BioC for HTS - PDCB topic
Infrastructure and Input/Output 02

LCG Leonardo Collado Torres
lcollado@wintergenomics.com – lcollado@ibt.unam.mx

October 13th, 2010
Exercises

IRanges

GenomicRanges
Some practice

- From the `aln` object, extract the dinucleotide frequency for the last 2 cycles.

```r
> library(ShortRead)
> exptPath <- system.file("extdata", + package = "ShortRead")
> sp <- SolexaPath(exptPath)
> aln <- readAligned(sp, "s_2_export.txt")
> di <- colSums(dinucleotideFrequency(sread(narrow(aln, + start = 34, width = 2))))
> di
```
Some practice

<table>
<thead>
<tr>
<th>AA</th>
<th>AC</th>
<th>AG</th>
<th>AT</th>
<th>CA</th>
<th>CC</th>
<th>CG</th>
<th>CT</th>
<th>GA</th>
<th>GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>52</td>
<td>50</td>
<td>74</td>
<td>72</td>
<td>57</td>
<td>23</td>
<td>63</td>
<td>76</td>
<td>43</td>
</tr>
<tr>
<td>GG</td>
<td>GT</td>
<td>TA</td>
<td>TC</td>
<td>TG</td>
<td>TT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>49</td>
<td>64</td>
<td>67</td>
<td>65</td>
<td>115</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Given the GC percentage of all cycles, did you expect the results you observe?

```r
> gc <- alphabetByCycle(sread(aln))
> gc <- colSums(gc[c("C", "G"), ])/colSums(gc)
> diGC <- unlist(lapply(strsplit(names(di), ""), function(x) {
+   sum(x %in% c("C", "G"))
+ })))
> diGC

[1] 0 1 1 0 1 2 2 1 1 2 2 1 0 1 1 0```
Some practice

```r
> expected <- lapply(gc, function(x) {
+   res <- (x * diGC) + ((1 - x) * (2 - diGC))
+   res <- res/sum(res) * length(aln)
+   return(res)
+ })
> expected[[1]]

[1] 61.0 62.5 62.5 61.0 62.5 64.0 64.0
[8] 62.5 62.5 64.0 64.0 62.5 61.0 62.5
[15] 62.5 61.0

> res <- unlist(lapply(expected, function(x) {
+   t.test(di, round(x), paired = TRUE)$p.value
+ })))
> sum(res < 0.05)
```

5 / 85
Some practice

[1] 0

2. Which is the read with NN at the end?

> sum(di)

[1] 999

> table(as.character(sread(narrow(aln, + start = 34, width = 2))))

AA  AC  AG  AT  CA  CC  CG  CT  GA  GC
85  52  50  74  72  57  23  63  76  43
GG  GT  NN  TA  TC  TG  TT
44  49   1  64  67  65 115

> i <- which(as.character(sread(narrow(aln, + start = 34, width = 2)))) == + "NN")
> i
Some practice

[1] 882
> sread(aln[i])

A DNAStringSet instance of length 1
width seq
[1] 35 GNNNNNNNNNNNNNN...NNNNNNNNNNNNNN

3. Is there a significative difference vs the dinucleotide frequency of cycles 15 and 16?
> res[15:16]
[1] 0.9886094 0.9627654
> res[15:16] < 0.05
[1] FALSE FALSE
Some practice

▶ Load the na19240url object (note that fpt doesn’t wort at IBt).

   > library(Rsamtools)
   > load("na19240.Rda")

▶ Are all reads of the same length? If not, what is the distribution? Make a cumulative plot.

   > unlist(lapply(na19240bam[[1]],
       +     class))
Some practice

qname
"character"

rname
"factor"

pos
"integer"

mapq
"integer"

mrnm
"factor"

isize
"integer"

qual
"PhredQuality"

flag
"integer"

strand
"factor"

qwidth
"integer"

cigar
"character"

mpos
"integer"

seq
"DNASTringSet"
Some practice

```r
> l.seqs <- width(na19240bam[[1]]$seq)
> t.seqs <- table(l.seqs)
> t.seqs

l.seqs
  31  35  36  45  50  51
  61 776 6966 782  656 1631

> table(na19240bam[[1]]$qwidth)
  31  35  36  45  50  51
  60 719 6646 753  633 1619
```
Some practice

```r
> plot(x = as.numeric(names(t.seqs)),
+       y = cumsum(t.seqs)/sum(t.seqs) * 100, type = "o", col = "blue",
+       xlab = "Width", ylab = "Percent of Reads",
+       main = "Cumulative Distribution of Read Width")
```
Some practice

Cumulative Distribution of Read Width

- Width
- Percent of Reads

- Width: 35, 40, 45, 50
- Percent of Reads: 0, 20, 40, 60, 80, 100
Some practice

- Convert the PhredQuality instance to a quality matrix and make a plot of the median quality per cycle. Is there any trend in the quality?

```r
> as(na19240bam[[1]]$qual, "matrix")

> qual <- lapply(na19240bam[[1]]$qual, 
+     as.integer)

> mat <- matrix(NA, nrow = length(qual), 
+     ncol = max(unlist(lapply(qual, 
+         length))))

> for (i in 1:length(qual)) mat[i, 
+     1:length(qual[[i]])] <- qual[[i]] - 
+     33
```
Some practice

```r
> mat2 <- mat
> for (i in 1:nrow(mat2)) mat2[i, +  is.na(mat2[i, ])] <- 0
> medians <- apply(mat, 2, median, +  na.rm = TRUE)
> medians2 <- apply(mat2, 2, median)
> plot(1:length(medians), medians, +  type = "o", xlab = "Cycle", +  ylab = "Median quality", col = "blue", +  ylim = c(min(medians, medians2), +     max(medians, medians2)))
> lines(medians2, type = "o", col = "red", +     lty = 2)
```
Some practice
Some practice

- Make a third plot for the alphabet by cycle relative frequency (in percent). Do you observe anything unexpected?

```r
> abc <- alphabetByCycle(na19240bam[[1]]$seq)
> abc <- t(abc[rowSums(abc) > 0, ])
> abc <- abc/rowSums(abc) * 100
> abc <- as.data.frame(abc)
> library(lattice)
> print(xyplot(A + C + G + T + N ~ 1:nrow(abc), data = abc, auto.key = TRUE,
+     type = c("o", "g"), xlab = "Cycle",
+     ylab = "Nucleotide Frequency (in %)"))
```
Some practice
Overview

IRanges is an infrastructure package that

- will help us save memory
- allows us to manipulate data in ranges
- is the backbone for manipulating our HTS data
Rle

- **Rle** or Run length encoded objects are the main source of memory usage reduction in IRanges.
- The idea: if neighbor values are frequently repeated in a vector, we can now construct a kind of matrix where we have each value in one row and the number of times it appears on the second row.
- For example, we have the vector `x` which is made up of 0s and 1s:
  ```r
  > x <- round(runif(10000))
  > table(x)
  
x
  0   1
  4991 5009
  ```
> head(x)
[1] 0 0 1 0 1 1
> tail(x)
[1] 0 1 0 0 1 0

We can observe that 0s are 1s are adjacent to each other quite frequently. This kind of vector is a good candidate to transform into an Rle:

> library(IRanges)
> y <- Rle(x)
> y
Rle

'numeric' Rle of length 10000 with 5075 runs
  Lengths: 2 1 1 4 3 1 ... 1 1 1 2 1 1
  Values : 0 1 0 1 0 1 ... 1 0 1 0 1 0

- This allows us to save some memory:
  > print(object.size(x) - object.size(y),
    +       units = "Kb")
  17.4 Kb

- With larger vectors, we save more memory :) 
  > x2 <- round(runif(4e+06))
  > y2 <- Rle(x2)
  > print(object.size(x2) - object.size(y2),
    +       units = "Kb")
  7793.4 Kb
More on Rle

- Just like with vectors, several basic accessors have been implemented:
  
  ```r
  > x[2:3]
  [1] 0 1
  > y[2:3]
  'numeric' Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : 0 1
  > c(x[2:3], x[10])
  [1] 0 1 0
  > c(y[2:3], y[10])
  ```
More on Rle

'numeric' Rle of length 3 with 3 runs
  Lengths: 1 1 1
  Values : 0 1 0

> identical(c(y[2:3], y[10]), append(y[2:3], + y[10]))

[1] TRUE

▶ And if needed, you can always return to a vector:
> as.vector(y[1:10])

[1] 0 0 1 0 1 1 1 0 0

> identical(x, as.vector(y))

[1] TRUE
More on Rle

- Our object \( y \) is a numeric Rle. We can also create other types of Rles:

\[
\begin{align*}
&> z \leftarrow y > 0 \\
&> m \leftarrow \text{Rle}(\text{sample}(c("A", "C", "T", + "G"), 10000, \text{replace} = \text{TRUE})) \\
&> n \leftarrow \text{Rle}(\text{as.factor}(\text{sample}(1:10, + 10000, \text{replace} = \text{TRUE}))) \\
&> o \leftarrow \text{Rle}(\text{as.integer}(x)) \\
&> \text{identical}(y, o) \\
&[1] \text{FALSE}
\end{align*}
\]

- Using the function \texttt{strand} we can create a special kind of factor:
More on Rle

```r
> str <- strand(sample(c("+", "-"),
+     1000, replace = TRUE))
> class(str)
[1] "factor"
> levels(str)
[1] "+" "-" "*
> head(str)
[1] - - - + + +
Levels: + - *

▶ Without any problems, we can convert this strand factor into an Rle:
> Rle(str)
```

More on Rle

'factor' Rle of length 1000 with 490 runs

Lengths: 3 3 3 3 3 3 ... 1 4 1 1 1 1

Values: - + - + - + ... - + - + - +

Levels(3): + - *
Lets practice a bit

- Using the following vectors `a` and `b`, what are the mean and median `a` values for each level of the `b` factor. Transform them into Rle objects.

```r
> a <- round(runif(1e+05, min = 0, + max = 10000))
> b <- sample(c("+", "-"), 1e+05, + replace = TRUE)
```
First, a long solution where we transform our objects back to vectors in order to use the functions `mean` and `median`:

```r
> e <- Rle(a)
> f <- Rle(b)
> sapply(unique(f), function(x) {
+   set <- as.vector(e[f == x])
+   res <- c(mean(set), median(set))
+ })
```

- 
  [1,] 5006.663 4992.109
  [2,] 5017.000 4990.000
Solutions

- However, the above is not necessary since both mean and median have methods for Rle objects. Hence, we can solve it like this:

  ```r
  > sapply(unique(f), function(x) {
  + set <- e[f == x]
  + res <- c(mean(set), median(set))
  + })
  - +
  [1,] 5006.663 4992.109
  [2,] 5017.000 4990.000
  ```

- Yet, the ideal scenario is to use the `tapply` function :) Either one by one or all together.

  ```r
  > tapply(e, f, mean)
  ```
Solutions

\[ - + \]
\[
5006.663 \quad 4992.109
\]

\[
> \ tapply(e, f, median)
\]

\[ - + \]

\[
5017 \quad 4990
\]

\[
> \ tapply(e, f, function(x) {
+ \quad c(mean(x), median(x))
+ \})
\]

\`
[1] 5006.663 5017.000
\`

\`
[1] 4992.109 4990.000
\`
Solutions

- In this case we could have used `tapply` from the start with the vectors `a` and `b`:

  ```r
  > tapply(a, b, function(x) {
  +   c(mean(x), median(x))
  + })
  $`-`
  [1] 5006.663 5017.000
  
  $`+`
  [1] 4992.109 4990.000
  
- Basically, we can use `Rle`'s just as we would use vectors yet we get the memory advantage :)\(^1\)

- Btw, this was another solution:
Solutions

```r
> tapply(e, f, function(x) {
+ summary(x)[3:4]
+ })
```

```
Median  Mean
5017    5007
```

```
Median  Mean
4990    4992
```

\(^1\)To make full use of the advantage we shouldn't create the vectors, just create the Rles directly.
More on Rles

- Just like with vectors, we can reverse or access a subsection of an Rle

```r
> y

'numeric' Rle of length 10000 with 5075 runs
  Lengths: 2 1 1 4 3 1 ... 1 1 1 2 1 1
  Values : 0 1 0 1 0 1 ... 1 0 1 0 1 0

> rev(y)

'numeric' Rle of length 10000 with 5075 runs
  Lengths: 1 1 2 1 1 1 ... 1 3 4 1 1 2
  Values : 0 1 0 1 0 1 ... 1 0 1 0 1 0

> window(y, 2, 4)
```
More on Rles

'nNumeric' Rle of length 3 with 3 runs
   Lengths: 1 1 1
   Values : 0 1 0

▶ We can also get into the parts of an Rle object using:
   > head(runLength(y))
   [1]  2 1 1 4 3 1
   > head(runValue(y))
   [1] 0 1 0 1 0 1

▶ Remember the matrix idea that lead to Rles? Well, we can build that said matrix:
More on Rles

```r
> mat <- matrix(0, nrow = nrun(y),
+                ncol = 2)
> mat[, 1] <- runLength(y)
> mat[, 2] <- runValue(y)
> head(mat)

        [,1] [,2]
   [1,]    2  0
   [2,]    1  1
   [3,]    1  0
   [4,]    4  1
   [5,]    3  0
   [6,]    1  1

> y
```
More on Rles

'n numeric' Rle of length 10000 with 5075 runs
  Lengths: 2 1 1 4 3 1 ... 1 1 1 2 1 1
  Values : 0 1 0 1 0 1 ... 1 0 1 0 1 0

➤ We can also get the start and end positions for each run:
  > head(start(y))
  [1] 1 3 4 5 9 12
  > head(end(y))
  [1] 2 3 4 8 11 12

➤ There are plenty of other numerical and character methods for Rles which you find on the help page for Rle. For example:
  > cor(y, e[1:10000])
  [1] 0.001470080
More on Rles

```r
> range(e)

[1] 0 10000

We can also create list of Rle objects:

```r
g > rlelist <- RleList(y, e[1:10000])
> rlelist

SimpleRleList of length 2
[[1]]
'numeric' Rle of length 10000 with 5075 runs
  Lengths: 2 1 1 4 3 1 ... 1 1 1 2 1 1
  Values : 0 1 0 1 0 1 ... 1 0 1 0 1 0

[[2]]
'numeric' Rle of length 10000 with 10000 runs
```
More on Rles

Lengths:  1  1  1  ...  1  1
Values:   5614  7993  5994  ...  4163  9976
Another fundamental piece of IRanges is the ability to construct matrixes of ranges using IRanges. For example:

```r
> IR <- IRanges(start = 1:5, end = 6:10)
```

Data from an IRanges object can be easily accessed:

```r
> length(IR)
[1] 5
> IR[2]
```

IRanges of length 1
```
  start  end  width
[1] 2 7 6
```

```r
> start(IR[5])
```

IRanges

[1] 5
> end(IR[3])

[1] 8
> width(IR)

[1] 6 6 6 6 6

Once we have ranges, we can manipulate them:

> reduce(IR)

IRanges of length 1
   start   end width
[1]    1   10    10

> disjoin(IR)
### IRanges

<table>
<thead>
<tr>
<th>IRanges of length 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
</tr>
<tr>
<td>[1]</td>
</tr>
<tr>
<td>[2]</td>
</tr>
<tr>
<td>[3]</td>
</tr>
<tr>
<td>[4]</td>
</tr>
<tr>
<td>[5]</td>
</tr>
<tr>
<td>[6]</td>
</tr>
<tr>
<td>[7]</td>
</tr>
<tr>
<td>[8]</td>
</tr>
<tr>
<td>[9]</td>
</tr>
</tbody>
</table>

And find overlaps between ranges:


```r
> ov <- findOverlaps(IR, reduce(IR))
> as.matrix(ov)

<table>
<thead>
<tr>
<th>query</th>
<th>subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,]</td>
<td>1</td>
</tr>
<tr>
<td>[2,]</td>
<td>2</td>
</tr>
<tr>
<td>[3,]</td>
<td>3</td>
</tr>
<tr>
<td>[4,]</td>
<td>4</td>
</tr>
<tr>
<td>[5,]</td>
<td>5</td>
</tr>
</tbody>
</table>
```

query: subject

```r
1
1
1
1
1
```
Exercise

- Construct an IRanges object where we’ll have 1 range per every read.
- Use the position and width of the read from the aln object

```r
> aln

class: AlignedRead
length: 1000 reads; width: 35 cycles
chromosome: NM NM ... chr5.fa 29:255:255
position: NA NA ... 71805980 NA
strand: NA NA ... + NA
alignQuality: NumericQuality
alignData varLabels: run lane ... filtering contig
```
Solution

- We just need to be careful with the reads from the minus strand and those that did not map
  ```r
  > idx <- which(!is.na(position(aln)))
  > start <- position(aln[idx])
  > str <- strand(aln[idx]) == "-
  > start[str] <- start[str] - width(aln[idx])[str] + 1
  > reads <- IRanges(start = start,
                  +   width = width(aln[idx]))
  ```

- Once we have our reads in an IRanges object, we can get information such as the coverage:
  ```r
  > cov <- coverage(reads)
  > cov
  ```
Solution

'integer' Rle of length 195524766 with 810 runs

Lengths:  11907  35 ...  35
Values :   0  1 ...  1

➢ Or manipulate further the ranges:

> shift(IR, 10)

IRanges of length 5

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

> narrow(IR, start = 1, width = 2)
Solution

IRanges of length 5

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

> `flank(IR, 1)`
Solution

IRanges of length 5

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Exercise

- Use the function `findOverlaps` to find the overlaps between our reads.
- Avoid obvious and repetitive overlaps (like range 1 vs range 1).
Solution

▶ We need to change the default values for two arguments :)  
> \texttt{ovReads <- matchMatrix(findOverlaps(reads, + ignoreSelf = TRUE, ignoreRedundant = TRUE))}
> \texttt{ovReads}

<table>
<thead>
<tr>
<th>query</th>
<th>subject</th>
<th>overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,]</td>
<td>8</td>
<td>156</td>
</tr>
<tr>
<td>[2,]</td>
<td>54</td>
<td>104</td>
</tr>
<tr>
<td>[3,]</td>
<td>54</td>
<td>374</td>
</tr>
<tr>
<td>[4,]</td>
<td>104</td>
<td>374</td>
</tr>
<tr>
<td>[5,]</td>
<td>361</td>
<td>371</td>
</tr>
</tbody>
</table>

▶ Which reads overlap with the 100 upstream to other reads? Are the results the same?
Solution part B

- We’ll use flank to get the upstream regions :

```r
> ovReadsUp <- matchMatrix(findOverlaps(reads,  
+     flank(reads, 100)))
> ovReadsUp

query subject
[1,]  54   104
[2,]  54   374
[3,] 104   374
[4,] 156    8
```
RangedData

- The third main object from the IRanges package is the RangedData object.
- It is basically a table with an IRanges object inside of it:
  ```r
  rd <- RangedData(ranges = IR, space = rep("chr", 5), name = letters[1:5])
  ```
- Once we have a RangedData object, we can get the names, coverage per space, access the different extra columns (name in this case), or get the IRanges object inside of the RangedData:
  ```r
  names(rd)
  [1] "chr"
  coverage(rd)
  ```
RangedData

SimpleRleList of length 1
$chr
'integer' Rle of length 10 with 9 runs
  Lengths: 1 1 1 1 2 1 1 1 1
  Values : 1 2 3 4 5 4 3 2 1

> rd$name
[1] "a" "b" "c" "d" "e"

> rd$space
[1] chr chr chr chr chr chr
Levels: chr

> ranges(rd)
RangedData

CompressedIRangesList of length 1
$\text{chr}$
IRanges of length 5

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

- Using our object reads, build a RangedData where the space is the chromosome where the read aligned. You might need to use our object idx.
RangedData

- Get the summary statistics for the coverage of chr5 (exclude bases with coverage equal to 0).
Solution

First we build the RangedData object:

```r
> readsRD <- RangedData(ranges = reads,
+   space = chromosome(aln[idx]))
> names(readsRD)
```

```
[1] "chr10.fa" "chr11.fa"
[3] "chr12.fa" "chr13.fa"
[5] "chr14.fa" "chr15.fa"
[7] "chr16.fa" "chr17.fa"
[9] "chr18.fa" "chr19.fa"
[11] "chr1.fa"  "chr2.fa"
[13] "chr3.fa" "chr4.fa"
[15] "chr5.fa" "chr6.fa"
[17] "chr7.fa" "chr8.fa"
```
Solution

[19] "chr9.fa"  "chrM.fa"
[21] "chrUn_random.fa" "chrX.fa"
[23] "chrY_random.fa"

Next we get the coverage for each space (chromosome), and finally we get the summary statistics we wanted:

```r
> covRD <- lapply(readsRD, coverage)
> covRD[["chr5.fa"]]

SimpleRleList of length 1
$chr5.fa
'integer' Rle of length 140154350 with 58 runs
  Lengths:  3936448 . . .  35
  Values :  0 . . .  1
```
Solution

```r
> summary(covRD[,"chr5.fa"][[1]][covRD[,"chr5.fa"]][[1]] + 0)

               Min. 1st Qu.  Median      Mean 3rd Qu. Max.
               1       1       1       1       1       1
```

Max.
1
Overview

- While built on top of IRanges, GenomicRanges provides a biological-aware framework to work with :)
- The GRanges class outperforms the RangedData class
- Caution: some methods have yet to be implemented for GRanges objects
GRanges

- It’s very similar to RangedData as the minimum information includes an IRanges object.
- Yet, now it requires strand information as well as the names.
- Let's build a GRanges object using our previous IR object:

```r
> GR <- GRanges(seqnames = rep("chr", 5), ranges = IR, strand = rep("*", 5), someVar = letters[1:5])
> GR
```
### GRanges

GRanges with 5 ranges and 1 element

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
<th>someVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>chr [1, 6]</td>
<td>*</td>
<td>a</td>
</tr>
<tr>
<td>[5]</td>
<td>chr [5, 10]</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>
GRanges

[5] e

seqlengths
chr
NA

Note the seqlengths section. We can specify the length of each unique seqname. This information affects the result from the coverage function:

> coverage(GR)
GRanges

SimpleRleList of length 1
$chr
'integer' Rle of length 10 with 9 runs
   Lengths: 1 1 1 1 2 1 1 1 1
   Values : 1 2 3 4 5 4 3 2 1

> seqlengths(GR) <- 20
> coverage(GR)

SimpleRleList of length 1
$chr
'integer' Rle of length 20 with 10 runs
   Lengths: 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1
   Values : 1 2 3 4 5 4 3 2 1 0
GRanges

Similar to RangedData objects, we can access parts of our GRanges object with:

> strand(GR)

'factor' Rle of length 5 with 1 run
  Lengths: 5
  Values : *
Levels(3): + - *

> start(GR)

[1] 1 2 3 4 5

> end(GR)

[1] 6 7 8 9 10

> width(GR)
GRanges

[1] 6 6 6 6 6

> ranges(GR)

IRanges of length 5

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>[2]</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>[3]</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>[4]</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>[5]</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

> GR[2:3]
GRanges

GRanges with 2 ranges and 1 elementMetadata value

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
<th>someVar</th>
<th>seqlengths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;Rle&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;Rle&gt;</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td>chr</td>
<td>[2, 7]</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>chr</td>
<td>[3, 8]</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>someVar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;character&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>seqlengths</td>
<td>chr</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
GRanges

Do you remember the class **DataFrame**. Well, that’s the class of the part of a GRanges object that contains information for the extra variables. Basically, it’s a data.frame where each column can be a vector, an Rle, etc.

Data Frame with 5 rows and 1 column

```
someVar
<character>
1    a
2    b
3    c
4    d
5    e
```
GRanges

Data Frame with 5 rows and 1 column

<table>
<thead>
<tr>
<th></th>
<th>someVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
</tr>
</tbody>
</table>

[1] "a" "b" "c" "d" "e"

- Plus, just like IRanges, we can manipulate the ranges:

```r
> flank(GR[5], 1)
```
GRanges

GRanges with 1 range and 1 element

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Rle&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;Rle&gt;</td>
</tr>
<tr>
<td>[1]</td>
<td>chr</td>
<td>[4, 4] *</td>
</tr>
<tr>
<td>someVar</td>
<td>&lt;character&gt;</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

seqlengths
chr
20

> disjoin(GR[4:5])
GRanges

GRanges with 3 ranges and 0 elementMetadata values

```
seqnames   ranges      strand |
<Rle>      <IRanges>   <Rle>  |
[1]        chr [ 4, 4]   *   |
[3]        chr [10, 10]  *   |
```

seqlengths

```
chr
20
```

> `shift(GR[3], 2)`
**GRanges**

GRanges with 1 range and 1 element

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
<th>someVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Rle&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;Rle&gt;</td>
<td></td>
</tr>
<tr>
<td>[1] chr</td>
<td>[5, 10] *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>someVar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;character&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1] c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

seqlengths

| chr | 20 |
Exercise

- Lets repeat the previous exercise where we looked for overlaps between
  1. reads
  2. reads and the 100bp upstream of reads
- First, we’ll need to construct a GRanges object using the reads from the aln object.
Solution

- Lets construct the GRanges object:
  ```r
  > readsGR <- GRanges(seqnames = chromosome(aln[idx]),
  +    ranges = reads, strand = Rle(strand(aln[idx])))
  ```

- Next, lets find the overlaps between reads:
  ```r
  > findOverlaps(readsGR, ignoreSelf = TRUE,
  +      ignoreRedundant = FALSE)
  ```

- Sadly, that doesn’t work yet. So lets do it the hard way:
Solution

```r
> ov <- matchMatrix(findOverlaps(readsGR,
+     readsGR))
> removeSelf <- function(ov) {
+     ov2 <- NULL
+     for (i in 1:nrow(ov)) if (ov[i,
+         1] != ov[i, 2])
+         ov2 <- rbind(ov2, ov[i,
+             ])
+     return(ov2)
+ }
> removeRedundant <- function(ov) {
+     index <- apply(ov, 1, function(x) {
+         res <- TRUE
+         x <- as.vector(x)
```
Solution

```r
for (j in 1:nrow(ov)) {
  y <- as.vector(ov[j, ])
  if (identical(y, x))
    break
  if (identical(y, rev(x)))
    res <- FALSE
}
return(res)
}
return(ov[index, ])
```

```r
> ov <- removeRedundant(removeSelf(ov))
> ov
```
Solution

query subject
[1,] 8 156
[2,] 54 104
[3,] 361 371

► Our new result is slight different that our original result:

> ovReads

query subject
[1,] 8 156
[2,] 54 104
[3,] 54 374
[4,] 104 374
[5,] 361 371
Solution

Next, lets find the overlaps between reads and upstream regions of reads.

```r
> matchMatrix(findOverlaps(readsGR,
+    flank(readsGR, 100)))

query subject
[1,]   8   156
[2,]  54   104

> ovReadsUp

query subject
[1,]  54   104
[2,]  54   374
[3,] 104   374
[4,] 156    8
```
Solution

- Just as above, the result is different. The reason: overlaps in GRanges objects takes into account the strand!
GRangesList

- A follow up class to GRanges is GRangesList. That’s the default output of the `split` function:
  ```r
  > grList <- split(GR)
  > class(grList)
  [1] "GRangesList"
  attr(,"package")
  [1] "GenomicRanges"
  ```

- You don’t need double brackets to access the elements of a GRangesList:
  ```r
  > grList[1:2]
  ```
GRangesList

GRangesList of length 2
$1
GRanges with 1 range and 1 element

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Rle&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;Rle&gt;</td>
</tr>
</tbody>
</table>

[1]
chr    [1, 6]   * |

someVar

<character>

[1] a

$2
GRanges with 1 range and 1 element

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Rle&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;Rle&gt;</td>
</tr>
</tbody>
</table>

79 / 85
GRangesList

[1] chr [2, 7] * |

someVar
<character>

[1] b

seqlengths

chr
20

- Functions like coverage work with all the elements of a GRangesList. Accessors like strand work with each element individually:

> coverage(grList)
**GRangesList**

```
SimpleRleList of length 1
$chr
'integer' Rle of length 10 with 9 runs
   Lengths: 1 1 1 1 2 1 1 1 1
   Values : 1 2 3 4 5 4 3 2 1

> strand(grList)

CompressedRleList of length 5
$`1`
'factor' Rle of length 1 with 1 run
   Lengths: 1
   Values : *
Levels(3): + - *
```
GRangesList

`$`2`
'factor' Rle of length 1 with 1 run
   Lengths: 1
   Values : *
Levels(3): + - *

`$`3`
'factor' Rle of length 1 with 1 run
   Lengths: 1
   Values : *
Levels(3): + - *

`$`4`
'factor' Rle of length 1 with 1 run
**GRangesList**

- Lengths: 1
- Values: *
- Levels(3): + - *

$`5``$

'factor' Rle of length 1 with 1 run
- Lengths: 1
- Values: *
- Levels(3): + - *

- The idea behind GRangesList is that you can have all the exons of a given gene in a GRanges, and have one element in your GRangesList per every gene.
Session Information

> sessionInfo()

R version 2.12.0 Under development (unstable) (2010-09-08 r52880)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
[1] LC_CTYPE=en_US.utf8
[2] LC_NUMERIC=C
[3] LC_TIME=en_US.utf8
[4] LC_COLLATE=en_US.utf8
[5] LC_MONETARY=C
[6] LC_MESSAGES=en_US.utf8
[7] LC_PAPER=en_US.utf8
[8] LC_NAME=C
[9] LC_ADDRESS=C
[10] LC_TELEPHONE=C
[12] LC_IDENTIFICATION=C

attached base packages:
Session Information

[1] stats       graphics    grDevices
[4] utils       datasets    methods
[7] base

other attached packages:
[1] ShortRead_1.7.20
[2] Rsamtools_1.1.15
[3] lattice_0.19-11
[4] Biostrings_2.17.41
[5] GenomicRanges_1.1.25
[6] IRanges_1.7.34

loaded via a namespace (and not attached):
[1] Biobase_2.9.0  grid_2.12.0
[3] hwriter_1.2