Package: microseq (via r-universe)

August 20, 2024

Type Package

Title Basic Biological Sequence Handling

Version 2.1.6

Date 2023-08-21

Author Lars Snipen, Kristian Hovde Liland

Maintainer Lars Snipen <lars.snipen@nmbu.no>

Description Basic functions for microbial sequence data analysis. The idea is to use generic R data structures as much as possible, making R data wrangling possible also for sequence data.

License GPL-2

Depends R (>= 4.0.0), tibble, stringr, dplyr, data.table, rlang

Imports Rcpp (>= 0.12.0)

LazyData FALSE

ZipData TRUE

LinkingTo Rcpp (>= 0.12.0)

RoxygenNote >=7.2.3

Repository https://larssnip.r-universe.dev

RemoteUrl https://github.com/larssnip/microseq

RemoteRef HEAD

RemoteSha 1bf4da28cd4e52adc3423a0d5717524070d8321d

Contents

`microseq-package` *Basic Biological Sequence Analysis*

Description

A collection of functions for basic analysis of microbial sequence data.

Usage

```
microseq()
```

Details

Package:	microseq
Type:	Package
Version:	2.1.6
Date:	2023-08-21
License:	GPL-2

Author(s)

Lars Snipen, Kristian Hovde Liland
Maintainer: Lars Snipen <lars.snipen@nmbu.no>

backTranslate *Replace amino acids with codons*

Description

Replaces aligned amino acids with their original codon triplets.

Usage

```
backTranslate(aa.msa, nuc.ffn)
```

Arguments

aa.msa	A fasta object with a multiple alignment, see msalign.
nuc.ffn	A fasta object with the coding sequences, see readFasta.

Details

This function replaces the aligned amino acids in aa.msa with their original codon triplets. This is possible only when the nucleotide sequences in nuc.ffn are the exact nucleotide sequences behind the protein sequences that are aligned in aa.msa.

It is required that the first token of the 'Header' lines is identical for a protein sequence in aa.msa and its nucleotide version in 'nuc.ffn', otherwise it is impossible to match them. Thus, they may not appear in the same order in the two input fasta objects.

When aligning coding sequences, one should in general always align their protein sequences, to keep the codon structure, and then use backTranslate to convert this into a nucleotide alignment, if required.

If the nuclotide sequences contain the stop codons, these will be removed.

Value

A fasta object similar to aa.msa, but where each amino acid has been replace by its corresponding codon. All gaps '"-"' are replaced by triplets '"---"'.

Author(s)

Lars Snipen.

See Also

msalign, readFasta.

Examples

```
msa.file <- file.path(file.path(path.package("microseq"),"extdata"), "small.msa")
aa.msa <- readFasta(msa.file)
nuc.file <- file.path(file.path(path.package("microseq"),"extdata"), "small.ffn")
nuc <- readFasta(nuc.file)
nuc.msa <- backTranslate(aa.msa, nuc)
```

`findGenes` *Finding coding genes*

Description

Finding coding genes in genomic DNA using the Prodigal software.

Usage

```
findGenes(
  genome,
  prodigal.exe = "prodigal",
  faa.file = "",
  ffn.file = "",
  proc = "single",
  trans.tab = 11,
  mask.N = FALSE,
  bypass.SD = FALSE
)
```

Arguments

`genome`	A table with columns Header and Sequence, containing the genome sequence(s).
`prodigal.exe`	Command to run the external software prodigal on the system (text).
`faa.file`	If provided, prodigal will output all proteins to this fasta-file (text).
`ffn.file`	If provided, prodigal will output all DNA sequences to this fasta-file (text).
`proc`	Either `"single"` or `"meta"`, see below.
`trans.tab`	Either 11 or 4 (see below).
`mask.N`	Turn on masking of N's (logical)
`bypass.SD`	Bypass Shine-Dalgarno filter (logical)

Details

The external software Prodigal is used to scan through a prokaryotic genome to detect the protein coding genes. The text in `prodigal.exe` must contain the exact command to invoke barrnap on the system.

In addition to the standard output from this function, FASTA files with protein and/or DNA sequences may be produced directly by providing filenames in `faa.file` and `ffn.file`.

The input `proc` allows you to specify if the input data should be treated as a single genome (default) or as a metagenome. In the latter case the `genome` are (un-binned) contigs.

The translation table is by default 11 (the standard code), but table 4 should be used for Mycoplasma etc.

The `mask.N` will prevent genes having runs of N inside. The `bypass.SD` turn off the search for a Shine-Dalgarno motif.

Value

A GFF-table (see `readGFF` for details) with one row for each detected coding gene.

Note

The prodigal software must be installed on the system for this function to work, i.e. the command '`system("prodigal -h")`' must be recognized as a valid command if you run it in the Console window.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

`readGFF`, `gff2fasta`.

Examples

```
## Not run:
# This example requires the external prodigal software
# Using a genome file in this package.
genome.file <- file.path(path.package("microseq"),"extdata","small.fna")

# Searching for coding sequences, this is Mycoplasma (trans.tab = 4)
genome <- readFasta(genome.file)
gff.tbl <- findGenes(genome, trans.tab = 4)

# Retrieving the sequences
cds.tbl <- gff2fasta(gff.tbl, genome)

# You may use the pipe operator
library(ggplot2)
readFasta(genome.file) %>%
  findGenes(trans.tab = 4) %>%
```

```
  filter(Score >= 50) %>%
  ggplot() +
  geom_histogram(aes(x = Score), bins = 25)

## End(Not run)
```

`findOrfs` *Finding ORFs in genomes*

Description

Finds all ORFs in prokaryotic genome sequences.

Usage

```
findOrfs(genome, circular = F, trans.tab = 11)
```

Arguments

`genome`	A fasta object (`tibble`) with the genome sequence(s).
`circular`	Logical indicating if the genome sequences are completed, circular sequences.
`trans.tab`	Translation table.

Details

A prokaryotic Open Reading Frame (ORF) is defined as a sub-sequence starting with a start-codon (ATG, GTG or TTG), followed by an integer number of triplets (codons), and ending with a stop-codon (TAA, TGA or TAG, unless `trans.tab = 4`, see below). This function will locate all such ORFs in a genome.

The argument `genome` is a fasta object, i.e. a table with columns '`Header`' and '`Sequence`', and will typically have several sequences (chromosomes/plasmids/scaffolds/contigs). It is vital that the *first token* (characters before first space) of every '`Header`' is unique, since this will be used to identify these genome sequences in the output.

By default the genome sequences are assumed to be linear, i.e. contigs or other incomplete fragments of a genome. In such cases there will usually be some truncated ORFs at each end, i.e. ORFs where either the start- or the stop-codon is lacking. In the `orf.table` returned by this function this is marked in the '`Attributes`' column. The texts "Truncated=10" or "Truncated=01" indicates truncated at the beginning or end of the genomic sequence, respectively. If the supplied `genome` is a completed genome, with circular chromosome/plasmids, set the flag `circular = TRUE` and no truncated ORFs will be listed. In cases where an ORF runs across the origin of a circular genome sequences, the stop coordinate will be larger than the length of the genome sequence. This is in line with the specifications of the GFF3 format, where a '`Start`' cannot be larger than the corresponding '`End`'.

An alternative translation table may be specified, and as of now the only alternative implemented is table 4. This means codon TGA is no longer a stop, but codes for Tryptophan. This coding is used by some bacteria (e.g. under the orders Entomoplasmatales and Mycoplasmatales).

Note that for any given stop-codon there are usually multiple start-codons in the same reading frame. This function will return all such nested ORFs, i.e. the same stop position may appear multiple times. If you want ORFs with the most upstream start-codon only (LORFs), then filter the output from this function with `lorfs`.

Value

This function returns an `orf.table`, which is simply a `tibble` with columns adhering to the GFF3 format specifications (a `gff.table`), see `readGFF`. If you want to retrieve the actual ORF sequences, use `gff2fasta`.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

`readGFF`, `gff2fasta`, `lorfs`.

Examples

```
# Using a genome file in this package
genome.file <- file.path(path.package("microseq"),"extdata","small.fna")

# Reading genome and finding orfs
genome <- readFasta(genome.file)
orf.tbl <- findOrfs(genome)

# Pipeline for finding LORFs of minimum length 100 amino acids
# and collecting their sequences from the genome
findOrfs(genome) %>%
 lorfs() %>%
 filter(orfLength(., aa = TRUE) > 100) %>%
 gff2fasta(genome) -> lorf.tbl
```

`findrRNA` *Finding rRNA genes*

Description

Finding rRNA genes in genomic DNA using the barrnap software.

Usage

```
findrRNA(genome, barrnap.exe = "barrnap", bacteria = TRUE, cpu = 1)
```

Arguments

genome	A table with columns Header and Sequence, containing the genome sequence(s).
barrnap.exe	Command to run the external software barrnap on the system (text).
bacteria	Logical, the genome is either a bacteria (default) or an archea.
cpu	Number of CPUs to use, default is 1.

Details

The external software barrnap is used to scan through a prokaryotic genome to detect the rRNA genes (5S, 16S, 23S). The text in `barrnap.exe` must contain the exact command to invoke barrnap on the system.

Value

A GFF-table (see `readGFF` for details) with one row for each detected rRNA sequence.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

`readGFF`, `gff2fasta`.

Examples

```
## Not run:
# This example requires the external barrnap software
# Using a genome file in this package.
genome.file <- file.path(path.package("microseq"),"extdata","small.fna")

# Searching for rRNA sequences, and inspecting
genome <- readFasta(genome.file)
gff.tbl <- findrRNA(genome)
print(gff.table)

# Retrieving the sequences
rRNA <- gff2fasta(gff.tbl, genome)

## End(Not run)
```

gff2fasta *Retrieving annotated sequences*

Description

Retrieving from a genome the sequences specified in a gff.table.

Usage

```
gff2fasta(gff.table, genome)
```

Arguments

gff.table	A gff.table (tibble) with genomic features information.
genome	A fasta object (tibble) with the genome sequence(s).

Details

Each row in gff.table (see readGFF) describes a genomic feature in the genome, which is a tibble with columns 'Header' and 'Sequence'. The information in the columns Seqid, Start, End and Strand are used to retrieve the sequences from the 'Sequence' column of genome. Every Seqid in the gff.table must match the first token in one of the 'Header' texts, in order to retrieve from the correct 'Sequence'.

Value

A fasta object with one row for each row in gff.table. The Header for each sequence is a summary of the information in the corresponding row of gff.table.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

readGFF, findOrfs.

Examples

```
# Using two files in this package
gff.file <- file.path(path.package("microseq"),"extdata","small.gff")
genome.file <- file.path(path.package("microseq"),"extdata","small.fna")

# Reading the genome first
genome <- readFasta(genome.file)

# Retrieving sequences
gff.table <- readGFF(gff.file)
```

```
fa.tbl <- gff2fasta(gff.table, genome)

# Alternative, using piping
readGFF(gff.file) %>% gff2fasta(genome) -> fa.tbl
```

---

gregexpr *Extended* gregexpr *with substring retrieval*

Description

An extension of the function base::gregexpr enabling retrieval of the matching substrings.

Usage

```
gregexpr(
  pattern,
  text,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE,
  extract = FALSE
)
```

Arguments

pattern	Character string containing a regular expression (or character string for fixed = TRUE) to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are not allowed.
text	A character vector where matches are sought, or an object which can be coerced by as.character to a character vector.
ignore.case	If FALSE, the pattern matching is *case sensitive* and if TRUE, case is ignored during matching.
perl	Logical. Should perl-compatible regexps be used? Has priority over extended.
fixed	Logical. If TRUE, 'pattern' is a string to be matched as is. Overrides all conflicting arguments.
useBytes	Logical. If TRUE the matching is done byte-by-byte rather than character-by-character. See grep for details.
extract	Logical indicating if matching substrings should be extracted and returned.

Details

Extended version of base:gregexpr that enables the return of the substrings matching the pattern. The last argument 'extract' is the only difference to base::gregexpr. The default behaviour is identical to base::gregexpr, but setting extract=TRUE means the matching substrings are returned.

Value

It will either return what the base::gregexpr would (extract = FALSE) or a 'list' of substrings matching the pattern (extract = TRUE). There is one 'list' element for each string in 'text', and each list element contains a character vector of all matching substrings in the corresponding entry of 'text'.

Author(s)

Lars Snipen and Kristian Liland.

See Also

grep

Examples

```
sequences <- c("ACATGTCATGTCC", "CTTGTATGCTG")
gregexpr("ATG", sequences, extract = TRUE)
```

---

iupac2regex *Ambiguity symbol conversion*

Description

Converting DNA ambiguity symbols to regular expressions, and vice versa.

Usage

```
iupac2regex(sequence)
regex2iupac(sequence)
```

Arguments

sequence	Character vector containing DNA sequences.

Details

The DNA alphabet may contain ambiguity symbols, e.g. a W means either A or T. When using a regular expression search, these letters must be replaced by the proper regular expression, e.g. W is replaced by [AT] in the string. The iupac2regex makes this translation, while regex2iupac converts the other way again (replace [AT] with W).

Value

A string where the ambiguity symbol has been replaced by a regular expression (iupac2regex) or a regular expression has been replaced by an ambiguity symbol (regex2iupac).

Author(s)

Lars Snipen.

Examples

```
iupac2regex("ACWGT")
regex2iupac("AC[AG]GT")
```

`lorfs` *Longest ORF*

Description

Filtering an `orf.table` with ORF information to keep only the LORFs.

Usage

```
lorfs(orf.tbl)
```

Arguments

`orf.tbl`	A `tibble` with the nine columns of the GFF-format (see `findOrfs`).

Details

For every stop-codon there are usually multiple possible start-codons in the same reading frame (nested ORFs). The LORF (Longest ORF) is defined as the longest of these nested ORFs, i.e. the ORF starting at the most upstream start-codon matching the stop-codon.

Value

A `tibble` with a subset of the rows of the argument `orf.tbl`. After this filtering the Type variable in `orf.tbl` is changed to `"LORF"`. If you want to retrieve the LORF sequences, use `gff2fasta`.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

`readGFF`, `findOrfs`, `gff2fasta`.

Examples

```
# See the example in the Help-file for findOrfs.
```

msa2mat *Convert alignment to matrix*

Description

Converts a FASTA formatted multiple alignment to a matrix.

Usage

```
msa2mat(msa)
```

Arguments

msa	A fasta object with a multiple alignment, see msalign‘.

Details

This function converts the fasta object msa, containing a multiple alignment, to a matrix. This means each position in the alignment is a column in the matrix, and the content of the ‘Header’ column of msa is used as rownames of theh matrix.

Such a matrix is useful for conversion to a DNAbin object that is used by the ape package for reconstructing phylogenetic trees.

Value

A matrix where each row is a vector of aligned bases/amino acids.

Author(s)

Lars Snipen.

See Also

msalign, readFasta.

Examples

```
msa.file <- file.path(path.package("microseq"),"extdata", "small.msa")
msa <- readFasta(msa.file)
msa.mat <- msa2mat(msa)  # to use with ape::as.DNAbin(msa.mat)
```

`msalign` *Multiple alignment*

Description

Quickly computing a smallish multiple sequence alignment.

Usage

```
msalign(fsa.tbl, machine = "microseq::muscle")
```

Arguments

`fsa.tbl`	A fasta object (`tibble`) with input sequences.
`machine`	Function that does the 'dirty work'.

Details

This function computes a multiple sequence alignment given a set of sequences in a fasta object, see `readFasta` for more on fasta objects.

It is merely a wrapper for the function named in `machine` to avoid explicit writing and reading of files. This function should only be used for small data sets, since no result files are stored. For heavier jobs, use the `machine` function directly.

At present, the only `machine` function implemented is `muscle`, but other third-party `machines` may be included later.

Note that this function will run `muscle` with default settings, which is fine for small data sets.

Value

Results are returned as a fasta object, i.e. a `tibble` with columns 'Header' and 'Sequence'.

Author(s)

Lars Snipen.

See Also

`muscle`, `msaTrim`.

Examples

```
## Not run:
prot.file <- file.path(file.path(path.package("microseq"),"extdata"),"small.faa")
faa <- readFasta(prot.file)
msa <- msalign(faa)

## End(Not run)
```

`msaTrim` *Trimming multiple sequence alignments*

Description

Trimming a multiple sequence alignment by discarding columns with too many gaps.

Usage

```
msaTrim(msa, gap.end = 0.5, gap.mid = 0.9)
```

Arguments

`msa`	A fasta object containing a multiple alignment.
`gap.end`	Fraction of gaps tolerated at the ends of the alignment (0-1).
`gap.mid`	Fraction of gaps tolerated inside the alignment (0-1).

Details

A multiple alignment is trimmed by removing columns with too many indels (gap-symbols). Any columns containing a fraction of gaps larger than `gap.mid` are discarded. For this reason, `gap.mid` should always be farily close to 1.0 therwise too many columns may be discarded, destroying the alignment.

Due to the heuristics of multiple alignment methods, both ends of the alignment tend to be uncertain and most of the trimming should be done at the ends. Starting at each end, columns are discarded as long as their fraction of gaps surpasses `gap.end`. Typically `gap.end` can be much smaller than `gap.mid`, but if set too low you risk that all columns are discarded!

Value

The trimmed alignment is returned as a fasta object.

Author(s)

Lars Snipen.

See Also

`muscle`, `msalign`.

Examples

```
msa.file <- file.path(path.package("microseq"),"extdata", "small.msa")
msa <- readFasta(msa.file)
print(str_length(msa$Sequence))
msa.trimmed <- msaTrim(msa)
print(str_length(msa.trimmed$Sequence))
msa.mat <- msa2mat(msa)  # for use with ape::as.DNAbin(msa.mat)
```

muscle *Multiple alignment using MUSCLE*

Description

Computing a multiple sequence alignment using the MUSCLE software.

Usage

```
muscle(
  in.file,
  out.file,
  muscle.exe = "muscle",
  quiet = FALSE,
  diags = FALSE,
  maxiters = 16
)
```

Arguments

in.file	Name of FASTA file with input sequences.
out.file	Name of file to store the result.
muscle.exe	Command to run the external software muscle on the system (text).
quiet	Logical, quiet = FALSE produces screen output during computations.
diags	Logical, diags = TRUE gives faster but less reliable alignment.
maxiters	Maximum number of iterations.

Details

The software MUSCLE (Edgar, 2004) must be installed and available on the system. The text in muscle.exe must contain the exact command to invoke muscle on the system.

By default diags = FALSE but can be set to TRUE to increase speed. This should be done only if sequences are highly similar.

By default maxiters = 16. If you have a large number of sequences (a few thousand), or they are very long, then this may be too slow for practical use. A good compromise between speed and accuracy is to run just the first two iterations of the algorithm. On average, this gives accuracy equal to T-Coffee and speeds much faster than CLUSTALW. This is done by the option maxiters = 2.

Value

The result is written to the file specified in out.file.

Author(s)

Lars Snipen.

References

Edgar, R.C. (2004). MUSCLE: multiple sequence alignment with high accuracy and high throughput, Nucleic Acids Res, 32, 1792-1797.

See Also

msaTrim.

Examples

```
## Not run:
fa.file <- file.path(file.path(path.package("microseq"),"extdata"),"small.faa")
muscle(in.file = fa.file, out.file = "delete_me.msa")

## End(Not run)
```

---

orfLength *Length of ORF*

Description

Computing the lengths of all ORFs in an `orf.table`.

Usage

```
orfLength(orf.table, aa = FALSE)
```

Arguments

`orf.table`	A GFF-formatted `tibble`.
`aa`	Logical, length in amino acids instead of bases.

Details

By default, computes the length of an ORF in bases, including the stop codon. However, if `aa = TRUE`, then the length is in amino acids after translation. This aa-length is the base-length divided by 3 and minus 1, unless the ORF is truncated and lacks a stop codon.

Value

A vector of lengths.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

findOrfs.

Examples

```
# See the example in the Help-file for findOrfs.
```

---

orfSignature *Signature for each ORF*

Description

Creates a signature text for orfs in an orf.table.

Usage

```
orfSignature(orf.table, full = TRUE)
```

Arguments

orf.table	A tibble with ORF information.
full	Logical indicating type of signature.

Details

A signature is a text that uniquely identifies each ORF in an orf.table, which is a GFF-table with columns Seqid, Start, End and Strand.

The full signature (full = TRUE) contains the Seqid, Start, End and Strand information for each ORF, separated by semicolon ";". This text is always unique to each ORF. If full = FALSE the Signature will not contain the starting position information for each ORF. This means all nested ORFs ending at the same stop-codon will then get identical Signatures. This is useful for identifying which ORFs are nested within the same LORF.

Note that the signature you get with full = FALSE contains Seqid, then End if on the positive Strand, Start otherwise, and then the Strand.

Value

A text vector with the Signature for each ORF.

Author(s)

Lars Snipen.

See Also

findOrfs.

Examples

```
# Using a genome file in this package
genome.file <- file.path(path.package("microseq"),"extdata","small.fna")

# Reading genome and finding orfs
genome <- readFasta(genome.file)
orf.tbl <- findOrfs(genome)

# Compute signatures
signature.full <- orfSignature(orf.tbl)
signature.reduced <- orfSignature(orf.tbl, full = FALSE)
```

`readFasta` *Read and write FASTA files*

Description

Reads and writes biological sequences (DNA, RNA, protein) in the FASTA format.

Usage

```
readFasta(in.file)
writeFasta(fdta, out.file, width = 0)
```

Arguments

`in.file`	url/directory/name of (gzipped) FASTA file to read.
`fdta`	A `tibble` with sequence data, see 'Details' below.
`out.file`	Name of (gzipped) FASTA file to create.
`width`	Number of characters per line, or 0 for no linebreaks.

Details

These functions handle input/output of sequences in the commonly used FASTA format. For every sequence it is presumed there is one Header-line starting with a '>'. If filenames (`in.file` or `out.file`) have the extension `.gz` they will automatically be compressed/uncompressed.

The sequences are stored in a `tibble`, opening up all the possibilities in R for fast and easy manipulations. The content of the file is stored as two columns, '`Header`' and '`Sequence`'. If other columns are added, these will be ignored by `writeFasta`.

The default `width = 0` in `writeFasta` results in no linebreaks in the sequences (one sequence per line).

Value

readFasta returns a tibble with the contents of the (gzipped) FASTA file stored in two columns of text. The first, named 'Header', contains the headerlines and the second, named 'Sequence', contains the sequences.

writeFasta produces a (gzipped) FASTA file.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

readFastq.

Examples

```
## Not run:
# We need a FASTA-file to read, here is one example file:
fa.file <- file.path(file.path(path.package("microseq"),"extdata"),"small.ffn")

# Read and write
fdta <- readFasta(fa.file)
ok <- writeFasta(fdta[4:5,], out.file = "delete_me.fasta")

# Make use of dplyr to copy parts of the file to another file
readFasta(fa.file) %>%
  filter(str_detect(Sequence, "TGA$")) %>%
  writeFasta(out.file = "TGAstop.fasta", width = 80) -> ok

## End(Not run)
```

---

readFastq *Read and write FASTQ files*

Description

Reads and writes files in the FASTQ format.

Usage

```
readFastq(in.file)
writeFastq(fdta, out.file)
```

Arguments

in.file	url/directory/name of (gzipped) FASTQ file to read.
fdta	FASTQ object to write.
out.file	url/directory/name of (gzipped) FASTQ file to write.

Details

These functions handle input/output of sequences in the commonly used FASTQ format, typically used for storing DNA sequences (reads) after sequencing. If filenames (`in.file` or `out.file`) have the extension `.gz` they will automatically be compressed/uncompressed.

The sequences are stored in a `tibble`, opening up all the possibilities in R for fast and easy manipulations. The content of the file is stored as three columns, '`Header`', '`Sequence`' and '`Quality`'. If other columns are added, these will be ignored by `writeFastq`.

Value

`readFastq` returns a `tibble` with the contents of the (gzipped) FASTQ file stored in three columns of text. The first, named '`Header`', contains the headerlines, the second, named '`Sequence`', contains the sequences and the third, named '`Quality`' contains the base quality scores.

`writeFastq` produces a (gzipped) FASTQ file.

Note

These functions will only handle files where each entry spans one single line, i.e. not the (uncommon) multiline FASTQ format.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

codereadFasta.

Examples

```
## Not run:
# We need a FASTQ-file to read, here is one example file:
fq.file <- file.path(file.path(path.package("microseq"),"extdata"),"small.fastq.gz")

# Read and write
fdta <- readFastq(fq.file)
ok <- writeFastq(fdta[1:3,], out.file = "delete_me.fq")

# Make use of dplyr to copy parts of the file to another file
readFastq(fq.file) %>%
  mutate(Length = str_length(Sequence)) %>%
  filter(Length > 200) %>%
  writeFasta(out.file = "long_reads.fa") # writing to FASTA file

## End(Not run)
```

`readGFF` *Reading and writing GFF-tables*

Description

Reading or writing a GFF-table from/to file.

Usage

```
readGFF(in.file)
writeGFF(gff.table, out.file)
```

Arguments

`in.file`	Name of file with a GFF-table.
`gff.table`	A table (`tibble`) with genomic features information.
`out.file`	Name of file.

Details

A GFF-table is simply a `tibble` with columns adhering to the format specified by the GFF3 format, see https://github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md for details. There is one row for each feature.

The following columns should always be in a full `gff.table` of the GFF3 format:

- Seqid. A unique identifier of the genomic sequence on which the feature resides.
- Source. A description of the procedure that generated the feature, e.g. `"R-package micropan::findOrfs"`.
- Type The type of feature, e.g. `"ORF"`, `"16S"` etc.
- Start. The leftmost coordinate. This is the start if the feature is on the Sense strand, but the end if it is on the Antisense strand.
- End. The rightmost coordinate. This is the end if the feature is on the Sense strand, but the start if it is on the Antisense strand.
- Score. A numeric score (E-value, P-value) from the `Source`.
- Strand. A `"+"` indicates Sense strand, a `"-"` Antisense.
- Phase. Only relevant for coding genes. the values 0, 1 or 2 indicates the reading frame, i.e. the number of bases to offset the `Start` in order to be in the reading frame.
- Attributes. A single string with semicolon-separated tokens prociding additional information.

Missing values are described by `"."` in the GFF3 format. This is also done here, except for the numerical columns Start, End, Score and Phase. Here `NA` is used, but this is replaced by `"."` when writing to file.

The `readGFF` function will also read files where sequences in FASTA format are added after the GFF-table. This file section must always start with the line `##FASTA`. This fasta object is added to the GFF-table as an attribute (use `attr(gff.tbl, "FASTA")` to retrieve it).

Value

`readGFF` returns a `gff.table` with the columns described above.

`writeGFF` writes the supplied `gff.table` to a text-file.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

`findOrfs`, `lorfs`.

Examples

```
# Using a GFF file in this package
gff.file <- file.path(path.package("microseq"),"extdata","small.gff")

# Reading gff-file
gff.tbl <- readGFF(gff.file)
```

---

reverseComplement *Reverse-complementation of DNA*

Description

The standard reverse-complement of nucleotide sequences.

Usage

```
reverseComplement(nuc.sequences, reverse = TRUE)
```

Arguments

`nuc.sequences`	Character vector containing the nucleotide sequences.
`reverse`	Logical indicating if complement should be reversed.

Details

With '`reverse = FALSE`' the DNA sequence is only complemented, not reversed.

This function will handle the IUPAC ambiguity symbols, i.e. '`R`' is reverse-complemented to '`Y`' etc.

Value

A character vector of reverse-complemented sequences.

Author(s)

Lars Snipen and Kristian Hovde Liland.

See Also

iupac2regex, regex2iupac.

Examples

```
fa.file <- file.path(file.path(path.package("microseq"),"extdata"),"small.ffn")
fa <- readFasta(fa.file)
reverseComplement(fa$Sequence)

#' # Or, make use of dplyr to manipulate tables
readFasta(fa.file) %>%
  mutate(RevComp = reverseComplement(Sequence)) -> fa.tbl
```

---

translate *Translation according to the standard genetic code*

Description

The translation from DNA(RNA) to amino acid sequence according to the standard genetic code.

Usage

```
translate(nuc.sequences, M.start = TRUE, no.stop = TRUE, trans.tab = 11)
```

Arguments

nuc.sequences	Character vector containing the nucleotide sequences.
M.start	A logical indicating if the amino acid sequence should start with M regardless of start codon.
no.stop	A logical indicating if terminal stops (*) should be eliminated from the translated sequence
trans.tab	Translation table, either 11 or 4

Details

Codons are by default translated according to translation table 11, i.e. the possible start codons are ATG, GTG or TTG and stop codons are TAA, TGA and TAG. The only alternative implemented here is translation table 4, which is used by some bacteria (e.g. Mycoplasma, Mesoplasma). If trans.tab is 4, the stop codon TGA istranslated to W (Tryptophan).

Value

A character vector of translated sequences.

Author(s)

Lars Snipen and Kristian Hovde Liland.

Examples

```
fa.file <- file.path(file.path(path.package("microseq"),"extdata"),"small.ffn")
fa <- readFasta(fa.file)
translate(fa$Sequence)

# Or, make use of dplyr to manipulate tables
readFasta(fa.file) %>%
  mutate(Protein = translate(Sequence)) -> fa.tbl
```

Index