Short read quality assessment, remediation, and manipulation – lab

Martin Morgan*

June 20-23, 2011

These exercises introduce key *Bioconductor* packages and data structures for working with sequence data. We start by loading a package designed for the course.

```
> library(EMBL2011)
```

The package contains data sets, helper functions, and the course material. In particular, the slides and labs are available by starting *R* and entering

```
> browseVignettes(package="EMBL2011")
```

This launches a web browser; the links labeled `R` contain scripts that can be run as a short-cut to completing exercises.

1 Warming up

This section is for those who are not really comfortable with *R*; others may skip it. *R* is a computer programming language. It has some familiar (to programmers) data types and instructions, but has many unique features; in many ways it is easier to learn *R* as a non-programmer than as someone with experience in another programming language. Welcome!

A key concept in *R* is the variable. Variables are vector-valued and can be of different types.

```
> x <- 1:10                          # integer vector, 1, ..., 10
> y <- c(TRUE, TRUE, FALSE)          # logical vector
> const <- c(pi=3.14159, e=2.718282) # named numeric vector
> misc <-                            # list of heterogeneous types
+     list(x=1:10, alph=c("a", "b", "c"))
```

Vectors can be sub-set by logical value, name, or integer index. An important distinction is between subsetting with a single square bracket, which returns an instance of the original object, versus element selection with a double square bracket, which returns a specific element.

*mtmorgan@fhcrc.org

```
> x[4:2]
> const["pi"]   # a list of length 1, containing an element pi
> const[["pi"]] # not a list; the element pi
```

Vectors are a basic *R* type, but *R* itself and many additional packages define *classes* representing complicated data structures that the user is expected to manipulate.

```
> library(Biostrings)
> dna <- DNAString("AAACTCTAT")
> translate(dna)
```

The *R* help system is essential to discovering how to work with classes. The elements of this system include interactive use

```
> class(dna)
```

coupled with 'man' pages accessible within *R* or *via* a web browser.

```
> ?DNAString
> help.start()  # start the web-based help system
```

Bioconductor packages include *vignettes* that provide a rich textual description of how the package is supposed to be used.

```
> browseVignettes(package="ShortRead")
```

The package vignette page also lists `.R` files. These contain the script used in the vignette. Cut-and-paste from the script into an *R* session to speed through an exercise.

2 RNA-seq

Start an *R* session and evaluate the following lines of code, adjusted to reflect the location of data available for this workshop.

```
> fastqFiles <- file.choose()
> bamFiles <- file.choose()
> ## check
> stopifnot(length(list.files(fastqFiles)) == 2)
```

For these exercises we use a subset of the data from [1]. The experiment is described in detail later in the course, but involved RNAi combined with mRNA-seq in *D. melanogaster*. The data itself is available in GEO as part of experiment GSE18508. We look at a subset of samples, summarized in Table 1. There are three biological replicates of untreated and RNAi. The data were collected over a period when there were very rapid changes in technology; the samples with multiple runs are single-end reads, with fewer reads per run and hence several runs per sample. Later samples were paired-end runs with more reads per sample.

Table 1: GEO records from GSE18508

Id	Sample	Run
708	S2_DRSC_Untreated-1	SRR0317086
709		SRR0317096
710		SRR0317106
711		SRR0317116
712		SRR0317126
713		SRR0317136
714	S2_DRSC_Untreated-3	SRR0317147
715		SRR0317157
716	S2_DRSC_Untreated-4	SRR0317168
717		SRR0317178
718	S2_DRSC_CG8144_RNAi-1	SRR0317189
719		SRR0317199
720		SRR0317209
721		SRR0317219
722		SRR0317229
723		SRR0317239
724	S2_DRSC_CG8144_RNAi-3	SRR0317240
725		SRR0317250
726	S2_DRSC_CG8144_RNAi-4	SRR0317261
727		SRR0317271

2.1 Reads

Exercise 1

This exercise reads in 2 fastq files. These contain sequence and quality information from 1 million reads of a paired end run; the file name ending with '`_1.fastq`' is the first mate pair. The reads have been sampled randomly from a larger file.

a. *Use `list.files` with the `pattern` argument to retrieve the full path to the two files in the `fastqFiles` directory.*

b. *Name the elements of the list with the base name of each file path (i.e., the file name; use `basename`), using `sub` to remove `.fastq` from the file name.*

c. *Use `lapply` and the `readFastq` function to read each file in to your R session. Explore the object you have created.*

Solution: Here we list all files in the path `fastqFiles` whose name matches the pattern `fastq`, returning the full path to the file. `fls` is now a character vector of length 2. We name the elements of the vector with their base name, substituting the extension `.fastq`, with the empty string.

```
> fls <- list.files(fastqFiles, "fastq", full=TRUE)
> names(fls) <-sub(".fastq", "", basename(fls))
```

Next, iterate over the vector of file names, reading in each. The result is a list, each element of which is a *ShortReadQ* object. discover the class of an object with `class`. Find out information about the class with, e.g., `?ShortReadQ`. The first line of code below is surrounded by parentheses `()`; this is a convenience for both performing the action inside the parentheses (e.g., reading the files) and displaying the result (printing the content of `fq`). Notice how the output list retains the names of the input list, helping to avoid book-keeping mistakes.

```
> (fq <- lapply(fls, readFastq))
> class(fq)       # list
> class(fq[[1]])  # list subset with [[, class ShortReadQ
> names(fq)
```

The first line of code is surrounded in parentheses `(fq <- ...)`. This is a shortcut to both assign the variable and print the result.

Exercise 2
The goal of the next several exercises is to summarize mono- and dinucleotide use in our reads. Write a short function that takes as its input an instance of ShortReadQ, and returns a vector containing the number of A, C, G, T, and N (uncalled) nucleotides summarized over all reads. To do this, your function will

a. *Extract the reads from an instance of the ShortReadQ class, using* `sread`.

b. *Calculate the frequency of the letters used in each read, using* `alphabetFrequency`. *Consult the help page for this function,* `?alphabetFrequency`, *and use the* `baseOnly` *and* `collapse=TRUE` *arguments to ignore IUPAC ambiguity letters (there are none in our reads) and to report the nucleotide counts over all reads (rather than for each read separately).*

Solution: A function taking a *ShortReadQ* instance and returning nucleotide counts is

```
> fun <- function(x)
+     alphabetFrequency(sread(x), baseOnly=TRUE, collapse=TRUE)
```

Applying this to each element of our list of *ShortReadQ* instances results in (using `sapply` produces a matrix; note that the names of the files are again carried through).

```
> (mono <- sapply(fq, fun))
```

Exercise 3
Write a function that accepts as input a ShortReadQ instance and returns dinucleotide frequencies. Use the `dinucleotideFrequency` *function to do the calculation. The operation is a little trickier, because* `dinucleotideFrequency` *does not have an option to collapse counts over all reads. Instead...*

a. *Use* `dinucleotideFrequency` *to create a matrix, with each column representing a different combination of nucleotides and each row a different read.*

b. *Use* `colSums` *to sum up each column.*

Apply this function to each element of `fq`.

Solution: A function taking a *ShortReadQ* instance and returning dinucleotide counts is

```
> fun <- function(x)
+     colSums(dinucleotideFrequency(sread(x)))
```

Applying this function to the list of *ShortReadQ* instances and displaying the first four records of the result (using `head`) is

```
> head(di <- sapply(fq, fun), 4)
```

The idiom `head(di <- ...)` is another short-cut that simultaneously assigns a value to `di` and uses the object as the first argument to the `head` function; `head` prints the first 6 (by default; 4 in the example above) lines of `di`.

Exercise 4
The following exercises asks about the distribution of 'GC' nucleotide content between reads. Start by writing a function that...

a. *Uses* `alphabetFrequency` *to determine nucleotide use, but omit the* `collapse` *function argument so that the result is a matrix summarizing use in each read.*

b. *Subsets the nucleotide use matrix to select the 'G' and 'C' columns, and uses* `rowSums` *to sum the GC content for each read (row of the matrix).*

c. *Uses* `tabulate` *to count how many times reads with 0, 1, ... 37 'GC' occurs. A trick here is that* `tabulate` *ignores zeros; by adding 1 to each count, we get a vector where the first element is the number of reads with 0 'GC' nucleotides, the second element is the number of reads with 1 'GC' nucleotide, and so on.*

Solution: The distribution of GC content is:

```
> fun <- function(x)
+ {
+     abc <- alphabetFrequency(sread(x), baseOnly=TRUE)
+     gcPerRead <- rowSums(abc[,c("G", "C")])
+     wd <- unique(width(sread(x)))
+     tabulate(1 + gcPerRead, 1 + wd)
+ }
> (gc <- lapply(fq, fun))
```

Exercise 5
Visualization is an important tool for gaining insight. R has flexible built-in graphics commands, but additional packages provide expressive (lattice) and pretty (ggplot2) alternatives. This lab uses lattice. Most lattice functions expect a data frame in 'long' format, where one or more columns contain the data to be plotted and additional columns are indicator variables indicating which group the correspond rows belongs to.

Start by making a vector `bins` *to indicate GC content (0, 1, . . .). Use this in creating a 'long' data frame with. . .*

a. *A column containing the counts calculated previously, using the* `unlist` *function to make a single vector (rather than a list of length two, each with a vector).*

b. *A column containing our 'GC' content from* `bins`*, scaled to represent a proportion*

c. *A column indicating which sample the row comes from. The sample identifier is taken from the names of* `gc` *object, each repeated as many times as there are elements in the corresponding count vector.*

A powerful and expressive feature of lattice is the use of a formula to describe the relationship between plotted variables, in this case Count as a function of GC content. The `group` *argument plots subsets of data in the same panel.* `type` *control what gets plotted – a background grid and both lines and points.* `pch` *controls the type of point (try* `example(points)`*).* `auto.key` *control display of the legend.* `file` *is used in this vignette; it is not part of lattice.*

Solution: The following creates a 'long' data frame. . .

```
> bins <- seq_along(gc[[1]]) - 1
> df <- data.frame(Count=unlist(gc, use.names=FALSE),
+                  GC=bins / bins[length(bins)],
+                  Sample=rep(names(gc), sapply(gc, length)))
```

. . . and plots the result, Figure 1

```
> xyplot(Count~GC, group=Sample, df, type=c("g", "b"), pch=20,
+        auto.key=list(lines=TRUE, points=FALSE, columns=2),
+        file="gc")
```

Exercise 6
A simple null expectation is that dinucleotide frequencies are the product of the mononucleotide frequencies.

a. *Calculate the frequency of mononucleotides (ignoring 'N' for simplicity), and use the* `outer` *product, applied to each sample, to determine the expected dinucleotide frequency.*

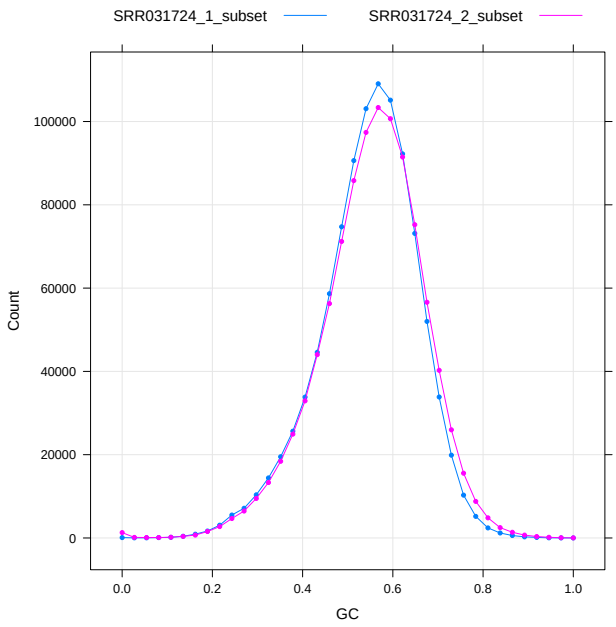

Figure 1: Read GC content.

b. *Express expected dinucleotide frequencies in each sample as counts by multiplying by the sample total dinucleotide count.*

c. *Use* `chisq.test` *to calculate a* χ^2 *test on each sample.*

Create a graphical display of the results, and comment on the results.

Solution:

```
> ## outer product of mononucleotides frequencies
> f <- mono[-5,] / colSums(mono[-5,])
> exp0 <- apply(f, 2, function(x) as.vector(outer(x, x)))
> ## expected values, as counts
> n <- colSums(di)
> exp <- exp0 * n[col(exp0)]
> mode(exp) <- "integer"
> head(exp, 3)
> ## Chi-squared test
> for (i in seq_len(ncol(di)))
+     print(chisq.test(cbind(di[,i], exp[,i])))
```

To display the data using *lattice*, create a 'long' data frame:

a. Coerce the observed and expected matrix to vectors, with `as.vector`.

b. Provide an indicator variable based on the row names of the observed matrix. *R*'s recycling rules mean that this vector is extended to the correct length. *R* represents a matrix as a vector with entries from column 1 first, column 2 next, etc. This means that the recycled row names align correctly with the vectors coerced from the matrix of observed or expected values.

c. Provide an indicator variable based on the column names of the matrix. Recycling can't be used directly (why not?); the `col` function provides an index that places the column names in the correct position.

d. Specify `stringsAsFactors=FALSE`, because we do not wish our string variables – `Pair` and `Sample` – to be treated as factors.

The conditioning bar | in the *lattice* formula indicates that a separate panel should be plotted for each sample (how does this differ from `group`?). the `panel` argument describes how each panel will be created – drawing a horizontal line of color gray (`panel.abline`) followed by placement of text at particular coordinates (`panel.text`; the 'usual' xy plot panel function is `panel.xyplot`). Note the use of ... to forward arguments not directly used by our custom panel function.

```
> ## display
> df <- data.frame(Observed=as.vector(di), Expected=as.vector(exp),
+                  Pair=rownames(di), Sample=colnames(di)[col(di)],
+                  stringsAsFactors=FALSE)
> xyplot(Observed - Expected ~ Expected | Sample, df, aspect="iso",
+        panel=function(..., subscripts) {
+            panel.abline(h=0, col="gray")
+            panel.text(..., labels=df$Pair[subscripts])
+        }, file="dinuc")
```

The result is in Figure 2, with the following interesting points:

a. The χ^2 test indicates departure from the null.

b. There is a strong bias against TA dinucleotides; knowledge of TA use in the reference genome would help us to understand whether this is a biological result or a technical artifact.

c. Similarly, G, C nucleotide pairs are most common.

d. The χ^2 test is not really satisfactory, because each nucleotide in a read is counted twice (as the first and then second nucleotide in the pair). Any suggestions for improving the analysis?

Exercise 7
Nucleotide counts in each cycle of a read provide important insight into sample preparation and technological artifacts.

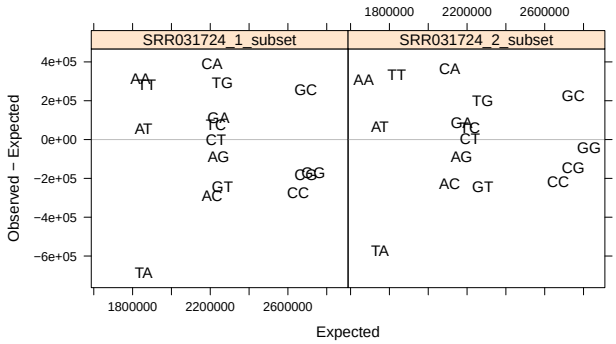

Figure 2: Deviation of observed from expected dinucleotide counts.

a. *Write a short function that extracts reads from a ShortReadQ object and uses* `alphabetByCycle` *to summarize nucleotide use by cycle. Subset the value returned by* `alphabetByCycle` *to include only the nucleotides A, C, G, T.*

b. *Apply this function to* `fq`*, and explore the result.*

c. *Display the result using* `lattice`*. To do this, cast the results into a long data frame, and then use* `xyplot`*. It is easiest to start with a simple plot, and to subsequently adjust display and formatting.*

Solution: Here we calculate and plot DNA alphabet use as a function of cycle. The `dnacol` object is a named vector defined in the *EMBL2011* package; the names are nucleotides and the values codes that represent colors.

```
> fun <- function(x)
+     alphabetByCycle(sread(x))[names(dnacol),]
> abc <- lapply(fq, fun)
> abc[[1]][,1:5]
```

The following creates a 'long' data frame and displays the result using *lattice*.

```
> ## create a 'long' data frame
> nuc <- rownames(abc[[1]])
```

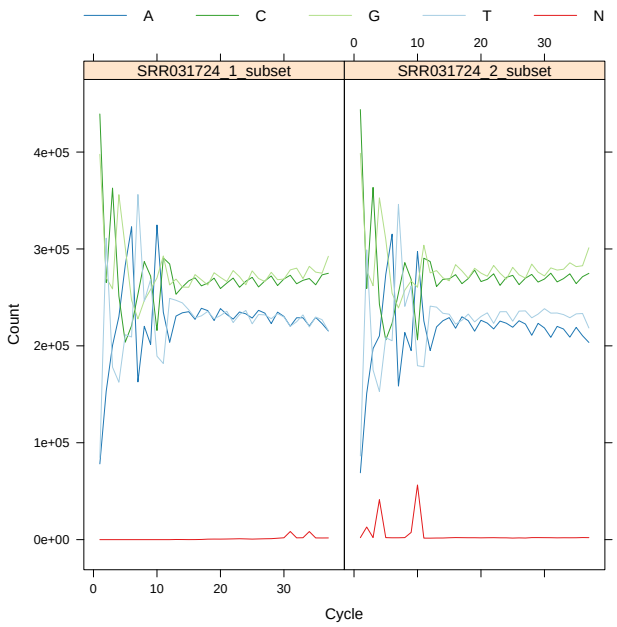

Figure 3: Alphabet-by-cycle.

```
> ncycle <- sapply(abc, ncol)
> cycle <- rep(sapply(ncycle, seq), each=5)
> sample <- rep(names(abc), sapply(abc, length))
> df <- data.frame(Count=unlist(abc, use.names=FALSE),
+                  Nucleotide=factor(nuc, levels=names(dnacol)),
+                  Cycle=cycle, Sample=sample)
> ## plot Count as a function of cycle, with separate panels
> ## for Sample, and with Nucleotide used to group lines
> ## within a panel.
> xyplot(Count ~ Cycle | Sample, group=Nucleotide, df, type="l",
+        key=list(lines=list(col=unname(dnacol)), columns=5,
+          text=list(lab=names(dnacol))),
+        col=dnacol, file="abc-read")
```

Interesting points (some well-known) from Figure 3 include:

a. Primers introduce initial bias, through cycle ≈ 12.

b. Under a null of uniform coverage, lines should be horizontal; but the plots suggest that A, T decrease in frequency as cycles progress. This is much more pronounced in early (GAI) Solexa / Illumina runs.

c. There is a weak but consistent periodic bias – e.g., every second cycle has relatively more A, T.

Exercise 8
Base quality decreases as cycle number increases. Calculate the average quality per cycle, and display the result. To calculate quality per cycle, write a short function that...

a. *Extracts the quality score from a ShortReadQ instance using the* `quality` *function.*

b. *Coerces the quality score to a numerical matrix representation using the* `as` *function. This matrix has as many rows as there are reads, and as many columns as there are cycles.*

c. *Calculate the average quality score of each cycle using* `colMeans`*.*

d. *Develop a second function that calculates an overall read quality using* `rowMeans`*.*

Apply these functions to each ShortReadQ instance, and visualize the result using lattice.

Solution: Calculating quality by cycle requires translating character encodings to their numeric representation. The average quality of the second read of the pair is lower than that of the first read.

```
> fun <- function(x)
+     colMeans(as(quality(x), "matrix"))
> abc <- sapply(fq, fun)
```

The following creates a 'long' data frame and displays the result.

```
> df <- data.frame(Mean=as.vector(abc), Cycle=seq_len(nrow(abc)),
+                  Sample=rep(colnames(abc), each=nrow(abc)))
> xyplot(Mean ~ Cycle, group=Sample, df, type="b", pch=20,
+        auto.key=list(lines=TRUE, points=FALSE, columns=2),
+        file="abc-quality")
```

Using the average quality of each read as a measure of 'overall quality', we have

```
> fun <- function(x)
+     rowMeans(as(quality(x), "matrix"))
> qual <- lapply(fq, fun)
> ## display -- hexbinplot to avoid plotting a million points
> xyplot(SRR031724_2_subset~ SRR031724_1_subset, qual,
+        aspect="iso", xbins=50,
+        panel=function(...) {
+            panel.hexbinplot(...)
+            panel.abline(0, 1, lty=3)
+        },
+        file="quality-by-read")
```

Interesting, mostly well-known, points from Figure 4 include:

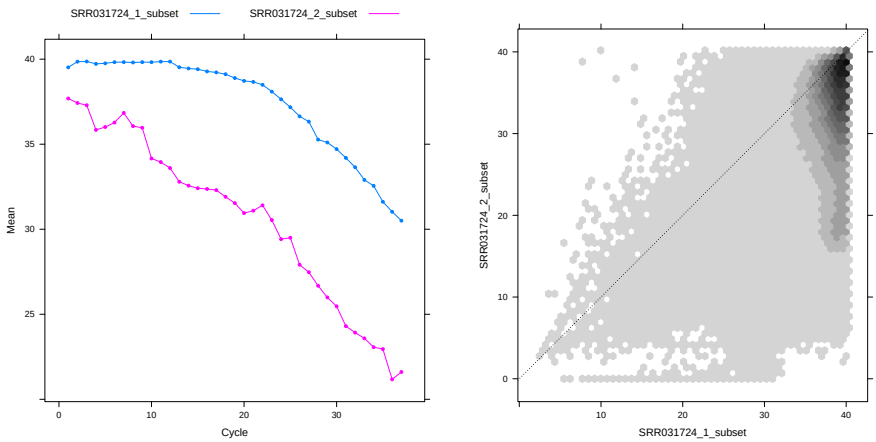

Figure 4: Quality-by-cycle (left) and distribution of read quality (right).

a. Average quality declines with cycle.

b. The second mate of paired reads is consistently lower quality.

c. Average quality and error are related (larger deviation from the diagonal at higher quality scores).

Exercise 9
Individual read sequences can be represented 1, 2, ... times in a sample. The overall extent to which reads are duplicated can indicate depth of coverage. Singleton sequences (represented exactly once) and sequences with very high representation often reflect limitations of technology. Create and interpret a figure that plots the cumulative number of reads as a function of the number of times a read occurs in a sample.

Solution:

```
> fun <-
+     function(x, sample)
+ {
+     t <- tables(sread(x)[c(TRUE, FALSE)], n=0)[["distribution"]]
+     with(t, data.frame(nOccurrences=nOccurrences,
+                        CummulativeReads=cumsum(nOccurrences * nReads),
+                        Sample=sample))
+ }
> tbl <- do.call(rbind, Map(fun, fq, names(fq)))
> ## display
> xyplot(CummulativeReads ~ log(nOccurrences), group=Sample, tbl,
```

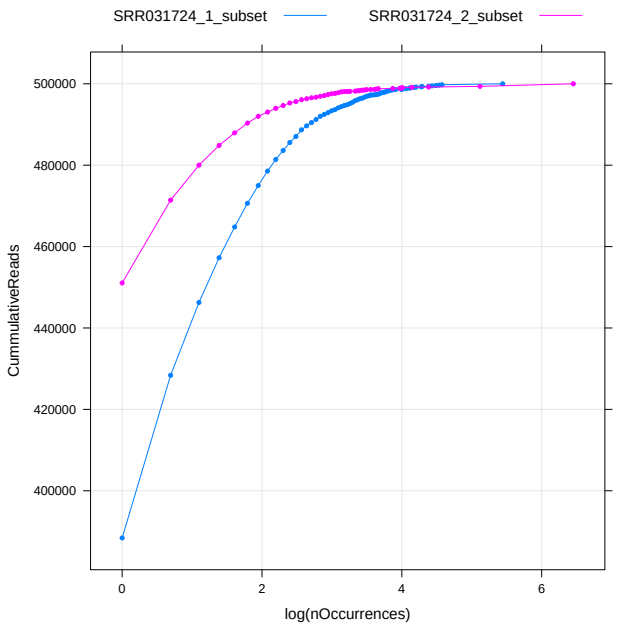

Figure 5: Cumulative frequency of reads occurring 1, 2, ... times.

```
+         type=c("b", "g"), pch=20,
+         auto.key=list(lines=TRUE, points=FALSE, columns=2),
+         file="freqseq")
```

Interesting points (Figure 5):

a. The intercept (singleton reads) will decrease as sample size increases – repeated observation of the same reads. The large value here implies limited coverage.

b. That the second of the read pair is above the first might partly reflect elevated error rates.

c. The most abundant reads in the second of the read pair reflects bias – particular sequences (degenerate?) become super-abundant.

Exercise 10

ShortRead can create a report summarizing several `fastq` *files. A short script (see the appendix) was used to summarize all fastq files; here we read in the summary and generate the report.*

a. *Load the data R object* `qa_GSM461176_81`*, representing the qa summary of GSM records 461176 through 461181. The path to the object is*

```
> qaFile <- system.file("data", "qa_GSM461176_81.rda",
+                       package="EMBL2011")
```

b. *Use the* `report` *function to generate a report, by default in a temporary directory.*

c. *Browse the report using* `browseURL`*.*

The 'Id' column in Table 1 is used to identify samples in the QA report; `_1` *or* `_2` *refers to the first and second mate in paired end runs. Reflect on the quality of the reads in this experiment.*

Solution:

```
> load(qaFile)
> rpt <- report(qa_GSM461176_81)
> browseURL(rpt)
```

If for some reason the report generation fails, a copy is available at

```
> rpt <- system.file("GSM461176_81_qa_report", "index.html",
+                    package="EMBL2011")
> browseURL(rpt)
```

Interesting points:

a. Substantial variation in number of reads per sample; low read counts in samples 709, 719, 723.

b. GC content of 718-723 is unusual – all from `S2_DRSC_CG8144_RNAi-1`.

c. Samples are quite heterogeneous in distribution of read qualities.

d. Read distributions are not super-saturated – opportunity for greater sequencing? Sample 719 is unusual – disproportionate representation of very common sequences.

e. All samples show initial primer bias; some samples, e.g., 708, 713, 717_2 show cycle-specific base trends. Sample 723 has unusual final nucleotides.

f. Samples differ in read length, maximum quality – likely different machines and chemistry. 708-713, 718-723 are longer with low-quality tails.

2.2 Alignments

The primary goal of this exercise is to become familiar with the *Rsamtools* package for querying BAM alignment files

Exercise 11
The goal of this exercise is to open several BAM files as a BamFileList, and to query one BAM file for the information about the reference sequences to which reads have been aligned (using the `seqinfo` *function) and the software tools used to perform the alignment (using* `scanBamHeader` *and parsing the return value).*

Solution: Create a list of `BamFile` objects, each pointing to a file and its index. Using a `BamFile` avoids loading the index each time the file is used.

```
> fls <- list.files(bamFiles, "bam$", full=TRUE)
> bam <- open(BamFileList(fls))
> names(bam) <- sub(".bam", "", basename(fls))
```

Summary information is available in the header of each file, e.g., querying for a summary of (reference) sequences and their lengths, or as tags available in the `text` element of the header (see the samtools web site for details of header content, including meaning of tags). For instance, our BAM files record the command line used to generate them.

```
> seqinfo(bam[[1]])
> h <- scanBamHeader(bam[[1]])[["text"]]
> noquote(unname(sapply(h[["@PG"]], strwrap)))
```

Exercise 12
Common operations on BAM files are available as functions. For instance, `countBam` *counts the number of reads in each BAM file. A more interesting use is to specify the* `param` *argument to this and other Rsamtools functions.*

a. *Create a GRanges object containing coordinates of four* Drosophila *genes; see the solution for one such set.*

b. *Create a ScanBamParam object to specify the regions on which BAM file operations are to be performed.*

c. *Use* `countBam` *to count reads in each of the regions.*

d. *Aligned reads have various flags that summarize the status of the read or alignment. Use* `countFlags` *to summarize flags of reads in one of the genes.*

Solution: `countBam` can be used to retrieve the number of reads aligned in each BAM file. N.B., `countBam` is not meant to be used for counting reads in complex regions as in an RNAseq analysis; see `?countGenomicOverlaps`.

```
> ## do not evaluate
> cnt <- countBam(bam)
```

The `ScanBamParam` function creates an object that allows easy access to particular portions of the file. For instance, the `which` argument can be used to select which genomic regions are accessed. The `which` argument could be one of several different objects, for instance a `GRanges` containing chromosomes and the regions of interest.

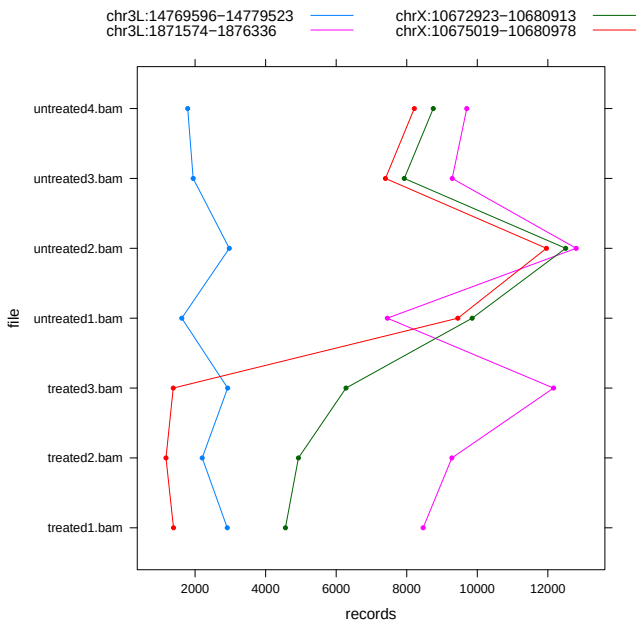

Figure 6: Number of reads aligning to regions of interest.

```
> which <-
+     GRanges(c("chr3L", "chrX", "chrX", "chr3L"),
+             IRanges(c(1871574, 10675019, 10672923, 14769596),
+                     c(1876336, 10680978, 10680913, 14779523)))
> param <- ScanBamParam(which=which)
> head(cnts <- countBam(bam, param=param))
> cnts$rgn <- with(cnts, sprintf("%s:%d-%d", space, start, end))
> xyplot(file~records, group=rgn, cnts, type="b", pch=20,
+        auto.key=list(lines=TRUE, points=FALSE, columns=2),
+        file="bam-regions")
```

The results are shown in Figure 6.

BAM files contain a `flag` field encoding information about the aligned reads, e.g., the strand to which they aligned, whether they are mate paired, whether the pairs are 'proper' in the eyes of the alignment tool producing the BAM file. Here we summarize reads satisfying various flags (note that a read may be tallied under more than one category, e.g., is a proper pair and is on the negative strand).

```
> param1 <- ScanBamParam(which=which[4])
> countFlags(bam, param=param1)
> bam2 <- bam[-c(1, 4, 5)]
```

Exercise 13
The previous exercise introduced BAM files, some aspects of the ScanBamParam class, and built-in functions for retrieving information from the file.

This and the next exercise illustrates how Rsamtools can be used more creatively. As a first step, suppose one is interested in the 'insert' distance between mapped reads of a mate pair. Construct a `ScanBamParam` *object with:*

a. *The* `flag` *argument, determined by the* `scanBamFlags` *function, to select the first read of each 'proper' mate pair.*

b. *The* `which` *argument and GRanges class created above, to restrict the query to a particular region of the genome.*

c. *The* `what` *argument to return the* `pos` *(alignment position) and* `mpos` *(mate read alignment position; arguments are described on* `?scanBam` *help page).*

Use this, the `BamFileList` *object, and the* `scanBam` *function to query the BAM files for the relevant information. The insert width is the absolute value of the difference between the* `pos` *and* `mpos` *locations; calculate this for each sample, and summarize as cumulative distribution.*

Solution: Here is the `ScanBamParam` object:

```
> param2 <- ScanBamParam(
+     what=c("pos", "mpos"), which=which[4],
+     flag=scanBamFlag(isProperPair=TRUE, isFirstMateRead=TRUE))
```

Now query the BAM files, and take a look, using `str`, at the list-of-list-of-lists returned:

```
> iwd0 <- lapply(bam2, scanBam, param=param2)
> str(iwd0)
```

Starting at the innermost list, the absolute value of the difference between each element of the `pos` and `mpos` vectors represents the insert width (not quite – what distance does this actually measure? What additional information would be required to measure the insert distance? Is this information available from the BAM file?). This information is itself represented in a list, so write a function that extracts the first element of a list, and then calculates the distance between mate pairs. Use `lapply` to calculate this distribution for each sample.

```
> fun <- function(elt)
+     with(elt[[1]], abs(pos - mpos))
> str(iwd <- lapply(iwd0, fun))
```

To visualize the distribution, use (from the *lattice* package) the `make.groups` function, invoked with `do.call`, to create a data frame with two columns, and suitable for use in `densityplot`.

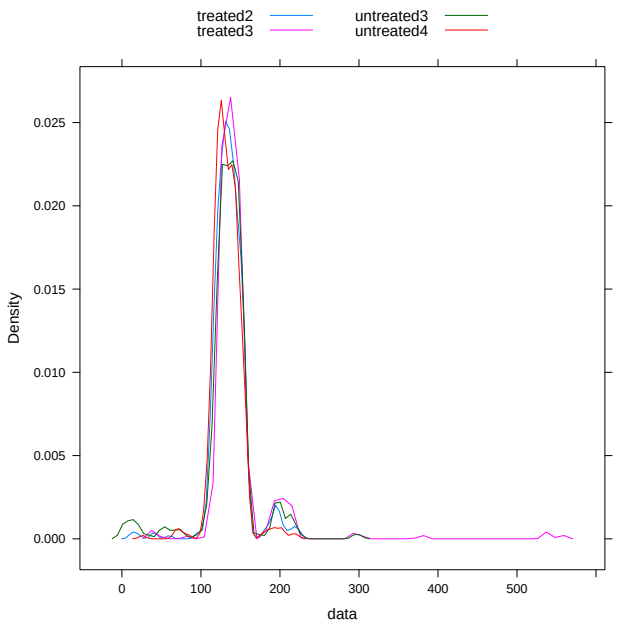

Figure 7: Insert size of proper mate pairs at `chr3L:1871574-1876336`.

```
> df <- do.call(make.groups, iwd)
> densityplot(~data, group=which, df, plot.points=FALSE,
+             auto.key=list(lines=TRUE, points=FALSE, columns=2),
+             file="insert")
```

3 Microbiome / 454 sequences

Microbiome sequencing projects characterize the representation of organisms in well-defined communities (e.g., flora of the human gut). Metagenomics aims one step further, asking about the biological functions provided by the suite of organisms found in a community. Both endeavors rely on assignment of longer (200-400bp) sequence reads to a reference database of taxa.

A typical microbiome work flow involves collection of DNA samples from a large number (10's to 100's) of individuals. Targeted PCR enriches samples for one or a few phylogenetically informative markers, e.g., specific regions of 16s RNA. A bar code is added to PCR-enriched DNA to allow sample multiplexing. Sequencing uses a technology, e.g., 454, that produces reads that are long enough to provide phylogenetic signal. Sequence processing requires reads to be de-multiplexed. There are typically PCR artifacts (e.g., primer sequence) to be cleaned. Pre-processed reads are aligned to curated collections of reference

genomes, with representation of taxa in the sample inferred from reads aligned to reference genome.

These exercises focus on read manipulation prior to classification, with emphasis on

1. Data input.
2. Sub-sequence extraction and manipulation
3. Pattern matching.

The data are a subset of bacterial 16s sequences sampled from a human body cavity. Down-stream analysis (beyond the scope of this tutorial) can use excellent *R* packages for community and phylogenetic analysis (e.g., *ape*, *vegan*).

Exercise 14
454 technology is different from the Illumina platform. Reads are initially made available as 'flows' in the `sff` *file format. Unfortunately, there are no Bioconductor packages that work directly with* `sff` *files. Instead, analysis begins with fastq-like data, typically presented as pairs of fasta sequence (*`.seq`*) and quality (*`.qual`*) files.*

Input the sample data using the `read454` *function from the ShortRead package. The data is represented as a ShortReadQ object, the same as seen earlier. Note that the read widths are longer (up to 342 cycles) and variable. Quality scores also follow a different pattern. Subset the reads to contain only those with width 250 or more, then use the* `narrow` *function to look at the average quality of the trailing 250 nucleotides.*

Solution: Here we read in the data.

```
> rp <- RochePath(barFiles)
> (bar <- read454(rp, "1.*fna", "1.*qual"))
> summary(width(bar))
```

Here we select reads with width 254 and look at the alphabet cycle at positions 100-110. Reads with this width

```
> bar254 <- bar[width(bar) == 254]
> alphabetByCycle(narrow(sread(bar254), 101, 110))[1:4,]
```

Exercise 15
Microbiome studies generally benefit from many individuals and relatively fewer sequences. Samples are therefore multiplexed by preceding the target sequence with a bar code, in this case associated with the first 8 nucleotides of each read.

a. *Use* `narrow` *function to isolate the bar codes.*
b. *Use* `table` *to determine the occurrence of each code.*

c. *Create a subset of reads corresponding to the most common bar code, and*

d. *Use* `narrow` *again, this time trimming the bar code and two adapter nucleotides (nucleotides 1-10) from the sequences.*

Solution: The following narrows the reads to positions 1-8 (containing the bar code), and then tabulates and sorts, using standard *R* functions, the number of times each bar code occurs. We then focus on the most abundant bar code.

```
> codes0 <- narrow(sread(bar), 1, 8)
> codes <- as.character(codes0)
> (cnt <- sort(table(codes), decreasing=TRUE)[1:5])
> (aBar <- bar[codes==names(cnt)[[1]]])
```

Now that `aBar` consists of reads from a single bar code, we can remove those nucleotides using `narrow` to focus on the 11th through final nucleotides

```
> (noBar <- narrow(aBar, 11, width(aBar)))
```

Exercise 16
A typical microbiome sample preparation involve targeted PCR amplification. In this case the (redundant) PCR primer is present in the read. Further, the primer is not always present fully intact. The primer sequence is

```
> pcrPrimer <- "GGACTACCVGGGTATCTAAT"
```

Trim the primer using the pattern-matching function `trimLRPatterns`*. Summarize the amount of trimming that has occurred by tabulating the difference between the width of the sequences with the bar codes removed, and the same sequences with the primer trimmed.*

Solution:

```
> (trimmed <-
+     trimLRPatterns(pcrPrimer, subject=noBar, Lfixed=FALSE))
> table(width(noBar) - width(trimmed))
```

References

[1] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Res.*, 21:193–202, Feb 2011.

A Appendix

These functions set paths to point to appropriate locations when the vignette is being produced.

```
> if (!interactive()) {
+     ## set up custom file paths
+     fastqFiles <- "~/EMBL/bigdata/fastq"
+     bamFiles <- "~/EMBL/bigdata/bam"
+     barFiles <- "~/EMBL/bigdata/bar"
+ }
> NA
```

These functions 'wrap' `xyplot` and `densityplot` so that they print to a file when the vignette is processed to a PDF.

```
> ## wrap lattice functions to print to file when run in batch mode
> plt <-
+     if (!interactive()) {
+         function(fun, file, ...) {
+             pdf(sprintf("%s.pdf", file))
+             print(fun(...))
+             invisible(dev.off() )
+         }
+     } else {
+         function(fun, file, ...) fun(...)
+     }
> xyplot <- function(..., file)
+     plt(lattice::xyplot, file, ...)
> densityplot <- function(..., file)
+     plt(lattice::densityplot, file, ...)
> dotplot <- function(..., file)
+     plt(lattice::dotplot, file, ...)
> ## don't open a web browser when run in batch mode
> if (!interactive())
+     browseURL <- function(...) {}
> NA
```

The QA report data was collated with the following script. For each fastq file, we create an identifier `id` from its file name, and read the file in with `readFastq`, and calculate summary statistics using `qa`. `doit` will perform an `lapply` operation on each file, but if the *multicore* package is available the operation will be in parallel. The result `qas` is a list of qa summaries; these are bound together into a single object using `rbind`.

```
> fun <- function(fl) {
+     id <- sub(".fastq$", "", basename(fl))
+     qa(readFastq(fl), id)
```

```
+ }
> fls <- list.files(pattern=".fastq$", full=TRUE)
> doit <-
+     if (suppressWarnings(require("multicore"))) {
+         mclapply
+     } else lapply
> qas <- doit(fls, fun)
> qa_GSM461176_81 <- do.call(rbind, qas)
```