IRanges
Bioconductor Infrastructure for Sequence Analysis

November 24, 2009
1 Introduction

2 Sequences
   - Background
   - RLEs

3 Ranges
   - Basic Manipulation
   - Simple Transformations
   - Ranges as Sets
   - Overlap

4 Views

5 Interval Datasets
   - Motivation
   - RangedData Representation
   - Accessing interval data
IRanges

- Supports the manipulation and analysis of:
  - Sequences (ordered collections of elements)
  - Ranges of indices into sequences
  - Data on ranges
- Emphasis on efficiency in space and time
- Metadata scheme for self-documenting objects and reproducible analysis
IRanges and High-throughput Sequencing

- The basis of much of the sequence analysis functionality in Bioconductor
- 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 within peak regions
The Two Towers of IRanges

- **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.
- **RangedData** - intervals and associated data on chromosomes/contigs. Essentially a data table that is sorted by the chromosomes/contigs indicator column.
Outline

1 Introduction

2 Sequences
   Background
   RLEs

3 Ranges
   Basic Manipulation
   Simple Transformations
   Ranges as Sets
   Overlap

4 Views

5 Interval Datasets
   Motivation
   RangedData Representation
   Accessing interval data
Almost every object manipulated by IRanges is a sequence:

- Atomic sequences (e.g. R vectors)
- Lists
- Data tables (two dimensions)
The number of genomic positions in a genome is often in the billions 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 problem of positional measures that constantly fluxuate, such as conservation scores.
Example sequence
Run-Length Encoding (RLE)

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 like any other sequence/vector:

### Basic

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

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

### Summary

```r
> sum(sRle > 0)

[1] 66
```

### Statistics

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

[1] 0.5142557
```
Splitting up \textit{Rle} by sequence

\textbf{Code}

\begin{verbatim}
> print(sRleList <- split(sRle, rep(c("chr1", 
+ "chr2"), each = length(sRle)/2)))

CompressedRleList of length 2
$chr1
'numeric' Rle of length 78 with 16 runs
  Lengths:  40 1 2 3 1 2 3 1 2 3 ...
  Values :  0 1 2 3 4 5 6 7 8 9 ...

$chr2
'numeric' Rle of length 78 with 8 runs
  Lengths:  5 2 12 3 1 2 3 50
  Values :  1 3 5 4 3 2 1 0
\end{verbatim}

\textit{RleList} supports most \textit{Rle} operations, element-wise.
External sequences

- Sequences derived from XSequence are references
- Memory not copied when containing object is modified
- Example: XString in Biostrings package, for storing biological sequences efficiently
Outline

1 Introduction

2 Sequences
   Background
   RLEs

3 Ranges
   Basic Manipulation
   Simple Transformations
   Ranges as Sets
   Overlap

4 Views

5 Interval Datasets
   Motivation
   RangedData Representation
   Accessing interval data
• Often interested in *consecutive* subsequences
• Consider the alphabet as a sequence:
  • \{A, B, C\} is a consecutive subsequence
  • The vowels would not be consecutive
• Compact representation: *range* (start and width)
• *Ranges* objects store a sequence of ranges
The *IRanges* class is a simple *Ranges* implementation.

### Code

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

![Diagram of IRanges object]
Low level data access

Accessors

> start(ir)

[1]  1  8 14 15 19 34 40

> end(ir)

[1]  12 13 19 29 24 35 46

> width(ir)

[1]  12  6  6 15  6  2  7
Basic Manipulation

Subsetting

Code

> ir[1:5]

IRanges of length 5

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>29</td>
<td>15</td>
</tr>
<tr>
<td>19</td>
<td>24</td>
<td>6</td>
</tr>
</tbody>
</table>
Splitting up *Ranges* by sequence

```r
> rl <- split(ir, c(rep("chr1", 2), rep("chr2", 3), "chr1", "chr2"))
> rl[1]

CompressedIRangesList of length 1
$chr1
IRanges of length 3
   start  end width
[1]  1  12  12
[2]  8  13   6
[3] 34  35   2
```

*RangesList* supports most *Ranges* operations, element-wise.
Shifting intervals

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

Code

```r
> shift(ir, 1)
```
Shifting intervals

Code

\[
\texttt{> shift(ir, 1)}
\]
One common operation in ChIP-seq experiments is to “grow” an alignment interval to an estimated fragment length.

Code

```r
> ir15 <- resize(ir, 15)
> print(ir15 <- resize(ir, 15, start = FALSE))

IRanges of length 7

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>-1</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>29</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>21</td>
<td>35</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>46</td>
<td>15</td>
</tr>
</tbody>
</table>
```
The previous operation created some negative start values. We can “clip” those negative values.

Code

```r
> restrict(ir15, 1)

IRanges of length 7

start end width
[1] 1  12  12
[2] 1  13  13
[3] 5  19  15
[4] 15  29  15
[6]  21  35  15
[7] 32  46  15
```
Normalizing ranges

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

**Code**

```r
> reduce(ir)
```
Normalizing ranges

Code

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

\[ \text{ir} \]

\[ \text{reduce}(ir) \]
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(ir)
```
Example: `gaps`

```r
> gaps(ir)
```

The diagram shows the original range `ir` and the gaps resulting from applying the `gaps` function.
Disjoining ranges

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

Code

```r
> disjoin(ir)
```
Disjoining ranges

Code

\[ \text{disjoin}(i) \]

```
> disjoin(ir)
```

![Diagram showing disjoin operation on intervals]
Overlap detection

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

Code

```r
> ol <- findOverlaps(ir, reduce(ir))
> 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
```
Coverage counts the number of ranges overlapping each position.

**Code**

```r
> cov <- coverage(ir)
```

[Graph showing coverage with ranges over each position]
Coverage over **multiple** sequences

coverage also works for *RangesList*:

```
Code

> covL <- coverage(rl)
> covL

SimpleRleList of length 2

$chr1
'integer' Rle of length 35 with 5 runs
  Lengths:  7  5  1  20  2
  Values :  1  2  1  0  1

$chr2
'integer' Rle of length 46 with 8 runs
  Lengths:  13  1  4  1  5  5  10  7
  Values :  0  1  2  3  2  1  0  1
```
Finding nearest neighbors

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

1. Introduction
2. Sequences
   - Background
   - RLEs
3. Ranges
   - Basic Manipulation
   - Simple Transformations
   - Ranges as Sets
   - Overlap
4. Views
5. Interval Datasets
   - Motivation
   - RangedData Representation
   - Accessing interval data
Ranges on Sequences: 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

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

Views on a 156-length Rle subject

views:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>67</td>
<td>21</td>
</tr>
<tr>
<td>86</td>
<td>100</td>
<td>15</td>
</tr>
</tbody>
</table>

Convenience

```r
> sViews <- slice(sRle, 4)
```
Slicing multiple sequences into views

Like many `Rle` operations, slice also works on `RleList`.

Slicing a `RleList`

```r
> sViewsList <- slice(sRleList, 4)
> sViewsList[1]

SimpleRleViewsList of length 1
$chr1
Views on a 78-length Rle subject

views:
  start   end  width
[1] 47 67 21 [ 4 5 5 6 6 6 7 ...]

Most `RleViews` methods also work on `RleViewsList`.
Summarizing windows

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

Sums

Maxima
Summarizing windows

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

### Sums

```r
> viewSums(sViews)
[1] 150  72

> viewSums(sViewsList)
SimpleNumericList of length 2
[["chr1"]]] 150
[["chr2"]]]  72
```
Summarizing windows

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

**Sums**

```r
> viewMaxs(sViews)
[1] 10  5
> viewMaxs(sViewsList)
SimpleNumericList of length 2
[["chr1"] 10
[["chr2"] 5
```
Summarizing windows

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

But how do we associate the summarized values with the original intervals?
Outline

1 Introduction

2 Sequences
   - Background
   - RLEs

3 Ranges
   - Basic Manipulation
   - Simple Transformations
   - Ranges as Sets
   - Overlap

4 Views

5 Interval Datasets
   - Motivation
   - RangedData Representation
   - Accessing interval data
Interval datasets

- 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
Motivation

Naive representation of interval dataset (1/2)

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

```r
> chr <- c("chr1", "chr2", "chr1")
> strand <- c("+", "+", "-")
> start <- c(3L, 4L, 1L)
> end <- c(7L, 5L, 3L)
> score <- c(1L, 3L, 2L)
> naiveTable <- data.frame(chr, strand, score, start, end)
> naiveTable

       chr strand score start end
   1  chr1    +     1    3   7
   2  chr2    +     3    4   5
   3  chr1    -     2    1   3
```
\textit{data.frame} objects are poorly suited for this data because operations are constantly performed within chromosome/contig.

\begin{verbatim}
Using by to loop over data.frame

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

naiveTable[["chr"]]: chr1
[1] 1 7
-------------------------------------

naiveTable[["chr"]]: chr2
[1] 4 5
\end{verbatim}
Instances are created using the `RangedData` constructor.

Interval starts and ends are wrapped in an `IRanges` constructor.

Chromosome/contig information is supplied to the `space` argument.

```r
c > rdTable <- RangedData(IRanges(start, end),
+  strand, score, space = chr)
```
RangedData sacrifices row order flexibility for efficiency.

**Code**

```r
> rdTable

RangedData with 3 rows and 2 value columns across 2 spaces

<table>
<thead>
<tr>
<th>space</th>
<th>ranges</th>
<th>strand</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;character&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;character&gt;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>1</td>
<td>chr1</td>
<td>[3, 7]</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>chr1</td>
<td>[1, 3]</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>chr2</td>
<td>[4, 5]</td>
<td>+</td>
</tr>
</tbody>
</table>
```
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.
Primary accessors

Get the ranges

```r
> ranges(rdTable)[1]

CompressedIRangesList of length 1
$chr1
IRanges of length 2
  start end width
[1]  3  7  5
[2]  1  3  3
```
Primary accessors

Get the ranges

Get the data values

```r
> values(rdTable)[1]

CompressedSplitDataFrameList of length 1
$chr1
Dataframe with 2 rows and 2 columns

  strand score
  <character> <integer>
  1    +    1
  2    -    2
```
Accessing built-in attributes

Each built-in feature attribute has a corresponding accessor method: start, end, strand, chrom, genome

Example

\texttt{\textgreater{} start(rdTable)}

\texttt{[1] 3 1 4}
Accessing interval data

Accessing data columns

Any data column (including strand) is accessible via $ and [].

Example

> rdTable$strand

[1] "+" "-" "+"
Overview of *RangedData* subsetting

- Often need to subset track features and data columns
- Example: limit the amount transferred to a genome browser
- Matrix style: $rd[i, j]$, where $i$ is feature index and $j$ is column index
- By chromosome: $rd[i]$, where $i$ indexes the chromosome
Subsetting examples and exercises

Examples

```r
> first10 <- rdTable[1:2, ]
> pos <- rdTable[rdTable$strand == "+", + ]
> chr1Table <- rdTable[1]
> scoreTable <- rdTable[, "score"]
```
Outline

1. Introduction
2. Sequences
   - Background
   - RLEs
3. Ranges
   - Basic Manipulation
   - Simple Transformations
   - Ranges as Sets
   - Overlap
4. Views
5. Interval Datasets
   - Motivation
   - RangedData Representation
   - Accessing interval data
Bridging the towers
Transitioning between *Rle(List)* and *RangedData*

Various paths between piecewise constant measures (*Rle(List)*) and interval datasets (*RangedData*)

- **Rle(List) to RangedData**
- Via *RleViews(List)*
- **RangedData to Rle(List)**
Bridging the towers
Transitioning between \textit{RleList} and \textit{RangedData}

\textbf{Rle(List) to RangedData}

\begin{verbatim}
> head(as(sRleList, "RangedData"), 3)

RangedData with 3 rows and 1 value column across 2 spaces

<table>
<thead>
<tr>
<th>space ranges</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;character&gt;</td>
<td>&lt;IRanges&gt;</td>
</tr>
<tr>
<td>chr1 [1, 40]</td>
<td>0</td>
</tr>
<tr>
<td>chr1 [41, 41]</td>
<td>1</td>
</tr>
<tr>
<td>chr1 [42, 43]</td>
<td>2</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{Via RleViews(List)}

\textbf{RangedData to Rle(List)}
**Bridging the towers**
Transitioning between *RleList* and *RangedData*

---

**Rle(List) to RangedData**

---

**Via RleViews(List)**

```r
> height <- unlist(viewMaxs(sViewsList))
> RangedData(sViewsList, height)
```

RangedData with 2 rows and 1 value column across 2 spaces

```
<table>
<thead>
<tr>
<th>space</th>
<th>ranges</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;character&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;numeric&gt;</td>
</tr>
<tr>
<td>1</td>
<td>chr1</td>
<td>[47, 67]</td>
</tr>
<tr>
<td>2</td>
<td>chr2</td>
<td>[ 8, 22]</td>
</tr>
</tbody>
</table>
```

---

**RangedData to Rle(List)**
Bridging the towers
Transitioning between *RleList* and *RangedData*

**Rle(List) to RangedData**

**Via RleViews(List)**

**RangedData to Rle(List)**

```r
> coverage(rdTable, weight = "score")[1]

SimpleRleList of length 1
$chr1
'integer' Rle of length 7 with 3 runs
  Lengths:  2 1 4
  Values :  2 3 1
```
Final Comments

- Just scratching the surface – much more under the hood. Exploration is encouraged.
- Trying to work around performance issues in R, but not entirely successful.
- Still in active development. Missing features or performance problems, let us know.