

Package ‘GenomicAlignments’

April 10, 2015

Title Representation and manipulation of short genomic alignments

Description Provides efficient containers for storing and manipulating short genomic alignments (typically obtained by aligning short reads to a reference genome). This includes read counting, computing the coverage, junction detection, and working with the nucleotide content of the alignments.

Version 1.2.2

Author Hervé Pagès, Valerie Obenchain, Martin Morgan

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

biocViews Genetics, Infrastructure, DataImport, Sequencing, RNASeq, SNP, Coverage, Alignment

Depends R (>= 2.10), methods, BiocGenerics (>= 0.11.3), S4Vectors (>= 0.2.2), IRanges (>= 1.99.27), GenomeInfoDb (>= 1.1.20), GenomicRanges (>= 1.17.42), Biostrings (>= 2.33.14), Rsamtools (>= 1.18.3)

Imports methods, stats, BiocGenerics, S4Vectors, IRanges, GenomicRanges, Biostrings, Rsamtools, BiocParallel

LinkingTo S4Vectors, IRanges

Suggests rtracklayer, BSgenome, GenomicFeatures, RNAseqData.HNRNPC.bam.chr14, pasillaBamSubset, TxDb.Hsapiens.UCSC.hg19.knownGene, TxDb.Dmelanogaster.UCSC.dm3.ensGene, BSgenome.Dmelanogaster.UCSC.dm3, BSgenome.Hsapiens.UCSC.hg19, DESeq, edgeR, RUnit, BiocStyle

License Artistic-2.0

Collate utils.R cigar-utils.R GAlignments-class.R
GAlignmentPairs-class.R GAlignmentsList-class.R
GappedReads-class.R OverlapEncodings-class.R
findMateAlignment.R readGAlignments.R junctions-methods.R
sequenceLayer.R pileLettersAt.R stackStringsFromBam.R
intra-range-methods.R coverage-methods.R setops-methods.R
findOverlaps-methods.R mapCoords-methods.R
encodeOverlaps-methods.R findCompatibleOverlaps-methods.R
summarizeOverlaps-methods.R findSpliceOverlaps-methods.R zzz.R

Video <https://www.youtube.com/watch?v=2KqBSbkfhRo> ,
https://www.youtube.com/watch?v=3PK_jx44QTs

R topics documented:

cigar-utils	2
coverage-methods	9
encodeOverlaps-methods	11
findCompatibleOverlaps-methods	14
findMateAlignment	16
findOverlaps-methods	20
findSpliceOverlaps-methods	23
GAlignmentPairs-class	25
GAlignments-class	30
GAlignmentsList-class	35
GappedReads-class	40
intra-range-methods	41
junctions-methods	43
map-methods	50
mapCoords-methods	51
OverlapEncodings-class	52
pileLettersAt	55
readGAlignments	57
sequenceLayer	63
setops-methods	69
stackStringsFromBam	70
summarizeOverlaps-methods	74
Index	84

cigar-utils	<i>CIGAR utility functions</i>
-------------	--------------------------------

Description

Utility functions for low-level CIGAR manipulation.

Usage

```
## ---- Supported CIGAR operations ----
CIGAR_OPS
```

```
## ---- Transform CIGARs into other useful representations ----
explodeCigarOps(cigar, ops=CIGAR_OPS)
explodeCigarOpLengths(cigar, ops=CIGAR_OPS)
cigarToRleList(cigar)
```

```

## ---= Summarize CIGARs ---=
cigarOpTable(cigar)

## ---= From CIGARs to ranges ---=
cigarRangesAlongReferenceSpace(cigar, flag=NULL,
                                N.regions.removed=FALSE, pos=1L, f=NULL,
                                ops=CIGAR_OPS, drop.empty.ranges=FALSE, reduce.ranges=FALSE,
                                with.ops=FALSE)

cigarRangesAlongQuerySpace(cigar, flag=NULL,
                             before.hard.clipping=FALSE, after.soft.clipping=FALSE,
                             ops=CIGAR_OPS, drop.empty.ranges=FALSE, reduce.ranges=FALSE,
                             with.ops=FALSE)

cigarRangesAlongPairwiseSpace(cigar, flag=NULL,
                               N.regions.removed=FALSE, dense=FALSE,
                               ops=CIGAR_OPS, drop.empty.ranges=FALSE, reduce.ranges=FALSE,
                               with.ops=FALSE)

extractAlignmentRangesOnReference(cigar, pos=1L,
                                   drop.D.ranges=FALSE, f=NULL)

## ---= From CIGARs to sequence lengths ---=
cigarWidthAlongReferenceSpace(cigar, flag=NULL,
                              N.regions.removed=FALSE)

cigarWidthAlongQuerySpace(cigar, flag=NULL,
                           before.hard.clipping=FALSE, after.soft.clipping=FALSE)

cigarWidthAlongPairwiseSpace(cigar, flag=NULL,
                              N.regions.removed=FALSE, dense=FALSE)

## ---= Narrow CIGARs ---=
cigarNarrow(cigar, start=NA, end=NA, width=NA)
cigarQNarrow(cigar, start=NA, end=NA, width=NA)

## ---= Translate coordinates between query and reference spaces ---=
queryLoc2refLoc(qloc, cigar, pos=1L)
queryLocs2refLocs(qlocs, cigar, pos=1L, flag=NULL)

```

Arguments

cigar	A character vector or factor containing the extended CIGAR strings. It can be of arbitrary length except for queryLoc2refLoc which only accepts a single CIGAR (as a character vector or factor of length 1).
ops	Character vector containing the extended CIGAR operations to actually consider. Zero-length operations or operations not listed ops are ignored.
flag	NULL or an integer vector containing the SAM flag for each read.

According to the SAM Spec v1.4, flag bit 0x4 is the only reliable place to tell whether a segment (or read) is mapped (bit is 0) or not (bit is 1). If `flag` is supplied, then `cigarRangesAlongReferenceSpace`, `cigarRangesAlongQuerySpace`, `cigarRangesAlongPairwiseSpace`, and `extractAlignmentRangesOnReference` don't produce any range for unmapped reads i.e. they treat them as if their CIGAR was empty (independently of what their CIGAR is). If `flag` is supplied, then `cigarWidthAlongReferenceSpace`, `cigarWidthAlongQuerySpace`, and `cigarWidthAlongPairwiseSpace` return NAs for unmapped reads.

<code>N.regions.removed</code>	TRUE or FALSE. If TRUE, then <code>cigarRangesAlongReferenceSpace</code> and <code>cigarWidthAlongReferenceSpace</code> report ranges/widths with respect to the "reference" space from which the N regions have been removed, and <code>cigarRangesAlongPairwiseSpace</code> and <code>cigarWidthAlongPairwiseSpace</code> report them with respect to the "pairwise" space from which the N regions have been removed.
<code>pos</code>	An integer vector containing the 1-based leftmost position/coordinate for each (eventually clipped) read sequence. Must have length 1 (in which case it's recycled to the length of <code>cigar</code>), or the same length as <code>cigar</code> .
<code>f</code>	NULL or a factor of length <code>cigar</code> . If NULL, then the ranges are grouped by alignment i.e. the returned <code>IRangesList</code> object has 1 list element per element in <code>cigar</code> . Otherwise they are grouped by factor level i.e. the returned <code>IRangesList</code> object has 1 list element per level in <code>f</code> and is named with those levels. For example, if <code>f</code> is a factor containing the chromosome for each read, then the returned <code>IRangesList</code> object will have 1 list element per chromosome and each list element will contain all the ranges on that chromosome.
<code>drop.empty.ranges</code>	Should empty ranges be dropped?
<code>reduce.ranges</code>	Should adjacent ranges coming from the same cigar be merged or not? Using TRUE can significantly reduce the size of the returned object.
<code>with.ops</code>	TRUE or FALSE indicating whether the returned ranges should be named with their corresponding CIGAR operation.
<code>before.hard.clipping</code>	TRUE or FALSE. If TRUE, then <code>cigarRangesAlongQuerySpace</code> and <code>cigarWidthAlongQuerySpace</code> report ranges/widths with respect to the "query" space to which the H regions have been added. <code>before.hard.clipping</code> and <code>after.soft.clipping</code> cannot both be TRUE.
<code>after.soft.clipping</code>	TRUE or FALSE. If TRUE, then <code>cigarRangesAlongQuerySpace</code> and <code>cigarWidthAlongQuerySpace</code> report ranges/widths with respect to the "query" space from which the S regions have been removed. <code>before.hard.clipping</code> and <code>after.soft.clipping</code> cannot both be TRUE.
<code>dense</code>	TRUE or FALSE. If TRUE, then <code>cigarRangesAlongPairwiseSpace</code> and <code>cigarWidthAlongPairwiseSpace</code> report ranges/widths with respect to the "pairwise" space from which the I, D, and N regions have been removed. <code>N.regions.removed</code> and <code>dense</code> cannot both be TRUE.
<code>drop.D.ranges</code>	Should the ranges corresponding to a deletion from the reference (encoded with a D in the CIGAR) be dropped? By default we keep them to be consistent with

	the pileup tool from SAMtools. Note that, when <code>drop.D.ranges</code> is <code>TRUE</code> , then <code>Ds</code> and <code>Ns</code> in the CIGAR are equivalent.
<code>start,end,width</code>	Vectors of integers. NAs and negative values are accepted and "solved" according to the rules of the SEW (Start/End/Width) interface (see <code>?solveUserSEW</code> for the details).
<code>qloc</code>	An integer vector containing "query-based locations" i.e. 1-based locations relative to the query sequence stored in the SAM/BAM file.
<code>qlocs</code>	A list of the same length as <code>cigar</code> where each element is an integer vector containing "query-based locations" i.e. 1-based locations relative to the corresponding query sequence stored in the SAM/BAM file.

Value

`CIGAR_OPS` is a predefined character vector containing the supported extended CIGAR operations: M, I, D, N, S, H, P, =, X. See p. 4 of the SAM Spec v1.4 at <http://samtools.sourceforge.net/> for the list of extended CIGAR operations and their meanings.

For `explodeCigarOps` and `explodeCigarOpLengths`: Both functions return a list of the same length as `cigar` where each list element is a character vector (for `explodeCigarOps`) or an integer vector (for `explodeCigarOpLengths`). The 2 lists have the same shape, that is, same `length()` and same `elementLengths()`. The *i*-th character vector in the list returned by `explodeCigarOps` contains one single-letter string per CIGAR operation in `cigar[i]`. The *i*-th integer vector in the list returned by `explodeCigarOpLengths` contains the corresponding CIGAR operation lengths. Zero-length operations or operations not listed in `ops` are ignored.

For `cigarToRleList`: A [CompressedRleList](#) object.

For `cigarOpTable`: An integer matrix with number of rows equal to the length of `cigar` and nine columns, one for each extended CIGAR operation.

For `cigarRangesAlongReferenceSpace`, `cigarRangesAlongQuerySpace`, `cigarRangesAlongPairwiseSpace`, and `extractAlignmentRangesOnReference`: An [IRangesList](#) object (more precisely a [CompressedIRangesList](#) object) with 1 list element per element in `cigar`. However, if `f` is a factor, then the returned [IRangesList](#) object can be a [SimpleIRangesList](#) object (instead of [CompressedIRangesList](#)), and in that case, has 1 list element per level in `f` and is named with those levels.

For `cigarWidthAlongReferenceSpace` and `cigarWidthAlongPairwiseSpace`: An integer vector of the same length as `cigar` where each element is the width of the alignment with respect to the "reference" and "pairwise" space, respectively. More precisely, for `cigarWidthAlongReferenceSpace`, the returned widths are the lengths of the alignments on the reference, `N` gaps included (except if `N.regions.removed` is `TRUE`). NAs or "*" in `cigar` will produce NAs in the returned vector.

For `cigarWidthAlongQuerySpace`: An integer vector of the same length as `cigar` where each element is the length of the corresponding query sequence as inferred from the CIGAR string. Note that, by default (i.e. if `before.hard.clipping` and `after.soft.clipping` are `FALSE`), this is the length of the query sequence stored in the SAM/BAM file. If `before.hard.clipping` or `after.soft.clipping` is `TRUE`, the returned widths are the lengths of the query sequences before hard clipping or after soft clipping. NAs or "*" in `cigar` will produce NAs in the returned vector.

For `cigarNarrow` and `cigarQNarrow`: A character vector of the same length as `cigar` containing the narrowed cigars. In addition the vector has an "rshift" attribute which is an integer vector of

the same length as `cigar`. It contains the values that would need to be added to the POS field of a SAM/BAM file as a consequence of this cigar narrowing.

For `queryLoc2refLoc`: An integer vector of the same length as `qloc` containing the "reference-based locations" (i.e. the 1-based locations relative to the reference sequence) corresponding to the "query-based locations" passed in `qloc`.

For `queryLocs2refLocs`: A list of the same length as `qlocs` where each element is an integer vector containing the "reference-based locations" corresponding to the "query-based locations" passed in the corresponding element in `qlocs`.

Author(s)

H. Pages and P. Aboyoun

References

<http://samtools.sourceforge.net/>

See Also

- The [sequenceLayer](#) function in the **GenomicAlignments** package for laying the query sequences alongside the "reference" or "pairwise" spaces.
- The [GAlignments](#) container for storing a set of genomic alignments.
- The [IRanges](#), [IRangesList](#), and [RleList](#) classes in the **IRanges** package.
- The [coverage](#) generic and methods for computing the coverage across a set of ranges or genomic ranges.

Examples

```
## -----
## A. CIGAR_OPS, explodeCigarOps(), explodeCigarOpLengths(),
##   cigarToRleList(), and cigarOpTable()
## -----

## Supported CIGAR operations:
CIGAR_OPS

## Transform CIGARs into other useful representations:
cigar1 <- "3H15M55N4M2I6M2D5M6S"
cigar2 <- c("40M2I9M", cigar1, "2S10M2000N15M", "3H33M5H")

explodeCigarOps(cigar2)
explodeCigarOpLengths(cigar2)
explodeCigarOpLengths(cigar2, ops=c("I", "S"))
cigarToRleList(cigar2)

## Summarize CIGARs:
cigarOpTable(cigar2)

## -----
## B. From CIGARs to ranges and to sequence lengths
```

```

## -----

## CIGAR ranges along the "reference" space:
cigarRangesAlongReferenceSpace(cigar1, with.ops=TRUE)[[1]]

cigarRangesAlongReferenceSpace(cigar1,
                                reduce.ranges=TRUE, with.ops=TRUE)[[1]]

ops <- setdiff(CIGAR_OPS, "N")

cigarRangesAlongReferenceSpace(cigar1, ops=ops, with.ops=TRUE)[[1]]

cigarRangesAlongReferenceSpace(cigar1, ops=ops,
                                reduce.ranges=TRUE, with.ops=TRUE)[[1]]

ops <- setdiff(CIGAR_OPS, c("D", "N"))

cigarRangesAlongReferenceSpace(cigar1, ops=ops, with.ops=TRUE)[[1]]

cigarWidthAlongReferenceSpace(cigar1)

pos2 <- c(1, 1001, 1, 351)

cigarRangesAlongReferenceSpace(cigar2, pos=pos2, with.ops=TRUE)

res1a <- extractAlignmentRangesOnReference(cigar2, pos=pos2)
res1b <- cigarRangesAlongReferenceSpace(cigar2,
                                        pos=pos2,
                                        ops=setdiff(CIGAR_OPS, "N"),
                                        reduce.ranges=TRUE)
stopifnot(identical(res1a, res1b))

res2a <- extractAlignmentRangesOnReference(cigar2, pos=pos2,
                                        drop.D.ranges=TRUE)
res2b <- cigarRangesAlongReferenceSpace(cigar2,
                                        pos=pos2,
                                        ops=setdiff(CIGAR_OPS, c("D", "N")),
                                        reduce.ranges=TRUE)
stopifnot(identical(res2a, res2b))

seqnames <- factor(c("chr6", "chr6", "chr2", "chr6"),
                  levels=c("chr2", "chr6"))
extractAlignmentRangesOnReference(cigar2, pos=pos2, f=seqnames)

## CIGAR ranges along the "query" space:
cigarRangesAlongQuerySpace(cigar2, with.ops=TRUE)
cigarWidthAlongQuerySpace(cigar1)
cigarWidthAlongQuerySpace(cigar1, before.hard.clipping=TRUE)

## CIGAR ranges along the "pairwise" space:
cigarRangesAlongPairwiseSpace(cigar2, with.ops=TRUE)
cigarRangesAlongPairwiseSpace(cigar2, dense=TRUE, with.ops=TRUE)

```

```

## -----
## C. PERFORMANCE
## -----

if (interactive()) {
  ## We simulate 20 millions aligned reads, all 40-mers. 95% of them
  ## align with no indels. 5% align with a big deletion in the
  ## reference. In the context of an RNAseq experiment, those 5% would
  ## be suspected to be "junction reads".
  set.seed(123)
  nreads <- 20000000L
  njunctionreads <- nreads * 5L / 100L
  cigar3 <- character(nreads)
  cigar3[] <- "40M"
  junctioncigars <- paste(
    paste(10:30, "M", sep=""),
    paste(sample(80:8000, njunctionreads, replace=TRUE), "N", sep=""),
    paste(30:10, "M", sep=""), sep="")
  cigar3[sample(nreads, njunctionreads)] <- junctioncigars
  some_fake_rnames <- paste("chr", c(1:6, "X"), sep="")
  rname <- factor(sample(some_fake_rnames, nreads, replace=TRUE),
    levels=some_fake_rnames)
  pos <- sample(80000000L, nreads, replace=TRUE)

  ## The following takes < 3 sec. to complete:
  system.time(irl1 <- extractAlignmentRangesOnReference(cigar3, pos=pos))

  ## The following takes < 4 sec. to complete:
  system.time(irl2 <- extractAlignmentRangesOnReference(cigar3, pos=pos,
    f=rname))

  ## The sizes of the resulting objects are about 240M and 160M,
  ## respectively:
  object.size(irl1)
  object.size(irl2)
}

## -----
## D. COMPUTE THE COVERAGE OF THE READS STORED IN A BAM FILE
## -----

## The information stored in a BAM file can be used to compute the
## "coverage" of the mapped reads i.e. the number of reads that hit any
## given position in the reference genome.
## The following function takes the path to a BAM file and returns an
## object representing the coverage of the mapped reads that are stored
## in the file. The returned object is an RleList object named with the
## names of the reference sequences that actually receive some coverage.

extractCoverageFromBAM <- function(file)
{
  ## This ScanBamParam object allows us to load only the necessary
  ## information from the file.
  param <- ScanBamParam(flag=scanBamFlag(isUnmappedQuery=FALSE,

```



```

                                isDuplicate=FALSE),
                                what=c("rname", "pos", "cigar"))
bam <- scanBam(file, param=param)[[1]]
## Note that unmapped reads and reads that are PCR/optical duplicates
## have already been filtered out by using the ScanBamParam object above.
irl <- extractAlignmentRangesOnReference(bam$cigar, pos=bam$pos,
                                         f=bam$rname)
irl <- irl[elementLengths(irl) != 0] # drop empty elements
coverage(irl)
}

library(Rsamtools)
f1 <- system.file("extdata", "ex1.bam", package="Rsamtools")
extractCoverageFromBAM(f1)

## -----
## E. cigarNarrow() and cigarQNarrow()
## -----

## cigarNarrow():
cigarNarrow(cigar1) # only drops the soft/hard clipping
cigarNarrow(cigar1, start=10)
cigarNarrow(cigar1, start=15)
cigarNarrow(cigar1, start=15, width=57)
cigarNarrow(cigar1, start=16)
#cigarNarrow(cigar1, start=16, width=55) # ERROR! (empty cigar)
cigarNarrow(cigar1, start=71)
cigarNarrow(cigar1, start=72)
cigarNarrow(cigar1, start=75)

## cigarQNarrow():
cigarQNarrow(cigar1, start=4, end=-3)
cigarQNarrow(cigar1, start=10)
cigarQNarrow(cigar1, start=19)
cigarQNarrow(cigar1, start=24)

```

coverage-methods

Coverage of a GAlignments or GAlignmentPairs object

Description

`coverage` methods for [GAlignments](#), [GAlignmentPairs](#), and [BamFile](#) objects.

NOTE: The `coverage` generic function and methods for [Ranges](#) and [RangesList](#) objects are defined and documented in the [IRanges](#) package. Methods for [GRanges](#) and [GRangesList](#) objects are defined and documented in the [GenomicRanges](#) package.

Usage

```
## S4 method for signature GAlignments
```

```

coverage(x, shift=0L, width=NULL, weight=1L,
         method=c("auto", "sort", "hash"), drop.D.ranges=FALSE)

## S4 method for signature GAlignmentPairs
coverage(x, shift=0L, width=NULL, weight=1L,
         method=c("auto", "sort", "hash"), drop.D.ranges=FALSE)

## S4 method for signature BamFile
coverage(x, shift=0L, width=NULL, weight=1L, ...,
         param=ScanBamParam())

## S4 method for signature character
coverage(x, shift=0L, width=NULL, weight=1L, ...,
         yieldSize=2500000L)

```

Arguments

x	A GAlignments , GAlignmentPairs , BamFile or character(1) object.
shift, width, weight	See coverage method for GRanges objects in the GenomicRanges package.
method	See ?coverage in the IRanges package for a description of this argument.
drop.D.ranges	Whether the coverage calculation should ignore ranges corresponding to D (deletion) in the CIGAR string.
...	Additional arguments passed to the coverage method for GAlignments objects.
param	An optional ScanBamParam object passed to readGAlignmentsFromBam .
yieldSize	An optional argument controlling how many records are input when iterating through a BamFile .

Details

The methods for [GAlignments](#) and [GAlignmentPairs](#) objects do:

```
coverage(grglist(x), ...)
```

The method for [BamFile](#) objects iterates through a BAM file, reading `yieldSize(x)` records (or all records, if `is.na(yieldSize(x))`) and calculating:

```
aln <- readGAlignmentsFromBam(x, param=param)
coverage(aln, shift=shift, width=width, weight=weight, ...)
```

The method for character vectors of length 1 creates a [BamFile](#) object from x and performs the calculation for `coverage,BamFile-method`.

Value

A named [RleList](#) object with one coverage vector per seqlevel in x.

See Also

- [coverage](#) in the **IRanges** package.
- [coverage-methods](#) in the **GenomicRanges** package.
- [RleList](#) objects in the **IRanges** package.
- [GAlignments](#) and [GAlignmentPairs](#) objects.
- [readGAlignmentsFromBam](#).

Examples

```
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")

gal <- readGAlignments(ex1_file)
stopifnot(identical(coverage(gal), coverage(as(gal, "GRangesList"))))

galp <- readGAlignmentPairs(ex1_file)
stopifnot(identical(coverage(galp), coverage(as(galp, "GRangesList"))))
```

encodeOverlaps-methods

Encode the overlaps between RNA-seq reads and the transcripts of a gene model

Description

In the context of an RNA-seq experiment, encoding the overlaps between the aligned reads and the transcripts of a given gene model can be used for detecting those overlaps that are *compatible* with the splicing of the transcript.

The central tool for this is the `encodeOverlaps` method for [GRangesList](#) objects, which computes the "overlap encodings" between a query and a subject, both list-like objects with list elements containing multiple ranges.

Other related utilities are also documented in this man page.

Usage

```
encodeOverlaps(query, subject, hits=NULL, ...)

## S4 method for signature GRangesList,GRangesList
encodeOverlaps(query, subject, hits=NULL,
               flip.query.if.wrong.strand=FALSE)

## Related utilities:

flipQuery(x, i)

selectEncodingWithCompatibleStrand(ovencA, ovencB,
```

```
query.strand, subject.strand, hits=NULL)
```

```
isCompatibleWithSplicing(x)
```

```
isCompatibleWithSkippedExons(x, max.skipped.exons=NA)
```

```
extractSteppedExonRanks(x, for.query.right.end=FALSE)
```

```
extractSpannedExonRanks(x, for.query.right.end=FALSE)
```

```
extractSkippedExonRanks(x, for.query.right.end=FALSE)
```

```
extractQueryStartInTranscript(query, subject, hits=NULL, ovenc=NULL,
                               flip.query.if.wrong.strand=FALSE,
                               for.query.right.end=FALSE)
```

Arguments

`query`, `subject` Typically [GRangesList](#) objects representing the the aligned reads and the transcripts of a given gene model, respectively. If the 2 objects don't have the same length, and if the `hits` argument is not supplied, then the shortest is recycled to the length of the longest (the standard recycling rules apply).

More generally speaking, `query` and `subject` must be list-like objects with list elements containing multiple ranges e.g. [RangesList](#) or [GRangesList](#) objects.

`hits` An optional [Hits](#) object typically obtained from a previous call to [findOverlaps\(query, subject\)](#).

Strictly speaking, `hits` only needs to be compatible with `query` and `subject`, that is, [queryLength\(hits\)](#) and [subjectLength\(hits\)](#) must be equal to `length(query)` and `length(subject)`, respectively.

Supplying `hits` is a convenient way to do `encodeOverlaps(query[queryHits(hits)], subject[subjectHits(hits)])`, that is, calling `encodeOverlaps(query, subject, hits)` is equivalent to the above, but is much more efficient, especially when `query` and/or `subject` are big. Of course, when `hits` is supplied, `query` and `subject` are not expected to have the same length anymore.

... Additional arguments for methods.

`flip.query.if.wrong.strand`

See the "OverlapEncodings" vignette located in this package (**GenomicAlignments**).

`x` For `flipQuery`: a [GRangesList](#) object.

For `isCompatibleWithSplicing`, `isCompatibleWithSkippedExons`, `extractSteppedExonRanks`, `extractSpannedExonRanks`, and `extractSkippedExonRanks`: an [OverlapEncodings](#) object, a factor, or a character vector.

`i` Subscript specifying the elements in `x` to flip. If missing, all the elements are flipped.

`ovencA`, `ovencB`, `ovenc`

[OverlapEncodings](#) objects.

`query.strand`, `subject.strand`

Vector-like objects containing the strand of the query and subject, respectively.

`max.skipped.exons`

Not supported yet. If `NA` (the default), the number of skipped exons must be 1 or more (there is no max).

for.query.right.end

If TRUE, then the information reported in the output is for the right ends of the paired-end reads. Using for.query.right.end=TRUE with single-end reads is an error.

Details

See ?OverlapEncodings for a short introduction to "overlap encodings".

The topic of working with overlap encodings is covered in details in the "OverlapEncodings" vignette located in this package (**GenomicAlignments**) and accessible with vignette("OverlapEncodings").

Value

For encodeOverlaps: An [OverlapEncodings](#) object. If hits is not supplied, this object is *parallel* to the longest of query and subject, that is, it has the length of the longest and the i-th encoding in it corresponds to the i-th element in the longest. If hits is supplied, then the returned object is *parallel* to it, that is, it has one encoding per hit.

For flipQuery: TODO

For selectEncodingWithCompatibleStrand: TODO

For isCompatibleWithSplicing and isCompatibleWithSkippedExons: A logical vector *parallel* to x.

For extractSteppedExonRanks, extractSpannedExonRanks, and extractSkippedExonRanks: TODO

For extractQueryStartInTranscript: TODO

Author(s)

H. Pages

See Also

- The [OverlapEncodings](#) class for a brief introduction to "overlap encodings".
- The [Hits](#) class defined and documented in the **S4Vectors** package.
- The "OverlapEncodings" vignette in this package.
- [findCompatibleOverlaps](#) for a specialized version of [findOverlaps](#) that uses encodeOverlaps internally to keep only the hits where the junctions in the aligned read are *compatible* with the splicing of the annotated transcript.
- The [GRangesList](#) class defined and documented in the **GenomicRanges** package.
- The [findOverlaps](#) generic function defined in the **IRanges** package.

Examples

```
## -----
## A. BETWEEN 2 RangesList OBJECTS
## -----
## In the context of an RNA-seq experiment, encoding the overlaps
## between 2 GRangesList objects, one containing the reads (the query),
```

```

## and one containing the transcripts (the subject), can be used for
## detecting hits between reads and transcripts that are "compatible"
## with the splicing of the transcript. Here we illustrate this with 2
## RangesList objects, in order to keep things simple:

## 4 aligned reads in the query:
read1 <- IRanges(c(7, 15, 22), c(9, 19, 23)) # 2 junctions
read2 <- IRanges(c(5, 15), c(9, 17)) # 1 junction
read3 <- IRanges(c(16, 22), c(19, 24)) # 1 junction
read4 <- IRanges(c(16, 23), c(19, 24)) # 1 junction
query <- IRangesList(read1, read2, read3, read4)

## 1 transcript in the subject:
tx <- IRanges(c(1, 4, 15, 22, 38), c(2, 9, 19, 25, 47)) # 5 exons
subject <- IRangesList(tx)

## Encode the overlaps:
ovenc <- encodeOverlaps(query, subject)
ovenc
encoding(ovenc)

## Reads that are "compatible" with the transcript can be detected with
## a regular expression (the regular expression below assumes that
## reads have at most 2 junctions):
regex0 <- "(:[fgij]:|:[jg]:|[gf]:|:[jg]:|:g:|[gf]:)"
grepl(regex0, encoding(ovenc)) # read4 is NOT "compatible"

## This was for illustration purpose only. In practise you dont need
## (and should not) use this regular expression, but use instead the
## isCompatibleWithSplicing() utility function:
isCompatibleWithSplicing(ovenc)

## -----
## B. BETWEEN 2 GRangesList OBJECTS
## -----
## With real RNA-seq data, the reads and transcripts will typically be
## stored in GRangesList objects. Please refer to the "OverlapEncodings"
## vignette in this package for realistic examples.

```

findCompatibleOverlaps-methods

Finding hits between reads and transcripts that are compatible with the splicing of the transcript

Description

In the context of an RNA-seq experiment, `findCompatibleOverlaps` (or `countCompatibleOverlaps`) can be used for finding (or counting) hits between reads and transcripts that are *compatible* with the splicing of the transcript.

Usage

```
findCompatibleOverlaps(query, subject)
countCompatibleOverlaps(query, subject)
```

Arguments

query A [GAlignments](#) or [GAlignmentPairs](#) object representing the aligned reads.
subject A [GRangesList](#) object representing the transcripts.

Details

findCompatibleOverlaps is a specialized version of [findOverlaps](#) that uses [encodeOverlaps](#) internally to keep only the hits where the junctions in the aligned read are *compatible* with the splicing of the annotated transcript.

The topic of working with overlap encodings is covered in details in the "OverlapEncodings" vignette located in this package ([GenomicAlignments](#)) and accessible with `vignette("OverlapEncodings")`.

Value

A [Hits](#) object for findCompatibleOverlaps.
An integer vector *parallel* to (i.e. same length as) query.

Author(s)

H. Pages

See Also

- The [findOverlaps](#) generic function defined in the [IRanges](#) package.
- The [encodeOverlaps](#) generic function and [OverlapEncodings](#) class.
- The "OverlapEncodings" vignette in this package.
- [GAlignments](#) and [GAlignmentPairs](#) objects.
- [GRangesList](#) objects in the [GenomicRanges](#) package.

Examples

```
## Here we only show a simple example illustrating the use of
## countCompatibleOverlaps() on a very small data set. Please
## refer to the "OverlapEncodings" vignette in the GenomicAlignments
## package for a comprehensive presentation of "overlap
## encodings" and related tools/concepts (e.g. "compatible"
## overlaps, "almost compatible" overlaps etc...), and for more
## examples.
```

```
## sm_treated1.bam contains a small subset of treated1.bam, a BAM
## file containing single-end reads from the "Pasilla" experiment
## (RNA-seq, Fly, see the pasilla data package for the details)
## and aligned to reference genome BDGP Release 5 (aka dm3 genome on
```

```

## the UCSC Genome Browser):
sm_treated1 <- system.file("extdata", "sm_treated1.bam",
                           package="GenomicAlignments", mustWork=TRUE)

## Load the alignments:
flag0 <- scanBamFlag(isDuplicate=FALSE, isNotPassingQualityControls=FALSE)
param0 <- ScanBamParam(flag=flag0)
gal <- readGAlignments(sm_treated1, use.names=TRUE, param=param0)

## Load the transcripts (IMPORTANT: Like always, the reference genome
## of the transcripts must be *exactly* the same as the reference
## genome used to align the reads):
library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
exbytx <- exonsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene, by="tx", use.names=TRUE)

## Number of "compatible" transcripts per alignment in gal:
gal_ncomptx <- countCompatibleOverlaps(gal, exbytx)
mcols(gal)$ncomptx <- gal_ncomptx
table(gal_ncomptx)
mean(gal_ncomptx >= 1)
## --> 33% of the alignments in gal are "compatible" with at least
## 1 transcript in exbytx.

## Keep only alignments compatible with at least 1 transcript in
## exbytx:
compgal <- gal[gal_ncomptx >= 1]
head(compgal)

```

findMateAlignment *Pairing the elements of a GAlignments object*

Description

Utilities for pairing the elements of a [GAlignments](#) object.

NOTE: Until BioC 2.13, `findMateAlignment` was the power horse used by [readGAlignmentPairsFromBam](#) for pairing the records loaded from a BAM file containing aligned paired-end reads. Starting with BioC 2.14, [readGAlignmentPairsFromBam](#) relies on [readGAlignmentsListFromBam](#) which itself relies on [scanBam](#)(`BamFile(asMates=TRUE)`, ...) for the pairing.

Usage

```

findMateAlignment(x)
makeGAlignmentPairs(x, use.names=FALSE, use.mcols=FALSE)

## Related low-level utilities:
getDumpedAlignments()
countDumpedAlignments()
flushDumpedAlignments()

```


Arguments

x	A named GAlignments object with metadata columns flag, mrnm, and mpos. Typically obtained by loading aligned paired-end reads from a BAM file with: <pre>param <- ScanBamParam(what=c("flag", "mrnm", "mpos")) x <- readGAlignmentsFromBam(..., use.names=TRUE, param=param)</pre>
use.names	Whether the names on the input object should be propagated to the returned object or not.
use.mcols	Names of the metadata columns to propagate to the returned GAlignmentPairs object.

Details

Pairing algorithm used by findMateAlignment: findMateAlignment is the power horse used by makeGAlignmentPairs for pairing the records loaded from a BAM file containing aligned paired-end reads.

It implements the following pairing algorithm:

- First, only records with flag bit 0x1 (multiple segments) set to 1, flag bit 0x4 (segment unmapped) set to 0, and flag bit 0x8 (next segment in the template unmapped) set to 0, are candidates for pairing (see the SAM Spec for a description of flag bits and fields). findMateAlignment will ignore any other record. That is, records that correspond to single-end reads, or records that correspond to paired-end reads where one or both ends are unmapped, are discarded.
- Then the algorithm looks at the following fields and flag bits:
 - (A) QNAME
 - (B) RNAME, RNEXT
 - (C) POS, PNEXT
 - (D) Flag bits 0x10 (segment aligned to minus strand) and 0x20 (next segment aligned to minus strand)
 - (E) Flag bits 0x40 (first segment in template) and 0x80 (last segment in template)
 - (F) Flag bit 0x2 (proper pair)
 - (G) Flag bit 0x100 (secondary alignment)

2 records rec1 and rec2 are considered mates iff all the following conditions are satisfied:

- (A) QNAME(rec1) == QNAME(rec2)
- (B) RNEXT(rec1) == RNAME(rec2) and RNEXT(rec2) == RNAME(rec1)
- (C) PNEXT(rec1) == POS(rec2) and PNEXT(rec2) == POS(rec1)
- (D) Flag bit 0x20 of rec1 == Flag bit 0x10 of rec2 and Flag bit 0x20 of rec2 == Flag bit 0x10 of rec1
- (E) rec1 corresponds to the first segment in the template and rec2 corresponds to the last segment in the template, OR, rec2 corresponds to the first segment in the template and rec1 corresponds to the last segment in the template
- (F) rec1 and rec2 have same flag bit 0x2
- (G) rec1 and rec2 have same flag bit 0x100

Timing and memory requirement of the pairing algorithm: The estimated timings and memory requirements on a modern Linux system are (those numbers may vary depending on your hardware and OS):

nb of alignments	time	required memory
8 millions	28 sec	1.4 GB
16 millions	58 sec	2.8 GB
32 millions	2 min	5.6 GB
64 millions	4 min 30 sec	11.2 GB

This is for a `GAlignments` object coming from a file with an "average nb of records per unique QNAME" of 2.04. A value of 2 (which means the file contains only primary reads) is optimal for the pairing algorithm. A greater value, say > 3, will significantly degrade its performance. An easy way to avoid this degradation is to load only primary alignments by setting the `isNotPrimaryRead` flag to `FALSE` in `ScanBamParam()`. See examples in `?readGAlignmentPairsFromBam` for how to do this.

Ambiguous pairing: The above algorithm will find almost all pairs unambiguously, even when the same pair of reads maps to several places in the genome. Note that, when a given pair maps to a single place in the genome, looking at (A) is enough to pair the 2 corresponding records. The additional conditions (B), (C), (D), (E), (F), and (G), are only here to help in the situation where more than 2 records share the same QNAME. And that works most of the times. Unfortunately there are still situations where this is not enough to solve the pairing problem unambiguously.

For example, here are 4 records (loaded in a `GAlignments` object) that cannot be paired with the above algorithm:

Showing the 4 records as a `GAlignments` object of length 4:

`GAlignments` with 4 alignments and 2 metadata columns:

	seqnames	strand	cigar	qwidth	start	end
	<Rle>	<Rle>	<character>	<integer>	<integer>	<integer>
SRR031714.2658602	chr2R	+	21M384N16M	37	6983850	6984270
SRR031714.2658602	chr2R	+	21M384N16M	37	6983850	6984270
SRR031714.2658602	chr2R	-	13M372N24M	37	6983858	6984266
SRR031714.2658602	chr2R	-	13M378N24M	37	6983858	6984272
	width	njunc	mrnm	mpos		
	<integer>	<integer>	<factor>	<integer>		
SRR031714.2658602	421	1	chr2R	6983858		
SRR031714.2658602	421	1	chr2R	6983858		
SRR031714.2658602	409	1	chr2R	6983850		
SRR031714.2658602	415	1	chr2R	6983850		

Note that the BAM fields show up in the following columns:

- QNAME: the names of the `GAlignments` object (unnamed col)
- RNAME: the `seqnames` col
- POS: the `start` col
- RNEXT: the `mrnm` col
- PNEXT: the `mpos` col

As you can see, the aligner has aligned the same pair to the same location twice! The only difference between the 2 aligned pairs is in the CIGAR i.e. one end of the pair is aligned twice to the same location with exactly the same CIGAR while the other end of the pair is aligned twice to the same location but with slightly different CIGARs.

Now showing the corresponding flag bits:

	isPaired	isProperPair	isUnmappedQuery	hasUnmappedMate	isMinusStrand
[1,]	1	1	0	0	0
[2,]	1	1	0	0	0
[3,]	1	1	0	0	1
[4,]	1	1	0	0	1

	isMateMinusStrand	isFirstMateRead	isSecondMateRead	isNotPrimaryRead
[1,]		1	0	1
[2,]		1	0	1
[3,]		0	1	0
[4,]		0	1	0

	isNotPassingQualityControls	isDuplicate
[1,]		0
[2,]		0
[3,]		0
[4,]		0

As you can see, rec(1) and rec(2) are second mates, rec(3) and rec(4) are both first mates. But looking at (A), (B), (C), (D), (E), (F), and (G), the pairs could be rec(1) <-> rec(3) and rec(2) <-> rec(4), or they could be rec(1) <-> rec(4) and rec(2) <-> rec(3). There is no way to disambiguate! So findMateAlignment is just ignoring (with a warning) those alignments with ambiguous pairing, and dumping them in a place from which they can be retrieved later (i.e. after findMateAlignment has returned) for further examination (see "Dumped alignments" subsection below for the details). In other words, alignments that cannot be paired unambiguously are not paired at all. Concretely, this means that readGAlignmentPairs is guaranteed to return a GAlignmentPairs object where every pair was formed in a non-ambiguous way. Note that, in practice, this approach doesn't seem to leave aside a lot of records because ambiguous pairing events seem pretty rare.

Dumped alignments: Alignments with ambiguous pairing are dumped in a place ("the dump environment") from which they can be retrieved with getDumpedAlignments() after findMateAlignment has returned.

Two additional utilities are provided for manipulation of the dumped alignments: countDumpedAlignments for counting them (a fast equivalent to length(getDumpedAlignments())), and flushDumpedAlignments to flush "the dump environment". Note that "the dump environment" is automatically flushed at the beginning of a call to findMateAlignment.

Value

For findMateAlignment: An integer vector of the same length as x, containing only positive or NA values, where the i-th element is interpreted as follow:

- An NA value means that no mate or more than 1 mate was found for x[i].
- A non-NA value j gives the index in x of x[i]'s mate.

For `makeGAlignmentPairs`: A [GAlignmentPairs](#) object where the pairs are formed internally by calling `findMateAlignment` on `x`.

For `getDumpedAlignments`: NULL or a [GAlignments](#) object containing the dumped alignments. See "Dumped alignments" subsection in the "Details" section above for the details.

For `countDumpedAlignments`: The number of dumped alignments.

Nothing for `flushDumpedAlignments`.

Author(s)

H. Pages

See Also

- [GAlignments](#) and [GAlignmentPairs](#) objects.
- [readGAlignmentsFromBam](#) and [readGAlignmentPairsFromBam](#).

Examples

```
bamfile <- system.file("extdata", "ex1.bam", package="Rsamtools",
                      mustWork=TRUE)
param <- ScanBamParam(what=c("flag", "mrnm", "mpos"))
x <- readGAlignmentsFromBam(bamfile, use.names=TRUE, param=param)

mate <- findMateAlignment(x)
head(mate)
table(is.na(mate))
galp0 <- makeGAlignmentPairs(x)
galp <- makeGAlignmentPairs(x, use.name=TRUE, use.mcols="flag")
galp
colnames(mcols(galp))
colnames(mcols(first(galp)))
colnames(mcols(last(galp)))
```

findOverlaps-methods *Finding overlapping genomic alignments*

Description

Finds range overlaps between a [GAlignments](#), [GAlignmentPairs](#), or [GAlignmentsList](#) object, and another range-based object.

NOTE: The [findOverlaps](#) generic function and methods for [Ranges](#) and [RangesList](#) objects are defined and documented in the **IRanges** package. The methods for [GRanges](#) and [GRangesList](#) objects are defined and documented in the **GenomicRanges** package.

Usage

```
## S4 method for signature GAlignments,GAlignments
findOverlaps(query, subject,
             maxgap = 0L, minoverlap = 1L,
             type = c("any", "start", "end", "within"),
             select = c("all", "first"),
             ignore.strand = FALSE)

## S4 method for signature GAlignments,GAlignments
countOverlaps(query, subject,
             maxgap = 0L, minoverlap = 1L,
             type = c("any", "start", "end", "within"),
             ignore.strand = FALSE)

## S4 method for signature GAlignments,GAlignments
overlapsAny(query, subject,
            maxgap = 0L, minoverlap = 1L,
            type = c("any", "start", "end", "within"),
            ignore.strand = FALSE)

## S4 method for signature GAlignments,GAlignments
subsetByOverlaps(query, subject,
                 maxgap = 0L, minoverlap = 1L,
                 type = c("any", "start", "end", "within"),
                 ignore.strand = FALSE)
```

Arguments

query, subject A [GAlignments](#), [GAlignmentPairs](#), or [GAlignmentsList](#) object for either query or subject. A vector-like object containing ranges for the other one.

maxgap, minoverlap, type, select
See [findOverlaps](#) in the **IRanges** package for a description of these arguments.

ignore.strand When set to TRUE, the strand information is ignored in the overlap calculations.

Details

When the query or the subject (or both) is a [GAlignments](#) object, it is first turned into a [GRangesList](#) object (with `as(, "GRangesList")`) and then the rules described previously apply. [GAlignmentsList](#) objects are coerced to [GAlignments](#) then to a [GRangesList](#). Feature indices are mapped back to the original [GAlignmentsList](#) list elements.

When the query is a [GAlignmentPairs](#) object, it is first turned into a [GRangesList](#) object (with `as(, "GRangesList")`) and then the rules described previously apply.

Value

For `findOverlaps` either a [Hits](#) object when `select = "all"` or an integer vector otherwise.
For `countOverlaps` an integer vector containing the tabulated query overlap hits.

For `overlapsAny` a logical vector of length equal to the number of ranges in query indicating those that overlap any of the ranges in subject.

For `subsetByOverlaps` an object of the same class as query containing the subset that overlapped at least one entity in subject.

See Also

- [findOverlaps](#).
- [Hits-class](#).
- [GRanges-class](#).
- [GRangesList-class](#).
- [GAlignments-class](#).
- [GAlignmentPairs-class](#).
- [GAlignmentsList-class](#).

Examples

```
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
galn <- readGAlignments(ex1_file)

subject <- granges(galn)[1]

## Note the absence of query no. 9 (i.e. galn[9]) in this result:
as.matrix(findOverlaps(galn, subject))

## This is because, by default, findOverlaps()/countOverlaps() are
## strand specific:
galn[8:10]
countOverlaps(galn[8:10], subject)
countOverlaps(galn[8:10], subject, ignore.strand=TRUE)

## Count alignments in galn that DO overlap with subject vs those
## that do NOT:
table(overlapsAny(galn, subject))
## Extract those that DO:
subsetByOverlaps(galn, subject)

## GAlignmentsList
galist <- GAlignmentsList(galn[8:10], galn[3000:3002])
gr <- GRanges(c("seq1", "seq1", "seq2"),
              IRanges(c(15, 18, 1233), width=1),
              strand=c("-", "+", "+"))

countOverlaps(galist, gr)
countOverlaps(galist, gr, ignore.strand=TRUE)
findOverlaps(galist, gr)
findOverlaps(galist, gr, ignore.strand=TRUE)
```

 findSpliceOverlaps-methods

Classify ranges (reads) as compatible with existing genomic annotations or as having novel splice events

Description

The findSpliceOverlaps function identifies ranges (reads) that are compatible with a specific transcript isoform. The non-compatible ranges are analyzed for the presence of novel splice events.

Usage

```
findSpliceOverlaps(query, subject, ignore.strand=FALSE, ...)

## S4 method for signature GRangesList,GRangesList
findSpliceOverlaps(query, subject, ignore.strand=FALSE, ..., cds=NULL)

## S4 method for signature GAlignments,GRangesList
findSpliceOverlaps(query, subject, ignore.strand=FALSE, ..., cds=NULL)

## S4 method for signature GAlignmentPairs,GRangesList
findSpliceOverlaps(query, subject, ignore.strand=FALSE, ..., cds=NULL)

## S4 method for signature BamFile,ANY
findSpliceOverlaps(query, subject, ignore.strand=FALSE, ...,
  param=ScanBamParam(), singleEnd=TRUE)
```

Arguments

query	A GRangesList , GAlignments , GAlignmentPairs , or BamFile object containing the reads. Can also be a single string containing the path to a BAM file. Single or paired-end reads are specified with the <code>singleEnd</code> argument (default FALSE). Paired-end reads can be supplied in a BAM file or GAlignmentPairs object. Single-end are expected to be in a BAM file, GAlignments or GRanges object.
subject	A GRangesList containing the annotations. This list is expected to contain exons grouped by transcripts.
ignore.strand	When set to TRUE, strand information is ignored in the overlap calculations.
...	Additional arguments such as <code>param</code> and <code>singleEnd</code> used in the method for BamFile objects. See below.
cds	Optional GRangesList of coding regions for each transcript in the subject. If provided, the "coding" output column will be a logical vector indicating if the read falls in a coding region. When not provided, the "coding" output is NA.
param	An optional ScanBamParam instance to further influence scanning, counting, or filtering.

singleEnd A logical value indicating if reads are single or paired-end. See [summarizeOverlaps](#) for more information.

Details

When a read maps compatibly and uniquely to a transcript isoform we can quantify the expression and look for shifts in the balance of isoform expression. If a read does not map in compatible way, novel splice events such as splice junctions, novel exons or retentions can be quantified and compared across samples.

findSpliceOverlaps detects which reads (query) match to transcripts (subject) in a compatible fashion. Compatibility is based on both the transcript bounds and splicing pattern. Assessing the splicing pattern involves comparison of the read splices (i.e., the N operations in the CIGAR) with the transcript introns. For paired-end reads, the inter-read gap is not considered a splice junction. The analysis of non-compatible reads for novel splice events is under construction.

Value

The output is a [Hits](#) object with the metadata columns defined below. Each column is a logical indicating if the read (query) met the criteria.

- compatible: Every splice (N) in a read alignment matches an intron in an annotated transcript. The read does not extend into an intron or outside the transcript bounds.
- unique: The read is compatible with only one annotated transcript.
- strandSpecific: The query (read) was stranded.

Author(s)

Michael Lawrence and Valerie Obenchain <vobencha@fhcrc.org>

See Also

- [GRangesList](#) objects in the **GenomicRanges** package.
- [GAlignments](#) and [GAlignmentPairs](#) objects.
- [BamFile](#) objects in the **Rsamtools** package.

Examples

```
## -----
## Isoform expression :
## -----
## findSpliceOverlaps() can assist in quantifying isoform expression
## by identifying reads that map compatibly and uniquely to a
## transcript isoform.
library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
library(pasillaBamSubset)
se <- untreated1_chr4() ## single-end reads
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
exbytx <- exonsBy(txdb, "tx")
cdsbytx <- cdsBy(txdb, "tx")
```



```

param <- ScanBamParam(which=GRanges("chr4", IRanges(1e5,3e5)))
sehits <- findSpliceOverlaps(se, exbytx, cds=cdsbytx, param=param)

## Tally the reads by category to get an idea of read distribution.
lst <- lapply(mcols(sehits), table)
nms <- names(lst)
tbl <- do.call(rbind, lst[nms])
tbl

## Reads compatible with one or more transcript isoforms.
rnms <- rownames(tbl)
tbl[rnms == "compatible", "TRUE"]/sum(tbl[rnms == "compatible",])

## Reads compatible with a single isoform.
tbl[rnms == "unique", "TRUE"]/sum(tbl[rnms == "unique",])

## All reads fall in a coding region as defined by
## the txdb annotation.
lst[["coding"]]

## Check : Total number of reads should be the same across categories.
lapply(lst, sum)

## -----
## Paired-end reads :
## -----
## singleEnd is set to FALSE for a BAM file with paired-end reads.
pe <- untreated3_chr4()
hits2 <- findSpliceOverlaps(pe, exbytx, singleEnd=FALSE, param=param)

## In addition to BAM files, paired-end reads can be supplied in a
## GAlignmentPairs object.
genes <- GRangesList(
  GRanges("chr1", IRanges(c(5, 20), c(10, 25)), "+"),
  GRanges("chr1", IRanges(c(5, 22), c(15, 25)), "+"))
galp <- GAlignmentPairs(
  GAlignments("chr1", 5L, "11M4N6M", strand("+")),
  GAlignments("chr1", 50L, "6M", strand("-")),
  isProperPair=TRUE)
findSpliceOverlaps(galp, genes)

```

GAlignmentPairs-class *GAlignmentPairs* objects

Description

The GAlignmentPairs class is a container for "genomic alignment pairs".

Details

A GAlignmentPairs object is a list-like object where each element describes a pair of genomic alignment.

An "alignment pair" is made of a "first" and a "last" alignment, and is formally represented by a [GAlignments](#) object of length 2. It is typically representing a hit of a paired-end read to the reference genome that was used by the aligner. More precisely, in a given pair, the "first" alignment represents the hit of the first end of the read (aka "first segment in the template", using SAM Spec terminology), and the "last" alignment represents the hit of the second end of the read (aka "last segment in the template", using SAM Spec terminology).

In general, a GAlignmentPairs object will be created by loading records from a BAM (or SAM) file containing aligned paired-end reads, using the `readGAlignmentPairs` function (see below). Each element in the returned object will be obtained by pairing 2 records.

Constructor

`GAlignmentPairs(first, last, isProperPair, names=NULL)`: Low-level GAlignmentPairs constructor. Generally not used directly.

Accessors

In the code snippets below, `x` is a GAlignmentPairs object.

`length(x)`: Return the number of alignment pairs in `x`.

`names(x)`, `names(x) <- value`: Get or set the names on `x`. See [readGAlignmentPairs](#) for how to automatically extract and set the names when reading the alignments from a file.

`first(x, invert.strand=FALSE)`, `last(x, invert.strand=FALSE)`: Get the "first" or "last" alignment for each alignment pair in `x`. The result is a [GAlignments](#) object of the same length as `x`. If `invert.strand=TRUE`, then the strand is inverted on-the-fly, i.e. "+" becomes "-", "-" becomes "+", and "*" remains unchanged.

`left(x)`: Get the "left" alignment for each alignment pair in `x`. By definition, the "left" alignment in a pair is the alignment that is on the + strand. If this is the "first" alignment, then it's returned as-is by `left(x)`, but if this is the "last" alignment, then it's returned by `left(x)` with the strand inverted.

`right(x)`: Get the "right" alignment for each alignment pair in `x`. By definition, the "right" alignment in a pair is the alignment that is on the - strand. If this is the "first" alignment, then it's returned as-is by `right(x)`, but if this is the "last" alignment, then it's returned by `right(x)` with the strand inverted.

`seqnames(x)`: Get the name of the reference sequence for each alignment pair in `x`. When reading the alignments from a BAM file, this comes from the RNAME field which has the same value for the 2 records in a pair ([makeGAlignmentPairs](#), the function used by [readGAlignmentPairsFromBam](#) for doing the pairing, rejects pairs with incompatible RNAME values).

`strand(x)`, `strand(x) <- value`: Get or set the strand for each alignment pair in `x`. By definition (and in a somewhat arbitrary way) the strand of an alignment pair is the strand of the "first" alignment in the pair. In a GAlignmentPairs object, the strand of the "last" alignment in a pair is typically (but not always) the opposite of the strand of the "first" alignment. Note that, currently, [readGAlignmentPairsFromBam](#), the function used internally by

`readGAlignmentPairs` for doing the pairing, rejects pairs where the "first" and "last" alignments are on the same strand, but those pairs might be supported in the future.

`njunc(x)`: Equivalent to `njunc(first(x)) + njunc(last(x))`.

`isProperPair(x)`: Get the "isProperPair" flag bit (bit 0x2 in SAM Spec) set by the aligner for each alignment pair in `x`.

`seqinfo(x)`, `seqinfo(x) <- value`: Get or set the information about the underlying sequences. `value` must be a [Seqinfo](#) object.

`seqlevels(x)`, `seqlevels(x) <- value`: Get or set the sequence levels. `seqlevels(x)` is equivalent to `seqlevels(seqinfo(x))` or to `levels(seqnames(x))`, those 2 expressions being guaranteed to return identical character vectors on a `GAlignmentPairs` object. `value` must be a character vector with no NAs. See [?seqlevels](#) for more information.

`seqlengths(x)`, `seqlengths(x) <- value`: Get or set the sequence lengths. `seqlengths(x)` is equivalent to `seqlengths(seqinfo(x))`. `value` can be a named non-negative integer or numeric vector eventually with NAs.

`isCircular(x)`, `isCircular(x) <- value`: Get or set the circularity flags. `isCircular(x)` is equivalent to `isCircular(seqinfo(x))`. `value` must be a named logical vector eventually with NAs.

`genome(x)`, `genome(x) <- value`: Get or set the genome identifier or assembly name for each sequence. `genome(x)` is equivalent to `genome(seqinfo(x))`. `value` must be a named character vector eventually with NAs.

`seqnameStyle(x)`: Get or set the seqname style for `x`. Note that this information is not stored in `x` but inferred by looking up `seqnames(x)` against a seqname style database stored in the `seqnames.db` metadata package (required). `seqnameStyle(x)` is equivalent to `seqnameStyle(seqinfo(x))` and can return more than 1 seqname style (with a warning) in case the style cannot be determined unambiguously.

Vector methods

In the code snippets below, `x` is a `GAlignmentPairs` object.

`x[i]`: Return a new `GAlignmentPairs` object made of the selected alignment pairs.

List methods

In the code snippets below, `x` is a `GAlignmentPairs` object.

`x[[i]]`: Extract the `i`-th alignment pair as a [GAlignments](#) object of length 2. As expected `x[[i]][1]` and `x[[i]][2]` are respectively the "first" and "last" alignments in the pair.

`unlist(x, use.names=TRUE)`: Return the [GAlignments](#) object conceptually defined by `c(x[[1]], x[[2]], ..., x[[length(x)]])`. `use.names` determines whether `x` names should be propagated to the result or not.

Coercion

In the code snippets below, `x` is a `GAlignmentPairs` object.

`grglist(x, use.mcols=FALSE, order.as.in.query=FALSE, drop.D.ranges=FALSE)`

Return a [GRangesList](#) object of length `length(x)` where the *i*-th element represents the ranges (with respect to the reference) of the *i*-th alignment pair in `x`.

If `use.mcols` is `TRUE` and `x` has metadata columns on it (accessible with `mcols(x)`), they're propagated to the returned object.

IMPORTANT: The strand of the ranges coming from the "last" alignment in the pair is *always* inverted.

The `order.as.in.query` toggle affects the order of the ranges *within* each top-level element of the returned object.

If `FALSE` (the default), then the "left" ranges are placed before the "right" ranges, and, within each left or right group, are ordered from 5' to 3' in elements associated with the plus strand and from 3' to 5' in elements associated with the minus strand. More formally, the *i*-th element in the returned [GRangesList](#) object can be defined as `c(gr11[[i]], gr12[[i]])`, where `gr11` is `grglist(left(x))` and `gr12` is `grglist(right(x))`.

If `TRUE`, then the "first" ranges are placed before the "last" ranges, and, within each first or last group, are *always* ordered from 5' to 3', whatever the strand is. More formally, the *i*-th element in the returned [GRangesList](#) object can be defined as `c(gr11[[i]], gr12[[i]])`, where `gr11` is `grglist(first(x), order.as.in.query=TRUE)` and `gr12` is `grglist(last(x, invert.strand=TRUE), order.as.in.query=TRUE)`.

Note that the relationship between the 2 [GRangesList](#) objects obtained with `order.as.in.query` being respectively `FALSE` or `TRUE` is simpler than it sounds: the only difference is that the order of the ranges in elements associated with the *minus* strand is reversed.

Finally note that, in the latter, the ranges are *always* ordered consistently with the original "query template", that is, in the order defined by walking the "query template" from the beginning to the end.

If `drop.D.ranges` is `TRUE`, then deletions (Ds in the CIGAR) are treated like junctions (Ns in the CIGAR), that is, the ranges corresponding to deletions are dropped.

`granges(x, use.mcols=FALSE)`: Return a [GRanges](#) object of length `length(x)` where each range is obtained by merging all the ranges within the corresponding top-level element in `grglist(x)`.

If `use.mcols` is `TRUE` and `x` has metadata columns on it (accessible with `mcols(x)`), they're propagated to the returned object.

`as(x, "GRangesList")`, `as(x, "GRanges")`: Alternate ways of doing `grglist(x, use.mcols=TRUE)` and `granges(x, use.mcols=TRUE)`, respectively.

`as(x, "GAlignments")`: Equivalent of `unlist(x, use.names=TRUE)`.

Other methods

In the code snippets below, `x` is a [GAlignmentPairs](#) object.

`show(x)`: By default the `show` method displays 5 head and 5 tail elements. This can be changed by setting the global options `showHeadLines` and `showTailLines`. If the object length is less than (or equal to) the sum of these 2 options plus 1, then the full object is displayed. Note that these options also affect the display of [GRanges](#) and [GAlignments](#) objects, as well as other objects defined in the `IRanges` and `Biostrings` packages (e.g. [Ranges](#) and [XStringSet](#) objects).

Author(s)

H. Pages

See Also

- [readGAlignmentPairs](#) for reading aligned paired-end reads from a file (typically a BAM file) into a GAlignmentPairs object.
- [GAlignments](#) objects for handling aligned single-end reads.
- [makeGAlignmentPairs](#) for pairing the elements of a [GAlignments](#) object into a GAlignmentPairs object.
- [junctions-methods](#) for extracting and summarizing junctions from a GAlignmentPairs object.
- [coverage-methods](#) for computing the coverage of a GAlignmentPairs object.
- [findOverlaps-methods](#) for finding range overlaps between a GAlignmentPairs object and another range-based object.
- [seqinfo](#) in the **GenomeInfoDb** package for getting/setting/modifying the sequence information stored in an object.
- The [GRanges](#) and [GRangesList](#) classes defined and documented in the **GenomicRanges** package.

Examples

```
library(Rsamtools) # for the ex1.bam file
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
galp <- readGAlignmentPairs(ex1_file, use.names=TRUE)
galp

length(galp)
head(galp)
head(names(galp))
first(galp)
last(galp)
last(galp, invert.strand=TRUE)
left(galp)
right(galp)
seqnames(galp)
strand(galp)
head(njunc(galp))
table(isProperPair(galp))
seqlevels(galp)

## Rename the reference sequences:
seqlevels(galp) <- sub("seq", "chr", seqlevels(galp))
seqlevels(galp)

galp[[1]]
unlist(galp)

grglist(galp) # a GRangesList object
grglist(galp, order.as.in.query=TRUE)
```

```
stopifnot(identical(unname(elementLengths(grglist(galp))), njunc(galp) + 2L))
granges(galp) # a GRanges object
```

GAlignments-class *GAlignments objects*

Description

The GAlignments class is a simple container whose purpose is to store a set of genomic alignments that will hold just enough information for supporting the operations described below.

Details

A GAlignments object is a vector-like object where each element describes a genomic alignment i.e. how a given sequence (called "query" or "read", typically short) aligns to a reference sequence (typically long).

Typically, a GAlignments object will be created by loading records from a BAM (or SAM) file and each element in the resulting object will correspond to a record. BAM/SAM records generally contain a lot of information but only part of that information is loaded in the GAlignments object. In particular, we discard the query sequences (SEQ field), the query qualities (QUAL), the mapping qualities (MAPQ) and any other information that is not needed in order to support the operations or methods described below.

This means that multi-reads (i.e. reads with multiple hits in the reference) won't receive any special treatment i.e. the various SAM/BAM records corresponding to a multi-read will show up in the GAlignments object as if they were coming from different/unrelated queries. Also paired-end reads will be treated as single-end reads and the pairing information will be lost (see [?AlignmentPairs](#) for how to handle aligned paired-end reads).

Each element of a GAlignments object consists of:

- The name of the reference sequence. (This is the RNAME field in a SAM/BAM record.)
- The strand in the reference sequence to which the query is aligned. (This information is stored in the FLAG field in a SAM/BAM record.)
- The CIGAR string in the "Extended CIGAR format" (see the SAM Format Specifications for the details).
- The 1-based leftmost position/coordinate of the clipped query relative to the reference sequence. We will refer to it as the "start" of the query. (This is the POS field in a SAM/BAM record.)
- The 1-based rightmost position/coordinate of the clipped query relative to the reference sequence. We will refer to it as the "end" of the query. (This is NOT explicitly stored in a SAM/BAM record but can be inferred from the POS and CIGAR fields.) Note that all positions/coordinates are always relative to the first base at the 5' end of the plus strand of the reference sequence, even when the query is aligned to the minus strand.

- The genomic intervals between the "start" and "end" of the query that are "covered" by the alignment. Saying that the full [start,end] interval is covered is the same as saying that the alignment contains no junction (no N in the CIGAR). It is then considered to be a simple alignment. Note that a simple alignment can have mismatches or deletions (in the reference). In other words, a deletion (encoded with a D in the CIGAR) is NOT considered to introduce a gap in the coverage, but a junction is.

Note that the last 2 items are not explicitly stored in the GAlignments object: they are inferred on-the-fly from the CIGAR and the "start".

Optionally, a GAlignments object can have names (accessed thru the `names` generic function) which will be coming from the QNAME field of the SAM/BAM records.

The rest of this man page will focus on describing how to:

- Access the information stored in a GAlignments object in a way that is independent from how the data are actually stored internally.
- How to create and manipulate a GAlignments object.

Constructor

`GAlignments(seqnames=Rle(factor()), pos=integer(), cigar=character(0),`
 Low-level GAlignments constructor. Generally not used directly. Named arguments in ...
 are used as metadata columns.

Accessors

In the code snippets below, `x` is a GAlignments object.

`length(x)`: Return the number of alignments in `x`.

`names(x)`, `names(x) <- value`: Get or set the names on `x`. See `readGAlignments` for how to automatically extract and set the names when reading the alignments from a file.

`seqnames(x)`, `seqnames(x) <- value`: Get or set the name of the reference sequence for each alignment in `x` (see Details section above for more information about the RNAME field of a SAM/BAM file). `value` can be a factor, or a 'factor' `Rle`, or a character vector.

`rname(x)`, `rname(x) <- value`: Same as `seqnames(x)` and `seqnames(x) <- value`.

`strand(x)`, `strand(x) <- value`: Get or set the strand for each alignment in `x` (see Details section above for more information about the strand of an alignment). `value` can be a factor (with levels `+`, `-` and `*`), or a 'factor' `Rle`, or a character vector.

`cigar(x)`: Returns a character vector of length `length(x)` containing the CIGAR string for each alignment.

`qwidth(x)`: Returns an integer vector of length `length(x)` containing the length of the query *after* hard clipping (i.e. the length of the query sequence that is stored in the corresponding SAM/BAM record).

`start(x)`, `end(x)`: Returns an integer vector of length `length(x)` containing the "start" and "end" (respectively) of the query for each alignment. See Details section above for the exact definitions of the "start" and "end" of a query. Note that `start(x)` and `end(x)` are equivalent to `start(granges(x))` and `end(granges(x))`, respectively (or, alternatively, `min(rglist(x))` and `max(rglist(x))`, respectively).

- `width(x)`: Equivalent to `width(granges(x))` (or, alternatively, to `end(x) - start(x) + 1L`). Note that this is generally different from `qwidth(x)` except for alignments with a trivial CIGAR string (i.e. a string of the form "`<n>M`" where `<n>` is a number).
- `njunc(x)`: Returns an integer vector of the same length as `x` containing the number of junctions (i.e. N operations in the CIGAR) in each alignment. Equivalent to `unname(elementLengths(rglist(x))) - 1L`.
- `seqinfo(x)`, `seqinfo(x) <- value`: Get or set the information about the underlying sequences. `value` must be a [Seqinfo](#) object.
- `seqlevels(x)`, `seqlevels(x) <- value`: Get or set the sequence levels. `seqlevels(x)` is equivalent to `seqlevels(seqinfo(x))` or to `levels(seqnames(x))`, those 2 expressions being guaranteed to return identical character vectors on a `GAlignments` object. `value` must be a character vector with no NAs. See [?seqlevels](#) for more information.
- `seqlengths(x)`, `seqlengths(x) <- value`: Get or set the sequence lengths. `seqlengths(x)` is equivalent to `seqlengths(seqinfo(x))`. `value` can be a named non-negative integer or numeric vector eventually with NAs.
- `isCircular(x)`, `isCircular(x) <- value`: Get or set the circularity flags. `isCircular(x)` is equivalent to `isCircular(seqinfo(x))`. `value` must be a named logical vector eventually with NAs.
- `genome(x)`, `genome(x) <- value`: Get or set the genome identifier or assembly name for each sequence. `genome(x)` is equivalent to `genome(seqinfo(x))`. `value` must be a named character vector eventually with NAs.
- `seqnameStyle(x)`: Get or set the seqname style for `x`. Note that this information is not stored in `x` but inferred by looking up `seqnames(x)` against a seqname style database stored in the **seqnames.db** metadata package (required). `seqnameStyle(x)` is equivalent to `seqnameStyle(seqinfo(x))` and can return more than 1 seqname style (with a warning) in case the style cannot be determined unambiguously.

Coercion

In the code snippets below, `x` is a `GAlignments` object.

```
grglist(x, use.mcols=FALSE, order.as.in.query=FALSE, drop.D.ranges=FALSE)
rglist(x, use.mcols=FALSE, order.as.in.query=FALSE, drop.D.ranges=FALSE)
```

Return either a [GRangesList](#) or a [RangesList](#) object of length `length(x)` where the `i`-th element represents the ranges (with respect to the reference) of the `i`-th alignment in `x`.

More precisely, the [RangesList](#) object returned by `rglist(x)` is a [CompressedIRangesList](#) object.

If `use.mcols` is `TRUE` and `x` has metadata columns on it (accessible with `mcols(x)`), they're propagated to the returned object.

The `order.as.in.query` toggle affects the order of the ranges *within* each top-level element of the returned object.

If `FALSE` (the default), then the ranges are ordered from 5' to 3' in elements associated with the plus strand (i.e. corresponding to alignments located on the plus strand), and from 3' to 5' in elements associated with the minus strand. So, whatever the strand is, the ranges are in ascending order (i.e. left-to-right).

If `TRUE`, then the order of the ranges in elements associated with the *minus* strand is reversed. So they end up being ordered from 5' to 3' too, which means that they are now in descending

order (i.e. right-to-left). It also means that, when `order.as.in.query=TRUE` is used, the ranges are *always* ordered consistently with the original "query template", that is, in the order defined by walking the "query template" from the beginning to the end.

If `drop.D.ranges` is `TRUE`, then deletions (D operations in the CIGAR) are treated like junctions (N operations in the CIGAR), that is, the ranges corresponding to deletions are dropped. See Details section above for more information.

`granges(x, use.mcols=FALSE)`, `ranges(x)`: Return either a [GRanges](#) or a [Ranges](#) object of length `length(x)` where each element represents the regions in the reference to which a query is aligned.

More precisely, the [Ranges](#) object returned by `ranges(x)` is an [IRanges](#) object.

If `use.mcols` is `TRUE` and `x` has metadata columns on it (accessible with `mcols(x)`), they're propagated to the returned object.

`as(x, "GRangesList")`, `as(x, "GRanges")`, `as(x, "RangesList")`, `as(x, "Ranges")`: Alternate ways of doing `grglist(x, use.mcols=TRUE)`, `granges(x, use.mcols=TRUE)`, `rglist(x, use.mcols=TRUE)`, and `ranges(x)`, respectively.

In the code snippet below, `x` is a [GRanges](#) object.

`as(from, "GAlignments")`: Creates a `GAlignments` object from a [GRanges](#) object. The metadata columns are propagated. `cigar` values are created from the sequence width unless a "cigar" metadata column already exists in `from`.

Subsetting and related operations

In the code snippets below, `x` is a `GAlignments` object.

`x[i]`: Return a new `GAlignments` object made of the selected alignments. `i` can be a numeric or logical vector.

Combining

`c(...)`: Concatenates the `GAlignments` objects in `...`

Other methods

`show(x)`: By default the `show` method displays 5 head and 5 tail elements. This can be changed by setting the global options `showHeadLines` and `showTailLines`. If the object length is less than (or equal to) the sum of these 2 options plus 1, then the full object is displayed. Note that these options also affect the display of [GRanges](#) and [GAlignmentPairs](#) objects, as well as other objects defined in the [IRanges](#) and [Biostrings](#) packages (e.g. [Ranges](#) and [DNAStringSet](#) objects).

Author(s)

H. Pages and P. Aboyoun

References

<http://samtools.sourceforge.net/>

See Also

- [readGAlignments](#) for reading genomic alignments from a file (typically a BAM file) into a GAlignments object.
- [GAlignmentPairs](#) objects for handling aligned paired-end reads.
- [junctions-methods](#) for extracting and summarizing junctions from a GAlignments object.
- [coverage-methods](#) for computing the coverage of a GAlignments object.
- [findOverlaps-methods](#) for finding overlapping genomic alignments.
- [seqinfo](#) in the **GenomeInfoDb** package for getting/setting/modifying the sequence information stored in an object.
- The [GRanges](#) and [GRangesList](#) classes defined and documented in the **GenomicRanges** package.
- The [CompressedIRangesList](#) class defined and documented in the **IRanges** package.

Examples

```

library(Rsamtools) # for the ex1.bam file
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
gal <- readGAlignments(ex1_file, param=ScanBamParam(what="flag"))
gal

## -----
## A. BASIC MANIPULATION
## -----
length(gal)
head(gal)
names(gal) # no names by default
seqnames(gal)
strand(gal)
head(cigar(gal))
head(qwidth(gal))
table(qwidth(gal))
head(start(gal))
head(end(gal))
head(width(gal))
head(njunc(gal))
seqlevels(gal)

## Rename the reference sequences:
seqlevels(gal) <- sub("seq", "chr", seqlevels(gal))
seqlevels(gal)

grglist(gal) # a GRangesList object
stopifnot(identical(unname(elementLengths(grglist(gal))), njunc(gal) + 1L))
granges(gal) # a GRanges object
rglist(gal) # a CompressedIRangesList object
stopifnot(identical(unname(elementLengths(rglist(gal))), njunc(gal) + 1L))
ranges(gal) # an IRanges object

## Modify the number of lines in show

```

```

options(showHeadLines=3)
options(showTailLines=2)
gal

## Revert to default
options(showHeadLines=NULL)
options(showTailLines=NULL)

## -----
## B. SUBSETTING
## -----
gal[strand(gal) == "-"]
gal[grep("I", cigar(gal), fixed=TRUE)]
gal[grep("N", cigar(gal), fixed=TRUE)] # no junctions

## A confirmation that none of the alignments contains junctions (in
## other words, each alignment can be represented by a single genomic
## range on the reference):
stopifnot(all(njunc(gal) == 0))

## Different ways to subset:
gal[6] # a GAlignments object of length 1
grglist(gal)[[6]] # a GRanges object of length 1
rglist(gal)[[6]] # a NormalIRanges object of length 1

## Unlike N operations, D operations dont introduce gaps:
ii <- grep("D", cigar(gal), fixed=TRUE)
gal[ii]
njunc(gal[ii])
grglist(gal[ii])

## qwidth() vs width():
gal[qwidth(gal) != width(gal)]

## This MUST return an empty object:
gal[cigar(gal) == "35M" & qwidth(gal) != 35]
## but this doesnt have too:
gal[cigar(gal) != "35M" & qwidth(gal) == 35]

```

GAlignmentsList-class *GAlignmentsList* objects

Description

The GAlignmentsList class is a container for storing a collection of [GAlignments](#) objects.

Details

A GAlignmentsList object contains a list of [GAlignments](#) objects. The majority of operations on this page are described in more detail on the GAlignments man page, see `?GAlignments`.

Constructor

`GAlignmentsList(...)`: Creates a `GAlignmentsList` from a list of `GAlignments` objects.

Accessors

In the code snippets below, `x` is a `GAlignmentsList` object.

`length(x)`: Return the number of elements in `x`.

`names(x)`, `names(x) <- value`: Get or set the names of the elements of `x`.

`seqnames(x)`, `seqnames(x) <- value`: Get or set the name of the reference sequences of the alignments in each element of `x`.

`rname(x)`, `rname(x) <- value`: Same as `seqnames(x)` and `seqnames(x) <- value`.

`strand(x)`, `strand(x) <- value`: Get or set the strand of the alignments in each element of `x`.

`cigar(x)`: Returns a character list of length `length(x)` containing the CIGAR string for the alignments in each element of `x`.

`qwidth(x)`: Returns an integer list of length `length(x)` containing the length of the alignments in each element of `x` *after* hard clipping (i.e. the length of the query sequence that is stored in the corresponding SAM/BAM record).

`start(x)`, `end(x)`: Returns an integer list of length `length(x)` containing the "start" and "end" (respectively) of the alignments in each element of `x`.

`width(x)`: Returns an integer list of length `length(x)` containing the "width" of the alignments in each element of `x`.

`njunc(x)`: Returns an integer list of length `x` containing the number of junctions (i.e. N operations in the CIGAR) for the alignments in each element of `x`.

`seqinfo(x)`, `seqinfo(x) <- value`: Get or set the information about the underlying sequences. `value` must be a `Seqinfo` object.

`seqlevels(x)`, `seqlevels(x) <- value`: Get or set the sequence levels of the alignments in each element of `x`.

`seqlengths(x)`, `seqlengths(x) <- value`: Get or set the sequence lengths for each element of `x`. `seqlengths(x)` is equivalent to `seqlengths(seqinfo(x))`. `value` can be a named non-negative integer or numeric vector eventually with NAs.

`isCircular(x)`, `isCircular(x) <- value`: Get or set the circularity flags for the alignments in each element in `x`. `value` must be a named logical list eventually with NAs.

`genome(x)`, `genome(x) <- value`: Get or set the genome identifier or assembly name for the alignments in each element of `x`. `value` must be a named character list eventually with NAs.

`seqnameStyle(x)`: Get or set the `seqname` style for alignments in each element of `x`.

Coercion

In the code snippets below, `x` is a `GAlignmentsList` object.

`granges(x, use.mcols=FALSE, ignore.strand=FALSE)`, `ranges(x)`: Return either a `GRanges` or a `IRanges` object of length `length(x)`. Note this coercion IGNORES the `cigar` information.

The resulting ranges span the entire range, including any gaps or spaces between paired-end reads.

If use.mcols is TRUE and x has metadata columns on it (accessible with mcols(x)), they're propagated to the returned object.

granges coercion supports ignore.strand to allow ranges of opposite strand to be combined (see examples). All ranges in the resulting GRanges will have strand '*'.

grglist(x, use.mcols=FALSE, ignore.strand=FALSE), rglis(x, use.mcols=FALSE): Return either a GRangesList or a IRangesList object of length length(x). This coercion RESPECTS the cigar information. The resulting ranges are fragments of the original ranges that do not include gaps or spaces between paired-end reads.

If use.mcols is TRUE and x has metadata columns on it (accessible with mcols(x)), they're propagated to the returned object.

grglist coercion supports ignore.strand to allow ranges of opposite strand to be combined (see examples). All ranges in the resulting GRangesList will have strand '*'.

as(x, "GRangesList"), as(x, "GRanges"), as(x, "RangesList"), as(x, "Ranges"): Alternate ways of doing grglist(x, use.mcols=TRUE), granges(x, use.mcols=TRUE), rglis(x, use.mcols=TRUE), and ranges(x), respectively.

as.data.frame(x, row.names = NULL, optional = FALSE, ..., value.name = "value", use.outer.mcols = ...): Coerces x to a data.frame. See as.data.frame on the List man page for details (?List).

as(x, "GAlignmentsList"): Here x is a GAlignmentPairs object. Return a GAlignmentsList object of length length(x) where the i-th list element represents the ranges of the i-th alignment pair in x.

Subsetting and related operations

In the code snippets below, x is a GAlignmentsList object.

x[i], x[[i]] <- value: Get or set list elements i. i can be a numeric or logical vector. value must be a GAlignments.

x[[i]], x[[[i]]] <- value: Same as x[i], x[[i]] <- value.

x[i, j], x[[i, j]] <- value: Get or set list elements i with optional metadata columns j. i can be a numeric, logical or missing. value must be a GAlignments.

Combining

c(...): Concatenates the GAlignmentsList objects in ...

Author(s)

Valerie Obenchain <vobencha@fhcrc.org>

References

<http://samtools.sourceforge.net/>

See Also

- [readGAlignmentsList](#) for reading genomic alignments from a file (typically a BAM file) into a GAlignmentsList object.
- [GAlignments](#) and [GAlignmentPairs](#) objects for handling aligned single- and paired-end reads, respectively.
- [junctions-methods](#) for extracting and summarizing junctions from a GAlignmentsList object.
- [findOverlaps-methods](#) for finding range overlaps between a GAlignmentsList object and another range-based object.
- [seqinfo](#) in the **GenomeInfoDb** package for getting/setting/modifying the sequence information stored in an object.
- The [GRanges](#) and [GRangesList](#) classes defined and documented in the **GenomicRanges** package.

Examples

```
gal1 <- GAlignments(
  seqnames=Rle(factor(c("chr1", "chr2", "chr1", "chr3")),
    c(1, 3, 2, 4)),
  pos=1:10, cigar=paste0(10:1, "M"),
  strand=Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
  names=head(letters, 10), score=1:10)

gal2 <- GAlignments(
  seqnames=Rle(factor(c("chr2", "chr4")), c(3, 4)), pos=1:7,
  cigar=c("5M", "3M2N3M2N3M", "5M", "10M", "5M1N4M", "8M2N1M", "5M"),
  strand=Rle(strand(c("-", "+")), c(4, 3)),
  names=tail(letters, 7), score=1:7)

galist <- GAlignmentsList(noGaps=gal1, Gaps=gal2)

## -----
## A. BASIC MANIPULATION
## -----

length(galist)
names(galist)
seqnames(galist)
strand(galist)
head(cigar(galist))
head(qwidth(galist))
head(start(galist))
head(end(galist))
head(width(galist))
head(njunc(galist))
seqlevels(galist)

## Rename the reference sequences:
seqlevels(galist) <- sub("chr", "seq", seqlevels(galist))
seqlevels(galist)
```

```

grglist(galist) # a GRangesList object
rglist(galist) # an IRangesList object

## -----
## B. SUBSETTING
## -----

galist[strand(galist) == "-"]
has_junctions <- sapply(galist,
                        function(x) any(grepl("N", cigar(x), fixed=TRUE)))
galist[has_junctions]

## Different ways to subset:
galist[2]           # a GAlignments object of length 1
galist[[2]]        # a GAlignments object of length 1
grglist(galist[2]) # a GRangesList object of length 1
rglist(galist[2])  # a NormalIRangesList object of length 1

## -----
## C. mcols()/elementMetadata()
## -----

## Metadata can be defined on the individual GAlignment elements
## and the overall GAlignmentsList object. By default, level=between
## extracts the GAlignmentsList metadata. Using level=within
## will extract the metadata on the individual GAlignments objects.

mcols(galist) ## no metadata on the GAlignmentsList object
mcols(galist, level="within")

## -----
## D. readGAlignmentsList()
## -----

library(pasillaBamSubset)

## file as character.
f1 <- untreated3_chr4()
galist1 <- readGAlignmentsList(f1)

galist1[1:3]
length(galist1)
table(elementLengths(galist1))

## When file is a BamFile, asMates must be TRUE. If FALSE,
## the data are treated as single-end and each list element of the
## GAlignmentsList will be of length 1. For single-end data
## use readGAlignments() instead of readGAlignmentsList().
bf <- BamFile(f1, yieldSize=3, asMates=TRUE)
readGAlignmentsList(bf)

```

```

## Use a param to fine tune the results.
param <- ScanBamParam(flag=scanBamFlag(isProperPair=TRUE))
galist2 <- readGAlignmentsList(f1, param=param)
length(galist2)

## -----
## E. COERCION
## -----

## The granges() and grlist() coercions support ignore.strand to
## allow ranges from different strand to be combined. In this example
## paired-end reads aligned to opposite strands were read into a
## GAlignmentsList. If the desired operation is to combine these ranges,
## regardless of gaps or the space between pairs, ignore.strand must be TRUE.
granges(galist[1])
granges(galist[1], ignore.strand=TRUE)

## grglist() splits ranges by gap and the space between list elements.
galist <- GAlignmentsList(noGaps=gal1, Gaps=gal2)
grglist(galist)
grglist(galist, ignore.strand=TRUE)

```

GappedReads-class *(Legacy) GappedReads objects*

Description

The GappedReads class extends the [GAlignments](#) class.

A GappedReads object contains all the information contained in a [GAlignments](#) object plus the sequences of the queries. Those sequences can be accessed via the `qseq` accessor.

Constructor

GappedReads objects are typically created when reading a file containing aligned reads with the [readGappedReads](#) function.

Accessors

In the code snippets below, `x` is a GappedReads object.

`qseq(x)`: Extracts the sequences of the queries as a [DNAStrngSet](#) object.

Author(s)

H. Pages and P. Aboyoun

References

<http://samtools.sourceforge.net/>

See Also

- [GAlignments](#) objects.
- [readGappedReads](#).

Examples

```

greads_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
greads <- readGappedReads(greads_file)
greads
qseq(greads)

```

intra-range-methods *Intra range transformations of a GAlignments or GAlignmentsList object*

Description

This man page documents intra range transformations of a [GAlignments](#) or [GAlignmentsList](#) object.

See [?intra-range-methods](#) and [?inter-range-methods](#) in the **IRanges** package for a quick introduction to intra range and inter range transformations.

Intra range methods for [GRanges](#) and [GRangesList](#) objects are defined and documented in the **GenomicRanges** package.

Usage

```

## S4 method for signature GAlignments
narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)
## S4 method for signature GAlignmentsList
narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)

## S4 method for signature GAlignments
qnarrow(x, start=NA, end=NA, width=NA)
## S4 method for signature GAlignmentsList
qnarrow(x, start=NA, end=NA, width=NA)

```

Arguments

x A [GAlignments](#) or [GAlignmentsList](#) object.

start, end, width Vectors of integers. NAs and negative values are accepted and "solved" according to the rules of the SEW (Start/End/Width) interface (see [?solveUserSEW](#) for more information about the SEW interface).

 See [?intra-range-methods](#) for more information about the start, end, and width arguments.

use.names See [?intra-range-methods](#).

Details

- `() narrow` on a [GAlignments](#) object behaves like on a [Ranges](#) object. See [?intra-range-methods](#) for the details.

A major difference though is that it returns a [GAlignments](#) object instead of a [Ranges](#) object. Unlike with `qnarrow` (see below), the `start/end/width` arguments here describe the narrowing on the reference side, not the query side.

- `() qnarrow` on a [GAlignments](#) object behaves like `narrow` except that the `start/end/width` arguments here specify the narrowing with respect to the query sequences. `qnarrow` on a [GAlignmentsList](#) object returns a [GAlignmentsList](#) object.

Value

An object of the same class as, and *parallel* to (i.e. same length and names as), the original object `x`.

Note

There is no difference between `narrow` and `qnarrow` when all the alignments have a simple CIGAR (i.e. no indels or junctions).

Author(s)

H. Pages and V. Obenchain <vobencha@fhcrc.org>

See Also

- [GAlignments](#) and [GAlignmentsList](#) objects.
- The [intra-range-methods](#) man page in the **IRanges** package.
- The [intra-range-methods](#) man page in the **GenomicRanges** package.

Examples

```
## -----
## A. ON A GAlignments OBJECT
## -----
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
gal <- readGAlignments(ex1_file, param=ScanBamParam(what="flag"))
gal

## This trims 3 nucleotides on the left and 5 nucleotides on the right
## of each alignment:
qnarrow(gal, start=4, end=-6)
## Note that the start and end arguments specify what part of each
## query sequence should be kept (negative values being relative to the
## right end of the query sequence), not what part should be trimmed.

## Trimming on the left doesnt change the "end" of the queries.
qnarrow(gal, start=21)
stopifnot(identical(end(qnarrow(gal, start=21)), end(gal)))
```

```
## -----
## B. ON A GAlignmentsList OBJECT
## -----
gal1 <- GAlignments(
  seqnames=Rle(factor(c("chr1", "chr2", "chr1", "chr3")),
    c(1, 3, 2, 4)),
  pos=1:10, cigar=paste0(10:1, "M"),
  strand=Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
  names=head(letters, 10), score=1:10)

gal2 <- GAlignments(
  seqnames=Rle(factor(c("chr2", "chr4")), c(3, 4)), pos=1:7,
  cigar=c("5M", "3M2N3M2N3M", "5M", "10M", "5M1N4M", "8M2N1M", "5M"),
  strand=Rle(strand(c("-", "+")), c(4, 3)),
  names=tail(letters, 7), score=1:7)

galist <- GAlignmentsList(noGaps=gal1, Gaps=gal2)
galist

qnarrows(galist)
```

junctions-methods

Extract junctions from genomic alignments

Description

Given an object `x` containing genomic alignments (e.g. a `GAlignments`, `GAlignmentPairs`, or `GAlignmentsList` object), `junctions(x)` extracts the junctions from it and `summarizeJunctions(x)` extracts and summarizes them.

`readTopHatJunctions` and `readSTARJunctions` are utilities for importing the junction file generated by the TopHat and STAR aligners, respectively.

Usage

```
## junctions() and summarizeJunctions()
## -----

junctions(x, use.mcols=FALSE, ...)

## S4 method for signature GAlignments
junctions(x, use.mcols=FALSE)

## S4 method for signature GAlignmentPairs
junctions(x, use.mcols=FALSE)

## S4 method for signature GAlignmentsList
junctions(x, use.mcols=FALSE, ignore.strand=FALSE)
```

```

## summarizeJunctions() and NATURAL_INTRON_MOTIFS
## -----

summarizeJunctions(x, with.revmap=FALSE, genome=NULL)

NATURAL_INTRON_MOTIFS

## Utilities for importing the junction file generated by some aligners
## -----

readTopHatJunctions(file, file.is.raw.juncs=FALSE)

readSTARJunctions(file)

```

Arguments

x	A GAlignments , GAlignmentPairs , or GAlignmentsList object.
use.mcols	TRUE or FALSE (the default). Whether the metadata columns on x (accessible with <code>mcols(x)</code>) should be propagated to the returned object or not.
...	Additional arguments, for use in specific methods.
ignore.strand	TRUE or FALSE (the default). If set to TRUE, then the strand of x is set to "*" prior to any computation.
with.revmap	TRUE or FALSE (the default). If set to TRUE, then a revmap metadata column is added to the output of <code>summarizeJunctions</code> . This metadata column is an IntegerList object representing the mapping from each element in the output (i.e. each junction) to the corresponding elements in the input x.
genome	NULL (the default), or a BSgenome object containing the sequences of the reference genome that was used to align the reads, or the name of this reference genome specified in a way that is accepted by the <code>getBSgenome</code> function defined in the BSgenome software package. In that case the corresponding BSgenome data package needs to be already installed (see <code>?getBSgenome</code> in the BSgenome package for the details). If genome is supplied, then the <code>intron_motif</code> and <code>intron_strand</code> metadata columns are computed (based on the dinucleotides found at the intron boundaries) and added to the output of <code>summarizeJunctions</code> . See the Value section below for a description of these metadata columns.
file	The path (or a connection) to the junction file generated by the aligner. This file should be the <code>junctions.bed</code> or <code>new_list.juncs</code> file for <code>readTopHatJunctions</code> , and the <code>SJ.out.tab</code> file for <code>readSTARJunctions</code> .
file.is.raw.juncs	TRUE or FALSE (the default). If set to TRUE, then the input file is assumed to be a TopHat <code>.juncs</code> file instead of the <code>junctions.bed</code> file generated by TopHat. A TopHat <code>.juncs</code> file can be obtained by passing the <code>junctions.bed</code> file thru TopHat's <code>bed_to_juncs</code> script. See the TopHat manual at http://tophat.cbcb.umd.edu/manual.shtml for more information.

Details

An N operation in the CIGAR of a genomic alignment is interpreted as a junction. `junctions(x)` will return the genomic ranges of all junctions found in `x`.

More precisely, if `x` is a [GAlignments](#) object, `junctions(x)` is equivalent to:

```
psetdiff(granges(x), grglist(x, order.as.in.query=TRUE))
```

On a `x` is a [GAlignmentPairs](#) object, it's equivalent to (but faster than):

```
mendoapply(c, junctions(first(x)), junctions(last(x)))
```

`NATURAL_INTRON_MOTIFS` is a predefined character vector containing the 5 natural intron motifs described at <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC84117/>.

Value

`junctions(x)` returns the genomic ranges of the junctions in a [GRangesList](#) object *parallel* to `x` (i.e. with 1 list element per element in `x`). If `x` has names on it, they're propagated to the returned object. If `use.mcols` is `TRUE` and `x` has metadata columns on it (accessible with `mcols(x)`), they're propagated to the returned object.

`summarizeJunctions` returns the genomic ranges of the unique junctions in `x` in an unstranded [GRanges](#) object with the following metadata columns:

- `score`: The total number of alignments crossing each junction, i.e., that have the junction encoded in their CIGAR.
- `plus_score` and `minus_score`: The strand-specific number of alignments crossing each junction.
- `revmap`: [Only if `with.revmap` was set to `TRUE`.] An [IntegerList](#) object representing the mapping from each element in the output (i.e. each junction) to the corresponding elements in input `x`.
- `intron_motif` and `intron_strand`: [Only if genome was supplied.] The intron motif and strand for each junction, based on the dinucleotides found in the genome sequences at the intron boundaries. The `intron_motif` metadata column is a factor whose levels are the 5 natural intron motifs stored in predefined character vector `NATURAL_INTRON_MOTIFS`. If the dinucleotides found at the intron boundaries don't match any of these natural intron motifs, then `intron_motif` and `intron_strand` are set to `NA` and `*`, respectively.

`readTopHatJunctions` and `readSTARJunctions` return the junctions reported in the input file in a stranded [GRanges](#) object. With the following metadata columns for `readTopHatJunctions` (when reading in the `junctions.bed` file):

- `name`: An id assigned by TopHat to each junction. This id is of the form `JUNC00000017` and is unique within the `junctions.bed` file.
- `score`: The total number of alignments crossing each junction.

With the following metadata columns for `readSTARJunctions`:

- `intron_motif` and `intron_strand`: The intron motif and strand for each junction, based on the code found in the input file (0: non-canonical, 1: GT/AG, 2: CT/AC, 3: GC/AG, 4: CT/GC, 5: AT/AC, 6: GT/AT). Note that of the 5 natural intron motifs stored in predefined character vector `NATURAL_INTRON_MOTIFS`, only the first 3 are assigned codes by the STAR software (2 codes per motif, one if the intron is on the plus strand and one if it's on the minus strand). Thus the `intron_motif` metadata column is a factor with only 3 levels. If code is 0, then `intron_motif` and `intron_strand` are set to NA and *, respectively.
- `um_reads`: The number of uniquely mapping reads crossing the junction (a pair where the 2 alignments cross the same junction is counted only once).
- `mm_reads`: The number of multi-mapping reads crossing the junction (a pair where the 2 alignments cross the same junction is counted only once).

See STAR manual at <https://code.google.com/p/rna-star/> for more information.

Author(s)

H. Pages

References

<http://www.ncbi.nlm.nih.gov/pmc/articles/PMC84117/> for the 5 natural intron motifs stored in predefined character vector `NATURAL_INTRON_MOTIFS`.

TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions

- TopHat2 paper: <http://genomebiology.com/2013/14/4/r36>
- TopHat2 software and manual: <http://tophat.cbcb.umd.edu/>

STAR: ultrafast universal RNA-seq aligner

- STAR paper: <http://bioinformatics.oxfordjournals.org/content/early/2012/10/25/bioinformatics.bts635>
- STAR software and manual: <https://code.google.com/p/rna-star/>

See Also

- `GAlignments`, `GAlignmentPairs`, and `GAlignmentsList` objects.
- The `GRanges` and `GRangesList` classes defined and documented in the **GenomicRanges** package.
- The `IntegerList` class defined and documented in the **IRanges** package.
- The `getBSgenome` function in the **BSgenome** package, for searching the installed BSgenome data packages for the specified genome and returning it as a `BSgenome` object.
- The `readGAlignments` and `readGAlignmentPairs` functions for reading genomic alignments from a file.
- The `extractList` function in the **IRanges** package, for extracting groups of elements from a vector-like object and returning them into a `List` object.


```

mcols(junc_summary)$score2 <- junc_score2

## If the name of the reference genome is specified thru the genome
## argument (in which case the corresponding BSgenome data package needs
## to be installed), then summarizeJunctions() returns the intron motif
## and strand for each junction.
## Since the reads in RNAseqData.HNRNPC.bam.chr14 were aligned to
## the hg19 genome, the following requires that you have
## BSgenome.Hsapiens.UCSC.hg19 installed:
junc_summary <- summarizeJunctions(gal, with.revmap=TRUE, genome="hg19")
mcols(junc_summary)$score2 <- junc_score2 # putting score2 back

## The "intron_motif" metadata column is a factor whose levels are the
## 5 natural intron motifs stored in predefined character vector
## NATURAL_INTRON_MOTIFS:
table(mcols(junc_summary)$intron_motif)

## -----
## C. STRANDED RNA-seq PROTOCOL
## -----

## Here is a simple test for checking whether the RNA-seq protocol was
## stranded or not:
strandedTest <- function(plus_score, minus_score)
  (sum(plus_score ^ 2) + sum(minus_score ^ 2)) /
  sum((plus_score + minus_score) ^ 2)

## The result of this test is guaranteed to be >= 0.5 and <= 1.
## If, for each junction, the strand of the crossing alignments looks
## random (i.e. "plus_score" and "minus_score" are close), then
## strandedTest() will return a value close to 0.5. If it doesnt look
## random (i.e. for each junction, one of "plus_score" and "minus_score"
## is much bigger than the other), then strandedTest() will return a
## value close to 1.

## If the reads are single-end, the test is meaningful when applied
## directly on junc_summary. However, for the test to be meaningful
## on paired-end reads, it needs to be applied on the first and last
## alignments separately:
junc_summary1 <- summarizeJunctions(first(galp))
junc_summary2 <- summarizeJunctions(last(galp))
strandedTest(mcols(junc_summary1)$plus_score,
             mcols(junc_summary1)$minus_score)
strandedTest(mcols(junc_summary2)$plus_score,
             mcols(junc_summary2)$minus_score)
## Both values are close to 0.5 which suggests that the RNA-seq protocol
## used for this experiment was not stranded.

## -----
## UTILITIES FOR IMPORTING THE JUNCTION FILE GENERATED BY SOME ALIGNERS
## -----

## The TopHat aligner generates a junctions.bed file where it reports

```



```

## all the junctions satisfying some "quality" criteria (see the TopHat
## manual at http://tophat.cbcb.umd.edu/manual.shtml for more
## information). This file can be loaded with readTopHatJunctions():
runname <- names(RNAseqData.HNRNPC.bam.chr14_BAMFILES)[1]
junctions_file <- system.file("extdata", "tophat2_out", runname,
                             "junctions.bed",
                             package="RNAseqData.HNRNPC.bam.chr14")
th_junctions <- readTopHatJunctions(junctions_file)

## Comparing the "TopHat junctions" with the result of
## summarizeJunctions():
th_junctions14 <- th_junctions
seqlevels(th_junctions14, force=TRUE) <- "chr14"
mcols(th_junctions14)$intron_strand <- strand(th_junctions14)
strand(th_junctions14) <- "*"

## All the "TopHat junctions" are in junc_summary:
stopifnot(all(th_junctions14 %in% junc_summary))

## But not all the junctions in junc_summary are reported by TopHat
## (thats because TopHat reports only junctions that satisfy some
## "quality" criteria):
is_in_th_junctions14 <- junc_summary %in% th_junctions14
table(is_in_th_junctions14) # 32 junctions are not in TopHats
                           # junctions.bed file
junc_summary2 <- junc_summary[is_in_th_junctions14]

## junc_summary2 and th_junctions14 contain the same junctions in
## the same order:
stopifnot(all(junc_summary2 == th_junctions14))

## Lets merge their metadata columns. We use our own version of
## merge() for this, which is stricter (it checks that the common
## columns are the same in the 2 data frames to merge) and also
## simpler:
merge2 <- function(df1, df2)
{
  common_colnames <- intersect(colnames(df1), colnames(df2))
  lapply(common_colnames,
         function(colname)
           stopifnot(all(df1[, colname] == df2[, colname])))
  extra_mcolnames <- setdiff(colnames(df2), colnames(df1))
  cbind(df1, df2[, extra_mcolnames, drop=FALSE])
}

mcols(th_junctions14) <- merge2(mcols(th_junctions14),
                               mcols(junc_summary2))

## Here is a peculiar junction reported by TopHat:
idx0 <- which(mcols(th_junctions14)$score2 == 0L)
th_junctions14[idx0]
gal[mcols(th_junctions14)$revmap[[idx0]]]
## The junction is crossed by 5 alignments (score is 5), all of which

```

```
## have a mapping quality of 0!
```

map-methods

Mapping ranges between sequences

Description

map and pmap are defunct. Use mapCoords and pmapCoords instead.

A method for translating a set of input ranges through a [GAlignments](#) object. Returns a GRangesMapping object.

NOTE: The [map](#) generic function is defined and documented in the **IRanges** package. A method for translating a set of input ranges through a [GRangesList](#) object is defined and documented in the **GenomicRanges** package.

Usage

```
## S4 method for signature GenomicRanges,GAlignments
map(from, to)
```

Arguments

from	The input ranges to map, usually a GenomicRanges
to	The alignment between the sequences in from and the sequences in the result.

Details

Each element in to is taken to represent the alignment of a (read) sequence. The CIGAR string is used to translate the input ranges to be relative to the read start. This is useful, for example, when determining the cycle (read position) at which a particular genomic mismatch occurs.

Value

An object of class GRangesMapping, which consists of a Hits instance and GRanges instance.

Author(s)

M. Lawrence

See Also

[mapCoords](#) and [pmapCoords](#) methods.

mapCoords-methods *Mapping ranges between sequences*

Description

A method for translating a set of input ranges through a [GAlignments](#) object. Returns a [GenomicRanges](#) object.

NOTE: The [mapCoords](#) generic function is defined and documented in the **IRanges** package. A method for translating a set of input ranges through a [GRangesList](#) object is defined and documented in the **GenomicRanges** package.

Usage

```
## S4 method for signature GenomicRanges,GAlignments
mapCoords(x, to, ...)
```

Arguments

x	The input ranges to map, usually a GRanges .
to	The alignment between the sequences in x and the sequences in the result.
...	Arguments passed to other methods.

Details

Each element in to is taken to represent the alignment of a (read) sequence. The CIGAR string is used to translate the input ranges to be relative to the read start. This is useful, for example, when determining the cycle (read position) at which a particular genomic mismatch occurs.

Value

A [GRanges](#) object of mapped coordinates with matching data as metadata columns ('queryHits' and 'subjectHits').

Matching data are the result of calling `findOverlaps` with type 'within' on ranges in x (the query) and the ranges in to (the subject). Matching can be many-to-one or one-to-many; one row is reported for each match.

Author(s)

M. Lawrence

See Also

The generic [mapCoords-methods](#) in the **IRanges** package.

Examples

```
bf <- system.file("extdata", "ex1.bam", package="Rsamtools", mustWork=TRUE)
gal <- readGAlignmentsFromBam(bf)
gr <- GRanges(c("seq1", "seq1", "seq2"), IRanges(c(3, 69, 1529), width=5))
mapCoords(gr, gal)
```

 OverlapEncodings-class

OverlapEncodings objects

Description

The OverlapEncodings class is a container for storing the "overlap encodings" returned by the [encodeOverlaps](#) function.

Usage

```
## OverlapEncodings accessors:

## S4 method for signature OverlapEncodings
length(x)
## S4 method for signature OverlapEncodings
Loffset(x)
## S4 method for signature OverlapEncodings
Roffset(x)
## S4 method for signature OverlapEncodings
encoding(x)
## S4 method for signature OverlapEncodings
levels(x)
## S4 method for signature OverlapEncodings
flippedQuery(x)

## S4 method for signature OverlapEncodings
Lencoding(x)
## S4 method for signature OverlapEncodings
Rencoding(x)

## S4 method for signature OverlapEncodings
njunc(x)
## S4 method for signature OverlapEncodings
Lnjunc(x)
## S4 method for signature OverlapEncodings
Rnjunc(x)

## Coercing an OverlapEncodings object:

## S4 method for signature OverlapEncodings
```

```

as.data.frame(x, row.names=NULL, optional=FALSE, ...)

## Low-level related utilities:

## S4 method for signature character
Lencoding(x)
## S4 method for signature character
Rencoding(x)
## S4 method for signature character
njunc(x)
## S4 method for signature character
Lnjunc(x)
## S4 method for signature character
Rnjunc(x)

## S4 method for signature factor
Lencoding(x)
## S4 method for signature factor
Rencoding(x)
## S4 method for signature factor
njunc(x)
## S4 method for signature factor
Lnjunc(x)
## S4 method for signature factor
Rnjunc(x)

```

Arguments

x	An OverlapEncodings object. For the low-level utilities, x can also be a character vector or factor containing encodings.
row.names	NULL or a character vector.
optional, ...	Ignored.

Details

Given a query and a subject of the same length, both list-like objects with top-level elements typically containing multiple ranges (e.g. [RangesList](#) objects), the "overlap encoding" of the *i*-th element in query and *i*-th element in subject is a character string describing how the ranges in query[[*i*]] are *qualitatively* positioned relatively to the ranges in subject[[*i*]].

The [encodeOverlaps](#) function computes those overlap encodings and returns them in an OverlapEncodings object of the same length as query and subject.

The topic of working with overlap encodings is covered in details in the "OverlapEncodings" vignette located in this package ([GenomicAlignments](#)) and accessible with `vignette("OverlapEncodings")`.

OverlapEncodings accessors

In the following code snippets, *x* is an OverlapEncodings object typically obtained by a call to [encodeOverlaps](#)(query, subject).

`length(x)`: Get the number of elements (i.e. encodings) in `x`. This is equal to `length(query)` and `length(subject)`.

`Loffset(x)`, `Roffset(x)`: Get the "left offsets" and "right offsets" of the encodings, respectively. Both are integer vectors of the same length as `x`.

Let's denote $Q_i = \text{query}[[i]]$, $S_i = \text{subject}[[i]]$, and $[q_1, q_2]$ the range covered by Q_i i.e. $q_1 = \min(\text{start}(Q_i))$ and $q_2 = \max(\text{end}(Q_i))$, then `Loffset(x)[i]` is the number L of ranges at the *head* of S_i that are strictly to the left of all the ranges in Q_i i.e. L is the greatest value such that $\text{end}(S_i)[k] < q_1 - 1$ for all k in `seq_len(L)`. Similarly, `Roffset(x)[i]` is the number R of ranges at the *tail* of S_i that are strictly to the right of all the ranges in Q_i i.e. R is the greatest value such that $\text{start}(S_i)[\text{length}(S_i) + 1 - k] > q_2 + 1$ for all k in `seq_len(L)`.

`encoding(x)`: Factor of the same length as `x` where the i -th element is the encoding obtained by comparing each range in Q_i with all the ranges in $tS_i = S_i[(1+L):(\text{length}(S_i)-R)]$ (tS_i stands for "trimmed S_i "). More precisely, here is how this encoding is obtained:

1. All the ranges in Q_i are compared with $tS_i[1]$, then with $tS_i[2]$, etc... At each step (one step per range in tS_i), comparing all the ranges in Q_i with $tS_i[k]$ is done with `rangeComparisonCodeToLetter(compare(Q_i, tS_i[k]))`. So at each step, we end up with a vector of M single letters (where M is `length(Q_i)`).
2. Each vector obtained previously (1 vector per range in tS_i , all of them of length M) is turned into a single string (called "encoding block") by pasting its individual letters together.
3. All the encoding blocks (1 per range in tS_i) are pasted together into a single long string and separated by colons (":"). An additional colon is prepended to the long string and another one appended to it.
4. Finally, a special block containing the value of M is prepended to the long string. The final string is the encoding.

`levels(x)`: Equivalent to `levels(encoding(x))`.

`flippedQuery(x)`: Whether or not the top-level element in query used for computing the encoding was "flipped" before the encoding was computed. Note that this flipping generally affects the "left offset", "right offset", in addition to the encoding itself.

`Lencoding(x)`, `Rencoding(x)`: Extract the "left encodings" and "right encodings" of paired-end encodings.

Paired-end encodings are obtained by encoding paired-end overlaps i.e. overlaps between paired-end reads and transcripts (typically). The difference between a single-end encoding and a paired-end encoding is that all the blocks in the latter contain a "--" separator to mark the separation between the "left encoding" and the "right encoding".

See the "Overlap encodings" vignette located in this package for examples of paired-end encodings.

`njunc(x)`, `Lnjunc(x)`, `Rnjunc(x)`: Extract the number of junctions in each encoding by looking at their first block (aka special block). If an element x_i in `x` is a paired-end encoding, then `Lnjunc(x_i)`, `Rnjunc(x_i)`, and `njunc(x_i)`, return `njunc(Lencoding(x_i))`, `njunc(Rencoding(x_i))`, and `Lnjunc(x_i) + Rnjunc(x_i)`, respectively.

Coercing an OverlapEncodings object

In the following code snippets, `x` is an `OverlapEncodings` object.

`as.data.frame(x)`: Return `x` as a data frame with columns "Loffset", "Roffset" and "encoding".

Author(s)

H. Pages

See Also

- The "OverlapEncodings" vignette in this package.
- The [encodeOverlaps](#) function for computing "overlap encodings".
- The [compare](#) function in the **IRanges** package for the interpretation of the strings returned by encoding.
- The [GRangesList](#) class defined and documented in the **GenomicRanges** package.

Examples

```
example(encodeOverlaps) # to generate the ovenc object

length(ovenc)
Loffset(ovenc)
Roffset(ovenc)
encoding(ovenc)
levels(ovenc)
nlevels(ovenc)
flippedQuery(ovenc)
njunc(ovenc)

as.data.frame(ovenc)
njunc(levels(ovenc))
```

pileLettersAt	<i>Pile the letters of a set of aligned reads on top of a set of individual genomic positions</i>
---------------	---

Description

`pileLettersAt` extracts the letters/nucleotides of a set of reads that align to a set of individual genomic positions of interest. The extracted letters are returned as "piles of letters" (one per genomic position of interest) stored in an [XStringSet](#) (typically [DNAStringSet](#)) object.

Usage

```
pileLettersAt(x, seqnames, pos, cigar, at)
```

Arguments

x	An XStringSet (typically DNAStrngSet) object containing N <i>unaligned</i> read sequences (a.k.a. the query sequences) reported with respect to the + strand.
seqnames	A factor- Rle <i>parallel</i> to x. For each i, seqnames[i] must be the name of the reference sequence of the i-th alignment.
pos	An integer vector <i>parallel</i> to x. For each i, pos[i] must be the 1-based position on the reference sequence of the first aligned letter in x[[i]].
cigar	A character vector <i>parallel</i> to x. Contains the extended CIGAR strings of the alignments.
at	A GRanges object containing the individual genomic positions of interest. seqlevels(at) must be identical to levels(seqnames).

Details

x, seqnames, pos, cigar must be 4 *parallel* vectors describing N aligned reads.

Value

An **XStringSet** (typically **DNAStrngSet**) object *parallel* to at (i.e. with 1 string per individual genomic position).

Author(s)

H. Pages

See Also

- **DNAStrngSet** objects in the **Biostrings** package.
- **GRanges** objects in the **GenomicRanges** package.
- The **stackStringsFromBam** function for stacking the read sequences (or their quality strings) stored in a BAM file on a region of interest.
- **GAlignments** objects.
- **cigar-utils** for the CIGAR utility functions used internally by pileLettersAt.
- The SAMtools mpileup command available at <http://samtools.sourceforge.net/> as part of the SAMtools project.

Examples

```
## Input

## - A BAM file:
bamfile <- BamFile(system.file("extdata", "ex1.bam", package="Rsamtools"))
seqinfo(bamfile) # to see the seqlevels and seqlengths
stackStringsFromBam(bamfile, param="seq1:1-21") # a quick look at
                                                # the reads

## - A GRanges object containing Individual Genomic Positions Of
```



```

## Interest:
my_IGPOI <- GRanges(Rle(c("seq1", "seq2"), c(7, 2)),
                   IRanges(c(1:5, 21, 1575, 1513:1514), width=1))

## Some preliminary message on my_IGPOI

seqinfo(my_IGPOI) <- merge(seqinfo(my_IGPOI), seqinfo(bamfile))
seqlevels(my_IGPOI) <- seqlevelsInUse(my_IGPOI)

## Load the BAM file in a GAlignments object. We load only the reads
## aligned to the sequences in seqlevels(my_IGPOI) and we filter out
## reads not passing quality controls (flag bit 0x200) and PCR or
## optical duplicates (flag bit 0x400). See ?ScanBamParam and the SAM
## Spec for more information.

which <- as(seqinfo(my_IGPOI), "GRanges")
flag <- scanBamFlag(isNotPassingQualityControls=FALSE,
                   isDuplicate=FALSE)
what <- c("seq", "qual")
param <- ScanBamParam(flag=flag, what=c("seq", "qual"), which=which)
gal <- readGAlignmentsFromBam(bamfile, param=param)
seqlevels(gal) <- seqlevels(my_IGPOI)

## Extract the read sequences (a.k.a. query sequences) and quality
## strings. Both are reported with respect to the + strand.

qseq <- mcols(gal)$seq
qual <- mcols(gal)$qual

nucl_piles <- pileLettersAt(qseq, seqnames(gal), start(gal), cigar(gal),
                           my_IGPOI)
qual_piles <- pileLettersAt(qual, seqnames(gal), start(gal), cigar(gal),
                           my_IGPOI)
mcols(my_IGPOI)$nucl_piles <- nucl_piles
mcols(my_IGPOI)$qual_piles <- qual_piles
my_IGPOI

## Finally, to summarize A/C/G/T frequencies at each position:
alphabetFrequency(nucl_piles, baseOnly=TRUE)

```

readGAlignments

Reading genomic alignments from a file

Description

Read genomic alignments from a file (typically a BAM file) into a [GAlignments](#), [GAlignmentPairs](#), [GAlignmentsList](#), or [GappedReads](#) object.

Usage

```

## Front-ends
readGAlignments(file, format="BAM", use.names=FALSE, ...)
readGAlignmentPairs(file, format="BAM", use.names=FALSE, ...)
readGAlignmentsList(file, format="BAM", use.names=FALSE, ...)
readGappedReads(file, format="BAM", use.names=FALSE, ...)

## BAM specific back-ends
readGAlignmentsFromBam(file, index=file, ..., use.names=FALSE,
                       param=NULL, with.which_label=FALSE)

readGAlignmentPairsFromBam(file, index=file, use.names=FALSE,
                           param=NULL, with.which_label=FALSE)

readGAlignmentsListFromBam(file, index=file, ..., use.names=FALSE,
                            param=ScanBamParam(), with.which_label=FALSE)

readGappedReadsFromBam(file, index=file, use.names=FALSE,
                       param=NULL, with.which_label=FALSE)

```

Arguments

file	The path to the file to read or a BamFile object. Can also be a BamViews object for <code>readGAlignmentsFromBam</code> .
format	Only "BAM" (the default) is supported for now.
use.names	Use the query template names (QNAME field) as the names of the returned object? If not (the default), then the returned object has no names.
...	Arguments passed to other methods.
index	The path to the index file of the BAM file to read. Must be given <i>without</i> the '.bai' extension. See scanBam in the Rsamtools packages for more information.
param	NULL or a ScanBamParam object. Like for scanBam , this influences what fields and which records are imported. However, note that the fields specified thru this ScanBamParam object will be loaded <i>in addition</i> to any field required for generating the returned object (GAlignments , GAlignmentPairs , or GappedReads object), but only the fields requested by the user will actually be kept as meta-data columns of the object. By default (i.e. <code>param=NULL</code> or <code>param=ScanBamParam()</code>), no additional field is loaded. The flag used is <code>scanBamFlag(isUnmappedQuery=FALSE)</code> for <code>readGAlignmentsFromBam</code> , <code>readGappedReadsFromBam</code> and <code>readGAlignmentsListFromBam</code> (i.e. only records corresponding to mapped reads are loaded), and <code>scanBamFlag(isUnmappedQuery=FALSE, isPaired=TRUE)</code> for <code>readGAlignmentPairsFromBam</code> (i.e. only records corresponding to paired-end reads with both ends mapped are loaded).
with.which_label	TRUE or FALSE (the default). If TRUE and if <code>param</code> has a <code>which</code> component, a "which_label" metadata column is added to the returned GAlignments or GappedReads object, or to the <code>first</code> and <code>last</code> components of the returned

[GAlignmentPairs](#) object. In the case of `readGAlignmentsListFromBam`, it's added as an *inner* metadata column, that is, the metadata column is placed on the [GAlignments](#) object obtained by unlisting the returned [GAlignmentsList](#) object.

The purpose of this metadata column is to unambiguously identify the range in which where each element in the returned object originates from. The labels used to identify the ranges are normally of the form "seq1:12250-246500", that is, they're the same as the names found on the outer list that `scanBam` would return if called with the same `param` argument. If some ranges are duplicated, then the labels are made unique by appending a unique suffix to all of them. The "which_label" metadata column is represented as a factor-[Rle](#).

Details

See [?GAlignments](#) for a description of [GAlignments](#) objects.

See [?GappedReads](#) for a description of [GappedReads](#) objects.

Front-ends: `readGAlignments` reads a file containing aligned reads as a [GAlignments](#) object. `readGAlignmentPairs` reads a file containing aligned paired-end reads as a [GAlignmentPairs](#) object.

`readGAlignmentsList` reads a file containing aligned reads as a [GAlignmentsList](#) object.

`readGappedReads` reads a file containing aligned reads as a [GappedReads](#) object.

By default (i.e. `use.names=FALSE`), the resulting object has no names. If `use.names` is `TRUE`, then the names are constructed from the query template names (QNAME field in a SAM/BAM file). Note that the 2 records in a pair (when using `readGAlignmentPairs` or the records in a group (when using `readGAlignmentsList`) have the same QNAME.

These functions are just front-ends that delegate to a format-specific back-end function depending on the supplied format argument. The `use.names` argument and any extra argument are passed to the back-end function. Only the BAM format is supported for now via the `read*FromBam` back-end functions.

BAM specific back-ends: When file is [BamViews](#) object `readGAlignmentsFromBam` visits each path in `bamPaths(file)`, returning the result of `readGAlignmentsFromBam` applied to the specified path. When `index` is missing, it is set equal to `bamIndicies(file)`. Only reads in `bamRanges(file)` are returned (if `param` is supplied, `bamRanges(file)` takes precedence over `bamWhich(param)`). The return value is a [SimpleList](#) object, with elements of the list corresponding to each path. `bamSamples(file)` is available as metadata columns (accessed with `mcols`) of the returned [SimpleList](#) object.

`readGAlignmentPairsFromBam` proceeds in 2 steps:

1. Load the BAM file into a [GAlignmentsList](#) object with `readGAlignmentsListFromBam` (see below);
2. Turn this [GAlignmentsList](#) object into a [GAlignmentPairs](#) object. Only list elements marked with `mate_status` "mated" go into the returned [GAlignmentPairs](#) object.

See [?GAlignmentPairs](#) for a description of [GAlignmentPairs](#) objects.

`readGAlignmentsListFromBam` pairs records into mates according to the pairing criteria described below. The 1st mate will always be 1st in the [GAlignmentsList](#) list elements that have `mate_status` set to "mated", and the 2nd mate will always be 2nd.

A `GAlignmentsList` is returned with a ‘mate_status’ metadata column on the outer list elements. `mate_status` is a factor with 3 levels indicating mate status, ‘mated’, ‘ambiguous’ or unmated.

Mate status:

- mated: primary or non-primary pairs
- ambiguous: multiple segments matching to the same location (indistinguishable)
- unmated: mate does not exist or is unmapped

When the ‘file’ argument is a `BamFile`, ‘asMates=TRUE’ must be set, otherwise the data are treated as single-end reads. See the ‘asMates’ section of [?BamFile](#) for details.

Flags, tags and ranges may be specified in the `ScanBamParam` for fine tuning of results.

See [?GAlignmentsList-class](#) for a description of `GAlignmentsList` objects.

Pairing criteria: This section describes the pairing criteria used by `readGAlignmentsListFromBam` and `readGAlignmentPairsFromBam`.

- First, only records with flag bit 0x1 (multiple segments) set to 1, flag bit 0x4 (segment unmapped) set to 0, and flag bit 0x8 (next segment in the template unmapped) set to 0, are candidates for pairing (see the SAM Spec for a description of flag bits and fields). Records that correspond to single-end reads, or records that correspond to paired-end reads where one or both ends are unmapped, will remain unmated.
- Then the following fields and flag bits are considered:
 - (A) QNAME
 - (B) RNAME, RNEXT
 - (C) POS, PNEXT
 - (D) Flag bits 0x10 (segment aligned to minus strand) and 0x20 (next segment aligned to minus strand)
 - (E) Flag bits 0x40 (first segment in template) and 0x80 (last segment in template)
 - (F) Flag bit 0x2 (proper pair)
 - (G) Flag bit 0x100 (secondary alignment)

2 records `rec1` and `rec2` are considered mates iff all the following conditions are satisfied:

- (A) `QNAME(rec1) == QNAME(rec2)`
- (B) `RNEXT(rec1) == RNAME(rec2)` and `RNEXT(rec2) == RNAME(rec1)`
- (C) `PNEXT(rec1) == POS(rec2)` and `PNEXT(rec2) == POS(rec1)`
- (D) Flag bit 0x20 of `rec1` == Flag bit 0x10 of `rec2` and Flag bit 0x20 of `rec2` == Flag bit 0x10 of `rec1`
- (E) `rec1` corresponds to the first segment in the template and `rec2` corresponds to the last segment in the template, OR, `rec2` corresponds to the first segment in the template and `rec1` corresponds to the last segment in the template
- (F) `rec1` and `rec2` have same flag bit 0x2
- (G) `rec1` and `rec2` have same flag bit 0x100

Note that this is actually the pairing criteria used by `scanBam` (when the `BamFile` passed to it has the `asMates` toggle set to TRUE), which `readGAlignmentsListFromBam` and `readGAlignmentPairsFromBam` call behind the scene. It is also the pairing criteria used by `findMateAlignment`.

Value

A [GAlignments](#) object for readGAlignmentsFromBam.

A [GAlignmentPairs](#) object for readGAlignmentPairsFromBam. Note that a BAM (or SAM) file can in theory contain a mix of single-end and paired-end reads, but in practise it seems that single-end and paired-end are not mixed. In other words, the value of flag bit 0x1 (isPaired) is the same for all the records in a file. So if readGAlignmentPairsFromBam returns a [GAlignmentPairs](#) object of length zero, this almost certainly means that the BAM (or SAM) file contains alignments for single-end reads (although it could also mean that the user-supplied [ScanBamParam](#) is filtering out everything, or that the file is empty, or that all the records in the file correspond to unmapped reads).

A [GAlignmentsList](#) object for readGAlignmentsListFromBam. When the list contains paired-end reads a metadata data column of mate_status is added to the object. See details in the ‘Bam specific back-ends’ section on this man page.

A [GappedReads](#) object for readGappedReadsFromBam.

Note

BAM records corresponding to unmapped reads are always ignored.

Starting with Rsamtools 1.7.1 (BioC 2.10), PCR or optical duplicates are loaded by default (use scanBamFlag(isDuplicate=FALSE) to drop them).

Author(s)

H. Pages <hpages@fhcrc.org> and Valerie Obenchain <vobencha@fhcrc.org>

See Also

- [scanBam](#) and [ScanBamParam](#) in the **Rsamtools** package.
- [GAlignments](#), [GAlignmentPairs](#), [GAlignmentsList](#), and [GappedReads](#) objects.
- [findMateAlignment](#).

Examples

```
## -----
## A. readGAlignmentsFromBam()
## -----

## Simple use:
bamfile <- system.file("extdata", "ex1.bam", package="Rsamtools",
                      mustWork=TRUE)
gal1 <- readGAlignmentsFromBam(bamfile)
gal1
names(gal1)

## Using the use.names arg:
gal2 <- readGAlignmentsFromBam(bamfile, use.names=TRUE)
gal2
head(names(gal2))
```

```

## Using the param arg to drop PCR or optical duplicates as well as
## secondary alignments, and to load additional BAM fields:
param <- ScanBamParam(flag=scanBamFlag(isDuplicate=FALSE,
                                     isNotPrimaryRead=FALSE),
                    what=c("qual", "flag"))
gal3 <- readGAlignmentsFromBam(bamfile, param=param)
gal3
mcols(gal3)

## Using the param arg to load reads from particular regions.
## Note that if we werent providing a what argument here, all the
## BAM fields would be loaded:
which <- RangesList(seq1=IRanges(1000, 2000),
                   seq2=IRanges(c(100, 1000), c(1000, 2000)))
param <- ScanBamParam(which=which)
gal4 <- readGAlignmentsFromBam(bamfile, param=param)
gal4

## Note that a given record is loaded one time for each region it
## belongs to (this is a scanBam() feature, readGAlignmentsFromBam()
## is based on scanBam()):
which <- IRangesList(seq2=IRanges(c(1563, 1567), width=1))
param <- ScanBamParam(which=which)
gal5 <- readGAlignmentsFromBam(bamfile, param=param)
gal5

## Use with.which_label=TRUE to identify the range in which
## where each element in gal5 originates from.
gal5 <- readGAlignmentsFromBam(bamfile, param=param,
                             with.which_label=TRUE)
gal5

## Using the param arg to load tags. Except for MF and Aq, the tags
## specified below are predefined tags (see the SAM Spec for the list
## of predefined tags and their meaning).
param <- ScanBamParam(tag=c("MF", "Aq", "NM", "UQ", "H0", "H1"),
                    what="isize")
gal6 <- readGAlignmentsFromBam(bamfile, param=param)
mcols(gal6) # "tag" cols always after "what" cols

## With a BamViews object:
fls <- system.file("extdata", "ex1.bam", package="Rsamtools",
                  mustWork=TRUE)
bv <- BamViews(fls,
              bamSamples=DataFrame(info="test", row.names="ex1"),
              auto.range=TRUE)
aln <- readGAlignmentsFromBam(bv)
aln
aln[[1]]
aln[colnames(bv)]
mcols(aln)

## -----

```

```

## B. readGAlignmentPairsFromBam()
## -----
galp1 <- readGAlignmentPairsFromBam(bamfile)
head(galp1)
names(galp1)
## Using the param arg to drop PCR or optical duplicates as well as
## secondary alignments (dropping secondary alignments can help make the
## pairing algorithm run significantly faster, see ?findMateAlignment):
param <- ScanBamParam(flag=scanBamFlag(isDuplicate=FALSE,
                                     isNotPrimaryRead=FALSE))
galp2 <- readGAlignmentPairsFromBam(bamfile, use.names=TRUE, param=param)
galp2
head(galp2)
head(names(galp2))

## -----
## C. readGAlignmentsListFromBam()
## -----
library(pasillaBamSubset)

## file as character.
fl <- untreated3_chr4()
galist1 <- readGAlignmentsListFromBam(fl)
galist1[1:3]
length(galist1)
table(elementLengths(galist1))

## When file is a BamFile, asMates must be TRUE. If FALSE,
## the data are treated as single-end and each list element of the
## GAlignmentsList will be of length 1. For single-end data
## use readGAlignments().
bf <- BamFile(fl, yieldSize=3, asMates=TRUE)
readGAlignmentsList(bf)

## Use a param to fine tune the results.
param <- ScanBamParam(flag=scanBamFlag(isProperPair=TRUE))
galist2 <- readGAlignmentsListFromBam(fl, param=param)
length(galist2)

## -----
## D. readGappedReadsFromBam()
## -----
greads1 <- readGappedReadsFromBam(bamfile)
greads1
names(greads1)
qseq(greads1)
greads2 <- readGappedReadsFromBam(bamfile, use.names=TRUE)
head(greads2)
head(names(greads2))

```

Description

sequenceLayer can lay strings that belong to a given space (e.g. the "query" space) alongside another space (e.g. the "reference" space) by removing/injecting substrings from/into them, using the supplied CIGARs.

Its primary use case is to lay the read sequences stored in a BAM file (which are considered to belong to the "query" space) alongside the "reference" space. It can also be used to remove the parts of the read sequences that correspond to soft-clipping. More generally it can lay strings that belong to any supported space alongside any other supported space. See the Details section below for the list of supported spaces.

Usage

```
sequenceLayer(x, cigar, from="query", to="reference",
             D.letter="-", N.letter=".",
             I.letter="-", S.letter="+", H.letter="+")
```

Arguments

x	An XStringSet object containing strings that belong to a given space.
cigar	A character vector or factor of the same length as x containing the extended CIGAR strings (one per element in x).
from, to	A single string specifying one of the 8 supported spaces listed in the Details section below. from must be the current space (i.e. the space the strings in x belong to) and to is the space alongside which to lay the strings in x.
D.letter, N.letter, I.letter, S.letter, H.letter	A single letter used as a filler for injections. More on this in the Details section below.

Details

The 8 supported spaces are: "reference", "reference-N-regions-removed", "query", "query-before-hard-clipping", "query-after-soft-clipping", "pairwise", "pairwise-N-regions-removed", and "pairwise-dense".

Each space can be characterized by the extended CIGAR operations that are *visible* in it. A CIGAR operation is said to be *visible* in a given space if it "runs along it", that is, if it's associated with a block of contiguous positions in that space (the size of the block being the length of the operation). For example, the M/=X operations are *visible* in all spaces, the D/N operations are *visible* in the "reference" space but not in the "query" space, the S operation is *visible* in the "query" space but not in the "reference" or in the "query-after-soft-clipping" space, etc...

Here are the extended CIGAR operations that are *visible* in each space:

1. reference: M, D, N, =, X
2. reference-N-regions-removed: M, D, =, X
3. query: M, I, S, =, X
4. query-before-hard-clipping: M, I, S, H, =, X
5. query-after-soft-clipping: M, I, =, X
6. pairwise: M, I, D, N, =, X

7. pairwise-N-regions-removed: M, I, D, =, X
8. pairwise-dense: M, =, X

sequenceLayer lays a string that belongs to one space alongside another by (1) removing the substrings associated with operations that are not *visible* anymore in the new space, and (2) injecting substrings associated with operations that become *visible* in the new space. Each injected substring has the length of the operation associated with it, and its content is controlled via the corresponding `*.letter` argument.

For example, when going from the "query" space to the "reference" space (the default), the I- and S-substrings (i.e. the substrings associated with I/S operations) are removed, and substrings associated with D/N operations are injected. More precisely, the D-substrings are filled with the letter specified in `D.letter`, and the N-substrings with the letter specified in `N.letter`. The other `*.letter` arguments are ignored in that case.

Value

An `XStringSet` object of the same class and length as `x`.

Author(s)

H. Pages

See Also

- The `stackStringsFromBam` function for stacking the read sequences (or their quality strings) stored in a BAM file on a region of interest.
- The `readGAlignmentsFromBam` function for loading read sequences from a BAM file (via a `GAlignments` object).
- The `extractAt` and `replaceAt` functions in the **Biostrings** package for extracting/replacing arbitrary substrings from/in a string or set of strings.
- `cigar-utils` for the CIGAR utility functions used internally by `sequenceLayer`.

Examples

```
## -----
## A. FROM "query" TO "reference" SPACE
## -----

## Load read sequences from a BAM file (they will be returned in a
## GAlignments object):
bamfile <- system.file("extdata", "ex1.bam", package="Rsamtools")
param <- ScanBamParam(what="seq")
gal <- readGAlignmentsFromBam(bamfile, param=param)
qseq <- mcols(gal)$seq # the read sequences (aka query sequences)

## Lay the query sequences alongside the reference space. This will
## remove the substrings associated with insertions to the reference
## (I operations) and soft clipping (S operations), and will inject new
## substrings (filled with "-") where deletions from the reference (D
## operations) and skipped regions from the reference (N operations)
```

```

## occurred during the alignment process:
qseq_on_ref <- sequenceLayer(qseq, cigar(gal))

## A typical use case for doing the above is to compute 1 consensus
## sequence per chromosome. The code below shows how this can be done
## in 2 extra steps.

## Step 1: Compute one consensus matrix per chromosome.
qseq_on_ref_by_chrom <- splitAsList(qseq_on_ref, seqnames(gal))
pos_by_chrom <- splitAsList(start(gal), seqnames(gal))

cm_by_chrom <- lapply(names(pos_by_chrom),
  function(seqname)
    consensusMatrix(qseq_on_ref_by_chrom[[seqname]],
                    as.prob=TRUE,
                    shift=pos_by_chrom[[seqname]]-1,
                    width=seqlengths(gal)[[seqname]]))
names(cm_by_chrom) <- names(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.

## Step 2: 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) {
    ## Because consensusString() doesn't like consensus matrices
    ## with columns that contain only zeroes (and you will have
    ## columns like that for chromosome positions that don't
    ## receive any coverage), we need to "fix" cm first.
    idx <- colSums(cm) == 0
    cm["+", idx] <- 1
    DNASTring(consensusString(cm, ambiguityMap="N"))
  })

## consensusString() provides some flexibility to let you extract
## the consensus in different ways. See ?consensusString in the
## Biostrings package for the details.

## Finally, note that the read quality strings can also be used as
## input for sequenceLayer():
param <- ScanBamParam(what="qual")
gal <- readGAlignmentsFromBam(bamfile, param=param)
qual <- mcols(gal)$qual # the read quality strings
qual_on_ref <- sequenceLayer(qual, cigar(gal))
## Note that since the "-" letter is a valid quality code, there is
## no way to distinguish it from the "-" letters inserted by
## sequenceLayer().

## -----
## B. FROM "query" TO "query-after-soft-clipping" SPACE

```

```

## -----

## Going from "query" to "query-after-soft-clipping" simply removes
## the substrings associated with soft clipping (S operations):
qseq <- DNASTringSet(c("AAAGTCGAA", "TTACGATTAN", "GGATAATTTT"))
cigar <- c("3H10M", "2S7M1S2H", "2M1I1M3D2M4S")
clipped_qseq <- sequenceLayer(qseq, cigar,
                             from="query", to="query-after-soft-clipping")

sequenceLayer(clipped_qseq, cigar,
              from="query-after-soft-clipping", to="query")

sequenceLayer(clipped_qseq, cigar,
              from="query-after-soft-clipping", to="query",
              S.letter="-")

## -----
## C. BRING QUERY AND REFERENCE SEQUENCES TO THE "pairwise" or
## "pairwise-dense" SPACE
## -----

## Load read sequences from a BAM file:
library(RNaseqData.HNRNPC.bam.chr14)
bamfile <- RNaseqData.HNRNPC.bam.chr14_BAMFILES[1]
param <- ScanBamParam(what="seq",
                     which=GRanges("chr14", IRanges(1, 25000000)))
gal <- readGAlignmentsFromBam(bamfile, param=param)
qseq <- mcols(gal)$seq # the read sequences (aka query sequences)

## Load the corresponding reference sequences from the appropriate
## BSgenome package (the reads in RNaseqData.HNRNPC.bam.chr14 were
## aligned to hg19):
library(BSgenome.Hsapiens.UCSC.hg19)
rseq <- getSeq(Hsapiens, as(gal, "GRanges")) # the reference sequences

## Bring qseq and rseq to the "pairwise" space.
## For qseq, this will remove the substrings associated with soft
## clipping (S operations) and inject substrings (filled with "-")
## associated with deletions from the reference (D operations) and
## skipped regions from the reference (N operations). For rseq, this
## will inject substrings (filled with "-") associated with insertions
## to the reference (I operations).
qseq2 <- sequenceLayer(qseq, cigar(gal),
                      from="query", to="pairwise")
rseq2 <- sequenceLayer(rseq, cigar(gal),
                      from="reference", to="pairwise")

## Sanity check: qseq2 and rseq2 should have the same shape.
stopifnot(identical(elementLengths(qseq2), elementLengths(rseq2)))

## A closer look at reads with insertions and deletions:
cigar_op_table <- cigarOpTable(cigar(gal))
head(cigar_op_table)

```



```

function(cigar) cigarWidthAlongQuerySpace(cigar),
function(cigar) cigarWidthAlongQuerySpace(cigar,
                                           before.hard.clipping=TRUE),
function(cigar) cigarWidthAlongQuerySpace(cigar,
                                           after.soft.clipping=TRUE),
function(cigar) cigarWidthAlongPairwiseSpace(cigar),
function(cigar) cigarWidthAlongPairwiseSpace(cigar,
                                              N.regions.removed=TRUE),
function(cigar) cigarWidthAlongPairwiseSpace(cigar, dense=TRUE)
)

cigar <- c("3H2S4M1D2M2I1M5N3M6H", "5M1I3M2D4M2S")

seq <- list(
  BStringSet(c(A="AAAA-BBC....DDD", B="AAAAABBB--CCCC")),
  BStringSet(c(A="AAAA-BBCDDD", B="AAAAABBB--CCCC")),
  BStringSet(c(A="++AAAABBiCDDD", B="AAAAAiBBBCCCC++")),
  BStringSet(c(A="+++++AAAABBiCDDD+++++", B="AAAAAiBBBCCCC++")),
  BStringSet(c(A="AAAABBiCDDD", B="AAAAAiBBBCCCC")),
  BStringSet(c(A="AAAA-BBiC....DDD", B="AAAAAiBBB--CCCC")),
  BStringSet(c(A="AAAA-BBiCDDD", B="AAAAAiBBB--CCCC")),
  BStringSet(c(A="AAAABBCDDD", B="AAAAABBBCCCC"))
)

stopifnot(all(sapply(1:8,
                    function(i) identical(width(seq[[i]]), cigarWidth[[i]](cigar))
                    )))

sequenceLayer2 <- function(x, cigar, from, to)
  sequenceLayer(x, cigar, from=from, to=to, I.letter="i")

identical_XStringSet <- function(target, current)
{
  ok1 <- identical(class(target), class(current))
  ok2 <- identical(names(target), names(current))
  ok3 <- all(target == current)
  ok1 && ok2 && ok3
}

res <- sapply(1:8, function(i) {
  sapply(1:8, function(j) {
    target <- seq[[j]]
    current <- sequenceLayer2(seq[[i]], cigar,
                             from=SPACES[i], to=SPACES[j])
    identical_XStringSet(target, current)
  })
})
stopifnot(all(res))

```

Description

Performs set operations on [GAlignments](#) objects.

Usage

```
## S4 method for signature GAlignments,GRanges
pintersect(x, y, ...)
## S4 method for signature GRanges,GAlignments
pintersect(x, y, ...)
```

Arguments

`x, y` A [GAlignments](#) object and a [GRanges](#) object. They must have the same length.
`...` Further arguments to be passed to or from other methods.

Value

A [GAlignments](#) object *parallel* to (i.e. same length as) `x` and `y`.

See Also

- The [GAlignments](#) class.
- The [setops-methods](#) man page in the **GenomicRanges** package.

Examples

```
## Parallel intersection of a GAlignments and a GRanges object:
bamfile <- system.file("extdata", "ex1.bam", package="Rsamtools")
gal <- readGAlignments(bamfile)
pintersect(gal, shift(as(gal, "GRanges"), 6L))
```

`stackStringsFromBam` *Stack the read sequences stored in a BAM file on a region of interest*

Description

`stackStringsFromBam` stacks the read sequences (or their quality strings) stored in a BAM file over a user-specified region.

`alphabetFrequencyFromBam` computes the alphabet frequency of the reads over a user-specified region.

Both functions take into account the CIGAR of each read to "lay" the read sequence (or its quality string) alongside the reference space. This step ensures that each nucleotide in a read is associated with the correct position on the reference sequence.

Usage

```
stackStringsFromBam(file, index=file, param,
                    what="seq", use.names=FALSE,
                    D.letter="-", N.letter=".",
                    Lpadding.letter="+", Rpadding.letter="+")

alphabetFrequencyFromBam(file, index=file, param, what="seq", ...)
```

Arguments

file, index	The path to the BAM file to read, and to the index file of the BAM file to read, respectively. The latter is given <i>without</i> the '.bai' extension. See scanBam for more information.
param	A ScanBamParam object containing exactly 1 genomic region (i.e. <code>unlist(bamWhich(param))</code> must have length 1). Alternatively, param can be a GRanges or RangesList object containing exactly 1 genomic region (the strand will be ignored in case of a GRanges object), or a character string specifying a single genomic region (in the "chr14:5201-5300" format).
what	A single string. Either "seq" or "qual". If "seq" (the default), the read sequences will be stacked. If "qual", the read quality strings will be stacked.
use.names	Use the query template names (QNAME field) as the names of the returned object? If not (the default), then the returned object has no names.
D.letter, N.letter	A single letter used as a filler for injections. The 2 arguments are passed down to the sequenceLayer function. See <code>?sequenceLayer</code> for more details.
Lpadding.letter, Rpadding.letter	A single letter to use for padding the sequences on the left, and another one to use for padding on the right. The 2 arguments are passed down to the stackStrings function defined in the Biostrings package. See <code>?stackStrings</code> in the Biostrings package for more details.
...	Further arguments to be passed to alphabetFrequency .

Details

stackStringsFromBam performs the 3 following steps:

1. Load the read sequences (or their quality strings) from the BAM file. Only the read sequences that overlap with the specified region are loaded. This is done with the [readGAlignmentsFromBam](#) function. Note that if the file contains paired-end reads, the pairing is ignored.
2. Lay the sequences alongside the reference space, using their CIGARs. This is done with the [sequenceLayer](#) function.
3. Stack them on the specified region. This is done with the [stackStrings](#) function defined in the **Biostrings** package.

alphabetFrequencyFromBam also performs steps 1. and 2. but, instead of stacking the sequences at step 3., it computes the nucleotide frequencies for each genomic position in the specified region.

Value

For `stackStringsFromBam`: A rectangular (i.e. constant-width) `DNAStringSet` object (if what is "seq") or `BStringSet` object (if what is "qual").

For `alphabetFrequencyFromBam`: By default a matrix like one returned by `alphabetFrequency`. The matrix has 1 row per nucleotide position in the specified region.

Note

TWO IMPORTANT CAVEATS ABOUT `stackStringsFromBam`:

Specifying a big genomic region, say ≥ 100000 bp, can require a lot of memory (especially with high coverage reads) and is not recommended. See the `pileLettersAt` function for piling the read letters on top of a set of individual genomic positions, which is more flexible and more memory efficient.

Paired-end reads are treated as single-end reads (i.e. they're not paired).

Author(s)

H. Pages

See Also

- The `pileLettersAt` function for piling the letters of a set of aligned reads on top of a set of individual genomic positions.
- The `readGAlignmentsFromBam` function for loading read sequences (or their quality strings) from a BAM file (via a `GAlignments` object).
- The `sequenceLayer` function for laying read sequences alongside the reference space, using their CIGARs.
- The `stackStrings` function in the **Biostrings** package for stacking an arbitrary `XStringSet` object.
- The `alphabetFrequency` function in the **Biostrings** package.
- The SAMtools mpileup command available at <http://samtools.sourceforge.net/> as part of the SAMtools project.

Examples

```
## -----
## A. EXAMPLE WITH TOY DATA
## -----

bamfile1 <- BamFile(system.file("extdata", "ex1.bam", package="Rsamtools"))

region1 <- GRanges("seq1", IRanges(1, 60)) # region of interest

## Stack the read sequences:
stackStringsFromBam(bamfile1, param=region1)

## Compute the "consensus matrix" (1 column per nucleotide position
```



```

## in the region of interest:
af <- alphabetFrequencyFromBam(bamfile1, param=region1, baseOnly=TRUE)
cm1a <- t(af[ , DNA_BASES])
cm1a

## Stack their quality strings:
stackStringsFromBam(bamfile1, param=region1, what="qual")

## Control the number of reads to display:
options(showHeadLines=18)
options(showTailLines=6)
stackStringsFromBam(bamfile1, param=GRanges("seq1", IRanges(61, 120)))

stacked_qseq <- stackStringsFromBam(bamfile1, param="seq2:1509-1519")
stacked_qseq # deletion in read 13
af <- alphabetFrequencyFromBam(bamfile1, param="seq2:1509-1519",
                               baseOnly=TRUE)
cm1b <- t(af[ , DNA_BASES]) # consensus matrix
cm1b

## Sanity check:
stopifnot(identical(consensusMatrix(stacked_qseq)[DNA_BASES, ], cm1b))

stackStringsFromBam(bamfile1, param="seq2:1509-1519", what="qual")

## -----
## B. EXAMPLE WITH REAL DATA
## -----

library(RNAseqData.HNRNPC.bam.chr14)
bamfile2 <- BamFile(RNAseqData.HNRNPC.bam.chr14_BAMFILES[1])

## Region of interest:
region2 <- GRanges("chr14", IRanges(19650095, 19650159))

readGAlignments(bamfile2, param=ScanBamParam(which=region2))

stackStringsFromBam(bamfile2, param=region2)

af <- alphabetFrequencyFromBam(bamfile2, param=region2, baseOnly=TRUE)
cm2 <- t(af[ , DNA_BASES]) # consensus matrix
cm2

## -----
## C. COMPUTE READ CONSENSUS SEQUENCE FOR REGION OF INTEREST
## -----

## Lets write our own little naive function to go from consensus matrix
## to consensus sequence. For each nucleotide position in the region of
## interest (i.e. each column in the matrix), we select the letter with
## highest frequency. We also use special letter "*" at positions where
## there is a tie, and special letter "." at positions where all the
## frequencies are 0 (a particular type of tie):

```

```

cm_to_cs <- function(cm)
{
  stopifnot(is.matrix(cm))
  nr <- nrow(cm)
  rnames <- rownames(cm)
  stopifnot(!is.null(rnames) && all(nchar(rnames) == 1L))
  selection <- apply(cm, 2,
    function(x) {
      i <- which.max(x)
      if (x[i] == 0L)
        return(nr + 1L)
      if (sum(x == x[i]) != 1L)
        return(nr + 2L)
      i
    })
  paste0(c(rnames, ".", "*")[selection], collapse="")
}

cm_to_cs(cm1a)
cm_to_cs(cm1b)
cm_to_cs(cm2)

## Note that the consensus sequences we obtain are relative to the
## plus strand of the reference sequence.

```

summarizeOverlaps-methods

Perform overlap queries between reads and genomic features

Description

summarizeOverlaps extends findOverlaps by providing options to resolve reads that overlap multiple features.

Usage

```

## S4 method for signature GRanges,GAlignments
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,
  preprocess.reads=NULL)

## S4 method for signature GRangesList,GAlignments
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,
  preprocess.reads=NULL)

## S4 method for signature GRanges,GRanges
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,

```

```

preprocess.reads=NULL)

## S4 method for signature GRangesList,GRanges
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,
  preprocess.reads=NULL)

## S4 method for signature GRanges,GAlignmentPairs
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,
  preprocess.reads=NULL)

## S4 method for signature GRangesList,GAlignmentPairs
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,
  preprocess.reads=NULL)

## mode funtions
Union(features, reads, ignore.strand=FALSE, inter.feature=TRUE)
IntersectionStrict(features, reads, ignore.strand=FALSE, inter.feature=TRUE)
IntersectionNotEmpty(features, reads, ignore.strand=FALSE, inter.feature=TRUE)

## S4 method for signature GRanges,BamFile
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,
  singleEnd=TRUE, fragments=FALSE, param=ScanBamParam(),
  preprocess.reads=NULL)

## S4 method for signature BamViews,missing
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ..., inter.feature=TRUE,
  singleEnd=TRUE, fragments=FALSE, param=ScanBamParam(),
  preprocess.reads=NULL)

```

Arguments

- | | |
|----------|---|
| features | <p>A GRanges or a GRangesList object of genomic regions of interest. When a GRanges is supplied, each row is considered a feature. When a GRangesList is supplied, each higher list-level is considered a feature. This distinction is important when defining overlaps.</p> <p>When features is a BamViews the reads argument is missing. Features are extracted from the <code>bamRanges</code> and the reads from <code>bamPaths</code>. Metadata from <code>bamPaths</code> and <code>bamSamples</code> are stored in the <code>colData</code> of the resulting SummarizedExperiment object. <code>bamExperiment</code> metadata are stored in the <code>exptData</code>.</p> |
| reads | <p>A GRanges, GRangesList, GAlignments, GAlignmentsList, GAlignmentPairs, BamViews or BamFileList object that represents the data to be counted by <code>summarizeOverlaps</code>.</p> <p><code>reads</code> is missing when a BamViews object is the only argument supplied to</p> |

	<p><code>summarizeOverlaps</code>. <code>reads</code> are the files specified in <code>bamPaths</code> of the <code>BamViews</code> object.</p>
<code>mode</code>	<p><code>mode</code> can be one of the pre-defined count methods such as "Union", "IntersectionStrict", or "IntersectionNotEmpty" or it can be a user supplied count function. For a custom count function, the input arguments must match those of the pre-defined options and the function must return a vector of counts the same length as the annotation ('features' argument). See examples for details.</p> <p>The pre-defined options are designed after the counting modes available in the HTSeq package by Simon Anders (see references).</p> <ul style="list-style-type: none"> • "Union" : (Default) Reads that overlap any portion of exactly one feature are counted. Reads that overlap multiple features are discarded. This is the most conservative of the 3 modes. • "IntersectionStrict" : A read must fall completely "within" the feature to be counted. If a read overlaps multiple features but falls "within" only one, the read is counted for that feature. If the read is "within" multiple features, the read is discarded. • "IntersectionNotEmpty" : A read must fall in a unique disjoint region of a feature to be counted. When a read overlaps multiple features, the features are partitioned into disjoint intervals. Regions that are shared between the features are discarded leaving only the unique disjoint regions. If the read overlaps one of these remaining regions, it is assigned to the feature the unique disjoint region came from. • user supplied function : A function can be supplied as the mode argument. It must (1) have arguments that correspond to features, reads, <code>ignore.strand</code> and <code>inter.feature</code> arguments (as in the defined mode functions) and (2) return a vector of counts the same length as features.
<code>ignore.strand</code>	A logical indicating if strand should be considered when matching.
<code>inter.feature</code>	<p>(Default TRUE) A logical indicating if the counting mode should be aware of overlapping features. When TRUE (default), reads mapping to multiple features are dropped (i.e., not counted). When FALSE, these reads are retained and a count is assigned to each feature they map to.</p> <p>There are 6 possible combinations of the mode and <code>inter.feature</code> arguments. When <code>inter.feature=FALSE</code> the behavior of modes 'Union' and 'IntersectionStrict' are essentially 'countOverlaps' with 'type=any' and 'type=within', respectively. 'IntersectionNotEmpty' does not reduce to a simple countOverlaps because common (shared) regions of the annotation are removed before counting.</p>
<code>preprocess.reads</code>	<p>A function applied to the reads before counting. The first argument should be reads and the return value should be an object compatible with the reads argument to the counting modes, Union, IntersectionStrict and IntersectionNotEmpty.</p> <p>The distinction between a user-defined 'mode' and user-defined 'preprocess.reads' function is that in the first case the user defines how to count; in the second case the reads are preprocessed before counting with a pre-defined mode. See examples.</p>

...	Additional arguments passed to functions or methods called from within summarizeOverlaps. For BAM file methods arguments may include singleEnd, fragments or param which apply to reading records from a file (see below). Providing count.mapped.reads=TRUE include additional passes through the BAM file to collect statistics similar to those from countBam.
singleEnd	(Default TRUE) A logical indicating if reads are single or paired-end. In Bioconductor > 2.12 it is not necessary to sort paired-end BAM files by qname. When counting with summarizeOverlaps, setting singleEnd=FALSE will trigger paired-end reading and counting. It is fine to also set asMates=TRUE in the BamFile but is not necessary when singleEnd=FALSE.
fragments	(Default FALSE) A logical value indicating if singletons, reads with unmapped pairs and other fragments should be included in counting. Applies to paired-end data only. When fragments=FALSE the readGAlignmentPairs function is used to read in the data. Only the read pairs mated by the algorithm are kept and counted. When fragments=TRUE singleEnd should be FALSE. readAlignmentsList is used to read in the data. Read pairs mated by the algorithm as well as all singletons, reads with unmapped pairs and other fragments are kept and counted. Because more records are kept, generally counts will be higher with fragments=TRUE. See ?readGAlignmentsListFromBam for the count algorithm details.
param	An optional ScanBamParam instance to further influence scanning, counting, or filtering. See ?BamFile for details of how records are returned when both yieldSize is specified in a BamFile and which is defined in a ScanBamParam.

Details

summarizeOverlaps offers counting modes to resolve reads that overlap multiple features. The mode argument defines a set of rules to resolve the read to a single feature such that each read is counted a maximum of once. New to GenomicRanges >= 1.13.9 is the inter.feature argument which allows reads to be counted for each feature they overlap. When inter.feature=TRUE the counting modes are aware of feature overlap and reads overlapping multiple features are dropped and not counted. When inter.feature=FALSE multiple feature overlap is ignored and reads are counted once for each feature they map to. This essentially reduces modes 'Union' and 'IntersectionStrict' to countOverlaps with type="any", and type="within", respectively. 'IntersectionNotEmpty' is not reduced to a derivative of countOverlaps because the shared regions are removed before counting.

The BamViews, BamFile and BamFileList methods summarize overlaps across one or several files. The latter uses bplapply; control parallel evaluation using the register interface in the BiocParallel package.

features : A 'feature' can be any portion of a genomic region such as a gene, transcript, exon etc. When the features argument is a GRanges the rows define the features. The result will be the same length as the GRanges. When features is a GRangesList the highest list-level defines the features and the result will be the same length as the GRangesList.

When inter.feature=TRUE, each count mode attempts to assign a read that overlaps multiple features to a single feature. If there are ranges that should be considered together (e.g., exons

by transcript or cds regions by gene) the [GRangesList](#) would be appropriate. If there is no grouping in the data then a [GRanges](#) would be appropriate.

paired-end reads : Paired-end reads are counted as a single hit if one or both parts of the pair are overlapped. Paired-end records can be counted in a [GAlignmentPairs](#) container or BAM file.

Counting pairs in BAM files:

- The `singleEnd` argument should be `FALSE`.
- When reads are supplied as a `BamFile` or `BamFileList`, the `asMates` argument to the `BamFile` should be `TRUE`.
- When `fragments` is `FALSE`, a `GAlignmentPairs` object is used in counting (pairs only).
- When `fragments` is `TRUE`, a `GAlignmentsList` object is used in counting (pairs, singletons, unmapped mates, etc.)

Value

A [SummarizedExperiment](#) object. The `assays` slot holds the counts, `rowData` holds the annotation from features.

When `reads` is a `BamFile` or `BamFileList` `colData` is an empty `DataFrame` with a single row named 'counts'. If `count.mapped.reads=TRUE`, `colData` holds the output of `countBam` in 3 columns named 'records' (total records), 'nucleotides' and 'mapped' (mapped records).

When `features` is a `BamViews` `colData` includes 2 columns named `bamSamples` and `bamIndices`.

In all other cases, `colData` has columns of 'object' (class of reads) and 'records' (length of reads).

Author(s)

Valerie Obenchain <vobencha@fhcrc.org>

References

HTSeq : <http://www-huber.embl.de/users/anders/HTSeq/doc/overview.html>

htseq-count : <http://www-huber.embl.de/users/anders/HTSeq/doc/count.html>

See Also

- The **DESeq**, **DEXSeq** and **edgeR** packages.
- The [GAlignments](#) and [GAlignmentPairs](#) classes.
- The [BamFileList](#) and [BamViews](#) classes in the **Rsamtools** package.
- The [readGAlignments](#) and [readGAlignmentPairs](#) functions.

Examples

```
reads <- GAlignments(
  names = c("a", "b", "c", "d", "e", "f", "g"),
  seqnames = Rle(c(rep("chr1", "chr2"), 3), "chr1"),
  pos = as.integer(c(1400, 2700, 3400, 7100, 4000, 3100, 5200)),
  cigar = c("500M", "100M", "300M", "500M", "300M",
            "50M200N50M", "50M150N50M"),
  strand = strand(rep("+", 7)))
```

```

gr <- GRanges(
  seqnames = c(rep("chr1", 7), rep("chr2", 4)), strand = "+",
  ranges = IRanges(c(1000, 3000, 3600, 4000, 4000, 5000, 5400,
    2000, 3000, 7000, 7500),
  width = c(500, 500, 300, 500, 900, 500, 500,
    900, 500, 600, 300),
  names=c("A", "B", "C1", "C2", "D1", "D2", "E", "F",
    "G", "H1", "H2"))
groups <- factor(c(1,2,3,3,4,4,5,6,7,8,8))
gr1 <- splitAsList(gr, groups)
names(gr1) <- LETTERS[seq_along(gr1)]

## -----
## Counting modes.
## -----

## First count with a GRanges as the features. Union is the
## most conservative counting mode followed by IntersectionStrict
## then IntersectionNotEmpty.
counts1 <-
  data.frame(union=assays(summarizeOverlaps(gr, reads))$counts,
    intStrict=assays(summarizeOverlaps(gr, reads,
      mode="IntersectionStrict"))$counts,
    intNotEmpty=assays(summarizeOverlaps(gr, reads,
      mode="IntersectionNotEmpty"))$counts)

colSums(counts1)

## Split the features into a GRangesList and count again.
counts2 <-
  data.frame(union=assays(summarizeOverlaps(gr1, reads))$counts,
    intStrict=assays(summarizeOverlaps(gr1, reads,
      mode="IntersectionStrict"))$counts,
    intNotEmpty=assays(summarizeOverlaps(gr1, reads,
      mode="IntersectionNotEmpty"))$counts)

colSums(counts2)

## The GRangesList (gr1 object) has 8 features whereas the GRanges
## (gr object) has 11. The affect on counting can be seen by looking
## at feature H with mode Union. In the GRanges this feature is
## represented by ranges H1 and H2,
gr[c("H1", "H2")]

## and by list element H in the GRangesList,
gr1["H"]

## Read "d" hits both H1 and H2. This is considered a multi-hit when
## using a GRanges (each range is a separate feature) so the read was
## dropped and not counted.
counts1[c("H1", "H2"), ]

## When using a GRangesList, each list element is considered a feature.

```

```

## The read hits multiple ranges within list element H but only one
## list element. This is not considered a multi-hit so the read is counted.
counts2["H", ]

## -----
## Counting multi-hit reads.
## -----

## The goal of the counting modes is to provide a set of rules that
## resolve reads hitting multiple features so each read is counted
## a maximum of once. However, sometimes it may be desirable to count
## a read for each feature it overlaps. This can be accomplished by
## setting inter.feature to FALSE.

## When inter.feature=FALSE, modes Union and IntersectionStrict
## essentially reduce to countOverlaps() with type="any" and
## type="within", respectively.

## When inter.feature=TRUE only features "A", "F" and "G" have counts.
se1 <- summarizeOverlaps(gr, reads, mode="Union", inter.feature=TRUE)
assays(se1)$counts

## When inter.feature=FALSE all 11 features have a count. There are
## 7 total reads so one or more reads were counted more than once.
se2 <- summarizeOverlaps(gr, reads, mode="Union", inter.feature=FALSE)
assays(se2)$counts

## -----
## Counting BAM files.
## -----

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

## (i) Single-end :

## Large files can be iterated over in chunks by setting a
## yieldSize on the BamFile.
bf_s <- BamFile(untreated1_chr4(), yieldSize=50000)
se_s <- summarizeOverlaps(exbygene, bf_s, singleEnd=TRUE)
table(assays(se_s)$counts > 0)

## When a character (file name) is provided as reads instead
## of a BamFile object summarizeOverlaps() will create a BamFile
## and set a reasonable default yieldSize.

## (ii) Paired-end :

## A paired-end file may contain singletons, reads with unmapped
## pairs or reads with more than two fragments. When fragments=FALSE
## only reads paired by the algorithm are included in the counting.
nofrag <- summarizeOverlaps(exbygene, untreated3_chr4(),

```



```

                                singleEnd=FALSE, fragments=FALSE)
table(assays(nofrag)$counts > 0)

## When fragments=TRUE all singletons, reads with unmapped pairs
## and other fragments will be included in the counting.
bf <- BamFile(untreated3_chr4(), asMates=TRUE)
frag <- summarizeOverlaps(exbygene, bf, singleEnd=FALSE, fragments=TRUE)
table(assays(frag)$counts > 0)

## As expected, using fragments=TRUE results in a larger number
## of total counts because singletons, unmapped pairs etc. are
## included in the counting.

## Total reads in the file:
countBam(untreated3_chr4())

## Reads counted with fragments=FALSE:
sum(assays(nofrag)$counts)

## Reads counted with fragments=TRUE:
sum(assays(frag)$counts)

## -----
## Count tables for DESeq or edgeR.
## -----

fls <- list.files(system.file("extdata", package="GenomicAlignments"),
                  recursive=TRUE, pattern="*bam$", full=TRUE)
names(fls) <- basename(fls)
bf <- BamFileList(fls, index=character(), yieldSize=1000)
genes <- GRanges(
  seqnames = c(rep("chr2L", 4), rep("chr2R", 5), rep("chr3L", 2)),
  ranges = IRanges(c(1000, 3000, 4000, 7000, 2000, 3000, 3600,
                    4000, 7500, 5000, 5400),
                  width=c(rep(500, 3), 600, 900, 500, 300, 900,
                           300, 500, 500)))
se <- summarizeOverlaps(genes, bf)

## When the reads are BAM files, the colData contains summary
## information from a call to countBam().
colData(se)

## Create count tables.
library(DESeq)
deseq <- newCountDataSet(assays(se)$counts, rownames(colData(se)))
library(edgeR)
edger <- DGEList(assays(se)$counts, group=rownames(colData(se)))

## -----
## Filter records by map quality before counting.
## (user-supplied mode function)
## -----

```

```

## The mode argument can take a custom count function whose
## arguments are the same as those in the current counting modes
## (i.e., Union, IntersectionNotEmpty, IntersectionStrict).
## In this example records are filtered by map quality before counting.

mapq_filter <- function(features, reads, ignore.strand, inter.feature) {
  require(GenomicAlignments) # needed for parallel evaluation
  Union(features, reads[mcols(reads)$mapq >= 20],
        ignore.strand=ignore.strand,
        inter.feature=inter.feature)
}

genes <- GRanges("seq1", IRanges(seq(1, 1500, by=200), width=100))
param <- ScanBamParam(what="mapq")
fl <- system.file("extdata", "ex1.bam", package="Rsamtools")
se <- summarizeOverlaps(genes, fl, mode=mapq_filter, param=param)
assays(se)$counts

## The count function can be completely custom (i.e., not use the
## pre-defined count functions at all). Requirements are that
## the input arguments match the pre-defined modes and the output
## is a vector of counts the same length as features.

my_count <- function(features, reads, ignore.strand, inter.feature) {
  ## perform filtering, or subsetting etc.
  require(GenomicAlignments) # needed for parallel evaluation
  countOverlaps(features, reads)
}

## -----
## Preprocessing reads before counting with a standard count mode.
## (user-supplied preprocess.reads function)
## -----

## The preprocess.reads argument takes a function that is
## applied to the reads before counting with a pre-defined mode.

ResizeReads <- function(reads, width=1, fix="start", ...) {
  reads <- as(reads, "GRanges")
  stopifnot(all(strand(reads) != "*"))
  resize(reads, width=width, fix=fix, ...)
}

## By default ResizeReads() counts reads that overlap on the 5 end:
summarizeOverlaps(gr1, reads, mode=Union, preprocess.reads=ResizeReads)

## Count reads that overlap on the 3 end by passing new values
## for width and fix:
summarizeOverlaps(gr1, reads, mode=Union, preprocess.reads=ResizeReads,
                  width=1, fix="end")

## -----
## summarizeOverlaps() with BamViews.

```

```
## -----  
  
## bamSamples and bamPaths metadata are included in the colData.  
## bamExperiment metadata is put into the exptData slot.  
fl <- system.file("extdata", "ex1.bam", package="Rsamtools", mustWork=TRUE)  
rngs <- GRanges(c("seq1", "seq2"), IRanges(1, c(1575, 1584)))  
samp <- DataFrame(info="test", row.names="ex1")  
view <- BamViews(fl, bamSamples=samp, bamRanges=rngs)  
se <- summarizeOverlaps(view, mode=Union, ignore.strand=TRUE)  
colData(se)  
exptData(se)
```

Index

*Topic **classes**

- GAlignmentPairs-class, 25
- GAlignments-class, 30
- GAlignmentsList-class, 35
- GappedReads-class, 40
- OverlapEncodings-class, 52

*Topic **manip**

- cigar-utils, 2
- findMateAlignment, 16
- junctions-methods, 43
- pileLettersAt, 55
- readGAlignments, 57
- sequenceLayer, 64
- stackStringsFromBam, 70

*Topic **methods**

- coverage-methods, 9
- encodeOverlaps-methods, 11
- findCompatibleOverlaps-methods, 14
- findOverlaps-methods, 20
- findSpliceOverlaps-methods, 23
- GAlignmentPairs-class, 25
- GAlignments-class, 30
- GAlignmentsList-class, 35
- GappedReads-class, 40
- intra-range-methods, 41
- junctions-methods, 43
- OverlapEncodings-class, 52
- pileLettersAt, 55
- sequenceLayer, 64
- setops-methods, 69
- stackStringsFromBam, 70
- summarizeOverlaps-methods, 74

*Topic **utilities**

- coverage-methods, 9
- encodeOverlaps-methods, 11
- findCompatibleOverlaps-methods, 14
- findOverlaps-methods, 20
- findSpliceOverlaps-methods, 23
- intra-range-methods, 41

- setops-methods, 69
- summarizeOverlaps-methods, 74
- [[,GAlignmentPairs,ANY,ANY-method
(GAlignmentPairs-class), 25
- alphabetFrequency, 71, 72
- alphabetFrequencyFromBam
(stackStringsFromBam), 70
- as.data.frame,GAlignments-method
(GAlignments-class), 30
- as.data.frame,OverlapEncodings-method
(OverlapEncodings-class), 52
- as.data.frame.OverlapEncodings
(OverlapEncodings-class), 52

- BamFile, 9, 10, 23, 24, 58, 60, 77
- BamFileList, 75, 78
- BamViews, 58, 59, 75, 76, 78
- BSgenome, 44, 46
- BStringSet, 72

- c,GAlignmentPairs-method
(GAlignmentPairs-class), 25
- c,GAlignments-method
(GAlignments-class), 30
- c,GAlignmentsList-method
(GAlignmentsList-class), 35
- c,GappedReads-method
(GappedReads-class), 40
- cigar (GAlignments-class), 30
- cigar,GAlignments-method
(GAlignments-class), 30
- cigar,GAlignmentsList-method
(GAlignmentsList-class), 35
- cigar-utils, 2, 56, 65
- CIGAR_OPS (cigar-utils), 2
- cigarNarrow (cigar-utils), 2
- cigarOpTable (cigar-utils), 2
- cigarQNarrow (cigar-utils), 2

- cigarRangesAlongPairwiseSpace
(cigar-utils), 2
- cigarRangesAlongQuerySpace
(cigar-utils), 2
- cigarRangesAlongReferenceSpace
(cigar-utils), 2
- cigarToRleList (cigar-utils), 2
- cigarWidthAlongPairwiseSpace
(cigar-utils), 2
- cigarWidthAlongQuerySpace
(cigar-utils), 2
- cigarWidthAlongReferenceSpace
(cigar-utils), 2
- class:GAlignmentPairs
(GAlignmentPairs-class), 25
- class:GAlignments (GAlignments-class),
30
- class:GAlignmentsList
(GAlignmentsList-class), 35
- class:GappedReads (GappedReads-class),
40
- class:OverlapEncodings
(OverlapEncodings-class), 52
- coerce, GAlignmentPairs, GAlignments-method
(GAlignmentPairs-class), 25
- coerce, GAlignmentPairs, GAlignmentsList-method
(GAlignmentsList-class), 35
- coerce, GAlignmentPairs, GRanges-method
(GAlignmentPairs-class), 25
- coerce, GAlignmentPairs, GRangesList-method
(GAlignmentPairs-class), 25
- coerce, GAlignments, GRanges-method
(GAlignments-class), 30
- coerce, GAlignments, GRangesList-method
(GAlignments-class), 30
- coerce, GAlignments, Ranges-method
(GAlignments-class), 30
- coerce, GAlignments, RangesList-method
(GAlignments-class), 30
- coerce, GAlignmentsList, GRanges-method
(GAlignmentsList-class), 35
- coerce, GAlignmentsList, GRangesList-method
(GAlignmentsList-class), 35
- coerce, GAlignmentsList, Ranges-method
(GAlignmentsList-class), 35
- coerce, GAlignmentsList, RangesList-method
(GAlignmentsList-class), 35
- coerce, GenomicRanges, GAlignments-method
(GAlignments-class), 30
- compare, 55
- CompressedIRangesList, 5, 32, 34
- CompressedRleList, 5
- countCompatibleOverlaps
(findCompatibleOverlaps-methods),
14
- countDumpedAlignments
(findMateAlignment), 16
- countOverlaps (findOverlaps-methods), 20
- countOverlaps, GAlignmentPairs, GAlignmentPairs-method
(findOverlaps-methods), 20
- countOverlaps, GAlignmentPairs, Vector-method
(findOverlaps-methods), 20
- countOverlaps, GAlignments, GAlignments-method
(findOverlaps-methods), 20
- countOverlaps, GAlignments, GenomicRanges-method
(findOverlaps-methods), 20
- countOverlaps, GAlignments, GRangesList-method
(findOverlaps-methods), 20
- countOverlaps, GAlignments, Vector-method
(findOverlaps-methods), 20
- countOverlaps, GAlignmentsList, GAlignmentsList-method
(findOverlaps-methods), 20
- countOverlaps, GAlignmentsList, Vector-method
(findOverlaps-methods), 20
- countOverlaps, GenomicRanges, GAlignments-method
(findOverlaps-methods), 20
- countOverlaps, GRangesList, GAlignments-method
(findOverlaps-methods), 20
- countOverlaps, Vector, GAlignmentPairs-method
(findOverlaps-methods), 20
- countOverlaps, Vector, GAlignments-method
(findOverlaps-methods), 20
- countOverlaps, Vector, GAlignmentsList-method
(findOverlaps-methods), 20
- coverage, 6, 9–11
- coverage (coverage-methods), 9
- coverage, BamFile-method
(coverage-methods), 9
- coverage, character-method
(coverage-methods), 9
- coverage, GAlignmentPairs-method
(coverage-methods), 9
- coverage, GAlignments-method
(coverage-methods), 9
- coverage-methods, 9, 11, 29, 34
- DNAStrngSet, 33, 40, 55, 56, 72

- elementMetadata, GAlignmentsList-method
(GAlignmentsList-class), 35
- elementMetadata<- , GAlignmentPairs-method
(GAlignmentPairs-class), 25
- elementMetadata<- , GAlignments-method
(GAlignments-class), 30
- elementMetadata<- , GAlignmentsList-method
(GAlignmentsList-class), 35
- encodeOverlaps, 15, 52, 53, 55
- encodeOverlaps
(encodeOverlaps-methods), 11
- encodeOverlaps, GRangesList, GRangesList-method
(encodeOverlaps-methods), 11
- encodeOverlaps, Ranges, RangesList-method
(encodeOverlaps-methods), 11
- encodeOverlaps, RangesList, Ranges-method
(encodeOverlaps-methods), 11
- encodeOverlaps, RangesList, RangesList-method
(encodeOverlaps-methods), 11
- encodeOverlaps-methods, 11
- encodeOverlaps1
(encodeOverlaps-methods), 11
- encoding, OverlapEncodings-method
(OverlapEncodings-class), 52
- end, GAlignments-method
(GAlignments-class), 30
- end, GAlignmentsList-method
(GAlignmentsList-class), 35
- explodeCigarOpLengths (cigar-utils), 2
- explodeCigarOps (cigar-utils), 2
- extractAlignmentRangesOnReference
(cigar-utils), 2
- extractAt, 65
- extractList, 46
- extractQueryStartInTranscript
(encodeOverlaps-methods), 11
- extractSkippedExonRanks
(encodeOverlaps-methods), 11
- extractSkippedExonRanks, character-method
(encodeOverlaps-methods), 11
- extractSkippedExonRanks, factor-method
(encodeOverlaps-methods), 11
- extractSkippedExonRanks, OverlapEncodings-method
(encodeOverlaps-methods), 11
- extractSpannedExonRanks
(encodeOverlaps-methods), 11
- extractSpannedExonRanks, character-method
(encodeOverlaps-methods), 11
- extractSpannedExonRanks, factor-method
(encodeOverlaps-methods), 11
- extractSpannedExonRanks, OverlapEncodings-method
(encodeOverlaps-methods), 11
- extractSpannedExonRanks, OverlapEncodings-method
(encodeOverlaps-methods), 11
- extractSteppedExonRanks
(encodeOverlaps-methods), 11
- extractSteppedExonRanks, character-method
(encodeOverlaps-methods), 11
- extractSteppedExonRanks, factor-method
(encodeOverlaps-methods), 11
- extractSteppedExonRanks, OverlapEncodings-method
(encodeOverlaps-methods), 11
- findCompatibleOverlaps, 13
- findCompatibleOverlaps
(findCompatibleOverlaps-methods),
14
- findCompatibleOverlaps, GAlignmentPairs, GRangesList-method
(findCompatibleOverlaps-methods),
14
- findCompatibleOverlaps, GAlignments, GRangesList-method
(findCompatibleOverlaps-methods),
14
- findCompatibleOverlaps-methods, 14
- findMateAlignment, 16, 60, 61
- findMateAlignment2 (findMateAlignment),
16
- findOverlaps, 12, 13, 15, 20–22
- findOverlaps (findOverlaps-methods), 20
- findOverlaps, GAlignmentPairs, GAlignmentPairs-method
(findOverlaps-methods), 20
- findOverlaps, GAlignmentPairs, Vector-method
(findOverlaps-methods), 20
- findOverlaps, GAlignments, GAlignments-method
(findOverlaps-methods), 20
- findOverlaps, GAlignments, GRangesList-method
(findOverlaps-methods), 20
- findOverlaps, GAlignments, Vector-method
(findOverlaps-methods), 20
- findOverlaps, GAlignmentsList, GAlignmentsList-method
(findOverlaps-methods), 20
- findOverlaps, GAlignmentsList, Vector-method
(findOverlaps-methods), 20
- findOverlaps, GRangesList, GAlignments-method
(findOverlaps-methods), 20
- findOverlaps, Vector, GAlignmentPairs-method
(findOverlaps-methods), 20
- findOverlaps, Vector, GAlignments-method
(findOverlaps-methods), 20

- findOverlaps, Vector, GAlignmentsList-method
(findOverlaps-methods), 20
- findOverlaps-methods, 20, 29, 34, 38
- findSpliceOverlaps
(findSpliceOverlaps-methods),
23
- findSpliceOverlaps, BamFile, ANY-method
(findSpliceOverlaps-methods),
23
- findSpliceOverlaps, character, ANY-method
(findSpliceOverlaps-methods),
23
- findSpliceOverlaps, GAlignmentPairs, GRangesList-method
(findSpliceOverlaps-methods),
23
- findSpliceOverlaps, GAlignments, GRangesList-method
(findSpliceOverlaps-methods),
23
- findSpliceOverlaps, GRangesList, GRangesList-method
(findSpliceOverlaps-methods),
23
- findSpliceOverlaps-methods, 23
- first, 58
- first (GAlignmentPairs-class), 25
- first, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- flippedQuery (OverlapEncodings-class),
52
- flippedQuery, OverlapEncodings-method
(OverlapEncodings-class), 52
- flipQuery (encodeOverlaps-methods), 11
- flushDumpedAlignments
(findMateAlignment), 16
- GAlignmentPairs, 9–11, 15, 17, 19–21, 23,
24, 30, 33, 34, 37, 38, 43–46, 57–59,
61, 75, 78
- GAlignmentPairs
(GAlignmentPairs-class), 25
- GAlignmentPairs-class, 22, 25
- GAlignments, 6, 9–11, 15–18, 20, 21, 23, 24,
26–29, 35, 36, 38, 40–46, 50, 51,
56–59, 61, 65, 70, 72, 75, 78
- GAlignments (GAlignments-class), 30
- GAlignments-class, 22, 30
- GAlignmentsList, 20, 21, 41–44, 46, 57,
59–61, 75
- GAlignmentsList
(GAlignmentsList-class), 35
- GAlignmentsList-class, 22, 35
- GappedReads, 57–59, 61
- GappedReads (GappedReads-class), 40
- GappedReads-class, 40
- GenomicRanges, 50, 51
- getBSgenome, 44, 46
- getDumpedAlignments
(findMateAlignment), 16
- GRanges, 9, 10, 20, 23, 28, 29, 33, 34, 36, 38,
41, 45, 46, 51, 56, 70, 71, 75, 77, 78
- granges, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- granges, GAlignments-method
(GAlignments-class), 30
- granges, GAlignmentsList-method
(GAlignmentsList-class), 35
- GRanges-class, 22
- GRangesList, 9, 11–13, 15, 20, 21, 23, 24, 28,
29, 32, 34, 37, 38, 41, 45, 46, 50, 51,
55, 75, 77, 78
- GRangesList-class, 22
- grglist, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- grglist, GAlignments-method
(GAlignments-class), 30
- grglist, GAlignmentsList-method
(GAlignmentsList-class), 35
- Hits, 12, 13, 15, 21, 24
- Hits-class, 22
- IntegerList, 44–46
- IntersectionNotEmpty
(summarizeOverlaps-methods), 74
- IntersectionStrict
(summarizeOverlaps-methods), 74
- intra-range-methods, 41, 42
- introns (junctions-methods), 43
- IRanges, 6, 33, 36
- IRangesList, 4–6, 37
- isCompatibleWithSkippedExons
(encodeOverlaps-methods), 11
- isCompatibleWithSkippedExons, character-method
(encodeOverlaps-methods), 11
- isCompatibleWithSkippedExons, factor-method
(encodeOverlaps-methods), 11
- isCompatibleWithSkippedExons, OverlapEncodings-method
(encodeOverlaps-methods), 11

- isCompatibleWithSplicing
 - (encodeOverlaps-methods), 11
- isCompatibleWithSplicing, character-method
 - (encodeOverlaps-methods), 11
- isCompatibleWithSplicing, factor-method
 - (encodeOverlaps-methods), 11
- isCompatibleWithSplicing, OverlapEncodings-method
 - (encodeOverlaps-methods), 11
- isProperPair (GAlignmentPairs-class), 25
- isProperPair, GAlignmentPairs-method
 - (GAlignmentPairs-class), 25

- junctions (junctions-methods), 43
- junctions, GAlignmentPairs-method
 - (junctions-methods), 43
- junctions, GAlignments-method
 - (junctions-methods), 43
- junctions, GAlignmentsList-method
 - (junctions-methods), 43
- junctions-methods, 29, 34, 38, 43

- last, 58
- last (GAlignmentPairs-class), 25
- last, GAlignmentPairs-method
 - (GAlignmentPairs-class), 25
- left (GAlignmentPairs-class), 25
- left, GAlignmentPairs-method
 - (GAlignmentPairs-class), 25
- Lencoding (OverlapEncodings-class), 52
- Lencoding, character-method
 - (OverlapEncodings-class), 52
- Lencoding, factor-method
 - (OverlapEncodings-class), 52
- Lencoding, OverlapEncodings-method
 - (OverlapEncodings-class), 52
- length, GAlignmentPairs-method
 - (GAlignmentPairs-class), 25
- length, GAlignments-method
 - (GAlignments-class), 30
- length, OverlapEncodings-method
 - (OverlapEncodings-class), 52
- levels, OverlapEncodings-method
 - (OverlapEncodings-class), 52
- levels. OverlapEncodings
 - (OverlapEncodings-class), 52
- List, 46
- Lngap (OverlapEncodings-class), 52
- Lnjunc (OverlapEncodings-class), 52
- Lnjunc, character-method
 - (OverlapEncodings-class), 52
- Lnjunc, factor-method
 - (OverlapEncodings-class), 52
- Lnjunc, OverlapEncodings-method
 - (OverlapEncodings-class), 52
- Loffset (OverlapEncodings-class), 52
- Loffset, OverlapEncodings-method
 - (OverlapEncodings-class), 52

- makeGAlignmentPairs, 26, 29
- makeGAlignmentPairs
 - (findMateAlignment), 16
- makeGAlignmentsListFromFeatureFragments
 - (GAlignmentsList-class), 35
- map, 50
- map, GenomicRanges, GAlignments-method
 - (map-methods), 50
- map-methods, 50
- mapCoords, 50, 51
- mapCoords, GenomicRanges, GAlignments-method
 - (mapCoords-methods), 51
- mapCoords-methods, 51, 51

- names, 31
- names, GAlignmentPairs-method
 - (GAlignmentPairs-class), 25
- names, GAlignments-method
 - (GAlignments-class), 30
- names, GAlignmentsList-method
 - (GAlignmentsList-class), 35
- names<-, GAlignmentPairs-method
 - (GAlignmentPairs-class), 25
- names<-, GAlignments-method
 - (GAlignments-class), 30
- names<-, GAlignmentsList-method
 - (GAlignmentsList-class), 35
- narrow (intra-range-methods), 41
- narrow, GAlignments-method
 - (intra-range-methods), 41
- narrow, GAlignmentsList-method
 - (intra-range-methods), 41
- narrow, GappedReads-method
 - (intra-range-methods), 41
- NATURAL_INTRON_MOTIFS
 - (junctions-methods), 43
- ngap, character-method
 - (OverlapEncodings-class), 52

- ngap, factor-method
(OverlapEncodings-class), 52
- ngap, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- ngap, GAlignments-method
(GAlignments-class), 30
- ngap, GAlignmentsList-method
(GAlignmentsList-class), 35
- ngap, OverlapEncodings-method
(OverlapEncodings-class), 52
- njunc (GAlignments-class), 30
- njunc, character-method
(OverlapEncodings-class), 52
- njunc, factor-method
(OverlapEncodings-class), 52
- njunc, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- njunc, GAlignments-method
(GAlignments-class), 30
- njunc, GAlignmentsList-method
(GAlignmentsList-class), 35
- njunc, OverlapEncodings-method
(OverlapEncodings-class), 52

- OverlapEncodings, 12, 13, 15
- OverlapEncodings
(OverlapEncodings-class), 52
- OverlapEncodings-class, 52
- overlapsAny (findOverlaps-methods), 20
- overlapsAny, GAlignmentPairs, GAlignmentPairs-method
(findOverlaps-methods), 20
- overlapsAny, GAlignmentPairs, Vector-method
(findOverlaps-methods), 20
- overlapsAny, GAlignments, GAlignments-method
(findOverlaps-methods), 20
- overlapsAny, GAlignments, Vector-method
(findOverlaps-methods), 20
- overlapsAny, GAlignmentsList, GAlignmentsList-method
(findOverlaps-methods), 20
- overlapsAny, GAlignmentsList, Vector-method
(findOverlaps-methods), 20
- overlapsAny, Vector, GAlignmentPairs-method
(findOverlaps-methods), 20
- overlapsAny, Vector, GAlignments-method
(findOverlaps-methods), 20
- overlapsAny, Vector, GAlignmentsList-method
(findOverlaps-methods), 20

- pileLettersAt, 55, 72

- pintersect, GAlignments, GRanges-method
(setops-methods), 69
- pintersect, GRanges, GAlignments-method
(setops-methods), 69
- pmapCoords, 50

- qnarrow (intra-range-methods), 41
- qnarrow, GAlignments-method
(intra-range-methods), 41
- qnarrow, GAlignmentsList-method
(intra-range-methods), 41
- qnarrow, GappedReads-method
(intra-range-methods), 41
- qseq (GappedReads-class), 40
- qseq, GappedReads-method
(GappedReads-class), 40
- queryLength, 12
- queryLoc2refLoc (cigar-utils), 2
- queryLocs2refLocs (cigar-utils), 2
- qwidth (GAlignments-class), 30
- qwidth, GAlignments-method
(GAlignments-class), 30
- qwidth, GAlignmentsList-method
(GAlignmentsList-class), 35
- qwidth, GappedReads-method
(GappedReads-class), 40

- Ranges, 9, 20, 28, 33, 42
- ranges, GAlignments-method
(GAlignments-class), 30
- ranges, GAlignmentsList-method
(GAlignmentsList-class), 35
- RangesList, 9, 12, 20, 32, 53, 71
- readGAlignmentPairs, 19, 26, 27, 29, 46, 77, 78
- readGAlignmentPairs (readGAlignments), 57
- readGAlignmentPairsFromBam, 16, 18, 20, 26
- readGAlignmentPairsFromBam
(readGAlignments), 57
- readGAlignmentPairsFromBam, BamFile-method
(readGAlignments), 57
- readGAlignmentPairsFromBam, character-method
(readGAlignments), 57
- readGAlignments, 31, 34, 46, 57, 78
- readGAlignmentsFromBam, 10, 11, 20, 65, 71, 72

- readGAlignmentsFromBam
(readGAlignments), 57
- readGAlignmentsFromBam, BamFile-method
(readGAlignments), 57
- readGAlignmentsFromBam, BamViews-method
(readGAlignments), 57
- readGAlignmentsFromBam, character-method
(readGAlignments), 57
- readGAlignmentsList, 38, 77
- readGAlignmentsList (readGAlignments),
57
- readGAlignmentsListFromBam, 16, 77
- readGAlignmentsListFromBam
(readGAlignments), 57
- readGAlignmentsListFromBam, BamFile-method
(readGAlignments), 57
- readGAlignmentsListFromBam, character-method
(readGAlignments), 57
- readGappedReads, 40, 41
- readGappedReads (readGAlignments), 57
- readGappedReadsFromBam
(readGAlignments), 57
- readGappedReadsFromBam, BamFile-method
(readGAlignments), 57
- readGappedReadsFromBam, character-method
(readGAlignments), 57
- readSTARJunctions (junctions-methods),
43
- readTopHatJunctions
(junctions-methods), 43
- register, 77
- relistToClass, GAlignments-method
(GAlignmentsList-class), 35
- Rencoding (OverlapEncodings-class), 52
- Rencoding, character-method
(OverlapEncodings-class), 52
- Rencoding, factor-method
(OverlapEncodings-class), 52
- Rencoding, OverlapEncodings-method
(OverlapEncodings-class), 52
- replaceAt, 65
- rglist, GAlignments-method
(GAlignments-class), 30
- rglist, GAlignmentsList-method
(GAlignmentsList-class), 35
- right (GAlignmentPairs-class), 25
- right, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- Rle, 31, 56, 59
- RleList, 6, 10, 11
- rname (GAlignments-class), 30
- rname, GAlignments-method
(GAlignments-class), 30
- rname, GAlignmentsList-method
(GAlignmentsList-class), 35
- rname<- (GAlignments-class), 30
- rname<- , GAlignments-method
(GAlignments-class), 30
- rname<- , GAlignmentsList-method
(GAlignmentsList-class), 35
- Rngap (OverlapEncodings-class), 52
- Rnjunc (OverlapEncodings-class), 52
- Rnjunc, character-method
(OverlapEncodings-class), 52
- Rnjunc, factor-method
(OverlapEncodings-class), 52
- Rnjunc, OverlapEncodings-method
(OverlapEncodings-class), 52
- Roffset (OverlapEncodings-class), 52
- Roffset, OverlapEncodings-method
(OverlapEncodings-class), 52
- scanBam, 16, 58–61, 71
- ScanBamParam, 10, 23, 58, 61, 71, 77
- selectEncodingWithCompatibleStrand
(encodeOverlaps-methods), 11
- Seqinfo, 27, 32, 36
- seqinfo, 29, 34, 38
- seqinfo, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- seqinfo, GAlignments-method
(GAlignments-class), 30
- seqinfo, GAlignmentsList-method
(GAlignmentsList-class), 35
- seqinfo<- , GAlignmentPairs-method
(GAlignmentPairs-class), 25
- seqinfo<- , GAlignments-method
(GAlignments-class), 30
- seqinfo<- , GAlignmentsList-method
(GAlignmentsList-class), 35
- seqlevels, 27, 32
- seqlevelsInUse, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- seqnames, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- seqnames, GAlignments-method
(GAlignments-class), 30

- seqnames, GAlignmentsList-method
(GAlignmentsList-class), 35
- seqnames<- , GAlignments-method
(GAlignments-class), 30
- seqnames<- , GAlignmentsList-method
(GAlignmentsList-class), 35
- sequenceLayer, 6, 63, 71, 72
- setops-methods, 69, 70
- show, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- show, GAlignments-method
(GAlignments-class), 30
- show, GAlignmentsList-method
(GAlignmentsList-class), 35
- show, OverlapEncodings-method
(OverlapEncodings-class), 52
- SimpleIRangesList, 5
- SimpleList, 59
- solveUserSEW, 5, 41
- splitAsListReturnedClass, GAlignments-method
(GAlignments-class), 30
- stackStrings, 71, 72
- stackStringsFromBam, 56, 65, 70
- start, GAlignments-method
(GAlignments-class), 30
- start, GAlignmentsList-method
(GAlignmentsList-class), 35
- strand, GAlignmentPairs-method
(GAlignmentPairs-class), 25
- strand, GAlignments-method
(GAlignments-class), 30
- strand, GAlignmentsList-method
(GAlignmentsList-class), 35
- strand<- , GAlignmentPairs-method
(GAlignmentPairs-class), 25
- strand<- , GAlignments-method
(GAlignments-class), 30
- strand<- , GAlignmentsList-method
(GAlignmentsList-class), 35
- subjectLength, 12
- subsetByOverlaps
(findOverlaps-methods), 20
- subsetByOverlaps, GAlignmentPairs, GAlignmentPairs-method
(findOverlaps-methods), 20
- subsetByOverlaps, GAlignmentPairs, Vector-method
(findOverlaps-methods), 20
- subsetByOverlaps, GAlignments, GAlignments-method
(findOverlaps-methods), 20
- subsetByOverlaps, GAlignments, Vector-method
(findOverlaps-methods), 20
- subsetByOverlaps, GAlignmentsList, GAlignmentsList-method
(findOverlaps-methods), 20
- subsetByOverlaps, GAlignmentsList, Vector-method
(findOverlaps-methods), 20
- subsetByOverlaps, Vector, GAlignmentPairs-method
(findOverlaps-methods), 20
- subsetByOverlaps, Vector, GAlignments-method
(findOverlaps-methods), 20
- subsetByOverlaps, Vector, GAlignmentsList-method
(findOverlaps-methods), 20
- SummarizedExperiment, 75, 78
- summarizeJunctions (junctions-methods),
43
- summarizeOverlaps, 24
- summarizeOverlaps
(summarizeOverlaps-methods), 74
- summarizeOverlaps, BamViews, missing-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, BamFile-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, BamFileList-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, character-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, GAlignmentPairs-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, GAlignments-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, GAlignmentsList-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, GRanges-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRanges, GRangesList-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRangesList, BamFile-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRangesList, BamFileList-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRangesList, character-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRangesList, GAlignmentPairs-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRangesList, GAlignments-method
(summarizeOverlaps-methods), 74
- summarizeOverlaps, GRangesList, GAlignmentsList-method
(summarizeOverlaps-methods), 74

summarizeOverlaps,GRangesList,GRanges-method
 (summarizeOverlaps-methods), 74

summarizeOverlaps,GRangesList,GRangesList-method
 (summarizeOverlaps-methods), 74

summarizeOverlaps-methods, 74

Union (summarizeOverlaps-methods), 74

unlist,GAlignmentPairs-method
 (GAlignmentPairs-class), 25

updateObject,GAlignments-method
 (GAlignments-class), 30

updateObject,GAlignmentsList-method
 (GAlignmentsList-class), 35

validCigar (cigar-utils), 2

width,GAlignments-method
 (GAlignments-class), 30

width,GAlignmentsList-method
 (GAlignmentsList-class), 35

XStringSet, 28, 55, 56, 64, 65, 72