Package: FASE

February 23, 2023

Title: Analysis of RNA-Sequencing data using FASE.

Version: 0.1.24

Tutorial version: 2

Authors: Harsh Sharma, Trishna Pani, Ujjaini Dasgupta, Jyotsna Batra, Ravi Datta Sharma

Maintainer: Harsh Sharma <harshsharma.cb@gmail.com>

Description: This tutorial is for running the FASE pipeline. The pipeline is based on differential alternative splicing events and predicts the transcript structure and their concentration along with survival analysis. In fact this is the first kind of pipeline that takes advantage of differential alternative splicing events for finding novel transcripts that are neglected due to low expression in transcript level statistical analysis.

It has seven modules for downstream analysis of RNA-Sequencing data. These include: differential alternative splicing, transcript structure, transcript concentration, survival analysis, network analysis, differential gene expression, and differential junction expression.

License: GNU General Public License (GPL)

URL: https://github.com/harshsharma-cb/FASE

Citation: Sharma, H., Pani, T., Dasgupta, U., Batra, J., and Sharma, R.D., Prediction of transcript structure and concentration using RNA-Seq data, Briefings in Bioinformatics, 2023; Volume 24, Issue 2, March 2023, bbad022, https://doi.org/10.1093/bib/bbad022

1. Introduction

This tutorial is for running the FASE pipeline. The pipeline is based on differential alternative splicing events and predicts the transcript structure and their concentration along with survival analysis. In fact this is the first kind of pipeline that takes advantage of differential alternative splicing events for finding novel transcripts that are neglected due to low expression in transcript level statistical analysis.

It has seven modules for downstream analysis of RNA-Sequencing data. These include: differential alternative splicing, transcript structure, transcript concentration, survival analysis, network analysis, differential gene expression, and differential junction expression.

2. Installation

Most of the R libraries required by FASE will be installed by default along with FASE installation, which includes limma, edgeR, Rsubread, parallel, etc. Besides, following tools must be installed on Linux system before starting the analysis with FASE if pre-processing functions (see part 1 of the tutorial) are being used. However, if read counts have already been summarized, these tools are not required:

1. **fastq-dump** is required for converting SRA files to fastq files. It is required in order to pre-process sra files automatically.
2. **bowtie2** is required to build a genome index, for read mapping by tophat2.
3. **tophat2** is required for mapping reads to reference genome.
4. **samtools** is required to sort and index the bam files.

3. Sample case study

1. **SRA files:** In this tutorial, SRA files from NCBI SRA BioProject ID: PRJNA342407 have been used. In this study, one set of C. elegans (4 biological replicates for each condition) were grown on OP50: control (WT) and treated (eat-2(ad1116)(-)) and another set were grown on RNAi-seeded plates: control and treated (hrpu-1 or smg-2). For this tutorial, two samples each from control (WT) and eat-2 (-) should be downloaded using accession IDs: SRR5606854 (WT Day 1-_04), SRR5606849 (WT Day 1_03), SRR5606847 (eat-2(-) Day 1_03) and SRR5606848 (eat-2(-) Day 1_04). Alternative splicing, differential gene expression and differential junction expression would be compared between the WT and eat-2 (-) C. elegans. All four files are paired-end. For a comprehensive analysis, the SRA files should be renamed as WT-R1, WT-R2, eat2-R1 and eat2-R2, respectively. These files should be saved in an empty folder.
2. **GTF file:** GTF file is required for adding annotation to genes, junctions and alternative splicing events. GTF for C. elegans (WBcel235) can be downloaded from Ensembl (https://asia.ensembl.org/Caenorhabditis_elegans/Info/Index).
3. **Reference genome:** It is required by tophat2 for mapping the reads. Reference genome for C. elegans (WBcel235) can be downloaded from Ensembl (https://asia.ensembl.org/Caenorhabditis_elegans/Info/Index). GTF and reference genome should be saved together in a folder, separate from SRA files.

FASE needs to pre-processes RNA-Seq data before analysing alternative splicing, differential gene expression and differential junction expression. Therefore the tutorial is divided in two parts: Pre-processing and downstream analysis. Pre-processing can be performed by any of the three methods, depending on user preference:

1. Using automatic pre-processing function: **ppAuto**,
2. Using three sequential wrapper functions: **ppRawData**, **ppSumEIG** and **ppFASE**, or
3. Using manual tools and functions.

For this purpose, the pre-processing part is sub-divided in three sections.

NOTE:

1. Pre-processing needs to be done using only one of the three methods described in section 4.
2. Please ensure that FASE package is installed in the system before starting this tutorial.

4. Part 1: Pre-processing

4.1. Section A: Pre-processing using automatic function

Open a linux terminal in the directory containing reference genome.

Step 1: Running bowtie to build genome index on linux terminal. This will generate six files containing genome index in the same directory as reference genome. This step needs to be performed only once for a reference genome.

```
bowtie2-build Caenorhabditis_elegans.WBcel235.dna.toplevel.fa
Caenorhabditis_elegans.WBcel235.dna.toplevel
```

Open an R session in the directory containing SRA files.

NOTE: This folder should not contain anything other than the required SRA files.

Step 2: Loading FASE.

```
library(FASE)
```

Step 3: Declaring genomeBI with the path of reference genome index. Please note that reference genome filename should not contain ‘.fa’ extension while declaring genomeBI.

```
# path <- path to folder containing indexed genome
genomeBI <- 'path/Caenorhabditis_elegans.WBcel235.dna.toplevel'
```

Step 4: Parsing the downloaded GTF file for possible junctions which is required to find intron retention events using intronGTFparser function. An intron parsed GTF file will be generated by default with ’_corrected’ appended to the GTF filename, in the same directory as downloaded GTF file. This GTF file should be used in FASE.

NOTE: intronGTFparser needs to be run only once for a GTF file.

```
# path <- path to folder containing GTF file
gtf <- intronGTFparser('path/Caenorhabditis_elegans.WBcel235.99.gtf')
```

Step 5: Declaring number of cores to be used throughout the analysis. tophat2, samtools, featureCounts, EPrnaseq and iPrnaseq allow parallel processing.

```
p <- 1
```

Step 6: Running ppAuto function. ppAuto performs:

1. SRA conversion to fastq using fastq-dump,
2. read mapping using tophat2,
3. sorting and indexing bam files using samtools,
4. generation of junction read count matrix,
5. summarization of read counts for exons, introns and genes, and
6. generation of readMembershipMatrix, intronMembershipMatrix and Gcount matrix (required detection of alternative splicing events).

After completion of ppAuto command, following files would be generated in the folder containing SRA files: eight fastq files (with _1 and _2 appended to each SRA filename), srlist.Rdata that contains list of samples, four directories (WT-R1_tophat_out, WT-R2_tophat_out, eat2-R1_tophat_out and eat2-R2_tophat_out) containing tophat output, JunctionCounts.Rdata containing read counts for mapped junctions, counts_genes.Rdata containing read counts for all mapped genes, counts_exons.Rdata containing read counts for all mapped exons, counts_introns.Rdata contining read counts of all mapped introns, RMM.data containing read membership matrix (association of exons with exons, introns and junctions (skipping and flanking)), iMM.Rdata containing intron membership matrix (association of introns with exons, introns and junctions (skipping and flanking)) and Gcount.Rdata containing per gene meta-feature read counts.

```
ppAuto(p = p, genomeBI = genomeBI, gtf = gtf, pairedend = T, files = 'sra')
```

4.2. Section B: Pre-processing using manual wrapper functions

Open a linux terminal in the directory containing reference genome.

Step 1: Running bowtie to build genome index on linux terminal. This will generate six files containing genome index in the same directory as reference genome. This step needs to be performed only once for a reference genome.

```
bowtie2-build Caenorhabditis_elegans.WBcel235.dna.toplevel.fa
Caenorhabditis_elegans.WBcel235.dna.toplevel
```

Open an R session in the directory containing SRA files.

NOTE: This folder should not contain anything other than the required SRA files.

Step 2: Loading FASE.

```
library(FASE)
```

Step 3: Declaring genomeBI with the path of reference genome index. Please note that reference genome filename should not contain '.fa' extension while declaring genomeBI.

```r
# path <- path to folder containing indexed genome
genomeBI <- 'path/Caenorhabditis_elegans.WBcel235.dna.toplevel'
```

Step 4: Parsing the downloaded GTF file for possible junctions which is required to find intron retention events using intronGTFparser function. An intron parsed GTF file will be generated by default with '_corrected' appended to the GTF filename, in the same directory as downloaded GTF file. This GTF file should be used in FASE.

NOTE: intronGTFparser needs to be run only once for a GTF file.

```r
# path <- path to folder containing GTF file
gtf <- intronGTFparser('path/Caenorhabditis_elegans.WBcel235.99.gtf')
```

Step 5: Declaring number of cores to be used throughout the analysis. tophat2, samtools, featureCounts, EPrnaseq and iPrnaseq allow parallel processing.

```r
p <- 1
```

Step 6: Calling ppRawData function. ppRawData requires genome build index and intron parsed GTF file to:

1. convert SRA files to fastq files using fastq-dump,
2. map RNA-Seq reads to the reference genome using tophat2,
3. sort and index bam files using samtools, and
4. generate junction count matrix.

After completion of ppRawData command, following files would be generated in the folder containing SRA files: eight fastq files (with _1 and _2 appended to each SRA filename), srlist.Rdata that contains list of samples, four directories (WT-R1_tophat_out, WT-R2_tophat_out, eat2-R1_tophat_out and eat2-R2_tophat_out) containing tophat output and JunctionCounts.Rdata containing read counts for mapped junctions

```r
ppRawData(pairedend = T, p = p, genomeBI = genomeBI, gtf = gtf)
```

Step 7: Calling ppSumEIG function. ppSumEIG is a wrapper function for featureCounts. It performs read count summarization for genes, exons and introns. For this, it requires srlist generated by ppRawData as srlist.Rdata, intron parsed GTF file and sorted and indexed bam files (passed automatically within the function). After completion, ppSumEIG saves three files in current directory: counts_genes.Rdata, counts_exons.Rdata and counts_introns.Rdata.

```
# srlist (a vector of samples) was automatically generated and saved as
'srlist.Rdata' by ppRawData function. For further steps, it needs to be
loaded in terminal.

load(‘srlist.Rdata’)

ppSumEIG(p = p, gtf = gtf, pairedend = T, srlist = srlist)
```

Step 8: Running ppFASE. ppFASE generates three matrices required for alternative splicing analyses: read membership matrix (RMM) which contains association of exons with exons, introns and junctions (flanking and skipping) (saved as RMM.Rdata by ppFASE), intron membership matrix which contains association of introns with exons, introns and junctions (flanking and skipping) (saved as iMM.Rdata by ppFASE) and Gcount which is a list of gene-wise read count summarization of meta-features times samples (saved as Gcount.Rdata by ppFASE). ppFASE requires summarized intron read counts (generated by ppSumEIG and saved as counts_introns.Rdata), exon read counts (generated by ppSumEIG and saved as counts_exons.Rdata), junction matrix (generated by ppRawData and saved as JunctionCounts.Rdata) and intron parsed GTF file.

```
# loading summarized read counts for exons, introns and junctions for
ppFASE function.
load(‘counts_exons.Rdata’)
load(‘counts_introns.Rdata’)
load(‘JunctionCounts.Rdata’)

ppFASE(intronCount = intronCount, exonCount = exonCount, JunctionMatrix =
JunctionMatrix, gtf = gtf)
```

4.3. Section C: Pre-processing using manual commands and functions

Open a linux terminal in the directory containing reference genome.

Step 1: Running bowtie to build genome index on linux terminal. This will generate six files containing genome index in the same directory as reference genome.

NOTE: This step needs to be performed only once for a reference genome.

```
bowtie2-build Caenorhabditis_elegans.WBcel235.dna.toplevel.fa
Caenorhabditis_elegans.WBcel235.dna.toplevel
```

Open an R session in the directory containing SRA files.

Step 2: Generating and saving srlist (vector of samples)

```
srlist <- dir()
save(srlist, file = 'srlist.Rdata')
```

Step 3: Converting sra files to fastq files.

Open a linux terminal in the directory containing SRA files. Run fastq-dump to convert SRA files to fastq files. It will generate eight fastq files (two fastq files for each SRA file as the reads are paired-end).

```
fastq-dump -I -v --split-files WT-R1 WT-R2 eat2-R1 eat2-R2
```

Step 4: Running tophat.

Open a linux terminal (or in same terminal used for fastq-dump) in the directory containing the fastq files.

```
#path <- path to folder containing indexed genome.
tophat2 -o ./WT-R1_tophat_out -p 5 -N 6 -r -44 --min-intron-length 50
--max-intron-length 5000 --mate-std-dev 30 --read-edit-dist 6
path/Caenorhabditis_elegans.WBcel235.dna.toplevel ./WT-R1_1.fastq
./WT-R1_2.fastq
tophat2 -o ./WT-R2_tophat_out -p 5 -N 6 -r -44 --min-intron-length 50
--max-intron-length 5000 --mate-std-dev 30 --read-edit-dist 6
path/Caenorhabditis_elegans.WBcel235.dna.toplevel ./WT-R2_1.fastq
./WT-R2_2.fastq
tophat2 -o ./eat2-R1_tophat_out -p 5 -N 6 -r -44 --min-intron-length 50
--max-intron-length 5000 --mate-std-dev 30 --read-edit-dist 6
path/Caenorhabditis_elegans.WBcel235.dna.toplevel ./eat2-R1_1.fastq
./eat2-R1_2.fastq
tophat2 -o ./eat2-R2_tophat_out -p 5 -N 6 -r -44 --min-intron-length 50
--max-intron-length 5000 --mate-std-dev 30 --read-edit-dist 6
path/Caenorhabditis_elegans.WBcel235.dna.toplevel ./eat2-R2_1.fastq
./eat2-R2_2.fastq
```

Step 5: Sorting bam files.

Open a linux terminal in the directory containing the tophat output. Run samtools sort for sorting the bam files generated by tophat2. It will generate a file, accepted_hits_sorted.bam, for each tophat output. The unsorted bam files, accepted_hits.bam, can be deleted after running this command. @ option in samtools sort denotes number of cores.

```
samtools sort accepted_hits.bam -o accepted_hits_sorted.bam -@ 1
# repeat this step for each tophat output
```

Step 6: Indexing the sorted bam files.

Open a linux terminal in the directory containing the tophat output. Run samtools index for indexing the sorted bam files. It will generate a file, accepted_hits_sorted.bam.bai.

```
samtools index accepted_hits_sorted.bam

# repeat this for each tophat output.
# if parallel is installed in the system, the following command can be
used instead of running samtools index for each tophat output. j in this
command denotes the number of cores to be used by parallel.

parallel -j 4 'samtools index {}'  ::: ./*/*sorted.bam
```

Step 7: Counting junction reads.

Open an R session in the directory containing fastq files.

```
#Loading FASE
library(FASE)

#Declaring folderSRA as current directory and loading the list of sample
names (srlist).
folderSRA <- getwd()
load('srlist.Rdata')

#Creating list of junctions.bed files of all samples. This list will be
passed to getJunctionCountMatrix to form junction matrix.
jfiles <- unlist(lapply(srlist, function(x) paste(folderSRA, '/' ,x
,'_tophat_out/junctions.bed' ,sep = '', collapse = "")))

#Calling getJunctionCountMatrix. This function requires list of
junctions.bed files of all samples and returns a junction matrix (read
counts of junctions) using them.
JunctionMatrix <- getJunctionCountMatrix(jfiles)

#Adding sample names to the junction matrix.
colnames(JunctionMatrix) <- c(colnames(JunctionMatrix)[1:5], srlist)
save(JunctionMatrix, file = 'JunctionCounts.Rdata')
```

Step 8: Read count summarization.

Before summarization, GTF file should be parsed for possible junctions which is required to find intron retention events using intronGTFparser function. An intron parsed GTF file will be generated by default with '_corrected' appended to the GTF filename, in the same directory as downloaded GTF file. This GTF file should be used in FASE.

NOTE: intronGTFparser needs to be run only once for a GTF file.

Continue in the previous R session (used for making junction matrix).

```
# path <- path of the folder containing GTF file.
gtf <- intronGTFparser('path/Caenorhabditis_elegans.WBcel235.99.gtf')

# Creating list of bam files (required for featureCounts)
srlistbam <-
paste(folderSRA,'/',srlist,'_tophat_out/accepted_hits_sorted.bam',sep = '')

# Declaring number of cores to be used for featureCounts.
p <- 1

# Loading Rsubread package, which contains featureCounts function
library(Rsubread)

# Read summarization for exons. Summarization step requires srlistbam,
intron parsed gtf file and number of cores. The output of summarization is
a matrix of read counts of exons which can be saved as counts_exons.Rdata
for use in analysis part.
exonCount<- featureCounts(files  =  srlistbam, isPairedEnd  =  TRUE,
requireBothEndsMapped  =  TRUE, GTF.featureType  =  "exon", GTF.attrType  =
"gene_id", useMetaFeatures  =  FALSE, isGTFAnnotationFile  =  TRUE,
annot.ext  =  gtf, allowMultiOverlap  =  TRUE, nthreads  =  p)
save(exonCount, file  =  'counts_exons.Rdata')

# Read summarization for introns. Read count summarization for introns
requires srlistbam, intron parsed gtf file and number of cores. It returns
a matrix of read counts of introns which can be saved as
counts_introns.Rdata for use in analysis part.
intronCount<- featureCounts(files  =  srlistbam, isPairedEnd  =  TRUE,
requireBothEndsMapped = TRUE, GTF.featureType  =  "intron", GTF.attrType  =
"gene_id", useMetaFeatures  =  FALSE, isGTFAnnotationFile = TRUE, annot.ext
= gtf, allowMultiOverlap = TRUE, nthreads = p)
save(intronCount, file = 'counts_introns.Rdata')

# Read summarization for genes. Summarization step requires srlistbam,
intron parsed gtf file and number of cores. It returns a matrix of read
counts of genes which can be saved as counts_genes.Rdata for use in
analysis part.
geneCount<- featureCounts(files = srlistbam, isPairedEnd = TRUE,
requireBothEndsMapped = TRUE, GTF.featureType  =  "gene", GTF.attrType  =
"gene_id", useMetaFeatures  =  FALSE, isGTFAnnotationFile = TRUE, annot.ext
= gtf, allowMultiOverlap = TRUE, nthreads = p)
```

```r
save(geneCount, file = 'counts_genes.Rdata')
```

Step 9: Generating readMembershipMatrix (RMM).

RMM contains the association of exons with other exons, introns and junctions (skipping and flanking). It is required for finding cassette exon events. It also saves Annotation.Rdata file, which contains annotation for exons and introns. readMembershipMatrix function requires intron parsed GTF file and junction matrix (JunctionCounts.Rdata). readMembershipMatrix saves RMM.Rdata by default.

Continue in the previous R session (used for making junction matrix)

```r
# Loading junction matrix.
load('JunctionCounts.Rdata')

#Running RMM function.
RMM <- readMembershipMatrix(gtf, JunctionMatrix)
```

Step 10: Generating intronMembershipMatrix (iMM).

iMM contains the association of introns with exons, introns and junctions (skipping and flanking). It is required for finding intron retention events. intronMembershipMatrix function requires annotation file generated and saved by readMembershipMatrix as Annotation.Rdata. intronMembershipMatrix saves iMM.Rdata by default.

```r
#Loading Annotation.Rdata.
load('Annotation.Rdata')

#Running iMM function
iMM <- intronMembershipMatrix(annotation = annotation)
```

Step 10: Generating Gcount matrix.

Gcount matrix is a list of gene-wise read count summarization of meta-features times samples. It is required for finding both cassette exon and intron retention events. countMatrixGenes requires the junction matrix (saved as JunctionCounts.Rdata), intron read counts (saved as counts_introns.Rdata), exon read counts (saved as counts_exons.Rdata) and annotation (saved as Annotation.Rdata by readMembershipMatrix). It saves Gcount by default as Gcount.Rdata.

```r
# Loading exon counts, intron counts, annotation and junction matrix.
load('counts_exons.Rdata')
load('counts_introns.Rdata')
load("Annotation.Rdata")
load("JunctionCounts.Rdata")

#Running Gcount function
Gcount <- countMatrixGenes(JunctionMatrix, annotation = annotation,
intronList = intronCount, exonList =  exonCount)
```

5. Part 2: Downstream Analysis

Preparing design matrix, contrast matrix and Groups vector for limma.

```
#Design matrix describes the samples in the experiment. In this tutorial,
WT-R1 and WT-R2 belong to control group and eat-R1, and eat2-R2 belong to
eat2 group.
designM <- matrix(c(rep(1,2), rep(0,4), rep(1,2)), byrow = F, ncol = 2,
nrow = 4)
colnames(designM) <- c('control', 'eat2')
rownames(designM) <- c('WT_R1', 'WT_R2', 'eat2_R1', 'eat2_R2')

#Contrast matrix describes which sample groups should be compared. In this
tutorial, eat2 is being comapred with control samples.
contrastM <- matrix(c(-1, 1), ncol = 1)
rownames(contrastM) <- colnames(designM)
colnames(contrastM) <- 'control_vs_eat2'

#Groups vector shows the groups to which the samples belong to (in correct
sequence).
Groups <- c(1,1,2,2)

#saving these three objects. These would be required in multiple
downstream analyses.
save(designM, contrastM, Groups, file = 'DCmatrix.Rdata')
```

5.1. Alternative Splicing

5.1.1. Cassette Exon Events (EP)

EPrnaseq requires Gcount matrix and RMM, generated by ppAuto/ppFASE/ readMembershipMatrix. It also requires a design matrix, contrast matrix and Groups. These matrices can be prepared as mentioned in Part 2 step 1. Threshold parameter in EPrnaseq is the minimum number of reads that should map to a meta-feature (intron/exon/junction).

```
#loading Gcount and RMM
load('Gcount.Rdata')
load('RMM.Rdata')
laod('Annotation.Rdata')

#fitting the data to EPrnaseq function, which returns a ranked list of
cassette exon events for all contrasts.
fit<- EPrnaseq(Gcount = Gcount, RMM = RMM, designM = designM, contrastM =
contrastM, Groups = Groups, p = p, threshold = 3, annotation = annotation)
```

```
#getPvaluesByContrast returns EPrnaseq ranking for a given contrast. Any
contrast from contrast matrix can be passed as shown below. In this case,
there is only one contrast 'control_vs_eat2'.
control_vs_eat2 <- getPvaluesByContrast(fit, 'control_vs_eat2')

#saving the ranking for given contrast in csv format.
write.csv(control_vs_eat2, file = 'EP_control_vs_eat2.csv')

# OPTIONAL step.
# cpmCountsEP saves CPM counts and log2CPM expression values for ranked
cassette exon events of a particular contrast (in csv format). cpmCountsEP
function should be passed with the file containing ranking for a contrast,
design matrix and Groups.
cpmCountsEP('EP_control_vs_eat2.csv', designM = designM, Groups = Groups,
Gcount = Gcount)
```

5.1.2. Alternative Splicing: Intron Retention Events (IP)

iPrnaseq requires Gcount matrix and iMM, generated by ppAuto/ppFASE/ intronMembershipMatrix. It also requires a design matrix, contrast matrix and Groups. These matrices can be prepared as mentioned in Part 2 step 1. Threshold parameter in iPrnaseq is the minimum number of reads that should map to a meta-feature (intron/exon/junction).

```
# loading Gcount and iMM
load('Gcount.Rdata')
load('iMM.Rdata')
load('Annotation.Rdata')

# fitting the data to iPrnaseq function, which returns a ranked list of
intron retention events for all contrasts.
fit<- iPrnaseq(Gcount = Gcount, iMM = iMM, designM = designM, contrastM =
contrastM, Groups = Groups, p = p, threshold = 3, annotation = annotation)

# getPvaluesByContrast returns iPrnaseq ranking for a given contrast. Any
contrast from contrast matrix can be passed as shown below. In this case,
there is only one contrast 'control_vs_eat2'.
control_vs_eat2 <- getPvaluesByContrast(fit, 'control_vs_eat2')

# saving the ranking for given contrast in csv format.
write.csv(control_vs_eat2, file = 'IP_control_vs_eat2.csv')

# OPTIONAL step.
# cpmCountsEP saves CPM counts and log2CPM expression values for ranked
intron retention events of a particular contrast (in csv format). The
function requires the file containing ranking for a contrast, design
matrix and Groups.
cpmCountsEP('IP_control_vs_eat2.csv', designM = designM, Groups = Groups,
Gcount = Gcount)
```

5.2. Transcript Structure

transtruct is the function for finding transcript structure. It reauires Gocunt, RMM, iMM, annotation, geneID, EP event ID, IP event ID, design matrix and Groups. These can be generated during pre-processing as explained in Section 4.

```
#loading  Gcount, iMM, RMM and annotation.
load('Gcount.Rdata')
load('RMM.Rdata')
load('iMM.Rdata')
load('Annotation.Rdata')
load('DCmatrix.Rdata')

#providing geneID and ep event ID
geneID <- 'WBGene00019878'
ep.event <- 'EX100238'

#extracting matrices corresponding to geneID
index_gene <- match(geneID, names(RMM))
RMM <- RMM[[index_gene]]

index_gene <- match(geneID, names(iMM))
iMM <- iMM[[index_gene]]

index_gene <- match(geneID, names(Gcount))
Gcount <- Gcount[[index_gene]]

index_ann <- match(annotation$genes, geneID)
annotation <- annotation[!is.na(index_ann),, drop = FALSE]

#defining whether or not we want to use flanking intron(s) for finding
seed exon(s). Please note that some cassette exon events are selected by
ExonPointer algorithm on the basis of differentially expressed flanking
introns. For those EP events, keep.intron parameter should be set TRUE,
otherwise transcript structures would not be generated.
keep.intron <- FALSE
```

```
#main function
ts <- transtruct(ep.event  =  ep.event , RMM  =  RMM, iMM  =  iMM, Gcount
=  Gcount, designM  =  designM, Groups  =  Groups, annotation  =
annotation, keep.intron  =  keep.intron)
```

5.3. Transcript Concentration

#transconc is the function for finding transcript concentration. It finds concentration of each transcript structure generated by transtruct, both sample-wise and condition-wise.

```
#ts <- output of transtruct function
tc <- transconc(transtruct  =  ts, designM  =  designM)
```

5.4. Survival Analysis

For survival analysis, we are using expression data from Pani et. al. (2021) and the corresponding clinical data from TCGA-BRCA IDC. Normally, clinical data needs to be filtered and matched according to the expression data. However, for this tutorial, the required files can be downloaded from https://github.com/harshsharma-cb/FASE/survivaldata.Rdata. survivaldata.Rdata contains Gcount matrix, clinical data, RMM, and exon ID for a cassette exon event in Exon8 of CerS2 gene in the LuminalB sub-type.

```
load(‘survivaldata.Rdata’)
survival.result <- survFASE(Gcount = Gcount, clinical.data = clinical.data,
rmm = RMM, imm=NULL, eventID = exonID, threshold = threshold, design =
designM, Groups = Groups)
```

5.5. Network Analysis

FASE uses alternative splicing results to generate splice index which can either directly be used for network analysis (e.g. as heat score in HotNet2) or can be modified into a similarity score to be used as edge weight for an unweighted network (e.g. ClusterONE).

For this analysis, files generated in Section 5 Step 2 and Step 3 can be used. Alternatively, these files can also be downloaded from https://github.com/harshsharma-cb/FASE/Network_analysis_files

heatscore function calculates splice index using expression data of EP and IP events. It can be used as an input to those network analysis softwares which require node parameter (score), and calculate edge weight using that score, e.g. HotNet2.

```
heatscore <- heatscore(ep = "EP_control_vs_eat2.csv" , ep.exp =
"EP_control_vs_eat2_log2cpm.csv", ip = "IP_control_vs_eat2.csv", ip.exp =
"IP_control_vs_eat2_log2cpm.csv")
```

simscore function calculates similarity score for gene pairs. It can be used in network analysis softwares which require edge weight, e.g. ClusterONE.

```
simscore <- simscore(ep = "EP_control_vs_eat2.csv" , ep.exp =
"EP_control_vs_eat2_log2cpm.csv", ip = "IP_control_vs_eat2.csv", ip.exp =
"IP_control_vs_eat2_log2cpm.csv")
```

5.6. Differentially Expressed Genes (DEG)

DEG is a wrapper function for limma, which requires summarized read counts for genes (saved as counts_genes.Rdata by ppAuto/ppSumEIG), design matrix, contrast matrix and Groups. The output of DEG is an object of MArrayLM, which stores the result of fitting gene-wise linear models to the normalized intensities or log-ratios. The output is futher processed to obtain ranking of differentially expressed genes, their CPM read counts and log2CPM expression values.

```
# loading geneCount
load('counts_genes.Rdata')

#fitting the data using DEG. It returns an object of MArrayLM with several
statistics related to differential expression like p-value, adjusted
p-value, t-statistic, etc.
fit <- DEG(geneCount = geneCount, designM = designM, contrastM = contrastM,
Groups = Groups)

#addAnnotationDEG adds annotation to the DEG fit object, according to
given contrast. The function requires geneCount (generated by
ppAuto/ppSumEIG in counts_genes.Rdata), output of DEG and a contrast from
contrast matrix. It returns ranking of differentially expressed genes for
the given contrast, which can be saved in csv format.
control_vs_eat2 <- addAnnotationDEG(geneCount, fit, 'control_vs_eat2')

#saving the ranking for given contrast in csv format.
write.csv(control_vs_eat2, file = 'DEG_control_vs_eat2.csv')

#OPTIONAL step.
#cpmCountsDEG saves CPM counts and log2CPM expression values for ranked
differentially expressed genes of a particular contrast (in csv format).
cpmCountsDEG function should be passed the file containing ranking for a
contrast and geneCount.
```

```r
cpmCountsDEG(geneCount, 'DEG_control_vs_eat2.csv', designM, contrastM,
Groups)
```

5.7. Differentially Expressed Junctions (DEJ)

DEJ is a wrapper function for limma to find differentially expressed junctions. Before adding annotation to ranked differentially expressed junction using addAnnotationDEJ, the junction matrix should be annotated using JunctionMatrixAnnotation (output is saved by default as JunctionMatrixAnnotation.Rdata). DEJ requires junction matrix (generated by ppAuto/ppRawData and saved as JunctionCounts.Rdata), design matrix, contrast matrix and Groups.

```r
#loading junction matrix generated in Section 4.
load('JunctionCounts.Rdata')

#annotating junction matrix using GTF file and junction matrix. The output
will be saved as JunctionMatrixAnnotation.Rdata.
JunctionMatrixAnnotation(gtf = gtf, JunctionMatrix = JunctionMatrix)
load('JunctionMatrixAnnotation.Rdata')

#fitting the data using DEJ. It returns an object of MArrayLM with several
statistics related to differential expression like p-value, adjusted
p-value, t-statistic, etc.
fit <- DEJ(JunctionMatrix = JunctionMatrix, designM = designM, contrastM =
contrastM, Groups = Groups)

#addAnnotationDEJ adds annotation to the DEJ fit object, according to
given contrast. addAnnotationDEJ function requires annotated junction
matrix JunctionMatrixA (generated by JunctionMatrixAnnotation as
JunctionMatrixAnnotation.Rdata), output of DEJ and a contrast from
contrast matrix. It returns ranking of differentially expressed junctions
for the given contrast, which can be saved in csv format.
control_vs_eat2 <- addAnnotationDEJ(JunctionMatrixA, fit,
'control_vs_eat2')

#saving the ranking for given contrast in csv format.
write.csv(control_vs_eat2, file = 'DEJ_control_vs_eat2.csv')

#OPTIONAL step.
#cpmCountsDEJ saves CPM counts and log2CPM expression values for ranked
differentially expressed junctions of a particular contrast (in csv
format). cpmCountsDEJ function requires the file containing ranking for a
contrast and junction matrix.
```

```r
cpmCountsDEJ(JunctionMatrix, 'DEJ_control_vs_eat2.csv', designM = designM,
contrastM = contrastM, Groups = Groups)
```