IRanges Package

Design overview and framing of its role in BioC

July 29, 2009

1. Introduction
   - Purpose
   - Data Structures
   - Functionality
   - Metadata
2. Sequences
   - RLEs
   - Typed Lists
   - Other
3. Ranges
   - Basics
   - Ranges as sets
   - Overlap
4. Data on Ranges
   - Views
   - RangedData
5. Future direction

Outline

1. Introduction
    - Purpose
    - Data Structures
    - Functionality
    - Metadata
2. Sequences
    - RLEs
    - Typed Lists
    - Other
3. Ranges
    - Basics
    - Ranges as sets
    - Overlap
4. Data on Ranges
    - Views
    - RangedData
5. Future direction

IRanges Purpose

- Fulfill low-level Bioconductor sequence analysis requirements.
  - Add new low-level utilities and classes not in vanilla R.
  - Supplant inefficient vanilla R functionality, particularly concerning long vectors (e.g. `window` function).
- Sits below *eSet*-like representations of sequence experiments in packages like *ShortRead*.

As such, package name is misleading (prefer *Seqbase*), but changing name would be costly to the BioC community.

Lessons Learned

- S4 classes are useful because they declare form, but...
  - Creating many S4 objects in R level loop takes time.
  - S4 object structure consumes memory, which can build up when there are lots of instantiated objects.
  - Class definitions can change and good to version instantiated object.
  - Can become too infatuated with multiple inheritances.
  - Avoid *initialize* methods, if possible. Use constructors instead.
- Testing is a developer's (and researcher's) best friend.
  - Validity methods provide important run-time data checking.
  - Automated (*RUnit*) tests make crucial refactoring possible.
- Don't let the perfect be the enemy of the good.

IRanges Data Structures

- S4 *Sequence* class
  - Mimics vector "class hierarchy"
  - Typed list objects
  - Data tables that can store S4 Sequence objects
  - Self-describing (think *Biobase*'s *AnnotatedDataFrame* metadata slots)
- Structures for compressing data
  - Run-length encodings (RLEs) (e.g. coverage vector)
  - Sparse list objects (e.g. read mapping information)
- Integer ranges/intervals
- Data on integer ranges/intervals

IRanges Functionality

- Implements the vector “interface” for *Sequence* objects
- Typed list object operations
    - Simple looping operations
    - Within and across object manipulations (e.g. Ops, Math, Summary group generics)
- Efficient operations on compressed data objects
    - Full suite of methods for RLE objects
    - Smart looping on compressed list objects
- Comprehensive integer ranges/interval operations
- Some functionality for data on ranges
    - Initial focus on subscripting, merging based on ranges, and *apply functionality.
    - Use cases may call for wider functionality.

Two types of metadata

- Whole object description (*list*)
- Element metadata (*DataFrame*)
- Currently this feature is severely underutilized; metadata can be passed from one object to another as data is processed.

Outline

1. Introduction
    - Purpose
    - Data Structures
    - Functionality
    - Metadata
2. Sequences
    - RLEs
    - Typed Lists
    - Other
3. Ranges
    - Basics
    - Ranges as sets
    - Overlap
4. Data on Ranges
    - Views
    - RangedData
5. Future direction

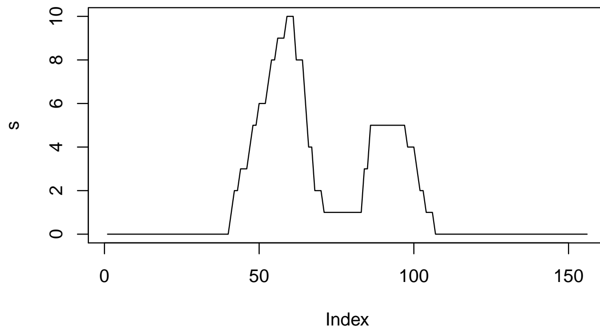
Sequence Subclasses

The *IRanges* package is chock full of *Sequence* subclasses:

Sequence Class Definition

```
> length(getClassDef("Sequence")@subclasses)

[1] 84

> head(names(getClassDef("Sequence")@subclasses),
+      8)

[1] "DataTable"       "AtomicList"
[3] "Rle"             "XSequence"
[5] "SimpleList"      "CompressedList"
[7] "DataFrameList"   "RangesList"

> slotNames(getClassDef("Sequence"))

[1] "elementMetadata" "elementType"
[3] "metadata"
```

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 instance 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 instance 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
```

Typed Lists

- Ordinary R list objects require element inspection and as such rarely used in method signature.
- Typed lists are list object whose elements inherit from a single class and more conducive to serve as method inputs.
- Typed lists in *IRanges* come in two basic flavors: “simple” and compressed (ideal for sparse lists).
- As with all *Sequence* classes, contain metadata slots.

List of Integers (1/2)

Typed list objects are well suited for method dispatch:

Simple List Type

```
> intList1 <- IntegerList(1:10, 1:100,
+       compress = FALSE)
> intList1

SimpleIntegerList: 2 elements

> (2L * intList1)[[1]]

 [1]  2  4  6  8 10 12 14 16 18 20

> intList2 <- IntegerList(11:20,
+       101:200, compress = FALSE)
> (intList1 + intList2)[[1]]

 [1] 12 14 16 18 20 22 24 26 28 30
```

List of Integers (2/2)

Compressed List Type

```
> xList <- lapply(1:1e+05, function(i) if (i%%100 ==
+     0) 1:10 else integer(0))
> cintList <- IntegerList(xList)
> system.time(sapply(xList, mean))

   user  system elapsed
  5.803   0.037   5.915

> system.time(sapply(cintList, mean))

   user  system elapsed
  0.797   0.018   0.816

> identical(sapply(xList, mean),
+     sapply(cintList, mean))

[1] TRUE
```

Sparse List of S4 Objects

Large lists of mostly empty S4 elements can take a large footprint:

Compressed List of Rle Objects

```
> empty <- Rle()
> empty

  'logical' Rle instance of length 0 with 0 runs
  Lengths:
  Values :

> print(object.size(lapply(1:1e+05,
+     function(i) empty)), units = "Mb")

69 Mb

> print(object.size(RleList(lapply(1:1e+05,
+     function(i) empty))), units = "Mb")

0.4 Mb
```

Other Sequence Types

- *DataTable* interface and *DataFrame* class
  - *data.frame* and *AnnotatedDataFrame* can't house S4 *Sequence* objects such as *Rle* and *IRanges*, *DNAStringSet*
  - A split version (*SplitDataFrameList*) can hold data across spaces (e.g. sequencing lanes, chromosomes, contigs, etc.).
- E**X**ternal 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
    - Purpose
    - Data Structures
    - Functionality
    - Metadata
2. Sequences
    - RLEs
    - Typed Lists
    - Other
3. Ranges
    - Basics
    - Ranges as sets
    - Overlap
4. Data on Ranges
    - Views
    - RangedData
5. Future direction

Ranges

- 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

Creating a Ranges object

The *IRanges* class is a simple *Ranges* implementation.

Code

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

Basic Ranges manipulation

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 Ranges manipulation

Subsetting

```
> ir[1:5]

IRanges instance:
    start end width
[1]     1  12    12
[2]     8  13     6
[3]    14  19     6
[4]    15  29    15
[5]    19  24     6
```

Normalizing ranges

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

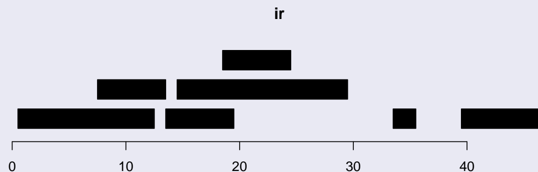
Code

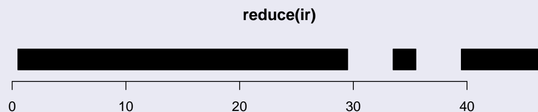
```
> reduce(ir)
```

Normalizing ranges

Code

```
> reduce(ir)
```





Set operations

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

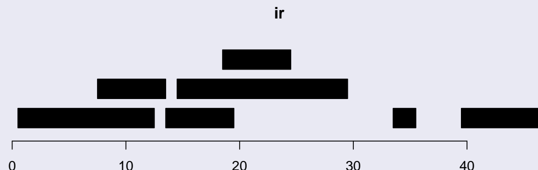
Example: gaps

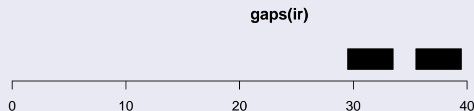
```
> gaps(ir)
```

Set operations

Example: gaps

```
> gaps(ir)
```





Disjoining ranges

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

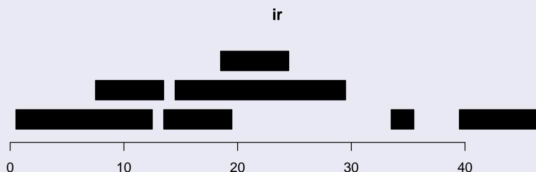
Code

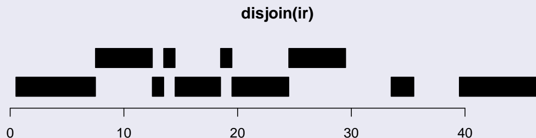
```
> disjoin(ir)
```

Disjoining ranges

Code

```
> disjoin(ir)
```





Overlap detection

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

Code

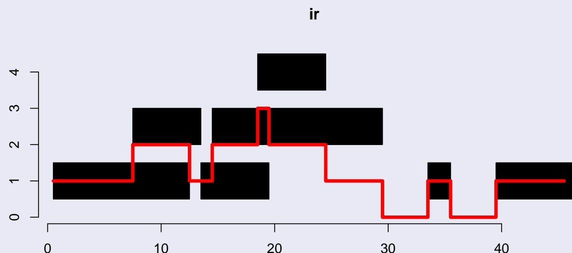
```
> ol <- overlap(reduce(ir), ir)
> as.matrix(ol)

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

Counting overlapping Ranges

coverage counts number of ranges over each position

Code

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



Finding nearest neighbors

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

Outline

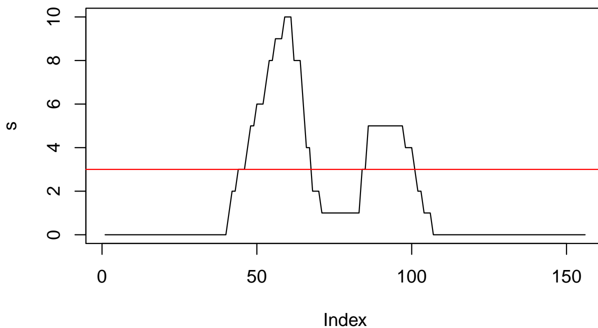
1. Introduction
   - Purpose
   - Data Structures
   - Functionality
   - Metadata
2. Sequences
   - RLEs
   - Typed Lists
   - Other
3. Ranges
   - Basics
   - Ranges as sets
   - Overlap
4. Data on Ranges
   - Views
   - RangedData
5. Future direction

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 ...]
[2]    86 100    15 [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: 2 elements

> viewMaxs(sViews)

[1] 10  5

> viewMaxs(sViewsList)

SimpleNumericList: 2 elements
```

RangedData

- Dataset where range is associated with a data row
- Holds ranges on multiple sequences (e.g. chromosomes/contigs)
- 3D data structure that departs from R conventions
  - In some context, feels like a list
  - In others, feels like a data.frame
- Serves as basic data structure for *rtracklayer*

Outline

1. Introduction
   - Purpose
   - Data Structures
   - Functionality
   - Metadata
2. Sequences
   - RLEs
   - Typed Lists
   - Other
3. Ranges
   - Basics
   - Ranges as sets
   - Overlap
4. Data on Ranges
   - Views
   - RangedData
5. Future direction

Short/Medium Term Goals

- Document biological sequencing experiment components in an *IRanges* context.
  - Genome browser track(s) = *RangedData*/*RangedDataList*
  - Coverage across chromosomes = *RleList*
  - Mapped ranges to genome = *CompressedIRangesList*
  - Data (sans ranges) across chroms = *SplitDataFrameList*
- Backfill functionality in current hot classes.
  - Add kernel smoother methods for *Rle*/*RleList*.
  - Further define *RangedData*.
- Optimize performance at choke points. (Accumulating coverage too slow?)
- Create (multiple) alignment data class and methods.