Package 'GenomicAlignments'

March 31, 2025

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.

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

URL https://bioconductor.org/packages/GenomicAlignments

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

BugReports https://github.com/Bioconductor/GenomicAlignments/issues

Version 1.42.0

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.0.0), methods, BiocGenerics (>= 0.37.0), S4Vectors (>= 0.27.12), IRanges (>= 2.23.9), GenomeInfoDb (>= 1.13.1), GenomicRanges (>= 1.55.3), SummarizedExperiment (>= 1.9.13), Biostrings (>= 2.55.7), Rsamtools (>= 1.31.2)

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

LinkingTo S4Vectors, IRanges

Suggests ShortRead, 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, DESeq2, edgeR, RUnit, knitr, BiocStyle

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 stackStringsFromGAlignments.R intra-range-methods.R coverage-methods.R setops-methods.R findOverlaps-methods.R coordinate-mapping-methods.R encodeOverlaps-methods.R findCompatibleOverlaps-methods.R summarizeOverlaps-methods.R findSpliceOverlaps-methods.R zzz.R

VignetteBuilder knitr

git_url https://git.bioconductor.org/packages/GenomicAlignments

git_branch RELEASE_3_20

git_last_commit 5a82c79

git_last_commit_date 2024-10-29

Repository Bioconductor 3.20

Date/Publication 2025-03-31

Author Hervé Pagès [aut, cre],
Valerie Obenchain [aut],
Martin Morgan [aut],
Fedor Bezrukov [ctb],
Robert Castelo [ctb],
Halimat C. Atanda [ctb] (Translated 'WorkingWithAlignedNucleotides' vignette from Sweave to RMarkdown / HTML.)

Maintainer Hervé Pagès <hpages.on.github@gmail.com>

Contents

cigar-utils *CIGAR utility functions*

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.
N.regions.removed	TRUE or FALSE. If TRUE, then cigarRangesAlongReferenceSpace and cigarWidthAlongReference report ranges/widths with respect to the "reference" space from which the N regions have been removed, and cigarRangesAlongPairwiseSpace and cigarWidthAlongPairwiseS report them with respect to the "pairwise" space from which the N regions have been removed.
pos	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 cigar), or the same length as cigar.
f	NULL or a factor of length cigar. If NULL, then the ranges are grouped by alignment i.e. the returned IRangesList object has 1 list element per element in cigar. Otherwise they are grouped by factor level i.e. the returned IRangesList object has 1 list element per level in f and is named with those levels. For example, if f is a factor containing the chromosome for each read, then the returned IRangesList object will have 1 list element per chromosome and each list element will contain all the ranges on that chromosome.
drop.empty.ranges	Should empty ranges be dropped?
reduce.ranges	Should adjacent ranges coming from the same cigar be merged or not? Using TRUE can significantly reduce the size of the returned object.
with.ops	TRUE or FALSE indicating whether the returned ranges should be named with their corresponding CIGAR operation.
before.hard.clipping	TRUE or FALSE. If TRUE, then cigarRangesAlongQuerySpace and cigarWidthAlongQuerySpace report ranges/widths with respect to the "query" space to which the H regions have been added. before.hard.clipping and after.soft.clipping cannot both be TRUE.

after.soft.clipping
: TRUE or FALSE. If TRUE, then cigarRangesAlongQuerySpace and cigarWidthAlongQuerySpace report ranges/widths with respect to the "query" space from which the S regions have been removed. before.hard.clipping and after.soft.clipping cannot both be TRUE.

dense
: TRUE or FALSE. If TRUE, then cigarRangesAlongPairwiseSpace and cigarWidthAlongPairwiseSp report ranges/widths with respect to the "pairwise" space from which the I, D, and N regions have been removed. N.regions.removed and dense cannot both be TRUE.

drop.D.ranges
: 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 drop.D.ranges is TRUE, then Ds and Ns in the CIGAR are equivalent.

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 the details).

qloc
: An integer vector containing "query-based locations" i.e. 1-based locations relative to the query sequence stored in the SAM/BAM file.

qlocs
: A list of the same length as cigar 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 elementNROWS(). 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)

Hervé Pagès & 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. 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.

flag0 <- scanBamFlag(isUnmappedQuery=FALSE, isDuplicate=FALSE)

extractCoverageFromBAM <- function(bamfile)
{
  stopifnot(is(bamfile, "BamFile"))
  ## This ScanBamParam object allows us to load only the necessary
  ## information from the file.
  param <- ScanBamParam(flag=flag0, what=c("rname", "pos", "cigar"))
  bam <- scanBam(bamfile, 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.
  f <- factor(bam$rname, levels=seqlevels(bamfile))
  irl <- extractAlignmentRangesOnReference(bam$cigar, pos=bam$pos, f=f)
  coverage(irl, width=seqlengths(bamfile))
}

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

## extractCoverageFromBAM() is equivalent but slightly more efficient
## than loading a GAlignments object and computing its coverage:
cvg2 <- coverage(readGAlignments(f1, param=ScanBamParam(flag=flag0)))
stopifnot(identical(cvg, cvg2))

## ---------------------------------------------------------------------
## D. 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)
```

```
## ---------------------------------------------------------------------
## E. 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)
}
```

`coverage-methods` *Coverage of a GAlignments, GAlignmentPairs, or GAlignmentsList object*

Description

`coverage` methods for GAlignments, GAlignmentPairs, GAlignmentsList, and BamFile objects.

NOTE: The `coverage` generic function and methods for IntegerRanges and IntegerRangesList 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", "naive"), drop.D.ranges=FALSE)
```

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

## S4 method for signature 'GAlignmentsList'
coverage(x, shift=0L, width=NULL, weight=1L, ...)

## 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, GAlignmentsList, or BamFile object, or the path to a BAM file.
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 readGAlignments.
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, drop.D.ranges=drop.D.ranges), ...)
```

The method for GAlignmentsList objects does:

```
coverage(unlist(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:

```
gal <- readGAlignments(x, param=param)
coverage(gal, 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.
- readGAlignments.
- BamFile objects in the **Rsamtools** package.

Examples

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

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

## Coverage of a GAlignments object:
gal <- readGAlignments(ex1_file)
cvg1 <- coverage(gal)
cvg1

## Coverage of a GAlignmentPairs object:
galp <- readGAlignmentPairs(ex1_file)
cvg2 <- coverage(galp)
cvg2

## Coverage of a GAlignmentsList object:
galist <- readGAlignmentsList(ex1_file)
cvg3 <- coverage(galist)
cvg3

table(mcols(galist)$mate_status)
mated_idx <- which(mcols(galist)$mate_status == "mated")
mated_galist <- galist[mated_idx]
mated_cvg3 <- coverage(mated_galist)
mated_cvg3

## Sanity checks:
stopifnot(identical(cvg1, cvg3))
stopifnot(identical( cvg2, mated_cvg3))

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

library(pasillaBamSubset)
## See '?pasillaBamSubset' for more information about the 2 BAM files
## included in this package.
reads <- readGAlignments(untreated3_chr4())
table(njunc(reads))  # data contains junction reads

## Junctions do NOT contribute to the coverage:
read1 <- reads[which(njunc(reads) != 0L)[1]]  # 1st read with a junction
read1  # cigar shows a "skipped region" of length 15306
```

```
grglist(read1)[[1]]  # the junction is between pos 4500 and 19807
coverage(read1)$chr4  # junction is not covered

## Sanity checks:
cvg <- coverage(reads)
read_chunks <- unlist(grglist(reads), use.names=FALSE)
read_chunks_per_chrom <- split(read_chunks, seqnames(read_chunks))
stopifnot(identical(sum(cvg), sum(width(read_chunks_per_chrom))))

galist <- readGAlignmentsList(untreated3_chr4())
stopifnot(identical(cvg, coverage(galist)))
```

---

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)

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. IntegerRangesList 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 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 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 `isCompatibleWithSkippedExons`: A logical vector *parallel* to `x`.

For `extractSteppedExonRanks`, `extractSpannedExonRanks`, and `extractSkippedExonRanks`: TODO

For `extractQueryStartInTranscript`: TODO

Author(s)

Hervé Pagès

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 IntegerRangesList 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
## IntegerRangesList 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)
```

```
## ------------------------------------------------------------------------
## 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 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` for `countCompatibleOverlaps`.

Author(s)

Hervé Pagès

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)
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
exbytx <- exonsBy(txdb, 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 `readGAlignmentPairs` for pairing the records loaded from a BAM file containing aligned paired-end reads. Starting with BioC 2.14, `readGAlignmentPairs` relies on `scanBam(BamFile(asMates=TRUE), ...)` for the pairing.

Usage

```
findMateAlignment(x)
makeGAlignmentPairs(x, use.names=FALSE, use.mcols=FALSE, strandMode=1)

## 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:
	`param <- ScanBamParam(what=c("flag", "mrnm", "mpos"))`
	`x <- readGAlignments(..., use.names=TRUE, param=param)`

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.
strandMode	Strand mode to set on the returned GAlignmentPairs object. See ?strandMode for more information.

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 Ox10 (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 `isSecondaryAlignment` flag to `FALSE` in ScanBamParam(). See examples in `?readGAlignmentPairs` 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 isSecondaryAlignment
[1,]                 1               0                1                     0
[2,]                 1               0                1                     0
[3,]                 0               1                0                     0
[4,]                 0               1                0                     0
     isNotPassingQualityControls isDuplicate
[1,]                           0           0
[2,]                           0           0
[3,]                           0           0
[4,]                           0           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 an 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)

Hervé Pagès

See Also

- GAlignments and GAlignmentPairs objects.
- `readGAlignments` and `readGAlignmentPairs`.

Examples

```
bamfile <- system.file("extdata", "ex1.bam", package="Rsamtools",
                       mustWork=TRUE)
param <- ScanBamParam(what=c("flag", "mrnm", "mpos"))
x <- readGAlignments(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 IntegerRanges and IntegerRangesList objects are defined and documented in the **IRanges** package. The methods for GRanges and GRangesList objects are defined and documented in the **GenomicRanges** package.

GAlignments, GAlignmentPairs, and GAlignmentsList objects also support `countOverlaps`, `overlapsAny`, and `subsetByOverlaps` thanks to the default methods defined in the **IRanges** package and to the `findOverlaps` method defined in this package and documented below.

Usage

```
## S4 method for signature 'GAlignments,GAlignments'
findOverlaps(query, subject,
    maxgap=-1L, minoverlap=0L,
    type=c("any", "start", "end", "within"),
    select=c("all", "first", "last", "arbitrary"),
    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

A Hits object when select="all" or an integer vector otherwise.

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 `singleEnd` 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 `param` and `singleEnd` 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 comparision 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.

Note

WARNING: The current implementation of `findSpliceOverlaps` doesn't work properly on paired-end reads where the 2 ends overlap!

Author(s)

Michael Lawrence and Valerie Obenchain

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", 5, "11M4N6M", strand("+")),
    GAlignments("chr1", 50, "6M", strand("-")))
findSpliceOverlaps(galp, genes)
```

GAlignmentPairs-class *GAlignmentPairs objects*

Description

The GAlignmentPairs class is a container for storing *pairs of genomic alignments*. These pairs are typically obtained by aligning paired-end reads to a reference genome or transcriptome.

Details

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

An *alignment pair* is made of a "first" and a "last"/"second" alignment, and is formally represented by a GAlignments object of length 2. In most applications, an *alignment pair* will represent an aligned paired-end read. In that case, the "first" member of the pair represents the alignment of the first end of the read (aka "first segment in the template", using SAM Spec terminology), and the "last" member of the pair represents the alignment 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, strandMode=1, isProperPair=TRUE, names=NULL)`**:** Low-level GAlignmentPairs constructor. Generally not used directly.

Accessors

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

`strandMode(x)`**,** `strandMode(x) <- value`**:** The *strand mode* is a per-object switch on GAlignmentPairs objects that controls the behavior of the `strand` getter. More precisely, it indicates how the strand of a pair should be inferred from the strand of the first and last alignments in the pair:

- 0: strand of the pair is always *.
- 1: strand of the pair is strand of its first alignment. This mode should be used when the paired-end data was generated using one of the following stranded protocols: Directional Illumina (Ligation), Standard SOLiD.
- 2: strand of the pair is strand of its last alignment. This mode should be used when the paired-end data was generated using one of the following stranded protocols: dUTP, NSR, NNSR, Illumina stranded TruSeq PE protocol.

These modes are equivalent to `strandSpecific` equal 0, 1, and 2, respectively, for the `featureCounts` function defined in the **Rsubread** package.

Note that, by default, the `readGAlignmentPairs` function sets the *strand mode* to 1 on the returned GAlignmentPairs object. The function has a `strandMode` argument to let the user set a different *strand mode*. The *strand mode* can also be changed any time with the `strandMode` setter or with `invertStrand`.

Also note that 3rd party programs TopHat2 and Cufflinks have a `--library-type` option to let the user specify which protocol was used. Please refer to the documentation of these programs for more information.

`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, real.strand=FALSE)`, `last(x, real.strand=FALSE)`, `second(x, real.strand=FALSE)`: Get the "first" or "last"/"second" alignment for each alignment pair in `x`. The result is a GAlignments object of the same length as `x`.

If `real.strand=TRUE`, then the strand is inverted on-the-fly according to the *strand mode* currently set on the object (see `strandMode(x)` above). More precisely, if `strandMode(x)` is 0, then the strand is set to * for the GAlignments object returned by both, `first()` and `last()`. If `strandMode(x)` is 1, then the strand of the object returned by `last()` is inverted. If `strandMode(x)` is 2, then the strand of the object returned by `first()` is inverted.

`seqnames(x)`: Get the sequence names of the pairs in `x` i.e. the name of the reference sequence for each alignment pair in `x`. The sequence name of a pair is the sequence name of the 2 alignments in the pair if they are the same (*concordant seqnames*), or NA if they differ (*discordant seqnames*).

The sequence names are returned in a factor-Rle object that is *parallel* to `x`, i.e. the i-th element in the returned object is the sequence name of the i-th pair in `x`.

`strand(x)`: Get the strand for each alignment pair in `x`. Obey `strandMode(x)` above to infer the strand of a pair. Return `*` for pairs with *discordant strand*, or for all pairs if `strandMode(x)` is 0.

`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.

`granges(x, use.names=TRUE, use.mcols=FALSE, on.discordant.seqnames=c("error", "drop", "split")), ra`
Return a GRanges object (for `granges()`) or IRanges) object (for `ranges()`).

If x contains no pairs with *discordant seqnames*, the operation is guaranteed to be successful and to return an object *parallel* to x, that is, an object where the i-th element is the range of the genomic region spanned by the i-th alignment in x (all gaps in the region are ignored).

If x contains pairs with discordant seqnames, then an error is raised, unless the `on.discordant.seqnames` argument is set to `"drop"` or `"split"`, in which case the pairs with discordant seqnames are either dropped or represented with 2 genomic ranges (or 2 ranges for `ranges()`) in the returned object. In that case, the returned object is NOT *parallel* to x.

If `use.names` is TRUE, then the names on x (if any) are propagated to the returned object. If `use.mcols` is TRUE, then the metadata columns on x (if any) are propagated to the returned object.

`grglist(x, use.mcols=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. The strand of the returned ranges obeys the *strand mode* currently set on the object (see `strandMode(x)` above).

More precisely, if `grl1` and `grl2` are `grglist(first(x, real.strand=TRUE), order.as.in.query=TRUE)` and `grglist(last(x, real.strand=TRUE), order.as.in.query=TRUE)`, respectively, then the i-th element in the returned GRangesList object is `c(grl1[[i]], grl2[[i]])` if `strandMode(x)` is 0 or 1, or `c(grl2[[i]], grl1[[i]])` if `strandMode(x)` is 2.

Note that, if `strandMode(x)` is 1 or 2, this results in the ranges being in consistent order with the original "query template", that is, being in the order defined by walking the "query template" from the beginning to the end.

If `use.names` is TRUE, then the names on x (if any) are propagated to the returned object. If `use.mcols` is TRUE, then the metadata columns on x (if any) are propagated to the returned object.

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.

`as(x, "GRanges"), as(x, "IntegerRanges"), as(x, "GRangesList")`: Alternate ways of doing `granges(x, use.names=TRUE, use.mcols=TRUE)`, `ranges(x, use.names=TRUE, use.mcols=TRUE)`, and `grglist(x, use.names=TRUE, 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 Vector derivatives defined in the **IRanges** and **Biostrings** packages (e.g. IRanges and XStringSet objects).

Author(s)

Hervé Pagès

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, strandMode=1)
galp

length(galp)
head(galp)
head(names(galp))

first(galp)
last(galp)
# or
second(galp)

strandMode(galp)
first(galp, real.strand=TRUE)
last(galp, real.strand=TRUE)
strand(galp)

strandMode(galp) <- 2
first(galp, real.strand=TRUE)
last(galp, real.strand=TRUE)
strand(galp)
```

```
seqnames(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

strandMode(galp) <- 1
grglist(galp)

## Alternatively the strand mode can be changed with invertStrand():
invertStrand(galp)

stopifnot(identical(unname(elementNROWS(grglist(galp))), njunc(galp) + 2L))

granges(galp)  # a GRanges object
```

GAlignments-class *GAlignments objects*

Description

The GAlignments class is a simple container which 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 ?GAlignmentPairs 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 expicitly 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(0), cigar=character(0), strand=NULL, names=NULL, .`
: 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, to `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(elementNROWS(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.

`granges(x, use.names=TRUE, use.mcols=FALSE)`, `ranges(x, use.names=TRUE, use.mcols=FALSE)`: Return a GRanges object (for `granges()`) or IRanges) object (for `ranges()`) *parallel* to `x` where the i-th element is the range of the genomic region spanned by the i-th alignment in `x`. All gaps in the region are ignored.

If `use.names` is TRUE, then the names on `x` (if any) are propagated to the returned object. If `use.mcols` is TRUE, then the metadata columns on `x` (if any) are propagated to the returned object.

`grglist(x, use.names=TRUE, use.mcols=FALSE, order.as.in.query=FALSE, drop.D.ranges=FALSE)`, `rglist(x`
Return either a GRangesList or a IntegerRangesList 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 IntegerRangesList object returned by `rglist(x)` is a CompressedIRangesList object.

If `use.names` is TRUE, then the names on `x` (if any) are propagated to the returned object. If `use.mcols` is TRUE, then the metadata columns on `x` (if any) are 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 decending 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.

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

Concatenation

`c(x, ..., ignore.mcols=FALSE)`: Concatenate GAlignments object `x` and the GAlignments objects in `...` together. See `?c` in the **S4Vectors** package for more information about concatenating Vector derivatives.

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. IntegerRanges and DNAStringSet objects).

Author(s)

Hervé Pagès 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)

## Invert the strand:
invertStrand(gal)

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

grglist(gal)  # a GRangesList object
```

```
stopifnot(identical(unname(elementNROWS(grglist(gal))), njunc(gal) + 1L))
granges(gal)  # a GRanges object
rglist(gal)   # a CompressedIRangesList object
stopifnot(identical(unname(elementNROWS(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 don't 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 doesn't 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.names=TRUE, use.mcols=FALSE, ignore.strand=FALSE)`, `ranges(x, use.names=TRUE, use.mcol`
: 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 junctions or spaces between paired-end reads.

  If `use.names` is TRUE, then the names on `x` (if any) are propagated to the returned object. If `use.mcols` is TRUE, then the metadata columns on `x` (if any) are 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.names=TRUE, use.mcols=FALSE, ignore.strand=FALSE)`, `rglist(x, use.names=TRUE, use.mcol`
: Return either a GRangesList or an 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 junctions or spaces between paired-end reads.

  If `use.names` is TRUE, then the names on `x` (if any) are propagated to the returned object. If `use.mcols` is TRUE, then the metadata columns on `x` (if any) are propagated to the returned object.

  `grglist` coercion supports `ignore.strand` to allow ranges of opposite strand to be combined (see examples). When `ignore.strand` is TRUE all ranges in the resulting GRangesList have strand '*'.

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

`as.data.frame(x, row.names = NULL, optional = FALSE, ..., value.name = "value", use.outer.mcols = FALSE`
: 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.

Concatenation

`c(x, ..., ignore.mcols=FALSE)`: Concatenate GAlignmentsList object `x` and the GAlignmentsList objects in `...` together. See ?`c` in the **S4Vectors** package for more information about concatenating Vector derivatives.

Author(s)

Valerie Obenchain

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.
fl <- untreated3_chr4()
galist1 <- readGAlignmentsList(fl)

galist1[1:3]
length(galist1)
table(elementNROWS(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(fl, yieldSize=3, asMates=TRUE)
readGAlignmentsList(bf)

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

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

## The granges() and grlist() coercions support 'ignore.strand' to
## allow ranges from different strands 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 junctions or the space between pairs, 'ignore.strand'
## must be TRUE.
granges(galist[1])
granges(galist[1], ignore.strand=TRUE)

## grglist()
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 `DNAStringSet` object.

Author(s)

Hervé Pagès

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'
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.

Details

- () `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 instead of the reference 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)

Hervé Pagès and Valerie Obenchain

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")
param <- ScanBamParam(what=c("seq", "qual"))
gal <- readGAlignments(ex1_file, param=param)
gal

## This trims 3 nucleotides on the left and 5 nucleotides on the right
## of each alignment:
gal2 <- qnarrow(gal, start=4, end=-6)
gal2

## Note that the 'start' and 'end' values are relative to the query
## sequences and specify the query substring that must be kept for each
## alignment. Negative values are relative to the right end of the query
## sequence.

## Also note that the metadata columns on 'gal' are propagated as-is so
## the "seq" and "qual" matadata columns must be adjusted "by hand" with
## narrow();
mcols(gal2)$seq <- narrow(mcols(gal)$seq, start=4, end=-6)
mcols(gal2)$qual <- narrow(mcols(gal)$qual, start=4, end=-6)
gal2

## Sanity checks:
stopifnot(identical(qwidth(gal2), width(mcols(gal2)$seq)))
stopifnot(identical(qwidth(gal2), width(mcols(gal2)$qual)))

## ---------------------------------------------------------------------
## 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

qnarrow(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() generic and methods
## -------------------------------

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 mcols(x)) 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 summarizeJunctions. This metadata column is an IntegerList object representing the mapping from each element in the ouput (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 getBSgenome function defined in the **BSgenome** software package. In that case the corresponding BSgenome data package needs to be already installed (see ?getBSgenome in the **BSgenome** package for the details). If genome is supplied, then the intron_motif and intron_strand metadata columns are computed (based on the dinucleotides found at the intron boundaries) and added to the output of summarizeJunctions. 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 *junctions.bed* or *new_list.juncs* file for readTopHatJunctions, and the *SJ.out.tab* file for readSTARJunctions.
file.is.raw.juncs	TRUE or FALSE (the default). If set to TRUE, then the input file is assumed to be a TopHat *.juncs* file instead of the *junctions.bed* file generated by TopHat. A TopHat *.juncs* file can be obtained by passing the *junctions.bed* file thru TopHat's *bed_to_juncs* 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, on a GAlignments object x, junctions(x) is equivalent to:

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

On a GAlignmentPairs object x, it's equivalent to (but faster than):

```
mendoapply(c, junctions(first(x, real.strand=TRUE)),
              junctions(last(x, real.strand=TRUE)))
```

Note that starting with BioC 3.2, the behavior of junctions on a GAlignmentPairs object has been slightly modified so that the returned ranges now have the *real strand* set on them. See the documentation of the real.strand argument in the man page of GAlignmentPairs objects for more information.

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 ouput (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)

Hervé Pagès

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

- The readGAlignments and readGAlignmentPairs functions for reading genomic alignments from a BAM file.
- GAlignments, GAlignmentPairs, and GAlignmentsList objects.
- GRanges and GRangesList objects implemented and documented in the **GenomicRanges** package.
- IntegerList objects implemented 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 extractList function in the **IRanges** package, for extracting groups of elements from a vector-like object and returning them into a List object.

Examples

```
library(RNAseqData.HNRNPC.bam.chr14)
bamfile <- RNAseqData.HNRNPC.bam.chr14_BAMFILES[1]

## ---------------------------------------------------------------------
## A. junctions()
## ---------------------------------------------------------------------

gal <- readGAlignments(bamfile)
table(njunc(gal))  # some alignments have 3 junctions!
juncs <- junctions(gal)
juncs

stopifnot(identical(unname(elementNROWS(juncs)), njunc(gal)))

galp <- readGAlignmentPairs(bamfile)
juncs <- junctions(galp)
juncs

stopifnot(identical(unname(elementNROWS(juncs)), njunc(galp)))

## ---------------------------------------------------------------------
## B. summarizeJunctions()
## ---------------------------------------------------------------------
```

```
## By default, only the "score", "plus_score", and "minus_score"
## metadata columns are returned:
junc_summary <- summarizeJunctions(gal)
junc_summary

## The "score" metadata column reports the total number of alignments
## crossing each junction, i.e., that have the junction encoded in their
## CIGAR:
median(mcols(junc_summary)$score)

## The "plus_score" and "minus_score" metadata columns report the
## strand-specific number of alignments crossing each junction:
stopifnot(identical(mcols(junc_summary)$score,
                    mcols(junc_summary)$plus_score +
                    mcols(junc_summary)$minus_score))

## If 'with.revmap' is TRUE, the "revmap" metadata column is added to
## the output. This metadata column is an IntegerList object represen-
## ting the mapping from each element in the ouput (i.e. a junction) to
## the corresponding elements in the input 'x'. Here we're going to use
## this to compute a 'score2' for each junction. We obtain this score
## by summing the mapping qualities of the alignments crossing the
## junction:
gal <- readGAlignments(bamfile, param=ScanBamParam(what="mapq"))
junc_summary <- summarizeJunctions(gal, with.revmap=TRUE)
junc_score2 <- sum(extractList(mcols(gal)$mapq,
                               mcols(junc_summary)$revmap))
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 doesn't 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.

## ---------------------------------------------------------------------
## D. 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, pruning.mode="coarse") <- "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
## (that's 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 TopHat's
                             # 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))

## Let's 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!
```

`mapToAlignments` *Map range coordinates between reads and genome space using CIGAR alignments*

Description

Map range coordinates between reads (local) and genome (reference) space using the CIGAR in a `GAlignments` object.

See ?`mapToTranscripts` in the **GenomicRanges** package for mapping coordinates between features in the transcriptome and genome space.

Usage

```
## S4 method for signature 'GenomicRanges,GAlignments'
mapToAlignments(x, alignments, ...)
## S4 method for signature 'GenomicRanges,GAlignments'
pmapToAlignments(x, alignments, ...)

## S4 method for signature 'GenomicRanges,GAlignments'
mapFromAlignments(x, alignments, ...)
## S4 method for signature 'GenomicRanges,GAlignments'
pmapFromAlignments(x, alignments, ...)
```

Arguments

`x`	`GenomicRanges` object of positions to be mapped. `x` must have names when mapping to the genome.

alignments — A GAlignments object that represents the alignment of x to the genome. The aligments object must have names. When mapping to the genome names are used to determine mapping pairs and in the reverse direction they are used as the seqlevels of the output object.

... — Arguments passed to other methods.

Details

These methods use a GAlignments object to represent the alignment between the ranges in x and the output. The following CIGAR operations in the "Extended CIGAR format" are used in the mapping algorithm:

- M, X, = Sequence match or mismatch
- I Insertion to the reference
- D Deletion from the reference
- N Skipped region from the reference
- S Soft clip on the read
- H Hard clip on the read
- P Silent deletion from the padded reference

- mapToAlignments, pmapToAlignments The CIGAR is used to map the genomic (reference) position x to local coordinates. The mapped position starts at

```
start(x) - start(alignments) + 1
```

  and is incremented or decremented as the algorithm walks the length of the CIGAR. A successful mapping in this direction requires that x fall within alignments.

  The seqlevels of the return object are taken from the alignments object and will be a name descriptive of the read or aligned region. In this direction, mapping is attempted between all elements of x and all elements of alignments.

- mapFromAlignments, pmapFromAlignments The CIGAR is used to map the local position x to genomic (reference) coordinates. The mapped position starts at

```
start(x) + start(alignments) - 1
```

  and is incremented or decremented as the algorithm walks the length of the CIGAR. A successful mapping in this direction requires that the width of alignments is <= the width of x.

  When mapping to the genome, name matching is used to determine the mapping pairs (vs attempting to match all possible pairs). Ranges in x are only mapped to ranges in alignments with the same name. Name matching is motivated by use cases such as differentially expressed regions where the expressed regions in x would only be related to a subset of regions in alignments, which may contains gene or transcript ranges.

- element-wise versions pmapToAlignments and pmapFromAlignments are element-wise (aka 'parallel') versions of mapToAlignments and mapFromAlignments. The i-th range in x is mapped to the i-th range in alignments; x and alignments must have the same length.

  Ranges in x that do not map (out of bounds) are returned as zero-width ranges starting at 0. These ranges are given the special seqname of "UNMAPPED". Note the non-parallel methods do not return unmapped ranges so the "UNMAPPED" seqname is unique to pmapToAlignments and pmapFromAlignments.

- strand By SAM convention, the CIGAR string is reported for mapped reads on the forward genomic strand. There is no need to consider strand in these methods. The output of these methods will always be unstranded (i.e., "*").

Value

An object the same class as `x`.

Parallel methods return an object the same shape as `x`. Ranges that cannot be mapped (out of bounds) are returned as zero-width ranges starting at 0 with a seqname of "UNMAPPED".

Non-parallel methods return an object that varies in length similar to a Hits object. The result only contains mapped records, out of bound ranges are not returned. `xHits` and `alignmentsHits` metadata columns indicate the elements of `x` and `alignments` used in the mapping.

When present, names from `x` are propagated to the output. When mapping locally, the seqlevels of the output are the names on the `alignment` object. When mapping globally, the output seqlevels are the seqlevels of `alignment` which are usually chromosome names.

Author(s)

V. Obenchain, M. Lawrence and H. Pagès

See Also

- ?`mapToTranscripts` in the in the **GenomicFeatures** package for methods mapping between transcriptome and genome space.
- http://samtools.sourceforge.net/ for a description of the Extended CIGAR format.

Examples

```
## ---------------------------------------------------------------------
## A. Basic use
## ---------------------------------------------------------------------

## 1. Map to local space with mapToAlignments()
## ---------------------------------------------------------------------

## Mapping to local coordinates requires 'x' to be within 'alignments'.
## In this 'x', the second range is too long and can't be mapped.
alignments <- GAlignments("chr1", pos=10, cigar="11M", names="read_A")
x <- GRanges("chr1", IRanges(c(12, 12), width=c(6, 20)))
mapToAlignments(x, alignments)

## The element-wise version of the function returns unmapped ranges
## as zero-width ranges with a seqlevel of "UNMAPPED":
pmapToAlignments(x, c(alignments, alignments))

## Mapping the same range through different alignments demonstrates
## how the CIGAR operations affect the outcome.
ops <- c("no-op", "junction", "insertion", "deletion")
x <- GRanges(rep("chr1", 4), IRanges(rep(12, 4), width=rep(6, 4), names=ops))
alignments <- GAlignments(rep("chr1", 4), pos=rep(10, 4),
                          cigar=c("11M", "5M2N4M", "5M2I4M", "5M2D4M"),
                          names=paste0("region_", 1:4))
pmapToAlignments(x, alignments)
```

```
## 2. Map to genome space with mapFromAlignments()
## ---------------------------------------------------------------------

## One of the criteria when mapping to genomic coordinates is that the
## shifted 'x' range falls within 'alignments'. Here the first 'x'
## range has a shifted start value of 14 (5 + 10 - 1 = 14) with a width of
## 2 and so is successfully mapped. The second has a shifted start of 29
## (20 + 10 - 1 = 29) which is outside the range of 'alignments'.
x <- GRanges("chr1", IRanges(c(5, 20), width=2, names=rep("region_A", 2)))
alignments <- GAlignments("chr1", pos=10, cigar="11M", names="region_A")
mapFromAlignments(x, alignments)

## Another characteristic of mapping this direction is the name matching
## used to determine pairs. Mapping is only attempted between ranges in 'x'
## and 'alignments' with the same name. If we change the name of the first 'x'
## range, only the second will be mapped to 'alignment'. We know the second
## range fails to map so we get an empty result.
names(x) <- c("region_B", "region_A")
mapFromAlignments(x, alignments)

## CIGAR operations: insertions reduce the width of the output while
## junctions and deletions increase it.
ops <- c("no-op", "junction", "insertion", "deletion")
x <- GRanges(rep("chr1", 4), IRanges(rep(3, 4), width=rep(5, 4), names=ops))
alignments <- GAlignments(rep("chr1", 4), pos=rep(10, 4),
                          cigar=c("11M", "5M2N4M", "5M2I4M", "5M2D4M"))
pmapFromAlignments(x, alignments)

## ---------------------------------------------------------------------
## B. TATA box motif: mapping from read -> genome -> transcript
## ---------------------------------------------------------------------

## The TATA box motif is a conserved DNA sequence in the core promoter
## region. Many eukaryotic genes have a TATA box located approximately
## 25-35 base pairs upstream of the transcription start site. The motif is
## the binding site of general transcription factors or histones and
## plays a key role in transcription.

## In this example, the position of the TATA box motif (if present) is
## located in the DNA sequence corresponding to read ranges. The local
## motif positions are mapped to genome coordinates and then mapped
## to gene features such as promoters regions.

## Load reads from chromosome 4 of D. melanogaster (dm3):
library(pasillaBamSubset)
fl <- untreated1_chr4()
gal <- readGAlignments(fl)

## Extract DNA sequences corresponding to the read ranges:
library(GenomicFeatures)
library(BSgenome.Dmelanogaster.UCSC.dm3)
dna <- extractTranscriptSeqs(BSgenome.Dmelanogaster.UCSC.dm3, grglist(gal))

## Search for the consensus motif TATAAA in the sequences:
box <- vmatchPattern("TATAAA", dna)

## Some sequences had more than one match:
```

```
table(elementNROWS(box))

## The element-wise function we'll use for mapping to genome coordinates
## requires the two input argument to have the same length. We need to
## replicate the read ranges to match the number of motifs found.

## Expand the read ranges to match motifs found:
motif <- elementNROWS(box) != 0
alignments <- rep(gal[motif], elementNROWS(box)[motif])

## We make the IRanges into a GRanges object so the seqlevels can
## propagate to the output. Seqlevels are needed in the last mapping step.
readCoords <- GRanges(seqnames(alignments), unlist(box, use.names=FALSE))

## Map the local position of the motif to genome coordinates:
genomeCoords <- pmapFromAlignments(readCoords, alignments)
genomeCoords

## We are interested in the location of the TATA box motifs in the
## promoter regions. To perform the mapping we need the promoter ranges
## as a GRanges or GRangesList.

## Extract promoter regions 50 bp upstream from the transcription start site:
library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
promoters <- promoters(txdb, upstream=50, downstream=0)

## Map the genome coordinates to the promoters:
names(promoters) <- mcols(promoters)$tx_name  ## must be named
mapToTranscripts(genomeCoords, promoters)
```

`OverlapEncodings-class` *OverlapEncodings objects*

Description

The OverlapEncodings class is a container for storing the "overlap encodings" returned by the `encodeOverlaps` function.

Usage

```
## -==- OverlapEncodings getters =-=-

## 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)

## -==- Coercing an OverlapEncodings object =-=-

## S4 method for signature 'OverlapEncodings'
as.data.frame(x, row.names=NULL, optional=FALSE, ...)

## -==- Low-level encoding utilities =-=-

encodingHalves(x, single.end.on.left=FALSE, single.end.on.right=FALSE,
                  as.factors=FALSE)
Lencoding(x, ...)
Rencoding(x, ...)

## S4 method for signature 'ANY'
njunc(x)

Lnjunc(x, single.end.on.left=FALSE)
Rnjunc(x, single.end.on.right=FALSE)

isCompatibleWithSplicing(x)
```

Arguments

x	An OverlapEncodings object. For the low-level encoding utilities, x can also be a character vector or factor containing encodings.
row.names	NULL or a character vector.
optional	Ignored.
...	Extra arguments passed to the as.data.frame method for OverlapEncodings objects are ignored. Extra arguments passed to Lencoding or Rencoding are passed down to encodingHalves.
single.end.on.left, single.end.on.right	By default the 2 halves of a single-end encoding are considered to be NAs. If single.end.on.left (resp. single.end.on.right) is TRUE, then the left (resp. right) half of a single-end encoding is considered to be the unmodified encoding.
as.factors	By default encodingHalves returns the 2 encoding halves as a list of 2 character vectors parallel to the input. If as.factors is TRUE, then it returns them as a list of 2 factors parallel to the input.

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. IntegerRangesList 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 this package (**GenomicAlignments**) and accessible with vignette("OverlapEncodings").

OverlapEncodings getters

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 Qi = query[[i]], Si = subject[[i]], and [q1,q2] the range covered by Qi i.e. q1 = min(start(Qi)) and q2 = max(end(Qi)), then Loffset(x)[i] is the number L of ranges at the *head* of Si that are strictly to the left of all the ranges in Qi i.e. L is the greatest value such that end(Si)[k] < q1 - 1 for all k in seq_len(L). Similarly, Roffset(x)[i] is the number R of ranges at the *tail* of Si that are strictly to the right of all the ranges in Qi i.e. R is the greatest value such that start(Si)[length(Si) + 1 - k] > q2 + 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 Qi with all the ranges in tSi = Si[(1+L):(length(Si)-R)] (tSi stands for "trimmed Si"). More precisely, here is how this encoding is obtained:

1. All the ranges in Qi are compared with tSi[1], then with tSi[2], etc... At each step (one step per range in tSi), comparing all the ranges in Qi with tSi[k] is done with rangeComparisonCodeToLetter(compare(Qi, tSi[k])). So at each step, we end up with a vector of M single letters (where M is length(Qi)).
2. Each vector obtained previously (1 vector per range in tSi, 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 tSi) 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.

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".

Low-level encoding utilities

In the following code snippets, x can be an OverlapEncodings object, or a character vector or factor containing encodings.

`encodingHalves(x, single.end.on.left=FALSE, single.end.on.right=FALSE, as.factors=FALSE)`:
Extract the 2 halves of paired-end encodings and return them as a list of 2 character vectors (or 2 factors) parallel to the input.

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

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

Equivalent to `encodingHalves(x, ...)[[1]]` and `encodingHalves(x, ...)[[2]]`, respectively.

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

`isCompatibleWithSplicing(x)`: Returns a logical vector *parallel* to `x` indicating whether the corresponding encoding describes a *splice compatible* overlap i.e. an overlap that is compatible with the splicing of the transcript.

WARNING: For paired-end encodings, `isCompatibleWithSplicing` considers that the encoding is *splice compatible* if its 2 halves are *splice compatible*. This can produce false positives if for example the right end of the alignment is located upstream of the left end in transcript space. The paired-end read could not come from this transcript. To eliminate these false positives, one would need to have access and look at the position of the left and right ends in transcript space. This can be done with `extractQueryStartInTranscript`.

Author(s)

Hervé Pagès

See Also

- The "OverlapEncodings" vignette in this package.
- The `encodeOverlaps` function for computing "overlap encodings".
- The `pcompare` 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

```
## ---------------------------------------------------------------------
## A. BASIC MANIPULATION OF AN OverlapEncodings OBJECT
## ---------------------------------------------------------------------

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))

## ---------------------------------------------------------------------
## B. WORKING WITH PAIRED-END ENCODINGS (POSSIBLY MIXED WITH SINGLE-END
##    ENCODINGS)
## ---------------------------------------------------------------------

encodings <- c("4:jmmm:agmm:aagm:aaaf:", "3--1:jmm--b:agm--i:")

encodingHalves(encodings)
encodingHalves(encodings, single.end.on.left=TRUE)
encodingHalves(encodings, single.end.on.right=TRUE)
encodingHalves(encodings, single.end.on.left=TRUE,
                          single.end.on.right=TRUE)

Lencoding(encodings)
Lencoding(encodings, single.end.on.left=TRUE)
Rencoding(encodings)
Rencoding(encodings, single.end.on.right=TRUE)

njunc(encodings)
Lnjunc(encodings)
Lnjunc(encodings, single.end.on.left=TRUE)
Rnjunc(encodings)
Rnjunc(encodings, single.end.on.right=TRUE)

## ---------------------------------------------------------------------
## C. DETECTION OF "SPLICE COMPATIBLE" OVERLAPS
## ---------------------------------------------------------------------

## Reads that are compatible with the splicing of 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 don't need
## (and should not) use this regular expression, but use instead the
## isCompatibleWithSplicing() utility function:
isCompatibleWithSplicing(ovenc)
```

`pileLettersAt` *Pile the letters of a set of aligned reads on top of a set of genomic positions*

Description

`pileLettersAt` extracts the letters/nucleotides of a set of reads that align to a set of 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 DNAStringSet) object containing N *unaligned* read sequences (a.k.a. the query sequences) reported with respect to the + strand.
`seqnames`	A factor-Rle *parallel* to `x`. For each `i`, `seqnames[i]` must be the name of the reference sequence of the i-th alignment.
`pos`	An integer vector *parallel* 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 *parallel* to `x`. Contains the extended CIGAR strings of the alignments.
`at`	A GPos object containing the genomic positions of interest. `seqlevels(at)` must be identical to `levels(seqnames)`. If `at` is not a GPos object, `pileLettersAt` will first try to turn it into one by calling the `GPos()` constructor function on it. So for example `at` can be a GRanges object (or any other GenomicRanges derivative), and, in that case, each range in it will be interpreted as a run of adjacent genomic positions. See `?GPos` in the **GenomicRanges** package for more information.

Details

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

Value

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

Author(s)

Hervé Pagès

See Also

- The `pileup` and `applyPileups` functions defined in the **Rsamtools** package, as well as the SAMtools mpileup command (available at `http://samtools.sourceforge.net/` as part of the SAMtools project), for more powerful flexible alternatives.
- The `stackStringsFromGAlignments` function for stacking the read sequences (or their quality strings) stored in a GAlignments object or a BAM file.
- DNAStringSet objects in the **Biostrings** package.
- GPos objects in the **GenomicRanges** package.
- GAlignments objects.
- cigar-utils for the CIGAR utility functions used internally by `pileLettersAt`.

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 GPos object containing Genomic Positions Of Interest:
my_GPOI <- GPos(c("seq1:1-5", "seq1:21-21", "seq1:1575-1575",
                  "seq2:1513-1514"))

## Some preliminary massage on 'my_GPOI'

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

## Load the BAM file in a GAlignments object. Note that we load only
## the reads aligned to the sequences in 'seqlevels(my_GPOI)'. Also,
## in order to be consistent with applyPileups() and SAMtools (m)pileup,
## we filter out the following BAM records:
##   - secondary alignments (flag bit 0x100);
##   - reads not passing quality controls (flag bit 0x200);
##   - PCR or optical duplicates (flag bit 0x400).
## See ?ScanBamParam and the SAM Spec for more information.

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

## 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_GPOI)
qual_piles <- pileLettersAt(qual, seqnames(gal), start(gal), cigar(gal),
                            my_GPOI)
mcols(my_GPOI)$nucl_piles <- nucl_piles
mcols(my_GPOI)$qual_piles <- qual_piles
my_GPOI

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

## Note that the pileup() function defined in the Rsamtools package
## can be used to obtain a similar result:
scanbam_param <- ScanBamParam(flag=flag, which=my_GPOI)
```

```
pileup_param <- PileupParam(max_depth=5000,
                            min_base_quality=0,
                            distinguish_strands=FALSE)
pileup(bamfile, scanBamParam=scanbam_param, pileupParam=pileup_param)
```

---

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

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

readGAlignmentPairs(file, index=file, use.names=FALSE, param=NULL,
                          with.which_label=FALSE, strandMode=1)

readGAlignmentsList(file, index=file, use.names=FALSE,
                          param=ScanBamParam(), with.which_label=FALSE,
                          strandMode=NA)

readGappedReads(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 readGAlignments.
index	The path to the index file of the BAM file to read. Must be given *without* the '.bai' extension. See scanBam in the **Rsamtools** packages for more information.
use.names	TRUE or FALSE. 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.
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 *in addition* 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 metadata columns of the object. By default (i.e. param=NULL or param=ScanBamParam()), no additional field is loaded. The flag used is scanBamFlag(isUnmappedQuery=FALSE) for readGAlignments, readGAlignmentsList, and readGappedReads. (i.e. only records corresponding to mapped reads are loaded), and scanBamFlag(isUnmappedQuery=FALSE, isPaired=TRUE, hasUnmappedMate=FALSE) for readGAlignmentPairs (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 param has a which component, a "which_label" metadata column is added to the returned GAlignments or GappedReads object, or to the first and last components of the returned GAlignmentPairs object. In the case of readGAlignmentsList, 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.

strandMode

Strand mode to set on the returned GAlignmentPairs or GAlignmentsList object. Note that the default value for this parameter is different for readGAlignmentPairs() and readGAlignmentsList(). See details below on readGAlignmentsList() and ?strandMode for more information.

Details

- readGAlignments reads a file containing aligned reads as a GAlignments object. See ?GAlignments for a description of GAlignments objects.

  When file is a BamViews object, readGAlignments visits each path in bamPaths(file), returning the result of readGAlignments 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.

- readGAlignmentPairs reads a file containing aligned paired-end reads as a GAlignmentPairs object. See ?GAlignmentPairs for a description of GAlignmentPairs objects.

- readGAlignmentsList reads a file containing aligned reads as a GAlignmentsList object. See ?GAlignmentsList for a description of GAlignmentsList objects. readGAlignmentsList 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':

  – 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 in the **Rsamtools** package for details.

  Note that, by default, strandMode=NA, which is different to the default value in readGAlignmentPairs() and which implies that, by default, the strand values in the returned GAlignmentsList object correspond to the original strand of the reads.

- readGappedReads reads a file containing aligned reads as a GappedReads object. See ?GappedReads for a description of GappedReads objects.

For all these functions, flags, tags and ranges may be specified in the supplied ScanBamParam object for fine tuning of results.

Value

A GAlignments object for `readGAlignments`.

A GAlignmentPairs object for `readGAlignmentPairs`. 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 `readGAlignmentPairs` returns a GAlignmentPairs object of length zero, this almost always 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 `readGAlignmentsList`. 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 `readGappedReads`.

Pairing criteria

This section describes the pairing criteria used by `readGAlignmentsList` and `readGAlignmentPairs`.

- 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 Ox10 (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 `readGAlignmentsList` and `readGAlignmentPairs` call behind the scene. It is also the pairing criteria used by `findMateAlignment`.

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)

Hervé Pagès and Valerie Obenchain

See Also

- `scanBam` and `ScanBamParam` in the **Rsamtools** package.
- GAlignments, GAlignmentPairs, GAlignmentsList, and GappedReads objects.
- IRangesList objects (used in the examples below to specify the `which` regions) in the **IRanges** package.

Examples

```
## ---------------------------------------------------------------------
## A. readGAlignments()
## ---------------------------------------------------------------------

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

## Using the 'use.names' arg:
gal2 <- readGAlignments(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,
                                       isSecondaryAlignment=FALSE),
                      what=c("qual", "flag"))
gal3 <- readGAlignments(bamfile, param=param)
gal3
mcols(gal3)

## Using the 'param' arg to load alignments from particular regions.
which <- IRangesList(seq1=IRanges(1000, 1100),
                     seq2=IRanges(c(1546, 1555, 1567), width=10))
param <- ScanBamParam(which=which)
gal4 <- readGAlignments(bamfile, use.names=TRUE, param=param)
gal4

## IMPORTANT NOTE: A given record is loaded one time for each region
## it overlaps with. We call this "duplicated record selection" (this
## is a scanBam() feature, readGAlignments() is based on scanBam()):
which <- IRangesList(seq2=IRanges(c(1555, 1567), width=10))
param <- ScanBamParam(which=which)
```

```
gal5 <- readGAlignments(bamfile, use.names=TRUE, param=param)
gal5  # record EAS114_26:7:37:79:581 was loaded twice

## This becomes clearer if we use 'with.which_label=TRUE' to identify
## the region in 'which' where each element in 'gal5' originates from.
gal5 <- readGAlignments(bamfile, use.names=TRUE, param=param,
                                 with.which_label=TRUE)
gal5

## Not surprisingly, we also get "duplicated record selection" when
## 'which' contains repeated or overlapping regions. Using the same
## regions as we did for 'gal4' above, except that now we're
## repeating the region on seq1:
which <- IRangesList(seq1=rep(IRanges(1000, 1100), 2),
                     seq2=IRanges(c(1546, 1555, 1567), width=10))
param <- ScanBamParam(which=which)
gal4b <- readGAlignments(bamfile, use.names=TRUE, param=param)
length(gal4b)  # > length(gal4), because all the records overlapping
               # with bases 1000 to 1100 on seq1 are now duplicated

## The "duplicated record selection" will artificially increase the
## coverage or affect other downstream results. It can be mitigated
## (but not completely eliminated) by first "reducing" the set of
## regions:
which <- reduce(which)
which
param <- ScanBamParam(which=which)
gal4c <- readGAlignments(bamfile, use.names=TRUE, param=param)
length(gal4c)  # < length(gal4), because the 2 first original regions
               # on seq2 were merged into a single one

## Note that reducing the set of regions didn't completely eliminate
## "duplicated record selection". Records that overlap the 2 reduced
## regions on seq2 (which$seq2) are loaded twice (like for 'gal5'
## above). See example D. below for how to completely eliminate
## "duplicated record selection".

## 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 <- readGAlignments(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)
## Note that the "readGAlignments" method for BamViews objects
## requires the ShortRead package to be installed.
aln <- readGAlignments(bv)
aln
aln[[1]]
aln[colnames(bv)]
```

```
mcols(aln)

## ---------------------------------------------------------------------
## B. readGAlignmentPairs()
## ---------------------------------------------------------------------
galp1 <- readGAlignmentPairs(bamfile)
head(galp1)
names(galp1)

## Here we use the 'param' arg to filter by proper pair, drop PCR /
## optical duplicates, and drop secondary alignments. Filtering by
## proper pair and dropping secondary alignments can help make the
## pairing algorithm run significantly faster:
param <- ScanBamParam(flag=scanBamFlag(isProperPair=TRUE,
                                       isDuplicate=FALSE,
                                       isSecondaryAlignment=FALSE))
galp2 <- readGAlignmentPairs(bamfile, use.names=TRUE, param=param)
galp2
head(galp2)
head(names(galp2))

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

## 'file' as character.
bam <- untreated3_chr4()
galist1 <- readGAlignmentsList(bam)
galist1[1:3]
length(galist1)
table(elementNROWS(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().
bamfile <- BamFile(bam, yieldSize=3, asMates=TRUE)
readGAlignmentsList(bamfile)

## Use a 'param' to fine tune the results.
param <- ScanBamParam(flag=scanBamFlag(isProperPair=TRUE))
galist2 <- readGAlignmentsList(bam, param=param)
galist2[1:3]
length(galist2)
table(elementNROWS(galist2))

## For paired-end data, we can set the 'strandMode' parameter to
## infer the strand of a pair from the strand of the first and
## last alignments in the pair
galist3 <- readGAlignmentsList(bam, param=param, strandMode=0)
galist3[1:3]
galist4 <- readGAlignmentsList(bam, param=param, strandMode=1)
galist4[1:3]
galist5 <- readGAlignmentsList(bam, param=param, strandMode=2)
galist5[1:3]
```

```
## ---------------------------------------------------------------------
## D. COMPARING 4 STRATEGIES FOR LOADING THE ALIGNMENTS THAT OVERLAP
##    WITH THE EXONIC REGIONS ON FLY CHROMMOSOME 4
## ---------------------------------------------------------------------
library(pasillaBamSubset)
bam <- untreated1_chr4()

library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
ex <- exons(txdb)
seqlevels(ex, pruning.mode="coarse") <- "chr4"
length(ex)

## Some of the exons overlap with each other:
isDisjoint(ex)  # FALSE
exonic_regions <- reduce(ex)
isDisjoint(exonic_regions)  # no more overlaps
length(exonic_regions)

## Strategy #1: slow and loads a lot of records more than once (see
## "duplicated record selection" in example A. above).
param1 <- ScanBamParam(which=ex)
gal1 <- readGAlignments(bam, param=param1)
length(gal1)  # many "duplicated records"

## Strategy #2: faster and generates less duplicated records but
## doesn't eliminate them.
param2 <- ScanBamParam(which=exonic_regions)
gal2 <- readGAlignments(bam, param=param2)
length(gal2)  # less "duplicated records"

## Strategy #3: fast and completely eliminates duplicated records.
gal0 <- readGAlignments(bam)
gal3 <- subsetByOverlaps(gal0, exonic_regions, ignore.strand=TRUE)
length(gal3)  # no "duplicated records"

## Note that, in this case using 'exonic_regions' or 'ex' makes no
## difference:
gal3b <- subsetByOverlaps(gal0, ex, ignore.strand=TRUE)
stopifnot(identical(gal3, gal3b))

## Strategy #4: strategy #3 however can require a lot of memory if the
## file is big because we load all the alignments into memory before we
## select those that overlap with the exonic regions. Strategy #4
## addresses this by loading the file by chunks.
bamfile <- BamFile(bam, yieldSize=50000)
open(bamfile)
while (length(chunk0 <- readGAlignments(bamfile))) {
    chunk <- subsetByOverlaps(chunk0, ex, ignore.strand=TRUE)
    cat("chunk0:", length(chunk0), "- chunk:", length(chunk), "\n")
    ## ... do something with 'chunk' ...
}
close(bamfile)

## ---------------------------------------------------------------------
## E. readGappedReads()
## ---------------------------------------------------------------------
```

```
greads1 <- readGappedReads(bamfile)
greads1
names(greads1)
qseq(greads1)
greads2 <- readGappedReads(bamfile, use.names=TRUE)
head(greads2)
head(names(greads2))
```

---

sequenceLayer *Lay read sequences alongside the reference space, using their CIGARs*

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 alonside 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-clipp
"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)

Hervé Pagès

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 `readGAlignments` 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 <- readGAlignments(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 <- readGAlignments(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("AAAGTTCGAA", "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 <- readGAlignments(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(elementNROWS(qseq2), elementNROWS(rseq2)))

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

I_idx <- which(cigar_op_table[ , "I"] >= 2)  # at least 2 insertions
qseq2[I_idx]
```

```
rseq2[I_idx]

D_idx <- which(cigar_op_table[ , "D"] >= 2)  # at least 2 deletions
qseq2[D_idx]
rseq2[D_idx]

## A closer look at reads with skipped regions:
N_idx <- which(cigar_op_table[ , "N"] != 0)
qseq2[N_idx]
rseq2[N_idx]

## A variant of the "pairwise" space is the "pairwise-dense" space.
## In that space, all indels and skipped regions are removed from 'qseq'
## and 'rseq'.
qseq3 <- sequenceLayer(qseq, cigar(gal),
                       from="query", to="pairwise-dense")
rseq3 <- sequenceLayer(rseq, cigar(gal),
                       from="reference", to="pairwise-dense")

## Sanity check: 'qseq3' and 'rseq3' should have the same shape.
stopifnot(identical(elementNROWS(qseq3), elementNROWS(rseq3)))

## Insertions were removed:
qseq3[I_idx]
rseq3[I_idx]

## Deletions were removed:
qseq3[D_idx]
rseq3[D_idx]

## Skipped regions were removed:
qseq3[N_idx]
rseq3[N_idx]

## ---------------------------------------------------------------------
## D. SANITY CHECKS
## ---------------------------------------------------------------------
SPACES <- c("reference",
            "reference-N-regions-removed",
            "query",
            "query-before-hard-clipping",
            "query-after-soft-clipping",
            "pairwise",
            "pairwise-N-regions-removed",
            "pairwise-dense")

cigarWidth <- list(
    function(cigar) cigarWidthAlongReferenceSpace(cigar),
    function(cigar) cigarWidthAlongReferenceSpace(cigar,
                                                  N.regions.removed=TRUE),
    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="++AAAABBiiCDDD", B="AAAAAiBBBCCCC++")),
    BStringSet(c(A="+++++AAAABBiiCDDD++++++", B="AAAAAiBBBCCCC++")),
    BStringSet(c(A="AAAABBiiCDDD", B="AAAAAiBBBCCCC")),
    BStringSet(c(A="AAAA-BBiiC.....DDD", B="AAAAAiBBB--CCCC")),
    BStringSet(c(A="AAAA-BBiiCDDD", 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))
```

---

setops-methods *Set operations on GAlignments objects*

Description

Performs set operations on GAlignments objects.

NOTE: The pintersect generic function and method for IntegerRanges objects is 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,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 GAlignments object or a BAM file*

Description

stackStringsFromGAlignments stacks the read sequences (or their quality strings) stored in a GAlignments object over a user-specified region.

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.

All these 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

```
stackStringsFromGAlignments(x, region, what="seq",
                    D.letter="-", N.letter=".",
                    Lpadding.letter="+",
                    Rpadding.letter="+")

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

`x`	A GAlignments object with the read sequences in the `"seq"` metadata column (if `what` is set to `"seq"`), or with the the read quality strings in the `"qual"` metadata column (if `what` is set to `"qual"`). Such an object is typically obtained by specifying `param=ScanBamParam(what=c("seq", "qual"))` when reading a BAM file with calling `readGAlignments()`.
`region`	A GRanges object with exactly 1 genomic range. The read sequences (or read quality strings) will be stacked over that region.
`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.
`D.letter`, `N.letter`	A single letter used as a filler for injections. The 2 arguments are passed down to the `sequenceLayer` function. See `?sequenceLayer` 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 `?stackStrings` in the **Biostrings** package for more details.
`file`, `index`	The path to the BAM file containing the reads, and to its index file, respectively. The latter is given *without* the '.bai' extension. See `scanBam` for more information.
`param`	A ScanBamParam object containing exactly 1 genomic region (i.e. `unlist(bamWhich(param))` must have length 1). Alternatively, `param` can be a GRanges or IntegerRangesList 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).
`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.
`...`	Further arguments to be passed to alphabetFrequency.

Details

`stackStringsFromGAlignments` performs the 3 following steps:

1. Subset GAlignments object `x` to keep only the alignments that overlap with the specified region.
2. Lay the sequences in `x` 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.

`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 readGAlignments function. Note that if the file contains paired-end reads, the pairing is ignored.
2. Same as stackStringsFromGAlignments.
3. Same as stackStringsFromGAlignments.

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 stackStringsFromGAlignments AND stackStringsFromBam:

Specifying a big genomic region, say >= 100000 bp, can require a lot of memory (especially with high coverage reads) so is not recommended. See the pileLettersAt function for piling the read letters on top of a set of 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)

Hervé Pagès

See Also

- The pileLettersAt function for piling the letters of a set of aligned reads on top of a set of genomic positions.
- The readGAlignments 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 directly from the BAM file:
stackStringsFromBam(bamfile1, param=region1, use.names=TRUE)

## or, alternatively, from a GAlignments object:
gal1 <- readGAlignments(bamfile1, param=ScanBamParam(what="seq"),
                        use.names=TRUE)
stackStringsFromGAlignments(gal1, 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))

## Stack the read sequences directly from the BAM file:
stackStringsFromBam(bamfile2, param=region2)

## or, alternatively, from a GAlignments object:
gal2 <- readGAlignments(bamfile2, param=ScanBamParam(what="seq"))
stackStringsFromGAlignments(gal2, 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
## ---------------------------------------------------------------------

## Let's 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=Union,
  ignore.strand=FALSE, inter.feature=TRUE, preprocess.reads=NULL, ...)

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

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

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

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

## S4 method for signature 'GRangesList,GAlignmentPairs'
summarizeOverlaps(
  features, reads, mode=Union,
  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=Union,
  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=Union,
  ignore.strand=FALSE, inter.feature=TRUE, singleEnd=TRUE,
  fragments=FALSE, param=ScanBamParam(), preprocess.reads=NULL, ...)
```

Arguments

features	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. When features is a BamViews the reads argument is missing. Features are extracted from the bamRanges and the reads from bamPaths. Metadata from bamPaths and bamSamples are stored in the colData of the resulting RangedSummarizedExperiment object. bamExperiment metadata are stored in the metadata slot.

`reads` A GRanges, GRangesList GAlignments, GAlignmentsList, GAlignmentPairs, BamViews or BamFileList object that represents the data to be counted by `summarizeOverlaps`.

`reads` is missing when a BamViews object is the only argument supplied to `summarizeOverlaps`. `reads` are the files specified in `bamPaths` of the BamViews object.

`mode` `mode` can be one of the pre-defined count methods such as "Union", "IntersectionStrict", or "IntersectionNotEmpty" or it 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.

The pre-defined options are designed after the counting modes available in the HTSeq package by Simon Anders (see references).

- "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`, `ignore.strand` and `inter.feature` arguments (as in the defined mode functions) and (2) return a vector of counts the same length as `features`.

`ignore.strand` A logical indicating if strand should be considered when matching.

`inter.feature` (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.

There are 6 possible combinations of the `mode` and `inter.feature` arguments. When `inter.feature=FALSE` 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.

`preprocess.reads`

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.

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.

`...` 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`.

A `BPPARAM` argument can be passed down to the `bplapply` called by `summarizeOverlaps`. The argument can be MulticoreParam(), SnowParam(), BatchJobsParam() or DoparParam(). See the BiocParallel package for details in specifying the params.

`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; applied to paired-end data only. `fragments` controls which function is used to read the data which subsequently affects which records are included in counting.

When `fragments=FALSE`, data are read with `readGAlignmentPairs` and returned in a `GAlignmentPairs` class. In this case, singletons, reads with unmapped pairs, and other fragments, are dropped.

When `fragments=TRUE`, data are read with `readGAlignmentsList` and returned in a `GAlignmentsList` class. This class holds 'mated pairs' as well as same-strand pairs, singletons, reads with unmapped pairs and other fragments. Because more records are kept, generally counts will be higher when `fragments=TRUE`.

The term 'mated pairs' refers to records paired with the algorithm described on the ?`readGAlignmentsList` man page.

`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; reads that overlap 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 RangedSummarizedExperiment object. The `assays` slot holds the counts, `rowRanges` 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

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 **DESeq2**, **DEXSeq** and **edgeR** packages.
- The RangedSummarizedExperiment class defined in the **SummarizedExperiment** package.
- 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(c("chr1", "chr2"), 3), "chr1")),
    pos = 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))
grl <- splitAsList(gr, groups)
names(grl) <- LETTERS[seq_along(grl)]

## ---------------------------------------------------------------------
## 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(grl, reads))$counts,
               intStrict=assays(summarizeOverlaps(grl, reads,
                                mode="IntersectionStrict"))$counts,
               intNotEmpty=assays(summarizeOverlaps(grl, reads,
                                  mode="IntersectionNotEmpty"))$counts)
colSums(counts2)

## The GRangesList ('grl' 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,
grl["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)

## ---------------------------------------------------------------------
## Use ouput of summarizeOverlaps() for differential expression analysis
## with DESeq2 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)

## Start differential expression analysis with the DESeq2 or edgeR
## package:
library(DESeq2)
deseq <- DESeqDataSet(se, design= ~ 1)
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, 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(grl, reads, mode=Union, preprocess.reads=ResizeReads)

## Count reads that overlap on the 3' end by passing new values
## for 'width' and 'fix':
summarizeOverlaps(grl, 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 metadata 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)
metadata(se)
```

Index