IRanges and *GenomicRanges*
An introduction

Kasper Daniel Hansen
<khansen@jhsph.edu>

CSAMA, Brixen 2011

Why you should care

IRanges and GRanges are data structures I use often to solve a variety of problems, typically related to annotating the genome.

These data structures are very fast and efficient.

Every R user who deals with genomic data should master these packages.

These two packages are themselves compelling arguments for learning R!

Aims

We provide an overview of the two packages *IRanges* and *GenomicRanges*.

At first, these two packages can appear daunting, with many classes and functions. Luckily, things are not as hard as the first appear.

Both packages now contain very extensive and useful vignettes. Read them!

Idea

A surprising amount of objects/tasks in computational biology can be formulated in terms of integer intervals, manipulation of integer intervals and overlap of integer intervals.

Objects: A transcript (a union of integer intervals), a collection of SNPs (intervals of width 1), TF binding sites, a collection of aligned short reads.

Tasks: Which TF binding sites hit the promoter of genes (overlap between two sets of intervals), which SNPs hit a collection of exons, which short reads hit a predetermined set of exons.

IRanges are collections of integer intervals. GRanges are like IRanges, but with an associated chromosome and strand, taking care of some book keeping.

Basic IRanges

Specify IRanges by 2 of start, end, width (SEW).

```
> library(IRanges)
> ir1 <- IRanges(start = c(1,3,5), end = c(3,5,7))
> ir1
```

```
IRanges of length 3
    start end width
[1]     1   3     3
[2]     3   5     3
[3]     5   7     3
```

```
> ir2 <- IRanges(start = c(1,3,5), width = 3)
> all.equal(ir1, ir2)
```

```
[1] TRUE
```

Assessor methods: start(), end(), width() and also replacement methods

```
> width(ir2) <- 1
> ir2
```

Basic IRanges

```
IRanges of length 3
    start end width
[1]     1   1     1
[2]     3   3     1
[3]     5   5     1
```

They may have names

```
> names(ir1) <- paste("A", 1:3, sep = "")
> ir1
```

```
IRanges of length 3
    start end width names
[1]     1   3     3    A1
[2]     3   5     3    A2
[3]     5   7     3    A3
```

and has a single dimension

```
> dim(ir1)
```

```
NULL
```

```
> length(ir1)
```

```
[1] 3
```

Aside: GRanges

GRanges are like IRanges but with a strand and a chromosome (seqnames)

```
> gr <- GRanges(seqnames = "chr1", strand = c("+", "-", "+"), ra
> gr
```

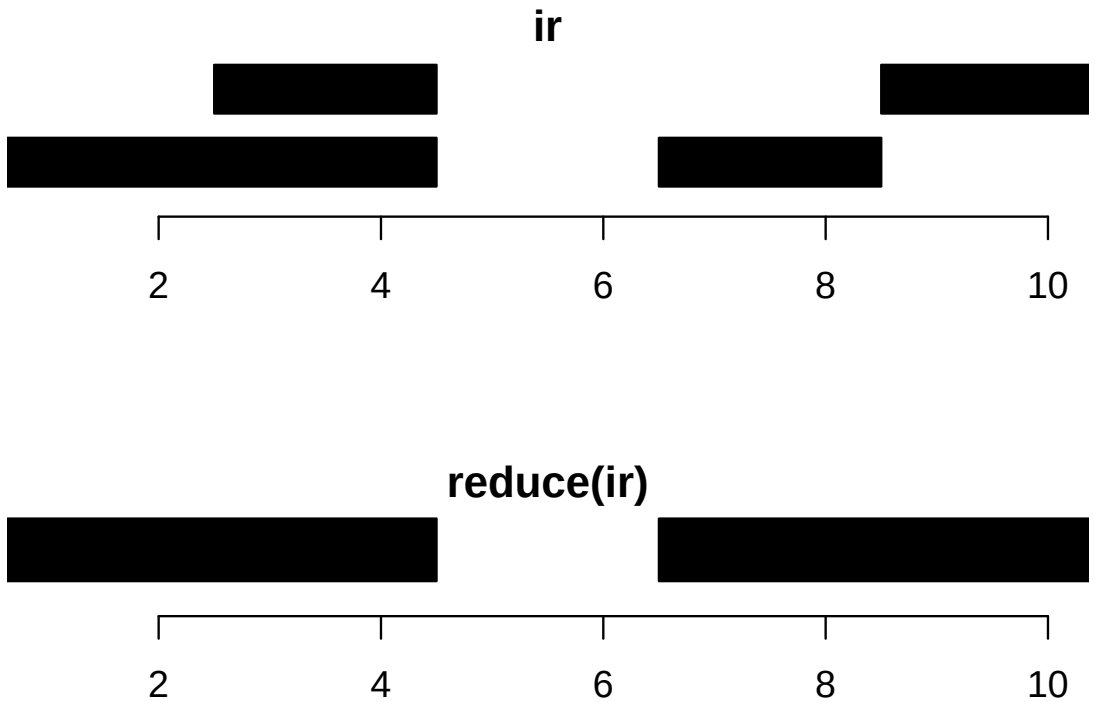
```
GRanges with 3 ranges and 0 elementMetadata values
   seqnames    ranges strand |
      <Rle> <IRanges>  <Rle> |
A1     chr1    [1, 3]      + |
A2     chr1    [3, 5]      - |
A3     chr1    [5, 7]      + |

seqlengths
 chr1
   NA
```

There are some additional differences, we will return to GRanges later.

Normal IRanges

A normal IRanges is a minimal representation of the IRanges viewed as a set. Each integer only occur in a single range and there are as few ranges as possible. In addition, it is ordered. Created by reduce().



Normal IRanges

```
> ir
```

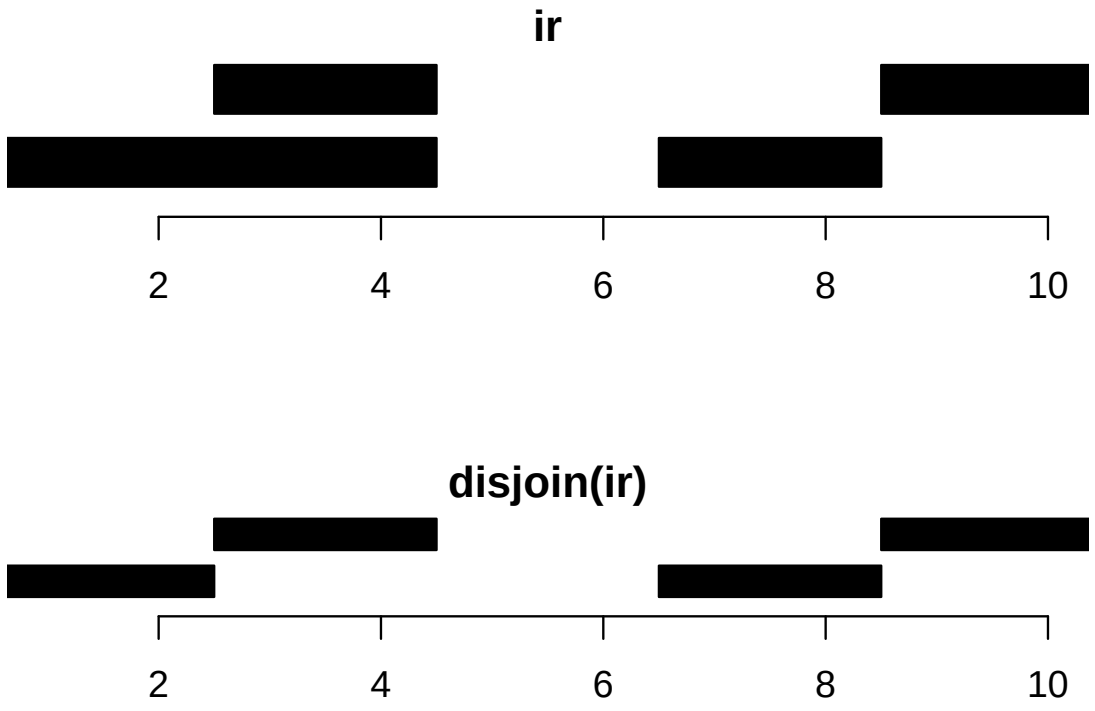
```
IRanges of length 4
    start end width
[1]     1   4     4
[2]     3   4     2
[3]     7   8     2
[4]     9  10     2
```

```
> reduce(ir)
```

```
IRanges of length 2
    start end width
[1]     1   4     4
[2]     7  10     4
```

"Which bases belong to an exon?"

Disjoin



"Which bases belong to the same set of exons?"

Manipulating IRanges

- shift(), narrow(), flank(), shift(), narrow(), restrict()
- union(), intersect(), setdiff() gaps() (returns normalized IRanges)
- punion(), pintersect(), psetdiff(), pgap()

Finding Overlaps

Finding (pairwise) overlaps between two IRanges is done by findOverlaps. This function is amazingly fast!

```
> ir1 <- IRanges(start = c(1,4,8), end = c(3,7,10))
> ir2 <- IRanges(start = c(3,4), width = 3)
> ov <- findOverlaps(ir1, ir2)
> ov
```

```
An object of class "RangesMatching"
Slot "matchMatrix":
     query subject
[1,]     1       1
[2,]     2       1
[3,]     2       2

Slot "DIM":
[1] 3 2
```

```
> matchMatrix(ov)
```

Finding Overlaps

```
     query subject
[1,]     1       1
[2,]     2       1
[3,]     2       2
```

queryHits(), subjectHits() (often used with unique())

For efficiency, there is also countOverlaps()

```
> countOverlaps(ir1, ir2)
```

```
[1] 1 2 0
```

```
> args(findOverlaps)
```

```
function (query, subject, maxgap = 0L, minoverlap = 1L, type = c
    "start", "end", "within", "equal"), select = c("all", "first
    "last", "arbitrary"), ...)
NULL
```

Finding nearest IRanges

nearest(), precede(), follow(). Watch out for ties!

```
> ir1
```

```
IRanges of length 3
    start end width
[1]     1   3     3
[2]     4   7     4
[3]     8  10     3
```

```
> ir2
```

```
IRanges of length 2
    start end width
[1]     3   5     3
[2]     4   6     3
```

```
> nearest(ir1, ir2)
```

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

IRangesList

An IRangesList is simply a list of IRanges.

Why a separate class for this?

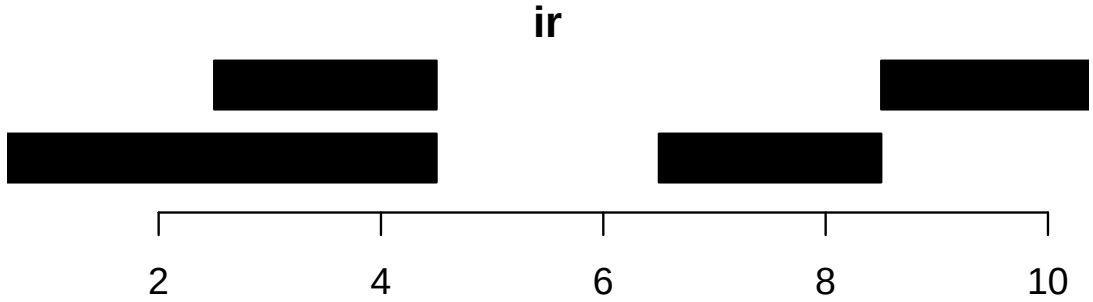
- Methods dispatching, "we know what we get".
- Easy to specify input and output.

Many XXList exists.

One biology use case: the set of transcripts of a gene. Each transcript is an IRanges, but we need something to glue the transcripts together.

Rle

Rle are "run-length encoded" vectors, which are very useful for representing coverage vectors efficiently.




```
> coverage(ir)
'integer' Rle of length 10 with 4 runs
  Lengths: 2 2 2 4
  Values : 1 2 0 1
> as.vector(coverage(ir))
 [1] 1 1 2 2 0 0 1 1 1 1
```

runLengths(), runValues()

Computing on Rle

Rle's can be extremely efficient, especially for really long vectors. They support are a wide variety of operations

```
> coverage(ir1) + coverage(ir2)
```

```
'integer' Rle of length 10 with 7 runs
  Lengths: 2 1 2 1 2 1 1
  Values : 1 2 3 2 1 2 3
```

```
> range(coverage(ir2))
```

```
[1] 0 2
```

```
> log(coverage(ir2))
```

```
'numeric' Rle of length 6 with 4 runs
  Lengths:                   2 ...
  Values :                -Inf ...
```

aggregate() allows you to apply functions to the Rle inside an IRanges

```
> aggregate(coverage(ir2), IRanges(start = 3, width = 4),
+           FUN = mean)
```

```
[1] 1.5
```

Views

Often we have a big object (think genome) and we are interested in subsets of this object.
These subsets may be thought of as IRanges inside the object. Think exons and genome sequence. Key idea: store big object and just keep the ranges.

```
> library(BSgenome.Scerevisiae.UCSC.sacCer1)
> vi <- Views(Scerevisiae$chr1,
+             start = seq(1, by = 10000, length = 20),
+             width = 1000)
> head(vi, n = 4)
```

```
  Views on a 230208-letter DNAString subject
subject: CCACACCACACCCACACAC...TGGGTGTGGTGTGTGTGGG
views:
    start   end width
[1]     1  1000  1000 [CCACACCACACC...GGGTGCTATGA]
[2] 10001 11000  1000 [AGAATGAATCGG...TGGATATAAAT]
[3] 20001 21000  1000 [ATTATAATCCTC...CGGTCTGGTGT]
[4] 30001 31000  1000 [GATTTGTTCAAA...TTTTTTTGTT]
```

Views

Views can also be made on coverage vectors

```
> coverage(ir)

'integer' Rle of length 10 with 4 runs
  Lengths: 2 2 2 4
  Values : 1 2 0 1

> Views(coverage(ir), start = 3, width = 4)

Views on a 10-length Rle subject

views:
    start end width
[1]     3   6     4 [2 2 0 0]
```

IRanges: important types of objects

- IRanges (and normal IRanges), IRangesList
- Rle
- Views
- RangedData: IRanges with metadata, think IRanges + data.frame

These types of objects allow you to accomplish many useful tasks in a very efficient manner, but they sometimes require a bit creativity.

Return to GRanges

GRanges are like IRanges with strand and chromosome. Strand can be "+", "-" and "*" (unknown strand or unstranded).

```
> gr
```

```
GRanges with 3 ranges and 0 elementMetadata values
   seqnames    ranges strand |
      <Rle> <IRanges>  <Rle> |
A1     chr1    [1, 3]      + |
A2     chr1    [3, 5]      - |
A3     chr1    [5, 7]      + |

seqlengths
 chr1
   NA
```

strand(), seqnmes(), ranges(), start(), end(), width().
They operate within a universe of seqlengths.

```
> seqlengths(gr) <- c("chr1" = 10)
> gaps(gr)
```

Return to GRanges

```
GRanges with 5 ranges and 0 elementMetadata values
    seqnames    ranges strand |
       <Rle> <IRanges>  <Rle> |
[1]     chr1   [4,  4]      + |
[2]     chr1   [8, 10]      + |
[3]     chr1   [1,  2]      - |
[4]     chr1   [6, 10]      - |
[5]     chr1   [1, 10]      * |

seqlengths
 chr1
   10
```

And because the have strand, some operations are now relative to the direction of transcription (upstream(), downstream()):

```
> flank(gr, 2, start = FALSE)
```

Return to GRanges

```
GRanges with 3 ranges and 0 elementMetadata values
   seqnames    ranges strand |
      <Rle> <IRanges>  <Rle> |
A1     chr1    [4, 5]      + |
A2     chr1    [1, 2]      - |
A3     chr1    [8, 9]      + |

seqlengths
 chr1
   10
```

Finally, GRanges may have associated metadata.

```
> values(gr) <- DataFrame(score = c(0.1, 0.5, 0.3))
> gr
```

Return to GRanges

```
GRanges with 3 ranges and 1 elementMetadata value
   seqnames    ranges strand |     score
      <Rle> <IRanges>  <Rle> | <numeric>
A1     chr1    [1, 3]      + |       0.1
A2     chr1    [3, 5]      - |       0.5
A3     chr1    [5, 7]      + |       0.3

seqlengths
 chr1
   10
```

values(), elementMetadata()

Usecase I

Suppose we want to identify TF binding sites that overlaps known SNPs.

snps: GRanges (of width 1)
TF: GRanges

```
> findOverlaps(snps, TF)
```

(watch out for strand)

Usecase II

Suppose we have a set of DMRs (think genomic regions) and a set of CpG Islands and we want to find all DMRs within 10kb of a CpG Island.

dmrs: GRanges
islands: GRanges

```
> findOverlaps(dmrs, resize(islands,
+         width = 20000 + width(islands), fix = "center"))
```

(watch out for strand)

Usecase III

Suppose we want to compute the average coverage of bases belonging to (known) exons.

reads: GRanges exons: GRanges

```
> bases <- reduce(exons)
> nBases <- sum(width(bases))
> nCoverage <- sum(Views(coverage(reads), bases))
> nCoverage / nBases
```

(watch out for strand)

Session Info

- R version 2.14.0 Under development (unstable) (2011-06-20 r56188), `x86_64-apple-darwin10.7.4`
- Locale: `en_US.utf-8/en_US.utf-8/en_US.utf-8/C/en_US.utf-8/en_US.utf-8`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: Biostrings 2.21.6, BSgenome 1.21.3, BSgenome.Scerevisiae.UCSC.sacCer1 1.3.17, GenomicRanges 1.5.12, IRanges 1.11.10
- Loaded via a namespace (and not attached): tools 2.14.0