Bioconductor cheat sheet

Michael Love

Install

For details go to http://bioconductor.org/install/

```
if (!requireNamespace("BiocManager"))
    install.packages("BiocManager")
BiocManager::install()
BiocManager::install(c("package1","package2")
BiocManager::valid() # are packages up to date?

# what Bioc version is release right now?
http://bioconductor.org/bioc-version
# what Bioc versions are release/devel?
http://bioconductor.org/js/versions.js
```

help within R

Simple help:

```
?functionName
?"eSet-class" # classes need the '-class' on the end
help(package="foo",help_type="html") # launch web browser help
vignette("topic")
browseVignettes(package="package") # show vignettes for the package
```

Help for advanced users:

```
functionName # prints source code
getMethod(method,"class")  # prints source code for method
selectMethod(method, "class") # will climb the inheritance to find method
showMethods(classes="class") # show all methods for class
methods(class="GRanges") # this will work in R >= 3.2
?"functionName,class-method" # method help for S4 objects, e.g.:
?"plotMA,data.frame-method" # from library(geneplotter)
?"method.class" # method help for S3 objects e.g.:
?"plot.lm"
sessionInfo() # necessary info for getting help
packageVersion("foo") # what version of package
```

Bioconductor support website: https://support.bioconductor.org

If you use RStudio, then you already get nicely rendered documentation using `?` or `help`. If you are a command line person, then you can use this alias to pop up a help page in your web browser with `rhelp functionName packageName`.

```
alias rhelp="Rscript -e 'args <- commandArgs(TRUE); help(args[2], package=args[3], help_type=\"html\");
```

debugging R

```
traceback() # what steps lead to an error
# debug a function
debug(myFunction) # step line-by-line through the code in a function
undebug(myFunction) # stop debugging
debugonce(myFunction) # same as above, but doesn't need undebug()
# also useful if you are writing code is to put
# the function browser() inside a function at a critical point
# this plus devtools::load_all() can be useful for programming
# to jump in function on error:
options(error=recover)
# turn that behavior off:
options(error=NULL)
# debug, e.g. estimateSizeFactors from DESeq2...
# debugging an S4 method is more difficult; this gives you a peek inside:
trace(estimateSizeFactors, browser, exit=browser, signature="DESeqDataSet")
```

Show package-specific methods for a class

These two long strings of R code do approximately the same thing: obtain the methods that operate on an object of a given class, which are defined in a specific package.

```
intersect(sapply(strsplit(as.character(methods(class="DESeqDataSet")), ","), `[`, 1), ls("package:DESeq
sub("Function: (.*) \\(package .*\\)","\\1",grep("Function",showMethods(classes="DESeqDataSet", where=g
```

Annotations

For AnnotationHub examples, see:

https://www.bioconductor.org/help/workflows/annotation/Annotation_Resources

The following is how to work with the organism database packages, and biomart.

AnnotationDbi

```
# using one of the annotation packges
library(AnnotationDbi)
library(org.Hs.eg.db) # or, e.g. Homo.sapiens
columns(org.Hs.eg.db)
keytypes(org.Hs.eg.db)
head(keys(org.Hs.eg.db, keytype="ENTREZID"))
# returns a named character vector, see ?mapIds for multiVals options
res <- mapIds(org.Hs.eg.db, keys=k, column="ENSEMBL", keytype="ENTREZID")

# generates warning for 1:many mappings
res <- select(org.Hs.eg.db, keys=k,
  columns=c("ENTREZID","ENSEMBL","SYMBOL"),
  keytype="ENTREZID")
```

biomaRt

```
# map from one annotation to another using biomart
library(biomaRt)
m <- useMart("ensembl", dataset = "hsapiens_gene_ensembl")
map <- getBM(mart = m,
  attributes = c("ensembl_gene_id", "entrezgene"),
  filters = "ensembl_gene_id",
```

```
  values = some.ensembl.genes)
```

Genomic ranges

GenomicRanges

```
library(GenomicRanges)
z <- GRanges("chr1",IRanges(1000001,1001000),strand="+")
start(z)
end(z)
width(z)
strand(z)
mcols(z) # the 'metadata columns', any information stored alongside each range
ranges(z) # gives the IRanges
seqnames(z) # the chromosomes for each ranges
seqlevels(z) # the possible chromosomes
seqlengths(z) # the lengths for each chromosome
```

Intra-range methods

Affects ranges independently

function	description
shift	moves left/right
narrow	narrows by relative position within range
resize	resizes to width, fixing start for +, end for -
flank	returns flanking ranges to the left +, or right -
promoters	similar to flank
restrict	restricts ranges to a start and end position
trim	trims out of bound ranges
+/-	expands/contracts by adding/subtracting fixed amount
*	zooms in (positive) or out (negative) by multiples

Inter-range methods

Affects ranges as a group

function	description
range	one range, leftmost start to rightmost end
reduce	cover all positions with only one range
gaps	uncovered positions within range
disjoin	breaks into discrete ranges based on original starts/ends

Nearest methods

Given two sets of ranges, `x` and `subject`, for each range in `x`, returns…

function	description
nearest	index of the nearest neighbor range in subject
precede	index of the range in subject that is directly preceded by the range in x
follow	index of the range in subject that is directly followed by the range in x

function	description
distanceToNearest	distances to its nearest neighbor in subject (Hits object)
distance	distances to nearest neighbor (integer vector)

A Hits object can be accessed with `queryHits`, `subjectHits` and `mcols` if a distance is associated.

set methods

If `y` is a GRangesList, then use `punion`, etc. All functions have default `ignore.strand=FALSE`, so are strand specific.

```
union(x,y)
intersect(x,y)
setdiff(x,y)
```

Overlaps

```
x %over% y  # logical vector of which x overlaps any in y
fo <- findOverlaps(x,y) # returns a Hits object
queryHits(fo)   # which in x
subjectHits(fo) # which in y
```

Seqnames and seqlevels

GenomicRanges and GenomeInfoDb

```
gr.sub <- gr[seqlevels(gr) == "chr1"]
seqlevelsStyle(x) <- "UCSC" # convert to 'chr1' style from "NCBI" style '1'
```

Sequences

Biostrings

see the Biostrings Quick Overview PDF

For naming, see cheat sheet for annotation

```
library(BSgenome.Hsapiens.UCSC.hg19)
dnastringset <- getSeq(Hsapiens, granges) # returns a DNAStringSet
# also Views() for Bioconductor >= 3.1
```

```
library(Biostrings)
dnastringset <- readDNAStringSet("transcripts.fa")
```

```
substr(dnastringset, 1, 10) # to character string
subseq(dnastringset, 1, 10) # returns DNAStringSet
Views(dnastringset, 1, 10) # lightweight views into object
complement(dnastringset)
reverseComplement(dnastringset)
matchPattern("ACGTT", dnastring) # also countPattern, also works on Hsapiens/genome
vmatchPattern("ACGTT", dnastringset) # also vcountPattern
letterFrequecy(dnastringset, "CG") # how many C's or G's
# also letterFrequencyInSlidingView
alphabetFrequency(dnastringset, as.prob=TRUE)
# also oligonucleotideFrequency, dinucleotideFrequency, trinucleotideFrequency
# transcribe/translate for imitating biological processes
```

Sequencing data

Rsamtools `scanBam` returns lists of raw values from BAM files

```
library(Rsamtools)
which <- GRanges("chr1",IRanges(1000001,1001000))
what <- c("rname","strand","pos","qwidth","seq")
param <- ScanBamParam(which=which, what=what)
# for more BamFile functions/details see ?BamFile
# yieldSize for chunk-wise access
bamfile <- BamFile("/path/to/file.bam")
reads <- scanBam(bamfile, param=param)
res <- countBam(bamfile, param=param)
# for more sophisticated counting modes
# see summarizeOverlaps() below

# quickly check chromosome names
seqinfo(BamFile("/path/to/file.bam"))

# DNAStringSet is defined in the Biostrings package
# see the Biostrings Quick Overview PDF
dnastringset <- scanFa(fastaFile, param=granges)
```

GenomicAlignments returns Bioconductor objects (GRanges-based)

```
library(GenomicAlignments)
ga <- readGAlignments(bamfile) # single-end
ga <- readGAlignmentPairs(bamfile) # paired-end
```

Transcript databases

GenomicFeatures

```
# get a transcript database, which stores exon, trancript, and gene information
library(GenomicFeatures)
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene

# or build a txdb from GTF file (e.g. downloadable from Ensembl FTP site)
txdb <- makeTranscriptDbFromGFF("file.GTF", format="gtf")

# or build a txdb from Biomart (however, not as easy to reproduce later)
txdb <- makeTranscriptDbFromBiomart(biomart = "ensembl", dataset = "hsapiens_gene_ensembl")

# in Bioconductor >= 3.1, also makeTxDbFromGRanges

# saving and loading
saveDb(txdb, file="txdb.sqlite")
loadDb("txdb.sqlite")

# extracting information from txdb
g <- genes(txdb) # GRanges, just start to end, no exon/intron information
tx <- transcripts(txdb) # GRanges, similar to genes()
e <- exons(txdb) # GRanges for each exon
ebg <- exonsBy(txdb, by="gene") # exons grouped in a GRangesList by gene
ebt <- exonsBy(txdb, by="tx") # similar but by transcript
```

```
# then get the transcript sequence
txSeq <- extractTranscriptSeqs(Hsapiens, ebt)
```

Summarizing information across ranges and experiments

The SummarizedExperiment is a storage class for high-dimensional information tied to the same GRanges or GRangesList across experiments (e.g., read counts in exons for each gene).

```
library(GenomicAlignments)
fls <- list.files(pattern="*.bam$")
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
ebg <- exonsBy(txdb, by="gene")
# see yieldSize argument for restricting memory
bf <- BamFileList(fls)
library(BiocParallel)
register(MulticoreParam(4))
# lots of options in the man page
# singleEnd, ignore.strand, inter.features, fragments, etc.
se <- summarizeOverlaps(ebg, bf)

# operations on SummarizedExperiment
assay(se) # the counts from summarizeOverlaps
colData(se)
rowRanges(se)
```

My preferred quantification method is Salmon, with `--gcBias` option enabled unless you know there is no GC dependence in the data, followed by tximport. Here is an example of usage:

```
coldata <- read.table("samples.txt")
rownames(coldata) <- coldata$id
files <- coldata$files; names(files) <- coldata$id
txi <- tximport(files, type="salmon", tx2gene=tx2gene)
dds <- DESeqDataSetFromTximport(txi, coldata, ~condition)
```

Another fast Bioconductor read counting method is featureCounts in Rsubread.

```
library(Rsubread)
res <- featureCounts(files, annot.ext="annotation.gtf",
  isGTFAnnotationFile=TRUE,
  GTF.featureType="exon",
  GTF.attrType="gene_id")
res$counts
```

RNA-seq gene-wise analysis

DESeq2

My preferred pipeline for DESeq2 users is to start with a lightweight transcript abundance quantifier such as Salmon and to use tximport, followed by `DESeqDataSetFromTximport`.

Here, `coldata` is a *data.frame* with `group` as a column.

```
library(DESeq2)
# from tximport
dds <- DESeqDataSetFromTximport(txi, coldata, ~ group)
# from SummarizedExperiment
dds <- DESeqDataSet(se, ~ group)
```

```
# from count matrix
dds <- DESeqDataSetFromMatrix(counts, coldata, ~ group)
# minimal filtering helps keep things fast
# one can set 'n' to e.g. min(5, smallest group sample size)
keep <- rowSums(counts(dds) >= 10) >= n
dds <- dds[keep,]
dds <- DESeq(dds)
res <- results(dds) # no shrinkage of LFC, or:
res <- lfcShrink(dds, coef = 2, type="apeglm") # shrink LFCs
```

edgeR

```
# this chunk from the Quick start in the edgeR User Guide
library(edgeR)
y <- DGEList(counts=counts,group=group)
keep <- filterByExpr(y)
y <- y[keep,]
y <- calcNormFactors(y)
design <- model.matrix(~group)
y <- estimateDisp(y,design)
fit <- glmFit(y,design)
lrt <- glmLRT(fit)
topTags(lrt)
# or use the QL methods:
qlfit <- glmQLFit(y,design)
qlft <- glmQLFTest(qlfit)
topTags(qlft)
```

limma-voom

```
library(limma)
design <- model.matrix(~ group)
y <- DGEList(counts)
keep <- filterByExpr(y)
y <- y[keep,]
y <- calcNormFactors(y)
v <- voom(y,design)
fit <- lmFit(v,design)
fit <- eBayes(fit)
topTable(fit)
```

Many more RNA-seq packages

Expression set

```
library(Biobase)
data(sample.ExpressionSet)
e <- sample.ExpressionSet
exprs(e)
pData(e)
fData(e)
```

Get GEO dataset

```
library(GEOquery)
e <- getGEO("GSE9514")
```

Microarray analysis

```
library(affy)
library(limma)
phenoData <- read.AnnotatedDataFrame("sample-description.csv")
eset <- justRMA("/celfile-directory", phenoData=phenoData)
design <- model.matrix(~ Disease, pData(eset))
fit <- lmFit(eset, design)
efit <- eBayes(fit)
topTable(efit, coef=2)
```

iCOBRA performance metrics

```
library(iCOBRA)
cd <- COBRAData(pval=pval.df, padj=padj.df, score=score.df, truth=truth.df)
cp <- calculate_performance(cd, binary_truth = "status", cont_truth = "logFC")
cobraplot <- prepare_data_for_plot(cp)
plot_fdrtprcurve(cobraplot)
# interactive shiny app:
COBRAapp(cd)
```