JunctionSeq Package User Manual

Stephen Hartley
National Human Genome Research Institute
National Institutes of Health

November 16, 2015

Contents

1 Overview

The JunctionSeq R package offers a powerful tool for detecting, identifying, and characterizing differential usage of transcript exons and/or splice junctions in next-generation, high-throughput RNA-Seq experiments.

"Differential usage" is defined as a differential expression of a particular feature or sub-unit of a gene (such as an exon or spice junction) relative to the overall expression of the gene as a whole (which may or may not itself be differentially expressed). The term was originally used by Anders et.al. [1] for their tool, DEXSeq, which is designed to detect differential usage of exonic sub-regions in RNA-Seq data. Tests for differential usage can be used as a proxy for detecting isoform-specific differential regulation such as isoform switching, alternative splicing, alternative start site usage, or alternative stop site usage.

JunctionSeq builds upon and expands the basic design put forth by DEXSeq, providing (among other things) the ability to test for both differential exon usage and differential splice junction usage. These two types of analyses are complementary: Exons represent a broad region on the transcripts, exons tend to have higher counts than the individual splice junctions (particularly for large exons) and as such tests for differential exon usage will often have higher power under ideal conditions. However, certain combinations of isoform differentials may not result in strong observed differentials in the individual exons, and up- or down-regulation in unannotated isoforms may not be detectable at all. The addition of splice junction usage analysis makes it possible to detect a broader array of isoform-regulation phenomena, as well as vastly improving the detection of differentials in unannotated isoforms.

JunctionSeq is *not* designed to detect changes in overall gene expression. Gene-level differential expression is best detected with tools designed specifically for that purpose such as DESeq2 [2] or edgeR [3].

The core advantage to JunctionSeq is that it provides powerful and comprehensive automated visualization tools designed to assist in the interpretation of the results. JunctionSeq produces an array of intuitive expression plots along with genome-wide browser tracks for use with IGV or the UCSC genome browser.

Additional help and documentation is available online. There is also a comprehensive walkthrough of the entire analysis pipeline, along with a full example dataset with example bam files.

2 Requirements

JunctionSeq requires R v3.1.1 as well as a number of R packages. The easiest way to install it is to use the automated installer. Note that to install JunctionSeq on windows, you must have Rtools.

```
source("http://hartleys.github.io/JunctionSeq/install/JS.install.R");
JS.install();
```

Alternatively it can be installed manually:

```
#CRAN package dependencies:
install.packages("statmod");
install.packages("plotrix");
install.packages("stringr");
install.packages("Rcpp");
install.packages("RcppArmadillo");
install.packages("locfit");

#Bioconductor dependencies:
source("http://bioconductor.org/biocLite.R");
biocLite();
biocLite("Biobase");
biocLite("BiocGenerics");
biocLite("BiocParallel");
biocLite("GenomicRanges");
biocLite("IRanges");
biocLite("S4Vectors");
biocLite("genefilter");
biocLite("geneplotter");
biocLite("SummarizedExperiment");

install.packages("http://hartleys.github.io/JunctionSeq/install/JunctionSeq_LATEST.tar.gz"
                 repos = NULL,
                 type = "source");
```

Generating splice junction counts: The easiest way to generate the counts for JunctionSeq is via the QoRTs software package. The QoRTs software package requires R version 3.0.2 or higher, as well as

(64-bit) java 6 or higher. QoRTs can be found online here.

Hardware: Both the JunctionSeq and QoRTs software packages will generally require at least 2-10 gigabytes of RAM to run. In general at least 4gb is recommended when available for the QoRTs runs, and at least 10gb for the JunctionSeq runs. R multicore functionality usually involves duplicating the R environment, so memory usage may be higher if multicore options are used.

Annotation: JunctionSeq requires a transcript annotation in the form of a gtf file. If you are using a annotation guided aligner (which is STRONGLY recommended) it is likely you already have a transcript gtf file for your reference genome. We recommend you use the same annotation gtf for alignment, QC, and downstream analysis. We have found the Ensembl "Gene Sets" gtf[1] suitable for these purposes. However, any format that adheres to the gtf file specification[2] will work.

Dataset: JunctionSeq requires aligned RNA-Seq data. Data can be paired-end or single-end, unstranded or stranded. For paired-end data it is strongly recommended (but not explicitly required) that the SAM/BAM files be sorted either by name or position (it does not matter which).

2.1 Alignment

QoRTs, which is used to generate read counts, is designed to run on paired-end or single-end next-gen RNA-Seq data. The data must first be aligned (or "mapped") to a reference genome. RNA-Star [4], GSNAP [5], and TopHat2 [6] are all popular and effective aligners for use with RNA-Seq data. The use of short-read or unspliced aligners such as BowTie, ELAND, BWA, or Novoalign is NOT recommended.

2.2 Recommendations

Using barcoding, it is possible to build a combined library of multiple distinct samples which can be run together on the sequencing machine and then demultiplexed afterward. In general, it is recommended that samples for a particular study be multiplexed and merged into "balanced" combined libraries, each containing equal numbers of each biological condition. If necessary, these combined libraries can be run across multiple sequencer lanes or runs to achieve the desired read depth on each sample.

This reduces "batch effects", reducing the chances of false discoveries being driven by sequencer artifacts or biases.

It is also recommended that the data be thoroughly examined and checked for artifacts, biases, or errors using the QoRTs quality control package.

[1] Which can be acquired from the Ensembl website
[2] See the gtf file specification here

3 Example Dataset

To allow users to test JunctionSeq and experiment with its functionality, an example dataset is available online here. It can be installed with the command:

```
install.packages("http://hartleys.github.io/JunctionSeq/install/JctSeqExData2_LATEST.tar.gz
```

A more complete walkthrough using this same dataset is available here. The bam files are available here.

The example dataset was taken from rat pineal glands. Sequence data from six samples (aka "biological replicates") are included, three harvested during the day, three at night. To reduce the file sizes to a more managable level, this dataset used only 3 out of the 6 sequencing lanes, and only the reads aligning to chromosome 14 were included. This yielded roughly 750,000 reads per sample. The example dataset, including aligned reads, QC data, example scripts, splice junction counts, and JunctionSeq results, is available online (see the JunctionSeq github page).

THIS EXAMPLE DATASET IS INTENDED FOR TESTING PURPOSES ONLY! The original complete dataset can be downloaded from GEO (the National Center for Biotechnology Information Gene Expression Omnibus), GEO series accession number GSE63309. The test dataset has been cut down to reduce file sizes and processing time, and a number of artificial "edge cases" were introduced for testing purposes. For example: in the gene annotation, one gene has an artificial transcript that lies on the opposite strand from the other transcripts, to ensure that JunctionSeq deals with that (unlikely) possibility in a sensible way. The results from this test analysis are not appropriate for anything other than testing!

Splice junction counts and annotation files generated from this example dataset are included in the `JcnSeqExData` R package, available online (see the JunctionSeq github page), which is what will be used by this vignette.

The annotation files can be accessed with the commands:

```
decoder.file <- system.file("extdata/annoFiles/decoder.bySample.txt",
                            package="JctSeqExData2",
                            mustWork=TRUE);
decoder <- read.table(decoder.file,
                      header=TRUE,
                      stringsAsFactors=FALSE);
gff.file <- system.file(
             "extdata/cts/withNovel.forJunctionSeq.gff.gz",
             package="JctSeqExData2",
             mustWork=TRUE);
```

The count files can be accessed with the commands:

```
countFiles.noNovel <- system.file(paste0("extdata/cts/",
               decoder$sample.ID,
               "/QC.spliceJunctionAndExonCounts.forJunctionSeq.txt.gz"),
```

```
                package="JctSeqExData2", mustWork=TRUE);

countFiles <- system.file(paste0("extdata/cts/",
                decoder$sample.ID,
                "/QC.spliceJunctionAndExonCounts.withNovel.forJunctionSeq.txt.gz"),
                package="JctSeqExData2", mustWork=TRUE);
```

4 Preparations

Once alignment and quality control has been completed on the study dataset, the splice-junction and gene counts must be generated via QoRTs.

To reduce batch effects, the RNA samples used for the example dataset were barcoded and merged together into a single combined library. This combined library was run on three HiSeq 2000 sequencer lanes. Thus, after demultiplexing each sample consisted of three "technical replicates". JunctionSeq is only designed to compare biological replicates, so QoRTs includes functions for generating counts for each technical replicate and then combining the counts across the technical replicates from each biological sample.

4.1 Generating raw counts via QoRTs

To generate read counts, you must run QoRTs on each aligned bam file. QoRTs includes a basic function that calculates a variety of QC metrics along with gene-level and splice-junction-level counts. All these functions can be performed in a single step and a single pass through the input alignment file, greatly simplifying the analysis pipeline.

For example, to run QoRTs on the first read-group of replicate `SHAM1_RG1` from the example dataset:

```
java -jar /path/to/jarfile/QoRTs.jar QC \
                    --stranded \
                    inputData/bamFiles/SHAM1_RG1.bam \
                    inputData/annoFiles/anno.gtf.gz \
                    rawCts/SHAM1_RG1/
```

Note that the `--stranded` option is required because this example dataset is strand-specific. Also note that QoRTs uses the original gtf annotation file, NOT the flattened gff file produced in section 4.3. More information on this command and on the available options can be found online here.

If Quality Control is being done seperately by other software packages or collaborators, the JunctionSeq counts can be generated without the rest of the QC data by setting the `--runFunctions` option:

```
java -jar /path/to/jarfile/QoRTs.jar QC \
 --stranded \
 --runFunction writeKnownSplices,writeNovelSplices,writeSpliceExon \
 inputData/bamFiles/SHAM1_RG1.bam \
 inputData/annoFiles/anno.gtf.gz \
 rawCts/SHAM1_RG1/
```

This will take much less time to run, as it does not generate the full battery of quality control metrics.

The above script will generate a count file `rawCts/SHAM1_RG1/QC.spliceJunctionAndExonCounts.forJunctio` This file contains both gene-level coverage counts, as well as coverage counts for transcript subunits such as exons and splice junction loci.

For more information about the quality control metrics provided by QoRTs and how to visualize, organize, and view them, see the QoRTs github page and documentation, available online here.

4.2 Merging Counts from Technical Replicates (If Needed)

QoRTs includes functions for merging all count data from various technical replicates. If your dataset does not include technical replicates, or if technical replicates have already been merged prior to the count-generation step then this step is unnecessary.

The example dataset has three such technical replicates per sample, which were aligned separately and counted separately. For the purposes of quality control QoRTs was run separately on each of these bam files (making it easier to discern any lane or run specific artifacts that might have occurred). It is then necessary to combine the read counts from each of these bam files.

QoRTs includes an automated utility for performing this merge. For the example dataset, the command would be:

```
java -jar /path/to/jarfile/QoRTs.jar \
                mergeAllCounts \
                rawCts/ \
                annoFiles/decoder.byUID.txt \
                cts/
```

The "`rawCts/`" and "`cts/`" directories are the relative paths to the input and output data, respectively. The "decoder" file should be a tab-delimited text file with column titles in the first row. One of the columns must be titled "sample.ID", and one must be labelled "unique.ID". The unique.ID must be unique and refers to the specific technical replicate, the sample.ID column indicates which biological sample each technical replicate belongs to.

4.3 (Option 1) Including Only Annotated Splice Junction Loci

If you wish to only test annotated splice junctions, then a simple flat annotation file can be generated for use by JunctionSeq. This file parses the input gtf annotation and assigns unique identifiers to each feature (exon or splice junction) belonging to each gene. These identifiers will match the identifiers listed in the count files produced by QoRTs in the count-generation step (see Section 4.1).

```
java -jar /path/to/jarfile/QoRTs.jar makeFlatGtf \
                    --stranded \
                    annoFiles/anno.gtf.gz \
                    annoFiles/JunctionSeq.flat.gff.gz
```

Note it is *vitally important* that the flat gff file and the read-counts be run using the same "strandedness" options (as described in Section 4.1). If the counts are generated in stranded mode, the gff file must also be generated in stranded mode.

4.4 (Option 2) Including Novel Splice Junction Loci

One of the core advantages of JunctionSeq over similar tools such as DEXSeq is the ability to test for differential usage of previously-undiscovered transcripts via novel splice junctions. Most advanced aligners have the ability to align read-pairs to both known and unknown splice junctions. QoRTs can produce counts for these novel splice junctions, and JunctionSeq can test them for differential usage.

Because many splice junctions that appear in the mapped file may only have a tiny number of mapped reads, it is generally desirable to filter out low-coverage splice junctions. Many such splice junctions may simply be errors, and in any case their coverage counts will be too low to detect differential effects.

In order to properly filter by read depth, size factors are needed. These can be generated by JunctionSeq (see Section 5.1.1), or generated from gene-level read counts using DESeq2, edgeR, or QoRTs. See the QoRTs vignette for more information on how to generate these size factors.

Once size factors have been generated, novel splice junctions can be selected and counted using the command:

```
java -jar /path/to/jarfile/QoRTs.jar QC \
                mergeNovelSplices  \
                --minCount 6 \
                --stranded \
                cts/ \
                sizeFactors.GEO.txt \
                annoFiles/anno.gtf.gz \
                cts/
```

Note: The file sizeFactors.GEO.txt contains two columns, "sample.ID" and "size.factor".

This utility finds all splice junctions that fall inside the bounds of any known gene. It then filters this set of splice junctions, selecting only the junction loci with mean normalized read-pair counts of greater than the assigned threshold (set to 6 read-pairs in the example above). It then gives each splice junction that passes this filter a unique identifier.

This utility produces two sets of output files. First it writes a .gff file containing the unique identifiers for each annotated and novel-and-passed-filter splice locus. Secondly, for each sample it produces a count file that includes these additional splice junctions (along with the original counts as well).

5 Differential Usage Analysis via JunctionSeq

5.1 Simple analysis pipeline

JunctionSeq includes a single function that loads the count data and performs a full analysis automatically. This function internally calls a number of sub-functions and returns a JunctionSeqCountSet object that holds the analysis results, dispersions, parameter estimates, and size factor data.

```
jscs <- runJunctionSeqAnalyses(sample.files = countFiles,
                               sample.names = decoder$sample.ID,
                               condition=factor(decoder$group.ID),
                               flat.gff.file = gff.file,
                               nCores = 6,
                               analysis.type = "junctionsAndExons"
                               );
```

The default analysis type, which is explicitly set in the above command, performs a "hybrid" analysis that tests for differential usage of both exons *and* splice junctions simultaniously. The methods used to test for differential splice junction usage are extremely similar to the methods used for differential exon usage other than the fact that the counts are derived from splice sites rather than exonic regions. These methods are based on the methods used by the DEXSeq package [1, 2].

The advantage to our method is that reads (or read-pairs) are never counted more than once in any given statistical test. This is not true in DEXSeq, as the long paired-end reads produced by current sequencers may span several exons and splice junctions and thus be counted several times. Our method also produces estimates that are more intuitive to interpret: our fold change estimates are defined as the fold difference between two conditions across a transcript sub-feature relative to the fold change between the same two conditions for the gene as a whole. The DEXSeq method instead calculates a fold change relative to the sum of the other sub-features, which may change depending on the analysis inclusion parameters.

The above function has a number of optional parameters, which may be relevant to some analyses:

`analysis.type` : By default JunctionSeq simultaniously tests both splice junction loci and exonic regions for differential usage (a "hybrid" analysis). This parameter can be used to limit analyses specifically to either splice junction loci or exonic regions.

`nCores` : The number of cores to use (note: this will only work when package BiocParallel is used on a Linux-based machine. Unfortunately R cannot run multiple threads on windows at this time).

`meanCountTestableThreshold` : The minimum normalized-read-count threshold used for filtering out low-coverage features.

`test.formula0` : The model formula used for the null hypothesis in the ANODEV analysis.

`test.formula1` : The model formula used for the alternative hypothesis in the ANODEV analysis.

`effect.formula` : The model formula used for estimating the effect size and parameter estimates.

`geneLevel.formula` : The model formula used for estimating the gene-level expression.

A full description of all these options and how they are used can be accessed using the command:

```
help(runJunctionSeqAnalyses);
```

5.1.1 Exporting size factors

As part of the analysis pipeline, JunctionSeq produces size factors which can be used to "normalize" all samples' read counts to a common scale. These can be accessed using the command:

```
writeSizeFactors(jscs, file = "sizeFactors.txt");
```

```
##       sample.ID size.factor sizeFactors.byGenes sizeFactors.byCountbins
## SAMP1     SAMP1       1.076                1.076                    1.102
## SAMP2     SAMP2       0.988                0.988                    1.039
## SAMP3     SAMP3       0.997                0.997                    0.975
## SAMP4     SAMP4       0.945                0.945                    0.975
## SAMP5     SAMP5       1.088                1.088                    1.105
## SAMP6     SAMP6       0.926                0.926                    0.880
```

These size factors can be used to normalize read counts for the purposes of adding novel splice junctions with a fixed coverage depth threshold (see Section 4.4). By default these normalization size factors are based on the gene-level counts.

5.2 Advanced Analysis Pipeline

Some advanced users may need to deviate from the standard analysis pipeline. They may want to use different size factors, apply multiple different models without having to reload the data from file each time, or use other advanced features offered by JunctionSeq.

First you must create a "design" data frame:

```
design <- data.frame(condition = factor(decoder$group.ID));
```

Note: the experimental condition variable MUST be named "condition".

Next, the data must be loaded into a JunctionSeqCountSet:

```
jscs = readJunctionSeqCounts(countfiles = countFiles,
                             samplenames = decoder$sample.ID,
                             design = design,
                             flat.gff.file = gff.file
                             );
```

Next, size factors must be created and loaded into the dataset:

```
#Generate the size factors and load them into the JunctionSeqCountSet:
jscs <- estimateJunctionSeqSizeFactors(jscs);
```

Next, we generate test-specific dispersion estimates:

```
jscs <- estimateJunctionSeqDispersions(jscs, nCores = 6);
```

Next, we fit these observed dispersions to a regression to create fitted dispersions:

```
jscs <- fitJunctionSeqDispersionFunction(jscs);
```

Next, we perform the hypothesis tests for differential splice junction usage:

```
jscs <- testForDiffUsage(jscs, nCores = 6);
```

Finally, we calculate effect sizes and parameter estimates:

```
jscs <- estimateEffectSizes( jscs, nCores = 6);
```

All these steps simply duplicates the behavior of the runJunctionSeqAnalyses function. However, far more options are available when run in this way. Full documentation of the various options for these commands:

```
help(readJunctionSeqCounts);
help(estimateJunctionSeqSizeFactors);
help(estimateJunctionSeqDispersions);
help(fitJunctionSeqDispersionFunction);
help(testForDiffUsage);
help(estimateEffectSizes);
```

5.3 Extracting test results

Once the differential splice junction usage analysis has been run, the results, including model fits, fold changes, p-values, and coverage estimates can all be written to file using the command:

```
writeCompleteResults(jscs,
                     outfile.prefix="./test",
                     save.jscs = TRUE
                     );
```

This produces a series of output files. The main results files are allGenes.results.txt.gz and sigGenes.results.txt.gz. The former includes rows for all genes, whereas the latter includes only genes that have one or more statistically significant differentially used splice junction locus. The columns for both are:

- *featureID*: The unique ID of the splice locus.
- *geneID*: The unique ID of the gene.
- *countbinID*: The sub-ID of the splice locus.
- *testable*: Whether the locus has sufficient coverage to test.
- *dispBeforeSharing*: The locus-specific dispersion estimate.
- *dispFitted*: The the fitted dispersion.
- *dispersion*: The final dispersion used for the hypothesis tests.
- *pvalue*: The raw p-value of the test for differential usage.

- *padjust*: The adjusted p-value, adjusted using the "FDR" method.
- *chr, start, end, strand*: The (1-based) positon of the exonic region or splice junction.
- *transcripts*: The list of known transcripts that contain this splice junction.
- *featureType*: The type of the feature (exonic region, novel splice junction or known splice junction)
- *baseMean*: The base mean normalized coverage counts for the locus.
- *HtestCoef(A/B)*: The interaction coefficient from the alternate hypothesis model fit used in the hypothesis tests. This is generally not used for anything but may be useful for certain testing diagnostics.
- *log2FC(A/B)*: The estimated log2 fold change found using the effect-size model fit. This is calculated using a different model than the model used for the hypothesis tests.

Note: If the biological condition has more than 2 categories then there will be multiple columns for the HtestCoef and log2FC. Each group will be compared with the reference group. If the supplied condition variable is supplied as a `factor` then the first "level" will be used as the reference group.

The writeCompleteResults function also writes a file: allGenes.expression.data.txt.gz. This file contains the raw counts, normalized counts, expression-level estimates by condition value (as normalized read counts), and relative expression estimates by condition value. These are the same values that are plotted in Section 6.2.

Finally, this function also produces splice junction track files suitable for use with the UCSC genome browser or the IGV genome browser. These files are described in detail in Section 6.3.3. If the save.jscs parameter is set to TRUE, then it will also save a binary representation of the JunctionSeqCountSet object.

6 Visualization and Interpretation

The interpretation of the results is almost as important as the actual analysis itself. Differential splice junction usage is an observed phenomenon, not an actual defined biological process. It can be caused by a number of underlying regulatory processes including alternative start sites, alternative end sites, transcript truncation, mRNA destabilization, alternative splicing, or mRNA editing. Depending on the quality of the transcript annotation, apparent differential splice junction usage can even be caused by simple gene-level differential expression, if (for example) a known gene and an unannotated gene overlap with one another but are independently regulated.

Many of these processes can be difficult to discern and differentiate. Therefore, it is imperative that the results generated by JunctionSeq be made as easy to interpret as possible.

JunctionSeq, especially when used in conjunction with the QoRTs software package, contains a number of tools for visualizing the results and the patterns of expression observed in the dataset.

A comprehensive battery of plots, along with a set of html files for easy navigation, can be generated with the command:

```
buildAllPlots(jscs=jscs,
              outfile.prefix = "./plots/",
              use.plotting.device = "png",
              FDR.threshold = 0.01
              );
```

This produces a number of sub-directories, and includes analysis-wide summary plots along with plots for every gene with one or more statistically significant exons or splice junctions. Alternatively, plots for a manually-specified gene list can be generated using the `gene.list` parameter. Other parameters include:

- `colorRed.FDR.threshold`: The adjusted-p-value threshold to use to determine when features should be colored as significant.
- `plot.gene.level.expression`: If `FALSE`, do not include the gene-level expression plot on the right side.
- `sequencing.type`: "paired-end" or "single-end". This option is purely cosmetic, and simply determines whether the y-axes will be labelled as "read-pairs" or "reads".
- `par.cex, anno.cex.text, anno.cex.axis, anno.cex.main`: The character expansions for various purposes. `par.cex` determines the cex value passed to `par`, which expands all other text, margins, etc. The other three parameters determine the size of text for the various labels.
- `plot.lwd, axes.lwd, anno.lwd, gene.lwd`: Line width variables for various lines.
- `base.plot.height, base.plot.width, base.plot.units`: The "base" width and height for all plots. Plots with transcripts drawn will be expanded vertically to fit the transcripts, plots with a large number of exons or junctions will be likewise be expanded horizontally. This functionality can be deactivated by setting `autoscale.height.to.fit.TX.annotation` to FALSE and `autoscale.width.to.fit.bins` to `Inf`. `html.compare.results.list`: If you have multiple studies that you want to directly compare, you can use this parameter to allow cross-linking between analyses from the html navigation pages. The results from each analysis must be placed in the same parent directory, and this parameter must be set to a named list of subdirectory names, corresponding to the `outfile.prefix` for each run of

`buildAllPlots`. In general you should also set the `gene.list` parameter, or else cross-links will fail if a gene is significant in one analysis but not the other. This parameter can also be used to cross-link different runs of `buildAllPlots` on the same dataset, to easily compare different parameter settings.

6.1 Summary Plots

JunctionSeq provides functions for two basic summary plots, used to display experiment-wide results. The first is the dispersion plot, which displays the dispersion estimates (y-axis) as a function of the base mean normalized counts (x-axis). The test-specific dispersions are displayed as blue density shading, and the "fitted" dispersion is displayed as a red line. JunctionSeq can also produce an "MA" plot, which displays the fold change("M") on the y-axis as a function of the overall mean normalized counts ("A") on the x-axis.

These plots can be generated with the command:

```
plotDispEsts(jscs);
plotMA(jscs, FDR.threshold=0.05);
```

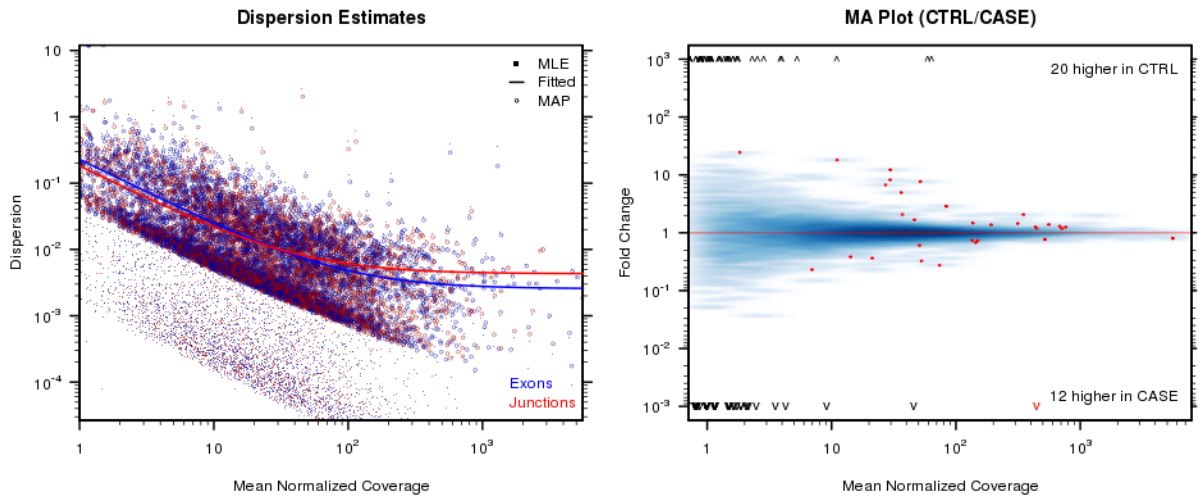

Figure 1: Summary Plots

6.2 Gene plots

Generally it is desired to print plots for all genes that have one or more splice junctions with statistically significant differential splice junction usage. By default, JunctionSeq uses an FDR-adjusted p-value threshold of 0.01.

For each selected gene, the `buildAllPlots` function will yield 6 plots:

- *geneID-expr.png*: Estimates of average coverage count over each feature, for each value of the biological condition. See Section 6.2.1.
- *geneID-expr-withTx.png*: as above, but with the transcript annotation displayed.
- *geneID-normCounts.png*: Normalized read counts for each sample, colored by biological condition. See Section 6.2.2.
- *geneID-normCounts-withTx.png*: as above, but with the transcript annotation displayed.
- *geneID-rExpr.png*: Expression levels relative to the overall gene-level expression. See Section 6.2.3.
- *geneID-rExpr-withTx.png*: as above, but with the transcript annotation displayed.

JunctionSeq will also generate an html index file for easy browsing of these output files.

6.2.1 Coverage/Expression Plots

Figure 2 displays the model estimates of the mean normalized exonic-region/splice-junction coverage counts for each biological condition (in this example: DAY vs NIGHT). Note that these values are not equal to the simple mean normalized read counts across all samples. Rather, these estimates are derived from the GLM parameter estimates (via linear contrasts).

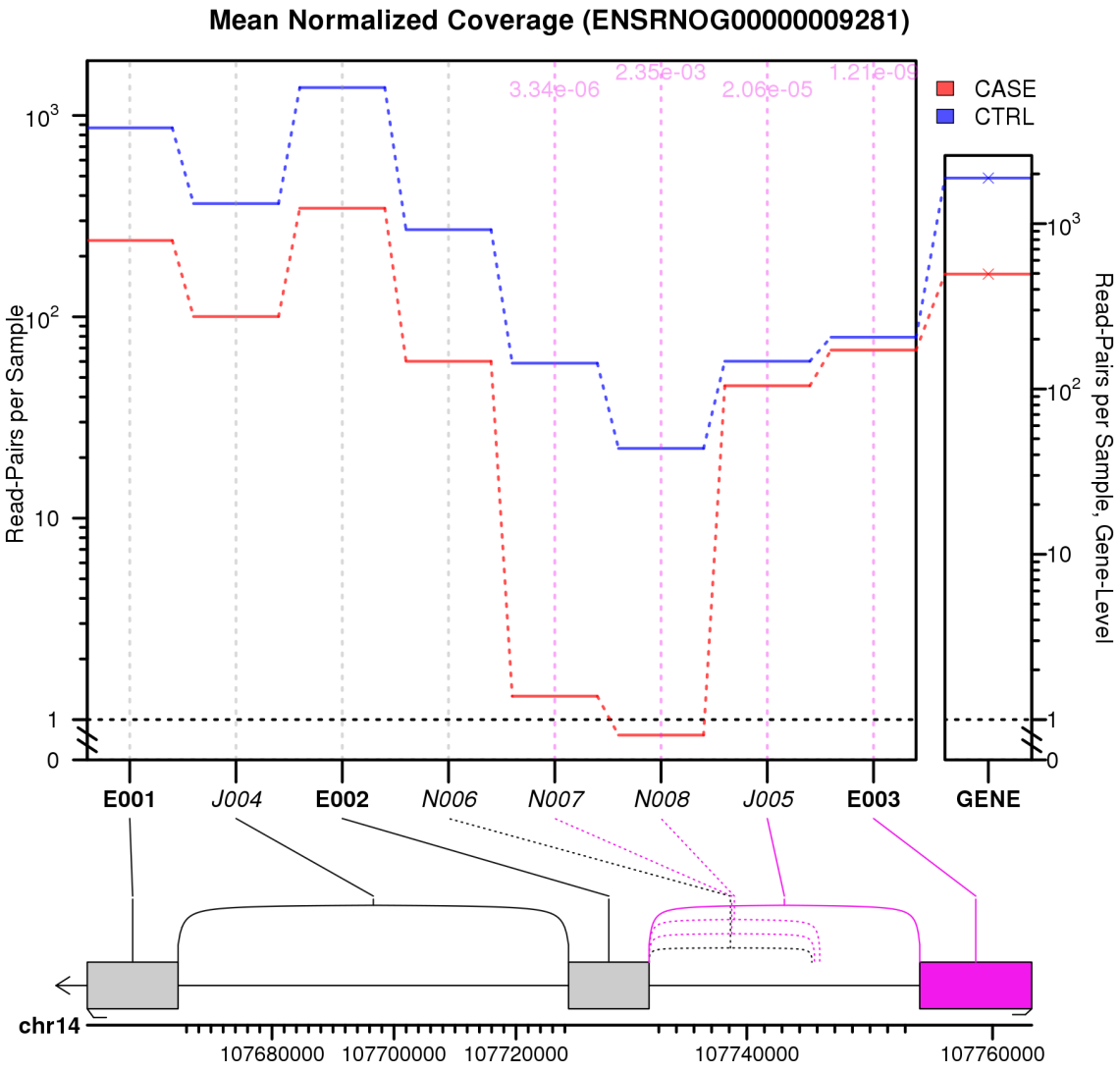

Figure 2: Feature coverage by biological condition. The plot includes 3 major frames. The top-left frame graphs the expression levels for each tested splice junction (aka the model estimates for the normalized read-pair counts). On the top-right is a box containing the estimates for the overall expression of the gene as a whole (plotted on a distinct y-axis scale). Junctions or exonic regions that are statisically significant are marked with vertical pink lines, and the significant p-values are displayed along the top of the plot. The bottom frame is a line drawing of the known exons for the given gene, as well as all splice junction. Known junctions are drawn with solid lines, novel junctions are dashed, and "untestable" junctions are drawn in grey. Additionally, statistically significant loci are colored pink. Between the two frames a set of lines connect the gene drawing to the expression plots.

There are a few things to note about this plot:

- The y-axis is log-transformed, except for the area between 0 and 1 which is plotted on a simple linear scale.
- In the line drawing at the bottom, the exons and introns are not drawn to a common scale. The exons are enlarged to improve readability. Each exon or intron is rescaled proportionately to the square-root of their width, then the exons are scaled up to take up 30 percent of the horizontal space. This can be adjusted via the "exon.rescale.factor" parameter (the default is 0.3), or turned off entirely by setting this parameter to -1. The proportionality function can also be changed using the `exonRescaleFunction` parameter. For mammalian genomes at least, we have found square-root-proportional function to be a good trade-off between making sure that features are distinct while still being able to visually identify which features are larger or smaller.
- Note that junction J007 is marked as significantly differentially used, even though it is NOT differentially expressed. This is because JunctionSeq does not test for simple differential expression. It tests for differential splice junction coverage relative to gene-wide expression. Therefore, if (as in this case) the gene as a whole is strongly differentially expressed, then a splice junction that is NOT differentially expressed is the one that is being differentially used.
- Similarly, splice junction N009 is differentially used in the opposite way: while the gene itself is somewhat differentially expressed (at a fold change of roughly 3-4x overall), this particular junction has a *massive* differential, far beyond that found in the gene as a whole (at 45x fold change). Thus, it is being differentially used.

Figure 3 displays the same information found in Figure 2, except with all known transcripts plotted beneath the main plot.

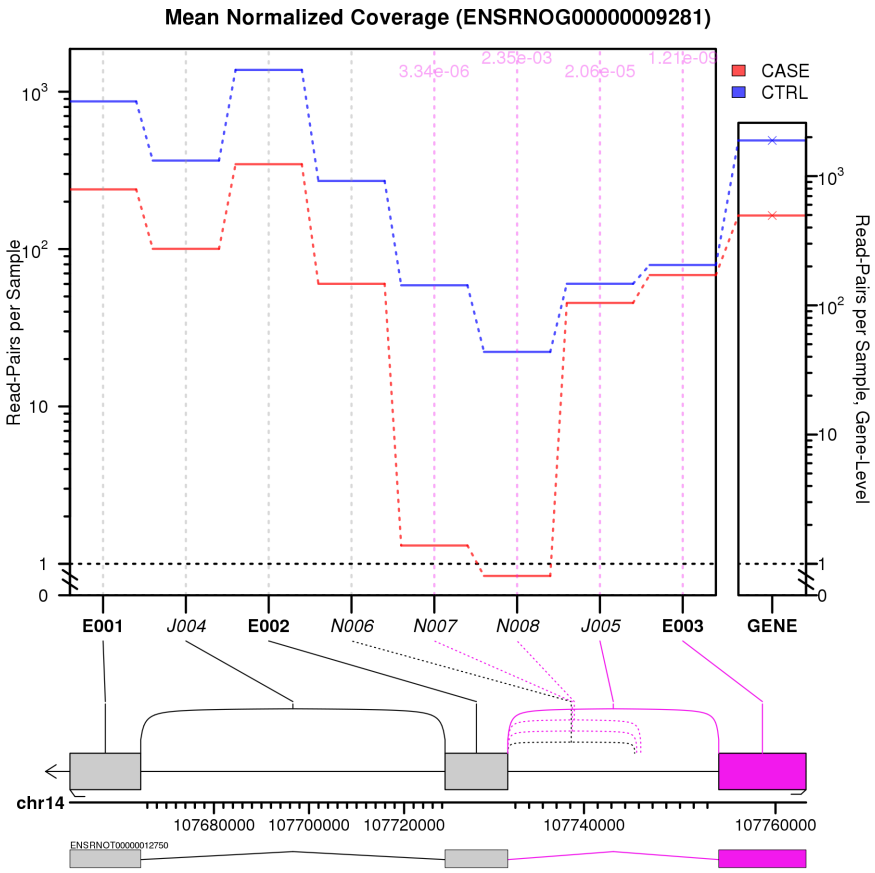

Figure 3: Feature coverage by biological condition, with annotated transcripts displayed. This plot is identical to the previous, except all annotated transcripts are displayed below the standard plot.

6.2.2 Normalized Count Plots

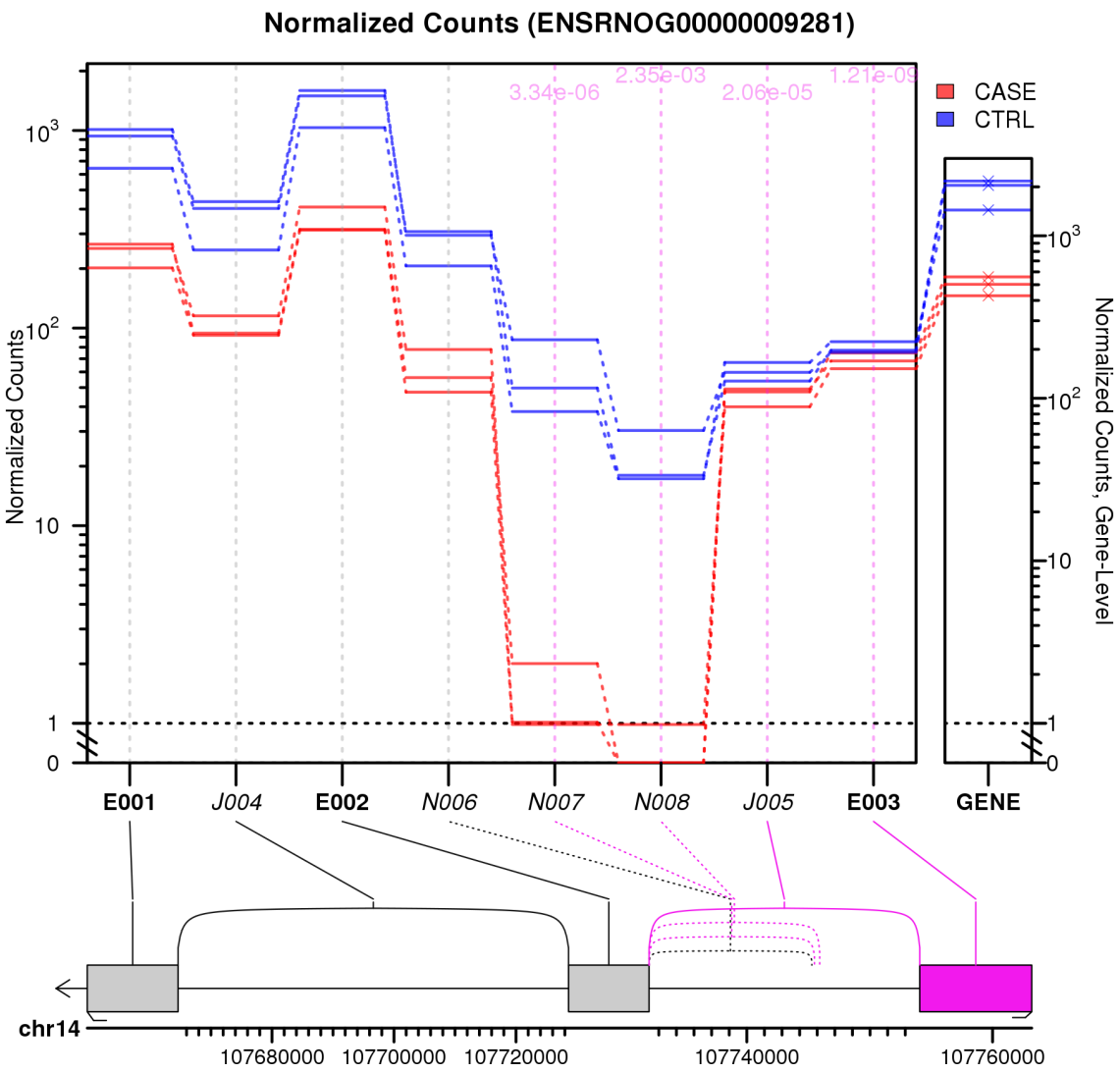

Figure 4: Normalized counts for each sample

Figure 4 displays the normalized coverage counts for each sample over each splice junction or exonic region. This will be identical to the counts displayed in Section 6.4.1 except that each read count will be normalized by the sample size factors so that the samples can be compared directly.

6.2.3 Relative Expression Plots

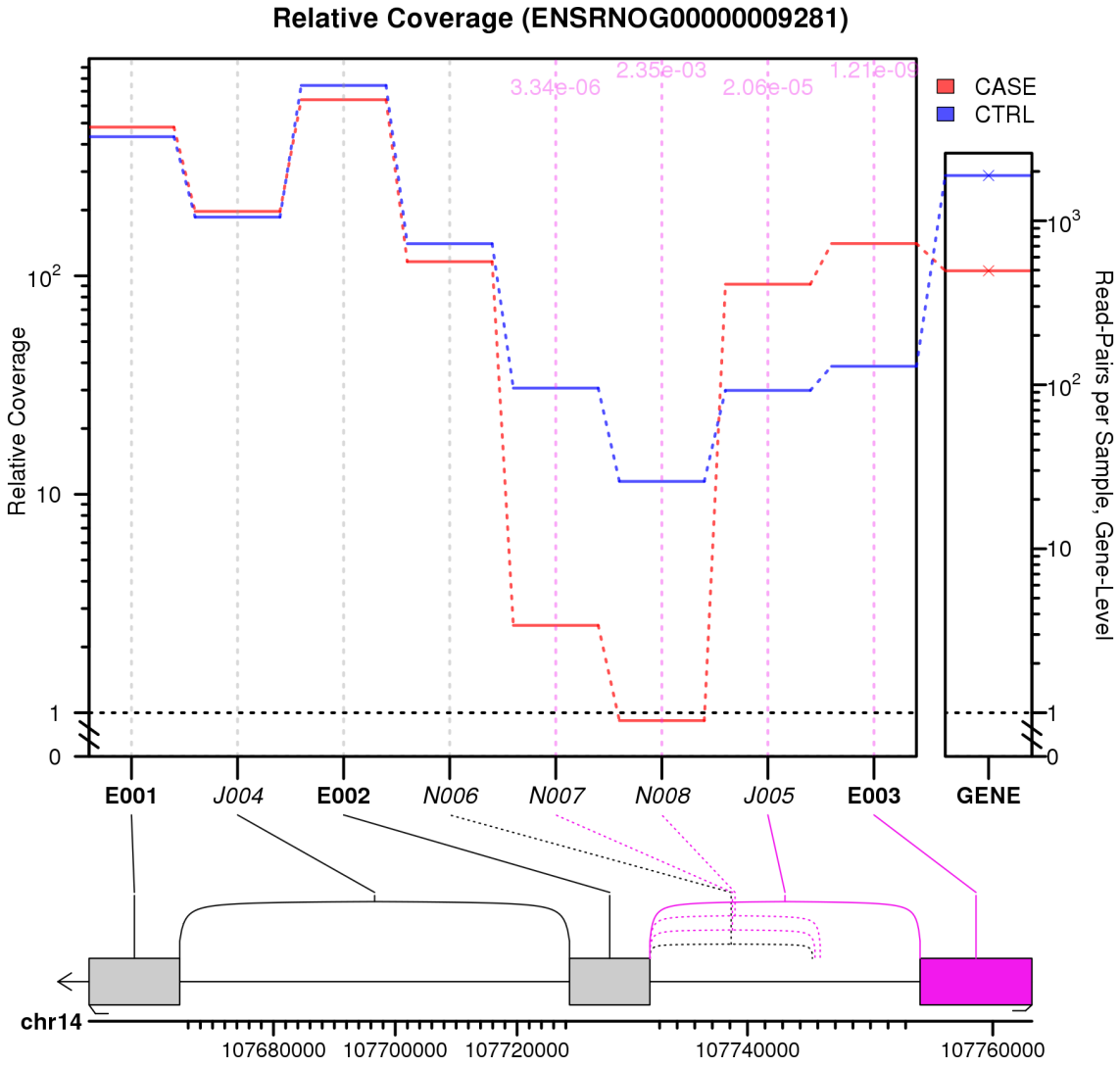

Figure 5: Relative splice junction coverage by biological condition.

Figure 5 displays the "relative" coverage for each splice junction or exonic region, *relative* to gene-wide expression (which is itself plotted in the right panel).

JunctionSeq is designed to detect differential expression even in the presence of gene-wide, multi-transcript differential expression. However it can be difficult to visually assess differential usage on differentially exprssed genes. This plot displays the coverage relative to the gene-wide expression. These estimates are derived from the GLM parameter estimates (via linear contrasts). The reported "fold changes" are ratios of these relative expression estimates.

6.3 Generating Genome Browser Tracks

Once potential genes of interest have been identified via JunctionSeq, it can be helpful to examine these genes manually on a genome browser (such as the UCSC genome browser or the IGV browser). This can assist in the identification of potential sources of artifacts or errors (such as repetitive regions or the presence of unannotated overlapping features) that may underlie false discoveries. To this end, the QoRTs and JunctionSeq software packages include a number of tools designed to assist in generating simple and powerful browser tracks designed to aid in the interpretation of the data and results.

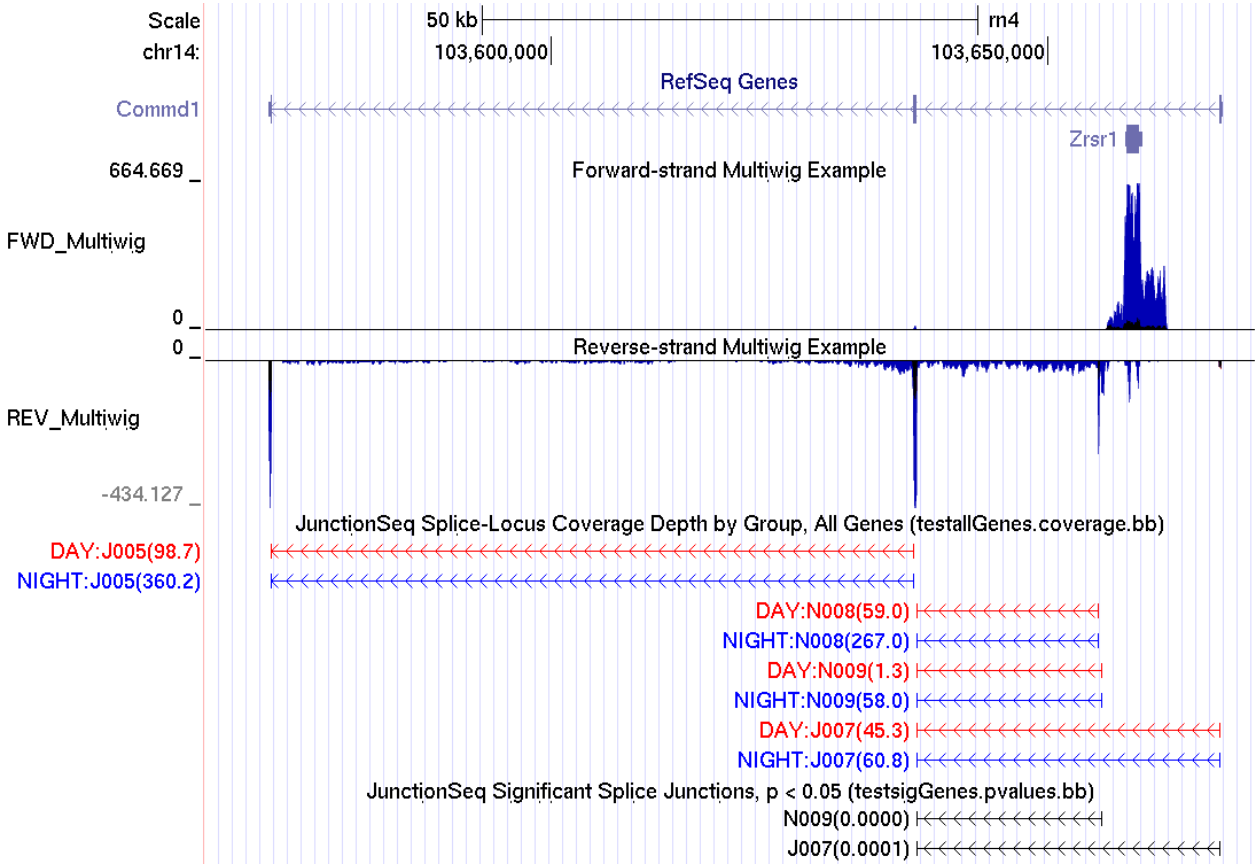

Figure 6: An example of the browser tracks that can be generated using QoRTs and JunctionSeq. The top two "MultiWig" tracks display the mean normalized coverage for 100-base-pair windows across the genome, by biological group (DAY or NIGHT). The top track displays the coverage across the forward (genomic) strand, and the second track displays the coverage across the reverse strand. In both tracks the mean normalized coverage depth across the three NIGHT samples is displayed in blue and the mean normalized coverage depth across the three DAY samples is displayed in red. The overlap is colored black. The third track displays the mean normalized coverage across all testable splice junction loci for each biological condition. Each junction is labelled with the condition ID (DAY or NIGHT), the splice junction ID (J for annotated, N for novel), followed by the mean normalized coverage across that junction and biological condition, in parentheses. Once again, DAY samples are displayed in red and NIGHT samples are displayed in blue. The final bottom track displays the splice junctions that exhibit statistically significant differential usage. Each junction is labelled with the splice junction ID and the p-value. These images were produced by the UCSC genome browser, and a browser session containing these tracks in this configuration is available online here

Advanced tracks like those displayed in Figure 6 can be used to visualize the data and can aid in determining the form of regulatory activity that underlies any apparent differential splice junction usage.

The wiggle files needed to produce such tracks can be generated via QoRTs and JunctionSeq. Configuring the multi-colored "MultiWig" tracks in the UCSC browser require the use of track hubs, the configuration of which is beyond the scope of this manual. More information on track hubs can be found on the UCSC browser documentation.

6.3.1 Wiggle Tracks

Both IGV and the UCSC browser can display "wiggle" tracks, which can be used to display coverage depth across the genome. QoRTs includes functions for generating these wiggle files for each sample/replicate, merging across technical replicates, and computing mean normalized coverages across multiple samples for each biological condition.

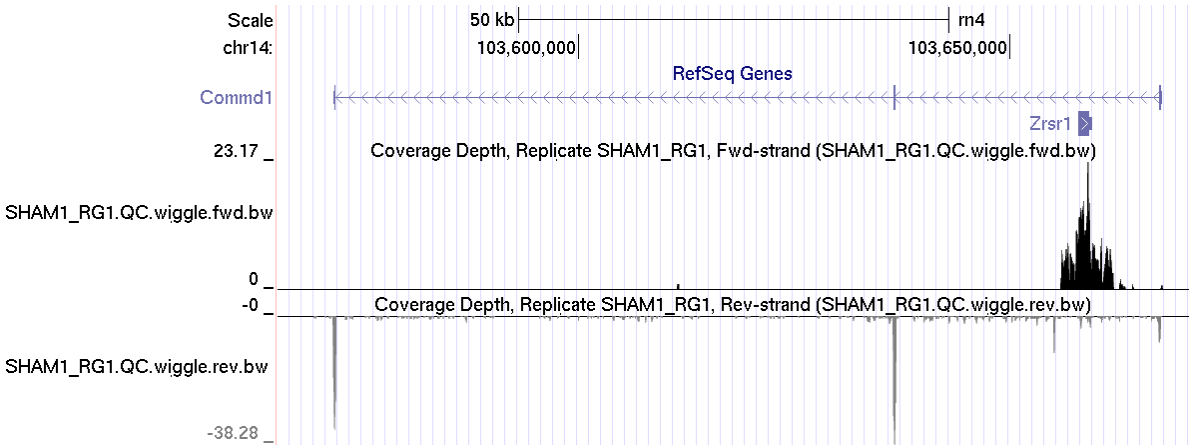

Figure 7: Two "wiggle" tracks displaying the forward- and reverse-strand coverage for replicate SHAM1_RG1. These tracks display the read-pair mean coverage depth for each 100-base-pair window across the whole genome. The reverse strand is displayed as negative values. These tracks have been loaded into the UCSC genome browser, and a browser session containing these tracks in this configuration is available online here

Figure 7 shows an example pair of "wiggle" files produced by QoRTs for replicate SHAM1_RG1. QoRTs includes two ways to generate such wiggle files from a sam/bam file:

The first way is to create these files at the same time as the read counts. To do this, simply add the "–chromSizes" parameter like so:

```
java -jar /path/to/jarfile/QoRTs.jar QC \
                    --stranded \
                    --chromSizes inputData/annoFiles/chrom.sizes \
                    inputData/bamFiles/SHAM1_RG1.chr14.bam \
                    inputData/annoFiles/rn4.anno.chr14.gtf.gz \
                    outputData/qortsData/SHAM1_RG1/
```

By default this will cause QoRTs to generate the wiggle file(s) for this sample. Note that if the –runFunctions parameter is being included, you must also include in the function list the function

"makeWiggles". Note that if the data is stranded (as in the example dataset), then two wiggle files will be generated, one for each strand.

Alternatively, the wiggle file(s) can be generated manually using the command:

```
java -jar /path/to/jarfile/QoRTs.jar QC \
                bamToWiggle \
                --stranded \
                --negativeReverseStrand \
                --includeTrackDefLine \
                inputData/bamFiles/SHAM1_RG1.chr14.bam \
                SHAM1_RG1 \
                inputData/annoFiles/rn4.chr14.chrom.sizes \
                outputData/qortsData/SHAM1_RG1/QC.wiggle
```

If this step is performed prior to merging technical replicates (see Section 4.2), and if the standard file-name conventions are followed (as displayed in the examples above), then the technical-replicate wiggle files will automatically be merged along with the other count information in by the QoRTs technical replicate merge utility.

6.3.2 Merging Wiggle Tracks

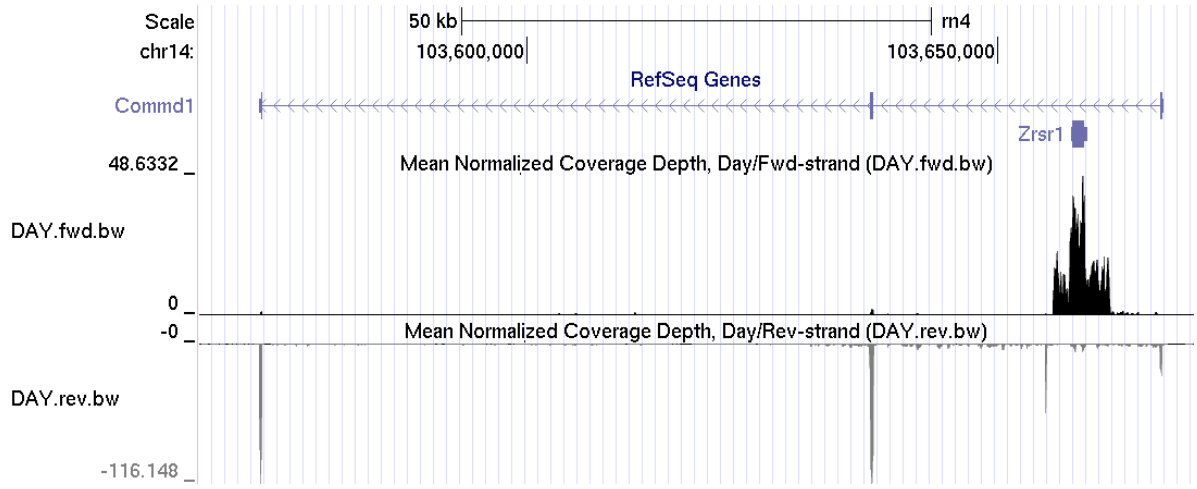

Figure 8: Two "wiggle" tracks displaying the forward- and reverse-strand coverage for the three "DAY" samples. These tracks display the mean normalized read-pair coverage depth for each 100-base-pair window across the whole genome. The reverse strand is displayed as negative values. These tracks have been loaded into the UCSC genome browser, and a browser session containing these tracks in this configuration is available online here.

QoRTs can generate wiggle files containing mean normalized coverage counts across a group of samples, as shown in Figure 8. These can be generated using the command:

```
java -jar /path/to/jarfile/QoRTs.jar QC \
                mergeWig  \
                --calcMean \
                --trackTitle DAY_FWD \
                --infilePrefix outputData/countTables/ \
                --infileSuffix /QC.wiggle.fwd.wig.gz \
                --sizeFactorFile sizeFactors.GEO.txt \
                --sampleList SHAM1,SHAM2,SHAM3 \
                outputData/DAY.fwd.wig.gz
```

The "–sampleList" parameter can also accept data from standard input ("-"), or a text file (which must end ".txt") containing a list of sample ID's, one on each line.

6.3.3 Splice Junction Tracks

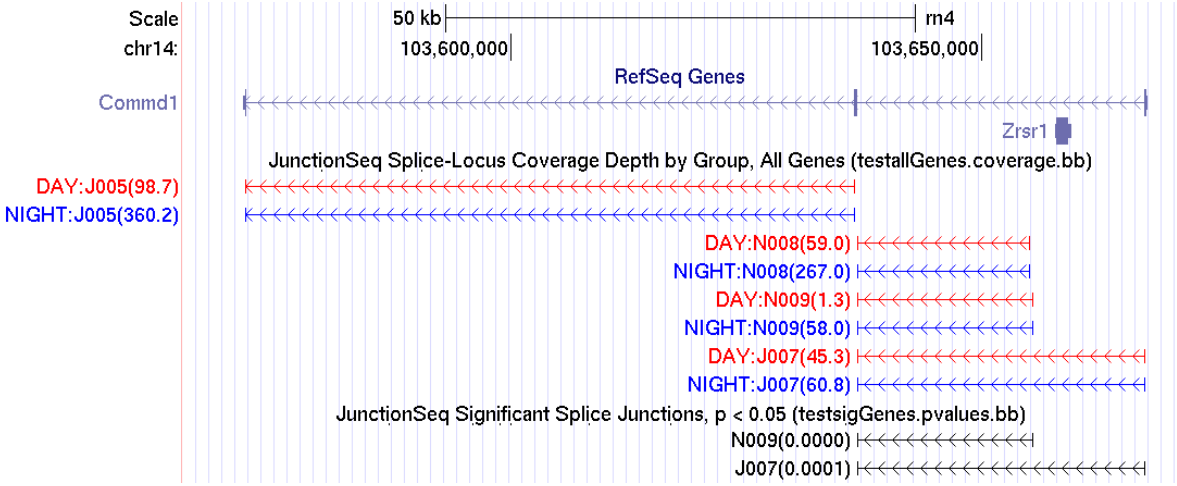

Figure 9: The first track displays the mean-normalized coverage for each splice junction for DAY (blue) and NIGHT (red) conditions. Each splice junction is marked with the condition ID (DAY or NIGHT), followed by the splice junction ID (J for annotated, N for novel), followed by the mean normalized read count in parentheses. The second track displays the splice junctions that display statistically-significant differential usage, along with the junction ID and the adjusted p-value in parentheses (rounded to 4 digits). This track has been loaded into the UCSC genome browser, and a browser session containing these tracks in this configuration is available online here.

Splice Junction Tracks are generated automatically by the `writeCompleteResults` function, assuming the `write.bedTracks` parameter has not been set to FALSE. By default, five bed tracks are generated:

- *allGenes.junctionCoverage.bed.gz*: A track that lists the mean normalized read (or read-pair) coverage for each splice junction and for each biological condition. These are not equal to the actual mean normalized counts, rather these are derived from the parameter estimates.
- *sigGenes.junctionCoverage.bed.gz*: Identical to the previous, but only for genes with statistically significant features.
- *allGenes.exonCoverage.bed.gz*: Similar to the Junction Coverage plots, but displaying coverage across exonic regions.
- *sigGenes.exonCoverage.bed.gz*: Identical to the previous, but only for genes with statistically significant features.
- *sigGenes.pvalues.bed.gz*: This track displays all statistically-significant loci (both exonic regions and splice junctions), with the adjusted p-value listed in parentheses.

These five files can be generated via the `writeCompleteResults` command (see Section 5.3)

Individual-sample or individual-replicate junction coverage tracks can also be generated via QoRTs. See the QoRTs documentation available online.

6.4 Additional Plotting Options

6.4.1 Raw Count Plots

Raw expression plots can be added using the command:

```
buildAllPlots(jscs=jscs,
              outfile.prefix = "./plots/",
              use.plotting.device = "png",
              FDR.threshold = 0.01,
              expr.plot = FALSE, normCounts.plot = FALSE,
              rExpr.plot = FALSE, rawCounts.plot = TRUE
              );
```

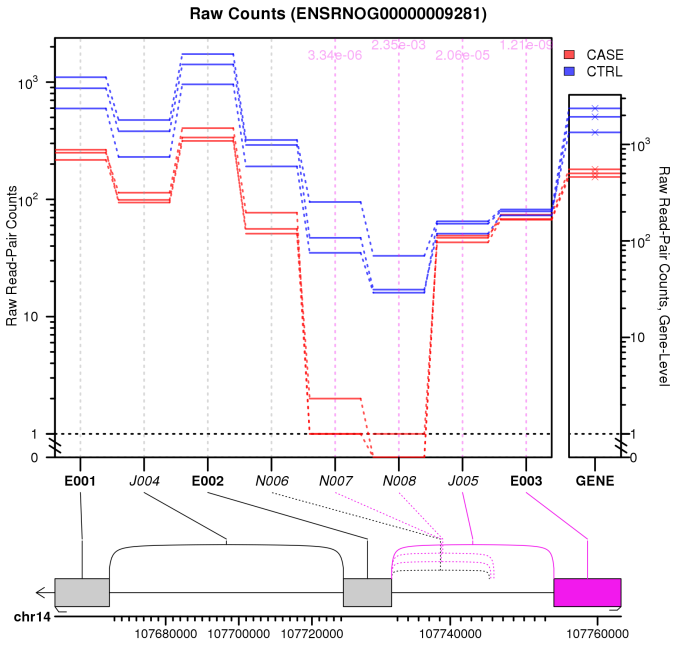

Figure 10: Raw counts for each sample

Figure 10 displays the raw (un-normalized) coverage counts for each sample over each splice junction or exonic region. This is equal to the number of reads (or read-pairs, for paired-end data) that bridge each junction. Note that these counts are not normalized, and are generally not directly comparable. Note that by default this plot is NOT generated by `buildAllPlots`. Generation of these plots must be turned on by adding the parameter: "`rawCounts.plot=TRUE`".

6.4.2 Additional Optional Parameters

By setting the `plot.exon.results` and `plot.junction.results` parameters to `FALSE`, we can exclude exons or junctions from the plots, respectively. There are numerous other options that can be used to generate plots either individually or in groups.

Just a few examples:

```
#Make a battery of exon-only plots for one gene only:
buildAllPlotsForGene(jscs=jscs, geneID = "ENSRNOG00000009281",
              outfile.prefix = "./plots/",
              use.plotting.device = "png",
              colorRed.FDR.threshold = 0.01,
             #Limit plotting to exons only:
              plot.junction.results = FALSE,
             #Change the fill color of significant exons:
              colorList = list(SIG.FEATURE.FILL.COLOR = "red"),
              );

#Make a set of Junction-Only Plots for a specific list of interesting genes:
buildAllPlots(jscs=jscs,
              gene.list = c("ENSRNOG00000009281"),
              outfile.prefix = "./plotsJCT/",
              use.plotting.device = "png",
              FDR.threshold = 0.01,
             #Do not graph exonic regions:
              plot.exon.results = FALSE,
              );
```

See Figure 11 and Figure 12.

For more information, use the help documentation:

```
help(buildAllPlots);
help(buildAllPlotsForGene);
help(plotJunctionSeqResultsForGene);
```

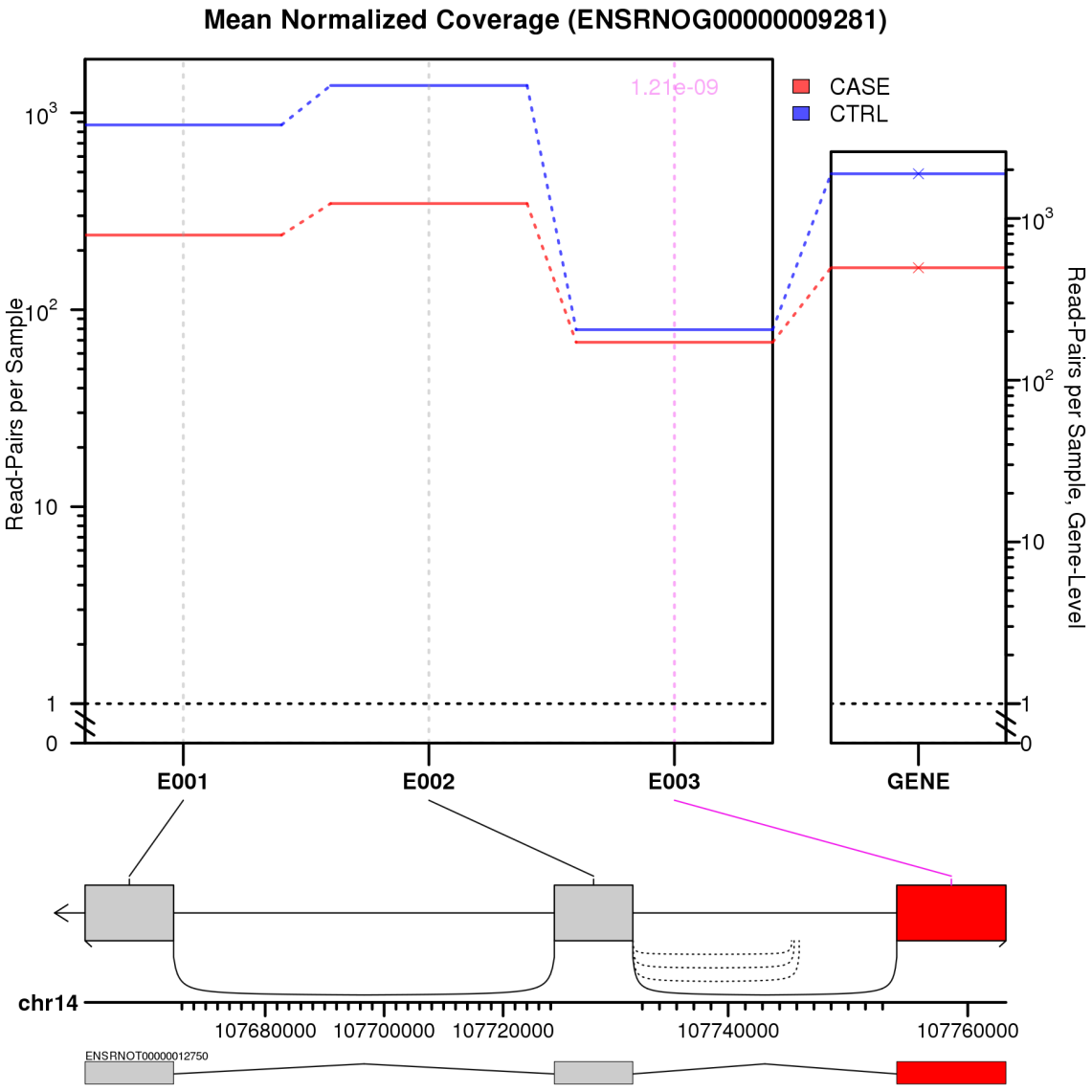

Figure 11: Exon expression plots. Note the altered fill color of the significant exon.

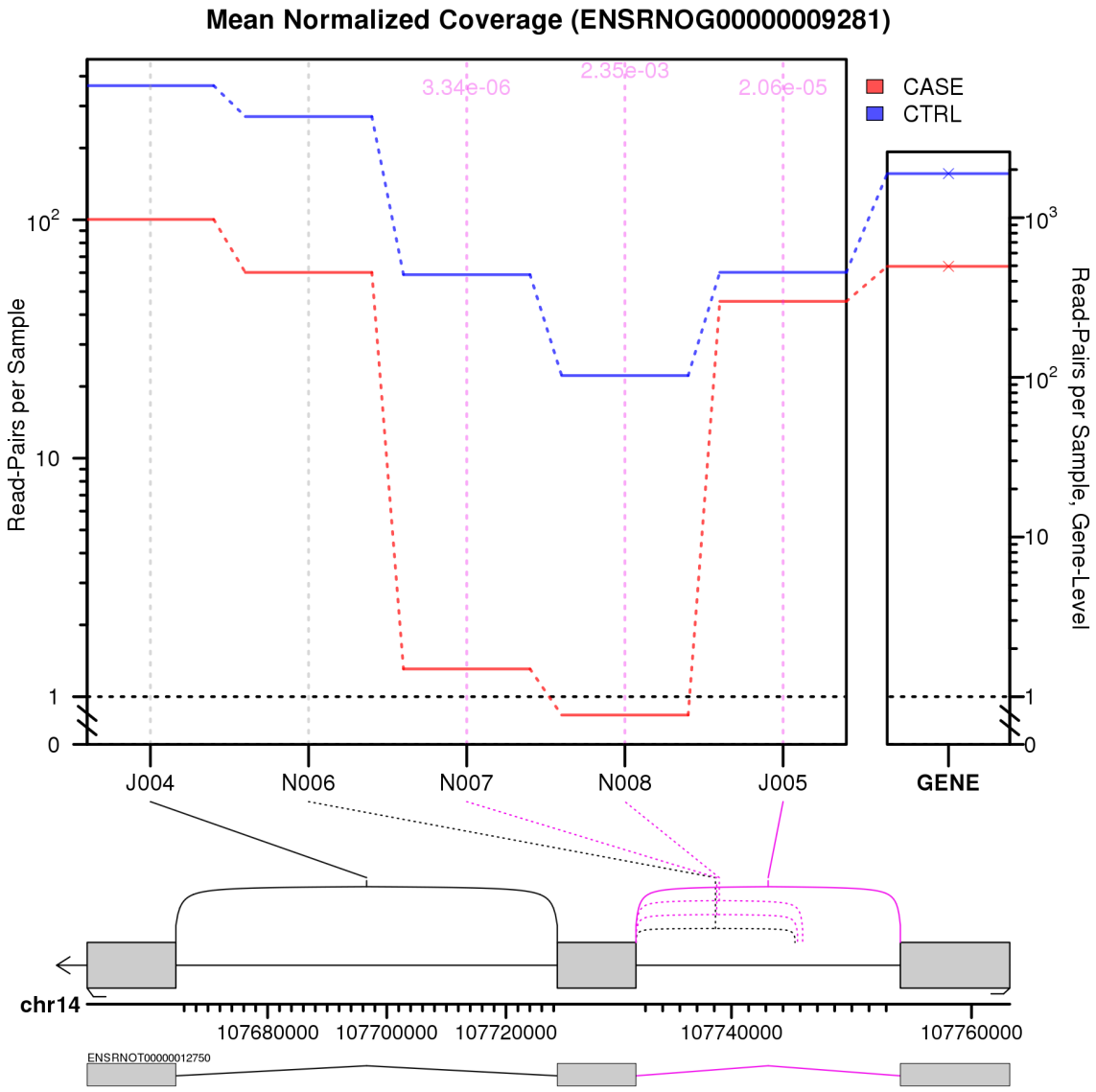

Figure 12: Splice Junction expression plots.

7 Statistical Methodology

The statistical methods are based on those described in (S. Anders et al., 2012), as implemented in the DEXSeq Bioconductor package. However these methods have been expanded, adapted, and altered in a number of ways in order to accurately and efficiently test for differential junction usage.

7.1 Preliminary Definitions

The methods used to generate exonic segment counting bins is idential to the methods used in DEXSeq. Briefly: for each gene the set of all transcripts are combined and exons are broken up into mutually-exclusive sub-segments.

Using the transcript annotation we generate a list of "features" for each gene. These features include all the mutually-exclusive exonic segments, along with all the splice junctions (both annotated splice junctions and novel splice junctions that pass the coverage thresholds).

For each sample i and feature j on gene g we define the counts:

$$k_{ji}^{\text{Feature}} = \#\text{ reads/pairs covering feature } j \text{ in sample } i \tag{1a}$$

and

$$k_{gi}^{\text{Gene}} = \#\text{ reads/pairs covering gene } g \text{ in sample } i \tag{1b}$$

Reads are counted towards an exonic segment feature if they overlap with any portion of that exonic segment. Reads are counted towards a splice junction feature if they map across the splice junction. Paired-end reads in which the two mates lie on opposite sides of a junction are NOT counted, as the specific splice junction between them cannot be reliably identified.

The count k_{gi}^{Gene} is calculated using the same methods already in general use for gene-level differential expression analysis (using the "union" rule). Briefly: any reads or read-pairs that cover any part of any of the exons of any one unique gene are counted towards that gene. Purely-intronic reads are ignored.

It should be noted that the QoRTs script that produces the exonic segments differs slightly in its output from the DEXSeq script `prepare_annotation.py`. This is because the DEXSeq script uses unordered data structures and thus the order in which elements appear is not defined. Additionally, for unstranded data QoRTs version operates in a slightly-different "unstranded" mode, in which genes that overlap across opposite strands are counted as overlapping.

7.2 Model Framework

Each feature is fitted to a separate statistical model. For a given feature j located on gene g, we define two counting bins: covering the feature j, and covering the gene g, but NOT covering the feature j. Thus we define y_{1i} and y_{0i} for each sample $i \in \{1, 2, \ldots, n\}$:

$$y_{1i} = k_{ji}^{\text{Feature}} \tag{2a}$$

and

$$y_{0i} = k_{gi}^{\text{Gene}} - k_{ji}^{\text{Feature}} \tag{2b}$$

Thus, y_{1i} is simply equal to the number of reads covering the feature in sample i, and y_{0i} is equal to the number of reads covering the gene but NOT covering the feature.

Note that while JunctionSeq generally uses methods similar to those used by DEXSeq, this framework differs from that used by DEXSeq on exon counting bins.

In the framework used by DEXSeq, the feature count (i.e. k_{ji}^{Feature}) is compared with the sum of all other feature counts for the given gene. This means that some reads may be counted more than once if they span multiple features. When reads are relatively short (as was typical when DEXSeq was designed) this effect is minimal, but it becomes more problematic as reads become longer. This problem is also exacerbated in genes with a large number of features in close succession. Under our framework, no read-pair is ever counted more than once in a given model.

As in DEXSeq, we assume that the count y_{bi} is a realization of a negative-binomial random variable Y_{bi}:

$$Y_{bi} \sim NegBin(\text{mean} = s_i \mu_{bi}, \text{dispersion} = \alpha_j) \tag{3}$$

Where α_j is the dispersion parameter for the current splice junction j, s_i is the normalization size factor for each sample i, and μ_{bi} is the mean for sample i and counting-bin b. Size factors s_i are estimated using the "geometric" normalization method, which is the default method used by DESeq, DESeq2, DEXSeq, and CuffDiff. Unlike with DEXSeq, these size factors are calculated using the gene-level counts k_{gi}^{Gene}, and thus does not implicitly assign excess weight to genes with a large number of features.

7.3 Dispersion Estimation

In many high-throughput sequencing experiments there are too few replicates to directly estimate the locus-specific dispersion term α_j for each splice junction j. This problem is well-characterized, and a number of different solutions have been proposed, the vast majority of which involve sharing information between loci across the genome. By default JunctionSeq uses the exact same method for dispersion estimation used by the more recent versions of the DEXSeq package (v1.14.0 and above) and by the DESeq2 package. This method is described at length in the DESeq2 methods paper [2].

Briefly: "feature-specific" estimates of dispersion are generated via a Cox-Reid adjusted-log-likelihood-based method, and these dispersions are fitted to a parametric trend. These "fitted" and "feature-specific" estimates of dispersion are combined, and we use the maximum *a posteriori* estimate as the final dispersion for each feature.

It is currently unclear whether splice junction loci and exonic segment loci can be reasonably expected to follow the same trend for dispersion. By default JunctionSeq fits the two types of features separately. For most datasets the difference appears to be negligible.

JunctionSeq offers a number of alternatives to this methodology, and can calculate dispersions using the methods of older versios of DEXSeq (v1.8.0 and earlier). See Section 8 for more information.

7.4 Hypothesis Testing

Other than our use of overall-gene counts and our use of splice junction counts, the methods used in the hypothesis test are identical to those used by the more recent versions of DEXSeq (v1.12.0 and higher).

"Differential usage" is not in and of itself a biological process. Rather, it is a **symptom** of transcript-specific differential regulation, which can take many forms.

Put simply: we are attempting to test whether the fold-change for the biological condition across feature j is the same as the fold-change for the biological condition across the gene g as a whole.

This can occur both with and without overall gene-level differential expression. Some features marked as "differentially used" might show a strong differential and be part of a gene that is not, as a whole, differentially expressed. Alternatively, a feature might be marked as "differentially used" if it displays flat expression but is part of a gene that otherwise shows extreme differentials. In both cases, the feature does not show the same pattern of expression relative to the gene as a whole. Thus, some form of transcript-specific differential regulation must be taking place.

In statistical terms: we are attempting to detect "interaction" between the counting-bin variable B and the experimental-condition variable C. Thus, two models are fitted to the mean μ_{bi}:

$$H_0 : \quad \log(\mu_{bi}) = \beta + \beta_b^B + \beta_i^S \tag{4a}$$

$$H_1 : \quad \log(\mu_{bi}) = \beta + \beta_b^B + \beta_i^S + \beta_{\rho_i b}^{CB} \tag{4b}$$

Where ρ_j is the biological condition (eg case/control status) of sample j.

Note that the bin-condition interaction term ($\beta_{\rho_i b}^{CB}$) is included, but the condition main-effect term ($\beta_{\rho i}^{C}$) is absent. This term can be omitted is because JunctionSeq is not designed to detect or assess gene-level differential expression. Thus there are two components that can be treated as "noise": variation in junction-level expression and variation in gene-level expression. As proposed by Anders et. al., we use a main-effects term for the sample ID (β_i^S), which subsumes the condition main-effect term. This subsumes both the differential trends and the random variations (noise) in the gene-level expression, improving the power towards detecting differential interaction between the count-bin term and the experimental-condition term.

These models can easily be extended to include confounding variables: For confounding variable τ, define the value of τ for each sample i as τ_i. Then we can define our null and alternative hypotheses:

$$H_0 : \quad \log(\mu_{bi}) = \beta + \beta_b^B + \beta_i^S + \beta_{\tau_i b}^{TB} \tag{5a}$$

$$H_1 : \quad \log(\mu_{bi}) = \beta + \beta_b^B + \beta_i^S + \beta_{\tau_i b}^{TB} + \beta_{\rho_i b}^{CB} \tag{5b}$$

7.5 Parameter Estimation

While the described statistical model is robust, efficient, and powerful, it cannot be used to effectively estimate the size of the differential effect or to produce informative parameter estimates.

For the purposes of estimating expression levels and the strength of differential effects, we create an entirely separate set of generalized linear models. For the purposes of hypothesis testing this model would be less powerful than the model used in section 7.4, but has the advantage of producing more intuitive and interpretable parameter estimates and fitted values. As before, we generate one set of generalized linear models per splice junction locus:

$$H_E : \quad \log(\mu_{bi}) = \beta + \beta_b^B + \beta_{\rho i}^C + \beta_{\rho_i b}^{CB} \tag{6}$$

These models appear similar to the models used for parameter estimation used by DEXSeq, however they differ considerably in that JunctionSeq models each feature separately, rather than attempting to include all features for a given gene in a single model. This "big model" method was originally used for both hypothesis tests and parameter estimation in DEXSeq, but later versions only use this framework for parameter estimation, as the "big model" was found to be less efficient and less consistent. This "big model" methodology also has a number of other weaknesses, including over-weighting of loci with a large number of alternative exonic regions. Reads covering such regions will be "double counted" several times over as they cover multiple regions. For all intents and purposes, DEXSeq counts such double counts as independent observations. JunctionSeq does not count any read or read-pair more than once in any model (unless alternative methodologies are selected, see Section 8).

Using linear contrasts, the parameter estimates $\hat{\beta}$, $\hat{\beta}^B_b$, $\hat{\beta}^C_{\rho i}$, and $\hat{\beta}^{CB}_{\rho_i b}$ can be used to calculate estimates of the effect size (fold change), as well as the mean normalized coverage over the sfeature for each condition. Additionally, the "relative" expression levels can be calculated for each condition, which indicates the expression of the splice junction, normalized relative to the overall gene-wide expression (which may be differentially expressed).

This model can be extended to include confounding variables in a manner similar to how the hypothesis test models can be extended (as in Equation 5).

$$H_E: \qquad \log(\mu_{bi}) = \beta + \beta^B_b + \beta^C_{\rho i} + \beta^{TB}_{\tau_i b} + \beta^{CB}_{\rho_i b} \tag{7}$$

8 For Advanced Users: Optional Alternative Methodologies

Since development began on JunctionSeq, a number of new methods have been developed for modelling expression in RNA-Seq data. Many of these more-advanced methods have been integrated into JunctionSeq. However, some advanced users with strong statistical backgrounds may prefer the older, more proven methods. Thus, the legacy methods are still optionally available to JunctionSeq users.

In addition, JunctionSeq includes a number of small methodological changes relative to the DEXSeq software upon which it was based. Any or all of these changes can be reverted as well, if desired.

All of these optional alternatives can be selected using the "method" parameters passed to `runJunctionSeqAnalyse`
and its sub-functions (see Section 5.2). These optional parameters are:

Additionally, there are a number of options that alter the fundamental methodology to be used. In general the defaults are sufficient, but advanced users may prefer alternative methods.

- `method.GLM` The default is "advanced" or, equivalently, "DESeq2-style". This uses the dispersion estimation methodology used by DESeq2 and DEXSeq v1.12.0 or higher to generate the initial (feature-specific) dispersion estimates. The alternative method is "simpleML" or, equivalently, "DEXSeq-v1.8.0-style". This uses a simpler maximum-likelihood-based method used by the original DESeq and by DEXSeq v1.8.0 or less.
- `method.dispFit` Determines the method used to generated "fitted" dispersion estimates. One of "parametric" (the default), "local", or "mean". See the DESeq2 documentation for more information.

- `method.dispFinal` Determines the method used to arrive at a "final" dispersion estimate. The default, "shrink" uses the maximum a posteriori estimate, combining information from both the fitted and feature-specific dispersion estimates. This is the method used by DESeq2 and DEXSeq v1.12.0 and above.
- `method.sizeFactors` Determines the method used to calculate normalization size factors. By default JunctionSeq uses gene-level expression. As an alternative, feature-level counts can be used as they are in DEXSeq. In practice the difference is almost always negligible.
- `method.countVectors` Determines the type of count vectors to be used in the model framework. By default JunctionSeq compares the counts for a specific feature against the counts across the rest of the gene minus the counts for the specific feature. Alternatively, the sum of all other features on the gene can be used, like in DEXSeq. The advantage to the default JunctionSeq behavior is that no read or read-pair is ever counted more than once in any model. Under DEXSeq, some reads may cover many exonic segments and thus be counted repeatedly.
- `method.expressionEstimation` Determines the methodology used to generate feature expression estimates and relative fold changes. By default each feature is modeled separately. Under the default count-vector method, this means that the resultant relative fold changes will be a measure of the relative fold change between the feature and the gene as a whole.
  Alternatively, the "feature-vs-otherFeatures" method builds a large, complex model containing all features belonging to the gene. The coefficients for each feature are then "balanced" using linear contrasts weighted by the inverse of their variance. In general we have found this method to produce very similar results but less efficiently and less consistently. Additionally, this alternative method "multi-counts" reads that cover more than one feature. This can result in over-weighting of exonic regions with a large number of annotated variations in a small genomic area, as each individual read or read-pair may be counted many times in the model.
  Under the default option, no read or read-pair is ever counted more than once in a given model.
- `fitDispersionsForExonsAndJunctionsSeparately` When running a junctionsAndExons-type analysis in which both exons and splice junctions are being tested simultaniously, this parameter determines whether a single fitted dispersion model should be fitted for both exons and splice junctions, or if separate fitted dispersions should be calculated for each. By default the dispersion fits are run separately.

8.1 Replicating DEXSeq Analysis via JunctionSeq

Using the alternative methods options described in Section 8, it is possible to run a standard DEXSeq analysis in JunctionSeq, replicating precisely the output that would be produced by DEXSeq on the given dataset.

```
jscs.DEX <- runJunctionSeqAnalyses(sample.files = countFiles,
                  sample.names = decoder$sample.ID,
                  condition = factor(decoder$group.ID),
                  flat.gff.file = gff.file,
                  nCores = 6,
                  analysis.type = "exonsOnly",
                  method.countVectors = "sumOfAllBinsForGene",
```

```
                    method.sizeFactors = "byCountbins",
                    method.expressionEstimation = "feature-vs-otherFeatures",
                    meanCountTestableThreshold = "auto",
                    use.multigene.aggregates = TRUE,
                    method.cooksFilter = TRUE,
                    optimizeFilteringForAlpha = 0.1,
#The following option reproduces a (very minor) bug in DEXSeq v1.12.1
#  This bug was fixed in subsequent versions of DEXSeq.
#  Included only for replicability. NOT FOR GENERAL USE!
                   replicateDEXSeqBehavior.useRawBaseMean = TRUE
                   );
```

Output files and plotting can be generated using the normal syntax.

```
writeCompleteResults(jscs.DEX,
                     outfile.prefix="./testDEX",
                     save.jscs = TRUE
                     );
buildAllPlots(jscs=jscs.DEX,
              outfile.prefix = "./plotsDEX/",
              use.plotting.device = "png",
              FDR.threshold = 0.01
              );
```

Note that these will be formatted and organized in the JunctionSeq style, including the numerous improvements to the plots, along with the browser tracks designed for use with IGC or the UCSC genome browser.

Using the above method, we can precisely reproduce the exact analyses that would be generated by the following commands in DEXSeq v1.12.1:

```
library("DEXSeq");
countFiles.dexseq <- system.file(paste0("extdata/cts/",
              decoder$sample.ID,
              "/QC.exonCounts.formatted.for.DEXSeq.txt.gz"),
              package="JctSeqExData2", mustWork=TRUE);
gff.dexseq <- system.file("extdata/annoFiles/DEXSeq.gtf",
              package="JctSeqExData2", mustWork=TRUE);
dxd <- DEXSeqDataSetFromHTSeq(
   countFiles.dexseq,
   sampleData = design,
   design = ~sample + exon + condition:exon,
   flattenedfile = gff.dexseq
);
dxd <- estimateSizeFactors(dxd);
dxd <- estimateDispersions(dxd);
```

```
dxd <- testForDEU( dxd);
dxd <- estimateExonFoldChanges(dxd, fitExpToVar = "condition");
dxr <- results(dxd);
write.table(dxr,file="dxr.out.txt");
```

Note that this will only reproduce the output of DEXSeq v1.12.0 and above, as earlier versions use a slightly different algorithm. The `replicateDEXSeqBehavior.useRawBaseMean` is **NOT FOR GENERAL USE**, as it reproduces an extremely minor bug in DEXSeq v1.12.1-1.14.0 in which the base mean is calculated using non-normalized counts. In most datasets this bug generally only causes a difference of a fraction of a percent across all metrics.

These options allow standard DEXSeq methods to be applied while still providing access to the improved output plots and tables generated by JunctionSeq.

Also note: the above analyses use the DEXSeq-formatted counts generated by QoRTs. The python scripts provided with DEXSeq produce very similar counts, but with a few very minor differences. Firstly: some elements will appear in a different order, as DEXSeq uses unsorted collections and thus the ordering cannot be externally reproduced. This includes the ordering of gene names in "aggregate genes", and the ordering of transcripts in the flattened gff file.

Additionally, for un-stranded data DEXSeq does not "flatten" genes that appear in overlapping regions on opposite strands. This only affects a very small number of genes, but can produce unusual results under these rare circumstances, and thus has not been reproduced. For un-stranded data QoRTs aggregates any genes that overlap on opposite strands.

8.2 Advanced generalized linear modelling

Since JunctionSeq uses generalized linear modelling, the experimental condition variable need not have only two values. Any categorical variable will do. For example:

```
threeLevelVariable <- c("GroupA","GroupA",
                        "GroupB","GroupB",
                        "GroupC","GroupC");
```

We can then carry out analysis normally:

```
jscs <- runJunctionSeqAnalyses(sample.files = countFiles,
                           sample.names = decoder$sample.ID,
                           condition=factor(threeLevelVariable),
                           flat.gff.file = gff.file,
                           nCores = 6,
                           analysis.type = "junctionsAndExons"
                           );
```

Similarly, we could theoretically add additional covariates to our analysis. Note that it is vitally important that every class have replicates, or else JunctionSeq will be unable to accurately assess the biological variance. As a result, any analysis involving covariates should have at least 8 samples.

```
#Artificially adding additional samples by using two of the samples twice:
# (Note: this is purely for use as an example. Never do this.)
countFiles.8 <- c(countFiles, countFiles[3],countFiles[6]);
#Make up some sample names, conditions, and covariates for these samples:
decoder.8 <- data.frame(
   sample.names = factor(paste0("SAMP",1:8)),
   condition    = factor(rep(c("CASE","CTRL"),each=4)),
   smokeStatus  = factor(rep(c("Smoker","Nonsmoker"),4))
)
print(decoder.8);
```

```
##   sample.names condition smokeStatus
## 1        SAMP1      CASE      Smoker
## 2        SAMP2      CASE   Nonsmoker
## 3        SAMP3      CASE      Smoker
## 4        SAMP4      CASE   Nonsmoker
## 5        SAMP5      CTRL      Smoker
## 6        SAMP6      CTRL   Nonsmoker
## 7        SAMP7      CTRL      Smoker
## 8        SAMP8      CTRL   Nonsmoker
```

To include covariates in the analysis, you will need to modify the various model formulae used by JunctionSeq:

```
jscs <- runJunctionSeqAnalyses(sample.files = countFiles.8,
                           sample.names = decoder.8$sample.names,
                           condition= decoder.8$condition,
                           use.covars = decoder.8[,"smokeStatus",drop=F],
                           flat.gff.file = gff.file,
                           nCores = 6,
                           analysis.type = "junctionsAndExons",
test.formula0  = ~ sample + countbin + smokeStatus : countbin,
test.formula1  = ~ sample + countbin + smokeStatus : countbin + condition : countbin,
effect.formula = ~ condition + smokeStatus + countbin +
                   smokeStatus : countbin + condition : countbin,
geneLevel.formula = ~ smokeStatus + condition
);
```

Warning: multivariate analysis using GLMs is an advanced task, and JunctionSeq modelling is already fairly complex. Multivariate GLMs are intended for use by advanced users only. It's quite easy to misunderstand what the models are doing, to do something wrong, or to misinterpret the results.

References

[1] Simon Anders, Alejandro Reyes, and Wolfgang Huber. Detecting differential usage of exons from RNA-seq data. *Genome Research*, 22:2008, 2012. doi:10.1101/gr.133744.111.

[2] Simon Anders Michael I Love, Wolfgang Huber. Moderated estimation of fold change and dispersion for rna-seq data with deseq2. *Genome Biology*, 15:550, 2014. URL: http://www.genomebiology.com/content/15/12/550.

[3] Mark D. Robinson and Gordon K. Smyth. Moderated statistical tests for assessing differences in tag abundance. *Bioinformatics*, 23:2881, 2007. URL: http://bioinformatics.oxfordjournals.org/cgi/content/abstract/23/21/2881, http://arxiv.org/abs/http://bioinformatics.oxfordjournals.org/cgi/reprint/23/21/2881.pdf arXiv:http://bioinformatics.oxfordjournals.org/cgi/reprint/23/21/2881.pdf, doi:10.1093/bioinformatics/btm453.

[4] 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. URL: http://bioinformatics.oxfordjournals.org/content/29/1/15.abstract, http://arxiv.org/abs/http://bioinformatics.oxfordjournals.org/content/29/1/15.full.pdf+html arXiv:http://bioinformatics.oxfordjournals.org/content/29/1/15.full.pdf+html, doi:10.1093/bioinformatics/bts635.

[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. URL: http://bioinformatics.oxfordjournals.org/content/26/7/873.abstract, http://arxiv.org/abs/http://bioinformatics.oxfordjournals.org/content/26/7/873.full.pdf+html arXiv:http://bioinformatics.oxfordjournals.org/content/26/7/873.full.pdf+html, doi:10.1093/bioinformatics/btq057.

[6] Daehwan Kim, Geo Pertea, Cole Trapnell, Harold Pimentel, Ryan Kelley, and Steven Salzberg. Tophat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biology*, 14(4):R36, 2013. URL: http://genomebiology.com/2013/14/4/R36, http://dx.doi.org/10.1186/gb-2013-14-4-r36 doi:10.1186/gb-2013-14-4-r36.

9 Session Information

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

```
sessionInfo()

## R version 3.2.2 (2015-08-14)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
```

```
## locale:
## [1] C
##
## attached base packages:
## [1] stats4    parallel  stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] JctSeqExData2_0.6.4        JunctionSeq_0.6.4
##  [3] SummarizedExperiment_1.0.1 Biobase_2.30.0
##  [5] GenomicRanges_1.22.1       GenomeInfoDb_1.6.1
##  [7] IRanges_2.4.1              S4Vectors_0.8.1
##  [9] BiocGenerics_0.16.1        knitr_1.11
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.1               highr_0.5.1
##  [3] formatR_1.2.1             RColorBrewer_1.1-2
##  [5] futile.logger_1.4.1       XVector_0.10.0
##  [7] futile.options_1.0.0      tools_3.2.2
##  [9] zlibbioc_1.16.0           statmod_1.4.22
## [11] annotate_1.48.0           evaluate_0.8
## [13] RSQLite_1.0.0             lattice_0.20-33
## [15] DBI_0.3.1                 RcppArmadillo_0.6.200.2.0
## [17] genefilter_1.52.0         stringr_1.0.0
## [19] locfit_1.5-9.1            grid_3.2.2
## [21] plotrix_3.6               AnnotationDbi_1.32.0
## [23] XML_3.98-1.3              survival_2.38-3
## [25] BiocParallel_1.4.0        geneplotter_1.48.0
## [27] lambda.r_1.1.7            magrittr_1.5
## [29] splines_3.2.2             BiocStyle_1.8.0
## [31] xtable_1.8-0              KernSmooth_2.23-15
## [33] stringi_1.0-1
```

10 Legal