Sequence Alignment of Short Read Data using Biostrings

Patrick Aboyoun
Fred Hutchinson Cancer Research Center
Seattle, WA 98008

13 November 2008

Contents

1 Introduction

While most researchers use sequence alignment software like ELAND, MAQ, and Bowtie to perform the bulk of short read mappings to a target genome, BioConductor contains a number of string matching/pairwise alignment tools in the Biostrings package that can be invaluable in answering complex scientific questions. These tools are naturally divided into four groups (`matchPDict`, `vmatchPattern`, `pairwiseAlignment`, and OTHER) that contain the following functions:

`matchPDict` : `matchPDict`, `countPDict`

`vmatchPattern` : `matchPattern`, `countPattern`, `vmatchPattern`, `vcountPattern`

`pairwiseAlignment` : `pairwiseAlignment`, `stringDist`

OTHER : `matchLRPatterns` (finds singleton paired-end matches), `matchProbePair` (finds theoretical amplicons), `matchPWM` (matches using a position weight matrix)

For more information on any of these functions, use `help(≪ function name ≫)` from within R.

Of the functions listed above, the `pairwiseAlignment` function has the most helper functions. These helper functions are in Table 1 below.

Table 2 shows the relative strenghts and weaknesses of the three main functional families and hints at how they can be used sequentially to find answers to multi-faceted questions.

Function	Description
`[`	Extracts the specified elements of the alignment object
`aligned`	Creates *XStringSet* alignments without insertions
`alphabet`	Extracts the allowable characters in the original strings
`as.character`	Creates a character vector version of `aligned`
`as.matrix`	Creates an "exploded" character matrix version of `aligned`
`compareStrings`	Creates character string mashups of the alignments
`consensusMatrix`	Computes a consensus matrix for the alignments
`consensusString`	Creates the string based on a 50% + 1 vote from the consensus matrix
`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
`pid`	Computes the percent sequence identity
`rep`	Replicates the elements of the alignment object
`score`	Extracts the pairwise sequence alignment scores
`summary`	Summarizes a pairwise sequence alignment
`toString`	Creates a concatenated string version of `aligned`
`type`	Extracts the type of pairwise sequence alignment
`Views`	Extracts the alignment ranges for the subject

Table 1: Functions for *PairwiseAlignment* objects.

2 Setup

This lab is designed as series of hands-on exercises where the students follow along with the instructor. The first exercise is to load the required packages:

Exercise 1
Start an R session and use the `library` *function to load the ShortRead software package and BSgenome.Mmusculus.UCSC.mm9 genome package along with its dependencies using the following commands:*

```
> library("ShortRead")
> library("BSgenome.Mmusculus.UCSC.mm9")
```

Exercise 2
Use the `packageDescription` *function to confirm that the loaded version of the Biostrings package is >= 2.10.3 and the IRanges package is >= 1.0.5.*

```
> packageDescription("Biostrings")$Version

[1] "2.10.6"

> packageDescription("IRanges")$Version

[1] "1.0.5"
```

matchPDict	vmatchPattern	pairwiseAlignment
Utilizes a fast string matching algorithm for multiple patterns.	Uses a fast string matching algorithm for multiple subjects.	Not practical for long strings.
Finds all occurrences with up to the specified # of mismatches.	Finds all occurrences with up to the specified # of mismatches.	Returns only the first occurrence of the best scoring alignment.
Supports removal of repeat masked regions.	Supports removal of repeat masked regions.	Cannot handle masked genomes.
Produces limited output: # of times a pattern matches and where they occur.	Produces limited output: # of times a pattern matches and where they occur.	Allows various summaries of alignments.
Does not support insertions or deletions.	Does support insertions and deletions.	Does support insertions and deletions.
Supports a limited number of mismatch penalty schemes.	Supports a limited number of mismatch penalty schemes.	Provides a flexible alignment framework, including quality-based scoring.

Table 2: Comparisons of string matching/alignment methods.

Seek assistance from one of the course assistants if you need help updating any of your BioConductor packages.

This lab also requires you have access to sample data.

Exercise 3
Copy the data from the distribution media to your local hard drive. Change the working directory in R to point to the data location.

```
> setwd(file.path("path", "to", "data"))
```

3 Finding Possible Contaminants in the Short Reads

The raw base-called sequences that are produced by high-throughput sequencing technologies like Solexa (Illumina), 454 (Roche), SOLiD (Applied Biosystems), and Helicos tend to contain experiment-related contaminants like adapters and PCR primers as well as "phantom" sequences like poly As. Functions like `countPDict`, `vcountPattern`, and `pairwiseAlignment` from the Biostrings package allow for the discovery of these troublesome sequences.

These raw base-called sequences can be read with functions like the `readXStringColumns` function and processed with functions like `tables`, which find the most common sequences, from the ShortRead package. While this course will be using pre-processed data for this exercise, the code to find the top short reads looks something like:

```
> sp <- list(experiment1 = SolexaPath(file.path("path", "to",
+     "experiment1")), experiment2 = SolexaPath(file.path("path",
+     "to", "experiment2")))
> patSeq <- paste("s_", 1:8, "_.*_seq.txt", sep = "")
> names(patSeq) <- paste("lane", 1:8, sep = "")
> topReads <- lapply(seq_len(length(sp)), function(i) {
+     print(experimentPath(sp[[i]]))
+     lapply(seq_len(length(patSeq)), function(j, n = 1000) {
+         cat("Reading", patSeq[[j]], "...")
```

```
+         x <- tables(readXStringColumns(baseCallPath(sp[[i]]),
+             pattern = patSeq[[j]], colClasses = c(rep(list(NULL),
+                 4), list("DNAString")))[[1]], n = n)[["top"]]
+         names(x) <- chartr("-", "N", names(x))
+         cat("done.\n")
+         x
+     })
+ })
> names(topReads) <- names(sp)
> for (i in seq_len(length(sp))) {
+     names(topReads[[i]]) <- names(patSeq)
+ }
```

Exercise 4
Use the `load` *function to load the pre-processed top short reads object from the data directory into your R session.*

```
> load(file.path("data", "topReads.rda"))
```

Exercise 5
Use the `class` *function to find the class of the* `topReads` *object and the* `names` *function to find the names of its components.*

```
> class(topReads)

[1] "list"

> names(topReads)

[1] "experiment1" "experiment2"
```

Exercise 6
Since `topReads` *is a list object, use the* `sapply` *function to find out what its elements are and what the names of the subcomponents are.*

```
> sapply(topReads, class)

experiment1 experiment2
     "list"      "list"

> sapply(topReads, names)

     experiment1 experiment2
[1,] "lane1"     "lane1"
[2,] "lane2"     "lane2"
[3,] "lane3"     "lane3"
[4,] "lane4"     "lane4"
[5,] "lane5"     "lane5"
[6,] "lane6"     "lane6"
[7,] "lane7"     "lane7"
[8,] "lane8"     "lane8"
```

Exercise 7
Thus `topReads` *is a list of list objects. Use nested* `sapply` *function calls to find the class of the underlying subcomponents.*

```
> sapply(topReads, sapply, class)
```

```
      experiment1 experiment2
lane1 "integer"   "integer"
lane2 "integer"   "integer"
lane3 "integer"   "integer"
lane4 "integer"   "integer"
lane5 "integer"   "integer"
lane6 "integer"   "integer"
lane7 "integer"   "integer"
lane8 "integer"   "integer"
```

Exercise 8

Subscript out the first element for each of the 8 lanes for both experiments by nesting an `sapply` *function call in a* `lapply` *function call.*

```
> lapply(topReads, sapply, "[", 1)
```

```
$experiment1
lane1.AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                      81237
lane2.AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                      95367
lane3.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      95122
lane4.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                     138645
lane5.AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                      72865
lane6.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                     103791
lane7.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                     103993
lane8.AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                     101645

$experiment2
lane1.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      46209
lane2.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      53886
lane3.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      96433
lane4.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      99389
lane5.AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                      22621
lane6.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      83085
lane7.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      80590
lane8.GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
                                      57792
```

The pre-processed data, `topReads`, loaded in the previous exercise, are in a *list* of *list*s of named integer vectors whose {name, value} pair represents the sequence and it corresponding number of occurrences. At a high level, the primary *list* elements represent two Solexa experiments and the secondary *list* elements representing the 8 lanes of a Solexa run. In both of these experiments, lanes {1-4, 6-8} contain mouse-related experimental data and lane 5 contains data from bacteriophage ϕX174.

The `lapply` function call in the above example, which extracts the most prevalent sequence in each of the lanes, shows that the top read is either `GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA` or all As. Given that the former sequence is the 33 base pairs of Solexa's genomic DNA/ChIP-seq adapter plus 3 As and the latter sequence of 36 As, it would appear that As are called when there is little information about a particular base.

Finding Poly N Sequences

When data are acquired through the ShortRead package, poly N sequences can be removed using the `polynFilter` function. Since we are operating on pre-processed data, we will have to remove poly N sequences using more rudimentary tools.

Exercise 9
Use the following steps to find the top sequences with with at least 34 nucleotides of a single type (A, C, T, G):

1. *Extract the named vector corresponding to the top sequence counts for experiment 1, lane 1.*
2. *Use the* `alphabetFrequency` *function to find the alphabet frequencies from a DNAStringSet based on the names of this named vector.*
3. *Use the parallel max,* `pmax`*, function to find the maximum number of occurrences for each of the four bases.*
4. *Create a DNAStringSet whose elements contain at least 34 bases of a single type.*

```
> lane1.1TopReads <- topReads[["experiment1"]][["lane1"]]
> alphabetCounts <- alphabetFrequency(DNAStringSet(names(lane1.1TopReads)),
+     baseOnly = TRUE)
> lane1.1MaxLetter <- pmax(alphabetCounts[, "A"], alphabetCounts[,
+     "C"], alphabetCounts[, "G"], alphabetCounts[, "T"])
> DNAStringSet(names(lane1.1TopReads[lane1.1MaxLetter >=
+     34]))

  A DNAStringSet instance of length 115
      width seq
  [1]    36 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  [2]    36 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
  [3]    36 AAAAAAAAAAAAAAAAAAACAAAAAAAAAAAAAA
  [4]    36 AAAAAAAAAAAAAAAAAACAAAAAAAAAAAAAAA
  [5]    36 AAAAAAAAAAAAAAAAAAGAAAAAAAAAAAAAAA
  [6]    36 AAAAAAAAAAAAAAAAAATAAAAAAAAAAAAAAA
  [7]    36 AAAAAAAAAAAAAAAAAANAAAAAAAAAAAAAAA
  [8]    36 TAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  [9]    36 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACA
  ...   ... ...
[107]    36 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAACACA
[108]    36 GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
```

```
[109]    36 AAAAAAAAAAAAAAAAAAAAATAAAAAAAAAAAAA
[110]    36 AAAAAAAAAAAAGAAAAAAAAAAAAAAAAAAAAAA
[111]    36 AAAAAAAAAAAATAAAAAAAAAAAAAAAAAAAAAA
[112]    36 AAAAAAAAAATAAAAAAAAAAAAAAAAAAAAAAAA
[113]    36 AAAAAAGAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[114]    36 AAAAATAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[115]    36 ANAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Finding Adapter-Like Sequences

While the Solexa's adapter is known not to map to the mouse genome,

Exercise 10
Show that Solexa's DNA/ChIP-seq adapter doesn't map to the mouse genome.

```
> adapter <- DNAString("GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTG")
```

Search the Mmusculus genome by first setting up a BSParams parameter object that utilizes the `countPattern` *function and then using the* `bsapply` *function to loop over the chromosomes. For more information, type* `help("BSParams-class")` *and* `help("bsapply")`.

```
> bsParams <- new("BSParams", X = Mmusculus, FUN = countPattern,
+     simplify = TRUE)
> bsapply(bsParams, pattern = adapter)
```

```
        chr1         chr2         chr3         chr4         chr5
           0            0            0            0            0
        chr6         chr7         chr8         chr9        chr10
           0            0            0            0            0
       chr11        chr12        chr13        chr14        chr15
           0            0            0            0            0
       chr16        chr17        chr18        chr19         chrX
           0            0            0            0            0
        chrY         chrM  chr1_random  chr3_random  chr4_random
           0            0            0            0            0
 chr5_random  chr7_random  chr8_random  chr9_random chr13_random
           0            0            0            0            0
chr16_random chr17_random  chrX_random  chrY_random chrUn_random
           0            0            0            0            0
```

repeated sequencing of the adapter is a great inefficiency within an experiment. These adapter-like sequences can distort quality assurance of the Solexa data and removing them upstream can help prevent distortions in downstream QA conclusions.

Exercise 11
Use the following steps to find the adapter-like sequences within the top reads:

1. *Create a DNAStringSet object containing the unique reads by first taking the names off of the top sequence tables through nested* `lapply` *operations, then removing the names of the experiments using the* `unname` *function, then using the* `unique` *function to find the unique set of reads, and then using the* `sort` *function to sort the sequences in alphabetical order.*

2. *Use the* `vcountPattern` *function to find the adapter-like sequences.*

3. Obtain the subset of adapter-like sequences.

```
> uniqueReads <- DNAStringSet(sort(unique(unname(unlist(lapply(topReads,
+     lapply, names))))))
> whichAdapters <- which(vcountPattern(adapter, uniqueReads,
+     max.mismatch = 4, with.indels = TRUE) > 0)
> adapterReads <- uniqueReads[whichAdapters]
> adapterReads

  A DNAStringSet instance of length 819
      width seq
  [1]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAA
  [2]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT
  [3]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTATAT
  [4]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
  [5]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAT
  [6]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGTAA
  [7]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTAAA
  [8]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT
  [9]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTTAT
  ...   ... ...
[811]    36 NATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT
[812]    36 NATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGNN
[813]    36 NATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTTAT
[814]    36 NATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTTNN
[815]    36 NATCGGAAGAGNTCGTATGCCGTCNTCTGCTTGNNN
[816]    36 NATCGGAAGAGNTCGTATGCCGTCNTCTGCTTNNNN
[817]    36 TATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT
[818]    36 TATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
[819]    36 TATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT
```

As the results above show, Solexa's 33-mer adapter is closely related to 819 unique short reads from the top reads lists.

Exercise 12
Use the following steps to find the number of unique adapter-like reads and the total number of these reads in each of the 8 lanes for the two experiments:

1. *Use nested* `lapply` *function calls to extract the adapter-like sequences from each of the Solexa lanes.*
2. *Use nested* `sapply` *function calls to get the number of unique adapter-like sequences.*
3. *Use nested* `sapply` *function calls to get the total number of adapter-like sequences.*

```
> topAdapterReads <- lapply(topReads, lapply, function(x) x[intersect(names(x),
+     as.character(adapterReads))])
> sapply(topAdapterReads, sapply, length)

      experiment1 experiment2
lane1         500         226
lane2         303         235
lane3         462         323
lane4         547         305
```

```
lane5           0           0
lane6         464         275
lane7         516         284
lane8         343         206
```

```
> sapply(topAdapterReads, sapply, sum)
```

```
      experiment1 experiment2
lane1      265463      158678
lane2      225519      178534
lane3      308251      303996
lane4      456932      290159
lane5           0           0
lane6      343988      255142
lane7      360014      252049
lane8      233244      177058
```

These adapter-like sequences are not wholely without value because they can provide some insight in where base call errors are most likely to occur for a particular sequence.

Exercise 13
Find the unique sequences from lane 1 of experiment 1 and their associated counts.

```
> lane1.1AdapterCounts <- topAdapterReads[["experiment1"]][["lane1"]]
> lane1.1AdapterReads <- DNAStringSet(names(lane1.1AdapterCounts))
> lane1.1AdapterReads
```

```
  A DNAStringSet instance of length 500
      width seq
  [1]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
  [2]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT
  [3]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT
  [4]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAT
  [5]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTATAT
  [6]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAA
  [7]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTAAA
  [8]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTTAT
  [9]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAA
  ...   ... ...
[492]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTCCTTTCAA
[493]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAG
[494]    36 GATCGGAAGAGCTCGTATGCCGTCTTTTGCTTGTAA
[495]    36 GATCGGAAGAGCTCGTATGTCGTCTTCTGCTTTGAA
[496]    36 GATCGGAAGAGCTCGTATGNCGTCTTCTGCTTAAAA
[497]    36 GATCGGAAGAGTCGTATGCCGTCTTCTGCTTGGATA
[498]    36 GATCGGAANAGCTCGTATGCCGTCTTCTGCTTAGAT
[499]    36 GATCGGGAGAGCTCGTATGCCGTCTTCTGCTTAGAT
[500]    36 GATTGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
```

Exercise 14
Use the `pairwiseAlignment` *function to fit the pairwise alignments of the adapter-like sequences against the adapter then summarize the results using the* `summary` *function.*

```
> lane1.1AdapterAligns <- pairwiseAlignment(lane1.1AdapterReads,
+     adapter, type = "patternOverlap")
> summary(lane1.1AdapterAligns, weight = lane1.1AdapterCounts)

Pattern Overlap Pairwise Alignment
Number of Alignments:  265463

Scores:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  27.75   57.52   57.52   59.09   65.40   65.40

Number of matches:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  30.00   32.00   32.00   32.27   33.00   33.00

Top 10 Mismatch Counts:
   SubjectPosition Subject Pattern  Count Probability
76              33       G       A 106988 0.403024150
78              33       G       T  41812 0.157505942
49              20       C       A  12558 0.047306028
77              33       G       C   7298 0.027491590
70              29       G       T   5686 0.021419181
51              20       C       N   2038 0.007677153
52              20       C       T   1996 0.007518939
50              20       C       G   1595 0.006008370
32              14       C       A   1487 0.005601534
34              14       C       T    902 0.003397837
```

Finding Over-Represented Sequences

Another potential source of data contamination is over-represented sequences. These sequences can be found by clustering the short reads.

Exercise 15
First find the unique sequences from lane 1 of experiment 2 and their associated counts.

```
> lane2.1TopCounts <- topReads[["experiment2"]][["lane1"]]
> lane2.1TopReads <- DNAStringSet(names(lane2.1TopCounts))
> lane2.1TopReads

  A DNAStringSet instance of length 1000
      width seq
   [1]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
   [2]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT
   [3]    36 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   [4]    36 ANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
   [5]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAT
   [6]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT
   [7]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTATAT
   [8]    36 GNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
   [9]    36 CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
   ...   ... ...
```

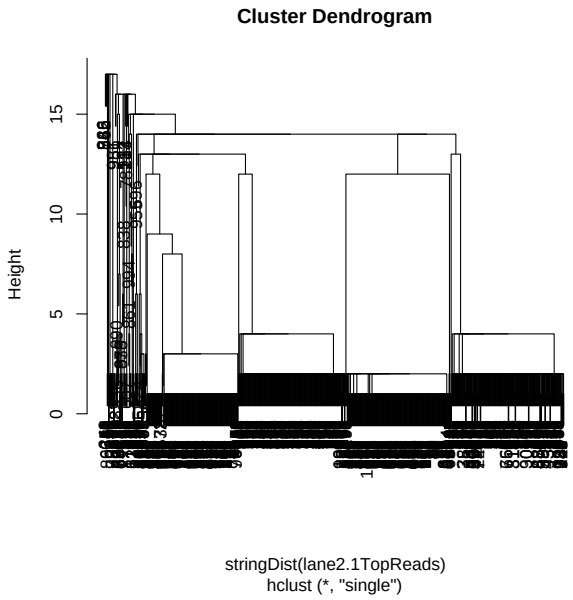

Figure 1: Clustering of Top Reads

```
 [992]    36 TGTCCACTGTAGGACGTGGAATATGGCAAGAAAACT
 [993]    36 ATTCCTCCCGACACATAATAATCAGAACAACAAATG
 [994]    36 ATTGATATACACTGTTCTACAAATCCCGTTTCCAAC
 [995]    36 ANNNNNNNNNAAAAANNNNANNAAAAAAAAAAAAAA
 [996]    36 ANNNNNNNNNNNNNNNNNNNNNNAANNNANNNNNN
 [997]    36 CATATTCCAGGTCCTACAGTGTGCATTTCTCATTTT
 [998]    36 CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNTN
 [999]    36 GATCGGAAGAGCTCGTATGCCGCCTTCTGCTTGGAT
[1000]    36 GATCGGAAGAGCTCGTATGCCGGTCTTCTGTTTAGA
```

Exercise 16
Then use the `stringDist` *function to generate the Levenshtein's edit distance amongst the reads, generate nearest-neighbor-based clustering using the* `hclust` *function, and classify the reads into clusters using the* `cutree` *function.*

```
> lane2.1Clust <- hclust(stringDist(lane2.1TopReads), method = "single")
> plot(lane2.1Clust)
> lane2.1Groups <- cutree(lane2.1Clust, h = 2)
> head(sort(table(lane2.1Groups), decreasing = TRUE))

lane2.1Groups
  1   9   8   3   2  10
226 200 197 161  34  27
```

The example above produces four interesting short read clusters: one representing poly As, one representing Solexa's adapter, and the remaining two coming from an unknown origin.

Exercise 17
Create a set of interesting sequences of unknown origin by using the `intersect` *function to find intersection of one of the interesting clusters with the reverse complement of the other interesting cluster.*

```
> reverseComplement(lane2.1TopReads[lane2.1Groups == 9])

  A DNAStringSet instance of length 200
      width seq
  [1]    36 AAATGAGAAATACACACTTTAGGACGTGAAATATGG
  [2]    36 AATGAGAAATACACACTTTAGGACGTGAAATATGGC
  [3]    36 TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
  [4]    36 AGAAATACACACTTTAGGACGTGAAATATGGCGAGG
  [5]    36 AATATGGCAAGAAAACTGAAAATCATGGAAAATGAG
  [6]    36 AAAATCACGGAAAATGAGAAATACACACTTTAGGAC
  [7]    36 AGAAAACTGAAAATCACGGAAAATGAGAAATACACA
  [8]    36 AGGACGTGGAATATGGCAAGAAAACTGAAAATCATG
  [9]    36 AAAATGAGAAATACACACTTTAGGACGTGAAATATG
  ...   ... ...
[192]    36 CTGAAAAAGGTGGAAAATTTAGAAATGTCCACTGTA
[193]    36 AATGGAAAATGAGAAACATCCACTTGACGACTTGAA
[194]    36 GAGAGAAAACTGAAAATCACGGAAAATGAGAAATAC
[195]    36 AAAATAATGGAAAATGAGAAACATCCACTTGACGAC
[196]    36 AGTGAAATATGGCGAGGAAAACTGAAAAAGGTGGAA
[197]    36 AAAACTGAAAATCATGGAAAATGAGAAACATCCACT
[198]    36 GGCGAGGAAAACTGAAAAAGGTGGAAAATTTAGAAA
[199]    36 AGAGAAACATCCACTTGACGACTTGAAAAATGACGA
[200]    36 AGGAAAATGAGAAATACACACTTTAGGACGTGAAAT

> lane2.1TopReads[lane2.1Groups == 8]

  A DNAStringSet instance of length 197
      width seq
  [1]    36 ACTGAAAATCACGGAAAATGAGAAATACACACTTTA
  [2]    36 AAACATCCACTTGACGACTTGAAAAATGACGAAATC
  [3]    36 TAGGACGTGGAATATGGCAAGAAAACTGAAAATCAT
  [4]    36 GGAATATGGCAAGAAAACTGAAAATCATGGAAAATG
  [5]    36 GTAGGACGTGGAATATGGCAAGAAAACTGAAAATCA
  [6]    36 TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
  [7]    36 CGTGAAATATGGCGAGGAAAACTGAAAAAGGTGGAA
  [8]    36 GAATATGGCAAGAAAACTGAAAATCATGGAAAATGA
  [9]    36 GAAAATCACGGAAAATGAGAAATACACACTTTAGGA
  ...   ... ...
[189]    36 ATCATGGAAAATGAGAAACATCCACTTGACGACTTG
[190]    36 ATTTAGAAATGTCCACTGTAGGACGTGGAATATGGC
[191]    36 TAGAAATGTCCACTGTAGGACGTGGAATATGGCAAG
[192]    36 TCCACTTGACGACTTGAAAAATGACGAAATCACTAA
[193]    36 TTAGAAATGTCCACTGTAGGACGTGGAATATGGCAA
[194]    36 AATTTAGAAATGTCCACTGTAGGACGTGGAATATGG
[195]    36 CGAAATCACTAAAAAACGTGAAAAATGAGAAATGCA
[196]    36 GAAATATGGCGAGGAAAACTGAAAAAGGTGGAAAAT
[197]    36 TGTCCACTGTAGGACGTGGAATATGGCAAGAAAACT

> unknownSeqs <- DNAStringSet(intersect(as.character(reverseComplement(lane2.1TopReads[lane2.1Groups ==
+     9])), as.character(lane2.1TopReads[lane2.1Groups ==
+     8])))
> unknownSeqs
```

```
  A DNAStringSet instance of length 155
      width seq
  [1]    36 AAATGAGAAATACACACTTTAGGACGTGAAATATGG
  [2]    36 AATGAGAAATACACACTTTAGGACGTGAAATATGGC
  [3]    36 TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
  [4]    36 AGAAATACACACTTTAGGACGTGAAATATGGCGAGG
  [5]    36 AATATGGCAAGAAAACTGAAAATCATGGAAAATGAG
  [6]    36 AAAATCACGGAAAATGAGAAATACACACTTTAGGAC
  [7]    36 AGAAAACTGAAAATCACGGAAAATGAGAAATACACA
  [8]    36 AGGACGTGGAATATGGCAAGAAAACTGAAAATCATG
  [9]    36 AAAATGAGAAATACACACTTTAGGACGTGAAATATG
  ...   ... ...
[147]    36 CACTTGACGACTTGAAAAATGACGAAATCACTAAAA
[148]    36 GGAAAATGAGAAACATCCACTTGACGACTTGAAAAA
[149]    36 CCTGGAATATGGCGAGAAAACTGAAAATCACGGAAA
[150]    36 GAAAATCATGGAAAATGAGAAACATCCACTTGACGA
[151]    36 TGAGAAATACACACTTTAGGACGTGAAATATGGCGA
[152]    36 ATGGCGAGAAAACTGAAAATCACGGAAAATGAGAAA
[153]    36 ACTGTAGGACGTGGAATATGGCAAGAAAACTGAAAA
[154]    36 TCCACTTGACGACTTGAAAAATGACGAAATCACTAA
[155]    36 AAAACTGAAAATCATGGAAAATGAGAAACATCCACT
```

Exercise 18
Create a set of interesting sequences and assoicated counts based upon the intersection created above.

```
> unknownCounts <- lane2.1TopCounts[as.character(unknownSeqs)] +
+     lane2.1TopCounts[as.character(reverseComplement(unknownSeqs))]
> unknownSeqs <- unknownSeqs[order(unknownCounts, decreasing = TRUE)]
> unknownCounts <- unknownCounts[order(unknownCounts, decreasing = TRUE)]
> head(unknownCounts)

TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
                                 387
GGAATATGGCAAGAAAACTGAAAATCATGGAAAATG
                                 375
ACTGAAAATCACGGAAAATGAGAAATACACACTTTA
                                 358
AAACATCCACTTGACGACTTGAAAAATGACGAAATC
                                 357
TAGGACGTGGAATATGGCAAGAAAACTGAAAATCAT
                                 354
GAAAATCACGGAAAATGAGAAATACACACTTTAGGA
                                 345
```

These sequences of unknown origin may be related and could potential assemble into a more informative larger sequence. This assembly can be performed using functions from the Biostrings package by first finding a starter, or seeding, sequences that can be grown using pairwise alignments of the starter sequences and the remaining sequences.

Exercise 19
Use the following step to find a starter or seed sequence to use in an assembly process by finding the unique sequence that closest related to the set of unknown sequences:

1. *Use the* `stringDist` *function to find the number of matches amongst the reads using an overlap alignment with a scoring scheme of* `{match = 1, mismatch = -Inf, gapExtension = -Inf}` *then convert the results into a matrix and loop over the rows to count how many times each unique read overlap with other unique reads at least 24 bases in the 36 bases reads.*

2. *Choose the unique sequence with the most similar unique sequences using the metric developed in the previous step.*

```
> submat <- nucleotideSubstitutionMatrix(match = 1, mismatch = -Inf)
> whichStarter <- which.max(apply(as.matrix(stringDist(unknownSeqs,
+     method = "substitutionMatrix", substitutionMatrix = submat,
+     gapExtension = -Inf, type = "overlap")), 1, function(x) sum(x >=
+     24)))
> starterSeq <- unknownSeqs[[whichStarter]]
> starterSeq

  36-letter "DNAString" instance
seq: TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
```

Exercise 20
Use the `pairwiseAlignment` *function to generate the pairwise alignments of all sequences against the starter sequence.*

```
> starterAlign <- pairwiseAlignment(unknownSeqs, starterSeq,
+     substitutionMatrix = submat, gapExtension = -Inf, type = "overlap")
```

Exercise 21
Assemble a sequence by using the starter sequence created above and the set of interesting sequences you found. The first step in this assembly is to create a function that generates a sequeunce through unanimous vote in a concensus matrix.

```
> unanimousChars <- function(x) {
+     letters <- c("A", "C", "G", "T")
+     mat <- consensusMatrix(x)[letters, , drop = FALSE]
+     paste(apply(mat, 2, function(y) {
+         z <- which(y != 0)
+         ifelse(length(z) == 1, letters[z], "?")
+     }), collapse = "")
+ }
```

Exercise 22
The next step is to find which alignments are in the "prefix" of the starter sequence. These are the sequences that overlap to the left of the start sequence.

```
> whichInPrefix <- (score(starterAlign) >= 10 & start(subject(starterAlign)) ==
+     1 & start(pattern(starterAlign)) != 1)
> prefix <- narrow(unknownSeqs[whichInPrefix], 1, start(pattern(starterAlign[whichInPrefix])) -
+     1)
> prefix <- DNAStringSet(paste(sapply(max(nchar(prefix)) -
+     nchar(prefix), polyn, nucleotides = "-"), as.character(prefix),
+     sep = ""))
> consensusMatrix(prefix, baseOnly = TRUE)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
A        1    2    0    0    5    0    0    0    0     0    11    12
C        0    0    0    0    0    6    7    0    0     0     0     0
G        0    0    3    4    0    0    0    0    9    10     0     0
T        0    0    0    0    0    0    0    8    0     0     0     0
other   25   24   23   22   21   20   19   18   17    16    15    14
      [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23]
A         0    14     0     0     0     0     0    20     0    22    23
C         0     0     0     0     0    18     0     0     0     0     0
G         0     0     0    16    17     0    19     0    21     0     0
T        13     0    15     0     0     0     0     0     0     0     0
other    13    12    11    10     9     8     7     6     5     4     3
      [,24] [,25] [,26]
A        24    25     0
C         0     0    26
G         0     0     0
T         0     0     0
other     2     1     0
```

```
> unanimousChars(prefix)
```

```
[1] "AAGGACCTGGAATATGGCGAGAAAAC"
```

Exercise 23

The corresponding step is to find which alignments are in the "suffix" of the starter sequence. These are the sequences that overlap to the right of the start sequence.

```
> whichInSuffix <- (score(starterAlign) >= 10 & end(subject(starterAlign)) ==
+     36 & end(pattern(starterAlign)) != 36)
> suffix <- narrow(unknownSeqs[whichInSuffix], end(pattern(starterAlign[whichInSuffix])) +
+     1, 36)
> suffix <- DNAStringSet(paste(as.character(suffix), sapply(max(nchar(suffix)) -
+     nchar(suffix), polyn, nucleotides = "-"), sep = ""))
> consensusMatrix(suffix, baseOnly = TRUE)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
A       26    0    0    0    0   21   20   19    0    17     0     0
C        0   25    0    0    0    0    0    0    0     0     0     0
G        0    0   24    0   22    0    0    0    0     0     0    15
T        0    0    0   23    0    0    0    0   18     0    16     0
other    0    1    2    3    4    5    6    7    8     9    10    11
      [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23]
A         0     0     0    11     0     0     8     7     6     5     0
C         0    13     0     0     0     0     0     0     0     0     4
G        14     0    12     0    10     9     0     0     0     0     0
T         0     0     0     0     0     0     0     0     0     0     0
other    12    13    14    15    16    17    18    19    20    21    22
      [,24] [,25] [,26]
A         0     0     1
C         0     0     0
G         0     2     0
T         3     0     0
other    23    24    25
```

```
> unanimousChars(suffix)

[1] "ACGTGAAATATGGCGAGGAAAACTGA"
```

Exercise 24
Combine the prefix and suffix with the starter sequence.

```
> extendedUnknown <- DNAString(paste(unanimousChars(prefix),
+     as.character(starterSeq), unanimousChars(suffix), sep = ""))
> extendedUnknown

  88-letter "DNAString" instance
seq: AAGGACCTGGAATATGGCGAGAAAACTGAAAA...TTTAGGACGTGAAATATGGCGAGGAAAACTGA
```

Exercise 25
Align the set of unknown sequences against the extended sequence.

```
> unknownAlign <- pairwiseAlignment(unknownSeqs, extendedUnknown,
+     substitutionMatrix = submat, gapExtension = -Inf, type = "overlap")
> table(score(unknownAlign))

 0  1  2  3  4  5  6  7  8  9 21 22 23 24 25 26 27 28 29 30 31 32 33 34
12 26 26  2  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2
35 36
 2 53
```

Exercise 26
Use the `countPDict` *function within nested* `sapply` *function calls to show the number of reads that map to the unknown sequence in the 8 lanes from the 2 experiments.*

```
> sapply(topReads, sapply, function(x) {
+     whichNoNs <- (alphabetFrequency(DNAStringSet(names(x)))[,
+         "N"] == 0)
+     x <- x[whichNoNs]
+     pdict <- PDict(DNAStringSet(names(x)))
+     whichMapped <- (countPDict(pdict, extendedUnknown) +
+         countPDict(pdict, reverseComplement(extendedUnknown))) >
+         0
+     sum(x[whichMapped])
+ })

      experiment1 experiment2
lane1        1577       10855
lane2        4627       10482
lane3        1284       10633
lane4        2219        8400
lane5           0           0
lane6        1659       13095
lane7        1823       11099
lane8        4657       14916
```

Exercise 27
Use the `countPattern` *function within a* `bsapply` *loop to find to which chromosome the extended unknown sequence maps.*

```
> params <- new("BSParams", X = Mmusculus, FUN = countPattern,
+     simplify = TRUE)
> unknownCountPattern <- bsapply(params, pattern = extendedUnknown)
> unknownCountPattern

        chr1         chr2         chr3         chr4         chr5
           0            1            0            0            0
        chr6         chr7         chr8         chr9        chr10
           0            0            0            0            0
       chr11        chr12        chr13        chr14        chr15
           0            0            0            0            0
       chr16        chr17        chr18        chr19         chrX
           0            0            0            0            0
        chrY         chrM  chr1_random  chr3_random  chr4_random
           0            0            0            0            0
 chr5_random  chr7_random  chr8_random  chr9_random chr13_random
           0            0            0            0            0
chr16_random chr17_random  chrX_random  chrY_random chrUn_random
           0            0            0            0            0
```

Exercise 28
Finally use the `matchPattern` *function to find the exact location on chromosome that it maps to.*

```
> mm9Chr2 <- Mmusculus[["chr2"]]
> mm9Ch2View <- matchPattern(extendedUnknown, mm9Chr2)
> mm9Ch2View

  Views on a 181748087-letter DNAString subject
subject: NNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...AGGTCTAGGGTTTGCGCCTGGATTACGGGT
views:
       start      end width
[1] 98507289 98507376    88 [AAGGACCTGGAATATGGCGA...ATATGGCGAGGAAAACTGA]
```

4 Aligning Bacteriophage Reads

Solexa's SOP includes dedicating lane 5 from a set of 8 to sequencing the bacterophage ϕX174 genome, a circular single-stranded genome with 5386 base pairs and the first to be sequenced in 1978. Analyzing the data from this lane can provide a check for a systematic failure of the sequencer.

Exercise 29
Read in one of the lane 5 export files from a Solexa run.

```
> sp <- SolexaPath(file.path("extdata", "ELAND", "080828_HWI-EAS88_0003"))
> phageReads <- readAligned(analysisPath(sp), "s_5_1_export.txt",
+     "SolexaExport")
```

Exercise 30
Find the unique number of reads and number of times they occurred.

```
> phageReadTable <- tables(sread(phageReads), n = Inf)[["top"]]
```

Exercise 31
Find which unique reads have uncalled bases and create a "clean" set of reads without any uncalled bases.

```
> whichNotClean <- grep("N", names(phageReadTable))
> head(phageReadTable[whichNotClean])

ANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN TNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
                              13320                                11892
CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN GNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
                               8978                                 7670
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN AANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
                               2308                                 1652

> cleanReadTable <- phageReadTable[-whichNotClean]
> head(cleanReadTable)

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA GATCTTTGGCGGCACGGAGCCGCGCATCACCTGTA
                              70947                                7561
GATCTCCCGAGCATCACCACATTACTGCGGTTATA CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
                               6740                                2535
GATCTCCATGGCATCACCACATTACTGCGGTTATA GACGTTTGGTCAGTTCCATCAACATCATAGCCAGA
                               2323                                 439
```

Exercise 32
Load the `phiX174Phage` *object and extract the New England BioLabs (NEB) version, the one used by Solexa, of the bacterophage $\phi X174$ genome, and extend the genome 34 bases to "linearize" the circular genome.*

```
> data(phiX174Phage)
> names(phiX174Phage)

[1] "Genbank" "RF70s"   "SS78"    "Bull"    "G97"     "NEB03"

> nebPhage <- phiX174Phage[[which(names(phiX174Phage) ==
+     "NEB03")]]
> nebPhage <- DNAString(paste(as.character(nebPhage), as.character(substr(nebPhage,
+     1, 34)), sep = ""))
> nebPhage

  5420-letter "DNAString" instance
seq: GAGTTTTATCGCTTCCATGACGCAGAAGTTAA...GTTTTATCGCTTCCATGACGCAGAAGTTAACA
```

Exercise 33
Show an aligned/unaligned breakdown of the read counts in the "Hoover" Solexa QA plot. This can be accomplished through the following steps:

1. *Use the* `PDict` *function to create pattern dictionaries for the cleaned reads and their reversed complement.*
2. *Use the* `countPDict` *function to find which reads map at least once to the phage genome.*
3. *Create an indicator variable that states whether or not a unique sequence maps to the phage genome.*

```
> posPDict <- PDict(DNAStringSet(names(cleanReadTable)),
+     max.mismatch = 2)
> negPDict <- PDict(reverseComplement(DNAStringSet(names(cleanReadTable))),
+     max.mismatch = 2)
> whichAlign <- rep(FALSE, length(phageReadTable))
> whichAlign[-whichNotClean] <- (countPDict(posPDict, nebPhage,
+     max.mismatch = 2) + countPDict(negPDict, nebPhage,
+     max.mismatch = 2) > 0)
```

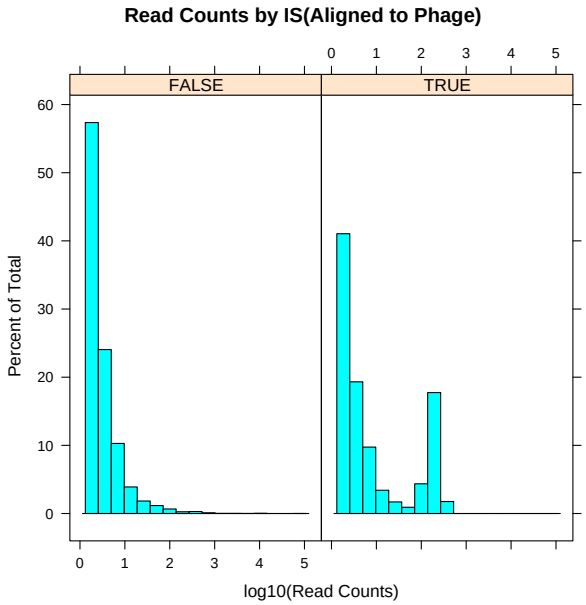

Figure 2: Hoover Plot Deconstructed

Exercise 34
Count the number of unique reads that map to the genome as well as the overall percentage of reads that map to the genome.

```
> table(whichAlign)

whichAlign
 FALSE   TRUE
312787 196626

> round(sapply(split(phageReadTable, whichAlign), sum)/sum(phageReadTable),
+     2)

FALSE  TRUE
 0.19  0.81
```

Exercise 35
Create a histogram, conditioned on alignment status, that shows the "Hoover" plot mentioned in the Short-Read vignette.

```
> print(histogram(~log10(phageReadTable[phageReadTable >
+     1]) | whichAlign[phageReadTable > 1], xlab = "log10(Read Counts)",
+     main = "Read Counts by IS(Aligned to Phage)"))
```

5 Session Information

```
> sessionInfo()
```

```
R version 2.8.0 Patched (2008-10-27 r46787)
i386-apple-darwin9.5.0

locale:
C/C/C/C/C/en_US.UTF-8

attached base packages:
[1] tools     stats     graphics  grDevices utils     datasets
[7] methods   base

other attached packages:
[1] BSgenome.Mmusculus.UCSC.mm9_1.3.11
[2] BSgenome_1.10.3
[3] ShortRead_1.0.3
[4] lattice_0.17-15
[5] Biobase_2.2.1
[6] Biostrings_2.10.6
[7] IRanges_1.0.5

loaded via a namespace (and not attached):
[1] Matrix_0.999375-16 grid_2.8.0
```