

*IRanges, GenomicRanges, and Biostrings*  
*Bioconductor* Infrastructure Packages for Sequence Analysis

Patrick Aboyoun

Fred Hutchinson Cancer Research Center

7-9 June, 2010

# Outline

Introduction

Genomic Intervals with Data

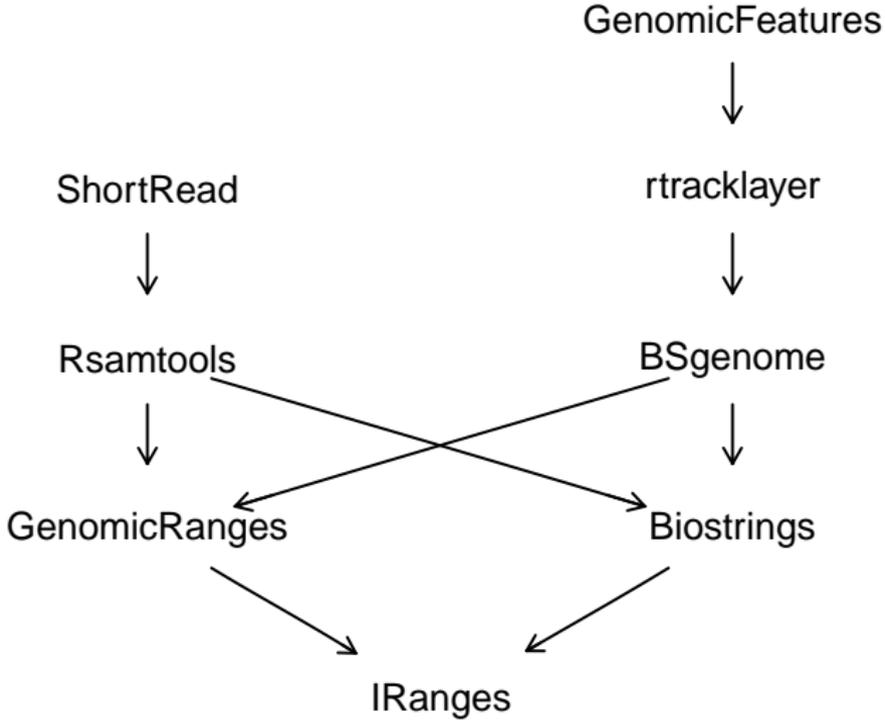
Coverage and Other Piecewise Constant Measures

Long Biological Strings

Developer's Notes

Resources

# Bioconductor Sequence Packages



# Bioconductor Sequence Infrastructure Packages

## *IRanges*

- ▶ Long sequences (compressed & pointer referenced)
- ▶ Views on long sequences
- ▶ Integer interval tools (e.g. interval overlap)
- ▶ Genomic intervals I (*RangedData*)

## *GenomicRanges*

- ▶ Genomic intervals II (*GenomicRanges*)
- ▶ Discontiguous genomic interval sets (*GenomicRangesList*)

## *Biostrings*

- ▶ Long DNA/RNA/amino acids sequences
- ▶ Sequence & PWM matching and pairwise alignment tools

# Bioconductor Sequence Infrastructure Classes

Ranges (as sequences & intervals)

*IRanges*

Genomic intervals with data

*GRanges*, *RangedData*

Genomic interval sets (e.g. spliced transcripts)

*GRangesList*

Long piecewise constant sequences

*Rle*, *RleList*

Long (biological) strings

*DNAString*, *RNAString*, *AAString*, *BString*, *DNAStringSet*, ...

Views on long sequences

*RleViews*, *RleListViews*, *XStringViews*, ...

# Concept I: Run-Length Encoding (RLE)

## Issue

- ▶ Chromosomes can be hundreds of million of base pairs long, making them hard to manage in computer memory.
- ▶ Fortunately, coverage vectors tend to follow an integer step function.

## Solution

- ▶ Run-length encoding (RLE) is a common compression technique for storing long sequences with lengthy repeats.
- ▶ An RLE couples values with run lengths, e.g. the vector 0, 0, 0, 1, 1, 2 would be represented as (3) 0's, (2) 1's, and (1) 2.
- ▶ The *IRanges* package uses the *Rle* and *RleList* classes to house coverage vectors.

# Concept II: Sequence Views

## Issue

- ▶ Chromosomes can be hundreds of million of base pairs long, making subsequence selection inefficient.

## Solution

- ▶ Store the original sequence using a pass-by-reference semantic.
- ▶ Associate ranges with the sequence to select subsequence.
- ▶ Example:
  - ▶ 7007-letter sequence:  
    <<SNIP-3000>>AGATTCA<<SNIP-4000>>
  - ▶ View range: [3001, 3007]
  - ▶ => 7-letter subsequence: AGATTCA

# Outline

Introduction

**Genomic Intervals with Data**

Coverage and Other Piecewise Constant Measures

Long Biological Strings

Developer's Notes

Resources

# Naive representation for intervals with data

## Data characteristics

- ▶ Genomic coordinates consist of chromosome, position, and potentially strand information
- ▶ May have additional values, such as GC content or alignment coverage

## *data.frame* approach

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

# BioC representations for intervals with data

## *GRanges*

- ▶ Used by *GenomicFeatures*, a transcript annotation generator
- ▶ Intervals not required to be grouped by chromosome/contig
- ▶ Methods strand aware
- ▶ *GRangesList* class can hold exons within spliced transcripts

## *RangedData*

- ▶ Used by *rtracklayer*, a genome browser interface
- ▶ Intervals grouped by chromosome/contig
- ▶ Methods strand unaware
- ▶ Preceded *GRanges* class

# GRanges construction

## GRanges constructor

- ▶ Instances are created using the `GRanges` constructor.
- ▶ Starts and ends are wrapped in an `IRanges` constructor.
- ▶ Chromosome/contig supplied to `seqnames` argument.
- ▶ Underlying sequence lengths can be supplied to `seqlengths` argument.

## GRanges example

```
> bioc <- GRanges(seqnames = chr,  
+                 ranges = IRanges(start = start, end = end),  
+                 strand = strand,  
+                 seqlengths = c("chr1" = 24, "chr2" = 18))
```

# GRanges display

## GRanges show method

```
> bioc
```

```
GRanges with 3 ranges and 0 elementMetadata values
```

	seqnames	ranges	strand	
	<Rle>	<IRanges>	<Rle>	
[1]	chr1	[3, 7]	+	
[2]	chr2	[4, 5]	+	
[3]	chr1	[1, 3]	-	

```
seqlengths
```

chr1	chr2
24	18

## Note

- ▶ Optional interval data would appear to the right of | divider.

# GRanges class decomposition

## GRanges slots

```
> getSlots("GRanges")
```

seqnames	ranges	strand	seqlengths
"Rle"	"IRanges"	"Rle"	"integer"
elementMetadata	elementType	metadata	
"ANY"	"character"	"list"	

## Notes

- ▶ If (mostly) sorted, *Rle* vectors reduce memory usage and provide faster group selection
- ▶ `elementMetadata` holds optional interval data
- ▶ `metadata` holds optional whole object info

# Interval operations

## Intra-interval

flank, resize, shift

## Inter-interval I

disjoin, gaps, reduce, range

## Inter-interval II

coverage

## Between two interval sets I

union, intersect, setdiff

## Between two interval sets II

punion, pintersect, psetdiff

## Between two interval sets III

findOverlaps, countOverlaps, %in%, match

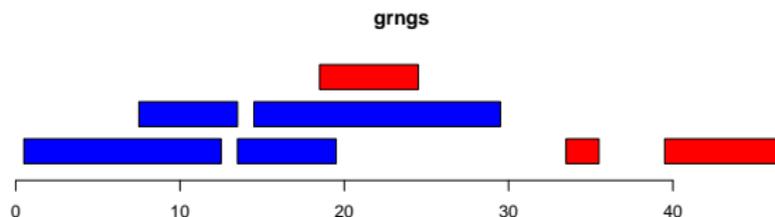
## Low level

start, end, width

# Creating a new *GRanges* 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))  
> grngs <- GRanges(seqnames = "chr1", ranges = ir,  
+                 strand = strand,  
+                 seqlengths = c("chr1" = 50))
```



blue = positive strand, red = negative strand

# GRanges subsetting

## seqselect

```
> seqselect(grngs, strand(grngs) == "--")
```

GRanges with 3 ranges and 0 elementMetadata values

	seqnames	ranges	strand	
	<Rle>	<IRanges>	<Rle>	
[1]	chr1	[19, 24]	-	
[2]	chr1	[34, 35]	-	
[3]	chr1	[40, 46]	-	

## seqlengths

```
chr1  
50
```

## Other functions

[, head, tail, window, subset, subsetByOverlaps

# Intra-interval (1/2)

## Shifting intervals

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

```
> shift(grngs, 1)
```

```
GRanges with 7 ranges and 0 elementMetadata values
```

	seqnames	ranges	strand	
	<Rle>	<IRanges>	<Rle>	
[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]	-	

```
seqlengths
```

```
chr1  
50
```

## Intra-interval (2/2)

### Resizing intervals

“Growing” alignment intervals to an estimated fragment length.

```
> resize(grngs, 10)
```

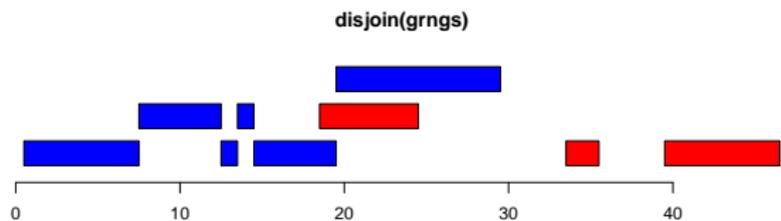
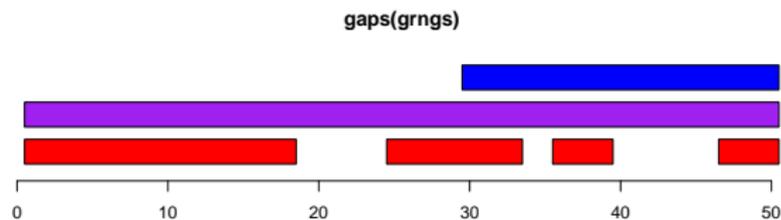
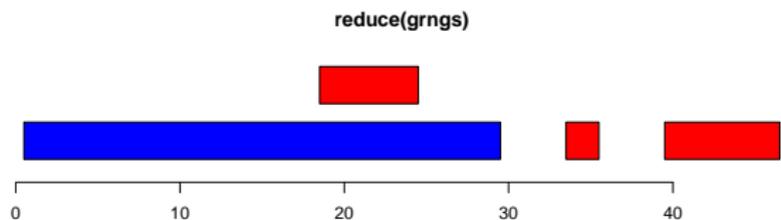
```
GRanges with 7 ranges and 0 elementMetadata values
```

	seqnames	ranges	strand	
	<Rle>	<IRanges>	<Rle>	
[1]	chr1	[ 1, 10]	+	
[2]	chr1	[ 8, 17]	+	
[3]	chr1	[14, 23]	+	
[4]	chr1	[15, 24]	+	
[5]	chr1	[15, 24]	-	
[6]	chr1	[26, 35]	-	
[7]	chr1	[37, 46]	-	

```
seqlengths
```

```
chr1  
50
```

# Inter-interval I



# Overlap detection

## Finding interval overlaps

`findOverlaps` and `countOverlaps` produce a mapping and a tabulation of interval overlaps, respectively

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

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

```
> countOverlaps(reduce(grngs), grngs)
```

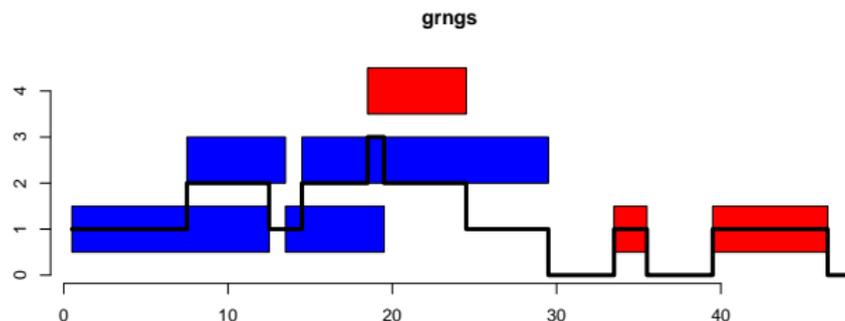
```
[1] 4 1 1 1
```

# Elementwise counts of overlapping intervals

## Coverage

- ▶ coverage counts number of ranges over each position
- ▶ Subset by strand to get stranded coverage

```
> cover <- coverage(grngs)
```



# Outline

Introduction

Genomic Intervals with Data

Coverage and Other Piecewise Constant Measures

Long Biological Strings

Developer's Notes

Resources

# Piecewise constant measures

## Issue restated

- ▶ 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”)

## *Rle* and *RleList* classes

- ▶ Solve the set of problems for positional measures that tend to have consecutively repeating values.
- ▶ *Do not* address the more general problem of positional measures that constantly fluxuate, such as conservation scores.

## Numerous *R*/*e* methods (1/2)

[1]	"!"	"["	"[<-"	"%in%"
[5]	"aggregate"	"as.character"	"as.complex"	"as.data.frame"
[9]	"as.factor"	"as.integer"	"as.logical"	"as.numeric"
[13]	"as.raw"	"as.vector"	"c"	"chartr"
[17]	"coerce"	"Complex"	"cor"	"cov"
[21]	"diff"	"end"	"findRange"	"findRun"
[25]	"gsub"	"IQR"	"is.na"	"is.unsorted"
[29]	"length"	"levels"	"levels<-"	"mad"
[33]	"match"	"Math"	"Math2"	"mean"
[37]	"median"	"nchar"	"nrun"	"Ops"
[41]	"paste"	"pmax"	"pmax.int"	"pmin"
[45]	"pmin.int"	"quantile"	"rep"	"rep.int"
[49]	"rev"	"runLength"	"runLength<-"	"runmean"
[53]	"runmed"	"runq"	"runsum"	"runValue"
[57]	"runValue<-"	"runwtsum"	"sd"	"seqselect"
[61]	"seqselect<-"	"shiftApply"	"show"	"slice"
[65]	"smoothEnds"	"sort"	"split"	"splitRanges"
[69]	"start"	"sub"	"substr"	"substring"
[73]	"summary"	"Summary"	"table"	"tolower"
[77]	"toupper"	"unique"	"var"	"Views"
[81]	"which"	"width"	"window"	

# Numerous *R*/*e* methods (2/2)

## Arith

`+`, `-`, `*`, `^`, `%%`, `%/%`, `/`

## Compare

`==`, `>`, `<`, `!=`, `<=`, `>=`

## Logic

`&`, `|`

## Math

`abs`, `sign`, `sqrt`, `ceiling`, `floor`, `trunc`, `cummax`, `cummin`, `cumprod`,  
`cumsum`, `log`, `log10`, `log2`, `log1p`, `acos`, `acosh`, `asin`, `asinh`, ...

## Math2

`round`, `signif`

## Summary

`max`, `min`, `range`, `prod`, `sum`, `any`, `all`

## Complex

`Arg`, `Conj`, `Im`, `Mod`, `Re`

## Coverage example

Coverage from a *Saccharomyces cerevisiae* (Yeast) experiment contained in two objects `posCover` & `negCover`

```
> c(class(posCover), class(negCover))
```

```
[1] "SimpleRleList" "SimpleRleList"
```

```
> posCover[["chrI"]]
```

```
'integer' Rle of length 230208 with 4814 runs
```

```
Lengths: 410 150 1470 150 269 ... 121 5 24 1252
```

```
Values : 0 1 0 1 0 ... 3 2 1 0
```

```
> negCover[["chrI"]]
```

```
'integer' Rle of length 230208 with 4641 runs
```

```
Lengths: 2267 22 128 22 456 ... 150 914 150 62
```

```
Values : 0 1 2 1 0 ... 1 0 1 0
```

```
> posCover[["chrI"]] + negCover[["chrI"]]
```

```
'integer' Rle of length 230208 with 8761 runs
```

```
Lengths: 410 150 1470 150 87 ... 126 914 150 62
```

```
Values : 0 1 0 1 0 ... 1 0 1 0
```

# Plotting coverage

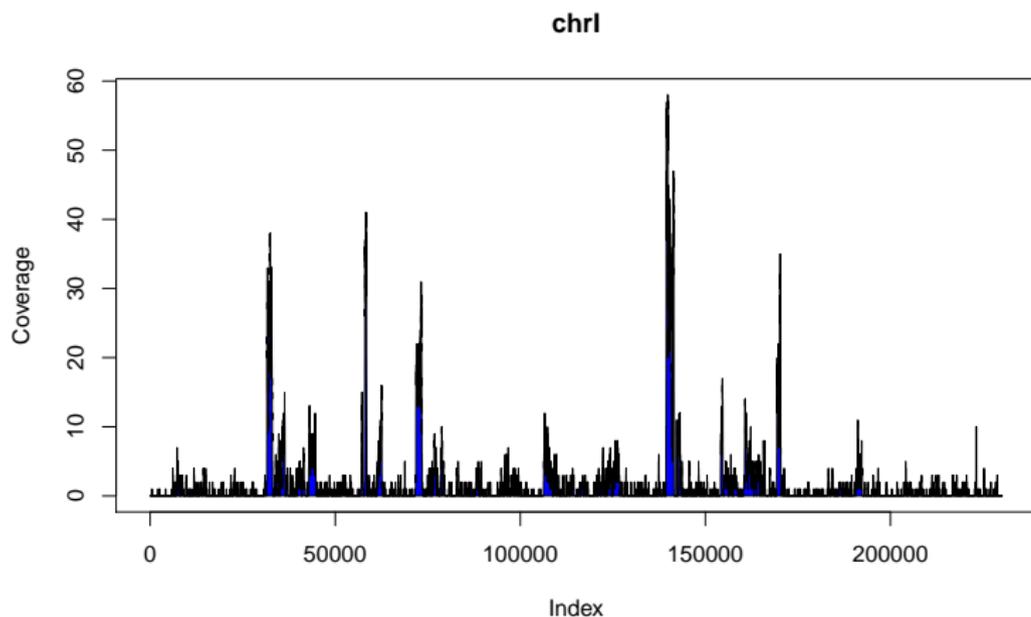
## Custom function

```
> plotCoverage <-  
+ function(x, chrom, start=1, end=length(x[[chrom]]), col="blue",  
+         xlab="Index", ylab="Coverage", main=chrom)  
+ {  
+   xWindow <- as.vector(window(x[[chrom]], start, end))  
+   x <- start:end  
+   xlim <- c(start, end)  
+   ylim <- c(0, max(xWindow))  
+   plot(x = start, y = 0, xlim = xlim, ylim = ylim,  
+        xlab = xlab, ylab = ylab, main = main, type = "n")  
+   polygon(c(start, x, end), c(0, xWindow, 0), col = col)  
+ }
```

# Plotting coverage on one strand

Plotting chr1+ coverage

```
> plotCoverage(posCover, "chr1")
```



# Plotting stranded coverage

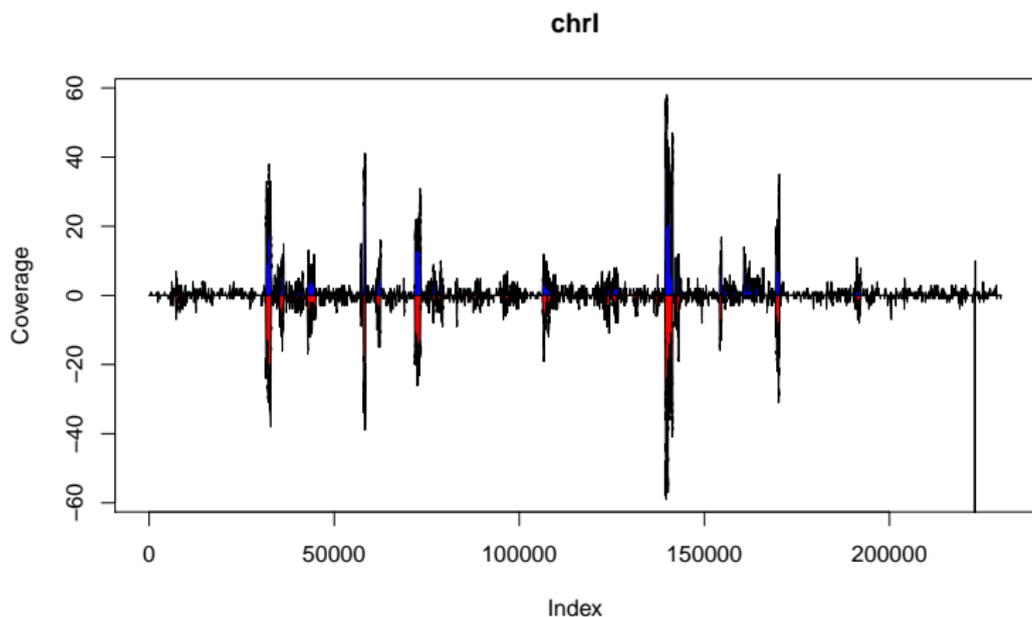
## Custom function

```
> plotCoverageStrands <-  
+ function(pos, neg, chrom, start=1, end=length(pos[[chrom]]),  
+         pos.col="blue", neg.col="red", xlab="Index",  
+         ylab="Coverage", main=chrom)  
+ {  
+   posWindow <- as.vector(window(pos[[chrom]], start, end))  
+   negWindow <- as.vector(window(neg[[chrom]], start, end))  
+   x <- start:end  
+   xlim <- c(start, end)  
+   ylim <- c(-1, 1) * min(max(posWindow), max(negWindow))  
+   plot(x = start, y = 0, xlim = xlim, ylim = ylim,  
+        xlab = xlab, ylab = ylab, main = main, type = "n")  
+   polygon(c(start, x, end), c(0, posWindow, 0), col = pos.col)  
+   polygon(c(start, x, end), c(0, -negWindow, 0), col = neg.col)  
+ }
```

# Plotting coverage on both strands

Plotting chr1 coverage, both strands

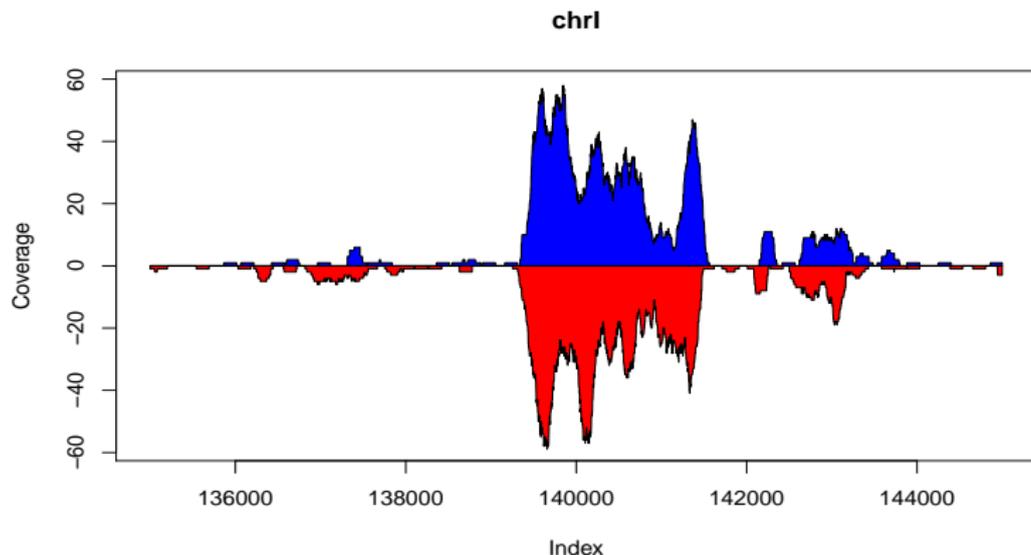
```
> plotCoverageStrands(posCover, negCover, "chr1")
```



# Plotting Coverage on both strands

## Plotting chr1 coverage, both strands

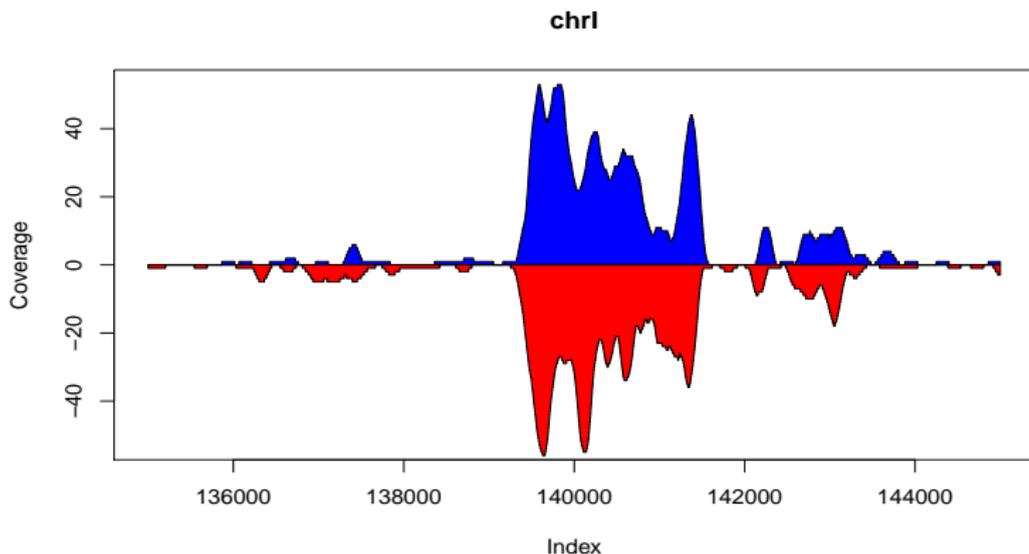
```
> plotCoverageStrands(posCover, negCover, "chr1", 135000, 145000)
```



# Smoothing coverage

## Running window mean

```
> posSmoothCover <- round(runmean(posCover, 75, endrule = "constant"))  
> negSmoothCover <- round(runmean(negCover, 75, endrule = "constant"))  
> plotCoverageStrands(posSmoothCover, negSmoothCover, "chrI", 135000, 145000)
```



# Combining coverage

## Combining coverage using "parallel" minimums

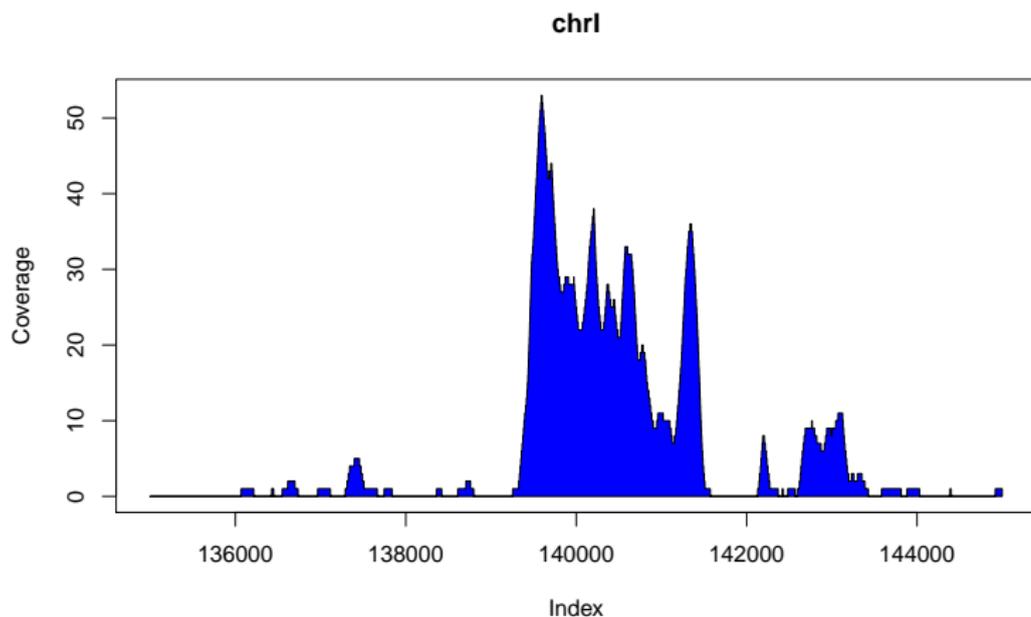
```
> combSmoothCover <- mendoapply(pmin,  
+                               posSmoothCover,  
+                               negSmoothCover)  
> identical(class(posSmoothCover), class(combSmoothCover))  
  
[1] TRUE
```

- ▶ The `mendoapply` function defined in *IRanges* packages as a member of the `apply` family.
  - ▶ Performs elementwise operations across multiple inputs of the same type.
  - ▶ Returns an object of the same type as the inputs.
- ▶ The minimum coverage value on either strand can be computed using `pmin`.

# Plotting combined coverage

## Plotting chr1, combined strands

```
> plotCoverage(combSmoothCover, "chr1", 135000, 145000)
```



## Island selection

```
> islands <- slice(combsmoothCover, lower=1)
> islandsWithWidePeaks <- islands[viewMaxs(islands) >= 8L &
+                               width(islands) >= 500L]
> islandsWithWidePeaks
```

SimpleRleViewsList of length 16

\$chrI

Views on a 230208-length Rle subject

views:

	start	end	width	
[1]	31430	33109	1680	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[2]	35071	36420	1350	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[3]	42744	44893	2150	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[4]	57855	58490	636	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[5]	61291	62604	1314	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[6]	71685	73511	1827	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[7]	106221	108629	2409	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[8]	139254	141574	2321	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[9]	142594	143428	835	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[10]	154057	154762	706	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]
[11]	169261	170429	1169	[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...]

...

<15 more elements>

## Common methods for *Views* objects

- ▶ Subset via `[]`, `[[`, etc.
- ▶ Manage edge cases via `trim` & `restrict`
- ▶ *Ranges* operations such as `start`, `end`, `width`, etc.
- ▶ Perform within view calculations via `viewSums`, `viewMins`, `viewMaxs`, `viewWhichMins`, `viewWhichMaxs`, `viewApply`

# Outline

Introduction

Genomic Intervals with Data

Coverage and Other Piecewise Constant Measures

**Long Biological Strings**

Developer's Notes

Resources

# Long biological string framework

## Biostrings string types

```
> library(Biostrings)
> names(completeSubclasses(getClass("XString")))

[1] "BString"    "DNAString" "RNAString" "AAString"
```

## DNA

```
> data(yeastSEQCHR1)
> c(class(yeastSEQCHR1), nchar(yeastSEQCHR1))

[1] "character" "230208"

> yeast1 <- DNAString(yeastSEQCHR1)
> yeast1

230208-letter "DNAString" instance
seq: CCACACCACACCCACACACCCACACACC...GGTGTGGTGTGGGTGTGGTGTGTGTGGG

> IUPAC_CODE_MAP

      A      C      G      T      M      R      W      S      Y
  "A"    "C"    "G"    "T"    "AC"   "AG"   "AT"   "CG"   "CT"
      K      V      H      D      B      N
  "GT"   "ACG"  "ACT"  "AGT"  "CGT"  "ACGT"
```

# List of strings

## Biostrings string list types

```
> head(names(completeSubclasses(getClass("XStringSet"))), 4)
[1] "BStringSet"   "DNABStringSet" "RNABStringSet" "AAStringSet"
```

## DNA strings

```
> data(srPhiX174)
> length(srPhiX174)
```

```
[1] 1113
```

```
> head(srPhiX174, 3)
```

```
  A DNABStringSet instance of length 3
  width seq
```

```
[1]   35 GTTATTATACCGTCAAGGACTGTGTGACTATTGAC
[2]   35 GGTGGTTATTATACCGTCAAGGACTGTGTGACTAT
[3]   35 TACCGTCAAGGACTGTGTGACTATTGACGTCCTTC
```

# XString class decomposition

## XString slots

```
> getSlots("XString")
```

shared	offset	length	elementMetadata
"SharedRow"	"integer"	"integer"	"ANY"
elementType	metadata		
"character"	"list"		

```
> getSlots("XStringSet")
```

pool	ranges	elementMetadata
"SharedRow_Pool"	"GroupedIRanges"	"ANY"
elementType	metadata	
"character"	"list"	

## Notes

- ▶ shared, offset, length, pool, and ranges slots regulate pass-by-reference semantic.
- ▶ metadata slot can be used to hold annotation information.

# Basic string utilities

## Subsequence selection

`subseq`, `Views`

## Letter frequencies

`alphabetFrequency`, `dinucleotideFrequency`,  
`trinucleotideFrequency`, `oligonucleotideFrequency`,  
`letterFrequencyInSlidingView`, `uniqueLetters`

## Letter consensus

`consensusMatrix`, `consensusString`

## Letter transformation

`reverse`, `complement`, `reverseComplement`, `translate`, `chartr`

## I/O

`read.DNAStringSet`, `read.RNAStringSet`, `read.AAStringSet`,  
`read.BStringSet`, `write.XStringSet`, `save.XStringSet`

# String matching/alignment utilities

## matchPDict

matchPDict, countPDict, whichPDict, vmatchPDict, vcountPDict, vwhichPDict

## vmatchPattern

matchPattern, countPattern, vmatchPattern, vcountPattern, neditStartingAt, neditEndingAt, isMatchingStartingAt, isMatchingEndingAt, which.isMatchingStartingAt, which.isMatchingEndingAt

## pairwiseAlignment

pairwiseAlignment, stringDist

## matchPWM

matchPWM, countPWM

## OTHER

matchLRPatterns, trimLRPatterns, matchProbePair, findPalindromes, findComplementedPalindromes

# Letter frequencies

## Single-letter frequencies

```
> alphabetFrequency(yeast1, baseOnly=TRUE)
```

```
      A      C      G      T other
69830 44643 45765 69970      0
```

## Multi-letter frequencies

```
> dinucleotideFrequency(yeast1)
```

```
      AA      AC      AG      AT      CA      CC      CG      CT      GA      GC
23947 12493 13621 19769 15224  9218  7089 13112 14478  8910
      GG      GT      TA      TC      TG      TT
 9438 12938 16181 14021 15617 24151
```

```
> head(trinucleotideFrequency(yeast1), 12)
```

```
      AAA      AAC      AAG      AAT      ACA      ACC      ACG      ACT      AGA      AGC      AGG      AGT
 8576  4105  4960  6306  3924  2849  2186  3534  4537  2680  2707  3697
```

# Basic transformations

## Standard transformations

```
> x
  21-letter "DNAString" instance
seq: TCAACGTTGAATAGCGTACCG
> reverseComplement(x)
  21-letter "DNAString" instance
seq: CGGTACGCTATTCAACGTTGA
> translate(x)
  7-letter "AAString" instance
seq: STLNSVP
```

## Bisulfite transformation

```
> library(BSgenome.Celegans.UCSC.ce2)
> alphabetFrequency(Celegans$chrII, baseOnly=TRUE)
  A      C      G      T  other
4878194 2769208 2762193 4869710      3
> chrIIBis <- chartr("C", "T", Celegans$chrII)
> alphabetFrequency(chrIIBis, baseOnly=TRUE)
  A      C      G      T  other
4878194      0 2762193 7638918      3
```

# Letter consensus

## Consensus matrix

```
> snippet <- subseq(head(sort(srPhiX174), 5), 1, 10)
> consensusMatrix(snippet, baseOnly=TRUE)
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]
A	5	5	1	0	4	2	1	0	1	0
C	0	0	1	0	0	2	0	0	0	0
G	0	0	3	4	0	0	0	4	0	3
T	0	0	0	1	1	1	4	1	4	2
other	0	0	0	0	0	0	0	0	0	0

## Consensus string

```
> consensusString(snippet)
```

```
[1] "AAGGAMTGTK"
```

```
> consensusString(snippet, ambiguityMap = "N", threshold = 0.5)
```

```
[1] "AAGGANTGTG"
```

# String matching

## Match counting

```
> data(phiX174Phage)
> genome <- phiX174Phage[["NEB03"]]
> negPhiX174 <- reverseComplement(srPhiX174)
> posCounts <- countPDict(PDict(srPhiX174), genome)
> negCounts <- countPDict(PDict(negPhiX174), genome)
> table(posCounts, negCounts)
```

```
      negCounts
posCounts  0
          0 1030
          1   83
```

## Match locations

```
> matchPDict(PDict(srPhiX174[posCounts > 0]), genome)
```

MIndex object of length 83

# Pairwise alignments

## Alignment scores

```
> data(phiX174Phage)
> posScore <- pairwiseAlignment(srPhiX174, genome,
+                               type = "global-local", scoreOnly = TRUE)
> negScore <- pairwiseAlignment(negPhiX174, genome,
+                               type = "global-local", scoreOnly = TRUE)
> cutoff <- max(pmin.int(posScore, negScore))
```

## Alignments

```
> pairwiseAlignment(srPhiX174[posScore > cutoff], genome,
+                   type = "global-local")
```

```
Global-Local PairwiseAlignedFixedSubject (1 of 1112)
pattern:  [1] GTTATTATACCGTCAAGGACTGTGTGACTATTGAC
subject: [2750] GTTATTATACCGTCAAGGACTGTGTGACTATTGAC
score: 69.36144
```

# Outline

Introduction

Genomic Intervals with Data

Coverage and Other Piecewise Constant Measures

Long Biological Strings

Developer's Notes

Resources

# Long compressed sequence classes (*IRanges*)

## *Rle*

- ▶ Compressed atomic vectors
- ▶ Methods for standard *R atomic vector* functions
- ▶ Concrete class with sub-typing at the slot level

## *CompressedList*

- ▶ Compressed list of S4 objects
- ▶ Methods for standard *R list* functions
- ▶ Virtual class with sub-typing at the subclass level

## *IRanges* (as Sequences)

- ▶ `as.integer` coercion
- ▶ Subscripting via `seqselect`, `window`, and `[]`

# Pointer referenced sequence classes

## *XVector (IRanges)*

- ▶ External pointer-based atomic vectors
- ▶ Virtual class
- ▶ Concrete subclasses:
  - ▶ *XRaw* – Underlies *Biostrings* infrastructure
  - ▶ *XInteger* – Experimental integer vector class
  - ▶ *XDouble* – Experimental real number vector class

## *XString (Biostrings)*

- ▶ Virtual class
- ▶ Concrete subclasses:
  - ▶ *BString* – Any “biological” sequence
  - ▶ *DNAStrng* – DNA sequence
  - ▶ *RNAStrng* – RNA sequence
  - ▶ *AAString* – Amino acid sequence

# Outline

Introduction

Genomic Intervals with Data

Coverage and Other Piecewise Constant Measures

Long Biological Strings

Developer's Notes

Resources

# Resources

## *Bioconductor* Web site

- ▶ '*IRanges*', '*GenomicRanges*', and '*Biostrings*' links.
- ▶ <http://bioconductor.org>
- ▶ 'Installation', 'Software', and 'Mailing lists' links.

## Help in *R*

- ▶ `help.start()` to view a help browser.
- ▶ `help(package = "Biostrings")`
- ▶ `?findOverlaps`
- ▶ `browseVignettes("GenomicRanges")`