An Overview of the *S4Vectors* package

Patrick Aboyoun, Michael Lawrence, Hervé Pagès

Edited: February 2018; Compiled: December 2, 2020

Contents

1 Introduction

The *S4Vectors* package provides a framework for representing vector-like and list-like objects as S4 objects. It defines two central virtual classes, *Vector* and *List*, and a set of generic functions that extend the semantic of ordinary vectors and lists in *R*. Package developers can easily implement vector-like or list-like objects as *Vector* and/or *List* derivatives. A few low-level *Vector* and *List* derivatives are implemented in the *S4Vectors* package itself e.g. *Hits*, *Rle*, and *DataFrame*). Many more are implemented in the *IRanges* and *GenomicRanges* infrastructure packages, and in many other Bioconductor packages.

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 *S4Vectors* to a particular problem domain will provide vignettes with relevant, realistic examples.

The *S4Vectors* package is available at bioconductor.org and can be downloaded via `BiocManager::install`:

```
> if (!require("BiocManager"))
+     install.packages("BiocManager")
> BiocManager::install("S4Vectors")
```

```
> library(S4Vectors)
```

2 Vector-like and list-like objects

In the context of the *S4Vectors* package, a vector-like object is an ordered finite collection of elements. All vector-like objects have three main properties: (1) a notion of length or number of elements, (2) the ability to extract elements to create new vector-like objects, and (3) the ability to be concatenated with one or more vector-like objects to form larger vector-like objects. The main functions for these three operations are `length`, `[`, and `c`. Supporting these operations provide a great deal of power and many vector-like object manipulations can be constructed using them.

Some vector-like objects can also have a list-like semantic, which means that individual elements can be extracted with `[[`.

In *S4Vectors* and many other Bioconductor packages, vector-like and list-like objects derive from the *Vector* and *List* virtual classes, respectively. Note that *List* is a subclass of *Vector*.

The following subsections describe each in turn.

2.1 Vector-like objects

As a first example of vector-like objects, we'll look at *Rle* objects. In *R*, atomic sequences are typically stored in atomic vectors. 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 *Rle* class defined in the *S4Vectors* package is used to represent a run-length encoded (compressed) sequence of *logical*, *integer*, *numeric*, *complex*, *character*, *raw*, or *factor* values. Note that the *Rle* class extends the *Vector* virtual class:

```
> showClass("Rle")

Class "Rle" [package "S4Vectors"]

Slots:

Name:             values           lengths   elementMetadata
Class:  vector_OR_factor  integer_OR_LLint DataFrame_OR_NULL

Name:           metadata
Class:              list

Extends:
Class "Vector", directly
Class "Annotated", by class "Vector", distance 2
Class "vector_OR_Vector", by class "Vector", distance 2
```

One way to construct *Rle* objects is through the *Rle* constructor function:

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

Rle objects are vector-like objects:

```
> length(xRle)

[1] 10000000

> xRle[1]

integer-Rle of length 1 with 1 run
  Lengths: 1
  Values : 0

> zRle <- c(xRle, yRle)
```

2.1.1 Subsetting a vector-like object

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 *S4Vectors* 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 *Rle* vector:

```
> xSnippet <- window(xRle, 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 Concatenating vector-like objects

The *S4Vectors* package uses two generic functions, `c` and `append`, for concatenating two *Vector* derivatives. The methods for *Vector* objects follow the definition that these two functions are given 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 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 subsequences of vector-like objects

In *R*, `for` looping can be an expensive operation. To compensate for this, the *S4Vectors* package provides `aggregate` and `shiftApply` methods (`shiftApply` is a new generic function defined in *S4Vectors*) to perform calculations over subsequences of vector-like objects.

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)
```

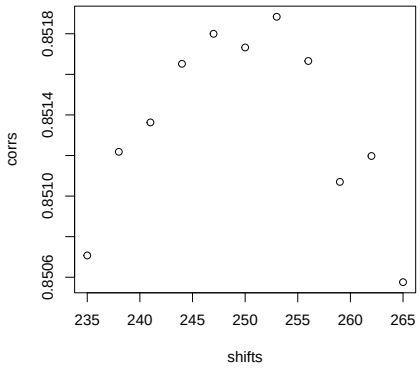

Figure 1: Correlation between `xRle` and `yRle` for various shifts

```
[1] 5 5 5 4 6 6 6 5
```

The `shiftApply` function is a looping operation involving two vector-like objects whose elements are lined up via a positional shift operation. For example, the elements of `xRle` and `yRle` were simulated from Poisson distributions with the mean of element i from `yRle` being equivalent to the mean of element i + 250 from `xRle`. If we did not know the size of the shift, we could estimate it by finding the shift that maximizes the correlation between `xRle` and `yRle`.

```
> cor(xRle, yRle)

[1] 0.5739224

> shifts <- seq(235, 265, by=3)
> corrs  <- shiftApply(shifts, yRle, xRle, FUN=cor)
```

```
> plot(shifts, corrs)
```

The result is shown in Fig. 1.

2.1.4 More on *Rle* objects

When there are lots of consecutive repeats, the memory savings through an RLE can be quite dramatic. For example, the `xRle` object occupies less than one third of the space of the original `xVector` object, while storing the same information:

```
> as.vector(object.size(xRle) / object.size(xVector))

[1] 0.3020726

> identical(as.vector(xRle), xVector)

[1] TRUE
```

The functions `runValue` and `runLength` extract the run values and run lengths from an *Rle* object respectively:

```
> head(runValue(xRle))

[1] 0 1 0 1 0 1

> head(runLength(xRle))

[1]  780    1  208    1 1599    1
```

The *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 a example of manipulating *Rle* objects using methods from the Ops group:

```
> 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 ...  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 ...  TRUE FALSE  TRUE FALSE
```

Here are some from the Summary group:

```
> range(xRle)

[1]  0 26

> sum(xRle > 0 | yRle > 0)

[1] 2105185
```

And here is one from the Math group:

```
> log1p(xRle)

numeric-Rle of length 10000000 with 1510219 runs
  Lengths:      780        1      208 ...       91        1      927
  Values : 0.000000 0.693147 0.000000 ... 0.000000 0.693147 0.000000
```

As with atomic vectors, the `cor` and `shiftApply` functions operate on *Rle* objects:

```
> cor(xRle, yRle)

[1] 0.5739224

> shiftApply(249:251, yRle, xRle,
+            FUN=function(x, y) {var(x, y) / (sd(x) * sd(y))})

[1] 0.8519138 0.8517324 0.8517725
```

For more information on the methods supported by the *Rle* class, consult the `Rle` man page.

2.2 List-like objects

Just as with ordinary *R list* objects, *List*-derived objects support `[[` for element extraction, `c` for concatenating, and `lapply`/`sapply` for looping. `lapply` and `sapply` are familiar to many *R* users since they are the standard functions for looping over the elements of an *R list* object.

In addition, the *S4Vectors* package introduces the `endoapply` function to perform an endomorphism equivalent to `lapply`, i.e. it returns a *List* derivative of the same class as the input rather than a *list* object.

An example of *List* derivative is the *DataFrame* class:

```
> showClass("DataFrame")

Class "DataFrame" [package "S4Vectors"]

Slots:

Name:           rownames             nrows          listData
Class: character_OR_NULL           integer              list

Name:        elementType   elementMetadata          metadata
Class:         character DataFrame_OR_NULL              list

Extends:
Class "RectangularData", directly
Class "SimpleList", directly
Class "DataFrame_OR_NULL", directly
Class "List", by class "SimpleList", distance 2
Class "Vector", by class "SimpleList", distance 3
Class "list_OR_List", by class "SimpleList", distance 3
Class "Annotated", by class "SimpleList", distance 4
Class "vector_OR_Vector", by class "SimpleList", distance 4

Known Subclasses:
Class "DFrame", directly, with explicit coerce
```

One way to construct *DataFrame* objects is through the *DataFrame* constructor function:

```
> df <- DataFrame(x=xRle, y=yRle)
> sapply(df, class)

    x     y
"Rle" "Rle"

> sapply(df, summary)

                x          y
Min.     0.0000000  0.0000000
1st Qu.  0.0000000  0.0000000
Median   0.0000000  0.0000000
Mean     0.9090338  0.9096009
3rd Qu.  0.0000000  0.0000000
Max.    26.0000000 27.0000000

> sapply(as.data.frame(df), summary)
```

```
                  x          y
Min.      0.0000000  0.0000000
1st Qu.   0.0000000  0.0000000
Median    0.0000000  0.0000000
Mean      0.9090338  0.9096009
3rd Qu.   0.0000000  0.0000000
Max.     26.0000000 27.0000000
```

```
> endoapply(df, `+`, 0.5)
```

```
DataFrame with 10000000 rows and 2 columns
             x     y
         <Rle> <Rle>
1          0.5   0.5
2          0.5   0.5
3          0.5   0.5
4          0.5   0.5
5          0.5   0.5
...        ...   ...
9999996    0.5   0.5
9999997    0.5   0.5
9999998    0.5   0.5
9999999    0.5   0.5
10000000   0.5   0.5
```

For more information on *DataFrame* objects, consult the `DataFrame` man page.

See the "An Overview of the *IRanges* package" vignette in the *IRanges* package for many more examples of *List* derivatives.

3 DataFrame and DataFrameList objects

TODO

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 *DataFrame* object. This *DataFrame* 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 *DataFrame* object annotates the corresponding element of the *Vector* object.

5 Session Information

Here is the output of `sessionInfo()` on the system on which this document was compiled:

```
R Under development (unstable) (2020-11-14 r79432)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.1 LTS

Matrix products: default
BLAS:   /home/biocbuild/bbs-3.13-bioc/R/lib/libRblas.so
LAPACK: /home/biocbuild/bbs-3.13-bioc/R/lib/libRlapack.so

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats4    parallel  stats     graphics  grDevices utils
[7] datasets  methods   base

other attached packages:
 [1] graph_1.69.0                ShortRead_1.49.0
 [3] GenomicAlignments_1.27.1    SummarizedExperiment_1.21.0
 [5] Biobase_2.51.0              MatrixGenerics_1.3.0
 [7] matrixStats_0.57.0          Rsamtools_2.7.0
 [9] GenomicRanges_1.43.0        GenomeInfoDb_1.27.1
[11] Biostrings_2.59.0           XVector_0.31.0
[13] BiocParallel_1.25.1         IRanges_2.25.4
[15] S4Vectors_0.29.5            BiocGenerics_0.37.0
[17] Matrix_1.2-18

loaded via a namespace (and not attached):
 [1] compiler_4.1.0         RColorBrewer_1.1-2
 [3] BiocManager_1.30.10    bitops_1.0-6
 [5] tools_4.1.0            zlibbioc_1.37.0
 [7] digest_0.6.27          evaluate_0.14
 [9] lattice_0.20-41        png_0.1-7
[11] rlang_0.4.9            DelayedArray_0.17.4
[13] yaml_2.2.1             xfun_0.19
[15] GenomeInfoDbData_1.2.4 hwriter_1.3.2
[17] knitr_1.30             grid_4.1.0
[19] jpeg_0.1-8.1           rmarkdown_2.5
[21] latticeExtra_0.6-29    htmltools_0.5.0
[23] BiocStyle_2.19.1       RCurl_1.98-1.2
[25] crayon_1.3.4
```