An Introduction to *biovizBase*

Tengfei Yin, Michael Lawrence, Dianne Cook

April 19, 2012

Contents

1 Introduction

The *biovizBase* package is designed to provide a set of utilities and color schemes serving as the basis for visualizing biological data, especially genomic data. Two other packages are currently built on this package, a static version of graphics is provided by the package *ggbio*, and an interactive version of graphics is provided by *visnab*(Currently not released).

In this vignette, we will introduce those color schemes and different utilities functions using simple examples and data sets. Utilities includes functions that precess the raw data, validate names, add attributes, and generate summaries such as fragment length, GC content, and mismatch information.

2 Color Schemes

The *biovizBase* package aims to provide a set of default color schemes for biological data, based on the following principles.

- Make biological sense. Data is displayed in a way that is similar to observed results under the microscope. (Example: giemsa stain results)
- Generate aesthetically pleasing colors based on well-defined color sets like *color brewer* [1]. Produce the appropriate color for *sequential, diverging, and qualitative* color schemes.
- Accommodate colorblind vision by creating color pallets that pass the color blind check on the *Vischeck* website [2] or use palette from package *dichromat* or use color-blind safe color palette checked by *ColorBrewer* website[3]. There are three types of colorblind checking strategy defined on these website.

  Deuteranope a form of red/green color deficit;

  Protanope another form of red/green color deficit;

  Tritanope a blue/yellow deficit- very rare.

Our color scheme try to pass color-blind checking points to make sure all the users can tell the difference between groups of data displayed. To make the implementation easy, we most time just use *dichromat* to check this, *dichromat* collapses red-green color distinctions to approximate the effect of the two common forms of red/green color blindness, protanopia and deuteranopia. Or we could simply implement proved color-blind safe palette from *dichromat* or *RColorBrewer*.

All color schemes have a general color generating function and a default color generating function. They are automatically stored in `options` as default when loading the package. Other packages built on *biovizBase* can use the default color scheme, ensuring consistent color themes across all static and interactive graphics. Users may also change the default color in the `options` to personalize the global color scheme to fit their needs.

```
> library(biovizBase)
> ## library(scales)
>
```

2.1 Colorblind Safe Palette

For graphics, it's important to make sure most people can tell the difference between colors on the plots, even for people with deficient or anomalous red-green vision.

[1] `http://colorbrewer2.org/`
[2] `http://www.vischeck.com/`
[3] `http://colorbrewer2.org/`

We will add more and more colorblind safe palette gradually, now we only supported palettes from two packages, *dichromat* or *RColorBrewer*. However, *RColorBrewer* doesn't provide information about colorblind palette. So we need to check manually on *ColorBrewer* website, and add this information with the palette information. For *dichromat* package, it doesn't have a palette information like `brewer.pal.info`, which contains three different types, **qual, div, seq** representing quality, divergent and sequential respectively, and also missing max colors information, so we integrate all these information and generate three palette information.

- `brewer.pal.blind.info` provides only colorblind safe palette subset.
- `dichromat.pal.blind.info` provides colorblind safe palette with category information and max color allowed.
- `blind.pal.info` integrate first two, provides a general palette information with extra column like pal.id, which used for function `colorBlindSafePal` as index for arguments `palette` or maxcolors for allowed number of color. *pkg* providing information about which package it is defined.

```
> head(blind.pal.info)
```

```
                maxcolors category          pkg pal.id
BluetoGray.8            8      div    dichromat      1
BluetoOrange.8          8      div    dichromat      2
BrowntoBlue.10         10      div    dichromat      3
BluetoOrange.10        10      div    dichromat      4
PiYG                   11      div RColorBrewer      5
PRGn                   11      div RColorBrewer      6
```

Then we defined a color generating function `colorBlindSafePal`, this function reading in a palette argument which could be a index number or names for palette defined in `blind.pal.info`. And return a color generating function, a `repeatable` argument will control, for number over max color numbers required, does it simply repeat it or just providing limited number of colors.

```
> ## with no arguments, return blind.pal.info
> head(colorBlindSafePal())
```

```
                maxcolors category          pkg pal.id
BluetoGray.8            8      div    dichromat      1
BluetoOrange.8          8      div    dichromat      2
BrowntoBlue.10         10      div    dichromat      3
BluetoOrange.10        10      div    dichromat      4
PiYG                   11      div RColorBrewer      5
PRGn                   11      div RColorBrewer      6
```

```
> ##
> mypalFun <- colorBlindSafePal("Set2")
> ## mypalFun(12, repeatable = FALSE) #only three
> mypalFun(11, repeatable = TRUE)  #repeat
```

```
 [1] "#66C2A5" "#FC8D62" "#8DA0CB" "#66C2A5" "#FC8D62" "#8DA0CB"
 [7] "#66C2A5" "#FC8D62" "#8DA0CB" "#66C2A5" "#FC8D62"
```

To Collapses red-green color distinctions to approximate the effect of the two common forms of red- green color blindness, protanopia and deuteranopia, we can use function `dichromat` from package *dichromat*, this save us the time to

We only show this as an examples and won't compare all other color schemes in the following sections. Please notice that

```
> ## for palette "Paried"
> mypalFun <- colorBlindSafePal(21)
> par(mfrow = c(1, 3))
> showColor(mypalFun(4))
> library(dichromat)
> showColor(dichromat(mypalFun(4), "deutan"))
> showColor(dichromat(mypalFun(4), "protan"))
```

Figure 1: Checking colors with two common type of color blindness. The first one is normal perception, second one for deuteranopia and last one for protanopia. Since we are using selected color palettes in this package, it should be fine with those types of blindness.

- If the categorical data contains many levels like amino acid, people cannot easily tell the difference anyway, we did the trick to simply repeat the colors. This might be useful for many other cases like grand linear view for chromosomes, since if the viewed orders of chromosomes is fixed it's OK to use repeated colors since they are not going to be layout as neighbors anyway.
- For schemes like cytobands, we try to follow the biological sense, in this case, we don't really check the color blindness.

2.2 Cytobands Color

Chemically staining the metaphase chromosomes results in a alternating dark and light banding pattern, which could provide information about abnormalities for chromosomes. Cytogenetic bands could also provide potential predictions of chromosomal structural characteristics, such as repeat structure content, CpG island density, gene density, and GC content.

biovizBase package provides utilities to get ideograms from the UCSC genome browser, as a wrapper around some functionality from *rtracklayer*. It gets the table for *cytoBand* and stores the table for certain species as a `GRanges` object.

We found a color setting scheme in package *geneplotter*, and we implemented it in biovisBase.

The function .cytobandColor will return a default color set. You could also get it from `options` after you load *biovizBase* package.

And we recommended function `getBioColor` to get the color vector you want, and names of the color is biological categorical data. This function hides interval color genenerators and also the complexity of getting color from options. You could specify whether you want to get colors by default or from options, in this way, you can temporarily edit colors in options and could change or all the graphics. This give graphics a uniform color scheme.

```
> getOption("biovizBase")$cytobandColor

     gneg    gpos25    gpos50    gpos75   gpos100      gvar     stalk
"grey100"  "grey90"  "grey70"  "grey40"   "grey0" "grey100"  "brown3"
     acen
 "brown4"

> getBioColor("CYTOBAND")

     gneg    gpos25    gpos50    gpos75   gpos100      gvar     stalk
"grey100"  "grey90"  "grey70"  "grey40"   "grey0" "grey100"  "brown3"
     acen
 "brown4"

> ## differece source from default or options.
> opts <- getOption("biovizBase")
> opts$DNABasesNColor[1] <- "red"
> options(biovizBase = opts)
> ## get from option(default)
> getBioColor("DNA_BASES_N")

        A         T         G         C         N
    "red" "#2C7BB6" "#D7191C" "#FDAE61" "#FFFFBF"

> ## get default fixed color
> getBioColor("DNA_BASES_N", source = "default")
```

```
        A         T         G         C         N
"#ABD9E9" "#2C7BB6" "#D7191C" "#FDAE61" "#FFFFBF"
```

```
> seqs <- c("A", "C", "T", "G", "G", "G", "C")
> ## get colors for a sequence.
> getBioColor("DNA_BASES_N")[seqs]
```

```
        A         C         T         G         G         G         C
    "red" "#FDAE61" "#2C7BB6" "#D7191C" "#D7191C" "#D7191C" "#FDAE61"
```

You can check the color scheme by calling the `plotColorLegend` function. or the `showColor`.

```
> cols <- getBioColor("CYTOBAND")
> plotColorLegend(cols, title  = "cytoband")
```

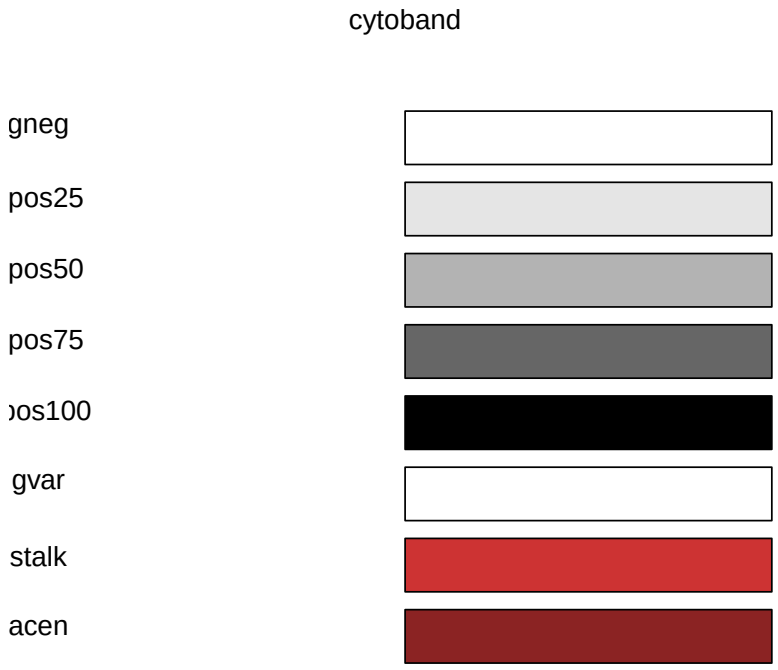

Figure 2: Legend for cytoband color

2.3 Strand Color

In the `GRanges` object, we have `strand` which contains three levels, `+`, `-`, `*`. We are using a qualitative color set from *Color Brewer* and check with *dichromat* as Figure3 shows, and we can see that this color set passes all three types of colorblind test. Therefore it should be a safe color set to use to color strand.

@

```
> par(mfrow = c(1, 3))
> cols <- getBioColor("STRAND")
> showColor(cols)
> showColor(dichromat(cols, "deutan"))
> showColor(dichromat(cols, "protan"))
```

Figure 3: Colorblind vision check for color of strand

2.4 Nucleotides Color

We start with the five most used nucleotides, **A,T,C,G,N**, most genome browsers have their own color scheme to represent nucleotides, We chose our color scheme based on the principles introduced above. Since in genetics, *GC-content* usually has special biological significance because GC pair is bound by three hydrogen bonds instead of two like AT pairs. So it has higher thermostability which could result in different significance, like higher annealing temperature in PCR. So we hope to choose warm colors for **G,C** and cold colors for **A,T**, and a color in between to represent **N**. They are chosen from a diverging color set of *color brewer*. So we should be able to easily tell the GC enriched region. Figure 4 shows the results from *dichromat*, and we can see this color set passes all two types of the colorblind test. It should be a safe color set to use to color the five most used nucleotides.

```
> getBioColor("DNA_BASES_N")

        A         T         G         C         N
    "red" "#2C7BB6" "#D7191C" "#FDAE61" "#FFFFBF"

>
```

2.5 Amino Acid Color and Other Schemes

We also include some other color schemes created based on existing object in package *Biostrings* and other customized color scheme. Please notice that the object name is not the same as the name in the options. On the left of =, it's name of object, most of them are defined in *Biostrings* and on the right, it's the name in options.

```
DNA_BASES_N = "DNABasesNColor"
DNA_BASES = "DNABasesColor"
DNA_ALPHABET = "DNAAlphabetColor"
RNA_BASES_N = "RNABasesNColor"
RNA_BASES = "RNABasesColor"
RNA_ALPHABET = "RNAAlphabetColor"
IUPAC_CODE_MAP = "IUPACCodeMapColor"
AMINO_ACID_CODE = "AminoAcidCodeColor"
AA_ALPHABET = "AAAlphabetColor"
STRAND = "strandColor"
CYTOBAND = "cytobandColor"
```

They all could be retrieved by calling function `getBioColor`.

2.6 Future Schemes

Current color schemes are most generated based on known object in R, which has a clear definition and classification. But we do have more interesting events or biological significance need to be color coded. Like most genome browser, they try to color code many events, for instance, color the insertion size which is larger/smaller than the estimated size; for paired RNA-seq data, we may color the paired reads mapped to a different chromosome.

We may include more color coded events in this package in next release.

```
> par(mfrow = c(1, 3))
> cols <- getBioColor("DNA_BASES_N", "default")
> showColor(cols, "name")
> cols.deu <- dichromat(cols, "deutan")
> names(cols.deu) <- names(cols)
> cols.pro <- dichromat(cols, "protan")
> names(cols.pro) <- names(cols)
> showColor(cols.deu, "name")
> showColor(cols.pro, "name")
```

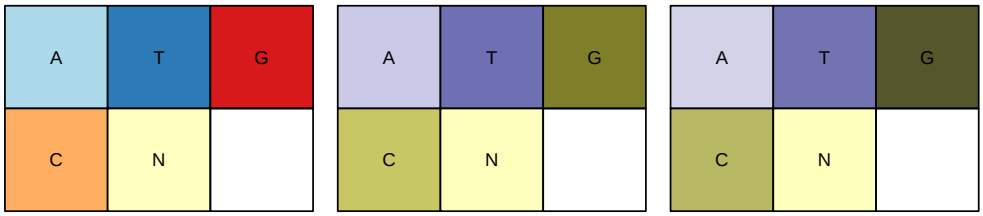

Figure 4: Colorblind vision check for color of nucleotide

3 Utilities

biovizBase serves as a basis for the visualization of biological data, especially for genomic data. *IRanges* and *GenomicRanges* are the two most important infrastructure packages to manipulate genomic data. They already have lots of useful and fast utilities for processing genomic data. Some other package such as *rtracklayer*, *Rsamtools*, *ShortRead*, *GenomicFeatures* provide common I/O for certain types of biological data and utilities for processing those raw data. Most of our utilities to be introduced in this section only manipulate the data in a simple and different way to get them ready for visualization. Most cases are only useful for visualization work, like adding brush color attributes to a `GRanges` object. Some of the other utilities are responsible for summarizing certain types of raw data, getting it ready to be visualized. Some of those utilities may be moved to a separate package later.

3.1 GRanges Related Manipulation

biovizBase mainly focuses on visualizing the genomic data, so we have some utilities for manipulating `GRanges` object. We are going to introduce these functions in the flow wing sub-sections. Overall, we hope to reduce people's work through these common utilities.

3.1.1 Adding Disjoint Levels

```
> library(GenomicRanges)
> set.seed(1)
> N <- 500
> gr <- GRanges(seqnames =
+               sample(c("chr1", "chr2", "chr3", "chrX", "chrY"),
+                      size = N, replace = TRUE),
+               IRanges(
+                       start = sample(1:300, size = N, replace = TRUE),
+                       width = sample(70:75, size = N,replace = TRUE)),
+               strand = sample(c("+", "-", "*"), size = N,
+                 replace = TRUE),
+               value = rnorm(N, 10, 3), score = rnorm(N, 100, 30),
+               group = sample(c("Normal", "Tumor"),
+                 size = N, replace = TRUE),
+               pair = sample(letters, size = N,
+                 replace = TRUE))
```

This is a tricky question. For example, for pair-end RNA-seq data, we may want to put the reads with the same *qname* on the same level, with nothing falling in between. For better visualization of the data, we may hope that adding invisible extensions to the reads will prevent closely neighbored reads from showing up on the same level.

`addStepping` function takes a `GenomicRanges` object and will add an extra column called **.levels** to the object. This function is essentially a wrapper around a function `disjointBins` but allows a more flexible way to assign levels to each entry. For example, if the arguments `group.name` is specified to one of the column in elementMetadata, the function will make sure

- Grouped intervals are in the same levels(if they are not overlapped each other).
- No entry is following between the grouped intervals.
- If extend.size is provided, it buffers the intervals and then computes the disjoint levels, thus ensuring that two closely positioned intervals will be assigned to different levels, a good practice for visualization.

For now, this function is only useful for visualization purposes.

```
> head(addStepping(gr))

GRanges with 6 ranges and 5 elementMetadata cols:
        seqnames     ranges strand |     value     score       group
           <Rle>  <IRanges>  <Rle> | <numeric> <numeric> <character>
   chr1     chr1 [160, 232]      - | 12.994463 118.45194      Normal
  chr11     chr1 [141, 213]      - |  9.350873  92.11885      Normal
  chr12     chr1 [ 39, 113]      * | 12.289738  65.92753       Tumor
  chr13     chr1 [242, 314]      - |  6.971489 102.75172      Normal
  chr14     chr1 [ 84, 158]      + |  9.309577 133.85467       Tumor
  chr15     chr1 [102, 174]      - | 12.610017 150.03296      Normal
              pair  stepping
        <character> <numeric>
   chr1           a         4
  chr11           y        19
  chr12           s        12
  chr13           u        24
  chr14           q         3
  chr15           y         9
  ---
  seqlengths:
   chr1 chr2 chr3 chrX chrY
     NA   NA   NA   NA   NA

> head(addStepping(gr, group.name = "pair"))

GRanges with 6 ranges and 5 elementMetadata cols:
        seqnames     ranges strand |     value     score       group
           <Rle>  <IRanges>  <Rle> | <numeric> <numeric> <character>
   chr1     chr1 [160, 232]      - | 12.994463 118.45194      Normal
  chr11     chr1 [141, 213]      - |  9.350873  92.11885      Normal
  chr12     chr1 [ 39, 113]      * | 12.289738  65.92753       Tumor
  chr13     chr1 [242, 314]      - |  6.971489 102.75172      Normal
  chr14     chr1 [ 84, 158]      + |  9.309577 133.85467       Tumor
  chr15     chr1 [102, 174]      - | 12.610017 150.03296      Normal
              pair  stepping
        <character> <numeric>
   chr1           a         1
  chr11           y        25
  chr12           s        19
  chr13           u        21
  chr14           q        17
  chr15           y        25
  ---
  seqlengths:
   chr1 chr2 chr3 chrX chrY
     NA   NA   NA   NA   NA

> gr.close <- GRanges(c("chr1", "chr1"), IRanges(c(10, 20), width = 9))
> addStepping(gr.close)

GRanges with 2 ranges and 1 elementMetadata col:
        seqnames    ranges strand |  stepping
```

```
          <Rle> <IRanges>  <Rle> | <numeric>
   chr1     chr1  [10, 18]      * |         1
  chr11     chr1  [20, 28]      * |         1
  ---
  seqlengths:
   chr1
     NA
```

```
> addStepping(gr.close, extend.size = 5)
```

```
GRanges with 2 ranges and 1 elementMetadata col:
        seqnames    ranges strand |  stepping
           <Rle> <IRanges>  <Rle> | <numeric>
   chr1     chr1  [10, 18]      * |         1
  chr11     chr1  [20, 28]      * |         2
  ---
  seqlengths:
   chr1
     NA
```

3.2 Shrink the Gaps

Sometime, in a gene centric view, we hope to truncate or shrink the gaps to better visualize the short reads or annotation data. It's **DANGEROUS** to shrink the gaps, since it only make sense in visualization. And even in the visualization the x-scale will be discontinued, and labels became somehow meaningless. **Make sure** you are not using the shrunk version of data when performing the down stream analysis.

This is a tricky question too, we hope to provide a flexible way to shrink the gaps. When we have multiple tracks, users would be responsible to shrink all the tracks based on the common gaps, otherwise there will be mis-aligned tracks.

`maxGap` computes a suitable estimated gap based on passed `GenomicRanges`

```
> gr.temp <- GRanges("chr1", IRanges(start = c(100, 250),
+                                    end = c(200, 300)))
> maxGap(gaps(gr.temp, start = min(start(gr.temp))))
```

```
[1] 0.1225
```

```
> maxGap(gaps(gr.temp, start = min(start(gr.temp))), ratio = 0.5)
```

```
[1] 24.5
```

`shrinkageFun` function will read in a `GenomicRanges` object which represents the gaps, and returns a function which alters a different `GenomicRanges` object, to shrink that object based on previously specified gaps shrinking information. You could use this function to treat multiple tracks(e.g. `GRanges`) to make sure they are shrunk based on the common gaps and the same ratio.

Be careful in the following situations.

- When use the same shrinkage function to shrink multiple tracks, make sure the gaps passed to `shrinkageFun` function is the common gaps across all tracks, otherwise, it doesn't make sense to cut a overlapped gap within one of the tracks.
- The default max gap is not 0, just for visualization purpose. If for estimation purpose, you might want to make sure you cut all the gaps.

And notice, after shrinking, the x-axis labes only provide approximate position as shown in Figure 5 and 6, because it's clipped. It's just for visualization purpose.

```
> gr1 <- GRanges("chr1", IRanges(start = c(100, 300, 600),
+                                end = c(200, 400, 800)))
> shrink.fun1 <- shrinkageFun(gaps(gr1), max.gap = maxGap(gaps(gr1), 0.15))
> shrink.fun2 <- shrinkageFun(gaps(gr1), max.gap = 0)
> head(shrink.fun1(gr1))

GRanges with 3 ranges and 1 elementMetadata col:
      seqnames     ranges strand |      .ori
         <Rle>  <IRanges>  <Rle> | <GRanges>
  [1]     chr1 [ 91, 191]      * |  ########
  [2]     chr1 [282, 382]      * |  ########
  [3]     chr1 [473, 673]      * |  ########
  ---
  seqlengths:
   chr1
     NA

> head(shrink.fun2(gr1))

GRanges with 3 ranges and 1 elementMetadata col:
      seqnames     ranges strand |      .ori
         <Rle>  <IRanges>  <Rle> | <GRanges>
  [1]     chr1 [  1, 101]      * |  ########
  [2]     chr1 [102, 202]      * |  ########
  [3]     chr1 [203, 403]      * |  ########
  ---
  seqlengths:
   chr1
     NA


> gr2 <- GRanges("chr1", IRanges(start = c(100, 350, 550),
+                                end = c(220, 500, 900)))
> gaps.gr <- intersect(gaps(gr1, start = min(start(gr1))),
+                      gaps(gr2, start = min(start(gr2))))
> shrink.fun <- shrinkageFun(gaps.gr, max.gap = maxGap(gaps.gr))
> head(shrink.fun(gr1))

GRanges with 3 ranges and 1 elementMetadata col:
      seqnames     ranges strand |      .ori
         <Rle>  <IRanges>  <Rle> | <GRanges>
  [1]     chr1 [100, 200]      * |  ########
  [2]     chr1 [222, 322]      * |  ########
  [3]     chr1 [474, 674]      * |  ########
  ---
  seqlengths:
   chr1
     NA

> head(shrink.fun(gr2))
```

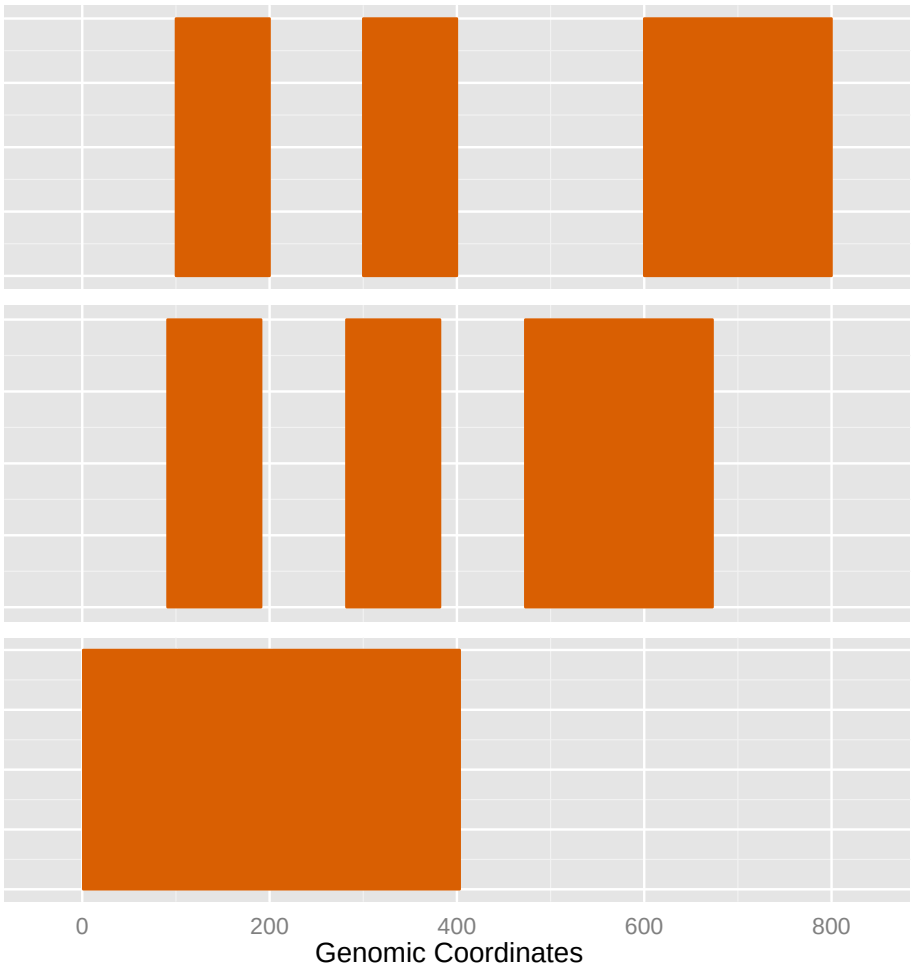

Figure 5: Shrink single GRanges. The first track is original GRanges, the second one use a ratio which shrink the GRanges a little bit, and default is to remove all gaps shown as the third track

```
GRanges with 3 ranges and 1 elementMetadata col:
      seqnames     ranges strand |      .ori
         <Rle>  <IRanges>  <Rle> | <GRanges>
  [1]     chr1 [100, 220]      * |  ########
  [2]     chr1 [272, 422]      * |  ########
  [3]     chr1 [424, 774]      * |  ########
  ---
  seqlengths:
   chr1
     NA
```

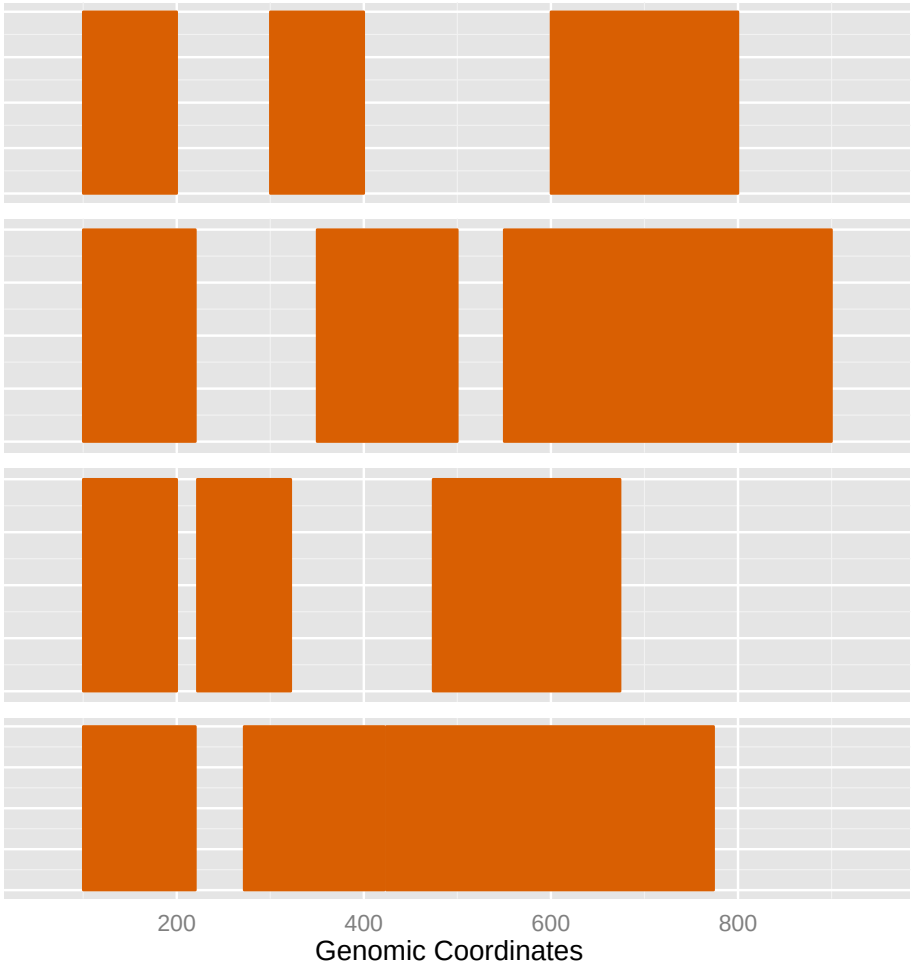

Figure 6: shrinkageFun demonstration for multiple GRanges, the top two tracks are the original tracks, please note how we clipped common gaps for those two tracks and shown as bottom two tracks.

3.3 GC content

As mentioned before, GC content is an interesting variable which may be related to various biological questions. So we need a way to compute GC content in a certain region of a reference genome.

`GCcontent` function is a wrapper around `getSeq` function in *BSgenome* package and `letterFrequency` in *Biostrings* package. It reads a `BSgenome` object and returns count/probability for **GC** content in specified region.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> GCcontent(Hsapiens, GRanges("chr1", IRanges(1e6, 1e6 + 1000)))
> GCcontent(Hsapiens, GRanges("chr1", IRanges(1e6, 1e6 + 1000)), view.width = 300)
```

3.4 Mismatch Summary

Compared to short-read alignment visualization, it's more useful to just show the summary of nucleotides of short reads per base and compare with the reference genome. We need a way to show the mismatched nucleotides, coverage at each position and proportion of mismatched nucleotides, and use the default color to indicate the type of nucleotide.

`pileupAsGRanges` function summarizes reads from bam files for nucleotides on single base units in a given region, which allows the downstream mismatch summary analysis. It's a wrapper around `applyPileup` function in Rsamtools package and more detailed control could be found under manual of PileupParam function in Rsamtools. `pileupAsGRanges` function returns a GRanges object which includes a summary of nucleotides, depth, and bam file path. This object could be read directly into the `pileupGRangesAsVariantTable` function for a mismatch summary.

This function returns a `GRanges` object with extra `elementMetadata`, counts for **A,C,T,G,N** and **depth** for coverage. **bam** indicates the bam file path. Each row is single base unit.

`pileupGRangesAsVariantTable` performs comparisons to the reference genome(a `BSgenome` object) and computes the mismatch summary for a certain region of reads. User need to make sure to pass the right reference genome to this function to get the right summary. This function drops the positions that have no reads and only keeps the regions with coverage in the summary. The result could be used to show stacked barchart for the mismatch summary.

This function returns a `GRanges` with the following elementMetadata information.

ref Reference base.

read Sequenced read at that position. Each type of **A,C,T,G,N** summarize counts at one position, if no counts detected, will not show it.

count Count for each nucleotide.

depth Coverage at that position.

match A logical value, indicate it's matched or not.

bam Indicate bam file path.

Sample raw data is from SRA(Short Read Archive), Accession: SRR027894 and subset the gene at chr10:6118023-6137427, which within gene RBM17. contains junction reads.

```
> library(Rsamtools)
> data(genesymbol)
> library(BSgenome.Hsapiens.UCSC.hg19)
> bamfile <- system.file("extdata", "SRR027894subRBM17.bam", package="biovizBase")
> test <- pileupAsGRanges(bamfile, region = genesymbol["RBM17"])
> test.match <- pileupGRangesAsVariantTable(test, Hsapiens)
> head(test[,-7])
> head(test.match[,-5])
```

3.5 Get an Ideogram

`getIdeogram` function is a wrapper of some functionality from *rtracklayer* to get certain table like cytoBand. A full table schema can be found here at *UCSC genome browser*. Please click *describe table schema*.

This function requires a network connection and will parse the data on the fly. The first argument of `getIdeogram` is `species`. If `missing`, the function will give you a choice hint, so you will not have to remember the name for the database you want, or you can simply get the database name for a different genome using the `ucscGenomes` function in *Rtracklayer*. The second argument `subchr` is used to subset the result by chromosome name. The third argument cytoband controls if you want to get the gieStain information/band information or not, which is useful for the visualization of the whole genome or single chromosome. You can see some examples in *ggbio*.

```
> library(rtracklayer)
> hg19IdeogramCyto <- getIdeogram("hg19", cytoband = TRUE)
> hg19Ideogram <- getIdeogram("hg19", cytoband = FALSE)
> unknowIdeogram <- getIdeogram()

Please specify genome

 1: hg19      2: hg18      3: hg17      4: hg16      5: felCat4
 6: felCat3   7: galGal3   8: galGal2   9: panTro3  10: panTro2
11: panTro1  12: bosTau4  13: bosTau3  14: bosTau2  15: canFam2
16: canFam1  17: loxAfr3  18: fr2      19: fr1      20: cavPor3
21: equCab2  22: equCab1  23: petMar1  24: anoCar2  25: anoCar1
26: calJac3  27: calJac1  28: oryLat2  29: mm9      30: mm8
31: mm7      32: monDom5  33: monDom4  34: monDom1  35: ponAbe2
36: ailMel1  37: susScr2  38: ornAna1  39: oryCun2  40: rn4
41: rn3      42: rheMac2  43: oviAri1  44: gasAcu1  45: tetNig2
46: tetNig1  47: xenTro2  48: xenTro1  49: taeGut1  50: danRer7
51: danRer6  52: danRer5  53: danRer4  54: danRer3  55: ci2
56: ci1      57: braFlo1  58: strPur2  59: strPur1  60: apiMel2
61: apiMel1  62: anoGam1  63: droAna2  64: droAna1  65: droEre1
66: droGri1  67: dm3      68: dm2      69: dm1      70: droMoj2
71: droMoj1  72: droPer1  73: dp3      74: dp2      75: droSec1
76: droSim1  77: droVir2  78: droVir1  79: droYak2  80: droYak1
81: caePb2   82: caePb1   83: cb3      84: cb1      85: ce6
86: ce4      87: ce2      88: caeJap1  89: caeRem3  90: caeRem2
91: priPac1  92: aplCal1  93: sacCer2  94: sacCer1

Selection:
```

Here is the example on how to get the genome names.

```
> head(ucscGenomes()$db)

[1] hg19    hg18    hg17    hg16    felCat4 felCat3
122 Levels: ailMel1 anoCar1 anoCar2 anoGam1 apiMel1 apiMel2 ...
```

We put the most used hg19 ideogram as our default data set, so you can simply load it and see what they look like. They are all returned by the `getIdeogram` function. The one with cytoband information has two special columns.

name Name of cytogenetic band

gieStain Giemsa stain results

```
> data(hg19IdeogramCyto)
> head(hg19IdeogramCyto)
```

```
GRanges with 6 ranges and 2 elementMetadata cols:
      seqnames               ranges strand |     name gieStain
         <Rle>            <IRanges>  <Rle> | <factor> <factor>
  [1]     chr1 [       0,  2300000]      * |   p36.33     gneg
  [2]     chr1 [ 2300000,  5400000]      * |   p36.32   gpos25
  [3]     chr1 [ 5400000,  7200000]      * |   p36.31     gneg
  [4]     chr1 [ 7200000,  9200000]      * |   p36.23   gpos25
  [5]     chr1 [ 9200000, 12700000]      * |   p36.22     gneg
  [6]     chr1 [12700000, 16200000]      * |   p36.21   gpos50
  ---
  seqlengths:
    chr1 chr10 chr11 chr12 chr13 chr14 ...  chr7  chr8  chr9  chrX  chrY
      NA    NA    NA    NA    NA    NA ...    NA    NA    NA    NA    NA
```

```
> data(hg19Ideogram)
> head(hg19Ideogram)
```

```
GRanges with 6 ranges and 0 elementMetadata cols:
                  seqnames         ranges strand
                     <Rle>      <IRanges>  <Rle>
  [1]                 chr1 [1, 249250621]      *
  [2] chr1_gl000191_random [1,    106433]      *
  [3] chr1_gl000192_random [1,    547496]      *
  [4]                 chr2 [1, 243199373]      *
  [5]                 chr3 [1, 198022430]      *
  [6]                 chr4 [1, 191154276]      *
  ---
  seqlengths:
                    chr1  chr1_gl000191_random ...                  chrM
               249250621                106433 ...                 16571
```

There are two simple functions to test if the ideogram is valid or not. `isIdeogram` simply tests if the result came from the `getIdeogram` function, making sure it's a `GenomicRanges` object with an extra column. `isSimpleIdeogram` only tests if it's `GenomicRanges` and does not require cytoband information. But it double checks to make sure there is only one entry per chromosome. This is useful to show stacked overview for genomes. Please check some examples in *ggbio* to draw stacked overview and single chromosome.

```
> isIdeogram(hg19IdeogramCyto)
```

```
[1] TRUE
```

```
> isIdeogram(hg19Ideogram)
```

```
[1] FALSE
```

```
> isSimpleIdeogram(hg19IdeogramCyto)
```

```
[1] FALSE
```

```
> isSimpleIdeogram(hg19Ideogram)
```

```
[1] TRUE
```

3.6 Other Utilities and Data Sets

We are not going to introduce other utilities in this vignette, please refer to the manual for more details, we have other function to transform a `GRanges` to a special format only for graphic purpose, such as function `transformGRangesForEvenSpace` and `transformGRangesToDfWithTicks` could be used for grand linear view or linked view as introduced in package *ggbio*.

We have introduced data sets like `hg19IdeogramCyto` and `hg19Ideogram` in the previous sections. We also have a data set called `genesymbol`, which is extracted from human annotation package and stored as `GRanges` object, with extra columns *symbol* and *ensembl_id*. For fast mapping, we use *symbol* as row names too.

This could be used for convenient overlapped subset with other annotation, and has potential use in a auto-complement drop list for gene search bar like most gene browsers have.

```
> data(genesymbol)
> head(genesymbol)

GRanges with 6 ranges and 2 elementMetadata cols:
       seqnames               ranges strand |      symbol
          <Rle>            <IRanges>  <Rle> | <character>
  A1BG    chr19 [58858174, 58864865]      - |         A1BG
   A2M    chr12 [ 9220304,  9268558]      - |          A2M
  NAT1     chr8 [18027971, 18081197]      + |         NAT1
  NAT1     chr8 [18067618, 18081197]      + |         NAT1
  NAT1     chr8 [18079177, 18081197]      + |         NAT1
  NAT2     chr8 [18248755, 18258723]      + |         NAT2
           ensembl_id
          <character>
  A1BG ENSG00000121410
   A2M ENSG00000175899
  NAT1 ENSG00000171428
  NAT1 ENSG00000171428
  NAT1 ENSG00000171428
  NAT2 ENSG00000156006
  ---
  seqlengths:
                    chr1                  chr10 ...                    chrY
                      NA                     NA ...                      NA

> genesymbol["RBM17"]

GRanges with 1 range and 2 elementMetadata cols:
        seqnames             ranges strand |      symbol
           <Rle>          <IRanges>  <Rle> | <character>
  RBM17    chr10 [6130949, 6159420]      + |       RBM17
             ensembl_id
            <character>
  RBM17 ENSG00000134453
  ---
  seqlengths:
                    chr1                  chr10 ...                    chrY
                      NA                     NA ...                      NA

>
```

4 Bugs Report and Features Request

Latest code are available on github `https://github.com/tengfei/biovizBase`

Please file bug/request on issue page, this is preferred way. or email me at yintengfei <at> gmail dot com.

It's a new package and under active development.

Thanks in advance for any feedback.

5 Acknowledgement

I wish to thank all those who helped me. Without them, I could not have started this project.

Genentech Sponsorship and valuable feed back and help for this project and my other project.

Jennifer Chang Feedback on this package

6 Session Information

```
> sessionInfo()

R version 2.15.0 (2012-03-30)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=C                 LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base

other attached packages:
[1] GenomicRanges_1.8.3 IRanges_1.14.2      BiocGenerics_0.2.0
[4] dichromat_1.2-4     biovizBase_1.4.2

loaded via a namespace (and not attached):
 [1] AnnotationDbi_1.18.0  BSgenome_1.24.0       Biobase_2.16.0
 [4] Biostrings_2.24.1     DBI_0.2-5             GenomicFeatures_1.8.1
 [7] Hmisc_3.9-3           RColorBrewer_1.0-5    RCurl_1.91-1
[10] RSQLite_0.11.1        Rsamtools_1.8.4       XML_3.9-4
[13] biomaRt_2.12.0        bitops_1.0-4.1        cluster_1.14.2
[16] colorspace_1.1-1      grid_2.15.0           lattice_0.20-6
[19] munsell_0.3           plyr_1.7.1            rtracklayer_1.16.1
[22] scales_0.2.0          stats4_2.15.0         stringr_0.6
[25] tools_2.15.0          zlibbioc_1.2.0
```