

Package ‘GenomicRanges’

October 9, 2013

Title Representation and manipulation of genomic intervals

Description The ability to efficiently store genomic annotations and alignments is playing a central role when it comes to analyze high-throughput sequencing data (a.k.a. NGS data). The package defines general purpose containers for storing genomic intervals as well as more specialized containers for storing alignments against a reference genome.

Version 1.12.5

Author P. Aboyoun, H. Pages and M. Lawrence

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

biocViews Genetics, Sequencing, HighThroughputSequencing, Annotation

Depends R (>= 2.10), methods, BiocGenerics (>= 0.5.4), IRanges (>= 1.17.33)

Imports methods, utils, stats, BiocGenerics, IRanges

LinkingTo IRanges

Suggests AnnotationDbi (>= 1.21.1), Biostrings (>= 2.25.3), Rsamtools (>= 1.11.24), BSgenome, rtracklayer, GenomicFeatures, VariantAnnotation, edgeR, DESeq, DEXSeq, EatonEtAlChIPseq (>= 0.0.3), leeBamViews, pasilla, pasillaBamSubset, org.Sc.sgd.db, TxDb.Dmelanogaster.UCSC.dm3.ensGene, seqnames.db, BSgenome.Scerevisiae.UCSC.sacCer2, BSgenome.Dmelanogaster.UCSC.dm3, RUnit, digest

License Artistic-2.0

Collate utils.R phicoef.R cigar-utils.R transcript-utils.R constraint.R makeSeqnameIds.R seqinfo.R strand-utils.R Seqinfo-class.R GenomicRanges-class.R GRanges-class.R GenomicRanges-comparison.R GenomicRangesList-class.R GRangesList-class.R GappedAlignments-class.R GAlignmentsList-class.R GappedAlignmentPairs-class.R SummarizedExperiment-class.R SummarizedExperiment-rowData-methods.R seqlevels-utils.R resolveHits-methods.R summarizeOverlaps.R

RangesMapping-methods.R RangedData-methods.R
 intra-range-methods.R inter-range-methods.R setops-methods.R
 findOverlaps-methods.R nearest-methods.R encodeOverlaps-methods.R coverage-methods.R
 findSpliceOverlaps-methods.R findSpliceOverlaps-utils.R
 test_GenomicRanges_package.R zzz.R

R topics documented:

cigar-utils	2
Constraints	7
coverage-methods	14
encodeOverlaps-methods	16
findOverlaps-methods	18
findSpliceOverlaps	22
GAlignmentsList-class	25
GappedAlignmentPairs-class	29
GappedAlignments-class	34
GenomicRanges-comparison	40
GenomicRangesList-class	43
GRanges-class	44
GRangesList-class	49
inter-range-methods	55
intra-range-methods	58
makeSeqnameIds	61
map-methods	62
nearest-methods	63
phicoef	66
seqinfo	67
Seqinfo-class	69
setops-methods	72
strand-utils	75
SummarizedExperiment-class	77
summarizeOverlaps	83
utils	87
Index	90

cigar-utils

CIGAR utility functions

Description

Utility functions for low-level CIGAR manipulation.

Usage

```

cigarOpTable(cigar)

cigarToQWidth(cigar, before.hard.clipping=FALSE)
cigarToWidth(cigar)

cigarQNarrow(cigar, start=NA, end=NA, width=NA)
cigarNarrow(cigar, start=NA, end=NA, width=NA)

cigarToIRanges(cigar,
               drop.D.ranges=FALSE, drop.empty.ranges=FALSE,
               reduce.ranges=TRUE)

cigarToIRangesListByAlignment(cigar, pos, flag=NULL,
                              drop.D.ranges=FALSE, drop.empty.ranges=FALSE,
                              reduce.ranges=TRUE)

cigarToIRangesListByRName(cigar, rname, pos, flag=NULL,
                          drop.D.ranges=FALSE, drop.empty.ranges=FALSE,
                          reduce.ranges=TRUE)

queryLoc2refLoc(qloc, cigar, pos=1)
queryLocs2refLocs(qlocs, cigar, pos, flag=NULL)

splitCigar(cigar)
cigarToRleList(cigar)
cigarToCigarTable(cigar)
summarizeCigarTable(x)

```

Arguments

cigar	A character vector/factor containing the extended CIGAR string for each read. For <code>cigarToIRanges</code> and <code>queryLoc2refLoc</code> , this must be a single string (i.e. a character vector/factor of length 1).
before.hard.clipping	Should the returned widths be the lengths of the reads before or after "hard clipping"? Hard clipping of a read is encoded with an H in the CIGAR. If NO (<code>before.hard.clipping=FALSE</code> , the default), then the returned widths are the lengths of the query sequences stored in the SAM/BAM file. If YES (<code>before.hard.clipping=TRUE</code>), then the returned widths are the lengths of the original reads.
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).
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 <code>drop.D.ranges</code> is TRUE, then Ds and Ns in the CIGAR are equivalent.

drop.empty.ranges	Should empty ranges be dropped?
reduce.ranges	Should adjacent ranges coming from the same cigar be merged or not? Using TRUE (the default) can significantly reduce the size of the returned object.
pos	An integer vector containing the 1-based leftmost position/coordinate for each (eventually clipped) read sequence.
flag	NULL or an integer vector containing the SAM flag for each read. According to the SAM specs, flag bit 0x004 has the following meaning: when bit 0x004 is ON then "the query sequence itself is unmapped". When flag is provided, <code>cigarToIRangesListByAlignment</code> and <code>cigarToIRangesListByRName</code> ignore these reads.
rname	A character vector/factor containing the name of the reference sequence associated with each read (i.e. the name of the sequence the read has been aligned to).
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.
x	A DataFrame produced by <code>cigarToCigarTable</code> .

Value

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

For `cigarToQWidth`: An integer vector of the same length as `cigar` where each element is the width of the query (i.e. the length of the query sequence) as inferred from the corresponding element in `cigar` (NAs in `cigar` will produce NAs in the returned vector).

For `cigarQNarrow` and `cigarNarrow`: 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 `cigarToWidth`: An integer vector of the same length as `cigar` where each element is the width of the alignment (i.e. its total length on the reference, gaps included) as inferred from the corresponding element in `cigar` (NAs in `cigar` will produce NAs in the returned vector).

For `cigarToIRanges`: An [IRanges](#) object describing where the bases in the read align with respect to an imaginary reference sequence assuming that the leftmost aligned base is at position 1 in the reference (i.e. at the first position).

For `cigarToIRangesListByAlignment`: A [CompressedIRangesList](#) object of the same length as `cigar`.

For `cigarToIRangesListByRName`: A named [IRangesList](#) object with one element ([IRanges](#)) per unique reference sequence.

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.

For splitCigar: A list of the same length as cigar where each element is itself a list with 2 elements of the same lengths, the 1st one being a raw vector containing the CIGAR operations and the 2nd one being an integer vector containing the lengths of the CIGAR operations.

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

For cigarToCigarTable: A frequency table of the CIGARs in the form of a [DataFrame](#) with two columns: cigar (a [CompressedRleList](#)) and count (an integer).

For summarizeCigarTable: A list with two elements: AlignedCharacters (integer) and Indels (matrix)

Author(s)

H. Pages and P. Aboyoun

References

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

See Also

[IRanges-class](#), [IRangesList-class](#), [coverage](#), [RleList-class](#)

Examples

```
## -----
## A. SIMPLE EXAMPLES
## -----

## With a cigar vector of length 1:
cigar1 <- "3H15M55N4M2I6M2D5M6S"

## cigarToQWidth()/cigarToWidth():
cigarToQWidth(cigar1)
cigarToQWidth(cigar1, before.hard.clipping=TRUE)
cigarToWidth(cigar1)

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

## 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)

## cigarToIRanges():
cigarToIRanges(cigar1)
cigarToIRanges(cigar1, reduce.ranges=FALSE)
cigarToIRanges(cigar1, drop.D.ranges=TRUE)

## With a cigar vector of length 4:
cigar2 <- c("40M", cigar1, "2S10M2000N15M", "3H25M5H")
pos <- c(1, 1001, 1, 351)
cigarToIRangesListByAlignment(cigar2, pos)
rname <- c("chr6", "chr6", "chr2", "chr6")
cigarToIRangesListByRName(cigar2, rname, pos)

cigarOpTable(cigar2)

splitCigar(cigar2)
cigarToRleList(cigar2)

cigarToCigarTable(cigar2)
cigarToCigarTable(cigar2)[,"cigar"]
cigarToCigarTable(cigar2)[,"count"]

summarizeCigarTable(cigarToCigarTable(cigar2))

## -----
## B. 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 <- sample(some_fake_rnames, nreads, replace=TRUE)
  pos <- sample(80000000L, nreads, replace=TRUE)

  ## The following takes < 5 sec. to complete:
  system.time(rglist <- cigarToIRangesListByAlignment(cigar3, pos))
}

```

```

## The following takes < 10 sec. to complete:
system.time(irl <- cigarToIRangesListByRName(cigar3, rname, pos))

## Internally, cigarToIRangesListByRName() turns 'rname' into a factor
## before starting the calculation. Hence it will run slightly
## faster if 'rname' is already a factor.
rname2 <- as.factor(rname)
system.time(irl2 <- cigarToIRangesListByRName(cigar3, rname2, pos))

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

## -----
## 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.

extractCoverageFromBAM <- function(file)
{
  ## This ScanBamParam object allows us to load only the necessary
  ## information from the file.
  param <- ScanBamParam(flag=scanBamFlag(isUnmappedQuery=FALSE,
                                         isDuplicate=FALSE),
                       what=c("rname", "pos", "cigar"))
  bam <- scanBam(file, param=param)[[1]]
  ## Note that unmapped reads and reads that are PCR/optical duplicates
  ## have already been filtered out by using the ScanBamParam object above.
  irl <- cigarToIRangesListByRName(bam$cigar, bam$rname, bam$pos)
  irl <- irl[elementLengths(irl) != 0] # drop empty elements
  coverage(irl)
}

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

```

Description

Attaching a Constraint object to an object of class A (the "constrained" object) is meant to be a convenient/reusable/extensible way to enforce a particular set of constraints on particular instances of A.

THIS IS AN EXPERIMENTAL FEATURE AND STILL VERY MUCH A WORK-IN-PROGRESS!

Details

For the developer, using constraints is an alternative to the more traditional approach that consists in creating subclasses of A and implementing specific validity methods for each of them. However, using constraints offers the following advantages over the traditional approach:

- The traditional approach often tends to lead to a proliferation of subclasses of A.
- Constraints can easily be re-used across different classes without the need to create any new class.
- Constraints can easily be combined.

All constraints are implemented as concrete subclasses of the Constraint class, which is a virtual class with no slots. Like the Constraint virtual class itself, concrete Constraint subclasses cannot have slots.

Here are the 7 steps typically involved in the process of putting constraints on objects of class A:

1. Add a slot named `constraint` to the definition of class A. The type of this slot must be `ConstraintORNULL`. Note that any subclass of A will inherit this slot.
2. Implements the `constraint()` accessors (getter and setter) for objects of class A. This is done by implementing the `"constraint"` method (getter) and replacement method (setter) for objects of class A (see the examples below). As a convenience to the user, the setter should also accept the name of a constraint (i.e. the name of its class) in addition to an instance of that class. Note that those accessors will work on instances of any subclass of A.
3. Modify the validity method for class A so it also returns the result of `checkConstraint(x, constraint(x))` (append this result to the result returned by the validity method).
4. Testing: Create `x`, an instance of class A (or subclass of A). By default there is no constraint on it (`constraint(x)` is `NULL`). `validObject(x)` should return `TRUE`.
5. Create a new constraint (`MyConstraint`) by extending the `Constraint` class, typically with `setClass("MyConstraint", contains="Constraint")`. This constraint is not enforcing anything yet so you could put it on `x` (with `constraint(x) <- "MyConstraint"`), but not much would happen. In order to actually enforce something, a `"checkConstraint"` method for signature `c(x="A", constraint="MyConstraint")` needs to be implemented.
6. Implement a `"checkConstraint"` method for signature `c(x="A", constraint="MyConstraint")`. Like validity methods, `"checkConstraint"` methods must return `NULL` or a character vector describing the problems found. Like validity methods, they should never fail (i.e. they should never raise an error). Note that, alternatively, an existing constraint (e.g. `SomeConstraint`) can be adapted to work on objects of class A by just defining a new `"checkConstraint"` method for signature `c(x="A", constraint="SomeConstraint")`. Also, stricter constraints can be built on top of existing constraints by extending one or more existing constraints (see the examples below).

7. Testing: Try `constraint(x) <- "MyConstraint"`. It will or will not work depending on whether `x` satisfies the constraint or not. In the former case, trying to modify `x` in a way that breaks the constraint on it will also raise an error.

Note

WARNING: This note is not true anymore as the constraint slot has been temporarily removed from `GenomicRanges` objects (starting with package `GenomicRanges` \geq 1.7.9).

Currently, only `GenomicRanges` objects can be constrained, that is:

- they have a constraint slot;
- they have `constraint()` accessors (getter and setter) for this slot;
- their validity method has been modified so it also returns the result of `checkConstraint(x, constraint(x))`.

More classes in the `GenomicRanges` and `IRanges` packages will support constraints in the near future.

Author(s)

H. Pages

See Also

[setClass](#), [is](#), [setMethod](#), [showMethods](#), [validObject](#), [GenomicRanges-class](#)

Examples

```
## The examples below show how to define and set constraints on
## GenomicRanges objects. Note that this is how the constraint()
## setter is defined for GenomicRanges objects:
#setReplaceMethod("constraint", "GenomicRanges",
#  function(x, value)
#  {
#    if (isSingleString(value))
#      value <- new(value)
#    if (!is(value, "ConstraintORNULL"))
#      stop("the supplied 'constraint' must be a ",
#           "Constraint object, a single string, or NULL")
#    x@constraint <- value
#    validObject(x)
#    x
#  }
#)

#selectMethod("constraint", "GenomicRanges") # the getter
#selectMethod("constraint<-", "GenomicRanges") # the setter

## We'll use the GRanges instance 'gr' created in the GRanges examples
## to test our constraints:
example(GRanges, echo=FALSE)
gr
```

```

#constraint(gr)

## -----
## EXAMPLE 1: The HasRangeTypeCol constraint.
## -----
## The HasRangeTypeCol constraint checks that the constrained object
## has a unique "rangeType" metadata column and that this column
## is a 'factor' Rle with no NAs and with the following levels
## (in this order): gene, transcript, exon, cds, 5utr, 3utr.

setClass("HasRangeTypeCol", contains="Constraint")

## Like validity methods, "checkConstraint" methods must return NULL or
## a character vector describing the problems found. They should never
## fail i.e. they should never raise an error.
setMethod("checkConstraint", c("GenomicRanges", "HasRangeTypeCol"),
  function(x, constraint, verbose=FALSE)
  {
    x_mcols <- mcols(x)
    idx <- match("rangeType", colnames(x_mcols))
    if (length(idx) != 1L || is.na(idx)) {
      msg <- c("'mcols(x)' must have exactly 1 column ",
              "named \"rangeType\"")
      return(paste(msg, collapse=""))
    }
    rangeType <- x_mcols[[idx]]
    .LEVELS <- c("gene", "transcript", "exon", "cds", "5utr", "3utr")
    if (!is(rangeType, "Rle") ||
        IRanges::anyMissing(runValue(rangeType)) ||
        !identical(levels(rangeType), .LEVELS))
    {
      msg <- c("'mcols(x)$rangeType' must be a ",
              "'factor' Rle with no NAs and with levels: ",
              paste(.LEVELS, collapse=", "))
      return(paste(msg, collapse=""))
    }
    NULL
  }
)

#\dontrun{
#constraint(gr) <- "HasRangeTypeCol" # will fail
#}
checkConstraint(gr, new("HasRangeTypeCol")) # with GenomicRanges >= 1.7.9

levels <- c("gene", "transcript", "exon", "cds", "5utr", "3utr")
rangeType <- Rle(factor(c("cds", "gene"), levels=levels), c(8, 2))
mcols(gr)$rangeType <- rangeType
#constraint(gr) <- "HasRangeTypeCol" # OK
checkConstraint(gr, new("HasRangeTypeCol")) # with GenomicRanges >= 1.7.9

## Use is() to check whether the object has a given constraint or not:
#is(constraint(gr), "HasRangeTypeCol") # TRUE

```

```

#\\dontrun{
#mcols(gr)$rangeType[3] <- NA # will fail
#}
mcols(gr)$rangeType[3] <- NA
checkConstraint(gr, new("HasRangeTypeCol")) # with GenomicRanges >= 1.7.9

## -----
## EXAMPLE 2: The GeneRanges constraint.
## -----
## The GeneRanges constraint is defined on top of the HasRangeTypeCol
## constraint. It checks that all the ranges in the object are of type
## "gene".

setClass("GeneRanges", contains="HasRangeTypeCol")

## The checkConstraint() generic will check the HasRangeTypeCol constraint
## first, and, only if it's satisfied, it will then check the GeneRanges
## constraint.
setMethod("checkConstraint", c("GenomicRanges", "GeneRanges"),
  function(x, constraint, verbose=FALSE)
  {
    rangeType <- mcols(x)$rangeType
    if (!all(rangeType == "gene")) {
      msg <- c("all elements in 'mcols(x)$rangeType' ",
              "must be equal to \"gene\"")
      return(paste(msg, collapse=""))
    }
    NULL
  }
)

#\\dontrun{
#constraint(gr) <- "GeneRanges" # will fail
#}
checkConstraint(gr, new("GeneRanges")) # with GenomicRanges >= 1.7.9

mcols(gr)$rangeType[] <- "gene"
## This replace the previous constraint (HasRangeTypeCol):
#constraint(gr) <- "GeneRanges" # OK
checkConstraint(gr, new("GeneRanges")) # with GenomicRanges >= 1.7.9

#is(constraint(gr), "GeneRanges") # TRUE
## However, 'gr' still indirectly has the HasRangeTypeCol constraint
## (because the GeneRanges constraint extends the HasRangeTypeCol
## constraint):
#is(constraint(gr), "HasRangeTypeCol") # TRUE
#\\dontrun{
#mcols(gr)$rangeType[] <- "exon" # will fail
#}
mcols(gr)$rangeType[] <- "exon"
checkConstraint(gr, new("GeneRanges")) # with GenomicRanges >= 1.7.9

## -----

```

```

## EXAMPLE 3: The HasGCCol constraint.
## -----
## The HasGCCol constraint checks that the constrained object has a
## unique "GC" metadata column, that this column is of type numeric,
## with no NAs, and that all the values in that column are >= 0 and <= 1.

setClass("HasGCCol", contains="Constraint")

setMethod("checkConstraint", c("GenomicRanges", "HasGCCol"),
  function(x, constraint, verbose=FALSE)
  {
    x_mcols <- mcols(x)
    idx <- match("GC", colnames(x_mcols))
    if (length(idx) != 1L || is.na(idx)) {
      msg <- c("'mcols(x)' must have exactly ",
              "one column named 'GC'")
      return(paste(msg, collapse=""))
    }
    GC <- x_mcols[[idx]]
    if (!is.numeric(GC) ||
        IRanges::anyMissing(GC) ||
        any(GC < 0) || any(GC > 1))
    {
      msg <- c("'mcols(x)$GC' must be a numeric vector ",
              "with no NAs and with values between 0 and 1")
      return(paste(msg, collapse=""))
    }
    NULL
  }
)

## This replace the previous constraint (GeneRanges):
#constraint(gr) <- "HasGCCol" # OK
checkConstraint(gr, new("HasGCCol")) # with GenomicRanges >= 1.7.9

#is(constraint(gr), "HasGCCol") # TRUE
#is(constraint(gr), "GeneRanges") # FALSE
#is(constraint(gr), "HasRangeTypeCol") # FALSE

## -----
## EXAMPLE 4: The HighGCRanges constraint.
## -----
## The HighGCRanges constraint is defined on top of the HasGCCol
## constraint. It checks that all the ranges in the object have a GC
## content >= 0.5.

setClass("HighGCRanges", contains="HasGCCol")

## The checkConstraint() generic will check the HasGCCol constraint
## first, and, if it's satisfied, it will then check the HighGCRanges
## constraint.
setMethod("checkConstraint", c("GenomicRanges", "HighGCRanges"),
  function(x, constraint, verbose=FALSE)

```

```

    {
      GC <- mcols(x)$GC
      if (!all(GC >= 0.5)) {
        msg <- c("all elements in 'mcols(x)$GC' ",
                "must be >= 0.5")
        return(paste(msg, collapse=""))
      }
      NULL
    }
  )

#\\dontrun{
#constraint(gr) <- "HighGCRanges" # will fail
#}
checkConstraint(gr, new("HighGCRanges")) # with GenomicRanges >= 1.7.9
mcols(gr)$GC[6:10] <- 0.5
#constraint(gr) <- "HighGCRanges" # OK
checkConstraint(gr, new("HighGCRanges")) # with GenomicRanges >= 1.7.9

#is(constraint(gr), "HighGCRanges") # TRUE
#is(constraint(gr), "HasGCCol") # TRUE

## -----
## EXAMPLE 5: The HighGCCGeneRanges constraint.
## -----
## The HighGCCGeneRanges constraint is the combination (AND) of the
## GeneRanges and HighGCRanges constraints.

setClass("HighGCCGeneRanges", contains=c("GeneRanges", "HighGCRanges"))

## No need to define a method for this constraint: the checkConstraint()
## generic will automatically check the GeneRanges and HighGCRanges
## constraints.

#constraint(gr) <- "HighGCCGeneRanges" # OK
checkConstraint(gr, new("HighGCCGeneRanges")) # with GenomicRanges >= 1.7.9

#is(constraint(gr), "HighGCCGeneRanges") # TRUE
#is(constraint(gr), "HighGCRanges") # TRUE
#is(constraint(gr), "HasGCCol") # TRUE
#is(constraint(gr), "GeneRanges") # TRUE
#is(constraint(gr), "HasRangeTypeCol") # TRUE

## See how all the individual constraints are checked (from less
## specific to more specific constraints):
#checkConstraint(gr, constraint(gr), verbose=TRUE)
checkConstraint(gr, new("HighGCCGeneRanges"), verbose=TRUE) # with
# GenomicRanges
# >= 1.7.9

## See all the "checkConstraint" methods:
showMethods("checkConstraint")

```

coverage-methods	<i>Coverage of a GRanges, GRangesList, GappedAlignments, or GappedAlignmentPairs object</i>
------------------	---

Description

[coverage](#) methods for [GRanges](#), [GRangesList](#), [GappedAlignments](#), and [GappedAlignmentPairs](#) objects.

Usage

```
## S4 method for signature 'GenomicRanges'
coverage(x, shift=0L, width=NULL, weight=1L, ...)
## S4 method for signature 'GappedAlignments'
coverage(x, shift=0L, width=NULL,
         weight=1L, drop.D.ranges=FALSE, ...)
## S4 method for signature 'GappedAlignmentPairs'
coverage(x, shift=0L, width=NULL,
         weight=1L, drop.D.ranges=FALSE, ...)
```

Arguments

x	A GRanges , GRangesList , GappedAlignments , or GappedAlignmentPairs object.
shift, width, weight, ...	See coverage in the IRanges package for a description of these optional arguments.
drop.D.ranges	Whether the coverage calculation should ignore ranges corresponding to D (deletion) in the CIGAR string.

Details

Here is how optional arguments `shift`, `width` and `weight` are handled when `x` is a [GRanges](#) object:

- `shift`, `weight`: can be either a numeric vector (integers) or a list. If a list, then it should be named by the sequence levels in `x` (i.e. by the names of the underlying sequences), and its elements are passed into the coverage method for [IRanges](#) objects. If a numeric vector, then it is first recycled to the length of `x`, then turned into a list with `split(shift, as.factor(seqnames(x)))`, and finally the elements of this list are passed into the coverage method for [IRanges](#) objects. Finally, if `x` is a [GRanges](#) object, then `weight` can also be a single string naming a metadata column to be used as the weights.
- `width`: can be either `NULL` or a numeric vector. If a numeric vector, then it should be named by the sequence levels in `x`. If `NULL` (the default), then it is replaced with `seqlengths(x)`. Like for `shift` and `weight`, its elements are passed into the coverage method for [IRanges](#) objects (if the element is `NA` then `NULL` is passed instead).

When `x` is a [GRangesList](#) object, `coverage(x, ...)` is equivalent to `coverage(unlist(x), ...)`.

When `x` is a [GappedAlignments](#) or [GappedAlignmentPairs](#) object, `coverage(x, ...)` is equivalent to `coverage(as(x, "GRangesList"), ...)`.

Value

Returns a named [RleList](#) object with one element ('integer' Rle) per underlying sequence in `x` representing how many times each position in the sequence is covered by the intervals in `x`.

Author(s)

P. Abouyou and H. Pages

See Also

- [coverage](#).
- [RleList-class](#).
- [GRanges-class](#).
- [GRangesList-class](#).
- [GappedAlignments-class](#).
- [GappedAlignmentPairs-class](#).

Examples

```
## Coverage of a GRanges object:
gr <- GRanges(
  seqnames=Rle(c("chr1", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
  ranges=IRanges(1:10, end=10),
  strand=Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
  seqlengths=c(chr1=11, chr2=12, chr3=13))
cvg <- coverage(gr)
pcvg <- coverage(gr[strand(gr) == "+"])
mcvg <- coverage(gr[strand(gr) == "-"])
scvg <- coverage(gr[strand(gr) == "*"])
stopifnot(identical(pcvg + mcvg + scvg, cvg))

## Coverage of a GRangesList object:
gr1 <- GRanges(seqnames="chr2",
  ranges=IRanges(3, 6),
  strand = "+")
gr2 <- GRanges(seqnames=c("chr1", "chr1"),
  ranges=IRanges(c(7,13), width=3),
  strand=c("+", "-"))
gr3 <- GRanges(seqnames=c("chr1", "chr2"),
  ranges=IRanges(c(1, 4), c(3, 9)),
  strand=c("-", "-"))
grl <- GRangesList(gr1=gr1, gr2=gr2, gr3=gr3)
stopifnot(identical(coverage(grl), coverage(unlist(grl))))

## Coverage of a GappedAlignments or GappedAlignmentPairs object:
```

```

library(Rsamtools) # because file ex1.bam is in this package
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
galn <- readGappedAlignments(ex1_file)
stopifnot(identical(coverage(galn), coverage(as(galn, "GRangesList"))))
galp <- readGappedAlignmentPairs(ex1_file)
stopifnot(identical(coverage(galp), coverage(as(galp, "GRangesList"))))

```

encodeOverlaps-methods

encodeOverlaps method for GRangesList objects, and related utilities

Description

encodeOverlaps method for [GRangesList](#), and related utilities.

Usage

```

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

## Low-level utils:

flipQuery(x, i)

selectEncodingWithCompatibleStrand(ovencA, ovencB,
                                   query.strand, subject.strand, hits=NULL)

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

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

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

## High-level convenience wrappers:

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

```

Arguments

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

- For `isCompatibleWithSplicing`, `isCompatibleWithSkippedExons`, `extractSteppedExonRanks`, `extractSpannedExonRanks`, and `extractSkippedExonRanks`: an [OverlapEncodings](#) object, a factor or a character vector.
- `i` Subscript specifying the elements in `x` to flip. If missing, all the elements are flipped.
- `ovencA`, `ovencB`, `ovenc` [OverlapEncodings](#) objects.
- `query`, `subject` [GRangesList](#) objects except for `findCompatibleOverlaps` and `countCompatibleOverlaps` where `query` must be a [GappedAlignments](#) or [GappedAlignmentPairs](#) object.
- `hits` A [Hits](#) object. See ?'encodeOverlaps' for a description of how a supplied [Hits](#) object is handled.
- `flip.query.if.wrong.strand`
See the "Overlap encodings" vignette in the [GenomicRanges](#) package.
- `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

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.

The topic of working with overlap encodings is covered in details in the "Overlap encodings" vignette in the [GenomicRanges](#) package.

Author(s)

H. Pages

See Also

- The "Overlap encodings" vignette in the [GenomicRanges](#) package.
- The [findOverlaps](#) generic function defined in the [IRanges](#) package.
- The [OverlapEncodings](#) class defined in the [IRanges](#) package.
- The [GRangesList](#), [GappedAlignments](#), and [GappedAlignmentPairs](#) classes.

Examples

```
## Here we only show a simple example illustrating the use of
## countCompatibleOverlaps() on a very small data set. Please
## refer to the "Overlap encodings" vignette in the GenomicRanges
## package for a more 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="GenomicRanges", mustWork=TRUE)

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

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

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

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

findOverlaps-methods *Finding overlapping genomic ranges*

Description

Finds interval overlaps between a [GRanges](#), [GRangesList](#), [GappedAlignments](#), [GAlignmentsList](#), or [GappedAlignmentPairs](#) object, and another object containing ranges.

Usage

```
## S4 method for signature 'GenomicRanges,GenomicRanges'
findOverlaps(query, subject,
             maxgap = 0L, minoverlap = 1L,
             type = c("any", "start", "end", "within", "equal"),
             select = c("all", "first", "last", "arbitrary"),
             ignore.strand = FALSE)
## S4 method for signature 'GenomicRanges,GenomicRanges'
countOverlaps(query, subject,
             maxgap = 0L, minoverlap = 1L,
             type = c("any", "start", "end", "within", "equal"),
             ignore.strand = FALSE)
## S4 method for signature 'GenomicRanges,GenomicRanges'
overlapsAny(query, subject,
            maxgap = 0L, minoverlap = 1L,
            type = c("any", "start", "end", "within", "equal"),
            ignore.strand = FALSE)
## S4 method for signature 'GenomicRanges,GenomicRanges'
subsetByOverlaps(query, subject,
                maxgap = 0L, minoverlap = 1L,
                type = c("any", "start", "end", "within", "equal"),
                ignore.strand = FALSE)
```

Arguments

query, subject A [GRanges](#), [GRangesList](#), [GappedAlignments](#), [GAlignmentsList](#), or [GappedAlignmentPairs](#) object. [RangesList](#) and [RangedData](#) are also accepted for one of query or subject.

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 and the subject are [GRanges](#) or [GRangesList](#) objects, `findOverlaps` uses the triplet (sequence name, range, strand) to determine which features (see paragraph below for the definition of feature) from the query overlap which features in the subject, where a strand value of "*" is treated as occurring on both the "+" and "-" strand. An overlap is recorded when a feature in the query and a feature in the subject have the same sequence name, have a compatible pairing of strands (e.g. "+"/"+", "-"/"-", "*"/"+", "*"/"-", etc.), and satisfy the interval overlap requirements. Strand is taken as "*" for [RangedData](#) and [RangesList](#).

In the context of `findOverlaps`, a feature is a collection of ranges that are treated as a single entity. For [GRanges](#) objects, a feature is a single range; while for [GRangesList](#) objects, a feature is a list element containing a set of ranges. In the results, the features are referred to by number, which run from 1 to `length(query)/length(subject)`.

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

[GAlignmentsList](#) objects are coerced to [GappedAlignments](#) then to a [GRangesList](#). Feature indices are mapped back to the original [GAlignmentsList](#) list elements.

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

Value

For `findOverlaps` either a [Hits](#) object when `select = "all"` or an integer vector otherwise.

For `countOverlaps` an integer vector containing the tabulated query overlap hits.

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

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

For `RangedData` and `RangesList`, with the exception of `subsetByOverlaps`, the results align to the unlisted form of the object. This turns out to be fairly convenient for `RangedData` (not so much for `RangesList`, but something has to give).

Author(s)

P. Aboyoun, S. Falcon, M. Lawrence, N. Gopalakrishnan and H. Pages

See Also

- [findOverlaps](#).
- [Hits-class](#).
- [GRanges-class](#).
- [GRangesList-class](#).
- [GappedAlignments-class](#).
- [GAlignmentsList-class](#).
- [GappedAlignmentPairs-class](#).

Examples

```
## -----
## WITH GRanges AND/OR GRangesList OBJECTS
## -----

## GRanges object:
gr <-
  GRanges(seqnames =
    Rle(c("chr1", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
    ranges =
    IRanges(1:10, width = 10:1, names = head(letters,10)),
    strand =
    Rle(strand(c("-", "+", "*+", "+", "-")),
      c(1, 2, 2, 3, 2)),
    score = 1:10,
```

```

        GC = seq(1, 0, length=10))
gr
## GRangesList object:
gr1 <-
  GRanges(seqnames = "chr2", ranges = IRanges(4:3, 6),
          strand = "+", score = 5:4, GC = 0.45)
gr2 <-
  GRanges(seqnames = c("chr1", "chr1"),
          ranges = IRanges(c(7,13), width = 3),
          strand = c("+", "-"), score = 3:4, GC = c(0.3, 0.5))
gr3 <-
  GRanges(seqnames = c("chr1", "chr2"),
          ranges = IRanges(c(1, 4), c(3, 9)),
          strand = c("-", "-"), score = c(6L, 2L), GC = c(0.4, 0.1))
gr1 <- GRangesList("gr1" = gr1, "gr2" = gr2, "gr3" = gr3)

## Overlapping two GRanges objects:
table(!is.na(findOverlaps(gr, gr1, select="arbitrary")))
countOverlaps(gr, gr1)
findOverlaps(gr, gr1)
subsetByOverlaps(gr, gr1)

countOverlaps(gr, gr1, type = "start")
findOverlaps(gr, gr1, type = "start")
subsetByOverlaps(gr, gr1, type = "start")

findOverlaps(gr, gr1, select = "first")
findOverlaps(gr, gr1, select = "last")

findOverlaps(gr1, gr)
findOverlaps(gr1, gr, type = "start")
findOverlaps(gr1, gr, type = "within")
findOverlaps(gr1, gr, type = "equal")

## Overlapping a GRanges and a GRangesList object:
table(!is.na(findOverlaps(gr1, gr, select="first")))
countOverlaps(gr1, gr)
findOverlaps(gr1, gr)
subsetByOverlaps(gr1, gr)
countOverlaps(gr1, gr, type = "start")
findOverlaps(gr1, gr, type = "start")
subsetByOverlaps(gr1, gr, type = "start")
findOverlaps(gr1, gr, select = "first")

## Overlapping two GRangesList objects:
countOverlaps(gr1, rev(gr1))
findOverlaps(gr1, rev(gr1))
subsetByOverlaps(gr1, rev(gr1))

## -----
## WITH GappedAlignments AND/OR GAlignmentsList OBJECTS
## -----

```

```

library(Rsamtools) # because file ex1.bam is in this package
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
galn <- readGappedAlignments(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	<i>Classify ranges (reads) as compatible with existing genomic annotations or as having novel splice events</i>
--------------------	---

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

```

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

## Low-level utils:

## High-level convenience wrappers (coming soon):
#summarizeSpliceOverlaps(query, subject, ignore.strand=FALSE, ...)
```

Arguments

query	character name of a Bam file, a BamFile , GappedAlignments , GappedAlignmentPairs or a GRangesList object containing the reads. Single or paired-end reads are specified with the <code>singleEnd</code> argument (default FALSE). Paired-end reads can be supplied in a Bam file or GappedAlignmentPairs object. Single-end are expected to be in a Bam file, GappedAlignments or GRanges object.
subject	A GRangesList containing the annotations. This list is expected to be exons by transcripts.
ignore.strand	When set to TRUE, strand information is ignored in the overlap calculations.
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.
...	

Details

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

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

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.

Additional methods

```
## Methods in Rsamtools :
```

```
findSpliceOverlaps(character,ANY(query, subject, ignore.strand=FALSE, ..., param=ScanBamParam(),
singleEnd=TRUE, cds=NULL)
```

```
findSpliceOverlaps(BamFile,ANY(query, subject, ignore.strand=FALSE, ..., param=ScanBamParam(),
singleEnd=TRUE, cds=NULL)
```

Author(s)

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

See Also

- The [GRangesList](#), [GappedAlignments](#), and [GappedAlignmentPairs](#) classes.

Examples

```
## -----
## Isoform expression :
## -----
## findSpliceOverlaps() can assist in quantifying isoform expression
## by identifying reads that map compatibly and uniquely to a
## transcript isoform.
library(Rsamtools)
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
## GappedAlignmentPairs object.
genes <- GRangesList(
  GRanges("chr1", IRanges(c(5, 20), c(10, 25)), "+"),
  GRanges("chr1", IRanges(c(5, 22), c(15, 25)), "+"))
galp <- GappedAlignmentPairs(
  GappedAlignments("chr1", 5L, "11M4N6M", strand("+")),
  GappedAlignments("chr1", 50L, "6M", strand("-")),
  isProperPair=TRUE)
findSpliceOverlaps(galp, genes)
```

GAlignmentsList-class *GAlignmentsList* objects

Description

The GAlignmentsList class is a container for storing a collection of GappedAlignments objects.

Details

A GAlignmentsList object contains a list of GappedAlignments. The majority of operations on this page are described in more detail on the GappedAlignments man page, see ?GappedAlignments.

Constructors

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

`readGAlignmentsList(file, format="BAM", use.names=FALSE, ...)`: Read a file containing aligned reads as a GAlignmentsList object. Note that this function is just a front-end that delegates to the `readBamGAlignmentsList` function defined in the Rsamtools package. See ?`readBamGAlignmentsList` for more information.

The `param` and `asProperPairs` arguments described on the `readBamGAlignmentsList` man page fine tune the records are returned. `param` is specified by the standard `ScanBamParam()` options. When `asProperPairs=TRUE` (default), only the proper pairs are returned. This is the same QC filtering that is done by `readBamGappedAlignmentPairs` but the records are returned in a GAlignmentsList instead of GappedAlignmentsPairs object. If `asProperPairs=FALSE`, all records in the file are retained (i.e., no QC) and the list structure is split by read id (QNAME in SAM/BAM). This format allows the flexibility to retain singletons, pairs, and multi-fragment reads in a single class.

`makeGAlignmentsListFromFeatureFragments(seqnames=Rle(factor()))`,
Constructs a GAlignmentsList from a list of fragmented features.

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`.

`ngap(x)`: Returns an integer list of length `x` containing the number of gaps (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 in each element of `x`. `value` must be a list of [Seqinfo](#) objects.

`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, ignore.strand=FALSE)`, `ranges(x)`: Return either a [GRanges](#) or a [IRanges](#) object of length `length(x)`. Note this coercion IGNORES the cigar information. The resulting ranges span the entire range, including any gaps or spaces between paired-end reads.

The `granges` coercion supports `ignore.strand` to allow ranges of opposite strand to be combined (see examples).

`grglist(x)`, `rglist(x)`: Return either a [GRangesList](#) or a [IRangesList](#) object of length `length(x)`. This coercion RESPECTS the cigar information. The resulting ranges are fragments of the original ranges that do not include gaps or spaces between paired-end reads.

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

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 `GappedAlignments`.

`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 `GappedAlignments`.

Combining

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

Other methods

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

`qnarrow(x, start=NA, end=NA, width=NA)`: Return a new `GAlignmentsList` object of the same length as `x` describing how the narrowed query sequences align to the reference. The `start/end/width` arguments describe how to narrow the query sequences. They must be 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).

Author(s)

Valerie Obenchain <vobencha@fhcrc.org>

References

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

See Also

- [GappedAlignments-class](#).
- [GappedAlignmentPairs-class](#).
- [readBamGappedAlignments](#).
- [readBamGappedAlignmentPairs](#).
- [findOverlaps-methods](#).

Examples

```
gal1 <- GappedAlignments(
  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 <- GappedAlignments(
  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(ngap(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) == "-"]
gaps <- sapply(galist, function(x) any(grepl("N", cigar(x), fixed=TRUE)))
galist[gaps]

## Different ways to subset:
galist[2] # a GappedAlignments object of length 1
galist[[2]] # a GappedAlignments 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 GappedAlignment elements
## and the overall GAlignmentsList object. By default, 'level=between'
## extracts the GAlignmentsList metadata. Using 'level=within'
## will extract the metadata on the individual GappedAlignments.

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

## -----
## D. readBamGAlignmentsList()
## -----

library(pasillaBamSubset)
library(Rsamtools)
fl <- sortBam(untreated3_chr4(), tempfile(), byQname=TRUE)
bf <- BamFile(fl, index=character(0), yieldSize=50, obeyQname=TRUE)
galist <- readGAlignmentsList(bf, asProperPairs=FALSE)
galist

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

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

## grglist() splits ranges by gap and the space between each list element
galist <- GAlignmentsList(noGaps=gal1, Gaps=gal2)
grglist(galist)

```

GappedAlignmentPairs-class

GappedAlignmentPairs objects

Description

The GappedAlignmentPairs class is a container for "alignment pairs".

Details

A `GappedAlignmentPairs` object is a list-like object where each element describes an "alignment pair".

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

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

Constructors

`readGappedAlignmentPairs(file, format="BAM", use.names=FALSE, ...)`: Read a file containing paired-end reads as a `GappedAlignmentPairs` object. By default (i.e. `use.names=FALSE`), the resulting object has no names. If `use.names` is `TRUE`, then the names are constructed from the query template names (QNAME field in a SAM/BAM file). Note that the 2 records in a pair of records have the same QNAME.

Note that this function is just a front-end that delegates to the format-specific back-end function specified via the `format` argument. The `use.names` argument and any extra argument are passed to the back-end function. Only the BAM format is supported for now. Its back-end is the `readBamGappedAlignmentPairs` function defined in the `Rsamtools` package. See [?readBamGappedAlignmentPairs](#) for more information (you might need to install and load the `Rsamtools` package first).

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

Accessors

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

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

`names(x)`, `names(x) <- value`: Get or set the names of `x`. See `readGappedAlignmentPairs` above for how to automatically extract and set the names from the file to read.

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

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

`right(x)`: Get the "right" alignment for each alignment pair in `x`. By definition, the "right" alignment in a pair is the alignment that is on the - strand. If this is the "first" alignment,

then it's returned as-is by `right(x)`, but if this is the "last" alignment, then it's returned by `right(x)` with the strand inverted.

`seqnames(x)`: Get the name of the reference sequence for each alignment pair in `x`. This comes from the RNAME field of the BAM file and has the same value for the 2 records in a pair (`makeGappedAlignmentPairs`, the function used by `readBamGappedAlignmentPairs` for doing the pairing, rejects pairs with incompatible RNAME values).

`strand(x)`, `strand(x) <- value`: Get or set the strand for each alignment pair in `x`. By definition (and in a somewhat arbitrary way) the strand of an alignment pair is the strand of the "first" alignment in the pair. In a `GappedAlignmentPairs` object, the strand of the "last" alignment in a pair is typically (but not always) the opposite of the strand of the "first" alignment. Note that, currently, `makeGappedAlignmentPairs`, the function used internally by `readBamGappedAlignmentPairs` for doing the pairing, rejects pairs where the "first" and "last" alignments are on the same strand, but those pairs might be supported in the future.

`ngap(x)`: Equivalent to `ngap(first(x)) + ngap(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 `GappedAlignmentPairs` 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 `GappedAlignmentPairs` object.

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

List methods

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

`x[[i]]`: Extract the *i*-th alignment pair as a [GappedAlignments](#) 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 [GappedAlignments](#) 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 [GappedAlignmentPairs](#) object.

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

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

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

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

If `FALSE` (the default), then the "left" ranges are placed before the "right" ranges, and, within each left or right group, are ordered from 5' to 3' in elements associated with the plus strand and from 3' to 5' in elements associated with the minus strand. More formally, the *i*-th element in the returned [GRangesList](#) object can be defined as `c(gr1[[i]], gr2[[i]])`, where `gr1` is `grglist(left(x))` and `gr2` is `grglist(right(x))`.

If `TRUE`, then the "first" ranges are placed before the "last" ranges, and, within each first or last group, are *always* ordered from 5' to 3', whatever the strand is. More formally, the *i*-th element in the returned [GRangesList](#) object can be defined as `c(gr1[[i]], gr2[[i]])`, where `gr1` is `grglist(first(x), order.as.in.query=TRUE)` and `gr2` is `grglist(last(x, invert.strand=TRUE), order.as.in.query=TRUE)`.

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

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

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

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

`introns(x)`: Extract the gaps (i.e. N operations in the CIGAR) of the "first" and "last" alignments of each pair as a [GRangesList](#) object of the same length as `x`. Equivalent to (but faster than):

```
introns1 <- introns(first(x))
introns2 <- introns(last(x, invert.strand=TRUE))
mendoapply(c, introns1, introns2)
```

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

Other methods

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

`show(x)`: The number of lines displayed in the `show` method are controlled with global options `showHeadLines` and `showTailLines`. If the sum of the two options is greater than the object length the full object is displayed. When these options are set they affect the display of `GRanges`, `GappedAlignments`, `GappedAlignmentPairs`, `Ranges` and `XString` objects.

Author(s)

H. Pages

See Also

- [GappedAlignments-class](#).
- [readBamGappedAlignmentPairs](#).
- [makeGappedAlignmentPairs](#).
- [GRangesList-class](#).
- [GRanges-class](#).
- [findOverlaps-methods](#).
- [coverage-methods](#).
- [seqinfo](#).

Examples

```
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
galp <- readGappedAlignmentPairs(ex1_file, use.names=TRUE)
galp

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

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

galp[[1]]
unlist(galp)
```

```

grglist(galp) # a GRangesList object
grglist(galp, order.as.in.query=TRUE)
stopifnot(identical(unname(elementLengths(grglist(galp))), ngap(galp) + 2L))

granges(galp) # a GRanges object

introns(galp) # a GRangesList object
stopifnot(identical(unname(elementLengths(introns(galp))), ngap(galp)))

```

GappedAlignments-class

GappedAlignments objects

Description

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

Details

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

Typically, a GappedAlignments 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 GappedAlignments 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 GappedAlignments 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 [?GappedAlignmentPairs](#) for how to handle aligned paired-end reads).

Each element of a GappedAlignments 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 has no gap (no N in the CIGAR). It is then considered 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, is NOT considered a gap.

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

Optionally, a GappedAlignments 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 GappedAlignments object in a way that is independent from how the data are actually stored internally.
- How to create and manipulate a GappedAlignments object.

Constructors

`readGappedAlignments(file, format="BAM", use.names=FALSE, ...)`: Read a file containing aligned reads as a GappedAlignments object. By default (i.e. `use.names=FALSE`), the resulting object has no names. If `use.names` is TRUE, then the names are constructed from the query template names (QNAME field in a SAM/BAM file).

Note that this function is just a front-end that delegates to the format-specific back-end function specified via the `format` argument. The `use.names` argument and any extra argument are passed to the back-end function. Only the BAM format is supported for now. Its back-end is the [readBamGappedAlignments](#) function defined in the Rsamtools package. See [?readBamGappedAlignments](#) for more information (you might need to install and load the Rsamtools package first).

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

Accessors

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

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

`names(x)`, `names(x) <- value`: Get or set the names of `x`. See [readGappedAlignments](#) above for how to automatically extract and set the names from the file to read.

`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).
- `ngap(x)`: Returns an integer vector of the same length as `x` containing the number of gaps (i.e. N operations in the CIGAR) for each alignment. Equivalent to `unname(elementLengths(rglist(x))) - 1L`.
- `seqinfo(x)`, `seqinfo(x) <- value`: Get or set the information about the underlying sequences. `value` must be a [Seqinfo](#) object.
- `seqlevels(x)`, `seqlevels(x) <- value`: Get or set the sequence levels. `seqlevels(x)` is equivalent to `seqlevels(seqinfo(x))` or to `levels(seqnames(x))`, those 2 expressions being guaranteed to return identical character vectors on a `GappedAlignments` 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 `GappedAlignments` object.

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

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

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

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

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

If `TRUE`, then the order of the ranges in elements associated with the *minus* strand is reversed. So they end up being ordered from 5' to 3' too, which means that they are now in descending order (i.e. right-to-left). It also means that, when `order.as.in.query=TRUE` is used, the ranges are *always* ordered consistently with the original "query template", that is, in the order defined by walking the "query template" from the beginning to the end.

If `drop.D.ranges` is `TRUE`, then deletions (D operations in the CIGAR) are treated like gaps (N operations in the CIGAR), that is, the ranges corresponding to deletions are dropped.

See Details section above for more information.

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

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

`introns(x)`: Extract the gaps (i.e. N operations in the CIGAR) as a [GRangesList](#) object of the same length as `x`. Equivalent to:

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

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

Subsetting and related operations

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

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

Combining

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

Other methods

`qnarrow(x, start=NA, end=NA, width=NA)`: `x` is a `GappedAlignments` object. Return a new `GappedAlignments` object of the same length as `x` describing how the narrowed query sequences align to the reference. The `start/end/width` arguments describe how to narrow the query sequences. They must be 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).

show(x): By default the show method displays 5 head and 5 tail lines. The number of lines can be altered by setting the global options showHeadLines and showTailLines. If the object length is less than the sum of the options, the full object is displayed. These options affect GRanges, GappedAlignments, Ranges, DataTable and XString objects.

Author(s)

H. Pages and P. Aboyoun

References

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

See Also

- [GappedAlignmentPairs-class](#).
- [readBamGappedAlignments](#).
- [GRangesList-class](#).
- [GRanges-class](#).
- [findOverlaps-methods](#).
- [coverage-methods](#).
- [seqinfo](#).
- [CompressedIRangesList-class](#).
- [setops-methods](#).

Examples

```
library(Rsamtools) # for ScanBamParam() and the ex1.bam file
ex1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
gal <- readGappedAlignments(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(ngap(gal))
seqlevels(gal)
```

```

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

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

## 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 gaps

## A confirmation that all the queries map to the reference with no
## gaps:
stopifnot(all(ngap(gal) == 0))

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

## D operations are NOT gaps:
ii <- grep("D", cigar(gal), fixed=TRUE)
gal[ii]
ngap(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]

## -----

```

```
## C. qnarrow()/narrow()
## -----
## Note that there is no difference between qnarrow() and narrow() when
## all the alignments are simple and with no indels.

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

## Trimming on the left doesn't change the "end" of the queries.
qnarrow(gal, start=21)
stopifnot(identical(end(qnarrow(gal, start=21)), end(gal)))
```

GenomicRanges-comparison

Comparing and ordering genomic ranges

Description

Methods for comparing and ordering the elements in one or more [GenomicRanges](#) objects.

Usage

```
## Element-wise (aka "parallel") comparison of 2 GenomicRanges objects
## -----

## S4 method for signature 'GenomicRanges,GenomicRanges'
e1 == e2

## S4 method for signature 'GenomicRanges,GenomicRanges'
e1 <= e2

## duplicated()
## -----

## S4 method for signature 'GenomicRanges'
duplicated(x, incomparables=FALSE, fromLast=FALSE,
           method=c("auto", "quick", "hash"))

## match()
## -----

## S4 method for signature 'GenomicRanges,GenomicRanges'
match(x, table, nomatch=NA_integer_, incomparables=NULL,
      method=c("auto", "quick", "hash"), ignore.strand=FALSE, match.if.overlap=FALSE)
```



```

## order() and related methods
## -----

## S4 method for signature 'GenomicRanges'
order(..., na.last=TRUE, decreasing=FALSE)

## S4 method for signature 'GenomicRanges'
rank(x, na.last=TRUE,
      ties.method=c("average", "first", "random", "max", "min"))

## Generalized element-wise (aka "parallel") comparison of 2 GenomicRanges
## objects
## -----

## S4 method for signature 'GenomicRanges,GenomicRanges'
compare(x, y)

```

Arguments

`e1`, `e2`, `x`, `table`, `y`
[GenomicRanges](#) objects.

`incomparables` Not supported.

`fromLast`, `method`, `nomatch`
 See ?[‘Ranges-comparison’](#) in the IRanges package for a description of these arguments.

`ignore.strand` Whether or not the strand should be ignored when comparing 2 genomic ranges.

`match.if.overlap`
 You temporarily need to explicitly provide this argument otherwise `match()` will issue a warning.
 Starting with BioC 2.12, the default behavior of `match()` on [GenomicRanges](#) objects has changed to use *equality* instead of *overlap* for comparing elements between [GenomicRanges](#) objects `x` and `table`. Now `x[i]` and `table[j]` are considered to match when they are equal (i.e. `x[i] == table[j]`), instead of when they overlap. This new behavior is consistent with base: `:match()`.
 If you need the new behavior, use `match.if.overlap=FALSE`.
 If you need the old behavior, you can either do: `findOverlaps(x, table, select="first")` which is the recommended way. Alternatively, you can call `match()` with `match.if.overlap=TRUE`.

`...` Additional [GenomicRanges](#) objects used for breaking ties.

`na.last` Ignored.

`decreasing` TRUE or FALSE.

`ties.method` A character string specifying how ties are treated. Only "first" is supported for now.

Details

Two elements of a [GenomicRanges](#) object (i.e. two genomic ranges) are considered equal iff they are on the same underlying sequence and strand, and have the same start and width. `duplicated()` and `unique()` on a [GenomicRanges](#) object are conforming to this.

The "natural order" for the elements of a [GenomicRanges](#) object is to order them (a) first by sequence level, (b) then by strand, (c) then by start, (d) and finally by width. This way, the space of genomic ranges is totally ordered. Note that the `reduce` method for [GenomicRanges](#) uses this "natural order" implicitly. Also, note that, because we already do (c) and (d) for regular ranges (see [?‘Ranges-comparison’](#)), genomic ranges that belong to the same underlying sequence and strand are ordered like regular ranges. `order()`, `sort()`, and `rank()` on a [GenomicRanges](#) object are using this "natural order".

Also the `==`, `!=`, `<=`, `>=`, `<` and `>` operators between 2 [GenomicRanges](#) objects are using this "natural order".

Author(s)

H. Pages

See Also

- The [GenomicRanges](#) class.
- [Ranges-comparison](#) in the `IRanges` package for comparing and ordering genomic ranges.
- [intra-range-methods](#) and [inter-range-methods](#) for intra and inter range transformations.
- [setops-methods](#) for set operations on [GenomicRanges](#) objects.
- [findOverlaps-methods](#) for finding overlapping genomic ranges.

Examples

```
gr0 <- GRanges(
  seqnames=Rle(c("chr1", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
  ranges=IRanges(c(1:9,7L), end=10),
  strand=Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
  seqlengths=c(chr1=11, chr2=12, chr3=13))
gr <- c(gr0, gr0[7:3])

## -----
## A. ELEMENT-WISE (AKA "PARALLEL") COMPARISON OF 2 GenomicRanges OBJECTS
## -----
gr[2] == gr[2] # TRUE
gr[2] == gr[5] # FALSE
gr == gr[4]
gr >= gr[3]

## -----
## B. duplicated(), unique()
## -----
duplicated(gr)
unique(gr)
```

```

## -----
## C. match(), %in%
## -----
table <- gr[1:7]
match(gr, table) # Warning! The warning will be removed in BioC 2.13.

## In the meantime, specify 'match.if.overlap=FALSE' to suppress the warning:
match(gr, table, match.if.overlap=FALSE)
match(gr, table, ignore.strand=TRUE, match.if.overlap=FALSE)

gr %in% table # Warning! The warning will be removed in BioC 2.13.
## In the meantime, use suppressWarnings() to suppress the warning:
suppressWarnings(gr %in% table)

## -----
## D. findMatches(), countMatches()
## -----
findMatches(gr, table)
countMatches(gr, table)

findMatches(gr, table, ignore.strand=TRUE)
countMatches(gr, table, ignore.strand=TRUE)

gr_levels <- unique(gr)
countMatches(gr_levels, gr)

## -----
## E. order() AND RELATED METHODS
## -----
order(gr)
sort(gr)
rank(gr)

## -----
## F. GENERALIZED ELEMENT-WISE COMPARISON OF 2 GenomicRanges OBJECTS
## -----
gr2 <- GRanges(c(rep("chr1", 12), "chr2"), IRanges(c(1:11, 6:7), width=3))
strand(gr2)[12] <- "+"
gr3 <- GRanges("chr1", IRanges(5, 9))

compare(gr2, gr3)
rangeComparisonCodeToLetter(compare(gr2, gr3))

```

GenomicRangesList-class

GenomicRangesList objects

Description

A GenomicRangesList is a [List](#) of [GenomicRanges](#). It is a virtual class; SimpleGenomicRangesList is the basic implementation. The subclass [GRangesList](#) provides special behavior and is particularly efficient for storing a large number of elements.

Constructor

GenomicRangesList(...): Constructs a SimpleGenomicRangesList with elements taken from the arguments in If the only argument is a list, the elements are taken from that list.

Coercion

as(from, "GenomicRangesList"): Supported from types include:

RangedDataList Each element of from is coerced to a GenomicRanges.

Author(s)

Michael Lawrence

See Also

[GRangesList](#), which differs from SimpleGenomicRangesList in that the GRangesList treats its elements as single, compound ranges, particularly in overlap operations. SimpleGenomicRangesList is just a barebones list for now, without that compound semantic.

GRanges-class

GRanges objects

Description

The GRanges class is a container for the genomic locations and their associated annotations.

Details

The GRanges class stores the sequences of genomic locations and associated annotations. Each element in the sequence is comprised of a sequence name, an interval, a [strand](#), and optional metadata columns (e.g. score, GC content, etc.). This information is stored in four components:

seqnames a 'factor' [Rle](#) object containing the sequence names.

ranges an [IRanges](#) object containing the ranges.

strand a 'factor' [Rle](#) object containing the [strand](#) information.

mcols a [DataFrame](#) object containing the metadata columns. Columns cannot be named "seqnames", "ranges", "strand", "seqlevels", "seqlengths", "isCircular", "start", "end", "width", or "element".

seqinfo a [Seqinfo](#) object containing information about the set of genomic sequences present in the GRanges object.

Constructor

`GRanges(seqnames = Rle(), ranges = IRanges(), strand = Rle("*", length(seqnames)), ...)`
 Creates a `GRanges` object.

`seqnames` `Rle` object, character vector, or factor containing the sequence names.

`ranges` `IRanges` object containing the ranges.

`strand` `Rle` object, character vector, or factor containing the strand information.

... Optional metadata columns. These columns cannot be named "start", "end", "width", or "element". A named integer vector "seqlength" can be used instead of `seqinfo`.

`seqlengths` an integer vector named with the sequence names and containing the lengths (or NA) for each `level(seqnames)`.

`seqinfo` a `Seqinfo` object containing allowed sequence names and lengths (or NA) for each `level(seqnames)`.

Coercion

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

`as(from, "GRanges")`: Creates a `GRanges` object from a `RangedData`, `RangesList`, `RleList` or `RleViewsList` object.

`as(from, "RangedData")`: Creates a `RangedData` object from a `GRanges` object. The `strand` and `metadata` columns become columns in the result. The `seqlengths(from)`, `isCircular(from)`, and `genome(from)` vectors are stored in the `metadata` columns of `ranges(rd)`.

`as(from, "RangesList")`: Creates a `RangesList` object from a `GRanges` object. The `strand` and `metadata` columns become *inner* `metadata` columns (i.e. `metadata` columns on the ranges). The `seqlengths(from)`, `isCircular(from)`, and `genome(from)` vectors become the `metadata` columns.

`as(from, "GappedAlignments")`: Creates a `GappedAlignments` 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`.

`as.data.frame(x, row.names = NULL, optional = FALSE, ...)`: Creates a `data.frame` with columns `seqnames` (factor), `start` (integer), `end` (integer), `width` (integer), `strand` (factor), as well as the additional `metadata` columns stored in `mcols(x)`. Pass an explicit `stringsAsFactors=TRUE/FALSE` argument via ... to override the default conversions for the `metadata` columns in `mcols(x)`.

Accessors

In the following code snippets, `x` is a `GRanges` object.

`length(x)`: Get the number of elements.

`seqnames(x)`, `seqnames(x) <- value`: Get or set the sequence names. `value` can be an `Rle` object, a character vector, or a factor.

`ranges(x)`, `ranges(x) <- value`: Get or set the ranges. `value` can be a `Ranges` object.

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

`strand(x)`, `strand(x) <- value`: Get or set the strand. `value` can be an `Rle` object, character vector, or factor.

`mcols(x, use.names=FALSE)`, `mcols(x) <- value`: Get or set the metadata columns. If `use.names=TRUE` and the metadata columns are not NULL, then the names of `x` are propagated as the row names of the returned [DataFrame](#) object. When setting the metadata columns, the supplied value must be NULL or a data.frame-like object (i.e. [DataTable](#) or `data.frame`) object holding element-wise metadata.

`elementMetadata(x)`, `elementMetadata(x) <- value`, `values(x)`, `values(x) <- value`: Alternatives to `mcols` functions. Their use is discouraged.

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

`seqlevels(x)`, `seqlevels(x, force=FALSE) <- 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 `GRanges` 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.

`score(x)`: Get the “score” column from the element metadata, if any.

Ranges methods

In the following code snippets, `x` is a `GRanges` object.

`start(x)`, `start(x) <- value`: Get or set `start(ranges(x))`.

`end(x)`, `end(x) <- value`: Get or set `end(ranges(x))`.

`width(x)`, `width(x) <- value`: Get or set `width(ranges(x))`.

Splitting and Combining

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

`append(x, values, after = length(x))`: Inserts the `values` into `x` at the position given by `after`, where `x` and `values` are of the same class.

`c(x, ...)`: Combines `x` and the `GRanges` objects in `...` together. Any object in `...` must belong to the same class as `x`, or to one of its subclasses, or must be NULL. The result is an object of the same class as `x`.

- `c(x, ..., ignore.mcols=FALSE)` If the GRanges objects have metadata columns (represented as one [DataFrame](#) per object), each such [DataFrame](#) must have the same columns in order to combine successfully. In order to circumvent this restraint, you can pass in an `ignore.mcols=TRUE` argument which will combine all the objects into one and drop all of their metadata columns.
- `split(x, f, drop=FALSE)`: Splits `x` according to `f` to create a [GRangesList](#) object. If `f` is a list-like object then `drop` is ignored and `f` is treated as if it was `rep(seq_len(length(f)), sapply(f, length))`, so the returned object has the same shape as `f` (it also receives the names of `f`). Otherwise, if `f` is not a list-like object, empty list elements are removed from the returned object if `drop` is `TRUE`.

Subsetting

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

- `x[i, j], x[i, j] <- value`: Get or set elements `i` with optional metadata columns `mcols(x)[, j]`, where `i` can be missing; an NA-free logical, numeric, or character vector; or a 'logical' Rle object.
- `x[i, j] <- value`: Replaces elements `i` and optional metadata columns `j` with `value`.
- `head(x, n = 6L)`: If `n` is non-negative, returns the first `n` elements of the GRanges object. If `n` is negative, returns all but the last `abs(n)` elements of the GRanges object.
- `rep(x, times, length.out, each)`: Repeats the values in `x` through one of the following conventions:
- `times` Vector giving the number of times to repeat each element if of length `length(x)`, or to repeat the whole vector if of length 1.
 - `length.out` Non-negative integer. The desired length of the output vector.
 - `each` Non-negative integer. Each element of `x` is repeated `each` times.
- `seqselect(x, start=NULL, end=NULL, width=NULL)`: Similar to `window`, except that multiple consecutive subsequences can be requested for concatenation. As such two of the three `start`, `end`, and `width` arguments can be used to specify the consecutive subsequences. Alternatively, `start` can take a Ranges object or something that can be converted to a Ranges object like an integer vector, logical vector or logical Rle. If the concatenation of the consecutive subsequences is undesirable, consider using [Views](#).
- `seqselect(x, start=NULL, end=NULL, width=NULL) <- value`: Similar to `window<-`, except that multiple consecutive subsequences can be replaced with a `value` whose length is a divisor of the number of elements it is replacing. As such two of the three `start`, `end`, and `width` arguments can be used to specify the consecutive subsequences. Alternatively, `start` can take a Ranges object or something that can be converted to a Ranges object like an integer vector, logical vector or logical Rle.
- `subset(x, subset)`: Returns a new object of the same class as `x` made of the subset using logical vector `subset`, where missing values are taken as `FALSE`.
- `tail(x, n = 6L)`: If `n` is non-negative, returns the last `n` elements of the GRanges object. If `n` is negative, returns all but the first `abs(n)` elements of the GRanges object.
- `window(x, start = NA, end = NA, width = NA, frequency = NULL, delta = NULL, ...)`: Extracts the subsequence `window` from the GRanges object using:
- `start, end, width` The start, end, or width of the window. Two of the three are required.

frequency, delta Optional arguments that specify the sampling frequency and increment within the window.

In general, this is more efficient than using "[" operator.

`window(x, start = NA, end = NA, width = NA, keepLength = TRUE) <- value`: Replaces the subsequence window specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must either be of class `class(x)`, belong to a subclass of `class(x)`, be coercible to `class(x)`, or be `NULL`. If `keepLength` is `TRUE`, the elements of `value` are repeated to create a `GRanges` object with the same number of elements as the width of the subsequence window it is replacing. If `keepLength` is `FALSE`, this replacement method can modify the length of `x`, depending on how the length of the left subsequence window compares to the length of `value`.

`x$name, x$name <- value`: Shortcuts for `mcols(x)$name` and `mcols(x)$name <- value`, respectively. Provided as a convenience, for `GRanges` objects *only*, and as the result of strong popular demand. Note that those methods are not consistent with the other `$` and `$<-` methods in the `IRanges/GenomicRanges` infrastructure, and might confuse some users by making them believe that a `GRanges` object can be manipulated as a `data.frame`-like object. Therefore we recommend using them only interactively, and we discourage their use in scripts or packages. For the latter, use `mcols(x)$name` and `mcols(x)$name <- value`, instead of `x$name` and `x$name <- value`, respectively.

Other methods

`show(x)`: By default the `show` method displays 5 head and 5 tail lines. The number of lines can be altered by setting the global options `showHeadLines` and `showTailLines`. If the object length is less than the sum of the options, the full object is displayed. These options affect `GRanges`, `GappedAlignments`, `Ranges`, `DataTable` and `XString` objects.

Author(s)

P. Aboyoun

See Also

[GRangesList-class](#), [seqinfo](#), [Vector-class](#), [Ranges-class](#), [Rle-class](#), [DataFrame-class](#), [intra-range-methods](#), [inter-range-methods](#), [setops-methods](#), [findOverlaps-methods](#), [nearest-methods](#), [coverage-methods](#)

Examples

```
seqinfo <- Seqinfo(paste0("chr", 1:3), c(1000, 2000, 1500), NA, "mock1")
gr <-
  GRanges(seqnames =
    Rle(c("chr1", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
    ranges = IRanges(
      1:10, width = 10:1, names = head(letters,10)),
    strand = Rle(
      strand(c("-", "+", "x", "+", "-")),
      c(1, 2, 2, 3, 2)),
    score = 1:10,
```



```

        GC = seq(1, 0, length=10),
        seqinfo=seqinfo)
gr

## Summarizing elements
table(seqnames(gr))
sum(width(gr))
summary(mcols(gr)[,"score"])

## Renaming the underlying sequences
seqlevels(gr)
seqlevels(gr) <- sub("chr", "Chrom", seqlevels(gr))
gr
seqlevels(gr) <- sub("Chrom", "chr", seqlevels(gr)) # revert

## Combining objects
gr2 <- GRanges(seqnames=Rle(c('chr1', 'chr2', 'chr3'), c(3, 3, 4)),
               IRanges(1:10, width=5), strand='- ',
               score=101:110, GC = runif(10),
               seqinfo=seqinfo)
gr3 <- GRanges(seqnames=Rle(c('chr1', 'chr2', 'chr3'), c(3, 4, 3)),
               IRanges(101:110, width=10), strand='- ',
               score=21:30,
               seqinfo=seqinfo)
some.gr <- c(gr, gr2)

## all.gr <- c(gr, gr2, gr3) ## (This would fail)
all.gr <- c(gr, gr2, gr3, ignore.mcols=TRUE)

## The number of lines displayed in the 'show' method
## are controlled with two global options.
longGR <- c(gr[,"score"], gr2[,"score"], gr3)
longGR
options("showHeadLines"=7)
options("showTailLines"=2)
longGR

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

```

GRangesList-class

GRangesList objects

Description

The GRangesList class is a container for storing a collection of GRanges objects. It is derived from GenomicRangesList.

Constructors

`GRangesList(...)`: Creates a `GRangesList` object using `GRanges` objects supplied in `...`.

`makeGRangesListFromFeatureFragments(seqnames=Rle(factor()), fragmentStarts=list(), fragmentEnds=list(), strand=factor(), mcols=mcols)`:
Constructs a `GRangesList` object from a list of fragmented features. See the Examples section below.

Coercion

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

`as.data.frame(x, row.names = NULL, optional = FALSE)`: Creates a `data.frame` with columns `element` (character), `seqnames` (factor), `start` (integer), `end` (integer), `width` (integer), `strand` (factor), as well as the additional metadata columns (accessed with `mcols(unlist(x))`).

`as.list(x, use.names = TRUE)`: Creates a list containing the elements of `x`.

`as(x, "IRangesList")`: Turns `x` into an `IRangesList` object.

`as(from, "GRangesList")`: Creates a `GRangesList` object from a `RangedDataList` object.

Accessors

In the following code snippets, `x` is a `GRanges` object.

`seqnames(x)`, `seqnames(x) <- value`: Get or set the sequence names in the form of an `RleList`. `value` can be an `RleList` or `CharacterList`.

`ranges(x)`, `ranges(x) <- value`: Get or set the ranges in the form of a `CompressedIRangesList`. `value` can be a `RangesList` object.

`strand(x)`, `strand(x) <- value`: Get or set the strand in the form of an `RleList`. `value` can be an `RleList` or `CharacterList` object.

`mcols(x, use.names=FALSE)`, `mcols(x) <- value`: Get or set the metadata columns. `value` can be `NULL`, or a `data.frame`-like object (i.e. `DataFrame` or `data.frame`) holding element-wise metadata.

`elementMetadata(x)`, `elementMetadata(x) <- value`, `values(x)`, `values(x) <- value`: Alternatives to `mcols` functions. Their use is discouraged.

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

`seqlevels(x)`, `seqlevels(x, force=FALSE) <- 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 `GRangesList` 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.
- `score(x)`: Get the “score” column from the element metadata, if any.

List methods

In the following code snippets, `x` is a `GRangesList` object.

- `length(x)`: Get the number of elements.
- `names(x)`, `names(x) <- value`: Get or set the names of the elements.
- `elementLengths(x)`: Get the length of each of the elements.
- `isEmpty(x)`: Returns a logical indicating either if the `GRangesList` has no elements or if all its elements are empty.

RangesList methods

In the following code snippets, `x` is a `GRangesList` object.

- `start(x)`, `start(x) <- value`: Get or set `start(ranges(x))`.
- `end(x)`, `end(x) <- value`: Get or set `end(ranges(x))`.
- `width(x)`, `width(x) <- value`: Get or set `width(ranges(x))`.
- `shift(x, shift, use.names=TRUE)`: Returns a new `GRangesList` object containing intervals with start and end values that have been shifted by integer vector `shift`. The `use.names` argument determines whether or not to keep the names on the ranges.
- `isDisjoint(x)` Return a vector of logical values indicating whether the ranges of each element of `x` are disjoint (i.e. non-overlapping).
- `disjoin(x, ...)` Returns an object of the same type as `x` containing disjoint ranges calculated on each element of `x`.

Combining

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

- `append(x, values, after = length(x))`: Inserts the `values` into `x` at the position given by `after`, where `x` and `values` are of the same class.
- `c(x, ...)`: Combines `x` and the `GRangesList` objects in `...` together. Any object in `...` must belong to the same class as `x`, or to one of its subclasses, or must be `NULL`. The result is an object of the same class as `x`.
- `unlist(x, recursive = TRUE, use.names = TRUE)`: Concatenates the elements of `x` into a single `GRanges` object.

Subsetting

In the following code snippets, `x` is a `GRangesList` object.

`x[i, j], x[[i, j]] <- value`: Get or set elements `i` with optional metadata columns `mcols(x)[, j]`, where `i` can be missing; an NA-free logical, numeric, or character vector; a 'logical' Rle object, or an `AtomicList` object.

`x[[i]], x[[[i]]] <- value`: Get or set element `i`, where `i` is a numeric or character vector of length 1.

`x$name, x$[name] <- value`: Get or set element name, where `name` is a name or character vector of length 1.

`head(x, n = 6L)`: If `n` is non-negative, returns the first `n` elements of the `GRangesList` object. If `n` is negative, returns all but the last `abs(n)` elements of the `GRangesList` object.

`rep(x, times, length.out, each)`: Repeats the values in `x` through one of the following conventions:

`times` Vector giving the number of times to repeat each element if of length `length(x)`, or to repeat the whole vector if of length 1.

`length.out` Non-negative integer. The desired length of the output vector.

`each` Non-negative integer. Each element of `x` is repeated `each` times.

`seqselect(x, start=NULL, end=NULL, width=NULL)`: Similar to `window`, except that multiple consecutive subsequences can be requested for concatenation. As such two of the three `start`, `end`, and `width` arguments can be used to specify the consecutive subsequences. Alternatively, `start` can take a `Ranges` object or something that can be converted to a `Ranges` object like an integer vector, logical vector or logical Rle. If the concatenation of the consecutive subsequences is undesirable, consider using [Views](#).

`seqselect(x, start=NULL, end=NULL, width=NULL) <- value`: Similar to `window<-`, except that multiple consecutive subsequences can be replaced by a value whose length is a divisor of the number of elements it is replacing. As such two of the three `start`, `end`, and `width` arguments can be used to specify the consecutive subsequences. Alternatively, `start` can take a `Ranges` object or something that can be converted to a `Ranges` object like an integer vector, logical vector or logical Rle.

`subset(x, subset)`: Returns a new object of the same class as `x` made of the subset using logical vector `subset`, where missing values are taken as `FALSE`.

`tail(x, n = 6L)`: If `n` is non-negative, returns the last `n` elements of the `GRanges` object. If `n` is negative, returns all but the first `abs(n)` elements of the `GRanges` object.

`window(x, start = NA, end = NA, width = NA, frequency = NULL, delta = NULL, ...)`: Extracts the subsequence window from the `GRanges` object using:

`start, end, width` The start, end, or width of the window. Two of the three are required.

`frequency, delta` Optional arguments that specify the sampling frequency and increment within the window.

In general, this is more efficient than using `"["` operator.

`window(x, start = NA, end = NA, width = NA, keepLength = TRUE) <- value`: Replaces the subsequence window specified on the left (i.e. the subsequence in `x` specified by `start`, `end` and `width`) by `value`. `value` must either be of class `class(x)`, belong to a subclass of `class(x)`, be coercible to `class(x)`, or be `NULL`. If `keepLength` is `TRUE`, the elements of

value are repeated to create a GRanges object with the same number of elements as the width of the subsequence window it is replacing. If `keepLength` is `FALSE`, this replacement method can modify the length of `x`, depending on how the length of the left subsequence window compares to the length of `value`.

Looping

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

`endoapply(X, FUN, ...)`: Similar to `lapply`, but performs an endomorphism, i.e. returns an object of `class(X)`.

`lapply(X, FUN, ...)`: Like the standard `lapply` function defined in the base package, the `lapply` method for GRangesList objects returns a list of the same length as `X`, with each element being the result of applying `FUN` to the corresponding element of `X`.

`Map(f, ...)`: Applies a function to the corresponding elements of given GRangesList objects.

`mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)`: Like the standard `mapply` function defined in the base package, the `mapply` method for GRangesList objects is a multivariate version of `sapply`.

`mendoapply(FUN, ..., MoreArgs = NULL)`: Similar to `mapply`, but performs an endomorphism across multiple objects, i.e. returns an object of `class(list(...)[[1]])`.

`Reduce(f, x, init, right = FALSE, accumulate = FALSE)`: Uses a binary function to successively combine the elements of `x` and a possibly given initial value.

f A binary argument function.

init An R object of the same kind as the elements of `x`.

right A logical indicating whether to proceed from left to right (default) or from right to left.

nomatch The value to be returned in the case when "no match" (no element satisfying the predicate) is found.

`sapply(X, FUN, ..., simplify=TRUE, USE.NAMES=TRUE)`: Like the standard `sapply` function defined in the base package, the `sapply` method for GRangesList objects is a user-friendly version of `lapply` by default returning a vector or matrix if appropriate.

The "range", "reduce" and "restrict" methods

In the code snippets below, `x` is a GRangesList object. The methods in this section are isomorphisms, that is, they are endomorphisms (i.e. they preserve the class of `x`) who also preserve the length & names & metadata columns of `x`. In addition, the `seqinfo` is preserved too.

`range(x)`: Applies `range` to each element in `x`. More precisely, it is equivalent to `endoapply(x, range)`.

`reduce(x, drop.empty.ranges=FALSE, min.gapwidth=1L)`: Applies `reduce` to each element in `x`. More precisely, it is equivalent to `endoapply(x, reduce, drop.empty.ranges=drop.empty.ranges,`

`restrict(x, start = NA, end = NA, keep.all.ranges = FALSE, use.names = TRUE)`: Applies `restrict` to each element in `x`.

`flank(x, width, start = TRUE, end = NA, keep.all.ranges = FALSE, use.names = TRUE, ignore.strand)`: Applies `flank` to each element in `x`.

Author(s)

P. Aboyoun & H. Pages

See Also

[GRanges-class](#), [seqinfo](#), [Vector-class](#), [RangesList-class](#), [RleList-class](#), [DataFrameList-class](#), [coverage-methods](#), [setops-methods](#), [findOverlaps-methods](#)

Examples

```
## Construction with GRangesList():
gr1 <-
  GRanges(seqnames = "chr2", ranges = IRanges(3, 6),
          strand = "+", score = 5L, GC = 0.45)
gr2 <-
  GRanges(seqnames = c("chr1", "chr1"),
          ranges = IRanges(c(7,13), width = 3),
          strand = c("+", "-"), score = 3:4, GC = c(0.3, 0.5))
gr3 <-
  GRanges(seqnames = c("chr1", "chr2"),
          ranges = IRanges(c(1, 4), c(3, 9)),
          strand = c("-", "-"), score = c(6L, 2L), GC = c(0.4, 0.1))
grl <- GRangesList("gr1" = gr1, "gr2" = gr2, "gr3" = gr3)
grl

## Summarizing elements:
elementLengths(grl)
table(seqnames(grl))

## Extracting subsets:
grl[seqnames(grl) == "chr1", ]
grl[seqnames(grl) == "chr1" & strand(grl) == "+", ]

## Renaming the underlying sequences:
seqlevels(grl)
seqlevels(grl) <- sub("chr", "Chrom", seqlevels(grl))
grl

## range() and reduce():
range(grl)
reduce(grl) # Doesn't really reduce anything but note the reordering
            # of the inner elements in the 3rd top-level element: the
            # ranges are reordered by sequence name first (the order of
            # the sequence names is dictated by the sequence levels),
            # and then by strand.
restrict(grl, start=3)
### flank
flank(grl, width =20)

## Coerce to IRangesList (seqnames and strand information is lost):
as(grl, "IRangesList")
```

```
## isDisjoint():
isDisjoint(gr1)

## disjoint():
disjoin(gr1) # metadata columns and order NOT preserved

## Construction with makeGRangesListFromFeatureFragments():
filepath <- system.file("extdata", "feature_fragments.txt",
                        package="GenomicRanges")
featfrags <- read.table(filepath, header=TRUE, stringsAsFactors=FALSE)
gr12 <- with(featfrags,
            makeGRangesListFromFeatureFragments(seqnames=targetName,
                                                fragmentStarts=targetStart,
                                                fragmentWidths=blockSizes,
                                                strand=strand))

names(gr12) <- featfrags$RefSeqID
gr12
```

inter-range-methods *Inter range transformations of a GenomicRanges object*

Description

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

This man page documents inter range transformations of a [GenomicRanges](#) object (i.e. of an object that belongs to the [GenomicRanges](#) class or one of its subclasses, this includes for example [GRanges](#) objects).

See ?[‘intra-range-methods’](#) for intra range transformations of a GenomicRanges object.

Usage

```
## S4 method for signature 'GenomicRanges'
range(x, ..., ignore.strand=FALSE, na.rm=FALSE)

## S4 method for signature 'GenomicRanges'
reduce(x, drop.empty.ranges=FALSE, min.gapwidth=1L,
       with.mapping=FALSE, with.inframe.attrib=FALSE, ignore.strand=FALSE)

## S4 method for signature 'GenomicRanges'
gaps(x, start=1L, end=seqlengths(x))

## S4 method for signature 'GenomicRanges'
disjoin(x, ignore.strand=FALSE)

## S4 method for signature 'GenomicRanges'
isDisjoint(x, ignore.strand=FALSE)
```

```
## S4 method for signature 'GenomicRanges'
disjointBins(x, ignore.strand=FALSE)
```

Arguments

`x` A [GenomicRanges](#) object.

`drop.empty.ranges`, `min.gapwidth`, `with.mapping`, `with.inframe.attrib`, `start`, `end`
See ?‘[inter-range-methods](#)‘ in the IRanges package.

`ignore.strand` TRUE or FALSE. Whether the strand of the input ranges should be ignored or not. See details below.

`...` For range, additional [GenomicRanges](#) objects to consider. Ignored otherwise.

`na.rm` Ignored.

Details

`range` returns an object of the same type as `x` containing range bounds for each distinct (seqname, strand) pairing. The names (`names(x)`) and the metadata columns in `x` are dropped.

`reduce` returns an object of the same type as `x` containing reduced ranges for each distinct (seqname, strand) pairing. The names (`names(x)`) and the metadata columns in `x` are dropped. See ?[reduce](#) for more information about range reduction and for a description of the optional arguments.

`gaps` returns an object of the same type as `x` containing complemented ranges for each distinct (seqname, strand) pairing. The names (`names(x)`) and the columns in `x` are dropped. For the start and end arguments of this `gaps` method, it is expected that the user will supply a named integer vector (where the names correspond to the appropriate seqlevels). See ?[gaps](#) for more information about range complements and for a description of the optional arguments.

`disjoin` returns an object of the same type as `x` containing disjoint ranges for each distinct (seqname, strand) pairing. The names (`names(x)`) and the metadata columns in `x` are dropped.

`isDisjoint` returns a logical value indicating whether the ranges in `x` are disjoint (i.e. non-overlapping).

`disjointBins` returns bin indexes for the ranges in `x`, such that ranges in the same bin do not overlap. If `ignore.strand=FALSE`, the two features cannot overlap if they are on different strands.

Author(s)

H. Pages and P. Aboyoun

See Also

- The [GenomicRanges](#) and [GRanges](#) classes.
- The [Ranges](#) class in the IRanges package.
- The [inter-range-methods](#) man page in the IRanges package.
- [GenomicRanges-comparison](#) for comparing and ordering genomic ranges.

Examples

```

gr <- GRanges(
  seqnames=Rle(paste("chr", c(1, 2, 1, 3), sep=""), c(1, 3, 2, 4)),
  ranges=IRanges(1:10, width=10:1, names=letters[1:10]),
  strand=Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
  score=1:10,
  GC=seq(1, 0, length=10)
)

gr

## -----
## range()
## -----
range(gr)

# -----
## reduce()
## -----
reduce(gr)

gr2 <- reduce(gr, with.mapping=TRUE)
mapping <- mcols(gr2)$mapping # an IntegerList

## Use the mapping from reduced to original ranges to group the original
## ranges by reduced range:
relist(gr[unlist(mapping)], mapping)

## Or use it to split the DataFrame of original metadata columns by
## reduced range:
relist(mcols(gr)[unlist(mapping), ], mapping) # a SplitDataFrameList

## [For advanced users] Use the mapping to compare the reduced ranges
## with the ranges they originate from:
expanded_gr2 <- rep(gr2, elementLengths(mapping))
reordered_gr <- gr[unlist(mapping)]
codes <- compare(expanded_gr2, reordered_gr)
## All the codes should translate to "d", "e", "g", or "h" (the 4 letters
## indicating that the range on the left contains the range on the right):
alphacodes <- rangeComparisonCodeToLetter(compare(expanded_gr2, reordered_gr))
stopifnot(all(alphacodes %in% c("d", "e", "g", "h")))

## On a big GRanges object with a lot of seqlevels:
mcols(gr) <- NULL
biggr <- c(gr, GRanges("chr1", IRanges(c(4, 1), c(5, 2)), strand="+"))
seqlevels(biggr) <- paste0("chr", 1:2000)
biggr <- rep(biggr, 25000)
set.seed(33)
seqnames(biggr) <- sample(factor(seqlevels(biggr), levels=seqlevels(biggr)),
  length(biggr), replace=TRUE)

biggr2 <- reduce(biggr, with.mapping=TRUE)
mapping <- mcols(biggr2)$mapping

```

```

expanded_biggr2 <- rep(biggr2, elementLengths(mapping))
reordered_biggr <- biggr[unlist(mapping)]
codes <- compare(expanded_biggr2, reordered_biggr)
alphacodes <- rangeComparisonCodeToLetter(compare(expanded_biggr2,
                                                reordered_biggr))

stopifnot(all(alphacodes %in% c("d", "e", "g", "h")))
table(alphacodes)

## -----
## gaps()
## -----
gaps(gr, start = 1, end = 10)

## -----
## disjoint(), isDisjoint(), disjointBins()
## -----
disjoin(gr)
isDisjoint(gr)
stopifnot(isDisjoint(disjoin(gr)))
disjointBins(gr)
stopifnot(all(sapply(split(gr, disjointBins(gr)), isDisjoint)))

```

intra-range-methods *Intra range transformations of a GenomicRanges object*

Description

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

This man page documents intra range transformations of a [GenomicRanges](#) object (i.e. of an object that belongs to the [GenomicRanges](#) class or one of its subclasses, this includes for example [GRanges](#) objects).

See ?‘[inter-range-methods](#)‘ for inter range transformations of a [GenomicRanges](#) object.

Usage

```

## S4 method for signature 'GenomicRanges'
shift(x, shift=0L, use.names=TRUE)

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

## S4 method for signature 'GenomicRanges'
flank(x, width, start=TRUE, both=FALSE,

```

```

        use.names=TRUE, ignore.strand=FALSE)

## S4 method for signature 'GenomicRanges'
promoters(x, upstream=2000, downstream=200, ...)

## S4 method for signature 'GenomicRanges'
resize(x, width, fix="start", use.names=TRUE,
       ignore.strand=FALSE)

## S4 method for signature 'GenomicRanges'
restrict(x, start=NA, end=NA, keep.all.ranges=FALSE,
        use.names=TRUE)

## S4 method for signature 'GenomicRanges'
trim(x, use.names=TRUE)

```

Arguments

`x` A [GenomicRanges](#) object.

`shift`, `use.names`, `start`, `end`, `width`, `both`, `fix`, `keep.all.ranges`, `upstream`, `downstream`
See ?'intra-range-methods'.

`ignore.strand` TRUE or FALSE. Whether the strand of the input ranges should be ignored or not.
See details below.

... Additional arguments to methods.

Details

- `shift` behaves like the `shift` method for [Ranges](#) objects. See ?'intra-range-methods' for the details.
- `()narrow` on [GenomicRanges](#) objects behaves like the `narrow` method for [Ranges](#) objects. See ?'intra-range-methods' for the details.
When `x` is a [GappedAlignments](#) object, `narrow(x)` returns a new [GappedAlignments](#) object the same length as `x` describing the narrowed alignments. Unlike with `qnarrow` now the `start/end/width` arguments describe the narrowing on the reference side, not the query side. Like with `qnarrow`, they must be 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).
- `flank` returns an object of the same type and length as `x` containing intervals of width `width` that flank the intervals in `x`. The `start` argument takes a logical indicating whether `x` should be flanked at the "start" (TRUE) or the "end" (FALSE), which for `strand(x) != "-"` is `start(x)` and `end(x)` respectively and for `strand(x) == "-"` is `end(x)` and `start(x)` respectively. The `both` argument takes a single logical value indicating whether the flanking region width positions extends *into* the range. If `both=TRUE`, the resulting range thus straddles the end point, with `width` positions on either side.
- `promoters` returns an object of the same type and length as `x` containing promoter ranges. Promoter ranges extend around the transcription start site (TSS) which is defined as `start(x)`.

The `upstream` and `downstream` arguments define the number of nucleotides in the 5' and 3' direction, respectively. The full range is defined as,

$(\text{start}(x) - \text{upstream})$ to $(\text{start}(x) + \text{downstream} - 1)$.

Ranges on the `*` strand are treated the same as those on the `+` strand. When no `seqlengths` are present in `x`, it is possible to have non-positive start values in the promoter ranges. This occurs when $(\text{TSS} - \text{upstream}) < 1$. In the equal but opposite case, the end values of the ranges may extend beyond the chromosome end when $(\text{TSS} + \text{downstream} + 1) > \text{'chromosome end'}$. When `seqlengths` are not `NA` the promoter ranges are kept within the bounds of the defined `seqlengths`.

- `resize` returns an object of the same type and length as `x` containing intervals that have been resized to width `width` based on the `strand(x)` values. Elements where `strand(x) == "+"` or `strand(x) == "*"` are anchored at `start(x)` and elements where `strand(x) == "-"` are anchored at the `end(x)`. The `use.names` argument determines whether or not to keep the names on the ranges.
- `restrict` returns an object of the same type and length as `x` containing restricted ranges for distinct `seqnames`. The `start` and `end` arguments can be a named numeric vector of `seqnames` for the ranges to be restricted or a numeric vector or length 1 if the restriction operation is to be applied to all the sequences in `x`. See ?'intra-range-methods' for more information about range restriction and for a description of the optional arguments.
- `trim` subsets the ranges in `x` to fall within the valid `seqlengths`. If no `seqlengths` are present, negative start values are set to 1L and end values are not touched.

Author(s)

P. Aboyoun and V. Obenchain <vobencha@fhcrc.org>

See Also

- The [GenomicRanges](#) and [GRanges](#) classes.
- The [Ranges](#) class in the [IRanges](#) package.
- The [intra-range-methods](#) man page in the [IRanges](#) package.

Examples

```
gr <- GRanges(
  seqnames=Rle(paste("chr", c(1, 2, 1, 3), sep=""), c(1, 3, 2, 4)),
  ranges=IRanges(1:10, width=10:1, names=letters[1:10]),
  strand=Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
  score=1:10,
  GC=seq(1, 0, length=10)
)
gr

shift(gr, 1)
narrow(gr[-10], start=2, end=-2)
flank(gr, 10)
resize(gr, 10)
restrict(gr, start=3, end=7)
```

```
gr <- GRanges("chr1", IRanges(rep(10, 3), width=6), c("+", "-", "*"))
promoters(gr, 2, 2)
```

makeSeqnameIds*Assigns a unique ID to each unique sequence name*

Description

Assign a unique ID to each unique sequence name. The returned IDs span 1:N where N is the number of unique sequence names provided.

Usage

```
makeSeqnameIds(seqnames, X.is.sexchrom=NA)
```

Arguments

seqnames A character vector or factor of sequence names.

X.is.sexchrom A logical indicating whether X refers to the sexual chromosome or to chromosome with Roman Numeral X.

Value

An integer vector the same length as seqnames. The values span 1:N where N is the number of unique sequence names provided.

Author(s)

H. Pages

Examples

```
library(BSgenome.Scerevisiae.UCSC.sacCer2)
makeSeqnameIds(seqnames(Scerevisiae))
makeSeqnameIds(seqnames(Scerevisiae)[c(1:5,5:1)])
```

map-methods

*Mapping ranges between sequences***Description**

The GenomicRanges package provides several methods for the `map` generic. They each translate a set of input ranges through a certain type of alignment and return a [RangesMapping](#) object.

Usage

```
## S4 method for signature 'GenomicRanges,GRangesList'
map(from, to)
## S4 method for signature 'GenomicRanges,GappedAlignments'
map(from, to)
```

Arguments

`from` The input ranges to map, usually a [GenomicRanges](#)
`to` The alignment between the sequences in `from` and the sequences in the result.

Details

The methods currently depend on the type of to:

GRangesList Each element is taken to represent an alignment of a sequence on a genome. The typical case is a set of transcript models, as might be obtained via `GenomicFeatures::exonsBy`. The method translates the input ranges to be relative to the transcript start. This is useful, for example, when predicting coding consequences of changes to the genomic sequence.

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

Value

An object of class `RangesMapping`. The GenomicRanges package provides some additional methods on this object:

`as(from, "GenomicRanges")`: Creates a [GenomicRanges](#) with `seqnames` and `ranges` from the space and `ranges` of `from`. The `hits` are coerced to a `DataFrame` and stored as the values of the result.

`granges(x)`: Like the above, except returns just the range information as a `GRanges`, without the matching information.

Author(s)

M. Lawrence

See Also

The [RangesMapping](#) class is the typical return value.

 nearest-methods

Finding the nearest genomic range neighbor

Description

The nearest, precede, follow, distance and distanceToNearest methods for [GenomicRanges](#) objects and subclasses.

Usage

```
## S4 method for signature 'GenomicRanges,GenomicRanges'
precede(x, subject, select=c("arbitrary", "all"), ignore.strand=FALSE, ...)
## S4 method for signature 'GenomicRanges,missing'
precede(x, subject, select=c("arbitrary", "all"), ignore.strand=FALSE, ...)

## S4 method for signature 'GenomicRanges,GenomicRanges'
follow(x, subject, select=c("arbitrary", "all"), ignore.strand=FALSE, ...)
## S4 method for signature 'GenomicRanges,missing'
follow(x, subject, select=c("arbitrary", "all"), ignore.strand=FALSE, ...)

## S4 method for signature 'GenomicRanges,GenomicRanges'
nearest(x, subject, select=c("arbitrary", "all"), ignore.strand=FALSE, ...)
## S4 method for signature 'GenomicRanges,missing'
nearest(x, subject, select=c("arbitrary", "all"), ignore.strand=FALSE, ...)

## S4 method for signature 'GenomicRanges,GenomicRanges'
distanceToNearest(x, subject, ignore.strand=FALSE, ...)
## S4 method for signature 'GenomicRanges,missing'
distanceToNearest(x, subject, ignore.strand=FALSE, ...)

## S4 method for signature 'GenomicRanges,GenomicRanges'
distance(x, y, ignore.strand=FALSE, ...)
```

Arguments

x	The query GenomicRanges instance.
subject	The subject GenomicRanges instance within which the nearest neighbors are found. Can be missing, in which case x is also the subject.
y	For the distance method, a GRanges instance. Cannot be missing. If x and y are not the same length, the shortest will be recycled to match the length of the longest.

<code>select</code>	Logic for handling ties. By default, all methods select a single interval (arbitrary for <code>nearest</code> , the first by order in <code>subject</code> for <code>precede</code> , and the last for <code>follow</code>). When <code>select="all"</code> a <code>Hits</code> object is returned with all matches for <code>x</code> . If <code>x</code> does not have a match in <code>subject</code> the <code>x</code> is not included in the <code>Hits</code> object.
<code>ignore.strand</code>	A logical indicating if the strand of the input ranges should be ignored. When <code>TRUE</code> , <code>strand</code> is set to <code>'+'</code> .
<code>...</code>	Additional arguments for methods.

Details

- `nearest`: Performs conventional nearest neighbor finding. Returns an integer vector containing the index of the nearest neighbor range in `subject` for each range in `x`. If there is no nearest neighbor `NA` is returned. For details of the algorithm see the man page in `IRanges`, `?nearest`.
- `precede`: For each range in `x`, `precede` returns the index of the range in `subject` that is directly preceded by the range in `x`. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in `subject`.
- `follow`: The opposite of `precede`, `follow` returns the index of the range in `subject` that is directly followed by the range in `x`. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in `subject`.
- **Orientation and Strand**: The relevant orientation for `precede` and `follow` is 5' to 3', consistent with the direction of translation. Because positional numbering along a chromosome is from left to right and transcription takes place from 5' to 3', `precede` and `follow` can appear to have 'opposite' behavior on the + and - strand. Using positions 5 and 6 as an example, 5 precedes 6 on the + strand but follows 6 on the - strand.
A range with `strand *` can be compared to ranges on either the + or - strand. Below we outline the priority when ranges on multiple strands are compared. When `ignore.strand=TRUE` all ranges are treated as if on the + strand.
 - `x` on + strand can match to ranges on both + and * strands. In the case of a tie the first range by order is chosen.
 - `x` on - strand can match to ranges on both - and * strands. In the case of a tie the first range by order is chosen.
 - `x` on * strand can match to ranges on any of +, - or * strands. In the case of a tie the first range by order is chosen.
- `distanceToNearest`: Returns the distance for each range in `x` to its nearest neighbor in the `subject`.
- `distance`: Returns the distance for each range in `x` to the range in `y`. The behavior of `distance` has changed in Bioconductor 2.12. See the man page `?distance` in `IRanges` for details.

Value

For `nearest`, `precede` and `follow`, an integer vector of indices in `subject`, or a `Hits` if `select="all"`.

For `distanceToNearest`, a `Hits` object with a column for the query index (`queryHits`), subject index (`subjectHits`) and the distance between the pair.

For `distance`, an integer vector of distances between the ranges in `x` and `y`.

Author(s)

P. Aboyoun and V. Obenchain <vobencha@fhcrc.org>

See Also

- The [GenomicRanges](#) and [GRanges](#) classes.
- The [Ranges](#) and [Hits](#) classes in the [IRanges](#) package.
- The [nearest-methods](#) man page in the [IRanges](#) package.
- [findOverlaps-methods](#) for finding just the overlapping ranges.

Examples

```
## -----
## precede() and follow()
## -----
query <- GRanges("A", IRanges(c(5, 20), width=1), strand="+")
subject <- GRanges("A", IRanges(rep(c(10, 15), 2), width=1),
                        strand=c("+", "+", "-", "-"))
precede(query, subject)
follow(query, subject)

strand(query) <- "-"
precede(query, subject)
follow(query, subject)

## ties choose first in order
query <- GRanges("A", IRanges(10, width=1), c("+", "-", "*"))
subject <- GRanges("A", IRanges(c(5, 5, 5, 15, 15, 15), width=1),
                        rep(c("+", "-", "*"), 2))
precede(query, subject)
precede(query, rev(subject))

## ignore.strand=TRUE treats all ranges as '+'
precede(query[1], subject[4:6], select="all", ignore.strand=FALSE)
precede(query[1], subject[4:6], select="all", ignore.strand=TRUE)

## -----
## nearest()
## -----
## When multiple ranges overlap an "arbitrary" range is chosen
query <- GRanges("A", IRanges(5, 15))
subject <- GRanges("A", IRanges(c(1, 15), c(5, 19)))
nearest(query, subject)

## select="all" returns all hits
nearest(query, subject, select="all")

## Ranges in 'x' will self-select when 'subject' is present
query <- GRanges("A", IRanges(c(1, 10), width=5))
nearest(query, query)
```

```

## Ranges in 'x' will not self-select when 'subject' is missing
nearest(query)

## -----
## distance(), distanceToNearest()
## -----
## Adjacent, overlap, separated by 1
query <- GRanges("A", IRanges(c(1, 2, 10), c(5, 8, 11)))
subject <- GRanges("A", IRanges(c(6, 5, 13), c(10, 10, 15)))
distance(query, subject)

## recycling
distance(query[1], subject)

## zero-width ranges
zw <- GRanges("A", IRanges(4,3))
stopifnot(distance(zw, GRanges("A", IRanges(3,4))) == 0L)
sapply(-3:3, function(i)
  distance(shift(zw, i), GRanges("A", IRanges(4,3))))

query <- GRanges(c("A", "B"), IRanges(c(1, 5), width=1))
distanceToNearest(query, subject)

```

phicoef

Calculate the "phi coefficient" between two binary variables

Description

The phicoef function calculates the "phi coefficient" between two binary variables.

Usage

```
phicoef(x, y=NULL)
```

Arguments

x, *y* Two logical vectors of the same length. If *y* is not supplied, *x* must be a 2x2 integer matrix (or an integer vector of length 4) representing the contingency table of two binary variables.

Value

The "phi coefficient" between the two binary variables. This is a single numeric value ranging from -1 to +1.

Author(s)

H. Pages

References

http://en.wikipedia.org/wiki/Phi_coefficient

Examples

```
set.seed(33)
x <- sample(c(TRUE, FALSE), 100, replace=TRUE)
y <- sample(c(TRUE, FALSE), 100, replace=TRUE)
phicoef(x, y)
phicoef(rep(x, 10), c(rep(x, 9), y))

stopifnot(phicoef(table(x, y)) == phicoef(x, y))
stopifnot(phicoef(y, x) == phicoef(x, y))
stopifnot(phicoef(x, !y) == - phicoef(x, y))
stopifnot(phicoef(x, x) == 1)
```

seqinfo

Accessing sequence information

Description

A set of generic functions for getting/setting sequence information from/on an object.

Usage

```
seqinfo(x)
seqinfo(x, new2old=NULL, force=FALSE) <- value
```

```
seqnames(x)
seqnames(x) <- value
```

```
seqlevels(x)
seqlevels(x, force=FALSE) <- value
```

```
seqlengths(x)
seqlengths(x) <- value
```

```
isCircular(x)
isCircular(x) <- value
```

```
genome(x)
genome(x) <- value
```

```
seqnameStyle(x)
seqnameStyle(x) <- value
```

Arguments

x	The object from/on which to get/set the sequence information.
new2old	<p>The new2old argument allows the user to rename, drop, add and/or reorder the "sequence levels" in x.</p> <p>new2old can be NULL or an integer vector with one element per row in Seqinfo object value (i.e. new2old and value must have the same length) describing how the "new" sequence levels should be mapped to the "old" sequence levels, that is, how the rows in value should be mapped to the rows in seqinfo(x). The values in new2old must be ≥ 1 and $\leq \text{length}(\text{seqinfo}(x))$. NAs are allowed and indicate sequence levels that are being added. Old sequence levels that are not represented in new2old will be dropped, but this will fail if those levels are in use (e.g. if x is a GRanges object with ranges defined on those sequence levels) unless force=TRUE is used (see below).</p> <p>If new2old=NULL, then sequence levels can only be added to the existing ones, that is, value must have at least as many rows as seqinfo(x) (i.e. $\text{length}(\text{values}) \geq \text{length}(\text{seqinfo}(x))$) and also $\text{seqlevels}(\text{values})[\text{seq_len}(\text{length}(\text{seqlevels}(x)))]$ must be identical to $\text{seqlevels}(x)$.</p>
force	Force dropping sequence levels currently in use. This is achieved by dropping the elements in x where those levels are used (hence typically reducing the length of x).
value	<p>Typically a Seqinfo object for the seqinfo setter.</p> <p>Either a named or unnamed character vector for the seqlevels setter.</p> <p>A vector containing the sequence information to store for the other setters.</p>

Details

Various classes implement methods for those generic functions.

The [Seqinfo](#) class plays a central role for those generics because:

- It has methods for all those generics (except seqinfo). That is, the seqnames, seqlevels, seqlengths, isCircular, genome and seqnameStyle getters and setters are defined for [Seqinfo](#) objects.
- For classes that implement it, the seqinfo getter should return a [Seqinfo](#) object.
- Default seqlevels, seqlengths, isCircular, genome, and seqnameStyle getters and setters are provided. By default, seqlevels(x) does seqlevels(seqinfo(x)), seqlengths(x) does seqlengths(seqinfo(x)), isCircular(x) does isCircular(seqinfo(x)), genome(x) does genome(seqinfo(x)), and seqnameStyle(x) does seqnameStyle(seqinfo(x)). So any class with a seqinfo getter will have all the above getters work out-of-the-box. If, in addition, the class defines a seqinfo setter, then all the corresponding setters will also work out-of-the-box.

See the [GRanges](#), [GRangesList](#), [GappedAlignments](#), and [GappedAlignmentPairs](#) classes for examples of classes that define the seqinfo getter and setter (those 4 classes are defined in the GenomicRanges package).

See the [TranscriptDb](#) class (defined in the GenomicFeatures package) for an example of a class that defines only the seqinfo getter (no setter).

The GenomicRanges package defines `seqinfo` and `seqinfo<-` methods for these low-level IRanges data structures: `List`, `RangesList` and `RangedData`. Those objects do not have the means to formally store sequence information. Thus, the wrappers simply store the `Seqinfo` object within `metadata(x)`. Initially, the metadata is empty, so there is some effort to generate a reasonable default `Seqinfo`. The names of any `List` are taken as the `seqnames`, and the universe of `RangesList` or `RangedData` is taken as the genome.

Note

The full list of methods defined for a given generic can be seen with e.g. `showMethods("seqinfo")` or `showMethods("seqnames")` (for the getters), and `showMethods("seqinfo<-")` or `showMethods("seqnames<-")` (for the setters aka *replacement methods*). Please be aware that this shows only methods defined in packages that are currently attached.

See Also

- [Seqinfo-class](#).
- [GRanges-class](#).
- [GRangesList-class](#).
- [GappedAlignments-class](#).
- [GappedAlignmentPairs-class](#).
- [TranscriptDb-class](#).

Examples

```
showMethods("seqinfo")
showMethods("seqinfo<-")
```

```
showMethods("seqnames")
showMethods("seqnames<-")
```

```
if (interactive())
  ?'GRanges-class'
```

Seqinfo-class

Seqinfo objects

Description

A `Seqinfo` object is a table-like object that contains basic information about a set of genomic sequences. The table has 1 row per sequence and 1 column per sequence attribute. Currently the only attributes are the length, circularity flag, and genome provenance (e.g. hg19) of the sequence, but more attributes might be added in the future as the need arises.

Details

Typically `Seqinfo` objects are not used directly but are part of higher level objects. Those higher level objects will generally provide a `seqinfo` accessor for getting/setting their `Seqinfo` component.

Constructor

`Seqinfo(seqnames, seqlengths=NA, isCircular=NA, genome=NA)`: Creates a Seqinfo object.

Accessor methods

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

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

`seqnames(x)`, `seqnames(x) <- value`: Get/set the names of the sequences in `x`. Those names must be non-NA, non-empty and unique. They are also called the *sequence levels* or the *keys* of the Seqinfo object.

Note that, in general, the end-user should not try to alter the sequence levels with `seqnames(x) <- value`. The recommended way to do this is with `seqlevels(x) <- value` as described below.

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

`seqlevels(x)`: Same as `seqnames(x)`.

`seqlevels(x) <- value`: Can be used to rename, drop, add and/or reorder the sequence levels. `value` must be either a named or unnamed character vector. When `value` has names, the names only serve the purpose of mapping the new sequence levels to the old ones. Otherwise (i.e. when `value` is unnamed) this mapping is implicitly inferred from the following rules:

(1) If the number of new and old levels are the same, and if the positional mapping between the new and old levels shows that some or all of the levels are being renamed, and if the levels that are being renamed are renamed with levels that didn't exist before (i.e. are not present in the old levels), then `seqlevels(x) <- value` will just rename the sequence levels. Note that in that case the result is the same as with `seqnames(x) <- value` but it's still recommended to use `seqlevels(x) <- value` as it is safer.

(2) Otherwise (i.e. if the conditions for (1) are not satisfied) `seqlevels(x) <- value` will consider that the sequence levels are not being renamed and will just perform `x <- x[value]`.

See below for some examples.

`seqlengths(x)`, `seqlengths(x) <- value`: Get/set the length for each sequence in `x`.

`isCircular(x)`, `isCircular(x) <- value`: Get/set the circularity flag for each sequence in `x`.

`genome(x)`, `genome(x) <- value`: Get/set the genome identifier or assembly name for each sequence in `x`.

`seqnameStyle(x)`: Get/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). The getter can return more than 1 seqname style (with a warning) in case the style cannot be determined unambiguously.

Subsetting

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

`x[i]`: A Seqinfo object can be subsetted only by name i.e. `i` must be a character vector. This is a convenient way to drop/add/reorder the rows (aka the sequence levels) of a Seqinfo object.

See below for some examples.

Coercion

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

`as.data.frame(x)`: Turns `x` into a data frame.

`as(x, "GRanges")`, `as(x, "GenomicRanges")`, `as(x, "RangesList")`: Turns `x` (with no NA lengths) into a `GRanges` or `RangesList`.

Combining Seqinfo objects

There are no `c` or `rbind` method for `Seqinfo` objects. Both would be expected to just append the rows in `y` to the rows in `x` resulting in an object of length `length(x) + length(y)`. But that would tend to break the constraint that the `seqnames` of a `Seqinfo` object must be unique keys.

So instead, a `merge` method is provided.

In the code snippet below, `x` and `y` are `Seqinfo` objects.

`merge(x, y)`: Merge `x` and `y` into a single `Seqinfo` object where the keys (aka the `seqnames`) are `union(seqnames(x), seqnames(y))`. If a row in `y` has the same key as a row in `x`, and if the 2 rows contain compatible information (NA values are compatible with anything), then they are merged into a single row in the result. If they cannot be merged (because they contain different `seqlengths`, and/or circularity flags, and/or genome identifiers), then an error is raised. In addition to check for incompatible sequence information, `merge(x, y)` also compares `seqnames(x)` with `seqnames(y)` and issues a warning if each of them has names not in the other. The purpose of these checks is to try to detect situations where the user might be combining or comparing objects based on different reference genomes.

`intersect(x, y)`: Finds the intersection between two `Seqinfo` objects by merging them and subsetting for the intersection of their sequence names. This makes it easy to avoid warnings about the objects not being subsets of each other during overlap operations.

Author(s)

H. Pages

See Also

[seqinfo](#)

Examples

```
## Note that all the arguments (except 'genome') must have the
## same length. 'genome' can be of length 1, whatever the lengths
## of the other arguments are.
x <- Seqinfo(seqnames=c("chr1", "chr2", "chr3", "chrM"),
             seqlengths=c(100, 200, NA, 15),
             isCircular=c(NA, FALSE, FALSE, TRUE),
             genome="toy")

length(x)
seqnames(x)
names(x)
seqlevels(x)
```

```

seqlengths(x)
isCircular(x)
genome(x)
seqnameStyle(x) # UCSC
seqnameStyle(x) <- "NCBI"
seqlevels(x)
seqnameStyle(x) # NCBI (ambiguous)
seqnameStyle(x) <- "UCSC"
seqlevels(x)

x[c("chrY", "chr3", "chr1")] # subset by names

## Rename, drop, add and/or reorder the sequence levels:
xx <- x
seqlevels(xx) <- sub("chr", "ch", seqlevels(xx)) # rename
xx
seqlevels(xx) <- rev(seqlevels(xx)) # reorder
xx
seqlevels(xx) <- c("ch1", "ch2", "chY") # drop/add/reorder
xx
seqlevels(xx) <- c(chY="Y", ch1="1", "22") # rename/reorder/drop/add
xx

y <- Seqinfo(seqnames=c("chr3", "chr4", "chrM"),
             seqlengths=c(300, NA, 15))
y
merge(x, y) # rows for chr3 and chrM are merged
suppressWarnings(merge(x, y))

## Note that, strictly speaking, merging 2 Seqinfo objects is not
## a commutative operation, i.e., in general 'z1 <- merge(x, y)'
## is not identical to 'z2 <- merge(y, x)'. However 'z1' and 'z2'
## are guaranteed to contain the same information (i.e. the same
## rows, but typically not in the same order):
suppressWarnings(merge(y, x))

## This contradicts what 'x' says about circularity of chr3 and chrM:
isCircular(y)[c("chr3", "chrM")] <- c(TRUE, FALSE)
y
if (interactive()) {
  merge(x, y) # raises an error
}

```

Description

Performs set operations on GRanges/GRangesList/GappedAlignments objects.

Usage

```

## Set operations
## S4 method for signature 'GRanges,GRanges'
union(x, y, ignore.strand=FALSE, ...)
## S4 method for signature 'GRanges,GRanges'
intersect(x, y, ignore.strand=FALSE, ...)
## S4 method for signature 'GRanges,GRanges'
setdiff(x, y, ignore.strand=FALSE, ...)

## Parallel set operations
## S4 method for signature 'GRanges,GRanges'
punion(x, y, fill.gap=FALSE, ignore.strand=FALSE, ...)
## S4 method for signature 'GRanges,GRanges'
pintersect(x, y, resolve.empty=c("none", "max.start", "start.x"), ignore.strand=FALSE, ...)
## S4 method for signature 'GappedAlignments,GRanges'
pintersect(x, y, ...)
## S4 method for signature 'GRanges,GRanges'
psetdiff(x, y, ignore.strand=FALSE, ...)

```

Arguments

<code>x, y</code>	<p>For <code>union</code>, <code>intersect</code>, <code>setdiff</code>, <code>pgap</code>: <code>x</code> and <code>y</code> must both be GRanges objects.</p> <p>For <code>punion</code>: one of <code>x</code> or <code>y</code> must be a GRanges object, the other one can be a GRanges or GRangesList object.</p> <p>For <code>pintersect</code>: one of <code>x</code> or <code>y</code> must be a GRanges object, the other one can be a GRanges, GRangesList or GappedAlignments object.</p> <p>For <code>psetdiff</code>: <code>x</code> and <code>y</code> can be any combination of GRanges and/or GRangesList objects, with the exception that if <code>x</code> is a GRangesList object then <code>y</code> must be a GRangesList too.</p> <p>In addition, for the "parallel" operations, <code>x</code> and <code>y</code> must be of equal length (i.e. <code>length(x) == length(y)</code>).</p>
<code>fill.gap</code>	Logical indicating whether or not to force a union by using the rule <code>start = min(start(x), start(y))</code> ,
<code>resolve.empty</code>	One of "none", "max.start", or "start.x" denoting how to handle ambiguous empty ranges formed by intersections. "none" - throw an error if an ambiguous empty range is formed, "max.start" - associate the maximum start value with any ambiguous empty range, and "start.x" - associate the start value of <code>x</code> with any ambiguous empty range. (See pintersect for the definition of an ambiguous range.)
<code>ignore.strand</code>	<p>For set operations: If set to <code>TRUE</code>, then the strand of <code>x</code> and <code>y</code> is set to "*" prior to any computation.</p> <p>For parallel set operations: If set to <code>TRUE</code>, the strand information is ignored in the computation and the result has the strand information of <code>x</code>.</p>
<code>...</code>	Further arguments to be passed to or from other methods.

Details

The pintersect methods involving [GRanges](#), [GRangesList](#) and/or [GappedAlignments](#) objects use the triplet (sequence name, range, strand) to determine the element by element intersection of features, where a strand value of "*" is treated as occurring on both the "+" and "-" strand.

The psetdiff methods involving [GRanges](#) and/or [GRangesList](#) objects use the triplet (sequence name, range, strand) to determine the element by element set difference of features, where a strand value of "*" is treated as occurring on both the "+" and "-" strand.

Value

For union, intersect, setdiff, and pgap: a [GRanges](#).

For punion and pintersect: when x or y is not a [GRanges](#) object, an object of the same class as this non-[GRanges](#) object. Otherwise, a [GRanges](#) object.

For psetdiff: either a [GRanges](#) object when both x and y are [GRanges](#) objects, or a [GRangesList](#) object when y is a [GRangesList](#) object.

Author(s)

P. Aboyoun

See Also

[setops-methods](#), [GRanges-class](#), [GRangesList-class](#), [GappedAlignments-class](#), [findOverlaps-methods](#)

Examples

```
## -----
## A. SET OPERATIONS
## -----

x <- GRanges("chr1", IRanges(c(2, 9) , c(7, 19)), strand=c("+", "-"))
y <- GRanges("chr1", IRanges(5, 10), strand="-")

union(x, y)
union(x, y, ignore.strand=TRUE)

intersect(x, y)
intersect(x, y, ignore.strand=TRUE)

setdiff(x, y)
setdiff(x, y, ignore.strand=TRUE)

## -----
## B. PARALLEL SET OPERATIONS
## -----

## Not run:
punion(x, shift(x, 7)) # will fail

## End(Not run)
```

```

union(x, shift(x, 7), fill.gap=TRUE)

pintersect(x, shift(x, 6))
## Not run:
pintersect(x, shift(x, 7)) # will fail

## End(Not run)
pintersect(x, shift(x, 7), resolve.empty="max.start")

psetdiff(x, shift(x, 7))

## -----
## C. MORE EXAMPLES
## -----

## GRanges object:
gr <- GRanges(seqnames=c("chr2", "chr1", "chr1"),
              ranges=IRanges(1:3, width = 12),
              strand=Rle(strand(c("-", "*", "-"))))

## GRangesList object
gr1 <- GRanges(seqnames="chr2",
              ranges=IRanges(3, 6))
gr2 <- GRanges(seqnames=c("chr1", "chr1"),
              ranges=IRanges(c(7,13), width = 3),
              strand=c("+", "-"))
gr3 <- GRanges(seqnames=c("chr1", "chr2"),
              ranges=IRanges(c(1, 4), c(3, 9)),
              strand=c("-", "-"))
grlist <- GRangesList(gr1=gr1, gr2=gr2, gr3=gr3)

## Parallel intersection of a GRanges and a GRangesList object
pintersect(gr, grlist)
pintersect(grlist, gr)

## Parallel intersection of a GappedAlignments and a GRanges object
library(Rsamtools) # because file ex1.bam is in this package
galn_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
galn <- readGappedAlignments(galn_file)
pintersect(galn, shift(as(galn, "GRanges"), 6L))

## Parallel set difference of a GRanges and a GRangesList object
psetdiff(gr, grlist)

## Parallel set difference of two GRangesList objects
psetdiff(grlist, shift(grlist, 3))

```

Description

Some useful strand methods.

Usage

```
## S4 method for signature 'missing'
strand(x)
## S4 method for signature 'character'
strand(x)
## S4 method for signature 'factor'
strand(x)
## S4 method for signature 'integer'
strand(x)
## S4 method for signature 'logical'
strand(x)
## S4 method for signature 'Rle'
strand(x)
## S4 method for signature 'DataTable'
strand(x)
## S4 replacement method for signature 'DataTable'
strand(x) <- value
```

Arguments

x	The object from which to obtain a strand factor, can be missing.
value	Replacement value for the strand.

Details

If x is missing, returns an empty factor with the standard levels that any strand factor should have: +, -, and *.

If x is a character vector or factor, it is coerced to a factor with the levels listed above.

If x is an integer vector, it is coerced to a factor with the levels listed above. 1 and -1 values in x are mapped to the + and - levels respectively. NAs in x produce NAs in the result.

If x is a logical vector, it is coerced to a factor with the levels listed above. FALSE and TRUE values in x are mapped to the + and - levels respectively. NAs in x produce NAs in the result.

If x is a 'logical'-Rle vector, it is transformed with `runValue(x) <- strand(runValue(x))` and returned.

If x inherits from DataTable, the "strand" column is returned as a factor with the levels listed above. If x has no "strand" column, this return value is populated with NAs.

Author(s)

Michael Lawrence

See Also

[strand](#)

Examples

```
strand()
strand(c("+", "-", NA, "*"))
strand(c(-1L, 1L, NA, -1L, NA))
strand(c(FALSE, FALSE, TRUE, NA, TRUE, FALSE))
strand(Rle(c(FALSE, FALSE, TRUE, NA, TRUE, FALSE)))
```

SummarizedExperiment-class

SummarizedExperiment instances

Description

The SummarizedExperiment class is a matrix-like container where rows represent ranges of interest (as a [GRanges](#) or [GRangesList](#)-class) and columns represent samples (with sample data summarized as a [DataFrame](#)-class). A SummarizedExperiment contains one or more assays, each represented by a matrix of numeric or other mode.

Usage

```
## Constructors
```

```
SummarizedExperiment(assays, ...)
## S4 method for signature 'SimpleList'
SummarizedExperiment(assays, rowData = GRangesList(),
  colData = DataFrame(), exptData = SimpleList(), ...,
  verbose = FALSE)
## S4 method for signature 'missing'
SummarizedExperiment(assays, ...)
## S4 method for signature 'list'
SummarizedExperiment(assays, ...)
## S4 method for signature 'matrix'
SummarizedExperiment(assays, ...)
```

```
## Accessors
```

```
assays(x, ..., withDimnames=TRUE)
assays(x, ...) <- value
assay(x, i, ...)
assay(x, i, ...) <- value
rowData(x, ...)
rowData(x, ...) <- value
colData(x, ...)
colData(x, ...) <- value
exptData(x, ...)
exptData(x, ...) <- value
```

```

## S4 method for signature 'SummarizedExperiment'
dim(x)
## S4 method for signature 'SummarizedExperiment'
dimnames(x)
## S4 replacement method for signature 'SummarizedExperiment,NULL'
dimnames(x) <- value
## S4 replacement method for signature 'SummarizedExperiment,list'
dimnames(x) <- value

## colData access

## S4 method for signature 'SummarizedExperiment'
x$name
## S4 replacement method for signature 'SummarizedExperiment,ANY'
x$name <- value
## S4 method for signature 'SummarizedExperiment,ANY,missing'
x[[i, j, ...]]
## S4 replacement method for signature 'SummarizedExperiment,ANY,missing,ANY'
x[[i, j, ...]] <- value

## rowData access
## see 'GRanges compatibility', below

## Subsetting

## S4 method for signature 'SummarizedExperiment'
x[i, j, ..., drop=TRUE]
## S4 replacement method for signature 'SummarizedExperiment,ANY,ANY,SummarizedExperiment'
x[i, j] <- value

## Combining

## S4 method for signature 'SummarizedExperiment'
cbind(..., deparse.level=1)
## S4 method for signature 'SummarizedExperiment'
rbind(..., deparse.level=1)

## Coercion

## S4 method for signature 'SummarizedExperiment'
updateObject(object, ..., verbose=FALSE)

```

Arguments

assays A list or SimpleList of matrix elements, or a matrix. All elements of the list must have the same dimensions, and dimension names (if present) must be consistent across elements and with the row names of rowData and colData.

rowData	A GRanges or GRangesList instance describing the ranges of interest. Row names, if present, become the row names of the SummarizedExperiment. The length of the GRanges or the GRangesList must equal the number of rows of the matrices in assays.
colData	An optional DataFrame describing the samples. Row names, if present, become the column names of the SummarizedExperiment.
exptData	An optional SimpleList of arbitrary content describing the overall experiment.
...	For SummarizedExperiment, S4 methods list and matrix, arguments identical to those of the SimpleList method. For assay, ... may contain withNames, which is forwarded to assays. For cbind, rbind, ... contains SummarizedExperiment objects to be combined. For other accessors, ignored.
verbose	A logical(1) indicating whether messages about data coercion during construction should be printed.
x, object	An instance of SummarizedExperiment-class.
i, j	For assay, assay<-, i is a integer or numeric scalar; see 'Details' for additional constraints. For [, SummarizedExperiment, [, SummarizedExperiment<-, i, j are instances that can act to subset the underlying rowData, colData, and matrix elements of assays. For [[, SummarizedExperiment, [[<-, SummarizedExperiment, i is a scalar index (e.g., character(1) or integer(1)) into a column of colData.
name	A symbol representing the name of a column of colData.
withDimnames	A logical(1), indicating whether dimnames should be applied to extracted assay elements.
drop	A logical(1), ignored by these methods.
value	An instance of a class specified in the S4 method signature or as outlined in 'Details'.
deparse.level	See ?base::cbind for a description of this argument.

Details

The SummarizedExperiment class is meant for numeric and other data types derived from a sequencing experiment. The structure is rectangular like a matrix, but with additional annotations on the rows and columns, and with the possibility to manage several assays simultaneously.

The rows of a SummarizedExperiment instance represent ranges (in genomic coordinates) of interest. The ranges of interest are described by a GRanges-class or a GRangesList-class instance, accessible using the rowData function, described below. The GRanges and GRangesList classes contains sequence (e.g., chromosome) name, genomic coordinates, and strand information. Each range can be annotated with additional data; this data might be used to describe the range or to summarize results (e.g., statistics of differential abundance) relevant to the range. Rows may or may not have row names; they often will not.

Each column of a `SummarizedExperiment` instance represents a sample. Information about the samples are stored in a `DataFrame`-class, accessible using the function `colData`, described below. The `DataFrame` must have as many rows as there are columns in the `SummarizedExperiment`, with each row of the `DataFrame` providing information on the sample in the corresponding column of the `SummarizedExperiment`. Columns of the `DataFrame` represent different sample attributes, e.g., tissue of origin, etc. Columns of the `DataFrame` can themselves be annotated (via the `mcols` function). Column names typically provide a short identifier unique to each sample.

A `SummarizedExperiment` can also contain information about the overall experiment, for instance the lab in which it was conducted, the publications with which it is associated, etc. This information is stored as a `SimpleList`-class, accessible using the `exptData` function. The form of the data associated with the experiment is left to the discretion of the user.

The `SummarizedExperiment` is appropriate for matrix-like data. The data are accessed using the `assays` function, described below. This returns a `SimpleList`-class instance. Each element of the list must itself be a matrix (of any mode) and must have dimensions that are the same as the dimensions of the `SummarizedExperiment` in which they are stored. Row and column names of each matrix must either be `NULL` or match those of the `SummarizedExperiment` during construction. It is convenient for the elements of `SimpleList` of assays to be named.

The `SummarizedExperiment` class has the following slots; this detail of class structure is not relevant to the user.

`exptData` A `SimpleList`-class instance containing information about the overall experiment.
`rowData` A `GRanges`-class instance defining the ranges of interest and associated metadata.
`colData` A `DataFrame`-class instance describing the samples and associated metadata.
`assays` A `SimpleList`-class instance, each element of which is a matrix summarizing data associated with the corresponding range and sample.

Constructor

Instances are constructed using the `SummarizedExperiment` function with arguments outlined above.

Coercion

Package version 1.9.59 introduced a new way of representing ‘assays’. If you have a serialized instance `x` of a `SummarizedExperiment` (e.g., from using the `save` function with a version of `GenomicRanges` prior to 1.9.59), it should be updated by invoking `x <- updateObject(x)`.

Accessors

In the following code snippets, `x` is a `SummarizedExperiment` instance.

`assays(x)`, `assays(x) <- value`: Get or set the assays. `value` is a list or `SimpleList`, each element of which is a matrix with the same dimensions as `x`.

`assay(x, i)`, `assay(x, i) <- value`: A convenient alternative (to `assays(x)[[i]]`, `assays(x)[[i]] <- value`) to get or set the `i`th (default first) assay element. `value` must be a matrix of the same dimension as `x`, and with dimension names `NULL` or consistent with those of `x`.

`rowData(x)`, `rowData(x) <- value`: Get or set the row data. `value` is a `GenomicRanges` instance. Row names of `value` must be `NULL` or consistent with the existing row names of `x`.

`colData(x)`, `colData(x) <- value`: Get or set the column data. `value` is a `DataFrame` instance. Row names of `value` must be `NULL` or consistent with the existing column names of `x`.

`exptData(x)`, `exptData(x) <- value`: Get or set the experiment data. `value` is a `list` or `SimpleList` instance, with arbitrary content.

`dim(x)`: Get the dimensions (ranges x samples) of the `SummarizedExperiment`.

`dimnames(x)`, `dimnames(x) <- value`: Get or set the dimension names. `value` is usually a list of length 2, containing elements that are either `NULL` or vectors of appropriate length for the corresponding dimension. `value` can be `NULL`, which removes dimension names. This method implies that `rownames`, `rownames<-`, `colnames`, and `colnames<-` are all available.

GRanges compatibility (rowData access)

Many `GRanges`-class and `GRangesList`-class operations are supported on ‘`SummarizedExperiment`’ and derived instances, using `rowData`.

Supported operations include: `compare`, `countOverlaps`, `coverage`, `disjointBins`, `distance`, `distanceToNearest`, `duplicated`, `end`, `end<-`, `findOverlaps`, `flank`, `follow`, `granges`, `isDisjoint`, `match`, `mcols`, `mcols<-`, `narrow`, `nearest`, `order`, `overlapsAny`, `precede`, `ranges`, `ranges<-`, `rank`, `resize`, `restrict`, `seqinfo`, `seqinfo<-`, `seqnames`, `seqselect`, `seqselect<-`, `shift`, `sort`, `split`, `splitAsListReturnedClass`, `start`, `start<-`, `strand`, `strand<-`, `subsetByOverlaps`, `width`, `width<-`.

Not all `GRanges`-class operations are supported, because they do not make sense for ‘`SummarizedExperiment`’ objects (e.g., `length`, `name`, `as.data.frame`, `c`, `splitAsList`), involve non-trivial combination or splitting of rows (e.g., `disjoin`, `gaps`, `reduce`, `unique`), or have not yet been implemented (`Ops`, `map`, `window`, `window<-`).

Subsetting

In the code snippets below, `x` is a `SummarizedExperiment` instance.

`x[i,j]`, `x[i,j] <- value`: Create or replace a subset of `x`. `i`, `j` can be numeric, logical, character, or missing. `value` must be a `SummarizedExperiment` instance with dimensions, dimension names, and assay elements consistent with the subset `x[i,j]` being replaced.

Additional subsetting accessors provide convenient access to `colData` columns

`x$name`, `x$name <- value` Access or replace column name in `x`.

`x[[i, ...]]`, `x[[i, ...]] <- value` Access or replace column `i` in `x`.

Combining

In the code snippets below, `...` are `SummarizedExperiment` instances to be combined.

`cbind(...)`, `rbind(...)`: `cbind` combines objects with identical ranges (`rowData`) but different samples (columns in assays). The `colnames` in `colData` must match or an error is thrown. Duplicate columns of `mcols(rowData(SummarizedExperiment))` must contain the same data.

`rbind` combines objects with different ranges (`rowData`) and the same subjects (columns in assays). Duplicate columns of `colData` must contain the same data.

`exptData` from all objects are combined into a `SimpleList` with no name checking.

Implementation and Extension

This section contains advanced material meant for package developers.

SummarizedExperiment is implemented as an S4 class, and can be extended in the usual way, using `contains="SummarizedExperiment"` in the new class definition.

In addition, the representation of the assays slot of SummarizedExperiment is as a virtual class Assays. This allows derived classes (`contains="Assays"`) to easily implement alternative requirements for the assays, e.g., backed by file-based storage like NetCDF or the `ff` package, while re-using the existing SummarizedExperiment class without modification. The requirements on Assays are list-like semantics (e.g., `sapply`, `[[` subsetting, `names`) with elements having matrix- or array-like semantics (e.g., `dim`, `dimnames`). These requirements can be made more precise if developers express interest.

The current assays slot is implemented as a reference class that has copy-on-change semantics. This means that modifying non-assay slots does not copy the (large) assay data, and at the same time the user is not surprised by reference-based semantics. Updates to non-assay slots are very fast; updating the assays slot itself can be 5x or more faster than with an S4 instance in the slot.

In a little more detail, a small reference class hierarchy (not exported from the GenomicRanges name space) defines a reference class ShallowData with a single field `data` of type ANY, and a derived class ShallowSimpleListAssays that specializes the type of data as SimpleList, and `contains=c("ShallowData", "Assays")`. The assays slot contains an instance of ShallowSimpleListAssays. Invoking `assays()` on a SummarizedExperiment re-dispatches from the assays slot to retrieve the SimpleList from the field of the reference class. This was achieved by implementing a generic (not exported) `value(x, name, ...)`, with a method implemented on SummarizedExperiment that retrieves a slot when `name` is a slot containing an S4 object in `x`, and a field when `name` is a slot containing a ShallowData instance in `x`. Copy-on-change semantics is maintained by implementing the `clone` method (`clone` methods are supposed to do a deep copy, update methods a shallow copy; the `clone` generic is introduced, and not exported, in the GenomicRanges package). The 'getter' and 'setter' code for methods implemented on SummarizedExperiment use `value` for slot access, and `clone` for replacement. This makes it easy to implement ShallowData instances for other slots if the need arises.

Author(s)

Martin Morgan, mtmorgan@fhcrc.org

See Also

[GRanges](#), [DataFrame](#), [SimpleList](#),

Examples

```
nrows <- 200; ncols <- 6
counts <- matrix(runif(nrows * ncols, 1, 1e4), nrows)
rowData <- GRanges(rep(c("chr1", "chr2"), c(50, 150)),
                  IRanges(floor(runif(200, 1e5, 1e6)), width=100),
                  strand=sample(c("+", "-"), 200, TRUE))
colData <- DataFrame(Treatment=rep(c("ChIP", "Input"), 3),
                    row.names=LETTERS[1:6])
sset <- SummarizedExperiment(assays=SimpleList(counts=counts),
```

```

                                rowData=rowData, colData=colData)
sset
assays(sset) <- endoapply(assays(sset), asinh)
head(assay(sset))

sset[, sset$Treatment == "ChIP"]

## cbind combines objects with the same ranges and different samples.
se1 <- sset
se2 <- se1[,1:3]
colnames(se2) <- letters[1:ncol(se2)]
cmb1 <- cbind(se1, se2)

## rbind combines objects with the same samples and different ranges.
se1 <- sset
se2 <- se1[1:50,]
rownames(se2) <- letters[1:nrow(se2)]
cmb2 <- rbind(se1, se2)

```

summarizeOverlaps

Perform overlap queries between reads and genomic features

Description

summarizeOverlaps extends findOverlaps by providing options to resolve reads that overlap multiple features. Each read is counted a maximum of once.

Usage

```

## S4 method for signature 'GRanges,GappedAlignments'
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ...)
## S4 method for signature 'GRangesList,GappedAlignments'
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ...)
## S4 method for signature 'GRanges,GappedAlignmentPairs'
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ...)
## S4 method for signature 'GRangesList,GappedAlignmentPairs'
summarizeOverlaps(
  features, reads, mode, ignore.strand=FALSE, ...)

```

Arguments

features A [GRanges](#) or a [GRangesList](#) containing genomic regions of interest. This will commonly be a [GRanges](#) of exons or transcripts or a [GRangesList](#) of exons by gene or transcripts by gene.

reads	<p>A GappedAlignments (single-end), GappedAlignmentPairs (paired-end), or BamFileList or BamViews objects containing the reads to be mapped to the genomic regions of interest.</p> <p>The BamFileList and BamViews methods can handle single or paired-end reads by appropriately specifying the <code>singleEnd</code> argument. Currently <code>singleEnd</code> is set once for all files in the list (i.e., the list cannot contain both single and paired-end read files.). For examples of iterating over Bam files See <code>?summarizeOverlaps, GRanges, BamFileLi</code></p>
mode	<p>Character name of a function that defines the counting method to be used. Modes include ‘Union’, ‘IntersectionStrict’, or ‘IntersectionNotEmpty’ and are designed after the counting modes in the HTSeq package by Simon Anders (see references). All methods are gap-aware and count a read a maximum of once. A user can provide their own count function as the <code>mode</code> argument when using the BamFileList or BamViews methods.</p> <ul style="list-style-type: none"> • Union : (Default) Reads that overlap any portion of exactly one feature are counted. Reads that overlap multiple features are discarded. For mode ‘Union’ gapped reads are handled the same as simple reads. If any portion of the gapped read hits >1 feature the read is discarded. The number of reads counted depends on the quality of the features (i.e., do the features overlap, have gaps, etc.). Of the three modes ‘Union’ tends to be the most conservative and often results in the lowest number of read counts. This is because the method does not attempt to resolve a read that hits multiple subjects but simply discards them. Both ‘IntersectionStrict’ and ‘IntersectionNotEmpty’ have rules that attempt to assign a ‘multi-hit read’ to a single feature. • IntersectionStrict : The read must fall completely within a single feature to be counted. A read can overlap multiple features but must fall within only one. In the case of gapped reads, all portions of the read fragment must fall within the same feature for the read to be counted. The fragments can overlap multiple features but collectively they must fall within only one. • IntersectionNotEmpty : For this counting mode, the features are partitioned into unique disjoint regions. This is accomplished by disjoining the feature ranges then removing ranges shared by more than one feature. The result is a group of non-overlapping regions each of which belong to a single feature. Simple and gapped reads are counted if, <ul style="list-style-type: none"> – the read or exactly 1 of the read fragments overlaps a unique disjoint region – the read or >1 read fragments overlap >1 unique disjoint region from the same feature
ignore.strand	A logical value indicating if strand should be considered when matching.
...	Additional arguments for other methods. If using multiple cores, you can pass arguments in here to be used by <code>mclapply</code> to indicate the number of cores to use etc.

Details

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`.

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 should be provided in a `GappedAlignmentPairs` container. Paired-end reads are counted the same as single-end reads with gaps.

The `BamFileList` and `BamViews` methods have an additional argument, `singleEnd`, to indicate if the bam files contain single or paired-end reads. See `?summarizeOverlaps,GRanges,BamFileList-method` for details.

Value

A `SummarizedExperiment` object. The assays slot holds the counts, rowData holds the features, colData will either be NULL or hold any metadata that was present in the reads.

Author(s)

Valerie Obenchain <vobencha@fhcrc.org>

References

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

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

See Also

- DESeq, DEXSeq and edgeR packages
- `BamFileList` and `BamViews` classes
- `GappedAlignments` and `GappedAlignmentPairs` classes
- `readGappedAlignments` and `readGappedAlignmentPairs`

Examples

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

```
## -----
## 'features' as GRanges
##
```

```
features <- 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)),
group_id = c("A", "B", "C", "C", "D", "D", "E", "F", "G", "H", "H"))

## Each row is a feature the read can 'hit'.
rowsAsFeatures <- data.frame(
  union = assays(summarizeOverlaps(features, reads))$counts,
  intStrict = assays(summarizeOverlaps(features, reads,
    mode="IntersectionStrict"))$counts,
  intNotEmpty = assays(summarizeOverlaps(features, reads,
    mode="IntersectionNotEmpty"))$counts,
  countOverlaps = countOverlaps(features, reads))

## Results from countOverlaps() are included to highlight how
## summarizeOverlaps() counts a read only once. For these 7
## reads, 'Union' is the most conservative counting mode, followed
## by 'Intersectionstrict' and then 'IntersectionNotEmpty'.

colSums(rowsAsFeatures)

## -----
## 'features' as GRangesList
##

## Each highest list-level is a feature the read can 'hit'.
lst <- split(features, mcols(features)[["group_id"]])
listAsFeatures <- data.frame(
  union = assays(summarizeOverlaps(lst, reads))$counts,
  intStrict = assays(summarizeOverlaps(lst, reads,
    mode="IntersectionStrict"))$counts,
  intNotEmpty = assays(summarizeOverlaps(lst, reads,
    mode="IntersectionNotEmpty"))$counts,
  countOverlaps = countOverlaps(lst, reads))

## -----
## 'reads' as BamFileList
##

## Examples of using summarizeOverlaps with Bam files are found
## at ?BamFile or '?summarizeOverlaps,GRanges,BamFileList-method'

## Count BAM files and prepare output for DESeq or edgeR analysis
library(Rsamtools)
fls <- list.files(system.file("extdata",package="GenomicRanges"),
  recursive=TRUE, pattern="*bam$", full=TRUE)
names(fls) <- basename(fls)
bf <- BamFileList(fls, index=character())
features <- 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)), "-",
        group_id=c(rep("A", 4), rep("B", 5), rep("C", 2)))

se <- summarizeOverlaps(features, bf)

library(DESeq)
deseq <- newCountDataSet(assays(se)$counts, rownames(colData(se)))

library(edgeR)
edger <- DGEList(assays(se)$counts, group=rownames(colData(se)))

## -----
## paired-end reads
##

## When paired-end reads are counted in a Bam file, the 'singleEnd'
## argument must be set to 'FALSE'. See the examples on the
## ?BamFile man page.

## Paired-end can also be counted in a GappedAlignmentPairs object.
## The readGappedAlignmentPairs() function outputs a GappedAlignmentPairs.
library("TxDb.Dmelanogaster.UCSC.dm3.ensGene")
exbygene <- exonsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene, "gene")
fl <- system.file("extdata", "untreated3_chr4.bam",
                  package="pasillaBamSubset")
reads <- readGappedAlignmentPairs(fl)
res <- summarizeOverlaps(exbygene, reads, "Union")
stopifnot(length(assays(res)$counts) == length(exbygene))

```

utils

seqlevels utility functions

Description

Rename or subset the seqlevels in a [GenomicRanges](#), [GRangesList](#) or [GappedAlignments](#) object.

Usage

```

## S4 method for signature 'GenomicRanges,character'
keepSeqlevels(x, value, ...)
## S4 method for signature 'GenomicRanges,character'
renameSeqlevels(x, value, ...)

```

Arguments

x The [GenomicRanges](#), [GRangesList](#) or [GappedAlignments](#) object in which the seqlevels will be removed or renamed.

value	For keepSeqlevels, a GRanges , GRangesList or GappedAlignments or character vector. x is subset on the seqlevels in value. For renameSeqlevels, a named character vector where the names are the 'old' and the values are the 'new' seqlevels.
...	Arguments passed to other functions.

Details

Many operations on [GRanges](#) objects require the seqlevels to match before a comparison can be made (e.g., `findOverlaps(type="within")`). `keepSeqlevels` and `renameSeqlevels` are convenience functions for subsetting and renaming the seqlevels of these objects.

`keepSeqlevels` subsets x based on the seqlevels provided in value. If value does not match any seqlevels in x the original x is returned. When x is a [GRangesList](#), there may be multiple chromosomes in a single list element. If not all chromosomes are specified in value, a reduced list element is returned. All empty list elements are dropped. See examples.

`renameSeqlevels` renames the seqlevels in x to those provided in value. value is a named character vector where the names are the 'old' seqlevel names and the values are the 'new'. If no names in value match the seqlevels in x the original x is returned.

Value

The x object with seqlevels removed or renamed. If x has no seqlevels (empty object) or no replacement values match the current seqlevels in x the unchanged x is returned.

Author(s)

Valerie Obenchain <vobencha@fhcrc.org>

See Also

[VCF](#)

Examples

```
gr1 <- GRanges(seqnames = c("chr1", "chr2"),
               ranges = IRanges(c(7,13), width = 3),
               strand = c("+", "-"), score = 3:4, GC = c(0.3, 0.5))
gr2 <- GRanges(seqnames = c("chr1", "chr1", "chr2", "chr3", "chr3"),
               ranges = IRanges(c(1, 4, 8, 9, 16), width=5),
               strand = "-", score = c(3L, 2L, 5L, 6L, 2L),
               GC = c(0.4, 0.1, 0.55, 0.20, 0.10))
gr3 <- GRanges(seqnames = c("CHROM4", "CHROM4"),
               ranges = IRanges(c(20, 45), width=6),
               strand = "+", score = c(2L, 5L), GC = c(0.30, 0.45))

## GRanges :
gr3_rename <- renameSeqlevels(gr3, c(CHROM4="chr4"))
gr3_rename

gr2_subset_chr <- keepSeqlevels(gr2, c("chr1", "chr2"))
```



```
gr2_subset_gr <- keepSeqlevels(gr2, gr1)
identical(gr2_subset_chr, gr2_subset_gr)

## GRangesList :
gr11 <- GRangesList("gr1" = gr1, "gr2" = gr2, "gr3" = gr3)
gr12 <- GRangesList("gr1" = gr1, "gr2" = gr2, "gr3" = gr3_rename)
gr11_rename <- renameSeqlevels(gr11, c(CHROM4="chr4"))
identical(gr11_rename, gr12)

gr11_subset <- keepSeqlevels(gr11, "chr3")

## GappedAlignments :
library(Rsamtools)
galn_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
galn <- readGappedAlignments(galn_file)

galn_rename <- renameSeqlevels(galn, c(seq2="chr2"))
galn_subset <- keepSeqlevels(galn_rename, gr1)
galn_subset

## See ?VCF for examples using renameSeqlevels and keepSeqlevels with
## VCF class objects
```

Index

*Topic **classes**

- Constraints, 7
- GAlignmentsList-class, 25
- GappedAlignmentPairs-class, 29
- GappedAlignments-class, 34
- Seqinfo-class, 69

*Topic **manip**

- cigar-utils, 2
- makeSeqnameIds, 61
- phicoef, 66

*Topic **methods**

- Constraints, 7
- coverage-methods, 14
- encodeOverlaps-methods, 16
- findOverlaps-methods, 18
- findSpliceOverlaps, 22
- GAlignmentsList-class, 25
- GappedAlignmentPairs-class, 29
- GappedAlignments-class, 34
- GenomicRanges-comparison, 40
- seqinfo, 67
- Seqinfo-class, 69
- setops-methods, 72
- strand-utils, 75
- summarizeOverlaps, 83
- utils, 87

*Topic **utilities**

- coverage-methods, 14
- encodeOverlaps-methods, 16
- findOverlaps-methods, 18
- findSpliceOverlaps, 22
- inter-range-methods, 55
- intra-range-methods, 58
- nearest-methods, 63
- setops-methods, 72
- summarizeOverlaps, 83
- utils, 87

<=, GenomicRanges, GenomicRanges-method
(GenomicRanges-comparison), 40

==, GenomicRanges, GenomicRanges-method
(GenomicRanges-comparison), 40

[, GAlignmentsList-method
(GAlignmentsList-class), 25

[, GRangesList-method
(GRangesList-class), 49

[, GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
29

[, GappedAlignments-method
(GappedAlignments-class), 34

[, GenomicRanges-method (GRanges-class),
44

[, Seqinfo-method (Seqinfo-class), 69

[, SummarizedExperiment-method
(SummarizedExperiment-class),
77

[<-, GAlignmentsList, ANY, ANY, ANY-method
(GAlignmentsList-class), 25

[<-, GRangesList, ANY, ANY, ANY-method
(GRangesList-class), 49

[<-, GenomicRanges, ANY, ANY, ANY-method
(GRanges-class), 44

[<-, SummarizedExperiment, ANY, ANY, SummarizedExperiment-meth
(SummarizedExperiment-class),
77

[[, GappedAlignmentPairs, ANY, ANY-method
(GappedAlignmentPairs-class),
29

[[, SummarizedExperiment, ANY, missing-method
(SummarizedExperiment-class),
77

[[<-, GAlignmentsList, ANY, ANY-method
(GAlignmentsList-class), 25

[[<-, GAlignmentsList-method
(GAlignmentsList-class), 25

[[<-, GRangesList, ANY, ANY-method
(GRangesList-class), 49

[[<-, GRangesList-method

- (GRangesList-class), 49
- [[<- , SummarizedExperiment, ANY, missing, ANY-method (SummarizedExperiment-class), 77
- [[<- , SummarizedExperiment, ANY, missing-method (SummarizedExperiment-class), 77
- \$, GenomicRanges-method (GRanges-class), 44
- \$, SummarizedExperiment-method (SummarizedExperiment-class), 77
- \$<- , GenomicRanges-method (GRanges-class), 44
- \$<- , SummarizedExperiment, ANY-method (SummarizedExperiment-class), 77
- \$<- , SummarizedExperiment-method (SummarizedExperiment-class), 77
- %in%, GAlignmentsList, GAlignmentsList-method (findOverlaps-methods), 18
- %in%, GAlignmentsList, Vector-method (findOverlaps-methods), 18
- %in%, GRangesList, GRangesList-method (findOverlaps-methods), 18
- %in%, GRangesList, GenomicRanges-method (findOverlaps-methods), 18
- %in%, GRangesList, RangedData-method (findOverlaps-methods), 18
- %in%, GRangesList, RangesList-method (findOverlaps-methods), 18
- %in%, GappedAlignmentPairs, Vector-method (findOverlaps-methods), 18
- %in%, GappedAlignments, GappedAlignments-method (findOverlaps-methods), 18
- %in%, GappedAlignments, Vector-method (findOverlaps-methods), 18
- %in%, GenomicRanges, GRangesList-method (findOverlaps-methods), 18
- %in%, GenomicRanges, GenomicRanges-method (GenomicRanges-comparison), 40
- %in%, GenomicRanges, RangedData-method (findOverlaps-methods), 18
- %in%, GenomicRanges, RangesList-method (findOverlaps-methods), 18
- %in%, RangedData, GRangesList-method (findOverlaps-methods), 18
- %in%, RangedData, GenomicRanges-method (findOverlaps-methods), 18
- %in%, RangesList, GRangesList-method (findOverlaps-methods), 18
- %in%, SummarizedExperiment, SummarizedExperiment-method (SummarizedExperiment-class), 77
- %in%, SummarizedExperiment, Vector-method (SummarizedExperiment-class), 77
- %in%, Vector, GAlignmentsList-method (findOverlaps-methods), 18
- %in%, Vector, GappedAlignmentPairs-method (findOverlaps-methods), 18
- %in%, Vector, GappedAlignments-method (findOverlaps-methods), 18
- %in%, Vector, SummarizedExperiment-method (SummarizedExperiment-class), 77
- as.data.frame, GAlignmentsList-method (GAlignmentsList-class), 25
- as.data.frame, GappedAlignments-method (GappedAlignments-class), 34
- as.data.frame, GenomicRanges-method (GRanges-class), 44
- as.data.frame, GRangesList-method (GRangesList-class), 49
- as.data.frame, Seqinfo-method (Seqinfo-class), 69
- assay (SummarizedExperiment-class), 77
- assay, SummarizedExperiment, ANY-method (SummarizedExperiment-class), 77
- assay, SummarizedExperiment, character-method (SummarizedExperiment-class), 77
- assay, SummarizedExperiment, missing-method (SummarizedExperiment-class), 77
- assay, SummarizedExperiment, numeric-method (SummarizedExperiment-class), 77
- assay<- (SummarizedExperiment-class), 77
- assay<- , SummarizedExperiment, character, matrix-method (SummarizedExperiment-class), 77

- assay<- , SummarizedExperiment, missing, matrix-method (SummarizedExperiment-class), 77
- assay<- , SummarizedExperiment, numeric, matrix-method (SummarizedExperiment-class), 77
- assays (SummarizedExperiment-class), 77
- assays, SummarizedExperiment-method (SummarizedExperiment-class), 77
- Assays-class (SummarizedExperiment-class), 77
- assays<- (SummarizedExperiment-class), 77
- assays<- , SummarizedExperiment, list-method (SummarizedExperiment-class), 77
- assays<- , SummarizedExperiment, SimpleList-method (SummarizedExperiment-class), 77

- BamFile, 23
- BamFileList, 84, 85
- BamViews, 84, 85

- c, GAlignmentsList-method (GAlignmentsList-class), 25
- c, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- c, GappedAlignments-method (GappedAlignments-class), 34
- c, GenomicRanges-method (GRanges-class), 44
- cbind, 79
- cbind, SummarizedExperiment-method (SummarizedExperiment-class), 77
- checkConstraint (Constraints), 7
- cigar (GappedAlignments-class), 34
- cigar, GAlignmentsList-method (GAlignmentsList-class), 25
- cigar, GappedAlignments-method (GappedAlignments-class), 34
- cigar-utils, 2
- cigarNarrow (cigar-utils), 2
- cigarOpTable (cigar-utils), 2
- cigarQNarrow (cigar-utils), 2
- cigarToCigarTable (cigar-utils), 2
- cigarToIRanges (cigar-utils), 2
- cigarToIRangesListByAlignment (cigar-utils), 2
- cigarToIRangesListByRName (cigar-utils), 2
- cigarToQWidth (cigar-utils), 2
- cigarToRleList (cigar-utils), 2
- cigarToWidth (cigar-utils), 2
- class:Constraint (Constraints), 7
- class:ConstraintORNULL (Constraints), 7
- class:GAlignmentsList (GAlignmentsList-class), 25
- class:GappedAlignmentPairs (GappedAlignmentPairs-class), 29
- class:GappedAlignments (GappedAlignments-class), 34
- class:GenomicRanges (GRanges-class), 44
- class:GenomicRangesList (GenomicRangesList-class), 43
- class:GRanges (GRanges-class), 44
- class:GRangesList (GRangesList-class), 49
- class:Seqinfo (Seqinfo-class), 69
- class:SimpleGenomicRangesList (GenomicRangesList-class), 43
- coerce, GAlignmentsList, GRanges-method (GAlignmentsList-class), 25
- coerce, GAlignmentsList, GRangesList-method (GAlignmentsList-class), 25
- coerce, GAlignmentsList, Ranges-method (GAlignmentsList-class), 25
- coerce, GAlignmentsList, RangesList-method (GAlignmentsList-class), 25
- coerce, GappedAlignmentPairs, GRanges-method (GappedAlignmentPairs-class), 29
- coerce, GappedAlignmentPairs, GRangesList-method (GappedAlignmentPairs-class), 29
- coerce, GappedAlignments, GRanges-method (GappedAlignments-class), 34
- coerce, GappedAlignments, GRangesList-method (GappedAlignments-class), 34
- coerce, GappedAlignments, Ranges-method (GappedAlignments-class), 34
- coerce, GappedAlignments, RangesList-method (GappedAlignments-class), 34

- (GappedAlignments-class), 34
- coerce, GenomicRanges, GappedAlignments-method (GRanges-class), 44
- coerce, GenomicRanges, RangedData-method (GRanges-class), 44
- coerce, GenomicRanges, RangesList-method (GRanges-class), 44
- coerce, GRangesList, CompressedIRangesList-method (GRangesList-class), 49
- coerce, GRangesList, IRangesList-method (GRangesList-class), 49
- coerce, RangedData, GRanges-method (GRanges-class), 44
- coerce, RangedDataList, GenomicRangesList-method (GenomicRangesList-class), 43
- coerce, RangedDataList, GRangesList-method (GRangesList-class), 49
- coerce, RangesList, GRanges-method (GRanges-class), 44
- coerce, RangesMapping, GenomicRanges-method (map-methods), 62
- coerce, RleList, GRanges-method (GRanges-class), 44
- coerce, RleViewsList, GRanges-method (GRanges-class), 44
- coerce, Seqinfo, GenomicRanges-method (Seqinfo-class), 69
- coerce, Seqinfo, GRanges-method (Seqinfo-class), 69
- coerce, Seqinfo, RangesList-method (Seqinfo-class), 69
- colData (SummarizedExperiment-class), 77
- colData, SummarizedExperiment-method (SummarizedExperiment-class), 77
- colData<- (SummarizedExperiment-class), 77
- colData<-, SummarizedExperiment, DataFrame-method (SummarizedExperiment-class), 77
- compare, 81
- compare, ANY, SummarizedExperiment-method (SummarizedExperiment-class), 77
- compare, GenomicRanges, GenomicRanges-method (GenomicRanges-comparison), 40
- compare, SummarizedExperiment, ANY-method (SummarizedExperiment-class), 77
- compare, SummarizedExperiment, SummarizedExperiment-method (SummarizedExperiment-class), 77
- CompressedIRangesList, 4, 37
- CompressedIRangesList-class, 38
- CompressedRleList, 5
- Constraint (Constraints), 7
- constraint (Constraints), 7
- Constraint-class (Constraints), 7
- constraint<- (Constraints), 7
- ConstraintORNULL (Constraints), 7
- ConstraintORNULL-class (Constraints), 7
- Constraints, 7
- countCompatibleOverlaps (encodeOverlaps-methods), 16
- countMatches, GAlignmentsList, GAlignmentsList-method (findOverlaps-methods), 18
- countMatches, GAlignmentsList, Vector-method (findOverlaps-methods), 18
- countMatches, GappedAlignmentPairs, Vector-method (findOverlaps-methods), 18
- countMatches, GappedAlignments, GappedAlignments-method (findOverlaps-methods), 18
- countMatches, GappedAlignments, Vector-method (findOverlaps-methods), 18
- countMatches, GenomicRanges, GenomicRanges-method (GenomicRanges-comparison), 40
- countMatches, GenomicRanges, GRangesList-method (findOverlaps-methods), 18
- countMatches, GenomicRanges, RangedData-method (findOverlaps-methods), 18
- countMatches, GenomicRanges, RangesList-method (findOverlaps-methods), 18
- countMatches, GRangesList, GenomicRanges-method (findOverlaps-methods), 18
- countMatches, GRangesList, GRangesList-method (findOverlaps-methods), 18
- countMatches, GRangesList, RangedData-method (findOverlaps-methods), 18
- countMatches, GRangesList, RangesList-method (findOverlaps-methods), 18
- countMatches, RangedData, GenomicRanges-method (findOverlaps-methods), 18
- countMatches, RangedData, GRangesList-method (findOverlaps-methods), 18
- countMatches, RangesList, GenomicRanges-method (findOverlaps-methods), 18

- countMatches, RangesList, GRangesList-method
(findOverlaps-methods), 18
- countMatches, Vector, GAlignmentsList-method
(findOverlaps-methods), 18
- countMatches, Vector, GappedAlignmentPairs-method
(findOverlaps-methods), 18
- countMatches, Vector, GappedAlignments-method
(findOverlaps-methods), 18
- countOverlaps, 81
- countOverlaps, GAlignmentsList, GAlignmentsList-method
(findOverlaps-methods), 18
- countOverlaps, GAlignmentsList, Vector-method
(findOverlaps-methods), 18
- countOverlaps, GappedAlignmentPairs, Vector-method
(findOverlaps-methods), 18
- countOverlaps, GappedAlignments, GappedAlignments-method
(findOverlaps-methods), 18
- countOverlaps, GappedAlignments, Vector-method
(findOverlaps-methods), 18
- countOverlaps, GenomicRanges, GenomicRanges-method
(findOverlaps-methods), 18
- countOverlaps, GenomicRanges, Vector-method
(findOverlaps-methods), 18
- countOverlaps, GRangesList, GRangesList-method
(findOverlaps-methods), 18
- countOverlaps, GRangesList, Vector-method
(findOverlaps-methods), 18
- countOverlaps, SummarizedExperiment, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- countOverlaps, SummarizedExperiment, Vector-method
(SummarizedExperiment-class), 77
- countOverlaps, Vector, GAlignmentsList-method
(findOverlaps-methods), 18
- countOverlaps, Vector, GappedAlignmentPairs-method
(findOverlaps-methods), 18
- countOverlaps, Vector, GappedAlignments-method
(findOverlaps-methods), 18
- countOverlaps, Vector, GenomicRanges-method
(findOverlaps-methods), 18
- countOverlaps, Vector, GRangesList-method
(findOverlaps-methods), 18
- countOverlaps, Vector, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- coverage, 5, 14, 15, 81
- coverage, GappedAlignmentPairs-method
(coverage-methods), 14
- coverage, GappedAlignments-method
(coverage-methods), 14
- coverage, GenomicRanges-method
(coverage-methods), 14
- coverage, GRangesList-method
(coverage-methods), 14
- coverage, SummarizedExperiment-method
(SummarizedExperiment-class),
coverage-methods, 14, 33, 38, 48, 54
- DataFrame, 4, 5, 44, 46, 47, 50, 77, 80, 82
- DataFrame-class, 48
- DataFrameList-class, 54
- DataTable, 46
- dim, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- dimnames, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- dimnames<-, SummarizedExperiment, list-method
(SummarizedExperiment-class), 77
- dimnames<-, SummarizedExperiment, NULL-method
(SummarizedExperiment-class), 77
- disjoin, GenomicRanges-method
(inter-range-methods), 55
- disjoin, GRangesList-method
(GRangesList-class), 49
- disjointBins, 81
- disjointBins, GenomicRanges-method
(inter-range-methods), 55
- disjointBins, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- distance, 81
- distance, ANY, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- distance, GenomicRanges, GenomicRanges-method
(nearest-methods), 63
- distance, SummarizedExperiment, ANY-method
(SummarizedExperiment-class), 77
- distance, SummarizedExperiment, SummarizedExperiment-method
(SummarizedExperiment-class),

- 77
- distanceToNearest, 81
- distanceToNearest, ANY, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- distanceToNearest, GenomicRanges, GenomicRanges-method
(nearest-methods), 63
- distanceToNearest, GenomicRanges, missing-method
(nearest-methods), 63
- distanceToNearest, SummarizedExperiment, ANY-method
(SummarizedExperiment-class),
77
- distanceToNearest, SummarizedExperiment, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- duplicate, 81
- duplicate, GenomicRanges-method
(GenomicRanges-comparison), 40
- duplicate, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- duplicate.GenomicRanges
(GenomicRanges-comparison), 40
- elementMetadata, GAlignmentsList-method
(GAlignmentsList-class), 25
- elementMetadata, GRangesList-method
(GRangesList-class), 49
- elementMetadata, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- elementMetadata<-, GAlignmentsList-method
(GAlignmentsList-class), 25
- elementMetadata<-, GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
29
- elementMetadata<-, GappedAlignments-method
(GappedAlignments-class), 34
- elementMetadata<-, GenomicRanges-method
(GRanges-class), 44
- elementMetadata<-, GRangesList-method
(GRangesList-class), 49
- elementMetadata<-, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- encodeOverlaps, 17
- encodeOverlaps, GRangesList, GRangesList-method
(encodeOverlaps-methods), 16
- encodeOverlaps-methods, 16
- end, 81
- end, GAlignmentsList-method
(GAlignmentsList-class), 25
- end, GappedAlignments-method
(GappedAlignments-class), 34
- end, GenomicRanges-method
(GRanges-class), 44
- end, GRangesList-method
(GRangesList-class), 49
- end, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- end<-, GenomicRanges-method
(GRanges-class), 44
- end<-, GRangesList-method
(GRangesList-class), 49
- end<-, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- exptData (SummarizedExperiment-class),
77
- exptData, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- exptData<-
(SummarizedExperiment-class),
77
- exptData<-, SummarizedExperiment, list-method
(SummarizedExperiment-class),
77
- exptData<-, SummarizedExperiment, SimpleList-method
(SummarizedExperiment-class),
77
- extractQueryStartInTranscript
(encodeOverlaps-methods), 16
- extractSkippedExonRanks
(encodeOverlaps-methods), 16
- extractSkippedExonRanks, character-method
(encodeOverlaps-methods), 16
- extractSkippedExonRanks, factor-method
(encodeOverlaps-methods), 16
- extractSkippedExonRanks, OverlapEncodings-method
(encodeOverlaps-methods), 16
- extractSpannedExonRanks
(encodeOverlaps-methods), 16
- extractSpannedExonRanks, character-method
(encodeOverlaps-methods), 16
- extractSpannedExonRanks, factor-method

- findOverlaps, SummarizedExperiment, Vector-method (SummarizedExperiment-class), 77
- findOverlaps, Vector, GAlignmentsList-method (findOverlaps-methods), 18
- findOverlaps, Vector, GappedAlignmentPairs-method (findOverlaps-methods), 18
- findOverlaps, Vector, GappedAlignments-method (findOverlaps-methods), 18
- findOverlaps, Vector, SummarizedExperiment-method (SummarizedExperiment-class), 77
- findOverlaps-methods, 18, 27, 33, 38, 42, 48, 54, 65, 74
- findSpliceOverlaps, 22
- findSpliceOverlaps, GappedAlignmentPairs, GRangesList-method (findSpliceOverlaps), 22
- findSpliceOverlaps, GappedAlignments, GRangesList-method (findSpliceOverlaps), 22
- findSpliceOverlaps, GRangesList, GRangesList-method (findSpliceOverlaps), 22
- findSpliceOverlaps-methods (findSpliceOverlaps), 22
- first (GappedAlignmentPairs-class), 29
- first, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- flank, 81
- flank, GenomicRanges-method (intra-range-methods), 58
- flank, GRangesList-method (GRangesList-class), 49
- flank, SummarizedExperiment-method (SummarizedExperiment-class), 77
- flipQuery (encodeOverlaps-methods), 16
- follow, 81
- follow, ANY, SummarizedExperiment-method (SummarizedExperiment-class), 77
- follow, GenomicRanges, GenomicRanges-method (nearest-methods), 63
- follow, GenomicRanges, missing-method (nearest-methods), 63
- follow, SummarizedExperiment, ANY-method (SummarizedExperiment-class), 77
- follow, SummarizedExperiment, SummarizedExperiment-method (SummarizedExperiment-class), 77
- GAlignmentsList, 18–20
- GAlignmentsList (GAlignmentsList-class), 25
- GAlignmentsList-class, 20, 25
- GappedAlignmentPairs, 14, 15, 17–20, 23, 24, 34, 68, 84, 85
- GappedAlignmentPairs (GappedAlignmentPairs-class), 29
- GappedAlignmentPairs-class, 15, 20, 27, 29, 38, 69
- GappedAlignments, 14, 15, 17–20, 23, 24, 30, 32, 68, 73, 74, 84, 85, 87
- GappedAlignments (GappedAlignments-class), 34
- GappedAlignments-class, 15, 20, 27, 33, 34, 69, 74
- gaps, 56
- gaps, GenomicRanges-method (inter-range-methods), 55
- genome (seqinfo), 67
- genome, ANY-method (seqinfo), 67
- genome, Seqinfo-method (Seqinfo-class), 69
- genome<- (seqinfo), 67
- genome<- , ANY-method (seqinfo), 67
- genome<- , Seqinfo-method (Seqinfo-class), 69
- GenomicRanges, 9, 40–42, 44, 55, 56, 58–60, 62, 63, 65, 87
- GenomicRanges (GRanges-class), 44
- GenomicRanges-class, 9
- GenomicRanges-class (GRanges-class), 44
- GenomicRanges-comparison, 40, 56
- GenomicRangesList (GenomicRangesList-class), 43
- GenomicRangesList-class, 43
- GenomicRangesORGRangesList-class (GRanges-class), 44
- GenomicRangesORmissing-class (GRanges-class), 44
- GRanges, 14, 18, 19, 23, 26, 32, 37, 55, 56, 58, 60, 65, 68, 73, 74, 77, 79–85, 88
- GRanges (GRanges-class), 44
- granges, 81
- granges (GappedAlignments-class), 34

- granges, GAlignmentsList-method
(GAlignmentsList-class), 25
- granges, GappedAlignmentPairs-method
(GappedAlignmentPairs-class), 29
- granges, GappedAlignments-method
(GappedAlignments-class), 34
- granges, RangesMapping-method
(map-methods), 62
- granges, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- GRanges-class, 15, 20, 33, 38, 44, 54, 69, 74
- GRangesList, 14–20, 23, 24, 27, 32, 37, 44, 47, 68, 73, 74, 77, 79, 81, 83, 85, 87, 88
- GRangesList (GRangesList-class), 49
- GRangesList-class, 15, 20, 33, 38, 48, 49, 69, 74
- grglist (GappedAlignments-class), 34
- grglist, GAlignmentsList-method
(GAlignmentsList-class), 25
- grglist, GappedAlignmentPairs-method
(GappedAlignmentPairs-class), 29
- grglist, GappedAlignments-method
(GappedAlignments-class), 34
- Hits, 17, 20, 23, 64, 65
- Hits-class, 20
- inter-range-methods, 42, 48, 55, 56
- intersect, GRanges, GRanges-method
(setops-methods), 72
- intersect, Seqinfo, Seqinfo-method
(Seqinfo-class), 69
- IntersectionNotEmpty
(summarizeOverlaps), 83
- IntersectionStrict (summarizeOverlaps), 83
- intra-range-methods, 42, 48, 58, 60
- introns (GappedAlignments-class), 34
- introns, GAlignmentsList-method
(GAlignmentsList-class), 25
- introns, GappedAlignmentPairs-method
(GappedAlignmentPairs-class), 29
- introns, GappedAlignments-method
(GappedAlignments-class), 34
- IRanges, 4, 14, 26, 37, 44
- IRanges-class, 5
- IRangesList, 4, 27, 50
- IRangesList-class, 5
- is, 9
- isCircular (seqinfo), 67
- isCircular, ANY-method (seqinfo), 67
- isCircular, Seqinfo-method
(Seqinfo-class), 69
- isCircular<- (seqinfo), 67
- isCircular<-, ANY-method (seqinfo), 67
- isCircular<-, Seqinfo-method
(Seqinfo-class), 69
- isCompatibleWithSkippedExons
(encodeOverlaps-methods), 16
- isCompatibleWithSkippedExons, character-method
(encodeOverlaps-methods), 16
- isCompatibleWithSkippedExons, factor-method
(encodeOverlaps-methods), 16
- isCompatibleWithSkippedExons, OverlapEncodings-method
(encodeOverlaps-methods), 16
- isCompatibleWithSplicing
(encodeOverlaps-methods), 16
- isCompatibleWithSplicing, character-method
(encodeOverlaps-methods), 16
- isCompatibleWithSplicing, factor-method
(encodeOverlaps-methods), 16
- isCompatibleWithSplicing, OverlapEncodings-method
(encodeOverlaps-methods), 16
- isDisjoint, 81
- isDisjoint, GenomicRanges-method
(inter-range-methods), 55
- isDisjoint, GRangesList-method
(GRangesList-class), 49
- isDisjoint, SummarizedExperiment-method
(SummarizedExperiment-class), 77
- isProperPair
(GappedAlignmentPairs-class), 29
- isProperPair, GappedAlignmentPairs-method
(GappedAlignmentPairs-class), 29
- keepSeqlevels (utils), 87
- keepSeqlevels, GappedAlignments, character-method
(utils), 87
- keepSeqlevels, GappedAlignments, GappedAlignments-method
(utils), 87

- keepSeqLevels, GappedAlignments, GenomicRanges-method (utils), [87](#)
- keepSeqLevels, GappedAlignments, GRangesList-method (utils), [87](#)
- keepSeqLevels, GenomicRanges, character-method (utils), [87](#)
- keepSeqLevels, GenomicRanges, GappedAlignments-method (utils), [87](#)
- keepSeqLevels, GenomicRanges, GenomicRanges-method (utils), [87](#)
- keepSeqLevels, GenomicRanges, GRangesList-method (utils), [87](#)
- keepSeqLevels, GRangesList, character-method (utils), [87](#)
- keepSeqLevels, GRangesList, GappedAlignments-method (utils), [87](#)
- keepSeqLevels, GRangesList, GenomicRanges-method (utils), [87](#)
- keepSeqLevels, GRangesList, GRangesList-method (utils), [87](#)

- lapply, [53](#)
- last (GappedAlignmentPairs-class), [29](#)
- last, GappedAlignmentPairs-method (GappedAlignmentPairs-class), [29](#)
- left (GappedAlignmentPairs-class), [29](#)
- left, GappedAlignmentPairs-method (GappedAlignmentPairs-class), [29](#)
- length, GappedAlignmentPairs-method (GappedAlignmentPairs-class), [29](#)
- length, GappedAlignments-method (GappedAlignments-class), [34](#)
- length, GenomicRanges-method (GRanges-class), [44](#)
- length, Seqinfo-method (Seqinfo-class), [69](#)
- List, [44](#)

- makeGAlignmentsListFromFeatureFragments (GAlignmentsList-class), [25](#)
- makeGappedAlignmentPairs, [31](#), [33](#)
- makeGRangesListFromFeatureFragments (GRangesList-class), [49](#)
- makeSeqnameIds, [61](#)
- map, [62](#)
- map, GenomicRanges, GappedAlignments-method (map-methods), [62](#)
- map, GenomicRanges, GRangesList-method (map-methods), [62](#)
- map-methods, [62](#)
- mapply, [53](#)
- match, [41](#), [81](#)
- match, GAlignmentsList, GAlignmentsList-method (findOverlaps-methods), [18](#)
- match, GAlignmentsList, Vector-method (findOverlaps-methods), [18](#)
- match, GappedAlignmentPairs, Vector-method (findOverlaps-methods), [18](#)
- match, GappedAlignments, GappedAlignments-method (findOverlaps-methods), [18](#)
- match, GappedAlignments, Vector-method (findOverlaps-methods), [18](#)
- match, GenomicRanges, GenomicRanges-method (GenomicRanges-comparison), [40](#)
- match, GenomicRanges, GRangesList-method (findOverlaps-methods), [18](#)
- match, GenomicRanges, RangedData-method (findOverlaps-methods), [18](#)
- match, GenomicRanges, RangesList-method (findOverlaps-methods), [18](#)
- match, GRangesList, GenomicRanges-method (findOverlaps-methods), [18](#)
- match, GRangesList, GRangesList-method (findOverlaps-methods), [18](#)
- match, GRangesList, RangedData-method (findOverlaps-methods), [18](#)
- match, GRangesList, RangesList-method (findOverlaps-methods), [18](#)
- match, RangedData, GenomicRanges-method (findOverlaps-methods), [18](#)
- match, RangedData, GRangesList-method (findOverlaps-methods), [18](#)
- match, RangesList, GenomicRanges-method (findOverlaps-methods), [18](#)
- match, RangesList, GRangesList-method (findOverlaps-methods), [18](#)
- match, SummarizedExperiment, SummarizedExperiment-method (SummarizedExperiment-class), [77](#)
- match, SummarizedExperiment, Vector-method (SummarizedExperiment-class), [77](#)
- match, Vector, GAlignmentsList-method

- (findOverlaps-methods), 18
- match, Vector, GappedAlignmentPairs-method (findOverlaps-methods), 18
- match, Vector, GappedAlignments-method (findOverlaps-methods), 18
- match, Vector, SummarizedExperiment-method (SummarizedExperiment-class), 77
- mcols, 80, 81
- mcols, SummarizedExperiment-method (SummarizedExperiment-class), 77
- mcols<-, SummarizedExperiment-method (SummarizedExperiment-class), 77
- merge, missing, Seqinfo-method (Seqinfo-class), 69
- merge, NULL, Seqinfo-method (Seqinfo-class), 69
- merge, Seqinfo, missing-method (Seqinfo-class), 69
- merge, Seqinfo, NULL-method (Seqinfo-class), 69
- merge, Seqinfo, Seqinfo-method (Seqinfo-class), 69

- names, 35
- names, GAlignmentsList-method (GAlignmentsList-class), 25
- names, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- names, GappedAlignments-method (GappedAlignments-class), 34
- names, GenomicRanges-method (GRanges-class), 44
- names, Seqinfo-method (Seqinfo-class), 69
- names<-, GAlignmentsList-method (GAlignmentsList-class), 25
- names<-, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- names<-, GappedAlignments-method (GappedAlignments-class), 34
- names<-, GenomicRanges-method (GRanges-class), 44
- names<-, Seqinfo-method (Seqinfo-class), 69
- narrow, 81
- narrow, GAlignmentsList-method (intra-range-methods), 58
- narrow, GappedAlignments-method (intra-range-methods), 58
- narrow, GenomicRanges-method (intra-range-methods), 58
- narrow, SummarizedExperiment-method (SummarizedExperiment-class), 77
- nearest, 81
- nearest, ANY, SummarizedExperiment-method (SummarizedExperiment-class), 77
- nearest, GenomicRanges, GenomicRanges-method (nearest-methods), 63
- nearest, GenomicRanges, missing-method (nearest-methods), 63
- nearest, SummarizedExperiment, ANY-method (SummarizedExperiment-class), 77
- nearest, SummarizedExperiment, missing-method (SummarizedExperiment-class), 77
- nearest, SummarizedExperiment, SummarizedExperiment-method (SummarizedExperiment-class), 77
- nearest-methods, 48, 63, 65
- ngap, GAlignmentsList-method (GAlignmentsList-class), 25
- ngap, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- ngap, GappedAlignments-method (GappedAlignments-class), 34

- Ops, GenomicRanges, numeric-method (intra-range-methods), 58
- order, 81
- order, GenomicRanges-method (GenomicRanges-comparison), 40
- order, SummarizedExperiment-method (SummarizedExperiment-class), 77
- OverlapEncodings, 17
- overlapsAny, 81
- overlapsAny, GAlignmentsList, GAlignmentsList-method (findOverlaps-methods), 18
- overlapsAny, GAlignmentsList, Vector-method (findOverlaps-methods), 18

- overlapsAny, GappedAlignmentPairs, Vector-method (findOverlaps-methods), 18
- overlapsAny, GappedAlignments, GappedAlignments-method (findOverlaps-methods), 18
- overlapsAny, GappedAlignments, Vector-method (findOverlaps-methods), 18
- overlapsAny, GenomicRanges, GenomicRanges-method (findOverlaps-methods), 18
- overlapsAny, GenomicRanges, GRangesList-method (findOverlaps-methods), 18
- overlapsAny, GenomicRanges, RangedData-method (findOverlaps-methods), 18
- overlapsAny, GenomicRanges, RangesList-method (findOverlaps-methods), 18
- overlapsAny, GRangesList, GenomicRanges-method (findOverlaps-methods), 18
- overlapsAny, GRangesList, GRangesList-method (findOverlaps-methods), 18
- overlapsAny, GRangesList, RangedData-method (findOverlaps-methods), 18
- overlapsAny, GRangesList, RangesList-method (findOverlaps-methods), 18
- overlapsAny, RangedData, GenomicRanges-method (findOverlaps-methods), 18
- overlapsAny, RangedData, GRangesList-method (findOverlaps-methods), 18
- overlapsAny, RangesList, GenomicRanges-method (findOverlaps-methods), 18
- overlapsAny, RangesList, GRangesList-method (findOverlaps-methods), 18
- overlapsAny, SummarizedExperiment, SummarizedExperiment (SummarizedExperiment-class), 77
- overlapsAny, SummarizedExperiment, Vector-method (SummarizedExperiment-class), 77
- overlapsAny, Vector, GAlignmentsList-method (findOverlaps-methods), 18
- overlapsAny, Vector, GappedAlignmentPairs-method (findOverlaps-methods), 18
- overlapsAny, Vector, GappedAlignments-method (findOverlaps-methods), 18
- overlapsAny, Vector, SummarizedExperiment-method (SummarizedExperiment-class), 77
- pgap, GRanges, GRanges-method (setops-methods), 72
- phicoef, 66
- pintersect, 73
- pintersect, GappedAlignments, GRanges-method (setops-methods), 72
- pintersect, GRanges, GappedAlignments-method (setops-methods), 72
- pintersect, GRanges, GRanges-method (setops-methods), 72
- pintersect, GRanges, GRangesList-method (setops-methods), 72
- pintersect, GRangesList, GRanges-method (setops-methods), 72
- pintersect, GRangesList, GRangesList-method (setops-methods), 72
- precede, 81
- precede, ANY, SummarizedExperiment-method (SummarizedExperiment-class), 77
- precede, GenomicRanges, GenomicRanges-method (nearest-methods), 63
- precede, GenomicRanges, missing-method (nearest-methods), 63
- precede, SummarizedExperiment, ANY-method (SummarizedExperiment-class), 77
- precede, SummarizedExperiment, SummarizedExperiment-method (SummarizedExperiment-class), 77
- promoters, GenomicRanges-method (intra-range-methods), 58
- promoters, GRangesList-method (GRangesList-class), 49
- psetdiff, GRanges, GRanges-method (setops-methods), 72
- psetdiff, GRanges, GRangesList-method (setops-methods), 72
- psetdiff, GRangesList, GRangesList-method (setops-methods), 72
- punion, GRanges, GRanges-method (setops-methods), 72
- punion, GRanges, GRangesList-method (setops-methods), 72
- punion, GRangesList, GRanges-method (setops-methods), 72
- qnarrow (GappedAlignments-class), 34
- qnarrow, GAlignmentsList-method (GAlignmentsList-class), 25
- qnarrow, GappedAlignments-method (GappedAlignments-class), 34

- queryLoc2refLoc (cigar-utils), 2
- queryLocs2refLocs (cigar-utils), 2
- qwidth (GappedAlignments-class), 34
- qwidth, GAlignmentsList-method (GAlignmentsList-class), 25
- qwidth, GappedAlignments-method (GappedAlignments-class), 34
- range, GenomicRanges-method (inter-range-methods), 55
- range, GRangesList-method (GRangesList-class), 49
- RangedData, 19
- RangedDataList, 50
- Ranges, 37, 56, 59, 60, 65
- ranges, 81
- ranges, GAlignmentsList-method (GAlignmentsList-class), 25
- ranges, GappedAlignments-method (GappedAlignments-class), 34
- ranges, GRanges-method (GRanges-class), 44
- ranges, GRangesList-method (GRangesList-class), 49
- ranges, SummarizedExperiment-method (SummarizedExperiment-class), 77
- Ranges-class, 48
- Ranges-comparison, 42
- ranges<-, GenomicRanges-method (GRanges-class), 44
- ranges<-, GRangesList-method (GRangesList-class), 49
- ranges<-, SummarizedExperiment-method (SummarizedExperiment-class), 77
- RangesList, 19, 37
- RangesList-class, 54
- RangesMapping, 62, 63
- rank, 81
- rank, GenomicRanges-method (GenomicRanges-comparison), 40
- rank, SummarizedExperiment-method (SummarizedExperiment-class), 77
- rbind, SummarizedExperiment-method (SummarizedExperiment-class), 77
- readBamGAlignmentsList, 25
- readBamGappedAlignmentPairs, 27, 30, 31, 33
- readBamGappedAlignments, 27, 35, 38
- readGAlignmentsList (GAlignmentsList-class), 25
- readGappedAlignmentPairs, 85
- readGappedAlignmentPairs (GappedAlignmentPairs-class), 29
- readGappedAlignments, 85
- readGappedAlignments (GappedAlignments-class), 34
- reduce, 56
- reduce, GenomicRanges-method (inter-range-methods), 55
- reduce, GRangesList-method (GRangesList-class), 49
- renameSeqlevels (utils), 87
- renameSeqlevels, GappedAlignments, character-method (utils), 87
- renameSeqlevels, GenomicRanges, character-method (utils), 87
- renameSeqlevels, GRangesList, character-method (utils), 87
- resize, 81
- resize, GenomicRanges-method (intra-range-methods), 58
- resize, SummarizedExperiment-method (SummarizedExperiment-class), 77
- restrict, 81
- restrict, GenomicRanges-method (intra-range-methods), 58
- restrict, GRangesList-method (GRangesList-class), 49
- restrict, SummarizedExperiment-method (SummarizedExperiment-class), 77
- rglist (GappedAlignments-class), 34
- rglist, GAlignmentsList-method (GAlignmentsList-class), 25
- rglist, GappedAlignments-method (GappedAlignments-class), 34
- right (GappedAlignmentPairs-class), 29
- right, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- Rle, 35, 36, 44, 45, 76

- Rle-class, 48
- RleList, 15
- RleList-class, 5, 15, 54
- rname (GappedAlignments-class), 34
- rname, GAlignmentsList-method (GAlignmentsList-class), 25
- rname, GappedAlignments-method (GappedAlignments-class), 34
- rname<- (GappedAlignments-class), 34
- rname<-, GAlignmentsList-method (GAlignmentsList-class), 25
- rname<-, GappedAlignments-method (GappedAlignments-class), 34
- rowData (SummarizedExperiment-class), 77
- rowData, SummarizedExperiment-method (SummarizedExperiment-class), 77
- rowData<- (SummarizedExperiment-class), 77
- rowData<-, SummarizedExperiment, GenomicRanges-method (SummarizedExperiment-class), 77
- rowData<-, SummarizedExperiment, GRangesList-method (SummarizedExperiment-class), 77

- sapply, 53
- score, GenomicRanges-method (GRanges-class), 44
- score, GRangesList-method (GRangesList-class), 49
- selectEncodingWithCompatibleStrand (encodeOverlaps-methods), 16
- Seqinfo, 26, 31, 36, 44, 46, 50, 68
- Seqinfo (Seqinfo-class), 69
- seqinfo, 33, 38, 48, 54, 67, 71, 81
- seqinfo, GAlignmentsList-method (GAlignmentsList-class), 25
- seqinfo, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- seqinfo, GappedAlignments-method (GappedAlignments-class), 34
- seqinfo, GRanges-method (GRanges-class), 44
- seqinfo, GRangesList-method (GRangesList-class), 49
- seqinfo, List-method (seqinfo), 67
- seqinfo, RangedData-method (seqinfo), 67
- seqinfo, RangesList-method (seqinfo), 67
- seqinfo, SummarizedExperiment-method (SummarizedExperiment-class), 77
- Seqinfo-class, 69, 69
- seqinfo<- (seqinfo), 67
- seqinfo<-, GAlignmentsList-method (GAlignmentsList-class), 25
- seqinfo<-, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- seqinfo<-, GappedAlignments-method (GappedAlignments-class), 34
- seqinfo<-, GenomicRanges-method (GRanges-class), 44
- seqinfo<-, GRangesList-method (GRangesList-class), 49
- seqinfo<-, List-method (seqinfo), 67
- seqinfo<-, RangedData-method (seqinfo), 67
- seqinfo<-, SummarizedExperiment-method (SummarizedExperiment-class), 77
- seqlengths (seqinfo), 67
- seqlengths, ANY-method (seqinfo), 67
- seqlengths, Seqinfo-method (Seqinfo-class), 69
- seqlengths<- (seqinfo), 67
- seqlengths<-, ANY-method (seqinfo), 67
- seqlengths<-, Seqinfo-method (Seqinfo-class), 69
- seqlevels, 31, 36, 46, 50
- seqlevels (seqinfo), 67
- seqlevels, ANY-method (seqinfo), 67
- seqlevels, Seqinfo-method (Seqinfo-class), 69
- seqlevels<- (seqinfo), 67
- seqlevels<-, ANY-method (seqinfo), 67
- seqlevels<-, Seqinfo-method (Seqinfo-class), 69
- seqnames, 81
- seqnames (seqinfo), 67
- seqnames, GAlignmentsList-method (GAlignmentsList-class), 25
- seqnames, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- seqnames, GappedAlignments-method

- (GappedAlignments-class), 34
- seqnames, GRanges-method
 - (GRanges-class), 44
- seqnames, GRangesList-method
 - (GRangesList-class), 49
- seqnames, RangedData-method (seqinfo), 67
- seqnames, RangesList-method (seqinfo), 67
- seqnames, Seqinfo-method
 - (Seqinfo-class), 69
- seqnames, SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
- seqnames<- (seqinfo), 67
- seqnames<-, GAlignmentsList-method
 - (GAlignmentsList-class), 25
- seqnames<-, GappedAlignments-method
 - (GappedAlignments-class), 34
- seqnames<-, GenomicRanges-method
 - (GRanges-class), 44
- seqnames<-, GRangesList-method
 - (GRangesList-class), 49
- seqnames<-, Seqinfo-method
 - (Seqinfo-class), 69
- seqnameStyle (seqinfo), 67
- seqnameStyle, ANY-method (seqinfo), 67
- seqnameStyle, Seqinfo-method
 - (Seqinfo-class), 69
- seqnameStyle<- (seqinfo), 67
- seqnameStyle<-, ANY-method (seqinfo), 67
- seqnameStyle<-, Seqinfo-method
 - (Seqinfo-class), 69
- seqselect, 81
- seqselect, GenomicRanges-method
 - (GRanges-class), 44
- seqselect, SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
- seqselect<-, GenomicRanges-method
 - (GRanges-class), 44
- seqselect<-, SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
- setClass, 9
- setdiff, GRanges, GRanges-method
 - (setops-methods), 72
- setMethod, 9
- setops-methods, 38, 42, 48, 54, 72, 74
- shift, 81
 - GenomicRanges-method
 - (intra-range-methods), 58
 - GRangesList-method
 - (GRangesList-class), 49
 - SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
 - GAlignmentsList-method
 - (GAlignmentsList-class), 25
 - GappedAlignmentPairs-method
 - (GappedAlignmentPairs-class), 29
 - GappedAlignments-method
 - (GappedAlignments-class), 34
 - GRanges-method
 - (GRanges-class), 44
 - GRangesList-method
 - (GRangesList-class), 49
 - Seqinfo-method (Seqinfo-class), 69
 - SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
- showMethods, 9
- SimpleGenomicRangesList-class
 - (GenomicRangesList-class), 43
- SimpleList, 80, 82
- solveUserSEW, 3, 27, 37, 59
- sort, 81
 - SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
- split, SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
- splitAsListReturnedClass, GappedAlignments-method
 - (GappedAlignments-class), 34
- splitAsListReturnedClass, GRRange
 - (GRanges-class), 44
- splitAsListReturnedClass, GRanges-method
 - (GRanges-class), 44
- splitAsListReturnedClass, SummarizedExperiment-method
 - (SummarizedExperiment-class), 77
- splitCigar (cigar-utils), 2
- start, 81
 - GAlignmentsList-method
 - (GAlignmentsList-class), 25
 - GappedAlignments-method

- (GappedAlignments-class), 34
- start, GenomicRanges-method
(GRanges-class), 44
- start, GRangesList-method
(GRangesList-class), 49
- start, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- start<-, GenomicRanges-method
(GRanges-class), 44
- start<-, GRangesList-method
(GRangesList-class), 49
- start<-, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- strand, 44, 76, 81
- strand, character-method (strand-utils),
75
- strand, DataTable-method (strand-utils),
75
- strand, factor-method (strand-utils), 75
- strand, GAlignmentsList-method
(GAlignmentsList-class), 25
- strand, GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
29
- strand, GappedAlignments-method
(GappedAlignments-class), 34
- strand, GRanges-method (GRanges-class),
44
- strand, GRangesList-method
(GRangesList-class), 49
- strand, integer-method (strand-utils), 75
- strand, logical-method (strand-utils), 75
- strand, missing-method (strand-utils), 75
- strand, Rle-method (strand-utils), 75
- strand, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- strand-utils, 75
- strand<-, DataTable-method
(strand-utils), 75
- strand<-, GAlignmentsList-method
(GAlignmentsList-class), 25
- strand<-, GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
29
- strand<-, GappedAlignments-method
(GappedAlignments-class), 34
- strand<-, GenomicRanges-method
(GRanges-class), 44
- strand<-, GRangesList-method
(GRangesList-class), 49
- strand<-, SummarizedExperiment-method
(SummarizedExperiment-class),
77
- subsetByOverlaps, 81
- subsetByOverlaps, GAlignmentsList, GAlignmentsList-method
(findOverlaps-methods), 18
- subsetByOverlaps, GAlignmentsList, Vector-method
(findOverlaps-methods), 18
- subsetByOverlaps, GappedAlignmentPairs, Vector-method
(findOverlaps-methods), 18
- subsetByOverlaps, GappedAlignments, GappedAlignments-method
(findOverlaps-methods), 18
- subsetByOverlaps, GappedAlignments, Vector-method
(findOverlaps-methods), 18
- subsetByOverlaps, GenomicRanges, GenomicRanges-method
(findOverlaps-methods), 18
- subsetByOverlaps, GenomicRanges, GRangesList-method
(findOverlaps-methods), 18
- subsetByOverlaps, GenomicRanges, RangedData-method
(findOverlaps-methods), 18
- subsetByOverlaps, GenomicRanges, RangesList-method
(findOverlaps-methods), 18
- subsetByOverlaps, GRangesList, GenomicRanges-method
(findOverlaps-methods), 18
- subsetByOverlaps, GRangesList, GRangesList-method
(findOverlaps-methods), 18
- subsetByOverlaps, GRangesList, RangedData-method
(findOverlaps-methods), 18
- subsetByOverlaps, GRangesList, RangesList-method
(findOverlaps-methods), 18
- subsetByOverlaps, RangedData, GenomicRanges-method
(findOverlaps-methods), 18
- subsetByOverlaps, RangedData, GRangesList-method
(findOverlaps-methods), 18
- subsetByOverlaps, RangesList, GenomicRanges-method
(findOverlaps-methods), 18
- subsetByOverlaps, RangesList, GRangesList-method
(findOverlaps-methods), 18
- subsetByOverlaps, SummarizedExperiment, SummarizedExperiment
(SummarizedExperiment-class),
77
- subsetByOverlaps, SummarizedExperiment, Vector-method
(SummarizedExperiment-class),

- 77
- subsetByOverlaps, Vector, GAlignmentsList-method (findOverlaps-methods), 18
- subsetByOverlaps, Vector, GappedAlignmentPairs-method (findOverlaps-methods), 18
- subsetByOverlaps, Vector, GappedAlignments-method (findOverlaps-methods), 18
- subsetByOverlaps, Vector, SummarizedExperiment-method (SummarizedExperiment-class), 77
- summarizeCigarTable (cigar-utils), 2
- SummarizedExperiment, 85
- SummarizedExperiment (SummarizedExperiment-class), 77
- SummarizedExperiment, list-method (SummarizedExperiment-class), 77
- SummarizedExperiment, matrix-method (SummarizedExperiment-class), 77
- SummarizedExperiment, missing-method (SummarizedExperiment-class), 77
- SummarizedExperiment, SimpleList-method (SummarizedExperiment-class), 77
- SummarizedExperiment-class, 77
- summarizeOverlaps, 83
- summarizeOverlaps, GRanges, GappedAlignmentPairs-method (summarizeOverlaps), 83
- summarizeOverlaps, GRanges, GappedAlignments-method (summarizeOverlaps), 83
- summarizeOverlaps, GRangesList, GappedAlignmentPairs-method (summarizeOverlaps), 83
- summarizeOverlaps, GRangesList, GappedAlignments-method (summarizeOverlaps), 83
- TranscriptDb, 68
- TranscriptDb-class, 69
- trim, GenomicRanges-method (intra-range-methods), 58
- Union (summarizeOverlaps), 83
- union, GRanges, GRanges-method (setops-methods), 72
- unlist, GappedAlignmentPairs-method (GappedAlignmentPairs-class), 29
- updateObject, GAlignmentsList-method (GAlignmentsList-class), 25
- updateObject, GappedAlignments-method (GappedAlignments-class), 34
- updateObject, GRanges-method (GRanges-class), 44
- updateObject, GRangesList-method (GRangesList-class), 49
- updateObject, Seqinfo-method (Seqinfo-class), 69
- updateObject, SummarizedExperiment-method (SummarizedExperiment-class), 77
- utils, 87
- validCigar (cigar-utils), 2
- validObject, 9
- values, SummarizedExperiment-method (SummarizedExperiment-class), 77
- values<-, SummarizedExperiment-method (SummarizedExperiment-class), 77
- VCF, 88
- Vector-class, 48, 54
- Views, 47, 52
- width, 81
- width, GAlignmentsList-method (GAlignmentsList-class), 25
- width, GappedAlignments-method (GappedAlignments-class), 34
- width, GenomicRanges-method (GRanges-class), 44
- width, GRangesList-method (GRangesList-class), 49
- width, SummarizedExperiment-method (SummarizedExperiment-class), 77
- width<-, GenomicRanges-method (GRanges-class), 44
- width<-, GRangesList-method (GRangesList-class), 49
- width<-, SummarizedExperiment-method (SummarizedExperiment-class), 77
- window, GenomicRanges-method (GRanges-class), 44