Package ‘Biostrings’

May 29, 2024

Title Efficient manipulation of biological strings

Description Memory efficient string containers, string matching algorithms, and other utilities, for fast manipulation of large biological sequences or sets of sequences.

biocViews SequenceMatching, Alignment, Sequencing, Genetics, DataImport, DataRepresentation, Infrastructure

URL https://bioconductor.org/packages/Biostrings

BugReports https://github.com/Bioconductor/Biostrings/issues

Version 2.72.0

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.0.0), BiocGenerics (>= 0.37.0), S4Vectors (>= 0.27.12), IRanges (>= 2.31.2), XVector (>= 0.37.1), GenomInfoDb

Imports methods, utils, grDevices, stats, crayon

LinkingTo S4Vectors, IRanges, XVector

Enhances Rmpi

Suggests graphics, pwalign, BSgenome (>= 1.13.14), BSgenome.Celegans.UCSC.ce2 (>= 1.3.11), BSgenome.Dmelanogaster.UCSC.dm3 (>= 1.3.11), BSgenome.Hsapiens.UCSC.hg18, drosophila2probe, hgu95av2probe, hgu133aprobe, GenomicFeatures (>= 1.3.14), hgu95av2cdf, affy (>= 1.41.3), affydata (>= 1.11.5), RUnit, BiocStyle, knitr

VignetteBuilder knitr

Contents

SparseList-class.R  MIndex-class.R  lowlevel-matching.R
trimLRPatterns.R  matchProbePair.R  matchPWM.R  findPalindromes.R
PDict-class.R  matchPDict.R  XStringPartialMatches-class.R
XStringQuality-class.R  QualityScaledXStringSet.R
pmatchPattern.R  needwunsQS.R  MultipleAlignment.R  matchprobes.R
moved_to_pwalign.R  zzz.R

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

Author Hervé Pagès [aut, cre],
Patrick Aboyoun [aut],
Robert Gentleman [aut],
Saikat DebRoy [aut],
Vince Carey [ctb],
Nicolas Delhomme [ctb],
Felix Ernst [ctb],
Wolfgang Huber [ctb] ('matchprobes' vignette),
Haleema Khan [ctb] (Converted 'matchprobes' vignette from Sweave to
RMarkdown),
Aidan Lakshman [ctb],
Kieran O'Neill [ctb],
Valerie Obenchain [ctb],
Marcel Ramos [ctb],
Albert Vill [ctb],
Jen Wokaty [ctb] (Converted 'matchprobes' vignette from Sweave to
RMarkdown),
Erik Wright [ctb]

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

Contents

AAString-class .................................................. 4
AMINO_ACID_CODE ........................................ 5
Biostrings internals ........................................ 6
chartr ......................................................... 6
detail ......................................................... 8
dinucleotideFrequencyTest .................................. 8
DNAString-class .............................................. 10
findPalindromes .............................................. 11
GENETIC_CODE .............................................. 14
<table>
<thead>
<tr>
<th>Function/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSeq</td>
<td>17</td>
</tr>
<tr>
<td>gregexpr2</td>
<td>18</td>
</tr>
<tr>
<td>HNF4-alpha</td>
<td>19</td>
</tr>
<tr>
<td>injectHardMask</td>
<td>19</td>
</tr>
<tr>
<td>IUPAC_CODE_MAP</td>
<td>21</td>
</tr>
<tr>
<td>letter</td>
<td>22</td>
</tr>
<tr>
<td>letterFrequency</td>
<td>23</td>
</tr>
<tr>
<td>longestConsecutive</td>
<td>29</td>
</tr>
<tr>
<td>lowlevel-matching</td>
<td>30</td>
</tr>
<tr>
<td>MaskedXString-class</td>
<td>35</td>
</tr>
<tr>
<td>maskMotif</td>
<td>37</td>
</tr>
<tr>
<td>match-utils</td>
<td>40</td>
</tr>
<tr>
<td>matchLRPatterns</td>
<td>41</td>
</tr>
<tr>
<td>matchPattern</td>
<td>43</td>
</tr>
<tr>
<td>matchPDict</td>
<td>48</td>
</tr>
<tr>
<td>matchPDict-inexact</td>
<td>57</td>
</tr>
<tr>
<td>matchProbePair</td>
<td>61</td>
</tr>
<tr>
<td>matchprobes</td>
<td>62</td>
</tr>
<tr>
<td>matchPWM</td>
<td>63</td>
</tr>
<tr>
<td>MIndex-class</td>
<td>66</td>
</tr>
<tr>
<td>mise</td>
<td>67</td>
</tr>
<tr>
<td>moved_to_pwalign</td>
<td>68</td>
</tr>
<tr>
<td>MultipleAlignment-class</td>
<td>69</td>
</tr>
<tr>
<td>needwunsQS</td>
<td>74</td>
</tr>
<tr>
<td>nucleotideFrequency</td>
<td>75</td>
</tr>
<tr>
<td>padAndClip</td>
<td>80</td>
</tr>
<tr>
<td>PDict-class</td>
<td>82</td>
</tr>
<tr>
<td>pmatchPattern</td>
<td>86</td>
</tr>
<tr>
<td>QualityScaledXStringSet-class</td>
<td>86</td>
</tr>
<tr>
<td>replaceAt</td>
<td>89</td>
</tr>
<tr>
<td>replaceLetterAt</td>
<td>93</td>
</tr>
<tr>
<td>reverseComplement</td>
<td>95</td>
</tr>
<tr>
<td>RNAString-class</td>
<td>97</td>
</tr>
<tr>
<td>seqinfo-methods</td>
<td>99</td>
</tr>
<tr>
<td>toComplex</td>
<td>100</td>
</tr>
<tr>
<td>translate</td>
<td>101</td>
</tr>
<tr>
<td>trimLRPatterns</td>
<td>104</td>
</tr>
<tr>
<td>xscat</td>
<td>107</td>
</tr>
<tr>
<td>XString-class</td>
<td>108</td>
</tr>
<tr>
<td>XStringPartialMatches-class</td>
<td>110</td>
</tr>
<tr>
<td>XStringQuality-class</td>
<td>111</td>
</tr>
<tr>
<td>XStringSet-class</td>
<td>113</td>
</tr>
<tr>
<td>XStringSet-comparison</td>
<td>119</td>
</tr>
<tr>
<td>XStringSet-io</td>
<td>122</td>
</tr>
<tr>
<td>XStringSetList-class</td>
<td>129</td>
</tr>
<tr>
<td>XStringViews-class</td>
<td>131</td>
</tr>
<tr>
<td>yeastSEQCHR1</td>
<td>133</td>
</tr>
</tbody>
</table>

Index 135
AAString-class

AAString objects

Description

An AAString object allows efficient storage and manipulation of a long amino acid sequence.

Usage

AAString(x="", start=1, nchar=NA)

## Predefined constants:
AA_ALPHABET # full Amino Acid alphabet
AA_STANDARD # first 20 letters only
AA_PROTEINOGENIC # first 22 letters only

Arguments

x A single string.
start, nchar Where to start reading from in x and how many letters to read.

Details

The AAString class is a direct XString subclass (with no additional slot). Therefore all functions and methods described in the XString man page also work with an AAString object (inheritance).

Unlike the BString container that allows storage of any single string (based on a single-byte character set) the AAString container can only store a string based on the Amino Acid alphabet (see below).

The Amino Acid alphabet

This alphabet contains all letters from the Single-Letter Amino Acid Code (see ?AMINO_ACID_CODE) plus "*" (the stop letter), "-" (the gap letter), "+" (the hard masking letter), and "." (the not a letter or not available letter). It is stored in the AA_ALPHABET predefined constant (character vector).

The alphabet() function returns AA_ALPHABET when applied to an AAString object.

Constructor-like functions and generics

In the code snippet below, x can be a single string (character vector of length 1) or a BString object.

AAString(x="", start=1, nchar=NA): Tries to convert x into an AAString object by reading nchar letters starting at position start in x.

Accessor methods

In the code snippet below, x is an AAString object.

alphabet(x): If x is an AAString object, then return the Amino Acid alphabet (see above). See the corresponding man pages when x is a BString, DNAString or RNAString object.
AMINO_ACID_CODE

Display

The letters in an AAString object are colored when displayed by the show() method. Set global option Biostrings.coloring to FALSE to turn off this coloring.

Author(s)

H. Pagès

See Also

AMINO_ACID_CODE, letter, XString-class, alphabetFrequency

Examples

AA_ALPHABET
a <- AAString("MARKSLEMSIR*")
length(a)
alphabet(a)

AMINO_ACID_CODE

The Single-Letter Amino Acid Code

Description

Named character vector mapping single-letter amino acid representations to 3-letter amino acid representations.

See Also

AAString, GENETIC_CODE

Examples

## See all the 3-letter codes
AMINO_ACID_CODE

## Convert an AAString object to a vector of 3-letter amino acid codes
aa <- AAString("LANDEECQW")
AMINO_ACID_CODE[strsplit(as.character(aa), NULL)[[1]]]
Description

Biostrings objects, classes and methods that are not intended to be used directly.

Author(s)

H. Pagès

chartr

Replace letters in a sequence or set of sequences

Description

Replace letters in a sequence or set of sequences.

Usage

## S4 method for signature 'ANY,ANY,XString'
chartr(old, new, x)

replaceAmbiguities(x, new="N")

Arguments

old          A character string specifying the characters to be replaced.
new          A character string specifying the replacements. It must be a single letter for replaceAmbiguities.
x            The sequence or set of sequences to translate. If x is an XString, XStringSet, XStringViews or MaskedXString object, then the appropriate chartr method is called, otherwise the standard chartr R function is called.

Details

See ?chartr for the details.

Note that, unlike the standard chartr R function, the methods for XString, XStringSet, XStringViews and MaskedXString objects do NOT support character ranges in the specifications.

replaceAmbiguities() is a simple wrapper around chartr() that replaces all IUPAC ambiguities with N.

Value

An object of the same class and length as the original object.
chartr

See Also

- `chartr` in the base package.
- The `replaceAt` function for extracting or replacing arbitrary subsequences from/in a sequence or set of sequences.
- The `replaceLetterAt` function for a DNA-specific single-letter replacement functions useful for SNP injections.
- `IUPAC_CODE_MAP` for the mapping between IUPAC nucleotide ambiguity codes and their meaning.
- `alphabetFrequency` (and `uniqueLetters`) for tabulating letters in (and extracting the unique letters from) a sequence or set of sequences.
- The `XString`, `XStringSet`, `XStringViews`, and `MaskedXString` classes.

Examples

```r
## A BASIC chartr() EXAMPLE
x <- BString("MiXeD cAsE 123")
chartr("iXs", "why", x)

## TRANSFORMING DNA WITH BISULFITE (AND SEARCHING IT...)
library(BSgenome.Celegans.UCSC.ce2)
chrII <- Celegans["chrII"]
alphabetFrequency(chrII)
pattern <- DNAString("TGGGTGTATTTA")

# Transforming and searching the + strand
plus_strand <- chartr("C", "T", chrII)
alphabetFrequency(plus_strand)
matchPattern(pattern, plus_strand)
matchPattern(pattern, chrII)

# Transforming and searching the - strand
minus_strand <- chartr("G", "A", chrII)
alphabetFrequency(minus_strand)
matchPattern(reverseComplement(pattern), minus_strand)
matchPattern(reverseComplement(pattern), chrII)

# replaceAmbiguities()

dna <- DNAStringSet(c("TTKYYY-GR", "", "NAASACVT"))
dna
replaceAmbiguities(dna)
```
**detailed**

*Show (display) detailed object content*

### Description

This is a variant of `show`, offering a more detailed display of object content.

### Usage

```r
detail(x, ...)
```

### Arguments

- **x**: An object. The default simply invokes `show`.
- **...**: Additional arguments. The default definition makes no use of these arguments.

### Value

None; the function is invoked for its side effect (detailed display of object content).

### Author(s)

Martin Morgan

### Examples

```r
origMAlign <-
  readDNAMultipleAlignment(filepath =
    system.file("extdata",
      "msx2_mRNA.aln",
      package="Biostrings"),
    format="clustal")

detail(origMAlign)
```

---

**dinucleotideFrequencyTest**

*Pearson’s chi-squared Test and G-tests for String Position Dependence*

### Description

Performs Person’s chi-squared test, G-test, or William’s corrected G-test to determine dependence between two nucleotide positions.
Usage

dinucleotideFrequencyTest(x, i, j, test = c("chisq", "G", "adjG"),
    simulate.p.value = FALSE, B = 2000)

Arguments

x  A DNAStringSet or RNAStringSet object.

i, j  Single integer values for positions to test for dependence.

test  One of "chisq" (Person's chi-squared test), "G" (G-test), or "adjG" (William's corrected G-test). See Details section.

simulate.p.value  a logical indicating whether to compute p-values by Monte Carlo simulation.

B  an integer specifying the number of replicates used in the Monte Carlo test.

Details

The null and alternative hypotheses for this function are:

H0:  positions i and j are independent

H1:  otherwise

Let O and E be the observed and expected probabilities for base pair combinations at positions i and j respectively. Then the test statistics are calculated as:

test="chisq":  stat = sum(abs(O - E)^2/E)

test="G":  stat = 2 * sum(O * log(O/E))

test="adjG":  stat = 2 * sum(O * log(O/E))/q, where q = 1 + ((df - 1)^2 - 1)/(6*length(x)*(df - 2))

Under the null hypothesis, these test statistics are approximately distributed chi-squared(df = ((distinct bases at i) - 1) * ((distinct bases at j) - 1)).

Value

An htest object. See help(chisq.test) for more details.

Author(s)

P. Aboyoun

References


See Also

nucleotideFrequencyAt, XStringSet-class, chisq.test

Examples

data(HNF4alpha)
dinucleotideFrequencyTest(HNF4alpha, 1, 2)
dinucleotideFrequencyTest(HNF4alpha, 1, 2, test = "G")
dinucleotideFrequencyTest(HNF4alpha, 1, 2, test = "adjG")

DNAString-class

DNAString objects

Description

A DNAString object allows efficient storage and manipulation of a long DNA sequence.

Details

The DNAString class is a direct XString subclass (with no additional slot). Therefore all functions and methods described in the XString man page also work with a DNAString object (inheritance).

Unlike the BString container that allows storage of any single string (based on a single-byte character set) the DNAString container can only store a string based on the DNA alphabet (see below). In addition, the letters stored in a DNAString object are encoded in a way that optimizes fast search algorithms.

The DNA alphabet

This alphabet contains all letters from the IUPAC Extended Genetic Alphabet (see ?IUPAC_CODE_MAP) plus ";" (the gap letter), "+" (the hard masking letter), and "." (the not a letter or not available letter). It is stored in the DNA_ALPHABET predefined constant (character vector).

The alphabet() function returns DNA_ALPHABET when applied to a DNAString object.

Constructor-like functions and generics

In the code snippet below, x can be a single string (character vector of length 1), a BString object or an RNAString object.

DNAString(x="", start=1, nchar=NA): Tries to convert x into a DNAString object by reading nchar letters starting at position start in x.
Accessor methods

In the code snippet below, \( x \) is a DNAString object.

\[
\text{alphabet}(x, \text{baseOnly}=\text{FALSE}): \text{If } x \text{ is a DNAString object, then return the DNA alphabet (see above). See the corresponding man pages when } x \text{ is a BString, RNAString or AString object.}
\]

Display

The letters in a DNAString object are colored when displayed by the \text{show()} method. Set global option \text{Biostrings.coloring} to \text{FALSE} to turn off this coloring.

Author(s)

H. Pagès

See Also

- The \text{DNAStringSet} class to represent a collection of DNAString objects.
- The \text{XString} and \text{RNAString} classes.
- \text{reverseComplement}
- \text{alphabetFrequency}
- \text{IUPAC_CODE_MAP}
- \text{letter}

Examples

\[
\begin{align*}
\text{DNA_BASES} \\
\text{DNA_ALPHABET} \\
dna & \leftarrow \text{DNAString("TTGAAAA-CTC-N")} \\
dna & \text{ # 'options(Biostrings.coloring=FALSE)' to turn off coloring} \\
\text{length}(dna) \\
\text{alphabet}(dna) & \text{ # DNA_ALPHABET} \\
\text{alphabet}(dna, \text{baseOnly}=\text{TRUE}) & \text{ # DNA_BASES}
\end{align*}
\]

findPalindromes  
\textbf{Searching a sequence for palindromes}

Description

The \text{findPalindromes} function can be used to find palindromic regions in a sequence.

\text{palindromeArmLength}, \text{palindromeLeftArm}, \text{and palindromeRightArm} are utility functions for operating on palindromic sequences. They should typically be used on the output of \text{findPalindromes}. 

findPalindromes

Usage

findPalindromes(subject, min.armlength=4,
    max.looplength=1, min.looplength=0, max.mismatch=0,
    allow.wobble=FALSE)
palindromeArmLength(x, max.mismatch=0, allow.wobble=FALSE)
palindromeLeftArm(x, max.mismatch=0, allow.wobble=FALSE)
palindromeRightArm(x, max.mismatch=0, allow.wobble=FALSE)

Arguments

subject An XString object containing the subject string, or an XStringViews object.
min.armlength An integer giving the minimum length of the arms of the palindromes to search
for.
max.looplength An integer giving the maximum length of "the loop" (i.e the sequence separ-
arating the 2 arms) of the palindromes to search for. Note that by default (max.looplength=1), findPalindromes will search for strict palindromes only.
min.looplength An integer giving the minimum length of "the loop" of the palindromes to search
for.
max.mismatch The maximum number of mismatching letters allowed between the 2 arms of
the palindromes to search for.
allow.wobble Logical indicating whether wobble base pairs (G/U or G/T base pairings) should
be treated as mismatches (the default) or matches.
x An XString object containing a 2-arm palindrome, or an XStringViews object
containing a set of 2-arm palindromes.

Details

The findPalindromes function finds palindromic substrings in a subject string. The palindromes
that can be searched for are either strict palindromes or 2-arm palindromes (the former being a
particular case of the latter) i.e. palindromes where the 2 arms are separated by an arbitrary sequence
called "the loop".

If the subject string is a nucleotide sequence (i.e. DNA or RNA), the 2 arms must contain sequences
that are reverse complement from each other. Otherwise, they must contain sequences that are the
same.

Value

findPalindromes returns an XStringViews object containing all palindromes found in subject
(one view per palindromic substring found).

palindromeArmLength returns the arm length (integer) of the 2-arm palindrome x. It will raise
an error if x has no arms. Note that any sequence could be considered a 2-arm palindrome if
we were OK with arms of length 0 but we are not: x must have arms of length greater or equal
to 1 in order to be considered a 2-arm palindrome. When applied to an XStringViews object x,
palindromeArmLength behaves in a vectorized fashion by returning an integer vector of the same
length as x.
**findPalindromes**

palindromeLeftArm returns an object of the same class as the original object x and containing the left arm of x.

palindromeRightArm does the same as palindromeLeftArm but on the right arm of x.

Like palindromeArmLength, both palindromeLeftArm and palindromeRightArm will raise an error if x has no arms. Also, when applied to an XStringViews object x, both behave in a vectorized fashion by returning an XStringViews object of the same length as x.

**Author(s)**

H. Pagès, with contributions from Erik Wright and Thomas McCarthy

**See Also**

maskMotif, matchPattern, matchLRPatterns, matchProbePair, XStringViews-class, DNAString-class

**Examples**

x0 <- BString("abbbabcbbabcccabbbbcbaabbbabcacab")

pals0a <- findPalindromes(x0, min.armlength=3, max.looplength=5)
pals0a
palindromeArmLength(pals0a)
palindromeLeftArm(pals0a)
palindromeRightArm(pals0a)

pals0b <- findPalindromes(x0, min.armlength=9, max.looplength=5, max.mismatch=3)
pals0b
palindromeArmLength(pals0b, max.mismatch=3)
palindromeLeftArm(pals0b, max.mismatch=3)
palindromeRightArm(pals0b, max.mismatch=3)

## Whitespaces matter:
x1 <- BString("Delia saw I was aileD")
palindromeArmLength(x1)
palindromeLeftArm(x1)
palindromeRightArm(x1)

x2 <- BString("was it a car or a cat I saw")
palindromeArmLength(x2)
palindromeLeftArm(x2)
palindromeRightArm(x2)

## On a DNA or RNA sequence:
x3 <- DNAString("CCGAAAACCAGTGCTGCGAGC")
findPalindromes(x3)
findPalindromes(RNAString(x3))

## Note that palindromes can be nested:
x4 <- DNAString("ACGTNAAGCTCCAAAATTTTCCACGGTNTACGT")
findPalindromes(x4, max.looplength=19)

## Treat wobble base pairings as matches:
x5 <- RNAString("AUGUUNNAGGCGU")
findPalindromes(x5, max.looplength=4, min.looplength=4)
findPalindromes(x5, max.looplength=4, min.looplength=4, max.mismatch=2)
findPalindromes(x5, max.looplength=4, min.looplength=4, allow.wobble=TRUE)

## A real use case:
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- DmelanogasterchrX
chrX_pals0 <- findPalindromes(chrX, min.armlength=40, max.looplength=80)
chrX_pals0
palindromeArmLength(chrX_pals0) # 251 70 262

## Allowing up to 2 mismatches between the 2 arms:
chrX_pals2 <- findPalindromes(chrX, min.armlength=40, max.looplength=80,
max.mismatch=2)
chrX_pals2
palindromeArmLength(chrX_pals2, max.mismatch=2) # 254 77 44 48 40 264

# The Standard Genetic Code and its known variants

Description

Two predefined objects (GENETIC_CODE and RNA_GENETIC_CODE) that represent The Standard Genetic Code.

Other genetic codes are stored in predefined table GENETIC_CODE_TABLE from which they can conveniently be extracted with getGeneticCode.

Usage

## The Standard Genetic Code:
GENETIC_CODE
RNA_GENETIC_CODE

## All the known genetic codes:
GENETIC_CODE_TABLE
getGeneticCode(id_or_name2="1", full.search=FALSE, as.data.frame=FALSE)

Arguments

id_or_name2 A single string that uniquely identifies the genetic code to extract. Should be one of the values in the id or name2 columns of GENETIC_CODE_TABLE.

full.search By default, only the id and name2 columns of GENETIC_CODE_TABLE are searched for an exact match with id_or_name2. If full.search is TRUE, then the search is extended to the name column of GENETIC_CODE_TABLE and id_or_name2 only needs to be a substring of one of the names in that column (also case is ignored).
Should the genetic code be returned as a data frame instead of a named character vector?

Details

Formally, a genetic code is a mapping between the 64 tri-nucleotide sequences (called codons) and amino acids.

The Standard Genetic Code (a.k.a. The Canonical Genetic Code, or simply The Genetic Code) is the particular mapping that encodes the vast majority of genes in nature. GENETIC_CODE and RNA_GENETIC_CODE are predefined named character vectors that represent this mapping.

All the known genetic codes are summarized in GENETIC_CODE_TABLE, which is a predefined data frame with one row per known genetic code. Use getGeneticCode to extract one genetic code at a time from this object.

Value

GENETIC_CODE and RNA_GENETIC_CODE are both named character vectors of length 64 (the number of all possible tri-nucleotide sequences) where each element is a single letter representing either an amino acid or the stop codon "*" (aka termination codon).

The names of the GENETIC_CODE vector are the DNA codons i.e. the tri-nucleotide sequences (directed 5' to 3') that are assumed to belong to the "coding DNA strand" (aka "sense DNA strand" or "non-template DNA strand") of the gene.

The names of the RNA_GENETIC_CODE are the RNA codons i.e. the tri-nucleotide sequences (directed 5' to 3') that are assumed to belong to the mRNA of the gene.

Note that the values in the GENETIC_CODE and RNA_GENETIC_CODE vectors are the same, only their names are different. The names of the latter are those of the former where all occurrences of T (thymine) have been replaced by U (uracil).

Finally, both vectors have an alt_init_codons attribute on them, that lists the alternative initiation codons. Note that codons that always translate to M (Methionine) (e.g. ATG in GENETIC_CODE or AUG in RNA_GENETIC_CODE) are omitted from the alt_init_codons attribute.

GENETIC_CODE_TABLE is a data frame that contains all the known genetic codes listed at ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt. The data frame has one row per known genetic code and the 5 following columns:

- name: The long and very descriptive name of the genetic code.
- name2: The short name of the genetic code (not all genetic codes have one).
- id: The id of the genetic code.
- AAs: A 64-character string representing the genetic code itself in a compact form (i.e. one letter per codon, the codons are assumed to be ordered like in GENETIC_CODE).
- Starts: A 64-character string indicating the Initiation Codons.

By default (i.e. when as.data.frame is set to FALSE), getGeneticCode returns a named character vector of length 64 similar to GENETIC_CODE i.e. it contains 1-letter strings from the Amino Acid alphabet (see ?AA_ALPHABET) and its names are identical to names(GENETIC_CODE). In addition it has an attribute on it, the alt_init_codons attribute, that lists the alternative initiation
codons. Note that codons that always translate to M (Methionine) (e.g. ATG) are omitted from the alt_init_codons attribute.

When as.data.frame is set to TRUE, getGeneticCode returns a data frame with 64 rows (one per codon), rownames (3-letter strings representing the codons), and the 2 following columns:

- **AA**: A 1-letter string from the Amino Acid alphabet (see ?AA_ALPHABET) representing the amino acid mapped to the codon (“*” is used to mark the stop codon).
- **Start**: A 1-letter string indicating an alternative mapping for the codon i.e. what amino acid the codon is mapped to when it’s the first translated codon.

The rownames of the data frame are identical to names(GENETIC_CODE).

Author(s)

H. Pagès

References

All the known genetic codes are described here:


The "official names" of the various codes ("Standard", "SGC0", "Vertebrate Mitochondrial", "SGC1", etc..) and their ids (1, 2, etc...) were taken from the print-form ASN.1 version of the above document (version 4.0 at the time of this writing):


See Also

- AA_ALPHABET and AMINO_ACID_CODE.
- The translate and trinucleotideFrequency functions.
- DNAString, RNAString, and AAString objects.

Examples

```r
## THE STANDARD GENETIC CODE

GENETIC_CODE

## Codon ATG is *always* translated to M (Methionine)
GENETIC_CODE["ATG"]

## Codons TTG and CTG are "normally" translated to L except when they are
## the first translated codon (a.k.a. start codon or initiation codon),
## in which case they are translated to M:
attr(GENETIC_CODE, "alt_init_codons")
GENETIC_CODE["TTG"]
GENETIC_CODE["CTG"]
```
getSeq

```r

sort(table(GENETIC_CODE))  # the same amino acid can be encoded by 1
   # to 6 different codons

RNA_GENETIC_CODE
all(GENETIC_CODE == RNA_GENETIC_CODE)  # TRUE

## ---------------------------------------------------------------------
## ALL THE KNOWN GENETIC CODES
## ---------------------------------------------------------------------

GENETIC_CODE_TABLE[1:3 , ]

getGeneticCode("SGC0")  # The Standard Genetic Code, again
stopifnot(identical(getGeneticCode("SGC0"), GENETIC_CODE))

getGeneticCode("SGC1")  # Vertebrate Mitochondrial

getGeneticCode("ascidian", full.search=TRUE)  # Ascidian Mitochondrial

## ---------------------------------------------------------------------
## EXAMINE THE DIFFERENCES BETWEEN THE STANDARD CODE AND A NON-STANDARD
## ONE
## ---------------------------------------------------------------------

idx <- which(GENETIC_CODE != getGeneticCode("SGC1"))
rbind(Standard=GENETIC_CODE[idx], SGC1=getGeneticCode("SGC1")[idx])
```

---

**Description**

A generic function for extracting a set of sequences (or subsequences) from a sequence container like a `BSgenome` object or other.

**Usage**

```r
getSeq(x, ...)
```

**Arguments**

- `x` A `BSgenome` object or any other supported object. Do `showMethods("getSeq")` to get the list of all supported types for `x`.
- `...` Any additional arguments needed by the specialized methods.
Value

An `XString` object or an `XStringSet` object or a character vector containing the extracted sequence(s). See man pages of individual methods for the details e.g. with `?getSeq,BSgenome-method` to access the man page of the method for `BSgenome` objects (make sure the BSgenome package is loaded first).

See Also

`getSeq,BSgenome-method, XString-class, XStringSet-class`

Examples

```r
## Note that you need to load the package(s) defining the specialized
## methods to have showMethods() display them and to be able to access
## their man pages:
library(BSgenome)
showMethods("getSeq")
```

Description

This is a replacement for the standard gregexpr function that does exact matching only. Standard `gregexpr()` misses matches when they are overlapping. The `gregexpr2` function finds all matches but it only works in "fixed" mode i.e. for exact matching (regular expressions are not supported).

Usage

```r
gregexpr2(pattern, text)
```

Arguments

- `pattern` character string to be matched in the given character vector
- `text` a character vector where matches are sought

Value

A list of the same length as `text` each element of which is an integer vector as in `gregexpr`, except that the starting positions of all (even overlapping) matches are given. Note that, unlike `gregexpr`, `gregexpr2` doesn’t attach a "match.length" attribute to each element of the returned list because, since it only works in "fixed" mode, then all the matches have the length of the pattern. Another difference with `gregexpr` is that with `gregexpr2`, the `pattern` argument must be a single (non-NA, non-empty) string.

Author(s)

H. Pagès
**HNF4alpha**

**See Also**

`gregexpr`, `matchPattern`

**Examples**

```r
gregexpr("aa", c("XaaaYaa", "a"), fixed=TRUE)
gregexpr2("aa", c("XaaaYaa", "a"))
```

---

**HNF4alpha**

Known HNF4alpha binding sequences

**Description**

Seventy one known HNF4alpha binding sequences

**Details**

A DNAStringSet containing 71 known binding sequences for HNF4alpha.

**Author(s)**

P. Aboyoun

**References**


**Examples**

```r
data(HNF4alpha)
HNF4alpha
```

---

**injectionHardMask**

Injecting a hard mask in a sequence

**Description**

`injectionHardMask` allows the user to "fill" the masked regions of a sequence with an arbitrary letter (typically the "+" letter).

**Usage**

```r
injectionHardMask(x, letter="+")
```
Arguments

\( x \) A MaskedXString or XStringViews object.

\( \text{letter} \) A single letter.

Details

The name of the \texttt{injectHardMask} function was chosen because of the primary use that it is intended for: converting a pile of active "soft masks" into a "hard mask". Here the pile of active "soft masks" refers to the active masks that have been put on top of a sequence. In Biostrings, the original sequence and the masks defined on top of it are bundled together in one of the dedicated containers for this: the \texttt{MaskedBString}, \texttt{MaskedDNAString}, \texttt{MaskedRNAString} and \texttt{MaskedAAString} containers (this is the \texttt{MaskedXString} family of containers). The original sequence is always stored unmodified in a \texttt{MaskedXString} object so no information is lost. This allows the user to activate/deactivate masks without having to worry about losing the letters that are in the regions that are masked/unmasked. Also this allows better memory management since the original sequence never needs to be copied, even when the set of active/inactive masks changes.

However, there are situations where the user might want to \textit{really} get rid of the letters that are in some particular regions by replacing them with a junk letter (e.g. "+") that is guaranteed to not interfere with the analysis that s/he is currently doing. For example, it’s very likely that a set of motifs or short reads will not contain the "+" letter (this could easily be checked) so they will never hit the regions filled with "+". In a way, it’s like the regions filled with "+" were masked but we call this kind of masking "hard masking".

Some important differences between "soft" and "hard" masking:

- \texttt{injectHardMask} creates a (modified) copy of the original sequence. Using "soft masking" does not.
- A function that is "mask aware" like \texttt{alphabetFrequency} or \texttt{matchPattern} will really skip the masked regions when "soft masking" is used i.e. they will not walk thru the regions that are under active masks. This might lead to some speed improvements when a high percentage of the original sequence is masked. With "hard masking", the entire sequence is walked thru.
- Matches cannot span over masked regions with "soft masking". With "hard masking" they can.

Value

An \texttt{XString} object of the same length as the original object \( x \) if \( x \) is a \texttt{MaskedXString} object, or of the same length as \( \text{subject}(x) \) if it’s an \texttt{XStringViews} object.

Author(s)

H. Pagès

See Also

\texttt{maskMotif}, \texttt{MaskedXString-class}, \texttt{replaceLetterAt}, \texttt{chartr}, \texttt{XString}, \texttt{XStringViews-class}
IUPAC_CODE_MAP

Examples

```r
## A. WITH AN XStringViews OBJECT
v2 <- Views("abCDefgHIJK", start=c(8, 3), end=c(14, 4))
injectHardMask(v2)
injectHardMask(v2, letter="=")
```

```r
## B. WITH A MaskedXString OBJECT
mask0 <- Mask(mask.width=29, start=c(3, 10, 25), width=c(6, 8, 5))
x <- DNAString("ACACAAGTAGATAGNACTNGAGAGACGC")
masks(x) <- mask0
x
subject <- injectHardMask(x)
```

```r
## Matches can span over masked regions with "hard masking":
matchPattern("ACggggggA", subject, max.mismatch=6)
## but not with "soft masking":
matchPattern("ACggggggA", x, max.mismatch=6)
```

---

IUPAC_CODE_MAP  The IUPAC Extended Genetic Alphabet

Description

The IUPAC_CODE_MAP named character vector contains the mapping from the IUPAC nucleotide ambiguity codes to their meaning.

The mergeIUPACLetters function provides the reverse mapping.

Usage

```r
IUPAC_CODE_MAP
mergeIUPACLetters(x)
```

Arguments

```r
x  A vector of non-empty character strings made of IUPAC letters.
```

Details

IUPAC nucleotide ambiguity codes are used for representing sequences of nucleotides where the exact nucleotides that occur at some given positions are not known with certainty.
Value

IUPAC_CODE_MAP is a named character vector where the names are the IUPAC nucleotide ambiguity codes and the values are their corresponding meanings. The meaning of each code is described by a string that enumerates the base letters ("A", "C", "G" or "T") associated with the code.

The value returned by mergeIUPACLetters is an unnamed character vector of the same length as its argument x where each element is an IUPAC nucleotide ambiguity code.

Author(s)

H. Pagès

References

http://www.chick.manchester.ac.uk/SiteSeer/IUPAC\_codes.html


See Also

DNAString, RNAString

Examples

```r
IUPAC_CODE_MAP
some_iupac_codes <- c("R", "M", "G", "N", "V")
IUPAC_CODE_MAP[some_iupac_codes]
mergeIUPACLetters(IUPAC_CODE_MAP[some_iupac_codes])

mergeIUPACLetters(c("Ca", "Acc", "aA", "MAAmC", "gM", "AB", "bS", "mk"))
```

---

**Description**

Extract a substring from a string by picking up individual letters by their position.

**Usage**

```r
letter(x, i)
```

**Arguments**

- `x`: A character vector, or an XString, XStringViews or MaskedXString object.
- `i`: An integer vector with no NAs.
Details

Unlike with the `substr` or `substring` functions, `i` must contain valid positions.

Value

A character vector of length 1 when `x` is an `XString` or `MaskedXString` object (the masks are ignored for the latter).

A character vector of the same length as `x` when `x` is a character vector or an `XStringViews` object.

Note that, because `i` must contain valid positions, all non-NA elements in the result are guaranteed to have exactly `length(i)` characters.

See Also

`subseq`, `XString-class`, `XStringViews-class`, `MaskedXString-class`

Examples

```r
x <- c("abcd", "ABC")
i <- c(3, 1, 1, 2, 1)

## With a character vector:
letter(x[1], 3:1)
letter(x, 3)
letter(x, i)
letter(x, 4)  # Error!

## With a BString object:
letter(BString(x[1]), i)  # returns a character vector
BString(x[1])[i]  # returns a BString object

## With an XStringViews object:
x2 <- as(BStringSet(x), "Views")
letter(x2, i)
```

letterFrequency

`Calculate the frequency of letters in a biological sequence, or the consensus matrix of a set of sequences`

Description

Given a biological sequence (or a set of biological sequences), the `alphabetFrequency` function computes the frequency of each letter of the relevant alphabet.

`letterFrequency` is similar, but more compact if one is only interested in certain letters. It can also tabulate letters "in common".

`letterFrequencyInSlidingView` is a more specialized version of `letterFrequency` for (non-masked) `XString` objects. It tallies the requested letter frequencies for a fixed-width view, or window, that is conceptually slide along the entire input sequence.
The consensusMatrix function computes the consensus matrix of a set of sequences, and the consensusString function creates the consensus sequence from the consensus matrix based upon specified criteria.

In this man page we call "DNA input" (or "RNA input") an XString, XStringSet, XStringViews or MaskedXString object of base type DNA (or RNA).

Usage

alphabetFrequency(x, as.prob=FALSE, ...)  
hasOnlyBaseLetters(x)  
uniqueLetters(x)

letterFrequency(x, letters, OR="|", as.prob=FALSE, ...)  
letterFrequencyInSlidingView(x, view.width, letters, OR="|", as.prob=FALSE)

consensusMatrix(x, as.prob=FALSE, shift=0L, width=NULL, ...)

## S4 method for signature 'matrix'
consensusString(x, ambiguityMap=“, threshold=0.5)  
## S4 method for signature 'DNAStringSet'
consensusString(x, ambiguityMap=IUPAC_CODE_MAP,  
threshold=0.25, shift=0L, width=NULL)  
## S4 method for signature 'RNAStringSet'
consensusString(x,  
ambiguityMap= 
structure(as.character(RNAStringSet(DNAStringSet(IUPAC_CODE_MAP))))),  
names= 
as.character(RNAStringSet(DNAStringSet(names(IUPAC_CODE_MAP))))),  
threshold=0.25, shift=0L, width=NULL)

Arguments

x
  An XString, XStringSet, XStringViews or MaskedXString object for alphabetFrequency, letterFrequency, or uniqueLetters.  
DNA or RNA input for hasOnlyBaseLetters.  
An XString object for letterFrequencyInSlidingView.  
A character vector, or an XStringSet or XStringViews object for consensusMatrix.  
A consensus matrix (as returned by consensusMatrix), or an XStringSet or XStringViews object for consensusString.

as.prob
  If TRUE then probabilities are reported, otherwise counts (the default).

view.width
  For letterFrequencyInSlidingView, the constant (e.g. 35, 48, 1000) size of the "window" to slide along x. The specified letters are tabulated in each window of length view.width. The rows of the result (see value) correspond to the various windows.

letters
  For letterFrequency or letterFrequencyInSlidingView, a character vector (e.g. "C", "CG", c("C", "G")) giving the letters to tabulate. When x is DNA or
RNA input, letters must come from alphabet(x). Except with OR=0, multi-character elements of letters (‘nchar’ > 1) are taken as groupings of letters into subsets, to be tabulated in common (“or”’d), as if their alphabetFrequency’s were added (Arithmetic). The columns of the result (see value) correspond to the individual and sets of letters which are counted separately. Unrelated (and, with some post-processing, related) counts may of course be obtained in separate calls.

OR

For letterFrequency or letterFrequencyInSlidingView, the string (default |) to use as a separator in forming names for the "grouped" columns, e.g. "CIG". The otherwise exceptional value 0 (zero) disables or’ing and is provided for convenience, allowing a single multi-character string (or several strings) of letters that should be counted separately. If some but not all letters are to be counted separately, they must reside in separate elements of letters (with ‘nchar’ 1 unless they are to be grouped with other letters), and OR cannot be 0.

ambiguityMap

Either a single character to use when agreement is not reached or a named character vector where the names are the ambiguity characters and the values are the combinations of letters that comprise the ambiguity (e.g. link{IUPAC_CODE_MAP}). When ambiguityMap is a named character vector, occurrences of ambiguous letters in x are replaced with their base alphabet letters that have been equally weighted to sum to 1. (See Details for some examples.)

threshold

The minimum probability threshold for an agreement to be declared. When ambiguityMap is a single character, threshold is a single number in (0, 1]. When ambiguityMap is a named character vector (e.g. link{IUPAC_CODE_MAP}), threshold is a single number in (0, 1/sum(nchar(ambiguityMap) == 1)].

...Further arguments to be passed to or from other methods.

For the XStringViews and XStringSet methods, the collapse argument is accepted.

Except for letterViews or letterFrequencyInSlidingView, and with DNA or RNA input, the baseOnly argument is accepted. If baseOnly is TRUE, the returned vector (or matrix) only contains the frequencies of the letters that belong to the "base" alphabet of x i.e. to the alphabet returned by alphabet(x, baseOnly=TRUE).

shift

An integer vector (recycled to the length of x) specifying how each sequence in x should be (horizontally) shifted with respect to the first column of the consensus matrix to be returned. By default (shift=0), each sequence in x has its first letter aligned with the first column of the matrix. A positive shift value means that the corresponding sequence must be shifted to the right, and a negative shift value that it must be shifted to the left. For example, a shift of 5 means that it must be shifted 5 positions to the right (i.e. the first letter in the sequence must be aligned with the 6th column of the matrix), and a shift of -3 means that it must be shifted 3 positions to the left (i.e. the 4th letter in the sequence must be aligned with the first column of the matrix).

width

The number of columns of the returned matrix for the consensusMatrix method for XStringSet objects. When width=NULL (the default), then this method returns a matrix that has just enough columns to have its last column aligned with the rightmost letter of all the sequences in x after those sequences have been...
shifted (see the shift argument above). This ensures that any wider consensus matrix would be a "padded with zeros" version of the matrix returned when width=NULL.

The length of the returned sequence for the consensusString method for XStringSet objects.

Details

alphabetFrequency, letterFrequency, and letterFrequencyInSlidingView are generic functions defined in the Biostrings package.

letterFrequency is similar to alphabetFrequency but specific to the letters of interest, hence more compact, especially with OR non-zero.

letterFrequencyInSlidingView yields the same result, on the sequence x, that letterFrequency would, if applied to the hypothetical (and possibly huge) XStringViews object consisting of all the intervals of length view.width on x. Taking advantage of the knowledge that successive "views" are nearly identical, for letter counting purposes, it is both lighter and faster.

For letterFrequencyInSlidingView, a masked (MaskedXString) object x is only supported through a cast to an (ordinary) XString such as unmasked (which includes its masked regions).

When consensusString is executed with a named character ambiguityMap argument, it weights each input string equally and assigns an equal probability to each of the base letters represented by an ambiguity letter. So for DNA and a threshold of 0.25, a "G" and an "R" would result in an "R" since 1/2 "G" + 1/2 "R" = 3/4 "G" + 1/4 "A" => "R"; two "G"'s and one "R" would result in a "G" since 2/3 "G" + 1/3 "R" = 5/6 "G" + 1/6 "A" => "G"; and one "A" and one "N" would result in an "N" since 1/2 "A" + 1/2 "N" = 5/8 "A" + 1/8 "C" + 1/8 "G" + 1/8 "T" => "N".

Value

alphabetFrequency returns an integer vector when x is an XString or MaskedXString object. When x is an XStringSet or XStringViews object, then it returns an integer matrix with length(x) rows where the i-th row contains the frequencies for x[[i]]. If x is a DNA, RNA, or AA input, then the returned vector is named with the letters in the alphabet. If the baseOnly argument is TRUE, then the returned vector has only 5 elements for DNA/RNA input (4 elements corresponding to the 4 nucleotides + the 'other' element) and 21 elements for AA input (20 elements corresponding to the 20 base amino acids + the 'other' element).

letterFrequency returns, similarly, an integer vector or matrix, but restricted and/or collated according to letters and OR.

letterFrequencyInSlidingView returns, for an XString object x of length (nchar) L, an integer matrix with L-view.width+1 rows, the i-th of which holding the letter frequencies of substring(x, i, i+view.width-1).

hasOnlyBaseLetters returns TRUE or FALSE indicating whether or not x contains only base letters (i.e. As, Cs, Gs and Ts for DNA input, As, Cs, Gs and Us for RNA input, or any of the 20 standard amino acids for AA input).

uniqueLetters returns a vector of 1-letter or empty strings. The empty string is used to represent the nul character if x happens to contain any. Note that this can only happen if the base class of x is BString.

An integer matrix with letters as row names for consensusMatrix.
A standard character string for consensusString.

Author(s)
H. Pagès and P. Aboyoun; H. Jaffee for letterFrequency and letterFrequencyInSlidingView

See Also
alphabet, coverage, oligonucleotideFrequency, countPDict, XString-class, XStringSet-class, XStringViews-class, MaskedXString-class, strsplit

Examples

```r
# --------------------------
# alphabetFrequency()
# --------------------------
data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)

alphabetFrequency(yeast1)
alphabetFrequency(yeast1, baseOnly=TRUE)

hasOnlyBaseLetters(yeast1)
uniqueLetters(yeast1)

# With input made of multiple sequences:
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
alphabetFrequency(probes[1:50], baseOnly=TRUE)
alphabetFrequency(probes, baseOnly=TRUE, collapse=TRUE)

# --------------------------
# letterFrequency()
# --------------------------
letterFrequency(probes[[1]], letters="ACGT", OR=0)
base_letters <- alphabet(probes, baseOnly=TRUE)
base_letters
letterFrequency(probes[[1]], letters=base_letters, OR=0)
base_letter_freqs <- letterFrequency(probes, letters=base_letters, OR=0)
head(base_letter_freqs)
GC_content <- letterFrequency(probes, letters="CG")
head(GC_content)
letterFrequency(probes, letters="CG", collapse=TRUE)

# --------------------------
# letterFrequencyInSlidingView()
# --------------------------
data(yeastSEQCHR1)
x <- DNAString(yeastSEQCHR1)
view.width <- 48
letters <- c("A", "CG")
two_columns <- letterFrequencyInSlidingView(x, view.width, letters)
```
head(two_columns)
tail(two_columns)
three_columns <- letterFrequencyInSlidingView(x, view.width, letters, OR=0)
head(three_columns)
tail(three_columns)
stopifnot(identical(two_columns[, "C"|"G"],
        three_columns[, "C"] + three_columns[, "G"]))

## Note that, alternatively, 'three_columns' can also be obtained by
## creating the views on 'x' (as a Views object) and by calling
## alphabetFrequency() on it. But, of course, that is be *much* less
## efficient (both, in terms of memory and speed) than using
## letterFrequencyInSlidingView():

v <- Views(x, start=seq_len(length(x) - view.width + 1), width=view.width)
v
three_columns2 <- alphabetFrequency(v, baseOnly=TRUE)[ , c("A", "C", "G")]
stopifnot(identical(three_columns2, three_columns))

## Set the width of the view to length(x) to get the global frequencies:
letterFrequencyInSlidingView(x, letters="ACGTN", view.width=length(x), OR=0)

## consensus*()
## ---------------------------------------------------------------------
## consensusString()
## ---------------------------------------------------------------------
## Read in ORF data:
file <- system.file("extdata", "someORF.fa", package="Biostrings")
orf <- readDNAStringSet(file)

## To illustrate, the following example assumes the ORF data
## to be aligned for the first 10 positions (patently false):
orf10 <- DNAStringSet(orf, end=10)
consensusMatrix(orf10, baseOnly=TRUE)

## The following example assumes the first 10 positions to be aligned
## after some incremental shifting to the right (patently false):
consensusMatrix(orf10, baseOnly=TRUE, shift=0:6)
consensusMatrix(orf10, baseOnly=TRUE, shift=0:6, width=10)

## For the character matrix containing the "exploded" representation
## of the strings, do:
as.matrix(orf10, use.names=FALSE)

## consensusMatrix() can be used to just compute the alphabet frequency
## for each position in the input sequences:
consensusMatrix(probes, baseOnly=TRUE)

## After sorting, the first 5 probes might look similar (at least on
## their first bases):
consensusString(sort(probes)[1:5])
consensusString(sort(probes)[1:5], ambiguityMap = "N", threshold = 0.5)

## Consensus involving ambiguity letters in the input strings
consensusString(DNAStringSet(c("NNNN","ACTG")))
longestConsecutive

longestConsecutive(seq, letter)

Arguments

seq Character vector.
letter Character vector of length 1, containing one single character.

Details

This function has been deprecated and won’t be replaced.
The elements of x can be in upper case, lower case or mixed. NAs are handled.

Value

An integer vector of the same length as x.

Author(s)

W. Huber
Examples

```r
## Not run:
v = c("AAACTGTGFG", "GGGAATT", "CCAAAAAAAAAAATT")
longestConsecutive(v, "A")

## End(Not run)
```

Description

In this man page we define precisely and illustrate what a "match" of a pattern `P` in a subject `S` is in the context of the Biostrings package. This definition of a "match" is central to most pattern matching functions available in this package: unless specified otherwise, most of them will adhere to the definition provided here.

`hasLetterAt` checks whether a sequence or set of sequences has the specified letters at the specified positions.

`neditAt`, `isMatchingAt` and `which.isMatchingAt` are low-level matching functions that only look for matches at the specified positions in the subject.

Usage

```r
hasLetterAt(x, letter, at, fixed=TRUE)

# neditAt() and related utils:
neditAt(pattern, subject, at=1,
with.indels=FALSE, fixed=TRUE)
neditStartingAt(pattern, subject, starting.at=1,
with.indels=FALSE, fixed=TRUE)
neditEndingAt(pattern, subject, ending.at=1,
with.indels=FALSE, fixed=TRUE)

# isMatchingAt() and related utils:
isMatchingAt(pattern, subject, at=1,
max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE)
isMatchingStartingAt(pattern, subject, starting.at=1,
max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE)
isMatchingEndingAt(pattern, subject, ending.at=1,
max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE)

# which.isMatchingAt() and related utils:
which.isMatchingAt(pattern, subject, at=1,
max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
follow.index=FALSE, auto.reduce.pattern=FALSE)
which.isMatchingStartingAt(pattern, subject, starting.at=1,
```
lowlevel-matching

max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
follow.index=FALSE, auto.reduce.pattern=FALSE)
which.isMatchingEndingAt(pattern, subject, ending.at=1,
max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
follow.index=FALSE, auto.reduce.pattern=FALSE)

Arguments

x          A character vector, or an XString or XStringSet object.
letter     A character string or an XString object containing the letters to check.
at, starting.at, ending.at
            An integer vector specifying the starting (for starting.at and at) or ending (for ending.at) positions of the pattern relatively to the subject. With auto.reduce.pattern (below), either a single integer or a constant vector of length nchar(pattern) (below), to which the former is immediately converted.
            For the hasLetterAt function, letter and at must have the same length.
pattern    The pattern string (but see auto.reduce.pattern, below).
subject    A character vector, or an XString or XStringSet object containing the subject sequence(s).
max.mismatch, min.mismatch
            Integer vectors of length >= 1 recycled to the length of the at (or starting.at, or ending.at) argument. More details below.
with.indels See details below.
fixed      Only with a DNAString or RNAString-based subject can a fixed value other than the default (TRUE) be used.
            If TRUE (the default), an IUPAC ambiguity code in the pattern can only match the same code in the subject, and vice versa. If FALSE, an IUPAC ambiguity code in the pattern can match any letter in the subject that is associated with the code, and vice versa. See IUPAC_CODE_MAP for more information about the IUPAC Extended Genetic Alphabet.
            fixed can also be a character vector, a subset of c("pattern", "subject").
            fixed=c("pattern", "subject") is equivalent to fixed=TRUE (the default).
            An empty vector is equivalent to fixed=FALSE. With fixed="subject", ambiguities in the pattern only are interpreted as wildcards. With fixed="pattern", ambiguities in the subject only are interpreted as wildcards.
follow.index Whether the single integer returned by which.isMatchingAt (and related utils) should be the first *value* in at for which a match occurred, or its *index* in at (the default).
auto.reduce.pattern Whether pattern should be effectively shortened by 1 letter, from its beginning for which.isMatchingStartingAt and from its end for which.isMatchingEndingAt, for each successive (at, max.mismatch) "pair".

Details

A "match" of pattern P in subject S is a substring S' of S that is considered similar enough to P according to some distance (or metric) specified by the user. 2 distances are supported by most
pattern matching functions in the Biostrings package. The first (and simplest) one is the “number of mismatching letters”. It is defined only when the 2 strings to compare have the same length, so when this distance is used, only matches that have the same number of letters as P are considered. The second one is the “edit distance” (aka Levenshtein distance): it’s the minimum number of operations needed to transform P into S’, where an operation is an insertion, deletion, or substitution of a single letter. When this metric is used, matches can have a different number of letters than P.

The neditAt function implements these 2 distances. If with.indels is FALSE (the default), then the first distance is used i.e. neditAt returns the “number of mismatching letters” between the pattern P and the substring S’ of S starting at the positions specified in at (note that neditAt is vectorized so a long vector of integers can be passed thru the at argument). If with.indels is TRUE, then the “edit distance” is used: for each position specified in at, P is compared to all the substrings S’ of S starting at this position and the smallest distance is returned. Note that this distance is guaranteed to be reached for a substring of length < 2*length(P) so, of course, in practice, P only needs to be compared to a small number of substrings for every starting position.

Value

hasLetterAt: A logical matrix with one row per element in x and one column per letter/position to check. When a specified position is invalid with respect to an element in x then the corresponding matrix element is set to NA.

neditAt: If subject is an XString object, then return an integer vector of the same length as at. If subject is an XStringSet object, then return the integer matrix with length(at) rows and length(subject) columns defined by:

\[
\text{sapply}(\text{unname}(\text{subject}), \\
\text{function}(x) \text{neditAt}(\text{pattern}, x, ...))
\]

neditStartingAt is identical to neditAt except that the at argument is now called starting.at.

neditEndingAt is similar to neditAt except that the at argument is now called ending.at and must contain the ending positions of the pattern relatively to the subject.

isMatchingAt: If subject is an XString object, then return the logical vector defined by:

\[
\text{min.mismatch} \leq \text{neditAt}(...) \leq \text{max.mismatch}
\]

If subject is an XStringSet object, then return the logical matrix with length(at) rows and length(subject) columns defined by:

\[
\text{sapply}(\text{unname}(\text{subject}), \\
\text{function}(x) \text{isMatchingAt}(\text{pattern}, x, ...))
\]

isMatchingStartingAt is identical to isMatchingAt except that the at argument is now called starting.at.

isMatchingEndingAt is similar to isMatchingAt except that the at argument is now called ending.at and must contain the ending positions of the pattern relatively to the subject.

which.isMatchingAt: The default behavior (follow.index=FALSE) is as follow. If subject is an XString object, then return the single integer defined by:
which(isMatchingAt(...))[1]

If subject is an `XStringSet` object, then return the integer vector defined by:

```r
sapply(unname(subject),
    function(x) which.isMatchingAt(pattern, x, ...))
```

If `follow.index=TRUE`, then the returned value is defined by:

```r
at[which.isMatchingAt(..., follow.index=FALSE)]
```

which.isMatchingStartingAt is identical to which.isMatchingAt except that the `at` argument is now called `starting.at`. which.isMatchingEndingAt is similar to which.isMatchingAt except that the `at` argument is now called `ending.at` and must contain the ending positions of the pattern relatively to the subject.

**See Also**

`nucleotideFrequencyAt`, `matchPattern`, `matchPDict`, `matchLRPatterns`, `trimLRPatterns`, `IUPAC_CODE_MAP`, `XString-class`, `align-utils` in the `pwalign` package

**Examples**

```r
## hasLetterAt()
##
## x <- DNAStringSet(c("AAACGT", "AACGT", "ACGT", "TAGGA"))
x <- DNAStringSet(c("AAACGT", "AACGT", "ACGT", "TAGGA"))
hasLetterAt(x, "AAAAAA", 1:6)

## hasLetterAt() can be used to answer questions like: "which elements
## in x have A at position 2 and a G at position 4?"
q1 <- hasLetterAt(x, \"AG\", c(2, 4))
which(rowSums(q1) == 2)

## or "how many probes in the drosophila2 chip have T, G, T, A at
## position 2, 4, 13 and 20, respectively?"
library(drosophila2probe)
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
q2 <- hasLetterAt(probes, \"TGTA\", c(2, 4, 13, 20))
sum(rowSums(q2) == 4)

## or "what's the probability to have an A at position 25 if there is
## one at position 13?"
q3 <- hasLetterAt(probes, \"AACGT\", c(13, 25, 25, 25, 25))
sum(q3[, 1] & q3[, 2]) / sum(q3[, 1])

## Probabilities to have other bases at position 25 if there is an A
## at position 13:
sum(q3[, 1] & q3[, 3]) / sum(q3[, 1]) # C
sum(q3[, 1] & q3[, 4]) / sum(q3[, 1]) # G
```
lowlevel-matching

sum(q3[,1] & q3[,5]) / sum(q3[,1]) # T

## See ?nucleotideFrequencyAt for another way to get those results.

subject <- DNAString("GTATA")

## Pattern "AT" matches subject "GTATA" at position 3 (exact match)
neditAt("AT", subject, at=3)
isMatchingAt("AT", subject, at=3)

## ... but not at position 1
neditAt("AT", subject)
isMatchingAt("AT", subject)

## ... unless we allow 1 mismatching letter (inexact match)
isMatchingAt("AT", subject, max.mismatch=1)

## Here we look at 6 different starting positions and find 3 matches if
## we allow 1 mismatching letter
isMatchingAt("AT", subject, at=0:5, max.mismatch=1)

## No match
neditAt("NT", subject, at=1:4)
isMatchingAt("NT", subject, at=1:4)

## 2 matches if N is interpreted as an ambiguity (fixed=FALSE)
neditAt("NT", subject, at=1:4, fixed=FALSE)
isMatchingAt("NT", subject, at=1:4, fixed=FALSE)

## max.mismatch != 0 and fixed=FALSE can be used together
neditAt("NCA", subject, at=0:5, fixed=FALSE)
isMatchingAt("NCA", subject, at=0:5, max.mismatch=1, fixed=FALSE)

some_starts <- c(10:-10, NA, 6)
subject <- DNAString("ACGTGCA")
is_matching <- isMatchingAt("CAT", subject, at=some_starts, max.mismatch=1)
some_starts[is_matching]

which.isMatchingAt("CAT", subject, at=some_starts, max.mismatch=1)
which.isMatchingAt("CAT", subject, at=some_starts, max.mismatch=1, follow.index=TRUE)

## WITH INDELS

subject <- BString("ABCDEFxxxCDEFxxxABBCDE")
neditAt("ABCDEF", subject, at=9)
neditAt("ABCDEF", subject, at=9, with.indels=TRUE)
isMatchingAt("ABCDEF", subject, at=9, max.mismatch=1, with.indels=TRUE)
isMatchingAt("ABCDEF", subject, at=9, max.mismatch=2, with.indels=TRUE)
neditAt("ABCDEF", subject, at=17)
neditAt("ABCDEF", subject, at=17, with.indels=TRUE)
neditEndingAt("ABCDEF", subject, ending.at=22)
neditEndingAt("ABCDEF", subject, ending.at=22, with.indels=TRUE)

MaskedXString-class  MaskedXString objects

Description

The MaskedBString, MaskedDNAString, MaskedRNAString and MaskedAAString classes are containers for storing masked sequences.

All those containers derive directly (and with no additional slots) from the MaskedXString virtual class.

Details

In Biostrings, a pile of masks can be put on top of a sequence. A pile of masks is represented by a MaskCollection object and the sequence by an XString object. A MaskedXString object is the result of bundling them together in a single object.

Note that, no matter what masks are put on top of it, the original sequence is always stored unmodified in a MaskedXString object. This allows the user to activate/deactivate masks without having to worry about losing the information stored in the masked/unmasked regions. Also this allows efficient memory management since the original sequence never needs to be copied (modifying it would require to make a copy of it first - sequences cannot and should never be modified in place in Biostrings), even when the set of active/inactive masks changes.

Accessor methods

In the code snippets below, x is a MaskedXString object. For masks(x) and masks(x) <- y, it can also be an XString object and y must be NULL or a MaskCollection object.

unmasked(x): Turns x into an XString object by dropping the masks.
masks(x): Turns x into a MaskCollection object by dropping the sequence.
masks(x) <- y: If x is an XString object and y is NULL, then this doesn’t do anything.
    If x is an XString object and y is a MaskCollection object, then this turns x into a MaskedXString object by putting the masks in y on top of it.
    If x is a MaskedXString object and y is NULL, then this is equivalent to x <- unmasked(x).
    If x is a MaskedXString object and y is a MaskCollection object, then this replaces the masks currently on top of x by the masks in y.
alphabet(x): Equivalent to alphabet(unmasked(x)). See ?alphabet for more information.
length(x): Equivalent to length(unmasked(x)). See ?'length,XString-method' for more information.
"maskedwidth" and related methods

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

maskedwidth(x): Get the number of masked letters in x. A letter is considered masked iff it's
masked by at least one active mask.

maskedratio(x): Equivalent to maskedwidth(x) / length(x).

nchar(x): Equivalent to length(x) – maskedwidth(x).

Coercion

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

as(x, "Views"): Turns x into a Views object where the views are the unmasked regions of the
original sequence ("unmasked" means not masked by at least one active mask).

Other methods

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

collapse(x): Collapses the set of masks in x into a single mask made of all active masks.
gaps(x): Reverses all the masks i.e. each mask is replaced by a mask where previously unmasked
regions are now masked and previously masked regions are now unmasked.

Author(s)

H. Pagès

See Also

• maskMotif
• injectHardMask
• alphabetFrequency
• reverseComplement
• XString-class
• MaskCollection-class
• Views-class

Examples

```
## A. MASKING BY POSITION

mask0 <- Mask(mask.width=29, start=c(3, 10, 25), width=c(6, 8, 5))
x <- DNAString("ACACACTAGATAGNACTNNGAGAGACGC")
length(x)  # same as width(mask0)
nchar(x)   # same as length(x)
masks(x) <- mask0
x
```
```
length(x) # has not changed
nchar(x)  # has changed
gaps(x)

## Prepare a MaskCollection object of 3 masks ('mymasks') by running the
## examples in the man page for these objects:
examle(MaskCollection, package="IRanges")

## Put it on 'x':
masks(x) <- mymasks
x
alphabetFrequency(x)

## Deactivate all masks:
active(masks(x)) <- FALSE
x

## Activate mask "C":
active(masks(x))['C'] <- TRUE
x

## Turn MaskedXString object into a Views object:
as(x, "Views")

## Drop the masks:
masks(x) <- NULL
x
alphabetFrequency(x)
```

## B. MASKING BY CONTENT

See ?maskMotif for masking by content

---

**maskMotif**

**Masking by content (or by position)**

**Description**

Functions for masking a sequence by content (or by position).

**Usage**

```
maskMotif(x, motif, min.block.width=1, ...)  
mask(x, start=NA, end=NA, pattern)
```
maskMotif

Arguments

- **x** The sequence to mask.
- **motif** The motif to mask in the sequence.
- **min.block.width** The minimum width of the blocks to mask.
- **...** Additional arguments for `matchPattern`.
- **start** An integer vector containing the starting positions of the regions to mask.
- **end** An integer vector containing the ending positions of the regions to mask.
- **pattern** The motif to mask in the sequence.

Value

A `MaskedXString` object for `maskMotif` and an `XStringViews` object for `mask`.

Author(s)

H. Pagès

See Also

`read.Mask`, `matchPattern`, `XString-class`, `MaskedXString-class`, `XStringViews-class`, `MaskCollection-class`

Examples

```r
## EXAMPLE 1
maskMotif(BString("AbcbcbEEE"), "bcb")
maskMotif(BString("AbcbcbEEE"), "bcb")

## maskMotif() can be used in an incremental way to mask more than 1
## motif. Note that maskMotif() does not try to mask again what's
## already masked (i.e. the new mask will never overlaps with the
## previous masks) so the order in which the motifs are masked actually
## matters as it will affect the total set of masked positions.

x0 <- BString("AbcbEEEEbcbbEEEcbcbcbc")
x1 <- maskMotif(x0, "E")
x1 <- maskMotif(x1, "b")
x2 <- maskMotif(x1, "bcb")
x2 <- maskMotif(x2, "b")
x3 <- maskMotif(x3, "b")

## Note that inverting the order in which "b" and "bcb" are masked would
## lead to a different final set of masked positions.
## Also note that the order doesn't matter if the motifs to mask don't
## overlap (we assume that the motifs are unique) i.e. if the prefix of
## each motif is not the suffix of any other motif. This is of course
```
## the case when all the motifs have only 1 letter.

## EXAMPLE 2

```r
x <- DNAString("ACACACTAGATAGNACTNNNGAGAGACGC")
```

## Mask the N-blocks
```r
x1 <- maskMotif(x, "N")
x1
as(x1, "Views")
gaps(x1)
as(gaps(x1), "Views")
```

## Mask the AC-blocks
```r
x2 <- maskMotif(x1, "AC")
x2
gaps(x2)
```

## Mask the GA-blocks
```r
x3 <- maskMotif(x2, "GA", min.block.width=5)
x3 # masks 2 and 3 overlap
gaps(x3)
```

## EXAMPLE 3

```r
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrU <- Dmelanogaster$chrU
chrU
alphabetFrequency(chrU)
chrU <- maskMotif(chrU, "N")
chrU
alphabetFrequency(chrU)
as(chrU, "Views")
as(gaps(chrU), "Views")
```

```r
mask2 <- Mask(mask.width=length(chrU),
              start=c(50000, 350000, 543900), width=25000)
names(mask2) <- "some ugly regions"
masks(chrU) <- append(masks(chrU), mask2)
chrU
as(chrU, "Views")
as(gaps(chrU), "Views")
```

## EXAMPLE 4

```
## Note that unlike maskMotif(), mask() returns an XStringViews object!

## masking "by position"
mask("AxyxyxBC", 2, 6)

## masking "by content"
mask("AxyxyxBC", "xyx")
noN_chrU <- mask(chrU, "N")
noN_chrU
alphabetFrequency(noN_chrU, collapse=TRUE)

---

match-utils

Utility functions operating on the matches returned by a high-level matching function

Description

Miscellaneous utility functions operating on the matches returned by a high-level matching function like `matchPattern`, `matchPDict`, etc...

Usage

mismatch(pattern, x, fixed=TRUE)
nmatch(pattern, x, fixed=TRUE)
nmismatch(pattern, x, fixed=TRUE)
## S4 method for signature 'MIndex'
coverage(x, shift=0L, width=NULL, weight=1L)
## S4 method for signature 'MaskedXString'
coverage(x, shift=0L, width=NULL, weight=1L)

Arguments

- **pattern**: The pattern string.
- **x**: An `XStringViews` object for `mismatch` (typically, one returned by `matchPattern(pattern, subject)`). An `MIndex` object for `coverage`, or any object for which a `coverage` method is defined. See ?coverage.
- **fixed**: See ?lowlevel-matching.
- **shift, width**: See ?coverage.
- **weight**: An integer vector specifying how much each element in x counts.

Details

The `mismatch` function gives the positions of the mismatching letters of a given pattern relatively to its matches in a given subject.

The `nmatch` and `nmismatch` functions give the number of matching and mismatching letters produced by the `mismatch` function.

The `coverage` function computes the "coverage" of a subject by a given pattern or set of patterns.
Value

mismatch: a list of integer vectors.
mismatch: an integer vector containing the length of the vectors produced by mismatch.
coverage: an Rle object indicating the coverage of x. See ?coverage for the details. If x is an MIndex object, the coverage of a given position in the underlying sequence (typically the subject used during the search that returned x) is the number of matches (or hits) it belongs to.

See Also

lowlevel-matching, matchPattern, matchPDict, XString-class, XStringViews-class, MIndex-class, coverage, align-utils in the pwalign package

Examples

```r
## mismatch() / nmismatch()
subject <- DNAString("ACGTGCA")
m <- matchPattern("NCA", subject, max.mismatch=1, fixed=FALSE)
mismatch("NCA", m)
nmismatch("NCA", m)

## coverage()
coverage(m)
## See ?matchPDict for examples of using coverage() on an MIndex object...
```

matchLRPatterns  Find paired matches in a sequence

Description

The matchLRPatterns function finds paired matches in a sequence i.e. matches specified by a left pattern, a right pattern and a maximum distance between the left pattern and the right pattern.

Usage

```r
matchLRPatterns(Lpattern, Rpattern, max.gaplength, subject, max.Lmismatch=0, max.Rmismatch=0, with.Lindels=FALSE, with.Rindels=FALSE, Lfixed=TRUE, Rfixed=TRUE)
```
Arguments

- **Lpattern**
  The left part of the pattern.

- **Rpattern**
  The right part of the pattern.

- **max.gaplength**
  The maximum length of the gap in the middle i.e. the maximum distance between the left and right parts of the pattern.

- **subject**
  An XString, XStringViews or MaskedXString object containing the target sequence.

- **max.Lmismatch**
  The maximum number of mismatching letters allowed in the left part of the pattern. If non-zero, an inexact matching algorithm is used (see the `matchPattern` function for more information).

- **max.Rmismatch**
  Same as `max.Lmismatch` but for the right part of the pattern.

- **with.Lindels**
  If TRUE then indels are allowed in the left part of the pattern. In that case `max.Lmismatch` is interpreted as the maximum "edit distance" allowed in the left part of the pattern.

- **with.Rindels**
  Same as `with.Lindels` but for the right part of the pattern.

- **Lfixed**
  Only with a DNAString or RNAString subject can a Lfixed value other than the default (TRUE) be used.
  With Lfixed=FALSE, ambiguities (i.e. letters from the IUPAC Extended Genetic Alphabet (see IUPAC_CODE_MAP) that are not from the base alphabet) in the left pattern and in the subject are interpreted as wildcards i.e. they match any letter that they stand for.
  Lfixed can also be a character vector, a subset of c("pattern", "subject"). Lfixed=c("pattern", "subject") is equivalent to Lfixed=TRUE (the default).
  An empty vector is equivalent to Lfixed=FALSE. With Lfixed="subject", ambiguities in the pattern only are interpreted as wildcards. With Lfixed="pattern", ambiguities in the subject only are interpreted as wildcards.

- **Rfixed**
  Same as Lfixed but for the right part of the pattern.

Value

An XStringViews object containing all the matches, even when they are overlapping (see the examples below), and where the matches are ordered from left to right (i.e. by ascending starting position).

Author(s)

H. Pagès

See Also

matchPattern, matchProbePair, trimLRPatterns, findPalindromes, reverseComplement, XString-class, XStringViews-class, MaskedXString-class
Examples

```r
library(BSgenome.Dmelanogaster.UCSC.dm3)
subject <- Dmelanogaster$chr3R
Lpattern <- "AGCTCCGAG"
Rpattern <- "TTGTTCACA"
matchLRPatterns(Lpattern, Rpattern, 500, subject) # 1 match

## Note that matchLRPatterns() will return all matches, even when they are
## overlapping:
subject <- DNAString("AAATTAACCCTT")
matchLRPatterns("AA", "TT", 0, subject) # 1 match
matchLRPatterns("AA", "TT", 1, subject) # 2 matches
matchLRPatterns("AA", "TT", 3, subject) # 3 matches
matchLRPatterns("AA", "TT", 7, subject) # 4 matches
```

---

matchPattern  String searching functions

Description

A set of functions for finding all the occurrences (aka "matches" or "hits") of a given pattern (typically short) in a (typically long) reference sequence or set of reference sequences (aka the subject)

Usage

```r
matchPattern(pattern, subject,
max.mismatch=0, min.mismatch=0,
with.indels=FALSE, fixed=TRUE,
algorithm="auto")

countPattern(pattern, subject,
max.mismatch=0, min.mismatch=0,
with.indels=FALSE, fixed=TRUE,
algorithm="auto")

vmatchPattern(pattern, subject,
max.mismatch=0, min.mismatch=0,
with.indels=FALSE, fixed=TRUE,
algorithm="auto", ...)

countPattern(pattern, subject,
max.mismatch=0, min.mismatch=0,
with.indels=FALSE, fixed=TRUE,
algorithm="auto", ...)```
Arguments

**pattern**
The pattern string.

**subject**
An XString, XStringViews or MaskedXString object for matchPattern and countPattern.
An XStringSet or XStringViews object for vmatchPattern and vcountPattern.

**max.mismatch, min.mismatch**
The maximum and minimum number of mismatching letters allowed (see `lowlevel-matching` for the details). If non-zero, an algorithm that supports inexact matching is used.

**with.indels**
If TRUE then indels are allowed. In that case, min.mismatch must be 0 and max.mismatch is interpreted as the maximum "edit distance" allowed between the pattern and a match. Note that in order to avoid pollution by redundant matches, only the "best local matches" are returned. Roughly speaking, a "best local match" is a match that is locally both the closest (to the pattern P) and the shortest. More precisely, a substring S' of the subject S is a "best local match" iff:

(a) \(nedit(P, S') \leq \max.mismatch\)
(b) for every substring S1 of S':
   \(nedit(P, S1) > nedit(P, S')\)
(c) for every substring S2 of S that contains S':
   \(nedit(P, S2) \geq nedit(P, S')\)

One nice property of "best local matches" is that their first and last letters are guaranteed to match the letters in P that they align with.

**fixed**
If TRUE (the default), an IUPAC ambiguity code in the pattern can only match the same code in the subject, and vice versa. If FALSE, an IUPAC ambiguity code in the pattern can match any letter in the subject that is associated with the code, and vice versa. See `lowlevel-matching` for more information.

**algorithm**
One of the following: "auto", "naive-exact", "naive-inexact", "boyer-moore", "shift-or" or "indels".

... Additional arguments for methods.

Details

Available algorithms are: “naive exact”, “naive inexact”, “Boyer-Moore-like”, “shift-or” and “indels”. Not all of them can be used in all situations: restrictions apply depending on the "search criteria" i.e. on the values of the pattern, subject, max.mismatch, min.mismatch, with.indels and fixed arguments.

It is important to note that the algorithm argument is not part of the search criteria. This is because the supported algorithms are interchangeable, that is, if 2 different algorithms are compatible with a given search criteria, then choosing one or the other will not affect the result (but will most likely affect the performance). So there is no "wrong choice" of algorithm (strictly speaking).

Using algorithm="auto" (the default) is recommended because then the best suited algorithm will automatically be selected among the set of algorithms that are valid for the given search criteria.
Value

An XStringViews object for matchPattern.
A single integer for countPattern.
An MIndex object for vmatchPattern.
An integer vector for vcountPattern, with each element in the vector corresponding to the number of matches in the corresponding element of subject.

Note

Use matchPDict if you need to match a (big) set of patterns against a reference sequence.
Use pairwiseAlignment from the pwalign package if you need to solve a (Needleman-Wunsch) global alignment, a (Smith-Waterman) local alignment, or an (ends-free) overlap alignment problem.

See Also

• lowlevel-matching
• matchPDict
• pairwiseAlignment
• mismatch
• matchLRPatterns
• matchProbePair
• maskMotif
• alphabetFrequency
• XStringViews class
• MIndex class
• pairwiseAlignment in the pwalign package

Examples

```r
## A. matchPattern()/countPattern()
## ---------------------------------------------------------------------
## A simple inexact matching example with a short subject:
x <- DNAString("AAGCGCGATATG")
m1 <- matchPattern("GCNNNAT", x)
m1
m2 <- matchPattern("GCNNNAT", x, fixed=FALSE)
m2
as.matrix(m2)

## With DNA sequence of yeast chromosome number 1:
data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)
```
matchPattern

PpiI <- "GAACNNNNNCTC"  # a restriction enzyme pattern
match1.PpiI <- matchPattern(PpiI, yeast1, fixed=FALSE)
match2.PpiI <- matchPattern(PpiI, yeast1, max.mismatch=1, fixed=FALSE)

## With a genome containing isolated Ns:
library(BSgenome.Celegans.UCSC.ce2)
chrII <- Celegans["chrII"]
alphabetFrequency(chrII)
matchPattern("N", chrII)
matchPattern("TGGGTGTCTTT", chrII)  # no match
matchPattern("TGGGTGTCTTT", chrII, fixed=FALSE)  # 1 match

## Using wildcards ("N") in the pattern on a genome containing N-blocks:
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- maskMotif(Dmelanogaster$chrX, "N")
as(chrX, "Views")  # 4 non masked regions
matchPattern("TTTATGNTTGGTA", chrX, fixed=FALSE)
## Can also be achieved with no mask:
matches(chrX) <- NULL
matchPattern("TTTATGNTTGGTA", chrX, fixed="subject")

## B. vmatchPattern()vcountPattern()

## Load Fly upstream sequences (i.e. the sequences 2000 bases upstream of
## annotated transcription starts):
dm3_upstream_filepath <- system.file("extdata",
    "dm3_upstream2000.fa.gz",
    package="Biostrings")
dm3_upstream <- readDNAStringSet(dm3_upstream_filepath)
dm3_upstream

Ebox <- DNAString("CANNTG")
subject <- dm3_upstream
mindex <- vmatchPattern(Ebox, subject, fixed="subject")
nmatch_per_seq <- elementNROWS(mindex)  # Get the number of matches per
    # subject element.
sum(nmatch_per_seq)  # Total number of matches.
table(nmatch_per_seq)

## Let's have a closer look at one of the upstream sequences with most
## matches:
i0 <- which.max(nmatch_per_seq)
subject0 <- subject[[i0]]
ir0 <- mindex[[i0]]  # matches in 'subject0' as an IRanges object
ir0
Views(subject0, ir0)  # matches in 'subject0' as a Views object

## C. WITH INDELS

##
```r
library(BSgenome.Celegans.UCSC.ce2)
subject <- Celegans$chrI
pattern1 <- DNAString("ACGGACCTAATGTTATC")
pattern2 <- DNAString("ACGGACCTVATGTTRTC")

## Allowing up to 2 mismatching letters doesn't give any match:
m1a <- matchPattern(pattern1, subject, max.mismatch=2)

## But allowing up to 2 edit operations gives 3 matches:
system.time(m1b <- matchPattern(pattern1, subject, max.mismatch=2,
                        with.indels=TRUE))
m1b

## pwalign::pairwiseAlignment() returns the (first) best match only:
if (interactive()) {
  library(pwalign)
  mat <- nucleotideSubstitutionMatrix(match=1, mismatch=0, baseOnly=TRUE)
  ## Note that this call to pairwiseAlignment() will need to
  ## allocate 733.5 Mb of memory (i.e. length(pattern) * length(subject)
  ## * 3 bytes).
  system.time(pwa <- pairwiseAlignment(pattern1, subject, type="local",
                        substitutionMatrix=mat,
                        gapOpening=0, gapExtension=1))
  pwa
}

## With IUPAC ambiguities in the pattern:
m2a <- matchPattern(pattern2, subject, max.mismatch=2,
                  fixed="subject")
m2b <- matchPattern(pattern2, subject, max.mismatch=2,
                  with.indels=TRUE, fixed="subject")

## All the matches in 'm1b' and 'm2a' should also appear in 'm2b':
stopifnot(suppressWarnings(all(ranges(m1b) %in% ranges(m2b))))
stopifnot(suppressWarnings(all(ranges(m2a) %in% ranges(m2b))))

## D. WHEN 'with.indels=TRUE', ONLY "BEST LOCAL MATCHES" ARE REPORTED

## With deletions in the subject:
subject <- BString("ACDEFxxxCDEFxxxABCE")
matchPattern("ABCDFF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDFF", subject, max.mismatch=2)

## With insertions in the subject:
subject <- BString("AiBCDiEFxxxABCDiiFxxxAiBCDEFxxxABCiDEF")
matchPattern("ABCDFF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDFF", subject, max.mismatch=2)

## With substitutions (note that the "best local matches" can introduce
## indels and therefore be shorter than 6):
subject <- BString("AsCDEFxxxABDCEFxxxBACDEFxxxABCEDF")
```

---

**matchPattern**

library(BSgenome.Celegans.UCSC.ce2)

subject <- Celegans$chrI

pattern1 <- DNAString("ACGGACCTAATGTTATC")

pattern2 <- DNAString("ACGGACCTVATGTTRTC")

## Allowing up to 2 mismatching letters doesn't give any match:
m1a <- matchPattern(pattern1, subject, max.mismatch=2)

## But allowing up to 2 edit operations gives 3 matches:
system.time(m1b <- matchPattern(pattern1, subject, max.mismatch=2,
                        with.indels=TRUE))

m1b

## pwalign::pairwiseAlignment() returns the (first) best match only:
if (interactive()) {
  library(pwalign)
  mat <- nucleotideSubstitutionMatrix(match=1, mismatch=0, baseOnly=TRUE)
  ## Note that this call to pairwiseAlignment() will need to
  ## allocate 733.5 Mb of memory (i.e. length(pattern) * length(subject)
  ## * 3 bytes).
  system.time(pwa <- pairwiseAlignment(pattern1, subject, type="local",
                        substitutionMatrix=mat,
                        gapOpening=0, gapExtension=1))
  pwa
}

## With IUPAC ambiguities in the pattern:
m2a <- matchPattern(pattern2, subject, max.mismatch=2,
                  fixed="subject")
m2b <- matchPattern(pattern2, subject, max.mismatch=2,
                  with.indels=TRUE, fixed="subject")

## All the matches in 'm1b' and 'm2a' should also appear in 'm2b':
stopifnot(suppressWarnings(all(ranges(m1b) %in% ranges(m2b))))
stopifnot(suppressWarnings(all(ranges(m2a) %in% ranges(m2b))))

## D. WHEN 'with.indels=TRUE', ONLY "BEST LOCAL MATCHES" ARE REPORTED

## With deletions in the subject:
subject <- BString("ACDEFxxxCDEFxxxABCE")
matchPattern("ABCDFF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDFF", subject, max.mismatch=2)

## With insertions in the subject:
subject <- BString("AiBCDiEFxxxABCDiiFxxxAiBCDEFxxxABCiDEF")
matchPattern("ABCDFF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDFF", subject, max.mismatch=2)

## With substitutions (note that the "best local matches" can introduce
## indels and therefore be shorter than 6):
subject <- BString("AsCDEFxxxABDCEFxxxBACDEFxxxABCEDF")
matchPattern("ABCDEF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDEF", subject, max.mismatch=2)

matchPDict

Matching a dictionary of patterns against a reference

Description

A set of functions for finding all the occurrences (aka "matches" or "hits") of a set of patterns (aka the dictionary) in a reference sequence or set of reference sequences (aka the subject)

The following functions differ in what they return: matchPDict returns the "where" information i.e. the positions in the subject of all the occurrences of every pattern; countPDict returns the "how many times" information i.e. the number of occurrences for each pattern; and whichPDict returns the "who" information i.e. which patterns in the input dictionary have at least one match.

vcountPDict and vwhichPDict are vectorized versions of countPDict and whichPDict, respectively, that is, they work on a set of reference sequences in a vectorized fashion.

This man page shows how to use these functions (aka the *PDict functions) for exact matching of a constant width dictionary i.e. a dictionary where all the patterns have the same length (same number of nucleotides).

See `?`matchPDict-inexact` for how to use these functions for inexact matching or when the original dictionary has a variable width.

Usage

matchPDict(pdict, subject,
    max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
    algorithm="auto", verbose=FALSE)
countPDict(pdict, subject,
    max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
    algorithm="auto", verbose=FALSE)
whichPDict(pdict, subject,
    max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
    algorithm="auto", verbose=FALSE)

vcountPDict(pdict, subject,
    max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
    algorithm="auto", collapse=FALSE, weight=1L,
    verbose=FALSE, ...)
vwhichPDict(pdict, subject,
    max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
    algorithm="auto", verbose=FALSE)
Arguments

pdict

A PDict object containing the preprocessed dictionary.

All these functions also work with a dictionary that has not been preprocessed (in other words, the pdict argument can receive an XStringSet object). Of course, it won’t be as fast as with a preprocessed dictionary, but it will generally be slightly faster than using matchPattern/countPattern or vmatchPattern/vcountPattern in a "lapply/sapply loop", because, here, looping is done at the C-level. However, by using a non-preprocessed dictionary, many of the restrictions that apply to preprocessed dictionaries don’t apply anymore. For example, the dictionary doesn’t need to be rectangular or to be a DNAStringSet object: it can be any type of XStringSet object and have a variable width.

subject

An XString or MaskedXString object containing the subject sequence for matchPDict, countPDict and whichPDict.

An XStringSet object containing the subject sequences for vcountPDict and vwhichPDict.

If pdict is a PDict object (i.e. a preprocessed dictionary), then subject must be of base class DNAString. Otherwise, subject must be of the same base class as pdict.

max.mismatch, min.mismatch

The maximum and minimum number of mismatching letters allowed (see ?isMatchingAt for the details). This man page focuses on exact matching of a constant width dictionary so max.mismatch=0 in the examples below. See ?matchPDict-inexact for inexact matching.

with.indels

Only supported by countPDict, whichPDict, vcountPDict and vwhichPDict at the moment, and only when the input dictionary is non-preprocessed (i.e. XStringSet).

If TRUE then indels are allowed. In that case, min.mismatch must be 0 and max.mismatch is interpreted as the maximum "edit distance" allowed between any pattern and any of its matches. See ?matchPattern for more information.

fixed

Whether IUPAC ambiguity codes should be interpreted literally or not (see ?isMatchingAt for more information). This man page focuses on exact matching of a constant width dictionary so fixed=TRUE in the examples below. See ?matchPDict-inexact for inexact matching.

algorithm

Ignored if pdict is a preprocessed dictionary (i.e. a PDict object). Otherwise, can be one of the following: "auto", "naive-exact", "naive-inexact", "boyer-moore" or "shift-or". See ?matchPattern for more information. Note that "indels" is not supported for now.

verbose

TRUE or FALSE.

collapse, weight

collapse must be FALSE, 1, or 2.

If collapse=FALSE (the default), then weight is ignored and vcountPDict returns the full matrix of counts (M0). If collapse=1, then M0 is collapsed "horizontally" i.e. it is turned into a vector with length equal to length(pdict).

If weight=1L (the default), then this vector is defined by rowSums(M0). If collapse=2, then M0 is collapsed "vertically" i.e. it is turned into a vector with
length equal to length(subject). If weight=1L (the default), then this vector is defined by colSums(M0).
If collapse=1 or collapse=2, then the elements in subject (collapse=1) or in pdict (collapse=2) can be weighted thru the weight argument. In that case, the returned vector is defined by M0 %*% rep(weight, length.out=length(subject)) and rep(weight, length.out=length(pdict)) %*% M0, respectively.

... Additional arguments for methods.

Details
In this man page, we assume that you know how to preprocess a dictionary of DNA patterns that can then be used with any of the *PDict functions described here. Please see ?PDict if you don’t.
When using the *PDict functions for exact matching of a constant width dictionary, the standard way to preprocess the original dictionary is by calling the PDict constructor on it with no extra arguments. This returns the preprocessed dictionary in a PDict object that can be used with any of the *PDict functions.

Value
If M denotes the number of patterns in the pdict argument (M <- length(pdict)), then matchPDict returns an MIndex object of length M, and countPDict an integer vector of length M.
whichPDict returns an integer vector made of the indices of the patterns in the pdict argument that have at least one match.
If N denotes the number of sequences in the subject argument (N <- length(subject)), then vcountPDict returns an integer matrix with M rows and N columns, unless the collapse argument is used. In that case, depending on the type of weight, an integer or numeric vector is returned (see above for the details).
vwhichPDict returns a list of N integer vectors.

Author(s)
H. Pagès

References

See Also
PDict-class, MIndex-class, matchPDict-inexact, isMatchingAt, coverage,MIndex-method, matchPattern, alphabetFrequency, DNAStringSet-class, XStringViews-class, MaskedDNAString-class

Examples
```R
## ---------------------------------------------------------------------
## A. A SIMPLE EXAMPLE OF EXACT MATCHING
## ---------------------------------------------------------------------
```
## Creating the pattern dictionary:
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe)
dict0 # The original dictionary.
length(dict0) # Hundreds of thousands of patterns.
pdict0 <- PDict(dict0) # Store the original dictionary in
# a PDict object (preprocessing).

## Using the pattern dictionary on chromosome 3R:
library(B5genome.Dmelanogaster.UCSC.dm3)
chr3R <- Dmelanogaster$chr3R # Load chromosome 3R
chr3R
mi0 <- matchPDict(pdict0, chr3R) # Search...

## Looking at the matches:
start_index <- startIndex(mi0) # Get the start index.
length(start_index) # Same as the original dictionary.

## Equivalent to start_index[[8220]].

## Starts of the 8220th pattern.
end_index <- endIndex(mi0) # Get the end index.

## Ends of the 8220th pattern.
nmatch_per_pat <- elementNROWS(mi0) # Get the number of matches per pattern.

## Total number of matches.
table(nmatch_per_pat)
i0 <- which(nmatch_per_pat == max(nmatch_per_pat))
pdict0[[i0]] # The pattern with most occurrences.

## And as an XStringViews object.

## Get the coverage of the original subject:

## Equivalent to start_index[[8220]].
cov3R <- as.integer(coverage(mi0, width=length(chr3R)))
max(cov3R)
mean(cov3R)

## Only 2.44% of chr3R is covered.
if (interactive()) {
  library(graphics)
  plotCoverage <- function(cx, start, end)
  {
    graphics::plot.new()
    graphics::plot.window(c(start, end), c(0, 20))
    graphics::axis(1)
    graphics::axis(2)
    graphics::axis(4)
    graphics::lines(start:end, cx[start:end], type="l")
  }
  plotCoverage(cov3R, 27600000, 27900000)
}

## B. NAMING THE PATTERNS
The names of the original patterns, if any, are propagated to the PDict and MIndex objects:

```r
names(dict0) <- mkAllStrings(letters, 4)[seq_len(length(dict0))]
dict0
pdict0n <- PDict(dict0)
names(pdict0n)[1:30]
pdict0n["abcd"]
m0n <- matchPDict(pdict0n, chr3R)
names(m0n)[1:30]
m0n["abcd"]
```

This is particularly useful when unlisting an MIndex object:

```r
unlist(m0)[1:10]
unlist(m0n)[1:10] # keep track of where the matches are coming from
```

If the number of matches is what matters only (without regarding their positions), then countPDict() will be faster, especially when there is a high number of matches:

```r
nmatch_per_pat0 <- countPDict(pdict0, chr3R)
stopifnot(identical(nmatch_per_pat0, nmatch_per_pat))
```

If (interactive()) {

```r
# What's the impact of the dictionary width on performance?
# Below is some code that can be used to figure out (will take a long
time to run). For different widths of the original dictionary, we look at:
# o pptime: preprocessing time (in sec.) i.e. time needed for
# building the PDict object from the truncated input
# sequences;
# o nnodes: nb of nodes in the resulting Aho-Corasick tree;
# o nupatt: nb of unique truncated input sequences;
# o matchtime: time (in sec.) needed to find all the matches;
# o totalcount: total number of matches.
getPDictStats <- function(dict, subject)
{
  ans_width <- width(dict[1])
  ans_pptime <- system.time(pdict <- PDict(dict))[["elapsed"]]
pptb <- pdict@threeparts@pptb
  ans_nnodes <- nnodes(pptb)
  ans_nupatt <- sum(!duplicated(pdict))
  ans_matchtime <- system.time(
    m0 <- matchPDict(pdict, subject)
  )[["elapsed"]]
  ans_totalcount <- sum(elementNROWS(m0))
  list(
    ans_width, ans_pptime, ans_nnodes, ans_nupatt, ans_matchtime, ans_totalcount
  )
}
```

```r
ans <- ทั้งนี้
```
```r
width=ans_width,
pptime=ans_pptime,
nnodes=ans_nnodes,
nupatt=ans_nupatt,
matchtime=ans_matchtime,
totalcount=ans_totalcount
)
stats <- lapply(8:25,
  function(width)
    getPDictStats(DNAStringSet(dict0, end=width), chr3R))
stats <- data.frame(do.call(rbind, stats))
stats
}

## D. USING A NON-PREPROCESSED DICTIONARY

dict3 <- DNAStringSet(mkAllStrings(DNA_BASES, 3))  # all trinucleotides
dict3 <- PDict(dict3)

## The 3 following calls are equivalent (from faster to slower):
res3a <- countPDict(pdict3, chr3R)
res3b <- countPDict(dict3, chr3R)
res3c <- sapply(dict3,
    function(pattern) countPattern(pattern, chr3R))
stopifnot(identical(res3a, res3b))
stopifnot(identical(res3a, res3c))

## One reason for using a non-preprocessed dictionary is to get rid of
## all the constraints associated with preprocessing, e.g., when
## preprocessing with PDict(), the input dictionary must be DNA and a
## Trusted Band must be defined (explicitly or implicitly).
## See '?PDict' for more information about these constraints.
## In particular, using a non-preprocessed dictionary can be
## useful for the kind of inexact matching that can't be achieved
## with a PDict object (if performance is not an issue).
## See '?matchPDict-inexact' for more information about inexact
## matching.

dictD <- xscat(dict3, "N", reverseComplement(dict3))

## The 2 following calls are equivalent (from faster to slower):
resDa <- matchPDict(dictD, chr3R, fixed=FALSE)
resDb <- sapply(dictD,
    function(pattern)
      matchPattern(pattern, chr3R, fixed=FALSE))
stopifnot(all(sapply(seq_len(length(dictD)),
    function(i)
      identical(resDa[[i]], as(resDb[[i]], "IRanges"))))
```
## A non-preprocessed dictionary can be of any base class i.e. BString, RNAString, and AAString, in addition to DNAString:

```r
matchPDict(AAStringSet(c("DARC", "EGH")), AAString("KMFPNDEGHSTTWTEE"))
```

## E. vcountPDict()

```r
# Load Fly upstream sequences (i.e. the sequences 2000 bases upstream of
# annotated transcription starts):

dm3_upstream_filepath <- system.file("extdata",
  "dm3_upstream2000.fa.gz",
  package="Biostrings")

dm3_upstream <- readDNAStringSet(dm3_upstream_filepath)
dm3_upstream

subject <- dm3_upstream[1:100]
mat1 <- vcountPDict(pdict0, subject)
dim(mat1) # length(pdict0) x length(subject)

nhit_per_probe <- rowSums(mat1)
table(nhit_per_probe)
```

```r
# Without vcountPDict(), 'mat1' could have been computed with:

mat2 <- sapply(unname(subject), function(x) countPDict(pdict0, x))

stopifnot(identical(mat1, mat2))
```

```r
## but using vcountPDict() is faster (10x or more, depending of the
## average length of the sequences in 'subject').
```

```r
if (interactive()) {

## This will fail (with message "allocMatrix: too many elements
## specified") because, on most platforms, vectors and matrices in R
## are limited to 2^31 elements:

subject <- dm3_upstream

vcountPDict(pdict0, subject)

length(pdict0) * length(dm3_upstream)

1 * length(pdict0) * length(dm3_upstream) # > 2^31

## But this will work:

nhit_per_seq <- vcountPDict(pdict0, subject, collapse=2)

sum(nhit_per_seq >= 1) # nb of subject sequences with at least 1 hit
table(nhit_per_seq) # max is 74

which.max(nhit_per_seq) # 1133

sum(countPDict(pdict0, subject[[1133]])) # 74

}
```

## F. RELATIONSHIP BETWEEN vcountPDict(), countPDict() AND
## vcountPattern()

```r
subject <- dm3_upstream

# The 4 following calls are equivalent (from faster to slower):

mat3a <- vcountPDict(pdict3, subject)
mat3b <- vcountPDict(dict3, subject)
```

mat3c <- sapply(dict3, function(pattern) vcountPattern(pattern, subject))
mat3d <- sapply(unname(subject), function(x) countPDict(pdict3, x))
stopifnot(identical(mat3a, mat3b))
stopifnot(identical(mat3a, t(mat3c)))
stopifnot(identical(mat3a, mat3d))

## The 3 following calls are equivalent (from faster to slower):

nhitpp3a <- vcountPDict(pdict3, subject, collapse=1) # rowSums(mat3a)
nhitpp3b <- vcountPDict(dict3, subject, collapse=1)
nhitpp3c <- sapply(dict3, function(pattern) sum(vcountPattern(pattern, subject)))
stopifnot(identical(nhitpp3a, nhitpp3b))
stopifnot(identical(nhitpp3a, nhitpp3c))

## The 3 following calls are equivalent (from faster to slower):

nhitps3a <- vcountPDict(pdict3, subject, collapse=2) # colSums(mat3a)
nhitps3b <- vcountPDict(dict3, subject, collapse=2)
nhitps3c <- sapply(unname(subject), function(x) sum(countPDict(pdict3, x)))
stopifnot(identical(nhitps3a, nhitps3b))
stopifnot(identical(nhitps3a, nhitps3c))

## G. vwhichPDict()

subject <- dm3_upstream

## The 4 following calls are equivalent (from faster to slower):

vwp3a <- vwhichPDict(pdict3, subject)
vwp3b <- vwhichPDict(dict3, subject)
vwp3c <- lapply(seq_len(ncol(mat3a)), function(j) which(mat3a[ , j] != 0L))
vwp3d <- lapply(unname(subject), function(x) whichPDict(pdict3, x))
stopifnot(identical(vwp3a, vwp3b))
stopifnot(identical(vwp3a, vwp3c))
stopifnot(identical(vwp3a, vwp3d))

table(sapply(vwp3a, length))
which.min(sapply(vwp3a, length))

## Get the trinucleotides not represented in upstream sequence 21823:

dict3[-vwp3a[[21823]]] # 2 trinucleotides

## Sanity check:
tnf <- trinucleotideFrequency(subject[[21823]])
stopifnot(all(names(tnf)[tnf == 0] == dict3[-vwp3a[[21823]]]))

## H. MAPPING PROBE SET IDS BETWEEN CHIPS WITH vwhichPDict()
```r
## probe.
## WARNING: This example takes about 10 minutes to run.
if (interactive()) {

library(hgu95av2probe)
library(hgu133aprobe)
probes1 <- DNAStringSet(hgu95av2probe)
probes2 <- DNAStringSet(hgu133aprobe)
pdict2 <- PDict(probes2)

## Get the mapping from probes1 to probes2 (based on exact matching):
map1to2 <- vwhichPDict(pdict2, probes1)

## The following helper function uses the probe level mapping to induce
## the mapping at the probe set IDs level (from hgu95av2 to hgu133a).
## To keep things simple, 2 probe set IDs are considered mapped iff
## each of them contains at least one probe mapped to one probe of
## the other:
mapProbeSetIDs1to2 <- function(psID)
  unique(hgu133aprobe$Probe.Set.Name[unlist(
    map1to2[hgu95av2probe$Probe.Set.Name == psID]
  )])

## Use the helper function to build the complete mapping:
psIDs1 <- unique(hgu95av2probe$Probe.Set.Name)
mapPSIDs1to2 <- lapply(psIDs1, mapProbeSetIDs1to2)  # about 3 min.
names(mapPSIDs1to2) <- psIDs1

## Do some basic stats:
table(sapply(mapPSIDs1to2, length))

## [ADVANCED USERS ONLY]
## An alternative that is slightly faster is to put all the probes
## (hgu95av2 + hgu133a) in a single PDict object and then query its
## 'dups0' slot directly. This slot is a Dups object containing the
## mapping between duplicated patterns.
## Note that we can do this only because all the probes have the
## same length (25) and because we are doing exact matching:
probes12 <- DNAStringSet(c(hgu95av2probe$sequence, hgu133aprobe$sequence))
pdict12 <- PDict(probes12)
dups0 <- pdict12@dups0

mapProbeSetIDs1to2alt <- function(psID)
{
  ii1 <- unique(togroup(dups0, which(hgu95av2probe$Probe.Set.Name == psID)))
  ii2 <- members(dups0, ii1) - length(probes1)
  ii2 <- ii2[ii2 >= 1L]
  unique(hgu133aprobe$Probe.Set.Name[ii2])
}

mapPSIDs1to2alt <- lapply(psIDs1, mapProbeSetIDs1to2alt)  # about 5 min.
names(mapPSIDs1to2alt) <- psIDs1
```

matchPDict-inexact

```r
## 'mapPSIDs1to2alt' and 'mapPSIDs1to2' contain the same mapping:
stopifnot(identical(lapply(mapPSIDs1to2alt, sort),
                   lapply(mapPSIDs1to2, sort)))
```

Description

The `matchPDict`, `countPDict` and `whichPDict` functions efficiently find the occurrences in a text (the subject) of all patterns stored in a preprocessed dictionary.

This man page shows how to use these functions for inexact (or fuzzy) matching or when the original dictionary has a variable width.

See `?matchPDict` for how to use these functions for exact matching of a constant width dictionary i.e. a dictionary where all the patterns have the same length (same number of nucleotides).

Details

In this man page, we assume that you know how to preprocess a dictionary of DNA patterns that can then be used with `matchPDict`, `countPDict` or `whichPDict`. Please see `?PDict` if you don’t.

`matchPDict` and family support different kinds of inexact matching but with some restrictions. Inexact matching is controlled via the definition of a Trusted Band during the preprocessing step and/or via the `max.mismatch`, `min.mismatch` and `fixed` arguments. Defining a Trusted Band is also required when the original dictionary is not rectangular (variable width), even for exact matching. See `?PDict` for how to define a Trusted Band.

Here is how `matchPDict` and family handle the Trusted Band defined on `pdict`:

- (1) Find all the exact matches of all the elements in the Trusted Band.
- (2) For each element in the Trusted Band that has at least one exact match, compare the head and the tail of this element with the flanking sequences of the matches found in (1).

Note that the number of exact matches found in (1) will decrease exponentially with the width of the Trusted Band. Here is a simple guideline in order to get reasonably good performance: if `TBW` is the width of the Trusted Band (`TBW <- tb.width(pdict)`) and `L` the number of letters in the subject (`L <- nchar(subject)`), then `L / (4^TBW)` should be kept as small as possible, typically < 10 or 20.

In addition, when a Trusted Band has been defined during preprocessing, then `matchPDict` and family can be called with `fixed=FALSE`. In this case, IUPAC ambiguity codes in the head or the tail of the `PDict` object are treated as ambiguities.

Finally, `fixed="pattern"` can be used to indicate that IUPAC ambiguity codes in the subject should be treated as ambiguities. It only works if the density of codes is not too high. It works whether or not a Trusted Band has been defined on `pdict`. 
Author(s)

H. Pagès

References


See Also

PDict-class, MIndex-class, matchPDict

Examples

```r
# A. USING AN EXPLICIT TRUSTED BAND
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe)
dict0 # the original dictionary

# Preprocess the original dictionary by defining a Trusted Band that
# spans nucleotides 1 to 9 of each pattern.
pdict9 <- PDict(dict0, tb.end=9)
pdict9
tail(pdict9)
sum(duplicated(pdict9))
table(patternFrequency(pdict9))

library(BSgenome.Dmelanogaster.UCSC.dm3)
chr3R <- Dmelanogaster$chr3R
chr3R
table(countPDict(pdict9, chr3R, max.mismatch=1))
table(countPDict(pdict9, chr3R, max.mismatch=3))
table(countPDict(pdict9, chr3R, max.mismatch=5))

# B. COMPARISON WITH EXACT MATCHING

# When the original dictionary is of constant width, exact matching
# (i.e. 'max.mismatch=0' and 'fixed=TRUE) will be more efficient with
# a full-width Trusted Band (i.e. a Trusted Band that covers the entire
# dictionary) than with a Trusted Band of width < width(dict0).
pdict0 <- PDict(dict0)
count0 <- countPDict(pdict0, chr3R)
count0b <- countPDict(pdict9, chr3R, max.mismatch=0)
identical(count0b, count0) # TRUE

# C. USING AN EXPLICIT TRUSTED BAND ON A VARIABLE WIDTH DICTIONARY
```
matchPDict-inexact

## Here is a small variable width dictionary that contains IUPAC ambiguities (pattern 1 and 3 contain an N):
dict0 <- DNAStringSet(c("TACCNG", "TAGT", "CGNNT", "AGTAG", "TAGT"))
## (Note that pattern 2 and 5 are identical.)

## If we only want to do exact matching, then it is recommended to use the widest possible Trusted Band i.e. to set its width to "min(width(dict0))" because this is what will give the best performance. However, when 'dict0' contains IUPAC ambiguities (like in our case), it could be that one of them is falling into the Trusted Band so we get an error (only base letters can go in the Trusted Band for now):
## Not run:
PDict(dict0, tb.end=min(width(dict0))) # Error!
## End(Not run)

## In our case, the Trusted Band cannot be wider than 3:
pdict <- PDict(dict0, tb.end=3)
tail(pdict)

subject <- DNAString("TAGTACCAGTTTCGGG")

m <- matchPDict(pdict, subject)
elementNROWS(m) # pattern 2 and 5 have 1 exact match
m[2]

## We can take advantage of the fact that our Trusted Band doesn't cover the entire dictionary to allow inexact matching on the uncovered parts (the tail in our case):

m <- matchPDict(pdict, subject, fixed=FALSE)
elementNROWS(m) # now pattern 1 has 1 match too
m[[1]]

m <- matchPDict(pdict, subject, max.mismatch=1)
elementNROWS(m) # now pattern 4 has 1 match too
m[[4]]

m <- matchPDict(pdict, subject, max.mismatch=1, fixed=FALSE)
elementNROWS(m) # now pattern 3 has 1 match too
m[[3]] # note that this match is "out of limit"
Views(subject, m[[3]])

m <- matchPDict(pdict, subject, max.mismatch=2)
elementNROWS(m) # pattern 4 gets 1 additional match
m[[4]]

## Unlist all matches:
unlist(m)
## D. WITH IUPAC AMBIGUITY CODES IN THE PATTERNS

The Trusted Band cannot contain IUPAC ambiguity codes so patterns
with ambiguity codes can only be preprocessed if we can define a
Trusted Band with no ambiguity codes in it.

```r
dict <- DNAStringSet(c("AAACAAKS", "GGGAAA", "TNCCGGG"))
pdict <- PDict(dict, tb.start=3, tb.width=4)
subject <- DNAString("AAACAATCCCGGGAAACAAGG")

matchPDict(pdict, subject)
matchPDict(pdict, subject, fixed="subject")
```

## Sanity checks:

```r
res1 <- as.list(matchPDict(pdict, subject))
res2 <- as.list(matchPDict(dict, subject))
res3 <- lapply(dict,
  function(pattern)
    as(matchPattern(pattern, subject), "IRanges"))
stopifnot(identical(res1, res2))
stopifnot(identical(res1, res3))
```

```r
res1 <- as.list(matchPDict(pdict, subject, fixed="subject"))
res2 <- as.list(matchPDict(dict, subject, fixed="subject"))
res3 <- lapply(dict,
  function(pattern)
    as(matchPattern(pattern, subject, fixed="subject"), "IRanges"))
stopifnot(identical(res1, res2))
stopifnot(identical(res1, res3))
```

## E. WITH IUPAC AMBIGUITY CODES IN THE SUBJECT

'fixed="pattern"' (or 'fixed=FALSE') can be used to indicate that
IUPAC ambiguity codes in the subject should be treated as ambiguities.

```r
pdict <- PDict(c("ACAC", "TCCG"))
matchPDict(pdict, DNAString("ACACAATCCCGGGAAACAAGG"))
matchPDict(pdict, DNAString("ACACAATCCCGGGAAACAAGG"), fixed="pattern")
matchPDict(pdict, DNAString("ACACAATCCCGGGAAACAAGG"), fixed="pattern")
matchPDict(pdict, DNAString("ACACAATCCCGGGAAACAAGG"), fixed="pattern")
matchPDict(pdict, DNAString("ACACAATCCCGGGAAACAAGG"), fixed="pattern")
```

```r
dict <- DNAStringSet(c("TTC", "CTT"))
pdict <- PDict(dict)
subject <- DNAString("CYTCACTTC")
mi1 <- matchPDict(pdict, subject, fixed="pattern")
mi2 <- matchPDict(dict, subject, fixed="pattern")
stopifnot(identical(as.list(mi1), as.list(mi2)))
```
matchProbePair

Find "theoretical amplicons" mapped to a probe pair

Description
In the context of a computer-simulated PCR experiment, one wants to find the amplicons mapped to a given primer pair. The matchProbePair function can be used for this: given a forward and a reverse probe (i.e. the chromosome-specific sequences of the forward and reverse primers used for the experiment) and a target sequence (generally a chromosome sequence), the matchProbePair function will return all the "theoretical amplicons" mapped to this probe pair.

Usage
matchProbePair(Fprobe, Rprobe, subject, algorithm="auto", logfile=NULL, verbose=FALSE)

Arguments
Fprobe
The forward probe.
Rprobe
The reverse probe.
subject
A DNAString object (or an XStringViews object with a DNAString subject) containing the target sequence.
algorithm
One of the following: "auto", "naive-exact", "naive-inexact", "boyer-moore" or "shift-or". See matchPattern for more information.
logfile
A file used for logging.
verbose
TRUE or FALSE.

Details
The matchProbePair function does the following: (1) find all the "plus hits" i.e. the Fprobe and Rprobe matches on the "plus" strand, (2) find all the "minus hits" i.e. the Fprobe and Rprobe matches on the "minus" strand and (3) from the set of all (plus_hit, minus_hit) pairs, extract and return the subset of "reduced matches" i.e. the (plus_hit, minus_hit) pairs such that (a) plus_hit <= minus_hit and (b) there are no hits (plus or minus) between plus_hit and minus_hit. This set of "reduced matches" is the set of "theoretical amplicons".

Value
An XStringViews object containing the set of "theoretical amplicons".

Author(s)
H. Pagès

See Also
matchPattern, matchLRPatterns, findPalindromes, reverseComplement, XStringViews-class
matchprobes

Examples

```r
library(BSgenome.Dmelanogaster.UCSC.dm3)
subject <- Dmelanogaster$chr3R

## With 20-nucleotide forward and reverse probes:
Fprobe <- "AGCTCCGAGTTCCTGCAATA"
Rprobe <- "CGTTGTTCACAAATATGCGG"
matchProbePair(Fprobe, Rprobe, subject) # 1 "theoretical amplicon"

## With shorter forward and reverse probes, the risk of having multiple
## "theoretical amplicons" increases:
Fprobe <- "AGCTCCGAGTTCC"
Rprobe <- "CGTTGTTCACAA"
matchProbePair(Fprobe, Rprobe, subject) # 2 "theoretical amplicons"
Fprobe <- "AGCTCCGAGTT"
Rprobe <- "CGTTGTTCACA"
matchProbePair(Fprobe, Rprobe, subject) # 9 "theoretical amplicons"
```

matchprobes

(Deprecated) A function to match a query sequence to the sequences of a set of probes.

Description

The query sequence, a character string (probably representing a transcript of interest), is scanned for the presence of exact matches to the sequences in the character vector records. The indices of the set of matches are returned.

The function is inefficient: it works on R’s character vectors, and the actual matching algorithm is of time complexity \( \text{length(query)} \times \text{length(records)} \)!

This function is now deprecated. See matchPattern, vmatchPattern and matchPDict for more efficient sequence matching functions.

Usage

```r
matchprobes(query, records, probepos=FALSE)
```

Arguments

- `query`: A character vector. For example, each element may represent a gene (transcript) of interest. See Details.
- `records`: A character vector. For example, each element may represent the probes on a DNA array.
- `probepos`: A logical value. If TRUE, return also the start positions of the matches in the query sequence.
Details

toupper is applied to the arguments query and records before matching. The intention of this is
to make the matching case-insensitive. The function is embarrassingly naive. The matching is done
using the C library function strstr.

Value

A list. Its first element is a list of the same length as the input vector. Each element of the list is a
numeric vector containing the indices of the probes that have a perfect match in the query sequence.
If probepos is TRUE, the returned list has a second element: it is of the same shape as described
above, and gives the respective positions of the matches.

Author(s)

R. Gentleman, Laurent Gautier, Wolfgang Huber

See Also

matchPattern, vmatchPattern, matchPDict

Examples

## Not run:
library(hgu95av2probe)
data("hgu95av2probe")
seq <- hgu95av2probe$sequence[1:20]
target <- paste(seq, collapse="")
matchprobes(target, seq, probepos=TRUE)

## End(Not run)
## Utility functions for basic manipulation of the Position Weight Matrix

maxWeights(x)
minWeights(x)
maxScore(x)
minScore(x)
unitScale(x)

## S4 method for signature 'matrix'
reverseComplement(x, ...)

### Arguments

**x**
For *PWM*: a rectangular character vector or rectangular DNASTRINGSet object ("rectangular" means that all elements have the same number of characters) with no IUPAC ambiguity letters, or a Position Frequency Matrix represented as an integer matrix with row names containing at least A, C, G and T (typically the result of a call to `consensusMatrix`).

For maxWeights, minWeights, maxScore, minScore, unitScale and reverseComplement:
- a Position Weight Matrix represented as a numeric matrix with row names A, C, G and T.

**type**
The type of Position Weight Matrix, either "log2probratio" or "prob". See Details section for more information.

**prior.params**
A positive numeric vector, which represents the parameters of the Dirichlet conjugate prior, with names A, C, G, and T. See Details section for more information.

**pwm**
A Position Weight Matrix represented as a numeric matrix with row names A, C, G and T.

**subject**
Typically a DNAString object. A Views object on a DNAString subject, a MaskedDNAString object, or a single character string, are also supported.

IUPAC ambiguity letters in subject are ignored (i.e. assigned weight 0) with a warning.

**min.score**
The minimum score for counting a match. Can be given as a character string containing a percentage (e.g. "85") of the highest possible score or as a single number.

**with.score**
TRUE or FALSE. If TRUE, then the score of each hit is included in the returned object in a metadata column named score. Say the returned object is hits, this metadata column can then be accessed with `mcols(hits)$score`.

**starting.at**
An integer vector specifying the starting positions of the Position Weight Matrix relatively to the subject.

... Additional arguments for methods.

### Details

The *PWM* function uses a multinomial model with a Dirichlet conjugate prior to calculate the estimated probability of base b at position i. As mentioned in the Arguments section, prior.params supplies the parameters for the DNA bases A, C, G, and T in the Dirichlet prior. These values result in a position independent initial estimate of the probabilities for the bases to be priorProbs =
prior.params/sum(prior.params) and the posterior (data infused) estimate for the probabilities for the bases in each of the positions to be postProbs = (consensusMatrix(x) + prior.params)/(length(x) + sum(prior.params)). When type = "log2probratio", the PWM = unitScale(log2(postProbs/priorProbs)). When type = "prob", the PWM = unitScale(postProbs).

Value

A numeric matrix representing the Position Weight Matrix for PWM.
A numeric vector containing the Position Weight Matrix-based scores for PWMscoreStartingAt.
An XStringViews object for matchPWM.
A single integer for countPWM.
A vector containing the max weight for each position in pwm for maxWeights.
A vector containing the min weight for each position in pwm for minWeights.
The highest possible score for a given Position Weight Matrix for maxScore.
The lowest possible score for a given Position Weight Matrix for minScore.
The modified numeric matrix given by (x - minScore(x)/ncol(x))/(maxScore(x) - minScore(x)) for unitScale.
A PWM obtained by reverting the column order in PWM x and by reassigning each row to its complementary nucleotide for reverseComplement.

Author(s)

H. Pagès and P. Aboyoun

References


See Also

consensusMatrix, matchPattern, reverseComplement, DNAString-class, XStringViews-class

Examples

## Data setup:
data(HNF4alpha)
library(B5genome.Dmelanogaster.UCSC.dm3)
chr3R <- Dmelanogaster$chr3R
chr3R

## Create a PWM from a PFM or directly from a rectangular DNAStringSet object:
pfm <- consensusMatrix(HNF4alpha)
pwm <- PWM(pfm) # same as 'PWM(HNF4alpha)'

## Perform some general routines on the PWM:
round(pwm, 2)
maxWeights(pwm)
maxScore(pwm)
reverseComplement(pwm)

## Score the first 5 positions:
PWMscoreStartingAt(pwm, chr3R, starting.at=1:5)

## Match the plus strand:
hits <- matchPWM(pwm, chr3R)
nhit <- countPWM(pwm, chr3R)  # same as 'length(hits)'

## Use 'with.score=TRUE' to get the scores of the hits:
hits <- matchPWM(pwm, chr3R, with.score=TRUE)
head(mcols(hits)$score)
min(mcols(hits)$score / maxScore(pwm))  # should be >= 0.8

## The scores can also easily be post-calculated:
scores <- PWMscoreStartingAt(pwm, subject(hits), start(hits))

## Match the minus strand:
matchPWM(reverseComplement(pwm), chr3R)

---

**MIndex-class**  
*MIndex objects*

**Description**

The MIndex class is the basic container for storing the matches of a set of patterns in a subject sequence.

**Details**

An MIndex object contains the matches (start/end locations) of a set of patterns found in an `XString` object called "the subject string" or "the subject sequence" or simply "the subject".  
`matchPDict` function returns an MIndex object.

**Accessor methods**

In the code snippets below, x is an MIndex object.

length(x): The number of patterns that matches are stored for.
names(x): The names of the patterns that matches are stored for.
startIndex(x): A list containing the starting positions of the matches for each pattern.
endIndex(x): A list containing the ending positions of the matches for each pattern.
elementNROWS(x): An integer vector containing the number of matches for each pattern.
Subsetting methods

In the code snippets below, \( x \) is an MIndex object.

\( x[[i]] \): Extract the matches for the \( i \)-th pattern as an IRanges object.

Coercion

In the code snippets below, \( x \) is an MIndex object.

\( \text{as}(x, \text{"CompressedIRangesList"}) \): Turns \( x \) into an CompressedIRangesList object. This coercion changes \( x \) from one IntegerRangesList subtype to another with the underlying IntegerRanges values remaining unchanged.

Other utility methods and functions

In the code snippets below, \( x \) and \( \text{mindex} \) are MIndex objects and \( \text{subject} \) is the XString object containing the sequence in which the matches were found.

\( \text{unlist}(x, \text{recursive=TRUE, use.names=TRUE}) \): Return all the matches in a single IRanges object. recursive and use.names are ignored.

\( \text{extractAllMatches}(\text{subject, mindex}) \): Return all the matches in a single XStringViews object.

Author(s)

H. Pagès

See Also

matchPDict, PDict-class, IRanges-class, XStringViews-class

Examples

```r
```

---

misc

Some miscellaneous stuff

Description

Some miscellaneous stuff.

Usage

\( \text{N50}(\text{csizes}) \)

Arguments

\( \text{csizes} \) A vector containing the contig sizes.
Value

N50: The N50 value as an integer.

The N50 contig size

**Definition** The N50 contig size of an assembly (aka the N50 value) is the size of the largest contig such that the contigs larger than that have at least 50% the bases of the assembly.

**How is it calculated?** It is calculated by adding the sizes of the biggest contigs until you reach half the total size of the contigs. The N50 value is then the size of the contig that was added last (i.e. the smallest of the big contigs covering 50% of the genome).

**What for?** The N50 value is a standard measure of the quality of a de novo assembly.

Author(s)

Nicolas Delhomme <delhomme@embl.de>

See Also

XStringSet-class

Examples

```r
# Generate 10 random contigs of sizes comprised between 100 and 10000:
my.contig <- DNAStringSet(
  sapply(
    sample(c(100:10000), 10),
    function(size)
      paste(sample(DNA_BASES, size, replace=TRUE), collapse=''))
)

# Get their sizes:
my.size <- width(my.contig)

# Calculate the N50 value of this set of contigs:
my.contig.N50 <- N50(my.size)
```

---

**Description**

Starting with BioC 3.19, the following functions are defined in the `pwalign` package:

- writePairwiseAlignments
- nucleotideSubstitutionMatrix
- errorSubstitutionMatrices
MultipleAlignment-class

Multiple Alignment objects

Description

The MultipleAlignment class is a container for storing multiple sequence alignments.

Usage

## Constructors:
DNAMultipleAlignment(x=character(), start=NA, end=NA, width=NA,
use.names=TRUE, rowmask=NULL, colmask=NULL)
RNAMultipleAlignment(x=character(), start=NA, end=NA, width=NA,
use.names=TRUE, rowmask=NULL, colmask=NULL)
AAMultipleAlignment(x=character(), start=NA, end=NA, width=NA,
use.names=TRUE, rowmask=NULL, colmask=NULL)

## Read functions:
readDNAMultipleAlignment(filepath, format)
readRNAMultipleAlignment(filepath, format)
readAAMultipleAlignment(filepath, format)
## Write funtions:
write.phylip(x, filepath)

## ... and more (see below)

### Arguments

- **x**: Either a character vector (with no NAs), or an `XString`, `XStringSet` or `XStringViews` object containing strings with the same number of characters. If writing out a Phylip file, then `x` would be a `MultipleAlignment` object.

- **start, end, width**: Either `NA`, a single integer, or an integer vector of the same length as `x` specifying how `x` should be "narrowed" (see `?narrow` in the `IRanges` package for the details).

- **use.names**: `TRUE` or `FALSE`. Should names be preserved?

- **filepath**: A character vector (of arbitrary length when reading, of length 1 when writing) containing the paths to the files to read or write. Note that special values like "" or "|cmd" (typically supported by other I/O functions in R) are not supported here. Also `filepath` cannot be a connection.

- **format**: Either "fasta" (the default), "stockholm", or "clustal".

- **rowmask**: a `NormalIRanges` object that will set masking for rows.

- **colmask**: a `NormalIRanges` object that will set masking for columns.

### Details

The `MultipleAlignment` class is designed to hold and represent multiple sequence alignments. The rows and columns within an alignment can be masked for ad hoc analyses.

### Accessor methods

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

- `unmasked(x)`: The underlying `XStringSet` object containing the multiple sequence alignment.

- `rownames(x)`: `NULL` or a character vector of the same length as `x` containing a short user-provided description or comment for each sequence in `x`.

- `rowmask(x), rowmask(x, append, invert) <- value`: Gets and sets the `NormalIRanges` object representing the masked rows in `x`. The `append` argument takes `union`, `replace` or `intersect` to indicate how to combine the new value with `rowmask(x)`. The `invert` argument takes a logical argument to indicate whether or not to invert the new mask. The `value` argument can be of any class that is coercible to a `NormalIRanges` via the `as` function.

- `colmask(x), colmask(x, append, invert) <- value`: Gets and sets the `NormalIRanges` object representing the masked columns in `x`. The `append` argument takes `union`, `replace` or `intersect` to indicate how to combine the new value with `colmask(x)`. The `invert` argument takes a logical argument to indicate whether or not to invert the new mask. The `value` argument can be of any class that is coercible to a `NormalIRanges` via the `as` function.
MultipleAlignment-class

maskMotif(x, motif, min.block.width=1, ...): Returns a MultipleAlignment object with a modified column mask based upon motifs found in the consensus string where the consensus string keeps all the columns but drops the masked rows.

- **motif** The motif to mask.
- **min.block.width** The minimum width of the blocks to mask.

... Additional arguments for matchPattern.

maskGaps(x, min.fraction, min.block.width): Returns a MultipleAlignment object with a modified column mask based upon gaps in the columns. In particular, this mask is defined by min.block.width or more consecutive columns that have min.fraction or more of their non-masked rows containing gap codes.

- **min.fraction** A value in \([0, 1]\) that indicates the minimum fraction needed to call a gap in the consensus string (default is 0.5).
- **min.block.width** A positive integer that indicates the minimum number of consecutive gaps to mask, as defined by min.fraction (default is 4).

nrow(x): Returns the number of sequences aligned in x.

ncol(x): Returns the number of characters for each alignment in x.

dim(x): Equivalent to c(nrow(x), ncol(x)).

maskednrow(x): Returns the number of masked aligned sequences in x.

maskedncol(x): Returns the number of masked aligned characters in x.

maskeddim(x): Equivalent to c(maskednrow(x), maskedncol(x)).

maskratio(x): Equivalent to maskeddim(x) / dim(x).

nchar(x): Returns the number of unmasked aligned characters in x, i.e. ncol(x) - maskedncol(x).

alphabet(x): Equivalent to alphabet(unmasked(x)).

Coercion

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

as(from, "DNAStringSet"), as(from, "RNAStringSet"), as(from, "AAStringSet"), as(from, "BStringSet"):

Creates an instance of the specified \(X\text{StringSet}\) object subtype that contains the unmasked regions of the multiple sequence alignment in x.

as.character(x, use.names): Convert x to a character vector containing the unmasked regions of the multiple sequence alignment. use.names controls whether or not rownames(x) should be used to set the names of the returned vector (default is TRUE).

as.matrix(x, use.names): Returns a character matrix containing the "exploded" representation of the unmasked regions of the multiple sequence alignment. use.names controls whether or not rownames(x) should be used to set the row names of the returned matrix (default is TRUE).

Utilities

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

consensusMatrix(x, as.prob, baseOnly): Creates an integer matrix containing the column frequencies of the underlying alphabet with masked columns being represented with NA values. If as.prob is TRUE, then probabilities are reported, otherwise counts are reported (the default). If baseOnly is TRUE, then the non-base letters are collapsed into an "other" category.
consensusString(x, ...): Creates a consensus string for x with the symbol "#" representing a masked column. See consensusString for details on the arguments.

consensusViews(x, ...): Similar to the consensusString method. It returns a XStringViews on the consensus string containing subsequence contigs of non-masked columns. Unlike the consensusString method, the masked columns in the underlying string contain a consensus value rather than the "#" symbol.

alphabetFrequency(x, as.prob, collapse): Creates an integer matrix containing the row frequencies of the underlying alphabet. If as.prob is TRUE, then probabilities are reported, otherwise counts are reported (the default). If collapse is TRUE, then returns the overall frequency instead of the frequency by row.

detail(x, invertColMask, hideMaskedCols): Allows for a full pager driven display of the object so that masked cols and rows can be removed and the entire sequence can be visually inspected. If hideMaskedCols is set to it's default value of TRUE then the output will hide all the the masked columns in the output. Otherwise, all columns will be displayed along with a row to indicate the masking status. If invertColMask is TRUE then any displayed mask will be flipped so as to represent things in a way consistent with Phylip style files instead of the mask that is actually stored in the MultipleAlignment object. Please notice that invertColMask will be ignored if hideMaskedCols is set to its default value of TRUE since in that case it will not make sense to show any masking information in the output. Masked rows are always hidden in the output.

Display

The letters in a DNAMultipleAlignment or RNAMultipleAlignment object are colored when displayed by the show() method. Set global option Biostrings.coloring to FALSE to turn off this coloring.

Author(s)
P. Aboyoun and M. Carlson

See Also
XStringSet-class, MaskedXString-class

Examples

```r
## create an object from file
origMAlign <- readDNAMultipleAlignment(filepath =
  system.file("extdata", 
  "msx2_mRNA.aln", 
  package="Biostrings"),
  format="clustal")

## list the names of the sequences in the alignment
rownames(origMAlign)

## rename the sequences to be the underlying species for MSX2
rownames(origMAlign) <- c("Human","Chimp","Cow","Mouse","Rat",
```

origMAlign

## See a detailed pager view
if (interactive()) {
detail(origMAlign)
}

## operations to mask rows
## For columns, just use colmask() and do the same kinds of operations
rowMasked <- origMAlign
rowmask(rowMasked) <- IRanges(start=1,end=3)
rowMasked

## remove rowumn masks
rowmask(rowMasked) <- NULL
rowMasked

## "select" rows of interest
rowmask(rowMasked, invert=TRUE) <- IRanges(start=4,end=7)
rowMasked

## or mask the rows that intersect with masked rows
rowmask(rowMasked, append="intersect") <- IRanges(start=1,end=5)
rowMasked

## TATA-masked
tataMasked <- maskMotif(origMAlign, "TATA")
colmask(tataMasked)

## automatically mask rows based on consecutive gaps
autoMasked <- maskGaps(origMAlign, min.fraction=0.5, min.block.width=4)
colmask(autoMasked)
autoMasked

## calculate frequencies
alphabetFrequency(autoMasked)
consensusMatrix(autoMasked, baseOnly=TRUE)[, 84:90]

## get consensus values
consensusString(autoMasked)
consensusViews(autoMasked)

## cluster the masked alignments
library(pwalign)
sdist <- pwalign::stringDist(as(autoMasked,"DNAStringSet"), method="hamming")
clust <- hclust(sdist, method = "single")
plot(clust)
fourgroups <- cutree(clust, 4)
fourgroups

## write out the alignement object (with current masks) to Phylip format
write.phylip(x = autoMasked, filepath = tempfile("foo.txt", tempdir()))
needwunsQS  
(Defunct) Needleman-Wunsch Global Alignment

Description

Simple gap implementation of Needleman-Wunsch global alignment algorithm.

Usage

needwunsQS(s1, s2, substmat, gappen = 8)

Arguments

s1, s2 an R character vector of length 1 or an XString object.
substmat matrix of alignment score values.
gappen penalty for introducing a gap in the alignment.

Details

Follows specification of Durbin, Eddy, Krogh, Mitchison (1998). This function is now defunct. Please use pairwiseAlignment from the pwalign instead.

Value

An instance of class "PairwiseAlignments".

Author(s)

Vince Carey (<stvjc@channing.harvard.edu>) (original author) and H. Pagès (current maintainer).

References


See Also

pairwiseAlignment and PairwiseAlignments-class in the pwalign package, substitution_matrices
Examples

```r
## Not run:
## This function is now defunct.
## Please use pairwiseAlignment() from the pwalign package instead.

## nucleotide alignment
mat <- matrix(-5L, nrow = 4, ncol = 4)
for (i in seq_len(4)) mat[i, i] <- 0L
rownames(mat) <- colnames(mat) <- DNA_ALPHABET[1:4]
s1 <- DNAString(paste(sample(DNA_ALPHABET[1:4], 1000, replace=TRUE), collapse=""))
s2 <- DNAString(paste(sample(DNA_ALPHABET[1:4], 1000, replace=TRUE), collapse=""))
nw0 <- needwunsQS(s1, s2, mat, gappen = 0)
nw1 <- needwunsQS(s1, s2, mat, gappen = 1)
nw5 <- needwunsQS(s1, s2, mat, gappen = 5)

## amino acid alignment
needwunsQS("PAWHEAE", "HEAGAWGHEE", substmat = "BLOSUM50")

## End(Not run)
```

nucleotideFrequency

`nucleotideFrequency` Calculate the frequency of oligonucleotides in a DNA or RNA sequence (and other related functions)

Description

Given a DNA or RNA sequence (or a set of DNA or RNA sequences), the `nucleotideFrequency` function computes the frequency of all possible oligonucleotides of a given length (called the "width" in this particular context) in a sliding window that is shifted step nucleotides at a time.

The `dinucleotideFrequency` and `trinucleotideFrequency` functions are convenient wrappers for calling `nucleotideFrequency` with width=2 and width=3, respectively.

The `nucleotideFrequencyAt` function computes the frequency of the short sequences formed by extracting the nucleotides found at some fixed positions from each sequence of a set of DNA or RNA sequences.

In this man page we call "DNA input" (or "RNA input") an `XString`, `XStringSet`, `XStringViews` or `MaskedXString` object of base type DNA (or RNA).

Usage

```r
nucleotideFrequency(x, width, step=1,
                     as.prob=FALSE, as.array=FALSE,
                     fast.moving.side="right", with.labels=TRUE, ...)
```

```
## S4 method for signature 'XStringSet'
nucleotideFrequency(x, width, step=1,
                     as.prob=FALSE, as.array=FALSE,
                     fast.moving.side="right", with.labels=TRUE,
                     ...)```
nucleotideFrequency

simplify.as="matrix")

dinucleotideFrequency(x, step=1,
    as.prob=FALSE, as.matrix=FALSE,
    fast.moving.side="right", with.labels=TRUE, ...)

trinucleotideFrequency(x, step=1,
    as.prob=FALSE, as.array=FALSE,
    fast.moving.side="right", with.labels=TRUE, ...)

nucleotideFrequencyAt(x, at,
    as.prob=FALSE, as.array=TRUE,
    fast.moving.side="right", with.labels=TRUE, ...)

## Some related functions:
oligonucleotideTransitions(x, left=1, right=1, as.prob=FALSE)

mkAllStrings(alphabet, width, fast.moving.side="right")

Arguments

x
   Any DNA or RNA input for the *Frequency and oligonucleotideTransitions functions.
   An XStringSet or XStringViews object of base type DNA or RNA for nucleotideFrequencyAt.

width
   The number of nucleotides per oligonucleotide for oligonucleotideFrequency.
   The number of letters per string for mkAllStrings.

step
   How many nucleotides should the window be shifted before counting the next oligonucleotide (i.e. the sliding window step; default 1). If step is smaller than width, oligonucleotides will overlap; if the two arguments are equal, adjacent oligonucleotides will be counted (an efficient way to count codons in an ORF); and if step is larger than width, nucleotides will be sampled step nucleotides apart.

at
   An integer vector containing the positions to look at in each element of x.

as.prob
   If TRUE then probabilities are reported, otherwise counts (the default).

as.array, as.matrix
   Controls the "shape" of the returned object. If TRUE (the default for nucleotideFrequencyAt) then it's a numeric matrix (or array), otherwise it's just a "flat" numeric vector i.e. a vector with no dim attribute (the default for the *Frequency functions).

fast.moving.side
   Which side of the strings should move fastest? Note that, when as.array is TRUE, then the supplied value is ignored and the effective value is "left".

with.labels
   If TRUE then the returned object is named.

...
   Further arguments to be passed to or from other methods.

simplify.as
   Together with the as.array and as.matrix arguments, controls the "shape" of the returned object when the input x is an XStringSet or XStringViews object. Supported simplify.as values are "matrix" (the default), "list" and
"collapsed". If simplify.as is "matrix", the returned object is a matrix with length(x) rows where the i-th row contains the frequencies for x[[i]]. If simplify.as is "list", the returned object is a list of the same length as length(x) where the i-th element contains the frequencies for x[[i]]. If simplify.as is "collapsed", then the the frequencies are computed for the entire object x as a whole (i.e. frequencies cumulated across all sequences in x).

left, right The number of nucleotides per oligonucleotide for the rows and columns respectively in the transition matrix created by oligonucleotideTransitions.

alphabet The alphabet to use to make the strings.

Value

If x is an XString or MaskedXString object, the *Frequency functions return a numeric vector of length 4^width. If as.array (or as.matrix) is TRUE, then this vector is formatted as an array (or matrix). If x is an XStringSet or XStringViews object, the returned object has the shape specified by the simplify.as argument.

Author(s)

H. Pagès and P. Aboyoun; K. Vlahovickef for the step argument

See Also

alphabetFrequency, alphabet, hasLetterAt, XString-class, XStringSet-class, XStringViews-class, MaskedXString-class, GENETIC_CODE, AMINO_ACID_CODE, reverseComplement, rev

Examples

## A. BASIC *Frequency() EXAMPLES

data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)

dinucleotideFrequency(yeast1)
trinucleotideFrequency(yeast1)
oligonucleotideFrequency(yeast1, 4)

## Get the counts of tetranucleotides overlapping by one nucleotide:
oligonucleotideFrequency(yeast1, 4, step=3)

## Get the counts of adjacent tetranucleotides, starting from the first
## nucleotide:
oligonucleotideFrequency(yeast1, 4, step=4)

## Subset the sequence to change the starting nucleotide (here we start
## counting from third nucleotide):
yeast2 <- subseq(yeast1, start=3)
oligonucleotideFrequency(yeast2, 4, step=4)

## Get the less and most represented 6-mers:
nucleotideFrequency

f6 <- oligonucleotideFrequency(yeast1, 6)
f6[f6 == min(f6)]
f6[f6 == max(f6)]

## Get the result as an array:
tri <- trinucleotideFrequency(yeast1, as.array=TRUE)
tri["A", "A", "C"] # == trinucleotideFrequency(yeast1)["AAC"]
tri["T", , ] # frequencies of trinucleotides starting with a "T"

## With input made of multiple sequences:
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
dfmat <- dinucleotideFrequency(probes) # a big matrix
dinucleotideFrequency(probes, simplify.as="collapsed")
dinucleotideFrequency(probes, simplify.as="collapsed", as.matrix=TRUE)

## B. OBSERVED DINUCLEOTIDE FREQUENCY VERSUS EXPECTED DINUCLEOTIDE FREQUENCY
## The expected frequency of dinucleotide "ab" based on the frequencies
## of its individual letters "a" and "b" is:
## exp_Fab = Fa * Fb / N if the 2 letters are different (e.g. CG)
## exp_Faa = Fa * (Fa-1) / N if the 2 letters are the same (e.g. TT)
## where Fa and Fb are the frequencies of "a" and "b" (respectively) and
## N the length of the sequence.

## Here is a simple function that implements the above formula for a
## DNAString object 'x'. The expected frequencies are returned in a 4x4
## matrix where the rownames and colnames correspond to the 1st and 2nd
## base in the dinucleotide:
expectedDinucleotideFrequency <- function(x)
{
  # Individual base frequencies.
  bf <- alphabetFrequency(x, baseOnly=TRUE)[DNA_BASES]
  (as.matrix(bf) %*% t(bf) - diag(bf)) / length(x)
}

## On Celegans chrI:
library(BSgenome.Celegans.UCSC.ce2)
chrI <- Celegans$chrI
obs_df <- dinucleotideFrequency(chrI, as.matrix=TRUE)
obs_df # CG has the lowest frequency
exp_df <- expectedDinucleotideFrequency(chrI)
## A sanity check:
stopifnot(as.integer(sum(exp_df)) == sum(obs_df))

## Ratio of observed frequency to expected frequency:
obs_df / exp_df # TA has the lowest ratio, not CG!

## C. nucleotideFrequencyAt()
nucleotideFrequencyAt(probes, 13)
nucleotideFrequencyAt(probes, c(13, 20))
nucleotideFrequencyAt(probes, c(13, 20), as.array=FALSE)

## nucleotideFrequencyAt() can be used to answer questions like: "how
## many probes in the drosophila2 chip have T, G, T, A at position
## 2, 4, 13 and 20, respectively?"
nucleotideFrequencyAt(probes, c(2, 4, 13, 20))["T", ",", "T", ",", "A"]
## or "what's the probability to have an A at position 25 if there is
## one at position 13?"
nf <- nucleotideFrequencyAt(probes, c(13, 25))
sum(nf["A", ",", "A"]) / sum(nf["A", "])  # A
## Probabilities to have other bases at position 25 if there is an A
## at position 13:
sum(nf["A", ",", "C"]) / sum(nf["A", "])  # C
sum(nf["A", ",", "G"]) / sum(nf["A", "])  # G
sum(nf["A", ",", "T"]) / sum(nf["A", "])  # T

## See ?hasLetterAt for another way to get those results.

## ---------------------------------------------------------------------
## D. oligonucleotideTransitions()
## ---------------------------------------------------------------------
## Get nucleotide transition matrices for yeast1
oligonucleotideTransitions(yeast1)
oligonucleotideTransitions(yeast1, 2, as.prob=TRUE)

## ---------------------------------------------------------------------
## E. ADVANCED *Frequency() EXAMPLES
## ---------------------------------------------------------------------
## Note that when dropping the dimensions of the 'tri' array, elements
## in the resulting vector are ordered as if they were obtained with
## 'fast.moving.side="left"':
triL <- trinucleotideFrequency(yeast1, fast.moving.side="left")
all(as.vector(tri) == triL)  # TRUE

## Convert the trinucleotide frequency into the amino acid frequency
## based on translation:
tri1 <- trinucleotideFrequency(yeast1)
names(tri1) <- GENETIC_CODE[names(tri1)]
sapply(split(tri1, names(tri1)), sum)  # 12512 occurrences of the stop codon

## When the returned vector is very long (e.g. width >= 10), using
## 'with.labels=FALSE' can improve performance significantly.
## Here for example, the observed speed up is between 25x and 500x:
f12 <- oligonucleotideFrequency(yeast1, 12, with.labels=FALSE)  # very fast!

## With the use of 'step', trinucleotideFrequency() is a very fast way to
## calculate the codon usage table in an ORF (or a set of ORFs).
## Taking the same example as in '?codons':
file <- system.file("extdata", "someORF.fa", package="Biostrings")
my_ORFs <- readDNAStringSet(file)
## Strip flanking 1000 nucleotides around each ORF and remove first
## sequence as it contains an intron:
my_ORFs <- DNAStringSet(my_ORFs, start=1001, end=-1001)[-1]
## Codon usage for each ORF:
codon_usage <- trinucleotideFrequency(my_ORFs, step=3)
## Codon usage across all ORFs:
global_codon_usage <- trinucleotideFrequency(my_ORFs, step=3,
  simplify.as="collapsed")
stopifnot(all(colSums(codon_usage) == global_codon_usage)) # sanity check

## Some related functions:
dict1 <- mkAllStrings(LETTERS[1:3], 4)
dict2 <- mkAllStrings(LETTERS[1:3], 4, fast.moving.side="left")
stopifnot(identical(reverse(dict1), dict2))

---

**padAndClip**  
*Pad and clip strings*

### Description

padAndClip first conceptually pads the supplied strings with an infinite number of padding letters on both sides, then clip them.  

stackStrings is a convenience wrapper to padAndClip that turns a variable-width set of strings into a rectangular (i.e. constant-width) set, by padding and clipping the strings, after conceptually shifting them horizontally.

### Usage

```r
padAndClip(x, views, Lpadding.letter=" ", Rpadding.letter=" ",
  remove.out.of.view.strings=FALSE)
```

```r
stackStrings(x, from, to, shift=0L,
  Lpadding.letter=" ", Rpadding.letter=" ",
  remove.out.of.view.strings=FALSE)
```

### Arguments

- **x**  
  An *XStringSet* object containing the strings to pad and clip.

- **views**  
  A *IntegerRanges* object (recycled to the length of x if necessary) defining the region to keep for each string. Because the strings are first conceptually padded with an infinite number of padding letters on both sides, regions can go beyond string limits.

- **Lpadding.letter, Rpadding.letter**  
  A single letter to use for padding on the left, and another one to use for padding on the right. Note that the default letter (" ") does not work if, for example, x is a *DNAStringSet* object, because the space is not a valid DNA letter (see ?DNA_ALPHABET). So the Lpadding.letter and Rpadding.letter arguments *must* be supplied if x is not a *BStringSet* object. For example, if x is a *DNAStringSet* object, a typical choice is to use "+".
padAndClip

remove.out.of.view.strings
TRUE or FALSE. Whether or not to remove the strings that are out of view in the returned object.

from, to
Another way to specify the region to keep for each string, but with the restriction that from and to must be single integers. So only 1 region can be specified, and the same region is used for all the strings.

shift
An integer vector (recycled to the length of x if necessary) specifying the amount of shifting (in number of letters) to apply to each string before doing pad and clip. Positive values shift to the right and negative values to the left.

Value
For padAndClip: An XStringSet object. If remove.out.of.view.strings is FALSE, it has the same length and names as x, and its "shape", which is described by the integer vector returned by width(), is the same as the shape of the views argument after recycling.

The class of the returned object is the direct concrete subclass of XStringSet that x belongs to or derives from. There are 4 direct concrete subclasses of the XStringSet virtual class: BStringSet, DNAStringSet, RNAStringSet, and AAStringSet. If x is an instance of one of those classes, then the returned object has the same class as x (i.e. in that case, padAndClip acts as an endomorphism). But if x derives from one of those 4 classes, then the returned object is downgraded to the class x derives from. In that case, padAndClip does not act as an endomorphism.

For stackStrings: Same as padAndClip. In addition it is guaranteed to have a rectangular shape i.e. to be a constant-width XStringSet object.

Author(s)
H. Pagès

See Also
- The stackStringsFromBam function in the GenomicAlignments package for stacking the read sequences (or their quality strings) stored in a BAM file on a region of interest.
- The XStringViews class to formally represent a set of views on a single string.
- The extractAt and replaceAt functions for extracting/replacing arbitrary substrings from/in a string or set of strings.
- The XStringSet class.
- The IntegerRanges class in the IRanges package.

Examples
```
x <- BStringSet(c(seq1="ABCD", seq2="abcdefghijk", seq3="", seq4="XYZ"))
padAndClip(x, IRanges(3, 8:5), Lpadding.letter=">", Rpadding.letter="<")
padAndClip(x, IRanges(1:-2, 7), Lpadding.letter=">", Rpadding.letter="<")
stackStrings(x, 2, 8)
stackStrings(x, -2, 8, shift=c(0, -11, 6, 7),
```

The PDict class is a container for storing a preprocessed dictionary of DNA patterns that can later be passed to the `matchPDict` function for fast matching against a reference sequence (the subject). PDict is the constructor function for creating new PDict objects.

**Usage**

```r
PDict(x, max.mismatch=NA, tb.start=NA, tb.end=NA, tb.width=NA, algorithm="ACtree2", skip.invalid.patterns=FALSE)
```

**Arguments**

- **x**: A character vector, a `DNAStringSet` object or an `XStringViews` object with a `DNAString` subject.
- **max.mismatch**: A single non-negative integer or `NA`. See the "Allowing a small number of mismatching letters" section below.
- **tb.start**, **tb.end**, **tb.width**: A single integer or `NA`. See the "Trusted Band" section below.
- **algorithm**: "ACtree2" (the default) or "Twobit".
- **skip.invalid.patterns**: This argument is not supported yet (and might in fact be replaced by the filter argument very soon).
Details

THIS IS STILL WORK IN PROGRESS!

If the original dictionary x is a character vector or an XStringViews object with a DNAString subject, then the PDict constructor will first try to turn it into a DNAStringSet object.

By default (i.e. if PDict is called with max.mismatch=NA, tb.start=NA, tb.end=NA and tb.width=NA) the following limitations apply: (1) the original dictionary can only contain base letters (i.e. only As, Cs, Gs, and Ts), therefore IUPAC ambiguity codes are not allowed; (2) all the patterns in the dictionary must have the same length ("constant width" dictionary); and (3) later matchPdict can only be used with max.mismatch=0.

A Trusted Band can be used in order to relax these limitations (see the "Trusted Band" section below).

If you are planning to use the resulting PDict object in order to do inexact matching where valid hits are allowed to have a small number of mismatching letters, then see the "Allowing a small number of mismatching letters" section below.

Two preprocessing algorithms are currently supported: algorithm="ACtree2" (the default) and algorithm="Twobit". With the "ACtree2" algorithm, all the oligonucleotides in the Trusted Band are stored in a 4-ary Aho-Corasick tree. With the "Twobit" algorithm, the 2-bit-per-letter signatures of all the oligonucleotides in the Trusted Band are computed and the mapping from these signatures to the 1-based position of the corresponding oligonucleotide in the Trusted Band is stored in a way that allows very fast lookup. Only PDict objects preprocessed with the "ACtree2" algo can then be used with matchPdict (and family) and with fixed="pattern" (instead of fixed=TRUE, the default), so that IUPAC ambiguity codes in the subject are treated as ambiguities. PDict objects obtained with the "Twobit" algo don’t allow this. See ?`matchPDict-inexact` for more information about support of IUPAC ambiguity codes in the subject.

Trusted Band

What’s a Trusted Band?

A Trusted Band is a region defined in the original dictionary where the limitations described above will apply.

Why use a Trusted Band?

Because the limitations described above will apply to the Trusted Band only! For example the Trusted Band cannot contain IUPAC ambiguity codes but the "head" and the "tail" can (see below for what those are). Also with a Trusted Band, if matchPdict is called with a non-null max.mismatch value then mismatching letters will be allowed in the head and the tail. Or, if matchPdict is called with fixed="subject", then IUPAC ambiguity codes in the head and the tail will be treated as ambiguities.

How to specify a Trusted Band?

Use the tb.start, tb.end and tb.width arguments of the PDict constructor in order to specify a Trusted Band. This will divide each pattern in the original dictionary into three parts: a left part, a middle part and a right part. The middle part is defined by its starting and ending nucleotide positions given relatively to each pattern thru the tb.start, tb.end and tb.width arguments. It must have the same length for all patterns (this common length is called the width of the Trusted Band). The left and right parts are defined implicitly: they are the parts that remain before (prefix) and after (suffix) the middle part, respectively. Therefore three DNAStringSet objects result from
this division: the first one is made of all the left parts and forms the head of the PDict object, the second one is made of all the middle parts and forms the Trusted Band of the PDict object, and the third one is made of all the right parts and forms the tail of the PDict object.

In other words you can think of the process of specifying a Trusted Band as drawing 2 vertical lines on the original dictionary (note that these 2 lines are not necessarily straight lines but the horizontal space between them must be constant). When doing this, you are dividing the dictionary into three regions (from left to right): the head, the Trusted Band and the tail. Each of them is a DNAStringSet object with the same number of elements than the original dictionary and the original dictionary could easily be reconstructed from those three regions.

The width of the Trusted Band must be \( \geq 1 \) because Trusted Bands of width 0 are not supported.

Finally note that calling \( \text{PDict} \) with \( \text{tb.start=} \text{NA}, \text{tb.end=} \text{NA} \) and \( \text{tb.width=} \text{NA} \) (the default) is equivalent to calling it with \( \text{tb.start=} 1, \text{tb.end=} -1 \) and \( \text{tb.width=} \text{NA} \), which results in a full-width Trusted Band i.e. a Trusted Band that covers the entire dictionary (no head and no tail).

### Allowing a small number of mismatching letters

[TODO]

### Accessor methods

In the code snippets below, \( x \) is a PDict object.

- \( \text{length}(x) \): The number of patterns in \( x \).
- \( \text{width}(x) \): A vector of non-negative integers containing the number of letters for each pattern in \( x \).
- \( \text{names}(x) \): The names of the patterns in \( x \).
- \( \text{head}(x) \): The head of \( x \) or NULL if \( x \) has no head.
- \( \text{tb}(x) \): The Trusted Band defined on \( x \).
- \( \text{tb.width}(x) \): The width of the Trusted Band defined on \( x \). Note that, unlike \( \text{width}(\text{tb}(x)) \), this is a single integer. And because the Trusted Band has a constant width, \( \text{tb.width}(x) \) is in fact equivalent to \( \text{unique}(\text{width}(\text{tb}(x))) \), or to \( \text{width}(\text{tb}(x))[1] \).
- \( \text{tail}(x) \): The tail of \( x \) or NULL if \( x \) has no tail.

### Subsetting methods

In the code snippets below, \( x \) is a PDict object.

- \( x[[i]] \): Extract the \( i \)-th pattern from \( x \) as a DNAString object.

### Other methods

In the code snippet below, \( x \) is a PDict object.

- \( \text{duplicated}(x) \): [TODO]
- \( \text{patternFrequency}(x) \): [TODO]

### Author(s)

H. Pagès
References


See Also

matchPDict, DNA_ALPHABET, IUPAC_CODE_MAP, DNAStringSet-class, XStringViews-class

Examples

```r
## A. NO HEAD AND NO TAIL (THE DEFAULT)
## ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe)
dict0 # The original dictionary.
length(dict0) # Hundreds of thousands of patterns.
unique(nchar(dict0)) # Patterns are 25-mers.
pdict0 <- PDict(dict0) # Store the original dictionary in a PDict object (preprocessing).

head(pdict0)
tb(pdict0)
tb.width(pdict0)

## B. NO HEAD AND A TAIL
## ~~~~~~~~~~~~~~~~~~~~~
dict1 <- c("ACNG", "GT", "CGT", "AC")
pdict1 <- PDict(dict1, tb.end=2)

width(pdict1) # 9 patterns are repeated 3 times.
pdict1[[3]]
```
pmatchPattern  Longest Common Prefix/Suffix/Substring searching functions

Description

Functions for searching the Longest Common Prefix/Suffix/Substring of two strings.

WARNING: These functions are experimental and might not work properly! Full documentation will come later.

Thanks for your comprehension!

Usage

\[
\begin{align*}
lcprefix(s1, s2) \\
lcsuffix(s1, s2) \\
lcsubstr(s1, s2) \\
pmatchPattern(pattern, subject, maxlength.out=1L)
\end{align*}
\]

Arguments

- **s1**: 1st string, a character string or an XString object.
- **s2**: 2nd string, a character string or an XString object.
- **pattern**: The pattern string.
- **subject**: An XString object containing the subject string.
- **maxlength.out**: The maximum length of the output i.e. the maximum number of views in the returned object.

See Also

- matchPattern, XStringViews-class, XString-class

QualityScaledXStringSet-class

QualityScaledBStringSet, QualityScaledDNAStringSet, QualityScaledRNAStringSet and QualityScaledAAStringSet objects

Description

The QualityScaledBStringSet class is a container for storing a BStringSet object with an XStringQuality object.

Similarly, the QualityScaledDNAStringSet (or QualityScaledRNAStringSet, or QualityScaledAAStringSet) class is a container for storing a DNAStringSet (or RNAStringSet, or AAStringSet) objects with an XStringQuality object.
Usage

## Constructors:

- `QualityScaledBStringSet(x, quality)`
- `QualityScaledDNAStringSet(x, quality)`
- `QualityScaledRNAStringSet(x, quality)`
- `QualityScaledAAStringSet(x, quality)`

## Read/write a QualityScaledXStringSet object from/to a FASTQ file:

- `readQualityScaledDNAStringSet(filepath, quality.scoring=c("phred", "solexa", "illumina"), nrec=-1L, skip=0L, seek.first.rec=FALSE, use.names=TRUE)`
- `writeQualityScaledXStringSet(x, filepath, append=FALSE, compress=FALSE, compression_level=NA)`

Arguments

- **x**: For the `QualityScaledXStringSet` constructors: Either a character vector, or an `XString`, `XStringSet` or `XStringViews` object. For `writeQualityScaledXStringSet`: A `QualityScaledDNAStringSet` object or other `QualityScaledXStringSet` derivative.
- **quality**: An `XStringQuality` derivative.
- **quality.scoring**: Specify the quality scoring used in the FASTQ file. Must be one of "phred" (the default), "solexa", or "illumina". If set to "phred" (or "solexa" or "illumina"), the qualities will be stored in a `PhredQuality` (or `SolexaQuality` or `IlluminaQuality`, respectively) object.

Details

The `QualityScaledBStringSet`, `QualityScaledDNAStringSet`, `QualityScaledRNAStringSet` and `QualityScaledAAStringSet` functions are constructors that can be used to "naturally" turn `x` into a `QualityScaledXStringSet` object of the desired base type.

Accessor methods

The `QualityScaledXStringSet` class derives from the `XStringSet` class hence all the accessor methods defined for an `XStringSet` object can also be used on a `QualityScaledXStringSet` object. Common methods include (in the code snippets below, `x` is a `QualityScaledXStringSet` object):

- `length(x)`: The number of sequences in `x`.
- `width(x)`: A vector of non-negative integers containing the number of letters for each element in `x`. 
nchar(x): The same as width(x).

names(x): NULL or a character vector of the same length as x containing a short user-provided description or comment for each element in x.

quality(x): The quality of the strings.

Subsetting and appending

In the code snippets below, x and values are XStringSet objects, and i should be an index specifying the elements to extract.

x[i]: Return a new QualityScaledXStringSet object made of the selected elements.

Author(s)

P. Aboyoun

See Also

• BStringSet, DNAStringSet, RNAStringSet, and AAStringSet objects.
• XStringQuality objects.
• readDNAStringSet and writeXStringSet for reading/writing a DNAStringSet object (or other XStringSet derivative) from/to a FASTA or FASTQ file.

Examples

```r
## QualityScaledXStringSet() CONSTRUCTORS
x1 <- DNAStringSet(c("TTGA", "CTCN"))
q1 <- PhredQuality(c("*+,-", "6789"))
qdna1 <- QualityScaledDNAStringSet(x1, q1)
qdna1

## READ/WRITE A QualityScaledDNAStringSet OBJECT FROM/TO A FASTQ FILE
filepath <- system.file("extdata", "s_1_sequence.txt",
package="Biostrings")

# By default, readQualityScaledDNAStringSet() assumes that the FASTQ file contains "Phred quality scores" (this is the standard Sanger variant to assess reliability of a base call):
qdna2 <- readQualityScaledDNAStringSet(filepath)
qdna2

outfile2a <- tempfile()
writeQualityScaledXStringSet(qdna2, outfile2a)

outfile2b <- tempfile()
```
writeQualityScaledXStringSet(qdna2, outfile2b, compress=TRUE)

## Use 'quality.scoring="solexa"' or 'quality.scoring="illumina"' if the
## quality scores are Solexa quality scores:
qdna3 <- readQualityScaledDNAStringSet(filepath, quality.scoring="solexa")

outfile3a <- tempfile()
writeQualityScaledXStringSet(qdna3, outfile3a)

outfile3b <- tempfile()
writeQualityScaledXStringSet(qdna3, outfile3b, compress=TRUE)

## Sanity checks:
stopifnot(identical(readLines(outfile2a), readLines(filepath)))
stopifnot(identical(readLines(outfile2a), readLines(outfile2b)))
stopifnot(identical(readLines(outfile3a), readLines(filepath)))
stopifnot(identical(readLines(outfile3a), readLines(outfile3b)))

replaceAt

**Extract/replace arbitrary substrings from/in a string or set of strings.**

**Description**

`extractAt` extracts multiple subsequences from `XString` object `x`, or from the individual sequences of `XStringSet` object `x`, at the ranges of positions specified thru `at`.

`replaceAt` performs multiple subsequence replacements (a.k.a. substitutions) in `XString` object `x`, or in the individual sequences of `XStringSet` object `x`, at the ranges of positions specified thru `at`.

**Usage**

```r
extractAt(x, at)
replaceAt(x, at, value="")
```

**Arguments**

- `x` An `XString` or `XStringSet` object.
- `at` Typically a `IntegerRanges` object if `x` is an `XString` object, and an `IntegerRangesList` object if `x` is an `XStringSet` object. Alternatively, the ranges can be specified with only 1 number per range (its start position), in which case they are considered to be empty ranges (a.k.a. zero-width ranges). So if `at` is a numeric vector, an `IntegerList` object, or a list of numeric vectors, each number in it is interpreted as the start position of a zero-width range. This is useful when using `replaceAt` to perform insertions. The following applies only if `x` is an `XStringSet` object:
  at is recycled to the length of `x` if necessary. If `at` is a `IntegerRanges` object (or a numeric vector), it is first turned into a `IntegerRangesList` object of length 1 and
then this `IntegerRangesList` object is recycled to the length of \( x \). This is useful for specifying the same ranges across all sequences in \( x \). The effective shape of \( at \) is described by its length together with the lengths of its list elements after recycling.

As a special case, `extractAt` accepts \( at \) and \( value \) to be both of length 0, in which case it just returns \( x \) unmodified (no-op).

`value` The replacement sequences.

If \( x \) is an `XString` object, \( value \) is typically a character vector or an `XStringSet` object that is recycled to the length of \( at \) (if necessary).

If \( x \) is an `XStringSet` object, \( value \) is typically a list of character vectors or a `CharacterList` or `XStringSetList` object. If necessary, it is recycled "vertically" first and then "horizontally" to bring it into the effective shape of \( at \) (see above). "Vertical recycling" is the usual recycling whereas "horizontal recycling" recycles the individual list elements.

As a special case, `extractAt` accepts \( at \) and \( value \) to be both of length 0, in which case it just returns \( x \) unmodified (no-op).

**Value**

For `extractAt`: An `XStringSet` object of the same length as \( at \) if \( x \) is an `XString` object. An `XStringSetList` object of the same length as \( x \) (and same effective shape as \( at \)) if \( x \) is an `XStringSet` object.

For `replaceAt`: An object of the same class as \( x \). If \( x \) is an `XStringSet` object, its length and names and metadata columns are preserved.

**Note**

Like `subseq` (defined and documented in the `XVector` package), `extractAt` does not copy the sequence data!

`extractAt` is equivalent to `extractList` (defined and documented in the `IRanges` package) when \( x \) is an `XString` object and at a `IntegerRanges` object.

**Author(s)**

H. Pagès

**See Also**

- The `subseq` and `subseq<-` functions in the `XVector` package for simpler forms of sub-sequence extractions and replacements.
- The `extractList` and `unstrsplit` functions defined and documented in the `IRanges` package.
- The `replaceLetterAt` function for a DNA-specific single-letter replacement functions useful for SNP injections.
- The `padAndClip` function for padding and clipping strings.
- The `XString`, `XStringSet`, and `XStringSetList` classes.
- The `IntegerRanges`, `IntegerRangesList`, `IntegerList`, and `CharacterList` classes defined and documented in the `IRanges` package.
Examples

## ---------------------------------------------------------------------
## (A) ON AN XString OBJECT
## ---------------------------------------------------------------------

```r
x <- BString("abcdefghijklm")

at1 <- IRanges(5:1, width=3)
extractAt(x, at1)
names(at1) <- LETTERS[22:26]
extractAt(x, at1)

at2 <- IRanges(c(1, 5, 12), c(3, 4, 12), names=c("X", "Y", "Z"))
extractAt(x, at2)
extractAt(x, rev(at2))

value <- c("+", "-", "+")
replaceAt(x, at2, value=value)
replaceAt(x, rev(at2), value=rev(value))

at3 <- IRanges(c(14, 1, 1, 1, 1, 11), c(13, 0, 10, 0, 0, 10))
value <- 1:6
replaceAt(x, at3, value=value)  # "24536klm1"
replaceAt(x, rev(at3), value=rev(value))  # "54236klm1"

## Deletions:
stopifnot(replaceAt(x, at2) == "defghjkm")
stopifnot(replaceAt(x, rev(at2)) == "defghjkm")
stopifnot(replaceAt(x, at3) == "klm")
stopifnot(replaceAt(x, rev(at3)) == "klm")

## Insertions:

at4 <- IRanges(c(6, 10, 2, 5), width=0)
stopifnot(replaceAt(x, at4, value="-") == "a-bcd-e-fghi-jklm")
stopifnot(replaceAt(x, start(at4), value="-") == "a-bcd-e-fghi-jklm")
at5 <- c(5, 1, 6, 5)  # 2 insertions before position 5
replaceAt(x, at5, value=c("+", "-", "+", "/"))

## No-ops:
stopifnot(replaceAt(x, NULL, value=NULL) == x)
stopifnot(replaceAt(x, at2, value=extractAt(x, at2)) == x)
stopifnot(replaceAt(x, at3, value=extractAt(x, at3)) == x)
stopifnot(replaceAt(x, at4, value=extractAt(x, at4)) == x)
stopifnot(replaceAt(x, at5, value=extractAt(x, at5)) == x)

## The order of successive transformations matters:
## T1: insert "+" before position 1 and 4
## T2: insert "-" before position 3

## T1 followed by T2
x2a <- replaceAt(x, c(1, 4), value="+")
x3a <- replaceAt(x2a, 3, value="-")
```
## T2 followed by T1
x2b <- replaceAt(x, 3, value="-")
x3b <- replaceAt(x2b, c(1, 4), value="+")

## T1 and T2 simultaneously:
x3c <- replaceAt(x, c(1, 3, 4), value=c("+", "-", "+"))

## T2 followed by T1
x2b <- replaceAt(x, 3, value="-")
x3b <- replaceAt(x2b, c(1, 4), value="+")

## T1 and T2 simultaneously:
x3c <- replaceAt(x, c(1, 3, 4), value=c("+", "-", "+"))

## Append "**" to 'x3c':
replaceAt(x3c, length(x3c) + 1L, value="**")

## Append "**" to 'x3c':
replaceAt(x3c, length(x3c) + 1L, value="**")

## (B) ON AN XStringSet OBJECT
## ---------------------------------------------------------------------
x <- BStringSet(c(seq1="ABCD", seq2="abcdefghijk", seq3="XYZ"))
at6 <- IRanges(c(1, 3), width=1)
extractAt(x, at=at6)
unstrsplit(extractAt(x, at=at6))
at7 <- IRangesList(IRanges(c(2, 1), c(3, 0)),
                   IRanges(c(7, 2, 12, 7), c(6, 5, 11, 8)),
                   IRanges(2, 2))

## Set inner names on 'at7'.
unlisted_at7 <- unlist(at7)
names(unlisted_at7) <-
    paste0("rg", sprintf("%02d", seq_along(unlisted_at7)))
at7 <- relist(unlisted_at7, at7)

extractAt(x, at7) # same as 'as(mapply(extractAt, x, at7), "List")'
extraaxtAt(x, at7[3]) # same as 'as(mapply(extractAt, x, at7[3]), "List")'

replaceAt(x, at7, value=extractAt(x, at7)) # no-op
replaceAt(x, at7) # deletions

at8 <- IRangesList(IRanges(1:5, width=0),
                   IRanges(c(6, 8, 10, 7, 2, 5),
                           width=c(0, 2, 0, 0, 0, 0)),
                   IRanges(c(1, 2, 1), width=c(0, 1, 0)))
replaceAt(x, at8, value="-")
value8 <- relist(paste0("[", seq_along(unlist(at8)), "]"), at8)
replaceAt(x, at8, value=value8)
replaceAt(x, at8, value=as(c("+", "-", "+"), "List"))

## Append "**" to all sequences:
replaceAt(x, as(width(x) + 1L, "List"), value="**")

---

## (C) ADVANCED EXAMPLES
library(hgu95av2probe)
probes <- DNAStringSet(hgu95av2probe)
replaceLetterAt

Replacing letters in a sequence (or set of sequences) at some specified locations

Description

replaceLetterAt first makes a copy of a sequence (or set of sequences) and then replaces some of the original letters by new letters at the specified locations.

.inplaceReplaceLetterAt is the IN PLACE version of replaceLetterAt: it will modify the original sequence in place i.e. without copying it first. Note that in place modification of a sequence is fundamentally dangerous because it alters all objects defined in your session that make reference to the modified sequence. NEVER use .inplaceReplaceLetterAt, unless you know what you are doing!

Usage

replaceLetterAt(x, at, letter, if.not.extending="replace", verbose=FALSE)

## NEVER USE THIS FUNCTION!
.inplaceReplaceLetterAt(x, at, letter)
**Arguments**

- **x**: A DNAString or rectangular DNAStringSet object.
- **at**: The locations where the replacements must occur. If `x` is a DNAString object, then `at` is typically an integer vector with no NAs but a logical vector or Rle object is valid too. Locations can be repeated and in this case the last replacement to occur at a given location prevails. If `x` is a rectangular DNAStringSet object, then `at` must be a matrix of logicals with the same dimensions as `x`.
- **letter**: The new letters. If `x` is a DNAString object, then `letter` must be a DNAString object or a character vector (with no NAs) with a total number of letters (\(\text{sum(nchar(letter))}\)) equal to the number of locations specified in `at`. If `x` is a rectangular DNAStringSet object, then `letter` must be a DNAStringSet object or a character vector of the same length as `x`. In addition, the number of letters in each element of `letter` must match the number of locations specified in the corresponding row of `at` (\(\text{all(width(letter) == rowSums(at))}\)).
- **if.not.extending**: What to do if the new letter is not "extending" the old letter? The new letter "extends" the old letter if both are IUPAC letters and the new letter is as specific or less specific than the old one (e.g. M extends A, Y extends Y, but Y doesn’t extend S). Possible values are "replace" (the default) for replacing in all cases, "skip" for not replacing when the new letter does not extend the old letter, "merge" for merging the new IUPAC letter with the old one, and "error" for raising an error. Note that the gap ("-"), and hard masking ("+"), letters are not extending or extended by any other letter. Also note that "merge" is the only value for the if.not.extending argument that guarantees the final result to be independent on the order the replacement is performed (although this is only relevant when `at` contains duplicated locations, otherwise the result is of course always independent on the order, whatever the value of if.not.extending is).
- **verbose**: When TRUE, a warning will report the number of skipped or merged letters.

**Details**

.inplaceReplaceLetterAt semantic is equivalent to calling replaceLetterAt with if.not.extending="merge" and verbose=FALSE.

Never use .inplaceReplaceLetterAt! It is used by the injectSNPs function in the BSgenome package, as part of the "lazy sequence loading" mechanism, for altering the original sequences of a BSgenome object at "sequence-load time". This alteration consists in injecting the IUPAC ambiguity letters representing the SNPs into the just loaded sequence, which is the only time where in place modification of the external data of an XString object is safe.

**Value**

A DNAString or DNAStringSet object of the same shape (i.e. length and width) as the original object `x` for replaceLetterAt.
reverseComplement

Sequence reversing and complementing

Description

Use these functions for reversing sequences and/or complementing DNA or RNA sequences.

Usage

complement(x, ...)
reverseComplement(x, ...)  

Arguments

x  A DNAString, RNAString, DNAStringSet, RNAStringSet, XStringViews (with DNAString or RNAString subject), MaskedDNAString or MaskedRNAString object for complement and reverseComplement.

...  Additional arguments to be passed to or from methods.
Details

See `reverse` for reversing an `XString`, `XStringSet` or `XStringViews` object.

If `x` is a `DNAString` or `RNAString` object, `complement(x)` returns an object where each base in `x` is "complemented" i.e. A, C, G, T in a `DNAString` object are replaced by T, G, C, A respectively and A, C, G, U in a `RNAString` object are replaced by U, G, C, A respectively.

Letters belonging to the IUPAC Extended Genetic Alphabet are also replaced by their complement (M <-> K, R <-> Y, S <-> S, V <-> B, W <-> W, H <-> D, N <-> N) and the gap ("-" ) and hard masking ("+" ) letters are unchanged.

`reverseComplement(x)` is equivalent to `reverse(complement(x))` but is faster and more memory efficient.

Value

An object of the same class and length as the original object.

See Also

`reverse`, `DNAString-class`, `RNAString-class`, `DNAStringSet-class`, `RNAStringSet-class`, `XStringViews-class`, `MaskedXString-class`, `chartr`, `findPalindromes`, `IUPAC_CODE_MAP`

Examples

## ---------------------------------------------------------------------
## A. SOME SIMPLE EXAMPLES
## ---------------------------------------------------------------------

```r
x <- DNAString("ACGT-YN-")
reverseComplement(x)
```

```r
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
probes
alphabetFrequency(probes, collapse=TRUE)
rcprobes <- reverseComplement(probes)
rcprobes
alphabetFrequency(rcprobes, collapse=TRUE)
```

## ---------------------------------------------------------------------
## B. OBTAINING THE MISMATCH PROBES OF A CHIP
## ---------------------------------------------------------------------

```r
pm2mm <- function(probes)
{
  probes <- DNAStringSet(probes)
  subseq(probes, start=13, end=13) <- complement(subseq(probes, start=13, end=13))
  probes
}
mmprobes <- pm2mm(probes)
mmprobes
alphabetFrequency(mmprobes, collapse=TRUE)
```
### C. SEARCHING THE MINUS STRAND OF A CHROMOSOME

Applying `reverseComplement()` to the pattern before calling `matchPattern()` is the recommended way of searching hits on the minus strand of a chromosome.

```r
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- Dmelanogaster$chrX
pattern <- DNAString("ACCAACNNGGTTG")
matchPattern(pattern, chrX, fixed=FALSE) # 3 hits on strand +
rcpattern <- reverseComplement(pattern)
matchPattern(rcpattern, chrX, fixed=FALSE) # 5 hits on strand -
```

Applying `reverseComplement()` to the subject instead of the pattern is not a good idea for 2 reasons:

1. Chromosome sequences are generally big and sometimes very big so computing the reverse complement of the positive strand will take time and memory proportional to its length.
2. Chromosome locations are generally given relatively to the positive strand, even for features located in the negative strand, so after doing this:

```r
chrXminus <- reverseComplement(chrX) # needs to allocate 22M of memory!
chrXminus
```

After doing this:

```r
matchPattern(pattern, chrXminus, fixed=FALSE)
```

You need to apply `reverseComplement()` again on the result if you want them to be relative to the positive strand:

```r
matchPattern(pattern, reverseComplement(chrX)) # DON'T DO THIS!
```

and finally to apply `rev()` to sort the matches from left to right (5'3' direction) like in m0:

```r
m3 <- rev(m2) # same as m0, finally!
```

### WARNING: Before you try the example below on human chromosome 1, be aware that it will require the allocation of about 500Mb of memory!

```r
if (interactive()) {
  library(BSgenome.Hsapiens.UCSC.hg18)
  chr1 <- Hsapiens$chr1
  matchPattern(pattern, reverseComplement(chr1)) # DON'T DO THIS!
  matchPattern(reverseComplement(pattern), chr1) # DO THIS INSTEAD
}
```

---

**RNAString-class**

**RNAString objects**

### Description

An RNAString object allows efficient storage and manipulation of a long RNA sequence.
Details

The RNAString class is a direct XString subclass (with no additional slot). Therefore all functions
and methods described in the XString man page also work with an RNAString object (inheritance).
Unlike the BString container that allows storage of any single string (based on a single-byte charac-
ter set) the RNAString container can only store a string based on the RNA alphabet (see below). In
addition, the letters stored in an RNAString object are encoded in a way that optimizes fast search
algorithms.

The RNA alphabet

This alphabet is the same as the DNA alphabet, except that "T" is replaced by "U". See ?DNA_ALPHABET
for more information about the DNA alphabet. The RNA alphabet is stored in the RNA_ALPHABET
predefined constant (character vector).

The alphabet() function returns RNA_ALPHABET when applied to an RNAString object.

Constructor-like functions and generics

In the code snippet below, x can be a single string (character vector of length 1), a BString object
or a DNAString object.

RNAString(x="", start=1, nchar=NA): Tries to convert x into an RNAString object by reading
nchar letters starting at position start in x.

Accessor methods

In the code snippet below, x is an RNAString object.

alphabet(x, baseOnly=FALSE): If x is an RNAString object, then return the RNA alphabet (see
above). See the corresponding man pages when x is a BString, DNAString or AAString object.

Display

The letters in an RNAString object are colored when displayed by the show() method. Set global
option Biostrings.coloring to FALSE to turn off this coloring.

Author(s)

H. Pagès

See Also

• The RNAStringSet class to represent a collection of RNAString objects.
• The XString and DNAString classes.
• reverseComplement
• alphabetFrequency
• IUPAC_CODE_MAP
• letter
Examples

RNA_BASES
RNA_ALPHABET
dna <- DNAString("TTGAAAA-CTC-N")
rna <- RNAString(dna)
rna # 'options(Biostrings.coloring=FALSE)' to turn off coloring

alphabet(rna) # RNA_ALPHABET
alphabet(rna, baseOnly=TRUE) # RNA_BASES

## When comparing an RNAString object with a DNAString object,
## U and T are considered equals:
rna == dna # TRUE

seqinfo-methods

seqinfo() method for DNAStringSet objects

Description

seqinfo methods for extracting the sequence information stored in a DNAStringSet object.

Usage

## S4 method for signature 'DNAStringSet'
seqinfo(x)

Arguments

x A DNAStringSet object.

Value

A Seqinfo object for the 'seqinfo' getter.
A DNAStringSet object containing sequence information for the 'seqinfo' setter.

See Also

getSeq, DNAStringSet-class,

Examples

## ------------------------------
## A. SIMPLE EXAMPLE
## ------------------------------

library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)

## Check metadata slot: empty
toComplex

Turning a DNA sequence into a vector of complex numbers

Description

The toComplex utility function turns a DNAString object into a complex vector.

Usage

toComplex(x, baseValues)

Arguments

x  A DNAString object.
baseValues  A named complex vector containing the values associated to each base e.g. c(A=1+0i, G=0+1i, T=-1+0i, C=0-1i)

Value

A complex vector of the same length as x.

Author(s)

H. Pagès

See Also

DNAString

Examples

seq <- DNAString("accacctgaccattgtcct")
baseValues1 <- c(A=1+0i, G=0+1i, T=-1+0i, C=0-1i)
toComplex(seq, baseValues1)

## GC content:
baseValues2 <- c(A=0, C=1, G=1, T=0)
sum(as.integer(toComplex(seq, baseValues2)))
## Note that there are better ways to do this (see ?alphabetFrequency)
translate  

Translating DNA/RNA sequences

Description

Functions for translating DNA or RNA sequences into amino acid sequences.

Usage

```r
## Translating DNA/RNA:
translate(x, genetic.code=GENETIC_CODE, no.init.codon=FALSE,
         if.fuzzy.codon="error")

## Extracting codons without translating them:
codons(x)
```

Arguments

- **x**
  A DNAStringSet, RNAStringSet, DNAString, RNAString, MaskedDNAString or MaskedRNAString object for translate.
  A DNAString, RNAString, MaskedDNAString or MaskedRNAString object for codons.

- **genetic.code**
  The genetic code to use for the translation of codons into Amino Acid letters. It must be represented as a named character vector of length 64 similar to predefined constant `GENETIC_CODE`. More precisely:
  - it must contain 1-letter strings in the Amino Acid alphabet;
  - its names must be identical to `names(GENETIC_CODE)`;
  - it must have an `alt_init_codons` attribute on it, that lists the alternative initiation codons.


- **no.init.codon**
  By default, translate() assumes that the first codon in a DNA or RNA sequence is the initiation codon. This means that the alt_init_codons attribute on the supplied genetic.code will be used to translate the alternative initiation codons. This can be changed by setting no.init.codon to TRUE, in which case the alt_init_codons attribute will be ignored.

- **if.fuzzy.codon**
  How fuzzy codons (i.e codon with IUPAC ambiguities) should be handled. Accepted values are:
  - "error": An error will be raised on the first occurrence of a fuzzy codon. This is the default.
  - "solve": Fuzzy codons that can be translated non ambiguously to an amino acid or to * (stop codon) will be translated. Ambiguous fuzzy codons will be translated to X.
- "error.if.X": Fuzzy codons that can be translated non ambiguously to an amino acid or to * (stop codon) will be translated. An error will be raised on the first occurrence of an ambiguous fuzzy codon.
- "X": All fuzzy codons (ambiguous and non-ambiguous) will be translated to X.

Alternatively if.fuzzy.codon can be specified as a character vector of length 2 for more fine-grained control. The 1st string and 2nd strings specify how to handle non-ambiguous and ambiguous fuzzy codons, respectively. The accepted values for the 1st string are:
- "error": Any occurrence of a non-ambiguous fuzzy codon will cause an error.
- "solve": Non-ambiguous fuzzy codons will be translated to an amino acid or to *.
- "X": Non-ambiguous fuzzy codons will be translated to X.

The accepted values for the 2nd string are:
- "error": Any occurrence of an ambiguous fuzzy codon will cause an error.
- "X": Ambiguous fuzzy codons will be translated to X.

All the 6 possible combinations of 1st and 2nd strings are supported. Note that if.fuzzy.codon=c("error", "error") is equivalent to if.fuzzy.codon="error", if.fuzzy.codon=c("solve", "error") is equivalent to if.fuzzy.codon="solve", if.fuzzy.codon=c("solve", "error") is equivalent to if.fuzzy.codon="error", and if.fuzzy.codon=c("X", "X") is equivalent to if.fuzzy.codon="X".

Details

translate reproduces the biological process of RNA translation that occurs in the cell. The input of the function can be either RNA or coding DNA. By default the Standard Genetic Code (see ?GENETIC_CODE) is used to translate codons into amino acids but the user can supply a different genetic code via the genetic.code argument.

codons is a utility for extracting the codons involved in this translation without translating them.

Value

For translate: An AAString object when x is a DNAString, RNAString, MaskedDNAString, or MaskedRNAString object. An AAStringSet object parallel to x (i.e. with 1 amino acid sequence per DNA or RNA sequence in x) when x is a DNAStringSet or RNAStringSet object. If x has names on it, they’re propagated to the returned object.

For codons: An XStringViews object with 1 view per codon. When x is a MaskedDNARNAString object, its masked parts are interpreted as introns and filled with the + letter in the returned object. Therefore codons that span across masked regions are represented by views that have a width > 3 and contain the + letter. Note that each view is guaranteed to contain exactly 3 base letters.

See Also

- AA_ALPHABET for the Amino Acid alphabet.
- **GENETIC_CODE** for The Standard Genetic Code and its known variants.
- The examples for `extractTranscriptSeqs` in the **GenomicFeatures** package for computing the full proteome of a given organism.
- The `reverseComplement` function.
- The DNAStringSet and AAStringSet classes.
- The XStringViews and MaskedXString classes.

## Examples

```r
# 1. BASIC EXAMPLES

dna1 <- DNAString("TTGATATGGCCCTTATAA")
translate(dna1)
## TTG is an alternative initiation codon in the Standard Genetic Code:
translate(dna1, no.init.codon=TRUE)

SGC1 <- getGeneticCode("SGC1") # Vertebrate Mitochondrial code
translate(dna1, genetic.code=SGC1)
## TTG is NOT an alternative initiation codon in the Vertebrate Mitochondrial code:
translate(dna1, genetic.code=SGC1, no.init.codon=TRUE)

## All 6 codons except 4th (CCC) are fuzzy:
dna2 <- DNAString("HTGATHTGRCCCYTRTRA")

# Not run:
  translate(dna2) # error because of fuzzy codons
  # End(Not run)

## Translate all fuzzy codons to x:
translate(dna2, if.fuzzy.codon="X")

## Or solve the non-ambiguous ones (3rd codon is ambiguous so cannot be solved):
translate(dna2, if.fuzzy.codon="solve")

## Fuzzy codons that are non-ambiguous with a given genetic code can become ambiguous with another genetic code, and vice versa:
translate(dna2, genetic.code=SGC1, if.fuzzy.codon="solve")

# 2. TRANSLATING AN OPEN READING FRAME

file <- system.file("extdata", "someORF.fa", package="Biostrings")
x <- readDNAStringSet(file)
x
```
The first and last 1000 nucleotides are not part of the ORFs:
```
x <- DNAStringSet(x, start=1001, end=-1001)
```

Before calling `translate()` on an ORF, we need to mask the introns
if any. We can get this information from the SGD database
(see http://www.yeastgenome.org/).

According to SGD, the 1st ORF (YAL001C) has an intron at 71..160
(see http://db.yeastgenome.org/cgi-bin/locus.pl?locus=YAL001C)
```
y1 <- x[[1]]
mask1 <- Mask(length(y1), start=71, end=160)
masks(y1) <- mask1
y1
```
```
translate(y1)
```

Codons:
```
codons(y1)
```
```
which(width(codons(y1)) != 3)
codons(y1)[20:28]
```

Translation on the '-' strand:
```
dna3 <- DNAStringSet(c("ATC", "GCTG", "CGACT")
translate(reverseComplement(dna3))
```

Translate sequences on both '+' and '-' strand across all
possible reading frames (i.e., codon position 1, 2 or 3):
```
dna3_subseqs <- lapply(1:3, function(pos)
  subseq(c(dna3, reverseComplement(dna3)), start=pos))
```
```
# Translation of 'dna3_subseqs' produces a list of length 3, each with
# 6 elements (3 '+' strand results followed by 3 '-' strand results).
lapply(dna3_subseqs, translate)
```

Note that `translate()` throws a warning when the length of the sequence
is not divisible by 3. To avoid this warning wrap the function in
```
suppressWarnings()
```

#### Description

The `trimLRPatterns` function trims left and/or right flanking patterns from sequences.
trimLRPatterns

Usage
trimLRPatterns(Lpattern = "", Rpattern = "", subject, max.Lmismatch = 0, max.Rmismatch = 0, with.Lindels = FALSE, with.Rindels = FALSE, Lfixed = TRUE, Rfixed = TRUE, ranges = FALSE)

Arguments

Lpattern
The left pattern.

Rpattern
The right pattern.

subject
An XString object, XStringSet object, or character vector containing the target sequence(s).

max.Lmismatch
Either an integer vector of length nLp = nchar(Lpattern) representing an absolute number of mismatches (or edit distance if with.Lindels is TRUE) or a single numeric value in the interval [0, 1) representing a mismatch rate when aligning terminal substrings (suffixes) of Lpattern with the beginning (prefix) of subject following the conventions set by neditStartingAt, isMatchingStartingAt, etc.

When max.Lmismatch is 0L or a numeric value in the interval [0, 1), it is taken as a "rate" and is converted to as.integer(1:nLp * max.Lmismatch), analogous to agrep (which, however, employs ceiling).

Otherwise, max.Lmismatch is treated as an integer vector where negative numbers are used to prevent trimming at the i-th location. When an input integer vector is shorter than nLp, it is augmented with enough -1s at the beginning to bring its length up to nLp. Elements of max.Lmismatch beyond the first nLp are ignored.

Once the integer vector is constructed using the rules given above, when with.Lindels is FALSE, max.Lmismatch[i] is the number of acceptable mismatches (errors) between the suffix substring(Lpattern, nLp - i + 1, nLp) of Lpattern and the first i letters of subject. When with.Lindels is TRUE, max.Lmismatch[i] represents the allowed "edit distance" between that suffix of Lpattern and subject, starting at position 1 of subject (as in matchPattern and isMatchingStartingAt).

For a given element s of the subject, the initial segment (prefix) substring(s, 1, j) of s is trimmed if j is the largest i for which there is an acceptable match, if any.

max.Rmismatch
Same as max.Lmismatch but with Rpattern, along with with.Rindels (below), and its initial segments (prefixes) substring(Rpattern, 1, i).

For a given element s of the subject, with nS = nchar(s), the terminal segment (suffix) substring(s, nS - j + 1, nS) of s is trimmed if j is the largest i for which there is an acceptable match, if any.

with.Lindels
If TRUE, indels are allowed in the alignments of the suffixes of Lpattern with the subject, at its beginning. See the with.indels arguments of the matchPattern and neditStartingAt functions for detailed information.

with.Rindels
Same as with.Lindels but for alignments of the prefixes of Rpattern with the subject, at its end. See the with.indels arguments of the matchPattern and neditEndingAt functions for detailed information.
Whether IUPAC extended letters in the left or right pattern should be interpreted as ambiguities (see `?lowlevel-matching` for the details).

If TRUE, then return the ranges to use to trim subject. If FALSE, then returned the trimmed subject.

A new `XString` object, `XStringSet` object, or character vector with the "longest" flanking matches removed, as described above.

P. Aboyoun and H. Jaffee

**Examples**

```r
Lpattern <- "TTCTGCTTG"
Rpattern <- "GATCGGAAG"
subject <- DNAString("TTCTGCTTGACGTGATCGGA")
subjectSet <- DNAStringSet(c("TGCTGACCGCGATCGG", "TTCTGCTTGATCGGAAG"))

## Only allow for perfect matches on the flanks
trimLRPatterns(Lpattern = Lpattern, subject = subject)
trimLRPatterns(Rpattern = Rpattern, subject = subject)
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet)

## Allow for perfect matches on the flanking overlaps
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet,
               max.Lmismatch = 0, max.Rmismatch = 0)

## Allow for mismatches on the flanks
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subject,
               max.Lmismatch = 0.2, max.Rmismatch = 0.2)
maxMismatches <- as.integer(0.2 * 1:9)
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet,
               max.Lmismatch = maxMismatches, max.Rmismatch = maxMismatches)

## Produce ranges that can be an input into other functions
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet,
               max.Lmismatch = 0, max.Rmismatch = 0, ranges = TRUE)
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subject,
               max.Lmismatch = 0.2, max.Rmismatch = 0.2, ranges = TRUE)
```
**xscat**

*Concatenate sequences contained in XString, XStringSet and/or XStringViews objects*

**Description**

This function mimics the semantic of `paste(..., sep="")` but accepts `XString`, `XStringSet` or `XStringViews` arguments and returns an `XString` or `XStringSet` object.

**Usage**

```r
xscat(...)  
```

**Arguments**

...  
One or more character vectors (with no NAs), `XString`, `XStringSet` or `XStringViews` objects.

**Value**

An `XString` object if all the arguments are either `XString` objects or character strings. An `XStringSet` object otherwise.

**Author(s)**

H. Pagès

**See Also**

`XString-class`, `XStringSet-class`, `XStringViews-class`, `paste`

**Examples**

```r
## Return a BString object:
xscat(BString("abc"), BString("EF"))
xscat(BString("abc"), "EF")
xscat("abc", "EF")

## Return a BStringSet object:
xscat(BStringSet("abc"), "EF")

## Return a DNAStringSet object:
xscat(c("t", "a"), DNAString("N"))

## Arguments are recycled to the length of the longest argument:
res1a <- xscat("x", LETTERS, c("3", "44", "555"))
res1b <- paste0("x", LETTERS, c("3", "44", "555"))
stopifnot(identical(as.character(res1a), as.character(res1b)))
```
## Concatenating big XStringSet objects:
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
mm <- complement(narrow(probes, start=13, end=13))
left <- narrow(probes, end=12)
right <- narrow(probes, start=14)
xscat(left, mm, right)

## Collapsing an XStringSet (or XStringViews) object with a small number of elements:
probes1000 <- as.list(probes[1:1000])
y1 <- do.call(xscat, probes1000)
y2 <- do.call(c, probes1000)  # slightly faster than the above
y1 == y2  # TRUE

## Note that this method won’t be efficient when the number of elements to collapse is big (> 10000) so we need to provide a collapse() (or xscollapse()) function in Biostrings that will be efficient at doing this. Please request this on the Bioconductor mailing list (http://bioconductor.org/help/mailing-list/) if you need it.

---

**XString-class**

**BString objects**

**Description**

The BString class is a general container for storing a big string (a long sequence of characters) and for making its manipulation easy and efficient.

The DNAString, RNAString and AAString classes are similar containers but with the more biology-oriented purpose of storing a DNA sequence (DNAString), an RNA sequence (RNAString), or a sequence of amino acids (AAString).

All those containers derive directly (and with no additional slots) from the XString virtual class.

**Details**

The 2 main differences between an XString object and a standard character vector are: (1) the data stored in an XString object are not copied on object duplication and (2) an XString object can only store a single string (see the XStringSet container for an efficient way to store a big collection of strings in a single object).

Unlike the DNAString, RNAString and AAString containers that accept only a predefined set of letters (the alphabet), a BString object can be used for storing any single string based on a single-byte character set.

**Constructor-like functions and generics**

In the code snippet below, \( x \) can be a single string (character vector of length 1) or an XString object.

\( \text{BString}(x='', \text{start}=1, \text{nchar}=\text{NA}) \): Tries to convert \( x \) into a BString object by reading \text{nchar} letters starting at position \text{start} in \( x \).
Accessor methods

In the code snippets below, x is an XString object.

alphabet(x): NULL for a BString object. See the corresponding man pages when x is a DNAString, RNAString or AString object.
leng(x): or nchar(x): Get the length of an XString object, i.e., its number of letters.

Coercion

In the code snippets below, x is an XString object.

as.character(x): Converts x to a character string.
toString(x): Equivalent to as.character(x).

Subsetting

In the code snippets below, x is an XString object.

x[i]: Return a new XString object made of the selected letters (subscript i must be an NA-free numeric vector specifying the positions of the letters to select). The returned object belongs to the same class as x.

Note that, unlike subseq, x[i] does copy the sequence data and therefore will be very inefficient for extracting a big number of letters (e.g. when i contains millions of positions).

Equality

In the code snippets below, e1 and e2 are XString objects.

e1 == e2: TRUE if e1 is equal to e2. FALSE otherwise.

Comparison between two XString objects of different base types (e.g. a BString object and a DNAString object) is not supported with one exception: a DNAString object and an RNAString object can be compared (see RNAString-class for more details about this).

Comparison between a BString object and a character string is also supported (see examples below).

e1 != e2: Equivalent to !(e1 == e2).

Author(s)

H. Pagès

See Also

subseq, letter, DNASTring-class, RNAString-class, AString-class, XStringSet-class, XStringViews-class, reverseComplement, compact, XVector-class
Examples

```r
b <- BString("I am a BString object")
b
length(b)

## Extracting a linear subsequence:
subseq(b)
subseq(b, start=3)
subseq(b, start=-3)
subseq(b, end=-3)
subseq(b, end=-3, width=5)

## Subsetting:
b2 <- b[length(b):1]  # better done with reverse(b)
as.character(b2)

b2 == b  # FALSE
b2 == as.character(b2)  # TRUE

## b[1:length(b)] is equal but not identical to b!
b == b[1:length(b)]  # TRUE
identical(b, 1:length(b))  # FALSE

## This is because subsetting an XString object with [ makes a copy
## of part or all its sequence data. Hence, for the resulting object,
## the internal slot containing the memory address of the sequence
## data differs from the original. This is enough for identical() to
## see the 2 objects as different.

## Compacting. As a particular type of XVector objects, XString
## objects can optionally be compacted. Compacting is done typically
## before serialization. See ?compact for more information.
```

---

**XStringPartialMatches-class**

**XStringPartialMatches objects**

### Description

WARNING: This class is currently under development and might not work properly! Full documentation will come later.

Please DO NOT TRY TO USE it for now. Thanks for your comprehension!

### Accessor methods

In the code snippets below, `x` is an XStringPartialMatches object.

- `subpatterns(x)`: Not ready yet.
- `pattern(x)`: Not ready yet.
**Standard generic methods**

In the code snippets below, \(x\) is an XStringPartialMatches objects, and \(i\) can be a numeric or logical vector.

\(x[i]\): Return a new XStringPartialMatches object made of the selected views. \(i\) can be a numeric vector, a logical vector, NULL or missing. The returned object has the same subject as \(x\).

**Author(s)**

H. Pagès

**See Also**

XStringViews-class, XString-class, letter

---

**Description**

Objects for storing string quality measures.

**Usage**

```r
## Constructors:
PhredQuality(x)
SolexaQuality(x)
IlluminaQuality(x)
```

```r
## alphabet and encoding
## S4 method for signature 'XStringQuality'
alphabet(x)
## S4 method for signature 'XStringQuality'
encoding(x)
```

**Arguments**

- \(x\)
  - Either a character vector, BString, BStringSet, integer vector, or number vector of error probabilities.

**Details**

PhredQuality objects store characters that are interpreted as \([0 - 99]\) quality measures by subtracting 33 from their ASCII decimal representation (e.g. ! = 0, " = 1, # = 2, ...). Quality measures \(q\) encode probabilities as \(-10 \times \log_{10}(p)\).

SolexaQuality objects store characters that are interpreted as \([-5 - 99]\) quality measures by subtracting 64 from their ASCII decimal representation (e.g. ; = -5, <= -4, = = -3, ...). Quality measures \(q\) encode probabilities as \(-10 \times (\log_{10}(p) - \log_{10}(1 - p))\).
IlluminaQuality objects store characters that are interpreted as \([0 - 99]\) quality measures by subtracting 64 from their ASCII decimal representation (e.g. @ = 0, A = 1, B = 2, ...). Quality measures \(q\) encode probabilities as \(-10 \times \log_{10}(p)\)

### Alphabet and encoding

In the code snippets below, \(x\) is an XStringQuality object.

- \(\text{alphabet}(x)\): Valid letters in this quality score; not all letters are encountered in actual sequencing runs.
- \(\text{encoding}(x)\): Map between letters and their corresponding integer encoding. Use \(\text{as.integer}\) and \(\text{as.numeric}\) to coerce objects to their integer and probability representations.

### Author(s)

P. Aboyoun

### See Also

- \(\text{pairwiseAlignment}\) and \(\text{PairwiseAlignments-class}\) in the \texttt{pwalign}\ package, \(\text{DNAString-class}, \text{BStringSet-class}\)

### Examples

```r
PhredQuality(0:40)
SolexaQuality(0:40)
IlluminaQuality(0:40)

pq <- PhredQuality(c("++-./", "0123456789;:"))
qs <- as(pq, "IntegerList") # quality scores
qs
as(qs, "PhredQuality")
p <- as(pq, "NumericList") # probabilities
as(p, "PhredQuality")

PhredQuality(seq(1e-4,0.5,length=10))
SolexaQuality(seq(1e-4,0.5,length=10))
IlluminaQuality(seq(1e-4,0.5,length=10))

x <- SolexaQuality(BStringSet(c(a="@ABC", b="abcd")))
as(x, "IntegerList") # quality scores
as(x, "NumericList") # probabilities
as.matrix(x) # quality scores
```
XStringSet-class  

XStringSet objects

Description

The BStringSet class is a container for storing a set of BString objects and for making its manipulation easy and efficient.

Similarly, the DNASTringSet (or RNASTringSet, or AASTringSet) class is a container for storing a set of DNASTring (or RNASTring, or AASTring) objects.

All those containers derive directly (and with no additional slots) from the XStringSet virtual class.

Usage

## Constructors:

BStringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)
DNASTringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)
RNASTringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)
AAStringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)

## Accessor-like methods:

## S4 method for signature 'character'

width(x)

## S4 method for signature 'XStringSet'

nchar(x, type="chars", allowNA=FALSE)

## ... and more (see below)

Arguments

x
Either a character vector (with no NAs), or an XString, XStringSet or XStringViews object.

start, end, width
Either NA, a single integer, or an integer vector of the same length as x specifying how x should be "narrowed" (see ?narrow for the details).

use.names
TRUE or FALSE. Should names be preserved?

Details

The BStringSet, DNASTringSet, RNASTringSet and AASTringSet functions are constructors that can be used to turn input x into an XStringSet object of the desired base type.

They also allow the user to "narrow" the sequences contained in x via proper use of the start, end and/or width arguments. In this context, "narrowing" means dropping a prefix or/and a suffix of each sequence in x. The "narrowing" capabilities of these constructors can be illustrated by the following property: if x is a character vector (with no NAs), or an XStringSet (or XStringViews) object, then the 3 following transformations are equivalent:
BStringSet(x, start=mystart, end=myend, width=mywidth):
subseq(BStringSet(x), start=mystart, end=myend, width=mywidth):
BStringSet(subseq(x, start=mystart, end=myend, width=mywidth)):

Note that, besides being more convenient, the first form is also more efficient on character vectors.

Accessor-like methods

In the code snippets below, x is an XStringSet object.

length(x): The number of sequences in x.

width(x): A vector of non-negative integers containing the number of letters for each element in x. Note that width(x) is also defined for a character vector with no NAs and is equivalent to nchar(x, type="bytes")

names(x): NULL or a character vector of the same length as x containing a short user-provided description or comment for each element in x. These are the only data in an XStringSet object that can safely be changed by the user. All the other data are immutable! As a general recommendation, the user should never try to modify an object by accessing its slots directly.

alphabet(x): Return NULL, DNA_ALPHABET, RNA_ALPHABET or AA_ALPHABET depending on whether x is a BStringSet, DNAStringSet, RNAStringSet or AAStringSet object.

nchar(x): The same as width(x).

Subsequence extraction and related transformations

In the code snippets below, x is a character vector (with no NAs), or an XStringSet (or XStringViews) object.

subseq(x, start=NA, end=NA, width=NA): Applies subseq on each element in x. See ?subseq for the details.

Note that this is similar to what substr does on a character vector. However there are some noticeable differences:
(1) the arguments are start and stop for substr;
(2) the SEW interface (start/end/width) interface of subseq is richer (e.g. support for negative start or end values); and (3) subseq checks that the specified start/end/width values are valid i.e., unlike substr, it throws an error if they define "out of limits" subsequences or subsequences with a negative width.

narrow(x, start=NA, end=NA, width=NA, use.names=TRUE): Same as subseq. The only differences are: (1) narrow has a use.names argument; and (2) all the things narrow and subseq work on (IRanges, XStringSet or XStringViews objects for narrow, XVector or XStringSet objects for subseq). But they both work and do the same thing on an XStringSet object.

threebands(x, start=NA, end=NA, width=NA): Like the method for IRanges objects, the threebands methods for character vectors and XStringSet objects extend the capability of narrow by returning the 3 set of subsequences (the left, middle and right subsequences) associated to the narrowing operation. See ?threebands in the IRanges package for the details.

subseq(x, start=NA, end=NA, width=NA) <- value: A vectorized version of the subseq<- method for XVector objects. See ? threebands for the details.
Subsetting and appending

In the code snippets below, \(x\) and \(values\) are XStringSet objects, and \(i\) should be an index specifying the elements to extract.

\(x[i]\): Return a new XStringSet object made of the selected elements.

\(x[[i]]\): Extract the \(i\)-th \textit{XString} object from \(x\).

\(\text{append}(x, values, after=\text{length}(x))\): Add sequences in \(values\) to \(x\).

Set operations

In the code snippets below, \(x\) and \(y\) are XStringSet objects.

\(\text{union}(x, y)\): Union of \(x\) and \(y\).

\(\text{intersect}(x, y)\): Intersection of \(x\) and \(y\).

\(\text{setdiff}(x, y)\): Asymmetric set difference of \(x\) and \(y\).

\(\text{setequal}(x, y)\): Set equality of \(x\) to \(y\).

Other methods

In the code snippets below, \(x\) is an XStringSet object.

\(\text{unlist}(x)\): Turns \(x\) into an \textit{XString} object by combining the sequences in \(x\) together. Fast equivalent to \(\text{do.call(c, as.list(x))}\).

\(\text{as.character}(x, \text{use.names=TRUE})\): Converts \(x\) to a character vector of the same length as \(x\).

The \text{use.names} argument controls whether or not \(\text{names}(x)\) should be propagated to the names of the returned vector.

\(\text{as.factor}(x)\): Converts \(x\) to a factor, via \(\text{as.character}(x)\).

\(\text{as.matrix}(x, \text{use.names=TRUE})\): Returns a character matrix containing the "exploded" representation of the strings. Can only be used on an XStringSet object with equal-width strings.

The \text{use.names} argument controls whether or not \(\text{names}(x)\) should be propagated to the row names of the returned matrix.

\(\text{toString}(x)\): Equivalent to \(\text{toString(as.character(x))}\).

\(\text{show}(x)\): By default the \text{show} method displays 5 head and 5 tail lines. The number of lines can be altered by setting the global options \text{showHeadLines} and \text{showTailLines}. If the object length is less than the sum of the options, the full object is displayed. These options affect GRanges, GAlignments, IRanges, and XStringSet objects.

Display

The letters in a DNAStringSet, RNAStringSet, or AAStringSet object are colored when displayed by the \text{show()} method. Set global option \text{Biostrings.coloring} to \text{FALSE} to turn off this coloring.

Author(s)

H. Pagès
See Also

- `readDNAStringSet` and `writeXStringSet` for reading/writing a `DNAStringSet` object (or other `XStringSet` derivative) from/to a FASTA or FASTQ file.
- `XStringSet-comparison`
- `XString` objects.
- `XStringViews` objects.
- `XStringSetList` objects.
- `subseq`, `narrow`, and `substr`.
- `compact`
- `XVectorList` objects.

Examples

```r
### A. USING THE XStringSet CONSTRUCTORS ON A CHARACTER VECTOR OR FACTOR
### Note that there is no XStringSet() constructor, but an XStringSet
### family of constructors: BStringSet(), DNAStringSet(), RNAStringSet(),
### etc...
xx <- c("#CTC-NACCAGTAT", "#TTGA", "TACCTAGAG")
width(xx)
x1 <- BStringSet(xx)
x1
## 3 equivalent ways to obtain the same BStringSet object:
BStringSet(xx, start=4, end=-3)
subseq(x1, start=4, end=-3)
BStringSet(subseq(xx, start=4, end=-3))

dna0 <- DNAStringSet(xx, start=4, end=-3)
dna0
# 'options(Biostrings.coloring=FALSE)' to turn off coloring

names(dna0)
names(dna0)[2] <- "seqB"
dna0

## When the input vector contains a lot of duplicates, turning it into
## a factor first before passing it to the constructor will produce an
## XStringSet object that is more compact in memory:
library(hgu95av2probe)
x2 <- sample(hgu95av2probe$sequence, 999000, replace=TRUE)
dna2a <- DNAStringSet(x2)
dna2b <- DNAStringSet(factor(x2))  # slower but result is more compact
object.size(dna2a)
object.size(dna2b)
```

### B. USING THE XStringSet CONSTRUCTORS ON A SINGLE SEQUENCE (XString
### OBJECT OR CHARACTER STRING)
## C. USING THE XStringSet CONSTRUCTORS ON AN XStringSet OBJECT

### library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)

probes

RNAStringSet(probes, start=2, end=-5)  # does NOT copy the sequence data!

### D. USING THE XStringSet CONSTRUCTORS ON AN ORDINARY list OF XString OBJECTS

```r
## Randomly extract 1 million 40-mers from C. elegans chrI:
extractRandomReads <- function(subject, nread, readlength)
{
  if (!is.integer(readlength))
    readlength <- as.integer(readlength)
  start <- sample(length(subject) - readlength + 1L, nread, replace=TRUE)
  DNAStringSet(subject, start=start, width=readlength)
}
```

```r
library(BSgenome.Celegans.UCSC.ce2)
rndreads <- extractRandomReads(Celegans$chrI, 1000000, 40)
```

## Notes:
- This takes only 2 or 3 seconds versus several hours for a solution using `substring()` on a standard character string.
- The short sequences in `rndreads` can be seen as the result of a simulated high-throughput sequencing experiment. A non-realistic one though because:
  (a) It assumes that the underlying technology is perfect (the generated reads have no technology induced errors).
  (b) It assumes that the sequenced genome is exactly the same as the reference genome.
  (c) The simulated reads can contain IUPAC ambiguity letters only because the reference genome contains them. In a real high-throughput sequencing experiment, the sequenced genome of course doesn't contain those letters, but the sequencer can introduce them in the generated reads to indicate ambiguous base-calling.
  (d) The simulated reads come from the plus strand only of a single chromosome.
- See the `getSeq()` function in the BSgenome package for how to circumvent (d) i.e. how to generate reads that come from the whole genome (plus and minus strands of all chromosomes).
probes10 <- head(probes, n=10)
set.seed(33)
shuffled_nucleotides <- lapply(probes10, sample)
shuffled_nucleotides

DNAStringSet(shuffled_nucleotides) # does NOT copy the sequence data!

## Note that the same result can be obtained in a more compact way with
## just:
set.seed(33)
endoapply(probes10, sample)

# ---------------------------------------------------------------------
# E. USING subseq() ON AN XStringSet OBJECT
# ---------------------------------------------------------------------
subseq(probes, start=2, end=-5)
subseq(probes, start=13, end=13) <- "N"
probes

## Add/remove a prefix:
subseq(probes, start=1, end=0) <- "--"
probes
subseq(probes, end=2) <- ""
probes

## Do more complicated things:
subseq(probes, start=4:7, end=7) <- c("YYYY", "YYY", "YY", "Y")
subseq(probes, start=4, end=6) <- subseq(probes, start=2:-5)
probes

# ---------------------------------------------------------------------
# F. UNLISTING AN XStringSet OBJECT
# ---------------------------------------------------------------------
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
unlist(probes)

# ---------------------------------------------------------------------
# G. COMPACTING AN XStringSet OBJECT
# ---------------------------------------------------------------------
## As a particular type of XVectorList objects, XStringSet objects can
## optionally be compacted. Compacting is done typically before
## serialization. See ?compact for more information.
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
y <- subseq(probes[1:12], start=5)
probes@pool
y@pool
object.size(probes)
object.size(y)
Description

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

Details

Element-wise (aka "parallel") comparison of 2 XStringSet objects is based on the lexicographic order between 2 BString, DNAString, RNAString, or AAString objects.

For DNAStringSet and RNAStringSet objects, the letters in the respective alphabets (i.e. DNA_ALPHABET and RNA_ALPHABET) are ordered based on a predefined code assigned to each letter. The code assigned to each letter can be retrieved with:

```r
dna_codes <- as.integer(DNAString(paste(DNA_ALPHABET, collapse="")))
names(dna_codes) <- DNA_ALPHABET

rna_codes <- as.integer(RNAString(paste(RNA_ALPHABET, collapse="")))
names(rna_codes) <- RNA_ALPHABET
```

Note that this order does NOT depend on the locale in use. Also note that comparing DNA sequences with RNA sequences is supported and in that case T and U are considered to be the same letter.

For BStringSet and AAStringSet objects, the alphabetical order is defined by the C collation. Note that, at the moment, AAStringSet objects are treated like BStringSet objects i.e. the alphabetical order is NOT defined by the order of the letters in AA_ALPHABET. This might change at some point.

pcompare() and related methods

In the code snippets below, x and y are XStringSet objects.

x == y, x != y, x <= y, x >= y, x < y, x > y: Equivalent to pcompare(x, y) == 0, pcompare(x, y) != 0, pcompare(x, y) <= 0, pcompare(x, y) >= 0, pcompare(x, y) < 0, and pcompare(x, y) > 0, respectively.
**XStringSet-comparison**

**order() and related methods**

In the code snippets below, x is an XStringSet object.

- `is.unsorted(x, strictly=FALSE)`: Return a logical values specifying if x is unsorted. The `strictly` argument takes logical value indicating if the check should be for _strictly_ increasing values.
- `order(x, decreasing=FALSE)`: Return a permutation which rearranges x into ascending or descending order.
- `rank(x, ties.method=c("first", "min"))`: Rank x in ascending order.
- `sort(x, decreasing=FALSE)`: Sort x into ascending or descending order.

**duplicated() and unique()**

In the code snippets below, x is an XStringSet object.

- `duplicated(x)`: Return a logical vector whose elements denotes duplicates in x.
- `unique(x)`: Return the subset of x made of its unique elements.

**match() and %in%**

In the code snippets below, x and table are XStringSet objects.

- `match(x, table, nomatch=NA_integer_)`: Returns an integer vector containing the first positions of an identical match in table for the elements in x.
- `x %in% table`: Returns a logical vector indicating which elements in x match identically with an element in table.

**is.na() and related methods**

In the code snippets below, x is an XStringSet object. An XStringSet object never contains missing values (these methods exist for compatibility).

- `is.na(x)`: Returns FALSE for every element.
- `anyNA(x)`: Returns FALSE.

**Author(s)**

H. Pagès

**See Also**

XStringSet-class, ==, is.unsorted, order, rank, sort, duplicated, unique, match, %in%
Examples

```r
# A. SIMPLE EXAMPLES

dna <- DNAStringSet(c("AAA", "TC", "", "TC", "AAA", "CAAC", "G"))
match(c("", "G", "AA", "TC"), dna)

library(drosophila2probe)
fly_probes <- DNAStringSet(drosophila2probe)
sum(duplicated(fly_probes))  # 481 duplicated probes

is.unsorted(fly_probes)  # TRUE
fly_probes <- sort(fly_probes)
is.unsorted(fly_probes)  # FALSE
is.unsorted(fly_probes, strictly=TRUE)  # TRUE, because of duplicates
is.unsorted(unique(fly_probes), strictly=TRUE)  # FALSE

# Nb of probes that are the reverse complement of another probe:
nb1 <- sum(reverseComplement(fly_probes) %in% fly_probes)
stopifnot(identical(nb1, 455L))  # 455 probes

# Probes shared between drosophila2probe and hgu95av2probe:
library(hgu95av2probe)
human_probes <- DNAStringSet(hgu95av2probe)
m <- match(fly_probes, human_probes)
stopifnot(identical(sum(!is.na(m)), 493L))  # 493 shared probes
```

## B. AN ADVANCED EXAMPLE

```r
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)

first5 <- narrow(probes, end=5)
last5 <- narrow(probes, start=-5)
b1 <- sum(first5 == reverseComplement(last5))
stopifnot(identical(b1, 17L))

# Percentage:
100 * nb2 / length(probes)  # 0.0064 %

# If the probes were random DNA sequences, a probe would have 1 chance
# out of 4^5 to have this property so the percentage would be:
100 / 4^5  # 0.008 %

# With randomly generated probes:
```
```r
set.seed(33)
random_dna <- sample(DNAString(paste(DNA_BASES, collapse="")),
sum(width(probes)), replace=TRUE)
random_probes <- successiveViews(random_dna, width(probes))
random_probes
random_probes <- as(random_probes, "XStringSet")
random_probes
random_first5 <- narrow(random_probes, end=5)
random_last5 <- narrow(random_probes, start=-5)

nb3 <- sum(random_first5 == reverseComplement(random_last5))
100 * nb3 / length(random_probes) # 0.099 %
```

---

**XStringSet-io**

*Read/write an XStringSet object from/to a file*

### Description

Functions to read/write an XStringSet object from/to a file.

### Usage

```r
## Read FASTA (or FASTQ) files in an XStringSet object:
readBStringSet(filepath, format="fasta",
nrec=-1L, skip=0L, seek.first.rec=FALSE,
use.names=TRUE, with.qualities=FALSE)
readDNAStringSet(filepath, format="fasta",
nrec=-1L, skip=0L, seek.first.rec=FALSE,
use.names=TRUE, with.qualities=FALSE)
readRNAStringSet(filepath, format="fasta",
nrec=-1L, skip=0L, seek.first.rec=FALSE,
use.names=TRUE, with.qualities=FALSE)
readAAStringSet(filepath, format="fasta",
nrec=-1L, skip=0L, seek.first.rec=FALSE,
use.names=TRUE, with.qualities=FALSE)

## Extract basic information about FASTA (or FASTQ) files
## without actually loading the sequence data:
fasta.seqlengths(filepath,
nrec=-1L, skip=0L, seek.first.rec=FALSE,
seqtype="B", use.names=TRUE)
fasta.index(filepath,
nrec=-1L, skip=0L, seek.first.rec=FALSE,
seqtype="B")

fastq.seqlengths(filepath,
nrec=-1L, skip=0L, seek.first.rec=FALSE)
```
fastq.geometry(filepath,
    nrec=-1L, skip=0L, seek.first.rec=FALSE)

## Write an XStringSet object to a FASTA (or FASTQ) file:
writeXStringSet(x, filepath, append=FALSE,
    compress=FALSE, compression_level=NA, format="fasta", ...)

## Serialize an XStringSet object:
saveXStringSet(x, objname, dirpath=".", save.dups=FALSE, verbose=TRUE)

### Arguments

**filepath**
A character vector (of arbitrary length when reading, of length 1 when writing) containing the path(s) to the file(s) to read or write. Reading files in gzip format (which usually have the `.gz` extension) is supported. Note that special values like `""` or `"|cmd"` (typically supported by other I/O functions in R) are not supported here. Also `filepath` cannot be a standard connection. However `filepath` can be an object as returned by `open_input_files`. This object can be used to read files by chunks. See "READ FILES BY CHUNK" in the examples section for the details.

**format**
Either "fasta" (the default) or "fastq".

**nrec**
Single integer. The maximum of number of records to read in. Negative values are ignored.

**skip**
Single non-negative integer. The number of records of the data file(s) to skip before beginning to read in records.

**seek.first.rec**
TRUE or FALSE (the default). If TRUE, then the reading function starts by setting the file position indicator at the beginning of the first line in the file that looks like the beginning of a FASTA (if `format` is "fasta") or FASTQ (if `format` is "fastq") record. More precisely this is the first line in the file that starts with a ‘>’ (for FASTA) or a ‘@’ (for FASTQ). An error is raised if no such line is found. Normal parsing then starts from there, and everything happens like if the file actually started there. In particular it will be an error if this first record is not a valid FASTA or FASTQ record. Using `seek.first.rec=TRUE` is useful for example to parse GFF3 files with embedded FASTA data.

**use.names**
TRUE (the default) or FALSE. If TRUE, then the returned vector is named. For FASTA the names are taken from the record description lines. For FASTQ they are taken from the record sequence ids. Dropping the names with `use.names=FALSE` can help reduce memory footprint e.g. for a FASTQ file containing millions of reads.

**with.qualities**
TRUE or FALSE (the default). This argument is only supported when reading a FASTQ file. If TRUE, then the quality strings are also read and returned in the qualities metadata column of the returned DNAStringSet object. Note that by default the quality strings are ignored. This helps reduce memory footprint if the FASTQ file contains millions of reads.
Note that using `readQualityScaledDNAStringSet()` is the preferred way to load a set of DNA sequences and their qualities from a FASTQ file into Bioconductor. Its main advantage is that it will return a `QualityScaledDNAStringSet` object instead of a `DNAStringSet` object, which makes handling of the qualities more convenient and less error prone. See "READ A FASTQ FILE AS A QualityScaledDNAStringSet OBJECT" in the Examples section below for more information.

**seqtype**

A single string specifying the type of sequences contained in the FASTA file(s). Supported sequence types:

- "B" for anything i.e. any letter is a valid one-letter sequence code.
- "DNA" for DNA sequences i.e. only letters in `DNA_ALPHABET` (case ignored) are valid one-letter sequence codes.
- "RNA" for RNA sequences i.e. only letters in `RNA_ALPHABET` (case ignored) are valid one-letter sequence codes.
- "AA" for Amino Acid sequences i.e. only letters in `AA_ALPHABET` (case ignored) are valid one-letter sequence codes.

Invalid one-letter sequence codes are ignored with a warning.

**x**

For `writeXStringSet`, the object to write to file.

For `saveXStringSet`, the object to serialize.

**append**

`TRUE` or `FALSE`. If `TRUE` output will be appended to file; otherwise, it will overwrite the contents of file. See `?cat` for the details.

**compress**

Like for the `save` function in base R, must be `TRUE` or `FALSE` (the default), or a single string specifying whether writing to the file is to use compression. The only type of compression supported at the moment is "gzip".

Passing `TRUE` is equivalent to passing "gzip".

**compression_level**

Not implemented yet.

... Further format-specific arguments.

If `format="fasta"`, the `width` argument can be used to specify the maximum number of letters per line of sequence. `width` must be a single integer.

If `format="fastq"`, the `qualities` argument can be used to specify the quality strings. `qualities` must be a `BStringSet` object. If the argument is omitted, then the quality strings are taken from the qualities metadata column of `x` (i.e. from `mcols(x)$qualities`). If `x` has no qualities metadata column and the qualities argument is omitted, then the fake quality ‘;’ is assigned to each letter in `x` and written to the FASTQ file. If `x` is a `QualityScaledXStringSet` and qualities is not defined, the qualities contained in `x` are used automatically.

**objname**

The name of the serialized object.

**dirpath**

The path to the directory where to save the serialized object.

**save.dups**

`TRUE` or `FALSE`. If `TRUE` then the `Dups` object describing how duplicated elements in `x` are related to each other is saved too. For advanced users only.

**verbose**

`TRUE` or `FALSE`. 
Details

gzip compression is supported by reading and writing functions on all platforms. readDNAStringSet and family (i.e. readBStringSet, readDNAStringSet, readRNAStringSet and readAAStringSet) load sequences from an input file (or multiple input files) into an XStringSet object. When multiple input files are specified, all must have the same format (i.e. FASTA or FASTQ) and files with different compression types can be mixed with non-compressed files. The files are read in the order they were specified and the sequences are stored in the returned object in the order they were read.

Only FASTA and FASTQ files are supported for now.

The fasta.seqlengths utility returns an integer vector with one element per FASTA record in the input files. Each element is the length of the sequence found in the corresponding record, that is, the number of valid one-letter sequence codes in the record. See description of the seqtype argument above for how to control the set of valid one-letter sequence codes.

The fasta.index utility returns a data frame with 1 row per FASTA record in the input files and the following columns:

- recno: The rank of the record in the (virtually) concatenated input files.
- fileno: The rank of the file where the record is located.
- offset: The offset of the record relative to the start of the file where it’s located. Measured in bytes.
- desc: The description line (a.k.a. header) of the record.
- seqlength: The length of the sequence in the record (not counting invalid letters).
- filepath: The path to the file where the record is located. Always a local file, so if the user specified a remote file, this column will contain the path to the downloaded file.

A subset of this data frame can be passed to readDNAStringSet and family for direct access to an arbitrary subset of sequences. More precisely, if fai is a FASTA index that was obtained with fasta.index(filepath, ..., seqtype="DNA"), then readDNAStringSet(fai[i, ]) is equivalent to readDNAStringSet(filepath, ...)[[i]] for any valid subscript i, except that the former only loads the requested sequences in memory and thus will be more memory efficient if only a small subset of sequences is requested.

The fastq.seqlengths utility returns the read lengths in an integer vector with one element per FASTQ record in the input files.

The fastq.geometry utility is a convenience wrapper around fastq.seqlengths that returns an integer vector of length 2 describing the geometry of the FASTQ files. The first integer gives the total number of FASTQ records in the files and the second element the common length of the reads (this common length is set to NA in case of variable length reads or if no FASTQ record was found). This compact representation of the geometry can be useful if the FASTQ files are known to contain fixed length reads.

writeXStringSet writes an XStringSet object to a file. Like with readDNAStringSet and family, only FASTA and FASTQ files are supported for now. WARNING: Please be aware that using writeXStringSet on a BStringSet object that contains the ‘\n’ (LF) or ‘\r’ (CR) characters or the FASTA markup characters ’>’ or ’:’ is almost guaranteed to produce a broken FASTA file!

Serializable an XStringSet object with saveXStringSet is equivalent to using the standard save mechanism. But it will try to reduce the size of x in memory first before calling save. Most of the times this leads to a much reduced size on disk.
References

http://en.wikipedia.org/wiki/FASTA_format

See Also

- BStringSet, DNAStringSet, RNAStringSet, and AAStringSet objects.
- readQualityScaledDNAStringSet and writeQualityScaledXStringSet for reading/writing a QualityScaledDNAStringSet object (or other QualityScaledXStringSet derivative) from/to a FASTQ file.

Examples

```r
## A. READ/ WRITE FASTA FILES

## Read a non-compressed FASTA files:
filepath1 <- system.file("extdata", "someORF.fa", package="Biostrings")
fasta.seqlengths(filepath1, seqtype="DNA")
x1 <- readDNAStringSet(filepath1)

## Read a gzip-compressed FASTA file:
filepath2 <- system.file("extdata", "someORF.fa.gz", package="Biostrings")
fasta.seqlengths(filepath2, seqtype="DNA")
x2 <- readDNAStringSet(filepath2)

## Sanity check:
stopifnot(identical(as.character(x1), as.character(x2)))

## Read 2 FASTA files at once:
filepath3 <- system.file("extdata", "fastaEx.fa", package="Biostrings")
fasta.seqlengths(c(filepath2, filepath3), seqtype="DNA")
x23 <- readDNAStringSet(c(filepath2, filepath3))

## Sanity check:
x3 <- readDNAStringSet(filepath3)
stopifnot(identical(as.character(x23), as.character(c(x2, x3))))

## Use a FASTA index to load only an arbitrary subset of sequences:
filepath4 <- system.file("extdata", "dm3_upstream2000.fa.gz", package="Biostrings")
fai <- fasta.index(filepath4, seqtype="DNA")
head(fai)
head(fai$desc)
i <- sample(nrow(fai), 10)  # randomly pick up 10 sequences
x4 <- readDNAStringSet(fai[i, ])

## Sanity check:
```

## B. READ/ WRITE FASTQ FILES

## Read a FASTQ file:
filepath5 <- system.file("extdata", "someORF.fq", package="Biostrings")
readFastq(filepath5, seqtype="DNA")

## Write a FASTQ file:
writeFastq(x1, filepath6, seqtype="DNA")

## Example of a valid FASTQ file:

```r
@HINT1
GAGTGCTGCTATG
+
####
AAGATAGATAGAT
```

```r
@HINT2
CGTCGACGCGCG
+
####
GCGCGCGCGCG
```

## Web Resources

```
stopifnot(identical(as.character(readDNAStringSet(filepath4)[i]),
            as.character(x4)))

## Write FASTA files:
out23a <- tempfile()
writeXStringSet(x23, out23a)
out23b <- tempfile()
writeXStringSet(x23, out23b, compress=TRUE)
file.info(c(out23a, out23b))$size

## Sanity checks:
stopifnot(identical(as.character(readDNAStringSet(out23a)),
            as.character(x23)))
stopifnot(identical(readLines(out23a), readLines(out23b)))

## B. READ/WRITE FASTQ FILES
## ---------------------------------------------------------------

filepath5 <- system.file("extdata", "s_1_sequence.txt",  
package="Biostrings")

fastq.geometry(filepath5)

## The quality strings are ignored by default:
reads <- readDNAStringSet(filepath5, format="fastq")
reads
mcols(reads)

## Use 'with.qualities=TRUE' to load them:
reads <- readDNAStringSet(filepath5, format="fastq", with.qualities=TRUE)
reads
mcols(reads)
mcols(reads)$qualities

## Each quality string contains one letter per nucleotide in the  
## corresponding read:
stopifnot(identical(width(mcols(reads)$qualities), width(reads)))

## Write the reads to a FASTQ file:
outfile <- tempfile()
writeXStringSet(reads, outfile, format="fastq")
outfile2 <- tempfile()
writeXStringSet(reads, outfile2, compress=TRUE, format="fastq")

## Sanity checks:
stopifnot(identical(readLines(outfile), readLines(filepath5)))
stopifnot(identical(readLines(outfile), readLines(outfile2)))

## C. READ FILES BY CHUNK
## ---------------------------------------------------------------

## readDNAStringSet() supports reading an arbitrary number of FASTA or
## FASTQ records at a time in a loop. This can be useful to process big FASTA or FASTQ files by chunk and thus avoids loading the entire file in memory. To achieve this the files to read from need to be opened with open_input_files() first. Note that open_input_files() accepts a vector of file paths and/or URLs.

### With FASTA files:
```
files <- open_input_files(filepath4)
i <- 0
while (TRUE) {
  i <- i + 1
  ## Load 4000 records at a time. Each new call to readDNAStringSet() picks up where the previous call left.
  dna <- readDNAStringSet(files, nrec=4000)
  if (length(dna) == 0L)
    break
  cat("processing chunk", i, "...
"
  ## do something with `dna`
}
```

### With FASTQ files:
```
files <- open_input_files(filepath5)
i <- 0
while (TRUE) {
  i <- i + 1
  ## Load 75 records at a time.
  reads <- readDNAStringSet(files, format="fastq", nrec=75)
  if (length(reads) == 0L)
    break
  cat("processing chunk", i, "...
"
  ## do something with `reads`
}
```

### IMPORTANT NOTE: Like connections, the object returned by open_input_files() can NOT be shared across workers in the context of parallelization!

## D. READ A FASTQ FILE AS A QualityScaledDNAStringSet OBJECT

Use readQualityScaledDNAStringSet() if you want the object to be returned as a QualityScaledDNAStringSet instead of a DNAStringSet object. See ?readQualityScaledDNAStringSet for more information.

### Note that readQualityScaledDNAStringSet() is a simple wrapper around readDNAStringSet() that does the following if the file contains "Phred quality scores" (which is the standard Sanger variant to assess reliability of a base call):
```
reads <- readDNAStringSet(filepath5, format="fastq", with.qualities=TRUE)
quals <- PhredQuality(mcols(reads)$qualities)
QualityScaledDNAStringSet(reads, quals)
```
## E. GENERATE FAKE READS AND WRITE THEM TO A FASTQ FILE

```r
library(BSgenome.Celegans.UCSC.ce2)

## Create a "sliding window" on chr I:
sw_start <- seq.int(1, length(Celegans$chrI)-50, by=50)
sw <- Views(Celegans$chrI, start=sw_start, width=10)
my_fake_reads <- as(sw, "XStringSet")
my_fake_ids <- sprintf("ID%06d", seq_len(length(my_fake_reads)))
names(my_fake_reads) <- my_fake_ids
my_fake_reads

## Fake quality ';'; will be assigned to each base in "my_fake_reads":
out2 <- tempfile()
writeXStringSet(my_fake_reads, out2, format="fastq")

## Passing qualities thru the 'qualities' argument:
my_fake_quals <- rep.int(BStringSet("DCBA@?>=<;"), length(my_fake_reads))
my_fake_quals
out3 <- tempfile()
writeXStringSet(my_fake_reads, out3, format="fastq",
qualities=my_fake_quals)
```

## F. SERIALIZATION

```r
saveXStringSet(my_fake_reads, "my_fake_reads", dirpath=tempdir())
```

### XStringSetList-class

**XStringSetList objects**

**Description**

The XStringSetList class is a virtual container for storing a list of XStringSet objects.

**Usage**

```r
## Constructors:
BStringSetList(..., use.names=TRUE)
DNAStringSetList(..., use.names=TRUE)
RNAStringSetList(..., use.names=TRUE)
AAStringSetList(..., use.names=TRUE)
```
Arguments

... Character vector(s) (with no NAs), or XStringSet object(s), or XStringViews object(s) to be concatenated into a XStringSetList.

use.names TRUE or FALSE. Should names be preserved?

Details

Concrete flavors of the XStringSetList container are the BStringSetList, DNAStringSetList, RNAStringSetList and AAStringSetList containers for storing a list of BStringSet, DNAStringSet, RNAStringSet and AAStringSet objects, respectively. These four containers are direct subclasses of XStringSetList with no additional slots.

Currently DNAStringSetList() and AAStringSetList() are the only XStringSetList constructors. The XStringSetList class itself is virtual and has no constructor.

Methods

The XStringSetList class extends the List class defined in the IRanges package. Using a less technical jargon, this just means that an XStringSetList object is a list-like object that can be manipulated like an ordinary list. Or, said otherwise, most of the operations that work on an ordinary list (e.g. length, names, [], [[], ., c, unlist, etc...) should work on an XStringSetList object. In addition, Bioconductor specific list operations like elementNROWS and PartitioningByEnd (defined in the IRanges package) are supported too.

Author(s)

H. Pagès

See Also

XStringSet-class, List-class

Examples

```r
# A. THE XStringSetList CONSTRUCTORS
#
# Currently DNAStringSetList() and AAStringSetList() are the only
# constructors. Others will be developed when the use case arises.

dna1 <- c("AAA", "AC", ",", "T", "GGATA")
dna2 <- c("G", "TT", "C")

x <- DNAStringSetList(dna1, dna2)
x

DNAStringSetList(DNAStringSet(dna1), DNAStringSet(dna2))
DNAStringSetList(dna1, DNAStringSet(dna2))
DNAStringSetList(DNAStringSet(dna1), dna2)
```
DNAStringSetList(dna1, RNAStringSet(DNAStringSet(dna2)))
DNAStringSetList(list(dna1, dna2))
DNAStringSetList(CharacterList(dna1, dna2))

## Empty object (i.e. zero-length):
DNAStringSetList()

## Not empty (length is 1):
DNAStringSetList(character(0))

## B. UNLISTING AN XStringSetList OBJECT
length(x)
elementNROWS(x)
unlist(x)
x[[1]]
x[[2]]
as.list(x)

names(x) <- LETTERS[1:length(x)]
x[["A"]]
x[["B"]]
as.list(x) # named list

---

XStringViews-class The XStringViews class

Description

The XStringViews class is the basic container for storing a set of views (start/end locations) on the same sequence (an XString object).

Details

An XStringViews object contains a set of views (start/end locations) on the same XString object called "the subject string" or "the subject sequence" or simply "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. An XStringViews object is in fact a particular case of an Views object (the XStringViews class contains the Views class) so it can be manipulated in a similar manner: see ?Views for more information. Note that two views can overlap and that a view can be "out of limits" i.e. it can start before the first letter of the subject or/and end after its last letter.

Constructor

Views(subject, start=NULL, end=NULL, width=NULL, names=NULL): See ?Views in the IRanges package for the details.
**Accessor-like methods**

All the accessor-like methods defined for Views objects work on XStringViews objects. In addition, the following accessors are defined for XStringViews objects:

- `nchar(x)`: A vector of non-negative integers containing the number of letters in each view. Values in `nchar(x)` coincide with values in `width(x)` except for "out of limits" views where they are lower.

**Other methods**

In the code snippets below, `x`, `object`, `e1` and `e2` are XStringViews objects, and `i` can be a numeric or logical vector.

- `e1 == e2`: A vector of logicals indicating the result of the view by view comparison. The views in the shorter of the two XStringViews object being compared are recycled as necessary. Like for comparison between XString objects, comparison between two XStringViews objects with subjects of different classes is not supported with one exception: when the subjects are DNAString and RNAString instances.

  Also, like with XString objects, comparison between an XStringViews object with a BString subject and a character vector is supported (see examples below).

- `e1 != e2`: Equivalent to `!(e1 == e2)`.

- `as.character(x, use.names=TRUE, check.limits=TRUE)`: Converts `x` to a character vector of the same length as `x`. The use.names argument controls whether or not names(`x`) should be propagated to the names of the returned vector. The check.limits argument controls whether or not an error should be raised if `x` has "out of limit" views. If check.limits is FALSE then "out of limit" views are trimmed with a warning.

- `as.data.frame(x, row.names = NULL, optional = FALSE, ...)`: Equivalent of `as.data.frame(as.character(x))`.

- `as.matrix(x, use.names=TRUE)`: Returns a character matrix containing the "exploded" representation of the views. Can only be used on an XStringViews object with equal-width views. The use.names argument controls whether or not names(`x`) should be propagated to the row names of the returned matrix.

- `toString(x)`: Equivalent to `toString(as.character(x))`.

**Author(s)**

H. Pagès

**See Also**

Views-class, gaps, XString-class, XStringSet-class, letter, MIndex-class

**Examples**

```r
## One standard way to create an XStringViews object is to use
## the Views() constructor.

## Views on a DNAString object:
s <- DNAString("-CTC-N")
```
v4 <- Views(s, start=3:0, end=5:8)
v4
subject(v4)
length(v4)
start(v4)
end(v4)
width(v4)

## Attach a comment to views #3 and #4:
names(v4)[3:4] <- "out of limits"
names(v4)

## A more programatical way to "tag" the "out of limits" views:
names(v4)[start(v4) < 1 | nchar(subject(v4)) < end(v4)] <- "out of limits"
## or just:
names(v4)[nchar(v4) < width(v4)] <- "out of limits"

## Two equivalent ways to extract a view as an XString object:
s2a <- v4[[2]]
s2b <- subseq(subject(v4), start=start(v4)[2], end=end(v4)[2])
identical(s2a, s2b) # TRUE

## It is an error to try to extract an "out of limits" view:
#v4[[3]] # Error!

v12 <- Views(DNAString("TAATAATG"), start=-2:9, end=0:11)
v12 == DNAString("TAA")
v12[v12 == v12[4]]
v12[v12 == v12[[1]]]
v12[3] == Views(RNAString("AU"), start=0, end=2)

## Here the first view doesn't even overlap with the subject:
Views(BString("aaa--b"), start=-3:4, end=-3:4 + c(3:6, 6:3))

## 'start' and 'end' are recycled:
subject <- "abcdefghij"
Views(subject, start=2:1, end=4)
Views(subject, start=5:7, end=nchar(subject))
Views(subject, start=1, end=5:7)

## Applying gaps() to an XStringViews object:
v2 <- Views("abcDefgHIJK", start=c(8, 3), end=c(14, 4))
gaps(v2)

## Coercion:
as(v12, "XStringSet") # same as 'as(v12, "DNAStringSet")'
rna <- as(v12, "RNAStringSet")
as(rna, "Views")

---

yeastSEQCHR1  
An annotation data file for CHR1 in the yeastSEQ package
Description

This is a single character string containing DNA sequence of yeast chromosome number 1. The data were obtained from the Saccharomyces Genome Database (ftp://genome-ftp.stanford.edu/pub/yeast/data\_download/sequence/genomic\_sequence/chromosomes/fasta/).

Details

Annotation based on data provided by Yeast Genome project.

Source data built: Yeast Genome data are built at various time intervals. Sources used were downloaded Fri Nov 21 14:00:47 2003 Package built: Fri Nov 21 14:00:47 2003

References

http://www.yeastgenome.org/DownloadContents.shtml

Examples

data(yeastSEQCHR1)
nchar(yeastSEQCHR1)
Index

!=, BString, character-method
   (XString-class), 108
!=, XString, XString-method
   (XString-class), 108
!=, XString, XStringViews-method
   (XStringViews-class), 131
!=, XStringViews, XString-method
   (XStringViews-class), 131
!=, XStringViews, XStringViews-method
   (XStringViews-class), 131
!=, character, BString-method
   (XString-class), 108
!=, character, XStringViews-method
   (XStringViews-class), 131

* classes
   AAString-class, 4
   DNAString-class, 10
   MaskedXString-class, 35
   MIndex-class, 66
   MultipleAlignment-class, 69
   PDict-class, 82
   QualityScaledXStringSet-class, 86
   RNAString-class, 97
   XString-class, 108
   XStringPartialMatches-class, 110
   XStringQuality-class, 111
   XStringSet-class, 113
   XStringSetList-class, 129
   XStringViews-class, 131

* datasets
   HNF4alpha, 19
   yeastSEQCHR1, 133

* data
   AMINO_ACID_CODE, 5
   GENETIC_CODE, 14
   IUPAC_CODE_MAP, 21

* distribution
   dinucleotideFrequencyTest, 8
* htest
   dinucleotideFrequencyTest, 8
* internal
   Biostrings internals, 6
* manip
   chartr, 6
   detail, 8
   getSeq, 17
   gregexpr2, 18
   injectHardMask, 19
   letterFrequency, 23
   longestConsecutive, 29
   maskMotif, 37
   matchprobes, 62
   matchPWM, 63
   misc, 67
   nucleotideFrequency, 75
   padAndClip, 80
   replaceAt, 89
   replaceLetterAt, 93
   reverseComplement, 95
   translate, 101
   xscat, 107
   XStringSet-io, 122

* methods
   AAString-class, 4
   chartr, 6
   DNAString-class, 10
   findPalindromes, 11
   letter, 22
   letterFrequency, 23
   lowlevel-matching, 30
   MaskedXString-class, 35
   maskMotif, 37
   match-utils, 40
   matchLRPatterns, 41
   matchPattern, 43
   matchPDict, 48
alphabet,XStringQuality-method
(XStringQuality-class), 111
alphabetFrequency, 5, 7, 11, 36, 45, 50, 77, 98
alphabetFrequency(letterFrequency), 23
alphabetFrequency, AAString-method
(letterFrequency), 23
alphabetFrequency, AAStringSet-method
(letterFrequency), 23
alphabetFrequency, DNAString-method
(letterFrequency), 23
alphabetFrequency, DNAStringSet-method
(letterFrequency), 23
alphabetFrequency, MaskedXString-method
(letterFrequency), 23
alphabetFrequency, MultipleAlignment-method
(MultipleAlignment-class), 69
alphabetFrequency, RNAString-method
(letterFrequency), 23
alphabetFrequency, RNAStringSet-method
(letterFrequency), 23
alphabetFrequency, XString-method
(letterFrequency), 23
alphabetFrequency, XStringSet-method
(letterFrequency), 23
alphabetFrequency, XStringViews-method
(XStringViews-class), 131
AMINO_ACID_CODE, 4, 5, 16, 77
anyNA,XStringSet-method
(XStringSet-comparison), 119
Arithmetic, 25
as.character, MaskedXString-method
(MaskedXString-class), 35
as.character, MultiColumn-method
(MultiAlignment-class), 69
as.character, XString-method
(XString-class), 108
as.character, XStringSet-method
(XStringSet-class), 113
as.character, XStringViews-method
(XStringViews-class), 131
as.data.frame, XStringSet-method
(XStringSet-class), 113
as.data.frame, XStringViews-method
(XStringViews-class), 131
as.factor, XStringSet-method
(XStringSet-class), 113
as.list, MTB_PDict-method (PDict-class), 82
as.list, SparseList-method (Biostrings internals), 6
as.matrix, MultipleAlignment-method
(MultipleAlignment-class), 69
as.matrix, XStringQuality-method
(XStringQuality-class), 69
as.matrix, XStringSet-method
(XStringSet-class), 113
as.matrix, XStringViews-method
(XStringViews-class), 131
as.vector, XString-method
(XString-class), 108
as.vector, XStringQuality-method
(XStringQuality-class), 111
as.vector, XStringSet-method
(XStringSet-class), 113
Biostats, 6
BSgenome, 17, 18, 94, 95
BString, 4, 10, 11, 26, 98, 111, 119
BString (XString-class), 108
BString-class (XString-class), 108
BStringSet, 80, 81, 86, 88, 111, 119,
124–126, 130
BStringSet (XStringSet-class), 113
BStringSet-class, 112
BStringSet-class (XStringSet-class), 113
BStringSetList (XStringSetList-class),
129
BStringSetList-class
(XStringSetList-class), 129
ByPos_MIndex-class (MIndex-class), 66
c, 24
cat, 124
celling, 105
CharacterList, 90
chartr, 6, 7, 20, 95, 96
chartr, ANY, ANY, MaskedXString-method
(chartr), 6
chartr, ANY, ANY, XString-method (chartr), 6
chartr, ANY, ANY, XStringSet-method
(chartr), 6
chartr, ANY, ANY, XStringViews-method
(chartr), 6
chisq.test, 10
class: AAMultipleAlignment
  (MultipleAlignment-class), 69
class: AASTring (AAString-class), 4
class: AASTringSet (XStringSet-class), 113
class: AASTringSetList
  (XStringSetList-class), 129
class: ACTree2 (PDict-class), 82
class: BString (XString-class), 108
class: BStringSet (XStringSet-class), 113
class: BStringSetList
  (XStringSetList-class), 129
class: ByPos_MIndex (MIndex-class), 66
class: DNAMultipleAlignment
  (MultipleAlignment-class), 69
class: DNAString (DNAString-class), 10
class: DNAStringSet (XStringSet-class), 113
class: DNAStringSetList
  (XStringSetList-class), 129
class: Expanded_TB_PDict
  (PDict-class), 82
class: IlluminaQuality
  (XStringQuality-class), 111
class: MaskedAAString
  (MaskedXString-class), 35
class: MaskedBString
  (MaskedXString-class), 35
class: MaskedDNAString
  (MaskedXString-class), 35
class: MaskedRNAString
  (MaskedXString-class), 35
class: MaskedRNAString
  (MaskedXString-class), 35
class: MaskedString
  (MaskedXString-class), 35
class: MIndex (MIndex-class), 66
class: MTB_PDict
  (PDict-class), 82
class: MultipleAlignment
  (MultipleAlignment-class), 69
class: PDict
  (PDict-class), 82
class: PDict3Parts
  (PDict-class), 82
class: PhredQuality
  (XStringQuality-class), 111
class: PreprocessedTB
  (PDict-class), 82
class: QualityScaledAAStringSet
  (QualityScaledXStringSet-class), 86
class: QualityScaledBStringSet
  (QualityScaledXStringSet-class), 86
class: QualityScaledDNAStringSet
  (QualityScaledXStringSet-class), 86
class: QualityScaledDNAStringSet
  (QualityScaledXStringSet-class), 86
class: QualityScaledDNAStringSet
  (QualityScaledXStringSet-class), 86
class: QualityScaledRNAStringSet
  (QualityScaledXStringSet-class), 86
class: QualityScaledRNAStringSet
  (QualityScaledXStringSet-class), 86
class: QualityScaledXStringSet
  (QualityScaledXStringSet-class), 86
class: RNAMultipleAlignment
  (MultipleAlignment-class), 69
class: RNAString (RNAString-class), 97
class: RNAStringSet (XStringSet-class), 113
class: RNAStringSetList
  (XStringSetList-class), 129
class: SolexaQuality
  (XStringQuality-class), 111
class: SparseList (Biostrings internals), 6
class: TB_PDict
  (PDict-class), 82
class: Twobit
  (PDict-class), 82
class: XString
  (XString-class), 108
class: XStringCodec (Biostrings internals), 6
class: XStringPartialMatches
  (XStringPartialMatches-class), 110
class: XStringQuality
  (XStringQuality-class), 111
class: XStringSet (XStringSet-class), 113
class: XStringSetList
  (XStringSetList-class), 129
class: XStringViews
  (XStringViews-class), 131
codons (translate), 101
codons, DNAString-method (translate), 101
codons, MaskedDNAString-method (translate), 101
codons, MaskedRNAString-method (translate), 101
codons, RNAString-method (translate), 101
coerce, AAString, MaskedAAString-method
  (MaskedXString-class), 35
coerce, ANY, AAStringSet-method
  (XStringSet-class), 113
coerce, ANY, BStringSet-method
coerce, ANY, DNAStringSet-method
(XStringSet-class), 113
coerce, ANY, RNAStringSet-method
(XStringSet-class), 113
coerce, ANY, XStringSet-method
(XStringSet-class), 113
coerce, BString, IlluminaQuality-method
(XStringQuality-class), 111
coerce, BString, MaskedBString-method
(MaskedXString-class), 35
coerce, BString, PhredQuality-method
(XStringQuality-class), 111
coerce, BString, SolexaQuality-method
(XStringQuality-class), 111
core, character, AAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, AAString-method
(XString-class), 108
core, character, BString-method
(XString-class), 108
core, character, DNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, DNAString-method
(XString-class), 108
core, character, IlluminaQuality-method
(XStringQuality-class), 111
core, character, PhredQuality-method
(XStringQuality-class), 111
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAString-method
(XString-class), 108
core, character, RNAString-method
(XString-class), 108
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAString-method
(XString-class), 108
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
coerce, character, AAString-method
(XString-class), 108
coerce, character, BString-method
(XString-class), 108
coerce, character, DNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, DNAString-method
(XString-class), 108
core, character, IlluminaQuality-method
(XStringQuality-class), 111
core, character, PhredQuality-method
(XStringQuality-class), 111
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAString-method
(XString-class), 108
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAString-method
(XString-class), 108
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
core, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
coerce, character, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
INDEX

(letterFrequency), 23
consensusMatrix, matrix-method
(letterFrequency), 23
consensusMatrix, MultipleAlignment-method
(MultipleAlignment-class), 69
consensusMatrix, XStringSet-method
(letterFrequency), 23
consensusMatrix, XStringViews-method
(letterFrequency), 23
consensusString, 72
consensusString (letterFrequency), 23
consensusString, AAMultipleAlignment-method
(MultipleAlignment-class), 69
consensusString, ANY-method
(letterFrequency), 23
consensusString, BStringSet-method
(letterFrequency), 23
consensusString, DNAMultipleAlignment-method
(MultipleAlignment-class), 69
consensusString, DNAStringSet-method
(letterFrequency), 23
consensusString, matrix-method
(MultipleAlignment-class), 69
consensusString, MultipleAlignment-method
(MultipleAlignment-class), 69
consensusString, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
consensusString, RNAStringSet-method
(letterFrequency), 23
consensusString, XStringViews-method
(letterFrequency), 23
consensusViews
(MultipleAlignment-class), 69
consensusViews, AAMultipleAlignment-method
(MultipleAlignment-class), 69
consensusViews, DNAMultipleAlignment-method
(MultipleAlignment-class), 69
consensusViews, MultipleAlignment-method
(MultipleAlignment-class), 69
consensusViews, RNAMultipleAlignment-method
(MultipleAlignment-class), 69
consensusViews
(MultipleAlignment-class), 69
countPattern, 49
countPattern (matchPattern), 43
countPattern, character-method
(matchPattern), 43
countPattern, MaskedXString-method
(matchPattern), 43
countPattern, XString-method
(matchPattern), 43
countPattern, XStringSet-method
(matchPattern), 43
countPattern, XStringViews-method
(matchPattern), 43
countPDict, 27
countPDict (matchPDict), 48
countPDict, MaskedXString-method
(matchPDict), 48
countPDict, XString-method
(matchPDict), 48
countPDict, XStringSet-method
(matchPDict), 48
countPDict, XStringViews-method
(matchPDict), 48
countPWM (matchPWM), 63
countPWM, character-method
(matchPWM), 63
countPWM, DNAString-method
(matchPWM), 63
countPWM, MaskedDNAString-method
(matchPWM), 63
countPWM, XStringViews-method
(matchPWM), 63
coverage, 27, 40, 41
coverage, MaskedXString-method
(match-utils), 40
coverage, MIndex-method
(match-utils), 40
deletion (moved_to_pwalign), 68
detail, 8
detail, MultipleAlignment-method
(MultipleAlignment-class), 69
dim, MultipleAlignment-method
(MultipleAlignment-class), 69
dinucleotideFrequency
(nucleotideFrequency), 75
dinucleotideFrequencyTest, 8
dinucleotideFrequencyTest, DNAStringSet-method
(nucleotideFrequencyTest), 8
dinucleotideFrequencyTest, RNAStringSet-method
(nucleotideFrequencyTest), 8
DNA_ALPHABET, 80, 85, 98, 114, 119, 124
DNA_ALPHABET (DNAString-class), 10
DNA_BASES (DNAString-class), 10
DNAMultipleAlignment
(MultipleAlignment-class), 69
DNAMultipleAlignment-class
(MultipleAlignment-class), 69
DNAString, 4, 16, 22, 31, 42, 49, 61, 64, 82–84, 94–96, 98, 100–102, 108,
INDEX
INDEX

hasOnlyBaseLetters, XStringViews-method
(letterFrequency), 23
head, PDict3Parts-method (PDict-class), 82
head, TB_PDict-method (PDict-class), 82
HNF4alpha, 19
IlluminaQuality, 87
IlluminaQuality (XStringQuality-class), 111
IlluminaQuality-class
(XStringQuality-class), 111
indel (moved_to_pwalign), 68
initialize, AChree2-method
(PDict-class), 82
initialize, PreprocessedTB-method
(PDict-class), 82
initialize, Twobit-method (PDict-class), 82
initialize, XStringCodec-method
(Biostrings internals), 6
injectHardMask, 19, 36, 95
injectHardMask, MaskedXString-method
(injectHardMask), 19
injectHardMask, XStringViews-method
(injectHardMask), 19
injectSNPs, 94, 95
insertion (moved_to_pwalign), 68
IntegerList, 89, 90
IntegerRanges, 67, 80, 81, 89, 90
IntegerRangesList, 67, 89, 90
IRanges, 67, 114
IRanges-class, 67
is.na, XStringSet-method
(XStringSet-comparison), 119
is.unsorted, 120
isMatchingAt, 49, 50
isMatchingAt (lowlevel-matching), 30
isMatchingEndingAt (lowlevel-matching), 30
isMatchingEndingAt, character-method
(lowlevel-matching), 30
isMatchingEndingAt, XString-method
(lowlevel-matching), 30
isMatchingEndingAt, XStringSet-method
(lowlevel-matching), 30
isMatchingStartingAt, character-method
(lowlevel-matching), 30
isMatchingStartingAt, XString-method
(lowlevel-matching), 30
isMatchingStartingAt, XStringSet-method
(lowlevel-matching), 30
IUPAC_CODE_MAP, 7, 10, 11, 21, 31, 33, 42, 85, 95, 96, 98
lcprefix (pmatchPattern), 86
lcprefix, character, character-method
(pmatchPattern), 86
lcprefix, character, XString-method
(pmatchPattern), 86
lcprefix, XString, character-method
(pmatchPattern), 86
lcprefix, XString, XString-method
(pmatchPattern), 86
lcsubstr (pmatchPattern), 86
lcsubstr, character, character-method
(pmatchPattern), 86
lcsubstr, character, XString-method
(pmatchPattern), 86
lcsubstr, XString, character-method
(pmatchPattern), 86
lcsubstr, XString, XString-method
(pmatchPattern), 86
lcsuffix (pmatchPattern), 86
lcsuffix, character, character-method
(pmatchPattern), 86
lcsuffix, character, XString-method
(pmatchPattern), 86
lcsuffix, XString, character-method
(pmatchPattern), 86
lcsuffix, XString, XString-method
(pmatchPattern), 86
length, MaskedXString-method
(MaskedXString-class), 35
length, MIndex-method (MIndex-class), 66
length, PDict-method (PDict-class), 82
length, PDict3Parts-method
(PDict-class), 82
length, PreprocessedTB-method
(PDict-class), 82
length, SparseList-method (Biostrings
internals), 6
letter, 5, 11, 22, 98, 109, 111, 132
letter, character-method (letter), 22
letter, MaskedXString-method (letter), 22
INDEX

letter,XString-method (letter), 22
letter,XString-method (letter), 22
letterFrequency, 23
letterFrequency,MaskedXString-method (letterFrequency), 23
letterFrequency,XString-method (letterFrequency), 23
letterFrequency,XStringSet-method (letterFrequency), 23
letterFrequency,XStringViews-method (letterFrequency), 23
letterFrequencyInSlidingView (letterFrequency), 23
List, 130
List-class, 130
longestConsecutive, 29
lowlevel-matching, 30, 41, 45, 106
make_XString_from_string (Biostrings internals), 6
make_XString_from_string,XString-method (Biostrings internals), 6
make_XStringSet_from_strings (Biostrings internals), 6
make_XStringSet_from_strings,XStringSet-method (Biostrings internals), 6
mask (maskMotif), 37
MaskCollection, 35
MaskCollection-class, 36, 38
MaskedAAString, 20
MaskedAAString (MaskedXString-class), 35
MaskedAAString-class (MaskedXString-class), 35
MaskedBString, 20
MaskedBString (MaskedXString-class), 35
MaskedBString-class (MaskedXString-class), 35
maskeddim (MultipleAlignment-class), 69
maskeddim,MultipleAlignment-method (MultipleAlignment-class), 69
MaskedDNAString, 20, 64, 95, 101, 102
MaskedDNAString (MaskedXString-class), 35
MaskedDNAString-class, 50
MaskedDNAString-class (MaskedXString-class), 35
maskedncol (MultipleAlignment-class), 69
maskedncol,MultipleAlignment-method (MultipleAlignment-class), 69
maskedncol,MultipleAlignment-method (MultipleAlignment-class), 69
maskedncol,MultipleAlignment-method (MultipleAlignment-class), 69
maskedncol,MultipleAlignment-method (MultipleAlignment-class), 35
maskedncol,MultipleAlignment-method (MultipleAlignment-class), 69
MaskedRNAString, 20, 95, 101, 102
MaskedRNAString (MaskedXString-class), 35
MaskedRNAString-class (MaskedXString-class), 35
maskGaps (MultipleAlignment-class), 69
maskGaps,MultipleAlignment-method (MultipleAlignment-class), 69
maskMotif, 13, 20, 36, 37, 45
maskMotif,MaskedXString,character-method (maskMotif), 37
maskMotif,MaskedXString,XString-method (maskMotif), 37
maskMotif,MultipleAlignment,ANY-method (MultipleAlignment-class), 69
maskMotif,XString,ANY-method (maskMotif), 37
masks (MaskedXString-class), 35
masks,MaskedXString-method (MaskedXString-class), 35
masks,XString-method (MaskedXString-class), 35
masks<-, (MaskedXString-class), 35
masks<-,MaskedXString,MaskCollection-method (MaskedXString-class), 35
masks<-,MaskedXString,NULL-method (MaskedXString-class), 35
masks<-,XString,ANY-method (MaskedXString-class), 35
masks<-,XString,NULL-method (MaskedXString-class), 35
match, 120
INDEX

nchar,XString-method (XString-class), 108
nchar,XStringSet-method (XStringSet-class), 113
nchar,XStringSetList-method (XStringSetList-class), 129
nchar,XStringViews-method (XStringViews-class), 131
ncol,MultipleAlignment-method (MultipleAlignment-class), 69
nedit (moved_to_pwalign), 68
neditAt (lowlevel-matching), 30
neditEndingAt (lowlevel-matching), 30
neditEndingAt,character-method (lowlevel-matching), 30
neditEndingAt,XString-method (lowlevel-matching), 30
neditEndingAt,XStringSet-method (lowlevel-matching), 30
neditStartingAt (lowlevel-matching), 30
neditStartingAt,character-method (lowlevel-matching), 30
neditStartingAt,XString-method (lowlevel-matching), 30
neditStartingAt,XStringSet-method (lowlevel-matching), 30
neditStartingAt,XStringSet-method (lowlevel-matching), 30
neditStartingAt,XStringSet-method (lowlevel-matching), 30
needwunsQS, 74
nindel (moved_to_pwalign), 68
nmatch (match-utils), 40
nmatch,ANY,XStringViews-method (match-utils), 40
nmismatch (match-utils), 40
nmismatch,ANY,XStringViews-method (match-utils), 40
nnodes (PDdict-class), 82
nnodes,ACTree2-method (PDdict-class), 82
NormalIRanges, 70
nrow,MultipleAlignment-method (MultipleAlignment-class), 69
nucleotideFrequency, 75
nucleotideFrequencyAt, 10, 33
nucleotideFrequencyAt (nucleotideFrequency), 75
nucleotideFrequencyAt,XString-method (nucleotideFrequency), 75
nucleotideFrequencyAt,XStringSet-method (nucleotideFrequency), 75
nucleotideFrequencyAt,XStringViews-method (nucleotideFrequency), 75
nucleotideSubstitutionMatrix (moved_to_pwalign), 68
oligonucleotideFrequency, 27
oligonucleotideFrequency (nucleotideFrequency), 75
oligonucleotideFrequency,MaskedXString-method (nucleotideFrequency), 75
oligonucleotideFrequency,XString-method (nucleotideFrequency), 75
oligonucleotideFrequency,XStringSet-method (nucleotideFrequency), 75
oligonucleotideFrequency,XStringViews-method (nucleotideFrequency), 75
oligonucleotideTransitions (nucleotideFrequency), 75
order, 120
padAndClip, 80, 90
pairwiseAlignment, 45, 74, 112
PairwiseAlignments (moved_to_pwalign), 68
PairwiseAlignments (moved_to_pwalign), 68
PairwiseAlignments-class, 74, 112
PairwiseAlignmentsSingleSubject (moved_to_pwalign), 68
palindromeArmLength (findPalindromes), 11
palindromeArmLength,DNAString-method (findPalindromes), 11
palindromeArmLength,RNAString-method (findPalindromes), 11
palindromeArmLength,XString-method (findPalindromes), 11
palindromeArmLength,XStringSet-method (findPalindromes), 11
palindromeArmLength,XStringViews-method (findPalindromes), 11
palindromeArmLength,XStringViews-method (findPalindromes), 11
palindromeArmLength,XStringMethods-method (findPalindromes), 11
palindromeLeftArm (findPalindromes), 11
palindromeLeftArm,XString-method (findPalindromes), 11
palindromeLeftArm,XStringViews-method (findPalindromes), 11
palindromeRightArm (findPalindromes), 11
palindromeRightArm,XString-method (findPalindromes), 11
palindromeRightArm,XStringViews-method (findPalindromes), 11
parallel_slot_names, QualityScaledXStringSet-method (QualityScaledXStringSet-class), 86
PartitioningByEnd, 130
paste, 107
pattern (XStringPartialMatches-class), 110
pattern, XStringPartialMatches-method (XStringPartialMatches-class), 110
patternFrequencies (PDict-class), 82
patternFrequency, PDict-method (PDict-class), 82
pcompare, ANY, XStringSet-method (XStringSet-comparison), 119
pcompare, XStringSet, ANY-method (XStringSet-comparison), 119
pcompare, XStringSet, XStringSet-method (XStringSet-comparison), 119
PDict, 49, 50, 57
PDict (PDict-class), 82
PDict, AsIs-method (PDict-class), 82
PDict, character-method (PDict-class), 82
PDict, DNAStringSet-method (PDict-class), 82
PDict, probetable-method (PDict-class), 82
PDict, XStringViews-method (PDict-class), 82
PDict-class, 50, 58, 67, 82
PDict3Parts (PDict-class), 82
PDict3Parts-class (PDict-class), 82
PhredQuality, 87
PhredQuality (XStringQuality-class), 111
PhredQuality-class (XStringQuality-class), 111
pid (moved_to_pwalign), 68
pmatchPattern, 86
pmatchPattern, character-method (pmatchPattern), 86
pmatchPattern, XString-method (pmatchPattern), 86
pmatchPattern, XStringViews-method (pmatchPattern), 86
PreprocessedTB (PDict-class), 82
PreprocessedTB-class (PDict-class), 82
PWM (matchPWM), 63
PWM, character-method (matchPWM), 63
PhredQuality-class (XStringQuality-class), 111
PhredQuality-class (XStringQuality-class), 111
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
quality (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
QualityScaledDNAStringSet-class (QualityScaledXStringSet-class), 86
qualitySubstitutionMatrices (moved_to_pwalign), 68
rank, 120
read.Mask, 38
readAAMultipleAlignment (MultipleAlignment-class), 69
readAAStringSet (XStringSet-io), 122
readBStringSet (XStringSet-io), 122
INDEX

seqinfo<-,DNAStringSet-method
(seqinfo-methods), 99
seqtype (Biostrings internals), 6
seqtype,AAString-method
(XString-class), 108
seqtype,BString-method
(XString-class), 108
seqtype,MaskedXString-method
(MaskedXString-class), 35
seqtype,MultipleAlignment-method
(MultipleAlignment-class), 69
seqtype,MaskedXString-method
(MaskedXString-class), 35
seqtype<-,MaskedXString-method
(MaskedXString-class), 35
seqtype<-,XString-method
(XString-class), 108
seqtype<-,XStringSet-method
(XStringSet-class), 113
seqtype,XStringSetList-method
(XStringSetList-class), 129
seqtype,XStringViews-method
(XStringViews-class), 131
seqtype<-(Biostrings internals), 6
seqtype<-,MaskedXString-method
(MaskedXString-class), 35
seqtype<-,XString-method
(XString-class), 108
seqtype<-,XStringSet-method
(XStringSet-class), 113
seqtype<-,XStringSetList-method
(XStringSetList-class), 129
seqtype<-,XStringViews-method
(XStringViews-class), 131
show,8
show,ACtree2-method (PDict-class), 82
show,MaskedXString-method
(MaskedXString-class), 35
show,MIndex-method (MIndex-class), 66
show,MTB_PDict-method (PDict-class), 82
show,MultipleAlignment-method
(MultipleAlignment-class), 69
show,QualityScaledXStringSet-method
(QualityScaledXStringSet-class), 86
show,TB_PDict-method (PDict-class), 82
show,TwoBit-method (PDict-class), 82
show,XString-method (XString-class), 108
show,XStringPartialMatches-method
(XStringPartialMatches-class), 110
show,XStringSet-method
(XStringSet-class), 113
show,XStringSetList-method
(XStringSetList-class), 129
show,XStringViews-method
(XStringViews-class), 131
showAsCell,XString-method
(XString-class), 108
showAsCell,XStringSet-method
(XStringSet-class), 113
showAsCell,XStringSetList-method
(XStringSetList-class), 129
SolexaQuality,87
SolexaQuality (XStringQuality-class), 111
SolexaQuality-class
(XStringQuality-class), 111
sort,120
SparseList (Biostrings internals), 6
SparseList-class (Biostrings internals), 6
stackStrings (padAndClip), 80
stackStringsFromBam,81
startIndex (MIndex-class), 66
startIndex,ByPos_MIndex-method
(MIndex-class), 66
stringDist (moved_to_pwalign), 68
strsplit,27
subpatterns
(XStringPartialMatches-class), 110
subpatterns,XStringPartialMatches-method
(XStringPartialMatches-class), 110
subseq,23,90,109,114,116
subseq,character-method
(XStringSet-class), 113
subseq,MappedXString-method
(MappedXString-class), 35
subseq<-,character-method
(XStringSet-class), 113
subseq<-,XString-method
(XStringSet-class), 113
substitution matrices,74
substr,114,116
substr,XString-method (XString-class), 108
substring, 26
substring,XString-method
  (XString-class), 108
tail,PDict3Parts-method (PDict-class), 82
tail,TB_PDict-method (PDict-class), 82
tb,PDict3Parts-method (PDict-class), 82
tb,TB_PDict-method (PDict-class), 82
tb,width,PDict3Parts-method (PDict-class), 82
tb,width,PreprocessedTB-method (PDict-class), 82
tb.width,PDict3Parts-method (PDict-class), 82
tb.width,PreprocessedTB-method (PDict-class), 82
tb.width,TB_PDict-method (PDict-class), 82
TB_PDict (PDict-class), 82
TB_PDict-class (PDict-class), 82
threebands, 114
threebands,character-method (XStringSet-class), 113
toupper, 63
toComplex, 100
toComplex,DNAString-method (toComplex), 100
toString,MaskedXString-method
  (MaskedXString-class), 35
toString,XString-method
  (XString-class), 108
toString,XStringSet-method
  (XStringSet-class), 113
toString,XStringViews-method
  (XStringViews-class), 131
unalignov(moved_to_PWalign), 68
unique, 120
uniqueLetters, 7
uniqueLetters,letterFrequency-method
  (letterFrequency), 23
uniqueLetters,MaskedXString-method
  (letterFrequency), 23
uniqueLetters,XString-method
  (letterFrequency), 23
uniqueLetters,XStringSet-method
  (letterFrequency), 23
uniqueLetters,XStringViews-method
  (letterFrequency), 23
unitScale(matchPWM), 63
unlist,MIndex-method (MIndex-class), 66
unlist,XStringSet-method
  (XStringSet-class), 113
unmasked, 26
unmasked,MaskedXString-method (MaskedXString-class), 35
updateObject,XString-method
  (XString-class), 108
updateObject,XStringSet-method
  (XStringSet-class), 113
vcountPattern, 49
vcountPattern,matchPattern-method (matchPattern), 43
vcountPattern,MaskedXString-method (matchPattern), 43
vcountPattern,XString-method (matchPattern), 43
vcountPattern,XStringSet-method (matchPattern), 43
vcountPattern,XStringViews-method (matchPattern), 43
vcountPDict (matchPDict), 48
vcountPDict,MaskedXString-method (matchPDict), 48
vcountPDict,XString-method (matchPDict), 48
vcountPDict,XStringSet-method (matchPDict), 48
vcountPDict,XStringViews-method (matchPDict), 48
vmatchPattern,character-method (matchPattern), 43
vmatchPattern,XString-method (matchPattern), 43
vmatchPattern,XStringSet-method (matchPattern), 43
vmatchPattern,XStringViews-method (matchPattern), 43
vmatchPDict (matchPDict), 48
vmatchPDict,MaskedXString-method (matchPDict), 48
vmatchPDict,XString-method (matchPDict), 48
vmatchPDict,XStringSet-method (matchPDict), 48
vmatchPDict,XStringViews-method (matchPDict), 48
whichPDict,MaskedXString-method (matchPDict), 48
whichPDict,XString-method (matchPDict), 48
whichPDict,XStringSet-method (matchPDict), 48
whichPDict,XStringViews-method (matchPDict), 48
vwhichPDict (matchPDict), 48
vwhichPDict,MaskedXString-method (matchPDict), 48
vwhichPDict,XString-method (matchPDict), 48
vwhichPDict,XStringSet-method (matchPDict), 48
vwhichPDict,XStringViews-method (matchPDict), 48
which.isMatchingAt (lowlevel-matching), 30
which.isMatchingEndingAt (lowlevel-matching), 30
which.isMatchingEndingAt,character-method (lowlevel-matching), 30
which.isMatchingEndingAt,XString-method (lowlevel-matching), 30
which.isMatchingEndingAt,XStringSet-method (lowlevel-matching), 30
which.isMatchingStartingAt (lowlevel-matching), 30
which.isMatchingStartingAt,character-method (lowlevel-matching), 30
which.isMatchingStartingAt,XString-method (lowlevel-matching), 30
which.isMatchingStartingAt,XStringSet-method (lowlevel-matching), 30
whichPDict,57
whichPDict,MaskedXString-method (matchPDict), 48
whichPDict,XString-method (matchPDict), 48
whichPDict,XStringSet-method (matchPDict), 48
whichPDict,XStringViews-method (matchPDict), 48
whichPDict,MaskedXString-method (matchPDict), 48
whichPDict,XString-method (matchPDict), 48
whichPDict,XStringSet-method (matchPDict), 48
whichPDict,XStringViews-method (matchPDict), 48
whichPDict,MaskedXString-method (matchPDict), 48
whichPDict,XString-method (matchPDict), 48
whichPDict,XStringSet-method (matchPDict), 48
whichPDict,XStringViews-method (matchPDict), 48
width,character-method (XStringSet-class), 113
width,PDict-method (PDict-class), 82
width,PDict3Parts-method (PDict-class), 82
width,PreprocessedTB-method (PDict-class), 82
width0 (MIndex-class), 66
width0,MIndex-method (MIndex-class), 66
tables,character-method (XStringSet-class), 113
windows,QualityScaledXStringSet-method (QualityScaledXStringSet-class), 86
write.phylip (MultipleAlignment-class), 69
writePairwiseAlignments (moved_to_pwalign), 68
writeQualityScaledXStringSet, 126
writeQualityScaledXStringSet
  (QualityScaledXStringSet-class), 86
writeXStringSet, 88, 116
writeXStringSet (XStringSet-io), 122

xscat, 107
xscodes (Biostrings internals), 6
xscodes, ANY-method (Biostrings internals), 6
XString, 4, 6, 7, 10–12, 18, 20, 22–24, 26, 31,
  32, 35, 42, 44, 49, 66, 67, 70, 74, 75,
  77, 86, 87, 90, 94, 96, 98,
  105–107, 113, 115, 116, 131, 132
XString (XString-class), 108
XString-class, 5, 18, 23, 27, 33, 36, 38, 41,
  42, 77, 86, 106, 107, 108, 111, 132
XStringCodec (Biostrings internals), 6
XStringCodec-class (Biostrings internals), 6
XStringPartialMatches-class, 110
XStringQuality, 86–88
XStringQuality (XStringQuality-class), 111
XStringQuality-class, 111
XStringSet, 6, 7, 18, 24–26, 31–33, 44, 49,
  70, 71, 75–77, 80, 81, 87–90, 96,
  105–108, 116, 119, 120, 122, 125,
  129, 130
XStringSet (XStringSet-class), 113
XStringSet-class, 10, 18, 27, 68, 72, 77,
  106, 107, 109, 113, 120, 130, 132
XStringSet-comparison, 116, 119
XStringSet-io, 122
XStringSetList, 90, 116, 130
XStringSetList (XStringSetList-class),
  129
XStringSetList-class, 129
XStringViews, 6, 7, 12, 13, 20, 22–26, 38, 40,
  42, 44, 45, 61, 63, 67, 70, 72, 75–77,
  81–83, 87, 95, 96, 102, 103, 107, 113, 114, 116, 130
XStringViews (XStringViews-class), 131
XStringViews-class, 13, 20, 23, 27, 38, 41,
  42, 50, 61, 65, 67, 77, 85, 86, 96,
  107, 109, 111, 131
XVector, 114
XVector-class, 109

yeastSEQCHR1, 133

INDEX