

An Introduction to the GenomicRanges Package

Marc Carlson Patrick Aboyoun Hervé Pagès Martin Morgan

October 3, 2017; updated 16 November, 2016

Contents

1	Introduction	1
2	GRanges: Genomic Ranges	2
2.1	Splitting and combining <i>GRanges</i> objects	4
2.2	Subsetting <i>GRanges</i> objects	4
2.3	Basic interval operations for <i>GRanges</i> objects	7
2.4	Interval set operations for <i>GRanges</i> objects	9
3	GRangesList: Groups of Genomic Ranges	11
3.1	Basic <i>GRangesList</i> accessors	12
3.2	Combining <i>GRangesList</i> objects	13
3.3	Basic interval operations for <i>GRangesList</i> objects	15
3.4	Subsetting <i>GRangesList</i> objects	16
3.5	Looping over <i>GRangesList</i> objects	18
4	Interval overlaps involving GRanges and GRangesList objects	21
5	Session Information	21

1 Introduction

The *GenomicRanges* package serves as the foundation for representing genomic locations within the *Bioconductor* project. In the *Bioconductor* package hierarchy, it builds upon the *IRanges* (infrastructure) package and provides support for the *BSgenome* (infrastructure), *Rsamtools* (I/O), *ShortRead* (I/O & QA), *rtracklayer* (I/O), *GenomicFeatures* (infrastructure), *GenomicAlignments* (sequence reads), *VariantAnnotation* (called variants), and many other *Bioconductor* packages.

This package lays a foundation for genomic analysis by introducing three classes (*GRanges*, *GPos*, and *GRangesList*), which are used to represent genomic ranges, genomic positions, and groups of genomic ranges. This vignette focuses on the *GRanges* and *GRangesList* classes and their associated methods.

The *GenomicRanges* package is available at <https://bioconductor.org> and can be installed via *biocLite*:

```
> source("https://bioconductor.org/biocLite.R")
> biocLite("GenomicRanges")
```

A package only needs to be installed once. Load the package into an *R* session with

```
> library(GenomicRanges)
```

2 GRanges: Genomic Ranges

The *GRanges* class represents a collection of genomic ranges that each have a single start and end location on the genome. It can be used to store the location of genomic features such as contiguous binding sites, transcripts, and exons. These objects can be created by using the *GRanges* constructor function. For example,

```
> gr <- GRanges(
+   seqnames = Rle(c("chr1", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)),
+   ranges = IRanges(101:110, end = 111:120, names = head(letters, 10)),
+   strand = Rle(strand(c("-", "+", "*", "+", "-")), c(1, 2, 2, 3, 2)),
+   score = 1:10,
+   GC = seq(1, 0, length=10))
> gr
```

GRanges object with 10 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
a	chr1	[101, 111]	-	1	1
b	chr2	[102, 112]	+	2	0.888888888888889
c	chr2	[103, 113]	+	3	0.777777777777778
.
h	chr3	[108, 118]	+	8	0.222222222222222
i	chr3	[109, 119]	-	9	0.111111111111111
j	chr3	[110, 120]	-	10	0

seqinfo: 3 sequences from an unspecified genome; no seqlengths

```
> options(warn=2)
```

creates a *GRanges* object with 10 genomic ranges. The output of the *GRanges* show method separates the information into a left and right hand region that are separated by | symbols. The genomic coordinates (seqnames, ranges, and strand) are located on the left-hand side and the metadata columns (annotation) are located on the right. For this example, the metadata is comprised of score and GC information, but almost anything can be stored in the metadata portion of a *GRanges* object.

The components of the genomic coordinates within a *GRanges* object can be extracted using the *seqnames*, *ranges*, and *strand* accessor functions.

```
> 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

```
> ranges(gr)
```

IRanges object with 10 ranges and 0 metadata columns:

	start	end	width
	<integer>	<integer>	<integer>
a	101	111	11
b	102	112	11
c	103	113	11
.
h	108	118	11
i	109	119	11
j	110	120	11

```
> strand(gr)
```

```
factor-Rle of length 10 with 5 runs
Lengths: 1 2 2 3 2
Values : - + * + -
Levels(3): + - *
```

The genomic ranges can be extracted without corresponding metadata with `granges`

```
> granges(gr)
```

GRanges object with 10 ranges and 0 metadata columns:

	seqnames	ranges	strand
	<Rle>	<IRanges>	<Rle>
a	chr1	[101, 111]	-
b	chr2	[102, 112]	+
c	chr2	[103, 113]	+
.
h	chr3	[108, 118]	+
i	chr3	[109, 119]	-
j	chr3	[110, 120]	-

```
-----
seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

Annotations for these coordinates can be extracted as a *DataFrame* object using the `mcols` accessor.

```
> mcols(gr)
```

DataFrame with 10 rows and 2 columns

	score	GC
	<integer>	<numeric>
1	1	1.0000000
2	2	0.8888889
3	3	0.7777778
...
8	8	0.2222222
9	9	0.1111111
10	10	0.0000000

```
> mcols(gr)$score
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

Information about the lengths of the various sequences that the ranges are aligned to can also be stored in the *GRanges* object. So if this is data from *Homo sapiens*, we can set the values as:

```
> seqlengths(gr) <- c(249250621, 243199373, 198022430)
```

And then retrieves as:

```
> seqlengths(gr)
```

chr1	chr2	chr3
249250621	243199373	198022430

Methods for accessing the `length` and `names` have also been defined.

```
> names(gr)
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
> length(gr)
```

```
[1] 10
```

2.1 Splitting and combining GRanges objects

GRanges objects can be divided into groups using the `split` method. This produces a *GRangesList* object, a class that will be discussed in detail in the next section.

```
> sp <- split(gr, rep(1:2, each=5))
```

```
> sp
```

GRangesList object of length 2:

\$1

GRanges object with 5 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
a	chr1	[101, 111]	-	1	1
b	chr2	[102, 112]	+	2	0.888888888888889
c	chr2	[103, 113]	+	3	0.777777777777778
d	chr2	[104, 114]	*	4	0.666666666666667
e	chr1	[105, 115]	*	5	0.555555555555556

\$2

GRanges object with 5 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
f	chr1	[106, 116]	+	6	0.444444444444444
g	chr3	[107, 117]	+	7	0.333333333333333
h	chr3	[108, 118]	+	8	0.222222222222222
i	chr3	[109, 119]	-	9	0.111111111111111
j	chr3	[110, 120]	-	10	0

seqinfo: 3 sequences from an unspecified genome

Separate *GRanges* instances can be concatenated by using the `c` and `append` methods.

```
> c(sp[[1]], sp[[2]])
```

GRanges object with 10 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
a	chr1	[101, 111]	-	1	1
b	chr2	[102, 112]	+	2	0.888888888888889
c	chr2	[103, 113]	+	3	0.777777777777778
.
h	chr3	[108, 118]	+	8	0.222222222222222
i	chr3	[109, 119]	-	9	0.111111111111111
j	chr3	[110, 120]	-	10	0

seqinfo: 3 sequences from an unspecified genome

2.2 Subsetting GRanges objects

GRanges objects act like vectors of ranges, with the expected vector-like subsetting operations available

```
> gr[2:3]
```

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>

```

b      chr2 [102, 112]      + |      2 0.888888888888889
c      chr2 [103, 113]      + |      3 0.777777777777778
-----

```

```
seqinfo: 3 sequences from an unspecified genome
```

A second argument to the `[]` subset operator can be used to specify which metadata columns to extract from the *GRanges* object. For example,

```
> gr[2:3, "GC"]
```

GRanges object with 2 ranges and 1 metadata column:

```

seqnames      ranges strand |      GC
   <Rle>   <IRanges>  <Rle> |   <numeric>
b      chr2 [102, 112]      + | 0.888888888888889
c      chr2 [103, 113]      + | 0.777777777777778
-----

```

```
seqinfo: 3 sequences from an unspecified genome
```

Elements can also be assigned to the *GRanges* object. Here is an example where the second row of a *GRanges* object is replaced with the first row of *gr*.

```

> singles <- split(gr, names(gr))
> grMod <- gr
> grMod[2] <- singles[[1]]
> head(grMod, n=3)

```

GRanges object with 3 ranges and 2 metadata columns:

```

seqnames      ranges strand |      score      GC
   <Rle>   <IRanges>  <Rle> | <integer>   <numeric>
a      chr1 [101, 111]      - |          1          1
b      chr1 [101, 111]      - |          1          1
c      chr2 [103, 113]      + |          3 0.777777777777778
-----

```

```
seqinfo: 3 sequences from an unspecified genome
```

Here is a second example where the metadata for score from the third element is replaced with the score from the second row etc.

```

> grMod[2,1] <- singles[[3]][,1]
> head(grMod, n=3)

```

GRanges object with 3 ranges and 2 metadata columns:

```

seqnames      ranges strand |      score      GC
   <Rle>   <IRanges>  <Rle> | <integer>   <numeric>
a      chr1 [101, 111]      - |          1          1
b      chr2 [103, 113]      + |          3          1
c      chr2 [103, 113]      + |          3 0.777777777777778
-----

```

```
seqinfo: 3 sequences from an unspecified genome
```

There are methods to repeat, reverse, or select specific portions of *GRanges* objects.

```
> rep(singles[[2]], times = 3)
```

GRanges object with 3 ranges and 2 metadata columns:

```

seqnames      ranges strand |      score      GC
   <Rle>   <IRanges>  <Rle> | <integer>   <numeric>
b      chr2 [102, 112]      + |          2 0.888888888888889
b      chr2 [102, 112]      + |          2 0.888888888888889
b      chr2 [102, 112]      + |          2 0.888888888888889

```

```

-----
seqinfo: 3 sequences from an unspecified genome
> rev(gr)
GRanges object with 10 ranges and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle> <IRanges> <Rle> | <integer> <numeric>
j      chr3 [110, 120]      - |        10        0
i      chr3 [109, 119]      - |         9 0.111111111111111
h      chr3 [108, 118]      + |         8 0.222222222222222
.      ...      ...      ... |      ...      ...
c      chr2 [103, 113]      + |         3 0.777777777777778
b      chr2 [102, 112]      + |         2 0.888888888888889
a      chr1 [101, 111]      - |         1        1
-----
seqinfo: 3 sequences from an unspecified genome
> head(gr,n=2)
GRanges object with 2 ranges and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle> <IRanges> <Rle> | <integer> <numeric>
a      chr1 [101, 111]      - |         1        1
b      chr2 [102, 112]      + |         2 0.888888888888889
-----
seqinfo: 3 sequences from an unspecified genome
> tail(gr,n=2)
GRanges object with 2 ranges and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle> <IRanges> <Rle> | <integer> <numeric>
i      chr3 [109, 119]      - |         9 0.111111111111111
j      chr3 [110, 120]      - |        10        0
-----
seqinfo: 3 sequences from an unspecified genome
> window(gr, start=2,end=4)
GRanges object with 3 ranges and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle> <IRanges> <Rle> | <integer> <numeric>
b      chr2 [102, 112]      + |         2 0.888888888888889
c      chr2 [103, 113]      + |         3 0.777777777777778
d      chr2 [104, 114]      * |         4 0.666666666666667
-----
seqinfo: 3 sequences from an unspecified genome
> gr[IRanges(start=c(2,7), end=c(3,9))]
GRanges object with 5 ranges and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle> <IRanges> <Rle> | <integer> <numeric>
b      chr2 [102, 112]      + |         2 0.888888888888889
c      chr2 [103, 113]      + |         3 0.777777777777778
g      chr3 [107, 117]      + |         7 0.333333333333333
h      chr3 [108, 118]      + |         8 0.222222222222222
i      chr3 [109, 119]      - |         9 0.111111111111111

```

```
-----
seqinfo: 3 sequences from an unspecified genome
```

2.3 Basic interval operations for GRanges objects

Basic interval characteristics of *GRanges* objects can be extracted using the `start`, `end`, `width`, and `range` methods.

```
> g <- gr[1:3]
> g <- append(g, singles[[10]])
> start(g)
```

```
[1] 101 102 103 110
```

```
> end(g)
```

```
[1] 111 112 113 120
```

```
> width(g)
```

```
[1] 11 11 11 11
```

```
> range(g)
```

GRanges object with 3 ranges and 0 metadata columns:

	seqnames	ranges	strand
	<Rle>	<IRanges>	<Rle>
[1]	chr1	[101, 111]	-
[2]	chr2	[102, 113]	+
[3]	chr3	[110, 120]	-

```
-----
seqinfo: 3 sequences from an unspecified genome
```

The *GRanges* class also has many methods for manipulating the ranges. The methods can be classified as *intra-range methods*, *inter-range methods*, and *between-range methods*.

Intra-range methods operate on each element of a *GRanges* object independent of the other ranges in the object. For example, the `flank` method can be used to recover regions flanking the set of ranges represented by the *GRanges* object. So to get a *GRanges* object containing the ranges that include the 10 bases upstream of the ranges:

```
> flank(g, 10)
```

GRanges object with 4 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
a	chr1	[112, 121]	-	1	1
b	chr2	[92, 101]	+	2 0.888888888888889	
c	chr2	[93, 102]	+	3 0.777777777777778	
j	chr3	[121, 130]	-	10	0

```
-----
seqinfo: 3 sequences from an unspecified genome
```

And to include the downstream bases:

```
> flank(g, 10, start=FALSE)
```

GRanges object with 4 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
a	chr1	[91, 100]	-	1	1
b	chr2	[113, 122]	+	2 0.888888888888889	
c	chr2	[114, 123]	+	3 0.777777777777778	

```
j      chr3 [100, 109]      - |      10      0
-----
seqinfo: 3 sequences from an unspecified genome
```

Other examples of intra-range methods include `resize` and `shift`. The `shift` method will move the ranges by a specific number of base pairs, and the `resize` method will extend the ranges by a specified width.

```
> shift(g, 5)
```

GRanges object with 4 ranges and 2 metadata columns:

```
  seqnames      ranges strand |      score      GC
    <Rle> <IRanges> <Rle> | <integer> <numeric>
a      chr1 [106, 116]      - |          1          1
b      chr2 [107, 117]      + |          2 0.888888888888889
c      chr2 [108, 118]      + |          3 0.777777777777778
j      chr3 [115, 125]      - |         10          0
-----
seqinfo: 3 sequences from an unspecified genome
```

```
> resize(g, 30)
```

GRanges object with 4 ranges and 2 metadata columns:

```
  seqnames      ranges strand |      score      GC
    <Rle> <IRanges> <Rle> | <integer> <numeric>
a      chr1 [ 82, 111]      - |          1          1
b      chr2 [102, 131]      + |          2 0.888888888888889
c      chr2 [103, 132]      + |          3 0.777777777777778
j      chr3 [ 91, 120]      - |         10          0
-----
seqinfo: 3 sequences from an unspecified genome
```

The [GenomicRanges](#) help page `?intra-range-methods` summarizes these methods.

Inter-range methods involve comparisons between ranges in a single *GRanges* object. For instance, the `reduce` method will align the ranges and merge overlapping ranges to produce a simplified set.

```
> reduce(g)
```

GRanges object with 3 ranges and 0 metadata columns:

```
  seqnames      ranges strand
    <Rle> <IRanges> <Rle>
[1]   chr1 [101, 111]      -
[2]   chr2 [102, 113]      +
[3]   chr3 [110, 120]      -
-----
seqinfo: 3 sequences from an unspecified genome
```

Sometimes one is interested in the gaps or the qualities of the gaps between the ranges represented by your *GRanges* object. The `gaps` method provides this information: reduced version of your ranges:

```
> gaps(g)
```

GRanges object with 12 ranges and 0 metadata columns:

```
  seqnames      ranges strand
    <Rle> <IRanges> <Rle>
[1]   chr1 [  1, 249250621]      +
[2]   chr1 [  1,         100]      -
[3]   chr1 [112, 249250621]      -
...     ...           ...     ...
[10]  chr3 [  1,         109]      -
```

```
[11]      chr3 [121, 198022430]      -
[12]      chr3 [  1, 198022430]      *
```

```
-----
seqinfo: 3 sequences from an unspecified genome
```

The `disjoin` method represents a *GRanges* object as a collection of non-overlapping ranges:

```
> disjoin(g)
```

GRanges object with 5 ranges and 0 metadata columns:

```
      seqnames      ranges strand
      <Rle>    <IRanges>  <Rle>
[1]      chr1 [101, 111]      -
[2]      chr2 [102, 102]      +
[3]      chr2 [103, 112]      +
[4]      chr2 [113, 113]      +
[5]      chr3 [110, 120]      -
```

```
-----
seqinfo: 3 sequences from an unspecified genome
```

The `coverage` method quantifies the degree of overlap for all the ranges in a *GRanges* object.

```
> coverage(g)
```

RleList of length 3

\$chr1

integer-Rle of length 249250621 with 3 runs

```
Lengths:      100      11 249250510
Values :         0         1         0
```

\$chr2

integer-Rle of length 243199373 with 5 runs

```
Lengths:      101         1      10         1 243199260
Values :         0         1         2         1         0
```

\$chr3

integer-Rle of length 198022430 with 3 runs

```
Lengths:      109      11 198022310
Values :         0         1         0
```

See the [GenomicRanges](#) help page `?"inter-range-methods"` for additional help.

Between-range methods involve operations between two *GRanges* objects; some of these are summarized in the next section.

2.4 Interval set operations for *GRanges* objects

Between-range methods calculate relationships between different *GRanges* objects. Of central importance are `findOverlaps` and related operations; these are discussed below. Additional operations treat *GRanges* as mathematical sets of coordinates; `union(g, g2)` is the union of the coordinates in `g` and `g2`. Here are examples for calculating the union, the intersect and the asymmetric difference (using `setdiff`).

```
> g2 <- head(gr, n=2)
```

```
> union(g, g2)
```

GRanges object with 3 ranges and 0 metadata columns:

```
      seqnames      ranges strand
      <Rle>    <IRanges>  <Rle>
```

```

[1] chr1 [101, 111] -
[2] chr2 [102, 113] +
[3] chr3 [110, 120] -
-----
seqinfo: 3 sequences from an unspecified genome
> intersect(g, g2)
GRanges object with 2 ranges and 0 metadata columns:
      seqnames      ranges strand
      <Rle> <IRanges> <Rle>
[1] chr1 [101, 111] -
[2] chr2 [102, 112] +
-----
seqinfo: 3 sequences from an unspecified genome
> setdiff(g, g2)
GRanges object with 2 ranges and 0 metadata columns:
      seqnames      ranges strand
      <Rle> <IRanges> <Rle>
[1] chr2 [113, 113] +
[2] chr3 [110, 120] -
-----
seqinfo: 3 sequences from an unspecified genome

```

Related methods are available when the structure of the *GRanges* objects are 'parallel' to one another, i.e., element 1 of object 1 is related to element 1 of object 2, and so on. These operations all begin with a *p*, which is short for parallel. The methods then perform element-wise, e.g., the union of element 1 of object 1 with element 1 of object 2, etc. A requirement for these operations is that the number of elements in each *GRanges* object is the same, and that both of the objects have the same *seqnames* and strand assignments throughout.

```

> g3 <- g[1:2]
> ranges(g3[1]) <- IRanges(start=105, end=112)
> punion(g2, g3)
GRanges object with 2 ranges and 0 metadata columns:
      seqnames      ranges strand
      <Rle> <IRanges> <Rle>
a chr1 [101, 112] -
b chr2 [102, 112] +
-----
seqinfo: 3 sequences from an unspecified genome
> pintersect(g2, g3)
GRanges object with 2 ranges and 3 metadata columns:
      seqnames      ranges strand |   score          GC      hit
      <Rle> <IRanges> <Rle> | <integer>      <numeric> <logical>
a chr1 [105, 111] - |      1          1      1
b chr2 [102, 112] + |      2 0.888888888888889      1
-----
seqinfo: 3 sequences from an unspecified genome
> psetdiff(g2, g3)
GRanges object with 2 ranges and 0 metadata columns:
      seqnames      ranges strand
      <Rle> <IRanges> <Rle>
a chr1 [101, 104] -

```

```

b      chr2 [102, 101]      +
-----
seqinfo: 3 sequences from an unspecified genome

```

For more information on the *GRanges* classes be sure to consult the manual page.

```
> ?GRanges
```

A relatively comprehensive list of available methods is discovered with

```
> methods(class="GRanges")
```

3 *GRangesList*: Groups of Genomic Ranges

Some important genomic features, such as spliced transcripts that are comprised of exons, are inherently compound structures. Such a feature makes much more sense when expressed as a compound object such as a *GRangesList*. Whenever genomic features consist of multiple ranges that are grouped by a parent feature, they can be represented as a *GRangesList* object. Consider the simple example of the two transcript *GRangesList* below created using the *GRangesList* constructor.

```

> gr1 <- GRanges(
+   seqnames = "chr2",
+   ranges = IRanges(103, 106),
+   strand = "+",
+   score = 5L, GC = 0.45)
> gr2 <- GRanges(
+   seqnames = c("chr1", "chr1"),
+   ranges = IRanges(c(107, 113), width = 3),
+   strand = c("+", "-"),
+   score = 3:4, GC = c(0.3, 0.5))
> grl <- GRangesList("txA" = gr1, "txB" = gr2)
> grl

```

GRangesList object of length 2:

\$txA

GRanges object with 1 range and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45

\$txB

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
[1]	chr1	[107, 109]	+	3	0.3
[2]	chr1	[113, 115]	-	4	0.5

```

-----
seqinfo: 2 sequences from an unspecified genome; no seqlengths

```

The `show` method for a *GRangesList* object displays it as a named list of *GRanges* objects, where the names of this list are considered to be the names of the grouping feature. In the example above, the groups of individual exon ranges are represented as separate *GRanges* objects which are further organized into a list structure where each element name is a transcript name. Many other combinations of grouped and labeled *GRanges* objects are possible of course, but this example is expected to be a common arrangement.

3.1 Basic GRangesList accessors

Just as with *GRanges* object, the components of the genomic coordinates within a *GRangesList* object can be extracted using simple accessor methods. Not surprisingly, the *GRangesList* objects have many of the same accessors as *GRanges* objects. The difference is that many of these methods return a list since the input is now essentially a list of *GRanges* objects. Here are a few examples:

```
> seqnames(grl)

RleList of length 2
$txA
factor-Rle of length 1 with 1 run
  Lengths: 1
  Values : chr2
Levels(2): chr2 chr1

$txB
factor-Rle of length 2 with 1 run
  Lengths: 2
  Values : chr1
Levels(2): chr2 chr1

> ranges(grl)

IRangesList of length 2
$txA
IRanges object with 1 range and 0 metadata columns:
      start      end      width
  <integer> <integer> <integer>
[1]      103      106         4

$txB
IRanges object with 2 ranges and 0 metadata columns:
      start      end      width
  <integer> <integer> <integer>
[1]      107      109         3
[2]      113      115         3

> strand(grl)

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

$txB
factor-Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : + -
Levels(3): + - *

The length and names methods will return the length or names of the list and the seqlengths method will return the
set of sequence lengths.

> length(grl)

[1] 2
```

```
> names(grl)
[1] "txA" "txB"
> seqlengths(grl)
chr2 chr1
  NA   NA
```

The `elementNROWS` method returns a list of integers corresponding to the result of calling `NROW` on each individual *GRanges* object contained by the *GRangesList*. This is a faster alternative to calling `lapply` on the *GRangesList*.

```
> elementNROWS(grl)
txA txB
  1   2
```

`isEmpty` tests if a *GRangesList* object contains anything.

```
> isEmpty(grl)
[1] FALSE
```

In the context of a *GRangesList* object, the `mcols` method performs a similar operation to what it does on a *GRanges* object. However, this metadata now refers to information at the list level instead of the level of the individual *GRanges* objects.

```
> mcols(grl) <- c("Transcript A", "Transcript B")
> mcols(grl)
```

```
DataFrame with 2 rows and 1 column
      value
<character>
1 Transcript A
2 Transcript B
```

Element-level metadata can be retrieved by unlisting the *GRangesList*, and extracting the metadata

```
> mcols(unlist(grl))
```

```
DataFrame with 3 rows and 2 columns
      score      GC
<integer> <numeric>
1         5    0.45
2         3    0.30
3         4    0.50
```

3.2 Combining *GRangesList* objects

GRangesList objects can be unlisted to combine the separate *GRanges* objects that they contain as an expanded *GRanges*.

```
> ul <- unlist(grl)
> ul
```

GRanges object with 3 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
txA	chr2	[103, 106]	+	5	0.45
txB	chr1	[107, 109]	+	3	0.3
txB	chr1	[113, 115]	-	4	0.5

seqinfo: 2 sequences from an unspecified genome; no seqlengths

Append lists using `append` or `c`.

A [support site user](#) had two *GRangesList* objects with 'parallel' elements, and wanted to combined these element-wise into a single *GRangesList*. One solution is to use `pc()` – parallel (element-wise) `c()`. A more general solution is to concatenate the lists and then re-group by some factor, in this case the names of the elements.

```
> grl1 <- GRangesList(
+   gr1 = GRanges("chr2", IRanges(3, 6)),
+   gr2 = GRanges("chr1", IRanges(c(7,13), width = 3)))
> grl2 <- GRangesList(
+   gr1 = GRanges("chr2", IRanges(9, 12)),
+   gr2 = GRanges("chr1", IRanges(c(25,38), width = 3)))
> pc(grl1, grl2)
```

GRangesList object of length 2:

\$gr1

GRanges object with 2 ranges and 0 metadata columns:

	seqnames	ranges	strand
	<Rle>	<IRanges>	<Rle>
[1]	chr2	[3, 6]	*
[2]	chr2	[9, 12]	*

\$gr2

GRanges object with 4 ranges and 0 metadata columns:

	seqnames	ranges	strand
[1]	chr1	[7, 9]	*
[2]	chr1	[13, 15]	*
[3]	chr1	[25, 27]	*
[4]	chr1	[38, 40]	*

seqinfo: 2 sequences from an unspecified genome; no seqlengths

```
> grl3 <- c(grl1, grl2)
> regroup(grl3, names(grl3))
```

GRangesList object of length 2:

\$gr1

GRanges object with 2 ranges and 0 metadata columns:

	seqnames	ranges	strand
	<Rle>	<IRanges>	<Rle>
[1]	chr2	[3, 6]	*
[2]	chr2	[9, 12]	*

\$gr2

GRanges object with 4 ranges and 0 metadata columns:

	seqnames	ranges	strand
[1]	chr1	[7, 9]	*
[2]	chr1	[13, 15]	*
[3]	chr1	[25, 27]	*
[4]	chr1	[38, 40]	*

seqinfo: 2 sequences from an unspecified genome; no seqlengths

3.3 Basic interval operations for GRangesList objects

For interval operations, many of the same methods exist for *GRangesList* objects that exist for *GRanges* objects.

```
> start(grl)

IntegerList of length 2
[["txA"]] 103
[["txB"]] 107 113

> end(grl)

IntegerList of length 2
[["txA"]] 106
[["txB"]] 109 115

> width(grl)

IntegerList of length 2
[["txA"]] 4
[["txB"]] 3 3
```

These operations return a data structure representing, e.g., *IntegerList*, a list where all elements are integers; it can be convenient to use mathematical and other operations on **List* objects that work on each element, e.g.,

```
> sum(width(grl)) # sum of widths of each grl element

txA txB
  4   6
```

Most of the intra-, inter- and between-range methods operate on *GRangesList* objects, e.g., to shift all the *GRanges* objects in a *GRangesList* object, or calculate the coverage. Both of these operations are also carried out across each *GRanges* list member.

```
> shift(grl, 20)

GRangesList object of length 2:
$txA
GRanges object with 1 range and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle>  <IRanges>  <Rle> | <integer> <numeric>
[1]      chr2 [123, 126]      + |         5      0.45

$txB
GRanges object with 2 ranges and 2 metadata columns:
      seqnames      ranges strand | score GC
[1]      chr1 [127, 129]      + |   3 0.3
[2]      chr1 [133, 135]      - |   4 0.5
```

```
-----
seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

```
> coverage(grl)

RleList of length 2
$chr2
integer-Rle of length 106 with 2 runs
  Lengths: 102  4
  Values :  0  1

$chr1
```

```
integer-Rle of length 115 with 4 runs
Lengths: 106  3  3  3
Values :  0  1  0  1
```

3.4 Subsetting GRangesList objects

A *GRangesList* object behaves like a list: `[` returns a *GRangesList* containing a subset of the original object; `[[` or `$` returns the *GRanges* object at that location in the list.

```
> grl[1]
> grl[[1]]
> grl["txA"]
> grl$txB
```

In addition, subsetting a *GRangesList* also accepts a second parameter to specify which of the metadata columns you wish to select.

```
> grl[1, "score"]
```

GRangesList object of length 1:

\$txA

GRanges object with 1 range and 1 metadata column:

	seqnames	ranges	strand	score
	<Rle>	<IRanges>	<Rle>	<integer>
[1]	chr2	[103, 106]	+	5

seqinfo: 2 sequences from an unspecified genome; no seqlengths

```
> grl["txB", "GC"]
```

GRangesList object of length 1:

\$txB

GRanges object with 2 ranges and 1 metadata column:

	seqnames	ranges	strand	GC
	<Rle>	<IRanges>	<Rle>	<numeric>
[1]	chr1	[107, 109]	+	0.3
[2]	chr1	[113, 115]	-	0.5

seqinfo: 2 sequences from an unspecified genome; no seqlengths

The `head`, `tail`, `rep`, `rev`, and `window` methods all behave as you would expect them to for a list object. For example, the elements referred to by `window` are now list elements instead of *GRanges* elements.

```
> rep(grl[[1]], times = 3)
```

GRanges object with 3 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45
[2]	chr2	[103, 106]	+	5	0.45
[3]	chr2	[103, 106]	+	5	0.45

seqinfo: 2 sequences from an unspecified genome; no seqlengths

```
> rev(grl)
```

GRangesList object of length 2:

\$txB

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr1	[107, 109]	+	3	0.3
[2]	chr1	[113, 115]	-	4	0.5

\$txA

GRanges object with 1 range and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45

seqinfo: 2 sequences from an unspecified genome; no seqlengths

> head(grl, n=1)

GRangesList object of length 1:

\$txA

GRanges object with 1 range and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45

seqinfo: 2 sequences from an unspecified genome; no seqlengths

> tail(grl, n=1)

GRangesList object of length 1:

\$txB

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr1	[107, 109]	+	3	0.3
[2]	chr1	[113, 115]	-	4	0.5

seqinfo: 2 sequences from an unspecified genome; no seqlengths

> window(grl, start=1, end=1)

GRangesList object of length 1:

\$txA

GRanges object with 1 range and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45

seqinfo: 2 sequences from an unspecified genome; no seqlengths

> grl[IRanges(start=2, end=2)]

GRangesList object of length 1:

\$txB

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr1	[107, 109]	+	3	0.3
[2]	chr1	[113, 115]	-	4	0.5

```
-----
seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

3.5 Looping over GRangesList objects

For *GRangesList* objects there is also a family of apply methods. These include `lapply`, `sapply`, `mapply`, `endoapply`, `mendoapply`, `Map`, and `Reduce`.

The different looping methods defined for *GRangesList* objects are useful for returning different kinds of results. The standard `lapply` and `sapply` behave according to convention, with the `lapply` method returning a list and `sapply` returning a more simplified output.

```
> lapply(grl, length)
```

```
$txA
[1] 1
```

```
$txB
[1] 2
```

```
> sapply(grl, length)
```

```
txA txB
 1    2
```

As with *IRanges* objects, there is also a multivariate version of `sapply`, called `mapply`, defined for *GRangesList* objects. And, if you don't want the results simplified, you can call the `Map` method, which does the same things as `mapply` but without simplifying the output.

```
> grl2 <- shift(grl, 10)
> names(grl2) <- c("shiftTxA", "shiftTxB")
> mapply(c, grl, grl2)
```

```
$txA
GRanges object with 2 ranges and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle>      <IRanges> <Rle> | <integer> <numeric>
[1]    chr2 [103, 106]      + |         5     0.45
[2]    chr2 [113, 116]      + |         5     0.45
```

```
-----
seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

```
$txB
GRanges object with 4 ranges and 2 metadata columns:
      seqnames      ranges strand |      score      GC
      <Rle>      <IRanges> <Rle> | <integer> <numeric>
[1]    chr1 [107, 109]      + |         3     0.3
[2]    chr1 [113, 115]      - |         4     0.5
[3]    chr1 [117, 119]      + |         3     0.3
[4]    chr1 [123, 125]      - |         4     0.5
```

```
-----
seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

```
> Map(c, grl, grl2)
```

\$txA

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45
[2]	chr2	[113, 116]	+	5	0.45

seqinfo: 2 sequences from an unspecified genome; no seqlengths

\$txB

GRanges object with 4 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr1	[107, 109]	+	3	0.3
[2]	chr1	[113, 115]	-	4	0.5
[3]	chr1	[117, 119]	+	3	0.3
[4]	chr1	[123, 125]	-	4	0.5

seqinfo: 2 sequences from an unspecified genome; no seqlengths

Sometimes you will want to get back a modified version of the *GRangesList* that you originally passed in.

An endomorphism is a transformation of an object to another instance of the same class . This is achieved using the `endoapply` method, which will return the results as a *GRangesList* object.

```
> endoapply(grl, rev)
```

GRangesList object of length 2:

\$txA

GRanges object with 1 range and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45

\$txB

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
[1]	chr1	[113, 115]	-	4	0.5
[2]	chr1	[107, 109]	+	3	0.3

seqinfo: 2 sequences from an unspecified genome; no seqlengths

```
> mendoapply(c, grl, grl2)
```

GRangesList object of length 2:

\$txA

GRanges object with 2 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45
[2]	chr2	[113, 116]	+	5	0.45

\$txB

GRanges object with 4 ranges and 2 metadata columns:

	seqnames	ranges	strand	score	GC
--	----------	--------	--------	-------	----

```
[1] chr1 [107, 109] + | 3 0.3
[2] chr1 [113, 115] - | 4 0.5
[3] chr1 [117, 119] + | 3 0.3
[4] chr1 [123, 125] - | 4 0.5
```

```
-----
```

```
seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

The Reduce method will allow the *GRanges* objects to be collapsed across the whole of the *GRangesList* object.

```
> Reduce(c, grl)
```

```
GRanges object with 3 ranges and 2 metadata columns:
```

	seqnames	ranges	strand	score	GC
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>
[1]	chr2	[103, 106]	+	5	0.45
[2]	chr1	[107, 109]	+	3	0.3
[3]	chr1	[113, 115]	-	4	0.5

```
-----
```

```
seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Explicit element-wise operations (`lapply()` and friends) on *GRangesList* objects with many elements can be slow. It is therefore beneficial to explore operations that work on **List* objects directly (e.g., many of the 'group generic' operators, see `?S4groupGeneric`, and the set and parallel set operators (e.g., `union`, `punion`). A useful and fast strategy is to unlist the *GRangesList* to a *GRanges* object, operate on the *GRanges* object, then relist the result, e.g.,

```
> gr <- unlist(grl)
> gr$log_score <- log(gr$score)
> grl <- relist(gr, grl)
> grl
```

```
GRangesList object of length 2:
```

```
$txA
```

```
GRanges object with 1 range and 3 metadata columns:
```

	seqnames	ranges	strand	score	GC	log_score
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>	<numeric>
txA	chr2	[103, 106]	+	5	0.45	1.6094379124341

```
$txB
```

```
GRanges object with 2 ranges and 3 metadata columns:
```

	seqnames	ranges	strand	score	GC	log_score
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>	<numeric>
txB	chr1	[107, 109]	+	3	0.3	1.09861228866811
txB	chr1	[113, 115]	-	4	0.5	1.38629436111989

```
-----
```

```
seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

See also `?extractList`.

For more information on the *GRangesList* classes be sure to consult the manual page and available methods

```
> ?GRangesList
> methods(class="GRangesList") # _partial_ list
```

4 Interval overlaps involving GRanges and GRangesList objects

Interval overlapping is the process of comparing the ranges in two objects to determine if and when they overlap. As such, it is perhaps the most common operation performed on *GRanges* and *GRangesList* objects. To this end, the *GenomicRanges* package provides a family of interval overlap functions. The most general of these functions is `findOverlaps`, which takes a query and a subject as inputs and returns a *Hits* object containing the index pairings for the overlapping elements.

```
> mtch <- findOverlaps(gr, grl)
> as.matrix(mtch)
```

	queryHits	subjectHits
[1,]	1	1
[2,]	2	2
[3,]	3	2

As suggested in the sections discussing the nature of the *GRanges* and *GRangesList* classes, the index in the above matrix of hits for a *GRanges* object is a single range while for a *GRangesList* object it is the set of ranges that define a "feature".

Another function in the overlaps family is `countOverlaps`, which tabulates the number of overlaps for each element in the query.

```
> countOverlaps(gr, grl)
```

txA	txB	txB
1	1	1

A third function in this family is `subsetByOverlaps`, which extracts the elements in the query that overlap at least one element in the subject.

```
> subsetByOverlaps(gr, grl)
```

GRanges object with 3 ranges and 3 metadata columns:

	seqnames	ranges	strand	score	GC	log_score
	<Rle>	<IRanges>	<Rle>	<integer>	<numeric>	<numeric>
txA	chr2	[103, 106]	+	5	0.45	1.6094379124341
txB	chr1	[107, 109]	+	3	0.3	1.09861228866811
txB	chr1	[113, 115]	-	4	0.5	1.38629436111989

seqinfo: 2 sequences from an unspecified genome; no seqlengths

Finally, you can use the `select` argument to get the index of the first overlapping element in the subject for each element in the query.

```
> findOverlaps(gr, grl, select="first")
```

```
[1] 1 2 2
```

```
> findOverlaps(grl, gr, select="first")
```

```
[1] 1 2
```

5 Session Information

All of the output in this vignette was produced under the following conditions:

```
> sessionInfo()
```

```
R version 3.4.1 (2017-06-30)
```

```
Platform: x86_64-apple-darwin15.6.0 (64-bit)
```

Running under: OS X El Capitan 10.11.6

Matrix products: default

BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib

LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib

locale:

[1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:

[1] parallel stats4 stats graphics grDevices utils datasets methods
[9] base

other attached packages:

[1] BSgenome.Scerevisiae.UCSC.sacCer2_1.4.0	KEGGgraph_1.38.1
[3] KEGG.db_3.2.3	BSgenome.Hsapiens.UCSC.hg19_1.4.0
[5] BSgenome_1.44.2	rtracklayer_1.36.5
[7] edgeR_3.18.1	limma_3.32.7
[9] DESeq2_1.16.1	AnnotationHub_2.8.2
[11] TxDb.Athaliana.BioMart.plantmart22_3.0.1	TxDb.Hsapiens.UCSC.hg19.knownGene_3.2.2
[13] TxDb.Dmelanogaster.UCSC.dm3.ensGene_3.2.2	GenomicFeatures_1.28.5
[15] AnnotationDbi_1.38.2	GenomicAlignments_1.12.2
[17] Rsamtools_1.28.0	Biostrings_2.44.2
[19] XVector_0.16.0	SummarizedExperiment_1.6.5
[21] DelayedArray_0.2.7	matrixStats_0.52.2
[23] Biobase_2.36.2	pasillaBamSubset_0.14.0
[25] GenomicRanges_1.28.6	GenomeInfoDb_1.12.2
[27] IRanges_2.10.4	S4Vectors_0.14.6
[29] BiocGenerics_0.22.0	

loaded via a namespace (and not attached):

[1] bitops_1.0-6	bit64_0.9-7
[3] RColorBrewer_1.1-2	httr_1.3.1
[5] rprojroot_1.2	tools_3.4.1
[7] backports_1.1.1	R6_2.2.2
[9] rpart_4.1-11	Hmisc_4.0-3
[11] DBI_0.7	lazyeval_0.2.0
[13] colorspace_1.3-2	nnet_7.3-12
[15] gridExtra_2.3	bit_1.1-12
[17] curl_2.8.1	compiler_3.4.1
[19] graph_1.54.0	htmlTable_1.9
[21] scales_0.5.0	checkmate_1.8.4
[23] genefilter_1.58.1	stringr_1.2.0
[25] digest_0.6.12	foreign_0.8-69
[27] rmarkdown_1.6	base64enc_0.1-3
[29] pkgconfig_2.0.1	htmltools_0.3.6
[31] htmlwidgets_0.9	rlang_0.1.2
[33] RSQLite_2.0	BiocInstaller_1.26.1
[35] shiny_1.0.5	BiocParallel_1.10.1
[37] acepack_1.4.1	VariantAnnotation_1.22.3
[39] RCurl_1.95-4.8	magrittr_1.5
[41] GenomeInfoDbData_0.99.0	Formula_1.2-2
[43] Matrix_1.2-11	Rcpp_0.12.13
[45] munsell_0.4.3	stringi_1.1.5
[47] yaml_2.1.14	zlibbioc_1.22.0
[49] plyr_1.8.4	grid_3.4.1
[51] blob_1.1.0	lattice_0.20-35
[53] splines_3.4.1	annotate_1.54.0
[55] locfit_1.5-9.1	knitr_1.17

```
[57] geneplotter_1.54.0      biomaRt_2.32.1
[59] XML_3.98-1.9            evaluate_0.10.1
[61] latticeExtra_0.6-28     data.table_1.10.4
[63] httpuv_1.3.5            gtable_0.2.0
[65] ggplot2_2.2.1           mime_0.5
[67] xtable_1.8-2            survival_2.41-3
[69] tibble_1.3.4            memoise_1.1.0
[71] cluster_2.0.6           interactiveDisplayBase_1.14.0
[73] BiocStyle_2.4.1
```