Working with DNA strings and ranges: Exercises

Hervé Pagès*

9-10 December, 2010

Contents

*Fred Hutchinson Cancer Research Center, Seattle, WA 98008

1 Preliminaries

First let's make sure you have a working *R*-2.12 installation with all the packages needed for the course installed. To install any missing package (or to update the installed packages), start *R* and use the following command:

```
> source("http://bioconductor.org/course-packages/install-SeattleIntro2010.R")
```

2 Use case I: Extracting sequences from a reference genome

In this introductory use case, we learn how to extract DNA sequences from a *BSgenome data package* for a set of given locations. In particular we use the transcript, exon and CDS locations stored in a *TranscriptDb* object to extract the sequences of those features.

The *Bioconductor* data repositories provide a *BSgenome data package* for the sacCer2 genome (Yeast): the *BSgenome.Scerevisiae.UCSC.sacCer2* package. You should normally have it installed. It contains the full DNA sequences of the sacCer2 genome.

Also the *SeattleIntro2010* package contains a *TranscriptDb* object corresponding to this genome. Note that a *TranscriptDb* object contains positional (and relational) information about features but it does not contain sequences. If we need to extract the sequence of a given feature, an easy way is to query the *BSgenome.Scerevisiae.UCSC.sacCer2* package with the `getSeq` function.

Note that the result of this query is meaningful only if the *TranscriptDb* object contains positional annotations relative to the genome stored in the *BSgenome data package*. For example using a *TranscriptDb* object based on *BSgenome.Hsapiens.UCSC.hg18* to extract sequences from *BSgenome.Hsapiens.UCSC.hg19* would not make sense.

Exercise 1

- *Start R and load the SeattleIntro2010 package.*
- *Use* `system.file(package="SeattleIntro2010")` *to get the full path to the top-level folder of the installed package. (The top-level folder of an installed package should always be treated as read-only).*
- *Use* `list.files` *on the previous result.*
- *The TranscriptDb object that we are looking for is in the* `extdata` *subfolder. Use* `list.files(system.file("extdata", package="SeattleIntro2010"))` *to see it. It's the* `sacCer2_sgdGene.sqlite` *file.*
- *The* `sacCer2_sgdGene.sqlite` *file is an SQLite database that stores the transcript, exon and CDS locations relative to the sacCer2 genome as well as the relations between those features and their corresponding genes. This information was extracted from the "SGD Genes" track for sacCer2 at the*

UCSC Genome Browser, and formatted into an SQLite database that can be loaded in R with the `loadFeatures` *function from the GenomicFeatures package.*

- *Load the GenomicFeatures package and use the* `loadFeatures` *function to load the* `sacCer2_sgdGene.sqlite` *db into your session. Let's call* `txdb` *the returned object.*

`txdb` is a *TranscriptDb* object. You will learn more about those objects in the next session (in particular, how to make your own). For now, we're just going to extract all the exon locations from it and then query the *BSgenome.Scerevisiae.UCSC.sacCer2* with `getSeq` to extract their sequences.

Exercise 2

- *Extract all the exon locations from* `txdb` *with the* `exons` *function. The result is a GRanges object.*
- *How many exons? What GRanges accessor would you use to extract the lengths of those exons?*
- *What's the longest exon? Where is it located?*
- *Are there exons on the 2micron plasmid?*
- *Load the BSgenome.Scerevisiae.UCSC.sacCer2 package.*
- *There is only one symbol defined in this package, the* `Scerevisiae` *object (you can check this with* `ls("package:BSgenome.Scerevisiae.UCSC.sacCer2")`*). Display it.*
- *Get the lengths of the sequences with* `seqlengths(Scerevisiae)`*.*
- *Try to load any sequence with* `Scerevisiae[["some sequence name"]]`*.*
- *Have a quick look at the man page for the* `getSeq` *function (with* `?getSeq`*) and use it on* `Scerevisiae` *and the GRanges object created previously to extract the exons sequences. An important caveat here is to use* `as.character=FALSE`*.*

We end up with a *DNAStringSet* object containing our exon sequences. Later we will learn more about *DNAString* and *DNAStringSet* objects.

3 Use case II: Importing and manipulating a *GappedAlignments* object

An high-throughput sequencing experiment produces reads that need to be aligned against a reference genome. Although the *Bioconductor* software provides some fast pattern matching tools that can be used for aligning the reads,

this step is typically done with a third-party software like Bowtie, BWA, Eland, etc...

The *SeattleIntro2010* package contains reads from the Nagalakshmi et al. [1] experiment (Yeast RNA-seq). They have been aligned against the sacCer2 reference genome (using the BWA software) and stored in a BAM file. In order to keep the package to a reasonable size, only the reads from a single lane (oligo(dT)-primed, original) with hits on chromosome I to V have been kept. They are in the `SRR002051.chrI-V.bam` file.

Bioconductor provides several tools to load a BAM file:

- The low-level `scanBam` function from the *Rsamtools* package. Returns a list of lists.
- The middle-level `readGappedAlignments` function from the *GenomicRanges* package. Returns a *GappedAlignments* object.
- The high-level `readAligned` function from the *ShortRead* package. Returns an *AlignedRead* object.

Here we will focus on the middle-level solution. As we will see, *GappedAlignments* objects don't store as much information as *AlignedRead* objects (e.g. the read sequences, read qualities and alignment scores are not stored), but, unlike *AlignedRead* objects, they can store alignments with indels and gaps.

Exercise 3

- *Load the `SRR002051.chrI-V.bam` file (located in the `extdata` subfolder of the SeattleIntro2010 package) with the `readGappedAlignments` function from the GenomicRanges package. Let's call the result `galn`.*
- *Display `galn`. What's its class? What's its length?*

`galn` is a *GappedAlignments* object. According to the man page for those objects (`?GappedAlignments`):

> A GappedAlignments object is a vector-like object where each element describes an *alignment*, that is, how a given sequence (called "query" or "read", typically short) aligns to a reference sequence (typically long).

Alignments are also called "hits" or "matches". It's important to note that the relationship between the original set of queries to align (i.e. the input of the aligner software) and the elements of the *GappedAlignments* object is generally not one-to-one. Queries with no hits don't show up in the *GappedAlignments* object. Queries with multiple hits might have been part of the output of the aligner software and stored in the BAM file (by assigning more than one BAM record to the same query ID). In that case the resulting *GappedAlignments* object would have more than one element corresponding to the same query ID.

Like for *TranscriptDb*, the information stored in a *GappedAlignments* object is positional only (no sequence is stored), and positions are always 1-based with respect to the 5' end of the plus strand of the reference sequence.

The basic accessors for a *GappedAlignments* object are `rname`, `strand`, `cigar`, `qwidth`, `start`, `end`, `width` and `ngap`. They all return a vector of the same length as the *GappedAlignments* object itself.

Note that the "r" in `rname` and the "q" in `qwidth` stand for "reference" and "query", respectively, that is, the `rname` accessor gets (or sets) the name of the *reference* sequence for each alignment and the `qwidth` accessor gets the width of the *query* (i.e. its number of bases). Although they are not prefixed with "r", `start`, `end` and `width` are relative to the reference genome.

In the next exercise we look more closely at the difference between `qwidth` and `width`.

Exercise 4

- *Compare the output of* `table(qwidth(galn))` *and* `table(width(galn))`*. What we observe is that, even if all the queries are 33-base long, the portion of the reference sequence that they are aligned to can have a slightly different length. This is because those alignments contain indels (represented with I's or a D's in the CIGAR) or gaps (represented with N's in the CIGAR).*

- *Do the alignments have gaps?*

- *The GenomicRanges package provides some CIGAR utility functions documented in* `?\`cigar-utils\`` *(note the use of the backticks). In order to find the alignments that don't have a simple cigar (simple cigar means "only M's in it"), we're going to use the* `cigarOpTable` *function. Extract the CIGAR strings from the* `galn` *object and pass them to* `cigarOpTable`*.*

- *What we have now is a numeric matrix with one row per CIGAR string and one column per valid CIGAR letter* [1]*. Drop the M column and use* `rowSums` *to collapse the remaining columns into a single column.*

- *From here it should be easy to extract the index of alignments that don't have a simple cigar. Use this index to subset* `galn`*.*

One last thing before we move on to the next section.

Exercise 5

- *Turn* `galn` *into a GRanges object (let's call it* `hits`*). Tip: Consult the man page for GappedAlignments objects for how to do this.*

We will use this *GRanges* object in the next section.

[1] This set of CIGAR letters corresponds to the extended CIGAR specification as described in the SAMtools spec – see http://samtools.sourceforge.net/ for more information

4 Use case III: Looking at the reads that don't hit a (known) gene

In this section, we want to identify the reads that don't hit a (known) gene as well as the regions in the genomes covered by those reads. Because those reads are coming from an RNA-seq experiment, those regions would be good candidates for de-novo gene/transcript discovery.

Exercise 6

- *Load the* `sacCer2_sgdGene.sqlite` *file (located in the* `extdata` *subfolder of the SeattleIntro2010 package) with* `loadFeatures`*. Let's call* `txdb` *this TranscriptDb object.*
- *Use the* `countOverlaps` *function to find the elements in the* `hits` *object (from the previous section) that don't hit any gene. Note that there are basically 2 approaches to this: one based on the start/end of the transcripts and one based on the start/end of the exons. Let's call* `hits0` *the subset of* `hits` *made of those elements that don't hit any gene.*
- *Use the* `reduce` *function to extract the regions covered by* `hits0`*. Additionally you can perform a sanity check by making sure those regions don't intersect with the exons.*

We want to refine the way we've determined the above regions by using a method based on the *depth* of the coverage of the hits. More precisely we want to find the regions of the genome where the coverage of `hits0` is greater than (or equal to) some threshold (e.g. 10). An alternative way to formulate this is: we want to find the regions of the genome corresponding to all the bases that receive at least 10 hits.

Exercise 7

- *Compute the genome wide coverage of* `hits0`*. The result is a SimpleRleList object. This sounds complicated but it helps to think of it as a named list of Rle objects. The names of the list are the chromosomes and each top-level element in the list is an Rle object representing the coverage for the corresponding chromosome. Let's call this SimpleRleList object* `cov0`*. Like you would do with a standard* `list` *object, you can use the double-bracket operator (*`[[`*) to extract an element from* `cov0`*.*
- *Use the* `slice` *function to slice* `cov0` *horizontally. The result of this is a SimpleRleViewsList object. This sounds even more complicated but here again it helps to think of it as a named list of RleViews objects. The names of the list are still the chromosomes and each top-level element in the list now is an RleViews object i.e. a set of views on the coverage vector for the corresponding chromosome.*
- *Use the generic coercion function* `as` *to turn the result of the previous slicing into a* `GRanges` *object.*

There is an issue with the above method: we've lost the strand information. We might want to retain it if our final goal were to identify new transcripts. In the next exercise we improve our "pile up and slice" method to propagate the strand information stored in `hits0`.

Exercise 8

- *Split `hits0` in two GRanges object: one containing the hits located on the plus strand and one containing the hits located on the minus strand.*
- *Apply the "pile up and slice" method used in the previous exercise to each GRanges object. This transforms each GRanges object into another GRanges object.*
- *Note that the two GRanges objects obtained previously are unstranded. Add the strand information to them.*
- *Combine the two GRanges objects obtained previously with the generic function `c`.*
- *One last thing we might want to do is use `reduce` on the final result. The only effect of reducing here is to reorder the regions first by chromosome and then by strand (this is how `reduce` orders the elements of a GRanges object).*

Finally, to make it easier to repeat the transformation we've done to `hits0` in the previous exercise but now with different values of the threshold used in the slicing step, we want to wrap our code in a function.

Exercise 9

- *Write a function (`coveredRegions`) that takes a set of hits (in a GRanges object) and a threshold (`lower`) and returns a GRanges object containing the regions where the coverage of the hits is greater than (or equal to) the threshold. Make sure the returned GRanges object is stranded and reduced.*
- *Sanity check: compare `coveredRegions(hits0, 1)` with `reduce(hits0)`.*
- *Extract the DNA sequences corresponding to the regions returned by `coveredRegions(hits0, 10)`. Try with other threshold values.*

5 Use case IV: Measuring the complexity of the reads

A common task when dealing with HTS data is to filter out reads with a low complexity like poly-As or reads made of the repetition of the same 2-mer (dinucleotide) etc... In this section, we implement a simple function that takes a *DNAStringSet* object (the reads) and returns a score for each read based on

its complexity. The approach we use is inspired and adapted from the DUST algorithm.

The basic idea behind DUST is that a DNA sequence with a "poor trinucleotide content" (i.e. with a small number of *distinct* trinucleotides) is considered to have a low complexity. For example, the following sequences have a low complexity:

- AAAAAAAAAA: contains only 1 tri-nculeotide: AAA
- ATATATATATATATA: contains 2 tri-nucleotide: ATA and TAT

On the other hand, a DNA sequence with a "rich trinucleotide content" (i.e. many *distinct* trinucleotides) is considered to have a high complexity. For example, the following 36-mer:

- GGGCTACATGACGGTCCTGTATTTAGCCAGAGGATC

has the highest complexity a 36-mer can have because all the trinucleotides contained in it (34 in total) are distinct.

Here is how we will compute the score of a given DNA sequence:

- Count the number of occurences (frequency) of each possible trinucleotide: F_{AAA}, F_{AAC}, F_{AAG}, F_{AAT}, F_{ACA}, ..., F_{TTT} (64 in total).
- Substract 1 to the frequencies that are not zero. That is: for each F_{xxx}, if $F_{xxx} \geq 1$ then $F_{xxx} = F_{xxx} - 1$.
- $score = 1 - \frac{\sqrt{\sum F_{xxx}^2}}{L-3}$ where L is the length of the sequence.

A score of 0 indicates a poly-A, poly-C, poly-G or poly-T. A score of 1 can only be obtained by a sequence where all trinucleotides are distinct which implies that the sequence is no more than 66 bases long. Sequences longer than this cannot obtain a score of 1 and this scoring algorithm is not expected to give meaningful results on long sequences. Also, the score of very short sequences (i.e. 4 <= length <= 6) is not very meaningful either.

The *SeattleIntro2010* package contains reads from the Nagalakshmi et al. experiment stored in a FASTQ file (unaligned reads). Let's start by loading and "cleaning" them.

Exercise 10

- *Use the* `read.DNAStringSet` *function from the Biostrings package to load the* `SRR002051.reads1-50k.fastq` *file located in the* `extdata` *subfolder of the SeattleIntro2010 package. Note that, by default,* `read.DNAStringSet` *expects a FASTA file. Consult the man page for* `read.DNAStringSet` *to see how to read a FASTQ file.*
- *Use the* `alphabetFrequency` *function (with* `collapse=TRUE`*) on those reads.*

- *The scoring algorithm described above doesn't apply to reads that contain IUPAC ambiguity letters. Remove those reads. Tip: Try to use* `alphabetFrequency` *with* `baseOnly=TRUE`.

Let's call `cleanreads` the *DNAStringSet* object containing the result of the previous exercise. Now we are going to compute the score of those reads.

Exercise 11

- *Use the* `trinucleotideFrequency` *function from the Biostrings package to compute the trinucleotide frequencies of the clean reads. The result (let's call it* `tnf`*) is a matrix containing the trinucleotide counts for each input read. The matrix has 1 row per read and 64 columns (i.e. one column per each possible trinucleotide).*

- *Use the* `tnf` *matrix to compute the scores of the clean reads by following the steps described above. Tip 1: Note that, by performing vectorized arithmetic (a key feature of arithmetic operations in R), we can avoid the use of loops and be very fast. Tip 2: You can use* `rowSums` *on a numeric matrix to sum all the coefficients that belong to the same row, and this for all the rows.*

- *Wrap the previous code in a function i.e. implement the* `complexityOfReads` *function that computes the complexity scores of each element of a DNAStringSet object.*

- *Use this function to display the clean reads with a score of 0, then the reads with a score of 1.*

- *Plot the histogram of the complexities of the reads in* `cleanreads`.

6 Use case V: Pattern matching

In this last use case, we're going to try to align our reads with the pattern matching tools available in the *Biostrings* package.

We can divide those tools in 2 categories:

- General purpose pattern matching/aligning tools:
  - `matchPattern` (and convenience wrapper `countPattern`);
  - `vmatchPattern` (and convenience wrapper `vcountPattern`);
  - `pairwiseAlignment`.
- Specialized pattern matching/finding tools:
  - `matchPDict` (and convenience wrappers `countPDict`, `whichPDict`);
  - `vmatchPDict` (and convenience wrappers `vcountPDict`, `vwhichPDict`);
  - `matchLRPatterns`;

- `matchProbePair`;
- `trimLRPatterns`;
- `matchPWM`;
- `findPalindromes`.

The `matchPDict` function (and family) is the tool of choice when it comes to matching a high number of short DNA sequences (the patterns) against a long reference DNA sequence (the subject). The `countPDict` and `whichPDict` functions are convenience wrappers for situations where counting the hits only is desired (`countPDict`), or when we just want to know which patterns have at least 1 hit (`whichPDict`). A common feature of all the `*PDict` matching functions is that, in order to be efficient, they all require to preprocess the set of patterns (aka "the pattern dictionary"). This is achieved with the `PDict` constructor:

```
> system.time(nhits <- countPDict(cleanreads, Scerevisiae$chrI))

   user  system elapsed
  17.96    0.00   18.03

> system.time(nhits2 <- countPDict(PDict(cleanreads), Scerevisiae$chrI))

   user  system elapsed
   1.25    0.00    1.25

> identical(nhits, nhits2)

[1] TRUE

> table(nhits)

nhits
    0     1     2     4
48648   164    23    45
```

Note that, by default, `matchPDict` and family look for exact matches only. See `?PDict` for how to allow a small numbers of mismatches (at the price of a significant slowdown though).

Let's start by illustrating the basic usage of `PDict`/`countPDict`.

Exercise 12

- *Preprocess the `cleanreads` dictionary with `PDict`.*
- *Use `countPDict` to count the number of hits per pattern against sacCer2 chromosome IV (the longest chromosome in Yeast).*

Note that, without taking any special precaution, the hit counts we got are for hits on the plus strand of chrIV only. However, a simple and efficient way to count the hits on the minus strand is to repeat the 2 steps above on the reverse complement of `cleanreads`.

Exercise 13

- *Use the* `reverseComplement` *function on* `cleanreads`*.*
- *Count the hits on the minus strand of chrIV.*
- *Finally add the 2 vectors of hit counts you got.*

Note that one way to make this faster (at the cost of more memory usage) is to preprocess the patterns and their reverse complements in the same *PDict* object, but this goes beyond the scope of this simple introduction to fast pattern matching with *Biostrings*.

In the next exercise, we want to match our clean reads against the *full* sacCer2 genome, by searching the plus and minus strands of each reference sequence. Again, we will do exact matching only.

Exercise 14

- *Write a function* `countSingleStrandGenomeHits`*, that takes 2 arguments: (1) a pattern dictionary (DNAStringSet) and, (2) a BSgenome object (e.g.* `Scerevisiae`*). The function will return the number of hits on the entire genome for each pattern in the input dictionary. The function should only count the hits on the plus strand.*
  - *Tip 1: Use* `seqnames` *on a BSgenome object to get its sequence names.*
  - *Tip 2: Loop over the sequence names with a* `for` *loop.*
  - *Tip 3: Before you start the loop, create an integer vector of the length of the input dictionary with* `nhits <- integer(length(dict))`*. It will be initialized with zeroes. Then, inside the loop, add the results for each sequence to* `nhits`*.*
- *Use* `countSingleStrandGenomeHits` *to count the hits of our clean reads against the full sacCer2 genome. The final result (let's call it* `gwnhits` *for "genome wide nhits") must take both strands into account.*
- *What's the percentage of reads that don't hit the genome? How could this be improved?*

7 Solutions and session information

7.1 Exercise 1

```
library(SeattleIntro2010)
filepath <- system.file("extdata", "sacCer2_sgdGene.sqlite",
                        package="SeattleIntro2010")
library(GenomicFeatures)
txdb <- loadFeatures(filepath)
```

7.2 Exercise 2

```
ex <- exons(txdb)
length(ex)  # number of exons
width(ex)  # lengths of exons
max(width(ex))  # length of longest exon
ex[which.max(width(ex))]  # longest exon
ex[seqnames(ex) == "2micron"]  # exons on the 2micron plasmid
library(BSgenome.Scerevisiae.UCSC.sacCer2)
Scerevisiae  # displays Scerevisiae
seqlengths(Scerevisiae)
Scerevisiae[["chrI"]]
getSeq(Scerevisiae, ex, as.character=FALSE)
```

7.3 Exercise 3

```
filepath <- system.file("extdata", "SRR002051.chrI-V.bam",
                        package="SeattleIntro2010")
galn <- readGappedAlignments(filepath)
class(galn)
length(galn)
```

7.4 Exercise 4

```
table(qwidth(galn))  # one single value (33)
table(width(galn))  # most are 33 but not all!
any(ngap(galn) != 0)  # no gaps
cigtable <- cigarOpTable(cigar(galn))
dim(cigtable)  # big matrix! (many rows)
colnames(cigtable)  # M is the first column
idx <- which(rowSums(cigtable[ , -1]) != 0)  # final index
galn[idx]
```

7.5 Exercise 5

```
hits <- grg(galn)
```

7.6 Exercise 6

```
hits0 <- hits[countOverlaps(hits, exons(txdb)) == 0]
reduce(hits0)
intersect(reduce(hits0), exons(txdb))  # sanity check
```

7.7 Exercise 7

```
cov0 <- coverage(hits0)  # SimpleRleList object
cov0[["chrV"]]  # Rle object
```

```
slice(cov0, lower=200)  # SimpleRleViewsList object
as(slice(cov0, lower=200), "GRanges")
```

7.8 Exercise 8

```
plushits0 <- hits0[strand(hits0) == "+"]
minushits0 <- hits0[strand(hits0) == "-"]
plusregions0 <- as(slice(coverage(plushits0), lower=200), "GRanges")
minusregions0 <- as(slice(coverage(minushits0), lower=200), "GRanges")
strand(plusregions0) <- "+"
strand(minusregions0) <- "-"
c(plusregions0, minusregions0)
reduce(c(plusregions0, minusregions0))
```

7.9 Exercise 9

```
coveredRegions <- function(hits, lower)
{
  plushits <- hits[strand(hits) == "+"]
  minushits <- hits0[strand(hits) == "-"]
  plusregions <- as(slice(coverage(plushits), lower=lower), "GRanges")
  minusregions <- as(slice(coverage(minushits), lower=lower), "GRanges")
  strand(plusregions) <- "+"
  strand(minusregions) <- "-"
  reduce(c(plusregions, minusregions))
}
identical(coveredRegions(hits0, 1), reduce(hits0))  # TRUE
getSeq(Scerevisiae, coveredRegions(hits0, 10), as.character=FALSE)
getSeq(Scerevisiae, coveredRegions(hits0, 1000), as.character=FALSE)
```

7.10 Exercise 10

```
filepath <- system.file("extdata", "SRR002051.reads1-50k.fastq",
                        package="SeattleIntro2010")
reads <- read.DNAStringSet(filepath, format="fastq")
alphabetFrequency(reads, collapse=TRUE)
cleanreads <- reads[alphabetFrequency(reads, baseOnly=TRUE)[ , "other"] == 0]
```

7.11 Exercise 11

```
tnf <- trinucleotideFrequency(cleanreads)

tnf <- tnf - 1
tnf[tnf < 0] <- 0
scores <- 1 - sqrt(rowSums(tnf^2)) / (width(cleanreads) - 3)
```

```
complexityOfReads <- function(reads)
{
    tnf <- trinucleotideFrequency(reads)
    tnf <- tnf - 1
    tnf[tnf < 0] <- 0
    1 - sqrt(rowSums(tnf^2)) / (width(reads) - 3)
}

cleanreads[complexityOfReads(cleanreads) == 0]
cleanreads[complexityOfReads(cleanreads) == 1]

plot(hist(complexityOfReads(cleanreads), breaks=50))
```

7.12 Exercise 12

```
pdict <- PDict(cleanreads)
library(BSgenome.Scerevisiae.UCSC.sacCer2)
nhits <- countPDict(pdict, Scerevisiae[["chrIV"]])
```

7.13 Exercise 13

```
rc_cleanreads <- reverseComplement(cleanreads)
rc_pdict <- PDict(rc_cleanreads)
rc_nhits <- countPDict(rc_pdict, Scerevisiae[["chrIV"]])
nhits + rc_nhits
```

7.14 Exercise 14

```
countSingleStrandGenomeHits <- function(dict, genome)
{
  pdict <- PDict(dict)
  nhits <- integer(length(dict))
  for (seqname in seqnames(genome)) {
    nhits <- nhits + countPDict(pdict, genome[[seqname]])
  }
  nhits
}
plus_nhits <- countSingleStrandGenomeHits(cleanreads, Scerevisiae)
minus_nhits <- countSingleStrandGenomeHits(rc_cleanreads, Scerevisiae)
gwnhits <- plus_nhits + minus_nhits
sum(gwnhits == 0) / length(gwnhits)  # 80% of the reads have no hit!
```

7.15 Session information

- R version 2.12.0 Patched (2010-11-28 r53696), `i386-pc-mingw32`

- Locale: `LC_COLLATE=C`, `LC_CTYPE=English_United States.1252`, `LC_MONETARY=English_United States.1252`, `LC_NUMERIC=C`, `LC_TIME=English_United States.1252`
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: ALL 1.4.7, AnnotationDbi 1.12.0, BSgenome 1.18.2, BSgenome.Scerevisiae.UCSC.sacCer2 1.3.16, Biobase 2.10.0, Biostrings 2.18.2, DBI 0.2-5, GO.db 2.4.5, GenomicFeatures 1.2.3, GenomicRanges 1.2.2, IRanges 1.8.7, RSQLite 0.9-4, SeattleIntro2010 0.0.41, biomaRt 2.6.0, genefilter 1.32.0, hgu95av2.db 2.4.5, org.Hs.eg.db 2.4.6
- Loaded via a namespace (and not attached): RCurl 1.5-0.1, XML 3.2-0.2, annotate 1.28.0, rtracklayer 1.10.6, splines 2.12.0, survival 2.36-2, tools 2.12.0, xtable 1.5-6

References

[1] U. Nagalakshmi, Z. Wang, K. Waern, C. Shou, D. Raha, M. Gerstein, and M. Snyder. The transcriptional landscape of the yeast genome defined by RNA sequencing. *Science*, 320:1344–1349, Jun 2008.