1 Introduction

The IRanges package is designed to represent sequences, ranges representing indices along those sequences, and data related to those ranges. In this vignette, we will rely on simple, illustrative example datasets, rather than large, real-world data, so that each data structure and algorithm can be explained in an intuitive, graphical manner. We expect that packages that apply IRanges to a particular problem domain will provide vignettes with relevant, realistic examples.

The IRanges package is available at bioconductor.org and can be downloaded via biocLite:

```r
> source("http://bioconductor.org/biocLite.R")
> biocLite("IRanges")
> library(IRanges)
```

2 Vector objects

In the context of the IRanges package, a sequence is an ordered finite collection of elements. The IRanges packages represents two types of objects as sequences: (1) atomic sequences and (2) lists (or non-atomic sequences). The following subsections describe each in turn. All IRanges-derived sequences inherit from the Vector virtual class.

2.1 Atomic Vectors

In R, atomic sequences are typically stored in atomic vectors. The IRanges package includes an additional atomic sequence object type, Rle, which compresses an atomic sequence through run-length encoding.

We begin our discussion of atomic sequences using two Rle vectors.

```r
> set.seed(0)
> lambda <- c(rep(0.001, 4500), seq(0.001, 10, length = 500),
+ seq(10, 0.001, length = 500))
> xVector <- Rle(rpois(1e7, lambda))
> yVector <- Rle(rpois(1e7, lambda[c(251:length(lambda), 1:250)]))
```

All atomic sequences in R have three main properties: (1) a notion of length or number of elements, (2) the ability to extract elements to create new atomic sequences, and (3) the ability to be combined with one or more atomic sequences to form larger atomic sequences. The main functions for these three operations are length, [, and c.

```r
> length(xVector)

[1] 10000000
```
While these three methods may seem trivial, they provide a great deal of power and many atomic sequence manipulations can be constructed using them.

### 2.1.1 Vector Subsetting

As with ordinary R atomic vectors, it is often necessary to subset one sequence from another. When this subsetting does not duplicate or reorder the elements being extracted, the result is called a subsequence. In general, the [ function can be used to construct a new sequence or extract a subsequence, but its interface is often inconvenient and not amenable to optimization. To compensate for this, the IRanges package supports seven additional functions for sequence extraction:

1. **window** - Extracts a subsequence over a specified region.
2. **subset** - Extracts the subsequence specified by a logical vector.
3. **head** - Extracts a consecutive subsequence containing the first n elements.
4. **tail** - Extracts a consecutive subsequence containing the last n elements.
5. **rev** - Creates a new sequence with the elements in the reverse order.
6. **rep** - Creates a new sequence by repeating sequence elements.

The following code illustrates how these functions are used on an ordinary R integer vector:

```r
> xSnippet <- xVector[IRanges(4751, 4760)]
> xSnippet
integer-Rle of length 10 with 9 runs
   Lengths: 1 1 1 1 1 1 1 1 2
   Values : 4 6 5 4 6 2 6 7 5

> head(xSnippet)
integer-Rle of length 6 with 6 runs
   Lengths: 1 1 1 1 1 1
   Values : 4 6 5 4 6 2

> tail(xSnippet)
integer-Rle of length 6 with 5 runs
   Lengths: 1 1 1 1 2
   Values : 6 2 6 7 5

> rev(xSnippet)
integer-Rle of length 10 with 9 runs
   Lengths: 2 1 1 1 1 1 1 1 1
   Values : 5 7 6 2 6 4 5 6 4
```
> rep(xSnippet, 2)
integer-Rle of length 20 with 18 runs
  Lengths: 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2
  Values : 4 6 5 4 6 2 6 7 5 4 6 5 4 6 2 6 7 5
> subset(xSnippet, xSnippet >= 5L)
integer-Rle of length 7 with 5 runs
  Lengths: 1 1 2 1 2
  Values : 6 5 6 7 5

2.1.2 Combining Vectors

The IRanges package uses two generic functions, `c` and `append`, for combining two Vector objects. The methods for Vector objects follow the definition that these two functions are given the the base package.

> c(xSnippet, rev(xSnippet))
integer-Rle of length 20 with 17 runs
  Lengths: 1 1 1 1 1 1 1 1 4 1 1 1 1 1 1 1
  Values : 4 6 5 4 6 2 6 7 5 7 6 2 6 4 5 6 4
> append(xSnippet, xSnippet, after = 3)
integer-Rle of length 20 with 18 runs
  Lengths: 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2
  Values : 4 6 5 4 6 5 4 6 2 6 7 5 4 6 2 6 7 5

2.1.3 Looping over Vectors and Vector subsets

In R, for looping can be an expensive operation. To compensate for this, IRanges uses three generics, `endoapply`, `lapply`, and `sapply`, for looping over sequences and two generics, `aggregate` and `shiftApply`, to perform calculations over subsequences. The `lapply` and `sapply` functions are familiar to many R users since they are the standard functions for looping over the elements of an R list object. The `endoapply` function performs an endomorphism equivalent to `lapply`, i.e. returns a Vector object of the same class as the input rather than a list object. More will be given on these three functions in the Lists subsection.

The `aggregate` function combines sequence extraction functionality of the `window` function with looping capabilities of the `sapply` function. For example, here is some code to compute medians across a moving window of width 3 using the function `aggregate`:

> xSnippet
integer-Rle of length 10 with 9 runs
  Lengths: 1 1 1 1 1 1 1 1 2
  Values : 4 6 5 4 6 2 6 7 5
> aggregate(xSnippet, start = 1:8, width = 3, FUN = median)
[1] 5 5 5 4 6 6 5

The `shiftApply` function is a looping operation involving two sequences whose elements are lined up via a positional shift operation. For example, the elements of `xVector` and `yVector` were simulated from Poisson distributions with the mean of element i from `yVector` being equivalent to the mean of element i + 250 from `xVector`. If we did not know the size of the shift, we could estimate it by finding the shift that maximizes the correlation between `xVector` and `yVector`.
> cor(xVector, yVector)
[1] 0.5739224

> shifts <- seq(235, 265, by=3)
> corrs <- shiftApply(shifts, yVector, xVector, FUN = cor)

> plot(shifts, corrs)

The result is shown in Fig. 1.

2.1.4 Run Length Encoding

Up until this point we have used R atomic vectors to represent atomic sequences, but there are times when these object become too large to manage in memory. When there are lots of consecutive repeats in the sequence, the data can be compressed and managed in memory through a run-length encoding where a data value is paired with a run length. For example, the sequence \{1, 1, 1, 2, 3, 3\} can be represented as values = \{1, 2, 3\}, run lengths = \{3, 1, 2\}.

The \texttt{Rle} class in \texttt{IRanges} is used to represent a run-length encoded (compressed) sequence of \texttt{logical}, \texttt{integer}, \texttt{numeric}, \texttt{complex}, \texttt{character}, or \texttt{raw} values. One way to construct an \texttt{Rle} object is through the \texttt{Rle} constructor function:

> xRle <- Rle(xVector)
> yRle <- Rle(yVector)
> xRle

integer-Rle of length 10000000 with 1510219 runs

Lengths: 780 1 208 1 1599 1 \ldots 5 1 91 1 927
Values: 0 1 0 1 0 1 \ldots 0 1 0 1 0
When there are lots of consecutive repeats, the memory savings through an RLE can be quite dramatic. For example, the \texttt{xRle} object occupies less than one quarter of the space of the original \texttt{xVector} object, while storing the same information:

\begin{verbatim}
> as.vector(object.size(xRle) / object.size(xVector))
[1] 1
> identical(as.vector(xRle), xVector)
[1] FALSE
\end{verbatim}

The functions \texttt{runValue} and \texttt{runLength} extract the run values and run lengths from an \texttt{Rle} object respectively:

\begin{verbatim}
> head(runValue(xRle))
[1] 0 1 0 1 0 1
> head(runLength(xRle))
[1] 780 1 208 1 1599 1
\end{verbatim}

The \texttt{Rle} class supports many of the basic methods associated with R atomic vectors including the Ops, Math, Math2, Summary, and Complex group generics. Here is an example of manipulating \texttt{Rle} objects using methods from the Ops group:

\begin{verbatim}
> xRle > 0
logical-Rle of length 10000000 with 197127 runs
  Lengths: 780 1 208 1 1599 ... 1 91 1 927
  Values : FALSE TRUE FALSE TRUE FALSE ...
> xRle + yRle
integer-Rle of length 10000000 with 1957707 runs
  Lengths: 780 1 208 1 13 1 413 ... 5 1 91 1 507 1 419
  Values : 0 1 0 1 0 1 0 ... 0 1 0 1 0 1 0
> xRle > 0 | yRle > 0
logical-Rle of length 10000000 with 210711 runs
  Lengths: 780 1 208 1 13 ... 1 507 1 419
  Values : FALSE TRUE FALSE TRUE FALSE ...
\end{verbatim}

Here are some from the Summary group:

\begin{verbatim}
> range(xRle)
[1] 0 26
\end{verbatim}
And here is one from the Math group:

\[ \text{sum}(xRle > 0 \mid yRle > 0) \]

\[
\begin{array}{l}
[1] 2105185
\end{array}
\]

As with the atomic vectors, the \texttt{cor} and \texttt{shiftApply} functions operate on \textit{Rle} objects:

\[ \text{cor}(xRle, yRle) \]

\[
\begin{array}{l}
[1] 0.5739224
\end{array}
\]

\[ \text{shiftApply}(249:251, yRle, xRle, \text{FUN} = \text{function}(x, y) \text{ var}(x, y) / (\text{sd}(x) \ast \text{sd}(y))) \]

\[
\begin{array}{l}
[1] 0.8519138 0.8517324 0.8517725
\end{array}
\]

For more information on the methods supported by the \textit{Rle} class, consult the \textit{Rle} man page.

### 2.2 Lists

In many data analysis situation there is a desire to organize and manipulate multiple objects simultaneously. Typically this is done in \textit{R} through the usage of a list. While a list serves as a generic container, it does not confer any information about the specific class of its elements, provides no infrastructure to ensure type safety, and the S3 and S4 method dispatch mechanisms do not support method selection for lists with homogeneous object types. The \textit{List} virtual class defined in the \textit{IRanges} package addresses these issues. \textit{List} is a direct extension of \textit{Vector}.

#### 2.2.1 Lists of Atomic Vectors

The first type of lists we consider are those containing atomic sequences such as \textit{integer} vectors or \textit{Rle} objects. We may wish to define a method that retrieves the length of each atomic sequence element, without special type checking. To enable this, we define collection classes such as \textit{IntegerList} and \textit{RleList}, which inherit from the \textit{List} virtual class, for representing lists of \textit{integer} vectors and \textit{Rle} objects respectively.

\[ \text{getClassDef}("RleList") \]

Virtual Class "RleList" [package "IRanges"]

Slots:

Name: elementType elementMetadata metadata

Class: character DataTableORNULL list

Extends:

Class "AtomicList", directly
Class "List", by class "AtomicList", distance 2
Class "Vector", by class "AtomicList", distance 3
Class "Annotated", by class "AtomicList", distance 4

Known Subclasses: "RleViews", "CompressedRleList", "SimpleRleList"
As the class definition above shows, the \textit{RleList} class is virtual with subclasses \textit{SimpleRleList} and \textit{CompressedRleList}. A \textit{SimpleRleList} class uses a regular R list to store the underlying elements and the \textit{CompressedRleList} class stores the elements in an unlisted form and keeps track of where the element breaks are. The former “simple list” class is useful when the Rle elements are long and the latter “compressed list” class is useful when the list is long and/or sparse (i.e. a number of the list elements have length 0).

In fact, all of the atomic vector types (raw, logical, integer, numeric, complex, and character) have similar list classes that derive from the \textit{List} virtual class. For example, there is an \textit{IntegerList} virtual class with subclasses \textit{SimpleIntegerList} and \textit{CompressedIntegerList}.

Each of the list classes for atomic sequences, be they stored as vectors or \textit{Rle} objects, have a constructor function with a name of the appropriate list virtual class, such as \textit{IntegerList}, and an optional argument \texttt{compress} that takes an argument to specify whether or not to create the simple list object type or the compressed list object type. The default is to create the compressed list object type.

```r
> args(IntegerList)
function (..., compress = TRUE)
NULL

> cIntList1 <- IntegerList(x = xVector, y = yVector)
> cIntList1
IntegerList of length 2
[["x"]]
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[["y"]]
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

> sIntList2 <- IntegerList(x = xVector, y = yVector, compress = FALSE)
> sIntList2
IntegerList of length 2
[["x"]]
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[["y"]]
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

> ## sparse integer list
> xExploded <- lapply(xVector[1:5000], function(x) seq_len(x))
> cIntList2 <- IntegerList(xExploded)
> sIntList2 <- IntegerList(xExploded, compress = FALSE)
> object.size(cIntList2)
33080 bytes

> object.size(sIntList2)
253952 bytes
```

The \texttt{length} function returns the number of elements in a \textit{Vector}-derived object and, for a \textit{List}-derived object like “simple list” or “compressed list”, the \texttt{elementNROWS} function returns an integer vector containing the lengths of each of the elements:

```r
> length(cIntList2)
[1] 5000

> Rle(elementNROWS(cIntList2))
```
integer-Rle of length 5000 with 427 runs

Lengths: 780 1 208 1 1599 1 ... 1 1 1 1 1
Values : 0 1 0 1 0 1 ... 10 9 6 9 12

Just as with ordinary R list objects, List-derived object support the [[ for element extraction, c for combining, and lapply/sapply for looping. When looping over sparse lists, the “compressed list” classes can be much faster during computations since only the non-empty elements are looped over during the lapply/sapply computation and all the empty elements are assigned the appropriate value based on their status.

```
> system.time(sapply(xExploded, mean))
user  system elapsed
 0.044   0.000   0.044
> system.time(sapply(sIntList2, mean))
user  system elapsed
 0.044   0.000   0.045
> system.time(sapply(cIntList2, mean))
user  system elapsed
 0.048   0.000   0.048

> identical(sapply(xExploded, mean), sapply(sIntList2, mean))
[1] TRUE
> identical(sapply(xExploded, mean), sapply(cIntList2, mean))
[1] TRUE
```

Unlist ordinary R list objects, AtomicList objects support the Ops (e.g. +, ==, &), Math (e.g. log, sqrt), Math2 (e.g. round, signif), Summary (e.g. min, max, sum), and Complex (e.g. Re, Im) group generics.

```
> xRleList <- RleList(xRle, 2L * rev(xRle))
> yRleList <- RleList(yRle, 2L * rev(yRle))
> xRleList > 0

RleList of length 2
[[1]]
logical-Rle of length 10000000 with 197127 runs
Lengths: 780 1 208 1 1599 ... 1 91 1 927
Values : FALSE TRUE FALSE TRUE FALSE ... TRUE FALSE TRUE FALSE

[[2]]
logical-Rle of length 10000000 with 197127 runs
Lengths: 927 1 91 1 5 ... 1 208 1 780
Values : FALSE TRUE FALSE TRUE FALSE ... TRUE FALSE TRUE FALSE
```

```
> xRleList + yRleList
```
RleList of length 2
[[1]]
inRle of length 10000000 with 1957707 runs
   Lengths: 780 1 208 1 13 1 413 ... 5 1 91 1 507 1 419
   Values : 0 1 0 1 0 1 0 ... 0 1 0 1 0 1 0
[[2]]
inRle of length 10000000 with 1957707 runs
   Lengths: 419 1 507 1 91 1 5 ... 413 1 13 1 208 1 780
   Values : 0 2 0 2 0 2 0 ... 0 2 0 2 0 2 0

> sum(xRleList > 0 | yRleList > 0)
[1] 2105185 2105185

Since these atomic lists inherit from List, they can also use the looping function endoapply to perform endomorphisms.

> safe.max <- function(x) { if(length(x)) max(x) else integer(0) }
> endoapply(sIntList2, safe.max)

IntegerList of length 5000
[[1]] integer(0)
[[2]] integer(0)
[[3]] integer(0)
[[4]] integer(0)
[[5]] integer(0)
[[6]] integer(0)
[[7]] integer(0)
[[8]] integer(0)
[[9]] integer(0)
[[10]] integer(0)
...
<4990 more elements>

> endoapply(cIntList2, safe.max)

IntegerList of length 5000
[[1]] integer(0)
[[2]] integer(0)
[[3]] integer(0)
[[4]] integer(0)
[[5]] integer(0)
[[6]] integer(0)
[[7]] integer(0)
[[8]] integer(0)
[[9]] integer(0)
[[10]] integer(0)
...
<4990 more elements>

> endoapply(sIntList2, safe.max)[[1]]

integer(0)
3 Data Tables

To Do: **DataTable, DataFrame, DataFrameList, SplitDataFrameList**

4 Vector Annotations

Often when one has a collection of objects, there is a need to attach metadata that describes the collection in some way. Two kinds of metadata can be attached to a *Vector* object:

1. Metadata about the object as a whole: this metadata is accessed via the *metadata* accessor and is represented as an ordinary *list*;

2. Metadata about the individual elements of the object: this metadata is accessed via the *mcols* accessor (*mcols* stands for *metadata columns*) and is represented as a *DataTable* object (i.e. as an instance of a concrete subclass of *DataTable*, e.g. a *DataFrame* object). This *DataTable* object can be thought of as the result of binding together one or several vector-like objects (the metadata columns) of the same length as the *Vector* object. Each row of the *DataTable* object annotates the corresponding element of the *Vector* object.

5 Vector Ranges

When analyzing sequences, we are often interested in particular consecutive subsequences. For example, the a vector could be considered a sequence of lower-case letters, in alphabetical order. We would call the first five letters (*a* to *e*) a consecutive subsequence, while the subsequence containing only the vowels would not be consecutive. It is not uncommon for an analysis task to focus only on the geometry of the regions, while ignoring the underlying sequence values. A list of indices would be a simple way to select a subsequence. However, a sparser representation for consecutive subsequences would be a range, a pairing of a start position and a width, as used when extracting sequences with *window*.

When analyzing subsequences in *IRanges*, each range is treated as an observation. The virtual *Ranges* class represents lists of ranges, or, equivalently and as a derivative *IntegerList*, sequences of consecutive integers. The most commonly used implementation of *Ranges* is *IRanges*, which stores the starts and widths as ordinary integer vectors. To construct an *IRanges* instance, we call the *IRanges* constructor. Ranges are normally specified by passing two out of the three parameters: start, end and width (see *help(IRanges)* for more information).

```r
> ir1 <- IRanges(start = 1:10, width = 10:1)
> ir2 <- IRanges(start = 1:10, end = 11)
> ir3 <- IRanges(end = 11, width = 10:1)
> identical(ir1, ir2) & identical(ir2, ir3)

[1] FALSE
```

```r
> ir <- IRanges(c(1, 8, 14, 15, 19, 34, 40),
+   width = c(12, 6, 6, 15, 6, 2, 7))
```

All of the above calls construct an *IRanges* instance with the same ranges, using different combinations of the *start*, *end* and *width* parameters.

Accessing the starts, widths and ends is supported by every *Ranges* implementation.

```r
> start(ir)

[1] 1 8 14 15 19 34 40
```
For \texttt{IRanges} and some other \texttt{Ranges} derivatives, subsetting is also supported, by numeric and logical indices.

\begin{example}
\texttt{ir[1:4]}
\end{example}
\begin{verbatim}
IRanges object with 4 ranges and 0 metadata columns:

\begin{tabular}{lll}
\hline
\texttt{start} & \texttt{end} & \texttt{width} \\
\texttt{<integer>} & \texttt{<integer>} & \texttt{<integer>} \\
\hline
[1] & 1 & 12 & 12 \\
[2] & 8 & 13 & 6 \\
[3] & 14 & 19 & 6 \\
\hline
\end{tabular}
\end{verbatim}

\begin{example}
\texttt{ir[start(ir) <= 15]}
\end{example}
\begin{verbatim}
IRanges object with 4 ranges and 0 metadata columns:

\begin{tabular}{lll}
\hline
\texttt{start} & \texttt{end} & \texttt{width} \\
\texttt{<integer>} & \texttt{<integer>} & \texttt{<integer>} \\
\hline
[1] & 1 & 12 & 12 \\
[2] & 8 & 13 & 6 \\
[3] & 14 & 19 & 6 \\
\hline
\end{tabular}
\end{verbatim}

One may think of each range as a sequence of integer ranges, and \texttt{Ranges} is, in fact, derived from \texttt{IntegerList}.

\begin{example}
\texttt{ir[[1]]}
\end{example}
\begin{verbatim}
[1] 1 2 3 4 5 6 7 8 9 10 11 12
\end{verbatim}

In order to illustrate range operations, we'll create a function to plot ranges.

\begin{example}
\texttt{plotRanges <- function(x, xlim = x, main = deparse(substitute(x)),}
+ \texttt{col = "black", sep = 0.5, \ldots)}
+ \texttt{)}
+ \texttt{)}
\end{example}
\begin{verbatim}
+ \{ \texttt{height <- 1} \}
+ \texttt{if (is(xlim, "Ranges"))}
+ \texttt{xlim <- c(min(start(xlim)), max(end(xlim)))}
+ \texttt{bins <- disjointBins(IRanges(start(x), end(x) + 1))}
+ \texttt{plot.new()} \texttt{plot.window(xlim, c(0, max(bins)*(height + sep)))}
+ \texttt{ybottom <- bins * (sep + height) - height}
+ \texttt{rect(start(x)-0.5, ybottom, end(x)+0.5, ybottom + height, col = col, \ldots)}
+ \texttt{title(main)} \texttt{axis(1)} \texttt{\}}
\end{verbatim}

\begin{example}
\texttt{plotRanges(ir)}
\end{example}
5.1 Normality

Sometimes, it is necessary to formally represent a subsequence, where no elements are repeated and order is preserved. Also, it is occasionally useful to think of a Ranges object as a set, where no elements are repeated and order does not matter. While every Ranges object, as a Vector derivative, has an implicit ordering, one can enforce the same ordering for all such objects, so that ordering becomes inconsequential within that context.

The NormalIRanges class formally represents either a subsequence encoding or a set of integers. By definition a Ranges object is said to be normal when its ranges are: (a) not empty (i.e. they have a non-null width); (b) not overlapping; (c) ordered from left to right; (d) not even adjacent (i.e. there must be a non-empty gap between 2 consecutive ranges).

There are three main advantages of using a normal Ranges object: (1) it guarantees a subsequence encoding or set of integers, (2) it is compact in terms of the number of ranges, and (3) it uniquely identifies its information, which simplifies comparisons.

The reduce function reduces any Ranges object to a NormalIRanges by merging redundant ranges.

\[
\text{reduce}(\text{ir})
\]

IRanges object with 3 ranges and 0 metadata columns:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>46</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
\text{plotRanges(\text{reduce}(\text{ir}))}
\]
5.2 Lists of Ranges objects

It is common to manipulate collections of Ranges objects during an analysis. Thus, the IRanges package defines some specific classes for working with multiple Ranges objects.

The RangesList class asserts that each element is a Ranges object and provides convenience methods, such as start, end and width accessors that return IntegerList objects, aligning with the RangesList object. To explicitly construct a RangesList, use the RangesList function.

```r
> rl <- RangesList(ir, rev(ir))

> start(rl)

IntegerList of length 2
[[1]] 1 8 14 15 19 34 40
[[2]] 40 34 19 15 14 8 1
```

5.3 Vector Extraction

As the elements of a Ranges object encode consecutive subsequences, they may be used directly in sequence extraction. Note that when a normal Ranges is given as the index, the result is a subsequence, as no elements are repeated or reordered. If the sequence is a Vector subclass (i.e. not an ordinary vector), the canonical \[ \] function accepts a Ranges instance.

```r
> irextract <- IRanges(start = c(4501, 4901) , width = 100)
> xRle[irextract]

integer-Rle of length 200 with 159 runs
  Lengths: 12 1 1 1 2 1 1 1 1 2 ... 1 1 1 1 1 1 1 1 1
  Values : 0 1 0 2 0 1 0 1 0 1 ... 9 12 6 5 10 9 6 9 12
```

5.4 Finding Overlapping Ranges

The function findOverlaps detects overlaps between two Ranges objects.

```r
> ol <- findOverlaps(ir, reduce(ir))
> as.matrix(ol)

queryHits subjectHits
[1,] 1 1
[2,] 2 1
[3,] 3 1
[4,] 4 1
[5,] 5 1
[6,] 6 2
[7,] 7 3
```

5.5 Counting Overlapping Ranges

The function coverage counts the number of ranges over each position.

```r
> cov <- coverage(ir)
> plotRanges(ir)
> cov <- as.vector(cov)
> mat <- cbind(seq_along(cov)-0.5, cov)
```

13
> d <- diff(cov) != 0
> mat <- rbind(cbind(mat[d,1]+1, mat[d,2]), mat)
> mat <- mat[order(mat[,1]),]
> lines(mat, col="red", lwd=4)
> axis(2)

5.6 Finding Neighboring Ranges
The nearest function finds the nearest neighbor ranges (overlapping is zero distance). The precede and follow functions find the non-overlapping nearest neighbors on a specific side.

5.7 Transforming Ranges
Utilities are available for transforming a Ranges object in a variety of ways. Some transformations, like reduce introduced above, can be dramatic, while others are simple per-range adjustments of the starts, ends or widths.

5.7.1 Adjusting starts, ends and widths
Perhaps the simplest transformation is to adjust the start values by a scalar offset, as performed by the shift function. Below, we shift all ranges forward 10 positions.

> shift(ir, 10)

IRanges object with 7 ranges and 0 metadata columns:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>22</td>
<td>12</td>
</tr>
<tr>
<td>18</td>
<td>23</td>
<td>6</td>
</tr>
<tr>
<td>24</td>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>25</td>
<td>39</td>
<td>15</td>
</tr>
<tr>
<td>29</td>
<td>34</td>
<td>6</td>
</tr>
<tr>
<td>44</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td>50</td>
<td>56</td>
<td>7</td>
</tr>
</tbody>
</table>

There are several other ways to transform ranges. These include narrow, resize, flank, reflect, restrict, and threebands. For example narrow supports the adjustment of start, end and width values, which should be relative to each range. These adjustments are vectorized over the ranges. As its name suggests, the ranges can only be narrowed.
The `restrict` function ensures every range falls within a set of bounds. Ranges are contracted as necessary, and the ranges that fall completely outside of but not adjacent to the bounds are dropped, by default.

```r
> restrict(ir, start=2, end=3)
```

IRanges object with 1 range and 0 metadata columns:

```
start end width
[1] 2 3 2
```

The `threebands` function extends `narrow` so that the remaining left and right regions adjacent to the narrowed region are also returned in separate `Ranges` objects.

```r
> threebands(ir, start=1:5, width=2)
```

$left

IRanges object with 7 ranges and 0 metadata columns:

```
start end width
[1] 1 0 0
[2] 8 8 1
[3] 14 15 2
[4] 15 17 3
[5] 19 22 4
[6] 34 33 0
[7] 40 40 1
```

$middle

IRanges object with 7 ranges and 0 metadata columns:

```
start end width
[1] 1 2 2
[2] 9 10 2
[3] 16 17 2
[4] 18 19 2
[5] 23 24 2
[6] 34 35 2
[7] 41 42 2
```
The arithmetic operators +, -, and * change both the start and the end/width by symmetrically expanding or contracting each range. Adding or subtracting a numeric (integer) vector to a Ranges causes each range to be expanded or contracted on each side by the corresponding value in the numeric vector.

> ir + seq_len(length(ir))

The * operator symmetrically magnifies a Ranges object by a factor, where positive contracts (zooms in) and negative expands (zooms out).

> ir * -2 # double the width

WARNING: The semantic of these arithmetic operators might be revisited at some point. Please restrict their use to the context of interactive visualization (where they arguably provide some convenience) but avoid to use them programmatically.

5.7.2 Making ranges disjoint

A more complex type of operation is making a set of ranges disjoint, i.e. non-overlapping. For example, threebands returns a disjoint set of three ranges for each input range.

The disjoin function makes a Ranges object disjoint by fragmenting it into the widest ranges where the set of overlapping ranges is the same.
> disjoin(ir)

IRanges object with 10 ranges and 0 metadata columns:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>46</td>
<td>7</td>
</tr>
</tbody>
</table>

> plotRanges(disjoin(ir))

A variant of disjoin is disjointBins, which divides the ranges into bins, such that the ranges in each bin are disjoint. The return value is an integer vector of the bins.

> disjointBins(ir)

[1] 1 2 1 2 3 1 1

5.7.3 Other transformations

Other transformations include reflect and flank. The former “flips” each range within a set of common reference bounds.

> reflect(ir, IRanges(start(ir), width=width(ir)*2))

IRanges object with 7 ranges and 0 metadata columns:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>30</td>
<td>44</td>
<td>15</td>
</tr>
<tr>
<td>25</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>2</td>
</tr>
<tr>
<td>47</td>
<td>53</td>
<td>7</td>
</tr>
</tbody>
</table>
The `flank` returns ranges of a specified width that flank, to the left (default) or right, each input range. One use case of this is forming promoter regions for a set of genes.

```r
> flank(ir, width = seq_len(length(ir)))
```

IRanges object with 7 ranges and 0 metadata columns:

<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>6</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>33</td>
<td>6</td>
</tr>
<tr>
<td>33</td>
<td>39</td>
<td>7</td>
</tr>
</tbody>
</table>

### 5.8 Set Operations

Sometimes, it is useful to consider a `Ranges` object as a set of integers, although there is always an implicit ordering. This is formalized by `NormalIRanges`, above, and we now present versions of the traditional mathematical set operations `complement`, `union`, `intersect`, and `difference` for `Ranges` objects. There are two variants for each operation. The first treats each `Ranges` object as a set and returns a normal value, while the other has a “parallel” semantic like `pmin`/`pmax` and performs the operation for each range pairing separately.

The `complement` operation is implemented by the `gaps` and `pgap` functions. By default, `gaps` will return the ranges that fall between the ranges in the (normalized) input. It is possible to specify a set of bounds, so that flanking ranges are included.

```r
> gaps(ir, start=1, end=50)
```

IRanges object with 3 ranges and 0 metadata columns:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>36</td>
<td>39</td>
<td>4</td>
</tr>
<tr>
<td>47</td>
<td>50</td>
<td>4</td>
</tr>
</tbody>
</table>

```r
> plotRanges(gaps(ir, start=1, end=50), c(1,50))
```

`pgap` considers each parallel pairing between two `Ranges` objects and finds the range, if any, between them. Note that the function name is singular, suggesting that only one range is returned per range in the input.
The remaining operations, union, intersect and difference are implemented by the `union`, `intersect` and `setdiff` functions, respectively. These are relatively self-explanatory.

6 Vector Views

The IRanges package provides the virtual Views class, which stores a sequence together with an IRanges object defining ranges on the sequence. Each range is said to represent a view onto the sequence.

Here, we will demonstrate the RleViews class, where the sequence is of class Rle. Other Views implementations exist, such as XStringViews in the Biostrings package.

6.1 Creating Views

There are two basic constructors for creating views: the `Views` function based on indicators and the `slice` based on numeric boundaries.

```r
> xViews <- Views(xRle, xRle >= 1)
> xViews <- slice(xRle, 1)
> xViewsList <- slice(xRleList, 1)
```

6.2 Aggregating Views

While `sapply` can be used to loop over each window, the native functions `viewMaxs`, `viewMins`, `viewSums`, and `viewMeans` provide fast looping to calculate their respective statistical summaries.

```r
> head(viewSums(xViews))
[1] 1 1 1 1 1 2
> viewSums(xViewsList)

IntegerList of length 2
[[1]] 1 1 1 1 2 1 2 3 1 6 1 3 4 ... 12 6 37 10 8 11 6 4 5 1 1 5 1
[[2]] 2 2 10 2 2 10 8 12 22 16 20 74 12 ... 2 12 2 6 4 2 2 4 2 2 2 2 2
> head(viewMaxs(xViews))
[1] 1 1 1 1 1 2
> viewMaxs(xViewsList)

IntegerList of length 2
[[1]] 1 1 1 1 2 1 2 1 2 1 2 3 1 ... 3 5 2 5 6 2 8 3 2 2 1 1 2 1
[[2]] 2 2 4 2 4 2 4 6 16 4 12 10 4 10 6 ... 4 2 4 2 4 2 2 4 2 2 2 2 2
```

7 IRanges in Biological Sequence Analysis

The IRanges packages was primarily designed with biological sequence analysis in mind and Table I shows how some biological sequence analysis concepts are represented in the IRanges class system.
Table 1: *Vector* subclasses for Biological Sequence Analysis

<table>
<thead>
<tr>
<th>Biological Entity</th>
<th>Vector Subclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome browser track(s)</td>
<td>GRanges/GRangesList</td>
</tr>
<tr>
<td>Coverage across chromosomes/contigs</td>
<td>RleList</td>
</tr>
<tr>
<td>Mapped ranges to genome</td>
<td>CompressedIRangesList</td>
</tr>
<tr>
<td>Data (sans ranges) across chromosomes/contigs</td>
<td>SplitDataFrameList</td>
</tr>
</tbody>
</table>

> `toLatex(sessionInfo())`

- R version 3.3.3 (2017-03-06), x86_64-pc-linux-gnu
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=C`,
  `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`,
  `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: BiocGenerics 0.20.0, IRanges 2.8.2, S4Vectors 0.12.2
- Loaded via a namespace (and not attached): tools 3.3.3

Table 2: The output of `sessionInfo` on the build system after running this vignette.

8 Session Information