Pairwise Sequence Alignments

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

August 14, 2008

Contents

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 constant 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 $j \in \{1, \ldots, n_i\}$. 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

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

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 constant for character pairings or quality-dependent for character pairings. (Characters that align with a gap are penalized according to the "Gap Penalty" framework.)

Constant substitution scoring - Constant 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, 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 when two characters match is given by $b * \log_2((1-\epsilon_c) * n)$ and the substitution score when two characters don't match is given by $b * \log_2(\epsilon_c * (n/(n-1)))$, where b is the bit-scaling for the scoring.

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: [6] ed
subject: [7] ed
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] succ-e--ed
subject: [1] su--persed
score: 33.90868
```

The substitution scoring scheme is set using four arguments, three of which are quality-based related (*patternQuality*, *subjectQuality*, *qualityType*) and one is constant substitution related (*substitutionMatrix*). When the substitution scores are fixed by character pairing, the *substituionMatrix* argument takes a matrix with the appropriate alphabets as dimension names. The `nucleotideSubstitutionMatrix` function tranlates 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 *qualityType* argument sets the type of quality score to use (`"Phred"` or `"Solexa"`) and the *patternQuality* and *subjectQuality* arguments accept the equivalent of $[x - 99]$ numeric quality values for the respective strings. For `"Phred"` quality measures $Q \in [0, 99]$, the probability of an error in the base read is given by $10^{-Q/10}$ and for `"Solexa"` quality 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.

```
> 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: *PairwiseAlignment*, *PairwiseAlignmentSummary*, *AlignedXStringSet*, and *QualityAlignedXStringSet*. As the names suggest the *PairwiseAlignment* class holds the results of a fit from the `pairwiseAlignment` function

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

```
[1] "PairwiseAlignment"
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] "PairwiseAlignmentSummary"
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 constant 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 *PairwiseAlignmentSummary* 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 *PairwiseAlignment* and *AlignedXStringSet* respectively. These functions should be used in preference to direct slot extraction from the alignment objects.

Function	Description
`[`	Extracts the specified elements of the alignment object
`alphabet`	Extracts the allowable characters in the original strings
`as.character`	Converts the alignments to character strings
`consmat`	Computes a consensus matrix for the alignments
`compareStrings`	Creates character string mashups of the alignments
`coverage`	Computes the alignment coverage along the subject
`length`	Extracts the number of patterns aligned
`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
`pattern`, `subject`	Extracts the aligned pattern/subject
`rep`	Replicates the elements of the alignment object
`score`	Extracts the pairwise sequence alignment scores
`summary`	Summarizes a pairwise sequence alignment
`type`	Extracts the type of pairwise sequence alignment
`views`	Extracts the alignment ranges for the subject

Table 1: Functions for *PairwiseAlignment* 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)

[1] 3 3

> nchar(psa2)

[1] 8 9
```

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               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

> mismatchTable(psa2)

  PatternId PatternStart PatternEnd PatternSubstring
1         1            3          3                c
2         1            4          4                c
3         1            5          5                e
4         2            1          1                p
```

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

The `pattern` and `subject` functions extract the aligned pattern and subject objects for further analysis. Most of the actions that can be performed on *PairwiseAnalysis* objects can also be performed on *AlignedXString* and *QualityAlignedXString* 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(subject(psa2))
```

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
`quality`	Extracts the quality scores for a *QualityAlignedXString*
`rep`	Replicates the elements of the alignment object
`start`	Extracts the starting index of the aligned range
`width`	Extracts the width of the aligned range

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

```
     Length WidthSum
[1,]      0        0
[2,]      0        0

> 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 `nmismath` 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 the 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.

```
> 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] P-AW-HEAE
subject: [3] AGAWGHE-E
score: 1

> compareStrings(nwdemo)

[1] "?-AW-HE+E"
```

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
  0 13 10  8  7 11 18  9 15 15 18 16 10 11  9 13 13 18 18
  1 15 21 21 12 17 14  8 11 12 10 14 16  7 14 19 14 14 16

    18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
  0 14  9 19 13 19 19 12  6 15 18 12 15 16 17 19  6 13 18
  1 14 16 16 16 13 11  9 11 15 10 10 16 15 15 11  7 12  7

    36
  0 15
  1 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 TTTAAACTACTGGAATAAATGCAAGTGGACAAACGC
 [993]    36 TGGCAGATCGGAAGAGCTCGTATGCCGTCTTCAGCT
 [994]    36 TGAAATATGTCATCTCATACAAGCACGTACTCATTG
 [995]    36 CTCCGGTACACGCCTCGGTGCACACATAATTGGGAT
 [996]    36 GTTGATCGGAAGAGCTCGTATGCCGTCTTCTGCTTG
 [997]    36 AATGTGATGTCTCACTTCAAAGGCGGATCGGAAGAG
 [998]    36 AGCTCGTATGCCGTCTTCTGCTTGAAAAAATTATTC
 [999]    36 ACTAACTGCACTCCCGCACCATGATCGGAAGAGCTC
[1000]    36 CGTCTTCTGCTTGAAAATCAGGTGTTGGGCCTGCCG
```

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 <- 10000
> randomStrings <- apply(matrix(sample(DNA_ALPHABET[1:4],
+     36 * M, replace = TRUE), nrow = M), 1, paste,
+     collapse = "")
> randomStrings <- DNAStringSet(randomStrings)
> randomStrings
```

```
  A DNAStringSet instance of length 10000
        width seq
```

```
    [1]    36 AGGGTGTGATGATGATTACTAGCGTCGCGGAATAGA
    [2]    36 CTTGCCCAATACTAGCACAAGGTTAGTGGAAGCGTT
    [3]    36 TATTGTAGATTGGAAGGCCTAATGACGTCATTTATA
    [4]    36 TATGCTGAGGTAGCAAACGGGGGGCTTAACATCGTA
    [5]    36 TTGAATTGATTCTAGCCTCTAGGCGCTGCCCTAAAC
    [6]    36 TTTGCATCGCTCGGAGGGACGCACACCTGATACTCC
    [7]    36 CTCGAAGACACGCTGACTGTGTAACTGCGTAGTCAG
    [8]    36 TTTGGAGATTAAACGCAGTGCTTCCTAATATATTCG
    [9]    36 TCGTCCCCTAGGGTTTATATCCGGACGACACTCGAC
    ...   ... ...
 [9992]    36 CTAGCTACAATGCATACACTCGGGCATTTATGCGAA
 [9993]    36 AACGCGAAAAGGTAAAGGGACTAAGAGATTCAATTT
 [9994]    36 CGCGGGTTCGTATAGGCGTCCACGGGAGAGCAAATA
 [9995]    36 AAATACAGCCTTCAAATTGCCATTGGCTGGTGCATT
 [9996]    36 TGTTTGAGAGCTACCCCCAGCAGTAATGGCCCCTGA
 [9997]    36 CGGAAGCTATACCACTCAGGATACAGAAGCCAGATA
 [9998]    36 AGGACATTCTCAAACATTCCCCGCATAATGGGTGTG
 [9999]    36 TGGTGAGTCGAGCCTGCCAACTGGGTCGCGCGAGAA
[10000]    36 TCAACACGGGGCAGAGTTCTCTTCGCCCTCATTATC
```

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%
  -17   -16   -16   -16   -16   -16   -16   -16   -15   -15
 100%
  -13
```

```
> 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
  FALSE 28 31 29 19 28 32 17 26 27 28 30 26 18 23 32 27 32
  TRUE   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE 34 28 25 35 29 32 30 21 17 30 28 22  5  1  0  0  0
  TRUE   0  0  0  0  0  0  0  0  0  0  0  0 26 30 32 30 13

        34 35 36
  FALSE  0  0  0
  TRUE  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     7     8     8     8     8     8     8     9     9
 100%
   11
```

```
> 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
  FALSE 28 31 29 19 28 32 17 26 27 28  2  0  0  1  2  0  1
  TRUE   0  0  0  0  0  0  0  0  0  0 28 26 18 22 30 27 31

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  TRUE  34 27 24 35 29 32 30 21 17 30 28 22 31 31 32 30 13

        34 35 36
  FALSE  0  0  0
  TRUE  25 25 29
```

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

```
         0   1
  FALSE 452  45
  TRUE   55 448
```

Limiting consecutive matches to the ends provides better results, but it doesn't revolve 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%
4.000 4.000 4.000 4.000 4.000 4.000 5.000 5.000 5.000 5.001
 100%
9.000
```

```
> 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
  FALSE 28 31 29 19 28 32  2  1  0  3  3  0  0  2  3  6  3
  TRUE   0  0  0  0  0  0 15 25 27 25 27 26 18 21 29 21 29

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

        34 35 36
  FALSE 11  5 10
  TRUE  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%
 7.927024  7.927024  7.927024  7.927024  7.927024  7.927024
    99.6%     99.7%     99.8%     99.9%      100%
 7.927208  9.908780  9.908780  9.908826 17.835804

> 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
  FALSE 28 31 29 19 28 32  2  1  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

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  TRUE  34 28 25 35 29 32 30 21 17 30 28 22 31 31 32 30 13

        34 35 36
  FALSE  0  0  0
  TRUE  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 TGAAAGAAGGTAATTTGATTAAGCCCTTCGCAAAAC
    [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 TGAAATATGTCATCTCATACAAGCACGTACTCATTG
  [995]    36 CTCCGGTACACGCCTCGGTGCACACATAATTGGGAT
  [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 a Coliphage phiX174. 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`.

```
> data(phiX174)
> nchar(phiX174)

[1] 5386

> data(srPhiX174)
> srPhiX174

  A DNAStringSet instance of length 4915
       width seq
   [1]    35 TAATGTTTATGTTGGGTTCTTGGTTTGTTATAACT
   [2]    35 GTTATTATACCGTCAAGGACTGTGTGACTATTGAC
   [3]    35 GGTGGTTATTATACCGTCAAGGACTGTGTGACTAT
   [4]    35 TACCGTCAAGGACTGTGTGACTATTGACGTCCTTC
   [5]    35 GTACGCCGGGCAATAATGTTTATGTTGGTTTCATG
   [6]    35 GGTTTCATGGTTTGGTCTAACTTTACCGCTACTAA
   [7]    35 GGGCAATAATGTTTATGTTGGTTTCATGGTTTGGT
   [8]    35 GTCCTTCCTCGTACGCCGGGCAATAATGTTTATGT
   [9]    35 GTTGGTTTCATGGTTTGGTCTAACTTTACCGCTAC
   ...   ... ...
[4907]    35 TCTTCCTCGTACGCCGGGCAATAATGTTTATGTTG
[4908]    35 GCAATACTGTTTATGTTGGTTTTATGGTTTGGTCT
[4909]    35 GTACGCCGGGCAATAATGTTTATGTTGGTTTCGTG
[4910]    35 TGACTATTGACGTCCTTCCTCCTACGCCGGGCAAT
[4911]    35 ATACCGTCAAGGACTGTGTGACTATTGACTTCATT
[4912]    35 ACGTCCTTCCTCGTACGCCGAGCAATAATGTTTAT
[4913]    35 ACTGTGTGACTATTGACCTCCTTCCTCCTACTTCT
[4914]    35 ATAATGTTTATGTTGGTTTCATGGTTTTGTCTTAT
[4915]    35 ATCGTACGCCGGGCAATAATGTTTATGTTGGTTTC

> quPhiX174

  A BStringSet instance of length 4915
       width seq
   [1]    35 YYYYZYYYZTYTYCYJYYJJYJTPPYIDIAYYYGY
   [2]    35 ZYZZZZZZZZZYYZZYYYYYYYYYYYYYYYYYQYY
   [3]    35 ZZYZZYZZZZYYYYYYYYYYYYYYYYYYVYYYTY
   [4]    35 ZZZYZYYZYYZYYZYYYYYYYYYYYYYYYVYYYYY
   [5]    35 ZZYZZZZZZZZZYZTYYYYYYYYYYYYYYYYYNYT
   [6]    35 ZZZZZZYZYYZZZYYYYYYYYYYYYYYYYYSYYSY
   [7]    35 ZZZZZYZYYYZYYZYYYYYYYYYYYYSYQVYYYASY
   [8]    35 ZZZZZZZZZZYZZZZYYYYYYYYYYYYYYYYYYYY
   [9]    35 YYZYYZYYZYYYYZYYVQYYYYYYYYYYYYTYYYY
   ...   ... ...
[4907]    35 TYZZZZZYYYZYYZZYYYYYYYYYYYYYYYYRYYM
```

```
[4908]    35 QVSMSVSVVVSVPQSVVLGLVSFGK@ASAUFFJNJ
[4909]    35 YZZZYZYYYYYYZYYYYYYVQYYYYYYYYYYYAYQ
[4910]    35 VVVXVVVNSSXVVSVVVVVQVHPVVPUUGGJSAFG
[4911]    35 XXXSLQXSVVVVQQMVVQVSVAQVVUPAUGSQ@SN
[4912]    35 ZYZZYYYZYZZYYYYYYYYYGYYYDYYSYYYYYPY
[4913]    35 NXQSQNXVNVXQNVVVVHVAHVLKFUGLGFFJJDJ
[4914]    35 ZZZZZZZZZZZZYZZVYYYYVYYJQYYYIJYAYAYA
[4915]    35 YYYZYZYYYZYYQYYVYYYYJNYYYYYYYQSSYYY

> summary(wtPhiX174)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    1.00    1.00   11.72    1.00  965.00

> fullShortReads <- rep(srPhiX174, wtPhiX174)
> srPDict <- PDict(fullShortReads)
> table(countPDict(srPDict, phiX174))

    0     1
40811 16793
```

For these short reads, the `pairwiseAlignment` function finds that the small number of perfect matches is due to two locations on the Coliphage phiX174 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 Coliphage phiX174 genome is supplied to the *subject* argument of the `pairwiseAlignment` function to reduce the computation time.

```
> phiX174Substring <- substring(phiX174, 2793 -
+     34, 2811 + 34)
> alignPhiX174 <- pairwiseAlignment(srPhiX174, phiX174Substring,
+     patternQuality = quPhiX174, subjectQuality = 99L,
+     qualityType = "Solexa", type = "subjectOverlap")
> summary(alignPhiX174, weight = wtPhiX174)

Subject Overlap Pairwise Alignment
Number of Alignments:  57604

Scores:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -46.44   32.52   49.96   40.73   59.45   69.77

Number of matches:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  21.00   30.00   33.00   31.35   34.00   35.00

Top 10 Mismatch Counts:
    SubjectPosition Subject Pattern Count Probability
158              53       C       T 24619  0.95474288
104              35       C       T 24305  0.99700550
227              76       G       T  2300  0.10851616
```

```
206              69       A       T  1487  0.06036127
236              79       C       T  1459  0.07689065
171              58       A       C  1249  0.04815329
186              63       G       A  1169  0.04585393
200              67       T       G  1167  0.04539796
213              72       G       A  1163  0.05027450
241              81       A       G  1152  0.06501863
```

```
> splitchars <- strsplit(as.character(phiX174),
+     "")[[1]]
> splitchars[c(2793, 2811)] <- "T"
> phiX174Revised <- DNAString(paste(splitchars,
+     collapse = ""))
> table(countPDict(srPDict, phiX174Revised))

    0     1
10570 47034
```

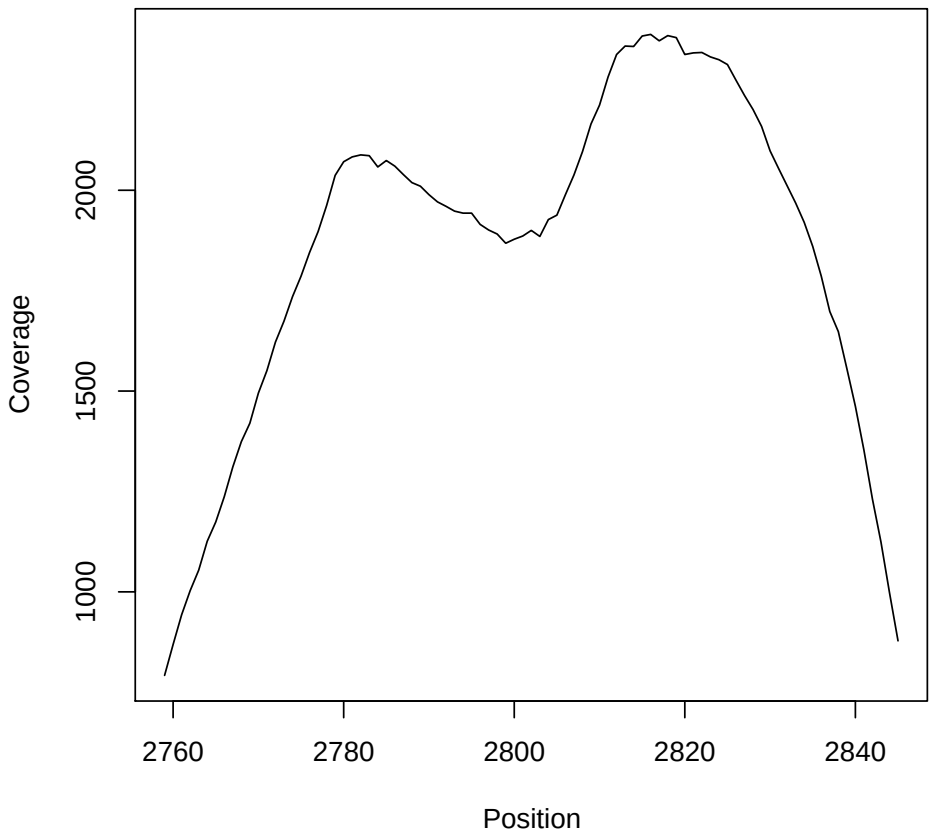
The following plot shows the coverage of the aligned short reads along the substring of the Coliphage phiX174 genome. Applying the `slice` function to the coverage shows the entire substring is covered by aligned short reads.

```
> coveragePhiX174 <- coverage(alignPhiX174, weight = wtPhiX174)
> plot((2793 - 34):(2811 + 34), coveragePhiX174,
+     xlab = "Position", ylab = "Coverage", type = "l")
> nchar(phiX174Substring)

[1] 87

> slice(coveragePhiX174, 0, includeLower = FALSE)

NormalIRanges object:
  start end width
1     1  87    87
```



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 Coliphage phiX174 genome. Do any short reads have a higher alignment score on this new sequence than on the orignal 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.

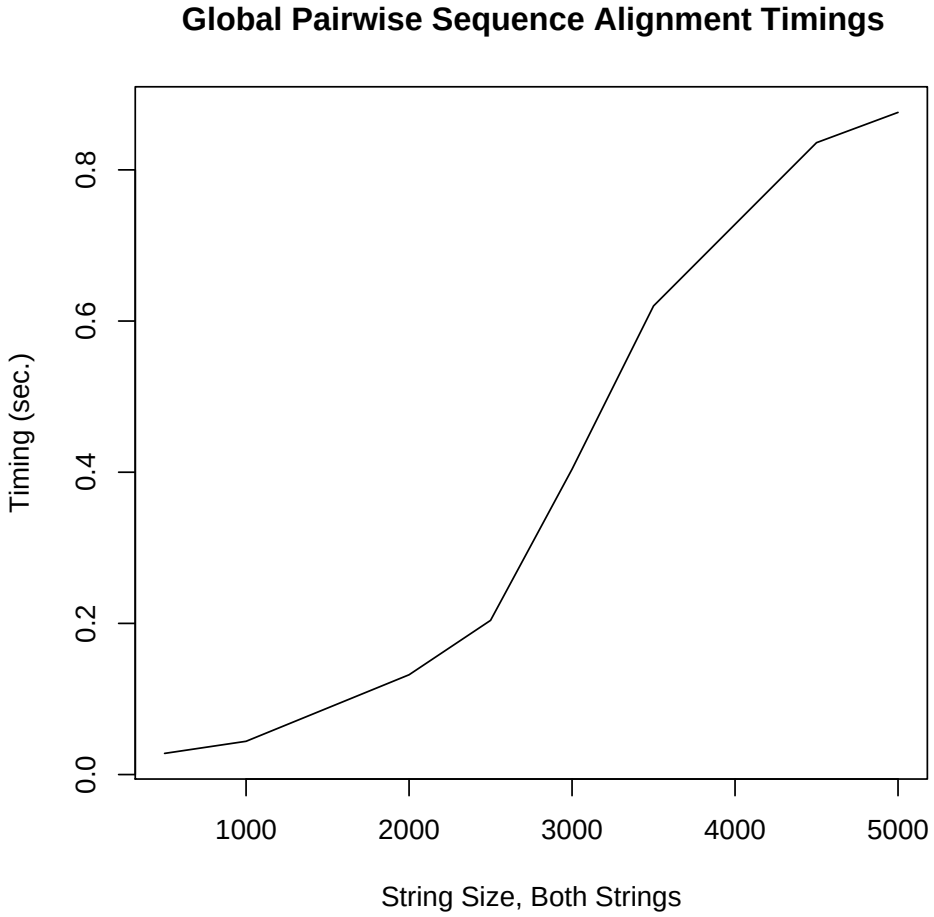
```
> N <- as.integer(seq(500, 5000, by = 500))
> timings <- rep(0, length(N))
> names(timings) <- as.character(N)
> for (i in seq_len(length(N))) {
+     string1 <- DNAString(paste(sample(DNA_ALPHABET[1:4],
+         N[i], replace = TRUE), collapse = ""))
+     string2 <- DNAString(paste(sample(DNA_ALPHABET[1:4],
+         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
0.024 0.048 0.084 0.136 0.312 0.412 0.624 0.740 0.844 0.848

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

              Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 0.40720000 0.02217497 18.363048 3.518801e-07
poly(N, 2)1 0.98161929 0.07012340 13.998455 2.247885e-06
poly(N, 2)2 0.08843145 0.07012340  1.261083 2.476806e-01

> plot(N, timings, xlab = "String Size, Both Strings",
+     ylab = "Timing (sec.)", type = "l", main = "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
0.016 0.028 0.044 0.076 0.112 0.156 0.204 0.264 0.332 0.400
> round((timings - scoreOnlyTimings)/timings, 2)
 500 1000 1500 2000 2500 3000 3500 4000 4500 5000
0.43 0.36 0.50 0.42 0.45 0.61 0.67 0.64 0.60 0.54
```

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 is still twice as fast?

[Answers provided in section 12.8.]

11 Computing alignment consensus matrices

The `consmat` 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 ACTTGTAAATATA...TTATTGTTGATAT YAL001C TFC3 SGDI...
[2]  5825 TTCCAAGGCCGAT...TTTCTATTCTCTT YAL002W VPS8 SGDI...
[3]  2987 CTTCATGTCAGCC...GTAGCTGCCTCAT YAL003W EFB1 SGDI...
[4]  3929 CACTCATATCGGG...CACGAAAAAGTAC YAL005C SSA1 SGDI...
[5]  2648 AGAGAAAGAGTTT...TGTGTGAACATAG YAL007C ERP2 SGDI...
[6]  2597 GTGTCCGGGCCTC...AGAATGTACTTTT YAL008W FUN14 SGD...
[7]  2780 CAAGATAATGTCA...AAAAAAAAATCAC YAL009W SPO7 SGDI...

> orf10 <- DNAStringSet(orf, end = 10)
> consmat(orf10)

      pos
letter         1         2         3         4         5
     A 0.2857143 0.2857143 0.2857143 0.0000000 0.5714286
     C 0.4285714 0.1428571 0.2857143 0.2857143 0.2857143
     G 0.1428571 0.1428571 0.1428571 0.2857143 0.1428571
     T 0.1428571 0.4285714 0.2857143 0.4285714 0.0000000
      pos
letter         6         7         8         9        10
     A 0.4285714 0.4285714 0.4285714 0.2857143 0.1428571
     C 0.1428571 0.0000000 0.0000000 0.2857143 0.4285714
     G 0.0000000 0.4285714 0.4285714 0.1428571 0.2857143
     T 0.4285714 0.1428571 0.1428571 0.2857143 0.1428571
```

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

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

```
+     fr <- colSums(alphabetFrequency(Lmers)[, lets])
+     fr <- fr/sum(fr)
+     sum(co * zlog(co/fr))
+ }
> infContent(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] zyzzyx
subject: [1] syzygy
score: -9.265214

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

Local Pairwise Alignment (1 of 1)
pattern: [4] zy
subject: [3] zy
score: 15.96347

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

Overlap Pairwise Alignment (1 of 1)
pattern: [1] zyzzy
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 *PairwiseAlignmentSummary* 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 `nmismath` 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 the get the character string versions of the alignments.

   ```
   > as.character(ex3)

           [,1]
   pattern "zyzzy"
   subject "zyg-y"
   ```

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.*

```
> 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
  0 13 10  8  7 11 18  9 15 15 18 16 10 11  9 13 13 18 18
  1 15 21 21 12 17 14  8 11 12 10 14 16  7 14 19 14 14 16

    18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
  0 14  9 19 13 19 19 12  6 15 18 12 15 16 17 19  6 13 18
  1 14 16 16 16 13 11  9 11 15 10 10 16 15 15 11  7 12  7

    36
  0 15
  1 14
```

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

```
  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 TTGAAAAATTAGGCCATGGCCACGGCGTATTCAACC
  [993]    36 AACATGATCGGAAGAGCTCGTATGCCGTCTTCTGCT
  [994]    36 TGAAACATTCAGCGTAAGCTGCTTAACGGTTTAGAC
  [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%
  -17   -16   -16   -16   -16   -16   -16   -16   -15   -15
 100%
  -13
```

```
> 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
  FALSE 28 31 29 19 28 32 17 26 27 28 30 26 18 23 32 27 32
  TRUE   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE 34 28 25 35 29 32 30 21 17 30 28 22  4  0  0  0  0
  TRUE   0  0  0  0  0  0  0  0  0  0  0  0 27 31 32 30 13

        34 35 36
  FALSE  0  0  0
  TRUE  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     7     8     8     8     8     8     8     9     9
 100%
   11
```

```
> 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
  FALSE 28 31 29 19 28 32 17 26 27 28  1  0  1  0  0  1  0
  TRUE   0  0  0  0  0  0  0  0  0  0 29 26 17 23 32 26 32

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  TRUE  34 28 25 35 29 32 30 21 17 30 28 22 31 31 32 30 13

        34 35 36
  FALSE  0  0  0
  TRUE  25 25 29
```

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

```
         0   1
  FALSE 464  38
  TRUE   43 455
```

```
> 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
  FALSE 28 31 29 19 28 32  1  0  1  4  1  0  1  2  0  2  5
  TRUE   0  0  0  0  0  0 16 26 26 24 29 26 17 21 32 25 27

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE  1  5  1  2  5  1  3  1  3  0  4  3  5  2  2  1  3
  TRUE  33 23 24 33 24 31 27 20 14 30 24 19 26 29 30 29 10

        34 35 36
  FALSE  2  2  6
  TRUE  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%
 7.927024  7.927024  7.927024  7.927024  7.927024  7.927024
    99.6%     99.7%     99.8%     99.9%      100%
 7.927208  9.908780  9.908780  9.908826 17.835804
```

```
> 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
  FALSE 28 31 29 19 28 32  1  0  1  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

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  TRUE  34 28 25 35 29 32 30 21 17 30 28 22 31 31 32 30 13

        34 35 36
  FALSE  0  0  0
  TRUE  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(ifelse(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(ifelse(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
  FALSE 28 31 29 19 28 31  1  2  1  1  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

        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  FALSE  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  TRUE  34 28 25 35 29 32 30 21 17 30 28 22 31 31 32 30 13

        34 35 36
  FALSE  0  0  0
  TRUE  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.)

```
> fullAlignPhiX174 <- pairwiseAlignment(srPhiX174,
+     phiX174, patternQuality = quPhiX174, subjectQuality = 99L,
+     qualityType = "Solexa", type = "subjectOverlap")
> summary(fullAlignPhiX174, weight = wtPhiX174)

Subject Overlap Pairwise Alignment
Number of Alignments:  57604

Scores:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -45.39   54.78   59.78   59.89   69.50   69.85

Number of matches:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  21.00   33.00   34.00   33.87   35.00   35.00

Top 10 Mismatch Counts:
    SubjectPosition Subject Pattern Count Probability
272            2811       C       T 24612  0.99817496
218            2793       C       T 24296  0.99708622
341            2834       G       T  2293  0.11469588
344            2835       G       T   790  0.04075736
326            2829       G       T   640  0.02836377
```

```
356            2839       A       T   476  0.02879961
185            2782       G       T   433  0.01728474
320            2827       A       T   317  0.01353313
350            2837       C       T   297  0.01670510
258            2807       A       C   266  0.01126689
```

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? *Yes, there are some alignments outside of the specified substring region.*

```
> fullCoveragePhiX174 <- coverage(fullAlignPhiX174,
+     weight = wtPhiX174)
> plot(fullCoveragePhiX174, xlab = "Position", ylab = "Coverage",
+     type = "l")
> slice(fullCoveragePhiX174, 0, includeLower = FALSE)
```

```
NormalIRanges object:
   start  end width
1    177  211    35
2    228  262    35
3    550  584    35
4   1028 1073    46
5   1195 1230    36
6   1351 1385    35
7   1651 1685    35
8   1933 1967    35
9   2513 2572    60
10  2602 2636    35
11  2745 2860   116
12  2929 2963    35
13  3209 3248    40
14  3814 3846    33
15  3964 3998    35
16  4161 4198    38
17  4559 4593    35
```

3. Use the `reverseComplement` function on the Coliphage phiX174 genome. Do any short reads have a higher alignment score on this new sequence than on the orignal sequence? *Yes, there are some strings with a higher score on the new sequence.*

```
> fullAlignRevCompPhiX174 <- pairwiseAlignment(srPhiX174,
+     reverseComplement(phiX174), patternQuality = quPhiX174,
+     subjectQuality = 99L, qualityType = "Solexa",
+     type = "subjectOverlap")
> table(score(fullAlignRevCompPhiX174) > score(fullAlignPhiX174))
```

```
FALSE  TRUE
 4905    10
```

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.*

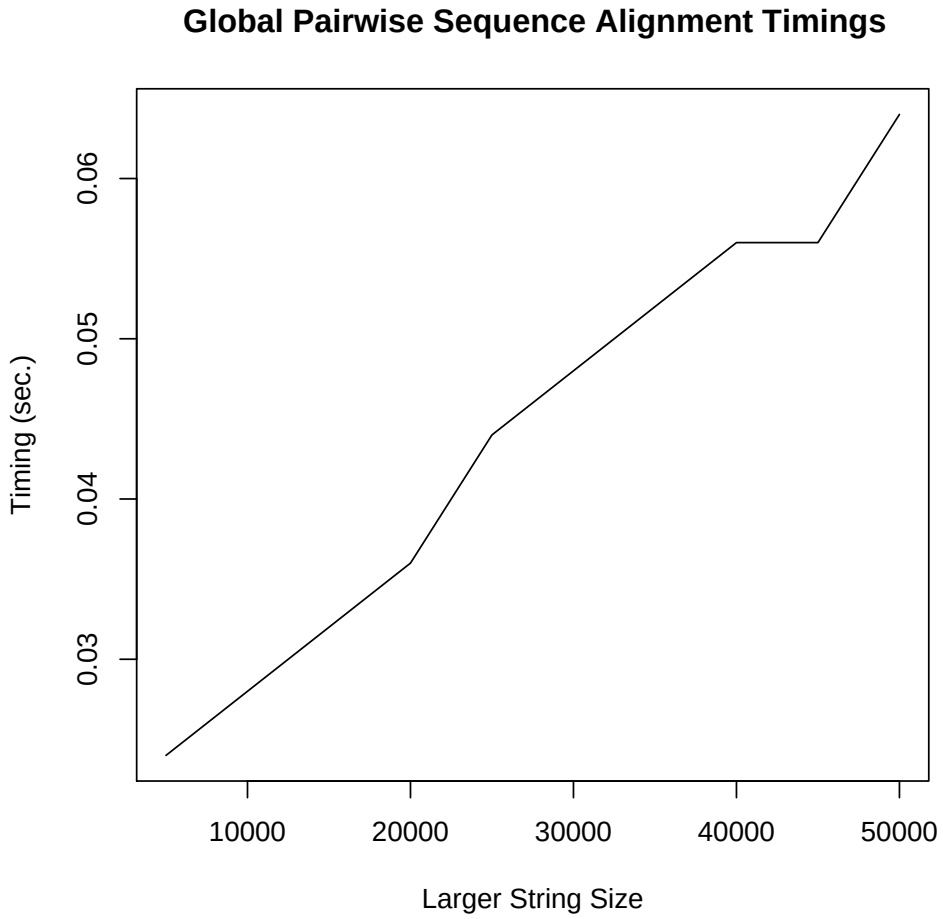
```
> N <- as.integer(seq(5000, 50000, by = 5000))
> newTimings <- rep(0, length(N))
> names(newTimings) <- as.character(N)
> for (i in seq_len(length(N))) {
+     string1 <- DNAString(paste(sample(DNA_ALPHABET[1:4],
+         35, replace = TRUE), collapse = ""))
+     string2 <- DNAString(paste(sample(DNA_ALPHABET[1:4],
+         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
0.024 0.028 0.032 0.040 0.044 0.044 0.052 0.060 0.060 0.064

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

               Estimate   Std. Error    t value
(Intercept)  0.044800000 0.0006903164 64.8977729
poly(N, 2)1  0.041836623 0.0021829723 19.1649814
poly(N, 2)2 -0.001392621 0.0021829723 -0.6379473
               Pr(>|t|)
(Intercept) 5.419046e-11
poly(N, 2)1 2.622428e-07
poly(N, 2)2 5.438044e-01

> plot(N, newTimings, xlab = "Larger String Size",
+     ylab = "Timing (sec.)", type = "l", main = "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 is 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
0.016 0.020 0.020 0.024 0.024 0.028 0.032 0.032 0.036 0.040

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

```
5000 10000 15000 20000 25000 30000 35000 40000 45000 50000
0.33  0.29  0.38  0.33  0.45  0.42  0.38  0.43  0.36  0.37
```

13 Session Information

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

```
> sessionInfo()

R version 2.8.0 Under development (unstable) (2008-08-09 r46277)
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
[6] datasets  methods   base

other attached packages:
[1] Biostrings_2.9.61 Biobase_2.1.7     IRanges_0.99.3
```

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.