Package ‘GenomicRanges’

April 23, 2016

Title Representation and manipulation of genomic intervals and variables defined along a genome

Description The ability to efficiently represent and manipulate genomic annotations and alignments is playing a central role when it comes to analyzing high-throughput sequencing data (a.k.a. NGS data). The GenomicRanges package defines general purpose containers for storing and manipulating genomic intervals and variables defined along a genome. More specialized containers for representing and manipulating short alignments against a reference genome, or a matrix-like summarization of an experiment, are defined in the GenomicAlignments and SummarizedExperiment packages respectively. Both packages build on top of the GenomicRanges infrastructure.

Version 1.22.4

Encoding UTF-8

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

Maintainer Bioconductor Package Maintainer `<maintainer@bioconductor.org>`

biocViews Genetics, Infrastructure, Sequencing, Annotation, Coverage, GenomeAnnotation

Depends R (>= 2.10), methods, BiocGenerics (>= 0.16.1), S4Vectors (>= 0.8.6), IRanges (>= 2.4.6), GenomeInfoDb (>= 1.1.20)

Imports utils, stats, XVector

LinkingTo S4Vectors, IRanges

Suggests Biobase, AnnotationDbi (>= 1.21.1), annotate, Biostrings (>= 2.25.3), Rsamtools (>= 1.13.53), SummarizedExperiment (>= 0.1.5), GenomicAlignments, rtracklayer, BSgenome, GenomicFeatures, Gviz, VariantAnnotation, AnnotationHub, DESeq, DEXSeq, edgeR, KEGGgraph, BiocStyle, digest, RUnit, BSgenome.Hsapiens.UCSC.hg19, BSgenome.Scerevisiae.UCSC.sacCer2, KEGG.db, hgu95av2.db, org.Hs.eg.db, org.Mm.eg.db, org.Sc.sgd.db, pasilla, pasillaBamSubset, TxDb.Athaliana.BioMart.plantsmart22, TxDb.Dmelanogaster.UCSC.dm3.ensGene, TxDb.Hsapiens.UCSC.hg19.knownGene

License Artistic-2.0

Collate utils.R phicoef.R transcript-utils.R constraint.R
strand-utils.R range-squeezers.R GenomicRanges-class.R
GRanges-class.R DelegatingGenomicRanges-class.R GNCList-class.R
GIntervalTree-class.R GenomicRanges-comparison.R
GenomicRangesList-class.R GRangesList-class.R
makeGRangesFromDataFrame.R SummarizedExperiment-class.R
SummarizedExperiment-rowData-methods.R RangedData-methods.R
intra-range-methods.R inter-range-methods.R coverage-methods.R
setops-methods.R findOverlaps-methods.R
findOverlaps-GIntervalTree-methods.R nearest-methods.R
mapCoords-methods.R absoluteRanges.R tileGenome.R
tile-methods.R genomicvars.R test_GenomicRanges_package.R zzz.R

NeedsCompilation yes

R topics documented:

absoluteRanges 3
Constraints 5
coverage-methods 11
DelegatingGenomicRanges-class 13
findOverlaps-methods 14
GenomicRanges-comparison 17
GenomicRangesList-class 21
genomicvars 21
GIntervalTree-class 25
GNCList-class 28
GRanges-class 30
GRangesList-class 38
inter-range-methods 42
intra-range-methods 46
makeGRangesFromDataFrame 49
makeSummarizedExperimentFromExpressionSet 52
mapCoords-methods 52
nearest-methods 54
phicoef 58
range-squeezers 59
setops-methods 60
strand-utils 63
SummarizedExperiment-class 65
tile-methods 72
tileGenome 73

Index **76**

`absoluteRanges` *Transform genomic ranges into "absolute" ranges*

Description

`absoluteRanges` transforms the genomic ranges in `x` into *absolute* ranges i.e. into ranges counted from the beginning of the virtual sequence obtained by concatenating all the sequences in the underlying genome (in the order reported by `seqlevels(x)`).

`relativeRanges` performs the reverse transformation.

NOTE: These functions only work on *small* genomes. See Details section below.

Usage

```
absoluteRanges(x)
relativeRanges(x, seqlengths)

## Related utility:
isSmallGenome(seqlengths)
```

Arguments

`x`	For `absoluteRanges`: a GenomicRanges object with ranges defined on a *small* genome (see Details section below). For `relativeRanges`: a Ranges object.
`seqlengths`	An object holding sequence lengths. This can be a named integer (or numeric) vector with no duplicated names as returned by `seqlengths()`, or any object from which sequence lengths can be extracted with `seqlengths()`. For `relativeRanges`, `seqlengths` must describe a *small* genome (see Details section below).

Details

Because `absoluteRanges` returns the *absolute* ranges in an IRanges object, and because an IRanges object cannot hold ranges with an end > `.Machine$integer.max` (i.e. >= 2^31 on most platforms), `absoluteRanges` cannot be used if the size of the underlying genome (i.e. the total length of the sequences in it) is > `.Machine$integer.max`. Utility function `isSmallGenome` is provided as a mean for the user to check upfront whether the genome is *small* (i.e. its size is <= `.Machine$integer.max`) or not, and thus compatible with `absoluteRanges` or not.

`relativeRanges` applies the same restriction by looking at the `seqlengths` argument.

Value

An IRanges object for `absoluteRanges`.

A GRanges object for `relativeRanges`.

`isSmallGenome` returns TRUE if the total length of the underlying sequences is <= `.Machine$integer.max` (e.g. Fly genome), FALSE if not (e.g. Human genome), or NA if it cannot be computed (because some sequence lengths are NA).

Author(s)

H. Pagès

See Also

- GRanges objects.
- IRanges objects in the **IRanges** package.
- Seqinfo objects and the `seqlengths` getter in the **GenomeInfoDb** package.
- genomicvars for manipulating genomic variables.
- The `tileGenome` function for putting tiles on a genome.

Examples

```
## ---------------------------------------------------------------------
## TOY EXAMPLE
## ---------------------------------------------------------------------

gr <- GRanges(Rle(c("chr2", "chr1", "chr3", "chr1"), 4:1),
              IRanges(1:10, width=5),
              seqinfo=Seqinfo(c("chr1", "chr2", "chr3"), c(100, 50, 20)))

ar <- absoluteRanges(gr)
ar

gr2 <- relativeRanges(ar, seqlengths(gr))
gr2

## Sanity check:
stopifnot(all(gr == gr2))

## ---------------------------------------------------------------------
## ON REAL DATA
## ---------------------------------------------------------------------

## With a "small" genome

library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
ex <- exons(txdb)
ex

isSmallGenome(ex)

## Note that because isSmallGenome() can return NA (see Value section
## above), its result should always be wrapped inside isTRUE() when
## used in an if statement:
```

```
if (isTRUE(isSmallGenome(ex))) {
    ar <- absoluteRanges(ex)
    ar

    ex2 <- relativeRanges(ar, seqlengths(ex))
    ex2  # original strand is not restored

    ## Sanity check:
    strand(ex2) <- strand(ex)  # restore the strand
    stopifnot(all(ex == ex2))
}

## With a "big" genome (but we can reduce it)

library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
ex <- exons(txdb)
isSmallGenome(ex)
## Not run:
    absoluteRanges(ex)  # error!

## End(Not run)

## However, if we are only interested in some chromosomes, we might
## still be able to use absoluteRanges():
seqlevels(ex, force=TRUE) <- paste0("chr", 1:10)
isSmallGenome(ex)  # TRUE!
ar <- absoluteRanges(ex)
ex2 <- relativeRanges(ar, seqlengths(ex))

## Sanity check:
strand(ex2) <- strand(ex)
stopifnot(all(ex == ex2))
```

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

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

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

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

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

Note

WARNING: This note is not true anymore as the `constraint` slot has been temporarily removed from GenomicRanges objects (starting with package GenomicRanges >= 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. Pagès

See Also

`setClass`, `is`, `setMethod`, `showMethods`, `validObject`, GenomicRanges-class

Examples

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

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

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

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

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

## Like validity methods, "checkConstraint" methods must return NULL or
## a character vector describing the problems found. They should never
## fail i.e. they should never raise an error.
setMethod("checkConstraint", c("GenomicRanges", "HasRangeTypeCol"),
    function(x, constraint, verbose=FALSE)
    {
        x_mcols <- mcols(x)
        idx <- match("rangeType", colnames(x_mcols))
        if (length(idx) != 1L || is.na(idx)) {
            msg <- c("'mcols(x)' must have exactly 1 column ",
                     "named \"rangeType\"")
            return(paste(msg, collapse=""))
        }
        rangeType <- x_mcols[[idx]]
        .LEVELS <- c("gene", "transcript", "exon", "cds", "5utr", "3utr")
        if (!is(rangeType, "Rle") ||
            S4Vectors:::anyMissing(runValue(rangeType)) ||
            !identical(levels(rangeType), .LEVELS))
        {
            msg <- c("'mcols(x)$rangeType' must be a ",
                     "'factor' Rle with no NAs and with levels: ",
                     paste(.LEVELS, collapse=", "))
            return(paste(msg, collapse=""))
        }
        NULL
    }
)

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

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

## Use is() to check whether the object has a given constraint or not:
#is(constraint(gr), "HasRangeTypeCol")  # TRUE
#\dontrun{
#mcols(gr)$rangeType[3] <- NA  # will fail
#}
mcols(gr)$rangeType[3] <- NA
checkConstraint(gr, new("HasRangeTypeCol"))  # with GenomicRanges >= 1.7.9

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

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

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

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

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

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

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

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

setMethod("checkConstraint", c("GenomicRanges", "HasGCCol"),
    function(x, constraint, verbose=FALSE)
    {
        x_mcols <- mcols(x)
        idx <- match("GC", colnames(x_mcols))
        if (length(idx) != 1L || is.na(idx)) {
            msg <- c("'mcols(x)' must have exactly ",
                     "one column named \"GC\"")
            return(paste(msg, collapse=""))
        }
        GC <- x_mcols[[idx]]
        if (!is.numeric(GC) ||
            S4Vectors:::anyMissing(GC) ||
            any(GC < 0) || any(GC > 1))
        {
            msg <- c("'mcols(x)$GC' must be a numeric vector ",
                     "with no NAs and with values between 0 and 1")
            return(paste(msg, collapse=""))
        }
        NULL
    }
)

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

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

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

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

## The checkConstraint() generic will check the HasGCCol constraint
## first, and, if it's statisfied, it will then check the HighGCRanges
## constraint.
setMethod("checkConstraint", c("GenomicRanges", "HighGCRanges"),
    function(x, constraint, verbose=FALSE)
    {
        GC <- mcols(x)$GC
        if (!all(GC >= 0.5)) {
            msg <- c("all elements in 'mcols(x)$GC' ",
                     "must be >= 0.5")
            return(paste(msg, collapse=""))
        }
        NULL
    }
)

#\dontrun{
#constraint(gr) <- "HighGCRanges"  # will fail
#}
```

```
checkConstraint(gr, new("HighGCRanges"))  # with GenomicRanges >= 1.7.9
mcols(gr)$GC[6:10] <- 0.5
#constraint(gr) <- "HighGCRanges"  # OK
checkConstraint(gr, new("HighGCRanges"))  # with GenomicRanges >= 1.7.9

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

## ---------------------------------------------------------------------
## EXAMPLE 5: The 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")
```

`coverage-methods` *Coverage of a GRanges or GRangesList object*

Description

`coverage` methods for GRanges and GRangesList objects.

NOTE: The `coverage` generic function and methods for Ranges and RangesList objects are defined and documented in the **IRanges** package. Methods for GAlignments and GAlignmentPairs objects are defined and documented in the **GenomicAlignments** package.

Usage

```
## S4 method for signature 'GenomicRanges'
coverage(x, shift=0L, width=NULL, weight=1L,
            method=c("auto", "sort", "hash"))

## S4 method for signature 'GRangesList'
coverage(x, shift=0L, width=NULL, weight=1L,
            method=c("auto", "sort", "hash"))
```

Arguments

`x`	A GRanges or GRangesList object.
`shift`	A numeric vector or a list-like object. If numeric, it must be parallel to `x` (recycled if necessary). If a list-like object, it must have 1 list element per seqlevel in `x`, and its names must be exactly `seqlevels(x)`. Alternatively, `shift` can also be specified as a single string naming a metadata column in `x` (i.e. a column in `mcols(x)`) to be used as the `shift` vector. See `?coverage` in the **IRanges** package for more information about this argument.
`width`	Either `NULL` (the default), or an integer vector. If `NULL`, it is replaced with `seqlengths(x)`. Otherwise, the vector must have the length and names of `seqlengths(x)` and contain NAs or non-negative integers. See `?coverage` in the **IRanges** package for more information about this argument.
`weight`	A numeric vector or a list-like object. If numeric, it must be parallel to `x` (recycled if necessary). If a list-like object, it must have 1 list element per seqlevel in `x`, and its names must be exactly `seqlevels(x)`. Alternatively, `weight` can also be specified as a single string naming a metadata column in `x` (i.e. a column in `mcols(x)`) to be used as the `weight` vector. See `?coverage` in the **IRanges** package for more information about this argument.
`method`	See `?coverage` in the **IRanges** package for a description of this argument.

Details

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

Value

A named RleList object with one coverage vector per seqlevel in `x`.

Author(s)

H. Pagès and P. Aboyoun

See Also

- `coverage` in the **IRanges** package.
- coverage-methods in the **GenomicAlignments** package.
- RleList objects in the **IRanges** package.
- GRanges and GRangesList objects.

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))))
```

`DelegatingGenomicRanges-class`
DelegatingGenomicRanges objects

Description

The `DelegatingGenomicRanges` class implements the virtual `GenomicRanges` class using a delegate `GenomicRanges`. This enables developers to create `GenomicRanges` subclasses that add specialized columns or other structure, while remaining agnostic to how the data are actually stored. See the Extending GenomicRanges vignette.

Author(s)

M. Lawrence

`findOverlaps-methods` *Finding overlapping genomic ranges*

Description

Various methods for finding/counting overlaps between objects containing genomic ranges. This man page describes the methods that operate on GenomicRanges or GRangesList objects.

NOTE: The `findOverlaps` generic function and methods for Ranges and RangesList objects are defined and documented in the **IRanges** package. The methods for GAlignments, GAlignmentPairs, and GAlignmentsList objects are defined and documented in the **GenomicAlignments** package.

Usage

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

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

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

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

Arguments

`query, subject`	A GRanges or GRangesList object. RangesList and RangedData are also accepted for one of `query` or `subject`.

`maxgap, minoverlap, type, algorithm`	See `findOverlaps` in the **IRanges** package for a description of these arguments.
`select`	When `select` is `"all"` (the default), the results are returned as a Hits object. Otherwise the returned value is an integer vector parallel to `query` (i.e. same length) containing the first, last, or arbitrary overlapping interval in `subject`, with `NA` indicating intervals that did not overlap any intervals in `subject`.
`ignore.strand`	When set to `TRUE`, the strand information is ignored in the overlap calculations.

Details

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

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

Value

For `findOverlaps` either a Hits object when `select="all"` or an integer vector otherwise.

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

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

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

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

Author(s)

P. Aboyoun, S. Falcon, M. Lawrence, and H. Pagès

See Also

- The Hits class for representing a set of hits between 2 vector-like objects.
- The `findOverlaps` generic function defined in the **IRanges** package.
- The GNCList constructor and class for preprocessing and representing a GenomicRanges or object as a data structure based on Nested Containment Lists.
- The GRanges and GRangesList classes.

Examples

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

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

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

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

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

## ---------------------------------------------------------------------
## MORE EXAMPLES
## ---------------------------------------------------------------------

table(!is.na(findOverlaps(gr, gr1, select="arbitrary")))
countOverlaps(gr, gr1)
findOverlaps(gr, gr1)
subsetByOverlaps(gr, gr1)
```

```
## Overlaps between a GRanges and a GRangesList object:

table(!is.na(findOverlaps(grl, gr, select="first")))
countOverlaps(grl, gr)
findOverlaps(grl, gr)
subsetByOverlaps(grl, gr)
countOverlaps(grl, gr, type="start")
findOverlaps(grl, gr, type="start")
subsetByOverlaps(grl, gr, type="start")
findOverlaps(grl, gr, select="first")

table(!is.na(findOverlaps(grl, gr1, select="first")))
countOverlaps(grl, gr1)
findOverlaps(grl, gr1)
subsetByOverlaps(grl, gr1)
countOverlaps(grl, gr1, type="start")
findOverlaps(grl, gr1, type="start")
subsetByOverlaps(grl, gr1, type="start")
findOverlaps(grl, gr1, select="first")

## Overlaps between two GRangesList objects:
countOverlaps(grl, rev(grl))
findOverlaps(grl, rev(grl))
subsetByOverlaps(grl, rev(grl))
```

`GenomicRanges-comparison`

Comparing and ordering genomic ranges

Description

Methods for comparing and ordering the elements in one or more GenomicRanges objects.

Usage

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

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

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

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

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

## S4 method for signature 'GenomicRanges'
is.unsorted(x, na.rm=FALSE, strictly=FALSE, ignore.strand=FALSE)

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

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

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

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

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

Arguments

`x, table, y`	GenomicRanges objects.
`incomparables`	Not supported.
`fromLast, method, nomatch`	See ?`` `Ranges-comparison` `` in the IRanges package for a description of these arguments.
`ignore.strand`	Whether or not the strand should be ignored when comparing 2 genomic ranges.
`na.rm`	Ignored.
`strictly`	Logical indicating if the check should be for *strictly* increasing values.
`...`	Additional GenomicRanges objects used for breaking ties.
`na.last`	Ignored.
`decreasing`	`TRUE` or `FALSE`.
`ties.method`	A character string specifying how ties are treated. Only `"first"` is supported for now.
`by`	An optional formula that is resolved against `as.env(x)`; the resulting variables are passed to `order` to generate the ordering permutation.

Details

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

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

`is.unsorted()`, `order()`, `sort()`, and `rank()` on a GenomicRanges object behave accordingly to this "natural order".

==, `!=`, <=, >=, < and > on GenomicRanges objects also behave accordingly to this "natural order".

Author(s)

H. Pagès, `is.unsorted` contributed by Pete Hickey

See Also

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

Examples

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

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

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

## ---------------------------------------------------------------------
## C. match(), %in%
## ---------------------------------------------------------------------
table <- gr[1:7]
match(gr, table)
match(gr, table, ignore.strand=TRUE)

gr %in% table

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

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

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

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

is.unsorted(gr, ignore.strand=TRUE)
gr2 <- sort(gr, ignore.strand=TRUE)
is.unsorted(gr2)  # TRUE
is.unsorted(gr2, ignore.strand=TRUE)  # FALSE

## TODO: Broken. Please fix!
#sort(gr, by = ~ seqnames + start + end) # equivalent to (but slower than) above

score(gr) <- rev(seq_len(length(gr)))

## TODO: Broken. Please fix!
#sort(gr, by = ~ score)

rank(gr)

## ---------------------------------------------------------------------
## F. GENERALIZED ELEMENT-WISE COMPARISON OF 2 GenomicRanges OBJECTS
## ---------------------------------------------------------------------
gr3 <- GRanges(c(rep("chr1", 12), "chr2"), IRanges(c(1:11, 6:7), width=3))
strand(gr3)[12] <- "+"
gr4 <- GRanges("chr1", IRanges(5, 9))

compare(gr3, gr4)
rangeComparisonCodeToLetter(compare(gr3, gr4))
```

`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`.

`as(from, "RangedDataList")`: Supported `from` types include:

GenomicRangesList Each element of `from` is coerced to a `RangedData`.

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.

`genomicvars` *Manipulating genomic variables*

Description

A *genomic variable* is a variable defined along a genome. Here are 2 ways a genomic variable is generally represented in Bioconductor:

1. as a named RleList object with one list element per chromosome;
2. as a metadata column on a *disjoint* GRanges object.

This man page documents tools for switching from one form to the other.

Usage

```
bindAsGRanges(...)
mcolAsRleList(x, varname)
binnedAverage(bins, numvar, varname)
```

Arguments

`...`	One or more genomic variables in the form of named RleList objects.
`x`	A *disjoint* GRanges object with metadata columns on it. A GRanges object is said to be *disjoint* if it contains ranges that do not overlap with each other. This can be tested with `isDisjoint`. See `` ?`isDisjoint,GenomicRanges-method` `` for more information about the `isDisjoint` method for GRanges objects.
`varname`	The name of the genomic variable. For `mcolAsRleList` this must be the name of the metadata column on `x` to be turned into an RleList object. For `binnedAverage` this will be the name of the metadata column that contains the binned average in the returned object.
`bins`	A GRanges object representing the genomic bins. Typically obtained by calling `tileGenome` with `cut.last.tile.in.chrom=TRUE`.
`numvar`	A named RleList object representing a numerical variable defined along the genome covered by `bins` (which is the genome described by `seqinfo(bins)`).

Details

`bindAsGRanges` allows to switch the representation of one or more genomic variables from the *named RleList* form to the *metadata column on a disjoint GRanges object* form by binding the supplied named RleList objects together and putting them on the same GRanges object. This transformation is lossless.

`mcolAsRleList` performs the opposite transformation and is also lossless (however the circularity flags and genome information in `seqinfo(x)` won't propagate). It works for any metadata column on `x` that can be put in Rle form i.e. that is an atomic vector or a factor.

`binnedAverage` computes the binned average of a numerical variable defined along a genome.

Value

For `bindAsGRanges`: a GRanges object with 1 metadata column per supplied genomic variable.

For `mcolAsRleList`: a named RleList object with 1 list element per seqlevel in `x`.

For `binnedAverage`: the GRanges object `bins` with an additional metadata column named `varname` containing the binned average.

Author(s)

H. Pagès

See Also

- RleList objects in the **IRanges** package.
- coverage,GenomicRanges-method for computing the coverage of a GRanges object.
- The `tileGenome` function for putting tiles on a genome.
- GRanges objects and isDisjoint,GenomicRanges-method for the `isDisjoint` method for GenomicRanges objects.

Examples

```
## ---------------------------------------------------------------------
## A. TWO WAYS TO REPRESENT A GENOMIC VARIABLE
## -----------------------------------------------------------------

## 1) As a named RleList object
## ----------------------------
## Let's create a genomic variable in the "named RleList" form:
library(BSgenome.Scerevisiae.UCSC.sacCer2)
set.seed(55)
my_var <- RleList(
    lapply(seqlengths(Scerevisiae),
        function(seqlen) {
            tmp <- sample(50L, seqlen, replace=TRUE)
            Rle(cumsum(tmp - rev(tmp)))
        }
    ),
    compress=FALSE)
my_var

## 2) As a metadata column on a disjoint GRanges object
## ----------------------------------------------------
gr1 <- bindAsGRanges(my_var=my_var)
gr1

gr2 <- GRanges(c("chrI:1-150",
                 "chrI:211-285",
                 "chrI:291-377",
                 "chrV:51-60"),
               score=c(0.4, 8, -10, 2.2),
               id=letters[1:4],
               seqinfo=seqinfo(Scerevisiae))
gr2

## Going back to the "named RleList" form:
mcolAsRleList(gr1, "my_var")
score <- mcolAsRleList(gr2, "score")
score
id <- mcolAsRleList(gr2, "id")
id
bindAsGRanges(score=score, id=id)

## Bind 'my_var', 'score', and 'id' together:
```

```
gr3 <- bindAsGRanges(my_var=my_var, score=score, id=id)

## Sanity checks:
stopifnot(identical(my_var, mcolAsRleList(gr3, "my_var")))
stopifnot(identical(score, mcolAsRleList(gr3, "score")))
stopifnot(identical(id, mcolAsRleList(gr3, "id")))
gr2b <- bindAsGRanges(score=score, id=id)
seqinfo(gr2b) <- seqinfo(gr2)
stopifnot(identical(gr2, gr2b))

## ---------------------------------------------------------------------
## B. BIND TOGETHER THE COVERAGES OF SEVERAL BAM FILES
## ---------------------------------------------------------------------

library(pasillaBamSubset)
library(GenomicAlignments)
untreated1_cvg <- coverage(BamFile(untreated1_chr4()))
untreated3_cvg <- coverage(BamFile(untreated3_chr4()))
all_cvg <- bindAsGRanges(untreated1=untreated1_cvg,
                         untreated3=untreated3_cvg)

## Keep regions with coverage:
all_cvg[with(mcols(all_cvg), untreated1 + untreated3 >= 1)]

## Plot the coverage profiles with the Gviz package:
library(Gviz)
plotNumvars <- function(numvars, region, name="numvars", ...)
{
    stopifnot(is(numvars, "GRanges"))
    stopifnot(is(region, "GRanges"), length(region) == 1L)
    gtrack <- GenomeAxisTrack()
    dtrack <- DataTrack(numvars,
                        chromosome=as.character(seqnames(region)),
                        name=name,
                        groups=colnames(mcols(numvars)), type="l", ...)
    plotTracks(list(gtrack, dtrack), from=start(region), to=end(region))
}
plotNumvars(all_cvg, GRanges("chr4:1-25000"),
            "coverage", col=c("red", "blue"))
plotNumvars(all_cvg, GRanges("chr4:1.03e6-1.08e6"),
            "coverage", col=c("red", "blue"))

## Sanity checks:
stopifnot(identical(untreated1_cvg, mcolAsRleList(all_cvg, "untreated1")))
stopifnot(identical(untreated3_cvg, mcolAsRleList(all_cvg, "untreated3")))

## ---------------------------------------------------------------------
## C. COMPUTE THE BINNED AVERAGE OF A NUMERICAL VARIABLE DEFINED ALONG A
##    GENOME
## ---------------------------------------------------------------------

## In some applications (e.g. visualization), there is the need to compute
## the average of a genomic variable for a set of predefined fixed-width
## regions (sometimes called "bins").
## Let's use tileGenome() to create such a set of bins:
bins1 <- tileGenome(seqinfo(Scerevisiae), tilewidth=100,
                    cut.last.tile.in.chrom=TRUE)

## Compute the binned average for 'my_var' and 'score':
bins1 <- binnedAverage(bins1, my_var, "binned_var")
bins1
bins1 <- binnedAverage(bins1, score, "binned_score")
bins1

## Binned average in "named RleList" form:
binned_var1 <- mcolAsRleList(bins1, "binned_var")
binned_var1
stopifnot(all.equal(mean(my_var), mean(binned_var1)))  # sanity check

mcolAsRleList(bins1, "binned_score")

## With bigger bins:
bins2 <- tileGenome(seqinfo(Scerevisiae), tilewidth=50000,
                    cut.last.tile.in.chrom=TRUE)
bins2 <- binnedAverage(bins2, my_var, "binned_var")
bins2 <- binnedAverage(bins2, score, "binned_score")
bins2

binned_var2 <- mcolAsRleList(bins2, "binned_var")
binned_var2
stopifnot(all.equal(mean(my_var), mean(binned_var2)))  # sanity check

mcolAsRleList(bins2, "binned_score")

## Not surprisingly, the "binned" variables are much more compact in
## memory than the original variables (they contain much less runs):
object.size(my_var)
object.size(binned_var1)
object.size(binned_var2)
```

`GIntervalTree-class` *GIntervalTree objects*

Description

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

WARNING: GIntervalTree objects are defunct. Please use GNCList objects instead. See ?`GNCList` for more information.

Details

A common type of query that arises when working with intervals is finding which intervals in one set overlap those in another. An efficient family of algorithms for answering such queries is known as the Interval Tree. The `GIntervalTree` class implements persistent Interval Trees for efficient querying of genomic intervals. It uses the IntervalForest class to store a set of trees, one for each `seqlevel` in a `GRanges` object.

The simplest approach for finding overlaps is to call the `findOverlaps` function on a Ranges or other object with range information. See the man page of `findOverlaps` for how to use this and other related functions. A `GIntervalTree` object is a derivative of GenomicRanges and stores its genomic ranges as a set of trees (a forest, with one tree per `seqlevel`) that is optimized for overlap queries. Thus, for repeated queries against the same subject, it is more efficient to create a `GIntervalTree` once for the subject using the constructor described below and then perform the queries against the `GIntervalTree` instance.

Like its GenomicRanges parent class, the GIntervalTree class stores the sequences of genomic locations and associated annotations. Each element in the sequence is comprised of a sequence name, an interval, a strand, and optional metadata columns (e.g. score, GC content, etc.). This information is stored in four components as in GenomicRanges, but two of these components are treated in a specific way:

`ranges` an IntervalForest object containing the ranges stored as a set of interval trees.

`seqnames` these are not stored directly in this class, but are obtained from the `partitioning` component of the IntervalForest object stored in `ranges`.

Note that `GIntervalTree` objects are not supported for GRanges objects with circular genomes.

Constructor

`GIntervalTree(x)`: Creates a `GIntervalTree` object from a GRanges object.

`x` a GRanges object containing the genomic ranges.

Coercion

`as(from, "GIntervalTree")`: Creates a `GIntervalTree` object from a GRanges object. `as(from, "GRanges")`: Creates a GRanges object from an `GIntervalTree` object

Accessors

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

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

`seqnames(x)`: Get the sequence names.

`ranges(x)`: Get the ranges as an IRanges object. This is for consistency with the `ranges` accessor for GRanges objects. To access the underlying IntervalForest object use the `obj@ranges` form.

`strand(x)`: Get the strand.

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

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

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

`seqlevels(x)`: Get the sequence levels. These are stored in the `partition` slot of the underlying IntervalForest object.

`seqlengths(x)`: Get the sequence lengths.

`isCircular(x)`: Get the circularity flags. Note that GIntervalTree objects are not supported for circular genomes.

`genome(x)`: Get or the genome identifier or assembly name for each sequence.

`seqlevelsStyle(x)`: Get the seqname style for `x`. See the seqlevelsStyle generic getter in the **GenomeInfoDb** package for more information.

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

Ranges methods

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

`start(x)`: Get `start(ranges(x))`.

`end(x)`: Get `end(ranges(x))`.

`width(x)`: Get `width(ranges(x))`.

Subsetting

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

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

Other methods

`show(x)`: By default the `show` method displays 5 head and 5 tail elements. This can be changed by setting the global options `showHeadLines` and `showTailLines`. If the object length is less than (or equal to) the sum of these 2 options plus 1, then the full object is displayed.

Author(s)

Hector Corrada Bravo, P. Aboyoun

See Also

`seqinfo`, IntervalForest, IntervalTree, findOverlaps-methods,

Examples

```
## GIntervalTree objects are defunct. Please use GNCList objects
## instead. See ?GNCList for more information.
```

`GNCList-class` *GNCList objects*

Description

The GNCList class is a container for storing the Nested Containment List representation of a vector of genomic ranges (typically represented as a GRanges object). To preprocess a GRanges object, simply call the `GNCList` constructor function on it. The resulting GNCList object can then be used for efficient overlap-based operations on the genomic ranges.

Usage

```
GNCList(x)
```

Arguments

`x`	The GRanges (or more generally GenomicRanges) object to preprocess.

Details

The **IRanges** package also defines the `NCList` and `NCLists` constructors and classes for preprocessing and representing a Ranges or RangesList object as a data structure based on Nested Containment Lists.

Note that GNCList objects are replacements for GIntervalTree objects. The latter are defunct starting with BioC 3.2.

See ?`NCList` in the **IRanges** package for some important differences between the new algorithm based on Nested Containment Lists and the old algorithm based on interval trees. In particular, the new algorithm supports preprocessing of a GenomicRanges object with ranges defined on circular sequences (e.g. on the mitochnodrial chromosome). See below for some examples.

Value

A GNCList object.

Author(s)

H. Pagès

References

Alexander V. Alekseyenko and Christopher J. Lee – Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. Bioinformatics (2007) 23 (11): 1386-1393. doi: 10.1093/bioinformatics/btl647

See Also

- The `NCList` and `NCLists` constructors and classs defined in the **IRanges** package.
- `findOverlaps` for finding/counting interval overlaps between two *range-based* objects.
- GRanges objects.

Examples

```
## The examples below are for illustration purpose only and do NOT
## reflect typical usage. This is because, for a one time use, it is
## NOT advised to explicitely preprocess the input for findOverlaps()
## or countOverlaps(). These functions will take care of it and do a
## better job at it (by preprocessing only what's needed when it's
## needed, and release memory as they go).

## ---------------------------------------------------------------------
## PREPROCESS QUERY OR SUBJECT
## ---------------------------------------------------------------------

query <- GRanges(Rle(c("chrM", "chr1", "chrM", "chr1"), 4:1),
                 IRanges(1:10, width=5), strand=rep(c("+", "-"), 5))
subject <- GRanges(Rle(c("chr1", "chr2", "chrM"), 3:1),
                   IRanges(6:1, width=5), strand="+")

## Either the query or the subject of findOverlaps() can be preprocessed:

ppsubject <- GNCList(subject)
hits1a <- findOverlaps(query, ppsubject)
hits1a
hits1b <- findOverlaps(query, ppsubject, ignore.strand=TRUE)
hits1b

ppquery <- GNCList(query)
hits2a <- findOverlaps(ppquery, subject)
hits2a
hits2b <- findOverlaps(ppquery, subject, ignore.strand=TRUE)
hits2b

## Note that 'hits1a' and 'hits2a' contain the same hits but not
## necessarily in the same order.
stopifnot(identical(sort(hits1a), sort(hits2a)))
## Same for 'hits1b' and 'hits2b'.
stopifnot(identical(sort(hits1b), sort(hits2b)))

## ---------------------------------------------------------------------
## WITH CIRCULAR SEQUENCES
## ---------------------------------------------------------------------

seqinfo <- Seqinfo(c("chr1", "chr2", "chrM"),
                   seqlengths=c(100, 50, 10),
                   isCircular=c(FALSE, FALSE, TRUE))
seqinfo(query) <- seqinfo[seqlevels(query)]
seqinfo(subject) <- seqinfo[seqlevels(subject)]

ppsubject <- GNCList(subject)
hits3 <- findOverlaps(query, ppsubject)
hits3

## Circularity introduces more hits:

stopifnot(all(hits1a %in% hits3))
new_hits <- setdiff(hits3, hits1a)
new_hits  # 1 new hit
query[queryHits(new_hits)]
subject[subjectHits(new_hits)]  # positions 11:13 on chrM are the same
                                # as positions 1:3

## Sanity checks:
stopifnot(identical(new_hits, Hits(9, 6, 10, 6)))
ppquery <- GNCList(query)
hits4 <- findOverlaps(ppquery, subject)
stopifnot(identical(sort(hits3), sort(hits4)))
```

`GRanges-class` *GRanges objects*

Description

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

Details

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

`seqnames` a 'factor' Rle object containing the sequence names.

`ranges` an IRanges object containing the ranges.

`strand` a 'factor' Rle object containing the strand information.

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

`seqinfo` a Seqinfo object containing information about the set of genomic sequences present in the GRanges object.

Constructor

`GRanges(seqnames=Rle(), ranges=NULL, strand=NULL, ..., seqlengths=NUL`
Creates a GRanges object.

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

`ranges` An IRanges object containing the ranges.

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

`...` Optional metadata columns. These columns cannot be named `"start"`, `"end"`, `"width"`, or `"element"`.

`seqlengths` An integer vector named with `levels(seqnames)` and containing the lengths (or NA) for each level in `levels(seqnames)`.

`seqinfo` A Seqinfo object containing allowed sequence names, lengths (or NA), and circularity flag, for each level in `levels(seqnames)`.

If `ranges` is not supplied and/or NULL then the constructor proceeds in 2 steps:

1. An initial GRanges object is created with `as(seqnames, "GRanges")`.
2. Then this GRanges object is updated according to whatever non-NULL remaining arguments were passed to the call to `GRanges()`.

Because of this behavior, `GRanges(x)` is equivalent to `as(x, "GRanges")`.

Coercion

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

`as(from, "GRanges")`: Creates a GRanges object from a character vector, a factor, or a RangedData, or RangesList object.

When `from` is a character vector (or a factor), each element in it must represent a genomic range in format `chr1:2501-2800` (unstranded range) or `chr1:2501-2800:+` (stranded range). `..` is also supported as a separator between the start and end positions. Strand can be `+`, `-`, `*`, or missing. The names on `from` are propagated to the returned GRanges object. See `as.character()` and `as.factor()` below for the reverse transformations.

Coercing a data.frame or DataFrame into a GRanges object is also supported. See `makeGRangesFromDataFrame` for the details.

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

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

`as.character(x, ignore.strand=FALSE)`: Turn GRanges object `x` into a character vector where each range in `x` is represented by a string in format `chr1:2501-2800:+`. If `ignore.strand` is TRUE or if *all* the ranges in `x` are unstranded (i.e. their strand is set to `*`), then all the strings in the output are in format `chr1:2501-2800`.

The names on `x` are propagated to the returned character vector. Its metadata (`metadata(x)`) and metadata columns (`mcols(x)`) are ignored.

See `as(from, "GRanges")` above for the reverse transformation.

`as.factor(x)`: Equivalent to

```
factor(as.character(x), levels=as.character(sort(unique(x))))
```

See `as(from, "GRanges")` above for the reverse transformation.

Note that `table(x)` is supported on a GRanges object. It is equivalent to, but much faster than, `table(as.factor(x))`.

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

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

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

Accessors

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

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

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

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

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

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

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

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

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

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

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

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

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

`seqlevelsStyle(x)`, `seqlevelsStyle(x) <- value`: Get or set the seqname style for `x`. See the seqlevelsStyle generic getter and setter in the **GenomeInfoDb** package for more information.

`score(x), score(x) <- value`: Get or set the “score” column from the element metadata.

`granges(x, use.mcols=FALSE)`: Gets a `GRanges` with only the range information from `x`, unless `use.mcols` is `TRUE`, in which case the metadata columns are also returned. Those columns will include any "extra column slots" if `x` is a specialized `GenomicRanges` derivative.

Ranges methods

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

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

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

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

Splitting and Combining

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

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

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

`c(x, ..., ignore.mcols=FALSE)` If the `GRanges` objects have metadata columns (represented as one DataFrame per object), each such DataFrame must have the same columns in order to combine successfully. In order to circumvent this restraint, you can pass in an `ignore.mcols=TRUE` argument which will combine all the objects into one and drop all of their metadata columns.

`split(x, f, drop=FALSE)`: Splits `x` according to `f` to create a GRangesList object. If `f` is a list-like object then `drop` is ignored and `f` is treated as if it was `rep(seq_len(length(f)), sapply(f, length))`, so the returned object has the same shape as `f` (it also receives the names of `f`). Otherwise, if `f` is not a list-like object, empty list elements are removed from the returned object if `drop` is `TRUE`.

Subsetting

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

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

`x[i, j] <- value`: Replaces elements `i` and optional metadata columns `j` with `value`.

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that a GRanges object can be used to as a subscript to subset a list-like object that has names on it. In that case, the names on the list-like object are interpreted as sequence names. In the code snippets below, `x` is a list or List object with names on it, and the subscript `gr` is a GRanges object with all its seqnames being valid `x` names.

`x[gr]`: Return an object of the same class as `x` and *parallel* to `gr`. More precisely, it's conceptually doing:

```
lapply(gr, function(gr1) x[[seqnames(gr1)]][ranges(gr1)])
```

Other methods

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

Author(s)

P. Aboyoun and H. Pagès

See Also

- `makeGRangesFromDataFrame` for making a GRanges object from a data.frame or DataFrame object.
- `seqinfo` for accessing/modifying information about the underlying sequences of a GRanges object.
- GenomicRanges-comparison for comparing and ordering genomic ranges.
- intra-range-methods and inter-range-methods for intra range and inter range transformations of a GRanges object.
- coverage-methods for computing the coverage of a GRanges object.
- setops-methods for set operations on GRanges objects.
- findOverlaps-methods for finding overlapping genomic ranges.
- nearest-methods finding the nearest genomic range neighbor.
- `absoluteRanges` for transforming genomic ranges into *absolute* ranges (i.e. into ranges on the sequence obtained by virtually concatenating all the sequences in a genome).
- `tileGenome` for putting tiles on a genome.
- genomicvars for manipulating genomic variables.
- GRangesList objects.
- Ranges objects in the **IRanges** package.
- Vector, Rle, and DataFrame objects in the **S4Vectors** package.

Examples

```
## ---------------------------------------------------------------------
## CONSTRUCTION
## ---------------------------------------------------------------------
## Specifying the bare minimum i.e. seqnames and ranges only. The
## GRanges object will have no names, no strand information, and no
## metadata columns:
gr0 <- GRanges(Rle(c("chr2", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
               IRanges(1:10, width=10:1))
gr0

## Specifying names, strand, metadata columns. They can be set on an
## existing object:
names(gr0) <- head(letters, 10)
strand(gr0) <- Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2))
```

```
mcols(gr0)$score <- 1:10
mcols(gr0)$GC <- seq(1, 0, length=10)
gr0

## ... or specified at construction time:
gr <- GRanges(Rle(c("chr2", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
              IRanges(1:10, width=10:1, names=head(letters, 10)),
              Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
              score=1:10, GC=seq(1, 0, length=10))
stopifnot(identical(gr0, gr))

## Specifying the seqinfo. It can be set on an existing object:
seqinfo <- Seqinfo(paste0("chr", 1:3), c(1000, 2000, 1500), NA, "mock1")
seqinfo(gr0) <- merge(seqinfo(gr0), seqinfo)
seqlevels(gr0) <- seqlevels(seqinfo)

## ... or specified at construction time:
gr <- GRanges(Rle(c("chr2", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
              IRanges(1:10, width=10:1, names=head(letters, 10)),
              Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
              score=1:10, GC=seq(1, 0, length=10),
              seqinfo=seqinfo)
stopifnot(identical(gr0, gr))

## ---------------------------------------------------------------------
## COERCION
## ---------------------------------------------------------------------
## From GRanges:
as.character(gr)
as.factor(gr)
as.data.frame(gr)

## From character to GRanges:
x1 <- "chr2:56-125"
as(x1, "GRanges")
as(rep(x1, 4), "GRanges")
x2 <- c(A=x1, B="chr1:25-30:-")
as(x2, "GRanges")

## From data.frame to GRanges:
df <- data.frame(chrom="chr2", start=11:15, end=20:24)
gr3 <- as(df, "GRanges")

## Alternatively, coercion to GRanges can be done by just calling the
## GRanges() constructor on the object to coerce:
gr1 <- GRanges(x1)  # same as as(x1, "GRanges")
gr2 <- GRanges(x2)  # same as as(x2, "GRanges")
gr3 <- GRanges(df)  # same as as(df, "GRanges")

## Sanity checks:
stopifnot(identical(as(x1, "GRanges"), gr1))
stopifnot(identical(as(x2, "GRanges"), gr2))
stopifnot(identical(as(df, "GRanges"), gr3))
```

```
## ---------------------------------------------------------------------
## SUMMARIZING ELEMENTS
## ---------------------------------------------------------------------
table(seqnames(gr))
table(strand(gr))
sum(width(gr))
table(gr)
summary(mcols(gr)[,”score”])

## The number of lines displayed in the 'show' method are controlled
## with two global options:
longGR  <- sample(gr, 25, replace=TRUE)
longGR
options(showHeadLines=7)
options(showTailLines=2)
longGR

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

## ---------------------------------------------------------------------
## RENAMING THE UNDERLYING SEQUENCES
## ---------------------------------------------------------------------
seqlevels(gr)
seqlevels(gr) <- sub(”chr”, ”Chrom”, seqlevels(gr))
gr
seqlevels(gr) <- sub(”Chrom”, ”chr”, seqlevels(gr)) # revert

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

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

## ---------------------------------------------------------------------
## USING A GRANGES OBJECT AS A SUBSCRIPT TO SUBSET ANOTHER OBJECT
## ---------------------------------------------------------------------
## Subsetting *by* a GRanges subscript is supported only if the object
## to subset is a named list-like object:
```

```
x <- RleList(chr1=101:120, chr2=2:-8, chr3=31:40)
x[gr]
```

`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(), fragmer`
Constructs a GRangesList object from a list of fragmented features. See the Examples section below.

Accessors

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

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

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

`elementLengths(x)`: Get the `length` of each of the list elements.

`isEmpty(x)`: Returns a logical indicating either if the GRangesList has no elements or if all its elements are empty.

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

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

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

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

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

`strand(x)`, `strand(x) <- value`: Get or set the strand in the form of an RleList. `value` can be an RleList, CharacterList or single character. `value` as a single character converts all ranges in `x` to the same `value`; for selective strand conversion (i.e., mixed “+” and “-”) use RleList or CharacterList.

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

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

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

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

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

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

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

`seqlevelsStyle(x)`, `seqlevelsStyle(x) <- value`: Get or set the seqname style for `x`. See the seqlevelsStyle generic getter and setter in the **GenomeInfoDb** package for more information.

`score(x), score(x) <- value`: Get or set the “score” metadata column.

Coercion

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

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

`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.

Subsetting

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

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

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

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

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

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

- `times` Vector giving the number of times to repeat each element if of length `length(x)`, or to repeat the whole vector if of length 1.
- `length.out` Non-negative integer. The desired length of the output vector.
- `each` Non-negative integer. Each element of `x` is repeated `each` times.

`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.

Combining

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

`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`.

`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.

`unlist(x, recursive = TRUE, use.names = TRUE)`: Concatenates the elements of `x` into a single GRanges object.

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.

Author(s)

P. Aboyoun & H. Pagès

See Also

GRanges-class, `seqinfo`, Vector-class, RangesList-class, RleList-class, DataFrameList-class, intra-range-methods, inter-range-methods, 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

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

## isDisjoint():
isDisjoint(grl)

## disjoin():
disjoin(grl)  # metadata columns and order NOT preserved

## 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
```

`inter-range-methods` *Inter range transformations of a GenomicRanges or GRangesList object*

Description

See ?`` `intra-range-methods` `` and ?`` `inter-range-methods` `` in the IRanges package for a quick introduction to intra range and inter range transformations.

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

See ?`` `intra-range-methods` `` for intra range transformations of a GenomicRanges object.

Usage

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

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

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

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

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

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

Arguments

`x`	A GenomicRanges object.
`drop.empty.ranges, min.gapwidth, with.revmap, with.inframe.attrib, start, end`	See ?`` `inter-range-methods` `` in the IRanges package.
`ignore.strand`	`TRUE` or `FALSE`. Whether the strand of the input ranges should be ignored or not. See details below.
`...`	For `range`, additional `GenomicRanges` objects to consider. Ignored otherwise.
`na.rm`	Ignored.

Details

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

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

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

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

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

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

On a GRangesList object: When they are supported on GRangesList object `x`, the above inter range transformations will apply the transformation to each of the list elements in `x` and return a list-like object *parallel* to `x` (i.e. with 1 list element per list element in `x`). If `x` has names on it, they're propagated to the returned object.

Author(s)

H. Pagès and P. Aboyoun

See Also

- The GenomicRanges and GRanges classes.
- The Ranges class in the IRanges package.

- The inter-range-methods man page in the IRanges package.
- GenomicRanges-comparison for comparing and ordering genomic ranges.

Examples

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

gr1 <- GRanges(seqnames="chr2", ranges=IRanges(3, 6),
               strand="+", score=5L, GC=0.45)
gr2 <- GRanges(seqnames="chr1",
               ranges=IRanges(c(10, 7, 19), width=5),
               strand=c("+", "-", "+"), score=3:5, GC=c(0.3, 0.5, 0.66))
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

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

## On a GRanges object:
range(gr)

## On a GRangesList object:
range(grl)
range(grl, ignore.strand=TRUE)

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

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

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

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

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

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

biggr2 <- reduce(biggr, with.revmap=TRUE)
revmap <- mcols(biggr2)$revmap
expanded_biggr2 <- rep(biggr2, elementLengths(revmap))
reordered_biggr <- biggr[unlist(revmap)]
codes <- compare(expanded_biggr2, reordered_biggr)
alphacodes <- rangeComparisonCodeToLetter(compare(expanded_biggr2,
                                                  reordered_biggr))
stopifnot(all(alphacodes %in% c("d", "e", "g", "h")))
table(alphacodes)

## On a GRangesList object:
reduce(grl)  # Doesn't really reduce anything but note the reordering
             # of the inner elements in the 2nd and 3rd list elements:
             # the ranges are reordered by sequence name first (which
             # should appear in the same order as in 'seqlevels(grl)'),
             # and then by strand.
reduce(grl, ignore.strand=TRUE)  # 2nd list element got reduced

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

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

`intra-range-methods` *Intra range transformations of a GRanges or GRangesList object*

Description

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

See ?`` `intra-range-methods` `` and ?`` `inter-range-methods` `` in the **IRanges** package for a quick introduction to intra range and inter range transformations.

Intra range methods for GAlignments and GAlignmentsList objects are defined and documented in the **GenomicAlignments** package.

See ?`` `inter-range-methods` `` for inter range transformations of a GenomicRanges or GRangesList object.

Usage

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

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

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

## S4 method for signature 'GenomicRanges'
flank(x, width, start=TRUE, both=FALSE,
      use.names=TRUE, ignore.strand=FALSE)
## S4 method for signature 'GRangesList'
flank(x, width, start=TRUE, both=FALSE,
      use.names=TRUE, ignore.strand=FALSE)

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

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

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

Arguments

`x`	A GenomicRanges or GRangesList object.
`shift, use.names, start, end, width, both, fix, keep.all.ranges, upstream, downstream`	See ``?`intra-range-methods` ``.
`ignore.strand`	`TRUE` or `FALSE`. Whether the strand of the input ranges should be ignored or not. See details below.
`...`	Additional arguments to methods.

Details

- `shift` behaves like the `shift` method for Ranges objects. See ``?`intra-range-methods` `` for the details.
- ()`narrow` on a GenomicRanges object behaves like on a Ranges object. See ``?`intra-range-methods` `` for the details.

  A major difference though is that it returns a GenomicRanges object instead of a Ranges object. The returned object is *parallel* (i.e. same length and names) to the original object `x`.
- `resize` returns an object of the same type and length as `x` containing intervals that have been resized to width `width` based on the `strand(x)` values. Elements where `strand(x) == "+"` or `strand(x) == "*"` are anchored at `start(x)` and elements where `strand(x) == "-"` are anchored at the `end(x)`. The `use.names` argument determines whether or not to keep the names on the ranges.
- `flank` returns an object of the same type and length as `x` containing intervals of width `width` that flank the intervals in `x`. The `start` argument takes a logical indicating whether `x` should be flanked at the "start" (`TRUE`) or the "end" (`FALSE`), which for `strand(x) != "-"` is `start(x)` and `end(x)` respectively and for `strand(x) == "-"` is `end(x)` and `start(x)` respectively. The `both` argument takes a single logical value indicating whether the flanking region `width` positions extends *into* the range. If `both=TRUE`, the resulting range thus straddles the end point, with `width` positions on either side.
- `promoters` returns an object of the same type and length as `x` containing promoter ranges. Promoter ranges extend around the transcription start site (TSS) which is defined as `start(x)`. The `upsteam` and `downstream` arguments define the number of nucleotides in the 5' and 3' direction, respectively. The full range is defined as,

  (start(x) - upstream) to (start(x) + downstream - 1).

  Ranges on the `*` strand are treated the same as those on the `+` strand. When no seqlengths are present in `x`, it is possible to have non-positive start values in the promoter ranges. This occurs when (TSS - `upstream`) < 1. In the equal but opposite case, the end values of the ranges may extend beyond the chromosome end when (TSS + `downstream` + 1) > 'chromosome end'.

When `seqlengths` are not `NA` the promoter ranges are kept within the bounds of the defined `seqlengths`.

- `restrict` returns an object of the same type and length as `x` containing restricted ranges for distinct seqnames. The `start` and `end` arguments can be a named numeric vector of seqnames for the ranges to be resticted or a numeric vector or length 1 if the restriction operation is to be applied to all the sequences in `x`. See `` ?`intra-range-methods` `` for more information about range restriction and for a description of the optional arguments.
- `trim` trims out-of-bound ranges located on non-circular sequences whose length is not NA.

Author(s)

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

See Also

- GenomicRanges, GRanges, and GRangesList objects.
- The intra-range-methods man page in the **IRanges** package.
- The Ranges class in the **IRanges** package.

Examples

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

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

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

## ---------------------------------------------------------------------
## B. ON A GRangesList OBJECT
## ---------------------------------------------------------------------
gr1 <- GRanges("chr2", IRanges(3, 6))
gr2 <- GRanges(c("chr1", "chr1"), IRanges(c(7,13), width=3),
               strand=c("+", "-"))
gr3 <- GRanges(c("chr1", "chr2"), IRanges(c(1, 4), c(3, 9)),
               strand="-")
```

```
grl <- GRangesList(gr1= gr1, gr2=gr2, gr3=gr3)
grl

resize(grl, width=20)
flank(grl, width=20)
restrict(grl, start=3)
```

`makeGRangesFromDataFrame` *Make a GRanges object from a data.frame or DataFrame*

Description

`makeGRangesFromDataFrame` takes a data-frame-like object as input and tries to automatically find the columns that describe genomic ranges. It returns them as a GRanges object.

`makeGRangesFromDataFrame` is also the workhorse behind the coercion method from data.frame (or DataFrame) to GRanges.

Usage

```
makeGRangesFromDataFrame(df,
                         keep.extra.columns=FALSE,
                         ignore.strand=FALSE,
                         seqinfo=NULL,
                         seqnames.field=c("seqnames", "seqname",
                                          "chromosome", "chrom",
                                          "chr", "chromosome_name",
                                          "seqid"),
                         start.field="start",
                         end.field=c("end", "stop"),
                         strand.field="strand",
                         starts.in.df.are.0based=FALSE)
```

Arguments

`df`	A data.frame or DataFrame object. If not, then the function tries to turn `df` into a data frame with `as.data.frame(df)`.
`keep.extra.columns`	`TRUE` or `FALSE` (the default). If `TRUE`, the columns in `df` that are not used to form the genomic ranges of the returned GRanges object are then returned as metadata columns on the object. Otherwise, they are ignored. If `df` has a `width` column, then it's always ignored.
`ignore.strand`	`TRUE` or `FALSE` (the default). If `TRUE`, then the strand of the returned GRanges object is set to `"*"`.
`seqinfo`	Either `NULL`, or a Seqinfo object, or a character vector of seqlevels, or a named numeric vector of sequence lengths. When not `NULL`, it must be compatible with the genomic ranges in `df` i.e. it must include at least the sequence levels represented in `df`.

`seqnames.field`	A character vector of recognized names for the column in `df` that contains the chromosome name (a.k.a. sequence name) associated with each genomic range. Only the first name in `seqnames.field` that is found in `colnames(df)` is used. If no one is found, then an error is raised.
`start.field`	A character vector of recognized names for the column in `df` that contains the start positions of the genomic ranges. Only the first name in `start.field` that is found in `colnames(df)` is used. If no one is found, then an error is raised.
`end.field`	A character vector of recognized names for the column in `df` that contains the end positions of the genomic ranges. Only the first name in `start.field` that is found in `colnames(df)` is used. If no one is found, then an error is raised.
`strand.field`	A character vector of recognized names for the column in `df` that contains the strand associated with each genomic range. Only the first name in `strand.field` that is found in `colnames(df)` is used. If no one is found or if `ignore.strand` is `TRUE`, then the strand of the returned GRanges object is set to `"*"`.
`starts.in.df.are.0based`	`TRUE` or `FALSE` (the default). If `TRUE`, then the start positions of the genomic ranges in `df` are considered to be *0-based* and are converted to *1-based* in the returned GRanges object. This feature is intended to make it more convenient to handle input that contains data obtained from resources using the "0-based start" convention. A notorious example of such resource is the UCSC Table Browser (`http://genome.ucsc.edu/cgi-bin/hgTables`).

Value

A GRanges object with one element per row in the input.

If the `seqinfo` argument was supplied, the returned object will have exactly the seqlevels specified in `seqinfo` and in the same order. Otherwise, the seqlevels are ordered according to the output of the `rankSeqlevels` function (except if `df` contains the seqnames in the form of a factor-Rle, in which case the levels of the factor-Rle become the seqlevels of the returned object and with no re-ordering).

If `df` has non-automatic row names (i.e. `rownames(df)` is not `NULL` and is not `seq_len(nrow(df))`), then they will be used to set names on the returned GRanges object.

Note

Coercing data.frame or DataFrame `df` into a GRanges object (with `as(df, "GRanges")`), or calling `GRanges(df)`, are both equivalent to calling `makeGRangesFromDataFrame(df, keep.extra.columns=TRUE)`.

Author(s)

H. Pagès, based on a proposal by Kasper Daniel Hansen

See Also

- GRanges objects.
- Seqinfo objects and the `rankSeqlevels` function in the **GenomeInfoDb** package.

- The `makeGRangesListFromFeatureFragments` function for making a GRangesList object from a list of fragmented features.
- The `getTable` function in the **rtracklayer** package for an R interface to the UCSC Table Browser.
- DataFrame objects in the **S4Vectors** package.

Examples

```
## ---------------------------------------------------------------------
## BASIC EXAMPLES
## ---------------------------------------------------------------------

df <- data.frame(chr="chr1", start=11:15, end=12:16,
                 strand=c("+","-","+","*","."), score=1:5)
df
makeGRangesFromDataFrame(df)  # strand value "." is replaced with "*"

## The strand column is optional:
df <- data.frame(chr="chr1", start=11:15, end=12:16, score=1:5)
makeGRangesFromDataFrame(df)

gr <- makeGRangesFromDataFrame(df, keep.extra.columns=TRUE)
gr2 <- as(df, "GRanges")  # equivalent to the above
stopifnot(identical(gr, gr2))
gr2 <- GRanges(df)        # equivalent to the above
stopifnot(identical(gr, gr2))

makeGRangesFromDataFrame(df, ignore.strand=TRUE)
makeGRangesFromDataFrame(df, keep.extra.columns=TRUE,
                             ignore.strand=TRUE)

makeGRangesFromDataFrame(df, seqinfo=paste0("chr", 4:1))
makeGRangesFromDataFrame(df, seqinfo=c(chrM=NA, chr1=500, chrX=100))
makeGRangesFromDataFrame(df, seqinfo=Seqinfo(paste0("chr", 4:1)))

## ---------------------------------------------------------------------
## ABOUT AUTOMATIC DETECTION OF THE seqnames/start/end/strand COLUMNS
## ---------------------------------------------------------------------

## Automatic detection of the seqnames/start/end/strand columns is
## case insensitive:
df <- data.frame(ChRoM="chr1", StarT=11:15, stoP=12:16,
                 STRAND=c("+","-","+","*","."), score=1:5)
makeGRangesFromDataFrame(df)

## It also ignores a common prefix between the start and end columns:
df <- data.frame(seqnames="chr1", tx_start=11:15, tx_end=12:16,
                 strand=c("+","-","+","*","."), score=1:5)
makeGRangesFromDataFrame(df)

## The common prefix between the start and end columns is used to
## disambiguate between more than one seqnames column:
```

```
df <- data.frame(chrom="chr1", tx_start=11:15, tx_end=12:16,
                 tx_chr="chr2", score=1:5)
makeGRangesFromDataFrame(df)

## ---------------------------------------------------------------------
## 0-BASED VS 1-BASED START POSITIONS
## ---------------------------------------------------------------------

if (require(rtracklayer)) {
  session <- browserSession()
  genome(session) <- "sacCer2"
  query <- ucscTableQuery(session, "Most Conserved")
  df <- getTable(query)

  ## A common pitfall is to forget that the UCSC Table Browser uses the
  ## "0-based start" convention:
  gr0 <- makeGRangesFromDataFrame(df, keep.extra.columns=TRUE)
  head(gr0)
  min(start(gr0))

  ## The start positions need to be converted into 1-based positions,
  ## to adhere to the convention used in Bioconductor:
  gr1 <- makeGRangesFromDataFrame(df, keep.extra.columns=TRUE,
                                  starts.in.df.are.0based=TRUE)
  head(gr1)
}
```

`makeSummarizedExperimentFromExpressionSet`

Make a SummarizedExperiment object from an ExpressionSet

Description

WARNING: The SummarizedExperiment class is deprecated and being replaced with the RangedSummarizedExperiment class defined in the new **SummarizedExperiment** package.

Please make sure to install and load the **SummarizedExperiment** package where the `makeSummarizedExperimentFromExpr` function is now defined and documented. Note that the function now returns a RangedSummarizedExperiment instance instead of a SummarizedExperiment instance.

`mapCoords-methods`

Mapping ranges between sequences

Description

These functions are defunct. Use `mapToTranscripts` from the **GenomicFeatures** package or `mapToAlignments` from the **GenomicAlignments** package instead.

A method for translating a set of input ranges through a GRangesList object. Returns a GenomicRanges object.

The generics for `mapCoords` and `pmapCoords` are defined in the **IRanges** package. A method for translating a set of input ranges through a GAlignments object is defined and in the **GenomicAlignments** package.

Usage

```
## S4 method for signature 'GenomicRanges,GRangesList'
mapCoords(from, to, ...,
          ignore.strand = TRUE, elt.hits = FALSE)

## S4 method for signature 'GenomicRanges,GRangesList'
pmapCoords(from, to, ...,
          ignore.strand = TRUE, elt.hits = FALSE)
```

Arguments

`from`	The input ranges to map, usually a `GRanges`.
`to`	The alignment between the sequences in `from` and the sequences in the result, usually a `GRangesList`.
`ignore.strand`	`logical`; When TRUE strand is ignored in overlap operations.
`elt.hits`	`logical`; When TRUE, the output includes a metadata column, eltHits, with indices of the inner list elements of `to` hit by `from`. Useful for identifying elements of `to` hit by `from`. See examples.
`...`	Arguments passed to other methods. Currently not used.

Details

DEFUNCT! Use `mapToTranscripts` from the **GenomicFeatures** package or `mapToAlignments` from the **GenomicAlignments** package instead.

Each element in `to` 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`. Each outer list element of the GRangesList represents a transcript while each each individual element is an exon in the transcript.

`mapCoords` and `pmapCoords` translate the ranges in `from` relative to the transcript start (i.e., start of all ranges in `to`). The widths of the individual elements (exons in this example) are concatenated and counting starts at the 5' or 3' end depending on strand. Translated coordinates are only reported for ranges in `from` that fall completely ‘within‘ ranges in `to`.

The transcript-centric coordinates are are useful, for example, when predicting coding consequences of changes to the genomic sequence.

`mapCoords` maps the i-th element in `from` to each element in `to` returning in a many-to-many mapping. In contrast, `pmapCoords` treats the two inputs as parallel vectors and maps the i-th element of `from` to the i-th element of `to` returning a maximum of one result per input element.

Value

A `GRanges` object of mapped coordinates with matching data as metadata columns `fromHits` and `toHits`. The ranges in the output `GRanges` are position relative to the outer list element of `to`; all individual list elements are concatenated and counting starts at the 5' or 3' end depending on strand.

Matching data are the result of calling `findOverlaps` with type 'within' on ranges in `from` (the query) and the ranges in `to` (the subject). In the case of `mapCoords` matching can be many-to-one or one-to-many; one row is reported for each match. For `pmapCoords` matching is one-to-one as the i-th element in `from` is only mapped to the i-th element in `to`.

When `elt.hits` is TRUE, the `eltHits` metadata column includes the index of inner list elements in `to` hit by `from`. In some applications it may be useful to identify the exact list element that was overlapped. These elements can be extracted with the combination of `toHits` (outer list index) and `eltHits` (inner list index).

Author(s)

M. Lawrence and V. Obenchain `vobencha@fhcrc.org`

See Also

- The generic mapCoords-methods in the IRanges package.
- Additional methods in the GenomicAlignments package mapCoords-methods.

Examples

```
## DEFUNCT! See ?mapToTranscripts in the GenomicFeatures package and
## ?mapToAlignments in the GenomicAlignments package.
```

---

nearest-methods *Finding the nearest genomic range neighbor*

Description

The `nearest`, `precede`, `follow`, `distance` and `distanceToNearest` methods for `GenomicRanges` objects and subclasses.

Usage

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

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

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

## S4 method for signature 'GenomicRanges,GenomicRanges'
distanceToNearest(x, subject, algorithm=c("nclist", "intervaltree"),
                  ignore.strand=FALSE, ...)
## S4 method for signature 'GenomicRanges,missing'
distanceToNearest(x, subject, algorithm=c("nclist", "intervaltree"),
                  ignore.strand=FALSE, ...)

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

Arguments

`x`	The query GenomicRanges instance.
`subject`	The subject GenomicRanges instance within which the nearest neighbors are found. Can be missing, in which case `x` is also the subject.
`y`	For the `distance` method, a `GRanges` instance. Cannot be missing. If `x` and `y` are not the same length, the shortest will be recycled to match the length of the longest.
`select`	Logic for handling ties. By default, all methods select a single interval (arbitrary for `nearest`, the first by order in `subject` for `precede`, and the last for `follow`). 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.
`ignore.strand`	A `logical` indicating if the strand of the input ranges should be ignored. When `TRUE`, strand is set to `'+'`.
`algorithm`	This argument is passed to `findOverlaps`, which `nearest` and `distanceToNearest` use internally. See ?`findOverlaps` for more information. Note that it will be removed in BioC 3.3 so please don't use it unless you have a good reason to do so (e.g. troubleshooting).
`...`	Additional arguments for methods.

Details

- nearest: Performs conventional nearest neighbor finding. Returns an integer vector containing the index of the nearest neighbor range in `subject` for each range in `x`. If there is no nearest neighbor `NA` is returned. For details of the algorithm see the man page in IRanges, ?`nearest`.

- precede: For each range in `x`, `precede` returns the index of the range in `subject` that is directly preceded by the range in `x`. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in `subject`.
- follow: The opposite of `precede`, `follow` returns the index of the range in `subject` that is directly followed by the range in `x`. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in `subject`.
- Orientation and Strand: The relevant orientation for `precede` and `follow` is 5' to 3', consistent with the direction of translation. Because positional numbering along a chromosome is from left to right and transcription takes place from 5' to 3', `precede` and `follow` can appear to have 'opposite' behavior on the `+` and `-` strand. Using positions 5 and 6 as an example, 5 precedes 6 on the `+` strand but follows 6 on the `-` strand.

  A range with strand `*` can be compared to ranges on either the `+` or `-` strand. Below we outline the priority when ranges on multiple strands are compared. When `ignore.strand=TRUE` all ranges are treated as if on the `+` strand.

  - x on + strand can match to ranges on both + and * strands. In the case of a tie the first range by order is chosen.
  - x on - strand can match to ranges on both - and * strands. In the case of a tie the first range by order is chosen.
  - x on * strand can match to ranges on any of +, - or * strands. In the case of a tie the first range by order is chosen.
- distanceToNearest: Returns the distance for each range in `x` to its nearest neighbor in the `subject`.
- distance: Returns the distance for each range in `x` to the range in `y`. The behavior of `distance` has changed in Bioconductor 2.12. See the man page `?distance` in IRanges for details.

Value

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

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

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

Author(s)

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

See Also

- The GenomicRanges and GRanges classes.
- The Ranges class in the **IRanges** package.
- The Hits class in the **S4Vectors** package.
- The nearest-methods man page in the IRanges package.
- findOverlaps-methods for finding just the overlapping ranges.
- The nearest-methods man page in the GenomicFeatures package.

Examples

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

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

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

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


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

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

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

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

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

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

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

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

## distance() with GRanges and TxDb see the
## ?'distance,GenomicRanges,TxDb-method' man
## page in the GenomicFeatures package.
```

`phicoef` *Calculate the "phi coefficient" between two binary variables*

Description

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

Usage

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

Arguments

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

Value

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

Author(s)

H. Pagès

References

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

Examples

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

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

`range-squeezers` *Squeeze the ranges out of a range-based object*

Description

S4 generic functions for squeezing the ranges out of a range-based object.

`granges` returns them as a GRanges object, `grglist` as a GRangesList object, and `rglist` as a RangesList object.

Usage

```
granges(x, use.mcols=FALSE, ...)
grglist(x, use.mcols=FALSE, ...)
rglist(x, use.mcols=FALSE, ...)
```

Arguments

`x`	A range-based object e.g. a SummarizedExperiment, GAlignments, GAlignmentPairs, or GAlignmentsList object.
`use.mcols`	`TRUE` or `FALSE` (the default). Whether the metadata columns on `x` (accessible with `mcols(x)`) should be propagated to the returned object or not.
`...`	Additional arguments, for use in specific methods.

Details

The **GenomicRanges** and **GenomicAlignments** packages define and document methods for various types of range-based objects (e.g. for SummarizedExperiment, GAlignments, GAlignmentPairs, and GAlignmentsList objects). Other Bioconductor packages might as well.

Note that these functions can be seen as a specific kind of *object getters* as well as functions performing coercion. For some objects (e.g. GAlignments), `as(x, "GRanges")`, `as(x, "GRangesList")`, and `as(x, "RangesList")`, are equivalent to `granges(x, use.mcols=TRUE)`, `grglist(x, use.mcols=TRUE)`, and `rglist(x, use.mcols=TRUE)`, respectively.

Value

A GRanges object for `granges`.

A GRangesList object for `grglist`.

A RangesList object for `rglist`.

If `x` is a vector-like object (e.g. GAlignments), the returned object is expected to be *parallel* to `x`, that is, the i-th element in the output corresponds to the i-th element in the input. If `x` has names on it, they're propagated to the returned object. If `use.mcols` is TRUE and `x` has metadata columns on it (accessible with `mcols(x)`), they're propagated to the returned object.

Author(s)

H. Pagès

See Also

- GRanges and GRangesList objects.
- RangesList objects in the **IRanges** package.
- SummarizedExperiment objects.
- GAlignments, GAlignmentPairs, and GAlignmentsList objects in the **GenomicAlignments** package.

Examples

```
## See ?GAlignments in the GenomicAlignments package for some
## examples.
```

`setops-methods` *Set operations on GRanges and GRangesList objects*

Description

Performs set operations on GRanges and GRangesList objects.

NOTE: The `punion`, `pintersect`, `psetdiff`, and `pgap` generic functions and methods for Ranges objects are defined and documented in the **IRanges** package.

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, drop.nohit.ranges=FALSE,
           ignore.strand=FALSE, strict.strand=FALSE)
## 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`: `x` and `y` can be any combination of GRanges and/or GRangesList objects. 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(y)),`
`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`.
`drop.nohit.ranges`	If TRUE then elements in `x` that don't intersect with their corresponding element in `y` are removed from the result (so the returned object is no more parallel to the input). If FALSE (the default) then nothing is removed and a `hit` metadata column is added to the returned object to indicate elements in `x` that intersect with the corresponding element in `y`. For those that don't, the reported intersection is a zero-width range that has the same start as `x`.
`strict.strand`	If set to FALSE (the default), features on the `"*"` strand are treated as occurring on both the `"+"` and `"-"` strand. If set to TRUE, the strand of intersecting elements must be strictly the same.
`...`	Further arguments to be passed to or from other methods.

Details

The `pintersect` methods involving GRanges and/or 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 (unless `strict.strand` is set to TRUE, in which case the strand of intersecting elements must be strictly the same).

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 object.

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 and H. Pagès

See Also

setops-methods, GRanges-class, GRangesList-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
## ---------------------------------------------------------------------

punion(x, shift(x, 6))
## 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))
pintersect(x, shift(x, 7))
```

```
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)

## For a fast 'mendoapply(intersect, grlist, as(gr, "GRangesList"))'
## call pintersect() with 'strict.strand=TRUE' and call reduce() on
## the result with 'drop.empty.ranges=TRUE':
reduce(pintersect(grlist, gr, strict.strand=TRUE),
       drop.empty.ranges=TRUE)

## 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,ANY'
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 strand levels" i.e. `+`, `-`, and `*`.

If `x` is a character vector or factor, it is coerced to a factor with the levels listed above. `NA` values in `x` are not accepted.

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

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

If `x` is a character-, factor-, integer-, or logical-Rle, it is transformed with `runValue(x) <- strand(runValue(x))` and returned.

If `x` inherits from `DataTable`, the `"strand"` column is passed thru `strand()` and returned. If `x` has no `"strand"` column, this return value is populated with `*`s.

Value

A factor (or factor-Rle) with the "standard strand levels" (i.e. `+`, `-`, and `*`) and no NAs.

Author(s)

M. Lawrence and H. Pagès

See Also

`strand`

Examples

```
strand()

x1 <- c("-", "*", "*", "+", "-", "*")
x2 <- factor(c("-", "-", "+", "-"))
x3 <- c(-1L, NA, NA, 1L, -1L, NA)
x4 <- c(TRUE, NA, NA, FALSE, TRUE, NA)

strand(x1)
strand(x2)
strand(x3)
strand(x4)

strand(Rle(x1))
strand(Rle(x2))
strand(Rle(x3))
strand(Rle(x4))

strand(DataFrame(score=2:-3))
strand(DataFrame(score=2:-3, strand=x3))
strand(DataFrame(score=2:-3, strand=Rle(x3)))

## Sanity checks:
target <- strand(x1)
stopifnot(identical(target, strand(x3)))
stopifnot(identical(target, strand(x4)))

stopifnot(identical(Rle(strand(x1)), strand(Rle(x1))))
stopifnot(identical(Rle(strand(x2)), strand(Rle(x2))))
stopifnot(identical(Rle(strand(x3)), strand(Rle(x3))))
stopifnot(identical(Rle(strand(x4)), strand(Rle(x4))))
```

`SummarizedExperiment-class`

SummarizedExperiment instances

Description

WARNING: The SummarizedExperiment class described here is deprecated and being replaced with the RangedSummarizedExperiment class defined in the new **SummarizedExperiment** package. Please make sure to install the **SummarizedExperiment** package before you attempt to use the `SummarizedExperiment()` constructor function. Note that this will return a RangedSummarizedExperiment instance instead of a SummarizedExperiment instance.

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

Usage

```
## Constructors

SummarizedExperiment(assays, ...)

## Accessors

assayNames(x, ...)
assayNames(x, ...) <- value
assays(x, ..., withDimnames=TRUE)
assays(x, ..., withDimnames=TRUE) <- value
assay(x, i, ...)
assay(x, i, ...) <- value
rowRanges(x, ...)
rowRanges(x, ...) <- value
colData(x, ...)
colData(x, ...) <- value
exptData(x, ...)
exptData(x, ...) <- value
## S4 method for signature 'SummarizedExperiment'
dim(x)
## S4 method for signature 'SummarizedExperiment'
dimnames(x)
## S4 replacement method for signature 'SummarizedExperiment,NULL'
dimnames(x) <- value
## S4 replacement method for signature 'SummarizedExperiment,list'
dimnames(x) <- value

## colData access

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

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

## Subsetting

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

```
x[i, j] <- value
## S4 method for signature 'SummarizedExperiment'
subset(x, subset, select, ...)

## Combining

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

## Coercion

## S4 method for signature 'SummarizedExperiment'
updateObject(object, ..., verbose=FALSE)
## S4 method for signature 'ExpressionSet,SummarizedExperiment'
coerce(from, to = "SummarizedExperiment", strict = TRUE)
## S4 method for signature 'SummarizedExperiment,ExpressionSet'
coerce(from, to = "ExpressionSet", strict = TRUE)
```

Arguments

`assays`	See ?`RangedSummarizedExperiment` in the **SummarizedExperiment** package.
`...`	For `SummarizedExperiment`, see ?`RangedSummarizedExperiment` in the **SummarizedExperiment** package. For `assay`, `...` may contain `withDimnames`, which is forwarded to `assays`. For `cbind, rbind`, `...` contains `SummarizedExperiment` objects to be combined. For other accessors, ignored.
`verbose`	A `logical(1)` indicating whether messages about data coercion during construction should be printed.
`x, object`	An instance of `SummarizedExperiment`-class.
`i, j`	For `assay`, `assay<-`, `i` is a integer or numeric scalar; see 'Details' for additional constraints. For `[,SummarizedExperiment`, `[,SummarizedExperiment<-`, `i`, `j` are instances that can act to subset the underlying `rowRanges`, `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`.
`subset`	An expression which, when evaluated in the context of `rowRanges(x)`, is a logical vector indicating elements or rows to keep: missing values are taken as false.
`select`	An expression which, when evaluated in the context of `colData(x)`, is a logical vector indicating elements or rows to keep: missing values are taken as false.
`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. Setting `withDimnames=FALSE` increases the speed and memory efficiency with which assays are extracted. `withDimnames=TRUE` in the getter `assays<-` allows efficient complex assignments (e.g., updating names of assays, `names(assays(x, withDimnames=FALSE)) = ...` is more efficient than `names(assays(x)) = ...`); it does not influence actual assignment of dimnames to assays.
`drop`	A `logical(1)`, ignored by these methods.
`value`	An instance of a class specified in the S4 method signature or as outlined in 'Details'.
`deparse.level`	See `?base::cbind` for a description of this argument.
`from`	the object to be coerced
`to`	the class to coerce to
`strict`	logical flag. If 'TRUE', the returned object must be strictly from the target class.

Details

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

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

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

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

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

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

`exptData` A SimpleList-class instance containing information about the overall experiment.

`rowData` A GRanges-class instance defining the ranges of interest and associated metadata. WARNING: The accessor for this slot is `rowRanges`, not `rowData`!

`colData` A DataFrame-class instance describing the samples and associated metadata.

`assays` A SimpleList-class instance, each element of which is a matrix summarizing data associated with the corresponding range and sample.

Constructor

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

Coercion

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

`as(from, "SummarizedExperiment")`: Creates a `SummarizedExperiment` object from a `ExpressionSet` object.

`as(from, "ExpressionSet")`: Creates a `ExpressionSet` object from a `SummarizedExperiment` object.

The following data mappings are used for coercion between `ExpressionSet` and `SummarizedExperiment`.

`assayData`	`assays`
`featureData`	`rowData`
`phenoData`	`colData`
`experimentData`, `annotation`, `protocolData`	`colData`

If the `SummarizedExperiment` being coerced uses `GRanges` to store it's range data that data will be included in the `featureData` of the `ExpressionSet`.

Because `ExpressionSet` objects require an assay named 'exprs' if the `SummarizedExperiment` object being coerced does not have an assay named 'exprs' the first assay will be renamed and a warning will be issued.

Accessors

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

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

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

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

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

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

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

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

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

GRanges compatibility (rowRanges access)

Many `GRanges-class` and `GRangesList-class` operations are supported on 'SummarizedExperiment' and derived instances, using `rowRanges`.

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

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

Subsetting

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

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

`subset(x, subset, select)`: Create a subset of `x` using an expression `subset` referring to columns of `rowRanges(x)` (including 'seqnames', 'start', 'end', 'width', 'strand', and `names(mcols(x))`) and / or `select` referring to column names of `colData(x)`.

Additional subsetting accessors provide convenient access to `colData` columns

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

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

Combining

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

`cbind(...)`, `rbind(...)`: `cbind` combines objects with identical ranges (`rowRanges`) but different samples (columns in `assays`). The colnames in `colData` must match or an error is thrown. Duplicate columns of `mcols(rowRanges(SummarizedExperiment))` must contain the same data. Data in `assays` are combined by name matching; if all names are NULL matching is by position. A mixture of names and NULL throws an error.

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

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

Implementation and Extension

This section contains advanced material meant for package developers.

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

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

The current `assays` slot is implemented as a reference class that has copy-on-change semantics. This means that modifying non-assay slots does not copy the (large) assay data, and at the same time the user is not surprised by reference-based semantics. Updates to non-assay slots are very fast; updating the assays slot itself can be 5x or more faster than with an S4 instance in the slot. One useful technique when working with `assay` or `assays` function is use of the `withDimnames=FALSE` argument, which benefits speed and memory use by not copying dimnames from the row- and colData elements to each assay.

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

Author(s)

Martin Morgan, mtmorgan@fhcrc.org

See Also

RangedSummarizedExperiment in the new **SummarizedExperiment** package for the replacement of the SummarizedExperiment class.

Examples

```
## WARNING: The SummarizedExperiment class is deprecated and being
## replaced with the RangedSummarizedExperiment class defined in the
## new SummarizedExperiment package. See ?RangedSummarizedExperiment
## in the SummarizedExperiment package for examples of how to create
## and manipulate RangedSummarizedExperiment objects.
```

`tile-methods` *Tile a GenomicRanges object*

Description

`tile` method for GenomicRanges. Partitions each range into a set of tiles. Tiles are defined in terms of their number or width.

Usage

```
## S4 method for signature 'GenomicRanges'
tile(x, n, width)
```

Arguments

`x`	A GenomicRanges object, like a `GRanges`.
`n`	The number of tiles to generate. See `?tile` in the **IRanges** package for more information about this argument.
`width`	The (maximum) width of each tile. See `?tile` in the **IRanges** package for more information about this argument.

Details

Splits `x` into a `GRangesList`, each element of which corresponds to a tile, or partition, of `x`. Specify the tile geometry with either `n` or `width` (not both). Passing `n` creates `n` tiles of approximately equal width, truncated by sequence end, while passing `width` tiles the region with ranges of the given width, again truncated by sequence end.

Value

A `GRangesList` object, each element of which corresponds to a tile.

Author(s)

M. Lawrence

See Also

`tile` in the **IRanges** package.

Examples

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

# split every range in half
tiles <- tile(gr, n = 2L)
stopifnot(all(elementLengths(tiles) == 2L))

# split ranges into subranges of width 2
# odd width ranges must contain one subrange of width 1
tiles <- tile(gr, width = 2L)
stopifnot(all(all(width(tiles) %in% c(1L, 2L))))
```

`tileGenome` *Put (virtual) tiles on a given genome*

Description

`tileGenome` returns a set of genomic regions that form a partitioning of the specified genome. Each region is called a "tile".

Usage

```
tileGenome(seqlengths, ntile, tilewidth, cut.last.tile.in.chrom=FALSE)
```

Arguments

`seqlengths`	Either a named numeric vector of chromosome lengths or a Seqinfo object. More precisely, if a named numeric vector, it must have a length >= 1, cannot contain NAs or negative values, and cannot have duplicated names. If a Seqinfo object, then it's first replaced with the vector of sequence lengths stored in the object (extracted from the object with the `seqlengths` getter), then the restrictions described previously apply to this vector.
`ntile`	The number of tiles to generate.
`tilewidth`	The desired tile width. The effective tile width might be slightly different but is guaranteed to never be more than the desired width.
`cut.last.tile.in.chrom`	Whether or not to cut the last tile in each chromosome. This is set to `FALSE` by default. Can be set to `TRUE` only when `tilewidth` is specified. In that case, a tile will never overlap with more than 1 chromosome and a GRanges object is returned with one element (i.e. one genomic range) per tile.

Value

If `cut.last.tile.in.chrom` is `FALSE` (the default), a GRangesList object with one list element per tile, each of them containing a number of genomic ranges equal to the number of chromosomes it overlaps with. Note that when the tiles are small (i.e. much smaller than the chromosomes), most of them only overlap with a single chromosome.

If `cut.last.tile.in.chrom` is `TRUE`, a GRanges object with one element (i.e. one genomic range) per tile.

Author(s)

H. Pagès, based on a proposal by M. Morgan

See Also

- genomicvars for an example of how to compute the binned average of a numerical variable defined along a genome.
- GRangesList and GRanges objects.
- Seqinfo objects and the `seqlengths` getter.
- IntegerList objects.
- Views objects.

Examples

```
## ---------------------------------------------------------------------
## A. WITH A TOY GENOME
## ---------------------------------------------------------------------

seqlengths <- c(chr1=60, chr2=20, chr3=25)

## Create 5 tiles:
tiles <- tileGenome(seqlengths, ntile=5)
tiles
elementLengths(tiles)  # tiles 3 and 4 contain 2 ranges

width(tiles)
## Use sum() on this IntegerList object to get the effective tile
## widths:
sum(width(tiles))  # each tile covers exactly 21 genomic positions

## Create 9 tiles:
tiles <- tileGenome(seqlengths, ntile=9)
elementLengths(tiles)  # tiles 6 and 7 contain 2 ranges

table(sum(width(tiles)))  # some tiles cover 12 genomic positions,
                          # others 11

## Specify the tile width:
tiles <- tileGenome(seqlengths, tilewidth=20)
length(tiles)  # 6 tiles
```

```
table(sum(width(tiles)))  # effective tile width is <= specified

## Specify the tile width and cut the last tile in each chromosome:
tiles <- tileGenome(seqlengths, tilewidth=24,
                    cut.last.tile.in.chrom=TRUE)
tiles
width(tiles)  # each tile covers exactly 24 genomic positions, except
              # the last tile in each chromosome

## Partition a genome by chromosome ("natural partitioning"):
tiles <- tileGenome(seqlengths, tilewidth=max(seqlengths),
                    cut.last.tile.in.chrom=TRUE)
tiles  # one tile per chromosome

## sanity check
stopifnot(all.equal(setNames(end(tiles), seqnames(tiles)), seqlengths))

## ---------------------------------------------------------------------
## B. WITH A REAL GENOME
## ---------------------------------------------------------------------

library(BSgenome.Scerevisiae.UCSC.sacCer2)
tiles <- tileGenome(seqinfo(Scerevisiae), ntile=20)
tiles

tiles <- tileGenome(seqinfo(Scerevisiae), tilewidth=50000,
                    cut.last.tile.in.chrom=TRUE)
tiles

## ---------------------------------------------------------------------
## C. AN APPLICATION: COMPUTE THE BINNED AVERAGE OF A NUMERICAL VARIABLE
##    DEFINED ALONG A GENOME
## ---------------------------------------------------------------------

## See '?genomicvars' for an example of how to compute the binned
## average of a numerical variable defined along a genome.
```

Index

∗Topic **classes**
  Constraints, 5
  GNCList-class, 28
∗Topic **manip**
  absoluteRanges, 3
  genomicvars, 21
  makeGRangesFromDataFrame, 49
  phicoef, 58
  tileGenome, 73
∗Topic **methods**
  Constraints, 5
  coverage-methods, 11
  findOverlaps-methods, 14
  GenomicRanges-comparison, 17
  GNCList-class, 28
  intra-range-methods, 46
  range-squeezers, 59
  setops-methods, 60
  strand-utils, 63
  tile-methods, 72
∗Topic **utilities**
  coverage-methods, 11
  findOverlaps-methods, 14
  inter-range-methods, 42
  intra-range-methods, 46
  nearest-methods, 54
  setops-methods, 60
  tile-methods, 72
[,GIntervalTree,ANY-method *(GIntervalTree-class)*, 25
[,GRangesList,ANY-method *(GRangesList-class)*, 38
[,GenomicRanges,ANY-method *(GRanges-class)*, 30
[,List,GenomicRanges-method *(GRanges-class)*, 30
[,SummarizedExperiment,ANY-method *(SummarizedExperiment-class)*, 65
[,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
[,list,GenomicRanges-method *(GRanges-class)*, 30
[<-,GRangesList,ANY,ANY,ANY-method *(GRangesList-class)*, 38
[<-,GenomicRanges,ANY,ANY,ANY-method *(GRanges-class)*, 30
[<-,SummarizedExperiment,ANY,ANY,SummarizedExperiment-meth *(SummarizedExperiment-class)*, 65
[[,SummarizedExperiment,ANY,missing-method *(SummarizedExperiment-class)*, 65
[[<-,GRangesList,ANY,ANY-method *(GRangesList-class)*, 38
[[<-,SummarizedExperiment,ANY,missing,ANY-method *(SummarizedExperiment-class)*, 65
[[<-,SummarizedExperiment,ANY,missing-method *(SummarizedExperiment-class)*, 65
$,GenomicRanges-method *(GRanges-class)*, 30
$,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
$<-,GenomicRanges-method *(GRanges-class)*, 30
$<-,SummarizedExperiment,ANY-method *(SummarizedExperiment-class)*, 65
$<-,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65

absoluteRanges, 3, *35*
as.character,GenomicRanges-method *(GRanges-class)*, 30

as.data.frame,GenomicRanges-method *(GRanges-class)*, 30
as.factor,GenomicRanges-method *(GRanges-class)*, 30
assay *(SummarizedExperiment-class)*, 65
assay,SummarizedExperiment,ANY-method *(SummarizedExperiment-class)*, 65
assay,SummarizedExperiment,character-method *(SummarizedExperiment-class)*, 65
assay,SummarizedExperiment,missing-method *(SummarizedExperiment-class)*, 65
assay,SummarizedExperiment,numeric-method *(SummarizedExperiment-class)*, 65
assay<- *(SummarizedExperiment-class)*, 65
assay<-,SummarizedExperiment,character,matrix-method *(SummarizedExperiment-class)*, 65
assay<-,SummarizedExperiment,missing,matrix-method *(SummarizedExperiment-class)*, 65
assay<-,SummarizedExperiment,numeric,matrix-method *(SummarizedExperiment-class)*, 65
assayNames *(SummarizedExperiment-class)*, 65
assayNames,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
assayNames<- *(SummarizedExperiment-class)*, 65
assayNames<-,SummarizedExperiment,character-method *(SummarizedExperiment-class)*, 65
assays *(SummarizedExperiment-class)*, 65
assays,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
Assays-class *(SummarizedExperiment-class)*, 65
assays<- *(SummarizedExperiment-class)*, 65
assays<-,SummarizedExperiment,list-method *(SummarizedExperiment-class)*, 65
assays<-,SummarizedExperiment,SimpleList-method *(SummarizedExperiment-class)*, 65

bindAsGRanges *(genomicvars)*, 21
binnedAverage *(genomicvars)*, 21

c,GenomicRanges-method *(GRanges-class)*, 30
cbind, *68*
cbind,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
checkConstraint *(Constraints)*, 5
class:Constraint *(Constraints)*, 5
class:ConstraintORNULL *(Constraints)*, 5
class:DelegatingGenomicRanges *(DelegatingGenomicRanges-class)*, 13
class:GenomicRanges *(GRanges-class)*, 30
class:GenomicRangesList *(GenomicRangesList-class)*, 21
class:GIntervalTree *(GIntervalTree-class)*, 25
class:GNCList *(GNCList-class)*, 28
class:GRanges *(GRanges-class)*, 30
class:GRangesList *(GRangesList-class)*, 38
class:SimpleGenomicRangesList *(GenomicRangesList-class)*, 21
coerce,character,GenomicRanges-method *(GRanges-class)*, 30
coerce,character,GRanges-method *(GRanges-class)*, 30
coerce,data.frame,GRanges-method *(makeGRangesFromDataFrame)*, 49
coerce,DataFrame,GRanges-method *(makeGRangesFromDataFrame)*, 49
coerce,ExpressionSet,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
coerce,factor,GenomicRanges-method *(GRanges-class)*, 30
coerce,factor,GRanges-method *(GRanges-class)*, 30

coerce,GenomicRanges,GNCList-method *(GNCList-class)*, 28
coerce,GenomicRanges,RangedData-method *(GRanges-class)*, 30
coerce,GenomicRanges,RangesList-method *(GRanges-class)*, 30
coerce,GenomicRangesList,RangedDataList-method *(GenomicRangesList-class)*, 21
coerce,GIntervalTree,GRanges-method *(GIntervalTree-class)*, 25
coerce,GNCList,GRanges-method *(GNCList-class)*, 28
coerce,GRanges,GIntervalTree-method *(GIntervalTree-class)*, 25
coerce,GRanges,GRangesList-method *(GRangesList-class)*, 38
coerce,GRangesList,CompressedIRangesList-method *(GRangesList-class)*, 38
coerce,GRangesList,IRangesList-method *(GRangesList-class)*, 38
coerce,GRangesList,RangesList-method *(GRangesList-class)*, 38
coerce,RangedData,GRanges-method *(GRanges-class)*, 30
coerce,RangedDataList,GenomicRangesList-method *(GenomicRangesList-class)*, 21
coerce,RangedDataList,GRangesList-method *(GRangesList-class)*, 38
coerce,RangesList,GRanges-method *(GRanges-class)*, 30
coerce,RleList,GRanges-method *(genomicvars)*, 21
coerce,RleViewsList,GRanges-method *(genomicvars)*, 21
coerce,Seqinfo,GenomicRanges-method *(GRanges-class)*, 30
coerce,Seqinfo,GRanges-method *(GRanges-class)*, 30
coerce,Seqinfo,RangesList-method *(GRanges-class)*, 30
coerce,SummarizedExperiment,ExpressionSet-method *(SummarizedExperiment-class)*, 65
coerce,SummarizedExperiment,RangedSummarizedExperiment-method *(SummarizedExperiment-class)*, 65
colData *(SummarizedExperiment-class)*, 65
colData,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
colData<- *(SummarizedExperiment-class)*, 65
colData<-,SummarizedExperiment,DataFrame-method *(SummarizedExperiment-class)*, 65
compare, *70*
compare *(GenomicRanges-comparison)*, 17
compare,ANY,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
compare,GenomicRanges,GenomicRanges-method *(GenomicRanges-comparison)*, 17
compare,SummarizedExperiment,ANY-method *(SummarizedExperiment-class)*, 65
compare,SummarizedExperiment,SummarizedExperiment-method *(SummarizedExperiment-class)*, 65
Constraint *(Constraints)*, 5
constraint *(Constraints)*, 5
Constraint-class *(Constraints)*, 5
constraint<- *(Constraints)*, 5
ConstraintORNULL *(Constraints)*, 5
ConstraintORNULL-class *(Constraints)*, 5
Constraints, 5
countOverlaps, *70*
countOverlaps *(findOverlaps-methods)*, 14
countOverlaps,GenomicRanges,GenomicRanges-method *(findOverlaps-methods)*, 14
countOverlaps,GenomicRanges,GIntervalTree-method *(findOverlaps-methods)*, 14
countOverlaps,GenomicRanges,GNCList-method *(findOverlaps-methods)*, 14
countOverlaps,GenomicRanges,Vector-method *(findOverlaps-methods)*, 14
countOverlaps,GNCList,GenomicRanges-method *(findOverlaps-methods)*, 14
countOverlaps,GRanges,GRangesList-method *(findOverlaps-methods)*, 14
countOverlaps,GRangesList,GRanges-method *(findOverlaps-methods)*, 14
countOverlaps,GRangesList,GRangesList-method *(findOverlaps-methods)*, 14
countOverlaps,GRangesList,Vector-method *(findOverlaps-methods)*, 14
countOverlaps,SummarizedExperiment,SummarizedExperiment-me

*(*SummarizedExperiment-class*)*, 65
countOverlaps,SummarizedExperiment,Vector-method *(*SummarizedExperiment-class*)*, 65
countOverlaps,Vector,GenomicRanges-method *(*findOverlaps-methods*)*, 14
countOverlaps,Vector,GRangesList-method *(*findOverlaps-methods*)*, 14
countOverlaps,Vector,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
coverage, *11–13*, *70*
coverage *(*coverage-methods*)*, 11
coverage,GenomicRanges-method, *23*
coverage,GenomicRanges-method *(*coverage-methods*)*, 11
coverage,GRangesList-method *(*coverage-methods*)*, 11
coverage,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
coverage-methods, 11, *13*, *35*, *41*

DataFrame, *26*, *30*, *32, 33*, *35*, *38*, *49–51*, *65*, *68, 69*
DataFrameList-class, *41*
DataTable, *26*, *32*
DelegatingGenomicRanges-class, 13
dim,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
dimnames,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
dimnames<-,SummarizedExperiment,list-method *(*SummarizedExperiment-class*)*, 65
dimnames<-,SummarizedExperiment,NULL-method *(*SummarizedExperiment-class*)*, 65
disjoin *(*inter-range-methods*)*, 42
disjoin,GenomicRanges-method *(*inter-range-methods*)*, 42
disjoin,GRangesList-method *(*inter-range-methods*)*, 42
disjointBins, *70*
disjointBins *(*inter-range-methods*)*, 42
disjointBins,GenomicRanges-method *(*inter-range-methods*)*, 42
disjointBins,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
distance, *70*
distance *(*nearest-methods*)*, 54
distance,ANY,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
distance,GenomicRanges,GenomicRanges-method *(*nearest-methods*)*, 54
distance,SummarizedExperiment,ANY-method *(*SummarizedExperiment-class*)*, 65
distance,SummarizedExperiment,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
distanceToNearest, *70*
distanceToNearest *(*nearest-methods*)*, 54
distanceToNearest,ANY,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
distanceToNearest,GenomicRanges,GenomicRanges-method *(*nearest-methods*)*, 54
distanceToNearest,GenomicRanges,missing-method *(*nearest-methods*)*, 54
distanceToNearest,SummarizedExperiment,ANY-method *(*SummarizedExperiment-class*)*, 65
distanceToNearest,SummarizedExperiment,SummarizedExperimen *(*SummarizedExperiment-class*)*, 65
DNAStringSet, *35*
duplicated, *70*
duplicated,GenomicRanges-method *(*GenomicRanges-comparison*)*, 17
duplicated,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
duplicated.GenomicRanges *(*GenomicRanges-comparison*)*, 17

elementMetadata,GIntervalTree-method *(*GIntervalTree-class*)*, 25
elementMetadata,GRangesList-method *(*GRangesList-class*)*, 38
elementMetadata,SummarizedExperiment-method *(*SummarizedExperiment-class*)*,

65
elementMetadata<-,GRangesList-method (GRangesList-class), 38
elementMetadata<-,SummarizedExperiment-method (SummarizedExperiment-class), 65
end, *70*
end,GenomicRanges-method (GRanges-class), 30
end,GIntervalTree-method (GIntervalTree-class), 25
end,GNCList-method (GNCList-class), 28
end,GRangesList-method (GRangesList-class), 38
end,SummarizedExperiment-method (SummarizedExperiment-class), 65
end<-,GenomicRanges-method (GRanges-class), 30
end<-,GRangesList-method (GRangesList-class), 38
end<-,SummarizedExperiment-method (SummarizedExperiment-class), 65
exptData (SummarizedExperiment-class), 65
exptData,SummarizedExperiment-method (SummarizedExperiment-class), 65
exptData<- (SummarizedExperiment-class), 65
exptData<-,SummarizedExperiment,list-method (SummarizedExperiment-class), 65
exptData<-,SummarizedExperiment,SimpleList-method (SummarizedExperiment-class), 65

findOverlaps, *14*, *15*, *26*, *29*, *55*, *70*
findOverlaps (findOverlaps-methods), 14
findOverlaps,GenomicRanges,GenomicRanges-method (findOverlaps-methods), 14
findOverlaps,GenomicRanges,GIntervalTree-method (findOverlaps-methods), 14
findOverlaps,GenomicRanges,GNCList-method (findOverlaps-methods), 14
findOverlaps,GenomicRanges,GRangesList-method (findOverlaps-methods), 14
findOverlaps,GenomicRanges,RangedData-method (findOverlaps-methods), 14
findOverlaps,GenomicRanges,RangesList-method (findOverlaps-methods), 14
findOverlaps,GNCList,GenomicRanges-method (findOverlaps-methods), 14
findOverlaps,GRangesList,GenomicRanges-method (findOverlaps-methods), 14
findOverlaps,GRangesList,GRangesList-method (findOverlaps-methods), 14
findOverlaps,GRangesList,RangedData-method (findOverlaps-methods), 14
findOverlaps,GRangesList,RangesList-method (findOverlaps-methods), 14
findOverlaps,RangedData,GenomicRanges-method (findOverlaps-methods), 14
findOverlaps,RangedData,GRangesList-method (findOverlaps-methods), 14
findOverlaps,RangesList,GenomicRanges-method (findOverlaps-methods), 14
findOverlaps,RangesList,GRangesList-method (findOverlaps-methods), 14
findOverlaps,SummarizedExperiment,SummarizedExperiment-met (SummarizedExperiment-class), 65
findOverlaps,SummarizedExperiment,Vector-method (SummarizedExperiment-class), 65
findOverlaps,Vector,SummarizedExperiment-method (SummarizedExperiment-class), 65
findOverlaps-methods, 14, *19*, *27*, *35*, *41*, *56*, *62*
flank, *70*
flank (intra-range-methods), 46
flank,GenomicRanges-method (intra-range-methods), 46
flank,GRangesList-method (intra-range-methods), 46
flank,SummarizedExperiment-method (SummarizedExperiment-class), 65
follow, *70*
follow (nearest-methods), 54
follow,ANY,SummarizedExperiment-method (SummarizedExperiment-class), 65
follow,GenomicRanges,GenomicRanges-method

(nearest-methods), 54
follow,GenomicRanges,missing-method (nearest-methods), 54
follow,SummarizedExperiment,ANY-method (SummarizedExperiment-class), 65
follow,SummarizedExperiment,SummarizedExperiment-method (SummarizedExperiment-class), 65

GAlignmentPairs, *11*, *14*, *35*, *59, 60*
GAlignments, *11*, *14*, *35*, *46*, *53*, *59, 60*
GAlignmentsList, *14*, *46*, *59, 60*
gaps, *43*
gaps (inter-range-methods), 42
gaps,GenomicRanges-method (inter-range-methods), 42
GenomicRanges, *3*, *6*, *14, 15*, *17–19*, *21*, *23*, *26*, *28*, *42, 43*, *46–48*, *53–56*, *72*
GenomicRanges (GRanges-class), 30
GenomicRanges-class, *7*
GenomicRanges-class (GRanges-class), 30
GenomicRanges-comparison, 17, *35*, *44*
GenomicRangesList (GenomicRangesList-class), 21
GenomicRangesList-class, 21
GenomicRangesORGRangesList-class (GRanges-class), 30
GenomicRangesORmissing-class (GRanges-class), 30
genomicvariables (genomicvars), 21
genomicvars, *4*, 21, *35*, *74*
getTable, *51*
GIntervalTree, *28*
GIntervalTree (GIntervalTree-class), 25
GIntervalTree-class, 25
GNCList, *15*, *25*
GNCList (GNCList-class), 28
GNCList-class, 28
GRanges, *3, 4*, *11–15*, *22, 23*, *26*, *28, 29*, *42*, *43*, *46*, *48–50*, *53*, *56*, *59–62*, *65*, *68–70*, *73, 74*
GRanges (GRanges-class), 30
granges, *70*
granges (range-squeezers), 59
granges,GenomicRanges-method (GRanges-class), 30
granges,GNCList-method (GNCList-class), 28
granges,SummarizedExperiment-method (SummarizedExperiment-class), 65
GRanges-class, 30, *41*, *62*
GRangesList, *11–15*, *21*, *33*, *35*, *46–48*, *51*, *53*, *59–62*, *65*, *68*, *70*, *74*
GRangesList (GRangesList-class), 38
GRangesList-class, 38, *62*
grglist (range-squeezers), 59

Hits, *15*, *55, 56*

IntegerList, *74*
inter-range-methods, *19*, *35*, *41*, 42, *44*
intersect,GRanges,GRanges-method (setops-methods), 60
IntervalForest, *26, 27*
IntervalTree, *27*
intra-range-methods, *19*, *35*, *41*, 46, *48*
IRanges, *3, 4*, *26*, *30, 31*, *35*
IRangesList, *39*
is, *7*
is.unsorted,GenomicRanges-method (GenomicRanges-comparison), 17
isDisjoint, *70*
isDisjoint (inter-range-methods), 42
isDisjoint,GenomicRanges-method, *23*
isDisjoint,GenomicRanges-method (inter-range-methods), 42
isDisjoint,GRangesList-method (inter-range-methods), 42
isDisjoint,SummarizedExperiment-method (SummarizedExperiment-class), 65
isSmallGenome (absoluteRanges), 3

lapply, *40*
length,GenomicRanges-method (GRanges-class), 30
length,GNCList-method (GNCList-class), 28
List, *21*, *34*

makeGRangesFromDataFrame, *31*, *35*, 49
makeGRangesListFromFeatureFragments, *51*
makeGRangesListFromFeatureFragments (GRangesList-class), 38
makeSummarizedExperimentFromExpressionSet, 52

mapCoords (mapCoords-methods), 52
mapCoords,GenomicRanges,GenomicRanges-method (mapCoords-methods), 52
mapCoords,GenomicRanges,GRangesList-method (mapCoords-methods), 52
mapCoords-methods, 52, *54*
mapply, *40*
mapToAlignments, *53*
mapToTranscripts, *53*
match, *70*
match,GenomicRanges,GenomicRanges-method (GenomicRanges-comparison), 17
mcolAsRleList (genomicvars), 21
mcols, *68*, *70*
mcols,SummarizedExperiment-method (SummarizedExperiment-class), 65
mcols<-,SummarizedExperiment-method (SummarizedExperiment-class), 65

names,GenomicRanges-method (GRanges-class), 30
names,GIntervalTree-method (GIntervalTree-class), 25
names,GNCList-method (GNCList-class), 28
names<-,GenomicRanges-method (GRanges-class), 30
narrow, *70*
narrow (intra-range-methods), 46
narrow,GenomicRanges-method (intra-range-methods), 46
narrow,GRangesList-method (intra-range-methods), 46
narrow,SummarizedExperiment-method (SummarizedExperiment-class), 65
NCList, *28*, *29*
NCLists, *28*, *29*
nearest, *70*
nearest (nearest-methods), 54
nearest,ANY,SummarizedExperiment-method (SummarizedExperiment-class), 65
nearest,GenomicRanges,GenomicRanges-method (nearest-methods), 54
nearest,GenomicRanges,missing-method (nearest-methods), 54
nearest,SummarizedExperiment,ANY-method (SummarizedExperiment-class), 65
nearest,SummarizedExperiment,missing-method (SummarizedExperiment-class), 65
nearest,SummarizedExperiment,SummarizedExperiment-method (SummarizedExperiment-class), 65
nearest-methods, *35*, 54, *56*

Ops,GenomicRanges,numeric-method (intra-range-methods), 46
order, *70*
order,GenomicRanges-method (GenomicRanges-comparison), 17
order,SummarizedExperiment-method (SummarizedExperiment-class), 65
overlapsAny, *70*
overlapsAny (findOverlaps-methods), 14
overlapsAny,GenomicRanges,GenomicRanges-method (findOverlaps-methods), 14
overlapsAny,GenomicRanges,GRangesList-method (findOverlaps-methods), 14
overlapsAny,GenomicRanges,RangedData-method (findOverlaps-methods), 14
overlapsAny,GenomicRanges,RangesList-method (findOverlaps-methods), 14
overlapsAny,GRangesList,GenomicRanges-method (findOverlaps-methods), 14
overlapsAny,GRangesList,GRangesList-method (findOverlaps-methods), 14
overlapsAny,GRangesList,RangedData-method (findOverlaps-methods), 14
overlapsAny,GRangesList,RangesList-method (findOverlaps-methods), 14
overlapsAny,RangedData,GenomicRanges-method (findOverlaps-methods), 14
overlapsAny,RangedData,GRangesList-method (findOverlaps-methods), 14
overlapsAny,RangesList,GenomicRanges-method (findOverlaps-methods), 14
overlapsAny,RangesList,GRangesList-method (findOverlaps-methods), 14
overlapsAny,SummarizedExperiment,SummarizedExperiment-meth (SummarizedExperiment-class), 65

overlapsAny,SummarizedExperiment,Vector-method (SummarizedExperiment-class), 65
overlapsAny,Vector,SummarizedExperiment-method (SummarizedExperiment-class), 65

pgap, *60*
pgap (setops-methods), 60
pgap,GRanges,GRanges-method (setops-methods), 60
phicoef, 58
pintersect, *60*
pintersect (setops-methods), 60
pintersect,GRanges,GRanges-method (setops-methods), 60
pintersect,GRanges,GRangesList-method (setops-methods), 60
pintersect,GRangesList,GRanges-method (setops-methods), 60
pintersect,GRangesList,GRangesList-method (setops-methods), 60
pmapCoords (mapCoords-methods), 52
pmapCoords,GenomicRanges,GRangesList-method (mapCoords-methods), 52
precede, *70*
precede (nearest-methods), 54
precede,ANY,SummarizedExperiment-method (SummarizedExperiment-class), 65
precede,GenomicRanges,GenomicRanges-method (nearest-methods), 54
precede,GenomicRanges,missing-method (nearest-methods), 54
precede,SummarizedExperiment,ANY-method (SummarizedExperiment-class), 65
precede,SummarizedExperiment,SummarizedExperiment-method (SummarizedExperiment-class), 65
promoters (intra-range-methods), 46
promoters,GenomicRanges-method (intra-range-methods), 46
promoters,GRangesList-method (intra-range-methods), 46
psetdiff, *60*
psetdiff (setops-methods), 60
psetdiff,GRanges,GRanges-method (setops-methods), 60
psetdiff,GRanges,GRangesList-method (setops-methods), 60
psetdiff,GRangesList,GRangesList-method (setops-methods), 60
punion, *60*
punion (setops-methods), 60
punion,GRanges,GRanges-method (setops-methods), 60
punion,GRanges,GRangesList-method (setops-methods), 60
punion,GRangesList,GRanges-method (setops-methods), 60

range (inter-range-methods), 42
range,GenomicRanges-method (inter-range-methods), 42
range,GRangesList-method (inter-range-methods), 42
range-squeezers, 59
RangedData, *14*
RangedDataList, *39*
RangedSummarizedExperiment, *52*, *65*, *67*, *71*
Ranges, *3*, *11*, *14*, *26*, *28*, *35*, *43*, *47*, *48*, *56*, *60*
ranges, *70*
ranges,DelegatingGenomicRanges-method (DelegatingGenomicRanges-class), 13
ranges,GIntervalTree-method (GIntervalTree-class), 25
ranges,GNCList-method (GNCList-class), 28
ranges,GRanges-method (GRanges-class), 30
ranges,GRangesList-method (GRangesList-class), 38
ranges,SummarizedExperiment-method (SummarizedExperiment-class), 65
Ranges-comparison, *19*
ranges<-,GenomicRanges-method (GRanges-class), 30
ranges<-,GRangesList-method (GRangesList-class), 38
ranges<-,SummarizedExperiment-method (SummarizedExperiment-class), 65
RangesList, *11*, *14*, *28*, *59*, *60*
RangesList-class, *41*

rank, *70*
rank,GenomicRanges-method (GenomicRanges-comparison), 17
rank,SummarizedExperiment-method (SummarizedExperiment-class), 65
rankSeqlevels, *50*
rbind,SummarizedExperiment-method (SummarizedExperiment-class), 65
reduce, *43*
reduce *(*inter-range-methods*)*, 42
reduce,GenomicRanges-method (inter-range-methods), 42
reduce,GRangesList-method (inter-range-methods), 42
relativeRanges *(*absoluteRanges*)*, 3
relistToClass,GRanges-method (GRangesList-class), 38
resize, *70*
resize *(*intra-range-methods*)*, 46
resize,GenomicRanges-method (intra-range-methods), 46
resize,GRangesList-method (intra-range-methods), 46
resize,SummarizedExperiment-method (SummarizedExperiment-class), 65
restrict, *70*
restrict *(*intra-range-methods*)*, 46
restrict,GenomicRanges-method (intra-range-methods), 46
restrict,GRangesList-method (intra-range-methods), 46
restrict,SummarizedExperiment-method (SummarizedExperiment-class), 65
rglist *(*range-squeezers*)*, 59
Rle, *22*, *30–32*, *35*, *64*
RleList, *12, 13*, *21–23*
RleList-class, *41*
rowData *(*SummarizedExperiment-class*)*, 65
rowData<- *(*SummarizedExperiment-class*)*, 65
rowRanges *(*SummarizedExperiment-class*)*, 65
rowRanges,SummarizedExperiment-method (SummarizedExperiment-class), 65
rowRanges<- (SummarizedExperiment-class), 65
rowRanges<-,SummarizedExperiment,GenomicRanges-method (SummarizedExperiment-class), 65
rowRanges<-,SummarizedExperiment,GRangesList-method (SummarizedExperiment-class), 65

sapply, *40*
score,GenomicRanges-method (GRanges-class), 30
score,GIntervalTree-method (GIntervalTree-class), 25
score,GRangesList-method (GRangesList-class), 38
score<-,GenomicRanges-method (GRanges-class), 30
score<-,GRangesList-method (GRangesList-class), 38
Seqinfo, *4*, *27*, *30–32*, *39*, *49*, *50*, *73, 74*
seqinfo, *27*, *35*, *41*, *70*
seqinfo,DelegatingGenomicRanges-method (DelegatingGenomicRanges-class), 13
seqinfo,GIntervalTree-method (GIntervalTree-class), 25
seqinfo,GNCList-method *(*GNCList-class*)*, 28
seqinfo,GRanges-method *(*GRanges-class*)*, 30
seqinfo,GRangesList-method (GRangesList-class), 38
seqinfo,SummarizedExperiment-method (SummarizedExperiment-class), 65
seqinfo<-,GenomicRanges-method (GRanges-class), 30
seqinfo<-,GRangesList-method (GRangesList-class), 38
seqinfo<-,SummarizedExperiment-method (SummarizedExperiment-class), 65
seqlengths, *3*, *4*, *73, 74*
seqlevels, *32*, *39*
seqlevelsStyle, *27*, *33*, *39*
seqnames, *70*

seqnames,DelegatingGenomicRanges-method (DelegatingGenomicRanges-class), 13
seqnames,GIntervalTree-method (GIntervalTree-class), 25
seqnames,GNCList-method (GNCList-class), 28
seqnames,GRanges-method (GRanges-class), 30
seqnames,GRangesList-method (GRangesList-class), 38
seqnames,SummarizedExperiment-method (SummarizedExperiment-class), 65
seqnames<-,GenomicRanges-method (GRanges-class), 30
seqnames<-,GRangesList-method (GRangesList-class), 38
setClass, *7*
setdiff,GRanges,GRanges-method (setops-methods), 60
setMethod, *7*
setops-methods, *19*, *35*, *41*, 60, *62*
ShallowData-class (SummarizedExperiment-class), 65
ShallowSimpleListAssays-class (SummarizedExperiment-class), 65
shift, *70*
shift (intra-range-methods), 46
shift,GenomicRanges-method (intra-range-methods), 46
shift,GRangesList-method (intra-range-methods), 46
shift,SummarizedExperiment-method (SummarizedExperiment-class), 65
show,GenomicRanges-method (GRanges-class), 30
show,GIntervalTree-method (GIntervalTree-class), 25
show,GRangesList-method (GRangesList-class), 38
show,SummarizedExperiment-method (SummarizedExperiment-class), 65
showMethods, *7*
SimpleGenomicRangesList-class (GenomicRangesList-class), 21
SimpleList, *68*, *69*
sort, *70*
sort,GenomicRanges-method (GenomicRanges-comparison), 17
sort,SummarizedExperiment-method (SummarizedExperiment-class), 65
sort.GenomicRanges (GenomicRanges-comparison), 17
split,SummarizedExperiment,ANY-method (SummarizedExperiment-class), 65
split,SummarizedExperiment-method (SummarizedExperiment-class), 65
start, *70*
start,GenomicRanges-method (GRanges-class), 30
start,GIntervalTree-method (GIntervalTree-class), 25
start,GNCList-method (GNCList-class), 28
start,GRangesList-method (GRangesList-class), 38
start,SummarizedExperiment-method (SummarizedExperiment-class), 65
start<-,GenomicRanges-method (GRanges-class), 30
start<-,GRangesList-method (GRangesList-class), 38
start<-,SummarizedExperiment-method (SummarizedExperiment-class), 65
strand, *26*, *30*, *64*, *70*
strand,character-method (strand-utils), 63
strand,DataTable-method (strand-utils), 63
strand,DelegatingGenomicRanges-method (DelegatingGenomicRanges-class), 13
strand,factor-method (strand-utils), 63
strand,GIntervalTree-method (GIntervalTree-class), 25
strand,GNCList-method (GNCList-class), 28

strand,GRanges-method *(*GRanges-class*)*, 30
strand,GRangesList-method *(*GRangesList-class*)*, 38
strand,integer-method *(*strand-utils*)*, 63
strand,logical-method *(*strand-utils*)*, 63
strand,missing-method *(*strand-utils*)*, 63
strand,NULL-method *(*strand-utils*)*, 63
strand,Rle-method *(*strand-utils*)*, 63
strand,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
strand-utils, 63
strand<-,DataTable,ANY-method *(*strand-utils*)*, 63
strand<-,GenomicRanges,ANY-method *(*GRanges-class*)*, 30
strand<-,GRangesList,ANY-method *(*GRangesList-class*)*, 38
strand<-,GRangesList,character-method *(*GRangesList-class*)*, 38
strand<-,SummarizedExperiment,ANY-method *(*SummarizedExperiment-class*)*, 65
subset,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
subsetByOverlaps, *70*
subsetByOverlaps *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GenomicRanges,GenomicRanges-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GenomicRanges,GRangesList-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GenomicRanges,RangedData-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GenomicRanges,RangesList-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GRangesList,GenomicRanges-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GRangesList,GRangesList-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GRangesList,RangedData-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,GRangesList,RangesList-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,RangedData,GenomicRanges-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,RangedData,GRangesList-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,RangesList,GenomicRanges-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,RangesList,GRangesList-method *(*findOverlaps-methods*)*, 14
subsetByOverlaps,SummarizedExperiment,SummarizedExperiment *(*SummarizedExperiment-class*)*, 65
subsetByOverlaps,SummarizedExperiment,Vector-method *(*SummarizedExperiment-class*)*, 65
subsetByOverlaps,Vector,SummarizedExperiment-method *(*SummarizedExperiment-class*)*, 65
SummarizedExperiment, *52*, *59*, *60*
SummarizedExperiment *(*SummarizedExperiment-class*)*, 65
SummarizedExperiment,list-method *(*SummarizedExperiment-class*)*, 65
SummarizedExperiment,matrix-method *(*SummarizedExperiment-class*)*, 65
SummarizedExperiment,missing-method *(*SummarizedExperiment-class*)*, 65
SummarizedExperiment,SimpleList-method *(*SummarizedExperiment-class*)*, 65
SummarizedExperiment-class, 65

tile, *72*
tile,GenomicRanges-method *(*tile-methods*)*, 72
tile-methods, 72
tileGenome, *4*, *22*, *23*, *35*, 73
trim,GenomicRanges-method *(*intra-range-methods*)*, 46

union,GRanges,GRanges-method *(*setops-methods*)*, 60
update,DelegatingGenomicRanges-method *(*DelegatingGenomicRanges-class*)*, 13
updateObject,GRanges-method *(*GRanges-class*)*, 30

updateObject,GRangesList-method
(GRangesList-class), 38
updateObject,SummarizedExperiment-method
(SummarizedExperiment-class),
65

validObject, *7*
values,SummarizedExperiment-method
(SummarizedExperiment-class),
65
values<-,SummarizedExperiment-method
(SummarizedExperiment-class),
65
Vector, *35*
Vector-class, *41*
Views, *74*

width, *70*
width,GenomicRanges-method
(GRanges-class), 30
width,GIntervalTree-method
(GIntervalTree-class), 25
width,GNCList-method (GNCList-class), 28
width,GRangesList-method
(GRangesList-class), 38
width,SummarizedExperiment-method
(SummarizedExperiment-class),
65
width<-,GenomicRanges-method
(GRanges-class), 30
width<-,GRangesList-method
(GRangesList-class), 38
width<-,SummarizedExperiment-method
(SummarizedExperiment-class),
65
window,GenomicRanges-method
(GRanges-class), 30