An introduction of making genomic plots with **circlize**

Zuguang Gu <z.gu@dkfz.de>

February 22, 2015

Since circos plots are mostly used in genomic research, the **circlize** package particularly provides functions which focus on genomic plots. These functions are synonymous to the basic circos graphic functions but expect special format of input data:

- `circos.genomicTrackPlotRegion`: create a new track and add graphics.
- `circos.genomicPoints`: low-level function, add points
- `circos.genomicLines`: low-level function, add lines
- `circos.genomicRect`: low-level function, add rectangles
- `circos.genomicText`: low-level function, add text
- `circos.genomicLink`: add links

The genomic functions are implemented by basic circos functions (e.g. `circos.trackPlotRegion`, `circos.points`), thus, you can customize your own plots by both genomic functions and basic circos functions.

1 Input data

Genomic circos functions expect input data as a data frame or a list of data frames in which there are at least three columns. The first column is genomic category (e.g. chromosome), the second column is the start positions in the genomic category and the third column is the end positions. Following columns are optional where numeric values or other related values are stored. Such data structure is known as *BED* format and is broadly used in genomic research.

circlize provides a simple function `generateRandomBed` which can generate random genomic data. Positions are uniformly generated from human genome and the number of regions on chromosomes approximately proportional to the length of chromosomes. In the function, `nr` and `nc` are number of rows and numeric columns that users want. Please note `nr` are not exactly the same as the number of rows which are returned by the function. `fun` is a self-defined function to generate random values.

```
library(circlize)
set.seed(999)

bed = generateRandomBed()
head(bed)

##    chr   start     end     value1
## 1 chr1   39485  159163 -0.1887635
## 2 chr1  897195 1041959 -0.2435220
## 3 chr1 1161957 1177159  0.3749953
## 4 chr1 1201513 1481406 -0.2600839
## 5 chr1 1487402 1531773 -0.4633990
## 6 chr1 1769949 2736215 -0.8159909

bed = generateRandomBed(nr = 200, nc = 4)
nrow(bed)
```

```
## [1] 205

bed = generateRandomBed(nc = 2, fun = function(k) runif(k))
head(bed)

##    chr   start     end     value1    value2
## 1 chr1   98740  566688 0.18303240 0.1725320
## 2 chr1  769960  887938 0.04453375 0.6378507
## 3 chr1  906851  933021 0.80237105 0.5433349
## 4 chr1 1241911 1243537 0.41103676 0.2260829
## 5 chr1 1385344 1410947 0.83473785 0.9218159
## 6 chr1 1498302 1585389 0.77238964 0.8390309
```

2 Initialize the layout

2.1 Initialize with cytoband data

2.1.1 Basic usage

Similar as general circos plots, the first step is to initialize the plot with genomic categories. In most situations, genomic categories are measured by chromosomes. The easiest way is to used `circos.initializeWithIdeogram` (figure 1 A):

```
par(mar = c(1, 1, 1, 1))
circos.initializeWithIdeogram()
```

By default, the function will initialize the plot with cytoband data of hg19. You can also use your own cytoband data by specifying the path of your cytoband file (no matter compressed or not) or providing your cytoband data as a data frame. An example for cytoband file is `http://hgdownload.soe.ucsc.edu/goldenpath/hg19/database/cytoBand.txt.gz`.

```
cytoband.file = paste0(system.file(package = "circlize"), "/extdata/cytoBand.txt")
circos.initializeWithIdeogram(cytoband.file)
cytoband.df = read.table(cytoband.file, colClasses = c("character", "numeric",
    "numeric", "character", "character"), sep = "\t")
circos.initializeWithIdeogram(cytoband.df)
circos.clear()
```

If you want to read cytoband data from file, please explicitly specify `colClasses` arguments and set the class of position columns as `numeric`. The reason is since positions are represented as integers, `read.table` would treat those numbers as `integer` by default. In initialization of circos plot, **circlize** needs to calculate the summation of all chromosome lengths. The sumation of such large integers would throw error of data overflow.

For simple use, users can also specify abbreviation of the species and the function will download cytoband file from UCSC server automatically (If it exists in UCSC. As you can guess, the URL template we use is `http://hgdownload.soe.ucsc.edu/goldenpath/$species/database/cytoBand.txt.gz`).

```
circos.initializeWithIdeogram(species = "hg18")
circos.initializeWithIdeogram(species = "mm10")
```

By default, the function will use all chromosomes which are available in cytoband data to initialize the circos plot. Users can also choose a subset of chromosomes by specifying `chromosome.index`. This argument is also for ordering chromosomes (figure 1 B).

```
circos.initializeWithIdeogram(chromosome.index = paste0("chr", 10:1))
```

2.1.2 Order chromosomes

Initialization step is important for circos plot. It controls orders of chromosomes which are going to be put on the circle. There are several ways to control the order:

- If `chromosome.index` is set, the order of `chromosome.index` is taken as order of chromosomes.
- If `chromosome.index` is not set and `sort.chr` is set to `TRUE`, chromosomes will be sorted first by numbers then by letters (figure 1 C).
- If `chromosome.index` and `sort.chr` are not set neither:
  - If `cytoband` is provided as a data frame, and if the first column is not a factor, the order of chromosomes would be `unique(cytoband[[1]])` (figure 1 D).
  - If `cytoband` is provided as a data frame, and if the first column is a factor, the order of chromosome would be `levels(cytoband[[1]])` (figure 1 E).
  - If `cytoband` is specified as a file path, or `species` is specified, the order of chromosomes depends on the original order in the source file.

```
cytoband = cytoband.df
circos.initializeWithIdeogram(cytoband, sort.chr = TRUE)
cytoband = cytoband.df
circos.initializeWithIdeogram(cytoband, sort.chr = FALSE)
cytoband[[1]] = factor(cytoband[[1]], levels = paste0("chr", c(22:1, "X", "Y")))
circos.initializeWithIdeogram(cytoband, sort.chr = FALSE)
```

circlize provides a function `read.cytoband` which can read/download and process cytoband data. In fact, `circos.initializeWithIdeogram` calls `read.cytoband` internally (actually, if there is no cytoband available, `read.chromInfo` will be called later). Please refer to the help page of the function for more details.

```
cytoband = read.cytoband()
cytoband = read.cytoband(file)
cytoband = read.cytoband(df)
cytoband = read.cytoband(species)
```

2.1.3 Pre-defined tracks

After the initialization of the circos plot, the function will additionally create a track where there are genomic axes and chromosome names, and create another track where there is an ideogram (depends on whether cytoband data is available). `plotType` can be used to control which kind of graphics need to be added (figure 1 F, G).

```
circos.initializeWithIdeogram(plotType = c("axis", "labels"))
circos.initializeWithIdeogram(plotType = NULL)
circos.clear()
```

2.1.4 Other general settings

Similar as general circos plot, the layout of circos plot can be controlled by `circos.par` (figure 1 H, I).

```
circos.par("start.degree" = 90)
circos.initializeWithIdeogram()
circos.clear()
circos.par("gap.degree" = rep(c(2, 4), 12))
circos.initializeWithIdeogram()
circos.clear()
```

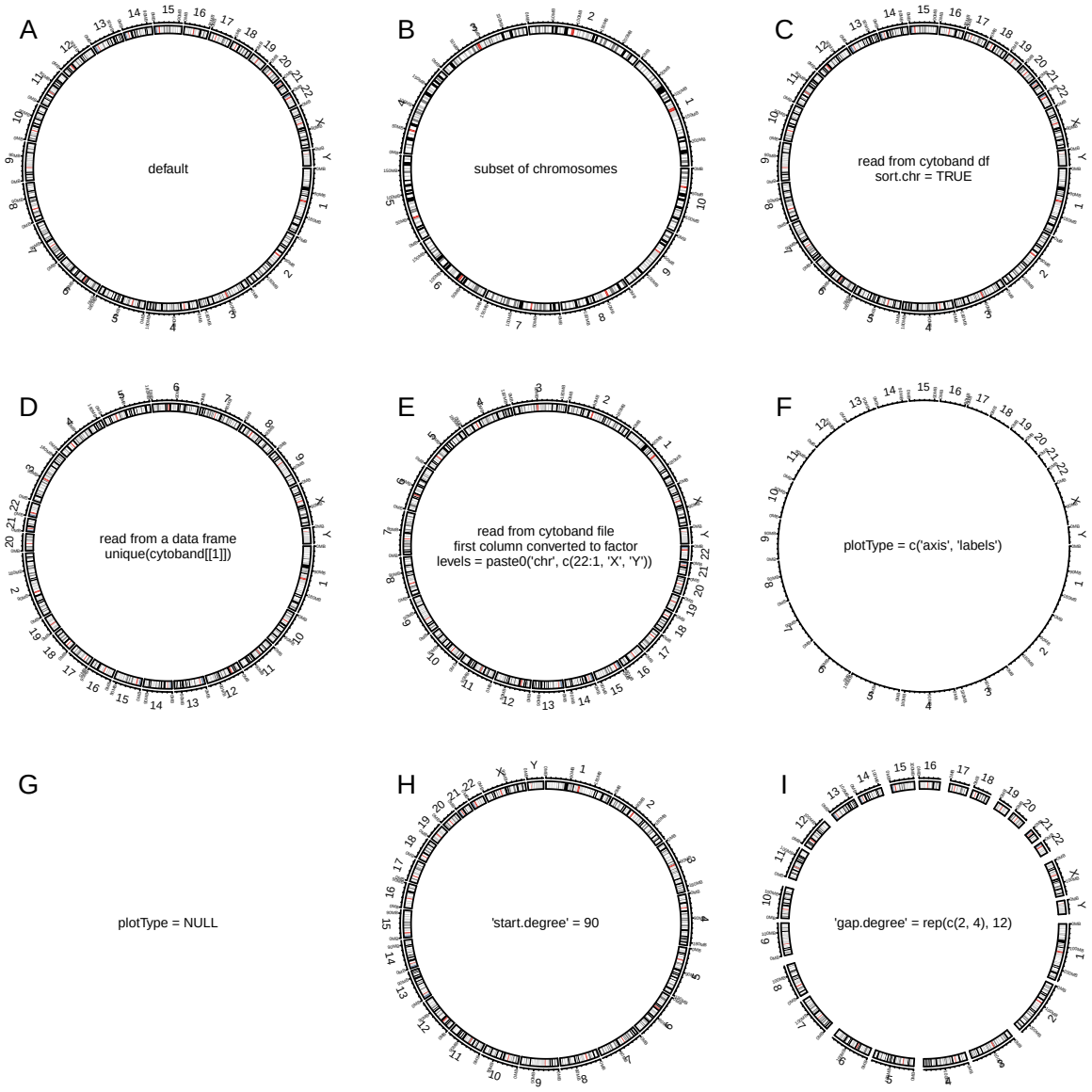

Figure 1: Different ways to initialize genomic circos plot. A) default; B) select subset of chromosomes; C) read from a data frame; D) the first column of the data frame is a factor; E) sort chromosomes; F) do not add ideogram; G) initialize the plot while plot nothing; H) set start degree of the plot; I) set gap degree.

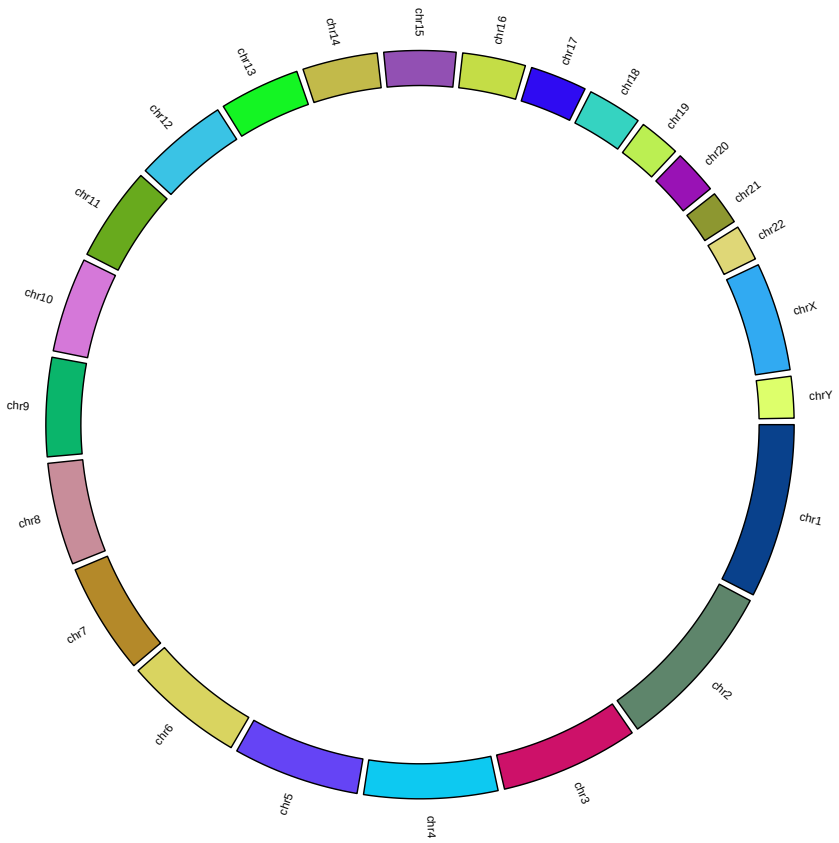

Figure 2: Customize ideogram.

2.2 Customize ideogram

The default behaviour of `circos.initializeWithIdeogram` is both initializing the layout and adding two tracks. But if `plotType` is set to `NULL`, the circular layout is only initialized but nothing is added. It provides possibility for users to completely design their own style of plots. In the following example, we use different colors to represent chromosomes and change the style of chromosome names (figure 2).

```
par(mar = c(1, 1, 1, 1))
circos.initializeWithIdeogram(plotType = NULL)
circos.trackPlotRegion(ylim = c(0, 1), panel.fun = function(x, y) {
    chr = get.cell.meta.data("sector.index")
    xlim = get.cell.meta.data("xlim")
    ylim = get.cell.meta.data("ylim")
    circos.rect(xlim[1], 0, xlim[2], 0.5, col = rand_color(1))
    circos.text(mean(xlim), 0.9, chr, cex = 0.5, facing = "clockwise",
        niceFacing = TRUE)
}, bg.border = NA)
```

```
circos.clear()
```

2.3 Initialize with general genomic category

Cytoband data is just a special case of genomic category. `circos.genomicInitialize` can initialize circular layout with any kind of genomic categories. In fact, `circos.initializeWithIdeogram` is implemented by `circos.genomicInitialize`. The input data for the function is also a data frame with at least three columns. The first column is genomic category (for cytoband data, it is chromosome name), and the next two columns are genomic positions in each genomic category. The range of each category will be inferred from the minimum position and the maximum position in corresponding category. In the following example, a circos plot is initialized with three genes.

```
df = data.frame(
    name  = c("TP53",  "TP63",    "TP73"),
    start = c(7565097, 189349205, 3569084),
    end   = c(7590856, 189615068, 3652765))
circos.genomicInitialize(df)
```

Note it is not necessary that the record for each gene is one row.

As explained in previous section, the order of genomic categories is controlled by the first column of `df` which depends on whether it is a factor or a simple vector. Alternative names can be assigned to each category and the order of alternative names correspond to the order of genomic categories.

```
circos.genomicInitialize(df)
circos.genomicInitialize(df, sector.names = c("tp53", "tp63", "tp73"))
circos.genomicInitialize(df, plotType)

circos.par(gap.degree = 2)
circos.genomicInitialize(df)
```

In following example, we plot the transcripts for TP53, TP63 and TP73 in a circular layout (figure 3). The object `tp_family` is a list of data frames which contain positions of exons for each transcript.

```
load(paste0(system.file(package = "circlize"), "/extdata/tp_family.RData"))
names(tp_family)

## [1] "TP73" "TP63" "TP53"

names(tp_family[["TP53"]])

##  [1] "ENST00000413465.2" "ENST00000359597.4" "ENST00000504937.1"
##  [4] "ENST00000269305.4" "ENST00000510385.1" "ENST00000504290.1"
##  [7] "ENST00000455263.2" "ENST00000420246.2" "ENST00000445888.2"
## [10] "ENST00000576024.1" "ENST00000509690.1" "ENST00000514944.1"
## [13] "ENST00000574684.1" "ENST00000505014.1" "ENST00000508793.1"
## [16] "ENST00000604348.1" "ENST00000503591.1"

head(tp_family[["TP53"]][[1]])

##          start     end
## exon_7 7565097 7565332
## exon_6 7577499 7577608
## exon_5 7578177 7578289
## exon_4 7578371 7578554
## exon_3 7579312 7579590
## exon_2 7579700 7579721

df = data.frame(gene = names(tp_family),
                start = sapply(tp_family, function(x) min(unlist(x))),
                end = sapply(tp_family, function(x) max(unlist(x))))
df
```

```
##      gene     start       end
## TP73 TP73   3569084   3652765
## TP63 TP63 189349205 189615068
## TP53 TP53   7565097   7590856
```

In the following code, we first create a track which identifies three genes.

```
circos.genomicInitialize(df)
circos.genomicTrackPlotRegion(ylim = c(0, 1),
    bg.col = c("#FF000040", "#00FF0040", "#0000FF40"),
    bg.border = NA, track.height = 0.05)
```

Next, we put each transcript line by line. It is simply adding lines and rectangles. The usage of `circos.genomicTrackPlotRegion` will be introduced in later sections.

```
n = max(sapply(tp_family, length))
circos.genomicTrackPlotRegion(ylim = c(0.5, n + 0.5),
    panel.fun = function(region, value, ...) {
        gn = get.cell.meta.data("sector.index")
        tr = tp_family[[gn]]  # all transcripts for this gene
        for(i in seq_along(tr)) {
            # for each transcript
            current_tr_start = min(tr[[i]]$start)
            current_tr_end = max(tr[[i]]$end)
            circos.lines(c(current_tr_start, current_tr_end),
                c(n - i, n - i), col = "#CCCCCC")
            circos.genomicRect(tr[[i]], ytop = n - i + 0.4,
                ybottom = n - i - 0.4, col = "orange", border = NA)
        }
}, bg.border = NA, track.height = 0.3)
circos.clear()
```

2.4 Zooming

`circos.genomicInitialize` is implemented by the general `circos.initialize` function, so same strategy can be applied to zoom chromosomes (figure 4). Just be careful with the order of chromosomes.

```
cytoband = read.cytoband()
df = cytoband$df
chromosome = cytoband$chromosome

# copy regions for the two zoomed chromosomes
zoom_df = df[df[[1]] %in% chromosome[1:2], ]
zoom_df[[1]] = paste0("zoom_", zoom_df[[1]])
df2 = rbind(df, zoom_df)

# attach ranges for two zoomed chromosomes
xrange = c(cytoband$chr.len, cytoband$chr.len[1:2])
normal_sector_index = seq_along(chromosome)
zoomed_sector_index = length(chromosome) + 1:2

# normalize in normal chromsomes and zoomed chromosomes separately
sector.width = c(xrange[normal_sector_index] / sum(xrange[normal_sector_index]),
                 xrange[zoomed_sector_index] / sum(xrange[zoomed_sector_index]))
```

Here we can only use `circos.genomicInitialize` to deal with zoomed chromosomes because chromosome with name "zoom_chr1" is not a normal chromosome.

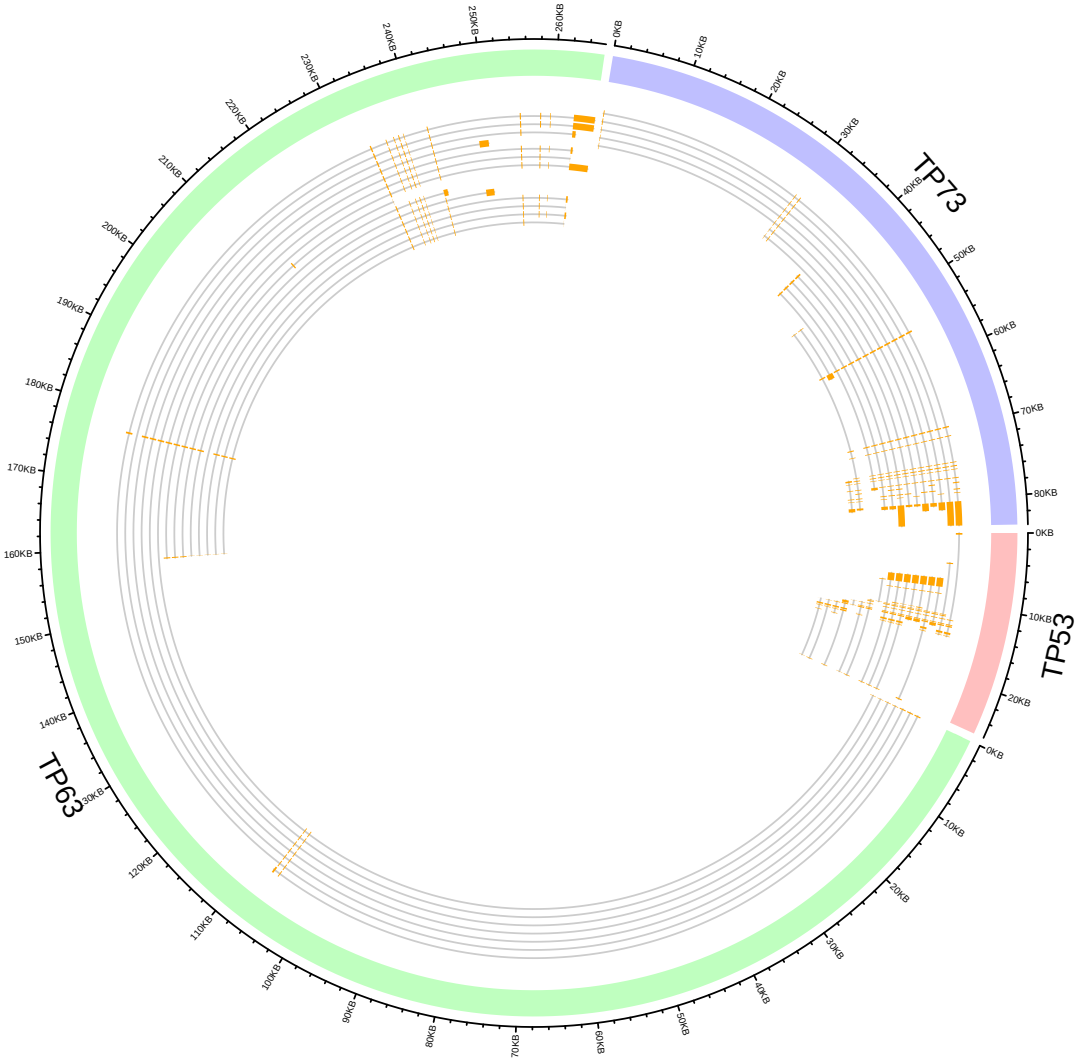

Figure 3: Alternative transcripts for genes.

```
par(mar = c(1, 1, 1, 1))
circos.par(start.degree = 90)
circos.genomicInitialize(df2, sector.width = sector.width)
```

In following tracks, data frames should also be extended with zoomed chromosomes. We can write a simple function to do the extension.

```
extend_zoomed_chromosome_in_bed = function(bed, chromosome, prefix = "zoom_") {
    zoom_bed = bed[bed[[1]] %in% chromosome, , drop = FALSE]
    zoom_bed[[1]] = paste0(prefix, zoom_bed[[1]])
    rbind(bed, zoom_bed)
}
```

Then add a new track:

```
bed = generateRandomBed(100)
circos.genomicTrackPlotRegion(extend_zoomed_chromosome_in_bed(bed, chromosome[1:2]),
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, pch = 16, cex = 0.5)
})
```

Also add link from original chromosome to the zoomed chromosome.

```
circos.link("chr1", get.cell.meta.data("cell.xlim", sector.index = "chr1"),
    "zoom_chr1", get.cell.meta.data("cell.xlim", sector.index = "zoom_chr1"),
    col = "#00000020")
circos.clear()
```

Same strategy can be applied if you only want to zoom sub regions inside a certain chromosome (just make them as pseudo chromosomes).

3 Create plotting regions

In following sections, chromosome will be used as the type of genomic category. In this section, we assume `data` is simply a data frame. For more complex situations and behaviour of the functions, we will introduce in the next section.

Similar as `circos.trackPlotRegion`, `circos.genomicTrackPlotRegion` also accepts a self-defined function `panel.fun` which is applied in every cell but with different form.

```
circos.genomicTrackPlotRegion(data, panel.fun = function(region, value, ...) {
    circos.genomicPoints(region, value, ...)
})
```

Inside `panel.fun`, users can use low-level genomic graphic functions to add basic graphics in each cell. `panel.fun` expects two arguments `region` and `value`. `region` is a data frame containing start position and end position in the current chromosome which is extracted from `data`, and you can think it corresponds to values on x-axes. `value` is also a data frame which contains other columns in `data`, and you can think it corresponds to values on y-axes. Besides, there should be a third arguments `...` which is mandatory and is used to pass user-invisible variables to inner functions (which we will explain in next sections).

Following codes demonstrate values for `region` and `value` when used inside `panel.fun`.

```
bed = generateRandomBed(nc = 1)
head(bed)

##    chr   start     end      value1
## 1 chr1   15804   90247  0.36038334
```

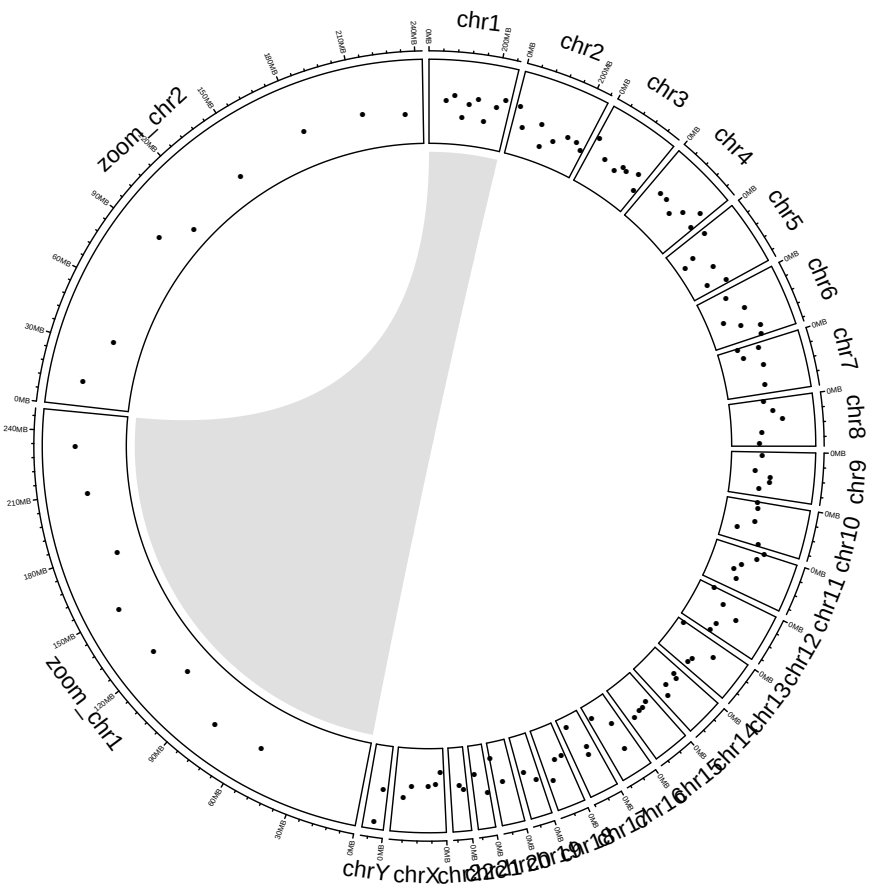

Figure 4: Zoom chromosomes.

```
## 2 chr1  122503  440777 -0.23412573
## 3 chr1  745473 1175215 -0.26368581
## 4 chr1 1662962 1781365 -0.94756937
## 5 chr1 1812844 1907062 -0.05621953
## 6 chr1 2060894 2353914  0.03344915

circos.initializeWithIdeogram(plotType = NULL)
circos.genomicTrackPlotRegion(bed, panel.fun = function(region, value, ...) {
    if(get.cell.meta.data("sector.index") == "chr1") {
        print(head(region))
        print(head(value))
    }
})

##     start     end
## 1   15804   90247
## 2  122503  440777
## 3  745473 1175215
## 4 1662962 1781365
## 5 1812844 1907062
## 6 2060894 2353914
##        value1
## 1  0.36038334
## 2 -0.23412573
## 3 -0.26368581
## 4 -0.94756937
## 5 -0.05621953
## 6  0.03344915

circos.clear()
```

Since `circos.genomicTrackPlotRegion` will create a new track, it needs values to calculate range of y-values to arrange data points. Users can either specify the index of numeric columns in `data` by `numeric.column` (**named index or numeric index**) or set `ylim`. If none of them are set, the function will try to look for all numeric columns in `data` (of course, excluding the first three columns), and set them as `numeric.column`.

```
circos.genomicTrackPlotRegion(data, ylim = c(0, 1),
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, ...)
})
circos.genomicTrackPlotRegion(data, numeric.column,
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, ...)
})
```

Since genomic functions are implemented by basic circos functions, you can use `circos.info` anywhere to get information of sectors and tracks.

3.1 Points

`circos.genomicPoints` is similar as `circos.points`. The difference is `circos.genomicPoints` expects a data frame containing genomic regions and a data frame containing values. The data column for plotting should be indicated by `numeric.column`. If the function is called inside `circos.genomicTrackPlotRegion` and users have been already set `numeric.column` in `circos.genomicTrackPlotRegion`, proper value of `numeric.column` will be passed to `circos.genomicPoints` through `...` in `panel.fun`. Which means, you need to add `...` as the final argument in `circos.genomicPoints` to get such information. If `numeric.column` is not set in both place, `circos.genomicPoints` will use all numeric columns detected in `value`.

Note here `numeric.column` is measured in `value` while `numeric.column` in `circos.genomicTrackPlotRegion` is measured in the complete data frame. There is a difference of 3! When `numeric.column` is passed to `circos.genomicPoints` internally, 3 is subtracted automatically. If you use character index instead of numeric index, you do not need to worry about it.

```
circos.genomicPoints(region, value, ...)
circos.genomicPoints(region, value, numeric.column = c(1, 2))
circos.genomicPoints(region, value, cex, pch)
circos.genomicPoints(region, value, sector.index, track.index)
```

If there is only one numeric column, graphical parameters such as `pch`, `cex` can be of length one or number of rows of `region`. If there are more than one numeric columns specified, points for each numeric column will be added iteratively, and the graphical parameters should be either length one or number of numeric columns specified.

`circos.genomicPoints` is implemented by `circos.points`. The basic idea of the implementation is like follows.

```
circos.genomicPoints = function(region, value, numeric.column = 1, ...) {
    x = (region[[2]] + region[[1]])/2
    y = value[[numeric.column]]
    circos.points(x, y, ...)
}
```

As shown above, `circos.genomicPoints` also provides arguments `sector.index` and `track.index`. The default values are current sector and current track. With these two arguments, you can add points by `circos.genomicPoints` outside `circos.genomicPlotTrackRegion`.

3.2 Lines

`circos.genomicLines` is similar as `circos.lines`. The setting of graphical parameters is similar as `circos.genomicPoints`.

```
circos.genomicLines(region, value, ...)
circos.genomicLines(region, value, numeric.column = c(1, 2))
circos.genomicLines(region, value, lwd, lty = "segment")
circos.genomicLines(region, value, area, baseline, border)
circos.genomicLines(region, value, sector.index, track.index)
```

For `lty`, we additionally provide a new option `segment` by which each genomic interval will represent as a 'horizontal' line.

3.3 Text

For `circos.genomicText`, the position of text can be specified either by `numeric.column` (index) or a separated vector `y`. The labels of text can be specified either by `labels.column` (index) or a vector `labels`.

```
circos.genomicText(region, value, ...)
circos.genomicText(region, value, y, labels)
circos.genomicText(region, value, numeric.column, labels.column)
circos.genomicText(region, value, facing, niceFacing, adj)
circos.genomicText(region, value, sector.index, track.index)
```

3.4 Rectangle

For `circos.genomicRect`, the positions of top and bottom of the rectangles can be specified by `ytop`, `ybottom` or `ytop.column`, `ybottom.column` (index).

```
circos.genomicRect(region, value, ytop = 1, ybottom = 0)
circos.genomicRect(region, value, ytop.column = 2, ybottom = 0)
circos.genomicRect(region, value, col, border)
```

One of the usage of `circos.genomicRect` is to plot heatmap on the circle. **circlize** provides a simple function `colorRamp2` to interpolate colors. The arguments of `colorRamp2` are break points and colors which correspond to the break points. `colorRamp2` returns a new function which can be used to generate new colors.

```
col_fun = colorRamp2(breaks = c(-1, 0, 1), colors = c("green", "black", "red"))
col_fun(c(-2, -1, -0.5, 0, 0.5, 1, 2))

## [1] "#00FF00FF" "#00FF00FF" "#008000FF" "#000000FF" "#800000FF" "#FF0000FF"
## [7] "#FF0000FF"

col_fun = colorRamp2(breaks = -log10(c(1, 0.05, 1e-4)),
    colors = c("green", "black", "red"))
p_value = c(0.8, 0.5, 0.001)
col_fun(-log10(p_value))

## [1] "#00EC00FF" "#00C400FF" "#A10000FF"
```

4 More details on `circos.genomicTrackPlotRegion`

The behaviour of `circos.genomicTrackPlotRegion` and `panel.fun` will be different according to different input data (e.g. is it a simple data frame or a list of data frames? If it is a data frame, how many numeric columns it has?) and different settings.

4.1 Normal mode

4.1.1 Input is a data frame

If input data is a simple data frame, `region` in `panel.fun` would be a data frame containing start position and end position in the current chromosome which is extracted from input data. `value` is also a data frame which contains columns in input data excluding the first three columns. Index of proper numeric columns will be passed by `...` if it is set in `circos.genomicTrackPlotRegion`. So if users want to use such information, they need to pass `...` to low-level genomic function such as `circos.genoimcPoints` as well.

If there are more than one numeric columns, all columns that are specified will be added to the plot.

```
bed = generateRandomBed(nc = 2)
# just note `numeric.column` is measured in `bed`
circos.genomicTrackPlotRegion(bed, numeric.column = 4,
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, ...)
        circos.genomicPoints(region, value)
        # here `numeric.column` is measured in `value`
        circos.genomicPoints(region, value, numeric.column = 1)
})
```

4.1.2 Input is a list of data frames

If input data is a list of data frames, `panel.fun` is applied on each data frame iteratively. Under such situation, `region` and `value` will contain corresponding data in the current data frame. The index for the current data frame can be get by `getI(...)`. For `numeric.column` argument, if input data is a list

of data frame, the length can only be one or the number of data frames, which means, there is only one numeric column that will be used in each data frame.

```
bedlist = list(generateRandomBed(), generateRandomBed())
circos.genomicTrackPlotRegion(bedlist,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicPoints(region, value, col = i, ...)
})

# column 4 in the first bed and column 5 in the second bed
circos.genomicTrackPlotRegion(bedlist, numeric.column = c(4, 5),
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicPoints(region, value, col = i, ...)
})
```

4.2 `stack` mode

`circos.genomicTrackPlotRegion` also supports a `stack` mode. Under `stack` mode, `ylim` is re-defined inside the function. The y-axis will be splitted into several layers with equal height and graphics are put on 'horizontal' layers (y = 1, 2, ...).

4.2.1 Input is a data frame

Under `stack` mode, when input data is a single data frame containing one or more numeric columns, each numeric column defined in `numeric.column` will be treated as a single unit. `ylim` is re-defined to `c(0.5, n+0.5)` in which `n` is number of numeric columns. `panel.fun` will be applied iteratively on each numeric column. In each iteration, in `panel.fun`, `region` is still the genomic regions in current genomic category, but `value` only contains current numeric column plus all non-numeric columns. All low-level genomic graphic functions will be applied on each 'horizontal line' `y = i` in which `i` is the index of current numeric column. The value of `i` can be obtained by `getI(...)`.

```
bed = generateRandomBed(nc = 2)
circos.genomicTrackPlotRegion(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicPoints(region, value, col = i, ...)
})
```

4.2.2 Input is a list of data frame

When input data is a list containing data frames, each data frame will be treated as a single unit. The situation is quite similar as described previously. `ylim` is re-defined to `c(0.5, n+0.5)` in which `n` is number of data frames. `panel.fun` will be applied iteratively on each data frame. In each iteration, in `panel.fun`, `region` is still the genomic regions in current chromosome, and `value` contains columns in current data frame excluding the first three columns. Still, graphics by low-level genomic functions will be added on the 'horizontal' lines.

```
bedlist = list(generateRandomBed(), generateRandomBed())
circos.genomicTrackPlotRegion(bedlist, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicPoints(region, value, ...)
})
```

Under `stack` mode, the differentce between using a data frame with multiple numeric columns and using a list of data frames is if using a data frame, all layers share the same genomic positions while if using a list of data frames, the genomic positions for each layer can be different.

5 Applications

In this section, we will show several real examples of making genomic plots under different modes.

5.1 Points

To make plots more clear to look at, we only add graphics in the 1/4 of the circle and initialize the plot only with chromosome 1 (figure 5).

```
par(mar = c(1, 1, 1, 1))
set.seed(999)

circos.par("track.height" = 0.1, start.degree = 90,
    canvas.xlim = c(0, 1), canvas.ylim = c(0, 1), gap.degree = 270)
circos.initializeWithIdeogram(chromosome.index = "chr1", plotType = NULL)
```

In track A, it is the most simple way to add points.

```
### track A
bed = generateRandomBed(nr = 300)
circos.genomicTrackPlotRegion(bed, panel.fun = function(region, value, ...) {
    circos.genomicPoints(region, value, pch = 16, cex = 0.5, ...)
})
```

In track B, under `stack` mode, points are added in one horizontal line (here we additionally add the dashed line) because there is only one numeric column in `bed`.

```
### track B
bed = generateRandomBed(nr = 300)
circos.genomicTrackPlotRegion(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, pch = 16, cex = 0.5, ...)

        i = getI(...)
        cell.xlim = get.cell.meta.data("cell.xlim")
        circos.lines(cell.xlim, c(i, i), lty = 2, col = "#00000040")
})
```

In track C, the input data is a list of two data frames. In the plot, sizes of the points correspond to the values of regions, and colors correspond to different data frames.

```
### track C
bed1 = generateRandomBed(nr = 300)
bed2 = generateRandomBed(nr = 300)
bed_list = list(bed1, bed2)
circos.genomicTrackPlotRegion(bed_list,
    panel.fun = function(region, value, ...) {
        cex = (value[[1]] - min(value[[1]]))/(max(value[[1]]) - min(value[[1]]))
        i = getI(...)
        circos.genomicPoints(region, value, cex = cex, pch = 16, col = i, ...)
})
```

In track D, plot the data frame list under `stack` mode.

```
### track D
circos.genomicTrackPlotRegion(bed_list, stack = TRUE,
    panel.fun = function(region, value, ...) {
        cex = (value[[1]] - min(value[[1]]))/(max(value[[1]]) - min(value[[1]]))
        i = getI(...)
```

```
        circos.genomicPoints(region, value, cex = cex, pch = 16, col = i, ...)
        cell.xlim = get.cell.meta.data("cell.xlim")
        circos.lines(cell.xlim, c(i, i), lty = 2, col = "#00000040")
})
```

In track E, the data frame has four numeric columns. Different colors are assigned to different columns. Note `value` is a data frame with four columns.

```
### track E
bed = generateRandomBed(nr = 300, nc = 4)
circos.genomicTrackPlotRegion(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, cex = 0.5, pch = 16, col = 1:4, ...)
})
```

In track F, the data frame has four columns but is plotted under `stack` mode. Note here `value` is a data frame with only one column (but it will be exexuted 4 times in each cell).

```
### track F
bed = generateRandomBed(nr = 300, nc = 4)
circos.genomicTrackPlotRegion(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        cex = (value[[1]] - min(value[[1]]))/(max(value[[1]]) - min(value[[1]]))
        i = getI(...)
        circos.genomicPoints(region, value, cex = cex, pch = 16, col = i, ...)

        cell.xlim = get.cell.meta.data("cell.xlim")
        circos.lines(cell.xlim, c(i, i), lty = 2, col = "#00000040")
})
circos.clear()
```

5.2 Lines

Initialize the plot with one quarter of the circle (figure 6).

```
par(mar = c(1, 1, 1, 1))
circos.par("track.height" = 0.1, start.degree = 90,
    canvas.xlim = c(0, 1), canvas.ylim = c(0, 1), gap.degree = 270)
circos.initializeWithIdeogram(chromosome.index = "chr1", plotType = NULL)
```

In track A, it is the most simple way to add lines.

```
### track A
bed = generateRandomBed(nr = 500)
circos.genomicTrackPlotRegion(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicLines(region, value, type = "l", ...)
})
```

In track B, add lines for a list of data frames. Different colors refers to different data frames.

```
### track B
bed1 = generateRandomBed(nr = 500)
bed2 = generateRandomBed(nr = 500)
bed_list = list(bed1, bed2)
circos.genomicTrackPlotRegion(bed_list,
    panel.fun = function(region, value, ...) {
```

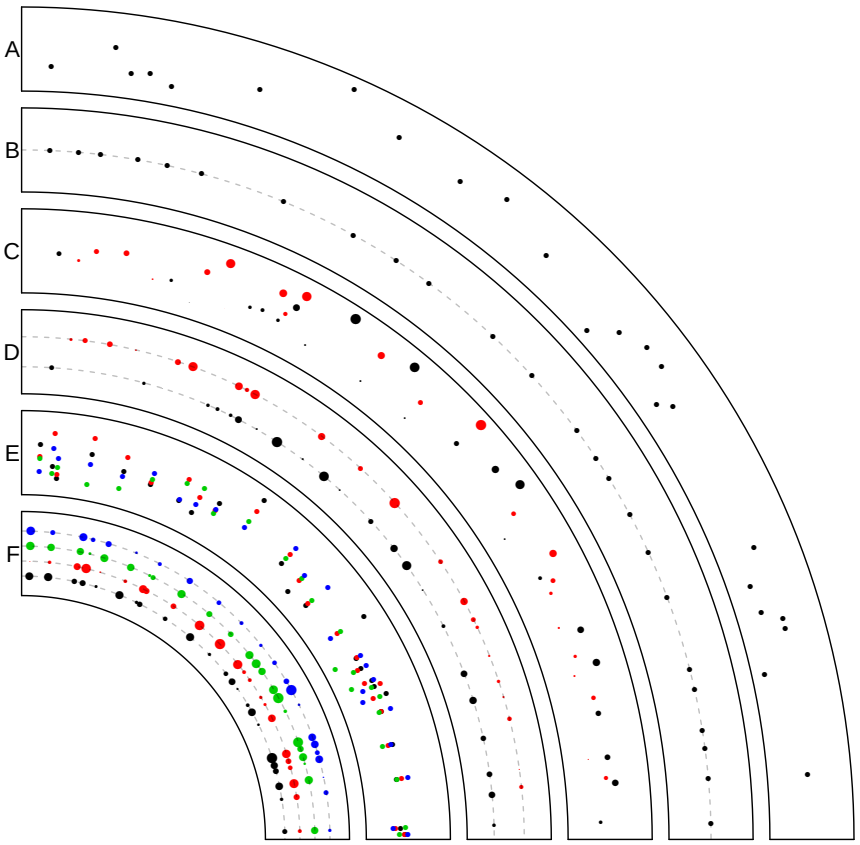

Figure 5: Add points under different modes.

```
        i = getI(...)
        circos.genomicLines(region, value, col = i, ...)
})
```

In track C, it is the `stack` mode of adding lines. Here the width of each line corresponds to the width of each genomic interval.

```
### track C
circos.genomicTrackPlotRegion(bed_list, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicLines(region, value, col = i, ...)
})
```

In track D, lines for the four numeric columns are added with different colors.

```
### track D
bed = generateRandomBed(nr = 500, nc = 4)
circos.genomicTrackPlotRegion(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicLines(region, value, col = 1:4, ...)
})
```

In track E, it is the `stack` mode for a data frame with more than one numeric columns.

```
### track E
bed = generateRandomBed(nr = 500, nc = 4)
circos.genomicTrackPlotRegion(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicLines(region, value, col = i, ...)
})
```

In track F, we specify `type` to `segment` and use different colors for segments. Note each segment is located at its corresponding y-value.

```
### track F
bed = generateRandomBed(nr = 200)
circos.genomicTrackPlotRegion(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicLines(region, value, type = "segment", lwd = 2,
            col = rand_color(nrow(region)), ...)
})
circos.clear()
```

5.3 Rectangles

Again, initialize the plot with one quarter of the circle (figure 7). Also, we want to map the values to colors by `f`.

```
par(mar = c(1, 1, 1, 1))
circos.par("track.height" = 0.1, start.degree = 90,
    canvas.xlim = c(0, 1), canvas.ylim = c(0, 1), gap.degree = 270)
circos.initializeWithIdeogram(chromosome.index = "chr1", plotType = NULL)
f = colorRamp2(breaks = c(-1, 0, 1), colors = c("green", "black", "red"))
```

In track A, `bed` has four numeric columns and `stack` mode is used to arrange the heatmap.

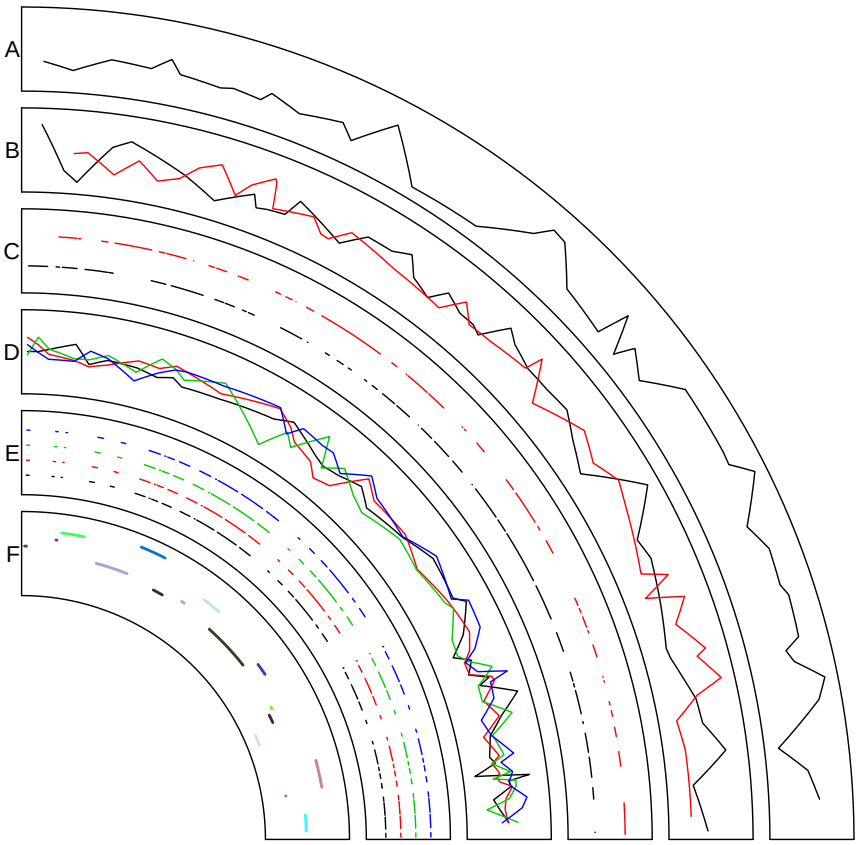

Figure 6: Add lines under different modes.

```
### track A
bed = generateRandomBed(nr = 100, nc = 4)
circos.genomicTrackPlotRegion(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        circos.genomicRect(region, value, col = f(value[[1]]), border = NA, ...)
})
```

In track B, add rectangles for a list of data frames. Comparing to track A, here genomic regions for each data frame are different, so the positions of rectangles in the first layer are different from that in the second layer.

Under `stack` mode, `ytop` and `ybottom` can be set to adjust the height of rectangles. It would be straightforward because for each layer, the position is `y = i` and the height of each rectangle is 1.

```
### track B
bed1 = generateRandomBed(nr = 100)
bed2 = generateRandomBed(nr = 100)
bed_list = list(bed1, bed2)
circos.genomicTrackPlotRegion(bed_list, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicRect(region, value, ytop = i + 0.4, ybottom = i - 0.4,
            col = f(value[[1]]), ...)
})
```

In track C, the plot are the same as track B, but without `stack` mode. Note here we explicitly specify `ylim`.

```
### track C
circos.genomicTrackPlotRegion(bed_list, ylim = c(0, 3),
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicRect(region, value, ytop = i + 0.4, ybottom = i - 0.4,
            col = f(value[[1]]), ...)
})
```

In track D, bars are added to the base line (`y = 0`).

```
### track D
bed = generateRandomBed(nr = 200)
circos.genomicTrackPlotRegion(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicRect(region, value, ytop.column = 1, ybottom = 0,
            col = ifelse(value[[1]] > 0, "red", "green"), ...)

        cell.xlim = get.cell.meta.data("cell.xlim")
        circos.lines(cell.xlim, c(0, 0), lty = 2, col = "#00000040")
})
circos.clear()
```

5.4 Mixed use of general circos functions

`panel.fun` is applied on each cell, which means, besides genomic functions, you can also use general circos functions to add more graphics. For example, some horizontal lines and texts are added to each cell and axes are put on top of each cell:

```
circos.genomicTrackPlotRegion(bed, ylim = c(-1, 1),
    panel.fun = function(region, value, ...) {
```

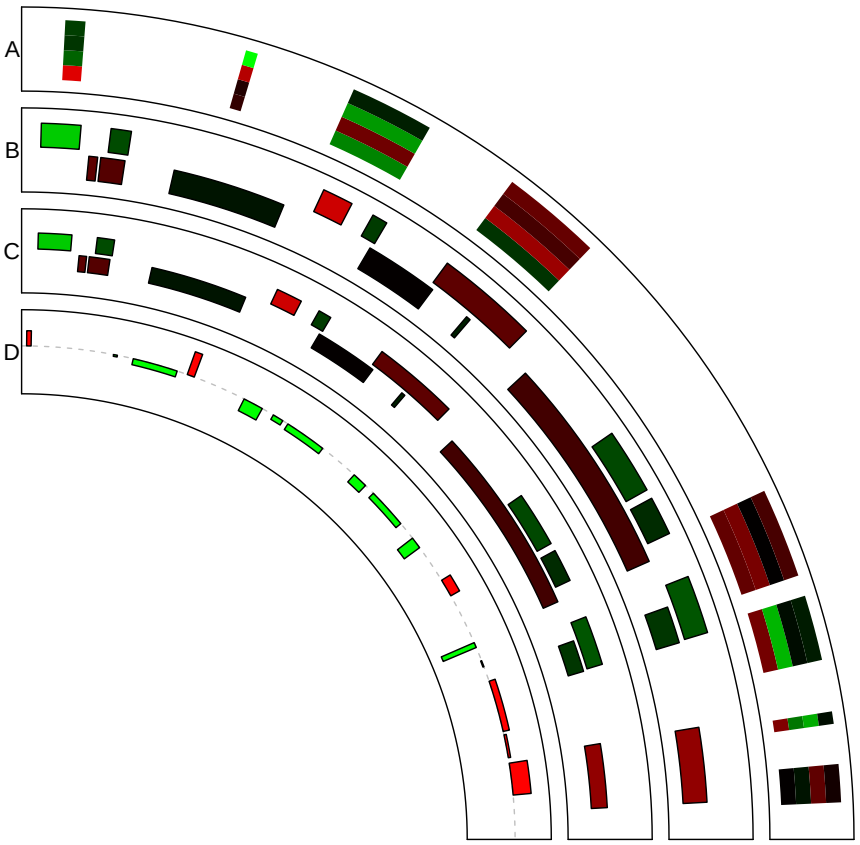

Figure 7: Add rectangles under different modes.

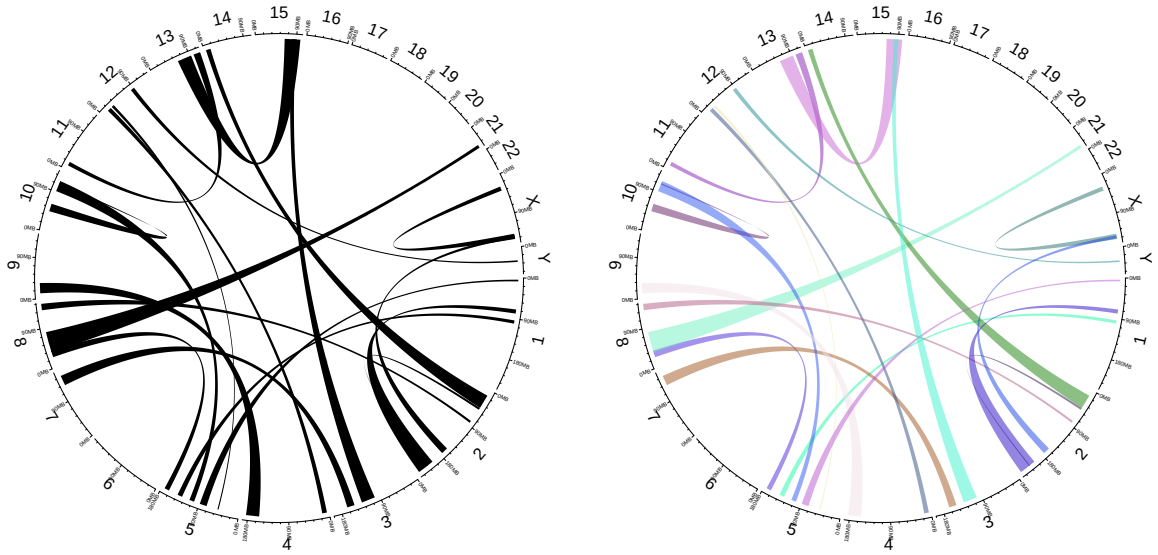

Figure 8: Add links from two sets of genomic regions.

```
        circos.genomicPoints(region, value, ...)

        cell.xlim = get.cell.meta.data("cell.xlim")
        for(h in c(-1, -0.5, 0, 0.5, 1)) {
            circos.lines(cell.xlim, c(0, 0), lty = 2, col = "grey")
        }
        circos.text(x, y, labels)
        circos.axis("top")
})
```

6 Links

`circos.genomicLink` expects two data frames and it will add links from genomic intervals in the first data frame to corresponding genomic intervals in the second data frame (figure 8). You can set graphical parameters as a single scalar or a vector.

```
bed1 = generateRandomBed(nr = 100)
bed1 = bed1[sample(nrow(bed1), 20), ]
bed2 = generateRandomBed(nr = 100)
bed2 = bed2[sample(nrow(bed2), 20), ]
circos.initializeWithIdeogram(plotType = c("axis", "labels"))
circos.genomicLink(bed1, bed2)
circos.clear()

circos.initializeWithIdeogram(plotType = c("axis", "labels"))
circos.genomicLink(bed1, bed2, col = rand_color(nrow(bed1), transparency = 0.5),
    border = NA)
```

7 Highlight chromosomes

`highlight.chromosome` provides a simple way to highlight chromosomes. Just remember to use transparent filled colors. The position of the highlighted regions can be fine-tuned by `padding` argument which are percent of corresponding height and width in the highlighted regions.

In following examples, we first create five more tracks with random points.

```
circos.par("track.height" = 0.1)
circos.initializeWithIdeogram(plotType = c("axis", "labels"))

for(i in 1:5) {
    bed = generateRandomBed(nr = 100)
    circos.genomicTrackPlotRegion(bed, panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, pch = 16, cex = 0.5, ...)
    })
}
```

Following are several ways of highlighting. we can highlight individual tracks by setting `track.index`. If two tracks or two chromosomes are neighbours, they will be taken as a union (figure 9).

```
highlight.chromosome(c("chrX", "chrY", "chr1"))
highlight.chromosome("chr3", col = "#00FF0040", padding = c(0.05, 0.05, 0.15, 0.05))
highlight.chromosome("chr5", col = NA, border = "red", lwd = 2,
    padding = c(0.05, 0.05, 0.15, 0.05))
highlight.chromosome("chr7", col = "#0000FF40", track.index = c(2, 4, 5))
highlight.chromosome(c("chr9", "chr10", "chr11"), col = NA, border = "green",
    lwd = 2, track.index = c(2, 4, 5))
highlight.chromosome(paste0("chr", c(1:22, "X", "Y")), col = "#FFFF0040",
    track.index = 6)
circos.clear()
```

`circos.info` would be helpful for you to find the correct track index. And if you only want to highlight some sub regions in one chromosome, use `draw.sector` directly.

8 High-level genomic functions

circlize implements several high-level functions which may help to visualize genomic data more effectively.

8.1 Position transformation

There is one representative situation when genomic position transformation needs to be applied. For example, there are regions which are quite densely located on the chromosome. If heatmap or text for those dense regions are to be plotted. They would be overlapped and hard to identify, also ugly to visualize. Thus, a way to transform original positions to new positions would help for the visualization.

Low-level genomic functions such as `circos.genomicPoints` all accept an argument `posTransform` to apply user-defined position transformation. Value for `posTransform` is a self-defined function which only accepts at least one argument: a data frame with two columns (start position and end position). Because `posTransform` is bound to low-level graphic functions, it is not necessary to add chromosome information in the input data. There is only one requirement for position transformation: Number of rows of regions should be the same before and after the transformation.

8.1.1 The default position transformation function

In **circlize**, there already provides a position transformation function `posTransform.default` which distributes positions uniformly in the current chromosome.

Following code does the transformation. The points are plotted with the new transformed regions.

```
circos.genomicTrackPlotRegion(data, panel.fun = function(region, value, ...) {
    circos.genomicPoints(region, value, posTransform = posTransform.default, ...)
})
```

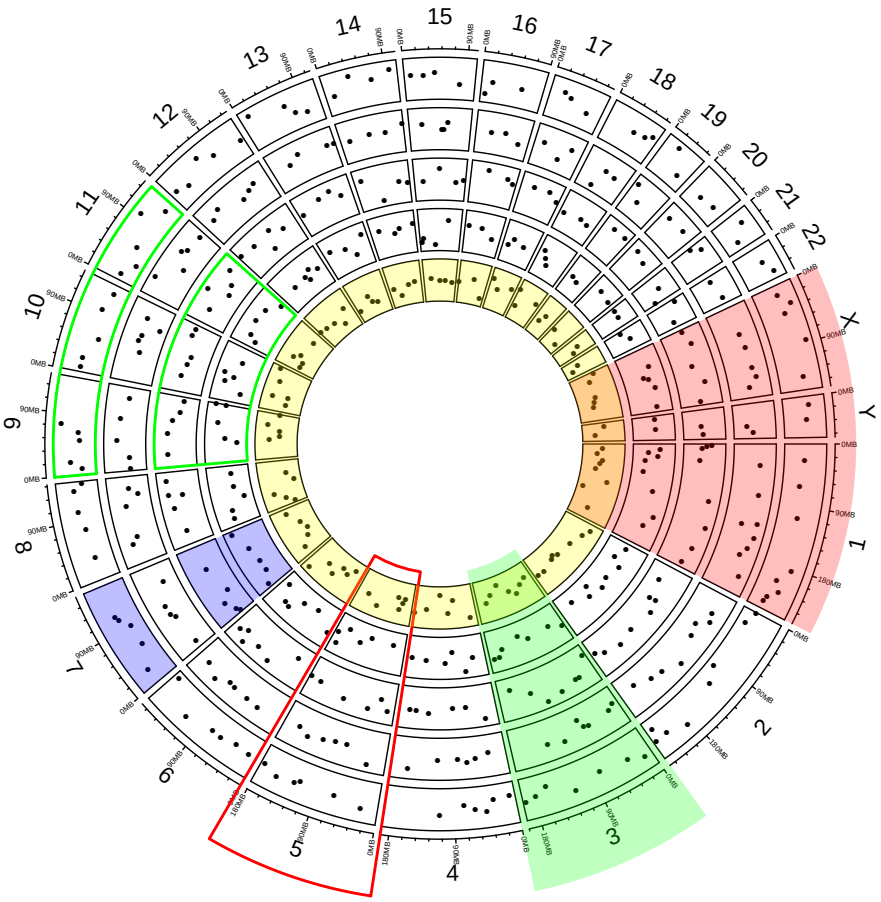

Figure 9: Highlight different chromosomes and tracks.

Of course we need identification lines to connect the untransformed regions to the transformed regions. There is a function called `circos.genomicPosTransformLines` which does the job. Same value for `posTransform` which was used before should be passed to `circos.genomicPosTransformLines`. Note `circos.genomicPosTransformLines` will create a new track to put such identification lines. In the function, `direction` controls whether the position transformed track is inside or outside the identification line track, and `horizontalLine` controls whether add lines to identify the regions.

```
circos.genomicPosTransformLines(data, posTransform = posTransform.default)
circos.genomicPosTransformLines(data, posTransform = posTransform.default,
    horizontalLine = "top")
circos.genomicPosTransformLines(data, posTransform = posTransform.default,
    direction = "outside")
```

In following example, we make heatmap for selected regions in the genome (figure 10 A).

```
circos.par(cell.padding = c(0, 0, 0, 0))
circos.initializeWithIdeogram()
bed = generateRandomBed(nr = 100, nc = 4)

# note how 'horizontalLine' works
circos.genomicPosTransformLines(bed, posTransform = posTransform.default,
    horizontalLine = "top", track.height = 0.1)

f = colorRamp2(breaks = c(-1, 0, 1), colors = c("green", "black", "red"))
circos.genomicTrackPlotRegion(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        circos.genomicRect(region, value, col = f(value[[1]]),
            border = f(value[[1]]), posTransform = posTransform.default, ...)
}, bg.border = NA)

circos.clear()
```

For the second example, the heatmap is plotted outside the identification lines, so here `direction` is set to `outside` (figure 10 B). Also note how we add the ideogram inside the circle.

```
circos.par(cell.padding = c(0, 0, 0, 0))
circos.initializeWithIdeogram(plotType = NULL)

circos.genomicTrackPlotRegion(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        circos.genomicRect(region, value, col = f(value[[1]]),
            border = f(value[[1]]), posTransform = posTransform.default, ...)
}, bg.border = NA)

circos.genomicPosTransformLines(bed, posTransform = posTransform.default,
    direction = "outside", horizontalLine = "bottom", track.height = 0.1)

cytoband = read.cytoband()$df
circos.genomicTrackPlotRegion(cytoband, stack = TRUE,
    panel.fun = function(region, value, ...) {
        circos.genomicRect(region, value, col = cytoband.col(value[[2]]), border = NA)
        xlim = get.cell.meta.data("xlim")
        ylim = get.cell.meta.data("ylim")
        circos.rect(xlim[1], ylim[1], xlim[2], ylim[2], border = "black")
}, track.height = 0.05, bg.border = NA)

circos.clear()
```

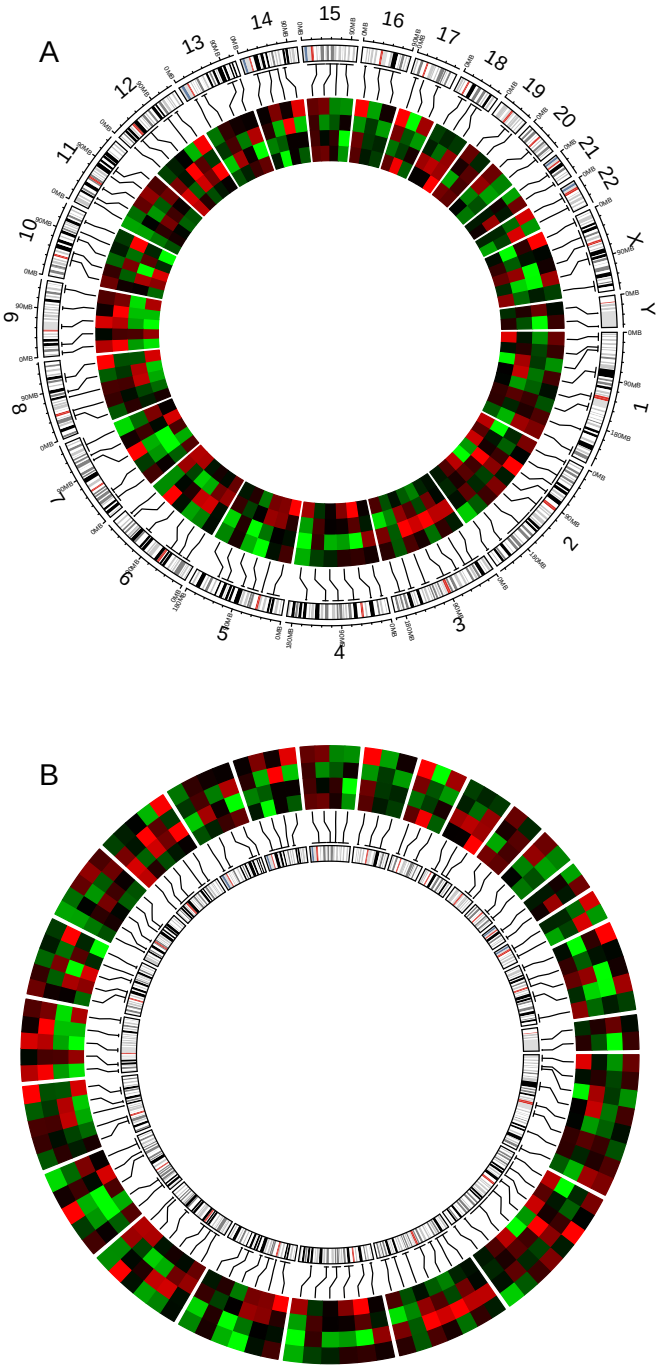

Figure 10: Position transformation with `posTransform.default`. A) transformation is inside; B) transformation is outside.

8.1.2 Position transformation for text

There is another position transformation function `posTransform.text` provided in **circlize** that can smartly position text on the circle. Normally, we don't want the text too away from the original position and also we want to get avoid of text overlapping. `posTransform.text` calculates the height of text and transform positions properly. Since such text position transformation relies on font size of text and which track the text is in, the usage of `posTransform.text` is a little bit complex. Currently, `posTransform.text` only makes sense if it is called inside `circos.genomicText` and only works when `facing` is set to `clockwise` or `reverse.clockwise`.

In following example code, using `posTransform.text` is quite similar as `posTransform.default` (but note this function is quit experimental):

```
bed = generateRandomBed(nr = 400, fun = function(k) rep("text", k))
circos.genomicTrackPlotRegion(bed, ylim = c(0, 1),
    panel.fun = function(region, value, ...) {
        circos.genomicText(region, value, y = 0, labels.column = 1,
            facing = "clockwise", adj = c(0, 0.5),
            posTransform = posTransform.text, cex = 0.8)
}, track.height = 0.1, bg.border = NA)
```

For the next track where position transformation lines are plotted. Code will be more complex. Since the calculation of the position of the line ends which corresponds to the transformed track relies on text settings (e.g. font size, font family) and track for the text (i.e. the track will affect the position of text), we need to go back to the track where text position transformation happened and use the same settings for the text. Such information should be collected when plotting transformation lines. The solution by **circlize** is to wrap `posTransform.text` with such information, then text information will be recovered and the transformation will be calculated in the correct track.

As we mentioned before, `posTransform` expect at least one argument, so the argument `value` is completely fine here.

```
i_track = get.cell.meta.data("track.index")  # the nearest track
# we put `y`, `labels`, ... into a self-defined function
# because these parameters will affect the text position
circos.genomicPosTransformLines(bed, direction = "outside",
    posTransform = function(region, value)
        posTransform.text(region, y = 0, labels = value[[1]],
            cex = 0.8, track.index = i_track)
)
```

If the transformation line track is plotted before the track where text position is transformed, things will be more complicated. Since `circos.genomicPosTransformLines` needs information of the text, but at that moment, track which transforms text positions has not be created. The solution is first creating an empty track for position transformation lines, then adding the position transformation track. Finally go back to the transformation line track to add transformation lines.

```
circos.genomicTrackPlotRegion(bed, ylim = c(0, 1), track.height = 0.1, bg.border = NA)
i_track = get.cell.meta.data("track.index")  # remember this empty track, we'll come back

circos.genomicTrackPlotRegion(bed, ylim = c(0, 1),
    panel.fun = function(region, value, ...) {
        circos.genomicText(region, value, y = 1, labels.column = 1,
            facing = "clockwise", adj = c(1, 0.5),
            posTransform = posTransform.text, cex = 0.8)
}, track.height = 0.1, bg.border = NA)
tr_track = get.cell.meta.data("track.index") # position transformation track

# because `circos.genomicPosTransformLines` is implemented by
# `circos.trackPlotRegion`, it accepts `track.index` argument.
circos.genomicPosTransformLines(bed,
```

```
    posTransform = function(region, value)
        posTransform.text(region, y = 1, labels = value[[1]],
            cex = 0.8, track.index = tr_track),
    direction = "inside", track.index = i_track
)
```

Padding of text after transformation can be set through `padding` argument to adjust the space between two neighbouring text. Of course, when add transformation lines, `padding` should also be wrapped in.

```
circos.genomicTrackPlotRegion(bed, ylim = c(0, 1),
    panel.fun = function(region, value, ...) {
        circos.genomicText(region, value, y = 0, labels.column = 1,
            facing = "clockwise", adj = c(0, 0.5), posTransform = posTransform.text,
            cex = 0.8, padding = 0.2)
}, track.height = 0.1, bg.border = NA)

i_track = get.cell.meta.data("track.index")  # previous track
circos.genomicPosTransformLines(bed,
    posTransform = function(region, value) posTransform.text(region, y = 0,
        labels = value[[1]], cex = 0.8, padding = 0.2, track.index = i_track),
    direction = "outside"
)
```

For examples and comparisons between `posTransform.default` and `posTransform.text` when visualizing text, please refer to figure 11. Source code is available in the help page of `posTransform.text`.

8.1.3 How about if I have a lot of labels?

Position transformation is applied inside each chromosome, which means, the transformed new positions are still located in the original chromosome. This would cause a problem that if you have a lot of labels to put, they will overlap with each other (figure 12 B). Unfortunately there is no simple solution to extend the positions outside of the chromosome.

Anyway, we provide a 'not-too-perfect' solution but the code is a little bit lengthy (figure 12 A). The idea is to put labels and transformation lines in one same track. Here we use `posTransform.text` directly to calculate the transformed positions.

To make a clear demonstration, we only add graphics in the first quarter of the circle, only on one chromosome.

```
set.seed(999)
bed = generateRandomBed(nr = 800, fun = function(k) rep("text", k))

par(mar = c(1, 1, 1, 1))
circos.par(start.degree = 75, canvas.xlim = c(0, 1), canvas.ylim = c(0, 1),
    gap.degree = 300, cell.padding = c(0, 0, 0, 0), track.margin = c(0, 0))
circos.initializeWithIdeogram(plotType = NULL, chromosome.index = "chr1")
```

The steps inside `panel.fun` are:

1. Original positions are plotted as dots.
2. In order to get rid of text overlapping, we separate the text into two groups. One group of text are plotted lower than the other group. In following code, the x-axis is splitted into 6 intervals and `region` and `value` are separated into groups by looking at whether the middle points of corresponding regions are in the odd intervals (i.e., the first interval, the third interval, ...) or even intervals. Of course, in this step, users should determine their own rule to separate the groups.
3. Text in different groups are assigned with different heights (y).

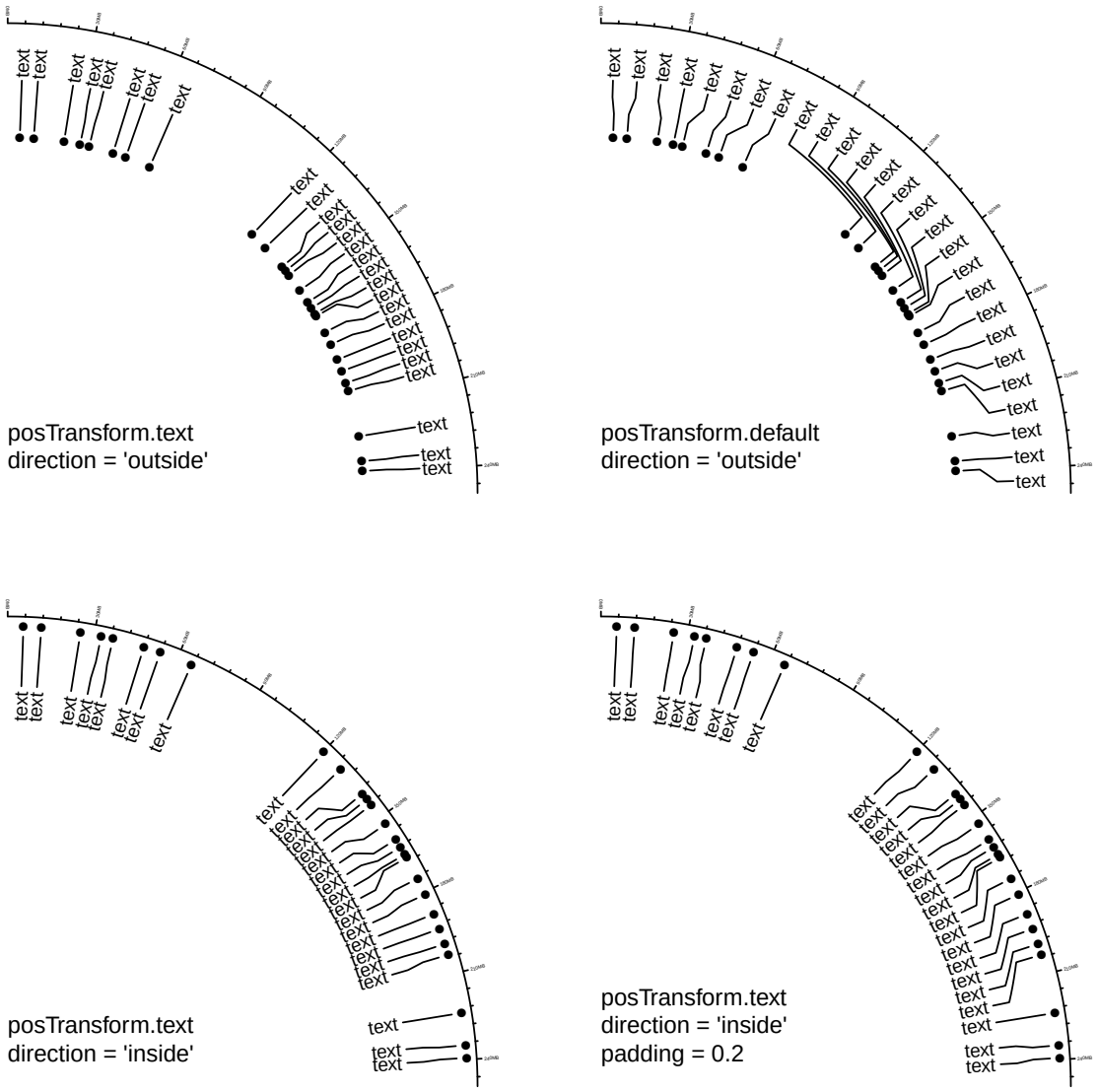

Figure 11: Transformation of text positions.

4. Do the position transformation on text and add labels.
5. Add the position transformation lines manually. Note the line for lower text and higher text are different. We do something more to make sure lines which points to higher text do not overlap with lower text too much.

```
circos.genomicTrackPlotRegion(bed, ylim = c(0, 1),
    panel.fun = function(region, value, ...) {

    # original positions
    circos.genomicPoints(region, data.frame(rep(0, nrow(region))), pch = 16)

    xlim = get.cell.meta.data("xlim")
    breaks = seq(xlim[1], xlim[2], length.out = 7)
    midpoints = (region[[1]] + region[[2]])/2
    for(i in seq_along(breaks)[-1]) {
        # index for current interval
        l = midpoints >= breaks[i - 1] & midpoints < breaks[i]
        # if there is no data in this interval
        if(sum(l) == 0) next

        # sub-regions in this interval
        sub_region = region[l, , drop = FALSE]
        sub_value = value[l, , drop = FALSE]

        # note here i == 2, 4, 6, ... corresponds to odd intervals
        # text in odd intervals are in lower position
        if(i %% 2 == 0) {
            y = 0.2
        } else {
            y = 0.8
        }

        # get the transformed position and add text with new positions
        tr_region = posTransform.text(sub_region, y = y, labels = sub_value[[1]],
            cex = 0.8, adj = c(0, 0.5))
        circos.genomicText(tr_region, sub_value, labels.column = 1, y = y,
            adj = c(0, 0.5), facing = "clockwise", niceFacing = TRUE, cex = 0.8)

        # add position transformation lines for odd intervals
        if(i %% 2 == 0) {
            for(i in seq_len(nrow(sub_region))) {
                x = c( (sub_region[i, 1] + sub_region[i, 2])/2,
                       (sub_region[i, 1] + sub_region[i, 2])/2,
                       (tr_region[i, 1] + tr_region[i, 2])/2,
                       (tr_region[i, 1] + tr_region[i, 2])/2)
                y = c(0, 0.2/3, 0.2/3*2, 0.2)
                circos.lines(x, y)
            }
        } else { # add position transformation lines for even intervals
            median_sub_region_midpoint = median(midpoints[l])
            sub_region_width = max(midpoints[l]) - min(midpoints[l])
            for(i in seq_len(nrow(sub_region))) {
                x = c( (sub_region[i, 1] + sub_region[i, 2])/2,
                       (sub_region[i, 1] + sub_region[i, 2])/2,
                      median_sub_region_midpoint +
                          sub_region_width*(i - nrow(sub_region))/nrow(sub_region) * 0.2,
                      median_sub_region_midpoint +
```

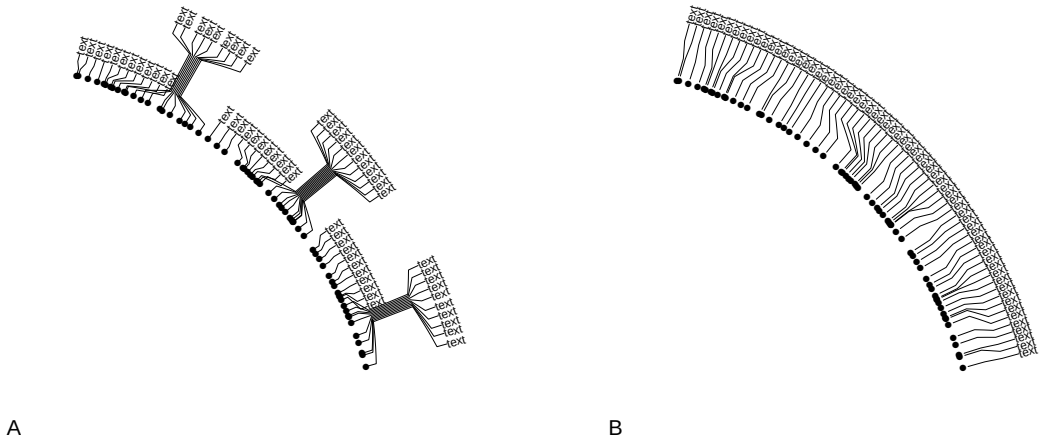

Figure 12: Position transformation for a lot of text. A) put text on two layers; B) put text on one layer.

```
                        sub_region_width*(i - nrow(sub_region))/nrow(sub_region) * 0.2,
                    (tr_region[i, 1] + tr_region[i, 2])/2,
                    (tr_region[i, 1] + tr_region[i, 2])/2)
                y = c(0, 0.1, 0.2, 0.6, 0.7, 0.8)
                circos.lines(x, y)
            }
        }
    }

}, track.height = 0.2, bg.border = NA)

circos.clear()
```

You can self-define positions of the 'necks' of the long transformation lines so that they will not overlap with the lower text.

8.2 Genomic density and Rainfall plot

`circos.genomicDensity` calculates how much a genomic window is covered by regions in `bed`. The input data can be a single data frame or a list of data frames.

```
circos.genomicDensity(bed)
circos.genomicDensity(bed, baseline = 0)
circos.genomicDensity(bed, window.size = 1e6)
circos.genomicDensity(bedlist, col = c("#FF000080", "#0000FF80"))
```

Rainfall plot can be used to visualize distribution of regions. On the plot, y-axis corresponds to the distance to neighbour regions (log10-based). So if there is a drop-down on the plot, it means there is a cluster of regions at that area (figure 13). The input data can be a single data frame or a list of data frames.

```
circos.genoimcRainfall(bed)
circos.genoimcRainfall(bedlist, col = c("red", "green"))
```

Following example makes a rainfall plot for differentially methylated regions (DMR) and their genomic densities. In the plot, red corresponds to hyper-methylated DMRs (gain of methylation) and blue corresponds to hypo-methylated DMRs (loss of methylation) (figure 13).

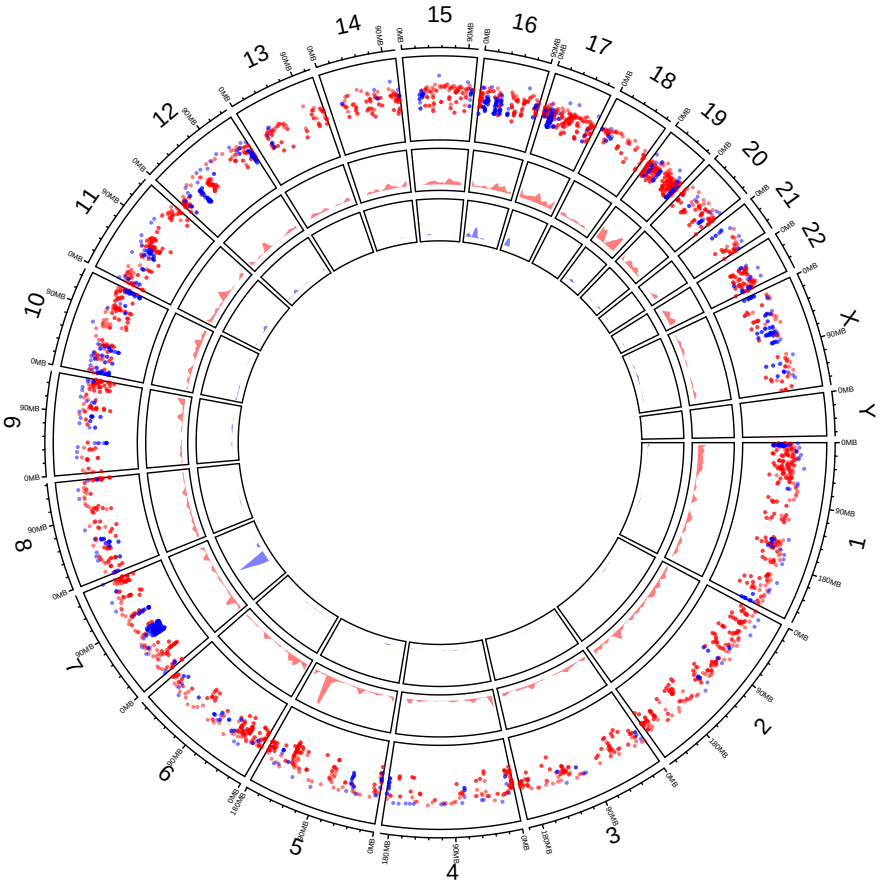

Figure 13: Genomic rainfall plot and densities.

```
load(paste0(system.file(package = "circlize"), "/extdata/DMR.RData"))
par(mar = c(1, 1, 1, 1))
circos.initializeWithIdeogram(plotType = c("axis", "labels"))

bed_list = list(DMR_hyper, DMR_hypo)
circos.genomicRainfall(bed_list, pch = 16, cex = 0.4, col = c("#FF000080", "#0000FF80"))
circos.genomicDensity(DMR_hyper, col = c("#FF000080"), track.height = 0.1)
circos.genomicDensity(DMR_hypo, col = c("#0000FF80"), track.height = 0.1)
```

```
circos.clear()
```