

Sequence Searching and Matching with Biostrings

*Aidan Lakshman**

*aidanlakshman@gmail.com

5 July 2026

Abstract

Biostrings encapsulates a wide variety of sequence matching functions, but using them can be relatively confusing. This vignette walks through examples of using the many functions available in Biostrings along with explanations of their output types.

Package

Biostrings 2.81.5

Contents

1	Brief Recap	3
1.1	XString objects	3
1.2	XStringSets	3
1.3	XStringViews	4
1.4	Summary	6
2	Low-Level Matching	6
2.1	Edit Distance	6
2.2	Simple Matching	7
2.3	Summary	9
3	Fixed Pattern Matching	9
3.1	matchPattern	9
3.2	countPattern	12
3.3	One Pattern, Multiple Subjects	12
3.4	Summary	14
4	Matching with PDicts	14
4.1	What is a PDict?	14
4.2	Exact Matching	15
4.3	Trusted Bands	17

Sequence Searching and Matching with Biostrings

4.4	Inexact Matching	18
4.5	PDict Matching Without a PDict	20
4.6	Performance Notes	20
5	Other Matching Utilities	20
5.1	Finding Mismatches	21
5.2	Finding Coverage.	22
6	Biological Applications	23
6.1	Finding Palindromes	24
6.2	Finding Amplicons for a Primer Pair.	26
6.3	Position Weight Matrix (PWM) Matching	28
7	Further Reading	29
8	Session info	29

1 Brief Recap

Before diving into searching and matching in Biostrings, let's first briefly review the central data structures and concepts in the package. If you're already familiar with XString objects and the various containers to hold them, feel free to skip forward to the next section.

1.1 XString objects

The focus of Biostrings is providing containers to efficiently work with biological data. Operations in Biostrings revolve around XString¹ objects. XString isn't actually a type; it's the collective name for a set of objects of the form `*String`:

- `DNAString`: holds DNA data (A/T/G/C)
- `RNAString`: holds RNA data (A/U/G/C)
- `AAString`: holds amino acid data
- `BString`: holds any arbitrary character

Aside from `BString`, all the XString types also support standard IUPAC ambiguity codes.

The simplest way to create XString objects is directly from a character object:

```
DNAString("ATGCATGC")
## 8-letter DNAString object
## seq: ATGCATGC

RNAString("AUGCAUGC")
## 8-letter RNAString object
## seq: AUGCAUGC

AAString("ARNDC*")
## 6-letter AAString object
## seq: ARNDC*

BString("I am a BString")
## 14-letter BString object
## seq: I am a BString
```

1.2 XStringSets

Single XString objects are nice, but many biological sequence analyses need to work with multiple sequences. Our primary container to achieve this functionality is the `XStringSet`² object. This container holds a set of XString objects that are all the same type (all DNA, all RNA, etc.). There is one XStringSet type for each XString type (i.e., DNA, RNA, AA, B).

It's also possible to create XStringSet objects from character vectors:

```
DNAStringSet(c("ATGC", "CGTA", "AAAA", "TT"))
## DNAStringSet object of length 4:
##      width seq
## [1]      4 ATGC
```

¹XString: a container that holds a single sequence of a particular type.

²XStringSet: a container that holds a collection of XString objects together.

Sequence Searching and Matching with Biostrings

```
## [2]      4 CGTA
## [3]      5 AAAAA
## [4]      2 TT
```

However, they are most often used to read in sequence sets from files, such as FASTA files:

```
filepath1 <- system.file("extdata", "someORF.fa", package="Biostrings")
readDNAStringSet(filepath1)
## DNAStringSet object of length 7:
##      width seq                                     names
## [1]  5573 ACTTGTAATATATATCTTTTATTT...CTTATCGACCTTATTGTTGATAT YAL001C TFC3 SGDI...
## [2]  5825 TTCCAAGGCCGATGAATTCGACT...AGTAAATTTTTTCTATTCTCTT YAL002W VPS8 SGDI...
## [3]  2987 CTTTCATGTCAGCCTGCACTTCTG...TGGTACTCATGTAGCTGCCTCAT YAL003W EFB1 SGDI...
## [4]  3929 CACTCATATCGGGGGTCTTACTT...TGTCCCGAAACACGAAAAAGTAC YAL005C SSA1 SGDI...
## [5]  2648 AGAGAAAGAGTTTCACCTCTTGA...ATATAATTTATGTGTGAACATAG YAL007C ERP2 SGDI...
## [6]  2597 GTGTCCGGGCCTCGCAGGCGTTC...AAGTTTTGGCAGAATGTACTTTT YAL008W FUN14 SGD...
## [7]  2780 CAAGATAATGTCAAAGTTAGTGG...GCTAAGGAAGAAAAAAAATCAC YAL009W SP07 SGDI...
```

Once we have an XStringSet, we can use single brackets (`[]`) to get a subset of the XStringSet, and double brackets (`[[[]]`) to get the component XStrings:

```
x <- DNAStringSet(c("ATGC", "CGTA", "AAAA", "TT"))

# Subsetting to a DNAStringSet
x[1]
## DNAStringSet object of length 1:
##      width seq
## [1]      4 ATGC

# Subsetting to a DNAString
x[[1]]
## 4-letter DNAString object
## seq: ATGC
```

1.3 XStringViews

When we search for patterns in a sequence, we're looking for specific subsequences within a single sequence that match whatever we're looking for. To implement that functionality, we have the XStringViews³ classes, which store "views" (subsequences) of a single XString object. These will be very important for searching and matching patterns later on.

We can directly construct an XStringViews from an XString object with the `Views` method by either specifying start/end indices, or using the width of the subsequences:

```
seq <- DNAString("AAATTGGGCC")

## Splitting into codons
Views(seq,
      start = seq(1, length(seq), by = 3),
      end = seq(3, length(seq), by = 3))
```

³XStringViews: Stores subsequences of a single XString object.

Sequence Searching and Matching with Biostrings

```
## Views on a 12-letter DNAString subject
## subject: AAATTTGGGCCC
## views:
##      start end width
## [1]    1  3    3 [AAA]
## [2]    4  6    3 [TTT]
## [3]    7  9    3 [GGG]
## [4]   10 12    3 [CCC]

## All overlapping subsequences of length 3
Views(seq,
      start = seq_len(length(seq) - 2),
      width = 3)
## Views on a 12-letter DNAString subject
## subject: AAATTTGGGCCC
## views:
##      start end width
## [1]    1  3    3 [AAA]
## [2]    2  4    3 [AAT]
## [3]    3  5    3 [ATT]
## [4]    4  6    3 [TTT]
## [5]    5  7    3 [TTG]
## [6]    6  8    3 [TGG]
## [7]    7  9    3 [GGG]
## [8]    8 10    3 [GGC]
## [9]    9 11    3 [GCC]
## [10]   10 12    3 [CCC]

## Views can also be "out of range":
Views(seq,
      end = seq_along(seq),
      width = 3)
## Views on a 12-letter DNAString subject
## subject: AAATTTGGGCCC
## views:
##      start end width
## [1]   -1  1    3 [ A]
## [2]    0  2    3 [ AA]
## [3]    1  3    3 [AAA]
## [4]    2  4    3 [AAT]
## [5]    3  5    3 [ATT]
## ...   ...   ...
## [8]    6  8    3 [TGG]
## [9]    7  9    3 [GGG]
## [10]   8 10    3 [GGC]
## [11]   9 11    3 [GCC]
## [12]  10 12    3 [CCC]
```

This class will be commonly returned from the pattern matching functions we'll discuss later, as well as some other internal functions (e.g., `codons`, which returns all the codons in a `DNAString`)

Sequence Searching and Matching with Biostrings

It's worth noting that we can initialize an `XStringSet` from an `XStringViews`, in case we want to save our subsequences as a new `XStringSet` object.

```
seq <- DNAString("AAATTGGGCC")

## `codons` returns an XStringViews
seq_codons <- codons(seq)

seq_codons
## Views on a 12-letter DNAString subject
## subject: AAATTGGGCC
## views:
##      start end width
## [1]      1   3     3 [AAA]
## [2]      4   6     3 [TTT]
## [3]      7   9     3 [GGG]
## [4]     10  12     3 [CCC]

## We can also convert it to a DNAStringSet
DNAStringSet(seq_codons)
## DNAStringSet object of length 4:
##      width seq
## [1]      3 AAA
## [2]      3 TTT
## [3]      3 GGG
## [4]      3 CCC
```

1.4 Summary

Knowledge of `XStrings`, `XStringSets`, and `XStringViews` are all that you'll need to be able to understand the rest of this vignette. However, there exist a number of additional containers and utility functions to work with these objects that I haven't mentioned here. For more information on these utility functions, look at `?XString`. Other containers that may be of interest include `XStringSetList`, `MaskedXString`, and `QualityScaledXStringSet`.

2 Low-Level Matching

Now we can start to look at how to perform some basic matching operations in Biostrings. Matching in Biostrings is done in terms of "patterns" and "subjects". A *pattern* is what we're searching for, and a *subject* is where we're searching. For example, if we wanted to find all `ATG` codons in an *E. coli* genome, `ATG` would be the pattern, and the *E. coli* genome is the subject. A *match* is a subsequence of the pattern that is equal to the pattern (or similar enough, if using things like ambiguity codes).

2.1 Edit Distance

Before we can get to matching, we first need a way to quantify the edit distance between two strings. The *edit distance* is the minimum number of operations it takes to transform one string into another. For instance, the edit distance between `AAA` and `AAB` is 1, since the only letter that differs between the two strings is the final character.

Sequence Searching and Matching with Biostrings

```
## the first string is the pattern,  
## the second is the subject  
neditAt(pattern = "AAAAA", subject = "AAAAB")  
## [1] 1  
  
neditAt("ABCDEFG", "AAADDDG")  
## [1] 4  
  
## Going from subject -> pattern  
## AACGCA -> AACGTA -> AACCTA -> AAGCTA -> GAGCTA: 4 edits  
neditAt("GAGCTA", "AACGCA")  
## [1] 4
```

However, this only works for strings of the same length. If we wanted to get the edit distance between two strings of unequal length, we'd also have to consider insertions and deletions (e.g., AAA and AC would have a distance of two via AAA -> AA -> AC). Considering insertions, deletions, and substitutions produces the Levenshtein distance, which is also supported by `neditAt` if we set `with.indels=TRUE`:

```
## AACGCA -> GACGCA -> GAGCA -> GAGCTA: 3 edits  
neditAt("GAGCTA", "AACGCA", with.indels = TRUE)  
## [1] 3
```

Note that this tries to match the pattern against the subject, so it only looks at the first `n` characters of the subject, where `n` is the length of the pattern. This can lead to some unintuitive results:

```
## Both of these will return 0, because nchar("ABC") = 3,  
## so we only look at the first 3 characters of the pattern ("ABC")  
neditAt("ABC", "ABCD")  
## [1] 0  
neditAt("ABC", "ABCD", with.indels = TRUE)  
## [1] 0  
  
## However, if we change the subject slightly  
## (note just showing the first 3 characters of the subject)  
neditAt("ABC", "DBCA") # DBCA -> ABCA: 1  
## [1] 1  
neditAt("ABC", "DCBA") # DCBA -> DCCA -> DBCA -> ABCA : 3  
## [1] 3  
neditAt("ABC", "DCBA", with.indels = TRUE) # DCBA -> ADCBA -> ABCBA: 2  
## [1] 2
```

2.2 Simple Matching

The previous result for `neditAt` probably seems unintuitive. Why is this function set up the way it is?

It turns out that we can use this `neditAt` function to find matches. We can define a “match” as a pattern that has an edit distance of 0 with the subject.

Sequence Searching and Matching with Biostrings

```
neditAt("ABC", "ABCDE") == 0
## [1] TRUE

isMatchingAt("ABC", "ABCDE")
## [1] TRUE
```

These functions have an “at” suffix because we can specify where we should start searching in the subject. By default, we start searching at the first character (1), but we can start anywhere:

```
neditAt("ABC", "BABCABC", at = 1)
## [1] 3

neditAt("ABC", "BABCABC", at = 2)
## [1] 0

neditAt("ABC", "BABCABC", at = 5)
## [1] 0
```

Both functions are vectorized to calculate for multiple starting positions at the same time:

```
subject <- "BABCABC"
edit_dists <- neditAt("ABC", subject, at = seq_len(nchar(subject)))
edit_dists
## [1] 3 0 3 3 0 3 3

## These are the positions of the matches!
which(edit_dists == 0)
## [1] 2 5

matches <- isMatchingAt("ABC", subject, at = seq_len(nchar(subject)))
which(matches)
## [1] 2 5
```

What if we want to be a little more lenient than exact matching? Previously, we defined a match as any location where `neditAt` returns 0. We can relax these conditions with the `min.mismatch` and `max.mismatch` arguments. With these, the definition of a “match” becomes anywhere where `min.mismatch <= neditAt <= max.mismatch`.

```
pattern <- "AAB"
subject <- "AABBAAA"
s <- seq_len(nchar(subject))
neditAt(pattern, subject, at = s)
## [1] 0 1 3 2 1 1 2

## Default is min.mismatch = max.mismatch = 0
isMatchingAt(pattern, subject, at = s)
## [1] TRUE FALSE FALSE FALSE FALSE FALSE FALSE

## If max.mismatch = 1, we get more matches
isMatchingAt(pattern, subject, at = s, max.mismatch = 1)
## [1] TRUE TRUE FALSE FALSE TRUE TRUE FALSE
```

Sequence Searching and Matching with Biostrings

```
## We can also restrict the lower bound
## This returns all patterns with neditAt == 2
isMatchingAt(pattern, subject, at = s, min.mismatch = 2, max.mismatch = 2)
## [1] FALSE FALSE FALSE TRUE FALSE FALSE TRUE
```

Note that since we're using `neditAt` to define a match, then we can also use the Levenshtein distance as our matcher:

```
neditAt("GAGC", "AACG", with.indels = FALSE)
## [1] 3
isMatchingAt("GAGCTA", "AACGCA", with.indels = FALSE, max.mismatch = 3)
## [1] FALSE

neditAt("GAGC", "AACG", with.indels = TRUE)
## [1] 2
isMatchingAt("GAGCTA", "AACGCA", with.indels = TRUE, max.mismatch = 3)
## [1] TRUE
```

2.3 Summary

With `isMatchingAt`, we have a basic way to get matches in pattern/subject syntax. All future sections will build off this foundational function.

Note that, while we used character vectors for the examples in this section, all the functions shown also work on `XString` and `XStringSet` objects.

3 Fixed Pattern Matching

In the previous section, we talked about how to check if a particular pattern has a match at a specific location in a sequence. While this is theoretically sufficient to do any arbitrary pattern matching, the API is pretty clunky. Ideally, we could just call a single function to automatically find a pattern throughout the subject, without having to worry about where to look.

3.1 `matchPattern`

The `matchPattern` function does exactly what we're looking for. This function has the same input arguments as `isMatchingAt`, but searches in the entire subject for the pattern rather than at specific points.

```
subject <- "BABCABC"
edit_dists <- neditAt("ABC", subject, at = seq_len(nchar(subject)))

which(edit_dists == 0)
## [1] 2 5

matchPattern("ABC", subject)
## Views on a 7-letter BString subject
## subject: BABCABC
```

Sequence Searching and Matching with Biostrings

```
## views:
##      start end width
## [1]    2   4    3 [ABC]
## [2]    5   7    3 [ABC]
```

The output is almost the same as running `isMatchingAt` over the entire subject, except that the return type is an `XStringViews` object.

`XStringViews` objects are perfect for pattern matching, because they can hold all the subsequences that match in the subject along with their start and end positions. Let's see what that looks like using `DNASTrings`:

```
pattern <- DNASTring("AG")
subject <- DNASTring("AGTAGCAGA")

matchPattern(pattern, subject)
## Views on a 9-letter DNASTring subject
## subject: AGTAGCAGA
## views:
##      start end width
## [1]     1   2     2 [AG]
## [2]     4   5     2 [AG]
## [3]     7   8     2 [AG]
```

Here we can see that the pattern `AG` is found starting at indices 1, 4, and 7. In this case, all of our matches are identical to the pattern, but this may not always be the case.

For instance, if we set `fixed = FALSE`, then we can use IUPAC ambiguity codes:

```
pattern <- DNASTring("AGY") # Y = T or G
subject <- DNASTring("AGTAGCAGAAGY")

## By default, ambiguities only match ambiguities
## (i.e., Y = Y)
matchPattern(pattern, subject)
## Views on a 12-letter DNASTring subject
## subject: AGTAGCAGAAGY
## views:
##      start end width
## [1]    10  12     3 [AGY]

## But with fixed = FALSE, it can match other letters
## (i.e., Y = Y, T, or G)
matchPattern(pattern, subject, fixed = FALSE)
## Views on a 12-letter DNASTring subject
## subject: AGTAGCAGAAGY
## views:
##      start end width
## [1]     1   3     3 [AGT]
## [2]     4   6     3 [AGC]
## [3]    10  12     3 [AGY]
```

Similarly, if we use the `max.mismatch` arguments, we can get variable length matches:

Sequence Searching and Matching with Biostrings

```
pattern <- DNASTring("AGY") # Y = T or G
subject <- DNASTring("AGTAGCAGAAGY")

## But with fixed = FALSE, it can match other letters
## (i.e., Y = Y, T, or G)
matchPattern(pattern, subject, fixed = FALSE, max.mismatch = 2)
## Views on a 12-letter DNASTring subject
## subject: AGTAGCAGAAGY
## views:
##      start end width
## [1]    1   3     3 [AGT]
## [2]    4   6     3 [AGC]
## [3]    7   9     3 [AGA]
## [4]    9  11     3 [AAG]
## [5]   10  12     3 [AGY]
```

Sometimes you may get too many matches to deal with. However, we can also match a pattern against an XStringViews subject. In other words, we can find matches *in our matches* to filter them out:

```
pattern1 <- DNASTring("ANN") # Any 3-mer beginning with A
pattern2 <- DNASTring("NNT") # Any 3-mer ending with T
subject <- DNASTring("ATGTTTAATTTATAATATTATAAAAT")

first_matches <- matchPattern(pattern1, subject, fixed = FALSE)
first_matches
## Views on a 26-letter DNASTring subject
## subject: ATGTTTAATTTATAATATTATAAAAT
## views:
##      start end width
## [1]    1   3     3 [ATG]
## [2]    7   9     3 [AAT]
## [3]    8  10     3 [ATT]
## [4]   12  14     3 [ATA]
## [5]   14  16     3 [AAT]
## [6]   15  17     3 [ATA]
## [7]   17  19     3 [ATT]
## [8]   21  23     3 [ATA]
## [9]   23  25     3 [AAA]
## [10]  24  26     3 [AAT]

second_matches <- matchPattern(pattern2, first_matches, fixed = FALSE)
second_matches
## Views on a 26-letter DNASTring subject
## subject: ATGTTTAATTTATAATATTATAAAAT
## views:
##      start end width
## [1]    7   9     3 [AAT]
## [2]    8  10     3 [ATT]
## [3]   14  16     3 [AAT]
## [4]   17  19     3 [ATT]
```

```
## [5] 24 26 3 [AAT]
```

3.2 countPattern

Sometimes we don't care specifically about what the matches are, we just want to know how many matches exist. For this, we have `countPattern`. `countPattern` returns the number of matches that would be found by `matchPattern`.

```
pattern <- DNAString("ANN")
subject <- DNAString("ATGTTTAATTTATAATTTTATAAAT")

length(matchPattern(pattern, subject, fixed = FALSE))
## [1] 10

countPattern(pattern, subject, fixed = FALSE)
## [1] 10
```

3.3 One Pattern, Multiple Subjects

What if we want to search for a single pattern in multiple subjects? A common example of this is searching for a motif across multiple chromosomes.

We can accomplish this workflow using the `vcountPattern` and `vmatchPattern`. The “v” stands for “vectorized”, since it operates across all the strings in the subject like a vectorized function.

While we can only use a single pattern (for now), the subject of these functions can have multiple sequences, like a character vector or `XStringSet`:

```
pattern <- DNAString("ATNG")
subjects <- DNAStringSet(c("ATTGATTAATGGATCG", "AGTGAGACAGATTG"))

vcountPattern(pattern, subjects, fixed = FALSE)
## [1] 3 1
vcountPattern(pattern, subjects, fixed = FALSE, max.mismatch = 2)
## [1] 7 9

vmatchPattern(pattern, subjects, fixed = FALSE, max.mismatch = 1)
## MIndex object of length 2
## [[1]]
## IRanges object with 5 ranges and 0 metadata columns:
##           start      end      width
##      <integer> <integer> <integer>
## [1]         1         4         4
## [2]         5         8         4
## [3]         8        11         4
## [4]         9        12         4
## [5]        13        16         4
##
## [[2]]
## IRanges object with 3 ranges and 0 metadata columns:
```

Sequence Searching and Matching with Biostrings

```
##           start      end    width
##      <integer> <integer> <integer>
## [1]         1        4        4
## [2]         7       10        4
## [3]        11       14        4
```

`vmatchPattern` returns an object we haven't seen before: an `MIndex`. This container stores matches for a set of a patterns against one or more subject sequences. Each entry in the `MIndex` is an `IRanges` object, which is basically an `XStringViews` without the substring of the match attached (i.e., just the start/end position of each match and its width).

```
pattern <- DNAString("ATNG")
subjects <- DNAStringSet(c("ATTGATTAATGGATCG", "AGTGAGACAGATTG"))
```

```
matches <- vmatchPattern(pattern, subjects, fixed = FALSE)
```

```
## 3 matches at indices 1, 9, and 13
matches[[1]]
## IRanges object with 3 ranges and 0 metadata columns:
##           start      end    width
##      <integer> <integer> <integer>
## [1]         1        4        4
## [2]         9       12        4
## [3]        13       16        4
```

```
## Note this is the same as in matchPattern
matchPattern(pattern, subjects[[1]], fixed = FALSE)
## Views on a 16-letter DNAString subject
## subject: ATTGATTAATGGATCG
## views:
##           start end width
## [1]         1   4    4 [ATTG]
## [2]         9  12    4 [ATGG]
## [3]        13  16    4 [ATCG]
```

```
## ...and same for the second match
matches[[2]]
## IRanges object with 1 range and 0 metadata columns:
##           start      end    width
##      <integer> <integer> <integer>
## [1]        11       14        4
matchPattern(pattern, subjects[[2]], fixed = FALSE)
## Views on a 14-letter DNAString subject
## subject: AGTGAGACAGATTG
## views:
##           start end width
## [1]        11  14    4 [ATTG]
```

3.4 Summary

`matchPattern` and `countPattern` are the first “real” matching functions we’ve seen. While it’s possible to do complex matching using `isMatchingAt` and `neditAt`, the `*Pattern` functions provide true search functionality that is useful in real scenarios.

One point of discussion we’ll skip in this vignette is the `algorithm` parameter of `matchPattern`. The best practice is to just leave it set to its default value of “auto”, which will automatically pick the best pattern for your search. If a particular algorithm can be used for your search, it will perform identically to all other usable algorithms, so you don’t need to worry too much about this parameter.

We unfortunately won’t be able to do “multiple pattern, multiple sequence” searching quite yet. For that, we’ll need `PDict` objects, which we’ll talk about later.

4 Matching with PDicts

4.1 What is a PDict?

PDict is short for “Preprocessed Dictionary”. A PDict object is essentially a container that holds a collection of DNA patterns to search with. The container preprocesses these patterns to allow fast matching against one or more subject sequences.

In the standard configuration, PDicts have the following limitations:

1. They can only contain DNA base letters (i.e., `ATGC`, no ambiguity codes allowed)
2. All sequences must be the same length
3. Only exact matching is supported

We’ll relax these limitations later.

```
patterns <- DNASTringSet(c("ATGC", "GGGG", "CCCC", "ATAT"))
pdict <- PDict(patterns)
pdict
## TB_PDict object of length 4 and width 4 (preprocessing algo="ACTree2")
```

The PDict object shows how many patterns it holds (“length 4”) and how long the sequences are (“width 4”). We can also recover the sequences inside of it:

```
pdict[[1]]
## 4-letter DNASTring object
## seq: ATGC
```

We can also provide named sequences, and the PDict will keep track of them:

```
patterns <- c(seq1 = "ATGC", seq2 = "CGTA")
pdict <- PDict(patterns)
names(pdict)
## [1] "seq1" "seq2"
```

It also mentions the preprocessing algorithm, which must be one of the following:

- `ACTree2`: The default processing algorithm. Supports matching against subjects that have ambiguity codes (i.e., `A` in the PDict could match `N` in a sequence).
- `Twobit`: Much faster than `ACTree2`, but can’t match against ambiguity codes in subjects.

Sequence Searching and Matching with Biostrings

Regardless of algorithm, we can't include ambiguity codes in the patterns stored in the PDict. However, we can get around this and unlock the full power of PDicts by using a "Trusted Band".

4.2 Exact Matching

Once we have a PDict, we can start matching. These functions are very similar to the `*Pattern` functions:

- `matchPDict` is equivalent to `matchPattern`
- `countPDict` is equivalent to `countPattern`
- `vmatchPDict` is similar to `vmatchPattern`
- `vcountPDict` is equivalent to `vcountPattern`

These enable us to do "many pattern, single subject" matching:

```
patterns <- DNASTringSet(c(seq1 = "ATGC", seq2 = "AATT", seq3 = "CCTA"))
pdict <- PDict(patterns)
subject <- DNASTring("ATGCAAAATTTAATTGC")

## How many matches does each pattern have?
countPDict(pdict, subject)
## [1] 1 2 0

## Where are the matches?
matchPDict(pdict, subject)
## MIndex object of length 3
## $seq1
## IRanges object with 1 range and 0 metadata columns:
##      start      end  width
##   <integer> <integer> <integer>
##   [1]      1      4      4
##
## $seq2
## IRanges object with 2 ranges and 0 metadata columns:
##      start      end  width
##   <integer> <integer> <integer>
##   [1]      7     10      4
##   [2]     12     15      4
##
## $seq3
## IRanges object with 0 ranges and 0 metadata columns:
##      start      end  width
##   <integer> <integer> <integer>
```

We also have `whichPDict`, which returns a vector indicating which patterns have at least one match:

```
## only the first and second patterns have matches
whichPDict(pdict, subject)
## [1] 1 2
```

Sequence Searching and Matching with Biostrings

We can also do “many pattern, many subject” matching with the `v*PDict` functions. However, the output types of these are different than previously. Let’s break them down one by one.

First, `vwhichPDict`:

```
subjects <- DNASTringSet(c(subj1 = "ATGCAAAATTTAATTGC", subj2 = "CCAATCCTACTATGC"))

vwhichPDict(pdct, subjects)
## [[1]]
## [1] 1 2
##
## [[2]]
## [1] 1 3
```

This returns a list with the output of calling `whichPDict` on each sequence in the subject. For this input, we find that patterns 1 and 2 have matches in the first sequence, and patterns 1 and 3 have matches in the second sequence.

If we wanted to find out how many matches each pattern has in each sequence, we’d use `vcountPDict`:

```
vcountPDict(pdct, subjects)
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    0
## [3,]    0    1
```

This function returns a matrix of counts. The rows correspond to patterns, and the columns correspond to subjects. Thus, since `vcountPDict(pdct, subjects)[2,1] = 2`, we know that pattern 2 has two matches in subject 1.

If we wanted to find out where those matches were, we’d use `vmatchPDict`:

```
vmatchPDict(pdct, subjects)
## MIndexList of length 2
## [1:"subj1"] 3 patterns, 3 matches
## [1:"subj2"] 3 patterns, 2 matches
```

The result of this is an `MIndexList` class. This is a similar concept to what we saw with the `MIndex` class when we discussed `matchPattern`. The `MIndex` class is essentially a list storing offsets of matches for a set of patterns against a subject, and the `MIndexList` class is a list storing an `MIndex` object for each subject.

```
all_matches <- vmatchPDict(pdct, subjects)

## first entry is the matches for all patterns against the first subject:
all_matches[[1]]
## MIndex object of length 3
## $seq1
## IRanges object with 1 range and 0 metadata columns:
##      start      end      width
##      <integer> <integer> <integer>
## [1]      1      4      4
##
```

Sequence Searching and Matching with Biostrings

```
## $seq2
## IRanges object with 2 ranges and 0 metadata columns:
##      start      end      width
##      <integer> <integer> <integer>
## [1]      7      10      4
## [2]     12     15      4
##
## $seq3
## IRanges object with 0 ranges and 0 metadata columns:
##      start      end      width
##      <integer> <integer> <integer>
```

Note that the last entry of `all_matches[[1]]` is empty, because the third pattern had no matches against the first subject.

Note: `vmatchPDict` is the *only* function that supports “many pattern, many subject” matching. `vmatchPattern` can only match a single pattern against one or more sequences.

4.3 Trusted Bands

PDicts in their default configuration are very limited. However, this changes when we define a “Trusted Band”. PDicts with a trusted band support ambiguity codes, variable length sequences, and fuzzy/inexact matching.

The trusted band is a fixed size subsequence of the patterns that obeys the prior limitations (no ambiguities, fixed size). When we do matching, the PDict can first search for matches in the trusted band to limit the search space, and then expand to the regions of each pattern outside the trusted band.

The trusted bands do not need to all be in the same position, but they all need to be the same length. For example, we could imagine the following sequences with trusted bands:

```
Sequences:
-----
TTGCTAMKT
AGCTAGTTAAG

Some Possible Trusted Bands:
-----
T | TGCT | AMKT
A | GCTA | GTTAAG

    T | TGCTA | MKT
AGC | TAGTT | AAG
```

Notice how the bands are all the same width and contain no ambiguity codes, but they don't have to all be in the same position in each sequence. In R, we could do this with the following:

```
patterns <- DNASTringSet(c("TTGCTAMKT", "AGCTAGTTAAG"))
pdict1 <- PDict(patterns, tb.start = 2, tb.width = 4)
pdict1
## TB_PDict object of length 2 and variable width (preprocessing algo="ACtree2"):
## - with a head of width 1
```

Sequence Searching and Matching with Biostrings

```
## - with a Trusted Band of width 4
## - with a tail of variable width (min=4 / max=6)

pdict2 <- PDict(patterns, tb.end = -4, tb.width = 5)
pdict2
## TB_PDict object of length 2 and variable width (preprocessing algo="ACtree2"):
## - with a head of variable width (min=1 / max=3)
## - with a Trusted Band of width 5
## - with a tail of width 3
```

We can either define the trusted band relative to the start (`tb.start`), or relative to the end (`tb.end`). Negative numbers are indices relative to the end of the sequence, so `-4` means “the fourth base counting from the end of the sequence”.

The trusted band splits the sequences into three parts: the head (before the band), the band, and the tail (part after the band). We can extract these to look at them:

```
head(pdict2)
## DNAStringSet object of length 2:
##      width seq
## [1]      1 T
## [2]      3 AGC
tb(pdict2)
## DNAStringSet object of length 2:
##      width seq
## [1]      5 TGCTA
## [2]      5 TAGTT
tail(pdict2)
## DNAStringSet object of length 2:
##      width seq
## [1]      3 MKT
## [2]      3 AAG
```

4.4 Inexact Matching

Given a PDict with a trusted band, we can start to do inexact matching. Inexact matching with a PDict that has a trusted band works in the following steps:

1. Find all exact matches using the trusted band of each pattern against the subjects.
2. For each match, compare the head and tail of the pattern against the regions of the subject flanking the match.

This means that an exact match has to be found to the trusted band before any inexact matching takes place. This is a key distinction, and can lead to some unintuitive behavior. For example, suppose we make two trusted bands from a given sequence:

Seq: AATTGGC

Trusted Band 1: AA | TTGG | C

Trusted Band 2: A | ATTG | GC

The trusted band is the center section, and the left and right regions are the head / tail. Now suppose we want to match against a target sequence with at most one mismatch:

Sequence Searching and Matching with Biostrings

```
Subject: ACTTGGC

Goal: Match with at most 1 mismatch

Trusted Band 1 finds a match:
AA | TTGG | C
AC  TTGG  C

Trusted Band 2 finds NO match.
A | ATTG | GC
A  CTTG  GC
(One mismatch, but no exact match in trusted band)
```

Because of this, the trusted band should be defined as the minimal match that must exist for a match to be correct.

To do this in R:

```
seq <- DNASTringSet(c("AATTGGC"))

# A | ATTG | GC
pdict <- PDict(seq, tb.start=2, tb.end=5)

subjects <- DNASTringSet(c("AATTGGCC", "CATTGGCT", "ACTTGGCG"))

# Pattern: A | ATTG | GC
# Seq 1: A  ATTG  GCC  (matches)
# Seq 2: C  ATTG  GCT  (matches, one mismatch)
# Seq 3: A  CTTG  GCG  (no exact match in TB)
vcountPDict(pdict, subjects, max.mismatch = 1L)
##      [,1] [,2] [,3]
## [1,]    1    1    0
```

The use of the trusted band allows us to use the other fuzzy matching techniques we used with the `*Pattern` functions.

We can match ambiguity codes using `fixed`:

```
pattern <- DNASTringSet("ATNC")

# | AT | NC
pdict <- PDict(pattern, tb.start = 1, tb.end = 2)

subject <- DNASTring("CCAATATACATNC")

## Only one match, since ambiguities are treated literally
matchPDict(pdict, subject)[[1]]
## IRanges object with 1 range and 0 metadata columns:
##      start      end      width
##      <integer> <integer> <integer>
## [1]        10        13         4

## Two matches; N in tail can match any nucleotide
```

Sequence Searching and Matching with Biostrings

```
matchPDict(pdDict, subject, fixed=FALSE)[[1]]
## IRanges object with 2 ranges and 0 metadata columns:
##      start      end      width
##      <integer> <integer> <integer>
## [1]      6      9      4
## [2]     10     13      4

## Since N matches anything, this matches any AT** pattern
matchPDict(pdDict, subject, fixed=FALSE, max.mismatch = 1)[[1]]
## IRanges object with 3 ranges and 0 metadata columns:
##      start      end      width
##      <integer> <integer> <integer>
## [1]      4      7      4
## [2]      6      9      4
## [3]     10     13      4
```

4.5 PDict Matching Without a PDict

It's possible to use the `*PDict` functions without a PDict, by just passing in a `DNAStrSet`. This is slightly slower than using a preprocessed dictionary, but will be slightly faster than using `matchPattern` in a loop, since `matchPDict` loops at the C level. Additionally, the preprocessing restrictions for PDicts *only apply to initializing the object*, so it's possible to call `matchPDict` for a variable width sequence set if the sequences aren't preprocessed:

```
## Patterns are all different lengths!
patterns <- DNAStrSet(c("AA", "ATGC", "TTGGCC", "T"))
subject <- DNAStr("AATTGGCCAATGC")
countPDict(patterns, subject)
## [1] 2 1 1 3
```

This is also the only way to use the `with.indels = TRUE` argument.

4.6 Performance Notes

Inexact matching can become extremely slow for large numbers of subjects and patterns, especially if the trusted band is small. The most time consuming part is the inexact matching of the head/tail with the flanking regions of the subject's matches. Larger trusted bands mean fewer exact matches, which means better performance. The Biostrings documentation recommends aiming for $\frac{L}{4^w} < 10$, where L is the number of letters in the subject, and w is the width of the trusted band. Larger trusted bands will always result in fewer, faster matches, so prioritize making it as large as you confidently can to get the matches you're looking for.

5 Other Matching Utilities

Biostrings also includes a couple general-purpose helper functions that operate on matches returned by matching functions (e.g., `matchPattern`, `matchPDict`).

5.1 Finding Mismatches

Fuzzy / inexact matching can match with some mismatches. It's often useful to figure out which letters were mismatching in an inexact match call. For this, we have the `mismatch` function, which provides the position of mismatching letters of a given pattern relative to its matches in the subject:

```
subject <- DNASTring("ATGCCGTA")
pattern <- DNASTring("GCGGAA")

matches <- matchPattern(pattern, subject, max.mismatch = 2)
matches
## Views on a 8-letter DNASTring subject
## subject: ATGCCGTA
## views:
##      start end width
## [1]      3   8     6 [GCCGTA]

## the pattern matched positions 3-8 of the subject
## pattern:  GCGGAA
## subject: ATGCCGTA
## mismatches: | | (positions 3 and 5 of the match)
mismatch(pattern = pattern, x = matches)
## [[1]]
## [1] 3 5

## How many mismatches were there overall?
nmismatch(pattern = pattern, x = matches)
## [1] 2
```

If we're matching using ambiguity codes, we can also use the `fixed` argument to treat ambiguities in the subject as ambiguities (and not literal matches)

```
subject <- DNASTring("ATNGTNCTN")
pattern <- DNASTring("ATC")

matches <- matchPattern(pattern, subject, fixed = FALSE, max.mismatch = 1L)
matches
## Views on a 9-letter DNASTring subject
## subject: ATNGTNCTN
## views:
##      start end width
## [1]      1   3     3 [ATN]
## [2]      4   6     3 [GTN]
## [3]      5   7     3 [TNC]
## [4]      7   9     3 [CTN]

## mismatches if we ignored ambiguities in subject
mismatch(pattern, matches)
## [[1]]
## [1] 3
##
```

Sequence Searching and Matching with Biostrings

```
## [[2]]
## [1] 1 3
##
## [[3]]
## [1] 1 2
##
## [[4]]
## [1] 1 3

## mismatches if we use ambiguities in subject
mismatch(pattern, matches, fixed = FALSE)
## [[1]]
## integer(0)
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 1
##
## [[4]]
## [1] 1
```

5.2 Finding Coverage

Given a set of matches on a subject, we may also be interested in determining how many matches we got at each position. In other words, how well does the pattern “cover” the sequence?

Let’s use the same example from the previous section, including ambiguities:

```
subject <- DNASTring("ATNGTNCTN")
pattern <- DNASTring("ATC")

matches <- matchPattern(pattern, subject, fixed = FALSE, max.mismatch = 1L)
matches
## Views on a 9-letter DNASTring subject
## subject: ATNGTNCTN
## views:
##      start end width
## [1]    1   3     3 [ATN]
## [2]    4   6     3 [GTN]
## [3]    5   7     3 [TNC]
## [4]    7   9     3 [CTN]
```

We can imagine overlaying the regions that match the pattern:

```
Subject:  A T N G T N C T N
Match 1:  - - -
Match 2:      - - -
Match 3:          - - -
```

Sequence Searching and Matching with Biostrings

```
Match 4:      - - -
```

If we then count how many matches we have for each position, we get the coverage:

```
Subject:  A T N G T N C T N
Match 1:  - - -
Match 2:      - - -
Match 3:      - - -
Match 4:      - - -
Coverage: 1 1 1 1 2 2 2 1 1
```

In code, this looks like:

```
subject <- DNAString("ATNGTNCTN")
pattern <- DNAString("ATC")

matches <- matchPattern(pattern, subject, fixed = FALSE, max.mismatch = 1L)

## This is a run-length encoding
coverage(matches)
## integer-Rle of length 9 with 3 runs
##   Lengths: 4 3 2
##   Values  : 1 2 1

## converting to a vector, we get:
as.vector(coverage(matches))
## [1] 1 1 1 1 2 2 2 1 1
```

If we're matching multiple patterns against a subject, `coverage` will give us the number of matches from *any* pattern for each position:

```
patterns <- DNAStringSet(c("ATC", "ATG"))
pd <- PDict(patterns, tb.start = 1, tb.end = 2)
subject <- DNAString("ATATATATATGAC")

countPDict(pd, subject, max.mismatch = 1L)
## [1] 5 5

matches <- matchPDict(pd, subject, max.mismatch = 1L)
as.vector(coverage(matches))
## [1] 2 2 4 2 4 2 4 2 4 2 2
```

6 Biological Applications

The previous sections have covered everything you need to know for matching sequences with Biostrings. In this last section, we'll cover some of the more application-specific functions you can use for biological analyses.

6.1 Finding Palindromes

Finding palindromic sequences is a critical analysis for many analyses. Biostrings provides the `findPalindromes` function to identify palindromic regions in a sequence.

By palindromic sequences, we mean two regions of a sequence that are reverse complements of each other, potentially separated by a non-matching region. These palindromic sequences can form hairpin loops, like so:

Palindromic Sequence: GAATTCAATCTAAGAATTC

This forms a hairpin:

```
      A A
    G A A T T C   T
    | | | | |   C
    C T T A A G   T
      A A
```

This palindrome has three regions: a left arm, a right arm, and a loop. The arms are the two regions that match, with left / right corresponding to which comes first / last in the sequence (respectively). The loop is the middle region that doesn't bind to itself.

In the previous example, we'd see the following regions:

Palindromic Sequence: GAATTCAATCTAAGAATTC

Separated into groups: GAATTC AATCTAA GAATTC

```
      /      |      \
    left arm  loop  right arm
```

We can find this with the `findPalindromes` function in Biostrings:

```
subject <- DNAString("GAATTCAATCTAAGAATTC")
findPalindromes(subject, max.looplevelength = 7)
## Views on a 19-letter DNAString subject
## subject: GAATTCAATCTAAGAATTC
## views:
##      start end width
## [1]      1  19    19 [GAATTCAATCTAAGAATTC]
```

In this case, we get a single view on the whole string, since the entire sequence is a palindrome. Note that we set `max.looplevelength` to control the maximum size of loop we look for. We can also set `min.armlength` and `max.armlength` to control the size of the arms, and `min.looplevelength` to control the size of the loop.

If we have a sequence with multiple palindromes, we can find them all at once:

```
subject <- DNAString("GAAGTTCTTCATGCATGTAACATG")

# Default parameters:
# - min.armlength = 4
# - max.looplevelength = 1 (strict palindromes only)
findPalindromes(subject, max.looplevelength = 3L)
## Views on a 24-letter DNAString subject
```

Sequence Searching and Matching with Biostrings

```
## subject: GAAGTTCTTCATGCATGTAACATG
## views:
##      start end width
## [1]    1  10    10 [GAAGTTCTTC]
## [2]   10  17     8 [CATGCATG]
## [3]   14  24    11 [CATGTAACATG]
```

Now we have three palindromes! However, it's a little difficult to tell what the palindromic regions are. We can identify the left/right arms with some helper functions:

```
subject <- DNASTring("GAAGTTCTTCATGCATGTAACATG")
palindromes <- findPalindromes(subject, max.looplevelength = 3L)

## lengths of all arms
palindromeArmLength(palindromes)
## [1] 4 4 5

## Get only the left arm
palindromeLeftArm(palindromes)
## Views on a 24-letter DNASTring subject
## subject: GAAGTTCTTCATGCATGTAACATG
## views:
##      start end width
## [1]    1   4     4 [GAAG]
## [2]   10  13     4 [CATG]
## [3]   14  18     5 [CATGT]

## Get only the right arm
palindromeRightArm(palindromes)
## Views on a 24-letter DNASTring subject
## subject: GAAGTTCTTCATGCATGTAACATG
## views:
##      start end width
## [1]    7  10     4 [CTTC]
## [2]   14  17     4 [CATG]
## [3]   20  24     5 [ACATG]
```

We can also set `allow.wobble` to `TRUE` to treat G-T or G-U pairings as matches:

```
subject <- DNASTring("ATGCGTAT")

## No matches, because G-T pairings are ignored
findPalindromes(subject)
## Views on a 8-letter DNASTring subject
## subject: ATGCGTAT
## views: NONE

## Allowing wobble base pairs, we find a match
findPalindromes(subject, allow.wobble = TRUE)
## Views on a 8-letter DNASTring subject
## subject: ATGCGTAT
## views:
```

Sequence Searching and Matching with Biostrings

```
##      start end width
## [1]    1   8     8 [ATGCGTAT]

## We can also just allow mismatches directly, which aren't limited to G-T pairs
findPalindromes(subject, max.mismatch = 1L)
## Views on a 8-letter DNAString subject
## subject: ATGCGTAT
## views:
##      start end width
## [1]    1   8     8 [ATGCGTAT]
```

`findPalindromes` will search for reverse complement sequences if the input is a `DNAString` or `RNAString`. If the input is a `BString` or `AAString`, it'll just search for regular palindromes:

```
subject <- BString("amanaplanacanalpanama racecar momomomom")
findPalindromes(subject)
## Views on a 39-letter BString subject
## subject: amanaplanacanalpanama racecar momomomom
## views:
##      start end width
## [1]    1  21    21 [amanaplanacanalpanama]
## [2]   22  30     9 [ racecar ]
## [3]   31  39     9 [momomomom]
```

6.2 Finding Amplicons for a Primer Pair

Another common workflow is identifying amplicons that match to a given pair of primers. For this, we have the `matchProbePair` function, which takes as input a pair (forward and reverse) of primers and a target sequence, and returns all the amplicons mapped to it.

Let's look at an example, using a primer pair derived directly from a yeast chromosome.

```
options(width = 70) # limiting the width of XString output
data("yeastSEQCHR1") # bundled with the Biostrings package
yeast_chr1 <- DNAString(yeastSEQCHR1)

## We'll target 500 bases starting at position 50,000
region_start <- 50000L
region_width <- 500L
probe_size <- 20L

## Getting some subsequences to act as primers
forward_probe <- subseq(yeast_chr1, start = region_start, width = probe_size)
reverse_probe <- subseq(
  yeast_chr1,
  end = region_start + region_width - 1,
  width = probe_size
)

## The reverse probe should be the reverse complement
reverse_probe <- reverseComplement(reverse_probe)
```

Sequence Searching and Matching with Biostrings

```
## Now we can identify the amplicons
matchProbePair(forward_probe, reverse_probe, subject = yeast_chr1)
## Views on a 230208-letter DNAString subject
## subject: CCACACCACACCCACACCCACACACCA...GGGTGTGGTGTGGGTGTGGTGTGTGTGGG
## views:
##      start   end width
## [1] 50000 50499   500 [AGAGGGTGATTCTTTCTCGA...CTTTCCCATAGTACCATTAA]
```

Under the hood, `matchProbePair` uses `matchPattern`, which is why it returns an `XStringViews` object. That means it supports all the features of `matchPattern`:

```
options(width = 70) # limiting the width of XString output

## Including mismatches
matchProbePair(
  forward_probe,
  reverse_probe,
  yeast_chr1,
  max.mismatch = 5
)
## Views on a 230208-letter DNAString subject
## subject: CCACACCACACCCACACCCACACACCA...GGGTGTGGTGTGGGTGTGGTGTGTGTGGG
## views:
##      start   end width
## [1] 50000 50499   500 [AGAGGGTGATTCTTTCTCG...CTTTCCCATAGTACCATTAA]
## [2] 85724 87210  1487 [AGAGCCTTATCTCTTTCTAG...CTTTCCACAGCATCTATAA]
## [3] 91467 134389 42923 [ATATGGTAATTTCTTTCTGT...CCTTCCATAATAGTTTAA]

## Not sure why you'd want to do this, but you can!
matchProbePair(
  forward_probe,
  reverse_probe,
  yeast_chr1,
  min.mismatch = 6,
  max.mismatch = 8
)
## Views on a 230208-letter DNAString subject
## subject: CCACACCACACCCACACCCACACACCA...GGGTGTGGTGTGGGTGTGGTGTGTGTGGG
## views:
##      start   end width
## [1]    603    834   232 [ACATTTTGATATCTATATC...CTATCATGGTATCACTAA]
## [2]   3085   3419   335 [AGACTGCCAAATTTTCTT...TTTTCAAAGAACTTCAA]
## [3]   4140   4257   118 [AGAGAGTACTTTTTTTGGA...CTAAAAAATTCAGTGTAT]
## [4]   5089   5337   249 [TGAAGATTGTATATGAAA...TCTAATCAGTTCATTAA]
## [5]   6956   7097   142 [TTTATAGTAGTAGTGTGAA...AGAAAGATATAAAAAATTA]
## ...     ...     ...
## [310] 227170 227460   291 [GTAAAAAGTATGAAGTGAAA...TTTCCCATTTAACATTTA]
## [311] 227687 227742    56 [TCAAAATAATATGTGCAA...CGAAAGCAATGGCCGCCA]
## [312] 228251 228324    74 [TGAGGACAACCTTTTACTC...CACAAGGTATCACATTAT]
## [313] 229121 229121     1 [A]
## [314] 229186 229328   143 [TTAACTTTACTAAAGGAAA...GTAAATGGGATCATTAA]
```

6.3 Position Weight Matrix (PWM) Matching

We can also find matches in sequences using position weight matrices (PWMs). This functions exactly like all the other matching functions we've covered, except we'll use a PWM as the pattern rather than an XString directly. We can create a PWM with the `PWM` function:

```
## Some possible TATA box binding motifs
motifs <- DNASTringSet(c("TATAAA", "TATATA", "TATAAT", "TATATT"))

## Converting into a PWM
pwm <- PWM(motifs)
pwm
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## A 0.0000000 0.1801462 0.0000000 0.1801462 0.1397077 0.1397077
## C 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## G 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## T 0.1801462 0.0000000 0.1801462 0.0000000 0.1397077 0.1397077
```

Now we can use our `countPWM` and `matchPWM` just like we would with a pattern, we're just going to use the PWM as our pattern:

```
options(width = 70) # limiting the width of XString output
data("yeastSEQCHR1") # bundled with the Biostrings package
yeast_chr1 <- DNASTring(yeastSEQCHR1)

## Getting the number of matching motifs
countPWM(pwm, yeast_chr1)
## [1] 7085

## Finding the matches
matchPWM(pwm, yeast_chr1)
## Views on a 230208-letter DNASTring subject
## subject: CCACACCACACCCACACCCACACACCA...GGGTGTGGTGTGGGTGTGGTGTGTGGG
## views:
##           start    end width
##      [1]    316    321     6 [CATATT]
##      [2]    342    347     6 [TAAATA]
##      [3]    372    377     6 [TTTATA]
##      [4]    408    413     6 [TGTATA]
##      [5]    410    415     6 [TATACT]
##      ...      ...      ...      ...
## [7081] 229813 229818     6 [TATCAA]
## [7082] 229826 229831     6 [TATGTA]
## [7083] 229908 229913     6 [GATATA]
## [7084] 229910 229915     6 [TATATT]
## [7085] 230071 230076     6 [TAGAAT]
```

As you can see, there are tons of matches. First, let's get an idea of how well each of these matches scored:

```
## finding matches with score included
matches <- matchPWM(pwm, yeast_chr1, with.score = TRUE)
```

Sequence Searching and Matching with Biostrings

```
## `mcols` references the metadata columns
all_scores <- mcols(matches)$score
table(round(all_scores, 2))
##
## 0.82 0.86    1
## 5867 637 581
```

The high scores are because we have such a small motif, but we do have some subsequences that scored higher than others. Let's trim out the lower scoring sequences:

```
## Way too many matches
countPWM(pwm, yeast_chr1)
## [1] 7085

## ...but we can limit the matches with min.score
countPWM(pwm, yeast_chr1, min.score = "86%")
## [1] 1218

## Score can also be expressed as a decimal value
countPWM(pwm, yeast_chr1, min.score = 0.95)
## [1] 581
```

There are a lot of features with the `*PWM` functions. You can use logged probability ratios (the default) or raw probabilities for scores, set priors for the estimated probability of bases at each position, and renormalize your PWMs to a unit scale. Check out `?PWM` for more details on these features.

7 Further Reading

If you're interested in diving more into the details of matching in Biostrings, check out these man pages:

- `?lowlevel-matching`: low-level matching
- `?matchPattern`
- `?PDict`
- `?matchPDict`
- `?matchPDict-inexact`
- `?match-utils`

Matching in Biostrings is still a work in progress, and there are many features we're still working on. Some of them are even noted in the man pages. If you're interested in contributing to Biostrings, feel free to file a [pull request](#)!

8 Session info

```
## R version 4.6.1 (2026-06-24)
## Platform: aarch64-apple-darwin23
## Running under: macOS Tahoe 26.3.1
##
## Matrix products: default
```

Sequence Searching and Matching with Biostrings

```
## BLAS: /Library/Frameworks/R.framework/Versions/4.6/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.6/Resources/lib/libRlapack.dylib; LAPACK version 3.12
##
## locale:
## [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/New_York
## tzcode source: internal
##
## attached base packages:
## [1] stats4 stats graphics grDevices utils datasets methods
## [8] base
##
## other attached packages:
## [1] Biostrings_2.81.5 Seqinfo_1.3.0 XVector_0.53.0
## [4] IRanges_2.47.2 S4Vectors_0.51.5 BiocGenerics_0.59.9
## [7] generics_0.1.4 BiocStyle_2.41.0
##
## loaded via a namespace (and not attached):
## [1] crayon_1.5.3 cli_3.6.6 knitr_1.51
## [4] rlang_1.3.0 xfun_0.59 otel_0.2.0
## [7] jsonlite_2.0.0 htmltools_0.5.9 sass_0.4.10
## [10] rmarkdown_2.31 evaluate_1.0.5 jquerylib_0.1.4
## [13] fastmap_1.2.0 yaml_2.3.12 lifecycle_1.0.5
## [16] bookdown_0.47 BiocManager_1.30.27 compiler_4.6.1
## [19] digest_0.6.39 R6_2.6.1 bslib_0.11.0
## [22] tools_4.6.1 cachem_1.1.0
```