GenomicRanges

April 20, 2011

`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, merge.ranges=TRUE)
cigarToIRangesListByAlignment(cigar, pos, flag=NULL, drop.D.ranges=FALSE)
cigarToIRangesListByRName(cigar, rname, pos, flag=NULL, drop.D.ranges=FALSE,
                          merge.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.

merge.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 bits 0x004 and 0x400 have the following meaning: when bit 0x004 is ON then "the query sequence itself is unmapped" and when bit 0x400 is ON then "the read is either a PCR duplicate or an optical duplicate". 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 CompressedNormalIRangesList 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, merge.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)
```

GRanges-findOverlaps

GRanges and GRangesList Interval Overlaps

Description

Finds interval overlaps between a GRanges/GRangesList object and a GRanges/GRangesList object.

Usage

```
  ## S4 method for signature 'GenomicRanges,GenomicRanges':
findOverlaps(query, subject, maxgap = 0L, minoverlap = 1L,
             type = c("any", "start", "end"), select = c("all", "first"))
  ## S4 method for signature 'GenomicRanges,GenomicRanges':
countOverlaps(query, subject, maxgap = 0L, minoverlap = 1L,
              type = c("any", "start", "end"))
  ## S4 method for signature 'GenomicRanges,GenomicRanges':
subsetByOverlaps(query, subject, maxgap = 0L, minoverlap = 1L,
                 type = c("any", "start", "end"))
  ## 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 or GRangesList object. RangesList and RangedData are also accepted for one of `query` or `subject` (`x` or `table` for `match`).
`maxgap`	A non-negative integer representing the maximum distance between a query interval and a subject interval.
`minoverlap`	Ignored.
`type`	The type of acceptable overlap: `"any"` - any overlap within `maxgap`, `"start"` - the start of the `query` overlaps the start of the `subject` within `maxgap`, and `"end"` - the end of the `query` overlaps the end of the `subject` within `maxgap`.
`select`	Overlaps to return: `"all"` - select all overlaps, and `"first"` - select the first overlap.
`nomatch`	The integer value to be returned in the case when no match is found.
`incomparables`	This value is ignored.

Details

The `findOverlaps` methods involving GRanges and GRangesList objects use 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)`.

Value

For `findOverlaps` either a RangesMatching 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 and S. Falcon and M. Lawrence

See Also

`findOverlaps`, GRanges, GRangesList

Examples

```
## 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")
```

```
## 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))
```

`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.

WARNING! This is work-in-progress. Expect frequent changes in functionalities.

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).

Most of the time, 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 ids (QNAME 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.

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".

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.

Constructor

`readGappedAlignments(file, format="BAM", ...)`: Read a file as a GappedAlignments object. The function is just a front-end that delegates to a format-specific back-end function (any extra argument is 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 package first).

Accessor methods

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

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

`rname(x)`: Returns a character factor of length `length(x)` containing the name of the reference sequence for each alignment.

`rname(x) <- value`: Replace the name of the reference sequence for each alignment. `value` must be a character factor/vector, or a 'character' Rle, or a 'factor' Rle, with the same length as `x`.

`strand(x)`: Returns a character factor of length `length(x)` (with levels +, - and *) containing the strand in the reference sequence to which the query is aligned.

`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).

`grglist(x)`, `grg(x)`, `rglist(x)`: Returns a GRangesList, a GRanges or a CompressedNormalIRangesList object of length `length(x)` where each element represents the regions in the reference to which a query is aligned. See Details section above for more information.

`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(grg(x))` and `end(grg(x))`, respectively (or, alternatively, to `min(rglist(x))` and `max(rglist(x))`, respectively).

`width(x)`: Equivalent to `width(grg(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`.

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.

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).

`pintersect(x, y)`: Either `x` is a GappedAlignments object and `y` is a GRanges object or `x` is a GRanges object and `y` is a GappedAlignments object. Returns a new GappedAlignments object of the same length as the GappedAlignments input arguments. Like with `narrow`, the resulting "parallel" intersection is with respect to the reference.

`coverage(x, shift=0L, width=NULL, weight=1L)`: `x` is a GappedAlignments object. Returns a named RleList object with one element (integer-Rle) per unique reference sequence. Each element represents `x`'s coverage of the corresponding reference sequence, that is, how many times each nucleotide position in the sequence is covered by the alignments in `x`. Note that the semantic of the `coverage` method for GappedAlignments objects is different from the semantic of the method for Ranges objects (the latter returns a single integer-Rle object representing the coverage of all ranges relatively to a unique imaginary reference sequence).

`findOverlaps(query, subject, ...)`,`countOverlaps(query, subject, ...)`, `subsetByOverlaps(query, subject, ...)`,`match(x, table, nomatch=NA_integer_, incomparables=NULL)`,`x %in% table`: `query` or `subject` or both are GappedAlignments objects. `findOverlaps(query, subject, ...)` is equivalent to `findOverlaps(grglist(qu subject, ...)` when `query` is a GappedAlignments object, or to `findOverlaps(query, grglist(subject), ...)` when `subject` is a GappedAlignments object, or to `findOverlaps(grglis grglist(subject), ...)` when both are GappedAlignments objects. The same apply to `countOverlaps(query, subject, ...)` and `subsetByOverlaps(query, subject, ...)`. See ¿`findOverlaps,GRangesList,GenomicRanges-method`',

¿countOverlaps,GRangesList,GenomicRanges-method' and ¿subsetByOverlaps,GRanges method' for more information (in particular for descriptions of the extra arguments and the returned object).

Author(s)

H. Pages and P. Aboyoun

References

http://samtools.sourceforge.net/

See Also

readBamGappedAlignments, GRangesList-class, NormalIRanges-class, CompressedNormalIRangesList-class, coverage, RleList-class, pintersect,GRanges,GRanges-method, findOverlaps,GRangesList,GenomicRanges-method, countOverlaps,GRangesList,GenomicRanges-method, subsetByOverlaps,GRangesList,GenomicRanges-method,

Examples

```
library(Rsamtools)  # the toy file below is there
aln1_file <- system.file("extdata", "ex1.bam", package="Rsamtools")
aln1 <- readGappedAlignments(aln1_file)
aln1

## ---------------------------------------------------------------------
## A. BASIC MANIPULATION
## ---------------------------------------------------------------------
length(aln1)
head(aln1)
head(rname(aln1))
levels(rname(aln1))

## Rename the reference sequences:
rname(aln1) <- sub("seq", "chr", rname(aln1))
levels(rname(aln1))

head(strand(aln1))
head(cigar(aln1))
head(qwidth(aln1))
table(qwidth(aln1))

grglist(aln1)  # a GRangesList object
grg(aln1)      # a GRanges object
rglist(aln1)   # a CompressedNormalIRangesList object
stopifnot(identical(elementLengths(grglist(aln1)), elementLengths(rglist(aln1))))

head(start(aln1))
head(end(aln1))
head(width(aln1))
head(ngap(aln1))

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

```
aln1[strand(aln1) == "-"]
aln1[grep("I", cigar(aln1), fixed=TRUE)]
aln1[grep("N", cigar(aln1), fixed=TRUE)]  # no gaps

## A confirmation that all the queries map to the reference with no
## gaps:
stopifnot(all(ngap(aln1) == 0))

## Different ways to subset:
aln1[6]             # a GappedAlignments object of length 1
grglist(aln1)[[6]]  # a GRanges object of length 1
rglist(aln1)[[6]]   # a NormalIRanges object of length 1

## Ds are NOT gaps:
ii <- grep("D", cigar(aln1), fixed=TRUE)
aln1[ii]
ngap(aln1[ii])
grglist(aln1[ii])

## qwidth() vs width():
aln1[qwidth(aln1) != width(aln1)]

## This MUST return an empty object:
aln1[cigar(aln1) == "35M" & qwidth(aln1) != 35]
## but this doesn't have too:
aln1[cigar(aln1) != "35M" & qwidth(aln1) == 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(aln1, 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(aln1, start=21)
stopifnot(identical(end(qnarrow(aln1, start=21)), end(aln1)))

## ---------------------------------------------------------------------
## D. coverage()
## ---------------------------------------------------------------------
coverage(aln1)

## ---------------------------------------------------------------------
## E. findOverlaps()/countOverlaps()
## ---------------------------------------------------------------------
findOverlaps(aln1, grg(aln1)[1])
sum(countOverlaps(aln1, grg(aln1)[1]))
subsetByOverlaps(aln1, grg(aln1)[1])
table(match(aln1, grg(aln1)[1]), useNA = "ifany")
table(aln1 %in% grg(aln1)[1])
```

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"`, `"seqlengths"`, `"isCircular"`, `"start"`, `"end"`, `"width"`, or `"element"`.

Constructor

`GRanges(seqnames = Rle(), ranges = IRanges(), strand = Rle("*", length(seqnames ..., seqlengths = structure(rep(NA_integer_, length(levels(seqnames))), names = levels(seqnames)))`: Creates a GRanges object.

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

`ranges` IRanges object containing the ranges.

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

`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 or RangesList 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)` and `isCircular(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)` and `isCircular(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.

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

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

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

`seqinfo(x)`: Returns a Seqinfo object containing information about the underlying sequences.

`seqlengths(x)`, `seqlengths(x) <- value`: Gets or sets the seqlengths. `seqlengths(x)` is formally equivalent to `seqlengths(seqinfo(x))`. `value` can be a named non-negative integer or numeric vector.

`isCircular(x)`: Formally equivalent to `isCircular(seqinfo(x))`.

`isCircularWithKnownLength(x)`: Formally equivalent to `isCircularWithKnownLength(seqinfo(x`

`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.

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

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

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)`: 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) == "+"` are anchored at `start(x)`, elements where `strand(x) == "-"` are anchored at the `end(x)`, and elements where `strand = "*"` are anchored at `(end(x) - start(x))%/%2`. 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.

`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. 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.

`coverage(x, shift = list(0L), width = as.list(seqlengths(x)), weight = list(1L))`: Returns a named RleList object with one element ('integer' Rle) per unique sequence name representing how many times each position in the sequence is covered by the intervals in `x`. The `shift`, `width`, and `weight` arguments take list arguments, possibly named by the unique sequence names in `x`, whose elements are passed into the `coverage` method for IRanges object. See `?coverage` for more information on these optional arguments.

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`.

`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.

`rev(x)`: Returns a new object of the same class as `x` made of the original elements in the reverse order.

`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`.

Author(s)

P. Aboyoun

See Also

GRangesList-class, Seqinfo-class, Sequence-class, Ranges-class, Rle-class, DataFrame-class

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)
coverage(gr)

# Changing sequence name
unique(seqnames(gr))
seqnames(gr) <- sub("chr", "Chrom", seqnames(gr))
gr

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

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

GRangesList-class *GRangesList objects*

Description

The GRangesList class is a container for storing a collection of GRanges objects.

Constructor

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

Coercion

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

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

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.

`seqinfo(x)`: Returns a Seqinfo object containing information about the underlying sequences.

`seqlengths(x)`, `seqlengths(x) <- value`: Gets or sets the seqlengths. `seqlengths(x)` is formally equivalent to `seqlengths(seqinfo(x))`. `value` can be a named non-negative integer or numeric vector.

`isCircular(x)`: Formally equivalent to `isCircular(seqinfo(x))`.

`isCircularWithKnownLength(x)`: Formally equivalent to `isCircularWithKnownLength(seqinfo(x`

`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.

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.

`coverage(x, shift = list(0L), width = list(NULL), weight = list(1L))`: Returns a named RleList object with one element ('integer' Rle) per unique sequence name representing how many times each position in the sequence is covered by the intervals in `x`. The `shift`, `width`, and `weight` arguments take list arguments, possibly named by the unique sequence names in `x`, whose elements are passed into the `coverage` method for IRanges object. See `?coverage` for more information on these optional arguments.

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.

`rev(x)`: Returns a new object of the same class as `x` made of the original elements in the reverse order.

`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 "reduce" method

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

reduce(x, drop.empty.ranges=FALSE, min.gapwidth=1L): Reduces separately each element in x and returns them in a GRangesList object with the same length & names & elementMetadata & seqinfo as x. In other words, it is equivalent to endoapply(x, reduce, drop.empty.ranges=drop.empty.ranges, min.gapwidth=min.gapwidth).

Author(s)

P. Aboyoun

See Also

GRanges-class, Seqinfo-class Sequence-class, RangesList-class, RleList-class, DataFrameList-class

Examples

```
## Construction using 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))
coverage(grl)

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

# Changing sequence name
unique(seqnames(grl))
seqnames(grl) <- sub("chr", "Chrom", seqnames(grl))
grl

# reduce():
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.
```

GRanges-setops *GRanges and GRangesList Set and Parallel Set Operations*

Description

Set and parallel set operations for GRanges/GRangesList objects.

Usage

```
  ## Set operations
  ## S4 method for signature 'GRanges,GRanges':
union(x, y)
  ## S4 method for signature 'GRanges,GRanges':
intersect(x, y)
  ## S4 method for signature 'GRanges,GRanges':
setdiff(x, y)
```

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

Arguments

x, y	GRanges or GRangesList objects of equal length (i.e. `length(x) == length(y)`). For `union`, `intersect`, `setdiff`, `x` and `y` must both be GRanges objects. For `pintersect`, `x` and `y` cannot both be GRangesList objects. For `psetdiff`, `x` and must be a GRanges object.
fill.gap	Logical indicating whether or not to force a union by using the rule `start = min(start(x), start(y)), end = max(end(x), end(y))`.
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.)
...	Further arguments to be passed to or from other methods.

Details

The `pintersect` methods involving GRanges and GRangesList 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 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`, and `setdiff` a GRanges.

For `punion` and `pintersect` either a GRanges object when both `x` and `y` are GRanges objects or a GRangesList object when one of the arguments is a GRangesList 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

GRanges, GRangesList, findOverlaps,GenomicRanges,GenomicRanges-method, pintersect,GRanges,GRanges-method

Examples

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


## Union, intersection, and set difference of two GRanges objects
union(gr2, gr3)
intersect(gr2, gr3)
setdiff(gr2, gr3)


## Parallel intersection of two GRanges objects
pintersect(gr2, shift(gr2, 3))

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


## Parallel set difference of two GRanges objects
psetdiff(gr2, shift(gr2, 3))

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

`Seqinfo-class` *Seqinfo objects*

Description

A Seqinfo object is a data.frame-like object that contains basic information about a set of genomic sequences. Currently only the length and circularity flag of each sequence is stored but more information might be added in the future.

Details

Typically Seqinfo objects are not used directly but are part of higher level objects.

Constructor

`Seqinfo(seqnames, seqlengths=NA, isCircular=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`.

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

`seqlengths(x)`, `seqlengths(x) <- value`: Gets/sets the lengths of the sequences in `x`.

`isCircular(x)`, `isCircular(x) <- value`: Gets/sets the circularity flags of the sequences in `x`.

`isCircularWithKnownLength(x)`: Formally defined as `(isCircular(x) %in% TRUE) & !is.na(seqlengths(x))`.

Coercion

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

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

Author(s)

H. Pages

Examples

```
## Coming soon...
```

---

seqnames *Accessing the sequence names and lengths stored in an object*

Description

The `seqnames` and `seqlengths` generics are meant as accessors for the sequence names and lengths stored in an object, respectively.

Usage

```
seqnames(x)
seqlengths(x)
```

Arguments

`x`	The object from which to access the sequence information.

Details

Various classes provide `seqnames` and `seqlengths` methods for getting or setting the sequence names and lengths of an object of the class, respectively.

See for example `¿seqnames,GRanges-method` for the GRanges class and `¿seqnames,GRangesList-method` for the GRangesList class, both classes being defined in the GenomicRanges package.

For more examples see `¿seqnames,BSgenome-method` for the BSgenome class defined in the BSgenome package and `¿seqnames,TranscriptDb-method` for the TranscriptDb class defined in the GenomicFeatures package (you might need to install and load these packages first).

The full list can be seen with `showMethods("seqnames")` and `showMethods("seqlengths")` (for the get methods) and `showMethods("seqnames<-")` and `showMethods("seqlengths<-")` (for the replacement methods).

See Also

seqnames,GRanges-method, seqnames,GRangesList-method, seqnames,BSgenome-method seqnames,TranscriptDb-method

Examples

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

showMethods("seqlengths")
showMethods("seqlengths<-")

if (interactive())
  ?`seqnames,GRanges-method`
```

---

strand *Accessing strand information*

Description

The `strand` generic is meant as an accessor for strand information. Four methods are defined by the GenomicRanges package, described below.

Usage

```
strand(x)
```

Arguments

`x`	The object from which to obtain a strand factor, can be missing.

Details

If x is missing, returns an empty factor with the standard levels that any strand factor should have: +, −, and ⋆ (for either).

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 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

Examples

```
strand()
strand(c("+", "-", NA, "*"))
strand(c(-1L, 1L, NA, -1L, NA))
strand(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 method for signature 'SummarizedExperiment,NULL':
dimnames(x) <- value
## S4 method for signature 'SummarizedExperiment,list':
dimnames(x) <- value

## Subsetting

## S4 method for signature 'SummarizedExperiment':
[(x, i, j, ..., drop=TRUE)
## S4 method for signature 'SummarizedExperiment,ANY,ANY,SummarizedExperiment':
[(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 instance describing the ranges of interest. Row names, if present, become the row names of the SummarizedExperiment.
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.

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 instance, accessible using the rowData function, described below. The GRanges class 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]] <- value) 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.

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))
```

Index