rCube - RNA-Rates in R

Leonhard Wachutka[1,*], Carina Demel[2], Julien Gagneur[1]

[1] Department of Informatics, Technical University of Munich, Munich, Germany

[2] Max Planck Institute for biophysical Chemistry, Göttingen, Germany

[*] wachutka (at) in.tum.de

August 16, 2017

Abstract

rCube provides a framework for the estimation of RNA metabolism Rates in R (R^3). The *rCube* package complements the recently published transient transcriptome sequencing (TT-seq) protocol. It has been shown, that 4sU-labeling and subsequent purification of RNA allows to monitor local RNA synthesis. Therefore, the information from TT-seq/4sU-seq and total RNA-seq samples is used to model RNA synthesis, splicing, and degradation rates based on first-order kinetics. The *rCube* package works on count data and provides a series of functionalities to extract them from the desired features. It allows to extract junctions and constitutive exons from feature annotations, count reads from BAM-files, and normalize different samples against each other using a variety of different methods.

rCube version: 1.1.0

If you use rCube in published research, please cite:

B. Schwalb, M. Michel, B. Zacher, K. Frühauf, C. Demel, A. Tresch, J. Gageur, and P. Cramer:
TT-seq maps the human transient transcriptome.
Science (2016). doi:10.1126/science.aad9841 [1]

Contents

1 Background

As described in [2], 4sU-seq allows to monitor changes in the RNA metabolism. If cells are exposed to 4sU, they rapidly take up this Uridine analog and incorporate it into newly-synthesized RNAs. This way, newly-synthesized RNAs are labeled and can be extracted from the total RNA in the sample. The longer the labeling time, e.i. the time from 4sU addition to harvesting the cells, the bigger is the proportion of labeled RNAs among all RNAs. Labeling with 4sU has been successfully combined with deep sequencing in various 4sU-seq and TT-seq studies ([1, 2, 3]). In order to quantify gene expression levels from sequencing experiments, usually read counts per gene are used to compare the expression between genes and samples.

Variations in read counts among different samples can have multiple reasons. One source of variation is the sequencing depth. Even replicates from the same experimental condition may exhibit different read counts based on how deep the samples were sequenced. These variations (up to biological variations) can be overcome by normalization to sequencing depth. In a typical RNA-seq experiment, one wants to compare different samples under different experimental conditions. After extracting the RNA from the cells, the same starting material is used for the library prepartion. Therefore the information is lost, if cells from different samples were expressing different amounts of RNA (both for individual genes or on a global scale). Artificial RNA spike-in sequences can be used to adjust for global sequencing variations between samples. Adding the same volumes of spike-ins to a defined number of cells can help to resolve this problem, as they are subject to the same technical biases than natural RNAs, but their read counts should not be influenced by biological processes. In the case of TT-seq/4sU-seq, we additionally want to rescale 4sU-labeled and total RNA-seq samples, so that the ratio of labeled RNA to total RNA read counts reflects the ratio of labeled RNA to total RNA amounts in the cell. This can be achieved by labeling some of the spike-ins with 4sU during the *in vitro* transcription. Then it is also possible to quantify the amount of unlabeled spike-ins (RNAs) that is not lost during labeled RNA purification, the so-called cross-contamination.

With the *rCube* package we propose two different methods to extract normalization parameters from spike-in read counts. The normalization based on artifical spike-ins and subsequent estimation of synthesis and degradation rates has been successfully implemented and applied in different studies: In human K562 cells, we investigated synthesis rates and half-lives of different RNA species under steady-state conditions [1]. In another study, we investigated the change of RNA synthesis immediately after T-cell stimulation [3].The sensitivity of TT-seq allowed us to monitor rapid changes in transcription from enhancers and promoters during the immediate response of T-cells to ionomycin and phorbol 12-myristate 13-acetate (PMA).

The *rCube* package offers a framework for two different experimental setups. First, a labeling time series can be used to extract robust synthesis, degradation, and splicing rates based on junctions. Hereby, a set of TT-seq samples with different labeling times and complementary total RNA-seq samples are used. A similar approach has been used for 4tU-seq labeling in *S. pombe* [4]. Second, synthesis and degradation rates for genes/exons/introns can be estimated from pairs

of TT-seq and total RNA-seq samples, as published in [1, 3]. This approach is useful to calculate conditional synthesis and degradation rates for different samples.

Synthesis rates correspond to the transcription rate at a specific genomic locus. Synthesis and degradation rates together determine the steady-state levels of mature RNA. The development of TT-seq led to the possiblity to study local synthesis and degradation rates at the level of individual phosphodiester bonds, as TT-seq provides a uniform read coverage even across long genes. The synthesis of individual phosphodiester bonds corresponds to the transcription rate (synthesis rate) of the corresponding bond, independent of its location within the gene. Degradation of phosphodiester bonds within exons or introns reflects exon and intron degradation rates, respectively. Degradation rates at donor or acceptor sites at junctions, however, correspond to the cleavage rates of donor or acceptor sites, respectively. All these rates can be estimated by reads mapping exclusively to exons, introns, or spanning exon-intron or intron-exon junctions, respectively. Reads mapping to exon-exon junctions, representing already spliced mRNA, can also be used to calculate synthesis and degradation rates for the corresponding junctions. These rates, however, translate to the junction formation and junction degradation rates. For more details, see also our recent review [5]. Taken together, counting reads on different regions of a gene allows to estimate phosphodieser bond specific RNA metabolism rates with *rCube*.

2 Getting started

This vignette provides a pipeline how to estimate RNA metabolism rates from TT-seq and RNA-seq data sets, starting from BAM. You will learn how to estimate sample specific size factors and cross-contamination rates from spike-in counts. These values can be used to normalize gene expression values obtained by RNA-Seq and thus estimate gene-specific synthesis and degradation rates (section 3). By extracting reads spanning junctions, splicing times can be estimated. For more robust estimation, multiple samples with different labeling times are taken into account (section 4).

Before starting, the package must be loaded by:

```
library("rCube")
```

2.1 Input and output data

The class used by *rCube* to store input data is called *rCubeExperiment*, which depends on *SummerizedExperiment*. An *rCubeExperiment* consists of

- an annotation in the form of a *GRanges* object,
- a sample information table in the form of a *data.frame*, and
- an assay, here a matrix containing read counts for all annotated regions in the corresponding samples.

Objects of this class are used as input for the whole workflow, starting from read counting, normalization, dispersion estimation, to rate estimation. Most of these steps return an updated and extended *rCubeExperiment* object.

The `rowRanges` of the *rCubeExperiment* is a *GRanges* annotation of features, for which RNA rates should be estimated. If you have only a gtf version of your annotation, you can import it via

```
library(rtracklayer)
granges <-import(gtffile)
```

In case you want to work on constitutive features, *rCube* provides a function to extract them from previously imported *GRanges* objects (see also section 3.1). If you want to work on junctions, *rCube* comes with a function to extract junctions *de novo* from the BAM files you are working on (see also section 4.2).

Experimental sample information in the `colData` should contain the following columns:

- sample: A unique sample name
- LT: A factor, stating if sample was labeled ('L' for TT-seq or 4sU-seq), or total RNA ('T' for total RNA-seq) was sequenced
- condition: A factor which distinuished different experimental conditions (but not the sequencing type)

- labelingTime: A numeric value indicating the labeling time with 4sU for each sample
- replicate: A factor giving replicate information (if no replicates were used, please use the same value for all samples)

This information can be either provided by a manually set up *data.frame* or this information can be extracted from the BAM-file names, if they follow the following convention: `{condition}_{L|T}_{labelingTime}_{replicate}.bam`

The output of *rCube* is stored in objects of type *rCubeRates*, which also depends of *SummarizedExperiment*. The assay here is a matrix with estimated synthesis and degradation rates for the respective features in `rowRanges`. The `colData` object contains the information about the rate type (synthesis, degradation, half-life), and the replicate for which the corresponding rate was estimated.

Please see also examples in sections 3 and 4.

2.2 Normalization functions

We provide different normalization schemes that can chosen in the `estimateSizeFactors` functions by the parameter `method`, namely: 'spikeinGLM', 'spikeinMean', and 'spikeinMedian'. All if these methods rely on the spike-in read counts.

In the 'spikeinGLM' method, we fit a generalized linear model (GLM) of the Negative Binomial family with a log link function. The response of the GLM are the observed spike-in counts, and the terms that specify the linear predictor of the response are comprised of:

- a sample specific factor (that reflects the sample specific size factor),
- a labeled sample specific factor (that reflects the control for cross contamination (only estimated for unlabeled spike-ins in labeled samples)), and
- a spike-in specific factor to allow for some spike-in specific variation e.g. due to sequence biases.

Additionally, the length of each spike-in is used as an offset, i.e. a known slope for the covariate.

The 'spikeinMean' and 'spikeinMedian' normalization methods estimate size factors based on the read distributions of labeled spike-ins among the different samples. They should give highly similar results unless one spike-in shows an outlier behaviour. For an estimate of cross-contamination of unlabeled RNA in labeled samples, the ratio of read counts for unlabeled spike-ins to read counts for labeled spike-ins is used.

In general, all three methods give highly correlated size factors.

2.3 Fitting functions

The estimation of RNA metabolism rates is wrapped in the `estimateRateByFirstOrderKinetics` function, where the parameter `method` allows to choose the type of fitting to be performed: If you want to compare RNA rates in different experimental setups, you might want to choose "single", then single synthesis and degradation rates are estimated for each condition. .If you select "series" the algorithm assumes you provide a labeling time series experiment and are working on junctions.

Both methods perform maximum-likelihood (ML) estimations, assuming negative binomial read distributions, with random initialization.

The estimation of these rates relies on a random initialization. To make the results more robust, it is possible to run multiple iterations of the algorithm with random initialization and the median of the resulting rates is returned. The default number of iterations is 3. To change this value, the `numberOfInterations` in the `elementMetadata` of the *rCubeExperiment* object has to be set:

```
elementMetadata(featureCounts)$numberOfIterations <- 10
```

Multiple replicates for the same condition can be used for a joint estimation. The user has to specify for which replicate or combination of replicates the results should be estimated. Therefore, the `replicate` parameter is a vector of all combinations that should be evaluated. For the joint estimation for multiple replicates, these have to be given as a string

separeted by a ":". If you want to estimate results for replicates 1 and 2 individually, as well as jointly, set the `replicate` parameter of the `estimateRateByFirstOrderKinetics` to `c(1, 2, "1:2")`:

```
estimateRateByFirstOrderKinetics(featureCounts,
                                 replicate=c(1, 2, "1:2"),
                                 method=c("single", "series"),
                                 BPPARAM=NULL)
```

2.4 Example data

The `inst/extdata` of the *rCube* package provides two example data sets that should illustrate the two different functionalities of *rCube*:

The first example data set, "TcellActivation", contains bam files from resting and activated Jurkat T-cells for TT-seq and RNA-seq samples. The bamfiles are restricted to the FOS gene (chr14:75278000-75283000) and the artifical spike-ins, subsampled to to 5% of the original read numbers to reduce file size. The full data sets are published in [3]. This example data is used to demonstrate the spike-in normalization method, and the estimation of synthesis and degradation rates for individual 4sU-labeled (TT-seq) and total RNA-seq pairs. A detailed working example is given in section 3.

The second data, "TimeSeriesExample", set is a labeling time series, where RNA has been labeled with 4sU for different time periods before RNA extraction. The bamfiles are restricted to the MYNN gene (chr3:169769500-169789800) and the artifical spike-ins, subsampled to to 20% of the original read numbers to reduce file size. An example workflow is provided in section 4.

3 Conditional synthesis and degradation rates for T-cell activation data

Example data sets from a T-cell activation experiment are stored in the `inst/extdata/TcellActivation` folder of the *rCube* package. In this part of the vignette, we will demonstrate

- how reads can be counted for (constitutive) exons and spike-ins,
- how the samples are normalized against each other based on the spike-in read counts,
- how synthesis and degradation rates are obtained for (constitutive) exons
- how gene-specific rates are obtained from exon-specific rates

Please note, the data set is downsampled (to yield approx 5% of the originial read numbers) due to size reasons, that's why fitting results are less acurate than when applied to multiple genes from deep-sequenced samples.

3.1 Gene model

working on exons/introns/genes... The estimation of synthesis and degradation rates with the *rCube* package relies on read counts. Dependent on the features, for which read counts are provided, the rates can reflect synthesis rates of exons, introns, or full genes. Especially degradation rates may differ between exons and introns. Therefore, the features, which should be used to estimate synthesis and degradation rates, and for which read counts are provided or should be obtained, need to be provided as a *GRanges* object.

Due to numerous transcript isoforms per gene, and the arising problem that for some bases their exonic or intronic nature cannot be unambigously identified, we propose to use the model of constitutive exons/introns from [6]. Hereby, all bases, that belong to an exon/intron in all (annotated) transcript isoforms of the same gene, are thought to be part of "constitutive" exons/introns. Figure 1 shows an example annotation from the FOS gene (not comprehensive) to illustrate how constitutive exons can be extracted from an exon annotation with the following code:

```
data("exampleExons")
exampleExons
```

```
## GRanges object with 7 ranges and 3 metadata columns:
##       seqnames              ranges strand |     type           gene_id
##          <Rle>           <IRanges>  <Rle> | <factor>       <character>
##   [1]    chr14 [75278774, 75279123]      + |     exon ENSG00000170345.9
##   [2]    chr14 [75278828, 75279123]      + |     exon ENSG00000170345.9
##   [3]    chr14 [75279877, 75280128]      + |     exon ENSG00000170345.9
##   [4]    chr14 [75279877, 75280128]      + |     exon ENSG00000170345.9
##   [5]    chr14 [75280560, 75280667]      + |     exon ENSG00000170345.9
##   [6]    chr14 [75280783, 75282230]      + |     exon ENSG00000170345.9
##   [7]    chr14 [75280783, 75281636]      + |     exon ENSG00000170345.9
##           transcript_id
##             <character>
##   [1] ENST00000303562.8
##   [2] ENST00000535987.5
##   [3] ENST00000303562.8
##   [4] ENST00000535987.5
##   [5] ENST00000303562.8
##   [6] ENST00000303562.8
##   [7] ENST00000535987.5
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

```
constitutiveExons <- createConstitutiveFeaturesGRangesFromGRanges(exampleExons,
                                                                  BPPARAM=NULL,
                                                                  ncores=1)
constitutiveExons
```

```
## GRanges object with 3 ranges and 2 metadata columns:
##           seqnames              ranges strand |                type
##              <Rle>           <IRanges>  <Rle> |            <factor>
##   CF00001    chr14 [75278828, 75279123]      + | constitutive feature
##   CF00002    chr14 [75279877, 75280128]      + | constitutive feature
##   CF00003    chr14 [75280783, 75281636]      + | constitutive feature
##                  gene_id
##                <character>
##   CF00001 ENSG00000170345.9
##   CF00002 ENSG00000170345.9
##   CF00003 ENSG00000170345.9
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Please note, for the subsequent workflow it is not necessary to extract constitutive exons. Any kind of *GRanges* object can be used as feature annotation (e.g. full genes, exons, introns, ...).

3.2 Experimental Design

The *rCube* package works on *rCubeExperiment* containers, that rely on the *SummerizedExperiment* class.

The `rowRanges` of the *rCubeExperiment* is a *GRanges* annotation of features, for which RNA rates should be estimated. Experimental sample information can be either provided by a experimental design matrix or this information can be extracted from the BAM-file names (when they fulfil the required structure).

We first look at the experimental design file `experimentalDesign.txt`, that can be imported as a *data.frame*.

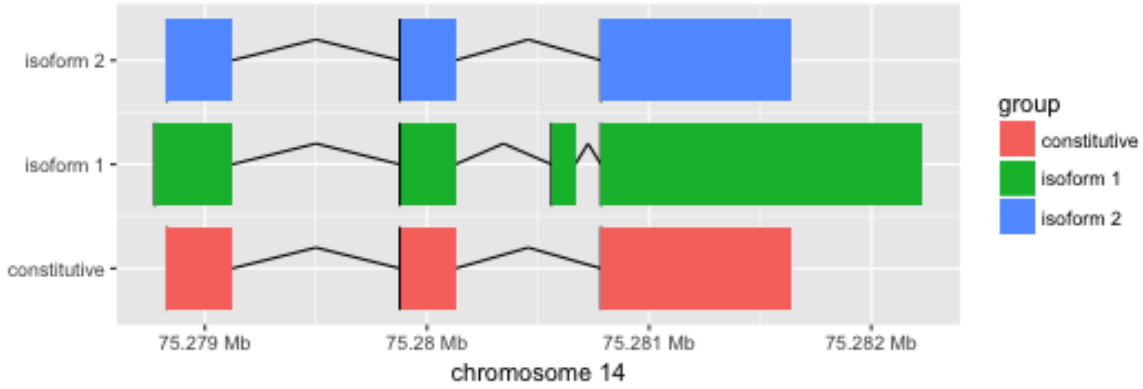

Figure 1: **Illustration of two transcript isoforms for the FOS gene and the resulting constitutive exons.**

```
folder <- system.file("extdata/TcellActivation", package='rCube')
expDesign <- read.delim(file.path(folder, "experimentalDesign.txt"))
expDesign
```

```
##            sample condition LT labelingTime replicate            filename
## 1   Resting_L_5_1   Resting  L            5         1   Resting_L_5_1.bam
## 2   Resting_L_5_2   Resting  L            5         2   Resting_L_5_2.bam
## 3   Resting_T_5_1   Resting  T            5         1   Resting_T_5_1.bam
## 4   Resting_T_5_2   Resting  T            5         2   Resting_T_5_2.bam
## 5 Activated_L_5_1 Activated  L            5         1 Activated_L_5_1.bam
## 6 Activated_L_5_2 Activated  L            5         2 Activated_L_5_2.bam
## 7 Activated_T_5_1 Activated  T            5         1 Activated_T_5_1.bam
## 8 Activated_T_5_2 Activated  T            5         2 Activated_T_5_2.bam
```

Together with the feature annotation, for which we want to estimate synthesis and degradation rates, we can construct the *rCubeExperiment*:

```
exonCounts <- setupExperiment(constitutiveExons, designMatrix=expDesign, files=NULL)
class(exonCounts)
```

```
## [1] "rCubeExperiment"
## attr(,"package")
## [1] "rCube"
```

Alternatively, the experimental design matrix can be constructed from the bam file names internally (see section 2.1).

```
bamfiles <- list.files(folder, pattern="*.bam$", full.names=TRUE)
basename(bamfiles)
```

```
## [1] "Activated_L_5_1.bam" "Activated_L_5_2.bam" "Activated_T_5_1.bam"
## [4] "Activated_T_5_2.bam" "Resting_L_5_1.bam"   "Resting_L_5_2.bam"
## [7] "Resting_T_5_1.bam"   "Resting_T_5_2.bam"
```

```
exonCounts <- setupExperiment(constitutiveExons, designMatrix=NULL, files=bamfiles)
```

The individual information from the *rCubeExperiment* can be assessed by:

```
# feature information
rowRanges(exonCounts)

# sample information
colData(exonCounts)
```

```
# read counts
assay(exonCounts)
```

The resulting *rCubeExperiment* object can now be used to count reads.

3.3 Counting

All RNA rate estimations of this package rely on read counts. These can be either provided as count matrices, or read counts can be obtained from BAM files using the *rCube* pipeline.

For read counting, we use the `readGAlignmentPairs` in a parallel fashion:

```
assay(exonCounts)
```

```
##         Activated_L_5_1 Activated_L_5_2 Activated_T_5_1 Activated_T_5_2
## CF00001              NA              NA              NA              NA
## CF00002              NA              NA              NA              NA
## CF00003              NA              NA              NA              NA
##         Resting_L_5_1 Resting_L_5_2 Resting_T_5_1 Resting_T_5_2
## CF00001            NA            NA            NA            NA
## CF00002            NA            NA            NA            NA
## CF00003            NA            NA            NA            NA
```

```
exonCounts <- countFeatures(exonCounts)

assay(exonCounts)
```

```
##         Activated_L_5_1 Activated_L_5_2 Activated_T_5_1 Activated_T_5_2
## CF00001              74             123               6               6
## CF00002             112             195               4               9
## CF00003             481             610              17              23
##         Resting_L_5_1 Resting_L_5_2 Resting_T_5_1 Resting_T_5_2
## CF00001             0             4             4             2
## CF00002             4             6             2             3
## CF00003            15            17             1             1
```

In case you already have counted reads on your features of interest, count matrices can be assigned to the correctly formatted, empty *rCubeExperiment* object.

3.4 Spike-ins

The artifical spike-in annotations and labeling information can be loaded via:

```
data("spikeins")
data("spikeinLabeling")
spikeinLengths <- width(spikeins)
```

3.5 Spike-in design

An empty *rCubeExperiment* for the artificial spike-ins additionally requires information about the length and the labeling status of each spikein, and can be constructed as follows:

```
spikeinCounts <- setupExperimentSpikeins(rows=spikeins,
                                         designMatrix=expDesign,
```

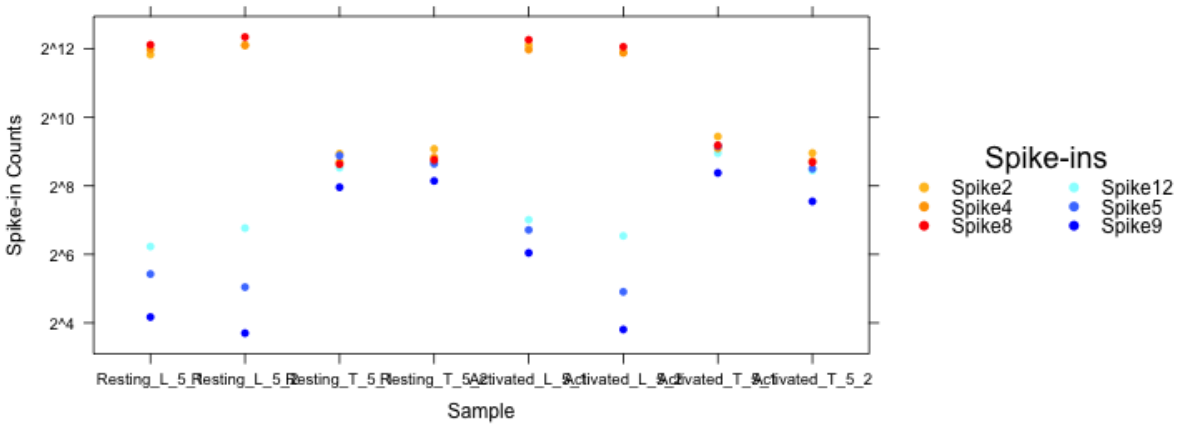

Figure 2: **Spike-in read counts in different samples.** The code that creates this figure is shown in the code chunk.

```
                                    length=spikeinLengths,
                                    labelingState=spikeinLabeling)
```

3.6 Spike-in counting

```
spikeinCounts <- countSpikeins(spikeinCounts)
assay(spikeinCounts)
```

```
##         Resting_L_5_1 Resting_L_5_2 Resting_T_5_1 Resting_T_5_2 Activated_L_5_1
## Spike2           3648          4416           492           541            4416
## Spike12            75           109           369           408             129
## Spike4           4060          4399           414           456            4031
## Spike5             43            33           472           401             105
## Spike8           4455          5205           397           432            4927
## Spike9             18            13           249           283              66
##         Activated_L_5_2 Activated_T_5_1 Activated_T_5_2
## Spike2             3791             695             498
## Spike12              93             496             349
## Spike4             3800             548             425
## Spike5               30             573             363
## Spike8             4272             582             412
## Spike9               14             333             187
```

The distribution of 4sU-labeled and unlabeld spike-ins among different samples can be illustrated by the function `plotSpikeinCountsVsSample`. Figure 2 shows the read counts of spike-ins in the TcellActivation example data set.

```
plotSpikeinCountsVsSample(spikeinCounts)
```

Naturally, labeled spike-ins (Spike2, Spike4, Spike8) should be enriched in labeled samples ("L"), whereas unlabeled spike-ins (Spike5, Spike9, Spike12) should be depleted from these samples. In total RNA-seq samples ("T"), all spike-ins should be present to a similar extend.

3.7 Size factor based on spike-ins

Using the previously determined spike-in read counts, we can now estimate sample specific size factors and cross-contamination values.

```
exonCounts <- estimateSizeFactors(exonCounts, spikeinCounts, method="spikeinGLM")
colnames(colData(exonCounts))
```

```
## [1] "filename"          "sample"             "condition"
## [4] "LT"                "labelingTime"       "replicate"
## [7] "sizeFactor"        "crossContamination"
```

```
exonCounts$sizeFactor
```

```
##   Resting_L_5_1   Resting_L_5_2   Resting_T_5_1   Resting_T_5_2 Activated_L_5_1
##          0.9121          1.0495          0.1024          0.1079          1.0000
## Activated_L_5_2 Activated_T_5_1 Activated_T_5_2
##          0.8884          0.1376          0.0935
```

```
exonCounts$crossContamination
```

```
##   Resting_L_5_1   Resting_L_5_2   Resting_T_5_1   Resting_T_5_2 Activated_L_5_1
##          0.0137          0.0130          1.0000          1.0000          0.0288
## Activated_L_5_2 Activated_T_5_1 Activated_T_5_2
##          0.0138          1.0000          1.0000
```

Note, the cross-contamination value for all total RNA-seq samples is 1, as 100% of the unlabeled RNAs are supposed to be in the sample. Additional fitting results are stored in the `metadata` of the resulting *rCubeExperiment* object.

```
metadata(exonCounts)
```

3.8 Providing gene-wise dispersion estimates

Usually, read counts in different RNA-seq samples underly fluctuations due to biological or technical variances. To take these fluctuations into account, we estimate each gene's dispersion. For each gene, a single dispersion estimate for all 4sU-Seq samples and for all Total RNA-Seq samples is needed. The wrapper function `estimateSizeDispersions` offers different methods to estimate a gene's dispersion Here, we can use the method provided in the DESeq2 package [7]. The DESeq algorithm is applied all genes, while separating the count table according to the RNA-Seq protocol (labeled or total RNA). It is possible to choose between all provided DESeq dispersion estimates, namely the genewise maximum likelihood dispersion estimate ("dispGeneEst"), the smooth curve fitted through the gene-wise disperion estimates ("dispFit") and the genewise dispersion estimates shrunken towards the fitted curve ("dispMAP", default). The input is an *rCubeExperiment* object with read counts for the features of interest. The function returns an updated *rCubeExperiment* object with two additional columns in the `rowRanges`, namely `dispersion_L` and `dispersion_T`.

```
exonCounts <- estimateSizeDispersions(exonCounts, method='DESeqDispGeneEst')
rowRanges(exonCounts)
```

```
## GRanges object with 3 ranges and 3 metadata columns:
##           seqnames              ranges strand |        group dispersion_L
##              <Rle>           <IRanges>  <Rle> |     <factor>    <numeric>
##   CF00001    chr14 [75278828, 75279123]      + | constitutive        1e-08
##   CF00002    chr14 [75279877, 75280128]      + | constitutive        1e-08
##   CF00003    chr14 [75280783, 75281636]      + | constitutive        1e-08
##           dispersion_T
##              <numeric>
##   CF00001        1e-08
##   CF00002        1e-08
##   CF00003        1e-08
```

```
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

3.9 Exon-specific synthesis and degradation rate estimates

After estimating size factor and cross-contamination rates per sample (see Section 3.7) and extracting feature-specific dispersion estimates (see Section 3.8), we can now estimate RNA synthesis and degradation rate for each feature and condition individually.

Let's set the number of iterations to 7 to yield more robust results:

```
elementMetadata(exonCounts)$numberOfInterations <- 7
```

In the following example, we will obtain individual results for replicate 1 and 2 and also results for a joint estimation.

```
rates <- estimateRateByFirstOrderKinetics(exonCounts,
                                          replicate=c(1, 2, "1:2"),
                                          method='single',
                                          BPPARAM=BiocParallel::MulticoreParam(1))
rates
```

```
## class: rCubeRates
## dim: 3 30
## metadata(0):
## assays(1): rates
## rownames(3): CF00001 CF00002 CF00003
## rowData names(4): group dispersion_L dispersion_T numberOfInterations
## colnames(30): Activated_synthesis_1 Resting_synthesis_1 ...
##   Activated_unlabeled.amount_1:2 Resting_unlabeled.amount_1:2
## colData names(4): condition rate replicate sample
```

3.10 Plotting and summarizing the results

The returned `rates` object is of the type *rCubeRates*, which also extends *SummarizedExperiment*. The `rowRanges` should be identical to the input data. The columns of the resulting matrix correspond to synthesis and degradation rates, for all indicated replicates and replicate combinations.

The resulting rates can be validated by looking at the correlation of measured read counts and fitted read counts.

```
plotFittedCounts(exonCounts, rates)
```

If rates were estimated independently for different replicates or replicate combinations, the correlation of the results can be plotted with:

```
plotResultsReplicates(rates)
```

If you want to get an average estimate for "top level features", e.g. full genes, from the exon-rates, you can run the `summarizeRates` function:

```
topLevelFeature <- GRanges(seqnames="chr14",
                           ranges=IRanges(start=75278774, end=75281636),
                           strand="+")
topLevelFeaturesRates <- summarizeRates(rates, topLevelFeature, by='mean')
```

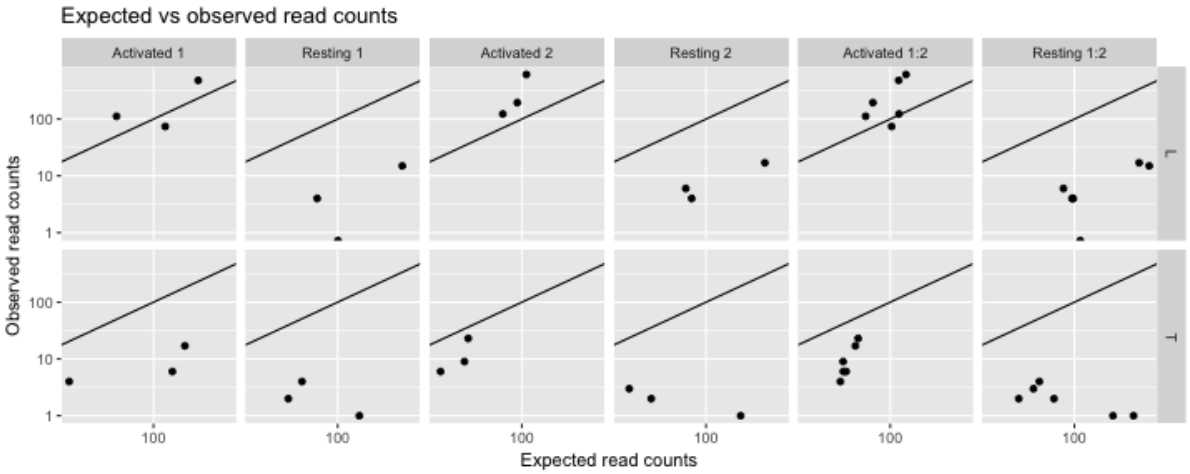

Figure 3: **Correlation of observed vs expected counts (based on synthesis and degradation rate estimates).** The code that creates this figure is shown in the code chunk.

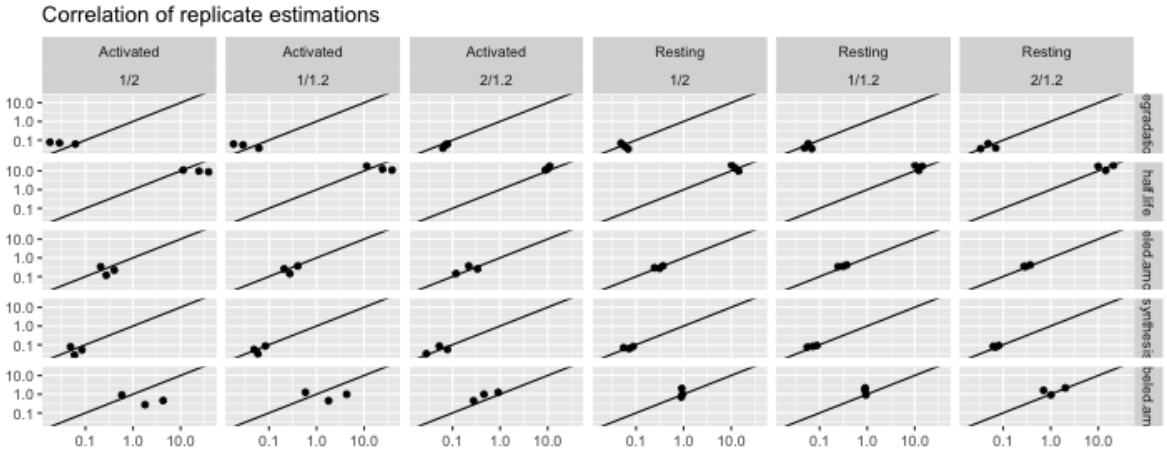

Figure 4: **Estimation results from independent estimations for different replicates/combinations of replicates.** The code that creates this figure is shown in the code chunk.

3.11 Synthesis rates in transcripts per cell per minute

So far, our estimated synthesis rates are on an arbitrary scale. Using the knowledge of initially used spike-in amounts and cell numbers for the experiment, the synthesis rates can be rescaled to the scale of transcripts per cell per minute. In the TcellActivation example, for each spikein 25ng of spike-ins were added to each sample constisting of RNA from 50 million cells. As reads were downsampled to yield only 5% of the originial reads, we have to account for that in this example. The synthesis rate adjustment can then achieved with the following function:

```
data("spikeinGenome")
adjustedSynthesis <- adjustSynthesisRate(spikeinCounts=spikeinCounts,
                                         spikeinGenome=spikeinGenome,
                                         featureRates=rates,
                                         spikeinAmount=25*0.05,
                                         cellNumber=5*10^7)
range(adjustedSynthesis)
```

```
## [1] 1.58 4.97
```

The result shows, that on average 1-5 transcripts are produced per cell per minute, which agrees well with previous estimates [8].

4 Labeling time series

Example data sets from a T-cell activation experiment are stored in the `inst/extdata/TimeSeriesExample` folder of the *rCube* package. In this part of the vignette, we will demonstrate

- how junction annotations can be extracted from bam files
- how
- how
- how

4.1 Read classification

4.2 Gene model

(exon, into and junctions) by gff de novo + gff

Junctions can also be extracted from the bam files, based on split reads. The following code shows how to extract all junctions, that are supported by at least 5 reads:

```
bamfiles <- list.files(system.file("extdata/TimeSeriesExample", package='rCube'),
                       pattern=".bam$", full.names=TRUE)
junctions <- createJunctionGRangesFromBam(bamfiles, support=5, ncores=1,
                                          BPPARAM=BiocParallel::MulticoreParam(1))
```

The junction annotation now contains entries for the three different types: "donor" (exon-intron boundary), "acceptor" (intron-exon boundary), and "junction" (intron):

```
head(junctions)
```

```
## GRanges object with 6 ranges and 1 metadata column:
##       seqnames                ranges strand |         typ
##          <Rle>             <IRanges>  <Rle> | <character>
##   [1]     chr3 [169773462, 169773463]      + |       donor
##   [2]     chr3 [169773463, 169774030]      + |    junction
##   [3]     chr3 [169773463, 169774264]      + |    junction
##   [4]     chr3 [169774030, 169774031]      + |    acceptor
##   [5]     chr3 [169774097, 169774098]      + |       donor
##   [6]     chr3 [169774098, 169774264]      + |    junction
##   -------
##   seqinfo: 200 sequences from an unspecified genome; no seqlengths
```

4.3 Experimental Design

We first look at the experimental design file `experimentalDesign.txt`, that can be imported as a *data.frame*.

```
folder <- system.file("extdata/TimeSeriesExample", package='rCube')
expDesign <- read.delim(file.path(folder, "experimentalDesign.txt"))
head(expDesign)
```

```
##              sample   condition LT labelingTime replicate             filename
## 1 TimeSeries_L_05_1 steadyState  L            5         1 TimeSeries_L_05_1.bam
## 2 TimeSeries_L_05_2 steadyState  L            5         2 TimeSeries_L_05_2.bam
## 3 TimeSeries_L_10_1 steadyState  L           10         1 TimeSeries_L_10_1.bam
## 4 TimeSeries_L_10_2 steadyState  L           10         2 TimeSeries_L_10_2.bam
## 5 TimeSeries_L_30_1 steadyState  L           30         1 TimeSeries_L_30_1.bam
```

```
## 6 TimeSeries_L_30_2 steadyState  L           30         2 TimeSeries_L_30_2.bam
```

Together with the junction annotation, we can now construct the *rCubeExperiment* container, which can then be used for counting and rate estimation:

```
junctionCounts <- setupExperiment(junctions, files=bamfiles)
class(junctionCounts)
```

```
## [1] "rCubeExperiment"
## attr(,"package")
## [1] "rCube"
```

4.4 Counting

Read counting for junctions includes three different types of reads:

- reads spanning exon-exon junctions, thereby representing already spliced mRNA
- reads spanning exon-intron junctions, thereby representing pre-mRNA
- reads spanning intron-exon junctions, thereby representing pre-mRNA

Read counts, that support each type of junction, can be obtained with the following code:

```
junctionCounts <- countJunctions(junctionCounts, BPPARAM=BiocParallel::MulticoreParam(1))
```

4.5 Spike-ins

The spike-ins used for the TimeSeries example are the same as in the TcellActivation example. Therefore we can use the same example data here:

```
data("spikeins")
data("spikeinLabeling")
spikeinLengths <- width(spikeins)
```

4.6 Spike-in design

An empty *rCubeExperiment* for the artificial spike-ins additionally requires information about the length and the labeling status of each spikein, and can be constructed as follows:

```
spikeinCounts <- setupExperimentSpikeins(rows=spikeins,
                                         designMatrix=expDesign,
                                         length=spikeinLengths,
                                         labelingState=spikeinLabeling)
```

Note, for counting, either your working directory should be set to the bamfile-folder, or you have to give absolute file paths:

4.7 Spike-in counting

Next, we can count reads that map to the spike-in sequences:

```
spikeinCounts <- countSpikeins(spikeinCounts)
assay(spikeinCounts)
```

```
##         TimeSeries_L_05_1 TimeSeries_L_05_2 TimeSeries_L_10_1 TimeSeries_L_10_2
## Spike2              26050             16427             16332             15916
## Spike12               445               300               223               268
## Spike4              36873             21667             26109             25032
## Spike5                 60                88                58                51
## Spike8               9774              7258              7888              7676
## Spike9                 14                25                15                12
##         TimeSeries_L_30_1 TimeSeries_L_30_2 TimeSeries_T_05_1 TimeSeries_T_05_2
## Spike2               5801              7532              3228              3428
## Spike12                81               103              2289              2335
## Spike4              11705             15186              3711              4027
## Spike5                 18                25              2969              3258
## Spike8               5947              6955              1352              1671
## Spike9                  2                 8              1303              1427
##         TimeSeries_T_10_1 TimeSeries_T_10_2 TimeSeries_T_30_1 TimeSeries_T_30_2
## Spike2               2886              3161              2982              3171
## Spike12              1781              1991              1494              2085
## Spike4               3231              3772              3584              4001
## Spike5               2587              2793              2921              3318
## Spike8               1351              1369              1595              1789
## Spike9               1256              1233              1434              1527
```

4.8 Size factor based on spike-in

Using the spike-in read counts, we can estimate a size factor that will be used later to adjust the junction read counts:

```
junctionCounts <- estimateSizeFactors(junctionCounts, spikeinCounts, method="spikeinMean")
colnames(colData(junctionCounts))
```

```
## [1] "filename"          "sample"            "condition"
## [4] "LT"                "labelingTime"      "replicate"
## [7] "sizeFactor"        "crossContamination"
```

```
junctionCounts$sizeFactor
```

```
## TimeSeries_L_05_1 TimeSeries_L_05_2 TimeSeries_L_10_1 TimeSeries_L_10_2
##            0.2182            0.1410            0.1538            0.1490
## TimeSeries_L_30_1 TimeSeries_L_30_2 TimeSeries_T_05_1 TimeSeries_T_05_2
##            0.0789            0.0978            0.0261            0.0293
## TimeSeries_T_10_1 TimeSeries_T_10_2 TimeSeries_T_30_1 TimeSeries_T_30_2
##            0.0240            0.0261            0.0265            0.0292
```

4.9 Estimate dispersion

```
junctionCounts <- estimateSizeDispersions(junctionCounts, method='Replicate')
rowRanges(junctionCounts)
```

```
## GRanges object with 123 ranges and 2 metadata columns:
##         seqnames                ranges strand |         typ dispersion
##            <Rle>             <IRanges>  <Rle> | <character>  <numeric>
##     [1]     chr3 [169773462, 169773463]      + |       donor         40
##     [2]     chr3 [169773463, 169774030]      + |    junction         40
##     [3]     chr3 [169773463, 169774264]      + |    junction         40
##     [4]     chr3 [169774030, 169774031]      + |    acceptor         40
```

```
##     [5]     chr3 [169774097, 169774098]      + |       donor         40
##     ...      ...                     ...    ... .         ...        ...
##   [119]    chrS9           [ 990,  991]      + |    acceptor         40
##   [120]    chrS9           [ 991,  992]      + |    acceptor         40
##   [121]    chrS9           [ 995,  996]      + |    acceptor         40
##   [122]    chrS9           [ 998,  999]      + |    acceptor         40
##   [123]    chrS9           [1001, 1002]      + |    acceptor         40
##   -------
##   seqinfo: 200 sequences from an unspecified genome; no seqlengths
```

4.10 Fit the rates

In deep-sequenced samples, you might want to filter your data, to only get more repliable estimates from highly covered junctions. This is not reasonable for our test set due to the downsampling. In general, filtering could be obtained via:

```
junctionCounts = junctionCounts[rowSums(assay(junctionCounts))>100]
```

```
junctionRates = estimateRateByFirstOrderKinetics(junctionCounts,
                                                 replicate=c(1,2,'1:2'),
                                                 method='series')
```

```
##
  |
  |                                                                      |   0%
  |
  |============                                                          |  17%
  |
  |========================                                              |  33%
  |
  |====================================                                  |  50%
  |
  |================================================                      |  67%
  |
  |============================================================          |  83%
  |
  |======================================================================| 100%
##
##
  |
  |                                                                      |   0%
  |
  |============                                                          |  17%
  |
  |========================                                              |  33%
  |
  |====================================                                  |  50%
  |
  |================================================                      |  67%
  |
  |============================================================          |  83%
  |
  |======================================================================| 100%
##
##
```

```
  |
  |                                                                      |   0%
  |
  |============                                                          |  17%
  |
  |========================                                              |  33%
  |
  |====================================                                  |  50%
  |
  |================================================                      |  67%
  |
  |============================================================          |  83%
  |
  |======================================================================| 100%
```

Now let's summarize the results obtained from the individual junctions of the MYNN gene:

```
topLevelFeaturesJunctions <- GRanges(seqnames="chr3",
                          ranges=IRanges(start=169769500, end=169789800),
                          strand="+")
topLevelFeaturesRates <- summarizeRates(junctionRates, topLevelFeaturesJunctions, by='median')

assay(topLevelFeaturesRates)
```

```
##      TimeSeries_synthesis_1 TimeSeries_degradation_1 TimeSeries_synthesis_2
## [1,]                  0.787                    1.56                   1.17
##      TimeSeries_degradation_2 TimeSeries_synthesis_1:2
## [1,]                      1.3                    1.31
##      TimeSeries_degradation_1:2
## [1,]                       1.67
```

5 Session Information

This vignette was generated using the following package versions:

```
sessionInfo()
```

```
## R version 3.4.1 (2017-06-30)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: OS X El Capitan 10.11.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel  stats4    stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] ggbio_1.25.3               ggplot2_2.2.1
```

```
##  [3] rCube_1.1.0                SummarizedExperiment_1.7.5
##  [5] DelayedArray_0.3.19        matrixStats_0.52.2
##  [7] Biobase_2.37.2             GenomicRanges_1.29.12
##  [9] GenomeInfoDb_1.13.4        IRanges_2.11.12
## [11] S4Vectors_0.15.5           BiocGenerics_0.23.0
## [13] knitr_1.17
##
## loaded via a namespace (and not attached):
##  [1] ProtGenerics_1.9.0            bitops_1.0-6
##  [3] bit64_0.9-7                   httr_1.2.1
##  [5] RColorBrewer_1.1-2            progress_1.1.2
##  [7] rprojroot_1.2                 tools_3.4.1
##  [9] backports_1.1.0               R6_2.2.2
## [11] rpart_4.1-11                  Hmisc_4.0-3
## [13] DBI_0.7                       lazyeval_0.2.0
## [15] colorspace_1.3-2              nnet_7.3-12
## [17] gridExtra_2.2.1               prettyunits_1.0.2
## [19] GGally_1.3.2                  DESeq2_1.17.13
## [21] curl_2.8.1                    bit_1.1-12
## [23] compiler_3.4.1                graph_1.55.0
## [25] htmlTable_1.9                 rtracklayer_1.37.3
## [27] scales_0.4.1                  checkmate_1.8.3
## [29] genefilter_1.59.0             RBGL_1.53.0
## [31] stringr_1.2.0                 digest_0.6.12
## [33] Rsamtools_1.29.0              foreign_0.8-69
## [35] rmarkdown_1.6                 XVector_0.17.0
## [37] base64enc_0.1-3               dichromat_2.0-0
## [39] htmltools_0.3.6               ensembldb_2.1.11
## [41] BSgenome_1.45.1               highr_0.6
## [43] htmlwidgets_0.9               rlang_0.1.2
## [45] RSQLite_2.0                   BiocInstaller_1.27.3
## [47] shiny_1.0.4                   BiocParallel_1.11.6
## [49] acepack_1.4.1                 VariantAnnotation_1.23.8
## [51] RCurl_1.95-4.8                magrittr_1.5
## [53] GenomeInfoDbData_0.99.1       Formula_1.2-2
## [55] Matrix_1.2-10                 Rcpp_0.12.12
## [57] munsell_0.4.3                 stringi_1.1.5
## [59] yaml_2.1.14                   MASS_7.3-47
## [61] zlibbioc_1.23.0               AnnotationHub_2.9.5
## [63] plyr_1.8.4                    grid_3.4.1
## [65] blob_1.1.0                    lattice_0.20-35
## [67] Biostrings_2.45.3             splines_3.4.1
## [69] GenomicFeatures_1.29.8        annotate_1.55.0
## [71] locfit_1.5-9.1                geneplotter_1.55.0
## [73] reshape2_1.4.2                biomaRt_2.33.4
## [75] XML_3.98-1.9                  evaluate_0.10.1
## [77] biovizBase_1.25.1             latticeExtra_0.6-28
## [79] data.table_1.10.4             httpuv_1.3.5
## [81] gtable_0.2.0                  reshape_0.8.7
## [83] assertthat_0.2.0              mime_0.5
## [85] xtable_1.8-2                  AnnotationFilter_1.1.3
## [87] survival_2.41-3               OrganismDbi_1.19.0
## [89] tibble_1.3.3                  GenomicAlignments_1.13.4
```

```
## [91] AnnotationDbi_1.39.2          memoise_1.1.0
## [93] cluster_2.0.6                 interactiveDisplayBase_1.15.0
## [95] BiocStyle_2.5.15
```

6 References

[1] Björn Schwalb, Margaux Michel, Benedikt Zacher, Carina Frühauf, Katja Demel, Achim Tresch, Julien Gagneur, and Patrick Cramer. TT-seq maps the human transient transcriptome. *Science*, 352(6290):1225–1228, 2016.

[2] Christian Miller, Björn Schwalb, Kerstin Maier, Daniel Schulz, Sebastian Dümcke, Benedikt Zacher, Andreas Mayer, Jasmin Sydow, Lisa Marcinowski, Lars Dölken, Dietmar E Martin, Achim Tresch, and Patrick Cramer. Dynamic transcriptome analysis measures rates of mRNA synthesis and decay in yeast. *Molecular Systems Biology*, 7(458):458, jan 2011. URL: http://www.nature.com/doifinder/10.1038/msb.2010.112http://dx.doi.org/10.1038/msb.2010.112, doi:10.1038/msb.2010.112.

[3] Margaux Michel, Carina Demel, Benedikt Zacher, Björn Schwalb, Stefan Krebs, Julien Gagneur, and Patrick Cramer. TT-seq captures enhancer landscapes immediately after T-cell stimulation. *Molecular Systems Biology*, 13(3):920, 2017. doi:10.15252/msb.20167507.

[4] Philipp Eser, Leonhard Wachutka, Kerstin C Maier, Carina Demel, Mariana Boroni, Srignanakshi Iyer, Patrick Cramer, and Julien Gagneur. Determinants of RNA metabolism in the Schizosaccharomyces pombe genome. *Molecular Systems Biology*, 12:857, 2016. URL: http://biorxiv.org/content/early/2015/08/26/025585.abstracthttp://msb.embopress.org/content/12/2/857.abstract, doi:10.15252/msb.20156526.

[5] Leonhard Wachutka and Julien Gagneur. Measures of RNA metabolism rates: Toward a definition at the level of single bonds. *Transcription*, page e1257972, nov 2016. URL: https://www.tandfonline.com/doi/full/10.1080/21541264.2016.1257972, doi:10.1080/21541264.2016.1257972.

[6] James H Bullard, Elizabeth Purdom, Kasper D Hansen, and Sandrine Dudoit. Evaluation of statistical methods for normalization and differential expression in mRNA-Seq experiments. *BMC Bioinformatics*, 11:94, 2010. doi:10.1186/1471-2105-11-94.

[7] Michael I Love, Wolfgang Huber, and Simon Anders. Moderated estimation of fold change and dispersion for RNA-Seq data with DESeq2. *Genome Biology*, 15(12):550, 2014.

[8] Michal Rabani, Raktima Raychowdhury, Marko Jovanovic, Michael Rooney, Deborah J. Stumpo, Andrea Pauli, Nir Hacohen, Alexander F. Schier, Perry J. Blackshear, Nir Friedman, Ido Amit, and Aviv Regev. High-Resolution Sequencing and Modeling Identifies Distinct Dynamic RNA Regulatory Strategies. *Cell*, 159(7):1698–1710, 2014. URL: http://dx.doi.org/10.1016/j.cell.2014.11.015, doi:10.1016/j.cell.2014.11.015.