Practical: Range-based containers

Hervé Pagès (hpages@fhcrc.org)

3 February 2014

Contents

1 An introduction to genomic ranges

A *genomic range* is a region on a genome that is made of contiguous nucleotides (i.e. no gaps). It is defined by a chromosome name (a.k.a. sequence name), a start and an end coordinate, and possibly a strand. Genomic ranges are used to locate various things on a reference genome such as genomic features (genes, transcripts, exons, repeat regions, etc...), aligned reads, or regions of interest (e.g. peaks detected in a ChIP-seq analysis).

In *Bioconductor*, the *GRanges* class, defined and documented in the *GenomicRanges* package, is used to represent a vector of genomic ranges.

1.1 Mandatory components of a *GRanges* object

A *GRanges* object contains the following mandatory components:

- A vector of sequence names, accessed with the `seqnames()` accessor.
- A vector of start coordinates, accessed with the `start()` accessor.
- A vector of end coordinates, accessed with the `end()` accessor.
- A vector of strand values (`+`, `-`, or `*`), accessed with the `strand()` accessor.

The vectors returned by the `seqnames()`, `start()`, `end()`, and `strand()` accessors have the same length, which is the number of genomic ranges in the object. By definition, this is also considered to be the length of the object and it can be obtained with `length()` (like with an ordinary vector).

Here is an example of a *GRanges* object of length 7 (i.e. with 7 genomic ranges in it):

```
gr1
```

```
## GRanges with 7 ranges and 0 metadata columns:
##       seqnames           ranges strand
##          <Rle>        <IRanges>  <Rle>
##   [1]      chr1 [ 10001,  10400]      -
##   [2]      chr1 [  5508,   5792]      *
##   [3]     chr19 [ 25644,  27263]      +
##   [4]      chrX [   685,   1017]      +
##   [5]      chrX [142501, 142501]      -
##   [6]      chrM [ 33077,  35076]      *
##   [7]     chr19 [  8001,   8421]      +
##   ---
##   seqlengths:
##     chr1 chr19  chrM  chrX
##       NA    NA    NA    NA
```

Here is another more realistic *GRanges* object representing a tiling of the full Yeast genome (with tiles of size 50000 nucleotides):

```
tiles
```

```
## GRanges with 251 ranges and 0 metadata columns:
##         seqnames           ranges strand
##            <Rle>        <IRanges>  <Rle>
##     [1]     chrI [     1,  50000]      *
##     [2]     chrI [ 50001, 100000]      *
##     [3]     chrI [100001, 150000]      *
##     [4]     chrI [150001, 200000]      *
##     [5]     chrI [200001, 230208]      *
##     ...      ...              ...    ...
##   [247]   chrXVI [850001, 900000]      *
##   [248]   chrXVI [900001, 948062]      *
##   [249]     chrM [     1,  50000]      *
##   [250]     chrM [ 50001,  85779]      *
##   [251]  2micron [     1,   6318]      *
##   ---
##   seqlengths:
##       chrI   chrII  chrIII   chrIV    chrV ...  chrXIV    chrXV  chrXVI     chrM 2micron
##     230208  813178  316617 1531919  576869 ...  784333 1091289  948062   85779    6318
```

An important convention to keep in mind about the start and end coordinates of a genomic range is that they are both *1-based* positions relative to the 5' end of the plus strand of the reference sequence, *even when the range is on the minus strand*. Also the nucleotides located at the start and end positions are both considered to be part of the range (the former being the left-most nucleotide of the range and the latter its right-most nucleotide).

1.2 Metadata columns of a *GRanges* object

In addition to the mandatory components described above, one or more *metadata columns* can be stored in a *GRanges* object in the form of a *DataFrame* that can be accessed with the `mcol()` accessor. The *DataFrame* has one row per genomic range and its columns are considered to be the metadata columns of the *GRanges* object.

Here is an example of a *GRanges* object with 2 metadata columns:

```
gr2
```

```
## GRanges with 7 ranges and 2 metadata columns:
##       seqnames           ranges strand |          desc GC_content
##          <Rle>        <IRanges>  <Rle> |   <character>  <numeric>
##   [1]     chr1 [ 10001,  10400]      - |          exon      0.550
##   [2]     chr1 [  5508,   5792]      * | repeat region      0.495
##   [3]    chr19 [ 25644,  27263]      + |          gene      0.562
##   [4]     chrX [   685,   1017]      + |          exon      0.530
##   [5]     chrX [142501, 142501]      - |          exon      0.552
##   [6]     chrM [ 33077,  35076]      * |           snp       <NA>
##   [7]    chr19 [  8001,   8421]      + | repeat region      0.510
##   ---
##   seqlengths:
##     chr1 chr19  chrM  chrX
##       NA    NA    NA    NA
```

Another example of a *GRanges* object with metadata columns is shown below. This object represents all the known protein coding regions (CDS) on the Human genome (positions are reported with respect to reference assembly hg19):

```
hg19_cds
```

```
## GRanges with 237533 ranges and 1 metadata column:
##            seqnames               ranges strand   |            gene_id
##               <Rle>            <IRanges>  <Rle>   |    <CharacterList>
##        [1]     chr1     [ 12190,  12227]      +   |          100287102
##        [2]     chr1     [ 12595,  12721]      +   |          100287102
##        [3]     chr1     [ 13403,  13639]      +   |          100287102
##        [4]     chr1     [ 69091,  70008]      +   |              79501
##        [5]     chr1     [324343, 324345]      +   | 100132062,100133331
##        ...      ...                  ...    ... ...                ...
##   [237529]     chrY [26959330, 26959332]      -   |        57054,57135
##   [237530]     chrY [27184245, 27184263]      -   |               9083
##   [237531]     chrY [27184956, 27185061]      -   |               9083
##   [237532]     chrY [27187916, 27188033]      -   |               9083
##   [237533]     chrY [27190093, 27190170]      -   |               9083
##   ---
##   seqlengths:
##                    chr1                 chr2 ...         chrUn_gl000249
##               249250621            243199373 ...                  38502
```

2 Working with *GRanges* objects

2.1 Basic manipulation

Exercise 1 *This exercise illustrates basic manipulation of GRanges objects.*

a. Load the package where the GRanges class is defined and documented.
b. Open the man page for the GRanges class and run the examples in it.
c. Display the `gr` object.
d. Use the `seqnames()` accessor to get the vector of sequence names. What is the class of this vector? Use `table()` on it to count the number of genomic ranges on each reference sequence.
e. In addition to the start and end coordinates that can be obtained with the `start()` and `end()` accessors, the width *of the ranges can be obtained with the `width()` accessor. By definition, the* width *of a genomic range*

is the number of nucleotides in it. For any range, we have $width = end - start + 1$*.*
Obtain the width of the ranges in `gr`*.*
How do you check that $width$ *is indeed the same as* $end - start + 1$ *programmatically?*

Solution:

a. The *GRanges* class is defined and documented in the *GenomicRanges* package.

```
library(GenomicRanges)
```

b. Use the `class?<classname>` or ``?`<classname>-class` `` syntax to open the man page for a given class:

```
class?GRanges
```

Since there is also a constructor function for *GRanges* objects (the `GRanges()` function) documented in the same man page, the man page can also be accessed with `?GRanges`.
To run the examples in a man page, you can copy/paste them into your current session or call the `example()` function on the same alias you used to open the man page (i.e. on the part following the `?`):

```
example(GRanges)
```

c. The code in the examples calls the `GRanges()` constructor function to create the `gr` object:

```
gr
## GRanges with 10 ranges and 2 metadata columns:
##     seqnames    ranges strand |     score                GC
##        <Rle> <IRanges>  <Rle> | <integer>         <numeric>
##   a     chr1  [ 1, 10]      - |         1                 1
##   b     chr2  [ 2, 10]      + |         2 0.888888888888889
##   c     chr2  [ 3, 10]      + |         3 0.777777777777778
##   d     chr2  [ 4, 10]      * |         4 0.666666666666667
##   e     chr1  [ 5, 10]      * |         5 0.555555555555556
##   f     chr1  [ 6, 10]      + |         6 0.444444444444444
##   g     chr3  [ 7, 10]      + |         7 0.333333333333333
##   h     chr3  [ 8, 10]      + |         8 0.222222222222222
##   i     chr3  [ 9, 10]      - |         9 0.111111111111111
##   j     chr3  [10, 10]      - |        10                 0
##   ---
##   seqlengths:
##    chr1 chr2 chr3
##    1000 2000 1500
```

d. The vector of sequence names can be obtained with the `seqnames()` accessor:

```
seqnames(gr)
## factor-Rle of length 10 with 4 runs
##   Lengths:    1    3    2    4
##   Values : chr1 chr2 chr1 chr3
## Levels(3): chr1 chr2 chr3
```

This is an *Rle* (Run Length Encoding) object:

```
class(seqnames(gr))
## [1] "Rle"
## attr(,"package")
## [1] "IRanges"
```

Using an *Rle* representation of a vector or factor is a simple and efficient way to reduce its size in memory (a.k.a. its *memory footprint*) when it has long *runs* of with the same value.

e. The *width* of the ranges in `gr`:

```
width(gr)
##  [1] 10  9  8  7  6  5  4  3  2  1
```

We can check that $width = end - start + 1$ with:

```
all(width(gr) == end(gr) - start(gr) + 1)
## [1] TRUE
```

Exercise 2 *This exercise illustrates more basic manipulation of GRanges objects.*

a. *Like ordinary vectors in R, GRanges objects can have names on them (one per vector element in the object). The mechanism for getting or setting the names is with the `names` getter and setter.*
   Obtain the names of `gr`.
   Replace the current names with upper case names (same names but upper case).
b. *Like ordinary vectors in R, GRanges objects can be subsetted with the `[` operator.*
   Subset `gr` to keep only the ranges located on the + strand.
c. *Obtain the metadata columns in `gr`.*
   Obtain the `GC` column.
   Subset `gr` to keep only the ranges for which the `GC` value is ¿= 0.3 and ¡= 0.6.
d. *Finally, add a new metadata column to `gr`.*

Solution:

a. We get the names with:

```
names(gr)
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

b. We use the `toupper()` function to translate the names from lower to upper case, and then we set the new names back on `gr` with the `names()` *setter* (i.e. we call `names()` on the *left side* of the assignment):

```
names(gr) <- toupper(names(gr))
```

c. To obtain all the metadata columns:

```
mcols(gr)
## DataFrame with 10 rows and 2 columns
##         score        GC
##     <integer> <numeric>
## 1           1    1.0000
## 2           2    0.8889
## 3           3    0.7778
## 4           4    0.6667
## 5           5    0.5556
## 6           6    0.4444
## 7           7    0.3333
## 8           8    0.2222
## 9           9    0.1111
## 10         10    0.0000
```

To obtain the `GC` column:

```
mcols(gr)$GC
##  [1] 1.0000 0.8889 0.7778 0.6667 0.5556 0.4444 0.3333 0.2222 0.1111 0.0000
```

To keep only the ranges for which GC is ¿= 0.3 and ¡= 0.6:

```
gr[mcols(gr)$GC >= 0.3 & mcols(gr)$GC <= 0.6]
## GRanges with 3 ranges and 2 metadata columns:
##     seqnames    ranges strand |     score                GC
##        <Rle> <IRanges>  <Rle> | <integer>         <numeric>
##   E     chr1   [5, 10]      * |         5 0.5555555555555556
##   F     chr1   [6, 10]      + |         6 0.4444444444444444
##   G     chr3   [7, 10]      + |         7 0.333333333333333
##   ---
##   seqlengths:
##    chr1 chr2 chr3
##    1000 2000 1500
```

Note that this subsetting can be performed more conveniently with `subset()`:

```
subset(gr, GC >= 0.3 & GC <= 0.6)
## GRanges with 3 ranges and 2 metadata columns:
##     seqnames    ranges strand |     score                GC
##        <Rle> <IRanges>  <Rle> | <integer>         <numeric>
##   E     chr1   [5, 10]      * |         5 0.555555555555556
##   F     chr1   [6, 10]      + |         6 0.444444444444444
##   G     chr3   [7, 10]      + |         7 0.333333333333333
##   ---
##   seqlengths:
##    chr1 chr2 chr3
##    1000 2000 1500
```

d. To set a metadata column on `gr`, we use the same form as for getting a metadata column but this time on the *left side* of the assignment:

```
mcols(gr)$id <- sprintf("ID%03d", seq_along(gr))
gr
## GRanges with 10 ranges and 3 metadata columns:
##     seqnames    ranges strand |     score        GC          id
##        <Rle> <IRanges>  <Rle> | <integer> <numeric> <character>
##   A     chr1  [ 1, 10]      - |         1    1.0000       ID001
##   B     chr2  [ 2, 10]      + |         2    0.8889       ID002
##   C     chr2  [ 3, 10]      + |         3    0.7778       ID003
##   D     chr2  [ 4, 10]      * |         4    0.6667       ID004
##   E     chr1  [ 5, 10]      * |         5    0.5556       ID005
##   F     chr1  [ 6, 10]      + |         6    0.4444       ID006
##   G     chr3  [ 7, 10]      + |         7    0.3333       ID007
##   H     chr3  [ 8, 10]      + |         8    0.2222       ID008
##   I     chr3  [ 9, 10]      - |         9    0.1111       ID009
##   J     chr3  [10, 10]      - |        10    0.0000       ID010
##   ---
##   seqlengths:
##    chr1 chr2 chr3
##    1000 2000 1500
```

2.2 Common range transformations

The ranges in a *GRanges* object can be transformed in different ways. Some of the most common range transformations are shown on figure˜1 below.

Exercise 3 *In this exercise we perform some range transformations on a GRanges object.*

a. *Use `shift` to shift the ranges in `gr` by 500 positions to the right. Store the result in `gr2`.*
b. *Find the man page for the `flank` method that operates on GRanges objects. Obtain the upstream flanking regions of width 100 for `gr2`, that is, the regions corresponding to the 100 nucleotides located upstream of each range in `gr2` when moving in the 5' to 3' direction.*

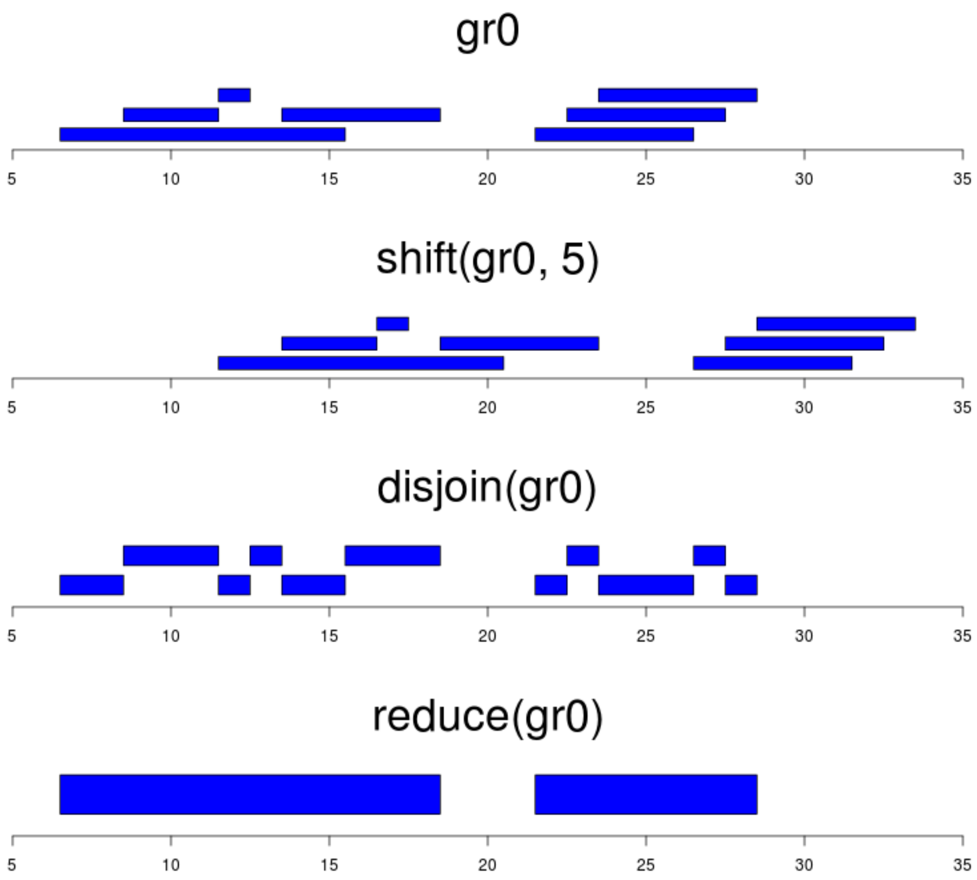

Figure 1: Range transformations `shift()`, `disjoin()`, and `reduce()`. For clarity, `gr0` is a *GRanges* object with all its ranges on the same reference sequence and strand.

c. *"Display" the `shift()` function. What kind of function is that? Which arguments are involved in the dispatch mechanism?*

d. *Use `showMethods` to list all the `shift` methods. Which method is used when we call `shift()` on a GRanges object? Try to open the man page for* this *method using the `?<something-here>` syntax.*

Solution:

a.
```
gr2 <- shift(gr, 500)
```

b.
```
flank(gr2, width=100)
## GRanges with 10 ranges and 3 metadata columns:
##     seqnames     ranges strand |     score        GC          id
##        <Rle>  <IRanges>  <Rle> | <integer> <numeric> <character>
##   A     chr1 [511, 610]      - |         1    1.0000       ID001
##   B     chr2 [402, 501]      + |         2    0.8889       ID002
##   C     chr2 [403, 502]      + |         3    0.7778       ID003
##   D     chr2 [404, 503]      * |         4    0.6667       ID004
##   E     chr1 [405, 504]      * |         5    0.5556       ID005
##   F     chr1 [406, 505]      + |         6    0.4444       ID006
##   G     chr3 [407, 506]      + |         7    0.3333       ID007
##   H     chr3 [408, 507]      + |         8    0.2222       ID008
##   I     chr3 [511, 610]      - |         9    0.1111       ID009
##   J     chr3 [511, 610]      - |        10    0.0000       ID010
##   ---
##   seqlengths:
##    chr1 chr2 chr3
##    1000 2000 1500
```

c. To “display” the `shift()` function, we just type the name of the function followed by ¡Enter¿, like with any other object in *R*. For an ordinary function, this displays the *body* of the function (i.e. its implementation). However, for `shift()`, we get something different:

```
shift
## standardGeneric for "shift" defined from package "IRanges"
##
## function (x, shift = 0L, use.names = TRUE)
## standardGeneric("shift")
## <environment: 0x4457510>
## Methods may be defined for arguments: x
## Use  showMethods("shift")  for currently available ones.
```

This is because `shift()` is not an ordinary function but an *S4 generic function*. A generic function doesn’t do the work: instead it calls the appropriate method for doing it. The selection of the appropriate method is done internally by the *dispatch mechanism*. The dispatch mechanism looks at the types of the arguments that are passed to the generic function and selects the most specific method compatible with these arguments. In the case of `shift()`, only 1 argument is considered for dispatch (the first argument `x`).

d. To list all the `shift` methods:

```
showMethods(shift)
## Function: shift (package IRanges)
## x="CompressedIRangesList"
## x="GRanges"
##     (inherited from: x="GenomicRanges")
## x="GRangesList"
## x="GenomicRanges"
## x="IRanges"
##     (inherited from: x="Ranges")
## x="IntervalForest"
## x="Ranges"
## x="RangesList"
## x="SummarizedExperiment"
## x="Views"
```

To see the method that is used when `x` is a *GRanges* object:

```
selectMethod(shift, "GRanges")
## Method Definition:
##
## function (x, shift = 0L, use.names = TRUE)
## {
##     ranges <- shift(ranges(x), shift, use.names = use.names)
##     clone(x, ranges = ranges)
## }
## <environment: namespace:GenomicRanges>
##
## Signatures:
##         x
## target  "GRanges"
## defined "GenomicRanges"
```

Pay attention to the last 2 lines. One complication here is that the method that is used when `x` is a *GRanges* object is not defined specifically for the *GRanges* class but for the *GenomicRanges* class which is a parent class of the *GRanges* class. We can check that `gr` is indeed a *GenomicRanges* object, even though it's not a *GenomicRanges instance*:

```
is(gr, "GenomicRanges")
## [1] TRUE
class(gr)
## [1] "GRanges"
## attr(,"package")
## [1] "GenomicRanges"
```

To open the man page for *this* method:

```
?`shift,GenomicRanges-method`
```

3 Working with *GRangesList* objects

So far we've seen *GRanges* objects and how to work on them. One limitation of the *GRanges* container is that it can only be used for storing genomic regions that are made of contiguous nucleotides (i.e. no gaps). However, some types of genomic regions (or genomic features) cannot always be represented with a single genomic range. For example, some aligners used on RNA-seq data can produce aligned reads (called *junction reads* or *gapped reads*) that need to be represented with 2 or more genomic ranges. In that case a *GRangesList* object needs to be used to store the ranges.

More formally speaking, a *GRangesList* object also contains genomic ranges but now they are divided into groups. Each group consists of one or more genomic ranges that represent a complex region (e.g. all the exons of a given gene, a junction read, etc...)

To keep things easy and intuitive, the *GRangesList* container has been designed so that it can be manipulated like an ordinary list for most usual list operations. For example, each individual group can be extracted with `[[`, and `length()` returns the total number of groups in the object.

Exercise 4 *In this exercise, we use a GRangesList object to store genomic features (e.g. exons) grouped by parent feature (e.g. transcript or gene). The input data we use for this is a* gene model.

a. *Load the TxDb.Hsapiens.UCSC.hg19.knownGene package. This package contains a* gene model *for Human i.e. the genomic ranges of all the known Human transcripts, exons, and CDS. The gene model can be accessed via the `TxDb.Hsapiens.UCSC.hg19.knownGene` object which is basically a pointer to an SQLite database containing the gene model. Display this object.*
b. *Use the `exonsBy()` function on the `TxDb.Hsapiens.UCSC.hg19.knownGene` object to extract the exons grouped by gene (check the `exonsBy` man page for how to do that). What kind of object is returned?*
c. *How many genes are represented in this object?*
d. *Note that this GRangesList object has names and that the names are Entrez Gene ids. Use `%in%` to check that Entrez Gene 1008 is present. Extract the exons of Entrez Gene 1008 in a GRanges object.*
   How many exons are in Entrez Gene 1008? On which chromosome and strand are they?
e. *The `elementLengths()` function is a very fast way to compute the length of each list element of a list-like object `x`. It's equivalent to (but much faster than) doing `sapply(x, length)`. Use it on the GRangesList object to compute the number of exons in each gene.*
   Which gene has the most exons? How many exons does it have?

Solution:

a.
```
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
TxDb.Hsapiens.UCSC.hg19.knownGene
## TranscriptDb object:
## | Db type: TranscriptDb
## | Supporting package: GenomicFeatures
## | Data source: UCSC
## | Genome: hg19
## | Organism: Homo sapiens
## | UCSC Table: knownGene
## | Resource URL: http://genome.ucsc.edu/
## | Type of Gene ID: Entrez Gene ID
## | Full dataset: yes
## | miRBase build ID: GRCh37
## | transcript_nrow: 82960
## | exon_nrow: 289969
## | cds_nrow: 237533
## | Db created by: GenomicFeatures package from Bioconductor
## | Creation time: 2013-09-19 00:15:15 -0700 (Thu, 19 Sep 2013)
## | GenomicFeatures version at creation time: 1.13.40
## | RSQLite version at creation time: 0.11.4
## | DBSCHEMAVERSION: 1.0
```

b.
```
ex_by_gn <- exonsBy(TxDb.Hsapiens.UCSC.hg19.knownGene, by="gene")
class(ex_by_gn)
## [1] "GRangesList"
## attr(,"package")
## [1] "GenomicRanges"
```

c. Number of genes:

```
length(ex_by_gn)
## [1] 23459
```

d. Is Entrez Gene 1008 present?

```
"1008" %in% names(ex_by_gn)
## [1] TRUE
```

To extract the exons in a *GRanges* object, we need to use `[[`, not `[`:

```
ex_by_gn[["1008"]]
## GRanges with 12 ranges and 2 metadata columns:
##        seqnames               ranges strand   |   exon_id   exon_name
##           <Rle>            <IRanges>  <Rle>   | <integer> <character>
##    [1]     chr5 [24487209, 24488262]      -   |     80706        <NA>
##    [2]     chr5 [24491685, 24491936]      -   |     80707        <NA>
##    [3]     chr5 [24492926, 24493034]      -   |     80708        <NA>
##    [4]     chr5 [24498507, 24498628]      -   |     80709        <NA>
##    [5]     chr5 [24505221, 24505357]      -   |     80710        <NA>
##    ...      ...                  ...    ... ...       ...         ...
##    [8]     chr5 [24535221, 24535388]      -   |     80713        <NA>
##    [9]     chr5 [24535812, 24535931]      -   |     80714        <NA>
##   [10]     chr5 [24537489, 24537783]      -   |     80715        <NA>
##   [11]     chr5 [24593369, 24593722]      -   |     80716        <NA>
##   [12]     chr5 [24644703, 24645085]      -   |     80717        <NA>
##   ---
##   seqlengths:
##                      chr1                  chr2 ...        chrUn_gl000249
##                249250621             243199373 ...                 38502
```

Also note that it’s important to subset with the `"1008"` *string* here, not the `1008` number. Why? Be aware that this is a common pitfall!

```
seqnames(ex_by_gn[["1008"]])
## factor-Rle of length 12 with 1 run
##   Lengths:   12
##   Values : chr5
## Levels(93): chr1 chr2 chr3 chr4 ... chrUn_gl000247 chrUn_gl000248 chrUn_gl000249
strand(ex_by_gn[["1008"]])
## factor-Rle of length 12 with 1 run
##   Lengths: 12
##   Values :  -
## Levels(3): + - *
```

All the exons in this gene are on chromosome 12 and minus strand.

e. Number of exons in each gene:

```
nex_per_gn <- elementLengths(ex_by_gn)
```

Max number of exons per gene:

```
max(nex_per_gn)
## [1] 481
```

Gene with the most exons:

```
which(nex_per_gn == max(nex_per_gn))
## 1302
## 4260
```

Entrez Gene ”1302”.
Alternatively, the ”which” and ”how many” questions can both be answered with `which.max()`:

```
which.max(nex_per_gn)
## 1302
## 4260
```

4 Working with aligned reads

4.1 Mapping the reads to the reference genome

A high-throughput sequencing experiment (a.k.a. HTS experiment, or new generation sequencing experiment, or NGS experiment) generates a huge quantity of short reads (typically millions). Depending on the technology used by the sequencing machine, the length of the reads can vary between 40 and 300 nucleotides. One of the first steps of almost any HTS data analysis is the *mapping* of the reads to the reference genome (or sometimes to the transcriptome). This *mapping* process consists in finding the genomic location of the read based on its similarity with the nucleotide content of the reference genome at this location. This step is very computational intensive and is usually performed outside *Bioconductor* with a 3rd party aligner like Bowtie, BWA, GSNAP, TopHat, etc...

Reads coming from an RNA-seq experiment (i.e. RNA was sequenced, not DNA) present an additional challenge: some of them, called *junction reads*, can span 2 or more exons separated by introns. In that case the aligner needs to be able to break the read into 2 or more pieces and map each individual piece to the reference genome. A few aligners like TopHat or GSNAP support *junction reads*. Others like Bowtie or BWA don’t.

The result of the alignment process is typically stored in BAM files. The BAM format (and its plain text version SAM) is a generic format for storing large nucleotide sequence alignments. It is specified and developped by the

SAMtools project (http://samtools.sourceforge.net/). The SAMtools project also provides various utilities for manipulating alignments in the SAM or BAM format.

4.2 Working with BAM files in *Bioconductor*

After the reads have been aligned and stored in BAM files (typically one file per sample or sequencing run), many packages are available in *Bioconductor* for performing different kinds of downstream analyses. See the High-ThroughputSequencing view under the Software and AssayTechnologies views on the *Bioconductor* website for a list of these packages (http://bioconductor.org/packages/release/BiocViews.html#___HighThroughputSequencing).

One of these packages, the *Rsamtools* package, provides an interface to the SAMtools utilities for manipulating BAM files, including for loading aligned reads in *R*. Other packages like the *ShortRead* package, provide some high-level tools for manipulating aligned reads. One container, the *GAlignments* container, has been specifically designed for storing aligned reads, including junction reads (a.k.a. gapped alignents). It is defined and documented in the *GenomicRanges* package.

The *parathyroidSE* package is a data package that contains aligned reads stored in BAM files. The paths to these BAM files can be obtained with:

```
library(parathyroidSE)
bamdir <- system.file("extdata", package="parathyroidSE")
bampaths <- list.files(bamdir, pattern="bam$", full.names=TRUE)
bampaths

## [1] "/home/mtmorgan/R/x86_64-unknown-linux-gnu-library/3.0-2.13/parathyroidSE/extdata/SRR479052.bam"
## [2] "/home/mtmorgan/R/x86_64-unknown-linux-gnu-library/3.0-2.13/parathyroidSE/extdata/SRR479053.bam"
## [3] "/home/mtmorgan/R/x86_64-unknown-linux-gnu-library/3.0-2.13/parathyroidSE/extdata/SRR479054.bam"
```

The aligned reads can be loaded from a BAM file with the `readGAlignmentsFromBam()` function from the *Rsamtools* package:

```
library(Rsamtools)
reads <- readGAlignmentsFromBam(bampaths[1])
reads

## GAlignments with 9973 alignments and 0 metadata columns:
##          seqnames strand       cigar    qwidth     start       end     width      ngap
##             <Rle>  <Rle> <character> <integer> <integer> <integer> <integer> <integer>
##      [1]       10      +        101M       101  59953037  59953137       101         0
##      [2]       10      -        101M       101  59953061  59953161       101         0
##      [3]       11      +        101M       101    209517    209617       101         0
##      [4]       10      +        101M       101 121691690 121691790       101         0
##      [5]       10      -        101M       101 121691702 121691802       101         0
##      ...      ...    ...         ...       ...       ...       ...       ...       ...
##   [9969]       14      -  71M117N30M       101  24035299  24035516       218         1
##   [9970]       16      +        101M       101  56466390  56466490       101         0
##   [9971]       16      -        101M       101  56466533  56466633       101         0
##   [9972]       MT      +        101M       101      6194      6294       101         0
##   [9973]       MT      -        101M       101      6316      6416       101         0
##   ---
##   seqlengths:
##             1        10        11        12 ...         9        MT         X         Y
##     249250621 135534747 135006516 133851895 ... 141213431     16569 155270560  59373566
```

The returned value is a *GAlignments* object. It is a *vector-like* container that has one element per aligned read. An important difference with *GRanges* objects is that an aligned read can be associated with more than one genomic range. We'll see later how to extract those ranges but before we do that, we first need to talk about the `cigar` component of the object.

The components displayed above can be accessed with accessor functions that are named like the component itself (e.g. `qwidth()` for the `qwidth` component, `ngap()` for the `ngap` component, etc...). One component of particular interest is the `cigar` component. It contains short strings called CIGAR strings that describe exactly the geometry of the alignments i.e. how each read aligns to the reference sequence. CIGAR strings are made of numbers and letters. Each letters represents a CIGAR operation and is preceeded by a number that specifies the length of the operation. The most important operations are `M`, `I`, `D`, and `N`. The format of the CIGAR strings is explained in the SAM Spec document hosted on the SAMtools website.

Exercise 5 *In this exercise we learn how to load aligned reads from a BAM file into a GAlignments object and do some basic manipulation on it.*

a. *Load the parathyroidSE package and use the `readGAlignmentsFromBam()` function from the Rsamtools package to read the third BAM file (SRR479054.bam). What kind of object is returned? Display it.*
b. *A gap in the alignment is called a "skipped region from the reference" in SAMtools jargon. Refer to the SAM Spec to find out how gaps are represented in the CIGAR strings.*
c. *Do some of the alignments have gaps? What does this tell us about the aligner that was used to align the reads?*
d. *The `ngap` component summarizes the number of gaps per alignment. What's the maximum number of gaps per alignment? Use `table()` to summarize the number of gaps per alignment.*

Solution:

a.
```
library(parathyroidSE)
bamdir <- system.file("extdata", package="parathyroidSE")
bampaths <- list.files(bamdir, pattern="bam$", full.names=TRUE)
reads <- readGAlignmentsFromBam(bampaths[3])
class(reads)
## [1] "GAlignments"
## attr(,"package")
## [1] "GenomicRanges"
reads
## GAlignments with 9973 alignments and 0 metadata columns:
##          seqnames strand       cigar    qwidth     start       end     width      ngap
##             <Rle>  <Rle> <character> <integer> <integer> <integer> <integer> <integer>
##      [1]        5      +    71M3I27M       101 118309615 118309712        98         0
##      [2]       20      +  48M104N53M       101  57484587  57484791       205         1
##      [3]       20      -  73M252N28M       101  57485064  57485416       353         1
##      [4]        9      +  90M243N11M       101 136223457 136223800       344         1
##      [5]        2      +        101M       101   9725148   9725248       101         0
##      ...      ...    ...         ...       ...       ...       ...       ...       ...
##   [9969]       16      +        101M       101  68849425  68849525       101         0
##   [9970]       16      -  5M1173N96M       101  68856124  68857397      1274         1
##   [9971]        3      +        101M       101  42700098  42700198       101         0
##   [9972]        3      -        101M       101  42700121  42700221       101         0
##   [9973]        2      +        101M       101  55200097  55200197       101         0
##   ---
##   seqlengths:
##             1        10        11        12 ...         9        MT         X         Y
##     249250621 135534747 135006516 133851895 ... 141213431     16569 155270560  59373566
```

b. A gap in the alignment is represented with the `N` operation.

c.
```
length(grep("N", cigar(reads)))
## [1] 3109
```

This tells us that the aligner that was used supports *junction reads*.

d. Maximum number of gaps per alignment:

```
max(ngap(reads))
## [1] 3
table(ngap(reads))
##
##    0    1    2    3
## 6864 2853  250    6
```

4.3 Extracting the genomic ranges from a *GAlignments* object

There are 2 ways to extract the genomic ranges from a *GAlignments* object: one that preserves the gaps, and one that doesn't (i.e. the gaps are ignored). This extraction is actually achieved by *coercing* the *GAlignments* object into a *GRangesList* or *GRanges* object.

Exercise 6 *In this exercise we learn how to extract the genomic ranges from a GAlignments object.*

a. *Coerce this GAlignments object into a GRangesList object. Note that this coercion produces an object that is parallel to the GAlignments object, that is, the i-th element in the GRangesList object corresponds to the i-th element (i.e. alignment) in the GAlignments object.*
b. *Use `elementLengths()` and `table()` to summarize the number of ranges per list element in the GRangesList object. How does this relate to the summary of number of gaps per alignment obtained earlier?*
c. *If we wanted to ignore the gaps in the reads, we would coerce the GAlignments object into a GRanges object. Do that.*

Solution:

a.
```
read_ranges <- as(reads, "GRangesList")
```

b.
```
table(elementLengths(read_ranges))
##
##    1    2    3    4
## 6864 2853  250    6
```

Number of ranges per alignment equals number of gaps plus one.

c.
```
as(reads, "GRanges")
## GRanges with 9973 ranges and 0 metadata columns:
##          seqnames                 ranges strand
##             <Rle>              <IRanges>  <Rle>
##      [1]        5 [118309615, 118309712]      +
##      [2]       20 [ 57484587,  57484791]      +
##      [3]       20 [ 57485064,  57485416]      -
##      [4]        9 [136223457, 136223800]      +
##      [5]        2 [  9725148,   9725248]      +
##      ...      ...                    ...    ...
##   [9969]       16   [68849425, 68849525]      +
##   [9970]       16   [68856124, 68857397]      -
##   [9971]        3   [42700098, 42700198]      +
##   [9972]        3   [42700121, 42700221]      -
##   [9973]        2   [55200097, 55200197]      +
##   ---
##   seqlengths:
##            1        10        11        12 ...         9        MT         X         Y
##    249250621 135534747 135006516 133851895 ... 141213431     16569 155270560  59373566
```

4.4 Computing the *coverage* of the reads

Finally, we can use the *GRangesList* object obtained in the previous exercise to compute the *coverage* of the reads, that is, the number of reads that cover each genomic position in the reference genome.

Exercise 7 *Computing the read coverage.*

a. *Use the `coverage()` function on `read_ranges` to compute the* coverage *of the reads.*
   What is the class of the object returned by `coverage()`? Display the object.
b. *Extract the coverage of chromosome 12 as an integer-Rle. What is the maximum coverage on chromosome 12?*
c. *Use `max()` on the full coverage object (RleList) to get the maximum coverage on each chromosome.*

Solution:

a.
```
cvg <- coverage(read_ranges)
class(cvg)
## [1] "SimpleRleList"
## attr(,"package")
## [1] "IRanges"
cvg
## RleList of length 25
## $`1`
## integer-Rle of length 249250621 with 1891 runs
##   Lengths:  565027       7       9      14      29 ...     100     480     101  100017
##   Values :       0       1       2       3       4 ...       1       0       1       0
##
## $`10`
## integer-Rle of length 135534747 with 1021 runs
##   Lengths:  294461      87     294      11       3 ...      92    1929       9  352312
##   Values :       0       1       0       1       2 ...       1       0       1       0
##
## $`11`
## integer-Rle of length 135006516 with 1316 runs
##   Lengths:  213325      30      63     811       8 ...      14    2285      86  871629
##   Values :       0       1       2       0       2 ...       1       0       1       0
##
## $`12`
## integer-Rle of length 133851895 with 1416 runs
##   Lengths:  404750      67      34      67  266509 ...   10445      26       1  319040
##   Values :       0       1       2       1       0 ...       0       2       1       0
##
## $`13`
## integer-Rle of length 115169878 with 363 runs
##   Lengths: 20567820       47       54       47 ...        3       98        3    77442
##   Values :        0        1        2        1 ...        1        2        1        0
##
```

```
## ...
## <20 more elements>
```

b. Coverage of chromosome 12:

```
cvg[["12"]]
## integer-Rle of length 133851895 with 1416 runs
##   Lengths:  404750       67       34       67   266509 ...    10445       26        1   319040
##   Values :       0        1        2        1        0 ...        0        2        1        0
max(cvg[["12"]])
## [1] 6
```

c. Maximum coverage on each chromosome:

```
max(cvg)
##  1 10 11 12 13 14 15 16 17 18 19  2 20 21 22  3  4  5  6  7  8  9 MT  X  Y
##  8  6 28  6  5 13 16  4  7  4  6  8  6  5  3  5  4  5 12 21  8 19 56  4  3
```