Intervals and Data on Intervals in BioC

The IRanges Package

November 18, 2009

Outline

NGS perspective on IRanges package

- Representation of information on chromosomes/contigs
  - Intervals with or without associated data
  - Piecewise constant measures (e.g. coverage)
- Vector and interval operations for these representations
  - Interval overlap calculations
  - Coverage area within peak regions
- Metadata scheme for self-documenting objects

Two most important classes in IRanges

- *RangedData* - intervals and associated data on chromosomes/contigs. It can be conceptualized as a data table that is sorted by the chromosomes/contigs indicator column.
- *RleList* - coverage (or other piecewise constant measures) on chromosomes/contigs. RLE is an initialism for run length encoding, a standard compression method in signal processing.

Outline

Intervals with corresponding data row

- Genomic coordinates consist of chromosome, position, and potentially strand information
- Each coordinate or set of coordinates may have additional values associated with it, such as GC content or alignment coverage
- A collection of intervals with data are commonly called tracks in genome browsers

Naive representation of intervals with data row (1/2)

- Tables in R are commonly stored in *data.frame* objects.

data.frame approach

```
> chr <- c("chr1", "chr2", "chr1")
> strand <- c("+", "+", "-")
> start <- c(3L, 4L, 1L)
> end <- c(7L, 5L, 3L)
> naiveTable <- data.frame(chr = chr,
+     strand = strand, start = start,
+     end = end)
> naiveTable

   chr strand start end
1 chr1      +     3   7
2 chr2      +     4   5
3 chr1      -     1   3
```

Naive representation of intervals with data row (2/2)

- *data.frame* objects are poorly suited for this data because operations are constantly performed within chromosome/contig.

Using by to loop over data.frame

```
> by(naiveTable, naiveTable[["chr"]],
+     function(x) range(x[c("start",
+         "end")]))

naiveTable[["chr"]]: chr1
[1] 1 7
------------------------------------
naiveTable[["chr"]]: chr2
[1] 4 5
```

RangedData construction

- Instances are created using the `RangedData` constructor.
- Interval starts and ends are wrapped in an *IRanges* constructor.
- Chromosome/contig information is supplied to space argument.

```
> rdTable <- RangedData(ranges = IRanges(start = start,
+     end = end), strand = strand, space = chr)
```

RangedData display

- The *RangedData* class sacrifices table row order flexibility for faster computational timings.

```
> rdTable
```

```
RangedData with 3 rows and 1 value column across 2 spaces
        space    ranges |      strand
  <character> <IRanges> | <character>
1        chr1    [3, 7] |           +
2        chr1    [1, 3] |           -
3        chr2    [4, 5] |           +
```

RangedData computation

Simplified looping

```
> range(ranges(rdTable))

CompressedIRangesList of length 2
$chr1
IRanges of length 1
    start end width
[1]     1   7     7

$chr2
IRanges of length 1
    start end width
[1]     4   5     2
```

RangedData class decomposition

- *RangedData*
  - *RangesList* - intervals on chromosomes/contigs. Extracted using the `ranges` function.
    - *Ranges* - intervals for a specific chromosome/contig. Most common subclass is *IRanges*.
  - *SplitDataFrameList* - data on chromosomes/contigs. Extracted using the `values` function.
    - *DataFrame* - data for a specific chromosome/contig.

Interval operations

Intra-interval : `flank`, `narrow`, `reflect`, `resize`, `restrict`, `shift`

Inter-interval I : `disjoin`, `gaps`, `reduce`, `range`

Inter-interval II : `coverage`

Between two interval sets I : `intersect`, `setdiff`, `union`

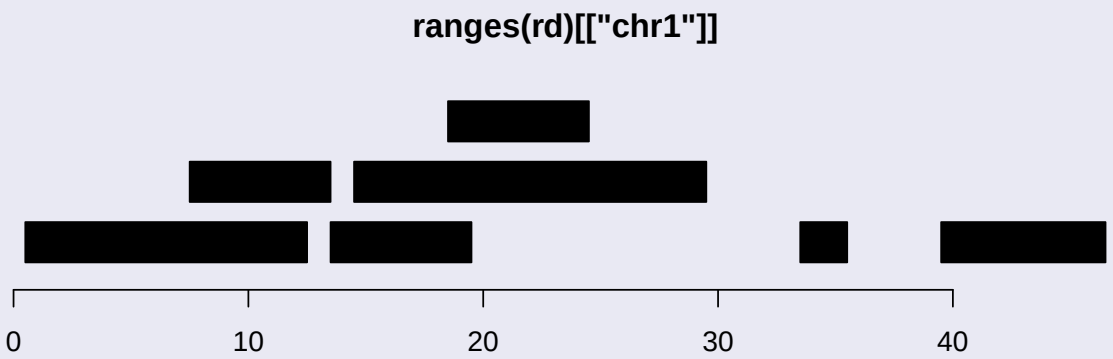
Between two interval sets II : `findOverlaps`, `%in%`, `match`

Low level : `start`, `end`, `width`, `space`, `universe`

Creating a new RangedData object

New object to use in interval operations

```
> ir <- IRanges(c(1, 8, 14, 15, 19, 34,
+     40), width = c(12, 6, 6, 15, 6,
+     2, 7))
> strand <- rep(c("+", "-"), c(4, 3))
> rd <- RangedData(ranges = ir, strand = strand,
+     space = "chr1")
```



Low level data access

Accessors

```
> start(rd)
[1]  1  8 14 15 19 34 40
> end(rd)
[1] 12 13 19 29 24 35 46
> width(rd)
[1] 12  6  6 15  6  2  7
```

RangedData subsetting

```
> rd[1:5, ]
```

```
RangedData with 5 rows and 1 value column across 1 space
        space    ranges |      strand
  <character> <IRanges> | <character>
1        chr1  [ 1, 12] |           +
2        chr1  [ 8, 13] |           +
3        chr1  [14, 19] |           +
4        chr1  [15, 29] |           +
5        chr1  [19, 24] |           -
```

Shifting intervals

- If your interval bounds are off by 1, you can shift them.

```
> rd2 <- rd
> ranges(rd2) <- shift(ranges(rd2), 1)
> rd2
```

```
RangedData with 7 rows and 1 value column across 1 space
       space    ranges |      strand
  <character> <IRanges> | <character>
1        chr1  [ 2, 13] |           +
2        chr1  [ 9, 14] |           +
3        chr1  [15, 20] |           +
4        chr1  [16, 30] |           +
5        chr1  [20, 25] |           -
6        chr1  [35, 36] |           -
7        chr1  [41, 47] |           -
```

Resizing intervals (1/2)

- One common operation in ChIP-seq experiments is to "grow" and alignment interval to an estimated fragment length.

```
> rd3 <- rd
> pos <- values(rd3)[, "strand"] == "+"
> ranges(rd3)[pos] <- resize(ranges(rd)[pos],
+     120)
> ranges(rd3)[!pos] <- resize(ranges(rd)[!pos],
+     120, start = FALSE)
```

Resizing intervals (2/2)

```
> rd3
```

```
RangedData with 7 rows and 1 value column across 1 space
        space     ranges |      strand
  <character>  <IRanges> | <character>
1        chr1 [  1, 120] |           +
2        chr1 [  8, 127] |           +
3        chr1 [ 14, 133] |           +
4        chr1 [ 15, 134] |           +
5        chr1 [-95,  24] |           -
6        chr1 [-84,  35] |           -
7        chr1 [-73,  46] |           -
```

Restricting interval bounds

- The previous operation created some negative start values. We can "clip" those negative values.

```
> ranges(rd3) <- restrict(ranges(rd3),
+     1)
> rd3
```

```
RangedData with 7 rows and 1 value column across 1 space
       space    ranges |      strand
  <character> <IRanges> | <character>
1        chr1 [ 1, 120] |           +
2        chr1 [ 8, 127] |           +
3        chr1 [14, 133] |           +
4        chr1 [15, 134] |           +
5        chr1 [ 1,  24] |           -
6        chr1 [ 1,  35] |           -
7        chr1 [ 1,  46] |           -
```

Normalizing intervals

- *Ranges* can represent a set of integers
- *NormalIRanges* formalizes this, with a compact, normalized representation
- `reduce` normalizes ranges

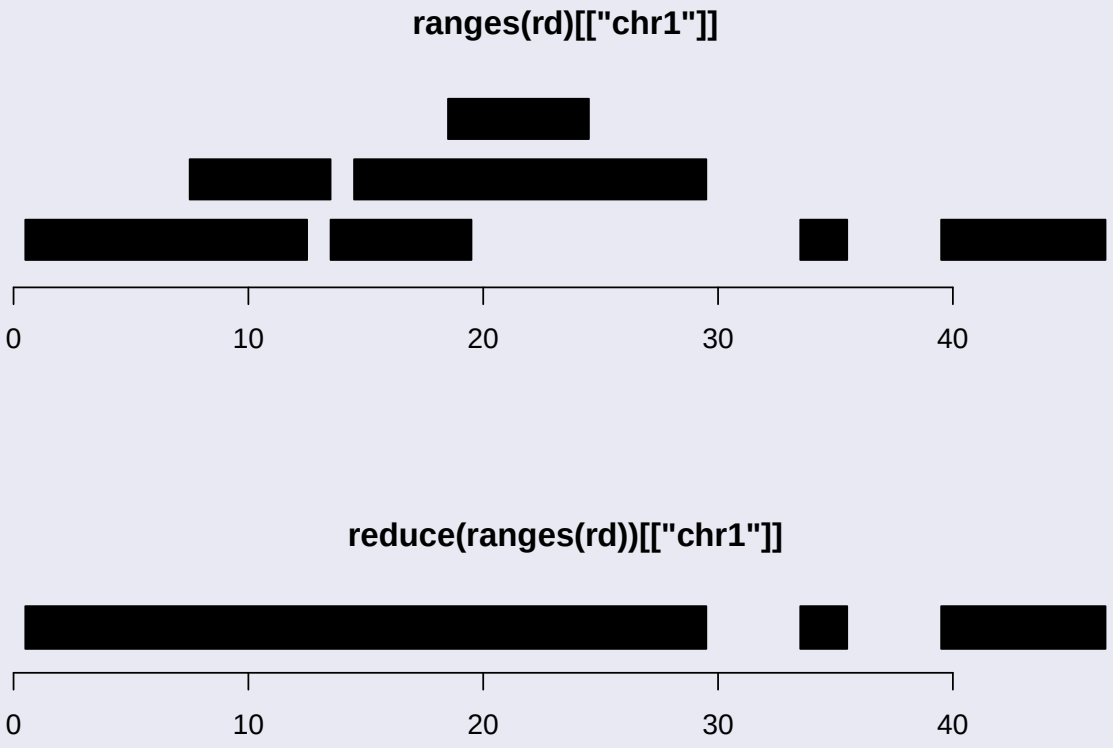
Code

```
> reduce(ranges(rd))
```

Normalizing intervals

Code

```
> reduce(ranges(rd))
```



Set operations

- *Ranges* as set of integers: `intersect`, `union`, `gaps`, `setdiff`
- Each range as integer set, in parallel: `pintersect`, `punion`, `pgap`, `psetdiff`

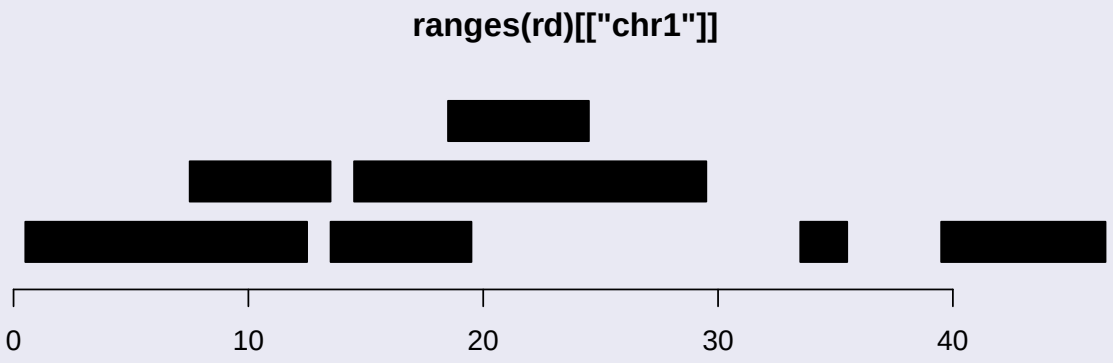
Example: `gaps`

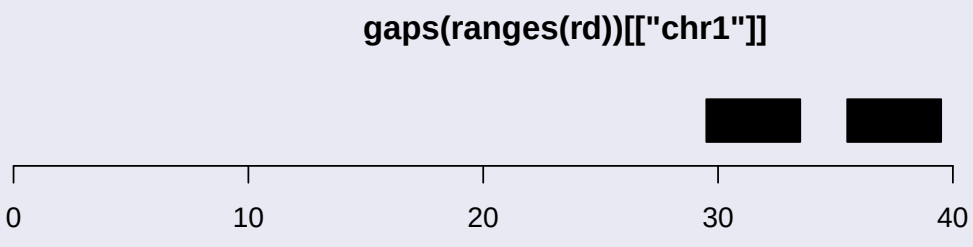
```
> gaps(ranges(rd))
```

Set operations

Example: gaps

```
> gaps(ranges(rd))
```





Disjoining intervals

- Disjoint ranges are non-overlapping
- `disjoin` returns the widest ranges where the overlapping ranges are the same

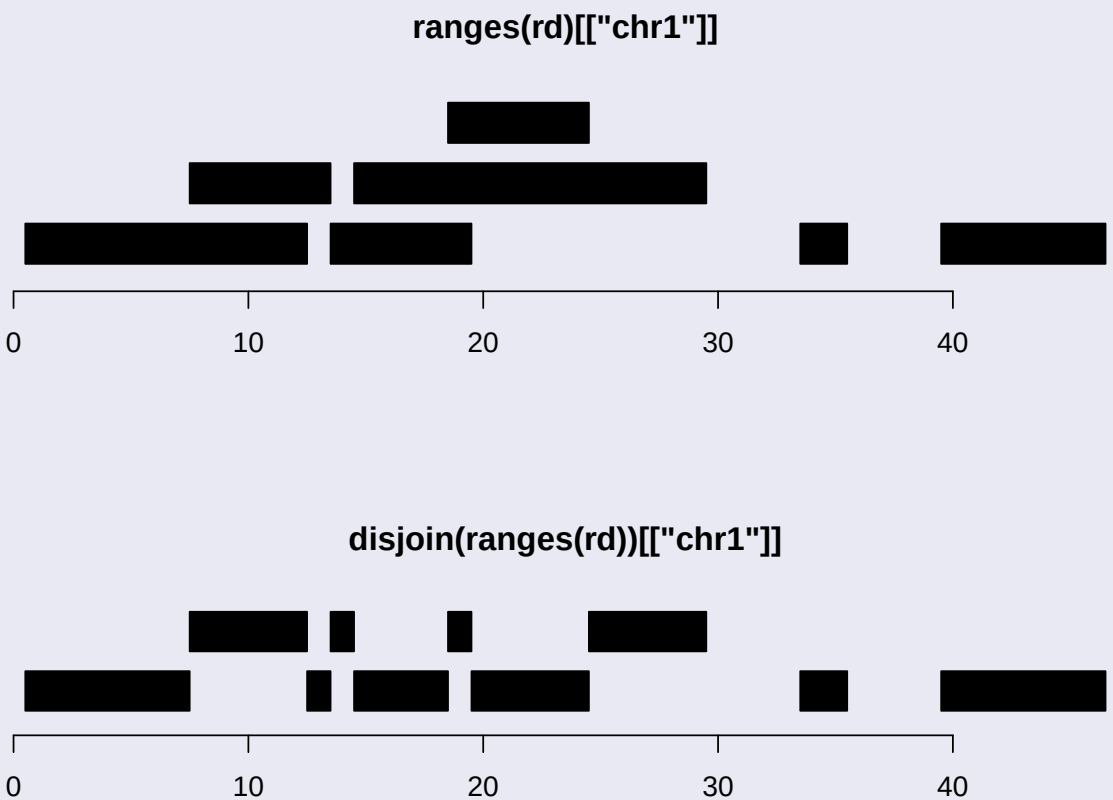
Code

```
> disjoin(ranges(rd))
```

Disjoining intervals

Code

```
> disjoin(ranges(rd))
```



Overlap detection

- `overlap` detects overlaps between two *Ranges* objects
- Uses interval tree for efficiency

Code

```
> ol <- findOverlaps(ranges(rd), reduce(ranges(rd)))
> as.matrix(ol)
```

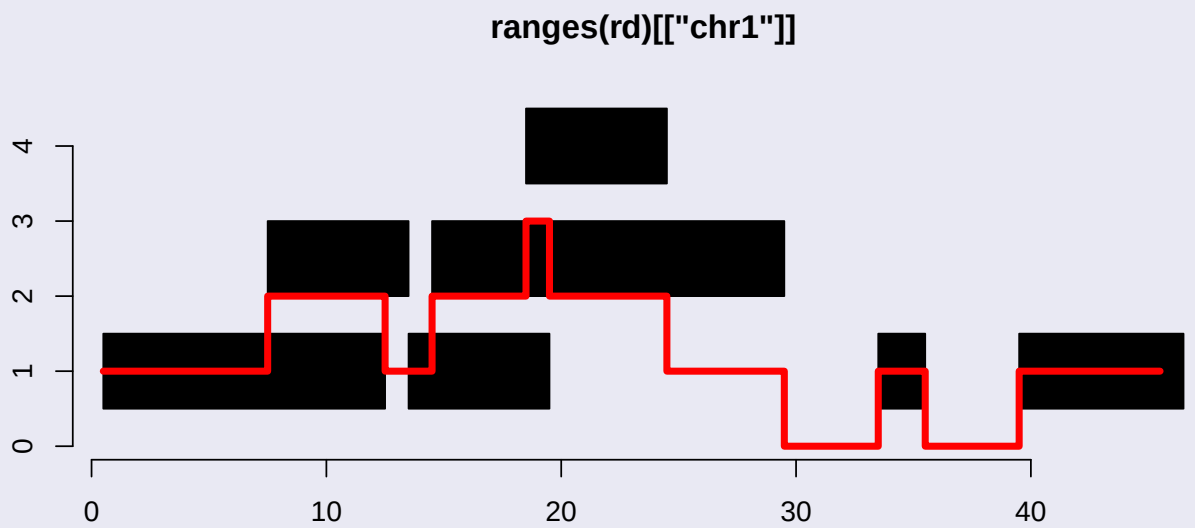
```
     query subject
[1,]     1       1
[2,]     2       1
[3,]     3       1
[4,]     4       1
[5,]     5       1
[6,]     6       2
[7,]     7       3
```

Counting overlapping intervals

`coverage` counts number of ranges over each position

Code

```
> cover <- coverage(ranges(rd))
```



Finding nearest neighbors

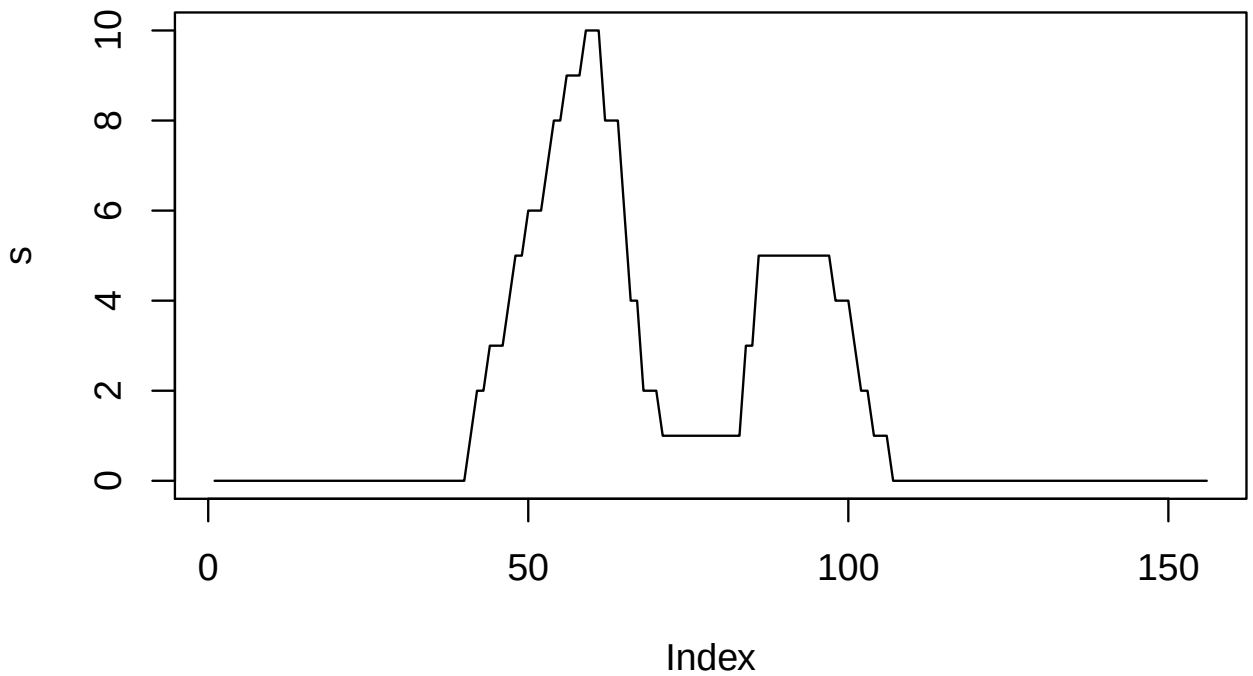
- `nearest` finds the nearest neighbor ranges (overlapping is zero distance)
- `precede`, `follow` find non-overlapping nearest neighbors on specific side

Outline

Positional piecewise constant measures

- The number of genomic positions in a genome is often in the billons for higher organisms, making it challenging to represent in memory.
- Some data across a genome tend to be sparse (i.e. large stretches of "no information")
- The *IRanges* packages solves the set of problems for positional measures that tend to have consecutively repeating values.
- The *IRanges* package *does not* address the more general problem of positional measures that constantly fluxuate, such as conservation scores.

Example sequence



Run-Length Encodings (RLEs)

Our example has many repeated values:

Code

```
> sum(diff(s) == 0)

[1] 133
```

Good candidate for compression by run-length encoding:

Code

```
> sRle <- Rle(s)
> sRle

'numeric' Rle of length 156 with 23 runs
  Lengths:  40 1 2 3 1 2 3 1 2 3 ...
  Values :  0 1 2 3 4 5 6 7 8 9 ...
```

Compression reduces size from 156 to 46.

Rle operations

The *Rle* object shares many method interfaces with vector:

Basic

```
> sRle > 0 | rev(sRle) > 0

'logical' Rle of length 156 with 3 runs
  Lengths:  40 76 40
  Values :  FALSE TRUE FALSE
```

Summary

```
> sum(sRle > 0)

[1] 66
```

Statistics

```
> cor(sRle, rev(sRle))

[1] 0.5142557
```

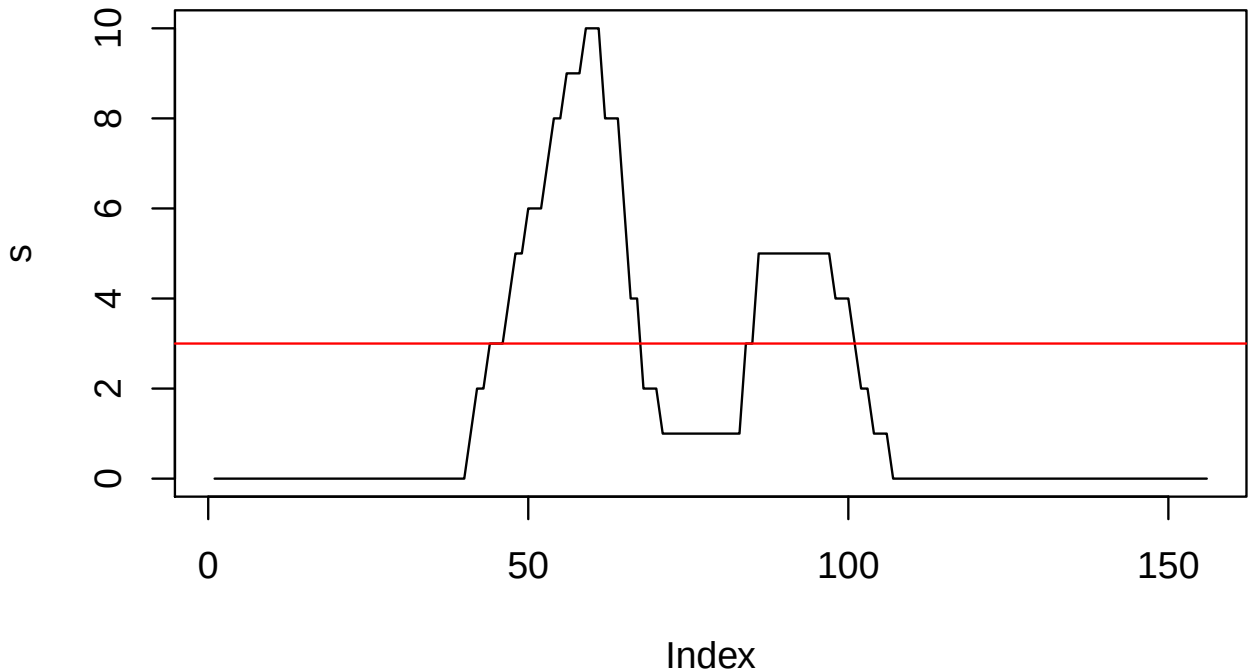
Outline

Views

- Associates a *Ranges* object with a sequence
- Sequences can be *Rle* or (in Biostrings) *XString*
- Extends *Ranges*, so supports the same operations

Slicing a Sequence into Views

Goal: find regions above cutoff of 3



Slicing a Sequence into Views

Goal: find regions above cutoff of 3

Using Rle

```
> Views(sRle, as(sRle > 3, "IRanges"))

Views on a 156-length Rle subject

views:
    start end width
[1]    47  67    21 [ 4  5  5  6  6  6 ...]
[2]    86 100    15 [5 5 5 5 5 5 5 5 5 ...]
```

Convenience

```
> sViews <- slice(sRle, 4)
> sViewsList <- RleViewsList(slice(sRle,
+     4), slice(rev(sRle), 4))
```

Summarizing windows

- Could `sapply` over each window
- Native functions available for common tasks: `viewMins`, `viewMaxs`, `viewSums`, ...

Code

```
> viewSums(sViews)

[1] 150  72

> viewSums(sViewsList)

SimpleNumericList of length 2
[[1]] 150 72
[[2]] 72 150

> viewMaxs(sViews)

[1] 10  5

> viewMaxs(sViewsList)
```