Package 'GenomicRanges'

September 24, 2012

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 1.8.13

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 (>= 0.1.12), IRanges (>= 1.13.36)

Imports methods, BiocGenerics, IRanges

LinkingTo IRanges

Suggests Rsamtools (>= 1.7.42), BSgenome, rtracklayer,GenomicFeatures, EatonEtAlChIPseq (>= 0.0.3), leeBamViews,edgeR, DESeq, rtracklayer, org.Sc.sgd.db,BSgenome.Scerevisiae.UCSC.sacCer2, DEXSeq, pasilla,pasillaBamSubset, VariantAnnotation, RUnit

License Artistic-2.0

Collate utils.R cigar-utils.R transcript-utils.R constraint.R seqinfo.R strand-utils.R Seqinfo-class.R GenomicRanges-class.R GenomicRangesList-class.R GenomicRanges-comparison.R GRanges-class.R GRangesList-class.R GappedAlignments-class.R GappedAlignmentPairs-class.R SummarizedExperiment-class.R coverage-methods.R setops-methods.R findOverlaps-methods.R encodeOverlaps-methods.R countGenomicOverlaps.R seqlevels-utils.R resolveHits-methods.R summarizeOverlaps.R RangesMapping-methods.R RangedData-methods.R test_GenomicRanges_package.R zzz.R

R topics documented:

cigar-utils . . . 2
Constraints . . . 7
countGenomicOverlaps . . . 12
coverage-methods . . . 15
encodeOverlaps-methods . . . 17
findOverlaps-methods . . . 19
GappedAlignmentPairs-class . . . 22
GappedAlignments-class . . . 25
GenomicRanges-comparison . . . 31
GenomicRangesList-class . . . 32
GRanges-class . . . 32
GRangesList-class . . . 39
map-methods . . . 43
seqinfo . . . 45
Seqinfo-class . . . 47
setops-methods . . . 49
strand-utils . . . 52
SummarizedExperiment-class . . . 53
summarizeOverlaps . . . 57
utils . . . 60

Index **63**

`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 `cigarToIRanges` and `queryLoc2refLoc`, 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 (`before.hard.clipping=FALSE`, the default), then the returned widths are the lengths of the query sequences stored in the SAM/BAM file. If YES (`before.hard.clipping=TRUE`), 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 `drop.D.ranges` 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, `cigarToIRangesListByAlignment` and `cigarToIRangesListByRName` 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 `cigarToCigarTable`.

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 sligthly
  ## 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)
```

`Constraints` *Enforcing constraints thru Constraint objects*

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 developper, 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 >= 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" column in its elementMetadata part 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)
    {
        emd <- elementMetadata(x)
        idx <- match("rangeType", colnames(emd))
        if (length(idx) != 1L || is.na(idx)) {
            msg <- c("'elementMetadata(x)' must have exactly 1 column ",
                     "named \"rangeType\"")
            return(paste(msg, collapse=""))
        }
        rangeType <- emd[[idx]]
        .LEVELS <- c("gene", "transcript", "exon", "cds", "5utr", "3utr")
        if (!is(rangeType, "Rle") ||
            IRanges:::anyMissing(runValue(rangeType)) ||
            !identical(levels(rangeType), .LEVELS))
        {
            msg <- c("'elementMetadata(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))
```

```
elementMetadata(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{
#elementMetadata(gr)$rangeType[3] <- NA  # will fail
#}
elementMetadata(gr)$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 statisfied, it will then check the GeneRanges
## constraint.
setMethod("checkConstraint", c("GenomicRanges", "GeneRanges"),
    function(x, constraint, verbose=FALSE)
    {
        rangeType <- elementMetadata(x)$rangeType
        if (!all(rangeType == "gene")) {
            msg <- c("all elements in 'elementMetadata(x)$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

elementMetadata(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{
#elementMetadata(gr)$rangeType[] <- "exon"  # will fail
#}
elementMetadata(gr)$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" column in its elementMetadata part, 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)
    {
        emd <- elementMetadata(x)
        idx <- match("GC", colnames(emd))
        if (length(idx) != 1L || is.na(idx)) {
            msg <- c("'elementMetadata(x)' must have exactly ",
                     "one column named \"GC\"")
            return(paste(msg, collapse=""))
        }
        GC <- emd[[idx]]
        if (!is.numeric(GC) ||
            IRanges:::anyMissing(GC) ||
            any(GC < 0) || any(GC > 1))
        {
            msg <- c("'elementMetadata(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 statisfied, it will then check the HighGCRanges
## constraint.
setMethod("checkConstraint", c("GenomicRanges", "HighGCRanges"),
    function(x, constraint, verbose=FALSE)
    {
```

```
        GC <- elementMetadata(x)$GC
        if (!all(GC >= 0.5)) {
            msg <- c("all elements in 'elementMetadata(x)$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
elementMetadata(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 HighGCGeneRanges constraint.
## ---------------------------------------------------------------------
## The HighGCGeneRanges constraint is the combination (AND) of the
## GeneRanges and HighGCRanges constraints.

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

#is(constraint(gr), "HighGCGeneRanges")  # 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("HighGCGeneRanges"), verbose=TRUE)  # with
                                                            # GenomicRanges
                                                            # >= 1.7.9

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

---

countGenomicOverlaps *Count Read Hits in Genomic Features*

Description

Count read hits per exon or transcript and resolve multi-hit reads.

WARNING: `countGenomicOverlaps` is deprecated in favor of `summarizeOverlaps`.

Usage

```
  ## S4 method for signature 'GRangesList,GRangesList'
countGenomicOverlaps(
    query, subject,
    type = c("any", "start", "end", "within", "equal"),
    resolution = c("none", "divide", "uniqueDisjoint"),
    ignore.strand = FALSE, splitreads = TRUE, ...)
```

Arguments

`query`	A GRangesList, or a GRanges of genomic features. These are the annotations that define the genomic regions and will often be the result of calling "exonsBy" or "transcriptsBy" on a TranscriptDb object. If a GRangesList is provided, each top level of the list represents a "super" such as a gene and each row is a "sub" such as an exon or transcript. When `query` is a GRanges all rows are considered to be of the same level (e.g., all genes, all exons or all transcripts).
`subject`	A GRangesList, GRanges, or GappedAlignments representing the data (e.g., reads). List structures as the `subject` are used to represent reads with multiple parts (i.e., gaps in the CIGAR). When a GappedAlignments is provided it is coerced to a GRangesList object. If any of the reads in the GappedAlignments have gaps, the corresponding GRangesList will have multiple elements for that top level list. When `subject` is a GRanges, it is assumed that all reads are simple and do not have multiple parts.
`type`	See `findOverlaps` in the IRanges package for a description of this argument.
`resolution`	A `character(1)` string of "none", "divide", or "uniqueDisjoint". These rule sets are used to distribute read hits when multiple queries are hit by the same subject. • "none" : No conflict resolution is performed. All subjects that hit more than 1 query are dropped. • "divide" : The hit from a single subject is divided equally among all queries that were hit. If a subject hit 4 queries each query is assigned 1/4 of a hit. • "uniqueDisjoint" : Queries hit by a common subject are partitioned into disjoint intervals. Any regions that are shared between the queries are discarded. If the read overlaps one of these remaining unique disjoint regions the hit is assigned to that feature. If the read overlaps both or none of the regions, no hit is assigned. Therefore, unlike the `divide` option, `uniqueDisjoint` does not resolve multi-hit conflict in all situations.
`ignore.strand`	A logical value indicating if strand should be considered when matching.
`splitreads`	A logical value indicating if split reads should be included.
`...`	Additional arguments, perhaps used by methods defined on this generic.

Details

The `countGenomicOverlaps` methods use the `findOverlaps` function in conjunction with a resolution method to identify overlaps and resolve subjects (reads) that match multiple queries (annotation regions). The usual `type` argument of `findOverlaps` is used to specify the type of overlap. The `resolution` argument is used to select a method to resolve the conflict when a subject hits more than 1 query. Here the term 'hit' means an overlap identified by `findOverlaps`.

The primary difference in the handling of split reads vs simple reads (i.e., no gap in the CIGAR) is the portion of the read hit each split read fragment has to contribute. All reads, whether simple or split, have an overall value of 1 to contribute to a query they hit. In the case of the split reads, this value is further divided by the number of fragments in the read. For example, if a split read has 3 fragments (i.e., two gaps in the CIGAR) each fragment has a value of 1/3 to contribute to the query they hit. As with the simple reads, depending upon the `resolution` chosen the value may be divided, fully assigned or discarded.

More detailed examples can be found in the `countGenomicOverlaps` vignette.

Value

A vector of counts

Author(s)

Valerie Obenchain and Martin Morgan

Examples

```
## Not run:
rng1 <- function(s, w)
GRanges(seq="chr1", IRanges(s, width=w), strand="+")

rng2 <- function(s, w)
GRanges(seq="chr2", IRanges(s, width=w), strand="+")

query <- GRangesList(A=rng1(1000, 500),
                     B=rng2(2000, 900),
                     C=rng1(c(3000, 3600), c(500, 300)),
                     D=rng2(c(7000, 7500), c(600, 300)),
                     E1=rng1(4000, 500), E2=rng1(c(4300, 4500), c(400, 400)),
                     F=rng2(3000, 500),
                     G=rng1(c(5000, 5600), c(500, 300)),
                     H1=rng1(6000, 500), H2=rng1(6600, 400))

subj <- GRangesList(a=rng1(1400, 500),
                      b=rng2(2700, 100),
                      c=rng1(3400, 300),
                      d=rng2(7100, 600),
                      e=rng1(4200, 500),
                      f=rng2(c(3100, 3300), 50),
                      g=rng1(c(5400, 5600), 50),
                      h=rng1(c(6400, 6600), 50))

## Overlap type = "any"
none <- countGenomicOverlaps(query, subj,
                             type="any", resolution="none")
divide <- countGenomicOverlaps(query, subj,
```

```
                                type="any", resolution="divide")
uniqueDisjoint <- countGenomicOverlaps(query, subj, type="any",
                                       resolution="uniqueDisjoint")
data.frame(none = none,
           divide = divide,
           uniqDisj = uniqueDisjoint)


## Split read with 4 fragments :
splitreads <- GRangesList(c(rng1(c(3000, 3200, 4000), 100), rng1(5400, 300)))
## Unlist both the splitreads and the query to see
## - read fragments 1 and 2 both hit query 3
## - read fragment 3 hits query 7
## - read fragment 4 hits query 11 and 12
findOverlaps(unlist(query), unlist(splitreads))

## Use countGenomicOverlaps to avoid double counting.
## Because this read has 4 parts each part contributes a count of 1/4.
## When resolution="none" only reads that hit a single region are counted.
split_none <- countGenomicOverlaps(query, splitreads, type="any",
                                   resolution="none")
## When resolution="divide" all reads are counted by dividing their count
## evenly between the regions they hit. Region 3 of the query was hit
## by two reads each contributing a count of 1/4. Region 7 was hit
## by one read contributing a count of 1/4. Regions 11 and 12 were both
## hit by the same read resulting in having to share (i.e., "divide") the
## single 1/4 hit read 4 had to contribute.
split_divide <- countGenomicOverlaps(query, splitreads,
                                     type="any", resolution="divide")

data.frame(none = split_none,
           divide = split_divide)

## End(Not run)
```

---

coverage-methods *Coverage of a GRanges, GRangesList, GappedAlignments, or GappedAlignmentPairs object*

Description

coverage methods for GRanges, GRangesList, GappedAlignments, and GappedAlignmentPairs objects.

Usage

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

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.

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 column in `elementMetadata(x)` 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. Aboyoun 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 methods and related utilities

Description

`encodeOverlaps` methods and related utilities.

WARNING (May 8, 2012): This man page is incomplete and was still a WORK IN PROGRESS at the time GenomicRanges 1.8 was released (as part of Bioconductor 2.10, released in April 2012). All the work on "overlap encodings" is now happening in the *devel* version of the GenomicRanges package (GenomicRanges 1.9, part of Bioconductor 2.11, at the time of this writing). To use "overlap encodings" and related tools, and to get an updated version of the "Overlap encodings" vignette, it is *strongly* recommended that you use Bioconductor 2.11. In particular please make sure that you always use the *latest* version of the GenomicRanges package (version 1.9.14 at the time of this writing, expect frequent updates).

In other words, this man page is superseded by the man page found in GenomicRanges 1.9 (part of Bioconductor 2.11). It won't be updated.

See `?useDevel` in the BiocInstaller package for how to use the devel version of Bioconductor (2.11 at the time of this writing).

See `?biocLite` in the BiocInstaller package for how to update all the installed packages.

Usage

```
flipQuery(x, i)

## S4 method for signature 'GRangesList,GRangesList,missing'
encodeOverlaps(query, subject, hits=NULL,
```

```
                flip.query.if.wrong.strand=FALSE)

selectEncodingWithCompatibleStrand(x, y,
                                   query.strand, subject.strand, hits=NULL)
isCompatibleWithSplicing(x)
isCompatibleWithSkippedExons(x, max.skipped.exons=NA)
extractSkippedExonRanks(x)
```

Arguments

`x`	For `flipQuery`: A GRangesList object. For `selectEncodingWithCompatibleStrand`: An OverlapEncodings object. For `isCompatibleWithSplicing`, `isCompatibleWithSkippedExons`, and `extractSkippedExonRa` 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.
`query, subject`	GRangesList, RangesList or Ranges objects.
`hits`	A Hits object. See ?‘`encodeOverlaps,ANY,ANY,Hits-method`‘ 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.
`y`	An OverlapEncodings object.
`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).

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.
- `findOverlaps`.
- OverlapEncodings-class.
- GRangesList-class.

Examples

```
## See the "Overlap encodings" vignette in the GenomicRanges package for
## some examples.
```

findOverlaps-methods *GRanges, GRangesList, GappedAlignments and GappedAlignmentPairs Interval Overlaps*

Description

Finds interval overlaps between a GRanges, GRangesList, GappedAlignments 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"), 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'
subsetByOverlaps(query, subject,
    maxgap = 0L, minoverlap = 1L,
    type = c("any", "start", "end", "within", "equal"),
    ignore.strand = FALSE)
## S4 method for signature 'GenomicRanges,GenomicRanges'
match(x, table,
    nomatch = NA_integer_, incomparables = NULL)
# Also:  x %in% table
```

Arguments

query, subject, x, table	A GRanges, GRangesList, GappedAlignments or GappedAlignmentPairs object. RangesList and RangedData are also accepted for one of query or subject (x or table for match).
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.
nomatch	The integer value to be returned in the case when no match is found.
incomparables	This value is ignored.

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.

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 when select = "first".

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

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

For match same as findOverlaps when select = "first".

For %in% the logical vector produced by !is.na(match(x, table)).

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.
- 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(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))
grlist <- GRangesList("gr1" = gr1, "gr2" = gr2, "gr3" = gr3)

## Overlapping two GRanges objects:
table(gr %in% gr1)
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(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(grlist %in% gr)
countOverlaps(grlist, gr)
findOverlaps(grlist, gr)
subsetByOverlaps(grlist, gr)
countOverlaps(grlist, gr, type = "start")
findOverlaps(grlist, gr, type = "start")
subsetByOverlaps(grlist, gr, type = "start")
findOverlaps(grlist, gr, select = "first")

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

## ---------------------------------------------------------------------
## WITH A GappedAlignments OBJECT
## ---------------------------------------------------------------------
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)

## Advanced examples:
subsetByOverlaps(galn, subject)
table(match(galn, subject), useNA = "ifany")
table(galn %in% subject)
```

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)`: Returns the number of alignment pairs in `x`.

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

`seqnames(x)`: Gets 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)`: Gets the strand for each alignment pair in `x`. By definition 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 *always* the opposite of the strand of the "first" alignment (`makeGappedAlignmentPairs`, the function used by `readBamGappedAlignmentPairs` for doing the pairing, rejects pairs where the "first" and "last" alignments are on the same strand).

`first(x, invert.strand=FALSE)`, `last(x, invert.strand=FALSE)`: Gets 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)`: Gets 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)`: Gets 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.

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

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

`seqlevels(x)`, `seqlevels(x) <- value`: Gets or sets 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`: Gets or sets 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`: Gets or sets 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`: Gets or sets 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.

Vector methods

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

`x[i]`: Returns 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)`: Returns 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)`:

Returns 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(grl1[[i]], grl2[[i]])`, where `grl1` is `grglist(left(x))` and `grl2` 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(grl1[[i]], grl2[[i]])`, where `grl1` is `grglist(first(x), order.as.in.query=TRUE)` and `grl2` is `grglist(last(x, invert.strand=TRUE), order.as.in`

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.

`as(x, "GRangesList")`: An alternate way of doing `grglist(x)`.

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))
seqnames(galp)
seqlevels(galp)

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

strand(galp)

first(galp)
last(galp)
last(galp, invert.strand=TRUE)
left(galp)
right(galp)
table(isProperPair(galp))

galp[[1]]
unlist(galp)

grglist(galp)
grglist(galp, order.as.in.query=TRUE)
```

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 expicitly 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=char

Low-level GappedAlignments constructor. Generally not used directly. Named arguments in ... are used as elementMetadata.

Accessors

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

length(x): Returns the number of alignments in x.

names(x), names(x) <- value: Gets or sets 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: Gets or sets 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: Gets or sets 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 length length(x) containing the number of gaps for each alignment. Equivalent to elementLengths(rglist(x)) - 1L.

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

seqlevels(x), seqlevels(x) <- value: Gets or sets 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: Gets or sets 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`: Gets or sets 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`: Gets or sets 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.

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)`:

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

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

See Details section above for more information.

`granges(x)`, `ranges(x)`: Returns 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.

`as(x, "GRangesList")`, `as(x, "GRanges")`, `as(x, "RangesList")`, `as(x, "Ranges")`: An alternate way 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]`: Returns 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. Returns 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).

`narrow(x, start=NA, end=NA, width=NA)`: `x` is a GappedAlignments object. Returns a new GappedAlignments object of 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).

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)
seqlevels(gal)

## Rename the reference sequences:
```

```
seqlevels(gal) <- sub("seq", "chr", seqlevels(gal))
seqlevels(gal)

strand(gal)
head(cigar(gal))
head(qwidth(gal))
table(qwidth(gal))

grglist(gal)  # a GRangesList object
granges(gal)  # a GRanges object
rglist(gal)   # a CompressedIRangesList object
ranges(gal)   # an IRanges object
stopifnot(identical(elementLengths(grglist(gal)), elementLengths(rglist(gal))))

head(start(gal))
head(end(gal))
head(width(gal))
head(ngap(gal))

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

## Ds 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.

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. The `duplicated` and `unique` methods for GenomicRanges objects are using this equality.

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. The `order`, `sort` and `rank` methods for GenomicRanges objects are using this "natural order".

Also the ==, !=, <=, >=, < and > operators between 2 GenomicRanges objects are using this "natural order".

See Also

GenomicRanges-class, Ranges-comparison

Examples

```
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))

duplicated(gr)
duplicated(c(gr[4], gr))
unique(gr)
unique(c(gr[4], gr))
order(gr)
sort(gr)
rank(gr)
```

```
gr[2] == gr[2]  # TRUE
gr[2] == gr[5]  # FALSE
gr == gr[4]
gr >= gr[3]
compare(gr, gr[3])
rangeComparisonCodeToLetter(compare(gr, gr[3]))
```

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 element metadata (e.g. score, GC content, etc.). This information is stored in four slots:

`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.

`elementMetadata` a DataFrame object containing the annotation columns. Columns cannot be named `"seqnames"`, `"ranges"`, `"strand"`, `"seqlevels"`, `"seqlengths"`, `"isCircular"`, `"genome"`, `"start"`, `"end"`, `"width"`, or `"element"`.

Constructor

`GRanges(seqnames = Rle(), ranges = IRanges(), strand = Rle("*", length(seqna`
`structure(rep(NA_integer_, length(levels(seqnames))), names = `
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.

`seqlengths` a named integer vector containing the sequence lengths for each `level(seqnames)`.

`...` Optional annotation columns for the `elementMetadata` slot. These columns cannot be named `"start"`, `"end"`, `"width"`, or `"element"`.

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 the values become columns in the result. The `seqlengths(from)`, `isCircular(from)`, and `genome(from)` vectors are stored in the element metadata of `ranges(rd)`.

`as(from, "RangesList")`: Creates a RangesList object from a GRanges object. The `strand` and values become element metadata on the ranges. The `seqlengths(from)`, `isCircular(from)`, and `genome(from)` vectors are stored in the element metadata.

`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 columns stored in `elementMetadata(x)`.

Accessors

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

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

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

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

`names(x)`, `names(x) <- value`: Gets or sets the names of the elements.

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

`elementMetadata(x)`, `elementMetadata(x) <- value`: Gets or sets the optional data columns. `value` can be a DataFrame, data.frame object, or NULL.

`values(x)`, `values(x) <- value`: Alternative to `elementMetadata` functions.

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

`seqlevels(x)`, `seqlevels(x, force=FALSE) <- value`: Gets or sets 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`: Gets or sets 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`: Gets or sets 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`: Gets or sets 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.

`score(x)`: Gets 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`: Gets or sets `start(ranges(x))`.

`end(x)`, `end(x) <- value`: Gets or sets `end(ranges(x))`.

`width(x)`, `width(x) <- value`: Gets or sets `width(ranges(x))`.

`flank(x, width, start = TRUE, both = FALSE, use.names = TRUE, ignore.strand=FALSE)`: Returns a new GRanges object 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 codeend(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.

`resize(x, width, use.names = TRUE)`: Returns a new GRanges object 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.

`shift(x, shift, use.names = TRUE)`: Returns a new GRanges 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.

`disjoin(x)`: Returns a new GRanges object containing disjoint ranges for each distinct (seqname, strand) pairing. The names (`names(x)`) and the columns in `x` are dropped.

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

`disjointBins(x, ignore.strand = FALSE)`: 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.

`gaps(x, start = 1L, end = seqlengths(x))`: Returns a new GRanges object 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.

`range(x, ...)`: Returns a new GRanges object containing range bounds for each distinct (seqname, strand) pairing. The names (`names(x)`) and the columns in `x` are dropped.

`reduce(x, drop.empty.ranges = FALSE, min.gapwidth = 1L)`: Returns a new GRanges object containing reduced ranges for each distinct (seqname, strand) pairing. The names (`names(x)`) and the columns in `x` are dropped. See `?reduce` for more information about range reduction and for a description of the optional arguments.

`restrict(x, start = NA, end = NA, keep.all.ranges = FALSE, use.names = TRUE)`: Returns a new GRanges object containing restricted ranges for distinct seqnames. The `start` and `end` arguments can be a named numeric vector of seqnames for the ranges to be resticted or a numeric vector or length 1 if the restriction operation is to be applied to all the sequences in `x`. See `?restrict` for more information about range restriction and for a description of the optional arguments.

`distance(x, y, ignore.strand = FALSE)`: Calculate the number of positions separating two features. The value is zero if the features overlap and `NA` if the features are on different sequences, or different strands (if `ignore.strand` is `FALSE`).

Other ranges utilities

In the code snippets below, `x` and `subject` are GRanges objects.

`precede(x, subject = x, select = c("arbitrary", "all"), ignore.strand = FALSE, ...)`: Identifies which subject(s) the query precedes. Returns the index of the range in `subject` that is directly preceded by the range in `x`. When ignore.strand = TRUE, the strand for both `x` and `subject` are set to `'+'`.

When `select = 'arbitrary'` an integer vector is returned with a single match per `x`. If no match is found an `NA` is returned. When `select = 'all'` a `Hits` object is returned with all matches for `x`. If `x` does not have a match in `subject` the `x` is not included in the `Hits` object. Overlapping ranges are excluded.

Matching by 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.

`follow(x, subject = x, select = c("arbitrary", "all"), ignore.strand = FALSE, ...)`: Identifies which subject(s) the query follows. Returns the index of the interval in `subject` that is directly followed by the range in `x`. When ignore.strand = TRUE, both `x` and `subject` strand are set to `'+'`.

When `select = 'arbitrary'` an integer vector is returned with a single match per `x`. If no match is found an `NA` is returned. When `select = 'all'` a `Hits` object is returned with all

matches for x. If x does not have a match in subject the x is not included in the Hits object. Overlapping ranges are excluded. See the documentation for precede for details of strand matching.

nearest(x, subject = x, ignore.strand = FALSE): Conventional nearest neighbor finder which returns an integer vector containing the index of the nearest neightbor range in subject for each range in x. If there is no nearest neighbor NA is returned. See the nearest man page in IRanges for a description of the algorithm used.

If ignore.strand = TRUE, both x and subject strand are set to "+". nearest calls precede and follow and thus the strand matching for * follows the conventions documented under those functions.

distanceToNearest(x, subject, ignore.strand = FALSE): Returns the distance for each range in x to its nearest neighbor in the subject.

If ignore.strand = TRUE, both x and subject strand are set to "+".

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, ..., .ignoreElementMetadata=TRUE) If the GRanges objects have associated elementMetadata (also known as values), each such DataFrame must have the same columns in order to combine successfully. In order to circumvent this restraint, you can pass in an .ignoreElementMetadata=TRUE argument which will combine all the objects into one and drop all of their elementMetadata.

split(x, f = seq_len(length(x)), drop = FALSE): Splits x into a GRangesList, according to f, dropping elements corresponding to unrepresented levels if drop is TRUE. Split factor f defaults to splitting each element of x into a separate element in the resulting GRangesList object.

Subsetting

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

x[i, j], x[i, j] <- value: Gets or sets elements i with optional elementMetadata columns elementMetadata(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 elementMetadata 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`.

Author(s)

P. Aboyoun

See Also

GRangesList-class, `seqinfo`, Vector-class, Ranges-class, Rle-class, DataFrame-class, coverage-methods, setops-methods, findOverlaps-methods

Examples

```
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

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

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

# Intra-interval operations
flank(gr, 10)
resize(gr, 10)
shift(gr, 1)

# Inter-interval operations
isDisjoint(gr)
disjoin(gr)
gaps(gr, start = 1, end = 10)
range(gr)
reduce(gr)
restrict(gr, start =3)

# Combining objects
gr2 <- GRanges(seqnames=Rle(c('Chrom1', 'Chrom2', 'Chrom3'), c(3, 3, 4)),
               IRanges(1:10, width=5), strand='-',
               score=101:110, GC = runif(10))
gr3 <- GRanges(seqnames=Rle(c('Chrom1', 'Chrom2', 'Chrom3'), c(3, 4, 3)),
               IRanges(101:110, width=10), strand='-',
               score=21:30)
some.gr <- c(gr, gr2)

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

# precede and follow :
# query on "+" strand
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)

# query on "+" strand
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))
```

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(), f`
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 columns stored in `elementMetadata(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`: Gets or sets the sequence names in the form of an RleList. `value` can be an RleList or CharacterList.

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

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

`elementMetadata(x)`, `elementMetadata(x) <- value`: Gets or sets the optional data columns for the GRangesList elements. `value` can be a DataFrame, data.frame object, or NULL.

`values(x)`, `values(x) <- value`: Alternative to `elementMetadata` functions.

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

`seqlevels(x)`, `seqlevels(x, force=FALSE) <- value`: Gets or sets 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`: Gets or sets 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`: Gets or sets 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`: Gets or sets 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.

`score(x)`: Gets the "score" column from the element metadata, if any.

List methods

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

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

`names(x)`, `names(x) <- value`: Gets or sets the names of the elements.

`elementLengths(x)`: Gets 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`: Gets or sets `start(ranges(x))`.

`end(x)`, `end(x) <- value`: Gets or sets `end(ranges(x))`.

`width(x)`, `width(x) <- value`: Gets or sets `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).

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`: Gets or sets elements `i` with optional values columns `values(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`: Gets or sets element `i`, where `i` is a numeric or character vector of length 1.

`x$name`, `x$name <- value`: Gets or sets 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 & elementMetadata 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, ign`
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(grl)

## Construction with makeGRangesListFromFeatureFragments():
filepath <- system.file("extdata", "feature_frags.txt",
                        package="GenomicRanges")
featfrags <- read.table(filepath, header=TRUE, stringsAsFactors=FALSE)
grl2 <- with(featfrags,
             makeGRangesListFromFeatureFragments(seqnames=targetName,
                                                 fragmentStarts=targetStart,
                                                 fragmentWidths=blockSizes,
                                                 strand=strand))
names(grl2) <- featfrags$RefSeqID
grl2
```

`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.

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
```

Arguments

x	The object from/on which to get/set the sequence information.
new2old	The new2old argument allows the user to rename, drop, add and/or reorder the "sequence levels" in x. 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 <= length(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). 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. length(values) >= length(seq and also seqlevels(values)[seq_len(length(seqlevels(x)))] must be identical to seqlevels(x).
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	Typically a Seqinfo object for the seqinfo setter. Either a named or unnamed character vector for the seqlevels setter. A vector containing the sequence information to store for the other setters.

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` and `genome` getters and setters are defined for Seqinfo objects.
- Classes that implement the `seqinfo` getter are typically expected to return a Seqinfo object.
- Default `seqlevels`, `seqlengths`, `isCircular` and `genome` 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))`, and `genome(x)` does `genome(seqinfo(x))`. So any class with a `seqinfo` getter that returns a Seqinfo object will have all those getters work out-of-the-box. If, in addition, the class defines a `seqinfo` setter, then all the related 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 flagi, 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)`: Gets the number of sequences in `x`.

`seqnames(x)`, `seqnames(x) <- value`: Gets/sets 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`: Gets/sets the length for each sequence in `x`.

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

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

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, "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.

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")
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
}
```

`setops-methods` *Set operations on GRanges/GRangesList/GappedAlignments objects*

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

`x, y`	For `union`, `intersect`, `setdiff`, `pgap`: `x` and `y` must both be GRanges objects. For `punion`: one of `x` or `y` must be a GRanges object, the other one can be a GRanges or GRangesList object. For `pintersect`: one of `x` or `y` must be a GRanges object, the other one can be a GRanges, GRangesList or GappedAlignments object. For `psetdiff`: `x` and `y` can be any combination of GRanges and/or GRangesList objects, with the exception that if `x` is a GRangesList object then `y` must be a GRangesList too. In addition, for the "parallel" operations, `x` and `y` must be of equal length (i.e. `length(x) == length(y)`).
`fill.gap`	Logical indicating whether or not to force a union by using the rule `start = min(start(x), start(`
`resolve.empty`	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 `x` with any ambiguous empty range. (See `pintersect` for the definition of an ambiguous range.)
`ignore.strand`	For set operations: If set to TRUE, then the strand of `x` and `y` is set to `"*"` prior to any computation. For parallel set operations: If set to TRUE, the strand information is ignored in the computation and the result has the strand information of `x`.
`...`	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

IRanges-setops, 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)
punion(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))
```

`strand-utils` *Strand utilities*

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 an eSet-like container where rows represent ranges of interest (as a GRanges-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 = GRanges(),
    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

## Subsetting

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

Arguments

assays	A `list` or `SimpleList` of matrix elements, or a `matrix`. Each element 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`, `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 other accessors, ignored.
verbose	A logical(1) indicating whether messages about data coercion during construction should be printed.
x	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'.

Details

The SummarizedExperiment class is meant for numeric and other data types derived from a sequencing experiment. The structure is rectangular, like an eSet in **Biobase**.

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 (analogous to annotations associated with genes in a eSet) 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 values function) in a fashion similar to the varMetadata facilities of the eSet class. 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.

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]] <- va to get or set the ith (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.

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`.

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"]
```

---

summarizeOverlaps *Count reads that map to genomic features*

Description

Count reads that map to genomic features with options to resolve reads that overlap multiple features.

Usage

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

Arguments

features	A GRanges or a GRangesList object of genomic regions of interest. When a GRanges is supplied, each row is considered a different feature. When a GRangesList is supplied, each highest list-level is considered a feature and the multiple elements are considered portions of the same feature. See examples or vignette for details.
reads	A GappedAlignments, BamFileList or a BamViews object.
mode	Character name of a function that defines the counting method to be used. Available counting modes include "Union", "IntersectionStrict", or "IntersectionNotEmpty" and are designed after the counting modes available in the HTSeq package by Simon Anders (see references). A user provided count function can be used as the mode with the BamFileList method for summarizedOverlaps. • "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. • "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, – 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
param	An optional ScanBamParam instance to further influence scanning, counting, or filtering of the BAM file.
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 mclapply to indicate the number of cores to use etc.

Details

In the context of summarizeOverlaps 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 to be overlapped. When features is a GRangesList the highest list-levels define the features.

summarizeOverlaps offers three mode functions to handle reads that overlap multiple features: "Union", "IntersectionStrict", and "IntersectionNotEmpty". These functions are patterned after the counting methods in the HTSeq package (see references). Each mode has a set of rules that dictate how a read is assigned. Reads are counted a maximum of once. Alternatively, users can provide their own counting function as the mode argument and take advantage of the infrastructure in summarizeOverlaps to count across multiple files and parse the results into a SummarizedExperiment object.

Currently reads must be input as either a BAM file or a GappedAlignments object. The information in the CIGAR field is used to determine if gapped reads are present.

NOTE : summarizeOverlaps does not currently handle paired-end reads.

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

http://www-huber.embl.de/users/anders/HTSeq/doc/overview.html home page for HTSeq

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

See Also

DESeq, DEXSeq and edgeR packages BamFileList BamViews

Examples

```
group_id <- c("A", "B", "C", "C", "D", "D", "E", "F", "G", "H", "H")
features <- GRanges(
    seqnames = Rle(c("chr1", "chr2", "chr1", "chr1", "chr2", "chr2",
        "chr1", "chr1", "chr2", "chr1", "chr1")),
    strand = strand(rep("+", length(group_id))),
    ranges = IRanges(
        start=c(1000, 2000, 3000, 3600, 7000, 7500, 4000, 4000, 3000,
            5000, 5400),
        width=c(500, 900, 500, 300, 600, 300, 500, 900, 500, 500, 500)),
   DataFrame(group_id)
)

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)))

## Results from countOverlaps are included to highlight how the
## modes in summarizeOverlaps count a read a maximum of once.

## When the 'features' argument is a GRanges, each row
## is treated as a different feature.
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))

## When the 'features' argument is a GRangesList, each
## highest list-level is a different feature.
lst <- split(features, values(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))

## Read across BAM files and package output for DESeq or edgeR analysis
library(Rsamtools)
library(DESeq)
library(edgeR)

fls = list.files(system.file("extdata",package="GenomicRanges"),
    recursive=TRUE, pattern="*bam$", full=TRUE)
bfl <- BamFileList(fls)
features <- GRanges(
    seqnames = Rle(c("chr2L", "chr2R", "chr2L", "chr2R", "chr2L", "chr2R",
        "chr2L", "chr2R", "chr2R", "chr3L", "chr3L")),
    strand = strand(rep("+", 11)),
    ranges = IRanges(start=c(1000, 2000, 3000, 3600, 7000, 7500, 4000, 4000,
        3000, 5000, 5400), width=c(500, 900, 500, 300, 600, 300, 500, 900,
        500, 500, 500))
)

solap <- summarizeOverlaps(features, bfl)

deseq <- newCountDataSet(countData=assays(solap)$counts,
                         conditions=rownames(colData(solap)))

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

`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 :
grl1 <- GRangesList("gr1" = gr1, "gr2" = gr2, "gr3" = gr3)
grl2 <- GRangesList("gr1" = gr1, "gr2" = gr2, "gr3" = gr3_rename)
grl1_rename <- renameSeqlevels(grl1, c(CHROM4="chr4"))
identical(grl1_rename, grl2)

grl1_subset <- keepSeqlevels(grl1, "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
GappedAlignmentPairs-class, 22
GappedAlignments-class, 25
Seqinfo-class, 47
∗Topic **manip**
cigar-utils, 2
∗Topic **methods**
Constraints, 7
countGenomicOverlaps, 12
coverage-methods, 15
encodeOverlaps-methods, 17
findOverlaps-methods, 19
GappedAlignmentPairs-class, 22
GappedAlignments-class, 25
GenomicRanges-comparison, 31
seqinfo, 45
Seqinfo-class, 47
setops-methods, 49
strand-utils, 52
summarizeOverlaps, 57
utils, 60
∗Topic **utilities**
countGenomicOverlaps, 12
coverage-methods, 15
encodeOverlaps-methods, 17
findOverlaps-methods, 19
setops-methods, 49
summarizeOverlaps, 57
utils, 60
<,GenomicRanges,GenomicRanges-method *(*GenomicRanges-comparison*)*, 31
<=,GenomicRanges,GenomicRanges-method *(*GenomicRanges-comparison*)*, 31
==,GenomicRanges,GenomicRanges-method *(*GenomicRanges-comparison*)*, 31
>,GenomicRanges,GenomicRanges-method *(*GenomicRanges-comparison*)*, 31
>=,GenomicRanges,GenomicRanges-method *(*GenomicRanges-comparison*)*, 31
[,GRangesList-method *(*GRangesList-class*)*, 39
[,GappedAlignmentPairs-method *(*GappedAlignmentPairs-class*)*, 22
[,GappedAlignments-method *(*GappedAlignments-class*)*, 25
[,GenomicRanges-method *(*GRanges-class*)*, 32
[,Seqinfo-method *(*Seqinfo-class*)*, 47
[,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 53
[<-,GRangesList,ANY,ANY,ANY-method *(*GRangesList-class*)*, 39
[<-,GenomicRanges,ANY,ANY,ANY-method *(*GRanges-class*)*, 32
[<-,SummarizedExperiment,ANY,ANY,SummarizedExperiment-m *(*SummarizedExperiment-class*)*, 53
[[,GappedAlignmentPairs,ANY,ANY-method *(*GappedAlignmentPairs-class*)*, 22
[[,SummarizedExperiment,ANY,missing-method *(*SummarizedExperiment-class*)*, 53
[[<-,GRangesList-method *(*GRangesList-class*)*, 39
[[<-,SummarizedExperiment,ANY,missing,ANY-method *(*SummarizedExperiment-class*)*, 53
$,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 53
$<-,SummarizedExperiment,ANY-method *(*SummarizedExperiment-class*)*, 53
%in%,ANY,GappedAlignments-method *(*findOverlaps-methods*)*, 19
%in%,GRangesList,GRangesList-method *(*findOverlaps-methods*)*, 19
%in%,GRangesList,GenomicRanges-method *(*findOverlaps-methods*)*, 19
%in%,GRangesList,RangedData-method *(*findOverlaps-methods*)*, 19
%in%,GRangesList,RangesList-method

(findOverlaps-methods), 19
%in%,GappedAlignmentPairs,ANY-method (findOverlaps-methods), 19
%in%,GappedAlignments,ANY-method (findOverlaps-methods), 19
%in%,GappedAlignments,GappedAlignments-method (findOverlaps-methods), 19
%in%,GenomicRanges,GRangesList-method (findOverlaps-methods), 19
%in%,GenomicRanges,GenomicRanges-method (findOverlaps-methods), 19
%in%,GenomicRanges,RangedData-method (findOverlaps-methods), 19
%in%,GenomicRanges,RangesList-method (findOverlaps-methods), 19
%in%,RangedData,GRangesList-method (findOverlaps-methods), 19
%in%,RangedData,GenomicRanges-method (findOverlaps-methods), 19
%in%,RangesList,GRangesList-method (findOverlaps-methods), 19
%in%,RangesList,GenomicRanges-method (findOverlaps-methods), 19

as.data.frame,GappedAlignments-method (GappedAlignments-class), 25
as.data.frame,GenomicRanges-method (GRanges-class), 32
as.data.frame,GRangesList-method (GRangesList-class), 39
as.data.frame,Seqinfo-method (Seqinfo-class), 47
assay (SummarizedExperiment-class), 53
assay,SummarizedExperiment,ANY-method (SummarizedExperiment-class), 53
assay,SummarizedExperiment,character-method (SummarizedExperiment-class), 53
assay,SummarizedExperiment,missing-method (SummarizedExperiment-class), 53
assay,SummarizedExperiment,numeric-method (SummarizedExperiment-class), 53
assay<- (SummarizedExperiment-class), 53
assay<-,SummarizedExperiment,character,matrix-method (SummarizedExperiment-class), 53
assay<-,SummarizedExperiment,missing,matrix-method (SummarizedExperiment-class), 53
assay<-,SummarizedExperiment,numeric,matrix-method (SummarizedExperiment-class), 53
assays (SummarizedExperiment-class), 53
assays,SummarizedExperiment-method (SummarizedExperiment-class), 53
assays<- (SummarizedExperiment-class), 53
assays<-,SummarizedExperiment,list-method (SummarizedExperiment-class), 53
assays<-,SummarizedExperiment,SimpleList-method (SummarizedExperiment-class), 53

BamFileList, *58*, *59*
BamViews, *58*, *59*

c,GappedAlignments-method (GappedAlignments-class), 25
c,GenomicRanges-method (GRanges-class), 32
checkConstraint (Constraints), 7
cigar (GappedAlignments-class), 25
cigar,GappedAlignments-method (GappedAlignments-class), 25
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:GappedAlignmentPairs (GappedAlignmentPairs-class), 22
class:GappedAlignments (GappedAlignments-class), 25
class:GenomicRanges (GRanges-class), 32
class:GenomicRangesList (GenomicRangesList-class), 32
class:GRanges (GRanges-class), 32
class:GRangesList (GRangesList-class), 39

class:Seqinfo (Seqinfo-class), 47
class:SimpleGenomicRangesList (GenomicRangesList-class), 32
coerce,GappedAlignmentPairs,GRangesList-method (GappedAlignmentPairs-class), 22
coerce,GappedAlignments,GRanges-method (GappedAlignments-class), 25
coerce,GappedAlignments,GRangesList-method (GappedAlignments-class), 25
coerce,GappedAlignments,Ranges-method (GappedAlignments-class), 25
coerce,GappedAlignments,RangesList-method (GappedAlignments-class), 25
coerce,GenomicRanges,RangedData-method (GRanges-class), 32
coerce,GenomicRanges,RangesList-method (GRanges-class), 32
coerce,GRangesList,CompressedIRangesList-method (GRangesList-class), 39
coerce,GRangesList,IRangesList-method (GRangesList-class), 39
coerce,RangedData,GRanges-method (GRanges-class), 32
coerce,RangedDataList,GenomicRangesList-method (GenomicRangesList-class), 32
coerce,RangedDataList,GRangesList-method (GRangesList-class), 39
coerce,RangesList,GRanges-method (GRanges-class), 32
coerce,RangesMapping,GenomicRanges-method (map-methods), 43
coerce,RleList,GRanges-method (GRanges-class), 32
coerce,RleViewsList,GRanges-method (GRanges-class), 32
coerce,Seqinfo,GenomicRanges-method (Seqinfo-class), 47
coerce,Seqinfo,RangesList-method (Seqinfo-class), 47
colData (SummarizedExperiment-class), 53
colData,SummarizedExperiment-method (SummarizedExperiment-class), 53
colData<- (SummarizedExperiment-class), 53
colData<-,SummarizedExperiment,DataFrame-method (SummarizedExperiment-class), 53
compare,GenomicRanges,GenomicRanges-method (GenomicRanges-comparison), 31
CompressedIRangesList, *4*, *28*
CompressedIRangesList-class, *29*
CompressedRleList, *4*
Constraint (Constraints), 7
constraint (Constraints), 7
Constraint-class (Constraints), 7
constraint<- (Constraints), 7
ConstraintORNULL (Constraints), 7
ConstraintORNULL-class (Constraints), 7
Constraints, 7
countGenomicOverlaps, 12
countGenomicOverlaps,GenomicRanges,GappedAlignments-method (countGenomicOverlaps), 12
countGenomicOverlaps,GenomicRanges,GenomicRanges-method (countGenomicOverlaps), 12
countGenomicOverlaps,GenomicRanges,GRangesList-method (countGenomicOverlaps), 12
countGenomicOverlaps,GRangesList,GappedAlignments-method (countGenomicOverlaps), 12
countGenomicOverlaps,GRangesList,GenomicRanges-method (countGenomicOverlaps), 12
countGenomicOverlaps,GRangesList,GRangesList-method (countGenomicOverlaps), 12
countOverlaps,ANY,GappedAlignments-method (findOverlaps-methods), 19
countOverlaps,GappedAlignmentPairs,ANY-method (findOverlaps-methods), 19
countOverlaps,GappedAlignments,ANY-method (findOverlaps-methods), 19
countOverlaps,GappedAlignments,GappedAlignments-method (findOverlaps-methods), 19
countOverlaps,GenomicRanges,GenomicRanges-method (findOverlaps-methods), 19
countOverlaps,GenomicRanges,GRangesList-method (findOverlaps-methods), 19
countOverlaps,GenomicRanges,RangedData-method (findOverlaps-methods), 19
countOverlaps,GenomicRanges,RangesList-method (findOverlaps-methods), 19
countOverlaps,GRangesList,GenomicRanges-method (findOverlaps-methods), 19
countOverlaps,GRangesList,GRangesList-method (findOverlaps-methods), 19
countOverlaps,GRangesList,RangedData-method (findOverlaps-methods), 19
countOverlaps,GRangesList,RangesList-method (findOverlaps-methods), 19
countOverlaps,RangedData,GenomicRanges-method (findOverlaps-methods), 19
countOverlaps,RangedData,GRangesList-method (findOverlaps-methods), 19
countOverlaps,RangesList,GenomicRanges-method (findOverlaps-methods), 19

countOverlaps,RangesList,GRangesList-method
(findOverlaps-methods), 19
coverage, *4*, *15, 16*
coverage,GappedAlignmentPairs-method
(coverage-methods), 15
coverage,GappedAlignments-method
(coverage-methods), 15
coverage,GenomicRanges-method
(coverage-methods), 15
coverage,GRangesList-method
(coverage-methods), 15
coverage-methods, 15, *25*, *29*, *37*, *42*

DataFrame, *3, 4*, *33*, *36*, *53*, *55–57*
DataFrame-class, *37*
DataFrameList-class, *42*
dim,SummarizedExperiment-method
(SummarizedExperiment-class),
53
dimnames,SummarizedExperiment-method
(SummarizedExperiment-class),
53
dimnames<-,SummarizedExperiment,list-method
(SummarizedExperiment-class),
53
dimnames<-,SummarizedExperiment,NULL-method
(SummarizedExperiment-class),
53
disjoin,GenomicRanges-method
(GRanges-class), 32
disjointBins,GenomicRanges-method
(GRanges-class), 32
distance,GenomicRanges,GenomicRanges-method
(GRanges-class), 32
duplicated,GenomicRanges-method
(GenomicRanges-comparison), 31

elementMetadata, *36*
elementMetadata,GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
22
elementMetadata,GappedAlignments-method
(GappedAlignments-class), 25
elementMetadata,GenomicRanges-method
(GRanges-class), 32
elementMetadata,GRangesList-method
(GRangesList-class), 39
elementMetadata<-,GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
22
elementMetadata<-,GappedAlignments-method
(GappedAlignments-class), 25
elementMetadata<-,GenomicRanges-method
(GRanges-class), 32
elementMetadata<-,GRangesList-method
(GRangesList-class), 39
encodeOverlaps,GRangesList,GRangesList,missing-method
(encodeOverlaps-methods), 17
encodeOverlaps-methods, 17
end,GappedAlignments-method
(GappedAlignments-class), 25
end,GenomicRanges-method
(GRanges-class), 32
end,GRangesList-method
(GRangesList-class), 39
end<-,GenomicRanges-method
(GRanges-class), 32
end<-,GRangesList-method
(GRangesList-class), 39
exptData (SummarizedExperiment-class),
53
exptData,SummarizedExperiment-method
(SummarizedExperiment-class),
53
exptData<-
(SummarizedExperiment-class),
53
exptData<-,SummarizedExperiment,list-method
(SummarizedExperiment-class),
53
exptData<-,SummarizedExperiment,SimpleList-method
(SummarizedExperiment-class),
53
extractSkippedExonRanks
(encodeOverlaps-methods), 17
extractSkippedExonRanks,character-method
(encodeOverlaps-methods), 17
extractSkippedExonRanks,factor-method
(encodeOverlaps-methods), 17
extractSkippedExonRanks,OverlapEncodings-method
(encodeOverlaps-methods), 17

findOverlaps, *13*, *18–20*
findOverlaps,ANY,GappedAlignments-method
(findOverlaps-methods), 19
findOverlaps,GappedAlignmentPairs,ANY-method
(findOverlaps-methods), 19
findOverlaps,GappedAlignments,ANY-method
(findOverlaps-methods), 19
findOverlaps,GappedAlignments,GappedAlignments-method
(findOverlaps-methods), 19
findOverlaps,GenomicRanges,GenomicRanges-method
(findOverlaps-methods), 19
findOverlaps,GenomicRanges,GRangesList-method
(findOverlaps-methods), 19

findOverlaps,GenomicRanges,RangedData-method (findOverlaps-methods), 19
findOverlaps,GenomicRanges,RangesList-method (findOverlaps-methods), 19
findOverlaps,GRangesList,GenomicRanges-method (findOverlaps-methods), 19
findOverlaps,GRangesList,GRangesList-method (findOverlaps-methods), 19
findOverlaps,GRangesList,RangedData-method (findOverlaps-methods), 19
findOverlaps,GRangesList,RangesList-method (findOverlaps-methods), 19
findOverlaps,RangedData,GenomicRanges-method (findOverlaps-methods), 19
findOverlaps,RangedData,GRangesList-method (findOverlaps-methods), 19
findOverlaps,RangesList,GenomicRanges-method (findOverlaps-methods), 19
findOverlaps,RangesList,GRangesList-method (findOverlaps-methods), 19
findOverlaps-methods, 19, *25*, *29*, *37*, *42*, *51*
first (GappedAlignmentPairs-class), 22
first,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22
flank,GenomicRanges-method (GRanges-class), 32
flank,GRangesList-method (GRangesList-class), 39
flipQuery (encodeOverlaps-methods), 17
follow,GenomicRanges,GenomicRanges-method (GRanges-class), 32
follow,GenomicRanges,missing-method (GRanges-class), 32

GappedAlignmentPairs, *15, 16*, *19, 20*, *26*, *46*
GappedAlignmentPairs (GappedAlignmentPairs-class), 22
GappedAlignmentPairs-class, *16*, *20*, 22, *29*, *46*
GappedAlignments, *13*, *15, 16*, *19, 20*, *22–24*, *46*, *50*, *58–61*
GappedAlignments (GappedAlignments-class), 25
GappedAlignments-class, *16*, *20*, *25*, 25, *46*, *51*
gaps, *35*
gaps,GenomicRanges-method (GRanges-class), 32
genome (seqinfo), 45
genome,ANY-method (seqinfo), 45
genome,Seqinfo-method (Seqinfo-class), 47
genome<- (seqinfo), 45
genome<-,ANY-method (seqinfo), 45
genome<-,Seqinfo-method (Seqinfo-class), 47
GenomicRanges, *8*, *31, 32*, *44*, *60, 61*
GenomicRanges (GRanges-class), 32
GenomicRanges-class, *8*, *31*
GenomicRanges-class (GRanges-class), 32
GenomicRanges-comparison, 31
GenomicRangesList (GenomicRangesList-class), 32
GenomicRangesList-class, 32
GenomicRangesORGRangesList-class (GRanges-class), 32
GenomicRangesORmissing-class (GRanges-class), 32
GRanges, *13*, *15, 16*, *19, 20*, *28*, *45, 46*, *50*, *53*, *55–58*, *61*
GRanges (GRanges-class), 32
granges (GappedAlignments-class), 25
granges,GappedAlignments-method (GappedAlignments-class), 25
granges,RangesMapping-method (map-methods), 43
GRanges-class, *16*, *20*, *25*, *29*, 32, *42*, *46*, *51*
GRangesList, *13*, *15, 16*, *18–20*, *24*, *28*, *32*, *36*, *46*, *50*, *55*, *58*, *60, 61*
GRangesList (GRangesList-class), 39
GRangesList-class, *16*, *18*, *20*, *25*, *29*, *37*, 39, *46*, *51*
grg (GappedAlignments-class), 25
grglist (GappedAlignments-class), 25
grglist,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22
grglist,GappedAlignments-method (GappedAlignments-class), 25

Hits, *18*, *20*
Hits-class, *20*

intersect,GRanges,GRanges-method (setops-methods), 49
IntersectionNotEmpty (summarizeOverlaps), 57
IntersectionStrict (summarizeOverlaps), 57
IRanges, *4*, *16*, *28*, *33*
IRanges-class, *4*
IRanges-setops, *51*
IRangesList, *4*, *39*

IRangesList-class, *4*
is, *8*
isCircular *(*seqinfo*)*, 45
isCircular,ANY-method *(*seqinfo*)*, 45
isCircular,Seqinfo-method *(*Seqinfo-class*)*, 47
isCircular<- *(*seqinfo*)*, 45
isCircular<-,ANY-method *(*seqinfo*)*, 45
isCircular<-,Seqinfo-method *(*Seqinfo-class*)*, 47
isCompatibleWithSkippedExons *(*encodeOverlaps-methods*)*, 17
isCompatibleWithSkippedExons,character-method *(*encodeOverlaps-methods*)*, 17
isCompatibleWithSkippedExons,factor-method *(*encodeOverlaps-methods*)*, 17
isCompatibleWithSkippedExons,OverlapEncodings-method *(*encodeOverlaps-methods*)*, 17
isCompatibleWithSplicing *(*encodeOverlaps-methods*)*, 17
isCompatibleWithSplicing,character-method *(*encodeOverlaps-methods*)*, 17
isCompatibleWithSplicing,factor-method *(*encodeOverlaps-methods*)*, 17
isCompatibleWithSplicing,OverlapEncodings-method *(*encodeOverlaps-methods*)*, 17
isDisjoint,GenomicRanges-method *(*GRanges-class*)*, 32
isDisjoint,GRangesList-method *(*GRangesList-class*)*, 39
isProperPair *(*GappedAlignmentPairs-class*)*, 22
isProperPair,GappedAlignmentPairs-method *(*GappedAlignmentPairs-class*)*, 22

keepSeqlevels *(*utils*)*, 60
keepSeqlevels,GappedAlignments,character-method *(*utils*)*, 60
keepSeqlevels,GappedAlignments,GappedAlignments-method *(*utils*)*, 60
keepSeqlevels,GappedAlignments,GenomicRanges-method *(*utils*)*, 60
keepSeqlevels,GappedAlignments,GRangesList-method *(*utils*)*, 60
keepSeqlevels,GenomicRanges,character-method *(*utils*)*, 60
keepSeqlevels,GenomicRanges,GappedAlignments-method *(*utils*)*, 60
keepSeqlevels,GenomicRanges,GenomicRanges-method *(*utils*)*, 60
keepSeqlevels,GenomicRanges,GRangesList-method *(*utils*)*, 60
keepSeqlevels,GRangesList,character-method *(*utils*)*, 60
keepSeqlevels,GRangesList,GappedAlignments-method *(*utils*)*, 60
keepSeqlevels,GRangesList,GenomicRanges-method *(*utils*)*, 60
keepSeqlevels,GRangesList,GRangesList-method *(*utils*)*, 60

lapply, *41*, *42*
last *(*GappedAlignmentPairs-class*)*, 22
last,GappedAlignmentPairs-method *(*GappedAlignmentPairs-class*)*, 22
left *(*GappedAlignmentPairs-class*)*, 22
left,GappedAlignmentPairs-method *(*GappedAlignmentPairs-class*)*, 22
length,GappedAlignmentPairs-method *(*GappedAlignmentPairs-class*)*, 22
length,GappedAlignments-method *(*GappedAlignments-class*)*, 25
length,GenomicRanges-method *(*GRanges-class*)*, 32
length,Seqinfo-method *(*Seqinfo-class*)*, 47
List, *32*

makeGappedAlignmentPairs, *23*, *25*
makeGRangesListFromFeatureFragments *(*GRangesList-class*)*, 39
map, *44*
map,GenomicRanges,GappedAlignments-method *(*map-methods*)*, 43
map,GenomicRanges,GRangesList-method *(*map-methods*)*, 43
map-methods, 43
mapply, *42*
match,ANY,GappedAlignments-method *(*findOverlaps-methods*)*, 19
match,GappedAlignmentPairs,ANY-method *(*findOverlaps-methods*)*, 19
match,GappedAlignments,ANY-method *(*findOverlaps-methods*)*, 19
match,GappedAlignments,GappedAlignments-method *(*findOverlaps-methods*)*, 19
match,GenomicRanges,GenomicRanges-method *(*findOverlaps-methods*)*, 19
match,GenomicRanges,GRangesList-method *(*findOverlaps-methods*)*, 19

match,GenomicRanges,RangedData-method
(findOverlaps-methods), 19
match,GenomicRanges,RangesList-method
(findOverlaps-methods), 19
match,GRangesList,GenomicRanges-method
(findOverlaps-methods), 19
match,GRangesList,GRangesList-method
(findOverlaps-methods), 19
match,GRangesList,RangedData-method
(findOverlaps-methods), 19
match,GRangesList,RangesList-method
(findOverlaps-methods), 19
match,RangedData,GenomicRanges-method
(findOverlaps-methods), 19
match,RangedData,GRangesList-method
(findOverlaps-methods), 19
match,RangesList,GenomicRanges-method
(findOverlaps-methods), 19
match,RangesList,GRangesList-method
(findOverlaps-methods), 19
merge,missing,Seqinfo-method
(Seqinfo-class), 47
merge,NULL,Seqinfo-method
(Seqinfo-class), 47
merge,Seqinfo,missing-method
(Seqinfo-class), 47
merge,Seqinfo,NULL-method
(Seqinfo-class), 47
merge,Seqinfo,Seqinfo-method
(Seqinfo-class), 47

names, *26*
names,GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
22
names,GappedAlignments-method
(GappedAlignments-class), 25
names,GenomicRanges-method
(GRanges-class), 32
names,Seqinfo-method (Seqinfo-class), 47
names<-,GappedAlignmentPairs-method
(GappedAlignmentPairs-class),
22
names<-,GappedAlignments-method
(GappedAlignments-class), 25
names<-,GenomicRanges-method
(GRanges-class), 32
names<-,Seqinfo-method (Seqinfo-class),
47
narrow,GappedAlignments-method
(GappedAlignments-class), 25
nearest,GenomicRanges,GenomicRanges-method
(GRanges-class), 32
nearest,GenomicRanges,missing-method
(GRanges-class), 32
ngap,GappedAlignments-method
(GappedAlignments-class), 25

order,GenomicRanges-method
(GenomicRanges-comparison), 31
OverlapEncodings, *18*
OverlapEncodings-class, *18*

pgap,GRanges,GRanges-method
(setops-methods), 49
pintersect, *50*
pintersect,GappedAlignments,GRanges-method
(setops-methods), 49
pintersect,GRanges,GappedAlignments-method
(setops-methods), 49
pintersect,GRanges,GRanges-method
(setops-methods), 49
pintersect,GRanges,GRangesList-method
(setops-methods), 49
pintersect,GRangesList,GRanges-method
(setops-methods), 49
pintersect,GRangesList,GRangesList-method
(setops-methods), 49
precede,GenomicRanges,GenomicRanges-method
(GRanges-class), 32
precede,GenomicRanges,missing-method
(GRanges-class), 32
psetdiff,GRanges,GRanges-method
(setops-methods), 49
psetdiff,GRanges,GRangesList-method
(setops-methods), 49
psetdiff,GRangesList,GRangesList-method
(setops-methods), 49
punion,GRanges,GRanges-method
(setops-methods), 49
punion,GRanges,GRangesList-method
(setops-methods), 49
punion,GRangesList,GRanges-method
(setops-methods), 49

qnarrow (GappedAlignments-class), 25
qnarrow,GappedAlignments-method
(GappedAlignments-class), 25
queryLoc2refLoc (cigar-utils), 2
queryLocs2refLocs (cigar-utils), 2
qwidth (GappedAlignments-class), 25
qwidth,GappedAlignments-method
(GappedAlignments-class), 25

range,GenomicRanges-method
(GRanges-class), 32

range,GRangesList-method (GRangesList-class), 39
RangedData, *19*
RangedDataList, *39*
Ranges, *18*, *28*
ranges,GappedAlignments-method (GappedAlignments-class), 25
ranges,GRanges-method (GRanges-class), 32
ranges,GRangesList-method (GRangesList-class), 39
Ranges-class, *37*
Ranges-comparison, *31*
ranges<-,GenomicRanges-method (GRanges-class), 32
ranges<-,GRangesList-method (GRangesList-class), 39
RangesList, *18, 19*, *28*
RangesList-class, *42*
RangesMapping, *44*
rank,GenomicRanges-method (GenomicRanges-comparison), 31
readBamGappedAlignmentPairs, *22, 23*, *25*
readBamGappedAlignments, *27*, *29*
readGappedAlignmentPairs (GappedAlignmentPairs-class), 22
readGappedAlignments (GappedAlignments-class), 25
reduce, *35*
reduce,GenomicRanges-method (GRanges-class), 32
reduce,GRangesList-method (GRangesList-class), 39
renameSeqlevels (utils), 60
renameSeqlevels,GappedAlignments,character-method (utils), 60
renameSeqlevels,GenomicRanges,character-method (utils), 60
renameSeqlevels,GRangesList,character-method (utils), 60
resize,GenomicRanges-method (GRanges-class), 32
restrict, *35*
restrict,GenomicRanges-method (GRanges-class), 32
restrict,GRangesList-method (GRangesList-class), 39
rglist (GappedAlignments-class), 25
rglist,GappedAlignments-method (GappedAlignments-class), 25
right (GappedAlignmentPairs-class), 22
right,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22
Rle, *27*, *33*, *53*
Rle-class, *37*
RleList, *16*
RleList-class, *4*, *16*, *42*
rname (GappedAlignments-class), 25
rname,GappedAlignments-method (GappedAlignments-class), 25
rname<- (GappedAlignments-class), 25
rname<-,GappedAlignments-method (GappedAlignments-class), 25
rowData (SummarizedExperiment-class), 53
rowData,SummarizedExperiment-method (SummarizedExperiment-class), 53
rowData<- (SummarizedExperiment-class), 53
rowData<-,SummarizedExperiment,GenomicRanges-method (SummarizedExperiment-class), 53
rowData<-,SummarizedExperiment,GRangesList-method (SummarizedExperiment-class), 53

sapply, *42*
ScanBamParam, *58*
score,GenomicRanges-method (GRanges-class), 32
score,GRangesList-method (GRangesList-class), 39
selectEncodingWithCompatibleStrand (encodeOverlaps-methods), 17
Seqinfo, *23*, *27*, *34*, *39*, *45, 46*
Seqinfo (Seqinfo-class), 47
seqinfo, *25*, *29*, *37*, *42*, 45, *48*
seqinfo,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22
seqinfo,GappedAlignments-method (GappedAlignments-class), 25
seqinfo,GRanges-method (GRanges-class), 32
seqinfo,GRangesList-method (GRangesList-class), 39
seqinfo,List-method (seqinfo), 45
seqinfo,RangedData-method (seqinfo), 45
seqinfo,RangesList-method (seqinfo), 45
Seqinfo-class, *46*, 47
seqinfo<- (seqinfo), 45
seqinfo<-,GappedAlignmentPairs-method (GappedAlignmentPairs-class),

22
seqinfo<-,GappedAlignments-method (GappedAlignments-class), 25
seqinfo<-,GenomicRanges-method (GRanges-class), 32
seqinfo<-,GRangesList-method (GRangesList-class), 39
seqinfo<-,List-method (seqinfo), 45
seqinfo<-,RangedData-method (seqinfo), 45
seqlengths (seqinfo), 45
seqlengths,ANY-method (seqinfo), 45
seqlengths,Seqinfo-method (Seqinfo-class), 47
seqlengths<- (seqinfo), 45
seqlengths<-,ANY-method (seqinfo), 45
seqlengths<-,Seqinfo-method (Seqinfo-class), 47
seqlevels, *23*, *27*, *34*, *39*
seqlevels (seqinfo), 45
seqlevels,ANY-method (seqinfo), 45
seqlevels,Seqinfo-method (Seqinfo-class), 47
seqlevels<- (seqinfo), 45
seqlevels<-,ANY-method (seqinfo), 45
seqlevels<-,Seqinfo-method (Seqinfo-class), 47
seqnames (seqinfo), 45
seqnames,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22
seqnames,GappedAlignments-method (GappedAlignments-class), 25
seqnames,GRanges-method (GRanges-class), 32
seqnames,GRangesList-method (GRangesList-class), 39
seqnames,RangedData-method (seqinfo), 45
seqnames,RangesList-method (seqinfo), 45
seqnames,Seqinfo-method (Seqinfo-class), 47
seqnames<- (seqinfo), 45
seqnames<-,GappedAlignments-method (GappedAlignments-class), 25
seqnames<-,GenomicRanges-method (GRanges-class), 32
seqnames<-,GRangesList-method (GRangesList-class), 39
seqnames<-,Seqinfo-method (Seqinfo-class), 47
seqselect,GenomicRanges-method (GRanges-class), 32
seqselect<-,GenomicRanges-method (GRanges-class), 32
setClass, *8*
setdiff,GRanges,GRanges-method (setops-methods), 49
setMethod, *8*
setops-methods, *29*, *37*, *42*, 49
shift,GenomicRanges-method (GRanges-class), 32
shift,GRangesList-method (GRangesList-class), 39
show,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22
show,GappedAlignments-method (GappedAlignments-class), 25
show,GenomicRanges-method (GRanges-class), 32
show,GRangesList-method (GRangesList-class), 39
show,Seqinfo-method (Seqinfo-class), 47
show,SummarizedExperiment-method (SummarizedExperiment-class), 53
showMethods, *8*
SimpleGenomicRangesList-class (GenomicRangesList-class), 32
SimpleList, *56*, *57*
solveUserSEW, *3*, *29*
sort,GenomicRanges-method (GenomicRanges-comparison), 31
split,GRanges-method (GRanges-class), 32
splitCigar (cigar-utils), 2
start,GappedAlignments-method (GappedAlignments-class), 25
start,GenomicRanges-method (GRanges-class), 32
start,GRangesList-method (GRangesList-class), 39
start<-,GenomicRanges-method (GRanges-class), 32
start<-,GRangesList-method (GRangesList-class), 39
strand, *33*, *53*
strand,character-method (strand-utils), 52
strand,DataTable-method (strand-utils), 52
strand,factor-method (strand-utils), 52
strand,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22

strand,GappedAlignments-method (GappedAlignments-class), 25
strand,GRanges-method (GRanges-class), 32
strand,GRangesList-method (GRangesList-class), 39
strand,integer-method (strand-utils), 52
strand,logical-method (strand-utils), 52
strand,missing-method (strand-utils), 52
strand,Rle-method (strand-utils), 52
strand-utils, 52
strand<-,DataTable-method (strand-utils), 52
strand<-,GappedAlignments-method (GappedAlignments-class), 25
strand<-,GenomicRanges-method (GRanges-class), 32
strand<-,GRangesList-method (GRangesList-class), 39
subsetByOverlaps,ANY,GappedAlignments-method (findOverlaps-methods), 19
subsetByOverlaps,GappedAlignmentPairs,ANY-method (findOverlaps-methods), 19
subsetByOverlaps,GappedAlignments,ANY-method (findOverlaps-methods), 19
subsetByOverlaps,GappedAlignments,GappedAlignments-method (findOverlaps-methods), 19
subsetByOverlaps,GenomicRanges,GenomicRanges-method (findOverlaps-methods), 19
subsetByOverlaps,GenomicRanges,GRangesList-method (findOverlaps-methods), 19
subsetByOverlaps,GenomicRanges,RangedData-method (findOverlaps-methods), 19
subsetByOverlaps,GenomicRanges,RangesList-method (findOverlaps-methods), 19
subsetByOverlaps,GRangesList,GenomicRanges-method (findOverlaps-methods), 19
subsetByOverlaps,GRangesList,GRangesList-method (findOverlaps-methods), 19
subsetByOverlaps,GRangesList,RangedData-method (findOverlaps-methods), 19
subsetByOverlaps,GRangesList,RangesList-method (findOverlaps-methods), 19
subsetByOverlaps,RangedData,GenomicRanges-method (findOverlaps-methods), 19
subsetByOverlaps,RangedData,GRangesList-method (findOverlaps-methods), 19
subsetByOverlaps,RangesList,GenomicRanges-method (findOverlaps-methods), 19
subsetByOverlaps,RangesList,GRangesList-method (findOverlaps-methods), 19
summarizeCigarTable (cigar-utils), 2
SummarizedExperiment, *58*, *59*
SummarizedExperiment (SummarizedExperiment-class), 53
SummarizedExperiment,list-method (SummarizedExperiment-class), 53
SummarizedExperiment,matrix-method (SummarizedExperiment-class), 53
SummarizedExperiment,missing-method (SummarizedExperiment-class), 53
SummarizedExperiment,SimpleList-method (SummarizedExperiment-class), 53
SummarizedExperiment-class, 53
summarizeOverlaps, 57
summarizeOverlaps,GRanges,GappedAlignments-method (summarizeOverlaps), 57
summarizeOverlaps,GRangesList,GappedAlignments-method (summarizeOverlaps), 57

TranscriptDb, *13*, *46*
TranscriptDb-class, *46*

Union (summarizeOverlaps), 57
union,GRanges,GRanges-method (setops-methods), 49
unique,GenomicRanges-method (GenomicRanges-comparison), 31
unlist,GappedAlignmentPairs-method (GappedAlignmentPairs-class), 22
updateObject,GappedAlignments-method (GappedAlignments-class), 25
updateObject,GRanges-method (GRanges-class), 32
updateObject,GRangesList-method (GRangesList-class), 39
updateObject,Seqinfo-method (Seqinfo-class), 47
utils, 60

validCigar (cigar-utils), 2
validObject, *8*
values, *36*, *55*
VCF, *61*
vector-class, *37*, *42*
Views, *37*, *41*

width,GappedAlignments-method (GappedAlignments-class), 25

width,GenomicRanges-method
(GRanges-class), 32
width,GRangesList-method
(GRangesList-class), 39
width<-,GenomicRanges-method
(GRanges-class), 32
width<-,GRangesList-method
(GRangesList-class), 39
window,GenomicRanges-method
(GRanges-class), 32