Representing sequencing data in Bioconductor

Mark Dunning mark.dunning@cruk.cam.ac.uk

Last modified: April 5, 2016

Contents

Introduction

The purpose of this practical is to familiarise you with the concept of strings and ranges in Bioconductor. These are fundamental objects upon which many analysis tools for next-generation sequencing data are built-upon.

1 Accessing Genome Sequence

The *BSgenome* package extends the *Biostrings* package to work with whole-genome sequence data. Its main purpose is to permit us to query genome sequences in an efficient manner. Several genomes are available in Bioconductor that have been built using this infrastructure.

Use Case: See what genomes are currently available through Bioconductor

```
library(BSgenome)
available.genomes()
```

comment: If your genome of interest is not currently available in this list, it is possible to create your own package. However, doing this is probably not for the novice user. See the "How to forge a BSgenome data package" vignette in the BSgenome package. `browseVignettes()`

Contained within each *BSgenome* package is a database object that we can use to make queries against. The name of the database is the same as the name of the package. However, as programmers don't like to type much, we often assign this to a shorter name

Use Case: Load the package that contains a representation of the Human Genome version hg19 and rename the database object to something more convenient.

```
library(BSgenome.Hsapiens.UCSC.hg19)
hg19 <- BSgenome.Hsapiens.UCSC.hg19
hg19
```

The names of each chromosome can be found by using the `names` function. Often in sequencing analysis in Bioconductor, we refer to these as the *seqnames*.

Use Case: How many chromosomes does the package have sequences for?

```
seqnames(hg19)
length(seqnames(hg19))
```

Each individual chromosome can be accessed using the double square brackets notation. If you have used lists in R, you might recognise this as the way of accessing items in a list *warning: You have to make sure that the chromosome name that you use is one of the valid names that is found in the* `seqnames` *of the genome.*

Use Case: Get the sequence for Chromosome 22. How many bases long is the sequence?

```
hg19[["chr22"]]
```

When we type the name of a `DNAString`, rather than displaying the contents of the object, R will produce a summary on the screen. This is useful if we don't want our screen filled with text. This summary gives a snapshot of the string contents and prints the overall length. However, if we want to perform further computations that use this length, we will need to calculate it ourselves.

Use Case: Save the length of the chromosome as a variable. *comment: HINT: What function do you use to find the length of a vector?*

```
mylen <- length(hg19[["chr22"]])
mylen
```

1.1 Alphabet Frequencies

The `alphabetFrequency` function is a very useful and efficient tool provided by *Biostrings* for calculating the number of occurences of each DNA base within a given `DNAString`. In fact, it can operate on any alphabet. To tell the function to use only valid DNA bases, we can use the `baseOnly=TRUE` argument. *comment: As usual, we can see the help page for the function for more information about the arguments:* `?alphabetFrequency`

Use Case: Find the overall number of A, C, T and Gs in the sequence for Chromosome 22. Express these as a proportion of the total length of the chromosome.

comment: You should have already calculated the length of chromsome 22 in a previous exercise.

```
alphabetFrequency(hg19[["chr22"]],baseOnly=TRUE)
alphabetFrequency(hg19[["chr22"]],baseOnly=TRUE) / mylen
```

We would hope that the proportion of each base will be roughly the same along the entire chromosome. However, there may be local variations that we could explore. To do this, we need a strategy for dividing the chromsome into equal-sized windows. Lets choose a window size of 1000 bases.

The best way to construct the windows is to use a `GRanges` object. To make such an object, we need to define a list of starting positions in the form of a vector. The `seq` function in R is one of the basic ways of creating a sequence of integers in the form of a vector. To use this function we have to specify a start and end position.

Use Case: Create a vector of start positions that can be used to divide up Chromosome 22 into 1000 base pair intervals. You will need to make sure that the last start position is 1000 less than the length of the chromosome.

```
startpos <- seq(1,mylen-1000,1000)
```

Use Case: Now calculate what the end position of each interval needs to be and create the `IRanges` object.

```
endpos <- startpos + 999
ir <- IRanges(startpos,endpos)
ir
```

comment: Rather than explicitly calculating the position, we can just specify the start postions and a width argument

```
ir <- IRanges(startpos,width=1000)
```

Use Case: Now construct a `GenomicRanges` representation. Choose an appropriate value for the sequence name so that we can operate on chromosome 22.

```
library(GenomicRanges)
rngs <- GRanges("chr22",ir)
```

warning: Again we need to be consistent about the chromosome names. The GRanges function will not check that the chromosome name you have given is valid!

```
sillyrng <- GRanges("sausages",ir)
sillyrng
```

The `getSeq` function can be used to obtain the genomic sequence from a genome object. We can give it a ranges object, and it will get the sequence for each individual range.

Use Case: Get the sequence for each interval that you defined, and then calculate the alphabet frequency.

```
winSeq <- getSeq(hg19,rngs)
winSeq
winAf <- alphabetFrequency(winSeq,baseOnly=TRUE)
head(winAf)
```

You will see that the output of this function is a matrix with 5 columns, and one row for each range. Each column gives the count of each DNA base, plus the end column for other bases (e.g. N). Recall that we can access columns and rows in a matrix using square brackets with row and column indices separated by a comma. e.g. `matrix[row,column]`. We can omit either index to give all rows or columns. `matrix[row,]` or `matrix[,column]`.

Use Case: Which window contains the most A bases? What interval on the genome does this correspond to?

```
aOrd <- order(winAf[,1],decreasing = TRUE)
head(winAf[aOrd,])
rngs[aOrd]
rngs[which.max(winAf[,1])]
```

We can attach extra information to our `GRanges` objects by using the `elementMetadata`. This metadata takes the form of a data frame and can be set and retrieved using the `mcols` function. This is especially convenient when we want to display any the results of any calculations we have performed alongside the actual ranges themselves.

Use Case: Put the alphabet frequency table as the metadata of your `GRanges` object. What effect does this have on how the object is displayed?

```
mcols(rngs) <- winAf
rngs
```

Once data have been stored in the `mcols` of a `GRanges` object, they can be referred to directly by using $ operator; as if they were columns or variables in a data frame.

Use Case: Access the counts of A bases using the $ syntax and reorder the ranges object. Verify that you get the same result as before.

```
rngs[order(rngs$A,decreasing = TRUE)]
```

GC-content is a known bias in NGS analysis that we often try and correct for; for example genes or regions with higher GC content tend to have more reads mapped within them.

Use Case: Calculate the GC content of each window and store these values in the `GRanges` object. Find which windows have over 70% GC content. Which ranges does these correspond to? Which window has the highest GC content.

```
gcContent <- (rngs$G + rngs$C) / 1000
mcols(rngs)$gcContent <- gcContent*100
hist(gcContent)
rngs[rngs$gcContent>70]
rngs[which.max(rngs$gcContent)]
```

When looking at the sequence of the chromosome, you have probably noticed that the start and end of the chromosome is purely N bases. This is referred to as *Masked* sequence. To find particular sequences or motifs in DNA we can use the matchPattern function.

1.2 Finding sequences

The basic usage of matchPattern is to take a basic character string, and to report whereabouts it occurs in another string. For instance, the following example will locate the start codon (ATG) in a defined sequence

```
s1 <- "aaaatgcagtaacccatgccc"
matchPattern("atg", s1)
```

Masked regions on the genome can be identified by long stretches of N bases

Use Case: Use the matchPattern function to find all occurrences of 10 consecutive Ns. Create a GRanges representation of the result.

```
n.matches <- matchPattern("NNNNNNNNNN", hg19[["chr22"]])
n.matches
maskRegions <- GRanges("chr22",
                       IRanges(start=start(n.matches),end=end(n.matches))
)
```

You'll notice that the ranges object we create contains intervals that are adjacent to each other and are part of the same pattern of N bases. For analysis purposes though, we probably want simplify this object so that we get unique regions.

Use Case: Create a set of unique masked regions using reduce to combine adjacent regions

```
maskRegions <- reduce(maskRegions)
```

If we want the places on the chromosome that are **not** masked, this is easily achieved with functions in IRanges.

Use Case: Create a GRanges object to represent all un-masked regions. Try using the gaps or setdiff functions.

```
unmaskRegions <- gaps(maskRegions)
unmaskRegions
wholechr <- GRanges("chr22",IRanges(1,mylen))
setdiff(wholechr,maskRegions)
```

2 Reading bam alignments

An obvious application of the *IRanges* and *GRanges* infrastructure is to facilitate the storage and querying of genomic alignments. The *Rsamtools* is one such package that allows aligned reads to be imported into R. However, we will use functionality from the *GenomicAlignments* package for simplicity.

An example bam file is provided in the `Day1` folder. If you wish, it may help your comprehension to also view this file in **IGV**.

2.1 Exploring and filtering the reads

Use Case: Read the example bam file into R as a `GRanges` object. How many reads are in the file?

```
library(GenomicAlignments)
mybam <- "Day1/HG00096.chr22.bam"
bam <- readGAlignments(file=mybam)
bam
```

comment: In IGV, try jumping to the region where the first few reads in this bam file are aligned. Hopefully, IGV should display the same information about the reads that we see in R!

Sometimes, we might want more than the standard fields from the bam file. In the following examples, we are also going to work with the mapping qualities and flags. We can also choose to use the names of each read as row identifers.

Use Case: Read the bam file again, this time importing the sequence reads, mapping qualities and flags. Also, import the read names. See how the resulting object differs from before.

```
bam <- readGAlignments(file=mybam,
                       param=ScanBamParam(what=c("seq","mapq","flag")),
                       use.names = TRUE)
bam
```

The extra information that we have read-in is part of the *'element metadata'* for the object. This metadata can be accessed using the `mcols` function, the result being a data-frame-like structure. As with standard data frames, particular columns can be extracted using the $ operator

Use Case: Find out what metadata is stored in the bam object. By referring to the columns of the metadata object, extract the mapping qualities and produce a histogram of these values.

```
mcols(bam)
hist(mcols(bam)$mapq)
```

The 'flag' field is also a useful indicator of read quality and the meaning of these flags is explained at this useful website

`https://broadinstitute.github.io/picard/explain-flags.html`

Use Case: Refer to the webpage that explains bam file flags and identify reads that have been flagged as PCR duplicates. Create an object containing only reads that are PCR duplicates, and further modify the bam object from the previous exercise to exclude PCR duplicates.

```
table(mcols(bam)$flag)
dupReads <- bam[mcols(bam)$flag==1024]
bamFilt <-bam[mcols(bam)$flag != 1024]
```

comment: In reality, this code is not sufficient to find all PCR duplicates, as it does not account for reads that have other flags set as well as the PCR duplicate flag. The code to remove all PCR duplicates would be more complex. Essentially we have to remove all flags in the range 1024 to 2048, and any flags over 3072

```
## Thanks to Tam Freestone-Bayes for the tip

 flag <- mcols(bam)$flag;
   a <- intersect(
     which(flag >= 1024),
     which(flag < 2048)
   )
   b <- which(flag >= 3072)
   dupIndices <- union(a, b)

   bamFilt2 <- bam[-dupIndices]
```

An alternative is provided by the *Rsamtools* package. There is a useful utility function `bamFlagAsBitMatrix` that can convert a vector of flags into a matrix where each column is a different flag that can be set. Each row of the matrix tells us which flags have been set for a particular read. Thus, we can read down the relevant column to identify the PCR duplicates.

```
flags <- mcols(bam)$flag
flagMat <- bamFlagAsBitMatrix(flags)
head(flagMat)
table(flagMat[,11])
bamFilt2 <- bam[-which(flagMat[,"isDuplicate"]==1)]
```

The mapping qualities reflect how unique and reliable a particular mapping to the genome is, and these values are produced by the particular aligner. If we want only reliable reads in our analysis, we can choose to keep only reads whose mapping quality exceeds a particular threshold.

Use Case: Produced a filtered object with only reads that have mapping quality greater than 30.

```
bamFilt <- bamFilt[mcols(bamFilt)$mapq > 30]
```

comment: We can in fact remove PCR duplicates when we read the bam file. Try the following section of code and verify that this object contains none of the PCR duplicate reads that you identified in the previous exercise

```
bam.nodups <- readGAlignments(file=mybam,
            param=ScanBamParam(flag=scanBamFlag(isDuplicate=FALSE)),
            use.names=TRUE)
names(dupReads) %in% names(bam.nodups)
```

comment: Dodgy reads may also be discovered by looking at the base frequency, or first few bases. Explore these data if you have time.

```
alignedSeqs <- mcols(bamFilt)$seq
seqFreq <- alphabetFrequency(alignedSeqs,baseOnly=TRUE)
subseq(alignedSeqs,1,10)
sort(table(subseq(alignedSeqs,1,10)),decreasing=TRUE)[1:5]
```

2.2 Finding 'peaks' in the aligned data

Several range operations will work on the `GappedAlignment` object, such as `coverage`.

Use Case: Create a coverage vector using `coverage` and see the output it produces. How can we get the coverage for chromosome 22 only? Why are there so many items in the output? *comment: Hint: This bam file was created by subsetting a bam file that was aligned to the whole-genome. Try using the `seqlevels` function on your aligned reads object*

```
cov <- coverage(bamFilt)
cov
names(cov)
cov[["22"]]
seqlevels(bamFilt)
```

The `slice` function, as it's name suggests, will return positions where the coverage exceeds a certain value.

Use Case: Identify regions with more than 400 reads covering and convert to an IRanges object.

```
slice(cov[["22"]],400)
HighCov <- ranges(slice(cov[["22"]], 400))
HighCov
```

For analysis, we might want to check that all the regions are the same length. We can do this using the `resize` function.

Use Case: Resize all the regions to a length of 100. Make sure that you have non-overlapping regions using `reduce`

```
HighCov <- resize(HighCov,100)
HighCov <- reduce(HighCov)
```

Now we are ready to find the reads that map within these peaks. An efficient way of creating a subset of reads based on genomic location is to use the overlapping functionality implemented in *GenomicAlignments*. The `findOverlaps` function will give indices for all ranges in the *query* and *subject* objects that overlap, which we can use to subset the objects. Alternatively ,the shortcut `%over%` is often useful.

Use Case: Restrict the alignments to just those in the high-coverage peaks. First, we will need to convert our peak regions to a GRanges object. How many reads are there inside these peaks in total?

```
HighCovGR <- GRanges("22", HighCov)
names(HighCovGR) <- paste("Region", 1:length(HighCov),sep="")
bam.sub <- bamFilt[bamFilt %over% HighCovGR]
bam.sub
length(bam.sub)
```

If we just wanted the number of overlapping reads, and not their details, we can use `countOverlaps`. The order is of arguments is important when we use this function.

Use Case: Use the `countOverlaps` to find out how many reads align to these high-coverage regions.

```
countOverlaps(HighCovGR,bamFilt)
sum(countOverlaps(HighCovGR,bamFilt))
```

comment: Why is `countOverlaps(bamFilt, HighCovGR)` not very informative in this example?

2.3 Reading a particular genomic region

If we were only really interested in a relatively small region of a chromosome, it is rather inefficient to read the whole bam file and then subset to the region we are interested in: especially if the file can run to several gigabytes in size. Fortunately, as the bam file is indexed, we can easily jump to the location we are interested in. This is achieved by using the `ScanBamParam` function with a GRanges object for the region we want to read.

Use Case: Read a portion of the bam file corresponding to the high-coverage regions rather the whole file. Verify that the same reads as the previous exercise are returned.

comment: You will need to carry out the same filtering on the file that you read-in. i.e. mapping quality over 30 and exclude PCR duplicates

```
bam.sub2 <- readGAlignments(file=mybam,use.names=TRUE,
param=ScanBamParam(which=HighCovGR,what=c("seq","mapq","flag")))
bam.sub2 <- bam.sub2[mcols(bam.sub2)$mapq >30 & mcols(bam.sub2)$flag != 1024]

length(bam.sub)
length(bam.sub2)
all(names(bam.sub)==names(bam.sub2))
```

3 Supplementary

3.1 Calculating Genome length

In this section, we will work through an example of how to calculate the length of the human genome, and express each chromosome as a percentage of the total length.

You will see that the code to calculate the length of a particular chromosome is virtually identical, apart from the variable name inside the [[]].

```
length(hg19[["chr22"]])
length(hg19[["chr21"]])
length(hg19[["chr20"]])
length(hg19[["chr19"]])
#etc.....
```

To save ourselves typing, and prevent errors, we often encapsulate tasks like this with a *for loop*. If you are unfamiliar with for loops, see the Appendix for a more-detailed discussion.

Use Case: What is the length of each chromosome? Calculate this using a for loop. You will need to 'loop' over the chromosome names and create a temporary variable to store the chromosome lengths as you calculate them.

```
chrLen <- NULL
cnames <- seqnames(hg19)

for(i in 1:length(cnames)){
  currentChr <- cnames[i]
  print(currentChr)
  chrLen[i] <- length(hg19[[currentChr]])
}
chrLen
names(chrLen) <- seqnames(hg19)
```

comment: R is very efficient at dealing with vectors, so in a lot of cases a for loop is not required. Often, we use a function that is part of the `apply` *family. This allows us to perform the same operation in fewer lines of code. An equivalent lapply statement to the above for loop is as follows:*

```
chrLen <- lapply(seqnames(hg19), function(x) length(hg19[[x]]))
chrLen
names(chrLen) <- seqnames(hg19)
```

To tidy-up a bit, we can discard unassembled, or alternate chromosomes. One way would be to only include chromosomes over a certain length.

Use Case: Select chromosomes that are over 5 million bases in length

```
chrLen <- chrLen[chrLen>5000000]
```

Use Case: Calculate the total length of the genome

```
totalLength <- sum(as.numeric(chrLen))
```

Use Case: Express each genome as a percentage of the total length. Visualise using a barplot.

```
barplot(100*(chrLen / totalLength))
```

We could also visualise the length as a cumulative distribution

```
barplot(cumsum(100*(chrLen / totalLength)))
```