

GenomicRanges HOWTOs

Bioconductor Team

Edited: July 2014; Compiled: July 31, 2014

Contents

1	Introduction	1
1.1	Purpose of this document	1
1.2	Prerequisites and additional recommended reading	2
1.3	Input data and terminology used across the HOWTOs	2
2	HOWTOs	2
2.1	How to read single-end reads from a BAM file	2
2.2	How to read paired-end reads from a BAM file	3
2.3	How to read and process a big BAM file by chunks in order to reduce memory usage	5
2.4	How to compute read coverage	6
2.5	How to find peaks in read coverage	7
2.6	How to retrieve a gene model from the UCSC genome browser	8
2.7	How to retrieve a gene model from Ensembl	9
2.8	How to load a gene model from a GFF or GTF file	10
2.9	How to retrieve a gene model from <i>AnnotationHub</i>	11
2.10	How to annotate peaks in read coverage	12
2.11	How to prepare a table of read counts for RNA-Seq differential gene expression	12
2.12	How to summarize junctions from a BAM file containing RNA-Seq reads	13
2.13	How to get the exon and intron sequences of a given gene	15
2.14	How to get the CDS and UTR sequences of genes associated with colorectal cancer	17
	2.14.1 Build a gene list	17
	2.14.2 Identify genomic coordinates	17
	2.14.3 Extract sequences from BSgenome	18
2.15	How to create DNA consensus sequences for read group ‘families’	19
	2.15.1 Sort reads into groups by start position	19
	2.15.2 Remove low frequency reads	21
	2.15.3 Create a consensus sequence for each read group family	22
2.16	How to compute binned averages along a genome	23
3	Session Information	25

1 Introduction

1.1 Purpose of this document

This document is a collection of *HOWTOs*. Each *HOWTO* is a short section that demonstrates how to use the containers and operations implemented in the *GenomicRanges* and related packages (*IRanges*, *Biostrings*, *Rsamtools*, *GenomicAlign*

[ments](#), [BSgenome](#), and [GenomicFeatures](#)) to perform a task typically found in the context of a high throughput sequence analysis.

Unless stated otherwise, the *HOWTOs* are self contained, independent of each other, and can be studied and reproduced in any order.

1.2 Prerequisites and additional recommended reading

We assume the reader has some previous experience with *R* and with basic manipulation of *GRanges*, *GRangesList*, *Rle*, *RleList*, and *DataFrame* objects. See the “An Introduction to Genomic Ranges Classes” vignette located in the [GenomicRanges](#) package (in the same folder as this document) for an introduction to these containers.

Additional recommended readings after this document are the “Software for Computing and Annotating Genomic Ranges” paper[?] and the “Counting reads with summarizeOverlaps” vignette located in the [GenomicAlignments](#) package.

To display the list of vignettes available in the [GenomicRanges](#) package, use `browseVignettes("GenomicRanges")`.

1.3 Input data and terminology used across the HOWTOs

In order to avoid repetition, input data, concepts and terms used in more than one *HOWTO* are described here:

- **The [pasillaBamSubset](#) data package:** contains both a BAM file with single-end reads (`untreated1_chr4`) and a BAM file with paired-end reads (`untreated3_chr4`). Each file is a subset of chr4 from the “Pasilla” experiment.


```
> library(pasillaBamSubset)
> untreated1_chr4()
[1] "/home/hpages/R/R-3.1.1/library/pasillaBamSubset/extdata/untreated1_chr4.bam"
> untreated3_chr4()
[1] "/home/hpages/R/R-3.1.1/library/pasillaBamSubset/extdata/untreated3_chr4.bam"
See ?pasillaBamSubset for more information.
> ?pasillaBamSubset
```
- **Gene models and TxDb objects:** A *gene model* is essentially a set of annotations that describes the genomic locations of the known genes, transcripts, exons, and CDS, for a given organism. In *Bioconductor* it is typically represented as a *TxDb* object but also sometimes as a *GRanges* or *GRangesList* object. The [GenomicFeatures](#) package contains tools for making and manipulating *TxDb* objects.

2 HOWTOs

2.1 How to read single-end reads from a BAM file

As sample data we use the [pasillaBamSubset](#) data package described in the introduction.

```
> library(pasillaBamSubset)
> un1 <- untreated1_chr4() # single-end reads
```

Several functions are available for reading BAM files into *R*:

```
readGAlignments()
readGAlignmentPairs()
readGAlignmentsList()
scanBam()
```

`scanBam` is a low-level function that returns a list of lists and is not discussed further here. See `?scanBam` in the [Rsamtools](#) package for more information.

Single-end reads can be loaded with the `readGAlignments` function from the [GenomicAlignments](#) package.

```
> library(GenomicAlignments)
> gal <- readGAlignments(un1)
```

Data subsets can be specified by genomic position, field names, or flag criteria in the `ScanBamParam`. Here we input records that overlap position 1 to 5000 on the negative strand with `flag` and `cigar` as metadata columns.

```
> what <- c("flag", "cigar")
> which <- GRanges("chr4", IRanges(1, 5000))
> flag <- scanBamFlag(isMinusStrand = TRUE)
> param <- ScanBamParam(which=which, what=what, flag=flag)
> neg <- readGAlignments(un1, param=param)
> neg
```

GAlignments with 37 alignments and 2 metadata columns:

	seqnames	strand	cigar	qwidth	start	end
	<Rle>	<Rle>	<character>	<integer>	<integer>	<integer>
[1]	chr4	-	75M	75	892	966
[2]	chr4	-	75M	75	919	993
[3]	chr4	-	75M	75	967	1041
...
[35]	chr4	-	75M	75	4997	5071
[36]	chr4	-	75M	75	4998	5072
[37]	chr4	-	75M	75	4999	5073

	width	njunc	flag	cigar
	<integer>	<integer>	<integer>	<character>
[1]	75	0	16	75M
[2]	75	0	16	75M
[3]	75	0	16	75M
...
[35]	75	0	16	75M
[36]	75	0	16	75M
[37]	75	0	16	75M

seqlengths:

chr2L	chr2R	chr3L	chr3R	chr4	chrM	chrX	chrYHet
23011544	21146708	24543557	27905053	1351857	19517	22422827	347038

Another approach to subsetting the data is to use `filterBam`. This function creates a new BAM file of records passing user-defined criteria. See `?filterBam` in the *Rsamtools* package for more information.

2.2 How to read paired-end reads from a BAM file

As sample data we use the *pasillaBamSubset* data package described in the introduction.

```
> library(pasillaBamSubset)
> un3 <- untreated3_chr4() # paired-end reads
```

Paired-end reads can be loaded with the `readGAlignmentPairs` or `readGAlignmentsList` function from the *GenomicAlignments* package. These functions use the same mate pairing algorithm but output different objects.

Let's start with `readGAlignmentPairs`:

```
> un3 <- untreated3_chr4()
> gapairs <- readGAlignmentPairs(un3)
```

The `GAlignmentPairs` class holds only pairs; reads with no mate or with ambiguous pairing are discarded. Each list element holds exactly 2 records (a mated pair). Records can be accessed as the `first` and `last` segments in a template or as `left` and `right` alignments. See `?GAlignmentPairs` in the *GenomicAlignments* package for more information.

```
> gapairs
```

```
GAlignmentPairs with 75346 alignment pairs and 0 metadata columns:
```

	seqnames	strand	:	ranges	--	ranges
	<Rle>	<Rle>	:	<IRanges>	--	<IRanges>
[1]	chr4	+	:	[169, 205]	--	[326, 362]
[2]	chr4	+	:	[943, 979]	--	[1086, 1122]
[3]	chr4	+	:	[944, 980]	--	[1119, 1155]
...
[75344]	chr4	+	:	[1348217, 1348253]	--	[1348215, 1348251]
[75345]	chr4	+	:	[1349196, 1349232]	--	[1349326, 1349362]
[75346]	chr4	+	:	[1349708, 1349744]	--	[1349838, 1349874]

```
---
```

```
seqlengths:
```

chr2L	chr2R	chr3L	chr3R	chr4	chrM	chrX	chrYHet
23011544	21146708	24543557	27905053	1351857	19517	22422827	347038

For readGAlignmentsList, mate pairing is performed when asMates is set to TRUE on the BamFile object, otherwise records are treated as single-end.

```
> galist <- readGAlignmentsList(BamFile(un3, asMates=TRUE))
```

GAlignmentsList is a more general 'list-like' structure that holds mate pairs as well as non-mates (i.e., singletons, records with unmapped mates etc.) A mates metadata column (accessed with mcols) indicates which records were paired and is set on both the individual GAlignments and the outer list elements.

```
> galist
```

```
GAlignmentsList of length 96632:
```

```
[[1]]
```

```
GAlignments with 2 alignments and 0 metadata columns:
```

	seqnames	strand	cigar	qwidth	start	end	width	njunc
[1]	chr4	+	37M	37	169	205	37	0
[2]	chr4	-	37M	37	326	362	37	0

```
[[2]]
```

```
GAlignments with 2 alignments and 0 metadata columns:
```

	seqnames	strand	cigar	qwidth	start	end	width	njunc
[1]	chr4	+	37M	37	946	982	37	0
[2]	chr4	-	37M	37	986	1022	37	0

```
[[3]]
```

```
GAlignments with 2 alignments and 0 metadata columns:
```

	seqnames	strand	cigar	qwidth	start	end	width	njunc
[1]	chr4	+	37M	37	943	979	37	0
[2]	chr4	-	37M	37	1086	1122	37	0

```
...
```

```
<96629 more elements>
```

```
---
```

```
seqlengths:
```

chr2L	chr2R	chr3L	chr3R	chr4	chrM	chrX	chrYHet
23011544	21146708	24543557	27905053	1351857	19517	22422827	347038

Non-mated reads are returned as groups by QNAME and contain any number of records. Here the non-mate groups range in size from 1 to 9.

```
> non_mates <- galist[unlist(mcols(galist)$mates) == FALSE]
> table(elementLengths(non_mates))
< table of extent 0 >
```

2.3 How to read and process a big BAM file by chunks in order to reduce memory usage

A large BAM file can be iterated through in chunks by setting a `yieldSize` on the *BamFile* object. As sample data we use the *pasillaBamSubset* data package described in the introduction.

```
> library(pasillaBamSubset)
> un1 <- untreated1_chr4()
> bf <- BamFile(un1, yieldSize=100000)
```

Iteration through a BAM file requires that the file be opened, repeatedly queried inside a loop, then closed. Repeated calls to `readGAlignments` without opening the file first result in the same 100000 records returned each time.

```
> open(bf)
> cvg <- NULL
> repeat {
+   chunk <- readGAlignments(bf)
+   if (length(chunk) == 0L)
+     break
+   chunk_cvg <- coverage(chunk)
+   if (is.null(cvg)) {
+     cvg <- chunk_cvg
+   } else {
+     cvg <- cvg + chunk_cvg
+   }
+ }
> close(bf)
> cvg
```

RleList of length 8

\$chr2L

integer-Rle of length 23011544 with 1 run
 Lengths: 23011544
 Values : 0

\$chr2R

integer-Rle of length 21146708 with 1 run
 Lengths: 21146708
 Values : 0

\$chr3L

integer-Rle of length 24543557 with 1 run
 Lengths: 24543557
 Values : 0

\$chr3R

integer-Rle of length 27905053 with 1 run
 Lengths: 27905053
 Values : 0

\$chr4

integer-Rle of length 1351857 with 122061 runs

```

Lengths: 891 27 5 12 13 45 ... 106 75 1600 75 1659
Values : 0 1 2 3 4 5 ... 0 1 0 1 0

...
<3 more elements>

```

2.4 How to compute read coverage

The “read coverage” is the number of reads that cover a given genomic position. Computing the read coverage generally consists in computing the coverage at each position in the genome. This can be done with the `coverage()` function.

As sample data we use the [pasillaBamSubset](#) data package described in the introduction.

```

> library(pasillaBamSubset)
> un1 <- untreated1_chr4() # single-end reads
> library(GenomicAlignments)
> reads1 <- readGAlignments(un1)
> cvg1 <- coverage(reads1)
> cvg1

RleList of length 8
$chr2L
integer-Rle of length 23011544 with 1 run
  Lengths: 23011544
  Values : 0

$chr2R
integer-Rle of length 21146708 with 1 run
  Lengths: 21146708
  Values : 0

$chr3L
integer-Rle of length 24543557 with 1 run
  Lengths: 24543557
  Values : 0

$chr3R
integer-Rle of length 27905053 with 1 run
  Lengths: 27905053
  Values : 0

$chr4
integer-Rle of length 1351857 with 122061 runs
  Lengths: 891 27 5 12 13 45 ... 106 75 1600 75 1659
  Values : 0 1 2 3 4 5 ... 0 1 0 1 0

...
<3 more elements>

Coverage on chr4:
> cvg1$chr4

integer-Rle of length 1351857 with 122061 runs
  Lengths: 891 27 5 12 13 45 ... 106 75 1600 75 1659
  Values : 0 1 2 3 4 5 ... 0 1 0 1 0

```

Average and max coverage:

```
> mean(cvg1$chr4)
[1] 11.33746
> max(cvg1$chr4)
[1] 5627
```

Note that `coverage()` is a generic function with methods for different types of objects. See `?coverage` for more information.

2.5 How to find peaks in read coverage

ChIP-Seq analysis usually involves finding peaks in read coverage. This process is sometimes called “peak calling” or “peak detection”. Here we’re only showing a naive way to find peaks in the object returned by the `coverage()` function. *Bioconductor* packages *BayesPeak*, *bumphunter*, *Starr*, *CexoR*, *exomePeak*, *RIPSeeker*, and others, provide sophisticated peak calling tools for ChIP-Seq, RIP-Seq, and other kind of high throughput sequencing data.

Let’s assume `cvg1` is the object returned by `coverage()` (see previous *HOWTO* for how to compute it). We can use the `slice()` function to find the genomic regions where the coverage is greater or equal to a given threshold.

```
> chr4_peaks <- slice(cvg1$chr4, lower=500)
> chr4_peaks
```

Views on a 1351857-length Rle subject

views:

```
      start      end width
[1]  86849   87364   516 [ 525  538  554  580  583  585  589 ...]
[2]  87466   87810   345 [4924 4928 4941 4943 4972 5026 5039 ...]
[3] 340791  340798     8 [508 512 506 530 521 519 518 501]
[4] 340800  340885    86 [500 505 560 560 565 558 564 559 555 ...]
[5] 348477  348483     7 [503 507 501 524 515 513 512]
[6] 348488  348571    84 [554 554 559 552 558 553 549 550 559 ...]
[7] 692512  692530    19 [502 507 508 518 520 522 524 526 547 ...]
[8] 692551  692657   107 [ 530  549  555  635  645  723  725 ...]
[9] 692798  692800     3 [503 500 503]
...     ...     ...   ... ...
[34] 1054306 1054306     1 [502]
[35] 1054349 1054349     1 [501]
[36] 1054355 1054444    90 [510 521 525 532 532 539 549 555 557 ...]
[37] 1054448 1054476    29 [502 507 516 517 508 517 525 528 532 ...]
[38] 1054479 1054482     4 [504 503 506 507]
[39] 1054509 1054509     1 [500]
[40] 1054511 1054511     1 [502]
[41] 1054521 1054623   103 [529 521 529 530 524 525 547 540 536 ...]
[42] 1054653 1054717    65 [520 519 516 528 526 585 591 589 584 ...]
```

```
> length(chr4_peaks) # nb of peaks
[1] 42
```

The weight of a given peak can be defined as the number of aligned nucleotides that belong to the peak (a.k.a. the area under the peak in mathematics). It can be obtained with `sum()`:

```
> sum(chr4_peaks)
```

```
[1] 1726347 1300700 4115 52301 3575 51233 10382 95103
[9] 1506 500 2051 500 5834 10382 92163 500
[17] 88678 1512 500 11518 14514 5915 3598 7821
[25] 511 508 503 500 1547 8961 43426 22842
[33] 503 502 501 51881 15116 2020 500 502
[41] 67010 40496
```

2.6 How to retrieve a gene model from the UCSC genome browser

See introduction for a quick description of what *gene models* and *TxDb* objects are. We can use the `makeTranscriptDbFromUCSC()` function from the *GenomicFeatures* package to import a UCSC genome browser track as a *TxDb* object.

```
> library(GenomicFeatures)
> ### Internet connection required! Can take several minutes...
> txdb <- makeTranscriptDbFromUCSC(genome="sacCer2", tablename="ensGene")
```

See `?makeTranscriptDbFromUCSC` in the *GenomicFeatures* package for more information.

Note that some of the most frequently used gene models are available as *TxDb* packages. A *TxDb* package consists of a pre-made *TxDb* object wrapped into an annotation data package. Go to http://bioconductor.org/packages/release/BiocViews.html#___TxDb to browse the list of available *TxDb* packages.

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> txdb
```

TxDb object:

```
| Db type: TranscriptDb
| Supporting package: GenomicFeatures
| Data source: UCSC
| Genome: hg19
| Organism: Homo sapiens
| UCSC Table: knownGene
| Resource URL: http://genome.ucsc.edu/
| Type of Gene ID: Entrez Gene ID
| Full dataset: yes
| miRBase build ID: GRCh37
| transcript_nrow: 82960
| exon_nrow: 289969
| cds_nrow: 237533
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2014-03-17 16:15:59 -0700 (Mon, 17 Mar 2014)
| GenomicFeatures version at creation time: 1.15.11
| RSQLite version at creation time: 0.11.4
| DBSCHEMAVERSION: 1.0
```

Extract the transcript coordinates from this gene model:

```
> transcripts(txdb)
```

GRanges with 82960 ranges and 2 metadata columns:

```
      seqnames      ranges strand |      tx_id
      <Rle>      <IRanges> <Rle> | <integer>
[1]   chr1      [11874, 14409]   +   |         1
[2]   chr1      [11874, 14409]   +   |         2
[3]   chr1      [11874, 14409]   +   |         3
...     ...                ...   ...     ...
```

```

[82958] chrY [27607404, 27607432] - | 78805
[82959] chrY [27635919, 27635954] - | 78806
[82960] chrY [59358329, 59360854] - | 78807
tx_name
<character>
[1] uc001aaa.3
[2] uc010nxq.1
[3] uc010nxr.1
...
[82958] uc004fwz.3
[82959] uc022cpd.1
[82960] uc011ncc.1
---
seqlengths:
      chr1          chr2 ...      chrUn_g1000249
249250621      243199373 ...      38502

```

2.7 How to retrieve a gene model from Ensembl

See introduction for a quick description of what *gene models* and *TxDb* objects are. We can use the `makeTranscriptDbFromBiomart()` function from the *GenomicFeatures* package to retrieve a gene model from the Ensembl Mart.

```

> library(GenomicFeatures)
> ### Internet connection required! Can take several minutes...
> txdb <- makeTranscriptDbFromBiomart(biomart="ensembl",
+                                   dataset="hsapiens_gene_ensembl")

```

See `?makeTranscriptDbFromBiomart` in the *GenomicFeatures* package for more information.

Note that some of the most frequently used gene models are available as *TxDb* packages. A *TxDb* package consists of a pre-made *TxDb* object wrapped into an annotation data package. Go to http://bioconductor.org/packages/release/BiocViews.html#___TxDb to browse the list of available *TxDb* packages.

```

> library(TxDb.Athaliana.BioMart.plantsmart21)
> txdb <- TxDb.Athaliana.BioMart.plantsmart21
> txdb

```

TxDb object:

```

| Db type: TranscriptDb
| Supporting package: GenomicFeatures
| Data source: BioMart
| Organism: Arabidopsis thaliana
| Resource URL: www.biomart.org:80
| BioMart database: plants_mart_21
| BioMart database version: ENSEMBL PLANTS 21 (EBI UK)
| BioMart dataset: athaliana_eg_gene
| BioMart dataset description: Arabidopsis thaliana genes (TAIR10 (2010-09-TAIR10))
| BioMart dataset version: TAIR10 (2010-09-TAIR10)
| Full dataset: yes
| miRBase build ID: NA
| transcript_nrow: 41671
| exon_nrow: 171013
| cds_nrow: 0
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2014-03-17 16:26:40 -0700 (Mon, 17 Mar 2014)
| GenomicFeatures version at creation time: 1.15.11

```

```
| RSQLite version at creation time: 0.11.4
| DBSCHEMAVERSION: 1.0
```

Extract the exon coordinates from this gene model:

```
> exons(txdb)

GRanges with 171013 ranges and 1 metadata column:
      seqnames      ranges strand | exon_id
      <Rle>        <IRanges> <Rle> | <integer>
 [1]          1      [3631, 3913]   + |         1
 [2]          1      [3996, 4276]   + |         2
 [3]          1      [4486, 4605]   + |         3
 ...          ...          ...     ... |         ...
[171011]      Pt [137869, 137940]   - |      171011
[171012]      Pt [144921, 145154]   - |      171012
[171013]      Pt [145291, 152175]   - |      171013
---
seqlengths:
  1  2  3  4  5 Mt Pt
NA NA NA NA NA NA NA
```

2.8 How to load a gene model from a GFF or GTF file

See introduction for a quick description of what *gene models* and *TxDb* objects are. We can use the `makeTranscriptDbFromGFF()` function from the [GenomicFeatures](#) package to import a GFF or GTF file as a *TxDb* object.

```
> library(GenomicFeatures)
> gff_file <- system.file("extdata", "a.gff3", package="GenomicFeatures")
> txdb <- makeTranscriptDbFromGFF(gff_file, format="gff3")
> txdb

TxDb object:
| Db type: TxDb
| Supporting package: GenomicFeatures
| Data source: /home/hpages/R/R-3.1.1/library/GenomicFeatures/extdata/a.gff3
| Organism: NA
| miRBase build ID: NA
| transcript_nrow: 488
| exon_nrow: 1268
| cds_nrow: 1258
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2014-07-31 11:09:34 -0700 (Thu, 31 Jul 2014)
| GenomicFeatures version at creation time: 1.17.13
| RSQLite version at creation time: 0.11.4
| DBSCHEMAVERSION: 1.0
```

See `?makeTranscriptDbFromGFF` in the [GenomicFeatures](#) package for more information.

Extract the exon coordinates grouped by gene from this gene model:

```
> exonsBy(txdb, by="gene")

GRangesList of length 488:
$gene:Solyc00g005000.2
GRanges with 2 ranges and 2 metadata columns:
      seqnames      ranges strand | exon_id
      <Rle>        <IRanges> <Rle> | <integer>
```

```
[1] SL2.40ch00 [16437, 17275] + | 1
[2] SL2.40ch00 [17336, 18189] + | 2
      exon_name
      <character>
[1] exon:Solyc00g005000.2.1.1
[2] exon:Solyc00g005000.2.1.2
```

```
$gene:Solyc00g005020.1
```

```
GRanges with 3 ranges and 2 metadata columns:
```

```
      seqnames      ranges strand | exon_id
[1] SL2.40ch00 [68062, 68211] + | 3
[2] SL2.40ch00 [68344, 68568] + | 4
[3] SL2.40ch00 [68654, 68764] + | 5
      exon_name
[1] exon:Solyc00g005020.1.1.1
[2] exon:Solyc00g005020.1.1.2
[3] exon:Solyc00g005020.1.1.3
```

```
$gene:Solyc00g005040.2
```

```
GRanges with 4 ranges and 2 metadata columns:
```

```
      seqnames      ranges strand | exon_id
[1] SL2.40ch00 [550920, 550945] + | 6
[2] SL2.40ch00 [551034, 551132] + | 7
[3] SL2.40ch00 [551218, 551250] + | 8
[4] SL2.40ch00 [551343, 551576] + | 9
      exon_name
[1] exon:Solyc00g005040.2.1.1
[2] exon:Solyc00g005040.2.1.2
[3] exon:Solyc00g005040.2.1.3
[4] exon:Solyc00g005040.2.1.4
```

```
...
```

```
<485 more elements>
```

```
---
```

```
seqlengths:
```

```
SL2.40ch00
      NA
```

2.9 How to retrieve a gene model from AnnotationHub

When a gene model is not available as a *GRanges* or *GRangesList* object or as a *Bioconductor* data package, it may be available on [AnnotationHub](#). In this *HOWTO*, will look for a gene model for *Drosophila melanogaster* on [AnnotationHub](#). Create a 'hub' and filter on *Drosophila melanogaster*:

```
> library(AnnotationHub)
> ### Internet connection required!
> hub <- AnnotationHub()
> filters(hub) <- list(Species="Drosophila melanogaster")
```

There are 87 files that match *Drosophila melanogaster*.

```
> length(hub)
> head(names(hub))
```

Retrieve a dm3 file as a *GRanges*.

```
> gr <- hub$goldenpath.dm3.database.ensGene_0.0.1.RData
> summary(gr)
```

The metadata fields contain the details of file origin and content.

```
> names(metadata(gr)[[2]])
> metadata(gr)[[2]]$Tags
```

Split the *GRanges* object by gene name to get a *GRangesList* object of transcript ranges grouped by gene.

```
> txbygn <- split(gr, gr$name)
```

You can now use *txbygn* with the *summarizeOverlaps* function to prepare a table of read counts for RNA-Seq differential gene expression.

Note that before passing *txbygn* to *summarizeOverlaps*, you should confirm that the *seqlevels* (chromosome names) in it match those in the BAM file. See *?renameSeqlevels*, *?keepSeqlevels* and *?seqlevels* for examples of renaming *seqlevels*.

2.10 How to annotate peaks in read coverage

[coming soon...]

2.11 How to prepare a table of read counts for RNA-Seq differential gene expression

Methods for RNA-Seq gene expression analysis generally require a table of counts that summarize the number of reads that overlap or 'hit' a particular gene. In this *HOWTO* we count with the *summarizeOverlaps* function from the *GenomicAlignments* package and create a count table from the results.

Other packages that provide read counting are *Rsubread* and *easyRNASeq*. The *parathyroidSE* package vignette contains a workflow on counting and other common operations required for differential expression analysis.

As sample data we use the *pasillaBamSubset* data package described in the introduction.

```
> library(pasillaBamSubset)
> un1 <- untreated1_chr4() # single-end reads
```

summarizeOverlaps requires the name of a BAM file(s) and a *gene model* to count against. See introduction for a quick description of what a *gene models* is. The gene model must match the genome build the reads in the BAM file were aligned to. For the *pasilla* data this is *dm3 Dmelanogaster* which is available as a *Bioconductor* package. Load the package and extract the exon ranges grouped by gene:

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> exbygene <- exonsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene, "gene")
```

exbygene is a *GRangesList* object with one list element per gene in the gene model.

summarizeOverlaps automatically sets a *yieldSize* on large BAM files and iterates over them in chunks. When reading paired-end data set the *singleEnd* argument to *FALSE*. See *?summarizeOverlaps* for details regarding the count modes and additional arguments.

```
> library(GenomicAlignments)
> se <- summarizeOverlaps(exbygene, un1, mode="IntersectionNotEmpty")
```

The return object is a *SummarizedExperiment* with counts in the *assays* slot.

```
> class(se)

[1] "SummarizedExperiment"
attr(,"package")
[1] "GenomicRanges"
```

```
> head(table(assays(se)$counts))
```

```
  0    1    2    3    4    5
15593  1    3    1    4    1
```

The count vector is the same length as exbygene:

```
> identical(length(exbygene), length(assays(se)$counts))
```

```
[1] TRUE
```

A copy of exbygene is stored in the rowData slot:

```
> rowData(se)
```

```
GRangesList of length 15682:
```

```
$FBgn0000003
```

```
GRanges with 1 range and 2 metadata columns:
```

	seqnames	ranges	strand	exon_id	exon_name
	<Rle>	<IRanges>	<Rle>	<integer>	<character>
[1]	chr3R	[2648220, 2648518]	+	45123	<NA>

```
$FBgn0000008
```

```
GRanges with 13 ranges and 2 metadata columns:
```

	seqnames	ranges	strand	exon_id	exon_name
[1]	chr2R	[18024494, 18024531]	+	20314	<NA>
[2]	chr2R	[18024496, 18024713]	+	20315	<NA>
[3]	chr2R	[18024938, 18025756]	+	20316	<NA>
...
[11]	chr2R	[18059821, 18059938]	+	20328	<NA>
[12]	chr2R	[18060002, 18060339]	+	20329	<NA>
[13]	chr2R	[18060002, 18060346]	+	20330	<NA>

```
...
```

```
<15680 more elements>
```

```
---
```

```
seqlengths:
```

	chr2L	chr2R	chr3L	...	chrXHet	chrYHet	chrUextra
	23011544	21146708	24543557	...	204112	347038	29004656

Two popular packages for RNA-Seq differential gene expression are *DESeq* and *edgeR*. Tables of counts per gene are required for both and can be easily created with a vector of counts. Here we use the counts from our *SummarizedExperiment* object:

```
> library(DESeq)
```

```
> deseq <- newCountDataSet(assays(se)$counts, rownames(colData(se)))
```

```
> library(edgeR)
```

```
> edger <- DGEList(assays(se)$counts, group=rownames(colData(se)))
```

2.12 How to summarize junctions from a BAM file containing RNA-Seq reads

As sample data we use the *pasillaBamSubset* data package described in the introduction.

```
> library(pasillaBamSubset)
```

```
> un1 <- untreated1_chr4() # single-end reads
```

```
> library(GenomicAlignments)
```

```
> reads1 <- readGAlignments(un1)
```

```
> reads1
```

```

GAlignments with 204355 alignments and 0 metadata columns:
      seqnames strand      cigar      qwidth      start      end
      <Rle> <Rle> <character> <integer> <integer> <integer>
 [1]   chr4     -      75M         75         892         966
 [2]   chr4     -      75M         75         919         993
 [3]   chr4     +      75M         75         924         998
 ...     ...     ...     ...     ...     ...
[204353] chr4     +      75M         75    1348268    1348342
[204354] chr4     -      75M         75    1348449    1348523
[204355] chr4     -      75M         75    1350124    1350198
      width      njunc
      <integer> <integer>
 [1]         75         0
 [2]         75         0
 [3]         75         0
 ...     ...     ...
[204353]     75         0
[204354]     75         0
[204355]     75         0

```

```
---
```

```
seqlengths:
```

```

chr2L  chr2R  chr3L  chr3R  chr4  chrM  chrX  chrYHet
23011544 21146708 24543557 27905053 1351857 19517 22422827 347038

```

For each alignment, the aligner generated a CIGAR string that describes its "geometry", that is, the locations of insertions, deletions and junctions in the alignment. See the SAM Spec available on the SAMtools website for the details (<http://samtools.sourceforge.net/>).

The `summarizeJunctions()` function from the *GenomicAlignments* package can be used to summarize the junctions in `reads1`.

```

> junc_summary <- summarizeJunctions(reads1)
> junc_summary

```

```

GRanges with 910 ranges and 3 metadata columns:
      seqnames      ranges strand |      score plus_score
      <Rle>      <IRanges> <Rle> | <integer> <integer>
 [1]   chr4  [ 5246, 11972]   * |         3         1
 [2]   chr4  [10346, 10637]   * |         1         1
 [3]   chr4  [27102, 27166]   * |        13         11
 ...     ...     ...     ...     ...     ...
[908]   chr4 [1333752, 1346734] * |         1         0
[909]   chr4 [1334150, 1347141] * |         1         1
[910]   chr4 [1334557, 1347539] * |         1         0
      minus_score
      <integer>
 [1]         2
 [2]         0
 [3]         2
 ...     ...
[908]         1
[909]         0
[910]         1

```

```
---
```

```
seqlengths:
```

```

chr2L  chr2R  chr3L  chr3R  chr4  chrM  chrX  chrYHet

```

```
23011544 21146708 24543557 27905053 1351857 19517 22422827 347038
```

See `?summarizeJunctions` in the [GenomicAlignments](#) package for more information.

2.13 How to get the exon and intron sequences of a given gene

The exon and intron sequences of a gene are essentially the DNA sequences of the introns and exons of all known transcripts of the gene. The first task is to identify all transcripts associated with the gene of interest. Our sample gene is the human TRAK2 which is involved in regulation of endosome-to-lysosome trafficking of membrane cargo. The Entrez gene id is '66008'.

```
> trak2 <- "66008"
```

The [TxDb.Hsapiens.UCSC.hg19.knownGene](#) data package contains the gene model corresponding to the UCSC 'Known Genes' track.

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
```

The transcript ranges for all the genes in the gene model can be extracted with the `transcriptsBy` function from the [GenomicFeatures](#) package. They will be returned in a named *GRangesList* object containing all the transcripts grouped by gene. In order to keep only the transcripts of the TRAK2 gene we will subset the *GRangesList* object using the `[[` operator.

```
> library(GenomicFeatures)
> trak2_txs <- transcriptsBy(txdb, by="gene")[[trak2]]
> trak2_txs
```

GRanges with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	tx_id	tx_name
	<Rle>	<IRanges>	<Rle>	<integer>	<character>
[1]	chr2	[202241930, 202316319]	-	12552	uc002uyb.4
[2]	chr2	[202259851, 202316319]	-	12553	uc002uyc.2

seqlengths:

chr1	chr2 ...	chrUn_gl000249
249250621	243199373 ...	38502

`trak2_txs` is a *GRanges* object with one range per transcript in the TRAK2 gene. The transcript names are stored in the `tx_name` metadata column. We will need them to subset the extracted intron and exon regions:

```
> trak2_tx_names <- mcols(trak2_txs)$tx_name
> trak2_tx_names
```

```
[1] "uc002uyb.4" "uc002uyc.2"
```

The exon and intron genomic ranges for all the transcripts in the gene model can be extracted with the `exonsBy` and `intronsByTranscript` functions, respectively. Both functions return a *GRangesList* object. Then we keep only the exon and intron for the transcripts of the TRAK2 gene by subsetting each *GRangesList* object by the TRAK2 transcript names.

Extract the exon regions:

```
> trak2_exbytx <- exonsBy(txdb, "tx", use.names=TRUE)[trak2_tx_names]
> elementLengths(trak2_exbytx)
```

```
uc002uyb.4 uc002uyc.2
      16          8
```

... and the intron regions:

```
> trak2_inbytx <- intronsByTranscript(txdb, use.names=TRUE)[trak2_tx_names]
> elementLengths(trak2_inbytx)
```

```
uc002uyb.4 uc002uyc.2
      15         7
```

Next we want the DNA sequences for these exons and introns. The `getSeq` function from the *Biostrings* package can be used to query a *BSgenome* object with a set of genomic ranges and retrieve the corresponding DNA sequences.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
```

Extract the exon sequences:

```
> trak2_ex_seqs <- getSeq(Hsapiens, trak2_exbytx)
> trak2_ex_seqs
```

DNASTringSetList of length 2

```
[["uc002uyb.4"]] GCTGGGAGAGTGGCTCTCCTTTGGCTTCCCAATTGTGTGGGGGCTGCCATT...
[["uc002uyc.2"]] GCTGGGAGAGTGGCTCTCCTTTGGCTTCCCAATTGTGTGGGGGCTGCCATT...
```

```
> trak2_ex_seqs[["uc002uyb.4"]]
```

A DNASTringSet instance of length 16

width seq

```
[1] 247 GCTGGGAGAGTGGCTCTCCTTTGGCTTCC...CGGACGACAGAGGATGCCGAACCACTCCA
[2] 290 GTCATGACTGTCCAAAGTATGATAATCAC...CAATCACAGAGACTCGGAGAGCATCACTG
[3] 195 ATGTCTGCTCCAATGAGGATCTCCCTGAA...CCTTGCTGAAGAGACTTCCGTTACATGA
... ..
[14] 267 GATCACAAACTCTGTATCACTGGCAGCAG...CATTACTTCAGCAGGTGGACCAGTTACAG
[15] 106 TTGCAACCGCCAACCCAGAAAGTGCCTG...CCCTCTGACATCACTCAGTTACCCCCAG
[16] 4012 CTCTGGGTTCCCTTCAATTATCCTGTGAA...TTAATAAACATGAGTAGCTGAATTTCA
```

```
> trak2_ex_seqs[["uc002uyc.2"]]
```

A DNASTringSet instance of length 8

width seq

```
[1] 247 GCTGGGAGAGTGGCTCTCCTTTGGCTTCC...CGGACGACAGAGGATGCCGAACCACTCCA
[2] 290 GTCATGACTGTCCAAAGTATGATAATCACA...CAATCACAGAGACTCGGAGAGCATCACTG
[3] 195 ATGTCTGCTCCAATGAGGATCTCCCTGAA...CCTTGCTGAAGAGACTTCCGTTACATGA
[4] 77 TTCTAGGCACAGACAGGGTGGAGCAGATGA...TCGACATGGTTACACATCTCCTGGCAGAG
[5] 117 AGGGATCGTGATCTGGAACCTCGCTGCTCGA...AGGAGCAATTGGGACAAGCCTTTGATCAA
[6] 210 GTTAATCAGCTGCAGCATGAGCTATGCAAG...AGAAGAGAATATGGCTCTTCGATCCAAG
[7] 79 GCTTGTACATAAAGACAGAAACTGTTACC...GCTTGTGCGGACTGTGTTAAAGAACTTC
[8] 317 GTGAAACAAATGCTCAGATGTCCAGAATGA...AGATATCATGAATAAATACTTTCAAGTCA
```

... and the intron sequences:

```
> trak2_in_seqs <- getSeq(Hsapiens, trak2_inbytx)
> trak2_in_seqs
```

DNASTringSetList of length 2

```
[["uc002uyb.4"]] GTAAGAGTGCCTGGGAAATCTGGGGCCTCACTTCTTCTCAGCTATATTTT...
[["uc002uyc.2"]] GTGAGTATTAACATATTCTCTTTGTACCTTTTTGGACAATTCTTTGGTAGG...
```

```
> trak2_in_seqs[["uc002uyb.4"]]
```

A DNASTringSet instance of length 15

width seq

```
[1] 2892 GTAAGAGTGCCTGGGAAATCTGGGGCCTC...GTCTCCCACTTTTTTTTTTTTTTTTAAAG
[2] 2001 GTGAGAAGAGTGTCTGGTTGAATATGTA...TGTATTTGCTCCCTAAAAATCTATTTAG
[3] 1218 GTAATAAATCAGTAAGGGCCCTTACTAAG...TTCCCTTCTTTGTTTGCATATTAG
... ..
```

```
[13] 6308 GTGAGTATTTTTTTACTCTTTTAGTTG...CTATAAATAGTTGTTTTTAACTATATTAG
[14] 12819 GTAAGTCCAGTTAATAAATATTGAAGTG...GATTCATTTACATAGACTCTCCTCTTTAG
[15] 30643 GTGAGTAAGCTGTCCGCGCAGAACCCGAA...GTTCTAGTCACTTGATGTTTTTGTTTTAG

> trak2_in_seqs[["uc002uyc.2"]]

A DNASTringSet instance of length 7
width seq
[1] 2057 GTGAGTATTAACATATTCTCTTTGTACCT...AATTTAAAAAATTTTTTTGCTTCCAAG
[2] 564 GTACGTTCAACCTAATTGCCATTTTCCTT...ATTGTCACATACTGATTTTTTTCTTGAAG
[3] 1022 GTAAGCCTTTGATCAAATGTCTGCAGTATG...CATGAAAATCAAGCATTTTATATGGACAG
[4] 1524 GTAGGAATATCTTTTCTTTCTCCAGTACAA...AAGAAAAGGTGATTTGGTATTTTAAACAG
[5] 6308 GTGAGTATTTTTTTACTCTTTTAGTTTGT...CTATAAATAGTTGTTTTTAACTATATTAG
[6] 12819 GTAAGTCCAGTTAATAAATATTGAAGTGC...GATTCATTTACATAGACTCTCCTCTTTAG
[7] 30643 GTGAGTAAGCTGTCCGCGCAGAACCCGAAC...GTTCTAGTCACTTGATGTTTTTGTTTTAG
```

2.14 How to get the CDS and UTR sequences of genes associated with colorectal cancer

In this *HOWTO* we extract the CDS and UTR sequences of genes involved in colorectal cancer. The workflow extends the ideas presented in the previous *HOWTO* and suggests an approach for identifying disease-related genes.

2.14.1 Build a gene list

We start with a list of gene or transcript ids. If you do not have pre-defined list one can be created with the [KEGG.db](#) and [KEGGgraph](#) packages. Updates to the data in the [KEGG.db](#) package are no longer available, however, the resource is still useful for identifying pathway names and ids.

Create a table of KEGG pathways and ids and search on the term 'cancer'.

```
> library(KEGG.db)
> pathways <- toTable(KEGGPATHNAME2ID)
> pathways[grep("cancer", pathways$path_name, fixed=TRUE),]
```

Use the "05210" id to query the KEGG web resource (accesses the currently maintained data).

```
> library(KEGGgraph)
> dest <- tempfile()
> retrieveKGML("05210", "hsa", dest, "internal")
```

The suffix of the KEGG id is the Entrez gene id. The `translateKEGGID2GeneID` simply removes the prefix leaving just the Entrez gene ids.

```
> crids <- as.character(parseKGML2DataFrame(dest)[,1])
> crgenes <- unique(translateKEGGID2GeneID(crids))
> head(crgenes)
```

2.14.2 Identify genomic coordinates

The list of gene ids is used to extract genomic positions of the regions of interest. The Known Gene table from UCSC will be the annotation and is available as a *Bioconductor* package.

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
```

If an annotation is not available as a *Bioconductor* annotation package it may be available in [AnnotationHub](#). Additionally, there are functions in [GenomicFeatures](#) which can retrieve data from UCSC and Ensembl to create a TxDb. See `?makeTranscriptDbFromUCSC` for more information.

As in the previous *HOWTO* we need to identify the transcripts corresponding to each gene. The transcript id (or name) is used to isolate the UTR and coding regions of interest. This grouping of transcript by gene is also used to re-group the final sequence results.

The `transcriptsBy` function outputs both the gene and transcript identifiers which we use to create a map between the two. The map is a `CharacterList` with gene ids as names and transcript ids as the list elements.

```
> txbygene <- transcriptsBy(txdb, "gene")[crgenes] ## subset on colorectal genes
> map <- relist(unlist(txbygene, use.names=FALSE)$tx_id, txbygene)
> map
```

Extract the UTR and coding regions.

```
> cds <- cdsBy(txdb, "tx")
> threeUTR <- threeUTRsByTranscript(txdb)
> fiveUTR <- fiveUTRsByTranscript(txdb)
```

Coding and UTR regions may not be present for all transcripts specified in `map`. Consequently, the subset results will not be the same length. This length discrepancy must be taken into account when re-listing the final results by gene.

```
> txid <- unlist(map, use.names=FALSE)
> cds <- cds[names(cds) %in% txid]
> threeUTR <- threeUTR[names(threeUTR) %in% txid]
> fiveUTR <- fiveUTR[names(fiveUTR) %in% txid]
```

Note the different lengths of the subset regions.

```
> length(txid) ## all possible transcripts
> length(cds)
> length(threeUTR)
> length(fiveUTR)
```

These objects are `GRangesLists` with the transcript id as the outer list element.

```
> cds
```

2.14.3 Extract sequences from BSgenome

The `BSgenome` packages contain complete genome sequences for a given organism.

Load the `BSgenome` package for homo sapiens.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> genome <- BSgenome.Hsapiens.UCSC.hg19
```

Use `extractTranscriptSeqs` to extract the UTR and coding regions from the `BSgenome`. This function retrieves the sequences for an any `GRanges` or `GRangesList` (i.e., not just transcripts like the name implies).

```
> threeUTR_seqs <- extractTranscriptSeqs(genome, threeUTR)
> fiveUTR_seqs <- extractTranscriptSeqs(genome, fiveUTR)
> cds_seqs <- extractTranscriptSeqs(genome, cds)
```

The return values are `DNASTringSet` objects.

```
> cds_seqs
```

Our final step is to collect the coding and UTR regions (currently organized by transcript) into groups by gene id. The `relist` function groups the sequences of a `DNASTringSet` object into a `DNASTringSetList` object, based on the specified `skeleton` argument. The `skeleton` must be a list-like object and only its shape (i.e. its element lengths) matters (its exact content is ignored). A simple form of `skeleton` is to use a partitioning object that we make by specifying the size of each partition. The partitioning objects are different for each type of region because not all transcripts had a coding or 3' or 5' UTR region defined.

```
> lst3 <- relist(threeUTR_seqs, PartitioningByWidth(sum(map %in% names(threeUTR))))
> lst5 <- relist(fiveUTR_seqs, PartitioningByWidth(sum(map %in% names(fiveUTR))))
> lstc <- relist(cds_seqs, PartitioningByWidth(sum(map %in% names(cds))))
```

There are 239 genes in `map` each of which have 1 or more transcripts. The table of element lengths shows how many genes have each number of transcripts. For example, 47 genes have 1 transcript, 48 genes have 2 etc.

```
> length(map)
> table(elementLengths(map))
```

The lists of DNA sequences all have the same length as `map` but one or more of the element lengths may be zero. This would indicate that data were not available for that gene. The tables below show that there was at least 1 coding region available for all genes (i.e., none of the element lengths are 0). However, both the 3' and 5' UTR results have element lengths of 0 which indicates no UTR data were available for that gene.

```
> table(elementLengths(lstc))
> table(elementLengths(lst3))
> names(lst3)[elementLengths(lst3) == 0L] ## genes with no 3' UTR data
> table(elementLengths(lst5))
> names(lst5)[elementLengths(lst5) == 0L] ## genes with no 5' UTR data
```

2.15 How to create DNA consensus sequences for read group 'families'

The motivation for this *HOWTO* comes from a study which explored the dynamics of point mutations. The mutations of interest exist with a range of frequencies in the control group (e.g., 0.1% - 50%). PCR and sequencing error rates make it difficult to identify low frequency events (e.g., < 20%).

When a library is prepared with Nextera, random fragments are generated followed by a few rounds of PCR. When the genome is large enough, reads aligning to the same start position are likely descendant from the same template fragment and should have identical sequences.

The goal is to eliminate noise by grouping the reads by common start position and discarding those that do not exceed a certain threshold within each family. A new consensus sequence will be created for each read group family.

2.15.1 Sort reads into groups by start position

Load the BAM file into a `GAlignments` object.

```
> library(Rsamtools)
> bamfile <- system.file("extdata", "ex1.bam", package="Rsamtools")
> param <- ScanBamParam(what=c("seq", "qual"))
> library(GenomicAlignments)
> gal <- readGAlignments(bamfile, use.names=TRUE, param=param)
```

Use the `sequenceLayer` function to *lay* the query sequences and quality strings on the reference.

```
> qseq <- setNames(mcols(gal)$seq, names(gal))
> qual <- setNames(mcols(gal)$qual, names(gal))
> qseq_on_ref <- sequenceLayer(qseq, cigar(gal),
+                             from="query", to="reference")
> qual_on_ref <- sequenceLayer(qual, cigar(gal),
+                             from="query", to="reference")
```

Split by chromosome.

```
> qseq_on_ref_by_chrom <- splitAsList(qseq_on_ref, seqnames(gal))
> qual_on_ref_by_chrom <- splitAsList(qual_on_ref, seqnames(gal))
> pos_by_chrom <- splitAsList(start(gal), seqnames(gal))
```


Quick look at 'qseq_on_ref_id': It's an IntegerList object with the same length and "shape" as 'qseq_on_ref'.

```
> qseq_on_ref_id
IntegerList of length 1934
[[1]] 1
[[2]] 2
[[3]] 3
[[4]] 4
[[5]] 5
[[6]] 6 7
[[7]] 8
[[8]] 9
[[9]] 10 11
[[10]] 12
...
<1924 more elements>
```

Remove the under represented ids from each list element of 'qseq_on_ref_id':

```
> qseq_on_ref_id2 <- endoapply(qseq_on_ref_id,
+   function(ids) ids[countMatches(ids, ids) >= 0.2 * length(ids)])
```

Remove corresponding sequences from 'qseq_on_ref':

```
> tmp <- unlist(qseq_on_ref_id2, use.names=FALSE)
> qseq_on_ref2 <- relist(unlist(qseq_on_ref, use.names=FALSE)[tmp],
+   qseq_on_ref_id2)
```

2.15.3 Create a consensus sequence for each read group family

Compute 1 consensus matrix per chromosome:

```
> split_factor <- rep.int(seqnames(gr), elementLengths(qseq_on_ref2))
> qseq_on_ref2 <- unlist(qseq_on_ref2, use.names=FALSE)
> qseq_on_ref2_by_chrom <- splitAsList(qseq_on_ref2, split_factor)
> qseq_pos_by_chrom <- splitAsList(start(gr), split_factor)
> cm_by_chrom <- lapply(names(qseq_pos_by_chrom),
+   function(seqname)
+     consensusMatrix(qseq_on_ref2_by_chrom[[seqname]],
+       as.prob=TRUE,
+       shift=qseq_pos_by_chrom[[seqname]]-1,
+       width=seqlengths(gr)[[seqname]]))
> names(cm_by_chrom) <- names(qseq_pos_by_chrom)
```

'cm_by_chrom' is a list of consensus matrices. Each matrix has 17 rows (1 per letter in the DNA alphabet) and 1 column per chromosome position.

```
> lapply(cm_by_chrom, dim)
```

```
$seq1
[1] 18 1575
```

```
$seq2
[1] 18 1584
```

Compute the consensus string from each consensus matrix. We'll put "+" in the strings wherever there is no coverage for that position, and "N" where there is coverage but no consensus.

```
> cs_by_chrom <- lapply(cm_by_chrom,
+   function(cm) {
+     ## need to "fix" 'cm' because consensusString()
+     ## doesn't like consensus matrices with columns
+     ## that contain only zeroes (e.g., chromosome
+     ## positions with no coverage)
+     idx <- colSums(cm) == 0L
+     cm["+", idx] <- 1
+     DNASTring(consensusString(cm, ambiguityMap="N"))
+   })
```

The new consensus strings.

```
> cs_by_chrom

$seq1
1575-letter "DNASTring" instance
seq: NANTAGNNNCTCANTTTAAANNTTNTTTTTN...AATNATANNTTNTTNTTTNTCTGNAC+++++

$seq2
1584-letter "DNASTring" instance
seq: ++++++...NNNANANANANCTNNA+++++
```

2.16 How to compute binned averages along a genome

In some applications, there is the need to compute the average of a variable along a genome for a set of predefined fixed-width regions (sometimes called "bins"). One such example is coverage. Coverage is an `RleList` with one list element per chromosome. Here we simulate a coverage list.

```
> library(BSgenome.Scerevisiae.UCSC.sacCer2)
> set.seed(22)
> cov <- RleList(
+   lapply(seqlengths(Scerevisiae),
+     function(len) Rle(sample(-10:10, len, replace=TRUE))),
+   compress=FALSE)
> head(cov, 3)

RleList of length 3
$chrI
integer-Rle of length 230208 with 219146 runs
Lengths:  1  1  1  1  1  1  1 ...  1  1  1  1  1  1  1
Values : -4 -1 10  0  7  5  2 ...  4 -2 -8  1 -10 -8 -10

$chrII
integer-Rle of length 813178 with 774522 runs
Lengths:  1  1  1  1  1  1  1 ...  1  1  1  2  2  1  1
Values : -3 -6 -7 -3  9 -4 -10 ... -3 -4 -5  2 -2 -8  0

$chrIII
integer-Rle of length 316617 with 301744 runs
Lengths:  1  1  1  1  1  1  1 ...  1  1  1  1  1  1  1
Values :  2 -3 -6  5  9  5  3 ...  4 -7 -10 -5 -10 -1 -3
```

Use the `tileGenome` function to create a set of bins along the genome.

```
> bins1 <- tileGenome(seqinfo(Scerevisiae), tilewidth=100,
+   cut.last.tile.in.chrom=TRUE)
```

We define the following function to compute the binned average of a numerical variable defined along a genome.

Arguments:

```
'bins': a GRanges object representing the genomic bins.
        Typically obtained by calling tileGenome() with
        'cut.last.tile.in.chrom=TRUE'.
'numvar': a named RleList object representing a numerical
          variable defined along the genome covered by 'bins', which
          is the genome described by 'seqinfo(bins)'.
'mcolname': the name to give to the metadata column that will
           contain the binned average in the returned object.
```

The function returns 'bins' with an additional metadata column named 'mcolname' containing the binned average.

```
> binnedAverage <- function(bins, numvar, mcolname)
+ {
+   stopifnot(is(bins, "GRanges"))
+   stopifnot(is(numvar, "RleList"))
+   stopifnot(identical(seqlevels(bins), names(numvar)))
+   bins_per_chrom <- split(ranges(bins), seqnames(bins))
+   means_list <- lapply(names(numvar),
+     function(seqname) {
+       views <- Views(numvar[[seqname]],
+         bins_per_chrom[[seqname]])
+       viewMeans(views)
+     })
+   new_mcol <- unsplit(means_list, as.factor(seqnames(bins)))
+   mcols(bins)[[mcolname]] <- new_mcol
+   bins
+ }
```

Compute the binned average for 'cov':

```
> bins1 <- binnedAverage(bins1, cov, "binned_cov")
> bins1
```

GRanges with 121639 ranges and 1 metadata column:

	seqnames	ranges	strand		binned_cov
	<Rle>	<IRanges>	<Rle>		<numeric>
[1]	chrI	[1, 100]	*		-0.66
[2]	chrI	[101, 200]	*		-0.05
[3]	chrI	[201, 300]	*		-1.56
...
[121637]	2micron	[6101, 6200]	*		-0.25
[121638]	2micron	[6201, 6300]	*		-0.54
[121639]	2micron	[6301, 6318]	*		-0.4444444444444444

seqlengths:

chrI	chrII	chrIII	chrIV	...	chrXV	chrXVI	chrM	2micron
230208	813178	316617	1531919	...	1091289	948062	85779	6318

The bin size can be modified with the `tilewidth` argument to `tileGenome`. For additional examples see `?tileGenome`.

3 Session Information

R version 3.1.1 (2014-07-10)

Platform: x86_64-unknown-linux-gnu (64-bit)

locale:

```
[1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C
[3] LC_TIME=en_US.UTF-8      LC_COLLATE=en_US.UTF-8
[5] LC_MONETARY=en_US.UTF-8  LC_MESSAGES=en_US.UTF-8
[7] LC_PAPER=en_US.UTF-8     LC_NAME=C
[9] LC_ADDRESS=C             LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
```

attached base packages:

```
[1] parallel stats graphics grDevices utils datasets
[7] methods base
```

other attached packages:

```
[1] BSgenome.Scerevisiae.UCSC.sacCer2_1.4.0
[2] BSgenome.Hsapiens.UCSC.hg19_1.4.0
[3] BSgenome_1.33.8
[4] rtracklayer_1.25.13
[5] edgeR_3.7.12
[6] limma_3.21.10
[7] DESeq_1.17.0
[8] lattice_0.20-29
[9] locfit_1.5-9.1
[10] TxDb.Dmelanogaster.UCSC.dm3.ensGene_2.14.0
[11] TxDb.Athaliana.BioMart.plantmart21_2.14.0
[12] TxDb.Hsapiens.UCSC.hg19.knownGene_2.14.0
[13] GenomicFeatures_1.17.13
[14] AnnotationDbi_1.27.9
[15] Biobase_2.25.0
[16] GenomicAlignments_1.1.22
[17] Rsamtools_1.17.32
[18] Biostrings_2.33.13
[19] XVector_0.5.7
[20] GenomicRanges_1.17.29
[21] GenomeInfoDb_1.1.18
[22] IRanges_1.99.23
[23] S4Vectors_0.1.2
[24] BiocGenerics_0.11.3
[25] pasillaBamSubset_0.3.0
```

loaded via a namespace (and not attached):

```
[1] annotate_1.43.5 BatchJobs_1.3 BBmisc_1.7
[4] BiocParallel_0.7.9 BiocStyle_1.3.4 biomaRt_2.21.1
[7] bitops_1.0-6 brew_1.0-6 checkmate_1.2
[10] codetools_0.2-8 DBI_0.2-7 digest_0.6.4
[13] fail_1.2 foreach_1.4.2 genefilter_1.47.6
[16] geneplotter_1.43.0 grid_3.1.1 iterators_1.0.7
[19] RColorBrewer_1.0-5 RCurl_1.95-4.3 RSQLite_0.11.4
[22] sendmailR_1.1-2 splines_3.1.1 stats4_3.1.1
```

```
[25] stringr_0.6.2      survival_2.37-7    tools_3.1.1
[28] XML_3.98-1.1       xtable_1.7-3      zlibbioc_1.11.1
```