared to any other counting bins. The testable bins are marked in the column `testable` of the feature data.

Then, a dispersion estimate is computed for each bin, and the obtained values are stored in the feature data column `dispBeforeSharing`. The following command performs these steps.

```
> ecs <- estimateDispersions( ecs )
```

Note that for a full, genome-wide data set, execution of this function can take a while. To indicate progress, a dot is printed on the console whenever 100 genes have been processed. If you have a machine with multiple cores, you may want to use the `nCores` option to instruct the function to parallelize the task over several CPU cores. See Section 6 and the function's help page for details.

Afterwards, the function `fitDispersionFunction` needs to be called, in which a dispersion-mean relation $\alpha(\mu) = \alpha_0 + \alpha_1/\mu$ is fitted to the individual CR dispersion values (as stored in `dispBeforeSharing`). The fit coefficients are stored in the slot `dispFitCoefs` and finally, for each exon, the maximum between the dispersion before sharing and the fitted dispersion value is taken as the exon's final dispersion value and stored in the `dispersion` slot.[3] See our paper [1] for the rationale behind this "sharing" scheme.

[3]Especially when the dispersion estimates are very large, this fit can be difficult, and has occasionally caused the function to fail. In these rare cases please contact the developers.

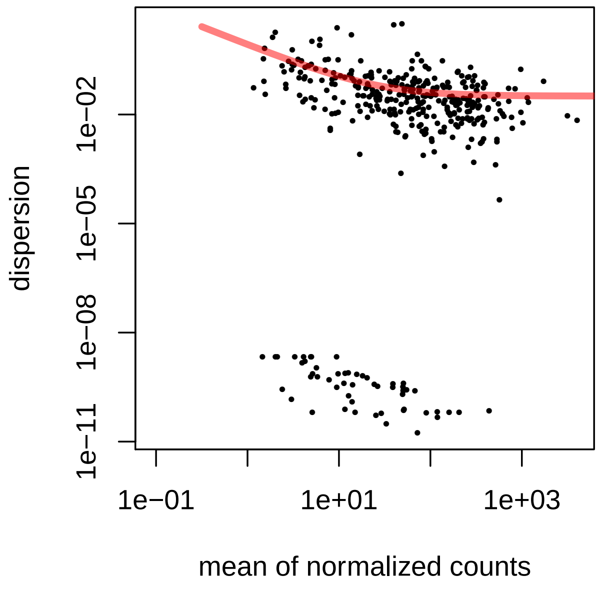

Figure 1: **Fit Diagnostics.** Per-gene dispersion estimates (shown by points) and the fitted mean-dispersion function (red line).

```
> ecs <- fitDispersionFunction( ecs )
```

We can have a look at the results of the CR estimation, the coefficients from the fit, and the fitted values:

```
> head( fData(ecs)$dispBeforeSharing )

[1] 0.0119 0.0021 0.0072 0.0076 0.0444     NA

> ecs@dispFitCoefs

     (Intercept) I(1/means[good])
           0.032            0.815

> head( fData(ecs)$dispFitted )

[1] 0.046 0.040 0.035 0.035 0.046 1.067
```

As a fit diagnostic, we plot the per-exon dispersion estimates versus the mean normalised count.

```
> plotDispEsts( ecs )
```

The plot (Figure 1) shows the estimates for all exons as dots and the fit as red line. The red line follows the trend of the dots in the upper cluster of dots. The lower cluster stems from exons for which the sample noise happens to fall below shot noise, i.e., their sample estimates of the dispersion is zero or close to zero and hence form another cluster at the bottom. The fact that these two clusters look so well separated is to a large extent an artefact of the logarithmic y-axis scaling. Inspect the fit and make sure that the regression line follows the trend of the points within the upper cluster.

The intercept coefficient of the dispersion fit (see above), which is the horizontal asymptote of the red line, can be understood as the square of the coefficient of variation of highly expressed exons, and is a good rough estimate of the overall noise in the data set and hence of the available power to infer differential exon usage,

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

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

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 *ExonCountSet* object at the beginning of the analysis, and `exon` is a factor with two levels, `this` and `others`. 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 [1] and used in older versions of *DEXSeq*; see Appendix C for further discussion.

Readers familiar with linear model formulae might find one aspect of Equation (2) 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, giving rise to a p value. Based on that, we can decide whether the null model (3) is sufficient to explain the data, or whether it may be rejected in favour of the alternative, model (2), 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.

```
> ecs <- testForDEU( ecs )
```

The function stores its results in the `pvalue` and `padjust` columns of the `featureData` slots of the *ExonCountSet* object with the results. Here, `pvalue` contains the p values from the χ^2 likelihood-ratio test, and `padjust` is the result of performing Benjamini-Hochberg adjustment for multiple testing on `pvalue`:

```
> head( fData(ecs)[, c( "pvalue", "padjust" ) ] )

                 pvalue padjust
FBgn0000256:E001   0.94       1
FBgn0000256:E002   0.47       1
FBgn0000256:E003   0.92       1
FBgn0000256:E004   0.81       1
FBgn0000256:E005   0.87       1
FBgn0000256:E006     NA      NA
```

For some usages, we may also want to estimate fold changes. To this end, we call `estimatelog2FoldChanges`:

```
> ecs <- estimatelog2FoldChanges( ecs )
```

This function stores its results in the feature data table, too.

A convenient way to extract all relevant columns of the feature data table is offered by the function `DEUresultTable`:

```
> res1 <- DEUresultTable(ecs)
> head( res1 )

                      geneID exonID dispersion pvalue padjust meanBase
FBgn0000256:E001 FBgn0000256   E001      0.046   0.94       1    58.34
FBgn0000256:E002 FBgn0000256   E002      0.040   0.47       1   103.33
FBgn0000256:E003 FBgn0000256   E003      0.035   0.92       1   326.48
FBgn0000256:E004 FBgn0000256   E004      0.035   0.81       1   253.65
FBgn0000256:E005 FBgn0000256   E005      0.046   0.87       1    60.64
FBgn0000256:E006 FBgn0000256   E006      1.067     NA      NA     0.79
                 log2fold(untreated/treated)
FBgn0000256:E001                       0.025
FBgn0000256:E002                      -0.045
FBgn0000256:E003                       0.025
FBgn0000256:E004                       0.038
FBgn0000256:E005                      -0.013
FBgn0000256:E006                       0.126
```

Controlling false discovery rate (FDR) at 0.1 (10%), we can now ask how many counting bins show evidence of differential exon usage:

```
> table ( res1$padjust < 0.1 )

FALSE  TRUE
  375     8
```

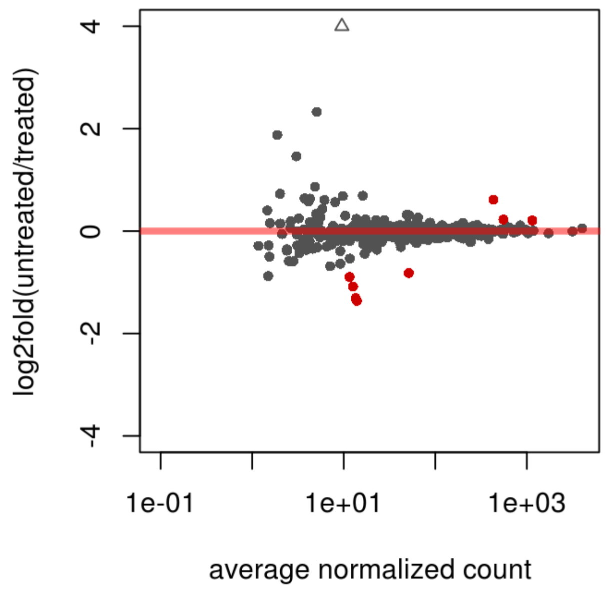

Figure 2: **MA plot.** Mean expression versus $\log_2$ fold change plot. Significant hits (at `padj`<0.1) are coloured in red.

We may also ask how many genes are affected

```
> table ( tapply( res1$padjust < 0.1, geneIDs(pasillaExons), any ) )
```

```
FALSE  TRUE 
   15     7 
```

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 2). There is of course nothing special about the number 0.1, and you can specify other thresholds in the call to `plotMA`.

```
> plotMA( ecs, FDR=0.1, ylim=c(-4,4), cex=0.8 )
```

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 *ExonCountSet* object (Section 2.6. 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 *ExonCountSet* object in Section 2.6, we provided in the sample table an additional column called `type`, which has been stored in the object:

```
> design(pasillaExons)

             condition        type
treated1fb     treated single-read
treated2fb     treated  paired-end
treated3fb     treated  paired-end
untreated1fb untreated single-read
untreated2fb untreated single-read
untreated3fb untreated  paired-end
untreated4fb untreated  paired-end
```

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

```
> formulaFullModel    <-  ~ sample + exon + type:exon + condition:exon
> formulaReducedModel <-  ~ sample + exon + type:exon
```

Compare these formulae with the default formulae (2, 3) given in Section 3.4. We have added, in both the full model and the reduced model, the term `type: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`.

To start a fresh analysis, now using these formulae instead of the default ones, we copy the example data into a new object.

```
> ecs2 <- pasillaExons
```

As before, we first estimate the size factors

```
> ecs2 <- estimateSizeFactors( ecs2 )
```

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 (2).

```
> ecs2 <- estimateDispersions( ecs2, formula = formulaFullModel )
```

The fit is performed as before.

```
> ecs2 <- fitDispersionFunction( ecs2 )
```

The test function now needs to be informed about both formulae

```
> ecs2 <- testForDEU( ecs2,
+          formula0 = formulaReducedModel, formula1 = formulaFullModel )
```

Finally, we get a summary table, as before.

```
> res2 <- DEUresultTable( ecs2 )
> head(res2)

                      geneID exonID dispersion pvalue padjust meanBase
FBgn0000256:E001 FBgn0000256   E001      0.026   0.90    0.99    58.34
FBgn0000256:E002 FBgn0000256   E002      0.020   0.40    0.99   103.33
FBgn0000256:E003 FBgn0000256   E003      0.014   0.85    0.99   326.48
FBgn0000256:E004 FBgn0000256   E004      0.015   0.53    0.99   253.65
FBgn0000256:E005 FBgn0000256   E005      0.056   0.88    0.99    60.64
FBgn0000256:E006 FBgn0000256   E006      1.097     NA      NA     0.79
```

How many significant DEU cases have we got this time?

```
> table( res2$padjust < 0.1 )

FALSE  TRUE
  374     9
```

We can now compare with the previous result:

```
> table( before = res1$padjust < 0.1, now = res2$padjust < 0.1 )

       now
before  FALSE TRUE
  FALSE   374    1
  TRUE      0    8
```

Accounting for the library type has allowed us to find one more hit. (With so few genes, this could be coincidence, of course. However, performing the analysis on the full pasilla data confirms that accounting for the covariate improves power.)

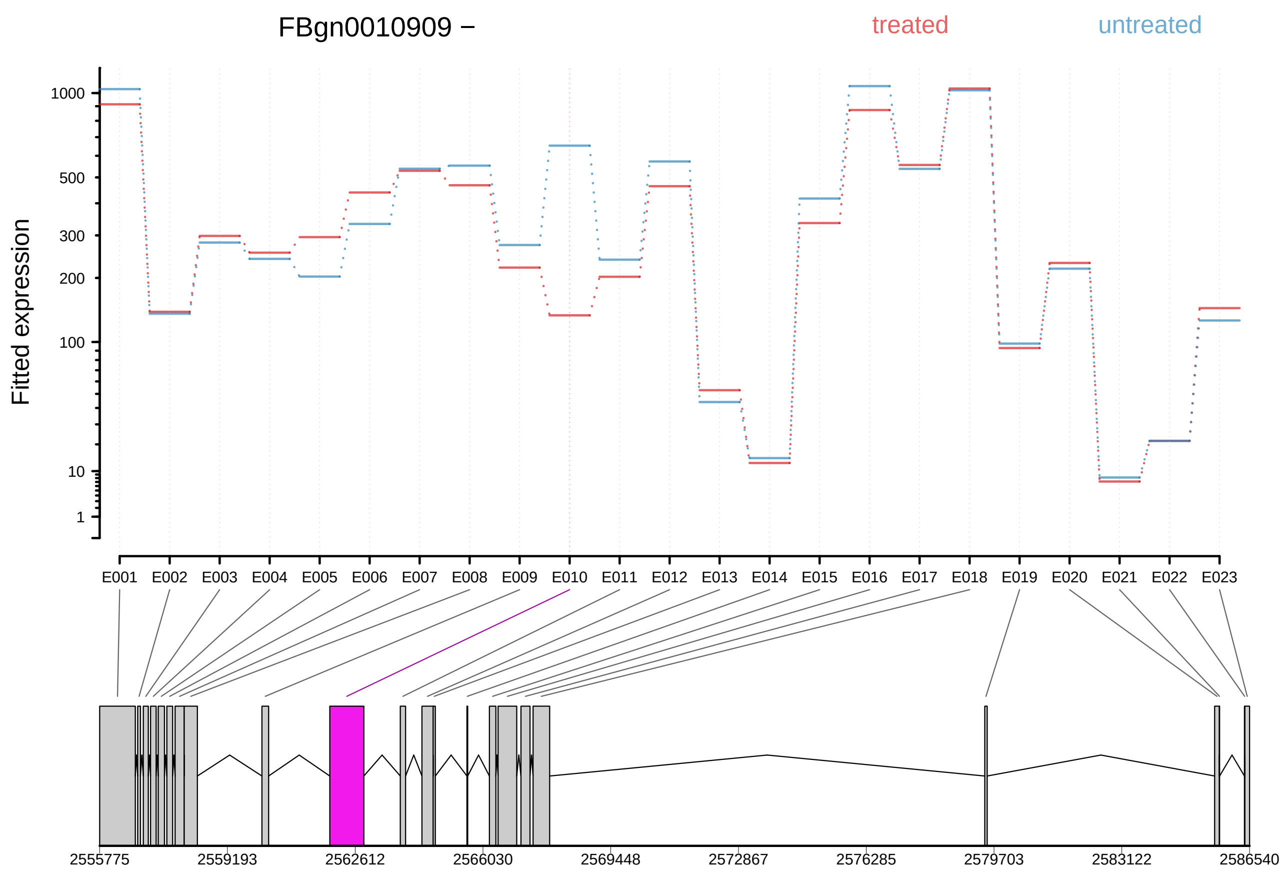

Figure 3: **Fitted expression.** The plot represents the expression estimates from a call to `testForDEU`. Shown in red is the exon that showed significant differential exon usage.

5 Visualization

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

```
> plotDEXSeq( ecs2, "FBgn0010909", legend=TRUE, cex.axis=1.2, cex=1.3, lwd=2 )
```

The result is shown in Figure 3. 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 *ExonCountSet* 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 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 4), which can be useful for putting differential exon usage results into the context of isoform regulation.

```
> plotDEXSeq( ecs2, "FBgn0010909", displayTranscripts=TRUE, legend=TRUE,
+     cex.axis=1.2, cex=1.3, lwd=2 )
```

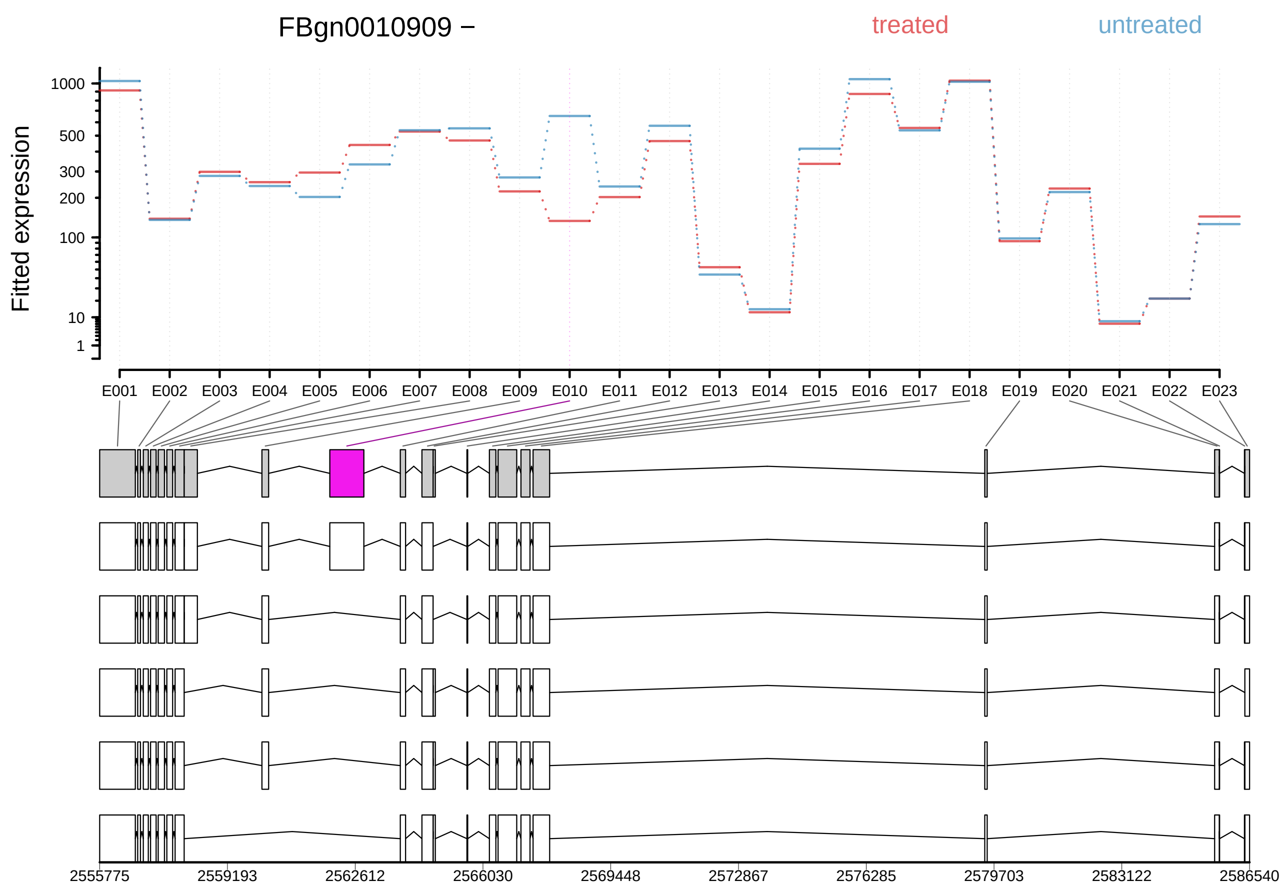

Figure 4: **Transcripts.** As in Figure 3, but including the annotated transcript models.

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 5).

```
> plotDEXSeq( ecs2, "FBgn0010909", expression=FALSE, norCounts=TRUE,
+    legend=TRUE, cex.axis=1.2, cex=1.3, lwd=2 )
```

As explained in Section 1, *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 expression from the plots. Use the (somewhat misnamed) option `splicing=TRUE` for this purpose.

```
> plotDEXSeq( ecs2, "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* [11] to create a result table with links to plots for the significant results, allowing a more detailed exploration of the results.

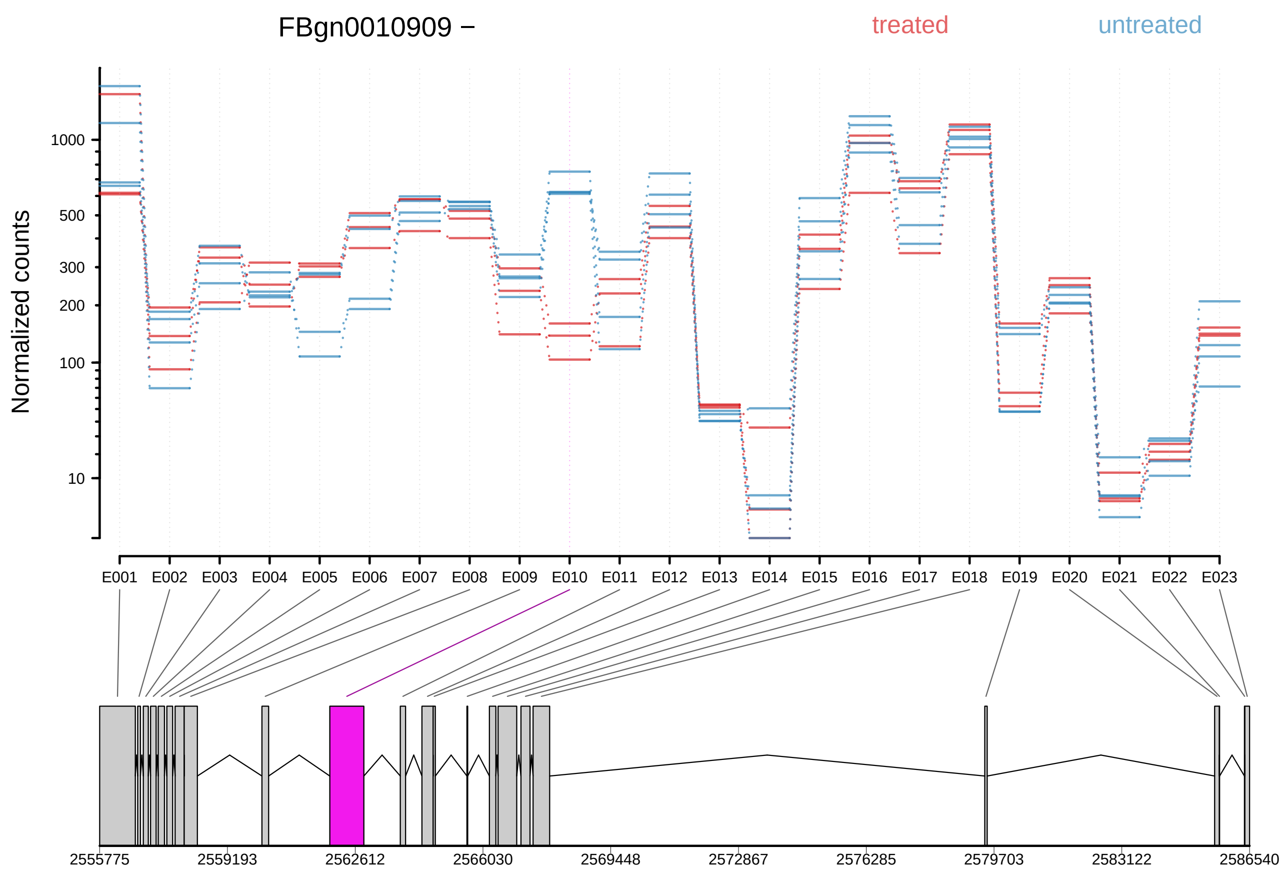

Figure 5: **Normalized counts.** As in Figure 3, with normalized count values of each exon in each of the samples.

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

6 Parallelization

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 two parts that are most time consuming – the functions `estimateDispersions` and `testForDEU` – can be parallelized by setting the two functions' parameter `nCores` to the number of CPU cores to be used. These functions will then distribute the *ExonCountSet* object into smaller objects that are processed in parallel on different cores. This functionality uses the *parallel* package, which hence has to be loaded first.

```
> data("pasillaExons", package="pasilla")
> library("parallel")
> pasillaExons <- estimateSizeFactors( pasillaExons )
> pasillaExons <- estimateDispersions( pasillaExons, nCores=16)
```

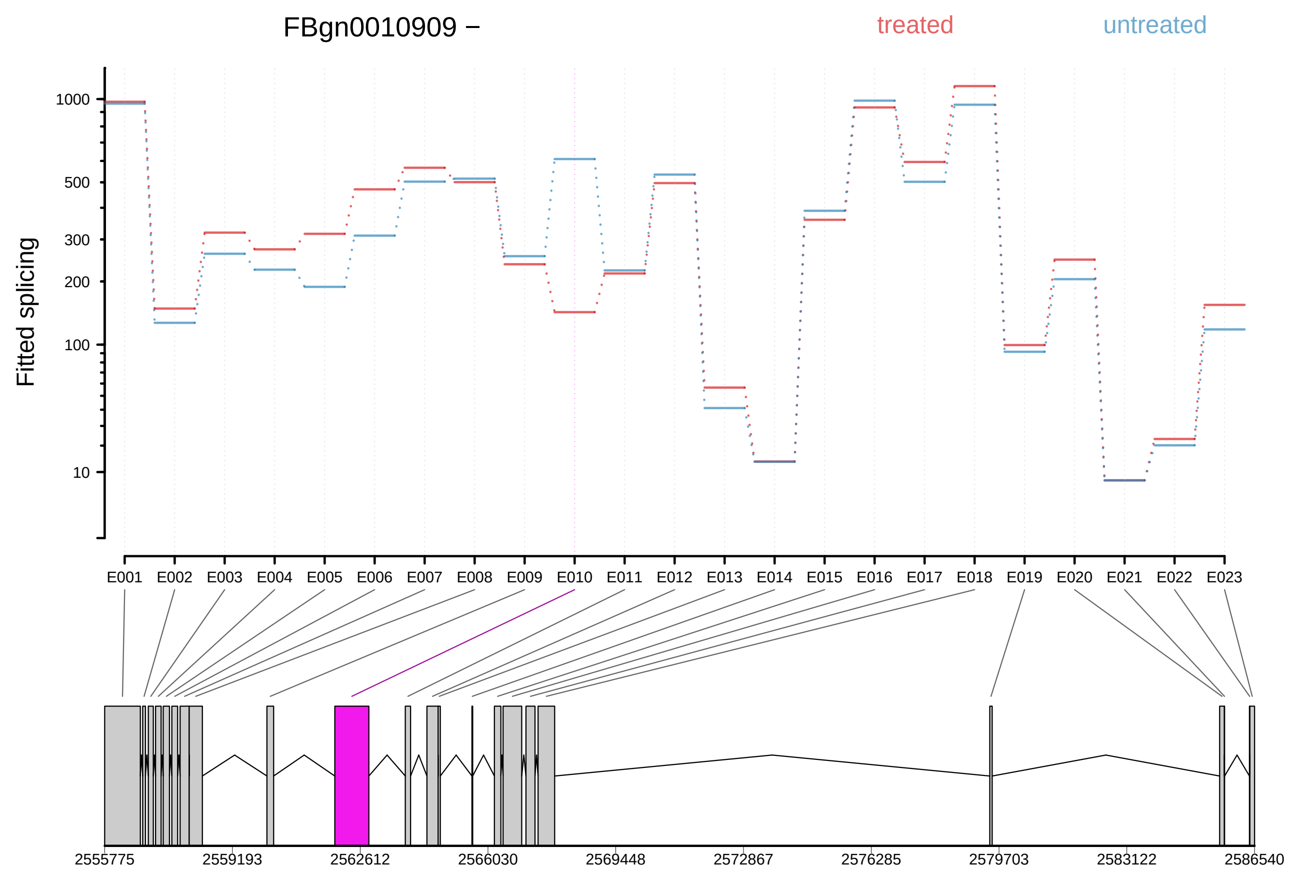

Figure 6: **Fitted splicing.** The plot represents the estimated effects, as in Figure 3, but after subtraction of overall changes in gene expression.

```
> pasillaExons <- fitDispersionFunction( pasillaExons )
> pasillaExons <- testForDEU( pasillaExons, nCores=16)
```

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 `doCompleteDEUAnalysis`, which runs the analysis shown above through a single function call.

In the simplest case, construct the `ExonCountSet` as shown in Section 2 or in Appendix A, then run `doCompleteDEUAnalysis` passing the `ExonCountSet` as only argument, and finally inspect the result with `DEUresultTable`. In the following, we also specify design formulae to account for library type as described in Section 4 and the `nCores` argument for parallelization (Section 6).

```
> data( "pasillaExons", package="pasilla" )
> pasillaExons <- doCompleteDEUAnalysis(
+    pasillaExons,
+    formula0 =  ~ sample + exon + type:exon,
+    formula1 =  ~ sample + exon + type:exon + condition:exon,
+    nCores = 1 )

> head( DEUresultTable( pasillaExons ) )

                      geneID exonID dispersion pvalue padjust meanBase
FBgn0000256:E001 FBgn0000256   E001      0.026   0.90    0.99    58.34
FBgn0000256:E002 FBgn0000256   E002      0.020   0.40    0.99   103.33
FBgn0000256:E003 FBgn0000256   E003      0.014   0.85    0.99   326.48
FBgn0000256:E004 FBgn0000256   E004      0.015   0.53    0.99   253.65
FBgn0000256:E005 FBgn0000256   E005      0.056   0.88    0.99    60.64
FBgn0000256:E006 FBgn0000256   E006      1.097     NA      NA     0.79
                 log2fold(untreated/treated)
FBgn0000256:E001                    -0.00094
FBgn0000256:E002                    -0.07526
FBgn0000256:E003                     0.01082
FBgn0000256:E004                     0.02590
FBgn0000256:E005                    -0.03983
FBgn0000256:E006                     0.09629
```

APPENDIX

A Preprocessing within R

As an alternative to the approach described in Section 2, users can also create *ExonCountSet* 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 [8]. 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("GenomicFeatures")
> library("GenomicRanges")
> library("Rsamtools")
```

We demonstrate the workflow briefly (for more details, see [8]) on the data set of Haglund et al. [12], 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 <- makeTranscriptDbFromBiomart( biomart="ensembl",
+    dataset="hsapiens_gene_ensembl" )
```

Next, we collapse the gene models into counting bins, analogous to Section 2.4.

```
> exonicParts <- disjointExons( hse, aggregateGenes=TRUE )
```

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 2.4. 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 `countReadsForDEXSeq`. 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 `buildExonCountSet` to build an `ExonCountSet` object. The middle argument in the following call is the design, indicating that the first two BAM files form one experimental condition and the third one the other. Generally, you will want to pass a *data.frame* with a sample table as in Section 2.6.

```
> ecs <- buildExonCountSet( SE, c("A", "A", "B"), exonicParts )
```

B Lower-level functions

The following functions are not needed in a standard analysis work-flow, but may be useful for special purposes. We list them here briefly; see the help pages of these function for further options (e. g., to specify formulae).

B.1 Count tables

While the function `counts` returns the whole read count table, the function `countTableForGene` returns the count table for a single gene:

```
> countTableForGene( pasillaExons, "FBgn0010909" )
```

Like the function `counts`, the function `countTableForGene` can also return normalized counts (i.e., counts divided by the size factors). Use the option `normalized=TRUE`.

```
> head( countTableForGene( pasillaExons, "FBgn0010909", normalized=TRUE ) )

     treated1fb treated2fb treated3fb untreated1fb untreated2fb untreated3fb
E001       1495        618        609         1161         1603          680
E002         91        140        195           70          129          186
E003        207        366        331          192          254          372
E004        314        250        197          232          284          218
E005        311        271        302          147          108          283
E006        364        446        511          192          215          499
     untreated4fb
E001          659
E002          171
E003          312
E004          223
E005          278
E006          439
```

The function `geneCountTable` computes a table of *gene counts*, which are obtained by summing the counts from all exons with the same geneID. This might be useful for the detection of differential expression of genes, where the table can be used as input e. g. for the packages *DESeq* or *edgeR*. This kind of table can also be produced with the package *GenomicRanges*, e. g. with the function `summarizeOverlaps`.

```
> head( geneCountTable( pasillaExons ) )

            treated1fb treated2fb treated3fb untreated1fb untreated2fb
FBgn0000256       1482        857        966         1169         2626
FBgn0000578       4386       2301       2827         3541         6381
FBgn0002921      11305       7135       8001         7433        11980
FBgn0003089          8          4          4            6            9
FBgn0010226        129        100        113          106          126
```

```
FBgn0010280       2693       1776       2187        2088        3963
            untreated3fb untreated4fb
FBgn0000256         1105         1101
FBgn0000578         3139         2725
FBgn0002921         5618         5991
FBgn0003089            4            6
FBgn0010226           60           99
FBgn0010280         2069         1981
```

Note that a read that mapped to several exons of a gene is counted for each of these exons by the `dexseq_count.py` script. The table returned `geneCountTable` will hence count the read several time for the gene, which may skew downstream analyses in subtle ways. Hence, we recommend to use `geneCountTable` with care and use more sophisticated counting schemes where appropriate.

B.2 Model frames

The function `constructModelFrame` returns the model frame used for the dispersion fits and the model fits involved in the likelihood ratio test. The terms used in the formulae passed to `estimateDispersions` and `testForDEU` refer to this model frame and hence must appear as column names.

```
> constructModelFrame( pasillaExons )
```

```
         sample condition        type sizeFactor   exon
1    treated1fb   treated single-read       1.34   this
2    treated2fb   treated  paired-end       0.80   this
3    treated3fb   treated  paired-end       0.92   this
4  untreated1fb untreated single-read       0.99   this
5  untreated2fb untreated single-read       1.57   this
6  untreated3fb untreated  paired-end       0.84   this
7  untreated4fb untreated  paired-end       0.83   this
8    treated1fb   treated single-read       1.34 others
9    treated2fb   treated  paired-end       0.80 others
10   treated3fb   treated  paired-end       0.92 others
11 untreated1fb untreated single-read       0.99 others
12 untreated2fb untreated single-read       1.57 others
13 untreated3fb untreated  paired-end       0.84 others
14 untreated4fb untreated  paired-end       0.83 others
```

For the visualization with `plotDEXSeq`, a GLM is fitted over the joint data from all exons of a gene. The function `modelFrameForGene` returns the model frame used for this fit for a single gene.

```
> mf <- modelFrameForGene( pasillaExons, "FBgn0010909" )
> head( mf )
```

```
      sample exon sizeFactor condition        type dispersion count
1 treated1fb E001        1.3   treated single-read      0.013  1997
2 treated1fb E002        1.3   treated single-read      0.025   122
3 treated1fb E003        1.3   treated single-read      0.015   276
4 treated1fb E004        1.3   treated single-read      0.015   420
5 treated1fb E005        1.3   treated single-read      0.051   416
6 treated1fb E006        1.3   treated single-read      0.020   486
```

B.3 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(pasillaExons) )
```

```
FBgn0000256:E001 FBgn0000256:E002 FBgn0000256:E003 FBgn0000256:E004
     FBgn0000256      FBgn0000256      FBgn0000256      FBgn0000256
FBgn0000256:E005 FBgn0000256:E006
     FBgn0000256      FBgn0000256
14470 Levels: FBgn0000003 FBgn0000008 FBgn0000014 FBgn0000015 ... FBgn0261575
```

```
> head( exonIDs(pasillaExons) )
```

```
FBgn0000256:E001 FBgn0000256:E002 FBgn0000256:E003 FBgn0000256:E004
          "E001"           "E002"           "E003"           "E004"
FBgn0000256:E005 FBgn0000256:E006
          "E005"           "E006"
```

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

C Methodological changes since publication of the paper

In our paper [1], 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.

For now, the function for our original approach (which we now call the "big model" or "BM" approach) are still included; all relevant functions, however, have been renamed to carry the suffix `_BM` in their name. The new approach, which is now default and is used by the work flow described in this vignette, has no special name (in some previous releases of *DEXSeq* which had included it first on an experimental basis, it was termed the "TRT" approach).

In the following, we describe the current default ("TRT") approach in detail (though the exposition assumes the reader's familiarity with our paper).

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

$$K_{ijl} \sim \mathrm{NB}(\mathrm{mean} = s_j \mu_{ijl}, \mathrm{dispersion} = \alpha_i), \tag{4}$$

now with the model specified in Equation (2), which we write out as

$$\log_2 \mu_{ijl} = \beta^{\mathrm{S}}_{ij} + l\beta^{\mathrm{E}}_{i} + \beta^{\mathrm{EC}}_{i\rho_j}. \tag{5}$$

This model is fit separately for each counting bin i. The coefficient β^{S}_{ij} accounts for the sample-specific contribution (factor `sample` in Equation (2)), the term β^{E}_{i} 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, $\beta^{\mathrm{EC}}_{i,\rho_j}$, 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^{EC}_{i0} 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. For the likelihood ratio test, this full model is fit and compared with the fit of the reduced model, which lacks the interaction term $\beta^{\mathrm{EC}}_{i\rho_j}$. As described in Section 4, alternative model formulae can be specified.

D Requirements on GTF files

In the initial preprocessing step described in Section 2.4, 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 3.0.2 (2013-09-25)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

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

other attached packages:
[1] DEXSeq_1.8.0       Biobase_2.22.0     BiocGenerics_0.8.0

loaded via a namespace (and not attached):
 [1] BiocStyle_1.0.0      Biostrings_2.30.0    GenomicRanges_1.14.0
 [4] IRanges_1.20.0       RCurl_1.95-4.1       Rsamtools_1.14.0
 [7] XML_3.98-1.1         XVector_0.2.0        biomaRt_2.18.0
[10] bitops_1.0-6         hwriter_1.3          statmod_1.4.18
```

```
[13] stats4_3.0.2        stringr_0.6.2       tools_3.0.2
[16] zlibbioc_1.8.0
```

F References

[1] Simon Anders, Alejandro Reyes, and Wolfgang Huber. Detecting differential usage of exons from RNA-seq data. *Genome Research*, 22:2008, 2012.

[2] Angela N. Brooks, Li Yang, Michael O. Duff, Kasper D. Hansen, Jung W. Park, Sandrine Dudoit, Steven E. Brenner, and Brenton R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*, 21:193–202, 2011.

[3] Alejandro Reyes. Data preprocessing and creation of the data objects pasillaGenes and pasillaExons. Vignette to Bioconductor data package *pasilla*, `http://bioconductor.org/packages/release/data/experiment/html/pasilla.html`, 2013.

[4] 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.

[5] Thomas D. Wu and Serban Nacu. Fast and SNP-tolerant detection of complex variants and splicing in short reads. *Bioinformatics*, 26(7):873–881, 2010.

[6] Alexander Dobin, Carrie A. Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R. Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[7] Simon Anders, Davis J. McCarthy, Yunshen Chen, Michal Okoniewski, Gordon K. Smyth, Wolfgang Huber, and Mark D. Robinson. Count-based differential expression analysis of RNA sequencing data using R and Bioconductor. *Nature Protocols*, 8:1765, 2013.

[8] Michael Love. SummarizedExperiment for RNA-Seq of primary cultures of parathyroid tumors by Haglund et al., J Clin Endocrinol Metab 2012. Vignette to Bioconductor data package *parathyroidSE*, `http://bioconductor.org/packages/release/data/experiment/html/parathyroidSE.html`, 2013.

[9] Seth Falcon, Martin Morgan, and Robert Gentleman. An introduction to Bioconductor's ExpressionSet class. Vignette, provided with Bioconductor package *Biobase*, `http://http://www.bioconductor.org/packages/release/bioc/vignettes/Biobase/inst/doc/ExpressionSetIntroduction.pdf`, 2007.

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

[11] Gregoire Pau and Wolfgang Huber. The hwriter package. *The R Journal*, 1:22, 2009.

[12] Felix Haglund, Ran Ma, Mikael Huss, Luqman Sulaiman, Ming Lu, Inga-Lena Nilsson, Anders Höög, C. Christofer Juhlin, Johan Hartman, and Catharina Larsson. Evidence of a functional estrogen receptor in parathyroid adenomas. *Journal of Clinical Endocrinology & Metabolism*, 97(12), 2012.