Package ‘cigarillo’

October 30, 2025

Title Efficient manipulation of CIGAR strings

Description CIGAR stands for Concise Idiosyncratic Gapped Alignment Report. CIGAR strings are found in the BAM files produced by most aligners and in the AIRR-formatted output produced by IgBLAST. The cigarillo package provides functions to parse and inspect CIGAR strings, trim them, turn them into ranges of positions relative to the ``query space'' or ``reference space'', and project positions or sequences from one space to the other. Note that these operations are low-level operations that the user rarely needs to perform directly. More typically, they are performed behind the scene by higher-level functionality implemented in other packages like Bioconductor packages GenomicAlignments and igblastr.

biocViews Infrastructure, Alignment, SequenceMatching, Sequencing

URL `https://bioconductor.org/packages/cigarillo`

BugReports `https://github.com/Bioconductor/cigarillo/issues`

Version 1.0.0

License Artistic-2.0

Encoding UTF-8

Depends methods, BiocGenerics, S4Vectors (>= 0.47.2), IRanges, Biostrings

Imports stats

LinkingTo S4Vectors, IRanges

Suggests Rsamtools, GenomicAlignments, RNAseqData.HNRNPC.bam.chr14, BSgenome.Hsapiens.UCSC.hg19, testthat, knitr, rmarkdown, BiocStyle

VignetteBuilder knitr

Collate utils.R cigar_ops_visibility.R explode_cigars.R tabulate_cigar_ops.R cigar_extent.R trim_cigars.R cigars_as_ranges.R project_positions.R project_sequences.R map_ref_ranges_to_query.R

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

git_branch RELEASE_3_22

git_last_commit 8775adf

git_last_commit_date 2025-10-29

Repository Bioconductor 3.22

Date/Publication 2025-10-30

Author Hervé Pagès [aut, cre] (ORCID: <https://orcid.org/0009-0002-8272-4522>),
Valerie Obenchain [aut],
Michael Lawrence [aut],
Patrick Aboyoun [ctb],
Fedor Bezrukov [ctb],
Martin Morgan [ctb]

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

Contents

`cigarillo-package` *Efficient manipulation of CIGAR strings*

Description

CIGAR stands for Concise Idiosyncratic Gapped Alignment Report. CIGAR strings are found in the BAM files produced by most aligners and in the AIRR-formatted output produced by IgBLAST.

The **cigarillo** package provides functions to parse and inspect CIGAR strings, trim them, turn them into ranges of positions relative to the "query space" or "reference space", and project positions or sequences from one space to the other. Note that these operations are low-level operations that the user rarely needs to perform directly. More typically, they are performed behind the scene by higher-level functionality implemented in other packages like Bioconductor packages **GenomicAlignments** and **igblastr**.

Details

For an overview of the functionality provided by the package, please refer to the vignette:

```
vignette("cigarillo", package="cigarillo")
```

Author(s)

Hervé Pagès, Valerie Obenchain, Michael Lawrence

With contributions from Martin Morgan, Patrick Aboyoun, and Fedor Bezrukov

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_positions` to project positions from query to reference space and vice versa.
- `project_sequences` to project sequences from one space to the other.
- The RleList class in the **IRanges** package.

`cigars_as_ranges` *Turn CIGAR strings into ranges of positions*

Description

Turn CIGAR strings into ranges of positions relative to the "query space", "reference space", or "pairwise alignment space".

Usage

```
cigars_as_ranges_along_ref(cigars,
          N.regions.removed=FALSE,
          flags=NULL, lmmpos=1L, f=NULL,
          ops=CIGAR_OPS, drop.empty.ranges=FALSE, reduce.ranges=FALSE,
          with.ops=FALSE, with.oplens=FALSE)

cigars_as_ranges_along_query(cigars,
          before.hard.clipping=FALSE, after.soft.clipping=FALSE,
          flags=NULL,
          ops=CIGAR_OPS, drop.empty.ranges=FALSE, reduce.ranges=FALSE,
          with.ops=FALSE, with.oplens=FALSE)

cigars_as_ranges_along_pwa(cigars,
          N.regions.removed=FALSE, dense=FALSE,
          flags=NULL,
          ops=CIGAR_OPS, drop.empty.ranges=FALSE, reduce.ranges=FALSE,
          with.ops=FALSE, with.oplens=FALSE)
```

Arguments

`cigars`	A character vector (or factor) containing CIGAR strings.
`N.regions.removed`	`TRUE` or `FALSE`. If `TRUE`, then `cigars_as_ranges_along_ref` reports ranges with respect to the "reference space" from which the N regions have been removed, and `cigars_as_ranges_along_pwa` reports them with respect to the "pairwise alignment space" from which the N regions have been removed.
`flags`	`NULL` or an integer vector parallel to `cigars` that contains the SAM/BAM flags corresponding to each CIGAR string. According to the SAM Spec v1.4, flag bit 0x4 is the only reliable place to tell whether a segment (or read) is mapped (bit is 0) or not (bit is 1). If the `flags` argument is supplied, then `cigars_as_ranges_along_ref`, `cigars_as_ranges_along_query`, and `cigars_as_ranges_along_pwa` don't produce any range for unmapped reads i.e. they treat them as if their CIGAR was empty (independently of what their CIGAR is).
`lmmpos`	An integer vector containing the 1-based leftmost mapping POSition of each alignment with respect to the "reference space". These are the 1-based leftmost positions/coordinates of each (eventually clipped) query sequence with respect to the subject. `lmmpos` must be a single integer, or an integer vector parallel to `cigars`.
`f`	`NULL` or a factor parallel to `cigars`. If `NULL` (the default), then the ranges are grouped by alignment i.e. the returned IRangesList object has 1 list element per element in `cigars`. Otherwise they are grouped by factor level i.e. the returned IRangesList object has 1 list element per level in `f` and is named with those levels. For example, if `f` is a factor containing the chromosome for each read, then the returned IRangesList object will have 1 list element per chromosome and each list element will contain all the ranges on that chromosome.
`ops`	Character vector where the elements are single letters representing valid CIGAR operations. Must be a subset of `CIGAR_OPS`. See ?`CIGAR_OPS` for more information. Only the operations listed in ops will be turned into ranges.
`drop.empty.ranges`	`TRUE` or `FALSE`. Should empty ranges be dropped?
`reduce.ranges`	`TRUE` or `FALSE`. Should adjacent ranges coming from the same cigar be merged or not? Using `TRUE` can significantly reduce the size of the returned object.
`with.ops`	`TRUE` or `FALSE`. Should the returned ranges be named/labeled with their corresponding CIGAR operation? Only supported when `f` is `NULL`.
`with.oplens`	`TRUE` or `FALSE`. If `with.oplens` is `TRUE`, then the returned IRangesList object will carry the lengths of the CIGAR operations in an inner metadata column named `oplen`. Only supported when `f` is `NULL`.

`before.hard.clipping`	`TRUE` or `FALSE`. If `TRUE`, then `cigars_as_ranges_along_query` reports ranges with respect to the "query space" to which the H regions have been added. Note that `before.hard.clipping` and `after.soft.clipping` cannot both be `TRUE`.
`after.soft.clipping`	`TRUE` or `FALSE`. If `TRUE`, then `cigars_as_ranges_along_query` reports ranges with respect to the "query space" from which the S regions have been removed. Note that `before.hard.clipping` and `after.soft.clipping` cannot both be `TRUE`.
`dense`	`TRUE` or `FALSE`. If `TRUE`, then `cigars_as_ranges_along_pwa` reports ranges with respect to the "pairwise alignment space" from which the I, D, and N regions have been removed. Note that `N.regions.removed` and `dense` cannot both be `TRUE`.

Value

An IRangesList object (more precisely a CompressedIRangesList object) with one list element per element in `cigars`.

However, if `f` is a factor, then the returned IRangesList object returned by `cigars_as_ranges_along_ref()` is a SimpleIRangesList object (instead of CompressedIRangesList). In that case it has one list element per level in `f`, and is named with those levels.

Author(s)

Hervé Pagès

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- `project_positions` to project positions from query to reference space and vice versa.
- `project_sequences` to project sequences from one space to the other.
- The IRanges and IRangesList classes in the **IRanges** package.

Examples

```
cigar1 <- "3H15M55N4M2I6M2D5M6S"
my_cigars <- c("40M2I9M", cigar1, "2S10M2000N15M", "3H33M5H")

## ---------------------------------------------------------------------
## Turn CIGAR strings into ranges along the "reference space"
## ---------------------------------------------------------------------
```

```
cigars_as_ranges_along_ref(cigar1, with.ops=TRUE, with.oplens=TRUE)[[1]]

cigars_as_ranges_along_ref(cigar1, reduce.ranges=TRUE,
                           with.ops=TRUE, with.oplens=TRUE)[[1]]

ops <- setdiff(CIGAR_OPS, "N")

cigars_as_ranges_along_ref(cigar1, ops=ops,
                           with.ops=TRUE, with.oplens=TRUE)[[1]]

cigars_as_ranges_along_ref(cigar1, ops=ops, reduce.ranges=TRUE,
                           with.ops=TRUE, with.oplens=TRUE)[[1]]

ops <- setdiff(CIGAR_OPS, c("D", "N"))

cigars_as_ranges_along_ref(cigar1, ops=ops,
                           with.ops=TRUE, with.oplens=TRUE)[[1]]

lmmpos <- c(1, 1001, 1,  351)

cigars_as_ranges_along_ref(my_cigars, lmmpos=lmmpos,
                           with.ops=TRUE, with.oplens=TRUE)

cigars_as_ranges_along_ref(my_cigars, lmmpos=lmmpos,
                           ops=setdiff(CIGAR_OPS, "N"),
                           reduce.ranges=TRUE)

cigars_as_ranges_along_ref(my_cigars, lmmpos=lmmpos,
                           ops=setdiff(CIGAR_OPS, c("D", "N")),
                           reduce.ranges=TRUE)

seqnames <- factor(c("chr6", "chr6", "chr2", "chr6"),
                   levels=c("chr2", "chr6"))
ops <- c("M", "=", "X", "I", "D")
cigars_as_ranges_along_ref(my_cigars, lmmpos=lmmpos, f=seqnames, ops=ops)

## ---------------------------------------------------------------------
## Turn CIGAR strings into ranges along the "query space"
## ---------------------------------------------------------------------

cigars_as_ranges_along_query(my_cigars, with.ops=TRUE, with.oplens=TRUE)

## ---------------------------------------------------------------------
## Turn CIGAR strings into ranges along the "pairwise alignment space"
## ---------------------------------------------------------------------

cigars_as_ranges_along_pwa(my_cigars, with.ops=TRUE, with.oplens=TRUE)
cigars_as_ranges_along_pwa(my_cigars, dense=TRUE,
                           with.ops=TRUE, with.oplens=TRUE)
```

`cigar_extent` *Calculate the number of positions spanned by a CIGAR string*

Description

The *extent* (or length) of an alignment is the number of positions that it spans. Note that positions can be counted with respect to the "reference space", "query space", or "pairwise alignment space". This means that the *extent* of a pairwise alignment depends on the space that we use to count positions.

The *extent* of a CIGAR string is simply the *extent* of the alignment that it describes.

The **cigarillo** package provides three functions to calculate the *extent* of a CIGAR string:

- `cigar_extent_along_ref` calculates the extent along the "reference space".
- `cigar_extent_along_query` calculates the extent along the "query space".
- `cigar_extent_along_pwa` calculates the extent along the "pairwise alignment space".

The three functions are vectorized.

Usage

```
cigar_extent_along_ref(cigars,
             N.regions.removed=FALSE,
             flags=NULL)

cigar_extent_along_query(cigars,
             before.hard.clipping=FALSE, after.soft.clipping=FALSE,
             flags=NULL)

cigar_extent_along_pwa(cigars,
             N.regions.removed=FALSE, dense=FALSE,
             flags=NULL)
```

Arguments

`cigars`	A character vector (or factor) containing CIGAR strings.
`N.regions.removed`	`TRUE` or `FALSE`. If `TRUE`, then `cigar_extent_along_ref` reports the CIGAR extents with respect to the "reference space" from which the N regions have been removed, and `cigar_extent_along_pwa` reports them with respect to the "pairwise alignment space" from which the N regions have been removed.
`flags`	`NULL` or an integer vector containing the SAM flag for each read. According to the SAM Spec v1.4, flag bit 0x4 is the only reliable place to tell whether a segment (or read) is mapped (bit is 0) or not (bit is 1). If the `flags` argument is supplied, then `cigar_extent_along_ref`, `cigar_extent_along_query`, and `cigar_extent_along_pwa` return `NA`s for unmapped reads.
`before.hard.clipping`	`TRUE` or `FALSE`. If `TRUE`, then `cigar_extent_along_query` reports the CIGAR extents with respect to the "query space" to which the H regions have been added. Note that `before.hard.clipping` and `after.soft.clipping` cannot both be `TRUE`.
`after.soft.clipping`	`TRUE` or `FALSE`. If `TRUE`, then `cigar_extent_along_query` reports the CIGAR extents with respect to the "query space" from which the S regions have been removed. Note that `before.hard.clipping` and `after.soft.clipping` cannot both be `TRUE`.

`dense`	`TRUE` or `FALSE`. If `TRUE`, then `cigar_extent_along_pwa` reports the CIGAR extents with respect to the "pairwise alignment space" from which the I, D, and N regions have been removed. Note that `N.regions.removed` and `dense` cannot both be `TRUE`.

Value

For `cigar_extent_along_ref` and `cigar_extent_along_pwa`: An integer vector of the same length as `cigars` where each element is the extent of the alignment with respect to the reference and pairwise space, respectively. More precisely, for `cigar_extent_along_ref`, the returned extents are the lengths of the alignments on the reference, N gaps included (except if `N.regions.removed` is `TRUE`). NAs or `"*"` in `cigars` will produce NAs in the returned vector.

For `cigar_extent_along_query`: An integer vector of the same length as `cigars` where each element is the length of the corresponding query sequence as inferred from the CIGAR string. Note that, by default (i.e. if `before.hard.clipping` and `after.soft.clipping` are `FALSE`), this is the length of the query sequence stored in the SAM/BAM file. If `before.hard.clipping` or `after.soft.clipping` is `TRUE`, the returned extents are the lengths of the query sequences before hard clipping or after soft clipping. NAs or `"*"` in `cigars` will produce NAs in the returned vector.

Author(s)

Hervé Pagès

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_positions` to project positions from query to reference space and vice versa.
- `project_sequences` to project sequences from one space to the other.

Examples

```
my_cigars <- c("40M2I9M", "3H15M55N4M2I6M2D5M6S",
               "2S10M2000N15M", "3H33M5H")

## Extents along the "reference space":
cigar_extent_along_ref(my_cigars)

## Extents along the "query space":
cigar_extent_along_query(my_cigars)
cigar_extent_along_query(my_cigars, before.hard.clipping=TRUE)

## Extents along the "pairwise alignment space":
cigar_extent_along_pwa(my_cigars)
cigar_extent_along_pwa(my_cigars, dense=TRUE)
```

`cigar_ops_visibility` *Visibility of CIGAR operations*

Description

CIGAR operations and their visibility in various *projection spaces*.

Usage

```
CIGAR_OPS

cigar_ops_visibility(ops=CIGAR_OPS)
```

Arguments

`ops`	Character vector where the elements are single letters representing valid CIGAR operations. Must be a subset of `CIGAR_OPS`.

Details

The 8 supported *projection spaces* are: "reference", "reference-N-regions-removed", "query", "query-before-hard-clipping", "query-after-soft-clipping", "pairwise", "pairwise-N-regions-removed", and "pairwise-dense".

Each space can be characterized by the extended CIGAR operations that are *visible* in it. A CIGAR operation is said to be *visible* in a given space if it "runs along it", that is, if it's associated with a block of contiguous positions in that space (the size of the block being the length of the operation). For example, the M/=/X operations are *visible* in all spaces, the D/N operations are *visible* in the "reference" space but not in the "query" space, the S operation is *visible* in the "query" space but not in the "reference" or in the "query-after-soft-clipping" space, etc...

Here are the extended CIGAR operations that are *visible* in each space:

1. reference: M, D, N, =, X
2. reference-N-regions-removed: M, D, =, X
3. query: M, I, S, =, X
4. query-before-hard-clipping: M, I, S, H, =, X
5. query-after-soft-clipping: M, I, =, X
6. pairwise: M, I, D, N, =, X
7. pairwise-N-regions-removed: M, I, D, =, X
8. pairwise-dense: M, =, X

Note that CIGAR operations M, =, and X are visible in all spaces.

Value

`CIGAR_OPS` is a predefined character vector containing the valid (extended) CIGAR operations: M, I, D, N, S, H, P, =, X. See official SAM/BAM Format specs at `https://samtools.github.io/hts-specs/SAMv1.pdf` for the list of extended CIGAR operations and their meaning.

`cigar_ops_visibility()` returns an 8-row integer matrix with 1 row per space and 1 column per CIGAR operation. The matrix is made of 0's and 1's indicating visibility.

Author(s)

Hervé Pagès

See Also

- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_positions` to project positions from query to reference space and vice versa.
- `project_sequences` to project sequences from one space to the other.

Examples

```
CIGAR_OPS  # valid CIGAR operations

cigar_ops_visibility()  # visibility in each "projection space"
```

`explode_cigars` *Explode CIGAR strings*

Description

Use `explode_cigar_ops()` (or `explode_cigar_oplens()`) to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.

Usage

```
explode_cigar_ops(cigars, ops=CIGAR_OPS)
explode_cigar_oplens(cigars, ops=CIGAR_OPS)

cigars_as_RleList(cigars)
```

Arguments

`cigars`	A character vector (or factor) containing CIGAR strings.
`ops`	Character vector where the elements are single letters representing valid CIGAR operations. Must be a subset of `CIGAR_OPS`. See ?`CIGAR_OPS` for more information. `explode_cigar_ops()` and `explode_cigar_oplens()` will ignore operations not listed in `ops` (in addition to 0-length operations which are always ignored).

Value

For `explode_cigar_ops` and `explode_cigar_oplens`: Both functions return a list parallel to `cigars` where each list element is a character vector (for `explode_cigar_ops`) or an integer vector (for `explode_cigar_oplens`). The two lists are guaranteed to have the same shape, that is, the same `length()` and same `lengths()`.

More precisely: The i-th character vector in the list returned by `explode_cigar_ops` contains one single-letter string per CIGAR operation in `cigars[i]`. The i-th integer vector in the list returned by `explode_cigar_oplens` contains the corresponding CIGAR operation lengths. Operations not listed in ops and 0-length operations are ignored.

For `cigars_as_RleList`: An RleList object.

Author(s)

Hervé Pagès, Martin Morgan, and Patrick Aboyoun

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_positions` to project positions from query to reference space and vice versa.
- `project_sequences` to project sequences from one space to the other.
- The RleList class in the **IRanges** package.

Examples

```
## ---------------------------------------------------------------------
## Turn CIGAR strings into other useful representations
## ---------------------------------------------------------------------

my_cigars <- c(
    "40M2I9M",
    "60M",
    "3H15M55N4M2I6M2D5M6S",
    "50=2X3=1X10=",
    "2S10M2000N15M",
    "3H33M5H"
)

cig_ops <- explode_cigar_ops(my_cigars)
cig_ops

cig_oplens <- explode_cigar_oplens(my_cigars)
cig_oplens
```

```
explode_cigar_ops(my_cigars, ops=c("I", "S"))
explode_cigar_oplens(my_cigars, ops=c("I", "S"))

cigs_as_rlelist <- cigars_as_RleList(my_cigars)
cigs_as_rlelist

## ---------------------------------------------------------------------
## Results can be coerced to CharacterList or IntegerList
## ---------------------------------------------------------------------

as(cig_ops, "CharacterList")
as(cig_oplens, "IntegerList")
as(cigs_as_rlelist, "CharacterList")

## ---------------------------------------------------------------------
## Sanity checks
## ---------------------------------------------------------------------

stopifnot(
    identical(as.list(runValue(cigs_as_rlelist)), cig_ops),
    identical(as.list(runLength(cigs_as_rlelist)), cig_oplens)
)
```

---

map_ref_ranges_to_query

Map ranges relative to reference space to query space

Description

Highly specialized utility functions whose main purpose is to support the `mapToAlignments` methods defined in the **GenomicAlignments** package. Only of interest to the authors/maintainers of these methods, and not really meant to be used by the end user.

Usage

```
map_ref_ranges_to_query(start, end, cigars, lmmpos)

fast_map_ref_ranges_to_query(start, end, cigars, lmmpos,
                                strictly.sort.hits=FALSE)
```

Arguments

`start, end`	Two parallel integer vectors containing the starts/ends of the ranges to map to the "query space". Note that the positions in the two vectors are expected to be relative to the "reference space".
`cigars`	A character vector (or factor) containing CIGAR strings.
`lmmpos`	An integer vector parallel to `cigars`. For each CIGAR string in `cigars`, `lmmpos` must contain the 1-based leftmost mapping POSition of the alignment described by the CIGAR string. Note that these positions must be relative to the "reference space".

`strictly.sort.hits`
: Whether the rows in the data.frame returned by `fast_map_ref_ranges_to_query()` should be sorted by `from_hit` first then by `to_hit` instead of by `from_hit` only. Note that when `strictly.sort.hits` is set to TRUE, `fast_map_ref_ranges_to_query()` is guaranted to return the exact same data.frame as `map_ref_ranges_to_query()`.

Details

`map_ref_ranges_to_query()` uses a naive and inefficient approach to find hits between the input ranges and the ranges implicitly defined by the (`cigars[j]`, `lmmpos[j]`) pairs.

`fast_map_ref_ranges_to_query()` is just a reimplementation of `map_ref_ranges_to_query()` that is based on `findOverlaps()`. It's hundreds times faster than `map_ref_ranges_to_query()` for medium size input (i.e. when nb of input ranges x nb of cigars is between 1e6 and 250e6), and thousands to hundreds of thousands times faster or more for big inputs (i.e. when nb of input ranges x nb of cigars is > 500e6).

Value

A 4-column data.frame with 1 hit per row. The columns are:

- `start`, `end`: start/end of input range relative to the "query space";
- `from_hit`: index of input range involved in hit;
- `to_hit`: index of (cigar,lmmpos) pair involved in hit.

The 4 columns are integer vectors.

Author(s)

Valerie Obenchain and Hervé Pagès

See Also

- The `mapToAlignments` methods defined in the **GenomicAlignments** package.
- `ref_pos_as_query_pos` to project positions that are defined along the "reference space" onto the "query space".
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `findOverlaps()` in the **IRanges** package.

Examples

```
set.seed(888)

## Random input ranges:
start <- sample(50000L, 10000, replace=TRUE)
end <- start + sample(15L, 10000, replace=TRUE) - 1L

## Random (cigar,lmmpos) pairs, kind of:
cigars <- sample(c("4M", "5M3I4M", "4M3D5M", "3M", "10M", "5M8N5M"),
                 25000, replace=TRUE)
lmmpos <- sample(50000L, 25000, replace=TRUE)

## map_ref_ranges_to_query():
system.time(df <- map_ref_ranges_to_query(start, end, cigars, lmmpos))
```

```
dim(df)
df[1:15, ]

## fast_map_ref_ranges_to_query() is about 300x-400x faster:
system.time(df2 <- fast_map_ref_ranges_to_query(start, end, cigars, lmmpos,
                                                strictly.sort.hits=TRUE))
stopifnot(identical(df, df2))
```

`project_positions` *Project positions from query to reference space and vice versa*

Description

`query_pos_as_ref_pos()` projects positions defined along the "query space" onto the "reference space", that is, it turns them into positions defined along the "reference space".

`ref_pos_as_query_pos()` does the opposite i.e. it projects positions that are defined along the "reference space" onto the "query space".

Usage

```
query_pos_as_ref_pos(query_pos, cigars, lmmpos, narrow.left)

ref_pos_as_query_pos(ref_pos, cigars, lmmpos, narrow.left)
```

Arguments

`query_pos`	An integer vector containing positions relative to the "query space".
`cigars`	A character vector (or factor) parallel to `query_pos` containing CIGAR strings.
`lmmpos`	An integer vector parallel to `cigars` and `query_pos`. For each CIGAR string in `cigars`, `lmmpos` must contain the 1-based leftmost mapping POSition of the alignment described by the CIGAR string. Note that these positions must be relative to the "reference space".
`ref_pos`	An integer vector containing positions relative to the "reference space".
`narrow.left`	For `query_pos_as_ref_pos()`: How should positions in the "query space" that fall within an insertion be treated? Such positions are peculiar, because, strictly speaking, they don't have corresponding positions in the "reference space". Instead, each of them falls *between* two adjacent positions in the "reference space". Another way to describe this situation is to say that each of them is mapped to a zero-width range along the "reference space". If `narrow.left` is `TRUE`, such position will be mapped to the position that is immediately on the left of the corresponding zero-width range on the "reference space". If `narrow.left` is `FALSE`, it will be mapped to the position that is immediately on the right of the corresponding zero-width range on the "reference space". For `ref_pos_as_query_pos()`: How should positions in the "reference space" that fall within a deletion be treated? Such positions are peculiar, because, strictly speaking, they don't have corresponding positions in the "query space". Instead, each of them falls *between*

two adjacent positions in the "query space". Another way to describe this situation is to say that each of them is mapped to a zero-width range along the "query space".

If `narrow.left` is `TRUE`, such position will be mapped to the position that is immediately on the left of the corresponding zero-width range on the "query space". If `narrow.left` is `FALSE`, it will be mapped to the position that is immediately on the right of the corresponding zero-width range on the "query space".

Value

An integer vector parallel to the input positions. `NA`s in the returned vector indicate input positions that cannot be mapped.

Author(s)

Michael Lawrence

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_sequences` to project sequences from one space to the other.

Examples

```
query_pos <- -1:11
cigars <- rep("5M3I2M", 13)
lmmpos <- rep(101, 13)
query_pos_as_ref_pos(query_pos, cigars, lmmpos, narrow.left=TRUE)
query_pos_as_ref_pos(query_pos, cigars, lmmpos, narrow.left=FALSE)
```

`project_sequences` *Project sequences from one space to the other*

Description

`project_sequences` projects sequences that belong to a given *projection space* (e.g. the "query space") onto another *projection space* (e.g. the "reference space") by removing/injecting substrings from/into them, based on their corresponding CIGAR string.

Its primary use case is to project the read sequences stored in a BAM file (which are considered to belong to the "query space") onto the "reference space". It can also be used to remove the parts of the read sequences that correspond to soft-clipping. More generally it can project sequences that belong to any supported space onto any other supported space. See the Details section below for the list of supported spaces.

Usage

```
project_sequences(x, cigars, from="query", to="reference",
                  I.letter="-", D.letter="-", N.letter=".",
                  S.letter="+", H.letter="+")
```

Arguments

`x`	An XStringSet derivative (e.g. BStringSet, DNAStringSet, or AAStringSet object) containing sequences that are considered to belong to the `from` space (see below).
`cigars`	A character vector (or factor) parallel to `x` containing CIGAR strings.
`from`, `to`	A single string specifying one of the 8 supported "projection spaces". See `?cigar_ops_visibility` for more information. `from` must be the current space (i.e. the space that the sequences in `x` belong to) and `to` is the space onto which the sequences in `x` must be projected.
`I.letter`, `D.letter`, `N.letter`, `S.letter`, `H.letter`	A single letter used as a filler for injections. More on this in the Details section below.

Details

See `?cigar_ops_visibility` for the 8 supported *projection spaces*.

`project_sequences` projects a sequence that belongs to one space onto another by (1) removing the substrings associated with operations that are no longer *visible* in the new space, and (2) injecting substrings associated with operations that become *visible* in the new space. Each injected substring has the length of the operation associated with it, and its content is controlled via the corresponding `*.letter` argument.

For example, when going from the "query" space to the "reference" space (the default), the I- and S-substrings (i.e. the substrings associated with I/S operations) are removed, and substrings associated with D/N operations are injected. More precisely, the D-substrings are filled with the letter specified in `D.letter`, and the N-substrings with the letter specified in `N.letter`. The other `*.letter` arguments are ignored in that case.

Value

An XStringSet derivative of the same class as input object `x`, and parallel to `x`. The names on `x`, if any, are propagated.

Author(s)

Hervé Pagès

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_positions` to project positions from query to reference space and vice versa.
- The `stackStringsFromBam` function in the **GenomicAlignments** package for stacking the read sequences (or their quality strings) stored in a BAM file on a region of interest.
- The `readGAlignments` function in the **GenomicAlignments** package for loading read sequences from a BAM file (as a GAlignments object).
- The `extractAt` and `replaceAt` functions in the **Biostrings** package for extracting/replacing arbitrary substrings from/in a string or set of strings.

Examples

```
library(GenomicAlignments)

## ---------------------------------------------------------------------
## A. FROM "query" TO "reference" SPACE
## ---------------------------------------------------------------------

## Load read sequences from a BAM file (they will be returned in a
## GAlignments object):
bamfile <- system.file("extdata", "ex1.bam", package="Rsamtools")
param <- ScanBamParam(what="seq")
gal <- readGAlignments(bamfile, param=param)
qseq <- mcols(gal)$seq  # the read sequences (aka query sequences)

## Project the query sequences onto the reference space. This will
## remove the substrings associated with insertions to the reference
## (I operations) and soft clipping (S operations), and will inject new
## substrings (filled with "-") where deletions from the reference (D
## operations) and skipped regions from the reference (N operations)
## occurred during the alignment process:
qseq_on_ref <- project_sequences(qseq, cigar(gal))

## A typical use case for doing the above is to compute 1 consensus
## sequence per chromosome. The code below shows how this can be done
## in 2 extra steps.

## Step 1: Compute one consensus matrix per chromosome.
qseq_on_ref_by_chrom <- splitAsList(qseq_on_ref, seqnames(gal))
pos_by_chrom <- splitAsList(start(gal), seqnames(gal))

cm_by_chrom <- lapply(names(pos_by_chrom),
```

```
    function(seqname)
        consensusMatrix(qseq_on_ref_by_chrom[[seqname]],
                        as.prob=TRUE,
                        shift=pos_by_chrom[[seqname]]-1,
                        width=seqlengths(gal)[[seqname]]))
names(cm_by_chrom) <- names(pos_by_chrom)

## 'cm_by_chrom' is a list of consensus matrices. Each matrix has 17
## rows (1 per letter in the DNA alphabet) and 1 column per chromosome
## position.

## Step 2: Compute the consensus string from each consensus matrix.
## We'll put "+" in the strings wherever there is no coverage for that
## position, and "N" where there is coverage but no consensus.
cs_by_chrom <- lapply(cm_by_chrom,
    function(cm) {
        ## Because consensusString() doesn't like consensus matrices
        ## with columns that contain only zeroes (and you will have
        ## columns like that for chromosome positions that don't
        ## receive any coverage), we need to "fix" 'cm' first.
        idx <- colSums(cm) == 0
        cm["+", idx] <- 1
        DNAString(consensusString(cm, ambiguityMap="N"))
    })

## consensusString() provides some flexibility to let you extract
## the consensus in different ways. See '?consensusString' in the
## Biostrings package for the details.

## Finally, note that the read quality strings can also be used as
## input for project_sequences():
param <- ScanBamParam(what="qual")
gal <- readGAlignments(bamfile, param=param)
qual <- mcols(gal)$qual  # the read quality strings
qual_on_ref <- project_sequences(qual, cigar(gal))
## Note that since the "-" letter is a valid quality code, there is
## no way to distinguish it from the "-" letters inserted by
## project_sequences().

## ---------------------------------------------------------------------
## B. FROM "query" TO "query-after-soft-clipping" SPACE
## ---------------------------------------------------------------------

## Going from "query" to "query-after-soft-clipping" simply removes
## the substrings associated with soft clipping (S operations):
qseq <- DNAStringSet(c("AAAGTTCGAA", "TTACGATTAN", "GGATAATTTT"))
cigars <- c("3H10M", "2S7M1S2H", "2M1I1M3D2M4S")
clipped_qseq <- project_sequences(qseq, cigars,
                                  from="query",
                                  to="query-after-soft-clipping")

project_sequences(clipped_qseq, cigars,
                  from="query-after-soft-clipping", to="query")

project_sequences(clipped_qseq, cigars,
                  from="query-after-soft-clipping", to="query",
                  S.letter="-")
```

```
## ---------------------------------------------------------------------
## C. BRING QUERY AND REFERENCE SEQUENCES TO THE "pairwise"
##    OR "pairwise-dense" SPACE
## ---------------------------------------------------------------------

## Load read sequences from a BAM file:
library(RNAseqData.HNRNPC.bam.chr14)
bamfile <- RNAseqData.HNRNPC.bam.chr14_BAMFILES[1]
param <- ScanBamParam(what="seq",
                      which=GRanges("chr14", IRanges(1, 25000000)))
gal <- readGAlignments(bamfile, param=param)
qseq <- mcols(gal)$seq  # the read sequences (aka query sequences)

## Load the corresponding reference sequences from the appropriate
## BSgenome package (the reads in RNAseqData.HNRNPC.bam.chr14 were
## aligned to hg19):
library(BSgenome.Hsapiens.UCSC.hg19)
rseq <- getSeq(Hsapiens, as(gal, "GRanges"))  # the reference sequences

## Bring 'qseq' and 'rseq' to the "pairwise" space.
## For 'qseq', this will remove the substrings associated with soft
## clipping (S operations) and inject substrings (filled with "-")
## associated with deletions from the reference (D operations) and
## skipped regions from the reference (N operations). For 'rseq', this
## will inject substrings (filled with "-") associated with insertions
## to the reference (I operations).
qseq2 <- project_sequences(qseq, cigar(gal),
                           from="query", to="pairwise")
rseq2 <- project_sequences(rseq, cigar(gal),
                           from="reference", to="pairwise")

## Sanity check: 'qseq2' and 'rseq2' should have the same shape.
stopifnot(identical(elementNROWS(qseq2), elementNROWS(rseq2)))

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

I_idx <- which(cigar_op_table[ , "I"] >= 2)  # at least 2 insertions
qseq2[I_idx]
rseq2[I_idx]

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

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

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

```
rseq3 <- project_sequences(rseq, cigar(gal),
                           from="reference", to="pairwise-dense")

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

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

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

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

`tabulate_cigar_ops` *Tabulate CIGAR operations*

Description

Count the occurences of CIGAR operations in a vector of CIGAR strings.

Usage

```
tabulate_cigar_ops(cigars, oplens.as.weights=FALSE)
```

Arguments

`cigars`	A character vector (or factor) containing CIGAR strings.
`oplens.as.weights`	`TRUE` or `FALSE`. Should the operation lengths be used as weights for the counts?

Value

An integer matrix with 1 row per CIGAR string in `cigars` and 1 column per CIGAR operation in `CIGAR_OPS`.

Author(s)

Patrick Aboyoun and Hervé Pagès

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.

- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- `trim_cigars_along_ref` and `trim_cigars_along_query` to trim CIGAR strings along the "reference space" and "query space", respectively.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_positions` to project positions from query to reference space and vice versa.
- `project_sequences` to project sequences from one space to the other.

Examples

```
my_cigars <- c(
    "40M2I9M",
    "60M",
    "3H15M55N4M2I6M2D5M6S",
    "50=2X3=1X10=",
    "2S10M2000N15M",
    "3H33M5H"
)

op_counts <- tabulate_cigar_ops(my_cigars)
op_counts

tabulate_cigar_ops(my_cigars, oplens.as.weights=TRUE)

## Get the total number of operations per CIGAR string:
rowSums(op_counts)  # a numeric vector parallel to 'my_cigars'

## Note that the above is equivalent to -- but much faster and more
## memory-efficient than -- 'lengths(explode_cigar_ops(my_cigars))'
## or 'lengths(explode_cigar_oplens(my_cigars))':
nop_per_cig <- as.integer(rowSums(op_counts))
stopifnot(
    identical(nop_per_cig, lengths(explode_cigar_ops(my_cigars))),
    identical(nop_per_cig, lengths(explode_cigar_oplens(my_cigars)))
)

## Identify CIGAR strings with indels:
has_indels <- rowSums(op_counts[ , c("I", "D")]) != 0
has_indels  # a logical vector parallel to 'my_cigars'

## Summarize the counts for the whole vector of CIGAR strings:
colSums(op_counts)
```

`trim_cigars` *Trim CIGAR strings along the reference or query space*

Description

The CIGAR string associated with a pairwise alignment describes the alignment in its entirety in the sense that it covers all the positions in the alignment. However, there might be situations where one is only interested in a particular portion of the alignment, that is, in the portion of the alignment that is left after trimming it by a given number of positions on its left and/or right ends. Furthermore, one might want to know the effect of this trimming on the original CIGAR string.

The **cigarillo** package provides two core functions, `trim_cigars_along_ref` and `trim_cigars_along_query`, to compute the CIGAR string that describes a "trimmed alignment". Both take:

- the original CIGAR string i.e. the CIGAR string that describes the alignment before trimming
- the numbers of left/right positions to trim

Both functions return the "trimmed CIGAR string", that is, the CIGAR string that describes the "trimmed alignment".

The only difference between the two function is how the numbers of left and right positions to trim are counted: with respect to the "reference space" for `trim_cigars_along_ref`, and with respect to the "query space" for `trim_cigars_along_query`.

Both functions are vectorized.

Usage

```
trim_cigars_along_ref(cigars, Lnpos=0L, Rnpos=0L)
trim_cigars_along_query(cigars, Lnpos=0L, Rnpos=0L)

## Wrappers to the above that do the same thing but via
## the "narrow()" interface:
narrow_cigars_along_ref(cigars, start=NA, end=NA, width=NA)
narrow_cigars_along_query(cigars, start=NA, end=NA, width=NA)
```

Arguments

`cigars`	A character vector (or factor) containing CIGAR strings.
`Lnpos`, `Rnpos`	The numbers of left/right positions to trim. Each of `Lnpos` and `Rnpos` must be a non-negative integer, or a vector of non-negative integers of the same length as `cigars`. Note that the numbers of left and right positions to trim are counted with respect to the "reference space" for `trim_cigars_along_ref`, and with respect to the "query space" for `trim_cigars_along_query`.
`start`, `end`, `width`	Vectors of integers. `NA`s and negative values are allowed and "solved" similarly to what `IRanges::narrow()` does. See `?IRanges::narrow` in the **IRanges** package for more information.

Value

A character vector of the same length as `cigars` that contains the "trimmed CIGAR strings".

In addition the vector has an "rshift" attribute which is an integer vector of the same length as `cigars`. It contains the values that would need to be added to the POS field (1-based leftmost mapping POSition) of a SAM/BAM file as a consequence of this trimming.

Author(s)

Hervé Pagès

See Also

- `cigar_ops_visibility` for an introduction to CIGAR operations and their visibility in various "projection spaces".
- explode_cigars to extract the letters (or lengths) of the CIGAR operations contained in a vector of CIGAR strings.
- `tabulate_cigar_ops` to count the occurences of CIGAR operations in a vector of CIGAR strings.
- cigar_extent for functions that calculate the *extent* of a CIGAR string, that is, the number of positions spanned by the alignment that it describes.
- cigars_as_ranges to turn CIGAR strings into ranges of positions.
- `project_positions` to project positions from query to reference space and vice versa.
- `project_sequences` to project sequences from one space to the other.

Examples

```
cigar1 <- "3H15M55N4M2I6M2D5M6S"

## trim_cigars_along_ref():
trim_cigars_along_ref(cigar1)  # only drops the soft/hard clipping
trim_cigars_along_ref(cigar1, Lnpos=9)
trim_cigars_along_ref(cigar1, Lnpos=14)
trim_cigars_along_ref(cigar1, Lnpos=14, Rnpos=16)
trim_cigars_along_ref(cigar1, Lnpos=15)
#trim_cigars_along_ref(cigar1, Lnpos=15, Rnpos=17)  # error! (empty cigar)
trim_cigars_along_ref(cigar1, Lnpos=70)
trim_cigars_along_ref(cigar1, Lnpos=71)
trim_cigars_along_ref(cigar1, Lnpos=74)

## trim_cigars_along_query():
trim_cigars_along_query(cigar1, Lnpos=3, Rnpos=2)
trim_cigars_along_query(cigar1, Lnpos=9)
trim_cigars_along_query(cigar1, Lnpos=18)
trim_cigars_along_query(cigar1, Lnpos=23)

## Using the "narrow()" interface:

stopifnot(
  ## narrow_cigars_along_ref() vs trim_cigars_along_ref():
  identical(narrow_cigars_along_ref(cigar1, start=10),
            trim_cigars_along_ref(cigar1, Lnpos=9)),
  identical(narrow_cigars_along_ref(cigar1, start=15),
            trim_cigars_along_ref(cigar1, Lnpos=14)),
  identical(narrow_cigars_along_ref(cigar1, start=15, width=57),
            trim_cigars_along_ref(cigar1, Lnpos=14, Rnpos=16)),
  identical(narrow_cigars_along_ref(cigar1, start=16),
            trim_cigars_along_ref(cigar1, Lnpos=15)),
  identical(narrow_cigars_along_ref(cigar1, start=71),
            trim_cigars_along_ref(cigar1, Lnpos=70)),
  identical(narrow_cigars_along_ref(cigar1, start=72),
            trim_cigars_along_ref(cigar1, Lnpos=71)),
  identical(narrow_cigars_along_ref(cigar1, start=75),
            trim_cigars_along_ref(cigar1, Lnpos=74)),

  ## narrow_cigars_along_query() vs trim_cigars_along_query():
```

```
  identical(narrow_cigars_along_query(cigar1, start=4, end=-3),
            trim_cigars_along_query(cigar1, Lnpos=3, Rnpos=2)),
  identical(narrow_cigars_along_query(cigar1, start=10),
            trim_cigars_along_query(cigar1, Lnpos=9)),
  identical(narrow_cigars_along_query(cigar1, start=19),
            trim_cigars_along_query(cigar1, Lnpos=18)),
  identical(narrow_cigars_along_query(cigar1, start=24),
            trim_cigars_along_query(cigar1, Lnpos=23))
)
```

Index