Working with DNA strings and ranges: Exercises

Hervé Pagès*

9-10 December, 2010

Contents

1 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.

*Fred Hutchinson Cancer Research Center, Seattle, WA 98008

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* `sacCer2_sgdGene.sqlite` *with* `loadFeatures`*. 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.*
- *What's the longest exon? 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.*
- *Try to load any sequence with* `Scerevisiae[["some sequence name"]]`*.*
- *Use the* `getSeq` *function on* `Scerevisiae` *and the GRanges object created previously to extract the exons sequences.*

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

2 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.*
- *Do the alignments have gaps?*
- *Use* `cigarOpTable` *to find the alignments that don't have a simple cigar (simple cigar means "only M's in it").*
- *Turn this object (*`galn`*) into a GRanges object (*`reads`*).*

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

3 Use case III: 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 4

- *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* `reads` *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* `reads0` *the subset of* `reads` *made of those elements that don't hit any gene.*
- *Use the* `reduce` *function to extract the regions covered by* `reads0`*.*

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 reads. More precisely we want to find the regions of the genome where the coverage of `reads0` 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 5

- *Compute the genome wide coverage of* `reads0`*. 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`*.*
- *Use the* `slice` *function to slice* `cov0` *horizontally.*
- *Use the generic 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 `reads0`.

Exercise 6

- *Split* `reads0` *in two GRanges object: one containing the reads located on the plus strand and one containing the reads 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 `reads0` in the previous exercise but now with different values of the threshold used for the slicing step, we want to wrap our code in a function.

Exercise 7

- *Write a function (*`coveredRegions`*) that takes a set of reads (in a GRanges object) and a threshold (*`lower`*) and returns a GRanges object containing the regions where the coverage of the reads is greater than (or equal to) the threshold. Make sure the returned GRanges object is stranded and reduced.*
- *Sanity check: compare* `coveredRegions(reads0, 1)` *with* `reduce(reads0)`.
- *Extract the DNA sequences corresponding to the regions returned by* `coveredRegions(reads0, 10)`. *Try with other threshold values.*

4 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 $F_{xxx} != 0$ 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 them.

Exercise 8

- *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* `trinucleotideFrequency` *function from the Biostrings package to compute the trinucleotide frequencies of all the 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).*
- *Compute the score of all the reads by following the steps described above. By using the vectorization capabilities of the arithmetic operations in R, we can avoid the use of loops for this and we can be very fast. 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.*

5 Use case V: Pattern matching

Exercise 9

-

6 Session information

- R version 2.12.1 beta (2010-12-07 r53813), `i386-apple-darwin9.8.0`
- Locale: `C`
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: ALL 1.4.7, AnnotationDbi 1.12.0, Biobase 2.10.0, DBI 0.2-5, GO.db 2.4.5, RSQLite 0.9-4, SeattleIntro2010 0.0.33, 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 1.4-3, XML 3.2-0, annotate 1.28.0, splines 2.12.1, survival 2.36-2, tools 2.12.1, 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.