Package ‘ggDNAvis’

March 6, 2026

Title 'ggplot2'-Based Tools for Visualising DNA Sequences and Modifications

Version 1.0.0

Description Uses 'ggplot2' to visualise either (a) a single DNA/RNA sequence split across multiple lines, (b) multiple DNA/RNA sequences, each occupying a whole line, or (c) base modifications such as DNA methylation called by modified bases models in Dorado or Guppy. Functions starting with visualise_<>() are the main plotting functions, and functions starting with extract_and_sort_<>() are key helper functions for reading files and reformatting data. Source code is available at <https://github.com/ejade42/ggDNAvis>, a full non-expert user guide is available at <https://ejade42.github.io/ggDNAvis/>, and an interactive web-app version of the software is available at <https://ejade42.github.io/ggDNAvis/articles/interactive_app.html>.

Imports cli, dplyr, ggnewscale, ggplot2 (>= 4.0.0), grid, magick, png, ragg, rlang, stats, stringr, tidyr, utils

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Suggests bslib, colourpicker, jsonlite, knitr, markdown, rmarkdown, shiny, shinyjs, testthat (>= 3.0.0)

Config/testthat/edition 3

Depends R (>= 3.5)

LazyData true

Language en-GB

URL https://ejade42.github.io/ggDNAvis/, https://github.com/ejade42/ggDNAvis, https://doi.org/10.64898/2026.03.02.708895

BugReports https://github.com/ejade42/ggDNAvis/issues

VignetteBuilder knitr

NeedsCompilation no

Author Evelyn Jade [aut, cre, cph] (ORCID: <https://orcid.org/0009-0003-7761-5425>)

Maintainer Evelyn Jade <evelynjade42@gmail.com>

Repository CRAN

Date/Publication 2026-03-06 03:10:02 UTC

Contents

---

bad_arg — *Emit an error message for an invalid function argument (generic* `ggDNAvis` *helper)*

Description

This function takes an argument name, a named list of arguments (presumably being iterated over for a particular validation check), and a message. Using `rlang::abort()`, it prints an error message of the form:

```
Argument '<argument_name>' <message>
Current value: <argument_value>
Current class: <class(argument_value)>
```

If the argument value is a named item (i.e. `names(arguments_list[[argument_name]])` is not null), or if `force_names` is `TRUE`, then the form will be:

```
Argument '<argument_name>' <message>
Current value: <argument_value>
Current names: <argument_names>
Current class: <class(argument_value)>
```

Usage

```
bad_arg(
  argument_name,
  arguments_list,
  message,
  class = "argument_value_or_type",
  force_names = FALSE
)
```

Arguments

`argument_name`	`character`. The name of the argument that caused the error
`arguments_list`	`list`. A named list where `arguments_list[[argument_name]]` is the value of the offending argument.
`message`	`character`. The message that should be printed to describe why the argument is invalid.

`class`	`character`. The class that the error should have. Defaults to `"argument_value_or_type"` for my own use.
`force_names`	`logical`. Whether the names argument should be printed even if names is `NULL`. Defaults to `FALSE`.

Value

Nothing, but causes an error exit via `rlang::abort()`

Examples

```
## Obviously this error-message function causes an error,
## so needs to be wrapped in try() for these examples

## Standard use
positive_args <- list(number = -1)
try(bad_arg("number", positive_args, "must be positive"))

## Automatically detects named item and prints names
named <- list(x = c("first item" = 1, "second item" = 7))
try(bad_arg("x", named, "is not acceptable"))

## Can force name printing
try(bad_arg("number", positive_args, "must be positive", force_names = TRUE))
```

`convert_base_to_number` *Map a single base to the corresponding number (generic* `ggDNAvis` *helper)*

Description

This function takes a single base and numerically encodes it for visualisation via `rasterise_matrix()`.

Encoding: `A = 1`, `C = 2`, `G = 3`, `T/U = 4`.

Usage

```
convert_base_to_number(base)
```

Arguments

`base`	`character`. A single DNA/RNA base to encode numerically (e.g. `"A"`).

Value

`integer`. The corresponding number.

Examples

```
convert_base_to_number("A")
convert_base_to_number("c")
convert_base_to_number("g")
convert_base_to_number("T")
convert_base_to_number("u")
```

`convert_locations_to_MM_vector` *Convert absolute index locations to MM tag (`write_modified_fastq()` helper)*

Description

This function takes a vector of modified base locations as absolute indices (i.e. a `1` would mean the first base in the sequence has been assessed for modification; a `15` would mean the 15th base has), and converts it to a vector in the format of the SAM/BAM MM tags. The MM tag defines a particular target base (e.g. `C` for methylation), and then stores the number of skipped instances of that base between sites where modification was assessed. In practice, this often means counting the number of non-CpG `C`s in between CpG `C`s. In a `GGC` repeat, this should be a bunch of `0`s as every `C` is in a CpG, but unique sequence will have many non-CpG `C`s.

This function is reversed by `convert_MM_vector_to_locations()`.

Usage

```
convert_locations_to_MM_vector(sequence, locations, target_base = "C")
```

Arguments

`sequence`	`character`. The DNA sequence about which the methylation information is being processed.
`locations`	`integer vector`. All of the base indices at which methylation/modification information was processed. Must all be instances of the target base.
`target_base`	`character`. The base type that has been assessed or skipped (defaults to `"C"`).

Value

`integer vector`. A component of a SAM MM tag, representing the number of skipped target bases in between each assessed base.

Examples

```
convert_locations_to_MM_vector(
    "GGCGGCGGCGGC",
    locations = c(3, 6, 9, 12),
    target_base = "C"
)

convert_locations_to_MM_vector(
    "GGCGGCGGCGGC",
    locations = c(1, 4, 7, 10),
    target_base = "G"
)

convert_locations_to_MM_vector(
    "GGCGGCGGCGGC",
    locations = c(1, 2, 4, 5, 7, 8, 10, 11),
    target_base = "G"
)
```

---

convert_MM_vector_to_locations

Convert MM tag to absolute index locations (deprecated helper)

Description

This function takes a sequence, a SAM-style vector of number of potential target bases to skip in between each target base that was actually assessed, and a target base type (defaults to `"C"` as 5-methylcytosine is most common).

It identifies the indices/locations of all instances of the target base within the sequence, and then goes along the vector of these indices, skipping them if requested by `skips`.

For example, the sequence `"GGCGGCGGCGGC"` with target `"C"` and skips `c(0, 0, 1)` would identify that the indices where `"C"` occurs are `c(3, 6, 9, 12)`. It would then take the first index, the second index, skip one, and take the fourth index i.e. return `c(3, 6, 12)`. If instead the skips were given as `c(0, 2)` it would take the first index, skip two, and take the fourth index i.e. return `c(3, 12)`. If the skips were given as `c(1, 1)` it would skip one, take the second index, skip one, and take the fourth index i.e. return `c(6, 12)`.

The length of `skips` corresponds to the number of indices/locations that will be returned (i.e. the length of the returned locations vector).

Ideally the length of `skips` plus the sum of `skips` (i.e. the number returned plus the total number skipped) is the same or less than the number of possible locations. If it is the same, then the last possible location will be taken; if it is less then some number of possible locations at the end will be skipped.

Important: if the length of `skips` plus the sum of `skips` is greater than the number of possible locations (instances of the target base within the sequence), then the total number of taken or skipped locations will be greater than the number of available locations. In this case, the returned vector will contain NA after the available locations have run out. In the example above, `skips = c(0, 0, 0, 0, 0)` would return `c(3, 6, 9, 12, NA)`, and `skips = c(0, 2, 0)` would return `c(3, 12, NA)`.

Therefore, if the target base is totally absent from the sequence (e.g. target `"A"` in `"GGCGGCGGCGGC"`), then any non-zero length of `skips` will return the same length of `NA`s e.g. `skips = c(0)` would return `NA`, and `skips = c(0, 1, 0)` would return `c(NA, NA, NA)`.

If `skips` has length zero, it will return `numeric(0)`.

This function is reversed by `convert_locations_to_MM_vector()`.

Usage

```
convert_MM_vector_to_locations(sequence, skips, target_base = "C")
```

Arguments

`sequence`	`character`. The DNA sequence about which the methylation information is being processed.
`skips`	`integer vector`. A component of a SAM MM tag, representing the number of skipped target bases in between each assessed base.
`target_base`	`character`. The base type that has been assessed or skipped (defaults to `"C"`).

Value

`integer vector`. All of the base indices at which methylation/modification information was processed. Will all be instances of the target base.

Examples

```
convert_MM_vector_to_locations(
    "GGCGGCGGCGGC",
    skips = c(0, 0, 0, 0),
    target_base = "C"
)

convert_MM_vector_to_locations(
    "GGCGGCGGCGGC",
    skips = c(1, 1, 1, 1),
    target_base = "G"
)

convert_MM_vector_to_locations(
    "GGCGGCGGCGGC",
    skips = c(0, 0, 2, 1, 0),
    target_base = "G"
)
```

`convert_modification_to_number_vector`
Convert string-ified modification probabilities and locations to a single vector of probabilities (`visualise_methylation()` helper)

Description

Takes modification locations (indices along the read signifying bases at which modification probability was assessed) and modification probabilities (the probability of modification at each assessed location, as an integer from 0 to 255), as comma-separated strings (e.g. `"1,5,25"`) produced from numerical vectors via `vector_to_string()`. Outputs a numerical vector of the modification probability for each base along the read. i.e. -2 for indices outside sequences, -1 for bases where modification was not assessed, and probability from 0-255 for bases where modification was assessed.

Usage

```
convert_modification_to_number_vector(
  modification_locations_str,
  modification_probabilities_str,
  max_length,
  sequence_length
)
```

Arguments

`modification_locations_str`	`character`. A comma-separated string representing a condensed numerical vector (e.g. `"3,6,9,12"`, produced via `vector_to_string()`) of the indices along the read at which modification was assessed. Indexing starts at 1.
`modification_probabilities_str`	`character`. A comma-separated string representing a condensed numerical vector (e.g. `"2,212,128,64"`, produced via `vector_to_string()`) of the probability of modification as an 8-bit (0-255) integer for each base where modification was assessed.
`max_length`	`integer`. How long the output vector should be.
`sequence_length`	`integer`. How long the sequence itself is. If smaller than `max_length`, the remaining spaces will be filled with `-2`s i.e. set to the background colour in `visualise_methylation()`.

Value

`numeric vector`. A vector of length `max_length` indicating the probability of methylation at each index along the read - 0 where methylation was not assessed, and probability from 0-255 where methylation was assessed.

Examples

```
convert_modification_to_number_vector(
    modification_locations_str = "3,6,9,12",
    modification_probabilities = "100,200,50,150",
    max_length = 15,
    sequence_length = 13
)
```

`convert_sequences_to_matrix` *Convert vector of sequences to character matrix (generic `ggDNAvis` helper)*

Description

This function takes a vector of sequences (e.g. input to `visualise_many_sequences()` or `visualise_methylation()`, or vector split from input to `visualise_single_sequence()`). It converts it into a matrix e.g. `c("GGCGGC", "", "ACGT", "")` would become:

```
G  G  C  G  G  C
NA NA NA NA NA NA
A  C  G  T  NA NA
NA NA NA NA NA NA
```

The resulting matrix can then be rasterised into a coordinate-value dataframe via `rasterise_matrix()`.

Usage

```
convert_sequences_to_matrix(sequences, line_length = NA, blank_value = NA)
```

Arguments

`sequences`	`character vector`. The sequences to transform into a matrix
`line_length`	`integer`. The width of the matrix. Set to `NA` (default) to automatically use the length of the longest sequence in `sequences`.
`blank_value`	`value`. The value that should be used to fill in blank/missing points of the matrix.

Value

`matrix`. A matrix of the sequences with one line per sequence, ready for rasterisation via `rasterise_matrix()`.

Examples

```
convert_sequences_to_matrix(
    sequences = c("GGCGGC", "", "ACGT", "")
)

convert_sequences_to_matrix(
    sequences = c("GGCGGC", "", "ACGT", ""),
    line_length = 10,
    blank_value = "X"
)
```

`convert_sequence_to_numbers`

Map a sequence to a vector of numbers (generic `ggDNAvis` helper)

Description

This function takes a sequence and encodes it as a vector of numbers for visualisation via `rasterise_matrix()`.

Encoding: `A = 1`, `C = 2`, `G = 3`, `T/U = 4`.

Usage

```
convert_sequence_to_numbers(sequence, length = NA)
```

Arguments

`sequence`	`character`. A DNA/RNA sequence (`A/C/G/T/U`) to be encoded numerically. No other characters allowed. Only one sequence allowed.
`length`	`integer`. How long the output numerical vector should be. If shorter than the sequence, the vector will include the first *n* bases up to this length. If longer than the sequence, the vector will be padded with 0s at the end. If left blank/set to `NA` (default), will output a vector the same length as the input sequence.

Value

`integer vector`. The numerical encoding of the input sequence, cut/padded to the desired length.

Examples

```
convert_sequence_to_numbers("ATCGATCG")
convert_sequence_to_numbers("ATCGATCG", length = NA)
convert_sequence_to_numbers("ATCGATCG", length = 4)
convert_sequence_to_numbers("ATCGATCG", length = 10)
```

`create_image_data` *Rasterise a vector of sequences into a numerical dataframe for ggplotting (generic `ggDNAvis` helper)*

Description

Takes a character vector of sequences (which are allowed to be empty `""` to act as a spacing line) and rasterises it into a dataframe that ggplot can read.

Usage

```
create_image_data(sequences)
```

Arguments

`sequences`	`character vector`. A vector of sequences for plotting, e.g. `c("ATCG", "", "GGCGGC", "")`. Each sequence will be plotted left-aligned on a new line.

Value

`dataframe`. Rasterised dataframe representation of the sequences, readable by `ggplot2::ggplot()`.

Examples

```
create_image_data(c("ATCG", "", "GGCGGC", ""))
```

`debug_join_vector_num` *Print a numeric vector to console (`ggDNAvis` debug helper)*

Description

Takes a numeric vector, and prints it to the console separated by `", "`.

This allows the output to be copy-pasted into a vector within an R script. Used for taking vector outputs and then writing them as literals within a script.

E.g. when given input `1:5`, prints `1, 2, 3, 4, 5`, which can be directly copy-pasted within `c()` to input that vector. Printing normally via `print(1:5)` instead prints `[1] 1 2 3 4 5`, which is not valid vector input so can't be copy-pasted directly.

See `debug_join_vector_str()` for the equivalent for character/string vectors.

Usage

```
debug_join_vector_num(vector)
```

Arguments

`vector`	`numeric vector`. Usually generated by some other function. This function allows copy-pasting the output to directly create a vector with this value.

Value

None (invisible `NULL`) - uses `cat()` to output directly to console.

Examples

```
debug_join_vector_num(1:5)
```

`debug_join_vector_str` *Print a character/string vector to console (*`ggDNAvis` *debug helper)*

Description

Takes a character/string vector, and prints it to the console separated by `", "`.

This allows the output to be copy-pasted into a vector within an R script. Used for taking vector outputs and then writing them as literals within a script.

E.g. when given input `strsplit("ABCD", split = "")[[1]]`, prints `"A", "B", "C", "D"`, which can be directly copy-pasted within `c()` to input that vector. Printing normally via `print(strsplit("ABCD", split = "")[[1]])` instead prints `[1] "A" "B" "C" "D"`, which is not valid vector input so can't be copy-pasted directly.

See `debug_join_vector_num()` for the equivalent for numeric vectors.

Usage

```
debug_join_vector_str(vector)
```

Arguments

`vector`	`character vector`. Usually generated by some other function. This function allows copy-pasting the output to directly create a vector with this value.

Value

None (invisible `NULL`) - uses `cat()` to output directly to console.

Examples

```
debug_join_vector_str(c("A", "B", "C", "D"))
```

`example_many_sequences` *Example multiple sequences data*

Description

A collection of made-up sequences in the style of long reads over a repeat region (e.g. *NOTCH2NLC*), with meta-data describing the participant each read is from and the family each participant is from. Can be used in `visualise_many_sequences()`, `visualise_methylation()`, and helper functions to visualise these sequences.

Generation code is available at `data-raw/example_many_sequences.R`

Usage

```
example_many_sequences
```

Format

`example_many_sequences`**:**

A dataframe with 23 rows and 10 columns:

family Participant family

individual Participant ID

read Unique read ID

sequence DNA sequence of the read

sequence_length Length (nucleotides) of the read

quality FASTQ quality scores for the read. Each character represents a score from 0 to 40 - see `fastq_quality_scores`.

These values are made up via `pmin(pmax(round(rnorm(n, mean = 20, sd = 10)), 0), 40)` i.e. sampled from a normal distribution with mean 20 and standard deviation 10, then rounded to integers between 0 and 40 (inclusive) - see `example_many_sequences.R`

methylation_locations Indices along the read (starting at 1) at which methylation probability was assessed i.e. CpG sites. Stored as a single character value per read, condensed from a numeric vector via `vector_to_string()`.

methylation_probabilities Probability of methylation (8-bit integer i.e. 0-255) for each assessed base. Stored as a single character value per read, condensed from a numeric vector via `vector_to_string()`.

These values are made up via `round(runif(n, min = 0, max = 255))` - see `example_many_sequences.R`

hydroxymethylation_locations Indices along the read (starting at 1) at which hydroxymethylation probability was assessed i.e. CpG sites. Stored as a single character value per read, condensed from a numeric vector via `vector_to_string()`.

hydroxymethylation_probabilities Probability of hydroxymethylation (8-bit integer i.e. 0-255) for each assessed base. Stored as a single character value per read, condensed from a numeric vector via `vector_to_string()`.

These values are made up via `round(runif(n, min = 0, max = 255 - this_base_methylation_probability))` such that the summed methylation and hydroxymethylation probability never exceeds 255 (100%) - see `example_many_sequences.R`

Examples

```
example_many_sequences
```

`extract_and_sort_methylation`

Extract methylation information from dataframe for visualisation

---

Description

`extract_methylation_from_dataframe()` is an alias for `extract_and_sort_methylation()` - see aliases.

This function takes a dataframe that contains methylation information in the form of locations (indices along the read signifying bases at which modification probability was assessed) and probabilities (the probability of modification at each assessed location, as an integer from 0 to 255).

Each observation/row in the dataframe represents one sequence (e.g. a Nanopore read). In the locations and probabilities column, each sequence (row) has many numbers associated. These are stored as one string per observation e.g. `"3,6,9,12"`, with the column representing a character vector of such strings (e.g. `c("3,6,9,12", "1,2,3,4")`).

This function calls `extract_and_sort_sequences()` on the relevant columns and returns a list of vectors stored in `$locations`, `$probabilities`, `$sequences`, and `$lengths`. These can then be used as input for `visualise_methylation()`.

Default arguments are set up to work with the included `example_many_sequences` data.

Usage

```
extract_and_sort_methylation(
  modification_data,
  ...,
  locations_colname = "methylation_locations",
  probabilities_colname = "methylation_probabilities",
  sequences_colname = "sequence",
  lengths_colname = "sequence_length",
  grouping_levels = c(family = 8, individual = 2),
  sort_by = "sequence_length",
  desc_sort = TRUE
)
```

Arguments

`modification_data`
: `dataframe`. A dataframe that must contain columns for methylation locations, methylation probabilities, sequence, and sequence length for each read. The former two should be condensed strings as produced by `vector_to_string()` e.g. `"1,2,3,4"`.

  See `example_many_sequences` for an example of a compatible dataframe.

`...`
: Used to recognise aliases e.g. American spellings or common misspellings - see aliases. If any American spellings do not work, please make a bug report at `https://github.com/ejade42/ggDNAvis/issues`.

`locations_colname`
: `character`. The name of the column within the input dataframe that contains methylation/modification location information. Defaults to `"methylation_locations"`.

  Values within this column must be a comma-separated string representing a condensed numerical vector (e.g. `"3,6,9,12"`, produced via `vector_to_string()`) of the indices along the read at which modification was assessed. Indexing starts at 1.

`probabilities_colname`
: `character`. The name of the column within the input dataframe that contains methylation/modification probability information. Defaults to `"methylation_probabilities"`.

  Values within this column must be a comma-separated string representing a condensed numerical vector (e.g. `"2,212,128,64"`, produced via `vector_to_string()`) of the probability of modification as an 8-bit (0-255) integer for each base where modification was assessed.

`sequences_colname`
: `character`. The name of the column within the input dataframe that contains the actual sequences. Defaults to `"sequence"`.

  Values within this column must be the actual sequences (e.g. `"GGCGGA"`).

`lengths_colname`
: `character`. The name of the column within the input dataframe that contains the length of each sequence. Defaults to `"sequence_length"`.

  Values within this column must be non-negative integers.

`grouping_levels`
: `named character vector`. What variables should be used to define the groups/chunks, and how large a gap should be left between groups at that level. Set to `NA` to turn off grouping.

  Defaults to `c("family" = 8, "individual" = 2)`, meaning the highest-level groups are defined by the `family` column, and there is a gap of 8 between each family. Likewise the second-level groups (within each family) are defined by the `individual` column, and there is a gap of 2 between each individual.

Any number of grouping variables and gaps can be given, as long as each grouping variable is a column within the dataframe. It is recommended that lower-level groups are more granular and subdivide higher-level groups (e.g. first divide into families, then into individuals within families).

To change the order of groups within a level, make that column a factor with the order specified e.g. `example_many_sequences$family <- factor(example_many_sequences$family, levels = c("Family 2", "Family 3", "Family 1"))` to change the order to Family 2, Family 3, Family 1.

`sort_by`	`character`. The name of the column within the dataframe that should be used to sort/order the rows within each lowest-level group. Set to `NA` to turn off sorting within groups. Recommended to be the length of the sequence information, as is the case for the default `"sequence_length"` which was generated via `example_many_sequences$sequence_length <- nchar(example_many_sequences$sequence)`.
`desc_sort`	`logical`. Boolean specifying whether rows within groups should be sorted by the `sort_by` variable descending (`TRUE`, default) or ascending (`FALSE`).

Value

`list`, containing `$locations` (`character vector`), `$probabilities` (`character vector`), `$sequences` (`character vector`), and `$lengths` (`integer vector`).

Examples

```
## See documentation for extract_and_sort_sequences()
## for more examples of changing sorting/grouping
extract_and_sort_methylation(
    example_many_sequences,
    locations_colname = "methylation_locations",
    probabilities_colname = "methylation_probabilities",
    sequences_colname = "sequence",
    lengths_colname = "sequence_length",
    grouping_levels = c("family" = 8, "individual" = 2),
    sort_by = "sequence_length",
    desc_sort = TRUE
)

extract_and_sort_methylation(
    example_many_sequences,
    locations_colname = "hydroxymethylation_locations",
    probabilities_colname = "hydroxymethylation_probabilities",
    sequences_colname = "sequence",
    lengths_colname = "sequence_length",
    grouping_levels = c("family" = 8, "individual" = 2),
    sort_by = "sequence_length",
    desc_sort = TRUE
)
```

`extract_and_sort_sequences`
Extract, sort, and add spacers between sequences in a dataframe

---

Description

`extract_sequences_from_dataframe()` is an alias for `extract_and_sort_sequences()` - see aliases.

This function takes a dataframe that contains sequences and metadata, recursively splits it into multiple levels of groups defined by `grouping_levels`, and adds breaks between each level of group as defined by `grouping_levels`. Within each lowest-level group, reads are sorted by `sort_by`, with order determined by `desc_sort`.

Default values are set up to work with the included dataset `example_many_sequences`.

The returned sequences vector is ideal input for `visualise_many_sequences()`.

Also called by `extract_methylation_from_dataframe()` to produce input for `visualise_methylation()`.

Usage

```
extract_and_sort_sequences(
  sequence_dataframe,
  sequence_variable = "sequence",
  grouping_levels = c(family = 8, individual = 2),
  sort_by = "sequence_length",
  desc_sort = TRUE
)
```

Arguments

`sequence_dataframe`
: `dataframe`. A dataframe containing the sequence information and all required meta-data. See `example_many_sequences` for an example of a compatible dataframe.

`sequence_variable`
: `character`. The name of the column within the dataframe containing the sequence information to be output. Defaults to `"sequence"`.

`grouping_levels`
: `named character vector`. What variables should be used to define the groups/chunks, and how large a gap should be left between groups at that level. Set to `NA` to turn off grouping.

  Defaults to `c("family" = 8, "individual" = 2)`, meaning the highest-level groups are defined by the `family` column, and there is a gap of 8 between each family. Likewise the second-level groups (within each family) are defined by the `individual` column, and there is a gap of 2 between each individual.

  Any number of grouping variables and gaps can be given, as long as each grouping variable is a column within the dataframe. It is recommended that lower-level groups are more granular and subdivide higher-level groups (e.g. first divide into families, then into individuals within families).

To change the order of groups within a level, make that column a factor with the order specified e.g. `example_many_sequences$family <- factor(example_many_sequences$family levels = c("Family 2", "Family 3", "Family 1"))` to change the order to Family 2, Family 3, Family 1.

`sort_by` `character`. The name of the column within the dataframe that should be used to sort/order the rows within each lowest-level group. Set to `NA` to turn off sorting within groups.

Recommended to be the length of the sequence information, as is the case for the default `"sequence_length"` which was generated via `example_many_sequences$sequence_length <- nchar(example_many_sequences$sequence)`.

`desc_sort` `logical`. Boolean specifying whether rows within groups should be sorted by the `sort_by` variable descending (`TRUE`, default) or ascending (`FALSE`).

Value

`character vector`. The sequences ordered and grouped as specified, with blank sequences (`""`) inserted as spacers as specified.

Examples

```
extract_and_sort_sequences(
    example_many_sequences,
    sequence_variable = "sequence",
    grouping_levels = c("family" = 8, "individual" = 2),
    sort_by = "sequence_length",
    desc_sort = TRUE
)

extract_and_sort_sequences(
    example_many_sequences,
    sequence_variable = "sequence",
    grouping_levels = c("family" = 3),
    sort_by = "sequence_length",
    desc_sort = FALSE
)

extract_and_sort_sequences(
    example_many_sequences,
    sequence_variable = "sequence",
    grouping_levels = NA,
    sort_by = "sequence_length",
    desc_sort = TRUE
)

extract_and_sort_sequences(
    example_many_sequences,
    sequence_variable = "sequence",
    grouping_levels = c("family" = 8, "individual" = 2),
    sort_by = NA
)

extract_and_sort_sequences(
    example_many_sequences,
    sequence_variable = "sequence",
    grouping_levels = NA,
    sort_by = NA
)

extract_and_sort_sequences(
    example_many_sequences,
    sequence_variable = "quality",
    grouping_levels = c("individual" = 3),
    sort_by = "quality",
    desc_sort = FALSE
)
```

`fastq_quality_scores` *Vector of the quality scores used by the FASTQ format*

Description

A vector of the characters used to indicate quality scores from 0 to 40 in the FASTQ format. These scores are related to the error probability p via $Q = -10 \log_{10}(p)$, so a Q-score of 10 (represented by `"+"`) means the error probability is 0.1, a Q-score of 20 (`"5"`) means the error probability is 0.01, and a Q-score of 30 (`"?"`) means the error probability is 0.001.

The character representations store Q-scores in one byte each by using ASCII encodings, where the Q-score for a character is its ASCII code minus 33 (e.g. `A` has an ASCII code of 65 and represents a Q-score of 32).

This vector contains the characters in order but starting with a score of 0, meaning the character at index n represents a Q-score of $n - 1$ e.g. the first character (`"!"`) represents a score of 0; the eleventh character (`"+"`) represents a score of 10.

The full set of possible score representations, in order and presented as a single string, is `!"#$%&'()*+,-./0123456789:;<=>`

Generation code is available at `data-raw/fastq_quality_scores.R`

Usage

```
fastq_quality_scores
```

Format

`fastq_quality_scores`:

A character vector of length 41

fastq_quality_scores The vector `c("!", '"', "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", ":", ";", "<", "=", ">", "?", "@", "A", "B", "C", "D", "E", "F", "G", "H", "I")`

Examples

```
fastq_quality_scores
```

`format_time_diff` *Format a difference between times (generic* `ggDNAvis` *helper)*

Description

This function takes two times (class `"POSIXct"`) and formats the difference between them nicely, with a certain number of numerical characters printed.

Note that the if the time difference rounded to the integer number of seconds (e.g. 1234 seconds) requires more space than the number of characters allocated (e.g. 3 characters) then it will go beyond the specified characters. However, this would be an exceptionally slow-running function. In normal monitoring use for `monitor()`, <1 second steps should be nearly universal, and <0.01 second steps are very common.

Usage

```
format_time_diff(new_time, old_time, characters_to_print = 4)
```

Arguments

`new_time`	`POSIXct`. The more recent (newer) of the two times to calculate a difference between.
`old_time`	`POSIXct`. The less recent (older) of the two times to calculate a difference between.
`characters_to_print`	`integer`. How many numeric digits should be printed.

Value

`character`. The formatted time difference in seconds.

Examples

```
## POSIXct time is a very large number of seconds
newer <- 1000000001
older <- 1000000000
format_time_diff(newer, older, 4)

newer <- 1000000456.45645
older <- 1000000000
format_time_diff(newer, older, 4)
format_time_diff(newer, older, 3)
format_time_diff(newer, older, 2)

newer <- 1000000000.011
older <- 1000000000
format_time_diff(newer, older, 4)
format_time_diff(newer, older, 3)
format_time_diff(newer, older, 2)
```

ggDNAvis_aliases   ggDNAvis *aliases*

Description

As of v1.0.0, ggDNAvis supports function and argument aliases. The code is entirely written with British spellings (e.g. visualise_methylation_colour_scale()), but should also accept American spellings (e.g. visualize_methylation_color_scale()). If any American spellings don't work, I most likely overlooked them and can easily fix, so please submit a bug report by creating a github issue (https://github.com/ejade42/ggDNAvis/issues).

All four major visualise_ functions have aliases to also accept visualize_:

- visualise_many_sequences() (visualize_many_sequences())
- visualise_methylation() (visualize_methylation())
- visualise_methylation_colour_scale() (visualize_methylation_color_scale())
- visualise_single_sequence() (visualize_single_sequence())

As of v1.0.0, extract_methylation_from_dataframe() has been renamed extract_and_sort_methylation() for consistency with extract_and_sort_sequences(). To preserve compatibility and ensure consistency, both functions now accept either name formulation:

- extract_and_sort_sequences() (extract_sequences_from_dataframe())
- extract_and_sort_methylation() (extract_methylation_from_dataframe())

The builtin dataset sequence_colour_palettes, like all colour arguments, also accepts color or col:

- sequence_colour_palettes (sequence_color_palettes & sequence_col_palettes)

The interactive shinyapp can be called via ggDNAvis_shinyapp() or ggDNAvis_shiny().

Additionally, the three rasterise_ helper functions also accept rasterize_:

- rasterise_matrix() (rasterize_matrix())
- rasterise_index_annotations() (rasterize_index_annotations())
- rasterise_probabilities() (rasterize_probabilities())

All arguments should have aliases configured. In particular, any _colour arguments should also accept _color or _col.

When more than one equivalent argument is provided, the 'canonical' (British) argument takes precedence, and will produce a warning message explaining this. For colours, _colour takes precedence over _color, which itself takes precedence over _col.

I have also tried to provide aliases for common argument misspellings. In particular, index_annotation_full_line also accepts any of index_annotations_full_lines, index_annotation_full_lines, or index_annotations_full_li
Likewise, index_annotations_above also accepts index_annotation_above.

Examples

```
d <- extract_methylation_from_dataframe(example_many_sequences)
## The resulting low colour will be green
visualise_methylation(
    d$locations,
    d$probabilities,
    d$sequences,
    index_annotation_lines = NA,
    outline_linewidth = 0,
    high_colour = "white",
    low_colour = "green",
    low_color = "orange",
    low_col = "purple"
)

## The resulting low colour will be orange
visualise_methylation(
    d$locations,
    d$probabilities,
    d$sequences,
    index_annotation_lines = NA,
    outline_linewidth = 0,
    high_colour = "white",
    low_color = "orange",
    low_col = "purple"
)

## The resulting low colour will be purple
visualise_methylation(
    d$locations,
    d$probabilities,
    d$sequences,
    index_annotation_lines = NA,
    outline_linewidth = 0,
    high_colour = "white",
    low_col = "purple"
)
```

`ggDNAvis_shinyapp` *Run the interactive* `ggDNAvis` *shinyapp*

Description

`ggDNAvis_shiny()` is an alias for `ggDNAvis_shinyapp()` - see aliases.

The `ggDNAvis` shinyapp is an interactive frontend for the `ggDNAvis` functions. Arguments can be configured via text/numerical/colour/checkbox entry rather than on the command line. In the future it will be hosted online, but is currently accessible only by running the shinyapp locally.

This function checks 'suggests' packages are present (not needed for main package, but needed for the shinyapp) and then runs the shinyapp in the `inst/shinyapp` directory.

Usage

```
ggDNAvis_shinyapp(themer = FALSE, return = FALSE)
```

Arguments

`themer`	`logical`. Whether the theme picker should be shown. Makes the app harder to use, not recommended except for dev purposes.
`return`	`logical`. Whether to return the shiny app object instead of running it. Intended for hosting the shinyapp.

Value

Nothing, or the shiny app object

Examples

```
## Not run:
## Run normally
ggDNAvis_shinyapp()
ggDNAvis_shinyapp(themer = FALSE, return = FALSE)

## Run with theme picker (dev)
ggDNAvis_shinyapp(themer = TRUE)

## Run, returning object (dev)
ggDNAvis_shinyapp(return = TRUE)

## End(Not run)
```

insert_at_indices — *Insert blank items at specified indices in a vector (visualise_many_sequences() helper)*

Description

This function takes a vector (e.g. the output of `extract_and_sort_sequences()`) and inserts a specified "blank" value at the specified indices. If `insert_before` is `TRUE` then the blank value will be inserted before each specified index, whereas if `insert_before` is `FALSE` then the blank value will be inserted after each specified index.

Usage

```
insert_at_indices(
  original_vector,
  insertion_indices,
  insert_before = TRUE,
  insert = "",
  vert = NA
)
```

Arguments

`original_vector`	`vector`. The vector to insert blanks into at specified locations (e.g. vector of sequences from extract_and_sort_sequences, but doesn't have to be).
`insertion_indices`	`integer vector`. The indices (1-indexed) at which blanks should be inserted. If length 0, no blanks will be inserted.
`insert_before`	`logical`. Whether blanks should be inserted before (`TRUE`, default) or after (`FALSE`) each specified index. Values must be sorted and unique.
`insert`	`value`. The value that should be inserted before/after each specified index. Defaults to `""`. If length 0, nothing will be inserted. If length > 1, multiple items will be inserted at each specified index.
`vert`	`numerical`. The vertical distance into the box that index annotations will be drawn. If set to `NA` (default) does nothing so that this function is more generalisable. If set to a number, then the `insert` will be repeated `ceiling(vert)` times each time it is inserted.

Value

`vector`. The original vector but with the `insert` value added before/after each specified index.

Examples

```
insert_at_indices(c("A", "B", "C", "D", "E"), c(2, 4))

insert_at_indices(
    c("A", "B", "C", "D", "E"),
    c(2, 4),
    insert_before = TRUE,
    insert = 0
)

insert_at_indices(
    c("A", "B", "C", "D", "E"),
    c(2, 4),
    insert_before = FALSE,
    insert = 0
)

insert_at_indices(
    original_vector = c("A", "B", "C", "D", "E"),
    insertion_indices = c(1, 4, 6),
    insert_before = TRUE,
    insert = c("X", "Y")
)

insert_at_indices(
    list("A", "B", "C", "D", "E"),
    c(2, 4),
    insert = TRUE
)

insert_at_indices(
    list("A", "B", "C", "D", "E"),
    c(2, 4),
    insert_before = FALSE,
    insert = list(TRUE, 7)
)

insert_at_indices(
    NA,
    c(1, 2),
    FALSE
)

insert_at_indices(
    c("A", "B", "C", "D", "E"),
    integer(0)
)
```

`merge_fastq_with_metadata` *Merge FASTQ data with metadata*

Description

Merge a dataframe of sequence and quality data (as produced by `read_fastq()` from an unmodified FASTQ file) with a dataframe of metadata, reverse-complementing sequences if required such that all reads are now in the forward direction. `merge_methylation_with_metadata()` is the equivalent function for working with FASTQs that contain DNA modification information.

FASTQ dataframe must contain columns of `"read"` (unique read ID), `"sequence"` (DNA sequence), and `"quality"` (FASTQ quality score). Other columns are allowed but not required, and will be preserved unaltered in the merged data.

Metadata dataframe must contain `"read"` (unique read ID) and `"direction"` (read direction, either `"forward"` or `"reverse"` for each read) columns, and can contain any other columns with arbitrary information for each read. Columns that might be useful include participant ID and family designations so that each read can be associated with its participant and family.

Important: A key feature of this function is that it uses the direction column from the metadata to identify which rows are reverse reads. These reverse reads will then be reversed-complemented and have quality scores reversed such that all reads are in the forward direction, ideal for consistent analysis or visualisation. The output columns are `"forward_sequence"` and `"forward_quality"`. Calls `reverse_sequence_if_needed()` and `reverse_quality_if_needed()` to implement the reversing - see documentation for these functions for more details.

Usage

```
merge_fastq_with_metadata(
  fastq_data,
  metadata,
  reverse_complement_mode = "DNA"
)
```

Arguments

`fastq_data`	`dataframe`. A dataframe contaning sequence and quality data, as produced by `read_fastq()`. Must contain a read id column (must be called `"read"`), a sequence column (`"sequence"`), and a quality column (`"quality"`). Additional columns are fine and will simply be included unaltered in the merged dataframe.
`metadata`	`dataframe`. A dataframe containing metadata for each read in `fastq_data`. Must contain a `"read"` column identical to the column of the same name in `fastq_data`, containing unique read IDs (this is used to merge the dataframes). Must also contain a `"direction"` column of `"forward"` and `"reverse"` (e.g. `c("forward", "forward", "reverse")`) indicating the direction of each read.

Important: Reverse reads will have their sequence and quality scores reversed such that every output read is now forward. These will be stored in columns called `"forward_sequence"` and `"forward_quality"`.

See `reverse_sequence_if_needed()` and `reverse_quality_if_needed()` documentation for details of how the reversing is implemented.

`reverse_complement_mode`

`character`. Whether reverse-complemented sequences should be converted to DNA (i.e. A complements to T), RNA (i.e. A complements to U), or not complemented at all. Must be `"DNA"` (default), `"RNA"`, or `"reverse_only"` to reverse sequence order without complementing. *Only affects reverse-complemented sequences. Sequences that were forward to begin with are not altered.*

Uses `reverse_complement()` via `reverse_sequence_if_needed()`.

Value

`dataframe`. A merged dataframe containing all columns from the input dataframes, as well as forward versions of sequences and qualities.

Examples

```
## Locate files
fastq_file <- system.file("extdata",
                          "example_many_sequences_raw.fastq",
                          package = "ggDNAvis")
metadata_file <- system.file("extdata",
                             "example_many_sequences_metadata.csv",
                             package = "ggDNAvis")

## Read files
fastq_data <- read_fastq(fastq_file)
metadata   <- read.csv(metadata_file)

## Merge data (including reversing if needed)
merge_fastq_with_metadata(
    fastq_data,
    metadata
)

## Merge data reversing but not complementing sequences
merge_fastq_with_metadata(
    fastq_data,
    metadata,
    reverse_complement_mode = "reverse_only"
)
```

merge_methylation_with_metadata

Merge methylation with metadata

Description

Merge a dataframe of methylation/modification data (as produced by `read_modified_fastq()`) with a dataframe of metadata, reversing sequence and modification information if required such that all information is now in the forward direction. `merge_fastq_with_metadata()` is the equivalent function for working with unmodified FASTQs (sequence and quality only).

Methylation/modification dataframe must contain columns of `"read"` (unique read ID), `"sequence"` (DNA sequence), `"quality"` (FASTQ quality score), `"sequence_length"` (read length), `"modification_types"` (a comma-separated string of SAMtools modification headers produced via `vector_to_string()` e.g. `"C+h?,C+m?"`), and, for each modification type, a column of comma-separated strings of modification locations (e.g. `"3,6,9,12"`) and a column of comma-separated strings of modification probabilities (e.g. `"255,0,64,128"`). See `read_modified_fastq()` for more information on how this dataframe is formatted and produced. Other columns are allowed but not required, and will be preserved unaltered in the merged data.

Metadata dataframe must contain `"read"` (unique read ID) and `"direction"` (read direction, either `"forward"` or `"reverse"` for each read) columns, and can contain any other columns with arbitrary information for each read. Columns that might be useful include participant ID and family designations so that each read can be associated with its participant and family.

Important: A key feature of this function is that it uses the direction column from the metadata to identify which rows are reverse reads. These reverse reads will then be reversed-complemented and have modification information reversed such that all reads are in the forward direction, ideal for consistent analysis or visualisation. The output columns are `"forward_sequence"`, `"forward_quality"`, `"forward_<modification_type>_locations"`, and `"forward_<modification_type>_probabilities"`.

Calls `reverse_sequence_if_needed()`, `reverse_quality_if_needed()`, `reverse_locations_if_needed()`, and `reverse_probabilities_if_needed()` to implement the reversing - see documentation for these functions for more details. If wanting to write reversed sequences to FASTQ via `write_modified_fastq()`, locations must be symmetric (e.g. CpG) and offset must be set to 1. Asymmetric locations are impossible to write to modified FASTQ once reversed because then e.g. cytosine methylation will be assessed at guanines, which SAMtools can't account for. Symmetrically reversing CpGs via `reversed_location_offset = 1` is the only way to fix this. **PLEASE READ THE `reverse_locations_if_needed()` DOCUMENTATION TO UNDERSTAND THE CHOICE OF OFFSET!**

Usage

```
merge_methylation_with_metadata(
  methylation_data,
  metadata,
  reversed_location_offset = 0,
  reverse_complement_mode = "DNA"
)
```

Arguments

methylation_data

dataframe. A dataframe contaning methylation/modification data, as produced by read_modified_fastq().

Must contain a read id column (must be called "read"), a sequence column ("sequence"), a quality column ("quality"), a sequence length column ("sequence_length"), a modification types column ("modification_types"), and, for each modification type listed in modification_types, a column of locations ("<modification_type>_locations") and a column of probabilities ("<modification_type>_probabilities"). Additional columns are fine and will simply be included unaltered in the merged dataframe.

See read_modified_fastq() documentation for more details about the expected dataframe format.

metadata

dataframe. A dataframe containing metadata for each read in methylation_data.

Must contain a "read" column identical to the column of the same name in methylation_data, containing unique read IDs (this is used to merge the dataframes). Must also contain a "direction" column of "forward" and "reverse" (e.g. c("forward", "forward", "reverse")) indicating the direction of each read.

Important: Reverse reads will have their sequence, quality scores, modification locations, and modification probabilities reversed such that every output read is now forward. These will be stored in columns called "forward_sequence", "forward_quality", "forward_<modification_type>_locations", and "forward_<modification_
If multiple modification types are present, multiple locations and probabilities columns will be created.

See reverse_sequence_if_needed(), reverse_quality_if_needed(), reverse_locations_if_nee
and reverse_probabilities_if_needed() documentation for details of how the reversing is implemented.

reversed_location_offset

integer. How much modification locations should be shifted by. Defaults to 0. This is important because if a CpG is assessed for methylation at the C, then reverse complementing it will give a methylation score at the G on the reverse-complemented strand. This is the most biologically accurate, but for visualising methylation it may be desired to shift the locations by 1 i.e. to correspond with the C in the reverse-complemented CpG rather than the G, which allows for consistent visualisation between forward and reverse strands. Setting (integer) values other than 0 or 1 will work, but may be biologically misleading so it is not recommended.

Highly recommended: if considering using this option, read the reverse_locations_if_needed() documentation to fully understand how it works.

reverse_complement_mode

character. Whether reverse-complemented sequences should be converted to DNA (i.e. A complements to T), RNA (i.e. A complements to U), or not complemented at all. Must be "DNA" (default), "RNA", or "reverse_only" to reverse sequence order without complementing. *Only affects reverse-complemented sequences. Sequences that were forward to begin with are not altered.*

Uses reverse_complement() via reverse_sequence_if_needed().

Value

dataframe. A merged dataframe containing all columns from the input dataframes, as well as forward versions of sequences, qualities, modification locations, and modification probabilities (with separate locations and probabilities columns created for each modification type in the modification data).

Examples

```
## Locate files
modified_fastq_file <- system.file("extdata",
                                   "example_many_sequences_raw_modified.fastq",
                                   package = "ggDNAvis")
metadata_file <- system.file("extdata",
                             "example_many_sequences_metadata.csv",
                             package = "ggDNAvis")

## Read files
methylation_data <- read_modified_fastq(modified_fastq_file)
metadata <- read.csv(metadata_file)

## Merge data (including reversing if needed)
merge_methylation_with_metadata(
    methylation_data,
    metadata,
    reversed_location_offset = 0
)

## Merge data with offset = 1
merge_methylation_with_metadata(
    methylation_data,
    metadata,
    reversed_location_offset = 1
)

## Merge data with offset = 1 but without complementing
merge_methylation_with_metadata(
    methylation_data,
    metadata,
    reversed_location_offset = 1,
    reverse_complement_mode = "reverse_only"
)
```

`monitor` *Continue performance monitoring (generic* `ggDNAvis` *helper)*

Description

This function is meant to be called frequently throughout a main function, and if verbose performance monitoring is enabled then it will print the elapsed time since (a) initialisation via `monitor_start()` and (b) since the last step was recorded via this function.

Usage

```
monitor(monitor_performance, start_time, previous_time, message)
```

Arguments

`monitor_performance`	`logical`. Whether verbose performance monitoring should be enabled.
`start_time`	`POSIXct`. The time at which the overarching function was initialised (generally via `monitor_start()`).
`previous_time`	`POSIXct`. The time at which the previous step was recorded (via a prior call to `monitor()`).
`message`	`character`. The message to be printed, generally indicating what this step is doing

Value

`POSIXct` the time at which the function was called, via `Sys.time()`.

Examples

```
## Initialise monitoring
start_time <- monitor_start(TRUE, "my_cool_function")

## Step 1
monitor_time <- monitor(TRUE, start_time, start_time, "performing step 1")
x <- 2 + 2

## Step 2
monitor_time <- monitor(TRUE, start_time, monitor_time, "performing step 2")
y <- 10.5^6 %% 345789

## Step 3
monitor_time <- monitor(TRUE, start_time, monitor_time, "performing step 3")
z <- y / x^2

## Conclude monitoring
monitor_time <- monitor(TRUE, start_time, monitor_time, "done")
```

`monitor_start` *Start performance monitoring (generic* `ggDNAvis` *helper)*

Description

This function takes a bool of whether verbose performance monitoring is on, as well as the name of the calling function, prints a monitoring initialisation message (if desired), and returns the start time.

Later monitoring steps are performed by `monitor()`

Usage

```
monitor_start(monitor_performance, function_name)
```

Arguments

`monitor_performance`	`logical`. Whether verbose performance monitoring should be enabled.
`function_name`	`character`. The name of the calling function, printed as part of the monitoring initialisation message.

Value

`POSIXct` the time at which the function was initialised, via `Sys.time()`.

Examples

```
## Initialise monitoring
start_time <- monitor_start(TRUE, "my_cool_function")

## Step 1
monitor_time <- monitor(TRUE, start_time, start_time, "performing step 1")
x <- 2 + 2

## Step 2
monitor_time <- monitor(TRUE, start_time, monitor_time, "performing step 2")
y <- 10.5^6 %% 345789

## Step 3
monitor_time <- monitor(TRUE, start_time, monitor_time, "performing step 3")
z <- y / x^2

## Conclude monitoring
monitor_time <- monitor(TRUE, start_time, monitor_time, "done")
```

rasterise_index_annotations *Process index annotations and rasterise to a x/y/layer dataframe (generic* ggDNAvis *helper)*

Description

`rasterize_index_annotations()` is an alias for `rasterise_index_annotations()`.

This function is called by `visualise_many_sequences()`, `visualise_methylation()`, and `visualise_single_sequence` to create the x/y position data for placing the index annotations on the graph. Its arguments are either intermediate variables produced by the visualisation functions, or arguments of the visualisation functions directly passed through.

Returns a dataframe with `x_position`, `y_position`, and `value` columns, where the values are the index annotations.

Usage

```
rasterise_index_annotations(
  new_sequences_vector,
  original_sequences_vector,
  index_annotation_lines,
  index_annotation_interval = 15,
  index_annotation_full_line = TRUE,
  index_annotations_above = TRUE,
  index_annotation_vertical_position = 1/3,
  index_annotation_always_first_base = TRUE,
  index_annotation_always_last_base = TRUE,
  sum_indices = FALSE,
  spacing = NA,
  offset_start = 0
)
```

Arguments

`new_sequences_vector`
: `vector`. The output of `insert_at_indices()` when used with identical arguments. Should be `original_sequences_vector` with some additional blank lines inserted.

`original_sequences_vector`
: `vector`. The vector of sequences used for plotting, that was originally given to `visualise_many_sequences()`/`visualise_methylation()` or split from the input sequence to `visualise_single_sequence()`. Must also have been used as input to `insert_at_indices()` to create `new_sequences_vector`.

`index_annotation_lines`
: `integer vector`. The lines (i.e. elements of `sequences_vector`) that should have their base incides annotated. 1-indexed e.g. `c(1, 10)` would annotate the

first and tenth elements of sequences_vector.

Extra lines are inserted above or below (depending on index_annotations_above) the selected lines - note that the line numbers come from sequences_vector, so are unaffected by these insertions.

Setting to NA disables index annotations (and prevents adding additional blank lines). Defaults to c(1) i.e. first sequence is annotated.

Note: visualise_methylation() and visualise_many_sequences() accept NA, NULL, numeric(0), 0, and FALSE as meaning "annotations off".

index_annotation_interval
: integer. The frequency at which numbers should be placed underneath indicating base index, starting counting from the leftmost base. Defaults to 15 (every 15 bases along each row).

  Setting to 0 disables index annotations (and prevents adding additional blank lines).

index_annotation_full_line
: logical. Whether index annotations should continue to the end of the longest sequence (TRUE, default) or should only continue as far as each selected line does (FALSE).

index_annotations_above
: logical. Whether index annotations should go above (TRUE, default) or below (FALSE) each line of sequence.

index_annotation_vertical_position
: numeric. How far annotation numbers should be rendered above (if index_annotations_above = TRUE) or below (if index_annotations_above = FALSE) each base. Defaults to 1/3.

  Not recommended to change at all. Strongly discouraged to set below 0 or above 1.

index_annotation_always_first_base
: logical. Whether to force the first base in each line to always be annotated regardless of whether it occurs at the index_annotation_interval. Defaults to TRUE.

index_annotation_always_last_base
: logical. Whether to force the last base in each line to always be annotated regardless of whether it occurs at the index_annotation_interval. Defaults to TRUE.

sum_indices
: logical. Whether indices should be counted separately along each line (FALSE, default) or summed along all annotated lines (TRUE). May behave unexpectedly if TRUE when annotate_full_lines is also TRUE.

spacing
: integer. The number of blank lines inserted for each index annotation. Set to NA (default) to infer from annotation_vertical_position.

offset_start
: integer. The number of blank lines not present at the start, that otherwise would be expected based on spacing or annotation_vertical_position. Defaults to 0.

Value

`dataframe`. A dataframe with columns `x`, `y`, and `value`, with one observation per annotation number that needs to be drawn onto the ggplot.

Examples

```
## Set up arguments (e.g. from visualise_many_sequences() call)
sequences_data <- example_many_sequences
index_annotation_lines <- c(1, 23, 37)
index_annotation_interval <- 10
index_annotations_above <- TRUE
index_annotation_full_line <- FALSE
index_annotation_vertical_position <- 1/3


## Create sequences vector
sequences <- extract_and_sort_sequences(
    example_many_sequences,
    grouping_levels = c("family" = 8, "individual" = 2)
)
sequences

## Insert blank rows as needed
new_sequences <- insert_at_indices(
    sequences,
    insertion_indices = index_annotation_lines,
    insert_before = index_annotations_above,
    insert = "",
    vert = index_annotation_vertical_position
)
new_sequences

## Create annnotation dataframe
rasterise_index_annotations(
    new_sequences_vector = new_sequences,
    original_sequences_vector = sequences,
    index_annotation_lines = index_annotation_lines,
    index_annotation_interval = 10,
    index_annotation_full_line = index_annotation_full_line,
    index_annotations_above = index_annotations_above,
    index_annotation_vertical_position = index_annotation_vertical_position,
    index_annotation_always_first_base = TRUE,
    index_annotation_always_last_base = TRUE,
    sum_indices = FALSE,
    spacing = NA, ## infer from vertical position
    offset_start = 0
)
```

rasterise_matrix	*Rasterise a matrix to an x/y/layer dataframe (generic* ggDNAvis *helper)*

Description

rasterize_matrix() is an alias for rasterise_matrix().

This function takes a matrix and rasterises it to a dataframe of x and y coordinates, such that the matrix occupies the space from (0, 0) to (1, 1) and each element of the matrix represents a rectangle with width 1/ncol(matrix) and height 1/nrow(matrix). The "layer" column of the dataframe is simply the value of each element of the matrix.

Usage

```
rasterise_matrix(image_matrix, drop_na = TRUE)
```

Arguments

image_matrix	matrix. A matrix (or anything that can be coerced to a matrix via base::as.matrix()).
drop_na	logical. A boolean specifying whether missing values should be dropped via tidyr::drop_na() (TRUE, default) or kept (FALSE).

Value

dataframe. A dataframe containing x and y coordinates for the centre of a rectangle per element of the matrix, such that the whole matrix occupies the space from (0, 0) to (1, 1). Additionally contains a layer column storing the value of each element of the matrix.

Examples

```
## Create numerical matrix
example_matrix <- matrix(1:16, ncol = 4, nrow = 4, byrow = TRUE)

## View
example_matrix

## Rasterise
rasterise_matrix(example_matrix)



## Create character matrix
example_matrix <- matrix(
    c("A", "B", "C", "D", "E",
      "F", "G", "H", "I", "J"),
    nrow = 2, ncol = 5, byrow = TRUE
)
```

```
## View
example_matrix

## Rasterise
rasterise_matrix(example_matrix)



## Create realistic DNA matrix
dna_matrix <- matrix(
    c(0, 0, 0, 0, 0, 0, 0, 0,
      3, 3, 2, 3, 3, 2, 4, 4,
      0, 0, 0, 0, 0, 0, 0, 0,
      4, 1, 4, 1, 0, 0, 0, 0),
    nrow = 4, ncol = 8, byrow = TRUE
)

## View
dna_matrix

## Rasterise
rasterise_matrix(dna_matrix)


## Create matrix with missing values
incomplete_matrix <- matrix(
    c(1, 2, 3, NA,
      5, NA, 7, 8),
    nrow = 2, ncol = 4, byrow = TRUE
)

## View
incomplete_matrix

## Rasterise, dropping NAs (default)
rasterise_matrix(incomplete_matrix, drop_na = TRUE)

## Rasterise, keeping NAs
rasterise_matrix(incomplete_matrix, drop_na = FALSE)
```

`rasterise_probabilities` *Create dataframe of locations and rendered probabilities (`visualise_methylation()` helper)*

Description

This function takes the locations/probabilities/sequences input to `visualise_methylation()`, as well as the scaling and rounding to apply to the probability text, and produces a dataframe of the x and y coordinates to draw each probability at (i.e. inside the coloured box for each assessed base) and the probability text to draw inside each box.

Usage

```
rasterise_probabilities(
  modification_locations,
  modification_probabilities,
  sequences,
  sequence_text_scaling = c(-0.5, 256),
  sequence_text_rounding = 2
)
```

Arguments

`modification_locations`	`character vector`. One character value for each sequence, storing a condensed string (e.g. `"3,6,9,12"`, produced via `vector_to_string()`) of the indices along the read at which modification was assessed. Indexing starts at 1.
`modification_probabilities`	`character vector`. One character value for each sequence, storing a condensed string (e.g. `"0,128,255,15"`, produced via `vector_to_string()`) of the probability of methylation/modification at each assessed base. Assumed to be Nanopore > SAM style modification stored as an 8-bit integer from 0 to 255, but changing other arguments could make this work on other scales.
`sequences`	`character vector`. One character value for each sequence, storing the actual DNA sequence.
`sequence_text_scaling`	`numeric vector, length 2`. The min and max possible probability values, used to facilitate scaling of the text in each to 0-1. Scaling is implemented as $\frac{p-min}{max}$, so custom scalings (e.g. scaled to 0-9 space) can be implemented by setting this values as required. Set to `c(0, 1)` to not scale at all i.e. print the raw integer probability values. It is recommended to also set `sequence_text_rounding = 0` to print integers as the default value of `2` will result in e.g. `"128.00"`. Set to `c(-0.5, 256)` (default, results in $\frac{p+0.5}{256}$) to scale to the centre of the probability spaces defined by the SAMtools spec, where integer p represents the probability space from $\frac{p}{256}$ to $\frac{p+1}{256}$. This is slightly better at representing the uncertainty compared to `c(0, 255)` as strictly speaking `0` represents the probability space from 0.000 to 0.004 and `255` represents the probability space from 0.996 to 1.000, so scaling them to 0.002 and 0.998 respectively is a more accurate representation of the probability space they each represent. Setting `c(0, 255)` would scale such that 0 is exactly 0.000 and 255 is exactly 1.000, which is not as accurate so it discouraged.

`sequence_text_rounding`
: `integer`. How many decimal places the text drawn in the boxes should be rounded to (defaults to 2). Ignored if `sequence_text_type` is `"sequence"` or `"none"`.

Value

`dataframe`. Dataframe of `x`, `y`, and `value` (i.e. probability to draw).

Examples

```
d <- extract_and_sort_methylation(example_many_sequences)

## Unscaled i.e. integers
rasterise_probabilities(
    d$locations,
    d$probabilities,
    d$sequences,
    sequence_text_scaling = c(0, 1),
    sequence_text_rounding = 0
)

## Scaled to 0-1, 3 dp
rasterize_probabilities(
    d$locations,
    d$probabilities,
    d$sequences,
    sequence_text_scaling = c(-0.5, 256),
    sequence_text_rounding = 3
)

## Default (i.e. scaled to 0-1, 2 dp)
rasterise_probabilities(
    d$locations,
    d$probabilities,
    d$sequences
)
```

`read_fastq` *Read sequence and quality information from FASTQ*

Description

This function simply reads a FASTQ file into a dataframe containing columns for read ID, sequence, and quality scores. Optionally also contains a column of sequence lengths.

See `fastq_quality_scores` for an explanation of quality.

Resulting dataframe can be written back to FASTQ via `write_fastq()`. To read/write a modified FASTQ containing modification information (SAM/BAM MM and ML tags) in the header lines, use `read_modified_fastq()` and `write_modified_fastq()`.

Usage

```
read_fastq(filename = file.choose(), calculate_length = TRUE, strip_at = TRUE)
```

Arguments

`filename`	`character`. The file to be read. Defaults to `file.choose()` to select a file interactively.
`calculate_length`	`logical`. Whether or not `sequence_length` column should be calculated and included.
`strip_at`	`logical`. Boolean value for whether "`@`" characters at the start of read IDs should automatically be stripped if they are present, via `strip_leading_at()`. These "`@`"s tend to be introduced by writing BAM to FASTQ via `samtools fastq` and can cause read IDs to not match between FASTQ data and metadata, causing metadata merging to fail.

Value

`dataframe`. A dataframe with `read`, `sequence`, `quality`, and optionally `sequence_length` columns.

Examples

```
## Locate file
fastq_file <- system.file("extdata",
                          "example_many_sequences_raw.fastq",
                          package = "ggDNAvis")

## View file
for (i in 1:16) {
    cat(readLines(fastq_file)[i], "\n")
}

## Read file to dataframe
read_fastq(fastq_file, calculate_length = FALSE)
read_fastq(fastq_file, calculate_length = TRUE)
```

`read_modified_fastq` *Read modification information from modified FASTQ*

Description

This function reads a modified FASTQ file (e.g. created by `samtools fastq -T MM,ML` from a BAM basecalled with a modification-capable model in Dorado or Guppy) to a dataframe.

By default, the dataframe contains columns for unique read id (`read`), sequence (`sequence`), sequence length (`sequence_length`), quality (`quality`), comma-separated (via `vector_to_string()`) modification types present in each read (`modification_types`), and for each modification type, a column of comma-separated modification locations (`<type>_locations`) and a column of comma-separated modification probabilities (`<type>_probabilities`).

Modification locations are the indices along the read at which modification was assessed e.g. a 3 indicates that the third base in the read was assessed for modifications of the given type. Modification probabilities are the probability that the given modification is present, given as an integer from 0-255 where integer p represents the probability space from $\frac{p}{256}$ to $\frac{p+1}{256}$.

To extract the numbers from these columns as numeric vectors to analyse, use `string_to_vector()` e.g. ``list_of_locations <- lapply(test_01$`C+h?_locations`, string_to_vector)``. Be aware that the SAM modification types often contain special characters, meaning the colname may need to be enclosed in backticks as in this example. Alternatively, use `extract_methylation_from_dataframe()` to create a list of locations, probabilities, and lengths ready for visualisation in `visualise_methylation()`. This works with any modification type extracted in this function, just provide the relevant colname when calling `extract_methylation_from_dataframe()`.

Optionally (by specifying debug = `TRUE`), the dataframe will also contain columns of the raw MM and ML tags (`<MM/ML>_raw`) and of the same tags with the initial label trimmed out (`<MM/ML>_tags`). This is not recommended in most situations but may help with debugging unexpected issues as it contains the raw data exactly from the FASTQ.

Dataframes produced by this function can be written back to modified FASTQ via `write_modified_fastq()`.

Usage

```
read_modified_fastq(filename = file.choose(), strip_at = TRUE, debug = FALSE)
```

Arguments

`filename`	`character`. The file to be read. Defaults to `file.choose()` to select a file interactively.
`strip_at`	`logical`. Boolean value for whether "`@`" characters at the start of read IDs should automatically be stripped if they are present, via `strip_leading_at()`. These "`@`"s tend to be introduced by writing BAM to FASTQ via `samtools fastq`

	and can cause read IDs to not match between FASTQ data and metadata, causing metadata merging to fail.
debug	`logical`. Boolean value for whether the extra `<MM/ML>_tags` and `<MM/ML>_raw` columns should be added to the dataframe. Defaults to `FALSE` as I can't imagine this is often helpful, but the option is provided to assist with debugging.

Value

`dataframe`. Dataframe of modification information, as described above.

Sequences can be visualised with `visualise_many_sequences()` and modification information can be visualised with `visualise_methylation()` (despite the name, any type of information can be visualised as long as it has locations and probabilities columns).

Can be written back to FASTQ via `write_modified_fastq()`.

Examples

```
## Locate file
modified_fastq_file <- system.file("extdata",
                                   "example_many_sequences_raw_modified.fastq",
                                   package = "ggDNAvis")

## View file
for (i in 1:16) {
    cat(readLines(modified_fastq_file)[i], "\n")
}

## Read file to dataframe
read_modified_fastq(modified_fastq_file, debug = FALSE)
read_modified_fastq(modified_fastq_file, debug = TRUE)
```

---

resolve_alias   *Resolve argument value when aliases are used (generic* ggDNAvis *helper)*

Description

See the aliases page for a general explanation of how aliases are used in `ggDNAvis`.

This function takes the name and value for the 'primary' form of an argument (generally British spellings in `ggDNAvis`), the name of an alternative 'alias' form, the dots (unrecognised argument) environment, and the default value of the 'primary' argument.

If the alias has not been used (i.e. the alias is not present in the dots env) or if the 'primary' value has been changed from the default, then the 'primary' value will be returned. (Note that if the alias is present in the dots env and the 'primary' value has been changed from the default, then the updated 'primary' value 'wins' and is returned, but with a warning that explains that both values were set and the 'alias' has been discarded).

If the alias has been used (i.e. the alias is present in the dots env) and the 'primary' value is the default, then the 'alias' value will be returned.

This function is most often used when called by `resolve_alias_map()`.

Usage

```
resolve_alias(primary_name, primary_val, primary_default, alias_name, dots_env)
```

Arguments

`primary_name`	`character`. The usual name of the argument.
`primary_val`	`value`. The value of the argument under its usual name.
`primary_default`	`value`. The default value of the argument under its usual name, used to determine if the primary argument has been explicitly set.
`alias_name`	`character`. An alternative alias name for the argument.
`dots_env`	`environment`. The environment created from the dots list. *WILL BE MODIFIED* by this function - alias is removed if it exists, to allow searching this environment for any unused arguments.

Value

`value`. Either `primary_val` or `alias_val`, depending on the logic above.

Examples

```
low_colour <- "blue" ## e.g. default value from function call
dots_env <- list2env(list(low_color = "pink")) ## e.g. low_color = "pink" set in function call
low_colour <- resolve_alias("low_colour", low_colour, "blue", "low_color", dots_env)
low_colour ## check to see what value was stored
```

`resolve_alias_map` *Process an alias map list (generic* `ggDNAvis` *helper)*

Description

See the aliases page for a general explanation of how aliases are used in `ggDNAvis`.

This function takes an alias map and the environment constructed from non-formal arguments (...) to the calling function, and optionally an environment to function inside, and works through the aliases provided in the map via `resolve_alias()`.

If any arguments were given that aren't in the alias map an error is raised.

Usage

```
resolve_alias_map(alias_map, dots_env, target_env = parent.frame())
```

Arguments

`alias_map`	`list`. A list where each entry takes the name of a formal argument in the calling function, and each value is a list containing `"default"` (the default value of the formal argument) and `"aliases"` (a character vector of all allowed aliases for the formal argument). Aliases are processed in the order given in the character vector, with earlier aliases taking precedence.
`dots_env`	`environment`. The environment created from the dots list. *WILL BE MODIFIED* by this function - alias is removed if it exists, to allow searching this environment for any unused arguments.
`target_env`	`environment`. The environment in which variables should be modified. Generally `parent.frame()` i.e. the calling function.

Value

Nothing (variables are modified within the `target_env`).

Examples

```
## Alias map (from within function code)
alias_map <- list(
   low_colour = list(default = "blue", aliases = c("low_color", "low_col")),
   high_colour = list(default = "red", aliases = c("high_color", "high_col"))
)

## Default values (would come from formal arguments)
low_colour = "blue" ## default
high_colour = "green" ## changed from default

## Extra arguments provided by name
dots_env <- list2env(list("low_col" = "black", "low_color" = "white", "high_color" = "orange"))

## Process
resolve_alias_map(alias_map, dots_env)

## See values
print(low_colour)
print(high_colour)
```

`reverse_complement` *Reverse complement a DNA/RNA sequence (generic* `ggDNAvis` *helper)*

Description

This function takes a string/character representing a DNA/RNA sequence and returns the reverse complement. Either DNA (`A/C/G/T`) or RNA (`A/C/G/U`) input is accepted.

By default, output is DNA (so `A` is reverse-complemented to `T`), but it can be set to output RNA (so `A` is reverse-complemented to `U`).

Alternatively, if `output_mode` is set to `"reverse_only"` then the sequence will be reversed as-is without being complemented. Note that this also skips sequence validation, meaning any string can be reversed even if it contains non-A/C/G/T/U characters.

Usage

```
reverse_complement(sequence, output_mode = "DNA")
```

Arguments

`sequence`	`character`. A DNA/RNA sequence (`A/C/G/T/U`) to be reverse-complemented. No other characters allowed. Only one sequence allowed.
`output_mode`	`character`. `"DNA"` (default) or `"RNA"` to determine whether `A` should be reverse-complemented to `T` or `U` respectively, or `"reverse_only"` to reverse the order of the characters without complementing.

Value

`character`. The reverse-complement of the input sequence.

Examples

```
reverse_complement("ATGCTAG")
reverse_complement("ATGCTAG", output_mode = "reverse_only")
reverse_complement("UUAUUAGC", output_mode = "RNA")
reverse_complement("AcGtU", output_mode = "DNA")
reverse_complement("aCgTU", output_mode = "RNA")
```

`reverse_locations_if_needed`
Reverse modification locations if needed (`merge_methylation_with_metadata()`* helper)*

Description

This function takes a vector of condensed modification locations/indices (e.g. `c("3,6,9,12", "1,4,7,10")`), a vector of directions (which must all be either `"forward"` or `"reverse"`, *not* case-sensitive), and a vector of sequence lengths (integers).

Returns a vector of condensed locations where reads that were originally forward are unchanged, and reads that were originally reverse are flipped to now be forward.

Optionally, a numerical offset can be set. If this is left at `0` (the default value), then a CpG assessed for methylation would be reverse-complemented to a CG with the modification information ascribed to the G (as the G is at the location where the actual modified C was on the other strand). However, setting the offset to `1` would shift all of the modification indices by 1 such that the modification is now ascribed to the C of the reverse-strand CG. This is beneficial for visualising the modifications as it ensures consistency between originally-forward and originally-reverse strands by making the modification score associated with each CpG site always be located at the C, but may be misleading for quantitative analysis. Setting the offset to anything other than `0` or `1` should work but may be biologically misleading, so produces a warning.

Called by `merge_methylation_with_metadata()` to create a forward dataset, alongside `reverse_sequence_if_needed()`, `reverse_quality_if_needed()`, and `reverse_probabilities_if_needed()`.

Example:

Forward sequence, with indices of Cs in CpGs numbered:

```
C C C A G G C G G C G G C G A C C G A
            7     10    13      17
```

(length = 19, locations = `"7,10,13,17"`, CpGs = 7-8, 10-11, 13-14, 17-18)

Reverse sequence, with indices of C in CpGs numbered:

```
T C G G T C G C C G C C G C C T G G G
  2       6     9     12
```

(length = 19, locations = `"2,6,9,12"`, CpGs = 2-3, 6-7, 9-10, 12-13)

As CG reverse-complements to itself, each CpG site has a 1:1 correspondence with a CpG site in the reverse strand. Many methylation calling models assess C-methylation at the C of each CpG. To map the locations from C to C, we take `19 - <index>` such that `"7,10,13,17"` becomes `"12,9,6,2"` and `"2,6,9,12"` becomes `"17,13,10,7"`. The symmetry of CpGs means mapping from C to C is also symmetric. *This is achieved by setting* `offset = 1`*, as mapping C to C involves shifting position by 1.*

Conversely, to map the locations from C to G (i.e. preserving the actual location of each modification, which is required if assessed locations are non-symmetric/don't reverse-complement to themselves like CpGs do), we take `20 - <index>` such that `"7,10,13,17"` becomes `"13,10,7,3"` i.e. the indices of the Gs in CpGs in the reverse sequence. Likewise `"2,6,9,12"` becomes `"18,14,11,8"` i.e. the indices of the Gs in CpGs in the forward sequence. *This is achieved by setting* `offset = 0`*, as mapping C to G preserves the actual original position at which each modification was assessed, but changes the base to its complement.*

In general, new locations are calculated as `(<length> + 1 - <offset>) - <index>`. Of course, output locations are reversed before returning so that they all return in ascending order, as is standard for all location vectors/strings.

If wanting to write reversed sequences to FASTQ via `write_modified_fastq()`, locations must be symmetric (e.g. CpG) and offset must be set to 1. Asymmetric locations are impossible to write to modified FASTQ once reversed because then e.g. cytosine methylation will be assessed at guanines, which SAMtools can't account for. Symmetrically reversing CpGs via `offset = 1` is the only way to fix this.

Usage

```
reverse_locations_if_needed(
  locations_vector,
  direction_vector,
  length_vector,
  offset = 0
)
```

Arguments

`locations_vector`
: `character vector`. The locations to be reversed for each sequence/read. Each read should have one character value, representing a comma-separated list of indices at which modification was assessed along the read e.g. `"3,6,9,12"` for all the `C`s in `GGCGGCGGCGGC`.

  These comma-separated characters/strings can be produced from numeric vectors via `vector_to_string()` and converted back to vectors via `string_to_vector()`.

`direction_vector`
: `character vector`. Whether each sequence is forward or reverse. Must contain only `"forward"` and `"reverse"`, but is not case sensitive. Must be the same length as `locations_vector` and `length_vector`.

`length_vector`
: `integer vector`. The length of each sequence. Needed for reversing locations as locations are stored relative to the start of the read i.e. relative to the end of the reverse read. Must be the same length as `locations_vector` and `direction_vector`.

`offset`
: `integer`. How much locations should be shifted by. Defaults to `0`. This is important because if a CpG is assessed for methylation at the C, then reverse complementing it will give a methylation score at the G on the reverse-complemented strand. This is the most biologically accurate, but for visualising methylation it may be desired to shift the locations by 1 i.e. to correspond with the C in the reverse-complemented CpG rather than the G, which allows for consistent visualisation between forward and reverse strands. Setting (integer) values other than `0` or `1` will work, but may be biologically misleading so it is not recommended.

Value

`character vector`. A vector of all forward versions of the input locations vector.

Examples

```
reverse_locations_if_needed(
    locations_vector = c("7,10,13,17", "2,6,9,12"),
    direction_vector = c("forward", "reverse"),
    length_vector = c(19, 19),
    offset = 0
)
```

```
reverse_locations_if_needed(
    locations_vector = c("7,10,13,17", "2,6,9,12"),
    direction_vector = c("forward", "reverse"),
    length_vector = c(19, 19),
    offset = 1
)
```

`reverse_probabilities_if_needed` *Reverse modification probabilities if needed (`merge_methylation_with_metadata()` helper)*

Description

This function takes a vector of condensed modification probabilities (e.g. `c("128,0,63,255", "3,78,1")` and a vector of directions (which must all be either `"forward"` or `"reverse"`, *not* case-sensitive), and returns a vector of condensed modification probabilities where those that were originally forward are unchanged, and those that were originally reverse are flipped to now be forward.

Called by `merge_methylation_with_metadata()` to create a forward dataset, alongside `reverse_sequence_if_needed()`, `reverse_quality_if_needed()`, and `reverse_locations_if_needed()`.

Usage

```
reverse_probabilities_if_needed(probabilities_vector, direction_vector)
```

Arguments

`probabilities_vector`
: `character vector`. The probabilities to be reversed for each sequence/read. Each read should have one character value, representing a comma-separated list of the modification probabilities for each assessed base along the read e.g. `"230,7,64,145"`. In most situations these will be 8-bit integers from 0 to 255, but this function will work on any comma-separated values.

  These comma-separated characters/strings can be produced from numeric vectors via `vector_to_string()` and converted back to vectors via `string_to_vector()`.

`direction_vector`
: `character vector`. Whether each sequence is forward or reverse. Must contain only `"forward"` and `"reverse"`, but is not case sensitive. Must be the same length as `probabilities_vector`.

Value

`character vector`. A vector of all forward versions of the input probabilities vector.

Examples

```
reverse_probabilities_if_needed(
    probabilities_vector = c("100,200,50", "100,200,50"),
    direction_vector = c("forward", "reverse")
)
```

`reverse_quality_if_needed` *Reverse qualities if needed (`merge_methylation_with_metadata()` helper)*

Description

This function takes a vector of FASTQ qualities and a vector of directions (which must all be either `"forward"` or `"reverse"`, *not* case-sensitive) and returns a vector of forward qualities.

Qualities of reads that were forward to begin with are unchanged, while qualities of reads that were reverse are now flipped to give the corresponding forward quality scores.

Called by `merge_methylation_with_metadata()` to create a forward dataset, alongside `reverse_sequence_if_needed()`, `reverse_locations_if_needed()`, and `reverse_probabilities_if_needed()`.

Usage

```
reverse_quality_if_needed(quality_vector, direction_vector)
```

Arguments

`quality_vector`	`character vector`. The qualities to be reversed. See `fastq_quality_scores` for an explanation of quality scores.
`direction_vector`	`character vector`. Whether each sequence is forward or reverse. Must contain only `"forward"` and `"reverse"`, but is not case sensitive. Must be the same length as `sequence_vector`.

Value

`character vector`. A vector of all forward versions of the input quality vector.

Examples

```
reverse_quality_if_needed(
    quality_vector = c("#^$&$*", "#^$&$*"),
    direction_vector = c("reverse", "forward")
)
```

reverse_sequence_if_needed

Reverse sequences if needed (`merge_methylation_with_metadata()` helper)

Description

This function takes a vector of DNA/RNA sequences and a vector of directions (which must all be either `"forward"` or `"reverse"`, *not* case-sensitive) and returns a vector of forward DNA/RNA sequences.

Sequences in the vector that were forward to begin with are unchanged, while sequences that were reverse are reverse-complemented via `reverse_complement()` to produce the forward sequence.

Called by `merge_methylation_with_metadata()` to create a forward dataset, alongside `reverse_quality_if_needed()`, `reverse_locations_if_needed()` and `reverse_probabilities_if_needed()`.

Usage

```
reverse_sequence_if_needed(
  sequence_vector,
  direction_vector,
  output_mode = "DNA"
)
```

Arguments

`sequence_vector`	`character vector`. The DNA or RNA sequences to be reversed, e.g. `c("ATCG", "GGCGGC", "AUUAUA")`. Accepts DNA, RNA, or mixed input.
`direction_vector`	`character vector`. Whether each sequence is forward or reverse. Must contain only `"forward"` and `"reverse"`, but is not case sensitive. Must be the same length as `sequence_vector`.
`output_mode`	`character`. Whether reverse-complemented sequences should be converted to DNA (i.e. A complements to T), RNA (i.e. A complements to U), or not complemented at all. Must be `"DNA"` (default), `"RNA"`, or `"reverse_only"` to reverse sequence order without complementing. *Only affects reverse-complemented sequences. Sequences that were forward to begin with are not altered.*

Value

`character vector`. A vector of all forward versions of the input sequence vector.

Examples

```
reverse_sequence_if_needed(
    sequence_vector = c("TAAGGC", "TAAGGC"),
    direction_vector = c("reverse", "forward")
)

reverse_sequence_if_needed(
    sequence_vector = c("UAAGGC", "UAAGGC"),
    direction_vector = c("reverse", "forward"),
    output_mode = "RNA"
)

reverse_sequence_if_needed(
    sequence_vector = c("TAAGGC", "TAAGGC"),
    direction_vector = c("reverse", "forward"),
    output_mode = "reverse_only"
)
```

`sequence_colour_palettes` *Colour palettes for sequence visualisations*

Description

`sequence_color_palettes` and `sequence_col_palettes` are aliases for `sequence_colour_palettes` - see aliases.

A collection of colour palettes for use with `visualise_single_sequence()` and `visualise_many_sequences()`. Each is a character vector of 4 colours, corresponding to A, C, G, and T/U in that order.

To use inside the visualisation functions, set `sequence_colours = sequence_colour_palettes$<palette_name>`

Generation code is available at `data-raw/sequence_colour_palettes.R`

Usage

```
sequence_colour_palettes
```

Format

`sequence_colour_palettes`**:**
A list of 6 length-4 character vectors

ggplot_style The shades of red, green, blue, and purple that `ggplot2::ggplot()` uses by default for a 4-way discrete colour scheme.

Values: `c("#F8766D", "#7CAE00", "#00BFC4", "#C77CFF")`

bright_pale Bright yellow, green, blue, and red in lighter pastel-like tones.

Values: `c("#FFDD00", "#40C000", "#00A0FF", "#FF4E4E")`

bright_pale2 Bright yellow, green, blue, and red in lighter pastel-like tones. The green (for C) is slightly lighter than bright_pale.

Values: `c("#FFDD00", "#30EC00", "#00A0FF", "#FF4E4E")`

bright_deep Bright orange, green, blue, and red in darker, richer tones.

Values: `c("#FFAA00", "#00BC00", "#0000DC", "#FF1E1E")`

sanger Green, blue, black, and red similar to a traditional Sanger sequencing readout.

Values: `c("#00B200", "#0000FF", "#000000", "#FF0000")`

accessible Light green, dark green, dark blue, and light blue as suggested by colorbrewer2.org for a 4-qualitative-category colourblind-safe palette.

Values: `c("#B2DF8A", "#33A02C", "#1F78B4", "#A6CEE3")`

Examples

```
## ggplot_style:
visualise_single_sequence(
    "ACGT",
    sequence_colours = sequence_colour_palettes$ggplot_style,
    index_annotation_interval = 0
)

## bright_pale:
visualise_single_sequence(
    "ACGT",
    sequence_colours = sequence_colour_palettes$bright_pale,
    index_annotation_interval = 0
)

## bright_pale2:
visualise_single_sequence(
    "ACGT",
    sequence_colours = sequence_colour_palettes$bright_pale2,
    index_annotation_interval = 0
)

## bright_deep:
visualise_single_sequence(
    "ACGT",
    sequence_colours = sequence_colour_palettes$bright_deep,
    sequence_text_colour = "white",
    index_annotation_interval = 0
)

## sanger:
visualise_single_sequence(
    "ACGT",
    sequence_colours = sequence_colour_palettes$sanger,
    sequence_text_colour = "white",
    index_annotation_interval = 0
)

## accessible:
visualise_single_sequence(
    "ACGT",
    sequence_colours = sequence_colour_palettes$accessible,
    sequence_text_colour = "black",
    index_annotation_interval = 0
)
```

string_to_vector — *Split a* `","`*-joined string back to a vector (generic* ggDNAvis *helper)*

Description

Takes a string (character) produced by `vector_to_string()` and recreates the vector.

Note that if a vector of multiple strings is input (e.g. `c("1,2,3", "9,8,7")`) the output will be a single concatenated vector (e.g. `c(1, 2, 3, 9, 8, 7)`).

If the desired output is a list of vectors, try `lapply()` e.g. `lapply(c("1,2,3", "9,8,7"), string_to_vector)` returns `list(c(1, 2, 3), c(9, 8, 7))`.

Usage

```
string_to_vector(string, type = "numeric", sep = ",")
```

Arguments

`string`	`character`. A comma-separated string (e.g. `"1,2,3"`) to convert back to a vector.
`type`	`character`. The type of the vector to be returned i.e. `"numeric"` (default), `"character"`, or `"logical"`.
`sep`	`character`. The character used to separate values in the string. Defaults to `","`. *Do not set to anything that might occur within one of the values.*

Value

`<type> vector`. The resulting vector (e.g. `c(1, 2, 3)`).

Examples

```
## String to numeric vector (default)
string_to_vector("1,2,3,4")
string_to_vector("1,2,3,4", type = "numeric")
string_to_vector("1;2;3;4", sep = ";")

## String to character vector
string_to_vector("A,B,C,D", type = "character")

## String to logical vector
```

```
string_to_vector("TRUE FALSE TRUE", type = "logical", sep = " ")

## By default, vector inputs are concatenated
string_to_vector(c("1,2,3", "4,5,6"))

## To create a list of vector outputs, use lapply()
lapply(c("1,2,3", "4,5,6"), string_to_vector)
```

`strip_leading_at` *Strip leading `@` from character vector*

Description

This function removes a single leading `@` character from each element of a character vector when present. This is intended to deal with SAMtools > FASTQ translation often prefixing read IDs with an "`@`", which can result in read ID mismatches and metadata merging fails.

Usage

```
strip_leading_at(string)
```

Arguments

`string`	`character vector`. A vector (e.g. read IDs) to strip of a single leading "`@`" each wherever one is present.

Value

`character vector`. The same string but with one "`@`" removed from each element that started with one.

Examples

```
strip_leading_at(c("read_1", "@read_2", "@@read_3", "", NA, NULL))
```

`vector_to_string` *Join a vector into a comma-separated string (generic `ggDNAvis` helper)*

Description

Takes a vector and condenses it into a single string by joining items with `","`. Reversed by `string_to_vector()`.

Usage

```
vector_to_string(vector, sep = ",")
```

Arguments

`vector`	`vector`. A vector (e.g. `c(1,2,3)`) to convert to a string.
`sep`	`character`. The character used to separate values in the string. Defaults to `","`. *Do not set to anything that might occur within one of the values.*

Value

`character`. The same vector but as a comma-separated string (e.g. `"1,2,3"`).

Examples

```
vector_to_string(c(1, 2, 3, 4))
vector_to_string(c("These", "are", "some", "words"))
vector_to_string(3:5, sep = ";")
```

`visualise_many_sequences` *Visualise many DNA/RNA sequences*

Description

`visualize_many_sequences()` is an alias for `visualise_many_sequences()` - see aliases.

This function takes a vector of DNA/RNA sequences (each sequence can be any length and they can be different lengths), and plots each sequence as base-coloured squares along a single line. Setting `filename` allows direct export of a png image with the correct dimensions to make every base a perfect square. Empty strings (`""`) within the vector can be utilised as blank spacing lines. Colours and pixels per square when exported are configurable.

Usage

```
visualise_many_sequences(
  sequences_vector,
  ...,
  sequence_colours = sequence_colour_palettes$ggplot_style,
  background_colour = "white",
  margin = 0.5,
  sequence_text_colour = "black",
  sequence_text_size = 16,
  index_annotation_lines = c(1),
  index_annotation_colour = "darkred",
  index_annotation_size = 12.5,
  index_annotation_interval = 15,
  index_annotations_above = TRUE,
  index_annotation_vertical_position = 1/3,
  index_annotation_full_line = TRUE,
  index_annotation_always_first_base = TRUE,
  index_annotation_always_last_base = TRUE,
  outline_colour = "black",
  outline_linewidth = 3,
  outline_join = "mitre",
  return = TRUE,
  filename = NA,
  force_raster = FALSE,
  render_device = ragg::agg_png,
  pixels_per_base = 100,
  monitor_performance = FALSE
)
```

Arguments

`sequences_vector`
: `character vector`. The sequences to visualise, often created from a dataframe via `extract_and_sort_sequences()`. E.g. `c("GGCGGC", "", "AGCTAGCTA")`.

`...`
: Used to recognise aliases e.g. American spellings or common misspellings - see aliases. If any American spellings do not work, please make a bug report at `https://github.com/ejade42/ggDNAvis/issues`.

`sequence_colours`
: `character vector`, length 4. A vector indicating which colours should be used for each base. In order: `c(A_colour, C_colour, G_colour, T/U_colour)`.

  Defaults to red, green, blue, purple in the default shades produced by ggplot with 4 colours, i.e. `c("#F8766D", "#7CAE00", "#00BFC4", "#C77CFF")`, accessed via `sequence_colour_palettes$ggplot_style`.

`background_colour`
: `character`. The colour the background should be drawn (defaults to white).

`margin`
: `numeric`. The size of the margin relative to the size of each base square. Defaults to `0.5` (half the side length of each base square).

  Note that index annotations can require a minimum margin size at the top or bottom if present above the first/below the last row. This is handled automatically but can mean the top/bottom margin is sometimes larger than the `margin` setting.

  Very small margins ($\leq$0.25) may cause thick outlines to be cut off at the edges of the plot. Recommended to either use a wider margin or a smaller `outline_linewidth`.

`sequence_text_colour`
: `character`. The colour of the text within the bases (e.g. colour of "A" letter within boxes representing adenosine bases). Defaults to black.

`sequence_text_size`
: `numeric`. The size of the text within the bases (e.g. size of "A" letter within boxes representing adenosine bases). Defaults to `16`. Set to `0` to hide sequence text (show box colours only).

`index_annotation_lines`
: `integer vector`. The lines (i.e. elements of `sequences_vector`) that should have their base incides annotated. 1-indexed e.g. `c(1, 10)` would annotate the first and tenth elements of `sequences_vector`.

  Extra lines are inserted above or below (depending on `index_annotations_above`) the selected lines - note that the line numbers come from `sequences_vector`, so are unaffected by these insertions.

  Setting to `NA` disables index annotations (and prevents adding additional blank lines). Defaults to `c(1)` i.e. first sequence is annotated.

  Note: `visualise_methylation()` and `visualise_many_sequences()` accept `NA`, `NULL`, `numeric(0)`, `0`, and `FALSE` as meaning "annotations off".

`index_annotation_colour`
: `character`. The colour of the little numbers underneath indicating base index (e.g. colour of "15" label under the 15th base). Defaults to dark red.

`index_annotation_size`
: `numeric`. The size of the little number underneath indicating base index (e.g. size of "15" label under the 15th base). Defaults to `12.5`.

  Setting to `0` disables index annotations (and prevents adding additional blank lines).

`index_annotation_interval`
: `integer`. The frequency at which numbers should be placed underneath indicating base index, starting counting from the leftmost base. Defaults to `15` (every 15 bases along each row).

  Setting to `0` disables index annotations (and prevents adding additional blank lines).

`index_annotations_above`
: `logical`. Whether index annotations should go above (`TRUE`, default) or below (`FALSE`) each line of sequence.

`index_annotation_vertical_position`
: `numeric`. How far annotation numbers should be rendered above (if `index_annotations_above = TRUE`) or below (if `index_annotations_above = FALSE`) each base. Defaults to `1/3`.

  Not recommended to change at all. Strongly discouraged to set below 0 or above 1.

`index_annotation_full_line`
: `logical`. Whether index annotations should continue to the end of the longest sequence (`TRUE`, default) or should only continue as far as each selected line does (`FALSE`).

`index_annotation_always_first_base`
: `logical`. Whether to force the first base in each line to always be annotated regardless of whether it occurs at the `index_annotation_interval`. Defaults to `TRUE`.

`index_annotation_always_last_base`
: `logical`. Whether to force the last base in each line to always be annotated regardless of whether it occurs at the `index_annotation_interval`. Defaults to `TRUE`.

`outline_colour`
: `character`. The colour of the box outlines. Defaults to black.

`outline_linewidth`
: `numeric`. The linewidth of the box outlines. Defaults to `3`. Set to `0` to disable box outlines.

`outline_join`
: `character`. One of `"mitre"`, `"round"`, or `"bevel"` specifying how outlines should be joined at the corners of boxes. Defaults to `"mitre"`. It would be unusual to need to change this.

`return`
: `logical`. Boolean specifying whether this function should return the ggplot object, otherwise it will return `invisible(NULL)`. Defaults to `TRUE`.

`filename`
: `character`. Filename to which output should be saved. If set to `NA` (default), no file will be saved. Recommended to end with `".png"`, but can change if render device is changed.

`force_raster`
: `logical`. Boolean specifying whether `ggplot2::geom_raster()` should be used even if it will remove text and outlines. Defaults to `FALSE`.

  To make the detailed plots with box outlines, sequence text, and index annotations, `ggplot2::geom_tile()` is used. However, `geom_tile` is slower for huge datasets, so there is an option to use `geom_raster` instead. `geom_raster` does not support box outlines, sequence text, or index annotations, but is much faster if only the colours are wanted.

  `geom_raster` is automatically used if it will not change the plot (i.e. if all extraneous elements are already off), but can be forced using this argument.

`render_device`
: `function/character`. Device to use when rendering. See `ggplot2::ggsave()` documentation for options. Defaults to `ragg::agg_png`. Can be set to `NULL` to infer from file extension, but results may vary between systems.

`pixels_per_base`
: `integer`. How large each box should be in pixels, if file output is turned on via setting `filename`. Corresponds to dpi of the exported image. Defaults to `100`.

  Large values (e.g. 100) are required to render small text properly. Small values (e.g. 20) will work when sequence/annotation text is off, and very small values (e.g. 10) will work when sequence/annotation text and outlines are all off.

`monitor_performance`
: `logical`. Boolean specifying whether verbose performance monitoring should be messaged to console. Defaults to `FALSE`.

Value

A ggplot object containing the full visualisation, or `invisible(NULL)` if `return = FALSE`. It is often more useful to use `filename = "myfilename.png"`, because then the visualisation is exported at the correct aspect ratio.

Examples

```
## Create sequences vector
sequences <- extract_and_sort_sequences(example_many_sequences)

## Visualise example_many_sequences with all defaults
## This looks ugly because it isn't at the right scale/aspect ratio
visualise_many_sequences(sequences)

## Export with all defaults rather than returning
visualise_many_sequences(
    sequences,
    filename = "example_vms_01.png",
    return = FALSE
)
## View exported image
image <- png::readPNG("example_vms_01.png")
unlink("example_vms_01.png")
grid::grid.newpage()
grid::grid.raster(image)

## Export while customising appearance
visualise_many_sequences(
    sequences,
    filename = "example_vms_02.png",
    return = FALSE,
    sequence_colours = sequence_colour_palettes$bright_pale,
    sequence_text_colour = "white",
    index_annotation_interval = 3,
    index_annotation_lines = 1:51,
    index_annotation_full_line = FALSE,
    index_annotation_always_first_base = FALSE,
    index_annotation_always_last_base = FALSE,
    background_colour = "lightgrey",
    outline_linewidth = 0,
    margin = 0
)
## View exported image
image <- png::readPNG("example_vms_02.png")
unlink("example_vms_02.png")
grid::grid.newpage()
grid::grid.raster(image)
```

visualise_methylation *Visualise methylation probabilities for many DNA sequences*

Description

`visualize_methylation()` is an alias for `visualise_methylation()` - see aliases.

This function takes vectors of modifications locations, modification probabilities, and sequence lengths (e.g. created by `extract_and_sort_methylation()`) and visualises the probability of methylation (or other modification) across each read.

Assumes that the three main input vectors are of equal length n and represent n sequences (e.g. Nanopore reads), where `locations` are the indices along each read at which modification was assessed, `probabilities` are the probability of modification at each assessed site, and `lengths` are the lengths of each sequence.

For each sequence, renders non-assessed (e.g. non-CpG) bases as `other_bases_colour`, renders background (including after the end of the sequence) as `background_colour`, and renders assessed bases on a linear scale from `low_colour` to `high_colour`.

Clamping means that the endpoints of the colour gradient can be set some distance into the probability space e.g. with Nanopore > SAM probability values from 0-255, the default is to render 0 as fully blue (`#0000FF`), 255 as fully red (`#FF0000`), and values in between linearly interpolated. However, clamping with `low_clamp = 100` and `high_clamp = 200` would set *all probabilities up to 100* as fully blue, *all probabilities 200 and above* as fully red, and linearly interpolate only over the `100-200` range.

A separate scalebar plot showing the colours corresponding to each probability, with any/no clamping values, can be produced via `visualise_methylation_colour_scale()`.

Usage

```
visualise_methylation(
  modification_locations,
  modification_probabilities,
  sequences,
  ...,
  low_colour = "blue",
  high_colour = "red",
  low_clamp = 0,
  high_clamp = 255,
  background_colour = "white",
  other_bases_colour = "grey",
  sequence_text_type = "none",
  sequence_text_scaling = c(-0.5, 256),
  sequence_text_rounding = 2,
  sequence_text_colour = "black",
  sequence_text_size = 16,
  index_annotation_lines = c(1),
  index_annotation_colour = "darkred",
  index_annotation_size = 12.5,
  index_annotation_interval = 15,
  index_annotations_above = TRUE,
  index_annotation_vertical_position = 1/3,
  index_annotation_full_line = TRUE,
  index_annotation_always_first_base = TRUE,
  index_annotation_always_last_base = TRUE,
  outline_colour = "black",
  outline_linewidth = 3,
  outline_join = "mitre",
  modified_bases_outline_colour = NA,
  modified_bases_outline_linewidth = NA,
  modified_bases_outline_join = NA,
  other_bases_outline_colour = NA,
  other_bases_outline_linewidth = NA,
  other_bases_outline_join = NA,
  margin = 0.5,
  return = TRUE,
  filename = NA,
  force_raster = FALSE,
  render_device = ragg::agg_png,
  pixels_per_base = 100,
  monitor_performance = FALSE
)
```

Arguments

`modification_locations`
: `character vector`. One character value for each sequence, storing a condensed string (e.g. `"3,6,9,12"`, produced via `vector_to_string()`) of the indices along the read at which modification was assessed. Indexing starts at 1.

`modification_probabilities`
: `character vector`. One character value for each sequence, storing a condensed string (e.g. `"0,128,255,15"`, produced via `vector_to_string()`) of the probability of methylation/modification at each assessed base.

  Assumed to be Nanopore > SAM style modification stored as an 8-bit integer from 0 to 255, but changing other arguments could make this work on other scales.

`sequences`
: `character vector`. One character value for each sequence, storing the actual DNA sequence.

`...`
: Used to recognise aliases e.g. American spellings or common misspellings - see aliases. If any American spellings do not work, please make a bug report at `https://github.com/ejade42/ggDNAvis/issues`.

`low_colour`
: `character`. The colour that should be used to represent minimum probability of methylation/modification (defaults to blue).

`high_colour`
: `character`. The colour that should be used to represent maximum probability of methylation/modification (defaults to red).

`low_clamp`
: `numeric`. The minimum probability below which all values are coloured `low_colour`. Defaults to `0` (i.e. no clamping). To specify a proportion probability in 8-bit form, multiply by 255 e.g. to low-clamp at 30% probability, set this to `0.3*255`.

`high_clamp`
: `numeric`. The maximum probability above which all values are coloured `high_colour`. Defaults to `255` (i.e. no clamping, assuming Nanopore > SAM style modification calling where probabilities are 8-bit integers from 0 to 255).

`background_colour`
: `character`. The colour the background should be drawn (defaults to white).

`other_bases_colour`
: `character`. The colour non-assessed (e.g. non-CpG) bases should be drawn (defaults to grey).

`sequence_text_type`
: `character`. What type of text should be drawn in the boxes. One of `"sequence"` (to draw the base sequence in the boxes, similar to `visualise_many_sequences()`), `"probability"` (to draw the numerical probability of methylation in each assessed box, optionally scaled via `sequence_text_scaling`), or `"none"` (default, to draw the boxes only with no text).

`sequence_text_scaling`
: `numeric vector, length 2`. The min and max possible probability values, used to facilitate scaling of the text in each to 0-1. Scaling is implemented as $\frac{p-min}{max}$, so custom scalings (e.g. scaled to 0-9 space) can be implemented by setting this values as required.

  Set to `c(0, 1)` to not scale at all i.e. print the raw integer probability values. It is recommended to also set `sequence_text_rounding = 0` to print integers as the default value of `2` will result in e.g. `"128.00"`.

  Set to `c(-0.5, 256)` (default, results in $\frac{p+0.5}{256}$) to scale to the centre of the probability spaces defined by the SAMtools spec, where integer p represents the probability space from $\frac{p}{256}$ to $\frac{p+1}{256}$. This is slightly better at representing the uncertainty compared to `c(0, 255)` as strictly speaking `0` represents the probability space from 0.000 to 0.004 and `255` represents the probability space from 0.996 to 1.000, so scaling them to 0.002 and 0.998 respectively is a more accurate representation of the probability space they each represent. Setting `c(0, 255)` would scale such that 0 is exactly 0.000 and 255 is exactly 1.000, which is not as accurate so it discouraged.

`sequence_text_rounding`
: `integer`. How many decimal places the text drawn in the boxes should be rounded to (defaults to `2`). Ignored if `sequence_text_type` is `"sequence"` or `"none"`.

`sequence_text_colour`
: `character`. The colour of the text within the bases (e.g. colour of "A" letter within boxes representing adenosine bases). Defaults to black.

`sequence_text_size`
: `numeric`. The size of the text within the bases (e.g. size of "A" letter within boxes representing adenosine bases). Defaults to `16`. Set to `0` to hide sequence text (show box colours only).

`index_annotation_lines`
: `integer vector`. The lines (i.e. elements of `sequences_vector`) that should have their base incides annotated. 1-indexed e.g. `c(1, 10)` would annotate the first and tenth elements of `sequences_vector`.

  Extra lines are inserted above or below (depending on `index_annotations_above`) the selected lines - note that the line numbers come from `sequences_vector`, so are unaffected by these insertions.

Setting to NA disables index annotations (and prevents adding additional blank lines). Defaults to c(1) i.e. first sequence is annotated.

Note: visualise_methylation() and visualise_many_sequences() accept NA, NULL, numeric(0), 0, and FALSE as meaning "annotations off".

index_annotation_colour
: character. The colour of the little numbers underneath indicating base index (e.g. colour of "15" label under the 15th base). Defaults to dark red.

index_annotation_size
: numeric. The size of the little number underneath indicating base index (e.g. size of "15" label under the 15th base). Defaults to 12.5.

  Setting to 0 disables index annotations (and prevents adding additional blank lines).

index_annotation_interval
: integer. The frequency at which numbers should be placed underneath indicating base index, starting counting from the leftmost base. Defaults to 15 (every 15 bases along each row).

  Setting to 0 disables index annotations (and prevents adding additional blank lines).

index_annotations_above
: logical. Whether index annotations should go above (TRUE, default) or below (FALSE) each line of sequence.

index_annotation_vertical_position
: numeric. How far annotation numbers should be rendered above (if index_annotations_above = TRUE) or below (if index_annotations_above = FALSE) each base. Defaults to 1/3.

  Not recommended to change at all. Strongly discouraged to set below 0 or above 1.

index_annotation_full_line
: logical. Whether index annotations should continue to the end of the longest sequence (TRUE, default) or should only continue as far as each selected line does (FALSE).

index_annotation_always_first_base
: logical. Whether to force the first base in each line to always be annotated regardless of whether it occurs at the index_annotation_interval. Defaults to TRUE.

index_annotation_always_last_base
: logical. Whether to force the last base in each line to always be annotated regardless of whether it occurs at the index_annotation_interval. Defaults to TRUE.

outline_colour
: character. The colour of the box outlines. Defaults to black.

outline_linewidth
: numeric. The linewidth of the box outlines. Defaults to 3. Set to 0 to disable box outlines.

`outline_join`	`character`. One of `"mitre"`, `"round"`, or `"bevel"` specifying how outlines should be joined at the corners of boxes. Defaults to `"mitre"`. It would be unusual to need to change this.
`modified_bases_outline_colour`	`character`. If `NA` (default), inherits from `outline_colour`. If not `NA`, overrides `outline_colour` for modification-assessed bases only.
`modified_bases_outline_linewidth`	`numeric`. If `NA` (default), inherits from `outline_linewidth`. If not `NA`, overrides `outline_linewidth` for modification-assessed bases only.
`modified_bases_outline_join`	`character`. If `NA` (default), inherits from `outline_join`. If not `NA`, overrides `outline_join` for modification-assessed bases only.
`other_bases_outline_colour`	`character`. If `NA` (default), inherits from `outline_colour`. If not `NA`, overrides `outline_colour` for non-modification-assessed bases only.
`other_bases_outline_linewidth`	`numeric`. If `NA` (default), inherits from `outline_linewidth`. If not `NA`, overrides `outline_linewidth` for non-modification-assessed bases only.
`other_bases_outline_join`	`character`. If `NA` (default), inherits from `outline_join`. If not `NA`, overrides `outline_join` for non-modification-assessed bases only.
`margin`	`numeric`. The size of the margin relative to the size of each base square. Defaults to `0.5` (half the side length of each base square). Note that index annotations can require a minimum margin size at the top or bottom if present above the first/below the last row. This is handled automatically but can mean the top/bottom margin is sometimes larger than the `margin` setting. Very small margins ($\leq$0.25) may cause thick outlines to be cut off at the edges of the plot. Recommended to either use a wider margin or a smaller `outline_linewidth`.
`return`	`logical`. Boolean specifying whether this function should return the ggplot object, otherwise it will return `invisible(NULL)`. Defaults to `TRUE`.
`filename`	`character`. Filename to which output should be saved. If set to `NA` (default), no file will be saved. Recommended to end with `".png"`, but can change if render device is changed.
`force_raster`	`logical`. Boolean specifying whether `ggplot2::geom_raster()` should be used even if it will remove text and outlines. Defaults to `FALSE`. To make the detailed plots with box outlines, sequence text, and index annotations, `ggplot2::geom_tile()` is used. However, `geom_tile` is slower for huge datasets, so there is an option to use `geom_raster` instead. `geom_raster` does not support box outlines, sequence text, or index annotations, but is much faster if only the colours are wanted. `geom_raster` is automatically used if it will not change the plot (i.e. if all extraneous elements are already off), but can be forced using this argument.

`render_device`	`function/character`. Device to use when rendering. See `ggplot2::ggsave()` documentation for options. Defaults to `ragg::agg_png`. Can be set to `NULL` to infer from file extension, but results may vary between systems.
`pixels_per_base`	`integer`. How large each box should be in pixels, if file output is turned on via setting `filename`. Corresponds to dpi of the exported image. Defaults to `100`. Large values (e.g. 100) are required to render small text properly. Small values (e.g. 20) will work when sequence/annotation text is off, and very small values (e.g. 10) will work when sequence/annotation text and outlines are all off.
`monitor_performance`	`logical`. Boolean specifying whether verbose performance monitoring should be messaged to console. Defaults to `FALSE`.

Value

A ggplot object containing the full visualisation, or `invisible(NULL)` if `return = FALSE`. It is often more useful to use `filename = "myfilename.png"`, because then the visualisation is exported at the correct aspect ratio.

Examples

```
## Extract info from dataframe
methylation_info <- extract_and_sort_methylation(example_many_sequences)

## Visualise example_many_sequences with all defaults
## This looks ugly because it isn't at the right scale/aspect ratio
visualise_methylation(
    methylation_info$locations,
    methylation_info$probabilities,
    methylation_info$sequences
)

## Export with all defaults rather than returning
visualise_methylation(
    methylation_info$locations,
    methylation_info$probabilities,
    methylation_info$sequences,
    filename = "example_vm_01.png",
    return = FALSE
)
## View exported image
image <- png::readPNG("example_vm_01.png")
unlink("example_vm_01.png")
grid::grid.newpage()
grid::grid.raster(image)

## Export with customisation
visualise_methylation(
    methylation_info$locations,
    methylation_info$probabilities,
    methylation_info$sequences,
    filename = "example_vm_02.png",
    return = FALSE,
    low_colour = "white",
    high_colour = "black",
    low_clamp = 0.3*255,
    high_clamp = 0.7*255,
    index_annotation_lines = c(1, 23, 37),
    index_annotation_interval = 3,
    index_annotation_full_line = FALSE,
    index_annotation_always_first_base = TRUE,
    index_annotation_always_last_base = TRUE,
    other_bases_colour = "lightblue1",
    other_bases_outline_linewidth = 1,
    other_bases_outline_colour = "grey",
    modified_bases_outline_linewidth = 3,
    modified_bases_outline_colour = "black",
    margin = 0.3
)
## View exported image
image <- png::readPNG("example_vm_02.png")
unlink("example_vm_02.png")
grid::grid.newpage()
grid::grid.raster(image)


## Export with customisation, viewing sequences
visualise_methylation(
    methylation_info$locations,
    methylation_info$probabilities,
    methylation_info$sequences,
    filename = "example_vm_03.png",
    return = FALSE,
    low_colour = "white",
    high_colour = "black",
    low_clamp = 0.3*255,
    high_clamp = 0.7*255,
    sequence_text_type = "sequence",
    sequence_text_colour = "red",
    index_annotation_lines = c(1, 23, 37),
    index_annotation_interval = 3,
    index_annotation_full_line = FALSE,
    index_annotation_always_first_base = TRUE,
    index_annotation_always_last_base = TRUE,
    other_bases_colour = "lightblue1",
    other_bases_outline_linewidth = 1,
    other_bases_outline_colour = "grey",
    modified_bases_outline_linewidth = 3,
    modified_bases_outline_colour = "black",
    margin = 0.3
)
## View exported image
```

```
image <- png::readPNG("example_vm_03.png")
unlink("example_vm_03.png")
grid::grid.newpage()
grid::grid.raster(image)


## Export with customisation, viewing probabilities
visualise_methylation(
    methylation_info$locations,
    methylation_info$probabilities,
    methylation_info$sequences,
    filename = "example_vm_04.png",
    return = FALSE,
    low_colour = "cyan",
    high_colour = "yellow",
    low_clamp = 0.3*255,
    high_clamp = 0.7*255,
    sequence_text_type = "probability",
    sequence_text_size = 10,
    sequence_text_colour = "black",
    index_annotation_lines = c(1, 23, 37),
    index_annotation_interval = 3,
    index_annotation_full_line = FALSE,
    index_annotation_always_first_base = TRUE,
    index_annotation_always_last_base = TRUE,
    other_bases_colour = "lightgreen",
    other_bases_outline_linewidth = 1,
    other_bases_outline_colour = "grey",
    modified_bases_outline_linewidth = 3,
    modified_bases_outline_colour = "black",
    margin = 0.3
)
## View exported image
image <- png::readPNG("example_vm_04.png")
unlink("example_vm_04.png")
grid::grid.newpage()
grid::grid.raster(image)


## Export with customisation, viewing probability integers
visualise_methylation(
    methylation_info$locations,
    methylation_info$probabilities,
    methylation_info$sequences,
    filename = "example_vm_05.png",
    return = FALSE,
    low_colour = "blue",
    high_colour = "red",
    low_clamp = 0.3*255,
    high_clamp = 0.7*255,
    sequence_text_type = "probability",
    sequence_text_scaling = c(0, 1),
    sequence_text_rounding = 0,
    sequence_text_size = 10,
    sequence_text_colour = "white",
    index_annotation_lines = c(1, 23, 37),
    index_annotation_interval = 3,
    index_annotation_full_line = FALSE,
    index_annotation_always_first_base = TRUE,
    index_annotation_always_last_base = TRUE,
    other_bases_outline_linewidth = 1,
    other_bases_outline_colour = "grey",
    modified_bases_outline_linewidth = 3,
    modified_bases_outline_colour = "black",
    margin = 0.3
)
## View exported image
image <- png::readPNG("example_vm_05.png")
unlink("example_vm_05.png")
grid::grid.newpage()
grid::grid.raster(image)
```

`visualise_methylation_colour_scale` *Visualise methylation colour scalebar*

Description

`visualize_methylation_color_scale()` is an alias for `visualise_methylation_colour_scale()` - see aliases.

This function creates a scalebar showing the colouring scheme based on methylation probability that is used in `visualise_methylation()`. Showing this is particularly important when the colour range is clamped via `low_clamp` and `high_clamp` (e.g. setting that all values below 100 are fully blue (`#0000FF`), all values above 200 are fully red (`#FF0000`), and colour interpolation occurs only in the range 100-200, rather than across the whole range 0-255). If clamping is off (default), then 0 is fully blue, 255 is fully read, and all values are linearly interpolated. NB: colours are configurable but default to blue = low modification probability and red = high modification probability.

Usage

```
visualise_methylation_colour_scale(
  low_colour = "blue",
  high_colour = "red",
  low_clamp = 0,
  high_clamp = 255,
  ...,
  full_range = c(0, 255),
  precision = 10^3,
  background_colour = "white",
  axis_location = "bottom",
  axis_title = NULL,
  do_axis_ticks = TRUE,
  outline_colour = "black",
  outline_linewidth = 1,
  monitor_performance = FALSE
)
```

Arguments

`low_colour`	`character`. The colour that should be used to represent minimum probability of methylation/modification (defaults to blue).
`high_colour`	`character`. The colour that should be used to represent maximum probability of methylation/modification (defaults to red).
`low_clamp`	`numeric`. The minimum probability below which all values are coloured `low_colour`. Defaults to `0` (i.e. no clamping). To specify a proportion probability in 8-bit form, multiply by 255 e.g. to low-clamp at 30% probability, set this to `0.3*255`.
`high_clamp`	`numeric`. The maximum probability above which all values are coloured `high_colour`. Defaults to `255` (i.e. no clamping, assuming Nanopore > SAM style modification calling where probabilities are 8-bit integers from 0 to 255).
`...`	Used to recognise aliases e.g. American spellings or common misspellings - see aliases. If any American spellings do not work, please make a bug report at `https://github.com/ejade42/ggDNAvis/issues`.
`full_range`	`numeric vector`, length 2. The total range of possible probabilities. Defaults to `c(0, 255)`, which is appropriate for Nanopore > SAM style modification calling where probabilities are 8-bit integers from 0 to 255. May need to be set to `c(0, 1)` if probabilites are instead stored as decimals. Setting any other value is advanced use and should be done for a good reason.
`precision`	`integer`. How many different shades should be rendered. Larger values give a smoother gradient. Defaults to `10^3` i.e. `1000`, which looks smooth to my eyes and isn't too intensive to calculate.
`background_colour`	`character`. The colour the background should be drawn (defaults to white).
`axis_location`	`character`. Which edge should be labelled. The gradient will always be along this axis (i.e. horizontal gradient for `"top"` or `"bottom"`, vertical gradient for `"left"` or `"right"`). Accepts `"top"` / `"north"`, `"bottom"` / `"south"`, `"left"` / `"west"`, and `"right"` / `"east"` (not case sensitive).
`axis_title`	`character`. The desired axis title for the edge selected by `axis_location`. Defaults to `NULL`.
`do_axis_ticks`	`logical`. Boolean specifying whether gradient axis ticks should be enabled (`TRUE`, default) or disabled (`FALSE`).
`outline_colour`	`character`. The colour of the scalebar outline. Defaults to black.
`outline_linewidth`	`numeric`. The linewidth of the scalebar outline. Defaults to `1`. Set to `0` to disable scalebar outline.

`monitor_performance`
: `logical`. Boolean specifying whether verbose performance monitoring should be messaged to console. Defaults to `FALSE`.

Value

ggplot of the scalebar.

Unlike the other `visualise_<>` functions in this package, does not directly export a png. This is because there are no squares that need to be rendered at a precise aspect ratio in this function. It can just be saved normally with `ggplot2::ggsave()` with any sensible combination of height and width.

Examples

```
## Defaults match defaults of visualise_methylation()
visualise_methylation_colour_scale()

## Use clamping and change colours
visualise_methylation_colour_scale(
    low_colour = "white",
    high_colour = "black",
    low_clamp = 0.3*255,
    high_clamp = 0.7*255,
    full_range = c(0, 255),
    background_colour = "lightblue1",
    axis_location = "bottom",
    axis_title = "Methylation probability"
)

## Lower precision = colour banding
visualise_methylation_colour_scale(
    precision = 10,
    do_axis_ticks = FALSE
)

## Left axis
visualise_methylation_colour_scale(
    precision = 100,
    axis_location = "WEST",
    axis_title = "vertical probability"
)
```

`visualise_single_sequence` *Visualise a single DNA/RNA sequence*

Description

`visualize_single_sequence()` is an alias for `visualise_single_sequence()` - see aliases.

This function takes a DNA/RNA sequence and returns a ggplot visualising it, with the option to directly export a png image with appropriate dimensions. Colours, line wrapping, index annotation interval, and pixels per square when exported are configurable.

Usage

```
visualise_single_sequence(
  sequence,
  ...,
  sequence_colours = sequence_colour_palettes$ggplot_style,
  background_colour = "white",
  line_wrapping = 75,
  spacing = 1,
  margin = 0.5,
  sequence_text_colour = "black",
  sequence_text_size = 16,
  index_annotation_colour = "darkred",
  index_annotation_size = 12.5,
  index_annotation_interval = 15,
  index_annotations_above = TRUE,
  index_annotation_vertical_position = 1/3,
  index_annotation_always_first_base = TRUE,
  index_annotation_always_last_base = TRUE,
  outline_colour = "black",
  outline_linewidth = 3,
  outline_join = "mitre",
  return = TRUE,
  filename = NA,
  force_raster = FALSE,
  render_device = ragg::agg_png,
  pixels_per_base = 100,
  monitor_performance = FALSE
)
```

Arguments

`sequence`	`character`. A DNA or RNA sequence to visualise e.g. `"AAATGCTGC"`.
`...`	Used to recognise aliases e.g. American spellings or common misspellings - see aliases. If any American spellings do not work, please make a bug report at `https://github.com/ejade42/ggDNAvis/issues`.
`sequence_colours`	`character vector`, length 4. A vector indicating which colours should be used for each base. In order: `c(A_colour, C_colour, G_colour, T/U_colour)`. Defaults to red, green, blue, purple in the default shades produced by ggplot

with 4 colours, i.e. `c("#F8766D", "#7CAE00", "#00BFC4", "#C77CFF")`, accessed via `sequence_colour_palettes$ggplot_style`.

`background_colour`
: `character`. The colour the background should be drawn (defaults to white).

`line_wrapping`
: `integer`. The number of bases that should be on each line before wrapping. Defaults to `75`. Recommended to make this a multiple of the repeat unit size (e.g. $3n$ for a trinucleotide repeat) if visualising a repeat sequence.

`spacing`
: `integer`. The number of blank lines between each line of sequence. Defaults to `1`.

  Be careful when setting to `0` as this means annotations have no space so might render strangely. Recommended to set `index_annotation_interval = 0` if doing so to disable annotations entirely.

`margin`
: `numeric`. The size of the margin relative to the size of each base square. Defaults to `0.5` (half the side length of each base square).

  Note that index annotations can require a minimum margin size at the top or bottom if present above the first/below the last row. This is handled automatically but can mean the top/bottom margin is sometimes larger than the `margin` setting.

  Very small margins ($\leq$0.25) may cause thick outlines to be cut off at the edges of the plot. Recommended to either use a wider margin or a smaller `outline_linewidth`.

`sequence_text_colour`
: `character`. The colour of the text within the bases (e.g. colour of "A" letter within boxes representing adenosine bases). Defaults to black.

`sequence_text_size`
: `numeric`. The size of the text within the bases (e.g. size of "A" letter within boxes representing adenosine bases). Defaults to `16`. Set to `0` to hide sequence text (show box colours only).

`index_annotation_colour`
: `character`. The colour of the little numbers underneath indicating base index (e.g. colour of "15" label under the 15th base). Defaults to dark red.

`index_annotation_size`
: `numeric`. The size of the little number underneath indicating base index (e.g. size of "15" label under the 15th base). Defaults to `12.5`.

  Setting to `0` disables index annotations (and prevents adding additional blank lines).

`index_annotation_interval`
: `integer`. The frequency at which numbers should be placed underneath indicating base index, starting counting from the leftmost base in each row. Defaults to `15` (every 15 bases along each row).

  Recommended to make this a factor/divisor of the line wrapping length (meaning the final base in each line is annotated), otherwise the numbering interval resetting at the beginning of each row will result in uneven intervals at each line break.

Setting to `0` disables index annotations (and prevents adding additional blank lines).

`index_annotations_above`
: `logical`. Whether index annotations should go above (`TRUE`, default) or below (`FALSE`) each line of sequence.

`index_annotation_vertical_position`
: `numeric`. How far annotation numbers should be rendered above (if `index_annotations_above = TRUE`) or below (if `index_annotations_above = FALSE`) each base. Defaults to `1/3`.

  Not recommended to change at all. Strongly discouraged to set below 0 or above 1.

`index_annotation_always_first_base`
: `logical`. Whether to force the first base in each line to always be annotated regardless of whether it occurs at the `index_annotation_interval`. Defaults to `TRUE`.

`index_annotation_always_last_base`
: `logical`. Whether to force the last base in each line to always be annotated regardless of whether it occurs at the `index_annotation_interval`. Defaults to `TRUE`.

`outline_colour`
: `character`. The colour of the box outlines. Defaults to black.

`outline_linewidth`
: `numeric`. The linewidth of the box outlines. Defaults to `3`. Set to `0` to disable box outlines.

`outline_join`
: `character`. One of `"mitre"`, `"round"`, or `"bevel"` specifying how outlines should be joined at the corners of boxes. Defaults to `"mitre"`. It would be unusual to need to change this.

`return`
: `logical`. Boolean specifying whether this function should return the ggplot object, otherwise it will return `invisible(NULL)`. Defaults to `TRUE`.

`filename`
: `character`. Filename to which output should be saved. If set to `NA` (default), no file will be saved. Recommended to end with `".png"`, but can change if render device is changed.

`force_raster`
: `logical`. Boolean specifying whether `ggplot2::geom_raster()` should be used even if it will remove text and outlines. Defaults to `FALSE`.

  To make the detailed plots with box outlines, sequence text, and index annotations, `ggplot2::geom_tile()` is used. However, `geom_tile` is slower for huge datasets, so there is an option to use `geom_raster` instead. `geom_raster` does not support box outlines, sequence text, or index annotations, but is much faster if only the colours are wanted.

  `geom_raster` is automatically used if it will not change the plot (i.e. if all extraneous elements are already off), but can be forced using this argument.

`render_device` `function/character`. Device to use when rendering. See `ggplot2::ggsave()` documentation for options. Defaults to `ragg::agg_png`. Can be set to `NULL` to infer from file extension, but results may vary between systems.

`pixels_per_base`
: `integer`. How large each box should be in pixels, if file output is turned on via setting `filename`. Corresponds to dpi of the exported image. Defaults to `100`.

  Large values (e.g. 100) are required to render small text properly. Small values (e.g. 20) will work when sequence/annotation text is off, and very small values (e.g. 10) will work when sequence/annotation text and outlines are all off.

`monitor_performance`
: `logical`. Boolean specifying whether verbose performance monitoring should be messaged to console. Defaults to `FALSE`.

Value

A ggplot object containing the full visualisation, or `invisible(NULL)` if `return = FALSE`. It is often more useful to use `filename = "myfilename.png"`, because then the visualisation is exported at the correct aspect ratio.

Examples

```
## Create sequence to visualise
sequence <- paste(c(rep("GGC", 72), rep("GGAGGAGGCGGC", 15)), collapse = "")

## Visualise with all defaults
## This looks ugly because it isn't at the right scale/aspect ratio
visualise_single_sequence(sequence)

## Export with all defaults rather than returning
visualise_single_sequence(
    sequence,
    filename = "example_vss_01.png",
    return = FALSE
)
## View exported image
image <- png::readPNG("example_vss_01.png")
unlink("example_vss_01.png")
grid::grid.newpage()
grid::grid.raster(image)

## Export while customising appearance
visualise_single_sequence(
    sequence,
    filename = "example_vss_02.png",
    return = FALSE,
    sequence_colours = sequence_colour_palettes$bright_pale,
    sequence_text_colour = "white",
    background_colour = "lightgrey",
    line_wrapping = 60,
    spacing = 2,
    outline_linewidth = 0,
    index_annotations_above = FALSE,
    index_annotation_always_first_base = FALSE,
    index_annotation_always_last_base = FALSE,
    margin = 0
)
## View exported image
image <- png::readPNG("example_vss_02.png")
unlink("example_vss_02.png")
grid::grid.newpage()
grid::grid.raster(image)
```

`write_fastq` *Write sequence and quality information to FASTQ*

Description

This function simply writes a FASTQ file from a dataframe containing columns for read ID, sequence, and quality scores.

See `fastq_quality_scores` for an explanation of quality.

Said dataframe can be produced from FASTQ via `read_fastq()`. To read/write a modified FASTQ containing modification information (SAM/BAM MM and ML tags) in the header lines, use `read_modified_fastq()` and `write_modified_fastq()`.

Usage

```
write_fastq(
  dataframe,
  filename = NA,
  ...,
  read_id_colname = "read",
  sequence_colname = "sequence",
  quality_colname = "quality",
  return = FALSE
)
```

Arguments

`dataframe`	Dataframe containing sequence information to write back to FASTQ. Must have columns for unique read ID and DNA sequence. Should also have a column for quality, unless wanting to fill in qualities with `"B"`.
`filename`	`character`. File to write the FASTQ to. Recommended to end with `.fastq` (warns but works if not). If set to `NA` (default), no file will be output, which may be useful for testing/debugging.

`...` Used to recognise aliases e.g. American spellings or common misspellings - see aliases. If any American spellings do not work, please make a bug report at https://github.com/ejade42/ggDNAvis/issues.

`read_id_colname`
: `character`. The name of the column within the dataframe that contains the unique ID for each read. Defaults to `"read"`.

`sequence_colname`
: `character`. The name of the column within the dataframe that contains the DNA sequence for each read. Defaults to `"sequence"`.

  The values within this column must be DNA sequences e.g. `"GGCGGC"`.

`quality_colname`
: `character`. The name of the column within the dataframe that contains the FASTQ quality scores for each read. Defaults to `"quality"`. If scores are not known, can be set to `NA` to fill in quality with `"B"`.

  If not `NA`, must correspond to a column where the values are the FASTQ quality scores e.g. `"$12\">/2C;4:9F8:816E,6C3*,"` - see `fastq_quality_scores`.

`return` `logical`. Boolean specifying whether this function should return the FASTQ (as a character vector of each line in the FASTQ), otherwise it will return `invisible(NULL)`. Defaults to `FALSE`.

Value

`character vector`. The resulting FASTQ file as a character vector of its constituent lines (or `invisible(NULL)` if `return` is `FALSE`). This is probably mostly useful for debugging, as setting `filename` within this function directly writes to FASTQ via `writeLines()`. Therefore, defaults to returning `invisible(NULL)`.

Examples

```
## Write to FASTQ (using filename = NA, return = FALSE
## to view as char vector rather than writing to file)
write_fastq(
    example_many_sequences,
    filename = NA,
    read_id_colname = "read",
    sequence_colname = "sequence",
    quality_colname = "quality",
    return = TRUE
)

## quality_colname = NA fills in quality with "B"
write_fastq(
    example_many_sequences,
    filename = NA,
    read_id_colname = "read",
    sequence_colname = "sequence",
    quality_colname = NA,
    return = TRUE
)
```

`write_modified_fastq` *Write modification information stored in dataframe back to modified FASTQ*

Description

This function takes a dataframe containing DNA modification information (e.g. produced by `read_modified_fastq()`) and writes it back to modified FASTQ, equivalent to what would be produced via `samtools fastq -T MM,ML`.

Arguments give the names of columns within the dataframe from which to read.

If multiple types of modification have been assessed (e.g. both methylation and hydroxymethylation), then multiple colnames must be provided for locations and probabilites, and multiple prefixes (e.g. `"C+h?"`) must be provided. **IMPORTANT:** These three vectors must all be the same length, and the modification types must be in a consistent order (e.g. if writing hydroxymethylation and methylation in that order, must do H then M in all three vectors and never vice versa).

If quality isn't known (e.g. there was a FASTA step at some point in the pipeline), the `quality` argument can be set to `NA` to fill in quality scores with `"B"`. This is the same behaviour as SAMtools v1.21 when converting FASTA to SAM/BAM then FASTQ. I don't really know why SAMtools decided the default quality should be "B" but there was probably a reason so I have stuck with that.

Default arguments are set up to work with the included `example_many_sequences` data.

Usage

```
write_modified_fastq(
  dataframe,
  filename = NA,
  ...,
  read_id_colname = "read",
  sequence_colname = "sequence",
  quality_colname = "quality",
 locations_colnames = c("hydroxymethylation_locations", "methylation_locations"),
  probabilities_colnames = c("hydroxymethylation_probabilities",
    "methylation_probabilities"),
  modification_prefixes = c("C+h?", "C+m?"),
  include_blank_tags = TRUE,
  return = FALSE
)
```

Arguments

`dataframe`
: `dataframe`. Dataframe containing modification information to write back to modified FASTQ. Must have columns for unique read ID, DNA sequence, and at least one set of locations and probabilities for a particular modification type (e.g. 5C methylation).

`filename`
: `character`. File to write the FASTQ to. Recommended to end with `.fastq` (warns but works if not). If set to `NA` (default), no file will be output, which may be useful for testing/debugging.

`...`
: Used to recognise aliases e.g. American spellings or common misspellings - see aliases. If any American spellings do not work, please make a bug report at `https://github.com/ejade42/ggDNAvis/issues`.

`read_id_colname`
: `character`. The name of the column within the dataframe that contains the unique ID for each read. Defaults to `"read"`.

`sequence_colname`
: `character`. The name of the column within the dataframe that contains the DNA sequence for each read. Defaults to `"sequence"`.

  The values within this column must be DNA sequences e.g. `"GGCGGC"`.

`quality_colname`
: `character`. The name of the column within the dataframe that contains the FASTQ quality scores for each read. Defaults to `"quality"`. If scores are not known, can be set to `NA` to fill in quality with `"B"`.

  If not `NA`, must correspond to a column where the values are the FASTQ quality scores e.g. `"$12\">/2C;4:9F8:816E,6C3*,"` - see `fastq_quality_scores`.

`locations_colnames`
: `character vector`. Vector of the names of all columns within the dataframe that contain modification locations. Defaults to `c("hydroxymethylation_locations", "methylation_locations")`.

  The values within these columns must be comma-separated strings of indices at which modification was assessed, as produced by `vector_to_string()`, e.g. `"3,6,9,12"`.

  Will fail if these locations are not instances of the target base (e.g. `"C"` for `"C+m?"`), as the SAMtools tag system does not work otherwise. One consequence of this is that if sequences have been reversed via `merge_methylation_with_metadata()` or helpers, they cannot be written to FASTQ *unless* modification locations are symmetric e.g. CpG *and* offset was set to `1` when reversing (see `reverse_locations_if_needed()`).

`probabilities_colnames`
: `character vector`. Vector of the names of all columns within the dataframe that contain modification probabilities. Defaults to `c("hydroxymethylation_probabilities", "methylation_probabilities")`.

  The values within the columns must be comma-separated strings of modification probabilities, as produced by `vector_to_string()`, e.g. `"0,255,128,78"`.

`modification_prefixes`
: `character vector`. Vector of the prefixes to be used for the MM tags specifying modification type. These are usually generated by Dorado/Guppy based on the original modified basecalling settings, and more details can be found in the SAM optional tag specifications. Defaults to `c("C+h?", "C+m?")`.

  `locations_colnames`, `probabilities_colnames`, and `modification_prefixes` must all have the same length e.g. 2 if there were 2 modification types assessed.

`include_blank_tags`
: `logical`. Boolean specifying what to do if a particular read has no assessed locations for a given modification type from `modification_prefixes`.

  If `TRUE` (default), blank tags will be written e.g. `"C+h?;"` (whereas a normal, non-blank tag looks like `"C+h?,0,0,0,0;"`). If `FALSE`, tags with no assessed locations in that read will not be written at all.

`return`
: `logical`. Boolean specifying whether this function should return the FASTQ (as a character vector of each line in the FASTQ), otherwise it will return `invisible(NULL)`. Defaults to `FALSE`.

Value

`character vector`. The resulting modified FASTQ file as a character vector of its constituent lines (or `invisible(NULL)` if `return` is `FALSE`). This is probably mostly useful for debugging, as setting `filename` within this function directly writes to FASTQ via `writeLines()`. Therefore, defaults to returning `invisible(NULL)`.

Examples

```
## Write to FASTQ (using filename = NA, return = FALSE
## to view as char vector rather than writing to file)
write_modified_fastq(
    example_many_sequences,
    filename = NA,
    read_id_colname = "read",
    sequence_colname = "sequence",
    quality_colname = "quality",
    locations_colnames = c("hydroxymethylation_locations",
                           "methylation_locations"),
    probabilities_colnames = c("hydroxymethylation_probabilities",
                               "methylation_probabilities"),
    modification_prefixes = c("C+h?", "C+m?"),
    return = TRUE
)

## Write methylation only, and fill in qualities with "B"
write_modified_fastq(
    example_many_sequences,
    filename = NA,
    read_id_colname = "read",
    sequence_colname = "sequence",
    quality_colname = NA,
    locations_colnames = c("methylation_locations"),
    probabilities_colnames = c("methylation_probabilities"),
    modification_prefixes = c("C+m?"),
    return = TRUE
)
```

Index