

Pairwise Sequence Alignments

Patrick Aboyoun
Gentleman Lab
Fred Hutchinson Cancer Research Center
Seattle, WA

April 2, 2009

Contents

1	Introduction	2
2	Pairwise Sequence Alignment Problems	2
3	Main Pairwise Sequence Alignment Function	3
3.1	Exercise 1	5
4	Pairwise Sequence Alignment Classes	5
4.1	Exercise 2	6
5	Pairwise Sequence Alignment Helper Functions	6
5.1	Exercise 3	10
6	Edit Distances	10
6.1	Exercise 4	11
7	Application: Using Evolutionary Models in Protein Alignments	12
7.1	Exercise 5	12
8	Application: Removing Adapters from Sequence Reads	12
8.1	Exercise 6	18
9	Application: Quality Assurance in Sequencing Experiments	18
9.1	Exercise 7	21
10	Computation Profiling	21
10.1	Exercise 8	24
11	Computing alignment consensus matrices	24
12	Exercise Answers	25
12.1	Exercise 1	25
12.2	Exercise 2	25
12.3	Exercise 3	26
12.4	Exercise 4	27
12.5	Exercise 5	27

12.6 Exercise 6	27
12.7 Exercise 7	31
12.8 Exercise 8	33

13 Session Information **35**

1 Introduction

In this document we illustrate how to perform pairwise sequence alignments using the `Biostrings` package through the use of the `pairwiseAlignment` function. This function aligns a set of *pattern* strings to a *subject* string in a global, local, or overlap (ends-free) fashion with or without affine gaps using either a fixed or quality-based substitution scoring scheme. This function’s computation time is proportional to the product of the two string lengths being aligned.

2 Pairwise Sequence Alignment Problems

The (Needleman-Wunsch) global, the (Smith-Waterman) local, and (ends-free) overlap pairwise sequence alignment problems are described as follows. Let string S_i have n_i characters $c_{(i,j)}$ with $j \in \{1, \dots, n_i\}$. A pairwise sequence alignment is a mapping of strings S_1 and S_2 to “gapped” substrings S'_1 and S'_2 that are defined by

$$\begin{aligned} S'_1 &= g_{(1,a_1)}c_{(1,a_1)} \cdots g_{(1,b_1)}c_{(1,b_1)}g_{(1,b_1+1)} \\ S'_2 &= g_{(1,a_2)}c_{(1,a_2)} \cdots g_{(1,b_2)}c_{(1,b_2)}g_{(1,b_2+1)} \end{aligned}$$

where

$$\begin{aligned} a_i, b_i &\in \{1, \dots, n_i\} \text{ with } a_i \leq b_i \\ g_{(i,j)} &= 0 \text{ or more gaps at the specified position } j \text{ for aligned string } i \\ length(S'_1) &= length(S'_2) \end{aligned}$$

Each of these pairwise sequence alignment problems is solved by maximizing the alignment *score*. An alignment score is determined by the type of pairwise sequence alignment (global, local, overlap), which sets the $[a_i, b_i]$ ranges for the substrings; the substitution scoring scheme, which sets the distance between aligned characters; and the gap penalties, which is divided into opening and extension components. The optimal pairwise sequence alignment is the pairwise sequence alignment with the largest score for the specified alignment type, substitution scoring scheme, and gap penalties. The pairwise sequence alignment types, substitution scoring schemes, and gap penalties influence alignment scores in the following manner:

Pairwise Sequence Alignment Types: The type of pairwise sequence alignment determines the substring ranges to apply the substitution scoring and gap penalty schemes. For the three primary (global, local, overlap) and two derivative (subject overlap, pattern overlap) pairwise sequence alignment types, the resulting substring ranges are as follows:

- Global - $[a_1, b_1] = [1, n_1]$ and $[a_2, b_2] = [1, n_2]$
- Local - $[a_1, b_1]$ and $[a_2, b_2]$
- Overlap - $\{[a_1, b_1] = [a_1, n_1], [a_2, b_2] = [1, b_2]\}$ or $\{[a_1, b_1] = [1, b_1], [a_2, b_2] = [a_2, n_2]\}$
- Subject Overlap - $[a_1, b_1] = [1, n_1]$ and $[a_2, b_2]$
- Pattern Overlap - $[a_1, b_1]$ and $[a_2, b_2] = [1, n_2]$

Substitution Scoring Schemes: The substitution scoring scheme sets the values for the aligned character pairings within the substring ranges determined by the type of pairwise sequence alignment. This scoring scheme can be fixed for character pairings or quality-dependent for character pairings. (Characters that align with a gap are penalized according to the “Gap Penalty” framework.)

Fixed substitution scoring - Fixed substitution scoring schemes associate each aligned character pairing with a value. These schemes are very common and include awarding one value for a match and another for a mismatch, Point Accepted Mutation (PAM) matrices, and Block Substitution Matrix (BLOSUM) matrices.

Quality-based substitution scoring - Quality-based substitution scoring schemes derive the value for the aligned character pairing based on the probabilities of character recording errors (3). Let ϵ_i be the probability of a character recording error. Assuming independence within and between recordings and a uniform background frequency of the different characters, the combined error probability of a mismatch when the underlying characters do match is $\epsilon_c = \epsilon_1 + \epsilon_2 - (n/(n-1)) * \epsilon_1 * \epsilon_2$, where n is the number of characters in the underlying alphabet. Using ϵ_c , the substitution score is given by $b * \log_2(\gamma_{(x,y)} * (1 - \epsilon_c) * n + (1 - \gamma_{(x,y)}) * \epsilon_c * (n/(n-1)))$, where b is the bit-scaling for the scoring and $\gamma_{(x,y)}$ is the probability that characters x and y represents the same underlying information (e.g. using IUPAC, $\gamma_{(A,A)} = 1$ and $\gamma_{(A,N)} = 1/4$).

Gap Penalties: Gap penalties are the values associated with the gaps within the substring ranges determined by the type of pairwise sequence alignment. These penalties are divided into *gap opening* and *gap extension* components, where the gap opening penalty is the cost for adding a new gap and the gap extension penalty is the incremental cost incurred along the length of the gap. A *constant gap penalty* occurs when there is a cost associated with opening a gap, but no cost for the length of a gap (i.e. gap extension is zero). A *linear gap penalty* occurs when there is no cost associated for opening a gap (i.e. gap opening is zero), but there is a cost for the length of the gap. An *affine gap penalty* occurs when both the gap opening and gap extension have a non-zero associated cost.

3 Main Pairwise Sequence Alignment Function

The `pairwiseAlignment` function solves the pairwise sequence alignment problems mentioned above. It aligns one or more strings specified in the *pattern* argument with a single string specified in the *subject* argument.

```
> library(Biostrings)
> pairwiseAlignment(pattern = c("succeed", "precede"), subject = "supersede")
```

```
Global Pairwise Alignment (1 of 2)
pattern: [1] succe--ed
subject: [1] sup-ersed
score: -12.39849
```

The type of pairwise sequence alignment is set by specifying the *type* argument to be one of "global", "local", "overlap", "subjectOverlap", and "patternOverlap".

```
> pairwiseAlignment(pattern = c("succeed", "precede"), subject = "supersede",
+   type = "local")
```

```
Local Pairwise Alignment (1 of 2)
pattern: [1] su
subject: [1] su
score: 15.96347
```

The gap penalties are regulated by the *gapOpening* and *gapExtension* arguments.

```
> pairwiseAlignment(pattern = c("succeed", "precede"), subject = "supersede",
+   gapOpening = 0, gapExtension = -1)
```

Global Pairwise Alignment (1 of 2)

```
pattern: [1] su-cce--ed
subject: [1] sup--ersed
score: 33.90868
```

The substitution scoring scheme is set using three arguments, two of which are quality-based related (*patternQuality*, *subjectQuality*) and one is fixed substitution related (*substitutionMatrix*). When the substitution scores are fixed by character pairing, the *substitutionMatrix* argument takes a matrix with the appropriate alphabets as dimension names. The *nucleotideSubstitutionMatrix* function translates simple match and mismatch scores to the full spectrum of IUPAC nucleotide codes.

```
> submat <- matrix(-1, nrow = 26, ncol = 26, dimnames = list(letters,
+   letters))
> diag(submat) <- 0
> pairwiseAlignment(pattern = c("succeed", "precede"), subject = "supersede",
+   substitutionMatrix = submat, gapOpening = 0, gapExtension = -1)
```

Global Pairwise Alignment (1 of 2)

```
pattern: [1] succe-ed
subject: [1] supersed
score: -5
```

When the substitution scores are quality-based, the *patternQuality* and *subjectQuality* arguments represent the equivalent of $[x - 99]$ numeric quality values for the respective strings, and the optional *fuzzyMatrix* argument represents how the closely two characters match on a $[0, 1]$ scale. The *patternQuality* and *subjectQuality* arguments accept quality measures in either a *PhredQuality* or *SolexaQuality* scaling. For *PhredQuality* measures $Q \in [0, 99]$, the probability of an error in the base read is given by $10^{-Q/10}$ and for *SolexaQuality* measures $Q \in [-5, 99]$, they are given by $1 - 1/(1 + 10^{-Q/10})$. The *qualitySubstitutionMatrices* function maps the *patternQuality* and *subjectQuality* scores to match and mismatch penalties. These three arguments will be demonstrated in later sections.

The final argument, *scoreOnly*, to the *pairwiseAlignment* function accepts a logical value to specify whether or not to return just the pairwise sequence alignment score. If *scoreOnly* is FALSE, the pairwise alignment with the maximum alignment score is returned. If more than one pairwise alignment has the maximum alignment score exists, the first alignment along the subject is returned. If there are multiple pairwise alignments with the maximum alignment score at the chosen subject location, then at each location along the alignment mismatches are given preference to insertions/deletions. For example, **pattern:** [1] ATTA; **subject:** [1] AT-A is chosen above **pattern:** [1] ATTA; **subject:** [1] A-TA if they both have the maximum alignment score.

```
> submat <- matrix(-1, nrow = 26, ncol = 26, dimnames = list(letters,
+   letters))
> diag(submat) <- 0
> pairwiseAlignment(pattern = c("succeed", "precede"), subject = "supersede",
+   substitutionMatrix = submat, gapOpening = 0, gapExtension = -1,
+   scoreOnly = TRUE)
```

```
[1] -5 -5
```

3.1 Exercise 1

1. Using `pairwiseAlignment`, fit the global, local, and overlap pairwise sequence alignment of the strings "syzygy" and "zyzzyx" using the default settings.
2. Do any of the alignments change if the `gapExtension` argument is set to `-Inf`?

[Answers provided in section 12.1.]

4 Pairwise Sequence Alignment Classes

Following the design principles of Bioconductor and R, the pairwise sequence alignment functionality in the `Biostrings` package keeps the end-user close to their data through the use of four specialty classes: `PairwiseAlignedFixedSubject`, `PairwiseAlignedFixedSubjectSummary`, `AlignedXStringSet`, and `QualityAlignedXStringSet`. The `PairwiseAlignedFixedSubject` class holds the results of a fit from the `pairwiseAlignment` function.

```
> psa1 <- pairwiseAlignment(pattern = c("succeed", "precede"),
+   subject = "supersede")
> class(psa1)
```

```
[1] "PairwiseAlignedFixedSubject"
attr(,"package")
[1] "Biostrings"
```

and the `pairwiseAlignmentSummary` function holds the results of a summarized pairwise sequence alignment.

```
> summary(psa1)
```

```
Global Pairwise Alignment
Number of Alignments: 2
```

```
Scores:
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-12.400	-10.550	-8.697	-8.697	-6.846	-4.995

```
Number of matches:
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
4.00	4.25	4.50	4.50	4.75	5.00

```
Top 4 Mismatch Counts:
```

SubjectPosition	Subject	Pattern	Count	Probability	
1	3	p	c	1	0.5
2	4	e	r	1	0.5
3	5	r	e	1	0.5
4	6	s	c	1	0.5

```
> class(summary(psa1))
```

```
[1] "PairwiseAlignedFixedSubjectSummary"
attr(,"package")
[1] "Biostrings"
```

The *AlignedXStringSet* and *QualityAlignedXStringSet* classes hold the “gapped” S'_i substrings with the former class holding the results when the pairwise sequence alignment is performed with a fixed substitution scoring scheme and the latter class a quality-based scoring scheme.

```
> class(pattern(psa1))

[1] "QualityAlignedXStringSet"
attr(,"package")
[1] "Biostrings"

> submat <- matrix(-1, nrow = 26, ncol = 26, dimnames = list(letters,
+   letters))
> diag(submat) <- 0
> psa2 <- pairwiseAlignment(pattern = c("succeed", "precede"),
+   subject = "supersede", substitutionMatrix = submat,
+   gapOpening = 0, gapExtension = -1)
> class(pattern(psa2))

[1] "AlignedXStringSet"
attr(,"package")
[1] "Biostrings"
```

4.1 Exercise 2

1. What is the primary benefit of formal summary classes like *PairwiseAlignedFixedSubjectSummary* and *summary.lm* to end-users?

[Answer provided in section 12.2.]

5 Pairwise Sequence Alignment Helper Functions

Tables 1 and 2 show functions that interact with objects of class *PairwiseAlignedFixedSubject* and *AlignedXStringSet* respectively. These functions should be used in preference to direct slot extraction from the alignment objects.

The `score`, `nmatch`, `nmismatch`, and `nchar` functions return numeric vectors containing information on the pairwise sequence alignment score, number of matches, number of mismatches, and number of aligned characters respectively.

```
> submat <- matrix(-1, nrow = 26, ncol = 26, dimnames = list(letters,
+   letters))
> diag(submat) <- 0
> psa2 <- pairwiseAlignment(pattern = c("succeed", "precede"),
+   subject = "supersede", substitutionMatrix = submat,
+   gapOpening = 0, gapExtension = -1)
> score(psa2)

[1] -5 -5

> nmatch(psa2)

[1] 4 4

> nmismatch(psa2)
```

Function	Description
[Extracts the specified elements of the alignment object
<code>aligned</code>	Creates <i>XStringSet</i> alignments without insertions
<code>alphabet</code>	Extracts the allowable characters in the original strings
<code>as.character</code>	Creates a character vector version of <code>aligned</code>
<code>as.matrix</code>	Creates an “exploded” character matrix version of <code>aligned</code>
<code>compareStrings</code>	Creates character string mashups of the alignments
<code>consensusMatrix</code>	Computes a consensus matrix for the alignments
<code>consensusString</code>	Creates the string based on a 50% + 1 vote from the consensus matrix
<code>coverage</code>	Computes the alignment coverage along the subject
<code>length</code>	Extracts the number of patterns aligned
<code>mismatchSummary</code>	Summarizes the information of the <code>mismatchTable</code>
<code>mismatchTable</code>	Creates a table for the mismatching positions
<code>nchar</code>	Computes the length of “gapped” substrings
<code>nindel</code>	Computes the number of insertions/deletions in the alignments
<code>nmatch</code>	Computes the number of matching characters in the alignments
<code>nmismatch</code>	Computes the number of mismatching characters in the alignments
<code>pattern, subject</code>	Extracts the aligned pattern/subject
<code>pid</code>	Computes the percent sequence identity
<code>rep</code>	Replicates the elements of the alignment object
<code>score</code>	Extracts the pairwise sequence alignment scores
<code>summary</code>	Summarizes a pairwise sequence alignment
<code>toString</code>	Creates a concatenated string version of <code>aligned</code>
<code>type</code>	Extracts the type of pairwise sequence alignment
<code>Views</code>	Extracts the alignment ranges for the subject

Table 1: Functions for *PairwiseAlignedFixedSubject* objects.

```
[1] 3 3
> nchar(psa2)
[1] 8 9
> aligned(psa2)
  A BStringSet instance of length 2
  width seq
[1]    9 succe-ed-
[2]    9 pr-ec-ede
> as.character(psa2)
[1] "succe-ed-" "pr-ec-ede"
> as.matrix(psa2)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,] "s"  "u"  "c"  "c"  "e"  "-"  "e"  "d"  "-"
[2,] "p"  "r"  "-"  "e"  "c"  "-"  "e"  "d"  "e"
> consensusMatrix(psa2)
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]
-	0	0	1	0	0	2	0	0	1
c	0	0	1	1	1	0	0	0	0
d	0	0	0	0	0	0	0	2	0
e	0	0	0	1	1	0	2	0	1
p	1	0	0	0	0	0	0	0	0
r	0	1	0	0	0	0	0	0	0
s	1	0	0	0	0	0	0	0	0
u	0	1	0	0	0	0	0	0	0

The `summary`, `mismatchTable`, and `mismatchSummary` functions return various summaries of the pairwise sequence alignments.

```
> summary(psa2)
```

Global Pairwise Alignment

Number of Alignments: 2

Scores:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-5	-5	-5	-5	-5	-5

Number of matches:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
4	4	4	4	4	4

Top 6 Mismatch Counts:

SubjectPosition	Subject	Pattern	Count	Probability
1	s	p	1	0.5
2	u	r	1	0.5
3	p	c	1	0.5
4	e	c	1	0.5
5	r	c	1	0.5
6	r	e	1	0.5

```
> mismatchTable(psa2)
```

PatternId	PatternStart	PatternEnd	PatternSubstring	SubjectStart
1	1	3	c	3
2	1	4	c	4
3	1	5	e	5
4	2	1	p	1
5	2	2	r	2
6	2	4	c	5

SubjectEnd	SubjectSubstring
1	p
2	e
3	r
4	s
5	u
6	r

```
> mismatchSummary(psa2)
```



```

$pattern
$pattern$position
  Position Count Probability
1         1     1         0.5
2         2     1         0.5
3         3     1         0.5
4         4     2         1.0
5         5     1         0.5
6         6     0         0.0
7         7     0         0.0

```

```

$subject
  SubjectPosition Subject Pattern Count Probability
1             1     s     p     1         0.5
2             2     u     r     1         0.5
3             3     p     c     1         0.5
4             4     e     c     1         0.5
5             5     r     c     1         0.5
6             5     r     e     1         0.5

```

Function	Description
[Extracts the specified elements of the alignment object
aligned, unaligned	Extracts the aligned/unaligned strings
alphabet	Extracts the allowable characters in the original strings
as.character, toString	Converts the alignments to character strings
coverage	Computes the alignment coverage
end	Extracts the ending index of the aligned range
indel	Extracts the insertion/deletion locations
length	Extracts the number of patterns aligned
mismatch	Extracts the position of the mismatches
mismatchSummary	Summarizes the information of the mismatchTable
mismatchTable	Creates a table for the mismatching positions
nchar	Computes the length of “gapped” substrings
nindel	Computes the number of insertions/deletions in the alignments
nmatch	Computes the number of matching characters in the alignments
nmismatch	Computes the number of mismatching characters in the alignments
rep	Replicates the elements of the alignment object
start	Extracts the starting index of the aligned range
toString	Creates a concatenated string containing the alignments
width	Extracts the width of the aligned range

Table 2: Functions for *AlignedXString* and *QualityAlignedXString* objects.

The `pattern` and `subject` functions extract the aligned pattern and subject objects for further analysis. Most of the actions that can be performed on *PairwiseAlignedFixedSubject* objects can also be performed on *AlignedXStringSet* and *QualityAlignedXStringSet* objects as well as operations including `start`, `end`, and `width` that extracts the start, end, and width of the alignment ranges.

```
> class(pattern(psa2))
```

```

[1] "AlignedXStringSet"
attr(,"package")
[1] "Biostrings"
> aligned(pattern(psa2))
  A BStringSet instance of length 2
  width seq
[1]    8 succe-ed
[2]    9 pr-ec-ede
> nindel(pattern(psa2))
      Length WidthSum
[1,]     1         1
[2,]     2         2
> start(subject(psa2))
[1] 1 1
> end(subject(psa2))
[1] 8 9

```

5.1 Exercise 3

For the overlap pairwise sequence alignment of the strings "syzygy" and "zyzzyx" with the `pairwiseAlignment` default settings, perform the following operations:

1. Use `nmatch` and `nmismatch` to extract the number of matches and mismatches respectively.
2. Use the `compareStrings` function to get the symbolic representation of the alignment.
3. Use the `as.character` function to get the character string versions of the alignments.
4. Use the `pattern` function to extract the aligned pattern and apply the `mismatch` function to it to find the locations of the mismatches.
5. Use the `subject` function to extract the aligned subject and apply the `aligned` function to it to get the aligned strings.

[Answers provided in section 12.3.]

6 Edit Distances

One of the earliest uses of pairwise sequence alignment is in the area of text analysis. In 1965 Vladimir Levenshtein considered a metric, now called the *Levenshtein edit distance*, that measures the similarity between two strings. This distance metric is equivalent to the negative of the score of a pairwise sequence alignment with a match cost of 0, a mismatch cost of -1, a gap opening penalty of 0, and a gap extension cost of -1.

The `stringDist` uses the internals of the `pairwiseAlignment` function to calculate the Levenshtein edit distance matrix for a set of strings.

There is also an implementation of approximate string matching using Levenshtein edit distance in the `agrep` (approximate grep) function of the `base R` package. As the following example shows, it is possible to replicate the `agrep` function using the `pairwiseAlignment` function. Since the `agrep` function is vectorized in x rather than $pattern$, these arguments are flipped in the call to `pairwiseAlignment`.

```

> agrepBioC <- function(pattern, x, ignore.case = FALSE,
+   value = FALSE, max.distance = 0.1) {
+   if (!is.character(pattern))
+     pattern <- as.character(pattern)
+   if (!is.character(x))
+     x <- as.character(x)
+   if (max.distance < 1)
+     max.distance <- ceiling(max.distance/nchar(pattern))
+   characters <- unique(unlist(strsplit(c(pattern, x),
+     "", fixed = TRUE)))
+   if (ignore.case)
+     substitutionMatrix <- outer(tolower(characters),
+       tolower(characters), function(x, y) -as.numeric(x !=
+         y))
+   else substitutionMatrix <- outer(characters, characters,
+     function(x, y) -as.numeric(x != y))
+   dimnames(substitutionMatrix) <- list(characters, characters)
+   distance <- -pairwiseAlignment(pattern = x, subject = pattern,
+     substitutionMatrix = substitutionMatrix, type = "patternOverlap",
+     gapOpening = 0, gapExtension = -1, scoreOnly = TRUE)
+   whichClose <- which(distance <= max.distance)
+   if (value)
+     whichClose <- x[whichClose]
+   whichClose
+ }
> cbind(base = agrep("laysy", c("1 lazy", "1", "1 LAZY"),
+   max = 2, value = TRUE), bioc = agrepBioC("laysy", c("1 lazy",
+   "1", "1 LAZY"), max = 2, value = TRUE))

      base      bioc
[1,] "1 lazy" "1 lazy"

> cbind(base = agrep("laysy", c("1 lazy", "1", "1 LAZY"),
+   max = 2, ignore.case = TRUE), bioc = agrepBioC("laysy",
+   c("1 lazy", "1", "1 LAZY"), max = 2, ignore.case = TRUE))

      base bioc
[1,]    1    1
[2,]    3    3

```

6.1 Exercise 4

1. Use the `pairwiseAlignment` function to find the Levenshtein edit distance between "syzygy" and "zyzzyx".
2. Use the `stringDist` function to find the Levenshtein edit distance for the vector `c("zyzzyx", "syzygy", "succeed", "precede", "supersede")`.

[Answers provided in section 12.4.]

7 Application: Using Evolutionary Models in Protein Alignments

When proteins are believed to descend from a common ancestor, evolutionary models can be used as a guide in pairwise sequence alignments. The two most common families evolutionary models of proteins used in pairwise sequence alignments are Point Accepted Mutation (PAM) matrices, which are based on explicit evolutionary models, and Block Substitution Matrix (BLOSUM) matrices, which are based on data-derived evolution models. The Biostrings package contains 5 PAM and 5 BLOSUM matrices (PAM30 PAM40, PAM70, PAM120, PAM250, BLOSUM45, BLOSUM50, BLOSUM62, BLOSUM80, and BLOSUM100) that can be used in the *substitutionMatrix* argument to the `pairwiseAlignment` function.

Here is an example pairwise sequence alignment of amino acids from Durbin, Eddy et al being fit by the `pairwiseAlignment` function using the BLOSUM50 matrix:

```
> data(BLOSUM50)
> BLOSUM50[1:4, 1:4]

  A  R  N  D
A  5 -2 -1 -2
R -2  7 -1 -2
N -1 -1  7  2
D -2 -2  2  8

> nwdemo <- pairwiseAlignment(AAString("PAWHEAE"), AAString("HEAGAWGHEE"),
+   substitutionMatrix = BLOSUM50, gapOpening = 0, gapExtension = -8)
> nwdemo

Global Pairwise Alignment (1 of 1)
pattern: [1] PA--W-HEAE
subject: [2] EAGAWGHE-E
score: 1

> compareStrings(nwdemo)

[1] "?A--W-HE+E"

> pid(nwdemo)

[1] 50
```

7.1 Exercise 5

1. Repeat the alignment exercise above using BLOSUM62, a gap opening penalty of -12, and a gap extension penalty of -4.
2. Explore to find out what caused the alignment to change.

[Answers provided in section 12.5.]

8 Application: Removing Adapters from Sequence Reads

Finding and removing uninteresting experiment process-related fragments like adapters is a common problem in genetic sequencing, and pairwise sequence alignment is well-suited to address this issue. When adapters are used to anchor or extend a sequence during the experiment process, they either intentionally or unintentionally become sequenced during the read process. The following code simulates what sequences with adapter fragments at either end could look like during an experiment.

```

> simulateReads <- function(N, adapter, experiment, substitutionRate = 0.01,
+   gapRate = 0.001) {
+   chars <- strsplit(as.character(adapter), "")[[1]]
+   sapply(seq_len(N), function(i, experiment, substitutionRate,
+     gapRate) {
+     width <- experiment[["width"]][i]
+     side <- experiment[["side"]][i]
+     randomLetters <- function(n) sample(DNA_ALPHABET[1:4],
+       n, replace = TRUE)
+     randomLettersWithEmpty <- function(n) sample(c("",
+       DNA_ALPHABET[1:4]), n, replace = TRUE, prob = c(1 -
+       gapRate, rep(gapRate/4, 4)))
+     nChars <- length(chars)
+     value <- paste(ifelse(rbinom(nChars, 1, substitutionRate),
+       randomLetters(nChars), chars), randomLettersWithEmpty(nChars),
+       sep = "", collapse = "")
+     if (side)
+       value <- paste(c(randomLetters(36 - width),
+         substring(value, 1, width)), sep = "",
+         collapse = "")
+     else value <- paste(c(substring(value, 37 - width,
+       36), randomLetters(36 - width)), sep = "",
+       collapse = "")
+     value
+   }, experiment = experiment, substitutionRate = substitutionRate,
+   gapRate = gapRate)
+ }
> adapter <- DNASTring("GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA")
> set.seed(123)
> N <- 1000
> experiment <- list(side = rbinom(N, 1, 0.5), width = sample(0:36,
+   N, replace = TRUE))
> table(experiment[["side"]], experiment[["width"]])

  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
0 13 10  8  7 11 18  9 15 15 18 16 10 11  9 13 13 18 18 14  9 19 13
1 15 21 21 12 17 14  8 11 12 10 14 16  7 14 19 14 14 16 14 16 16 16

 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
0 19 19 12  6 15 18 12 15 16 17 19  6 13 18 15
1 13 11  9 11 15 10 10 16 15 15 11  7 12  7 14

> adapterStrings <- simulateReads(N, adapter, experiment,
+   substitutionRate = 0.01, gapRate = 0.001)
> adapterStrings <- DNASTringSet(adapterStrings)
> adapterStrings

```

```

A DNASTringSet instance of length 1000
  width seq
[1] 36 CTGCTTGAAATTGCACGATAGTTGCATATGCTACAA
[2] 36 ATTTCTCCTTCTCAGGATCGGAAGAGCTCGTATGCC
[3] 36 TGAAAGAAGGTAATTTGATTAAGCCCTTCGCAAAAC

```

```

[4] 36 CAAACGATCGGAAGAGCTCGTATGCCGTCTTCTGCT
[5] 36 TCTCAGATCGGAAGAGCTCGTATGCCGTCTTCTGCT
[6] 36 CCGTCTTCTGCTTGAAACGTGAACAGGACAATGGCC
[7] 36 GGAAGCCAGATCGTAAGAGCTCGTATGCCGTCTTCT
[8] 36 CGGGTCCTGGTCCTGGGGCCATCCATGATCGGAAGA
[9] 36 TGGCACATCGCAGCTAAATCGACAGTACTATCATGA
... ..
[992] 36 TTTAAACTACTGGAATAAATGCAAGTGGACAAAACGC
[993] 36 TGGCAGATCGGAAGAGCTCGTATGCCGTCTTTCAGCT
[994] 36 TGAATATATGTCATCTCATACAAGCACGTAATTCATTG
[995] 36 CTCGGGTACACGCCTCGGTGCACACATAATTGGGAT
[996] 36 GTTGATCGGAAGAGCTCGTATGCCGTCTTCTGCTTG
[997] 36 AATGTGATGTCTCACTTCAAAGGCGGATCGGAAGAG
[998] 36 AGCTCGTATGCCGTCTTCTGCTTGAAAAAATTATTC
[999] 36 ACTAACTGCACTCCCGCACCATGATCGGAAGAGCTC
[1000] 36 CGTCTTCTGCTTGAAAAATCAGGTGTTGGGCCTGCCG

```

These simulated strings above have 0 to 36 characters from the adapters attached to either end. We can use completely random strings as a baseline for any pairwise sequence alignment methodology we develop to remove the adapter characters.

```

> M <- 5000
> randomStrings <- apply(matrix(sample(DNA_ALPHABET[1:4],
+ 36 * M, replace = TRUE), nrow = M), 1, paste, collapse = "")
> randomStrings <- DNAStrngSet(randomStrings)
> randomStrings

```

```

A DNAStrngSet instance of length 5000
width seq
[1] 36 AAGAGGGATTGATGGTATTTGTATTAGAATTATGAC
[2] 36 CGTGTGGGCTCGCAAGATTCACCGTGAAGGCTAACG
[3] 36 TAAATATCGTTCATGGACTGTTGTGCATAGGCGGCT
[4] 36 TGACTCGTCTTGGCAGGAGATTATGACTACACACCT
[5] 36 TATAGGATACTATTGCACTGTACATAAGGCCTCGTC
[6] 36 TTTCTGGGCAACTCCGGTCATTCCGTGCACGGGCGT
[7] 36 CTTACGGAATATGGAACCATCTGCCATTGCACCCTT
[8] 36 TCTCTCGGGATGCACTGTGAGAAACCTGCCCAAGG
[9] 36 TGCTGGTACACGCGCATTACGAGTGTCTCTAACTG
... ..
[4992] 36 GCATAAGGGCGTCATCAATATTGGCCGACTCACCGA
[4993] 36 GAGACCGGTCGGGACAAATATGAGCTTAAACAAGCG
[4994] 36 GCTGCCGGGGCGTTATTCCGCTCAATAAAGGGCCGG
[4995] 36 GAGATAATTATCCAAGACCCCTGTTCTACAGATTAT
[4996] 36 TTGGGTTTGTAGAAAGTACGCCTTAACCTCACGCAC
[4997] 36 CCAGGGGACAAGCCATTATTTAACGCTAACTTTCAA
[4998] 36 AAAGTGAAGCAATTATTCGTGCGACACAGCTATTGT
[4999] 36 TTTGAGGTAGCAGGGTGCCGGAGGCCGCTCGTCTC
[5000] 36 ATTCTATATCGAACTGCGAGCGTCGATGGAGGATGT

```

Since edit distances are easy to explain, it serves as a good place to start for developing a adapter removal methodology. Unfortunately given that it is based on a global alignment, it only is useful for filtering out sequences that are derived primarily from the adapter.

```

> submat1 <- nucleotideSubstitutionMatrix(match = 0, mismatch = -1,
+   baseOnly = TRUE)
> randomScores1 <- pairwiseAlignment(randomStrings, adapter,
+   substitutionMatrix = submat1, gapOpening = 0, gapExtension = -1,
+   scoreOnly = TRUE)
> quantile(randomScores1, seq(0.99, 1, by = 0.001))

```

```

99% 99.1% 99.2% 99.3% 99.4% 99.5% 99.6% 99.7% 99.8% 99.9% 100%
-16 -16 -16 -16 -16 -16 -16 -16 -16 -15 -14

```

```

> adapterAligns1 <- pairwiseAlignment(adapterStrings, adapter,
+   substitutionMatrix = submat1, gapOpening = 0, gapExtension = -1)
> table(score(adapterAligns1) > quantile(randomScores1, 0.999),
+   experiment[["width"]])

```

```

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32 17 26 27 28 30 26 18 23 32 27 32 34 28 25 35
TRUE   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE 29 32 30 21 17 30 28 22  5  1  0  0  0  0  0  0
TRUE   0  0  0  0  0  0  0  0  26 30 32 30 13 25 25 29

```

One improvement to removing adapters is to look at consecutive matches anywhere within the sequence. This is more versatile than the edit distance method, but it requires a relatively large number of consecutive matches and is susceptible to issues related to error related substitutions and insertions/deletions.

```

> submat2 <- nucleotideSubstitutionMatrix(match = 1, mismatch = -Inf,
+   baseOnly = TRUE)
> randomScores2 <- pairwiseAlignment(randomStrings, adapter,
+   substitutionMatrix = submat2, type = "local", gapOpening = 0,
+   gapExtension = -Inf, scoreOnly = TRUE)
> quantile(randomScores2, seq(0.99, 1, by = 0.001))

```

```

99% 99.1% 99.2% 99.3% 99.4% 99.5% 99.6% 99.7% 99.8% 99.9%
7.000 8.000 8.000 8.000 8.000 8.000 8.000 8.000 8.000 9.000 9.001
100%
11.000

```

```

> adapterAligns2 <- pairwiseAlignment(adapterStrings, adapter,
+   substitutionMatrix = submat2, type = "local", gapOpening = 0,
+   gapExtension = -Inf)
> table(score(adapterAligns2) > quantile(randomScores2, 0.999),
+   experiment[["width"]])

```

```

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32 17 26 27 28  2  0  0  1  2  0  1  0  1  1  0
TRUE   0  0  0  0  0  0  0  0  0  0  28 26 18 22 30 27 31 34 27 24 35

      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TRUE  29 32 30 21 17 30 28 22 31 31 32 30 13 25 25 29

```

```
> table(start(pattern(adapterAligns2)) > 37 - end(pattern(adapterAligns2)),
+       experiment[["side"]])
```

```
      0  1
FALSE 466 58
TRUE  41 435
```

Limiting consecutive matches to the ends provides better results, but it doesn't resolve the issues related to substitutions and insertions/deletions errors.

```
> submat3 <- nucleotideSubstitutionMatrix(match = 1, mismatch = -Inf,
+   baseOnly = TRUE)
> randomScores3 <- pairwiseAlignment(randomStrings, adapter,
+   substitutionMatrix = submat3, type = "overlap", gapOpening = 0,
+   gapExtension = -Inf, scoreOnly = TRUE)
> quantile(randomScores3, seq(0.99, 1, by = 0.001))
```

```
99% 99.1% 99.2% 99.3% 99.4% 99.5% 99.6% 99.7% 99.8% 99.9% 100%
  4     4     4     4     4     5     5     5     5     6     7
```

```
> adapterAligns3 <- pairwiseAlignment(adapterStrings, adapter,
+   substitutionMatrix = submat3, type = "overlap", gapOpening = 0,
+   gapExtension = -Inf)
> table(score(adapterAligns3) > quantile(randomScores3, 0.999),
+       experiment[["width"]])
```

```
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32 17  1  0  3  3  0  0  2  3  6  3  6  5  7  4
TRUE  0  0  0  0  0  0  0 25 27 25 27 26 18 21 29 21 29 28 23 18 31
```

```
      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE  8  7  2  4  7  8  5  4  7  6  4  8  4 11  5 10
TRUE  21 25 28 17 10 22 23 18 24 25 28 22  9 14 20 19
```

```
> table(end(pattern(adapterAligns3)) == 36, experiment[["side"]])
```

```
      0  1
FALSE 478 70
TRUE  29 423
```

Allowing for substitutions and insertions/deletions errors in the pairwise sequence alignments provides much better results for finding adapter fragments.

```
> randomScores4 <- pairwiseAlignment(randomStrings, adapter,
+   type = "overlap", scoreOnly = TRUE)
> quantile(randomScores4, seq(0.99, 1, by = 0.001))
```

```
99%      99.1%      99.2%      99.3%      99.4%      99.5%      99.6%
7.927024 7.927024 7.927024 7.927024 7.927024 7.927024 7.973007
99.7%      99.8%      99.9%      100%
9.908780 9.908780 9.908780 11.890536
```

```
> adapterAligns4 <- pairwiseAlignment(adapterStrings, adapter,
+   type = "overlap")
> table(score(adapterAligns4) > quantile(randomScores4, 0.999),
+       experiment[["width"]])
```



```

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32  2  1  0  0  0  0  0  0  0  0  0  0  0  0
TRUE  0  0  0  0  0  0 15 25 27 28 30 26 18 23 32 27 32 34 28 25 35

      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TRUE 29 32 30 21 17 30 28 22 31 31 32 30 13 25 25 29

```

```
> table(end(pattern(adapterAligns4)) == 36, experiment[["side"]])
```

```

      0  1
FALSE 488 20
TRUE  19 473

```

Using the results that allow for substitutions and insertions/deletions errors, the cleaned sequence fragments can be generated as follows:

```

> fragmentFound <- score(adapterAligns4) > quantile(randomScores4,
+ 0.999)
> fragmentFoundAt1 <- fragmentFound & (start(pattern(adapterAligns4)) ==
+ 1)
> fragmentFoundAt36 <- fragmentFound & (end(pattern(adapterAligns4)) ==
+ 36)
> cleanedStrings <- as.character(adapterStrings)
> cleanedStrings[fragmentFoundAt1] <- as.character(narrow(adapterStrings[fragmentFoundAt1],
+ end = 36, width = 36 - end(pattern(adapterAligns4[fragmentFoundAt1])))
> cleanedStrings[fragmentFoundAt36] <- as.character(narrow(adapterStrings[fragmentFoundAt36],
+ start = 1, width = start(pattern(adapterAligns4[fragmentFoundAt36])) -
+ 1))
> cleanedStrings <- DNASTringSet(cleanedStrings)
> cleanedStrings

```

A DNASTringSet instance of length 1000

```

width seq
[1] 26 TTGCACGATAGTTGCATATGCTACAA
[2] 15 ATTTCTCCTTCTCAG
[3] 36 TGAAAGAAGGTAATTTGATTAAGCCCTTCGCAAAC
[4] 5 CAAAC
[5] 5 TCTCA
[6] 19 CGTGAACAGGACAATGGCC
[7] 8 GGAAGCCA
[8] 26 CGGGTCCTGGTCCTGGGGCCATCCAT
[9] 36 TGGCACATCGCAGCTAAATCGACAGTACTATCATGA
... ..
[992] 36 TTTAAACTACTGGAATAAATGCAAGTGGACAAACGC
[993] 5 TGGCA
[994] 36 TGAAATATGTCATCTCATACAAGCACGTAATTCATTG
[995] 36 CTCCGGTACACGCTCGGTGCACACATAATTGGGAT
[996] 3 GTT
[997] 25 AATGTGATGTCTCACTTCAAAGGCG
[998] 9 AAATTATTC
[999] 22 ACTAACTGCACTCCCGCACCAT
[1000] 20 ATCAGGTGTTGGGCCTGCCG

```

8.1 Exercise 6

1. Rerun the simulation time using the `simulateReads` function with a *substitutionRate* of 0.005 and *gapRate* of 0.0005. How do the different pairwise sequence alignment methods compare?
2. (Advanced) Modify the `simulateReads` function to accept different equal length adapters on either side (left & right) of the reads. How would the methods for trimming the reads change?

[Answers provided in section 12.6.]

9 Application: Quality Assurance in Sequencing Experiments

Due to its flexibility, the `pairwiseAlignment` function is able to diagnose sequence matching-related issues that arise when `matchPDict` and its related functions don't find a match when aligning sequence fragments to a target. This section contains an example involving a short read Solexa sequencing experiment of bacteriophage ϕ X174 DNA produced by New England BioLabs (NEB). This experiment contains slightly less than 5000 unique short reads in `srPhiX174`, with quality measures in `quPhiX174`, and frequency for those short reads in `wtPhiX174`.

In order to demonstrate how to find sequence differences in the target, these short reads will be compared against the bacteriophage ϕ X174 genome NC_001422 from the GenBank database.

```
> data(phiX174Phage)
> genBankPhage <- phiX174Phage[[1]]
> nchar(genBankPhage)

[1] 5386

> data(srPhiX174)
> srPhiX174

A DNASTringSet instance of length 1113
  width seq
[1] 35 GTTATTATACCGTCAAGGACTGTGTGACTATTGAC
[2] 35 GGTGGTTATTATACCGTCAAGGACTGTGTGACTAT
[3] 35 TACCGTCAAGGACTGTGTGACTATTGACGTCCTTC
[4] 35 GTACGCCGGCAATAATGTTTATGTTGGTTTCATG
[5] 35 GGTTTCATGGTTTGGTCTAACTTTACCGCTACTAA
[6] 35 GGGCAATAATGTTTATGTTGGTTTCATGGTTTGGT
[7] 35 GTCCTTCCTCGTACGCCGGCAATAATGTTTATGT
[8] 35 GTTGGTTTCATGGTTTGGTCTAACTTTACCGCTAC
[9] 35 GCAATAATGTTTATGTTGGTTTCATGGTTTGGTCT
...
[1105] 35 AATAATGTTTATGTTGGTTTCATGTTTTTTTCTAA
[1106] 35 GGTGGTTATTATACCGTCAAGGACTTTGTGACTAT
[1107] 35 CGGGCAATAATGTTTATGTTGGTTTCATGTTTTGT
[1108] 35 ATTGACGTCCTTCCTCGTACGCCGGCAATAATGC
[1109] 35 ATAATGTTTATGTTGGTTTCATGGTTTGTCTATC
[1110] 35 GGGCAATAATGTTTATGTTGGTTTCATTTTTTTTT
[1111] 35 CAATAATGTTTATGTTGGTTTCATGGTTTGTTTTA
[1112] 35 GACGTCCTTCCTCGTACGCCGGCAATGATGTTTA
[1113] 35 ACGCCGGCAATAATGTTTATGTTGGTTTCATTGT

> quPhiX174
```

```

A BStringSet instance of length 1113
  width seq
[1] 35 ZYZZZZZZZZZYZZYYYYYYYYYYYYYYYYYQYY
[2] 35 ZZYZZYZZZZYYYYYYYYYYYYYYYYVYYTY
[3] 35 ZZZYZYZYZYZYYYYYYYYYYYYVYYYYY
[4] 35 ZZYZZZZZZZZYZYYYYYYYYYYYYNYT
[5] 35 ZZZZZYZYZYZZZYYYYYYYYYYYYSYYSY
[6] 35 ZZZZZYZYZYZYZYYYYYYYYYYYYSYQVYYASY
[7] 35 ZZZZZZZZZYZZZZZYYYYYYYYYYYYYYY
[8] 35 YYZYZZYZYZYZYZVQYYYYYYYYYYYTYYY
[9] 35 ZZZYZZZZZYZZZZZYQYYYYYYYQYYQGYNY
... ..
[1105] 35 ZZZZZYZZZYZYZYZYQYYQCYJAYYYQKYJYJJ
[1106] 35 VVLVVVVQVSQPVVVMQSSUJPLQVAGFPGFNFJJ
[1107] 35 ZZZZZYZYZYZZZZZYYYYYYYQYYYYYIAYYAI
[1108] 35 YYYZYZZYZYZYZPYWYYSVVYVWYVVMIVRWYG
[1109] 35 ZZZZZYZZZYZZZVYYYYVYYYQYYQCYQYQCT
[1110] 35 YYYTYYYYYTYYYYYYJTYYOAYIYYGYGAY
[1111] 35 ZZYZZZZZZZZVZYVYYYYVQYYIYQYAYW
[1112] 35 YZYZZYZYZYZYZVYVYVYVWVYVYVYVYV
[1113] 35 ZZYZZYZYZYZYZVZYVYVYVYVYVYVYVYV

```

```
> summary(wtPhiX174)
```

```

  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.00   2.00   3.00  48.34   6.00  965.00

```

```

> fullShortReads <- rep(srPhiX174, wtPhiX174)
> srPDict <- PDict(fullShortReads)
> table(countPDict(srPDict, genBankPhage))

```

```

  0    1
37018 16784

```

For these short reads, the `pairwiseAlignment` function finds that the small number of perfect matches is due to two locations on the bacteriophage ϕ X174 genome.

Unlike the `countPDict` function, the `pairwiseAlignment` function works off of the original strings, rather than `PDict` processed strings, and to be computationally efficient it is recommended that the unique sequences are supplied to the `pairwiseAlignment` function, and the frequencies of those sequences are supplied to the `weight` argument of functions like `summary`, `mismatchSummary`, and `coverage`. For the purposes of this exercise, a substring of the GenBank bacteriophage ϕ X174 genome is supplied to the `subject` argument of the `pairwiseAlignment` function to reduce the computation time.

```

> genBankSubstring <- substring(genBankPhage, 2793 - 34,
+   2811 + 34)
> genBankAlign <- pairwiseAlignment(srPhiX174, genBankSubstring,
+   patternQuality = SolexaQuality(quPhiX174), subjectQuality = SolexaQuality(99L),
+   type = "subjectOverlap")
> summary(genBankAlign, weight = wtPhiX174)

```

```

Subject Overlap Pairwise Alignment
Number of Alignments: 53802

```

Scores:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-45.08	35.81	50.07	41.24	59.50	67.35

Number of matches:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
21.00	31.00	33.00	31.46	34.00	35.00

Top 10 Mismatch Counts:

	SubjectPosition	Subject	Pattern	Count	Probability
101	53	C	T	22965	0.95536234
58	35	C	T	22849	0.99969373
153	76	G	T	1985	0.10062351
139	69	A	T	1296	0.05654697
159	79	C	T	1289	0.07289899
111	58	A	C	1153	0.04783637
144	72	G	A	1130	0.05248978
123	63	G	A	1130	0.04767731
133	67	T	G	1130	0.04721514
163	81	A	G	1103	0.06672313

```
> revisedPhage <- replaceLetterAt(genBankPhage, c(2793, 2811),  
+ "TT")  
> table(countPDict(srPDict, revisedPhage))
```

```
  0    1  
6768 47034
```

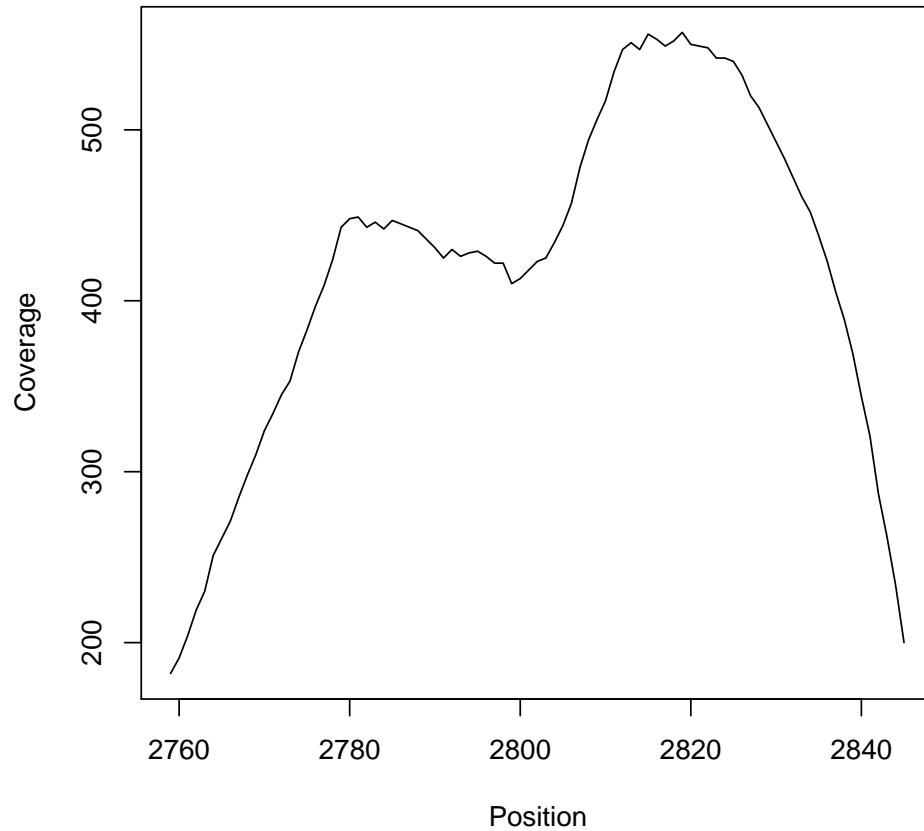
The following plot shows the coverage of the aligned short reads along the substring of the bacteriophage ϕ X174 genome. Applying the `slice` function to the coverage shows the entire substring is covered by aligned short reads.

```
> genBankCoverage <- coverage(genBankAlign, weight = wtPhiX174)  
> plot((2793 - 34):(2811 + 34), as.integer(genBankCoverage),  
+ xlab = "Position", ylab = "Coverage", type = "l")  
> nchar(genBankSubstring)
```

```
[1] 87
```

```
> slice(genBankCoverage, lower = 1)
```

```
Views on a 87-integer XRleInteger subject  
subject: 182 191 204 219 230 251 261 271 285 298 310 324 334 345 353 370 383 ...  
views:  
  start end width  
[1]    1  87   87 [182 191 204 219 230 251 261 271 285 298 310 ...]
```



9.1 Exercise 7

1. Rerun the “Subject Overlap” alignment of the short reads against the entire genome. (This may take a few minutes.)
2. Plot the coverage of these alignments and use the `slice` function to find the ranges of alignment. Are there any alignments outside of the substring region that was used above?
3. Use the `reverseComplement` function on the bacteriophage ϕ X174 genome. Do any short reads have a higher alignment score on this new sequence than on the original sequence?

[Answers provided in section 12.7.]

10 Computation Profiling

The `pairwiseAlignment` function uses a dynamic programming algorithm based on the Needleman-Wunsch and Smith-Waterman algorithms for global and local pairwise sequence alignments respectively. The algorithm consumes memory and computation time proportional to the product of the length of the two strings being aligned.

```

> N <- as.integer(seq(500, 5000, by = 500))
> timings <- rep(0, length(N))
> names(timings) <- as.character(N)
> for (i in seq_len(length(N))) {
+   string1 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   N[i], replace = TRUE), collapse = ""))
+   string2 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   N[i], replace = TRUE), collapse = ""))
+   timings[i] <- system.time(pairwiseAlignment(string1,
+   string2, type = "global"))[["user.self"]]
+ }
> timings

  500  1000  1500  2000  2500  3000  3500  4000  4500  5000
1.396 0.724 0.760 0.820 1.120 0.732 1.076 1.188 1.312 1.460

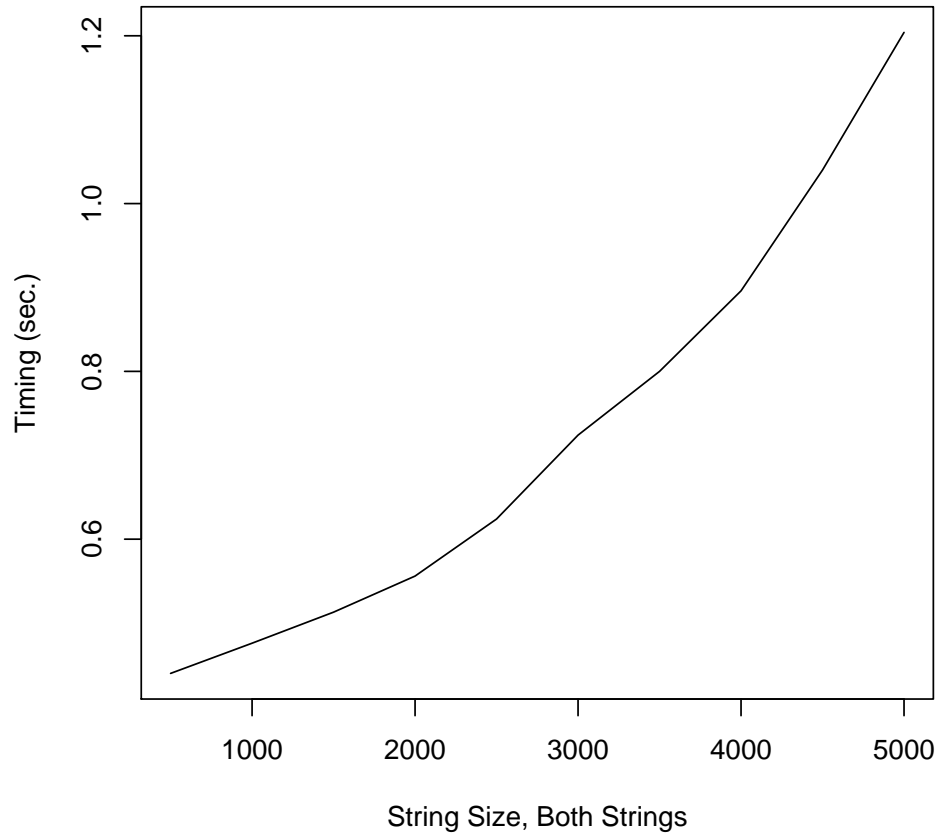
> coef(summary(lm(timings ~ poly(N, 2))))

      Estimate Std. Error  t value    Pr(>|t|)
(Intercept) 1.0588000 0.06316085 16.763549 6.575990e-07
poly(N, 2)1 0.3970075 0.19973214  1.987700 8.718645e-02
poly(N, 2)2 0.5365073 0.19973214  2.686134 3.126025e-02

> plot(N, timings, xlab = "String Size, Both Strings", ylab = "Timing (sec.)",
+   type = "l", main = "Global Pairwise Sequence Alignment Timings")

```

Global Pairwise Sequence Alignment Timings



When a problem only requires the pairwise sequence alignment score, setting the *scoreOnly* argument to TRUE will more than halve the computation time.

```
> scoreOnlyTimings <- rep(0, length(N))
> names(scoreOnlyTimings) <- as.character(N)
> for (i in seq_len(length(N))) {
+   string1 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   N[i], replace = TRUE), collapse = ""))
+   string2 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   N[i], replace = TRUE), collapse = ""))
+   scoreOnlyTimings[i] <- system.time(pairwiseAlignment(string1,
+   string2, type = "global", scoreOnly = TRUE))["user.self"]
+ }
> scoreOnlyTimings
 500 1000 1500 2000 2500 3000 3500 4000 4500 5000
0.416 0.440 0.464 0.476 0.520 0.572 0.620 0.684 0.764 0.821
> round((timings - scoreOnlyTimings)/timings, 2)
 500 1000 1500 2000 2500 3000 3500 4000 4500 5000
0.05 0.08 0.10 0.14 0.17 0.21 0.22 0.24 0.27 0.32
```

10.1 Exercise 8

1. Rerun the first set of profiling code, but this time fix the number of characters in `string1` to 35 and have the number of characters in `string2` range from 5000, 50000, by increments of 5000. What is the computational order of this simulation exercise?
2. Rerun the second set of profiling code using the simulations from the previous exercise with `scoreOnly` argument set to `TRUE`. Is it still twice as fast?

[Answers provided in section 12.8.]

11 Computing alignment consensus matrices

The `consensusMatrix` function is provided for computing a consensus matrix for a set of equal-length strings assumed to be aligned. To illustrate, the following application assumes the ORF data to be aligned for the first 10 positions (patently false):

```
> file <- system.file("extdata", "someORF.fa", package = "Biostrings")
> orf <- read.DNAStringSet(file, "fasta")
> orf
```

```
A DNAStringSet instance of length 7
  width seq                               names
[1] 5573 ACTTGTAATATATATCTTTT...TCGACCTTATTGTTGATAT YAL001C TFC3 SGDI...
[2] 5825 TTCCAAGGCCGATGAATTC...AATTTTTTTCTATTCTCTT YAL002W VPS8 SGDI...
[3] 2987 CTTTCATGTCAGCCTGCACT...ACTCATGTAGCTGCCTCAT YAL003W EFB1 SGDI...
[4] 3929 CACTCATATCGGGGGTCTT...CCGAAACACGAAAAAGTAC YAL005C SSA1 SGDI...
[5] 2648 AGAGAAAGAGTTTCACTTC...AATTTATGTGTGAACATAG YAL007C ERP2 SGDI...
[6] 2597 GTGTCCGGGCCTCGCAGGC...TTTGGCAGAATGTACTTTT YAL008W FUN14 SGD...
[7] 2780 CAAGATAATGTCAAAGTTA...AGGAAGAAAAAAAATCAC YAL009W SPO7 SGDI...
```

```
> orf10 <- DNAStringSet(orf, end = 10)
> consensusMatrix(orf10, baseOnly = TRUE, freq = TRUE)
```

```
      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
A    0.2857143 0.2857143 0.2857143 0.0000000 0.5714286 0.4285714
C    0.4285714 0.1428571 0.2857143 0.2857143 0.2857143 0.1428571
G    0.1428571 0.1428571 0.1428571 0.2857143 0.1428571 0.0000000
T    0.1428571 0.4285714 0.2857143 0.4285714 0.0000000 0.4285714
other 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
      [,7]      [,8]      [,9]      [,10]
A    0.4285714 0.4285714 0.2857143 0.1428571
C    0.0000000 0.0000000 0.2857143 0.4285714
G    0.4285714 0.4285714 0.1428571 0.2857143
T    0.1428571 0.1428571 0.2857143 0.1428571
other 0.0000000 0.0000000 0.0000000 0.0000000
```

The information content as defined by Hertz and Stormo 1995 is computed as follows:

```
> informationContent <- function(Lmers) {
+   zlog <- function(x) ifelse(x == 0, 0, log(x))
+   co <- consensusMatrix(Lmers, freq = TRUE)
+   lets <- rownames(co)
```



```

+   fr <- alphabetFrequency(Lmers, collapse = TRUE)[lets]
+   fr <- fr/sum(fr)
+   sum(co * zlog(co/fr), na.rm = TRUE)
+ }
> informationContent(orf10)

[1] 2.167186

```

12 Exercise Answers

12.1 Exercise 1

1. Using `pairwiseAlignment`, fit the global, local, and overlap pairwise sequence alignment of the strings "syzygy" and "zyzzyx" using the default settings.

```

> pairwiseAlignment("zyzzyx", "syzygy")

Global Pairwise Alignment (1 of 1)
pattern: [1] zzyzyx
subject: [1] syzygy
score: -9.265214

> pairwiseAlignment("zyzzyx", "syzygy", type = "local")

Local Pairwise Alignment (1 of 1)
pattern: [2] yz
subject: [2] yz
score: 15.96347

> pairwiseAlignment("zyzzyx", "syzygy", type = "overlap")

Overlap Pairwise Alignment (1 of 1)
pattern: [1] zyzy
subject: [3] zyg-y
score: 3.638040

```

2. Do any of the alignments change if the `gapExtension` argument is set to `-Inf`? *Yes, the overlap pairwise sequence alignment changes.*

```

> pairwiseAlignment("zyzzyx", "syzygy", type = "overlap",
+   gapExtension = -Inf)

Overlap Pairwise Alignment (1 of 1)
pattern: [1] zyzz
subject: [3] zygy
score: 3.349131

```

12.2 Exercise 2

1. What is the primary benefit of formal summary classes like `PairwiseAlignedFixedSubjectSummary` and `summary.lm` to end-users? *These classes allow the end-user to extract the summary output for further operations.*

```

> ex2 <- summary(pairwiseAlignment("zyzzyx", "syzygy"))
> nmatch(ex2)/nmismatch(ex2)

[1] 0.5

```

12.3 Exercise 3

For the overlap pairwise sequence alignment of the strings "syzygy" and "zyzzyx" with the `pairwiseAlignment` default settings, perform the following operations:

```

> ex3 <- pairwiseAlignment("zyzzyx", "syzygy", type = "overlap")

```

1. Use `nmatch` and `nmismatch` to extract the number of matches and mismatches respectively.

```

> nmatch(ex3)

[1] 3

> nmismatch(ex3)

[1] 1

```

2. Use the `compareStrings` function to get the symbolic representation of the alignment.

```

> compareStrings(ex3)

[1] "zy?+y"

```

3. Use the `as.character` function to get the character string versions of the alignments.

```

> as.character(ex3)

[1] "--zyzy"

```

4. Use the `pattern` function to extract the aligned pattern and apply the `mismatch` function to it to find the locations of the mismatches.

```

> mismatch(pattern(ex3))

[[1]]
[1] 3

```

5. Use the `subject` function to extract the aligned subject and apply the `aligned` function to it to get the aligned strings.

```

> aligned(subject(ex3))

A BStringSet instance of length 1
width seq
[1] 5 zyg-y

```

12.4 Exercise 4

1. Use the `pairwiseAlignment` function to find the Levenshtein edit distance between "syzygy" and "zyzzyx".

```
> submat <- matrix(-1, nrow = 26, ncol = 26, dimnames = list(letters,
+ letters))
> diag(submat) <- 0
> -pairwiseAlignment("zyzzyx", "syzygy", substitutionMatrix = submat,
+ gapOpening = 0, gapExtension = -1, scoreOnly = TRUE)
```

```
[1] 4
```

2. Use the `stringDist` function to find the Levenshtein edit distance for the vector `c("zyzzyx", "syzygy", "succeed", "precede", "supersede")`.

```
> stringDist(c("zyzzyx", "syzygy", "succeed", "precede",
+ "supersede"))
```

```
  1 2 3 4
2  4
3  7 6
4  7 7 5
5  9 8 5 5
```

12.5 Exercise 5

1. Repeat the alignment exercise above using BLOSUM62, a gap opening penalty of -12, and a gap extension penalty of -4.

```
> data(BLOSUM62)
> pairwiseAlignment(AAString("PAWHEAE"), AAString("HEAGAWGHEE"),
+ substitutionMatrix = BLOSUM62, gapOpening = -12, gapExtension = -4)
```

```
Global Pairwise Alignment (1 of 1)
pattern: [1] P---AWHEAE
subject: [1] HEAGAWGHEE
score: -9
```

2. Explore to find out what caused the alignment to change. *The sift in gap penalties favored infrequent long gaps to frequent short ones.*

12.6 Exercise 6

1. Rerun the simulation time using the `simulateReads` function with a `substitutionRate` of 0.005 and `gapRate` of 0.0005. How do the different pairwise sequence alignment methods compare? *The different methods are much more comprobable when the error rates are lower.*

```
> adapter <- DNASTring("GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAAAA")
> set.seed(123)
> N <- 1000
> experiment <- list(side = rbinom(N, 1, 0.5), width = sample(0:36,
+ N, replace = TRUE))
> table(experiment[["side"]], experiment[["width"]])
```

```

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
0 13 10  8  7 11 18  9 15 15 18 16 10 11  9 13 13 18 18 14  9 19 13
1 15 21 21 12 17 14  8 11 12 10 14 16  7 14 19 14 14 16 14 16 16 16

      22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
0 19 19 12  6 15 18 12 15 16 17 19  6 13 18 15
1 13 11  9 11 15 10 10 16 15 15 11  7 12  7 14

> ex6Strings <- simulateReads(N, adapter, experiment, substitutionRate = 0.005,
+   gapRate = 5e-04)
> ex6Strings <- DNAStrngSet(ex6Strings)
> ex6Strings

A DNAStrngSet instance of length 1000
width seq
[1] 36 CTGCTTGAAATTGCACGATAGTTGCATATGCTACAA
[2] 36 ATTTCTCCTTCTCAGGATCGGAAGAGCTCGTATGCC
[3] 36 TGAAAGAAGGTAATTTGATTAAGCCCTTCGCAAAAC
[4] 36 CAAACGATCGGAAGAGCTCGTATGCCGTCTTCTGCT
[5] 36 TCTCAGATCGGAAGAGCTCGTATGCCGTCTTCTGCT
[6] 36 CCGTCTTCTGCTTGAAACGTGAACAGGACAATGGCC
[7] 36 GGAAGCCAGATCGTAAGAGCTCGTATGCCGTCTTCT
[8] 36 CGGGTCTGGTCCTGGGGCCATCCATGATCGGAAGA
[9] 36 TGGCACATCGCAGCTAAATCGACAGTACTATCATGA
...
[992] 36 TTGAAAAATTAGGCCATGGCCACGGCGTATTCAACC
[993] 36 AACATGATCGGAAGAGCTCGTATGCCGTCTTCTGCT
[994] 36 TGAAACATTTCAGCGTAAGCTGCTTAACGTTTAGAC
[995] 36 ACTCGGGATCATCGGAAACGATAAGAACGTTGAGAT
[996] 36 TACGATCGGAAGCGCTCGTATGCCGTCTTCTGCTTG
[997] 36 TCATTGACATTACACAGCCTACTAGGATCGGAAGAG
[998] 36 AGCTCGTATGCCGTCTTCTGCTTGAAACATGTTTCA
[999] 36 CCGTAATTAGTTCCTACAGATCGATCGGAAGAGCTC
[1000] 36 CGTCTTCTGCTTGAAACGGCACACCTCAACGGGGAA

> submat1 <- nucleotideSubstitutionMatrix(match = 0, mismatch = -1,
+   baseOnly = TRUE)
> quantile(randomScores1, seq(0.99, 1, by = 0.001))

99% 99.1% 99.2% 99.3% 99.4% 99.5% 99.6% 99.7% 99.8% 99.9% 100%
-16 -16 -16 -16 -16 -16 -16 -16 -16 -15 -14

> ex6Aligns1 <- pairwiseAlignment(ex6Strings, adapter, substitutionMatrix = submat1,
+   gapOpening = 0, gapExtension = -1)
> table(score(ex6Aligns1) > quantile(randomScores1, 0.999),
+   experiment[["width"]])

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32 17 26 27 28 30 26 18 23 32 27 32 34 28 25 35
TRUE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

```

```

FALSE 29 32 30 21 17 30 28 22 4 0 0 0 0 0 0 0
TRUE 0 0 0 0 0 0 0 0 0 27 31 32 30 13 25 25 29

> submat2 <- nucleotideSubstitutionMatrix(match = 1, mismatch = -Inf,
+     baseOnly = TRUE)
> quantile(randomScores2, seq(0.99, 1, by = 0.001))

 99% 99.1% 99.2% 99.3% 99.4% 99.5% 99.6% 99.7% 99.8% 99.9%
7.000 8.000 8.000 8.000 8.000 8.000 8.000 8.000 9.000 9.001
100%
11.000

> ex6Aligns2 <- pairwiseAlignment(ex6Strings, adapter, substitutionMatrix = submat2,
+     type = "local", gapOpening = 0, gapExtension = -Inf)
> table(score(ex6Aligns2) > quantile(randomScores2, 0.999),
+     experiment[["width"]])

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32 17 26 27 28 1 0 1 0 0 1 0 0 0 0 0
TRUE 0 0 0 0 0 0 0 0 0 0 0 29 26 17 23 32 26 32 34 28 25 35

      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TRUE 29 32 30 21 17 30 28 22 31 31 32 30 13 25 25 29

> table(start(pattern(ex6Aligns2)) > 37 - end(pattern(ex6Aligns2)),
+     experiment[["side"]])

      0  1
FALSE 475 57
TRUE 32 436

> submat3 <- nucleotideSubstitutionMatrix(match = 1, mismatch = -Inf,
+     baseOnly = TRUE)
> ex6Aligns3 <- pairwiseAlignment(ex6Strings, adapter, substitutionMatrix = submat3,
+     type = "overlap", gapOpening = 0, gapExtension = -Inf)
> table(score(ex6Aligns3) > quantile(randomScores3, 0.999),
+     experiment[["width"]])

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32 17 0 1 4 1 0 1 2 0 2 5 1 5 1 2
TRUE 0 0 0 0 0 0 0 26 26 24 29 26 17 21 32 25 27 33 23 24 33

      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE 5 1 3 1 3 0 4 3 5 2 2 1 3 2 2 6
TRUE 24 31 27 20 14 30 24 19 26 29 30 29 10 23 23 23

> table(end(pattern(ex6Aligns3)) == 36, experiment[["side"]])

      0  1
FALSE 479 39
TRUE 28 454

```

```

> quantile(randomScores4, seq(0.99, 1, by = 0.001))

      99%      99.1%      99.2%      99.3%      99.4%      99.5%      99.6%
7.927024 7.927024 7.927024 7.927024 7.927024 7.927024 7.973007
      99.7%      99.8%      99.9%      100%
9.908780 9.908780 9.908780 11.890536

> ex6Aligns4 <- pairwiseAlignment(ex6Strings, adapter, type = "overlap")
> table(score(ex6Aligns4) > quantile(randomScores4, 0.999),
+       experiment[["width"]])

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 32  1  0  1  0  0  0  0  0  0  0  0  0  0  0
TRUE   0  0  0  0  0  0 16 26 26 28 30 26 18 23 32 27 32 34 28 25 35

      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TRUE  29 32 30 21 17 30 28 22 31 31 32 30 13 25 25 29

> table(end(pattern(ex6Aligns4)) == 36, experiment[["side"]])

      0  1
FALSE 486 17
TRUE   21 476

```

2. (Advanced) Modify the `simulateReads` function to accept different equal length adapters on either side (left & right) of the reads. How would the methods for trimming the reads change?

```

> simulateReads <- function(N, left, right = left, experiment,
+ substitutionRate = 0.01, gapRate = 0.001) {
+   leftChars <- strsplit(as.character(left), "")[[1]]
+   rightChars <- strsplit(as.character(right), "")[[1]]
+   if (length(leftChars) != length(rightChars))
+     stop("left and right adapters must have the same number of characters")
+   nChars <- length(leftChars)
+   sapply(seq_len(N), function(i) {
+     width <- experiment[["width"]][i]
+     side <- experiment[["side"]][i]
+     randomLetters <- function(n) sample(DNA_ALPHABET[1:4],
+     n, replace = TRUE)
+     randomLettersWithEmpty <- function(n) sample(c("",
+     DNA_ALPHABET[1:4]), n, replace = TRUE, prob = c(1 -
+     gapRate, rep(gapRate/4, 4)))
+     if (side) {
+       value <- paste(iffelse(rbinom(nChars, 1, substitutionRate),
+       randomLetters(nChars), rightChars), randomLettersWithEmpty(nChars),
+       sep = "", collapse = "")
+       value <- paste(c(randomLetters(36 - width),
+       substring(value, 1, width)), sep = "",
+       collapse = "")
+     }
+     else {
+       value <- paste(iffelse(rbinom(nChars, 1, substitutionRate),

```

```

+         randomLetters(nChars), leftChars), randomLettersWithEmpty(nChars),
+         sep = "", collapse = "")
+     value <- paste(c(substring(value, 37 - width,
+         36), randomLetters(36 - width)), sep = "",
+         collapse = "")
+     }
+     value
+ })
+ }
> leftAdapter <- adapter
> rightAdapter <- reverseComplement(adapter)
> ex6LeftRightStrings <- simulateReads(N, leftAdapter, rightAdapter,
+     experiment)
> ex6LeftAligns4 <- pairwiseAlignment(ex6LeftRightStrings,
+     leftAdapter, type = "overlap")
> ex6RightAligns4 <- pairwiseAlignment(ex6LeftRightStrings,
+     rightAdapter, type = "overlap")
> scoreCutoff <- quantile(randomScores4, 0.999)
> leftAligned <- start(pattern(ex6LeftAligns4)) == 1 & score(ex6LeftAligns4) >
+     pmax(scoreCutoff, score(ex6RightAligns4))
> rightAligned <- end(pattern(ex6RightAligns4)) == 36 & score(ex6RightAligns4) >
+     pmax(scoreCutoff, score(ex6LeftAligns4))
> table(leftAligned, rightAligned)

           rightAligned
leftAligned FALSE TRUE
      FALSE   171  392
      TRUE    437   0

> table(leftAligned | rightAligned, experiment[["width"]])

           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
FALSE 28 31 29 19 28 31  1  2  1  1  0  0  0  0  0  0  0  0  0  0  0
TRUE   0  0  0  0  0  1 16 24 26 27 30 26 18 23 32 27 32 34 28 25 35

           21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TRUE  29 32 30 21 17 30 28 22 31 31 32 30 13 25 25 29

```

12.7 Exercise 7

1. Rerun the “Subject Overlap” alignment of the short reads against the entire genome. (This may take a few minutes.)

```

> genBankFullAlign <- pairwiseAlignment(srPhiX174, genBankPhage,
+     patternQuality = SolexaQuality(quPhiX174), subjectQuality = SolexaQuality(99L),
+     type = "subjectOverlap")
> summary(genBankFullAlign, weight = wtPhiX174)

```

```

Subject Overlap Pairwise Alignment
Number of Alignments: 53802

```


12.8 Exercise 8

1. Rerun the first set of profiling code, but this time fix the number of characters in `string1` to 35 and have the number of characters in `string2` range from 5000, 50000, by increments of 5000. What is the computational order of this simulation exercise? *As expected, the growth in time is now linear.*

```
> N <- as.integer(seq(5000, 50000, by = 5000))
> newTimings <- rep(0, length(N))
> names(newTimings) <- as.character(N)
> for (i in seq_len(length(N))) {
+   string1 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   35, replace = TRUE), collapse = ""))
+   string2 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   N[i], replace = TRUE), collapse = ""))
+   newTimings[i] <- system.time(pairwiseAlignment(string1,
+   string2, type = "global"))[["user.self"]]
+ }
> newTimings

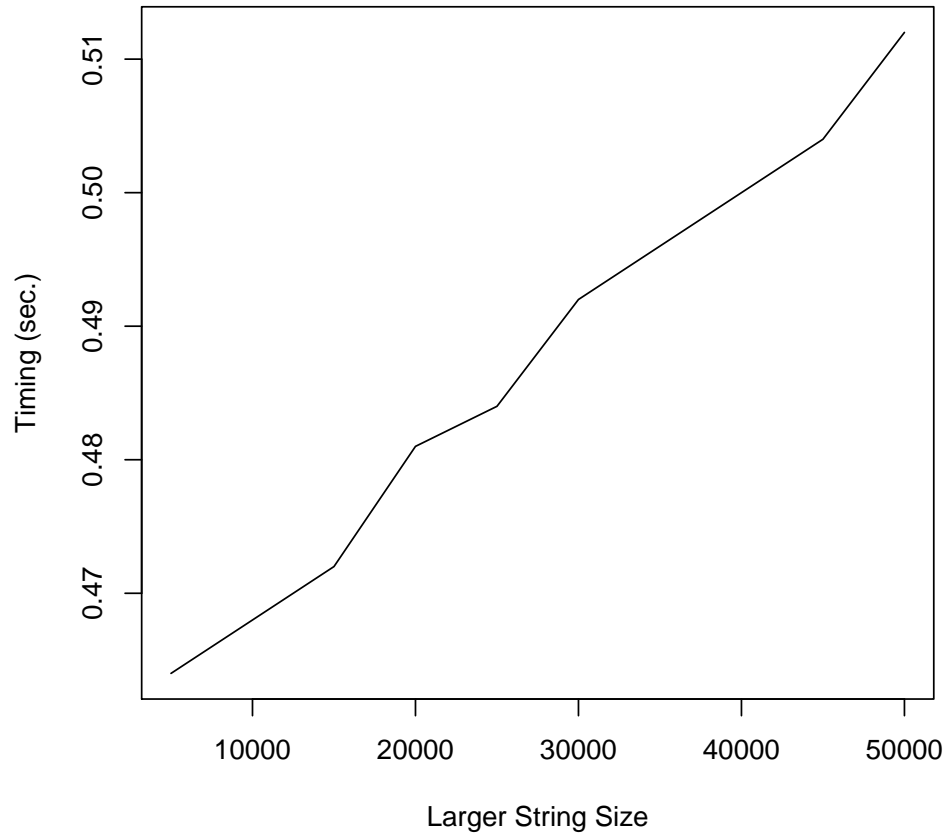
 5000 10000 15000 20000 25000 30000 35000 40000 45000 50000
0.456 0.468 0.484 0.757 0.476 0.492 0.492 0.500 0.504 0.512

> coef(summary(lm(newTimings ~ poly(N, 2))))

              Estimate Std. Error    t value    Pr(>|t|)
(Intercept) 0.514100000 0.02991528 17.18519699 5.546701e-07
poly(N, 2)1 0.003137747 0.09460043  0.03316842 9.744662e-01
poly(N, 2)2 -0.074070043 0.09460043 -0.78297790 4.593051e-01

> plot(N, newTimings, xlab = "Larger String Size", ylab = "Timing (sec.)",
+   type = "l", main = "Global Pairwise Sequence Alignment Timings")
```

Global Pairwise Sequence Alignment Timings



2. Rerun the second set of profiling code using the simulations from the previous exercise with `scoreOnly` argument set to `TRUE`. Is it still twice as fast? *Yes, it is still over twice as fast.*

```
> newScoreOnlyTimings <- rep(0, length(N))
> names(newScoreOnlyTimings) <- as.character(N)
> for (i in seq_len(length(N))) {
+   string1 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   35, replace = TRUE), collapse = ""))
+   string2 <- DNASTring(paste(sample(DNA_ALPHABET[1:4],
+   N[i], replace = TRUE), collapse = ""))
+   newScoreOnlyTimings[i] <- system.time(pairwiseAlignment(string1,
+   string2, type = "global", scoreOnly = TRUE))["user.self"]
+ }
> newScoreOnlyTimings

 5000 10000 15000 20000 25000 30000 35000 40000 45000 50000
0.444 0.448 0.448 0.460 0.456 0.456 0.460 0.464 0.468 0.468

> round((newTimings - newScoreOnlyTimings)/newTimings, 2)
```

```
5000 10000 15000 20000 25000 30000 35000 40000 45000 50000
0.04 0.04 0.05 0.04 0.06 0.07 0.07 0.07 0.07 0.09
```

13 Session Information

All of the output in this vignette was produced under the following conditions:

```
> sessionInfo()
```

```
R version 2.8.1 (2008-12-22)
x86_64-unknown-linux-gnu
```

```
locale:
```

```
LC_CTYPE=en_US;LC_NUMERIC=C;LC_TIME=en_US;LC_COLLATE=en_US;LC_MONETARY=C;LC_MESSAGES=en_US;LC_PAPER=en_US
```

```
attached base packages:
```

```
[1] tools      stats      graphics  grDevices  utils      datasets
[7] methods    base
```

```
other attached packages:
```

```
[1] Biostrings_2.10.22 IRanges_1.0.16
```

```
loaded via a namespace (and not attached):
```

```
[1] grid_2.8.1      lattice_0.17-20  Matrix_0.999375-23
```

References

- [1] Durbin, R., Eddy, S., Krogh, A., and Mitchison G. *Biological Sequence Analysis*. Cambridge UP 1998, sec 2.3.
- [2] Haubold, B. and Wiehe, T. *Introduction to Computational Biology*. Birkhauser Verlag 2006, Chapter 2.
- [3] Malde, K. The effect of sequence quality on sequence alignment. *Bioinformatics*, 24(7):897-900, 2008.
- [4] Smith, H.; Hutchison, C.; Pfannkoch, C.; and Venter, C. Generating a synthetic genome by whole genome assembly: {phi}X174 bacteriophage from synthetic oligonucleotides. *Proceedings of the National Academy of Sciences*, 100(26): 15440-15445, 2003.