systemPipeR Workflow for Ribo-Seq and polyRibo-Seq Experiments

Piyada Juntawong, Jeremie Bazin, Maureen Hummel, Julia Bailey-Serres and Thomas Girke
Email contact: thomas.girke@ucr.edu

August 6, 2016

Contents

1 Introduction

Ribo-Seq and polyRibo-Seq are a specific form of RNA-Seq gene expression experiments utilizing mRNA subpopulations directly bound to ribosomes. Compared to standard RNA-Seq, their readout of gene expression provides a better approximation of downstream protein abundance profiles due to their close association with translational processes. The most important difference among the two is that polyRibo-Seq utilizes polyribosomal RNA for sequencing, whereas Ribo-Seq is a footprinting approach restricted to sequencing RNA fragments protected by ribosomes (**???**).

The workflow presented in this vignette contains most of the data analysis steps described by **?** including functionalities useful for processing both polyRibo-Seq and Ribo-Seq experiments. To improve re-usability and adapt to recent changes of software versions (*e.g.* R, Bioconductor and short read aligners), the code has been optimized accordingly. Thus, the results obtained with the updated workflow are expected to be similar but not necessarily identical with the published results described in the original paper.

Relevant analysis steps of this workflow include read preprocessing, read alignments against a reference genome, counting of reads overlapping with a wide range of genomic features (*e.g.* CDSs, UTRs, uORFs, rRNAs, etc.), differential gene expression and differential ribosome binding analyses, as well as a variety of genome-wide summary plots for visualizing RNA expression trends. Functions are provided for evaluating the quality of Ribo-seq data, for identifying novel expressed regions in the genomes, and for gaining insights into gene regulation at the post-transcriptional and translational levels. For example, the functions `genFeatures` and `featuretypeCounts` can be used to quantify the expression output for all feature types included in a genome annotation (*e.g.* genes, introns, exons, miRNAs, intergenic regions, etc.). To determine the approximate read length of ribosome footprints in Ribo-Seq experiments, these feature type counts can be obtained and plotted for specific read lengths separately. Typically, the most abundant read length obtained for translated features corresponds to the approximate footprint length occupied by the ribosomes of a given organism group. Based on the results from several Ribo-Seq studies, these ribosome footprints are typically ~30 nucleotides long (**???**). However, their length can vary by several nucleotides depending upon the optimization of the RNA digestion step and various factors associated with translational regulation. For quality control purposes of Ribo-Seq experiments it is also useful to monitor the abundance of reads mapping to rRNA genes due to the high rRNA content of ribosomes. This information can be generated with the `featuretypeCounts` function described above.

Coverage trends along transcripts summarized for any number of transcripts can be obtained and plotted with the functions `featureCoverage` and `plotfeatureCoverage`, respectively. Their results allow monitoring of the phasing of ribosome movements along triplets of coding sequences. Commonly, high quality data will display here for the first nucleotide of each codon the highest depth of coverage computed for the 5' ends of the aligned reads.

Ribo-seq data can also be used to evaluate various aspects of translational control due to ribosome occupancy in upstream open reading frames (uORFs). The latter are frequently present in (or near) 5' UTRs of transcripts. For this, the function `predORFs` can be used to identify ORFs in the nucleotide sequences of transcripts or their subcomponents such as UTR regions. After scaling the resulting ORF coordinates back to the corresponding genome locations using `scaleRanges`,

one can use these novel features (*e.g.* uORFs) for expression analysis routines similar to those employed for pre-existing annotations, such as the exonic regions of genes. For instance, in Ribo-Seq experiments one can use this approach to systematically identify all transcripts occupied by ribosomes in their uORF regions. The binding of ribosomes to uORF regions may indicate a regulatory role in the translation of the downstream main ORFs and/or translation of the uORFs into functionally relevant peptides.

1.1 Experimental design

Typically, users want to specify here all information relevant for the analysis of their NGS study. This includes detailed descriptions of FASTQ files, experimental design, reference genome, gene annotations, etc.

2 Load workflow environment

2.1 Load packages and sample data

The *systemPipeR* package needs to be loaded to perform the analysis steps shown in this report (**?**). The package allows users to run the entire analysis workflow interactively or with a single command while also generating the corresponding analysis report. For details see *systemPipeR's* main vignette.

```
library(systemPipeR)
```

Load workflow environment with sample data into your current working directory. The sample data are described here.

```
library(systemPipeRdata)
genWorkenvir(workflow="ribseq")
setwd("riboseq")
```

In the workflow environments generated by `genWorkenvir` all data inputs are stored in a `data/` directory and all analysis results will be written to a separate `results/` directory, while the `systemPipeRIBOseq.Rnw` script and the `targets` file are expected to be located in the parent directory. The R session is expected to run from this parent directory. Additional parameter files are stored under `param/`.

To work with real data, users want to organize their own data similarly and substitute all test data for their own data. To rerun an established workflow on new data, the initial `targets` file along with the corresponding FASTQ files are usually the only inputs the user needs to provide.

If applicable users can load custom functions not provided by *systemPipeR*. Skip this step if this is not the case.

```
source("systemPipeRIBOseq_Fct.R")
```

2.2 Experiment definition provided by `targets` file

The `targets` file defines all FASTQ files and sample comparisons of the analysis workflow.

```
targetspath <- system.file("extdata", "targets.txt", package="systemPipeR")
targets <- read.delim(targetspath, comment.char = "#")[,1:4]
targets
```

```
                  FileName SampleName Factor SampleLong
1  ./data/SRR446027_1.fastq        M1A     M1  Mock.1h.A
2  ./data/SRR446028_1.fastq        M1B     M1  Mock.1h.B
3  ./data/SRR446029_1.fastq        A1A     A1   Avr.1h.A
4  ./data/SRR446030_1.fastq        A1B     A1   Avr.1h.B
5  ./data/SRR446031_1.fastq        V1A     V1   Vir.1h.A
```

```
6  ./data/SRR446032_1.fastq        V1B     V1   Vir.1h.B
7  ./data/SRR446033_1.fastq        M6A     M6  Mock.6h.A
8  ./data/SRR446034_1.fastq        M6B     M6  Mock.6h.B
9  ./data/SRR446035_1.fastq        A6A     A6   Avr.6h.A
10 ./data/SRR446036_1.fastq        A6B     A6   Avr.6h.B
11 ./data/SRR446037_1.fastq        V6A     V6   Vir.6h.A
12 ./data/SRR446038_1.fastq        V6B     V6   Vir.6h.B
13 ./data/SRR446039_1.fastq       M12A    M12 Mock.12h.A
14 ./data/SRR446040_1.fastq       M12B    M12 Mock.12h.B
15 ./data/SRR446041_1.fastq       A12A    A12  Avr.12h.A
16 ./data/SRR446042_1.fastq       A12B    A12  Avr.12h.B
17 ./data/SRR446043_1.fastq       V12A    V12  Vir.12h.A
18 ./data/SRR446044_1.fastq       V12B    V12  Vir.12h.B
```

3 Read preprocessing

3.1 Quality filtering and adaptor trimming

The following custom function trims adaptors hierarchically from the longest to the shortest match of the right end of the reads. If `internalmatch=TRUE` then internal matches will trigger the same behavior. The argument `minpatternlength` defines the shortest adaptor match to consider in this iterative process. In addition, the function removes reads containing Ns or homopolymer regions. More detailed information on read preprocessing is provided in *systemPipeR*'s main vignette.

```
args <- systemArgs(sysma="param/trim.param", mytargets="targets.txt")
fctpath <- system.file("extdata", "custom_Fct.R", package="systemPipeR")
source(fctpath)
iterTrim <- ".iterTrimbatch1(fq, pattern='ACACGTCT', internalmatch=FALSE, minpatternlength=6,
                             Nnumber=1, polyhomo=50, minreadlength=16, maxreadlength=100)"
preprocessReads(args=args, Fct=iterTrim, batchsize=100000, overwrite=TRUE, compress=TRUE)
writeTargetsout(x=args, file="targets_trim.txt", overwrite=TRUE)
```

3.2 FASTQ quality report

The following `seeFastq` and `seeFastqPlot` functions generate and plot a series of useful quality statistics for a set of FASTQ files including per cycle quality box plots, base proportions, base-level quality trends, relative k-mer diversity, length and occurrence distribution of reads, number of reads above quality cutoffs and mean quality distribution. The results are written to a PDF file named `fastqReport.pdf`.

```
args <- systemArgs(sysma="param/tophat.param", mytargets="targets_trim.txt")
fqlist <- seeFastq(fastq=infile1(args), batchsize=100000, klength=8)
pdf("./results/fastqReport.pdf", height=18, width=4*length(fqlist))
seeFastqPlot(fqlist)
dev.off()
```

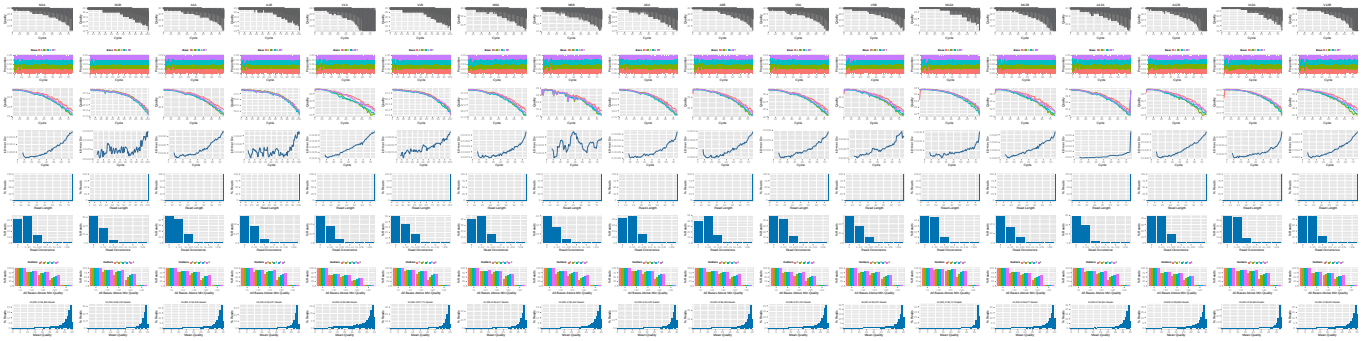

Figure 1: QC report for 18 FASTQ files.

4 Alignments

4.1 Read mapping with Bowtie2/Tophat2

The NGS reads of this project will be aligned against the reference genome sequence using Bowtie2/TopHat2 (**??**). The parameter settings of the aligner are defined in the `tophat.param` file.

```
args <- systemArgs(sysma="param/tophat.param", mytargets="targets.txt")
sysargs(args)[1] # Command-line parameters for first FASTQ file
```

Submission of alignment jobs to compute cluster, here using 72 CPU cores (18 `qsub` processes each with 4 CPU cores).

```
moduleload(modules(args))
system("bowtie2-build ./data/tair10.fasta ./data/tair10.fasta")
resources <- list(walltime="20:00:00", nodes=paste0("1:ppn=", cores(args)), memory="10gb")
reg <- clusterRun(args, conffile=".BatchJobs.R", template="torque.tmpl", Njobs=18, runid="01",
                  resourceList=resources)
waitForJobs(reg)
```

Check whether all BAM files have been created

```
file.exists(outpaths(args))
```

4.2 Read and alignment stats

The following provides an overview of the number of reads in each sample and how many of them aligned to the reference.

```
read_statsDF <- alignStats(args=args)
write.table(read_statsDF, "results/alignStats.xls", row.names=FALSE, quote=FALSE, sep="\t")
```

```
read.table(system.file("extdata", "alignStats.xls", package="systemPipeR"), header=TRUE)[1:4,]
```

```
  FileName Nreads2x Nalign Perc_Aligned Nalign_Primary Perc_Aligned_Primary
1      M1A   192918 177961     92.24697         177961             92.24697
2      M1B   197484 159378     80.70426         159378             80.70426
3      A1A   189870 176055     92.72397         176055             92.72397
4      A1B   188854 147768     78.24457         147768             78.24457
```

4.3 Create symbolic links for viewing BAM files in IGV

The `symLink2bam` function creates symbolic links to view the BAM alignment files in a genome browser such as IGV. The corresponding URLs are written to a file with a path specified under `urlfile`, here IGVurl.txt.

```
symLink2bam(sysargs=args, htmldir=c("~/.html/", "somedir/"),
            urlbase="http://biocluster.ucr.edu/~tgirke/",
            urlfile="./results/IGVurl.txt")
```

5 Read distribution across genomic features

The `genFeatures` function generates a variety of feature types from `TxDb` objects using utilities provided by the *GenomicFeatures* package.

5.1 Obtain feature types

The first step is the generation of the feature type ranges based on annotations provided by a GFF file that can be transformed into a `TxDb` object. This includes ranges for mRNAs, exons, introns, UTRs, CDSs, miRNAs, rRNAs, tRNAs, promoter and intergenic regions. In addition, any number of custom annotations can be included in this routine.

```
library(GenomicFeatures)
file <- system.file("extdata/annotation", "tair10.gff", package="systemPipeRdata")
txdb <- makeTxDbFromGFF(file=file, format="gff3", organism="Arabidopsis")
feat <- genFeatures(txdb, featuretype="all", reduce_ranges=TRUE, upstream=1000, downstream=0,
                    verbose=TRUE)
```

5.2 Count and plot reads of any length

The `featuretypeCounts` function counts how many reads in short read alignment files (BAM format) overlap with entire annotation categories. This utility is useful for analyzing the distribution of the read mappings across feature types, *e.g.* coding versus non-coding genes. By default the read counts are reported for the sense and antisense strand of each feature type separately. To minimize memory consumption, the BAM files are processed in a stream using utilities from the *Rsamtools* and *GenomicAlignment* packages. The counts can be reported for each read length separately or as a single value for reads of any length. Subsequently, the counting results can be plotted with the associated `plotfeaturetypeCounts` function.

The following generates and plots feature counts for any read length.

```
library(ggplot2); library(grid)
fc <- featuretypeCounts(bfl=BamFileList(outpaths(args), yieldSize=50000), grl=feat,
                        singleEnd=TRUE, readlength=NULL, type="data.frame")
p <- plotfeaturetypeCounts(x=fc, graphicsfile="results/featureCounts.pdf", graphicsformat="pdf",
                           scales="fixed", anyreadlength=TRUE, scale_length_val=NULL)
```

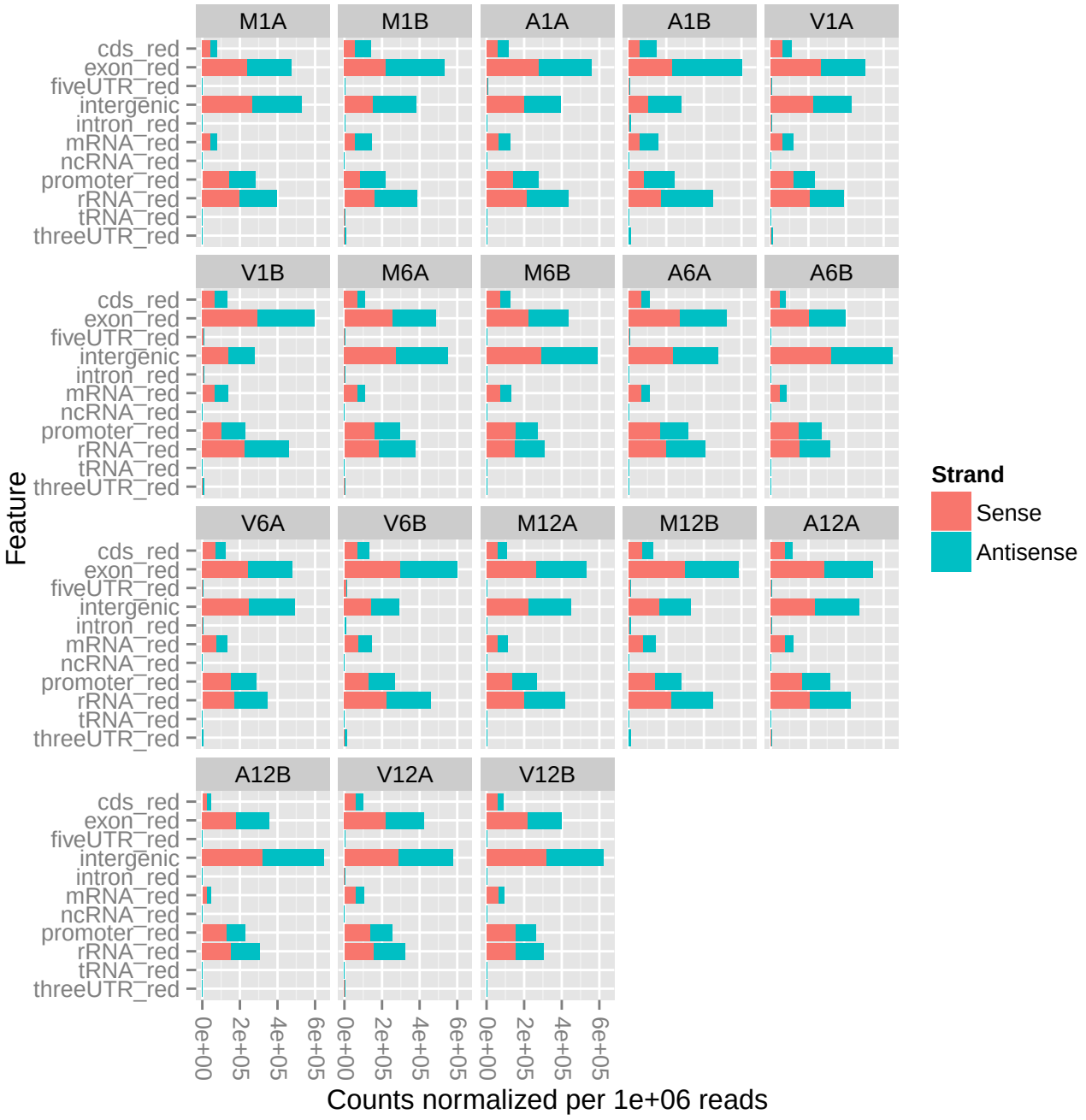

Figure 2: Read distribution plot across annotation features for any read length.

5.3 Count and plot reads of specific lengths

To determine the approximate read length of ribosome footprints in Ribo-Seq experiments, one can generate and plot the feature counts for specific read lengths separately. Typically, the most abundant read length obtained for translated features corresponds to the approximate footprint length occupied by the ribosomes.

```
fc2 <- featuretypeCounts(bfl=BamFileList(outpaths(args), yieldSize=50000), grl=feat,
                         singleEnd=TRUE, readlength=c(74:76,99:102), type="data.frame")
p2 <- plotfeaturetypeCounts(x=fc2, graphicsfile="results/featureCounts2.pdf", graphicsformat="pdf",
                            scales="fixed", anyreadlength=FALSE, scale_length_val=NULL)
```

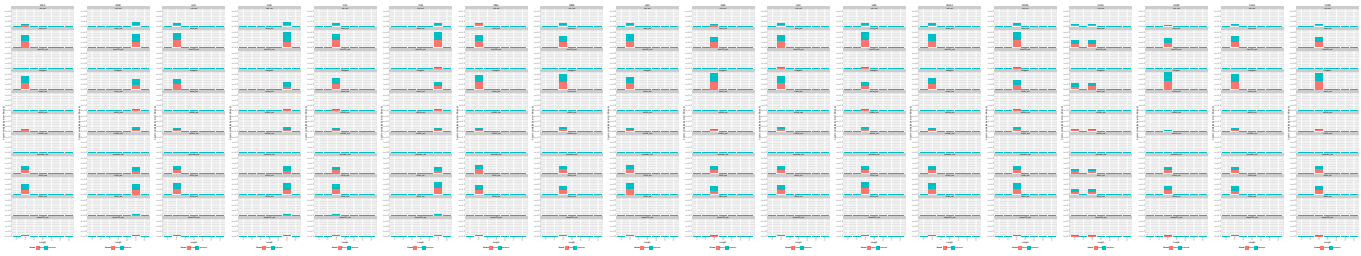

Figure 3: Read distribution plot across annotation features for specific read lengths.

6 Adding custom features to workflow

6.1 Predicting uORFs in 5' UTR regions

The function `predORF` can be used to identify open reading frames (ORFs) and coding sequences (CDSs) in DNA sequences provided as `DNAString` or `DNAStringSet` objects. The setting `mode='ORF'` returns continuous reading frames that begin with a start codon and end with a stop codon, while `mode='CDS'` returns continuous reading frames that do not need to begin or end with start or stop codons, respectively. Non-canonical start and stop condons are supported by allowing the user to provide any custom set of triplets under the `startcodon` and `stopcodon` arguments (*i.e.* non-ATG start codons). The argument `n` defines the maximum number of ORFs to return for each input sequence (*e.g.* `n=1` returns only the longest ORF). It also supports the identification of overlapping and nested ORFs. Alternatively, one can return all non-overlapping ORFs including the longest ORF for each input sequence with `n="all"` and `longest_disjoint=TRUE`.

```
library(systemPipeRdata); library(GenomicFeatures); library(rtracklayer)
gff <- system.file("extdata/annotation", "tair10.gff", package="systemPipeRdata")
txdb <- makeTxDbFromGFF(file=gff, format="gff3", organism="Arabidopsis")
futr <- fiveUTRsByTranscript(txdb, use.names=TRUE)
genome <- system.file("extdata/annotation", "tair10.fasta", package="systemPipeRdata")
dna <- extractTranscriptSeqs(FaFile(genome), futr)
uorf <- predORF(dna, n="all", mode="orf", longest_disjoint=TRUE, strand="sense")
```

To use the predicted ORF ranges for expression analysis given genome alignments as input, it is necessary to scale them to the corresponding genome coordinates. The function `scaleRanges` does this by transforming the mappings of spliced features (query ranges) to their corresponding genome coordinates (subject ranges). The method accounts for introns in the subject ranges that are absent in the query ranges. The above uORFs predicted in the provided 5' UTRs sequences using `predORF` are a typical use case for this application. These query ranges are given relative to the 5' UTR sequences and `scaleRanges` will convert them to the corresponding genome coordinates. The resulting `GRangesList` object (here `grl_scaled`) can be directly used for read counting as described in Section **??**.

```
grl_scaled <- scaleRanges(subject=futr, query=uorf, type="uORF", verbose=TRUE)
export.gff3(unlist(grl_scaled), "uorf.gff")
```

To confirm the correctness of the obtained uORF ranges, one can parse their corresponding DNA sequences from the reference genome with the `getSeq` function and then translate them with the `translate` function into proteins. Typically, the returned protein sequences should start with a `M` (corresponding to start codon) and end with a `*` (corresponding to stop codon). The following example does this for a single uORF containing three exons.

```
translate(unlist(getSeq(FaFile(genome), grl_scaled[[7]])))
```

6.2 Adding custom features to other feature types

If required custom feature ranges can be added to the standard features generated in Section **??**. The following does this for the uORF ranges predicted in Subsection **??**.

```
feat <- genFeatures(txdb, featuretype="all", reduce_ranges=FALSE)
feat <- c(feat, GRangesList("uORF"=unlist(grl_scaled)))
```

6.3 Predicting sORFs in intergenic regions

The following identifies continuous ORFs in intergenic regions. Note, `predORF` can only identify continuous ORFs in query sequences. The function does not identify and remove introns prior to the ORF prediction.

```
feat <- genFeatures(txdb, featuretype="intergenic", reduce_ranges=TRUE)
intergenic <- feat$intergenic
strand(intergenic) <- "+"
dna <- getSeq(FaFile(genome), intergenic)
names(dna) <- mcols(intergenic)$feature_by
sorf <- predORF(dna, n="all", mode="orf", longest_disjoint=TRUE, strand="both")
sorf <- sorf[width(sorf) > 60] # Remove sORFs below length cutoff, here 60bp
intergenic <- split(intergenic, mcols(intergenic)$feature_by)
grl_scaled_intergenic <- scaleRanges(subject=intergenic, query=sorf, type="sORF", verbose=TRUE)
export.gff3(unlist(grl_scaled_intergenic), "sorf.gff")
translate(getSeq(FaFile(genome), unlist(grl_scaled_intergenic)))
```

7 Genomic read coverage along transripts or CDSs

The `featureCoverage` function computes the read coverage along single and multi component features based on genomic alignments. The coverage segments of component features are spliced to continuous ranges, such as exons to transcripts or CDSs to ORFs. The results can be obtained with single nucleotide resolution (*e.g.* around start and stop codons) or as mean coverage of relative bin sizes, such as 100 bins for each feature. The latter allows comparisons of coverage trends among transcripts of variable length. Additionally, the results can be obtained for single or many features (*e.g.* any number of transcripts) at once. Visualization of the coverage results is facilitated by the downstream `plotfeatureCoverage` function.

7.1 Binned CDS coverage to compare many transcripts

```
grl <- cdsBy(txdb, "tx", use.names=TRUE)
fcov <- featureCoverage(bfl=BamFileList(outpaths(args)[1:2]), grl=grl[1:4], resizereads=NULL,
                        readlengthrange=NULL, Nbins=20, method=mean, fixedmatrix=FALSE,
                        resizefeatures=TRUE, upstream=20, downstream=20,
                        outfile="results/featureCoverage.xls", overwrite=TRUE)
```

7.2 Coverage upstream and downstream of start and stop codons

```
fcov <- featureCoverage(bfl=BamFileList(outpaths(args)[1:4]), grl=grl[1:12], resizereads=NULL,
                        readlengthrange=NULL, Nbins=NULL, method=mean, fixedmatrix=TRUE,
                        resizefeatures=TRUE, upstream=20, downstream=20,
                        outfile="results/featureCoverage.xls", overwrite=TRUE)
plotfeatureCoverage(covMA=fcov, method=mean, scales="fixed", extendylim=2, scale_count_val=10^6)
```

7.3 Combined coverage for both binned CDS and start/stop codons

```
library(ggplot2); library(grid)
fcov <- featureCoverage(bfl=BamFileList(outpaths(args)[1:2]), grl=grl[1:4], resizereads=NULL,
                        readlengthrange=NULL, Nbins=20, method=mean, fixedmatrix=TRUE,
                        resizefeatures=TRUE, upstream=20, downstream=20,
                        outfile="results/featureCoverage.xls", overwrite=TRUE)
pdf("./results/featurePlot.pdf", height=12, width=24)
plotfeatureCoverage(covMA=fcov, method=mean, scales="fixed", extendylim=2, scale_count_val=10^6)
dev.off()
```

Figure 4: Feature coverage plot with single nucleotide resolution around start and stop codons and binned coverage between them.

7.4 Nucleotide level coverage along entire transcripts/CDSs

```
fcov <- featureCoverage(bfl=BamFileList(outpaths(args)[1:2]), grl=grl[1:4], resizereads=NULL,
                        readlengthrange=NULL, Nbins=NULL, method=mean, fixedmatrix=FALSE,
                        resizefeatures=TRUE, upstream=20, downstream=20)
plotfeatureCoverage(covMA=fcov, method=mean, scales="fixed", scale_count_val=10^6)
```

8 Read quantification per annotation range

8.1 Read counting with `summarizeOverlaps` in parallel mode using multiple cores

Reads overlapping with annotation ranges of interest are counted for each sample using the `summarizeOverlaps` function (**?**). The read counting is preformed for exonic gene regions in a non-strand-specific manner while ignoring overlaps among different genes. Subsequently, the expression count values are normalized by *reads per kp per million mapped reads* (RPKM). The raw read count table (countDFeByg.xls) and the correspoding RPKM table (rpkmDFeByg.xls) are written to separate files in the `results` directory of this project. Parallelization is achieved with the *BiocParallel* package, here using 8 CPU cores.

```
library("GenomicFeatures"); library(BiocParallel)
txdb <- loadDb("./data/tair10.sqlite")
eByg <- exonsBy(txdb, by=c("gene"))
bfl <- BamFileList(outpaths(args), yieldSize=50000, index=character())
multicoreParam <- MulticoreParam(workers=8); register(multicoreParam); registered()
counteByg <- bplapply(bfl, function(x) summarizeOverlaps(eByg, x, mode="Union",
                                                         ignore.strand=TRUE,
                                                         inter.feature=FALSE,
                                                         singleEnd=TRUE))
countDFeByg <- sapply(seq(along=counteByg), function(x) assays(counteByg[[x]])$counts)
rownames(countDFeByg) <- names(rowRanges(counteByg[[1]])); colnames(countDFeByg) <- names(bfl)
rpkmDFeByg <- apply(countDFeByg, 2, function(x) returnRPKM(counts=x, ranges=eByg))
write.table(countDFeByg, "results/countDFeByg.xls", col.names=NA, quote=FALSE, sep="\t")
write.table(rpkmDFeByg, "results/rpkmDFeByg.xls", col.names=NA, quote=FALSE, sep="\t")
```

Sample of data slice of count table

```
read.delim("results/countDFeByg.xls", row.names=1, check.names=FALSE)[1:4,1:5]
```

Sample of data slice of RPKM table

```
read.delim("results/rpkmDFeByg.xls", row.names=1, check.names=FALSE)[1:4,1:4]
```

Note, for most statistical differential expression or abundance analysis methods, such as *edgeR* or *DESeq2*, the raw count values should be used as input. The usage of RPKM values should be restricted to specialty applications required by some users, *e.g.* manually comparing the expression levels among different genes or features.

8.2 Sample-wise correlation analysis

The following computes the sample-wise Spearman correlation coefficients from the `rlog` transformed expression values generated with the *DESeq2* package. After transformation to a distance matrix, hierarchical clustering is performed with the `hclust` function and the result is plotted as a dendrogram (sample_tree.pdf).

```
library(DESeq2, quietly=TRUE); library(ape,  warn.conflicts=FALSE)
countDF <- as.matrix(read.table("./results/countDFeByg.xls"))
colData <- data.frame(row.names=targetsin(args)$SampleName, condition=targetsin(args)$Factor)
dds <- DESeqDataSetFromMatrix(countData = countDF, colData = colData, design = ~ condition)
d <- cor(assay(rlog(dds)), method="spearman")
hc <- hclust(dist(1-d))
pdf("results/sample_tree.pdf")
plot.phylo(as.phylo(hc), type="p", edge.col="blue", edge.width=2, show.node.label=TRUE,
           no.margin=TRUE)
dev.off()
```

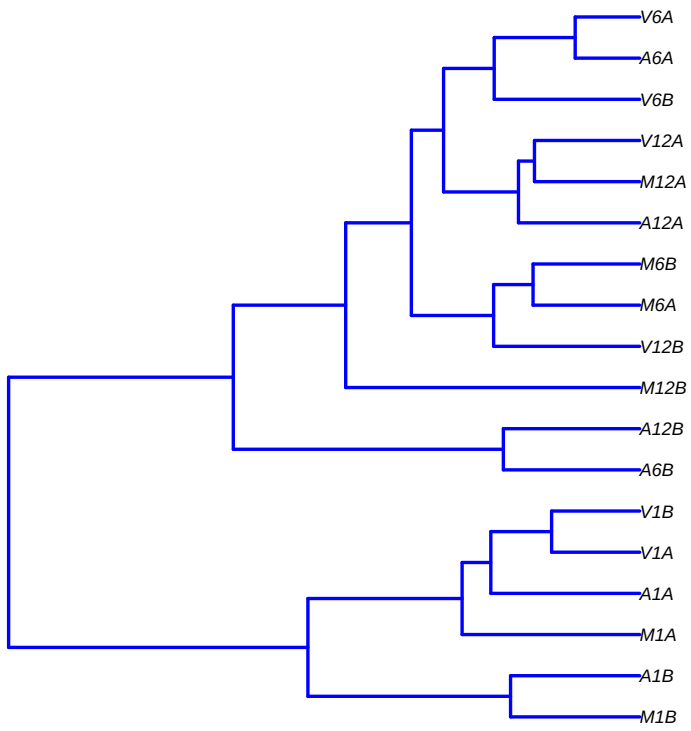

Figure 5: Correlation dendrogram of samples.

9 Analysis of differentially expressed genes with edgeR

The analysis of differentially expressed genes (DEGs) is performed with the glm method from the *edgeR* package (**?**). The sample comparisons used by this analysis are defined in the header lines of the `targets` file starting with `<CMP>`.

```
library(edgeR)
countDF <- read.delim("results/countDFeByg.xls", row.names=1, check.names=FALSE)
targets <- read.delim("targets.txt", comment="#")
cmp <- readComp(file="targets.txt", format="matrix", delim="-")
edgeDF <- run_edgeR(countDF=countDF, targets=targets, cmp=cmp[[1]], independent=FALSE, mdsplot="")
```

Add custom functional descriptions. Skip this step if `desc.xls` is not available.

```
desc <- read.delim("data/desc.xls")
desc <- desc[!duplicated(desc[,1]),]
descv <- as.character(desc[,2]); names(descv) <- as.character(desc[,1])
edgeDF <- data.frame(edgeDF, Desc=descv[rownames(edgeDF)], check.names=FALSE)
write.table(edgeDF, "./results/edgeRglm_allcomp.xls", quote=FALSE, sep="\t", col.names = NA)
```

Filter and plot DEG results for up and down regulated genes. The definition of '*up*' and '*down*' is given in the corresponding help file. To open it, type `?filterDEGs` in the R console.

```
edgeDF <- read.delim("results/edgeRglm_allcomp.xls", row.names=1, check.names=FALSE)
pdf("results/DEGcounts.pdf")
DEG_list <- filterDEGs(degDF=edgeDF, filter=c(Fold=2, FDR=1))
dev.off()
write.table(DEG_list$Summary, "./results/DEGcounts.xls", quote=FALSE, sep="\t", row.names=FALSE)
```

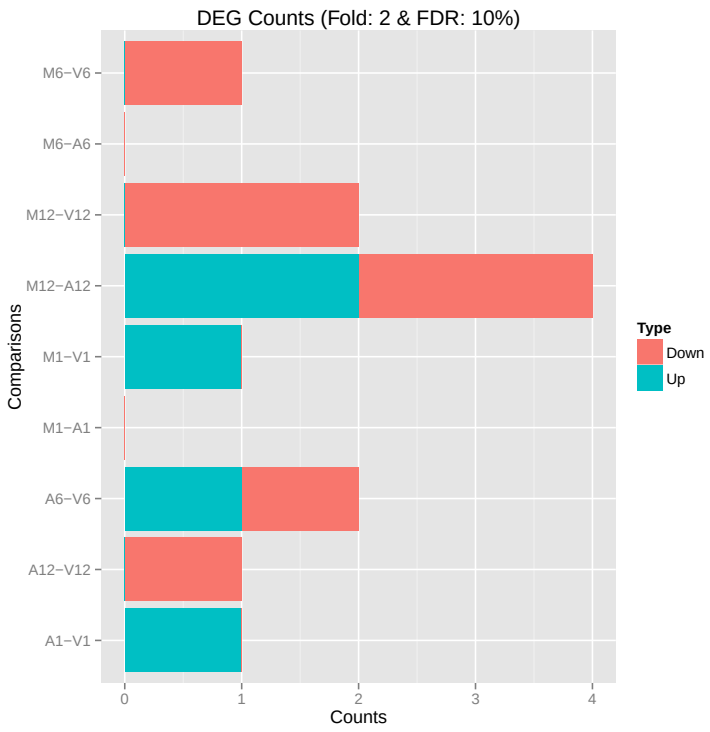

Figure 6: Up and down regulated DEGs with FDR of 1%.

The function `overLapper` can compute Venn intersects for large numbers of sample sets (up to 20 or more) and `vennPlot` can plot 2-5 way Venn diagrams. A useful feature is the possiblity to combine the counts from several Venn comparisons with the same number of sample sets in a single Venn diagram (here for 4 up and down DEG sets).

```
vennsetup <- overLapper(DEG_list$Up[6:9], type="vennsets")
vennsetdown <- overLapper(DEG_list$Down[6:9], type="vennsets")
pdf("results/vennplot.pdf")
vennPlot(list(vennsetup, vennsetdown), mymain="", mysub="", colmode=2, ccol=c("blue", "red"))
dev.off()
```

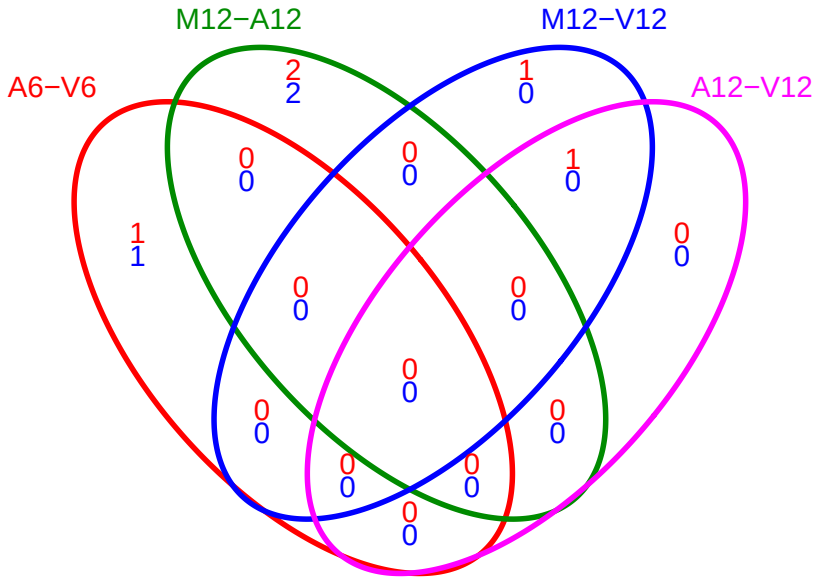

Figure 7: Venn Diagram for 4 Up and Down DEG Sets.

9.1 GO term enrichment analysis of DEGs

9.1.1 Obtain gene-to-GO mappings

The following shows how to obtain gene-to-GO mappings from *biomaRt* (here for *A. thaliana*) and how to organize them for the downstream GO term enrichment analysis. Alternatively, the gene-to-GO mappings can be obtained for many organisms from Bioconductor's `*.db` genome annotation packages or GO annotation files provided by various genome databases. For each annotation this relatively slow preprocessing step needs to be performed only once. Subsequently, the preprocessed data can be loaded with the `load` function as shown in the next subsection.

```
library("biomaRt")
listMarts() # To choose BioMart database
m <- useMart("ENSEMBL_MART_PLANT"); listDatasets(m)
m <- useMart("ENSEMBL_MART_PLANT", dataset="athaliana_eg_gene")
listAttributes(m) # Choose data types you want to download
go <- getBM(attributes=c("go_accession", "tair_locus", "go_namespace_1003"), mart=m)
go <- go[go[,3]!="",]; go[,3] <- as.character(go[,3])
```

```
go[go[,3]=="molecular_function", 3] <- "F"
go[go[,3]=="biological_process", 3] <- "P"
go[go[,3]=="cellular_component", 3] <- "C"
go[1:4,]
dir.create("./data/GO")
write.table(go, "data/GO/GOannotationsBiomart_mod.txt", quote=FALSE, row.names=FALSE,
            col.names=FALSE, sep="\t")
catdb <- makeCATdb(myfile="data/GO/GOannotationsBiomart_mod.txt", lib=NULL, org="",
                   colno=c(1,2,3), idconv=NULL)
save(catdb, file="data/GO/catdb.RData")
```

9.1.2 Batch GO term enrichment analysis

Apply the enrichment analysis to the DEG sets obtained the above differential expression analysis. Note, in the following example the `FDR` filter is set here to an unreasonably high value, simply because of the small size of the toy data set used in this vignette. Batch enrichment analysis of many gene sets is performed with the `GOCluster_Report` function. When `method="all"`, it returns all GO terms passing the p-value cutoff specified under the `cutoff` arguments. When `method="slim"`, it returns only the GO terms specified under the `myslimv` argument. The given example shows how a GO slim vector for a specific organism can be obtained from BioMart.

```
load("data/GO/catdb.RData")
DEG_list <- filterDEGs(degDF=edgeDF, filter=c(Fold=2, FDR=50), plot=FALSE)
up_down <- DEG_list$UporDown; names(up_down) <- paste(names(up_down), "_up_down", sep="")
up <- DEG_list$Up; names(up) <- paste(names(up), "_up", sep="")
down <- DEG_list$Down; names(down) <- paste(names(down), "_down", sep="")
DEGlist <- c(up_down, up, down)
DEGlist <- DEGlist[sapply(DEGlist, length) > 0]
BatchResult <- GOCluster_Report(catdb=catdb, setlist=DEGlist, method="all", id_type="gene",
                                CLSZ=2, cutoff=0.9, gocats=c("MF", "BP", "CC"),
                                recordSpecGO=NULL)
library("biomaRt"); m <- useMart("ENSEMBL_MART_PLANT", dataset="athaliana_eg_gene")
goslimvec <- as.character(getBM(attributes=c("goslim_goa_accession"), mart=m)[,1])
BatchResultslim <- GOCluster_Report(catdb=catdb, setlist=DEGlist, method="slim", id_type="gene",
                                    myslimv=goslimvec, CLSZ=10, cutoff=0.01,
                                    gocats=c("MF", "BP", "CC"), recordSpecGO=NULL)
```

9.1.3 Plot batch GO term results

The `data.frame` generated by `GOCluster_Report` can be plotted with the `goBarplot` function. Because of the variable size of the sample sets, it may not always be desirable to show the results from different DEG sets in the same bar plot. Plotting single sample sets is achieved by subsetting the input data frame as shown in the first line of the following example.

```
gos <- BatchResultslim[grep("M6-V6_up_down", BatchResultslim$CLID), ]
gos <- BatchResultslim
pdf("GOslimbarplotMF.pdf", height=8, width=10); goBarplot(gos, gocat="MF"); dev.off()
goBarplot(gos, gocat="BP")
goBarplot(gos, gocat="CC")
```

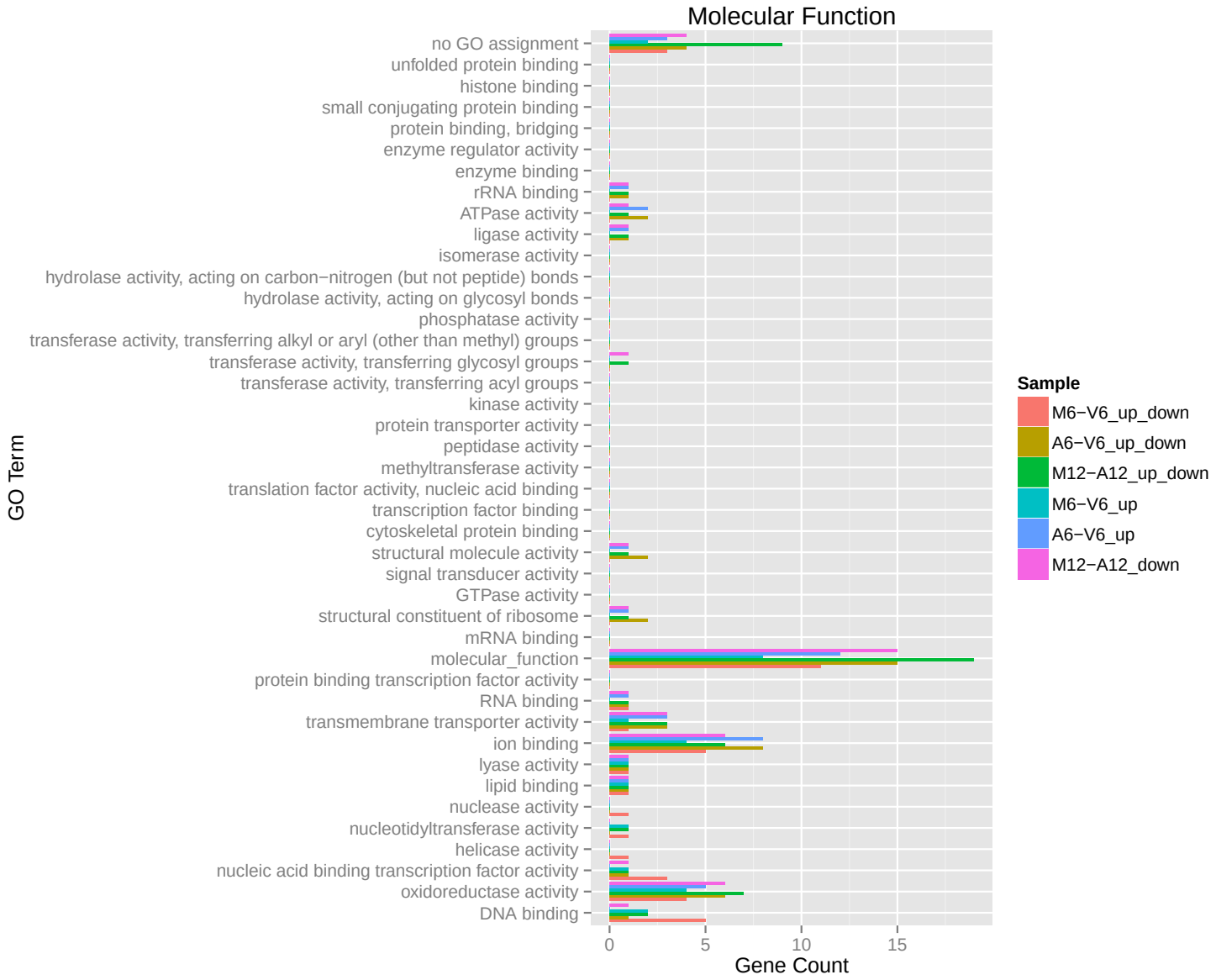

Figure 8: GO Slim Barplot for MF Ontology.

10 Differential ribosome loading analysis (translational efficiency)

Combinded with mRNA-Seq data, Ribo-Seq or polyRibo-Seq experiments can be used to study changes in translational efficiencies of genes and/or transcripts for different treatments. For test purposes the following generates a small test data set from the sample data used in this vignette, where two types of RNA samples (`assays`) are considered: polyribosomal mRNA (`Ribo`) and total mRNA (`mRNA`). In addition, there are two treatments (`conditions`): `M1` and `A1`.

```
library(DESeq2)
targetspath <- system.file("extdata", "targetsPE.txt", package="systemPipeR")
parampath <- system.file("extdata", "tophat.param", package="systemPipeR")
countDFeBygpath <- system.file("extdata", "countDFeByg.xls", package="systemPipeR")
```

```
args <- suppressWarnings(systemArgs(sysma=parampath, mytargets=targetspath))
countDFeByg <- read.delim(countDFeBygpath, row.names=1)
coldata <- DataFrame(assay=factor(rep(c("Ribo","mRNA"), each=4)),
                condition=factor(rep(as.character(targetsin(args)$Factor[1:4]), 2)),
                row.names=as.character(targetsin(args)$SampleName)[1:8])
coldata
```

```
DataFrame with 8 rows and 2 columns
       assay condition
    <factor>  <factor>
M1A     Ribo        M1
M1B     Ribo        M1
A1A     Ribo        A1
A1B     Ribo        A1
V1A     mRNA        M1
V1B     mRNA        M1
M6A     mRNA        A1
M6B     mRNA        A1
```

Differences in translational efficiencies can be calculated by ratios of ratios for the two conditions:

$$(Ribo\_A1/mRNA\_A1)/(Ribo\_M1/mRNA\_M1).$$

The latter can be modeled with the *DESeq2* package using the design '$\sim assay + condition + assay : condition$', where the interaction term '$assay : condition$' represents the ratio of ratios. Using the likelihood ratio test of *DESeq2*, which removes the interaction term in the reduced model, one can test whether the translational efficiency (ribosome loading) is different in condition `A1` than in `M1`.

```
dds <- DESeqDataSetFromMatrix(countData=as.matrix(countDFeByg[,rownames(coldata)]),
                              colData = coldata,
                              design = ~ assay + condition + assay:condition)
# model.matrix(~ assay + condition + assay:condition, coldata) # Corresponding design matrix
dds <- DESeq(dds, test="LRT", reduced = ~ assay + condition)
res <- DESeq2::results(dds)
head(res[order(res$padj),],4)
```

```
log2 fold change (MLE): assaymRNA.conditionM1
LRT p-value: '~ assay + condition + assay:condition' vs '~ assay + condition'
DataFrame with 4 rows and 6 columns
            baseMean log2FoldChange     lfcSE      stat       pvalue         padj
           <numeric>      <numeric> <numeric> <numeric>    <numeric>    <numeric>
AT5G01040   94.03820       5.824586 1.1642626  26.83953 2.210693e-07 2.254907e-05
AT5G01015   25.22693      -5.407541 1.1911336  19.52701 9.918751e-06 5.058563e-04
AT5G01100  540.71442       3.610760 0.8237736  18.22140 1.966567e-05 6.686328e-04
AT2G01021 9227.94454      -4.863878 1.2927551  12.74227 3.574829e-04 9.115813e-03
```

```
# write.table(res, file="transleff.xls", quote=FALSE, col.names = NA, sep="\t")
```

11 Clustering and heat maps

The following example performs hierarchical clustering on the `rlog` transformed expression matrix subsetted by the DEGs identified in the above differential expression analysis. It uses a Pearson correlation-based distance measure and complete linkage for cluster joining.

```
library(pheatmap)
geneids <- unique(as.character(unlist(DEG_list[[1]])))
y <- assay(rlog(dds))[geneids, ]
pdf("heatmap1.pdf")
pheatmap(y, scale="row", clustering_distance_rows="correlation",
         clustering_distance_cols="correlation")
dev.off()
```

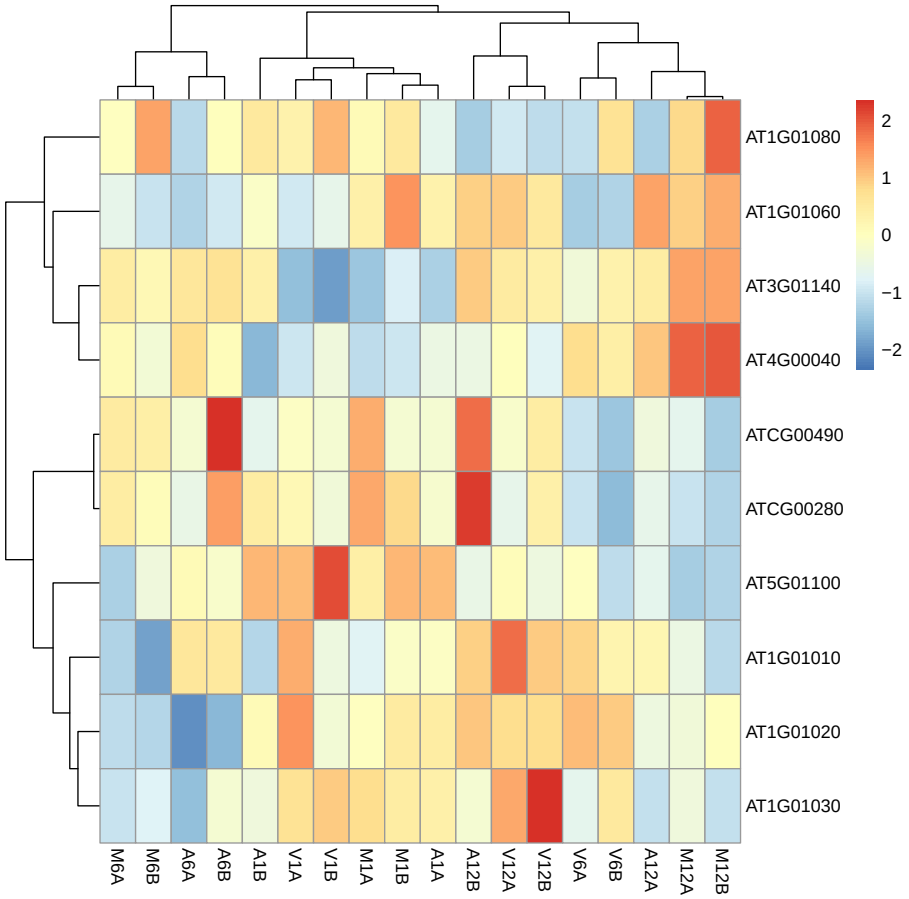

Figure 9: Heat map with hierarchical clustering dendrograms of DEGs.

12 Version Information

```
toLatex(sessionInfo())
```

- R version 3.3.1 (2016-06-21), `x86_64-apple-darwin13.4.0`
- Locale: `C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: Biobase 2.32.0, BiocGenerics 0.18.0, BiocParallel 1.6.4, BiocStyle 2.0.3, Biostrings 2.40.2, DESeq2 1.12.3, GenomeInfoDb 1.8.3, GenomicAlignments 1.8.4, GenomicRanges 1.24.2, IRanges 2.6.1, Rsamtools 1.24.0, S4Vectors 0.10.2, ShortRead 1.30.0, SummarizedExperiment 1.2.3, XVector 0.12.1, ape 3.5, ggplot2 2.1.0, knitr 1.13, systemPipeR 1.6.4
- Loaded via a namespace (and not attached): AnnotationDbi 1.34.4, AnnotationForge 1.14.2, BBmisc 1.10, BatchJobs 1.6, Category 2.38.0, DBI 0.4-1, Formula 1.2-1, GO.db 3.3.0, GOstats 2.38.1, GSEABase 1.34.0,

GenomicFeatures 1.24.5, Hmisc 3.17-4, Matrix 1.2-6, RBGL 1.48.1, RColorBrewer 1.1-2, RCurl 1.95-4.8, RSQLite 1.0.0, Rcpp 0.12.6, XML 3.98-1.4, acepack 1.3-3.3, annotate 1.50.0, backports 1.0.3, base64enc 0.1-3, biomaRt 2.28.0, bitops 1.0-6, brew 1.0-6, checkmate 1.8.1, chron 2.3-47, cluster 2.0.4, codetools 0.2-14, colorspace 1.2-6, data.table 1.9.6, digest 0.6.10, edgeR 3.14.0, evaluate 0.9, fail 1.3, foreign 0.8-66, formatR 1.4, genefilter 1.54.2, geneplotter 1.50.0, graph 1.50.0, grid 3.3.1, gridExtra 2.2.1, gtable 0.2.0, highr 0.6, htmltools 0.3.5, hwriter 1.3.2, labeling 0.3, lattice 0.20-33, latticeExtra 0.6-28, limma 3.28.17, locfit 1.5-9.1, magrittr 1.5, munsell 0.4.3, nlme 3.1-128, nnet 7.3-12, pheatmap 1.0.8, plyr 1.8.4, rjson 0.2.15, rmarkdown 1.0, rpart 4.1-10, rtracklayer 1.32.2, scales 0.4.0, sendmailR 1.2-1, splines 3.3.1, stringi 1.1.1, stringr 1.0.0, survival 2.39-5, tools 3.3.1, xtable 1.8-2, yaml 2.1.13, zlibbioc 1.18.0

13 Funding

This research was funded by National Science Foundation Grants IOS-0750811 and MCB-1021969, and a Marie Curie European Economic Community Fellowship PIOF-GA-2012-327954.