# RNA-Seq Tutorial (EBI, October 2011)

Nicolas Delhomme

November 2, 2013

## Contents

# 1 Introduction

This file describes a RNA-Seq analysis use-case. RNA-Seq [Mortazavi et al., 2008] was introduced as a new method to perform Gene Expression Analysis, using the advantages of the high throughput of *Next-Generation Sequencing* (NGS) machines. The goal of this use-case is to generate a count table for the selected genic features of interest, *i.e.* exons, transcripts, gene models, *etc.*

In the first part, the data will be read in R using Bioconductor [Gentleman et al., 2004] *ShortRead* [Morgan et al., 2009], *Rsamtools* and *GenomicRanges* packages. Then, annotation will be retrieved using the *biomaRt* [Durinck et al., 2005] and *genomeIntervals* packages. Finally the *IRanges* and *GenomicFeatures* packages will be used to define the reads coverage, and assign counts to genic features of interest.

In the second part, we will see how this process can be simplified by the use of the *easyRNASeq* package [Delhomme et al., 2012] and how more advanced pre-processing can be performed, such as *de-multiplexing*, *RPKM* "correction" or normalization using the *DESeq* or *edgeR* packages.

Finally, the count information will be exported as bed and wig formatted file, to be visualized into the UCSC genome browser or a stand alone genome browser like IGB.

The overall process is described in figure 1, page 3. First, the genomic and genic annotation will be retrieved from the selected/preferred source and converted into an appropriate `object`. In parallel, the sequenced reads' information (*e.g.* chromosome, position, strand, *etc.*) will be retrieved from the alignment file and, as well, converted to a similar `object`. Then, the reads contained in the `reads` object are summarized per genic annotation contained in the `annotation` object. This give raise to a count table that, finally, can be normalized using additional R packages.
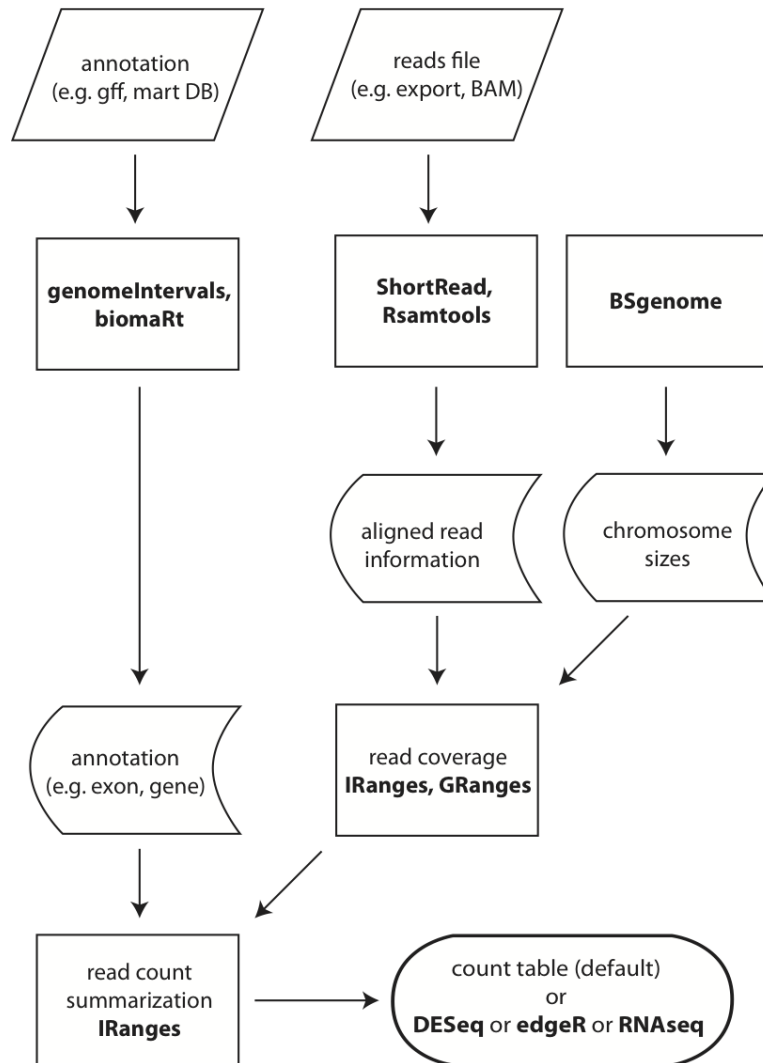
Figure 1: RNA-Seq Procedure Overview. The R packages used for the different steps are emphasized in bold face.

# 2 Walk through a single sample use case

In this section, we will mainly look into the details of generating a count table, *i.e.* how many sequencing reads can be assigned to given genic features. An expressed genic feature can be anything from an exon to a gene-model or, as recently published, enhancers [Kim et al., 2010]. In this section, you will learn how to read "raw" data in, load the related annotations, calculate the reads genomic coverage and deduce read counts per exons. But first, we need to load the tutorial library and data.

```
> library(RnaSeqTutorial)
```

## 2.1 Reading the data

There are different ways to read in NGS data into R, depending on the "raw" data format at hand. The next paragraphs will present three different approaches, introducing the *ShortRead*, *Rsamtools* and *GenomicRanges* packages capabilities.

**ShortRead**   *ShortRead* was the first NGS package developed to read in NGS data and is able to read almost every sequencer's manufacturer proprietary formats (with the notable exception of ABI color-space). First, an Illumina "export" file produced by a GenomeAnalyzer GAIIx will be read in; and then the same data set, in BAM format.

    **Illumina export**   The export file is read in using the `readAligned` function, and the resulting object displayed.

```
> library(ShortRead)
> aln<-readAligned(
+                   system.file("extdata",package="RnaSeqTutorial"),
+                   pattern="subset_export",type="SolexaExport")
> show(aln)

class: AlignedRead
length: 100000 reads; width: 36 cycles
chromosome: NM 1:0:0 ... chr2R chr2L
position: NA NA ... 20555556 13903608
strand: NA NA ... + -
alignQuality: NumericQuality
alignData varLabels: run lane ... filtering contig
```

    *N.B.*: The `system.file` retrieves the file path where a package was installed. Additional argument can be provided in a similar fashion to that of `file.path` to access directories and files within the package.

    The `readAligned` function returns an object of the *AlignedRead* class. The main slots can be accessed using similarly named accessors as described in the following code sample:

```
> chromosome(aln)
> levels(chromosome(aln))
> position(aln)[1:100]
> width(aln)[1:100]
```

```
> strand(aln)[1:100]
> sread(aln)
> quality(aln)
```

The Illumina "export" format contains every read sequenced on the platform, as well as these coming from overlapping sequence clusters. These sequences are flagged by the Illumina pipeline through a *chastity filter*. It is important for processing such "raw" data to remove these reads. This will be done in the next section: 2.2, page 10. First, we will just find out the value of the chastity filter field ("Y" or "N" whether it passes the filter or not) within the current object:

```
> alignData(aln)$filtering[1:100]
```

**BAM** The SAM/BAM format [Li et al., 2009] is becoming a *de-facto* standard for storing NGS aligned data. As a consequence, the *Rsamtools* was developed to import the *samtools* functionalities into the R environment. Subsequently, the *ShortRead* package was extended to use *Rsamtools* to load BAM files. SAM/BAM files can be sorted by chromosomal position, in which case an index can be created that will improve the subsequent retrieval of information within the BAM file. This can be done using the "samtools" command in a terminal, or using the Rsamtools package that implements in R most of the "samtools" and "bcftools" functionalities. In the following example, you will first create an index for the BAM file and then read the data into an *AlignedRead* class object.

```
> library(Rsamtools)
> file.copy(
+          system.file("extdata",
+                      "subset.bam",
+                      package="RnaSeqTutorial"),
+          getwd())

[1] TRUE

> indexFile <- indexBam("subset.bam")
> basename(indexFile)

[1] "subset.bam.bai"

> aln2 <- readAligned(getwd(),pattern="subset.bam$",type="BAM")
```

**Q1**: What differences exists between the `aln` and `aln2` objects?

**Answers** are provided in the Appendix A, page 32.

**Rsamtools**   A different, more flexible way to load BAM formatted files is to directly use the `scanBam` function from the *Rsamtools* package.

```
> aln2b <- scanBam(
+                  "subset.bam",
+                  index="subset.bam"
+                  )
> names(aln2b[[1]])

 [1] "qname"  "flag"   "rname"  "strand" "pos"    "qwidth" "mapq"   "cigar"
 [9] "mrnm"   "mpos"   "isize"  "seq"    "qual"
```

*N.B.* First, the *index* argument is the filename, *i.e.* the ".bai" extension is ignored. Second, (scanBam) returns a list of list, with the first list having a single element.

The inner lists contains an element per column of the BAM file formats, with the optional fields being ignored. The `scanBam` function extends the base R "scan" function. Additional parameters can be provided to select the reads: see the `scanBamParam` argument for filtering the reads based on their flag, cigar string, *etc.*; see the `ScanBamParam` to display which fields of the BAM file are to be retrieved.

**GenomicRanges**   Finally, the last example to load data uses the *GenomicRanges* package. With the increasing read length, it became possible to reliably map exon-exon junctions and therefore to report *gapped alignments* and alternative transcripts. Tools such as TopHat [Trapnell et al., 2009] have especially been developed for that purpose. The *GenomicRanges* package was implemented to deal with this new kind of data. As of today, most of the common aligners such as bowtie [Langmead et al., 2009], bwa [Li and Durbin, 2009], novoalign [Novocraft.com, 2009] or GSNAP [Wu and Nacu, 2010] supports gap alignments, an additional motivation to use the *GenomicRanges* package functionalities.

```
> library(GenomicRanges)
> aln3 <- readGAlignments(
+                         system.file("extdata",
```

```
+                                          "gapped.bam",
+                                          package="RnaSeqTutorial"),
+                           format="BAM")
```

The generated object of class *GAlignments*, is very different from the *AlignedRead* class objects you have seen so far.

**Q2** What are the most obvious differences?
**Q3**: Does the `aln3` object contains evidence of exon-exon junctions?
Tip: use the cigar functionalities: `cigar`, `cigarOpTable`, *etc.*

**Caveats**  Finally, before we go on with data filtering, a few caveats need to be mentioned.

**Illumina export**  The *ShortRead* package, when loading an export file does not retrieve all the possible information; *i.e.* the `id` slot of the *AlignedRead* object contains no valid information.

```
> head(id(aln))

  A BStringSet instance of length 6
    width seq
[1]     0
[2]     0
[3]     0
[4]     0
[5]     0
[6]     0
```

The *withMultiplexIndex*, *withPairedReadNumber*, *withId* and *withAll* arguments offer the possibility to retrieve such information.

```
> aln4<-readAligned(
+                     system.file("extdata",package="RnaSeqTutorial"),
+                     pattern="subset_export",type="SolexaExport",
+                     withId=TRUE)
> head(id(aln4))

  A BStringSet instance of length 6
    width seq
[1]     26 HWI-EAS225_90320:3:1:0:519
```

```
[2]     27 HWI-EAS225_90320:3:1:0:1860
[3]     27 HWI-EAS225_90320:3:1:0:1013
[4]     26 HWI-EAS225_90320:3:1:0:747
[5]     27 HWI-EAS225_90320:3:1:0:1512
[6]     26 HWI-EAS225_90320:3:1:0:990
```

**BAM**   In a BAM file, the reads are stored as they are aligned against the reference genome! Hence, these are not necessarily the "sequence" that have been read by the sequencer. The *ShortRead*, when loading a BAM file revert the reads to their original sequence.

```
> sel <- !is.na(strand(aln2)) & strand(aln2) %in% "-"
> aln2b[[1]]$seq[sel][1]

  A DNAStringSet instance of length 1
    width seq
[1]    36 AAAAAGTGGAGCCGCTCCTTTCCATTTTTGATTTCC

> sread(aln2[sel])[1]

  A DNAStringSet instance of length 1
    width seq
[1]    36 GGAAATCAAAAATGGAAAGGAGCGGCTCCACTTTTT

> reverseComplement(aln2b[[1]]$seq[sel][1])

  A DNAStringSet instance of length 1
    width seq
[1]    36 GGAAATCAAAAATGGAAAGGAGCGGCTCCACTTTTT
```

**Conclusion**   In that section, we have seen three packages that allow loading NGS data into R, as well as some caveats related to the format these data can be in. Depending on the data format, this step might only be the first one and some additional pre-processing might be necessary, as described in the next paragraph.

## 2.2 Filtering the data

As one can see, many reads do not pass the chastity filter and many reads do not align to the genome. In addition some of those reads do contain many **N**s; that is whenever Bustard, the Illumina base caller, could not perform a valid base call. All the chastity flagged reads should be filtered out. The N-containing reads can be filtered out according to the number of mismatch you are willing to have in your data. Filtering for failed chastity calls is not implemented in the *ShortRead* package, but is in the *easyRNASeq* package. In the following example, several filters are combined to keep reads that align to reference chromosomes, that do not have more than 2 Ns and that pass the chastity filter.

```
> library(easyRNASeq)
> nFilt <- nFilter(2)
> chrFilt <- chromosomeFilter(regex="chr")
> cFilt <- chastityFilter()
> filt <- compose(nFilt,chrFilt,cFilt)
> aln <- aln[filt(aln)]

> show(aln)
```

We are now left with 56,883 "valid" reads, which we want to assign to their respective exon. For this we need to get the proper genomic and genic information.

**Conclusion**   We've seen how to load in R the raw data (the aligned reads). We therefore have the information where these reads are located in the genome. Now, we want to discover if these loci covered by reads corresponds to interesting genomic or genic features, *e.g.* exons, promoters, *etc.*. To achieve this, we first need to load the genic/genomic annotation in R. This will be the topic of the next subsection.

## 2.3 Loading the annotation

To assign reads to exons, we need to know the genome composition, *i.e.* how many chromosomes, their names and sizes. In addition, we need to know the genic information, *i.e.* where are exons located, which gene they belong to, *etc.*

**Genomic information** The reads present in the "subset" file used previously, come from an RNA-Seq experiment conducted in *Drosophila melanogaster*. First, the genomic information for that organism need to be retrieved. You will use the *BSgenome* package for this.

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> chrSizes <-seqlengths(Dmelanogaster)
> chrSizes
```

| chr2L | chr2R | chr3L | chr3R | chr4 | chrX | chrU | chrM |
|---|---|---|---|---|---|---|---|
| 23011544 | 21146708 | 24543557 | 27905053 | 1351857 | 22422827 | 10049037 | 19517 |
| chr2LHet | chr2RHet | chr3LHet | chr3RHet | chrXHet | chrYHet | chrUextra | |
| 368872 | 3288761 | 2555491 | 2517507 | 204112 | 347038 | 29004656 | |

**Genic information** To retrieve the genic annotation, several solutions are available:

- the *biomaRt* package to fetch information from "Mart" databases

- the *genomeInterval* package to read data in from "gff" annotation files obtained from FlyBase [Tweedie et al., 2009], Ensembl [Flicek et al., 2011], UCSC, or using proprietary annotations.

- the *rtracklayer* package to import "gff", "bed" or "wig" files.

- the *GenomicFeatures* package to retrieve information from the UCSC databases or from the "Mart" databases through the *biomaRt* package interface.

These four methods have strengths and drawbacks and selecting the most appropriate is essentially depending on your computing environment, *e.g.* lots of memory vs. lots of disk space as well as on your computing proficiency. Shortly, the two first approaches will require the post-processing of the obtained data into a *RangedData* or *GRanges* class object. The first and last require an internet connection, at least for initially downloading

11

the data. Such data can be saved locally, not to have to download them again and again. Having a local frozen copy is anyway a good practice to ensure reproducibility. The last one requires to write an *SQLite* database locally. The two last returns object that do not need post-processing; however the *rtracklayer* `import` function is not robust to incorrect gff files and the *GenomicFeatures* is limited to the genomes available in its datasources.

**biomaRt** The *biomaRt* package remotely queries Mart services. To get the *Drosophila melanogaster* genomic information, a connection is established with the Ensembl fruit fly Mart database and queried for the information we need (gene ID, transcript ID, exon ID, position, *etc.*). To limit the amount of data to be retrieved, a filter is set to select for the chromosomes of interest. The *filters* argument defines the criteria to use as filters and the *values'* one defines the values that are accepted for these criteria.

```
> library(biomaRt)
> ensembl <- useMart("ensembl",
+                    dataset="dmelanogaster_gene_ensembl")
> exon.annotation<-getBM(
+                        c("ensembl_gene_id",
+                          "strand",
+                          "ensembl_transcript_id",
+                          "chromosome_name",
+                          "ensembl_exon_id",
+                          "exon_chrom_start",
+                          "exon_chrom_end"),
+                        mart=ensembl,
+                        filters="chromosome_name",
+                        values=c("2L","2R","3L","3R","4","X"))
```

As mentioned, the obtained `data.frame` needs to be converted into an object of class *RangedData* or *GRanges*. Note that the chromosome names retrieved from Ensembl, compliant with the *FlyBase* annotation are not UCSC compliant and need to be converted; *i.e.* the "chr" string needs to be prepended.

```
> exon.annotation$chromosome <- paste(
+                                      "chr",
+                                      exon.annotation$chromosome_name,
+                                      sep="")
```

```
> exon.range <- RangedData(
+                          IRanges(
+                                  start=exon.annotation$exon_chrom_start,
+                                  end=exon.annotation$exon_chrom_end),
+                          space=exon.annotation$chromosome,
+                          strand=exon.annotation$strand,
+                          transcript=exon.annotation$ensembl_transcript_id,
+                          gene=exon.annotation$ensembl_gene_id,
+                          exon=exon.annotation$ensembl_exon_id,
+                          universe = "Dm3"
+                          )
```

This created a *RangedData* class object.

```
> show(exon.range)
```

**Q4**: How would you create a *GRanges* object?

**genomeIntervals**    The `readGff3` of the *genomeIntervals* is a robust
and efficient way to load Generic Feature Format (gff) file. The latest version
of that format: *version 3* is used by most model organism websites, or similar
format have been derived from it, such as the Gene Transfer Format (gtf)
used among others by Ensembl. The `readGff3` is flexible enough to cope
with gtf formatted files too.

```
> library(genomeIntervals)
> gInterval<-readGff3(system.file("extdata",
+                                 "annot.gff",
+                                 package="RnaSeqTutorial"))
```

As for *biomaRt*, post-processing the obtained object is necessary. The
gff file has been retrieved from *FlyBase* and filtered for the "exon" type. It
contains the *gffAttributes ID*, *Name* and *Parent* defining the "exon ID", the
"gene ID" and the "transcript ID" respectively.

```
> exon.range2 <- RangedData(
+                           IRanges(
+                                   start=gInterval[,1],
+                                   end=gInterval[,2]),
+                           space=gInterval$seq_name,
+                           strand=gInterval$strand,
```

13

```
+                                  transcript=as.vector(
+                                    getGffAttribute(gInterval,"Parent")),
+                                  gene=as.vector(
+                                    getGffAttribute(gInterval,"Name")),
+                                  exon=as.vector(
+                                    getGffAttribute(gInterval,"ID")),
+                                  universe = "Dm3"
+                                  )
```

**Q5**: How would you create a *GRanges* object from the `gInterval`?

**Q6**: (optional) How would you *export* the "gAnnot.rda" file, present in the "data" directory into a gff version *3* formatted file?

**rtracklayer**  Importing a properly formatted "gff" file is straightforward.

```
> library(rtracklayer)
> exon.range3<-import.gff3(
+                     system.file("extdata",
+                               "annot.gff",
+                               package="RnaSeqTutorial")
+                     )
```

**GenomicFeatures**  The *GenomicFeatures* can retrieve data by connecting "UCSC" or by through the *biomaRt* package interface. The two related functions are `makeTranscriptDbFromUCSC` and `makeTranscript-DbFromBiomart`. It creates a TranscriptDb object that can be queried using the `transcript`, `exon` and `cds` functions.

```
> library(GenomicFeatures)

> dm3.tx <- makeTranscriptDbFromUCSC(
+                                       genome="dm3",
+                                       tablename="refGene")
> exon.range4 <- exons(dm3.tx)
> exon.range4
```

As you can see a certain amount of warnings are raised and the actual annotation looks different (somewhat less complete) than the previous ones. To avoid downloading the annotation everytime they are needed, the annotation object (`dm3.tx` in that case) can be saved to disk. This is actually a good practice as well to ensure the reproducibility of your analyses.

**Caveats** All the previous steps help retrieve the necessary genomic and genic information, however it is **essential** for the user to pay attention that the proper annotation are gathered. The *GenomicFeatures* is under very active development, so changes are to be expected. If your genome is not in it,do not hesitate to post on the mailing list,to make the changes happen! It is as well **essential** that you understand the content of your annotation and its consequences on your analysis.*I.e.* most of the previously described approaches will result in annotation containing some overlapping genic feature, *e.g.* genes on opposite strand, exons shared by transcripts, *etc.* Using them as is will results in counting some reads multiple times, introducing some possible bias. It is obviously important to avoid such behavior and this requires to check and validate you annotation.

## 2.4 Summarizing read counts per feature

Now that all the annotations have been retrieved (`exon.range2` and `chr-Sizes`) and the data loaded (`aln`), the read coverage can be extracted and then summarized per feature of interest.

**Calculating the coverage**

```
> cover <- coverage(aln,width=chrSizes)

> show(cover)
> show(cover$chr2R)
```

    *RleList* objects are clever way of encoding a coverage vector. Such a vector, contains one value per bp and is therefore greedy. However many successive bp might have the same coverage value and therefore could be considered as intervals that have a given coverage value. This is exactly what an *Rle* object does. It is constituted of "runs" that encode the length of an interval together with its coverage value.

**Q7**: How would you get the actual bp coverage from an *Rle* class object? Tip: select the "chr4" out of your `cover` *RleList*; it is the smallest.
**Q8**: How would you access the "run" lengths and values?

The *Rle* class derives from the *Sequence* one and therefore has numerous capabilities. The following example display some:

```
> runLength(cover$chr4)[1:3]

[1] 59510    36  2019

> runValue(cover$chr4)[1:3]

[1] 0 1 0

> r.start <- runLength(cover$chr4)[1]+1
> r.end <- sum(runLength(cover$chr4)[1:2])
> as.integer(cover$chr4)[r.start:r.end]

 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

> as.integer(window(cover$chr4,r.start,r.end))
```

```
[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

The previous example identifies a region covered by a single read and
retrieve the actual bp coverage from it. Note the use of the `window` function.
It significantly fastens such approaches; *i.e.* it works similarly to a *Views*.
*Views* is a general container for storing a set of "views" on an object; *i.e.*
a view simply records the position of interest on the object rather than
creating a copy of that object. For example, a view on a genomic sequence
such as a chromosome would simply register the start and the width rather
than the actual sequence.

**Aggregating the coverage per exon**   Now to convert the genomic cov-
erage into an exonic one, we need to use the genic annotation retrieved
previously. The coverage object: `cover` we have generated earlier describes
the number of reads overlapping every bp in the genome. To summarize its
values per exon, the simplest approach is to average the coverage for all bp
covered by a given exon.

```
> exon.coverage<-aggregate(
+                          cover[match(names(exon.range2),names(cover))],
+                          ranges(exon.range2),
+                          sum)
> exon.coverage <- ceiling(exon.coverage/unique(width(aln)))

> show(exon.coverage)
```

   **Tip**   Using Views actually make this approach much faster:

```
> viewSums(
+        Views(
+              cover[match(names(exon.range2),names(cover))],
+              ranges(exon.range2)))
```

   **Caveats**   Note the `match` that is done between the `names(exon.range2)`
and `names(cover)`.

**Q9**: Why is it so **essential**?
**Q10**: List the possible drawback of this approach.
**Q11**: Use the `countOverlaps` to overcome some of these limitations.

## 2.5 Conclusion

Now, you are done with processing that single sample. If you were to do this for many samples, it would be demanding and probably not ideally fail-safe. Rationalizing and automating that task was our motivation to build the *easyRNASeq* package. However, keep in mind that getting the proper annotation for your analysis is an essential step, as well when using the *easyRNASeq* package [Delhomme et al., 2012]. You'll probably want to avoid counting the same read multiple times, *e.g.* in overlapping exons.

# 3 Using easyRNASeq

Let us redo what was done in the previous sections. Note that most of the *RNAseq* object slots are optional. However, it is advised to set them, especially the *readLength* and the *organismName*; to help having a proper documentation of your analysis. The organismName slot is actually mandatory if you want to get genomic annotation using *biomaRt*. In that case, you need to provide the name as specified in the corresponding *BSgenome* package, *i.e.* "Dmelanogaster" for the *BSgenome.Dmelanogaster.UCSC.dm3* package.

## 3.1 easyRNASeq

```
> ## load the library
> library("easyRNASeq")
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> count.table <- easyRNASeq(system.file(
+                                       "extdata",
+                                       package="RnaSeqTutorial"),
+                          organism="Dmelanogaster",
+                          chr.sizes=as.list(seqlengths(Dmelanogaster)),
+                          readLength=36L,
+                          annotationMethod="rda",
+                          annotationFile=system.file(
+                             "data",
+                             "gAnnot.rda",
+                             package="RnaSeqTutorial"),
+                          format="bam",
+                          count="exons",
+                          pattern="[A,C,T,G]{6}\\.bam$")
> head(count.table)
> dim(count.table)
```

That is all. In one command, you got the count table for your 4 samples!

**Warnings**   As you could see when running the previous example, warnings were emitted and quite rightly so.

1. about the annotation: The annotation we are using here is redundant and this at two levels. First, some exons overlap. These are alternative exons from different transcript isoforms. Second, the annotation

19

contains the information about all the possible different transcript isoforms. This means that some exons are duplicated. Therefore counting by exons or transcripts using these annotation will result in counting some of the reads several times. There might be reasons one might want to do that, but as it is probably not what you want when performing an RNA-Seq analysis, the warning is emitted. As this can be a very significant source of error, all the examples here will emit this warning. The ideal solution is to provide an annotation object that contains no overlapping features. The `disjoin` function from the *IRanges* package offers a way to achieve this.

2. about potential naming issue in the input file: It is (sadly) very frequent that the sequencing facilities use different naming conventions for the chromosomes they report in the alignment files. It is therefore very frequent that the annotation provided to easyRNASeq uses different chromosome names than the alignment file. These warnings are there to inform you about this issue.

**Details**   The `easyRNASeq` function currently accepts the following `annotationMethods`:

- "biomaRt" use biomaRt to retrieve the annotation

- "env" use a *RangedData* class object present in the environment

- "gff" reads in a gff version 3 file

- "gtf" reads in a gtf file

- "rda" load an RData object. The object needs to be named `gAnnot` and of class *RangedData*.

The reads can be read in from BAM files or any format supported by *ShortRead*.

The reads can be summarized by:

- exons

- features (any features such as introns, enhancers, *etc.*)

- transcripts

20

- geneModels (a geneModel is the set of non overlapping loci (*i.e.* synthetic exons) that represents all the possible exons and UTRs of a gene. Such geneModels are essential when counting reads as they ensure that no reads will be accounted for several times. *E.g.*, a gene can have different isoforms, using different exons, overlapping exons, in which case summarizing by exons might result in counting a read several times, once per overlapping exon. *N.B.* Assessing differential expression between transcripts, based on synthetic exons is something possible since the release *2.14* of R, using the *DEXSeq* package available from Bioconductor.

The results can be exported in four different formats:

- count table (the default, a n (features) x m (samples) `matrix`).

- a *DESeq* [Anders and Huber, 2010] *countDataSet* class object. Useful to perform further analyses using the *DESeq* package.

- an *edgeR* [Robinson et al., 2010] *DGEList* class object. Useful to perform further analyses using the *edgeR* package.

- an *RNAseq* class object. Useful for performing additional pre-processing without re-loading the reads and annotations.

For more details and a complete overview of the *easyRNASeq* package capabilities, have a look at the *easyRNASeq* vignette.

```
> vignette("easyRNASeq")
```

The obtained results can optionally be normalized as *Reads per Kilobase of feature per Million reads in the library* (RPKM, Mortazavi et al. [2008]) or using the *DESeq* or *edgeR* packages.

**Q12**: From the same input files and annotations, generate an object of class *RNAseq* .
**Q13**: Summarize the counts per transcript and geneModels.

**Finally...** If you find *easyRNASeq* useful and apply it in the frame of your research for a publication, please cite it:

easyRNASeq: a bioconductor package for processing RNA-Seq data
Nicolas Delhomme; Ismael Padioleau; Eileen E. Furlong; Lars M. Steinmetz
Bioinformatics 2012; doi: 10.1093/bioinformatics/bts477

# 4 Advanced usage

In this section we will discuss about more advanced RNA-Seq pre-processing, such as de-multiplexing, normalizing or *de novo* identification of expressed regions.

## 4.1 Normalizing counts

A common way to normalize reads is to convert them to RPKM. This implies normalizing the read counts depending on the genic feature size (exon, transcript, gene model,...) and on the total number of reads sequenced for that library. *easyRNASeq* count tables can be easily transformed into RPKM, by using the RPKM method:

```
> feature.size = width(exon.range2)
> names(feature.size) = exon.range2$exon
> feature.size <- feature.size[!duplicated(names(feature.size))]
> lib.size=c("ACACTG.bam"=56643,
+    "ACTAGC.bam"=42698,
+    "ATGGCT.bam"=55414,
+    "TTGCGA.bam"=60740)
> head(RPKM(count.table,NULL,
+           lib.size=lib.size,
+           feature.size=feature.size))
```

**Q14**: Do the same for the object created at the **Q12** and **Q13** for every possible counts.

Such a count normalization is suited for visualization, but sub-optimal for further analyses . A better way of normalizing the data is to use either the *edgeR* or *DESeq* packages, provided you have got enough (biological) replicates. Refer to the *easyRNASeq*, the *DESeq* and the *edgeR* vignettes.

**Q15**: Perform the examples in the *easyRNASeq* vignette paragraph 3.7.

## 4.2 De-multiplexing samples

This part of the tutorial is now in the *easyRNASeq* vignette paragraph 4.
    **Q16**: Perform the examples in the *easyRNASeq* vignette paragraph 4.

    **Q17**: How would you access the barcode sequence in the `alns` object?


# 5 Visualizing the data

Before performing any more advanced analyses, it is crucial to be able to visualize the data. Many technical or procedural problems can be identified and resolved, making the residual filtered data of better quality.
The *rtracklayer* library provides the necessary functionalities to export data stored in *GenomicData* and *RangedData* class objects.

## 5.1 exporting the coverage

A common technical problem in NGS data is PCR amplification biases. These can be visualized quite easily by scanning the read coverage across chromosomes in a *Genome Browser*. To achieve this, one needs to export the coverage into a wig formatted file. The wig format expects constant span and constant steps. This is problematic since, if you have a step of 50bp and want a span of 50 bp, you'd need a chromosome whose size is a multiple of 50bp. Luckily the chromosome size is not enforced by Genome Browsers, so a small hack (5th line) does it (actually it is useless here as the chromosome 4 size is a natural multiple of the selected window size).

```
> library(rtracklayer)
> window.size <- 51
> rngs <- breakInChunks(length(cover[["chr4"]]),window.size)
> vals <- viewSums(Views(cover[["chr4"]],rngs))
> width(rngs)[width(rngs) != width(rngs)[1]] <- width(rngs)[1]
> silent <- export(
+                   RangedData(rngs,score=vals,universe="Dmelanogaster",space="chr4"),
+                   con="chr4.wig"
+                   )
```

## 5.2 exporting the normalized exon counts

Depending on the kind of experiments, other criteria can be visually assessed. For example, in the case of a gene over-expression in a sample, one could visually check the score obtained for that gene.

```
> exon.RPKM <- easyRNASeq(system.file(
+                                     "extdata",
+                                     package="RnaSeqTutorial"),
+                          organism="Dmelanogaster",
+                          chr.sizes=as.list(seqlengths(Dmelanogaster)),
+                          readLength=36L,
+                          annotationMethod="rda",
+                          annotationFile=system.file(
+                             "data",
+                             "gAnnot.rda",
+                             package="RnaSeqTutorial"),
+                          format="aln",
+                          count="exons",
+                          normalize=TRUE,
+                          pattern="subset_export",
+                          type="SolexaExport",
+                          filter=compose(
+                             chastityFilter(),
+                             nFilter(2),
+                             chromosomeFilter(regex="chr")))
> exons <- exon.range2
> exons <- exons[!duplicated(exons$exon),]
> exons$score <- exon.RPKM[,1]
> exons$name <- rownames(exon.RPKM)
> exons <- exons[exons$score>0,]
> export(exons,con="exons.bed")
```

## 5.3 conclusion

In this section, we have seen how to export the data in a lightweight fashion to be visualized in a Genome Browser. This step is an **essential** step for validating the data. The QA processes run on the raw or aligned data might reveal *technical* issues, but other biases might still be present in your data and the best (only?) way to control for those is visual. For example, one could compare replicates (in which case, biological replicates are best), by

plotting a scatterplot of both replicates and estimating their correlation. Keeping in mind the design of the experiments, helps design the necessary QA step one can do; *i.e.* In an RNA-Seq experiment, where a gene has been over-expressed, you would expect to be able to visualize it when comparing it to a control sample.

# 6 You are done, but still there is more to come...

New protocols, new packages are constantly being developed; making pre-processing NGS data a moving target. For example, it is known that the standard Illumina RNA-Seq protocol shows a bias in the first 12 nucleotides of every read. It is still unclear where this bias comes from (fragmentation, random hexamer priming, RNAseH sequence specificity), but there has been a couple of publication recently that proposes corrections for that bias [Li et al., 2010, Hansen et al., 2010].

Feedback and requests are very welcome. Just look at the *Final Remarks section* 8, page 29. for contact details.

# 7 Session Information

The version number of R[R Development Core Team, 2009] and packages
loaded for generating the vignette were:

```
R version 3.0.2 Patched (2013-10-30 r64123)
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] BSgenome.Dmelanogaster.UCSC.dm3_1.3.19
 [2] BSgenome_1.30.0
 [3] RnaSeqTutorial_0.0.14
 [4] easyRNASeq_1.8.1
 [5] ShortRead_1.20.0
 [6] Rsamtools_1.14.1
 [7] GenomicRanges_1.14.3
 [8] DESeq_1.14.0
 [9] lattice_0.20-24
[10] locfit_1.5-9.1
[11] Biostrings_2.30.0
[12] XVector_0.2.0
[13] IRanges_1.20.4
[14] edgeR_3.4.0
[15] limma_3.18.1
[16] biomaRt_2.18.0
[17] Biobase_2.22.0
[18] genomeIntervals_1.18.0
[19] BiocGenerics_0.8.0
```

```
[20] intervals_0.14.0

loaded via a namespace (and not attached):
 [1] AnnotationDbi_1.24.0 DBI_0.2-7             LSD_2.5
 [4] RColorBrewer_1.0-5   RCurl_1.95-4.1        RSQLite_0.11.4
 [7] XML_3.98-1.1         annotate_1.40.0       bitops_1.0-6
[10] genefilter_1.44.0    geneplotter_1.40.0    grid_3.0.2
[13] hwriter_1.3          latticeExtra_0.6-26   splines_3.0.2
[16] stats4_3.0.2         survival_2.37-4       tools_3.0.2
[19] xtable_1.7-1         zlibbioc_1.8.0
```

# 8  Final remarks

RNA-Seq is still maturating and a lot of new developments are to be expected. If you have any questions, comments, feel free to contact me: delhomme *at* embl *dot* de.

The author want to thank *Gabriella Rustici* for her time, comments and patience, as well as for organizing the course.

# References

Simon Anders and Wolfgang Huber. Differential expression analysis for sequence count data. *Genome Biology 2010 11:202*, 11(10):R106, Oct 2010.

Nicolas Delhomme, Ismaël Padioleau, Eileen E Furlong, and Larsm Steinmetz. easyrnaseq: a bioconductor package for processing rna-seq data. *Bioinformatics*, Jul 2012. doi: 10.1093/bioinformatics/bts477.

Steffen Durinck et al. Biomart and bioconductor: a powerful link between biological databases and microarray data analysis. *Bioinformatics*, 21(16): 3439–40, Aug 2005.

Paul Flicek et al. Ensembl 2011. *Nucleic Acids Research*, 39(Database issue): D800–6, Jan 2011.

Robert C Gentleman et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology 2010 11:202*, 5(10):R80, Jan 2004.

Kasper D Hansen, Steven E Brenner, and Sandrine Dudoit. Biases in illumina transcriptome sequencing caused by random hexamer priming. *Nucleic Acids Research*, Apr 2010.

Tae-Kyung Kim et al. Widespread transcription at neuronal activity-regulated enhancers. *Nature*, 465(7295):182–7, May 2010.

Ben Langmead et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology 2010 11:202*, 10(3): R25, Jan 2009.

Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–60, Jul 2009.

Heng Li et al. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–9, Aug 2009.

Jun Li, Hui Jiang, and Wing Hung Wong. Modeling non-uniformity in short-read rates in rna-seq data. *Genome Biology 2010 11:202*, 11(5):R50, May 2010.

Martin Morgan et al. Shortread: a bioconductor package for input, quality assessment and exploration of high-throughput sequence data. *Bioinformatics*, 25(19):2607–8, Oct 2009.

Ali Mortazavi et al. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature Methods*, 5(7):621–8, Jul 2008.

Novocraft.com. *Novoalign*. Novocraft.com, Selangor, Malaysia, 2009. URL `http://www.novocraft.com/main/index.php`.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Mark D Robinson et al. edger: a bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26(1): 139–40, Jan 2010.

C Trapnell et al. Tophat: discovering splice junctions with rna-seq. *Bioinformatics*, 25(9):1105–1111, May 2009.

Susan Tweedie et al. Flybase: enhancing drosophila gene ontology annotations. *Nucleic Acids Research*, 37(Database issue):D555–9, Jan 2009.

Thomas D Wu and Serban Nacu. Fast and snp-tolerant detection of complex variants and splicing in short reads. *Bioinformatics*, 26(7):873–81, Apr 2010.

# A  Appendix A: Solutions

- **Q1**: section 2.1, page 7: The `aln` and `aln2` objects differs by their number of reads. In the `aln2`, the *chastity filtered* reads have been ignored.

- **Q2**: section 2.1, page 8: The `aln3`, of class *GAlignments* from the *GenomicRanges* package does not contain any sequence, quality or id information.

- **Q3**: section 2.1, page 8: using the `ngap` function gives you the number of alignment having a gap, but not the size of that gap. This can be accessed using the `cigarOpTable` and `cigar` as in the example below:

```
> table(ngap(aln3))
> table(cigarOpTable(cigar(aln3))[,"N"])
```

- **Q4**: section 2.3, page 13: using the `GRanges` constructor as follow:

```
> exon.grange <- GRanges(
+                      IRanges(
+                              start=exon.annotation$exon_chrom_start,
+                              end=exon.annotation$exon_chrom_end),
+                      seqnames=Rle(exon.annotation$chromosome),
+                      strand=Rle(exon.annotation$strand),
+                      transcript=exon.annotation$ensembl_transcript_id,
+                      gene=exon.annotation$ensembl_gene_id,
+                      exon=exon.annotation$ensembl_exon_id,
+                      seqlengths = chrSizes[
+                        match(
+                              unique(exon.annotation$chromosome),
+                              names(chrSizes))]
+                      )
```

- **Q5**: section 2.3, page 14: using the `GRanges` constructor as follow:

```
> levels(gInterval$strand) <- c("-","+")
> exon.grange2 <- GRanges(
+                      IRanges(
+                              start=gInterval[,1],
+                              end=gInterval[,2]),
+                      seqnames=Rle(gInterval$seq_name),
```

```
+                               strand=Rle(gInterval$strand),
+                               transcript=as.vector(getGffAttribute(
+                                  gInterval,"Parent")),
+                               gene=as.vector(getGffAttribute(
+                                  gInterval,"Name")),
+                               exon=as.vector(getGffAttribute(
+                                  gInterval,"ID")),
+                               seqlengths = chrSizes[
+                                  match(
+                                          unique(gInterval$seq_name),
+                                          names(chrSizes))]
+                               )
```

- **Q6**: section 2.3, page 14: using the *rtracklayer* package, specifically the `export.gff3` of `export.gff` or the `export` function as follow. The only difference in using these functions is that the number of argument you need to provide somewhat increases as the function becomes more generic. The three following examples have the same results.

```
> library(rtracklayer)
> load(system.file("data",
+                    "gAnnot.rda",
+                    package="RnaSeqTutorial"))
> export.gff3(gAnnot,con="annot.gff")
> export.gff(gAnnot,con="annot.gff",version="3")
> export(gAnnot,con="annot.gff",version="3")
```

- **Q7**: section 2.4, page 16: simply coerce the *Rle* class object into an integer.

```
> as.integer(cover$chr4)
```

- **Q8**: section 2.4, page 16: use the `runLength` and `runValue` functions

```
> runLength(cover$chr4)
> runValue(cover$chr4)
```

- **Q9**: section 2.4, page 17: The names are different for both objects. It is therefore essential to make sure that there are ordered in the same way to avoid unexpected results. In standard R, names are optional for lists; that is the default behavior, so do not expect otherwise from packages until you've tested it. Better safe than sorry.

```
> names(exon.range2)
> names(cover)
> match(names(exon.range2),names(cover))
```

- **Q10**: section 2.4, page 17: The main drawback is an edge effect; *i.e.*
  reads that are spanning the exon boundaries, although valid will not
  be taken entirely into account, only the proportion of these reads that
  cover the exon will. In addition, as shown below, exons present in
  different isoforms will be several times accounted for.

```
> exon.coverage <- unlist(exon.coverage)
> names(exon.coverage) <- exon.range2$exon
> head(
+     sort(
+         exon.coverage[!duplicated(names(exon.coverage))],
+         decreasing=TRUE))
```

- **Q11**: section 2.4, page 17:

```
> sel <- chromosome(aln) != "chrM"
> aln <- aln[sel]
> exon.counts <- countOverlaps(
+                         exon.range2,
+                         split(IRanges(
+                                     start=position(aln),
+                                     width=width(aln)),
+                             chromosome(aln))
+                         )
```

We might want to compare that result with the former one, stored in
the exon.coverage.

```
> plot(
+     unlist(exon.coverage),
+     unlist(exon.counts),
+     log="xy",
+     main="countOverlap vs. aggregate",
+     xlab="aggregate",
+     ylab="CountOverlap",
+     pch="+",col=6)
> abline(0,1,lty=2,col="orange")
> table(unlist(exon.coverage) - unlist(exon.counts))
```

```

As you can see, the difference is not striking.

- **Q12**: section 3.1, page 21: as in the example, but add the `output-Format` argument as follow:

```
> rnaSeq <- easyRNASeq(system.file(
+                                  "extdata",
+                                  package="RnaSeqTutorial"),
+                       organism="Dmelanogaster",
+                       chr.sizes=as.list(seqlengths(Dmelanogaster)),
+                       readLength=36L,
+                       annotationMethod="rda",
+                       annotationFile=system.file(
+                          "data",
+                          "gAnnot.rda",
+                          package="RnaSeqTutorial"),
+                       format="bam",
+                       count="exons",
+                       pattern="bam$",
+                       outputFormat="RNAseq")
> show(rnaSeq)
```

- **Q13**: section 3.1, page 21: use the `transcriptCounts`, `geneCounts` to generate the counts and `readCounts` to access the results.

```
> rnaSeq <- transcriptCounts(rnaSeq)
> head(readCounts(rnaSeq,'transcripts'))
```

Summarizing by transcript introduces some complexity in the data analysis, *i.e.* exons part of different isoforms introduce a bias in the counts. For that reason, it might be better to have a first look at the data, summarized by genes. This, however, requires to combine all the alternative exons and UTRs present for every gene into a "gene model"; *i.e.* overlapping exons are merged into "synthetic" ones. This is what is performed when the arguments "count" and "summarization" are set to "genes" and "geneModels", respectively. A caveat not addressed by this procedure are genes overlapping on the same or opposite strands. If this occurs a warning will be emitted. If the reads were summarized by "geneModels" and the "outputFormat" argument was set to "RNAseq", one can use the "geneModel" accessor on the obtained object to access the computed gene models. They are stored in

35

an *RangedData* object and can be modified to address the caveat previously mentioned. To be strict, one would remove every overlapping loci and conserve only the other ones. Such a modified annotation can then be saved and used for the next `easyRNASeq` run.

It is not possible yet to summarize by "geneModels" using the `geneCounts` function. A meaningful error message is thrown if the `geneCounts` is used for that purpose.

```
> rnaSeq<-geneCounts(rnaSeq,summarization='geneModels')
```

This behavior will be corrected in the next release. At the moment, this has to be done using the `easyRNASeq` function directly.

```
> rnaSeq2 <- easyRNASeq(system.file(
+                                   "extdata",
+                                   package="RnaSeqTutorial"),
+                     organism="Dmelanogaster",
+                     chr.sizes=as.list(seqlengths(Dmelanogaster)),
+                     readLength=36L,
+                     annotationMethod="rda",
+                     annotationFile=system.file(
+                        "data",
+                        "gAnnot.rda",
+                        package="RnaSeqTutorial"),
+                     format="bam",
+                     count="genes",
+                     summarization="geneModels",
+                     pattern="bam$",
+                     outputFormat="RNAseq")
> head(readCounts(rnaSeq2,'genes','geneModels'))
```

- **Q14**: section 4.1, page 22: you can directly use the `rnaSeq` and `rnaSeq2` objects

```
> RPKM(rnaSeq,from="transcripts")
> RPKM(rnaSeq2,from="geneModels")
```

- **Q15**: Explanations are in the *easyRNASeq* package vignette.

- **Q16**: Explanations are in the *easyRNASeq* package vignette.

- **Q17**: section 4.2, page 23: For the Illumina protocol, the barcode is read in a separate sequencing reaction. The barcode sequence is reported as a field of the *export* file, and when the data is loaded using the`withAll` argument, it is accessible through:

```
> alignData(alns)$multiplexIndex
```

To view all the possible fields, do:

```
> varLabels(alignData(alns))
```