An introduction to *circlize* package

Zuguang Gu <z.gu@dkfz.de>

August 2, 2014

1 Introduction

Circular layout is very useful to represent complicated information, especially for genomic data. It has advantages to visualize data with long axes or large amount of categories, described with different measurements. Another unique feature for circular layout is it is effective to visualize relations between elements.

Circos (`http://circos.ca`) is an extraordinarily cool tool to draw such circular layout and it is broadly used in real applications, not just popular in Genomic but in a lot of other areas as well. It is not only a way to visualize data, but enhances the representation of scientific results into a level of aesthetics. Therefore, most people call figures with circular layout as 'circos plot'. Here the *circlize* package aims to implement *Circos* in R. One advantage for the implementation in R is that R is an ideal environment which provides seamless connection between data analysis and data visualization. This package is not a front-end wrapper to generate configuration files for *Circos*, but completely coded in R style by using R's elegant statistical and graphic engine. We aim to keep the flexibility and configurability of *Circos*, also make the package more straightforward to use and enhance it to support more types of graphics.

2 Principle of design

Since most of the figures are composed of simple graphics, such as points, lines, polygon (for filled colors) *et al*, *circlize* implements low-level graphical functions for adding graphics in circular layout, so that more higher level graphics can be easily comprised by low-level graphics. This principle ensures the generality that types of high-level graphics are not restricted by the software but determined by users.

Currently there are following graphical functions that can be used for plotting, they are similar to the functions without `circos.` prefix from the traditional graphical engine (you can also see the correspondence in figure 1):

- `circos.points`: add points in a cell, similar as `points`.
- `circos.lines`: add lines in a cell, similar as `lines`.
- `circos.rect`: add rectangle in a cell, similar as `rect`.
- `circos.polygon`: add polygon in a cell, similar as `polygon`.
- `circos.text`: add text in a cell, similar as `text`.
- `circos.axis`: add axis in a cell, functionally similar as `axis` but with more features.
- `circos.link`: this maybe the unique feature for circos layout to represent relationships between elements.

For adding points, lines and text in cells through the whole track (among several sectors), the following functions are available:

- `circos.trackPoints`: this can be replaced by `circos.points` through a `for` loop.
- `circos.trackLines`: this can be replaced by `circos.lines` through a `for` loop.
- `circos.trackText`: this can be replaced by `circos.text` through a `for` loop.

`circos.trackPlotRegion` $\longrightarrow$ `plot.default`

`circos.points` $\longrightarrow$ `points`

`circos.lines` $\longrightarrow$ `lines`

`circos.text` $\longrightarrow$ `text`

`circos.rect` $\longrightarrow$ `rect`

`circos.polygon` $\longrightarrow$ `polygon`

`circos.axis` $\longrightarrow$ `axis`

Figure 1: Correspondence between graphic functions in *circlize* and in traditional R graphic engine.

Functions to arrange the circular layout:

- `circos.trackPlotRegion`: create plotting regions of cells in a track
- `circos.updatePlotRegion`: update an existed cell
- `circos.par`: graphic parameters
- `circos.info`: print general parameters of current circos plot
- `circos.clear`: reset graphic parameters and internal variables

Theoretically, you are able to draw most kinds of circos figures by the above functions. As you will see, most of figures in the six vignettes are generated by *circlize* package.

The following part of this vignette is structured as follows: First there is an example to give a quick glance of how to implement a circular layout by *circlize*. Then it tells you the basic principle (or the order of using the circos functions) for plotting. After that there are detailed explanations of graphic parameters, coordinates transformation and low-level functions. Finally it will tell you some tricks for drawing more complicated circos plot.

3 A quick glance

Following is an example to show the basic feature and usage of *circlize* package. First let's generate some data. There needs a factor to represent categories, values on x-axis, and values on y-axis.

```
> set.seed(12345)
> n = 1000
> a = data.frame(factor = sample(letters[1:8], n, replace = TRUE),
+     x = rnorm(n), y = runif(n))
```

Initialize the layout. In this step, the `circos.initialize` function allocates sectors in the circle according to ranges of x-values in different categories. E.g, if there are two categories, range for x-values in the first category is `c(0, 2)` and range for x-values in the second category is `c(0, 1)`, the first category would hold approximately 67% areas of the circle. Here we only need x-values because all cells in a sector share the same x-ranges.

```
> library(circlize)
> par(mar = c(1, 1, 1, 1), lwd = 0.1, cex = 0.7)
> circos.par("default.track.height" = 0.1)
> circos.initialize(factors = a$factor, x = a$x)
```

Draw the first track (figure 2, top left). Before drawing any track we need to know that all tracks should firstly be created by `circos.trackPlotRegion`, then those low-level functions can be applied (recall in traditional R graphic engine, you need first call `plot.default` and then you can use functions such as `points` and `lines` to add graphics on it). Since xlims for cells in the track have already been defined in the initialization step, so here we only need to specify the ylim for each cell, either by `y` or `ylim` argument.

We also add axes for cells in the first track, The axis for each cell is added by `panel.fun` argument. `circos.trackPlotRegion` creates plotting region cell by cell and the `panel.fun` is actually executed immediately after the creation of the plotting region for a certain cell. So `panel.fun` actually means adding graphics in the "current cell". After that, add points through the whole track by `circos.trackPoints`. Finally, add two texts in a certain cell (the cell is specified by `sector.index` and `track.index` argument). When adding the second text, we do not specify `track.index` because the package knows we are now in the first track.

Here what should be noted is that the first track has a index number of 1. An internal variable which traces the tracks would set the 'current track index' to 1. So if the track index is not specified in the plotting functions such as `circos.trackPoints` and `circos.text` which are called after the creation of the track, the current track index would be used ad the default track index. (details would be explained in the following sections).

```
> circos.trackPlotRegion(factors = a$factor, y = a$y,
+     panel.fun = function(x, y) {
+         circos.axis()
+ })
> col = rep(c("#FF0000", "#00FF00"), 4)
> circos.trackPoints(a$factor, a$x, a$y, col = col,
+     pch = 16, cex = 0.5)
> circos.text(-1, 0.5, "left", sector.index = "a", track.index = 1)
> circos.text(1, 0.5, "right", sector.index = "a")
```

Draw the second track (figure 2, top right). There are histograms in the track. The `circos.trackHist` can also create a new track because drawing histogram is really high level, so we do not need to call `circos.trackPlotRegion`. The track index for this track is 2.

```
> bgcol = rep(c("#EFEFEF", "#CCCCCC"), 4)
> circos.trackHist(a$factor, a$x, bg.col = bgcol, col = NA)
```

Draw the third track (figure 2, middle left). Here some meta data for a cell can be obtained by `get.cell.meta.data`. This function needs `sector.index` and `track.index` arguments, and if they are not specified, it means it is the current sector index and the current track index.

```
> circos.trackPlotRegion(factors = a$factor, x = a$x, y = a$y,
+   panel.fun = function(x, y) {
+       grey = c("#FFFFFF", "#CCCCCC", "#999999")
+       i = get.cell.meta.data("sector.numeric.index")
+       circos.updatePlotRegion(bg.col = grey[i %% 3 + 1])
+       circos.points(x[1:10], y[1:10], col = "red", pch = 16, cex = 0.6)
+       circos.points(x[11:20], y[11:20], col = "blue", cex = 0.6)
+   })
```

You can update an existed cell by specifying `sector.index` and `track.index` in `circos.updatePlotRegion`. The function erases graphics which have been added. Here we erase graphics in one cell in track 2, sector `d` and re-add some points (figure 2, middle right). `circos.updatePlotRegion` can not modify the `xlim` and `ylim` of the cell as well as other settings related to the position of the cell.

```
> circos.updatePlotRegion(sector.index = "d", track.index = 2)
> circos.points(x = -2:2, y = rep(0, 5))
```

Draw the fourth track (figure 2, bottom left). Here you can choose different line types which is similar as `type` argument in `lines`.

```
> circos.trackPlotRegion(factors = a$factor, y = a$y)
> circos.trackLines(a$factor[1:100], a$x[1:100], a$y[1:100], type = "h")
```

Draw links (figure 2, bottom right). Links can be from point to point, point to interval or interval to interval. Some of the arguments would be explained in the following sections.

```
> circos.link("a", 0, "b", 0, h = 0.4)
> circos.link("c", c(-0.5, 0.5), "d", c(-0.5,0.5), col = "red",
+     border = "blue", h = 0.2)
> circos.link("e", 0, "g", c(-1,1), col = "green", lwd = 2, lty = 2)
```

Finally we need to reset the graphical parameters and internal variables, so that it will not mess up your next plotting.

```
> circos.clear()
```

The final figure looks like figure 2.

4 Details

In this section, more details of the package would be explained.

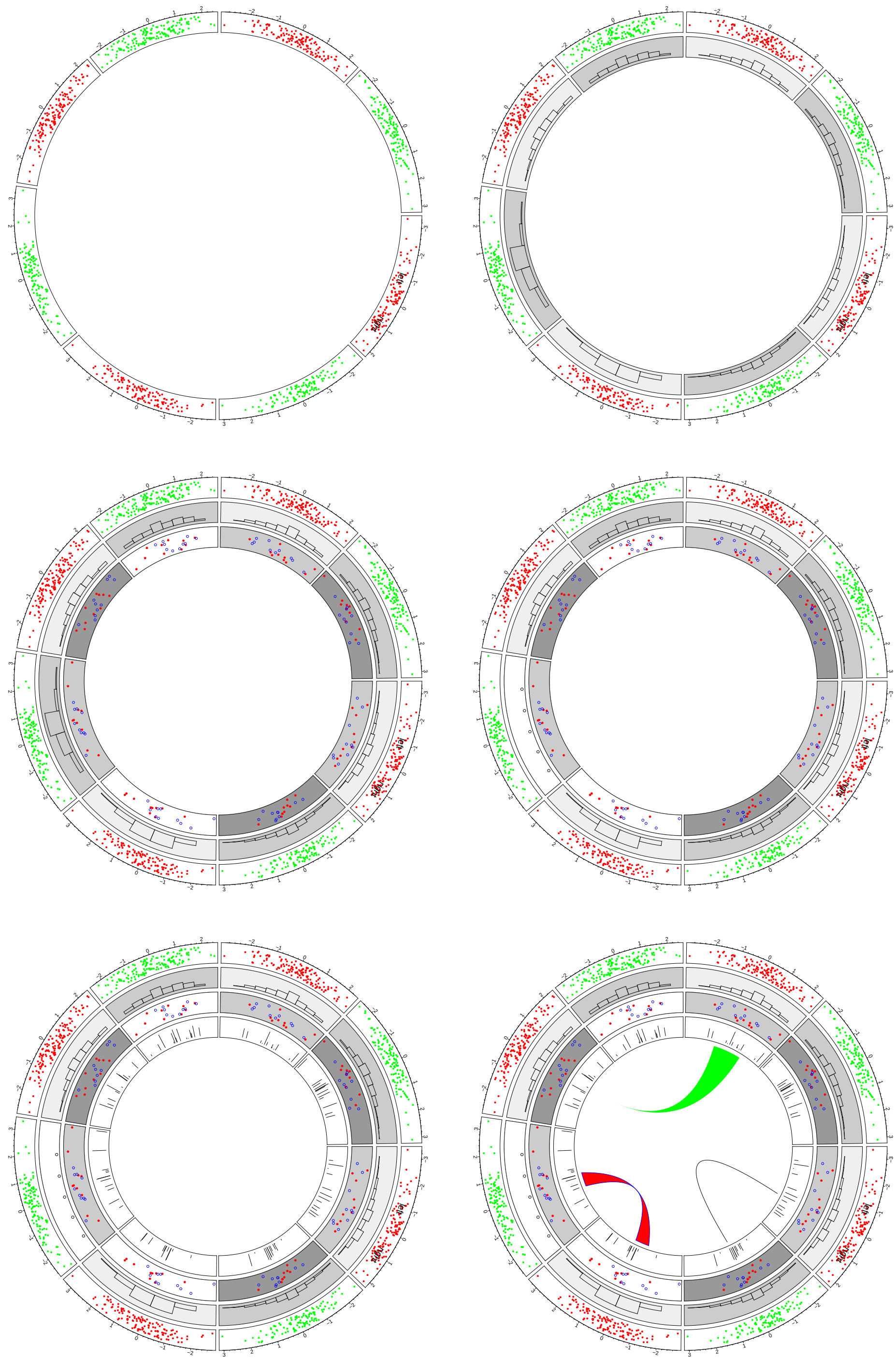

Figure 2: A step-by-step example by *circlize*

4.1 Coordinate transformation

There is a **data coordinate** in which the range for x-axis and y-axis is the range of data, a **polar coordinate** to allocates graphics on the circle and a **canvas coordinate** which really draws the graphics (figure 3). The package would first transform from the data coordinate to a polar coordinate and finally transform into the canvas coordinate.

The finnal canvas coordinate is in fact an ordinary coordinate in R plotting system with x-range from -1 to 1 and y-range from -1 to 1 by default.

It should be noted that the circular layout is always (or mostly except you want to draw something out of the circle) drawn inside the circle which has radius of 1 (unit circle), from outside to inside.

However, for users, they only need to imagine that each cell is a normal rectangular plotting region (data coordinate) in which x-lim and y-lim are ranges of data in the category respectively. The *circlize* package would know which cell you are drawing in and do all the transformations.

4.2 Rules for drawing circular layout

The rules for drawing circular layout is rather simple. It follows the sequence of "initialize - create track - add graphics - create track - add graphics - ... - clear" (figure 4). Details are as follows:

1. Initialize the layout using `circos.initialize`. Since circular layout in fact visualizes data which is in categories, there should be a factor and a x-range variable to allocate categories into sectors.

2. Create plotting regions for the new track and add graphics. The new track is created just inside the previously created one and the index of the track is added by 1 automatically. Only after the creation of the track can you add other graphics on it. There are three ways to do the plotting job.

   (a) After the creation of the track, use low-level graphic function like `circos.points`, `circos.lines`, ... to add graphics cell by cell. It always involves a `for` loop.

   (b) Use `circis.trackPoints`, `circos.trackLines`, ... to draw same style of graphics through all cells simultaneously. However, it is not recommended because it would make you a little confused and also it cannot draw complicated graphics.

   (c) Use `panel.fun` argument in `circos.trackPlotRegion` to draw graphics immediately after the creation of a certain cell. `panel.fun` needs two arguments `x` and `y` which are x-values and y-values that in the current category. This subset operation would be applied automatically. This is the most recommended way.

   Plotting regions for cells which have already been created can be updated by `circos.updatePlotRegion`. `circos.updatePlotRegion` will erase everything that you added before.

   Low level functions such as `circos.points` can be applied to any created cell by specifying `sector.index` and `track.index`.

3. Repeat step 2 to add more tracks on the circle.

4. Call `circos.clear` to do cleaning.

As mentioned above, there are three ways to add graphics on the created track. 1. create plotting regions for the whole track and then add graphics by specifying `sector.index` and `track.index`. In the following pseudo code, `x1`, `y1` are data points in a given cell, which means you need to do data subsetting by yourself.

```
> circos.initialize(factors, xlim)
> circos.trackPlotRegion(factors, ylim)
> for(sector.index in all.sector.index) {
+     circos.points(x1, y1, sector.index)
+     circos.lines(x2, y2, sector.index)
+ }
```

2. add graphics through a batch mode. This can be replaced by `circos.points` or `circos.lines` in a `for` loop. In the following code, you need to specifying the `factors` and now `x1` and `y1` are data points for all categories. The data points for a given cell will be subsetted according the `factors`.

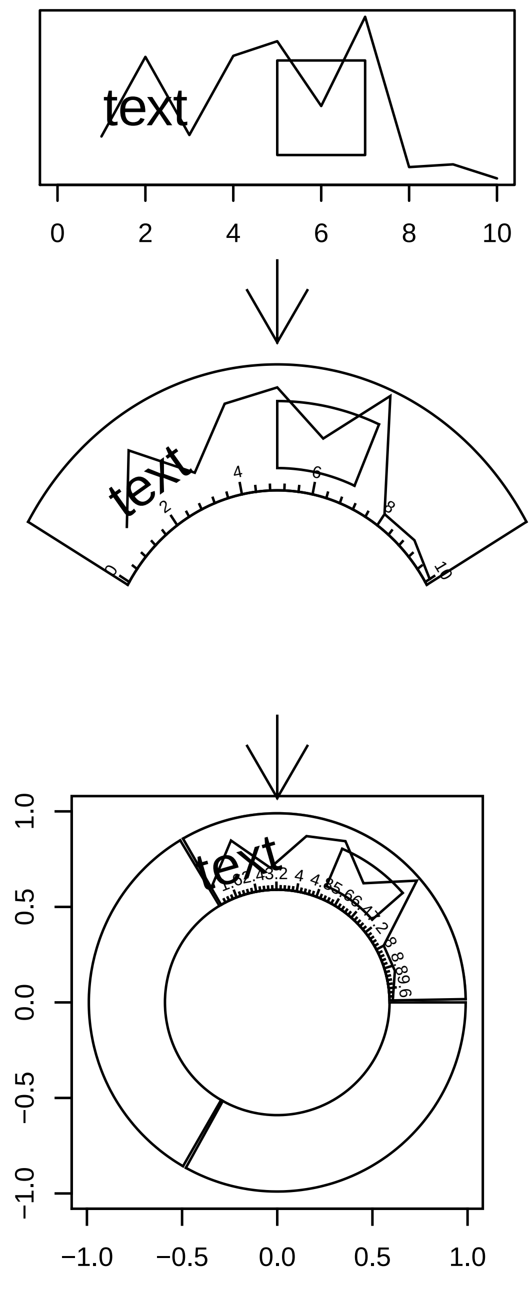

Figure 3: Transformation between different coordinates. Top: data coordinate; Middle: polar coordinate; Bottom: canvas coordinate.

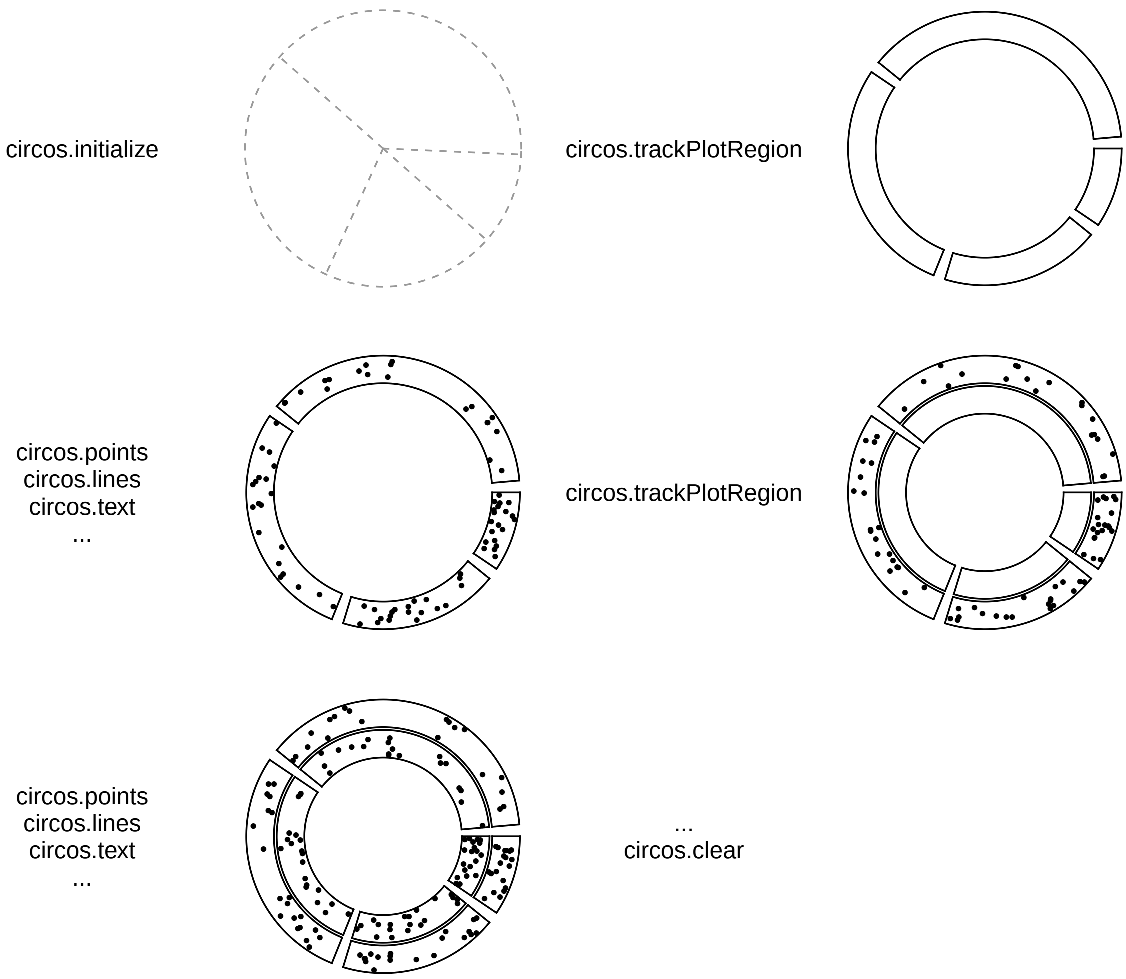

Figure 4: Order of drawing circular layout

```
> circos.initialize(factors, xlim)
> circos.trackPlotRegion(factors, ylim)
> circos.trackPoints(factors, x1, y1)
> circos.trackLines(factors, x2, y2)
```

3. use a panel function to add self-defined graphics as soon as the cell has been created. This is the way recommended since when you look at `panel.fun`, it is just like adding graphics in traditional R graphics system. There will be a more detailed explanation of `panel.fun` argument in the following section.

```
> circos.initialize(factors, xlim)
> circos.trackPlotRegion(factors, all_x, all_y, ylim,
+   panel.fun = function(x, y) {
+     circos.points(x, y)
+     circos.lines(x, y)
+ })
```

There is several internal variables keeping tracing of the current sector and track when applying `circos.trackPlotRegion` and `circos.updatePlotRegion`. So although functions like `circos.points`, `circos.lines` need to specify the index for sector and track, they will take the current calculated ones by default. As a result, if you draw points, lines, text, *et al* just after the creation of the track or cell, you do not need to set the sector index and the track index explicitly and it will be put in the most recently created cell. Note again, only `circos.trackPlotRegion` and `circos.updatePlotRegion` can reset the current track index and sector index.

Finally, in *circlize* package, function with prefix `circos.track` would affect all cells in a track.

4.3 Sectors and tracks

A circular layout is composed of sectors and tracks, as illustrated in figure 5. The red circle is the track and the blue one is the sector. The intersection of a sector and a track is called a cell which can be thought as an imaginary plotting region for data points in a certain category (data coordinate).

Sectors are first allocated on the circle and determined by `circos.initialize`, then track allocation is determined by `circos.trackPlotRegion`. `circos.initialize` needs a category variable and data value which implicates the range of data in each category. The range of data can be specified either by `x` or `xlim`.

```
> circos.initialize(factors, x)
> circos.initialize(factors, xlim)
```

There are something very important that should be noted in the initialization step. In this step, not only the width of each sector is assigned, but also the order of sectors on the circle is determined. **Order of the sectors are determined by the order of levels of the factor**. So if you want to change the order of the sectors, just change of the level of the `factors` variable. The following codes would generate different figures (figure 6):

```
> fa = c("d", "f", "e", "c", "g", "b", "a")
> f1 = factor(fa)
> circos.initialize(factors = f1, xlim = c(0, 1))
> f2 = factor(fa, levels = fa)
> circos.initialize(factors = f2, xlim = c(0, 1))
```

If `x` which is the x-values corresponding to `factors` is specified, the range for x-values in different categories would be calculated according to `factors` automatically. And if `xlim` is specified, it should be either a matrix which has same number of rows as the length of the level of `factors` or a two-element vector. If it is a two-element vector, it would be extended to a matrix which has the same number of rows as the length of `factors` levels. Here, every row in `xlim` corresponds to the x-ranges of a category and the order of rows in `xlim` corresponds to the order of levels of `factors`.

Under the default settings, width of sectors are calculated according to the range of data in each category. In some circumstance, you many want to manually set the width of each sector (figure 7). Normally it is not a good idea to change the default settings, since width of sectors can reflect useful

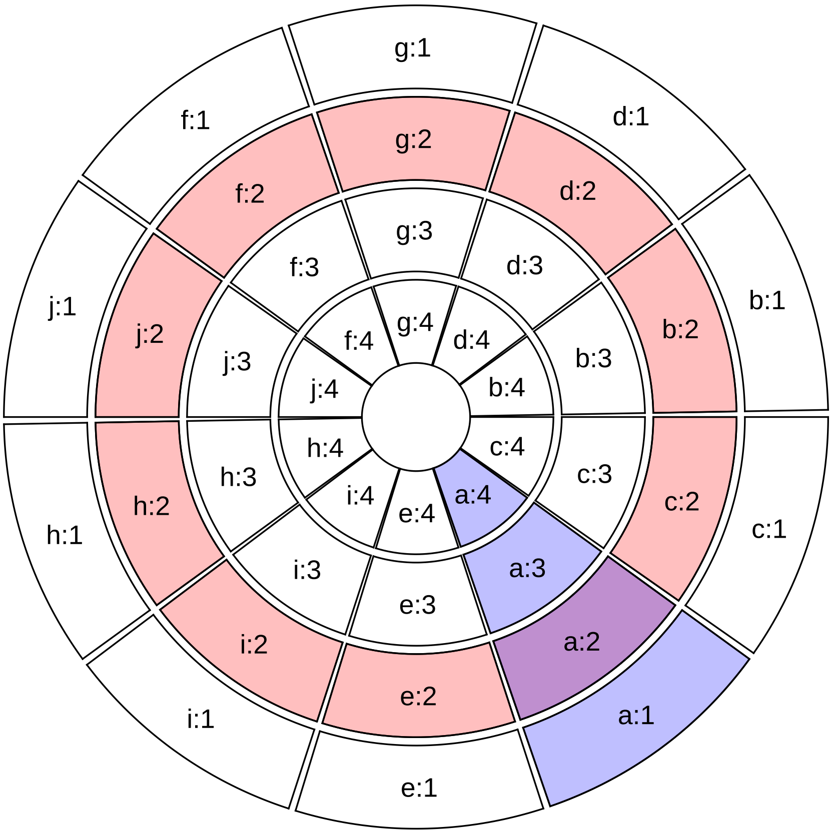

Figure 5: Coordinate in circos layout

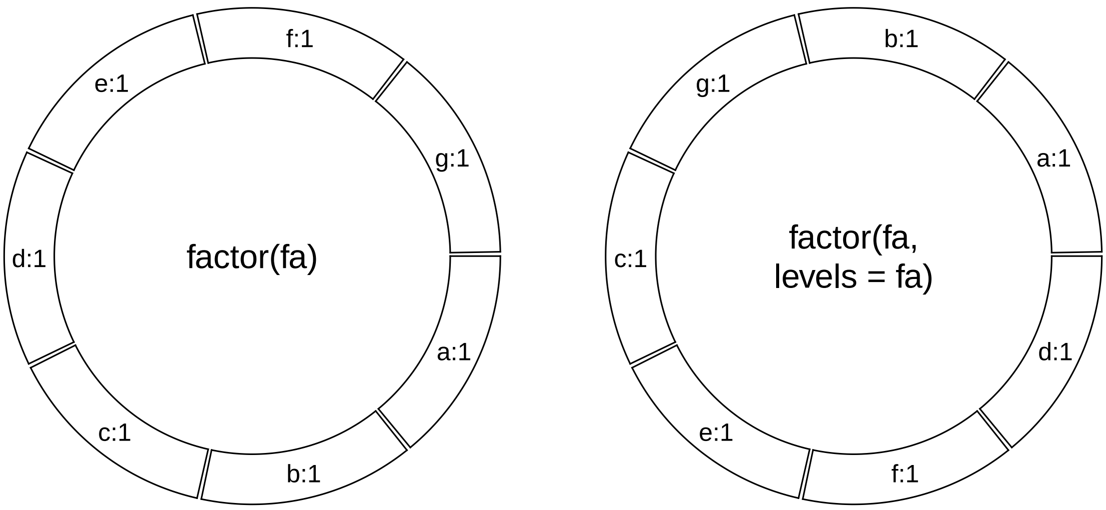

Figure 6: Different `factor` order in the initialization step.

information of your data. However, sometimes it is useful to set the width of sectors such as you want to draw your data only in half of the circle and in the other half of the circle, you want to zoom in some part of your data. For example, someone may want to draw 24 chromosomes in half of the circle and zoom in two chromosomes at the other half of the circle. The width of sectors can be manually set by `sector.width` argument in `circos.initialize`. The value for the argument should be a vector with length of either one or as same as the number of categories (again, order of `sector.width` vector corresponds to the order of levels of `factors`). `sector.width` is some kind of ratio or relative values, and the values will be scaled to percentage later (e.g. if you set `sector.width` to `c(1, 3)`, it will be scaled as `c(0.25, 0.75)`).

Example codes for sector zooming look like:

```
> factors = sample(letters[1:6], 100, replace = TRUE)
> x = rnorm(100)
> y = rnorm(100)
> zoomed_factor = factors[factors %in% c("a", "b")]
> zoomed_factor[zoomed_factor == "a"] = "a_zoomed"
> zoomed_factor[zoomed_factor == "b"] = "b_zoomed"
> zoomed_x = x[factors %in% c("a", "b")]
> zoomed_y = y[factors %in% c("a", "b")]
> # attached to the origin data
> factors = c(factors, zoomed_factor)
> factors = factor(factors, levels = c(letters[1:6], "a_zoomed", "b_zoomed"))
> x = c(x, zoomed_x)
> y = c(y, zoomed_y)
> xrange = tapply(x, factors, function(x) max(x) - min(x))
> # see how to scale sector 1:6 to first half of circle and scale sector 7:8
> # to the other half
> circos.initialize(factors, x = x,
+     sector.width = c(xrange[1:6]/sum(xrange[1:6]), xrange[7:8]/sum(xrange[7:8])))
> circos.trackPlotRegion(factors, x = x, y = y, panel.fun = function(x, y) {
+     circos.points(x, y)
+     xlim = get.cell.meta.data("xlim")
+     ylim = get.cell.meta.data("ylim")
+     sector.index = get.cell.meta.data("sector.index")
+     circos.text(mean(xlim), ylim[2] + 0.2*(ylim[2] - ylim[1]),
+         sector.index, adj = c(0.5, 0))
+ })
> # if you want to add links from unzoomed sectors to zoomed sectors
> circos.link("a", get.cell.meta.data("cell.xlim", sector.index = "a"),
+     "a_zoomed", get.cell.meta.data("cell.xlim", sector.index = "a_zoomed"),
+     border = NA, col = "#00000010")
> circos.clear()
```

Since all cells in one sector and in different tracks share the same x-ranges, for each track, we only need to specify the y-ranges for cells. Similar as `circos.initialize`, `circos.trackPlotRegion` can also receive either `y` or `ylim` argument to specify the range of y-values. There is also a `force.ylim` argument to specify whether all cells in one same track should share the same y-ranges. `force.ylim` is only used along with `y`.

```
> circos.trackPlotRegion(factors, y)
> circos.trackPlotRegion(factors, ylim)
```

In the track creation step, since all sectors are already allocated in the circle, if `factors` argument is not set, `circos.trackPlotRegion` would create plotting regions for all available sectors. Also, levels of `factors` do not need to be specified explicitly because the order of sectors has already be determined in the initialization step. If `factors` is just a vector, it would be converted to factor automatically. And if users just create cells for part of sectors in the track (not all sectors), in fact, cells in remaining unspecified sectors would also be created as well, but with no borders (pretending they are not created).

Cells are basic units in the circle and are independent with each other. After the creation of cells, they have self-contained meta values of x-lim and y-lim (range of data in the category, data coordinate,

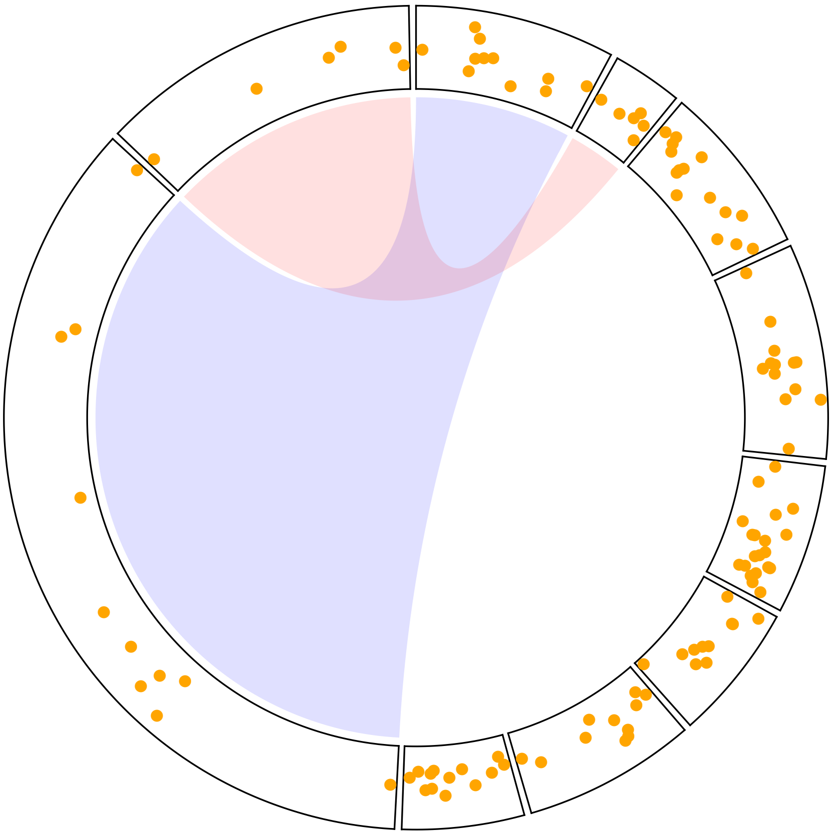

Figure 7: manually set the sector width

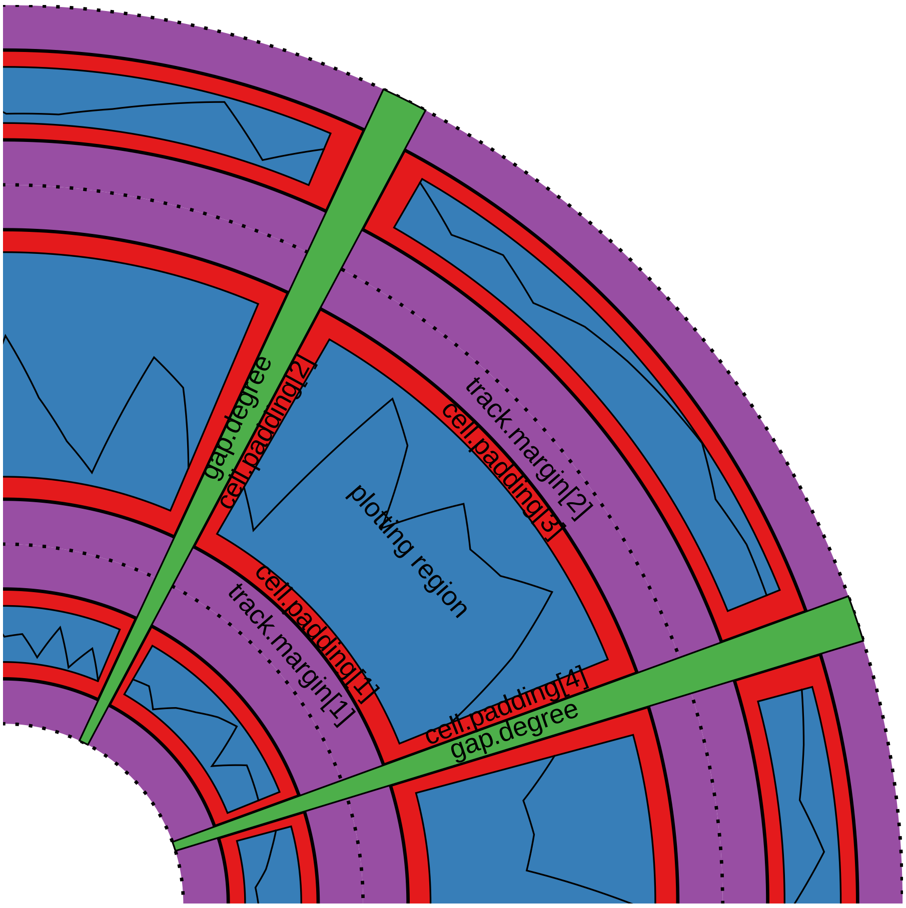

Figure 8: Regions for a cell

...). So if you are adding graphics in one cell, you do not need to consider things outside the cell and also you do not need to consider you are in the circle. Just pretending it is rectangle area.

4.4 Graphic parameters

Some basic parameters for the circular layout can be set through `circos.par`. The parameters are as follows, note some parameters can only be assigned before the initialization of the circular layout.

- `start.degree`: The starting degree where to put the first sector. Note this degree is measured in the standard polar coordinate which means it is always reverse clockwise. See figure 9.
- `gap.degree`: Gap between two neighbour sectors. It can be a single value which means all gaps share same degree, or a vector which has same length as levels of the factors. The first gap is after the first sector. See figure 9 and figure 8.
- `track.margin`: Like `margin` in Cascading Style Sheets (CSS), it is the blank area out of the plotting region, also outside of the borders. Since left and right margin are controlled by `gap.degree`, only bottom and top margin need to be set. The value for the `track.margin` is the percentage according to the radius of the unit circle. See figure 8.
- `cell.padding`: Padding of the cell. Like `padding` in Cascading Style Sheets (CSS), it is the blank area around the plotting regions, but within the borders. The parameter has four values, which control the bottom, left, top and right padding respectively. The first and the third padding values are the percentages according to the radius of the unit circle and the second and fourth values are the degrees. See figure 8.

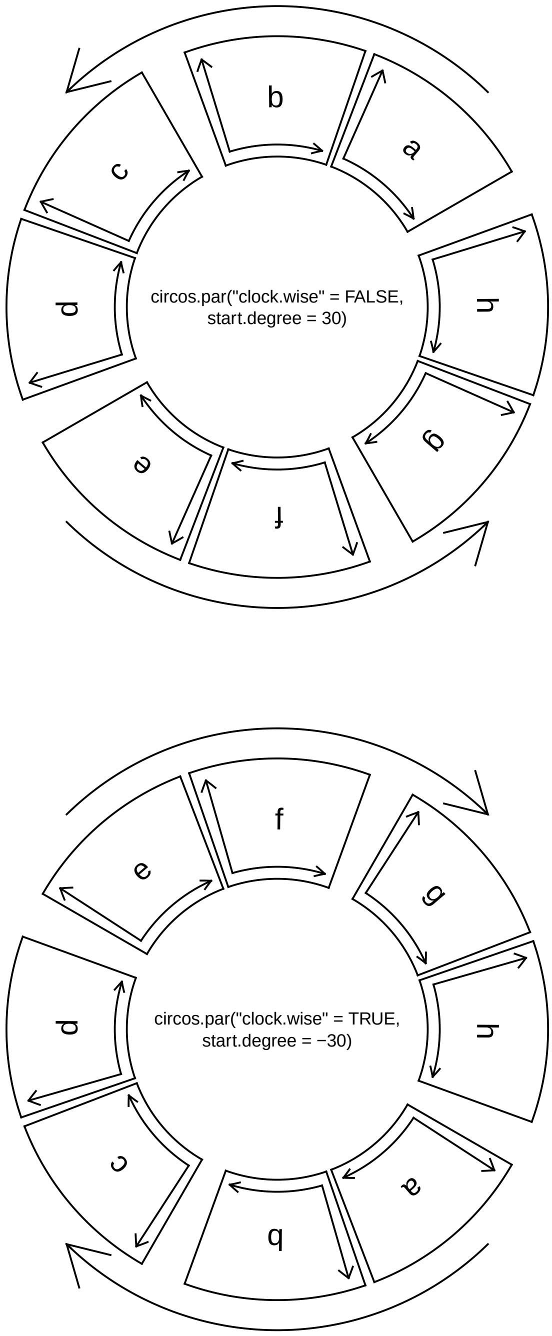

Figure 9: Sector directions. Sector orders are `a, b, ..., h`.

- `unit.circle.segments`: Since curves are simulated by a series of straight lines, this parameter controls the amount of segments to represent a curve. The minimal length of the line segment is the length of the unit circle (`2*pi`) divided by `unit.circle.segments`. More segments means better approximation for the curves while larger file size if you generate figures as PDF format. (The name of this parameter may be a little hard to understand. I will try to change it with a better one.)
- `default.track.height`: The default height of tracks. It is the percentage according to the radius of the unit circle. The height includes the top and bottom cell paddings but not the margins.
- `points.overflow.warning`: Since each cell is in fact not a real plotting region but only an ordinary rectangle (or more precisely, rectangle-like), it does not remove points that are plotted outside of the region. So if some points are out of the plotting region, by default, the package would continue drawing the points and print warnings. But in some circumstances, draw something out of the plotting region is useful, such as draw some legend or text. Set this value to `FALSE` to turn off the warnings.
- `canvas.xlim`: The coordinate for the canvas. The package is forced to draw unit circle, so the `xlim` and `ylim` for the canvas would be `c(-1, 1)` by default. However, you can set it to a more broad interval if you want to draw other things out of the circle. By choose proper `canvas.xlim` and `canvas.ylim`, you can draw part of the circle. E.g. setting `canvas.xlim` to `c(0, 1)` and `canvas.ylim` to `c(0, 1)` would only draw circle in the region of `(0, pi/2)`.
- `canvas.ylim`: The coordinate for the canvas.
- `clock.wise`: The order of drawing sectors. Default is `TRUE` which means clockwise (figure 9). But note that inside each cell, the direction of x-axis is always clockwise and direction of y-axis is always from inside to outside in the circle.

Default values for graphic parameters are in table 1.

parameter	default value
`start.degree`	`0`
`gap.degree`	`1`
`track.margin`	`c(0.01, 0.01)`
`cell.padding`	`c(0.02, 1.00, 0.02, 1.00)`
`unit.circle.segments`	`500`
`default.track.height`	`0.2`
`points.overflow.warning`	`TRUE`
`canvas.xlim`	`c(-1, 1)`
`canvas.ylim`	`c(-1, 1)`
`clock.wise`	`TRUE`

Table 1: Default graphic parameters

Parameters related to the allocation of sectors cannot be changed after the initialization of the circos layout. So `start.degree`, `gap.degree`, `canvas.xlim`, `canvas.ylim` and `clock.wise` can only be modified before `circos.initialize`. The second and the fourth element of `cell.padding` (left and right paddings) can not be modified either (or will be ignored).

4.5 Create plotting region

As described above, only after creating the plotting region can you add low-level graphics on it. The minimal set of arguments for this function is to set either `y` or `ylim` (remember order of rows of `ylim` corresponds to the order of levels of `factors` and if `factors` is not specified, then it corresponds to the order of default sectors which is determined in the initialization step) which will assign range of y-values for the track. `circos.trackPlotRegion` create tracks for all sectors although in some case only part of them is visible. If you only want to create the plotting regions but do not draw anything, you do not need to specify `factors` and `x`. While if you want to draw something by `panel.fun` as soon as you create the plotting region, then you need to specify `factors` and `x` to pass the categories and values to `panel.fun`. Graphic arguments such as `bg.border` and `bg.col` can either be a scalar or a vector. If it

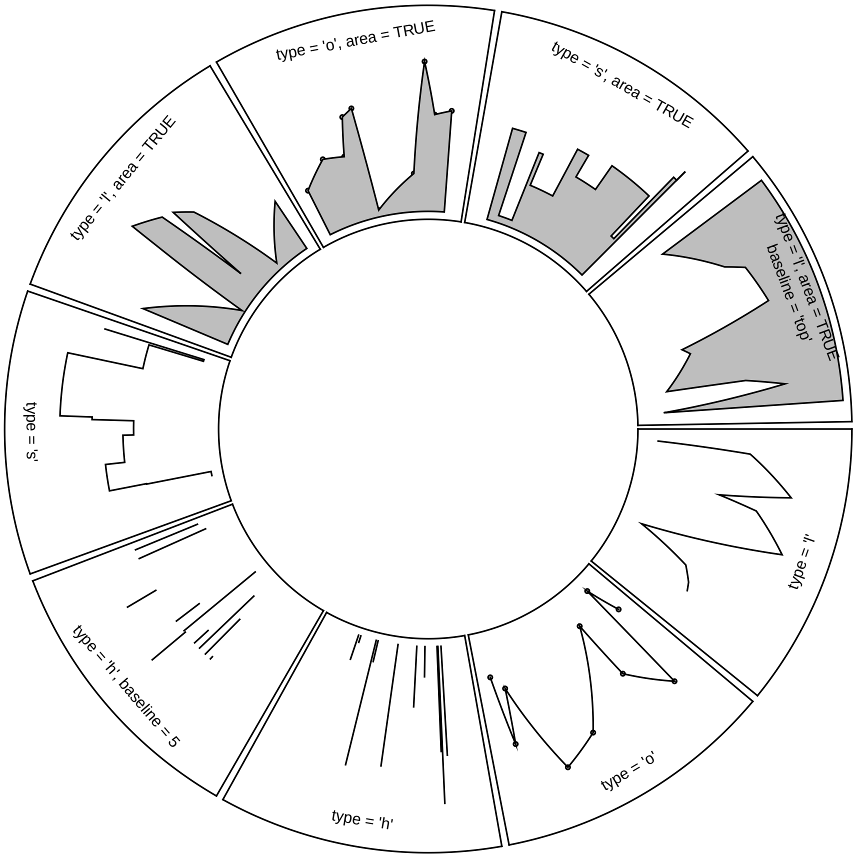

Figure 10: Line style settings

is a vector, the length must be equal to the length of levels of `factors` and the order of these graphics arguments is also as same as the order of levels of `factors`. With setting these graphics arguments you can create plot regions with different styles of borders and background colors. If you are confused with the `factors` orders, you can also customize the borders and background colors for the plotting region in `panel.fun` (using `get.cell.meta.data("cell.xlim")` and `get.cell.meta.data("cell.ylim")` to get the position of the plotting region).

4.6 Update plotting region

If `track.index` is specified in `circos.trackPlotRegion` and the specified track is already created, the track will be updated with new graphics. In this case, settings related to the positions of the track such as the height of the track can not be modified.

```
> circos.trackPlotRegion(data, ylim = c(0, 1), track.index = 1, ...)
```

For single cell, `circos.updatePlotRegion` can be used to erase all graphics that have been already plotted in the cell. In this case, you cannot re-define y-ranges in the cell.

```
> circos.updatePlotRegion(sector.index, track.index)
> circos.points(x, y, sector.index, track.index)
```

4.7 Points

Adding points by `circos.points` is similar as `points` function. Possible usage is:

```
> circos.points(x, y)
> circos.points(x, y, sector.index, track.index)
> circos.points(x, y, pch, col, cex)
```

Since `circos.points` is a low-level function, it can only be applied to those cells which have been created. If `sector.index` or `track.index` is not specified, it would use the 'current' index for sector and track which would be defined by the most recent `circos.trackPlotRegion` or `circos.updatePlotRegion`.

`circos.trackPoints` can draw points in the whole cells on a same track. However, it is the same if you use `circos.points` in a `for` loop.

4.8 Lines

Parameters for adding lines by `circos.lines` are similar to `lines` function, as illustrated in figure 10. One additional feature is that the areas under/above lines can be specified by `area` argument which can help to identify the direction of y-axes. Also the base line for the area can be set by `baseline`. `baseline` can be pre-defined string of `bottom` or `top`, or numeric values. `baseline` is also workable when `lty` is set to `h`.

Straight lines will be transformed to curves when mapping to circular layout (figure 11). Normally, curves can be approximated by a series of segmentations of straight lines. With more segmentations, there would be better approximations, but with larger size if you generate the graph file as pdf format, especially for huge genomic data. Thus, in this package, the number of the segmentation can be controlled by `circos.par("unit.circle.segments")`. The length of minimal segment is the length of the unit circle divided by `circos.par("unit.circle.segments")`. If you do not want such curve-transformations, you can set `straight` argument to `TRUE` (e.g. if it is a radical line.).

Possible usage for `circos.lines` is:

```
> circos.lines(x, y)
> circos.lines(x, y, sector.index, track.index)
> circos.lines(x, y, col, lwd, lty, type, straight)
> circos.lines(x, y, col, area, baseline, border)
```

Similar as `circos.points`, if no `sector.index` or `track.index` is specified, 'current' index would be used. Also, there is a `circos.trackLines` which is identical to `circos.lines` in a `for` loop.

4.9 Text

Only the facing of text by `circos.text` should be noted, as illustrated in figure 12. `srt` in `text` has been degenerated as `facing` in `circos.text` which support only six types of rotation pre-defined in `c("inside", "outside", "reverse.clockwise", "clockwise", "downward", "bending")`. But `adj` argument is still applicable in `circos.text`. Possible usage for `circos.text` is:

```
> circos.text(x, y, labels)
> circos.text(x, y, labels, sector.index, track.index)
> circos.text(x, y, labels, facing, adj, cex, col, font)
```

In some case, we may want to set the text facing more human-easy. For example, we want the text facing clockwise in right half of the circle while reverse-closewise in the left half of the circle. This can be easily done by setting `niceFacing` to `TRUE`. This option only works for `facing` value of `inside`, `outside`, `clockwise` and `reverse.clockwise`. When `niceFacing` is on, values for internal `facing` and `adj` will be re-defined according to the position of the texts in the circle. Please refer to figure 13 for examples.

There is also a `circos.trackText` in the package.

4.10 Rectangle

If you imagine the plotting region in a cell as Cartesian coordinate, then it draws rectangles. In the circle, the up and bottom edge become two arcs. Usage is similar as `rect`, but it can only draw one rectangle at a time.

```
> circos.rect(xleft, ybottom, xright, ytop)
> circos.rect(xleft, ybottom, xright, ytop, sector.index, track.index)
> circos.rect(xleft, ybottom, xright, ytop, col, border, lty, lwd)
```

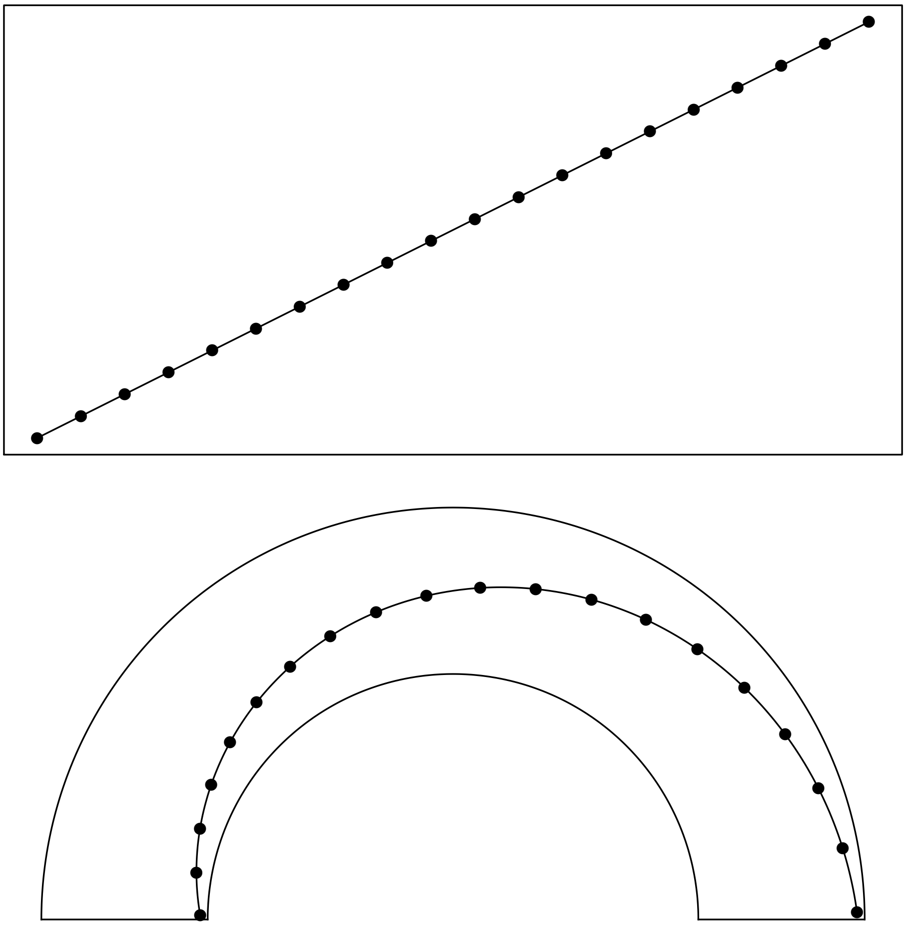

Figure 11: Straight lines will be transformed into curves in the circle

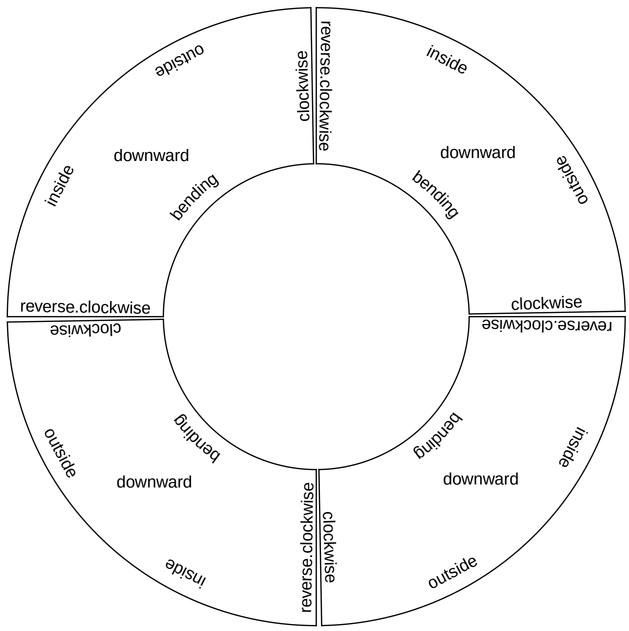

Figure 12: Text facing

4.11 Polygon

Similar as `circos.rect` and `polygon`, it draws a polygon through a series of points in a cell:

```
> circos.polygon(x, y)
> circos.polygon(x, y, sector.index, track.index)
> circos.polygon(x, y, col, border, lty, lwd)
```

In figure 14, the area of standard deviation of the smoothed line is drawn by `circos.polygon`. (Source code is in the examples section of `circos.polygon` help page.)

4.12 Axis

Because there may be no space to put y-axis, only adding x-axis for each cell is supported by `circos.axis`, as illustrated in figure 15. A lot of styles for axis can be set such as the position and length of major ticks, the number of minor ticks, the position and direction of the axis labels and the position of the x-axis. Note the adjustment of label strings is defined internally according to different label directions to ensure the start/end position of the string is located near the major tick.

In figure 15, axis styles in different sectors are :

- a: Major ticks are calculated automatically, other settings are default.
- b: Ticks are pointing to inside of the circle, facing of tick labels is set to `outside`.
- c: Position of x-axis is `bottom` of the cell.
- d: Ticks are pointing to inside of the circle, facing of tick labels is set to `reverse.clockwise`.
- e: Self-defined major ticks.
- f: Self-defined major ticks and tick labels, no minor ticks.
- g: No ticks for both major and minor ones, facing of tick labels is set to `reverse.clockwise`.

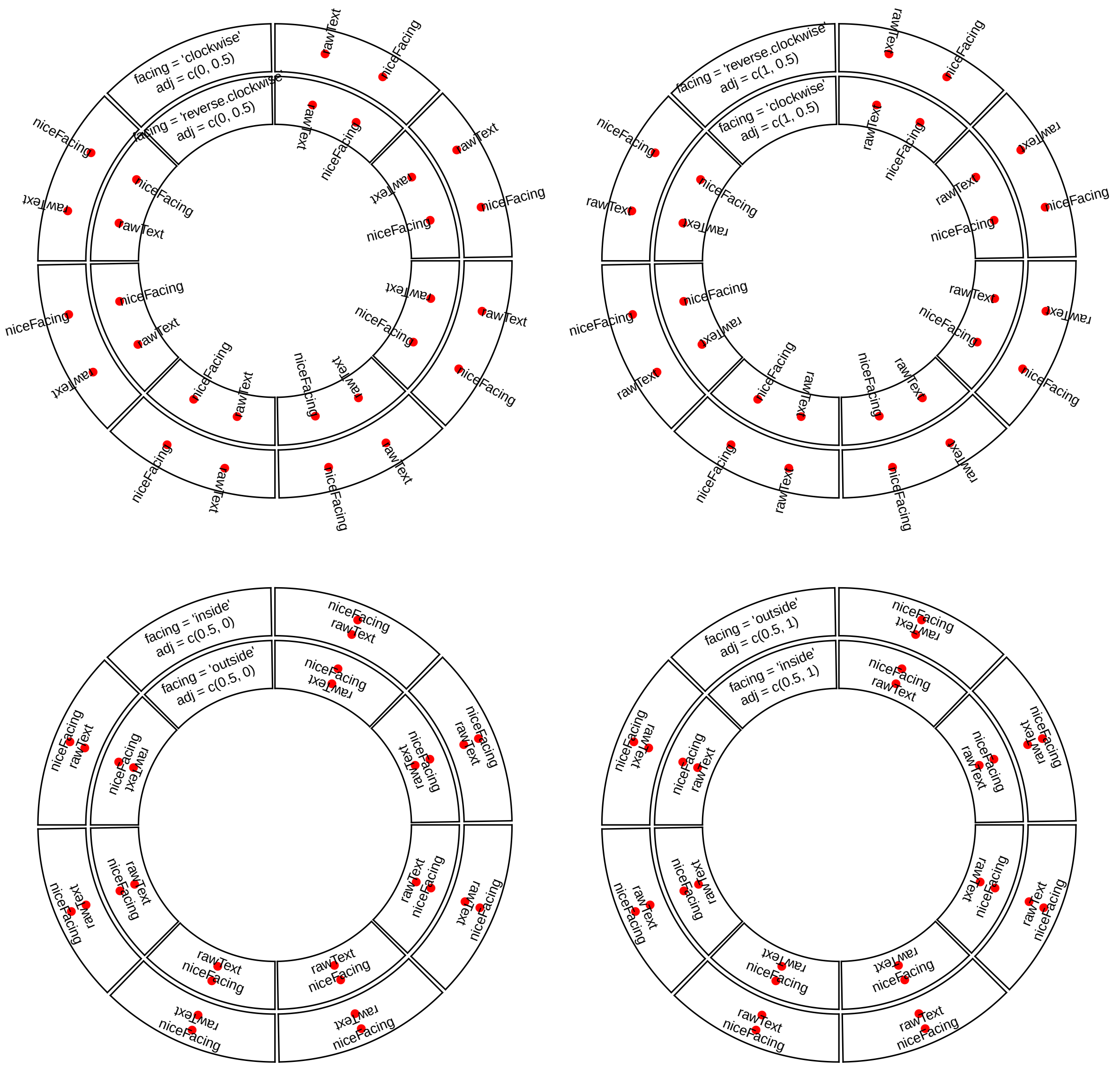

Figure 13: Human easy text facing. When `niceFacing` is on, settings in upper two figures are actually identical. Same as the settings in two bottom figures. Red dots represent positions of the texts.

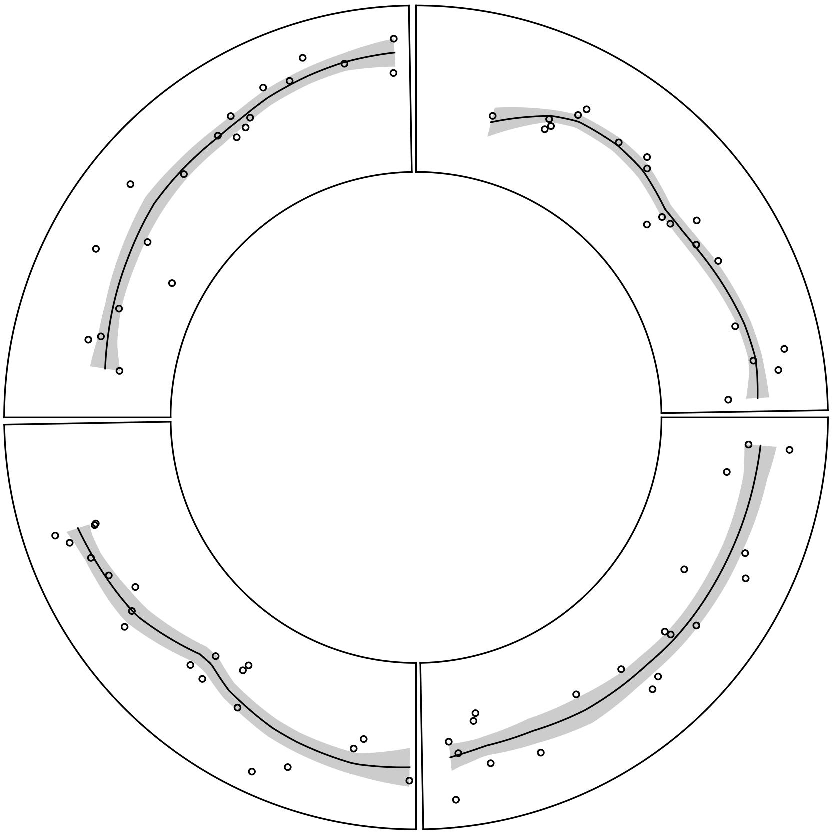

Figure 14: Area of standard deviation of the smoothed line

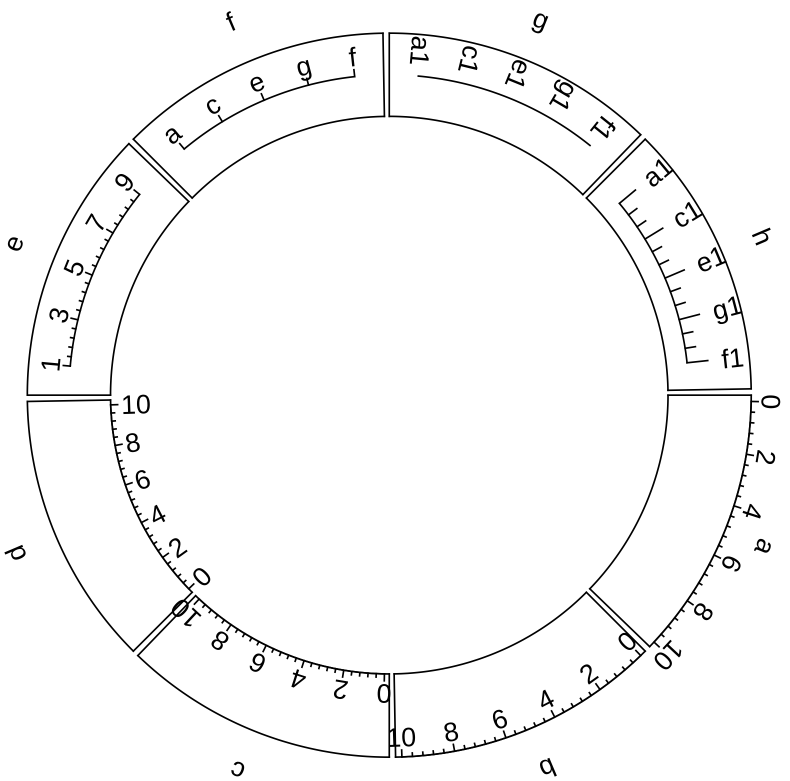

Figure 15: Axis settings

- h: Number of minor ticks between two major ticks is set to 2. Length of ticks is longer and axis labels are more away from ticks. Facing of tick labels is set to `clockwise`.

The facing of labels text can also be optimized by `labels.niceFacing` (by default it is `TRUE`).

For `circos.axis`, possible usage is as follows. `h` can be pre-defined string of `bottom` or `top`, or numeric values.

```
> circos.axis(h)
> circos.axis(h, sector.index, track.index)
> circos.axis(h, major.at, labels, major.tick)
> circos.axis(h, major.at, labels, major.tick, labels.font, labels.cex,
+             labels.facing, labels.away.percentage)
> circos.axis(h, major.at, labels, major.tick, minor.ticks,
+             major.tick.percentage, lwd)
```

If you really want the y-axes, you can implement one by yourself. It is just a combination of lines and text by using `circos.lines` and `circos.text`

4.13 Links

`circos.link` draws links from points and intervals (figure 16, top left). If both ends are points, then the link is represented as a line. If one of the ends is an interval, the link would be a belt/ribbon. The link is in fact a quadratic Bezier curve. Links do not hold any position as tracks, so they can be overlapping with tracks.

Possible usage for `circos.link` is:

```
> circos.link(sector.index1, 0, sector.index2, 0)
> circos.link(sector.index1, c(0, 1), sector.index2, 0)
> circos.link(sector.index1, c(0, 1), sector.index2, c(1, 2))
> circos.link(sector.index1, c(0, 1), sector.index2, 0, col, lwd, lty, border)
```

The position of the 'root' of the link is controlled by `rou`. By default, it is the end position of the most recently created track. So normally, you don't need to care about this setting. If you want to dive deeper, the default value of `rou` is defined as:

```
> tracks = get.all.track.index()
> n = length(tracks)
> get.cell.meta.data("cell.bottom.radius", track.index = tracks[n]) -
+     get.cell.meta.data("track.margin", track.index = tracks[n])[1] -
+     circos.par("track.margin")[2]
```

By default, the two roots of the link are located in a same circle. The positions of two roots can be assigned with different values by `rou1` and `rou2` (figure 16, top right).

```
> circos.link(sector.index1, 0, sector.index2, 0, rou)
> circos.link(sector.index1, 0, sector.index2, 0, rou1, rou2)
```

The height of the link can be controlled by `h` argument in `circos.link`.

When the link represents as a belt/ribbon (i.e. link from point to interval or from interval to interval), It can not ensure that one border is always below or above the other. Which means, in some cases, the two borders would intersect and it would make the link so ugly. It happens especially when position of the two ends are too close or the width of one end is extremely large while the width of the other end is too small. In that case, users can manually set height of the top and bottom border by `h` and `h2` (figure 16, bottom left).

```
> circos.link(sector.index1, 0, sector.index2, 0, h)
> circos.link(sector.index1, 0, sector.index2, 0, h, h2)
```

You can also control the shape of the link by `w` and `w2` (`w2` controls the shape of bottom border). Examples are in figure 16, bottom right. For more explanation of `w`, please refer to `http://en.wikipedia.org/wiki/B%C3%A9zier_curve#Rational_B.C3.A9zier_curves`.

```
> circos.link(sector.index1, 0, sector.index2, 0, w)
> circos.link(sector.index1, 0, sector.index2, 0, w, w2)
```

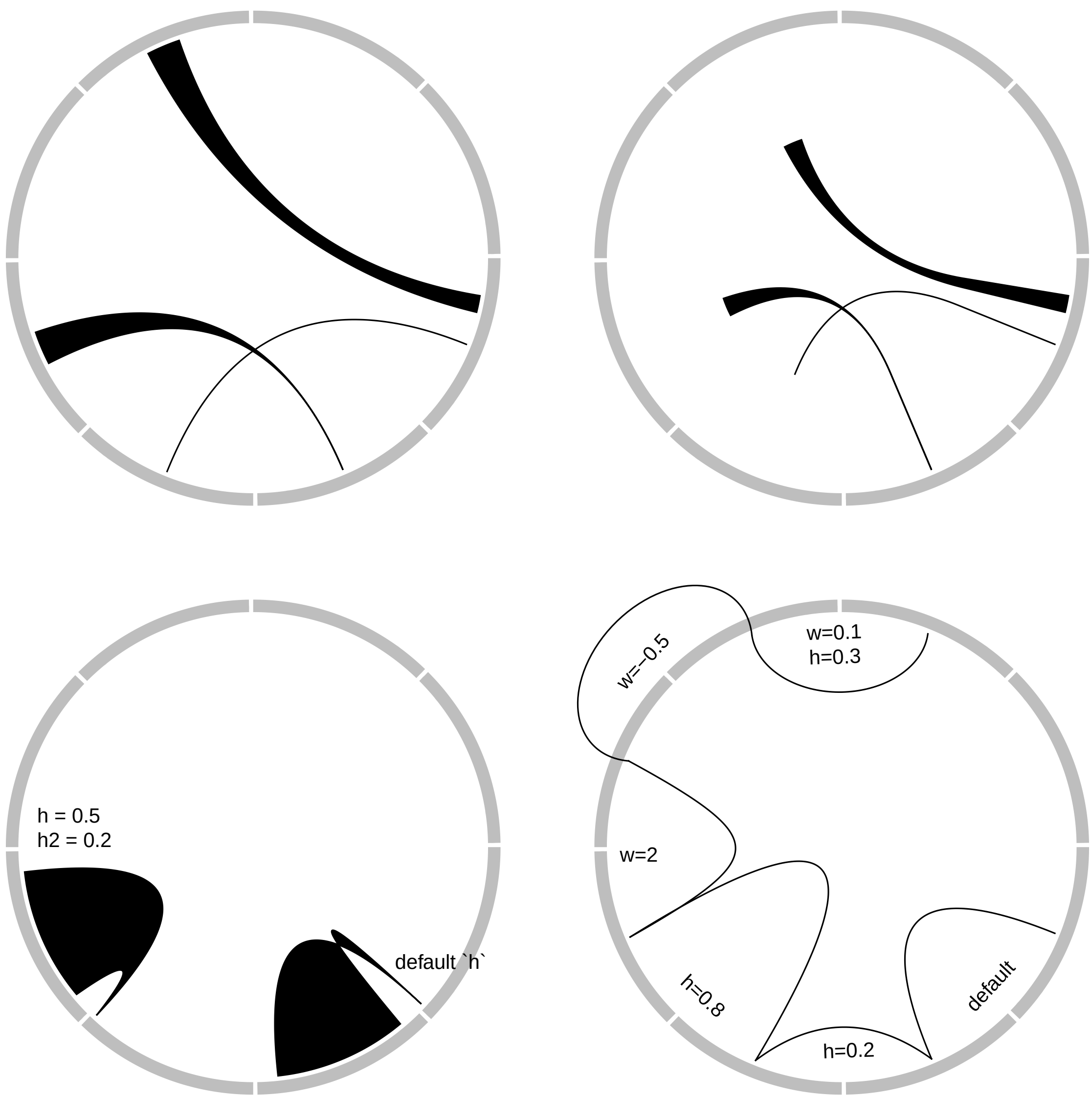

Figure 16: Drawing links. Top right: set different positions of roots; Bottom left: set different height of two borders. Bottom right: set different `h` and `w`

4.14 The `panel.fun` argument in `circos.trackPlotRegion`

`panel.fun` argument in `circos.trackPlotRegion` is useful to apply plotting as soon as the cell has been created. This self-defined function needs two arguments `x` and `y` which are data points that belong to this cell. The value for such values are automatically extracted from `x` and `y` in `circos.trackPlotRegion` function according to the category argument `factors`. In the following example, `x` in category `a` in `panel.fun` would be `1:3` and `y` values are `5:3`. If `x` or `y` in `circos.trackPlotRegion` is `NULL`, then `x` or `y` inside `panel.fun` is also `NULL`.

```
> factors = c("a", "a", "a", "b", "b")
> x = 1:5
> y = 5:1
> circos.trackPlotRegion(factors = factors, x = x, y = y,
+     panel.fun = function(x, y) {
+         circos.points(x, y)
+     })
```

In `panel.fun`, one thing important is that if you use any low-level graphic functions, you don't need to specify `sector.index` and `track.index` explicitly. Remember that when applying `circos.trackPlotRegion`, cells in the track are created one after one. When a cell is created, *circlize* would set the sector index and track index of the cell as the 'current' index for the sector and track. When the cell is created, `panel.fun` would be executed immediately. Without specifying `sector.index` and `track.index`, the 'current' one would be used and that's exactly what you need.

The advantage of `panel.fun` is that it makes you feel you are using graphical functions in traditional graphics engine (You can see it is the same of using `circos.points(x, y)` and `points(x, y)`). And you just pretend to draw in regular plotting regions. It will be much easier for users to understand and customize new graphics.

Inside `panel.fun`, more information of the 'current' cell can be obtained through `get.cell.meta.data`. Also this function takes the 'current' sector and 'current' track by default, Explanation of `get.cell.meta.data` can be found in following section.

4.15 High level plotting functions

With those low-level graphical functions such as `circos.points`, `circos.lines`, more high-level functions can be easily implemented. The package provides a high-level function `circos.trackHist` which draws histograms or the density distributions of data (figure 17). Users can learn how to implement high-level functions to support graphs such as barplot, heatmap, ... according to the source code of `circos.trackHist`. In `circos.trackHist`, it first calls `hist` or `density` to calculate the distribution, then creates a new track, finally uses `circos.rect` or `circos.lines` to draw histograms or density distributions.

In figure 17, the first track is histograms in which all the `ylim` are the same. The second track is histograms in which `force.ylim` is `FALSE`. The third and the fourth tracks are density distributions in which ylims are forced same or not.

In figure 18 you would see heatmap and cluster dendrograms in a circular layout. Heatmap is series of grids which can be drawn by `circos.rect`. Dendrograms are series of lines which can be drawn by `circos.lines`. However, x-values for heatmaps and dendrograms are not really x-values but just index for the grid/leaf (i.e., 1, 2, ... for grid/leaf 1, 2, ...), so it would be hard (or not proper) to make them as general functions for circos plotting. Thus we do not provide such `circos.heatmap` or `circos.dendrogram` in the package for public use. Anyway, we still wrote a not-full-functional `circos.dendrogram` which can be found at `http://jokergoo.github.io/circlize/example/genomic_heatmap.html`. If you want to draw heatmap or dendrogram by your own, this may be helpful for you, especially when you want to customize a complicated phylogenic tree.

4.16 Other functions

`draw.sector` can be used to draw sectors or part of a ring. This is useful if you want to highlight some part of your circos plot. As you can think, this function needs arguments of the position of circle center, the start degree and the end degree for sectors, and radius for two edges (or one edge) which may be the up or bottom border of a cell. These information can be obtained by `get.cell.meta.data`. E.g.

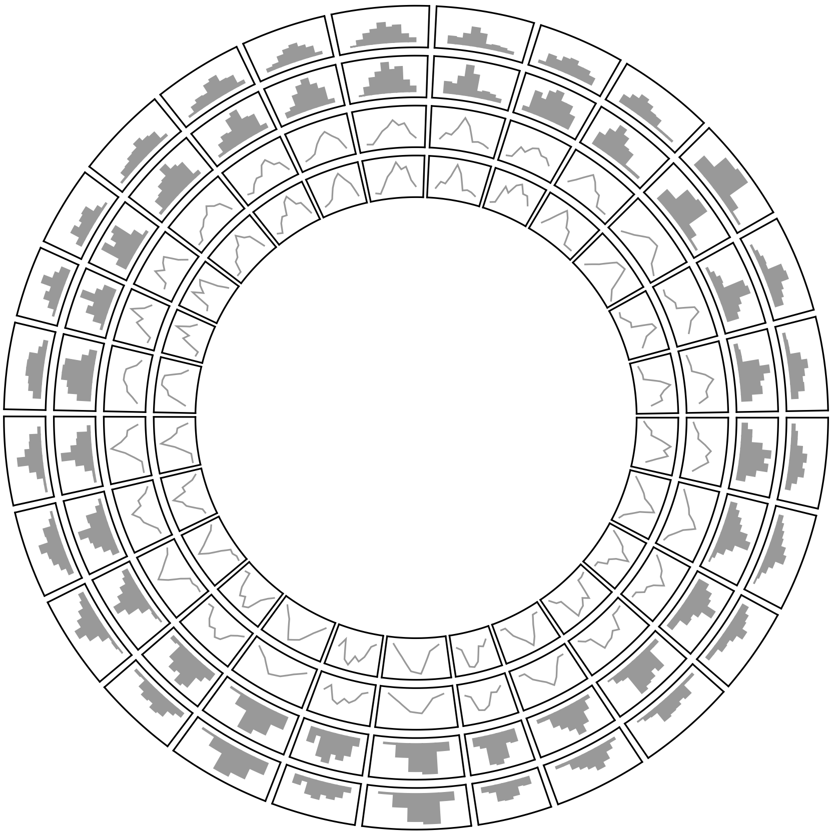

Figure 17: Histograms

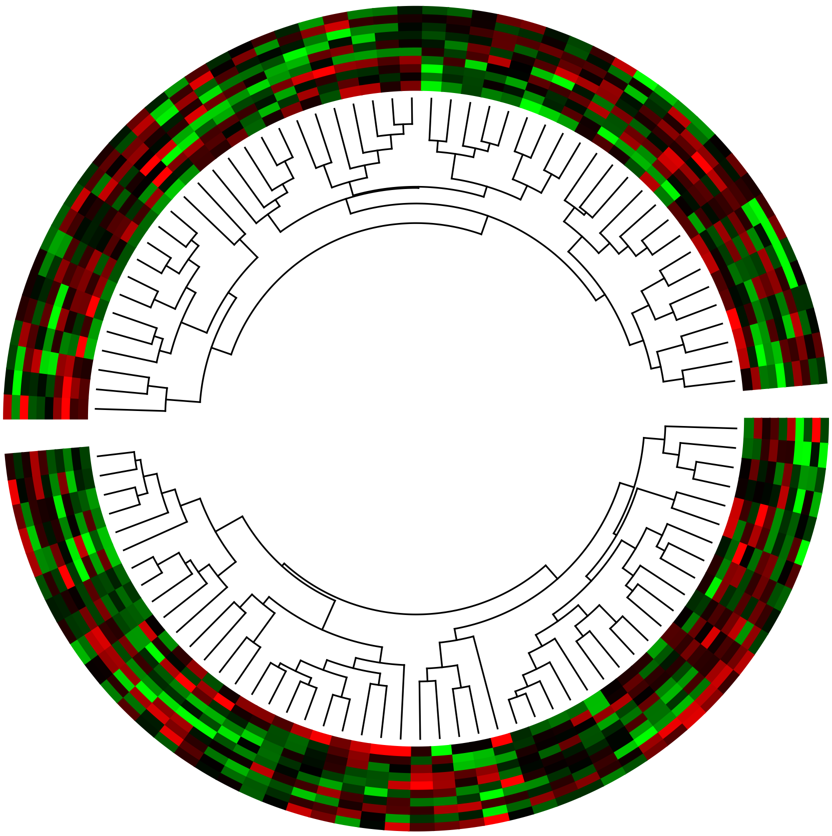

Figure 18: Heatmap with clustering

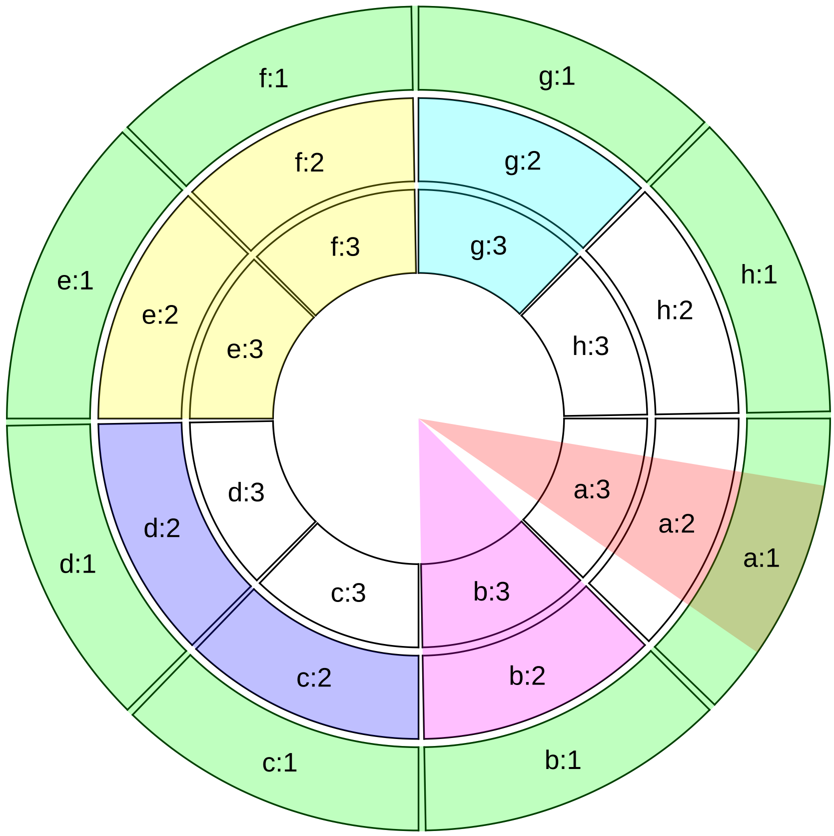

Figure 19: Hightlight sectors

the start degree and end degree can be obtained through `cell.start.degree` and `cell.end.degree`, and the position of the top border and bottom border on the circle radius can be obtained through `cell.top.radius` and `cell.bottom.radius`. An example is as follows and see figure 19 in which different colors correspond to different regions that need to be highlighted.

Degrees and radius for any points in a given cell can be obtained by the core function `circlize`, so you can highlight any region in the circular layout. Note `circlize` always returns a matrix with two columns in which the first column are degrees and the second column are radius for the points.

```
> circlize(x, y, sector.index, track.index)
> circlize(x, y)
```

Remember the color should be semi-transparent in the highlighted area. Usage for `draw.sector` is:

```
> draw.sector(center, start.degree, end.degree, rou1)
> draw.sector(center, start.degree, end.degree, rou1, rou2)
> draw.sector(center, start.degree, end.degree, rou1, rou2, col, border, lwd, lty)
```

`get.cell.meta.cell` can provide detailed information for a cell. It needs the index of sector and track as arguments. As usual, it would use 'current' index by default.

```
> get.cell.meta.data(name)
> get.cell.meta.data(name, sector.index, track.index)
```

Items that can be extracted by `get.cell.meta.data` are:

- `sector.index`: The name (label) for the sector

- `sector.numeric`.index: Numeric index for the sector. It is the numeric order of levels of `factors` in initialization step.
- `track.index`: Numeric index for the track
- `xlim`: Minimal and maximal values on the x-axis
- `ylim`: Minimal and maximal values on the y-axis
- `xcenter`: mean of `xlim`
- `ycenter`: mean of `ylim`
- `xrange`: Range of `xlim`
- `yrange`: Range of `ylim`
- `cell.xlim`: Minimal and maximal values on the x-axis extended by cell paddings
- `cell.ylim`: Minimal and maximal values on the y-axis extended by cell paddings
- `xplot`: Right and left border degree for the plotting region in the unit circle. The first element corresponds to the start point of values on x-axis (`cell.xlm[1]`) and the second element corresponds to the end point of values on x-axis (`cell.xlim[2]`) Since x-axis in data coordinate in cells are always clockwise, `xplot[1]` is larger than `xplot[2]`.
- `yplot`: Bottom and top radius value for borders of the plotting region. It is the value of radius of arc corresponding to inner border or outer border.
- `cell.start.degree`: Same as `xplot[1]`
- `cell.end.degree`: Same as `xplot[2]`
- `cell.bottom.radius`: Same as `yplot[1]`
- `cell.top.radius`: Same as `yplot[2]`
- `track.margin`: Margins for the cell
- `cell.padding`: Paddings for the cell

With information returned by `get.cell.meta.data`, you can customize the plotting region for each cell. The following two tracks have the same style. The advantages for the second track is that you can add other graphics freely such as reference lines.

```
> factors = letters[1:3]
> circos.initialize(factors = factors, xlim = c(0, 1))
> circos.trackPlotRegion(ylim = c(0, 1), bg.border = NA, bg.col = 1:3)
> circos.trackPlotRegion(ylim = c(0, 1), bg.border = NA, bg.col = NA,
+     panel.fun = function(x, y) {
+         cell.xlim = get.cell.meta.data("cell.xlim")
+         cell.ylim = get.cell.meta.data("cell.ylim")
+         i = get.cell.meta.data("sector.numeric.index")
+         circos.rect(cell.xlim[1], cell.ylim[1], cell.xlim[2], cell.ylim[2],
+             col = i, border = NA)
+
+                 seg = seq(cell.ylim[1], cell.ylim[2], length = 5)
+                 seg = seg[-1]
+                 seg = seg[-length(seg)]
+                 for(h in seg) {
+                         circos.lines(cell.xlim, c(h, h), lty = 2, col = "grey")
+                 }
+ })
```

4.17 Get information of your circos plot

You can get basic information of your current circos plot by `circos.info`. The function can be applied at any time.

```
> factors = letters[1:3]
> circos.initialize(factors = factors, xlim = c(1, 2))
> circos.info()

All your sectors:
[1] "a" "b" "c"

All your tracks:
integer(0)

Your current sector.index is
Your current track.index is 0

> circos.trackPlotRegion(ylim = c(0, 1))
> circos.info(sector.index = "a", track.index = 1)

sector index: a
track index: 1
xlim: [1, 2]
ylim: [0, 1]
cell.ylim: [-0.1, 1.1]
cell.ylim: [-0.1, 1.1]
xplot (degree): [360, 241]
yplot (radius): [0.79, 0.99]
track.margin: c(0.01, 0.01)
cell.padding: c(0.02, 1, 0.02, 1)

Your current sector.index is c
Your current track.index is 1
```

It can also add labels onto every cell by `circos.info(plot = TRUE)`.

4.18 Do not forget `circos.clear`

You should always call `circos.clear` to complete the circos plottings. Because there are several parameters for circos plot which can only be set before `circos.initialize`. So before you draw the next circos plot, you need to reset these parameters.

4.19 A simple example of implementing high level graphics

We will show you a simple example (figure 20) which combine several low-level graphical functions to construct a complicated graphic for specific purpose.

In the following code, we first draw reference lines both from x-direction and y-direction. Then draw two rectangles which covers region of `y > 1` and `y < -1`. Finally, add points to the region with different colors.

```
> library(circlize)
> factors = sample(letters[1:6], 100, replace = TRUE)
> x = rnorm(100)
> y = rnorm(100)
> par(mar = c(1, 1, 1, 1))
> circos.initialize(factors = factors, x = x)
> circos.trackPlotRegion(factors = factors, x = x, y = y, bg.col = "#EEEEEE",
+     bg.border = NA, track.height = 0.4, panel.fun = function(x, y) {
+
```

```
+     cell.xlim = get.cell.meta.data("cell.xlim")
+     cell.ylim = get.cell.meta.data("cell.ylim")
+
+     # reference lines
+     for(xi in seq(cell.xlim[1], cell.xlim[2], length.out = 10)) {
+         circos.lines(c(xi, xi), cell.ylim, lty = 2, col = "white")
+     }
+     for(yi in seq(cell.ylim[1], cell.ylim[2], length.out = 5)) {
+         circos.lines(cell.xlim, c(yi, yi), lty = 2, col = "white")
+     }
+
+         # region with different colors
+     xlim = get.cell.meta.data("xlim")
+     ylim = get.cell.meta.data("ylim")
+     circos.rect(xlim[1], 1, xlim[2], ylim[2], col = "#FF000020", border = NA)
+     circos.rect(xlim[1], ylim[1], xlim[2], -1, col = "#00FF0020", border = NA)
+
+         # points with different colors
+     circos.points(x[y >= 1], y[y >= 1], pch = 16, cex = 0.8, col = "red")
+     circos.points(x[y <= -1], y[y <= -1], pch = 16, cex = 0.8, col = "green")
+     circos.points(x[y > -1 & y < 1], y[y > -1 & y < 1], pch = 16, cex = 0.5)
+ })
> circos.clear()
```

5 Advanced technique

5.1 Draw part of the circos layout

`canvas.xlim` and `canvas.ylim` in `circos.par` is useful to draw only part of circle. In the example, only sectors between 0° to 90° are plotted (figure 21). First, four sectors with the same width are initialized. Then only the first sector is drawn with points and lines. From figure 21, we in fact created the whole circle, but only a quarter of the circle is in the canvas region. Codes are as follows.

```
> library(circlize)
> par(mar = c(1, 1, 1, 1))
> circos.par("canvas.xlim" = c(0, 1), "canvas.ylim" = c(0, 1),
+     "clock.wise" = FALSE, "gap.degree" = 0)
> factors = letters[1:4]
> circos.initialize(factors = factors, xlim = c(0, 1))
> circos.trackPlotRegion(factors = factors, ylim = c(0, 1), bg.border = NA)
> circos.updatePlotRegion(sector.index = "a", bg.border = "black")
> x1 = runif(100)
> y1 = runif(100)
> circos.points(x1, y1, pch = 16, cex = 0.5)
> circos.trackPlotRegion(factors = factors, ylim = c(0, 1), bg.border = NA)
> circos.updatePlotRegion(sector.index = "a", bg.border = "black")
> circos.lines(1:100/100, y1, pch = 16, cex = 0.5)
> circos.clear()
```

In the second situation, you don't need some sectors or cells but still you need to draw the whole circle. Remember when you are creating new track with `circos.trackPlotRegion` and set `bg.col` and `bg.border` to `NA`, it means create the new track and draw nothing. After that, you can use `circos.updatePlotRegion` to update these invisible cells of interest to add graphics on it (figure 22).

```
> library(circlize)
> par(mar = c(1, 1, 1, 1))
> factors = letters[1:4]
> circos.initialize(factors = factors, xlim = c(0, 1))
```

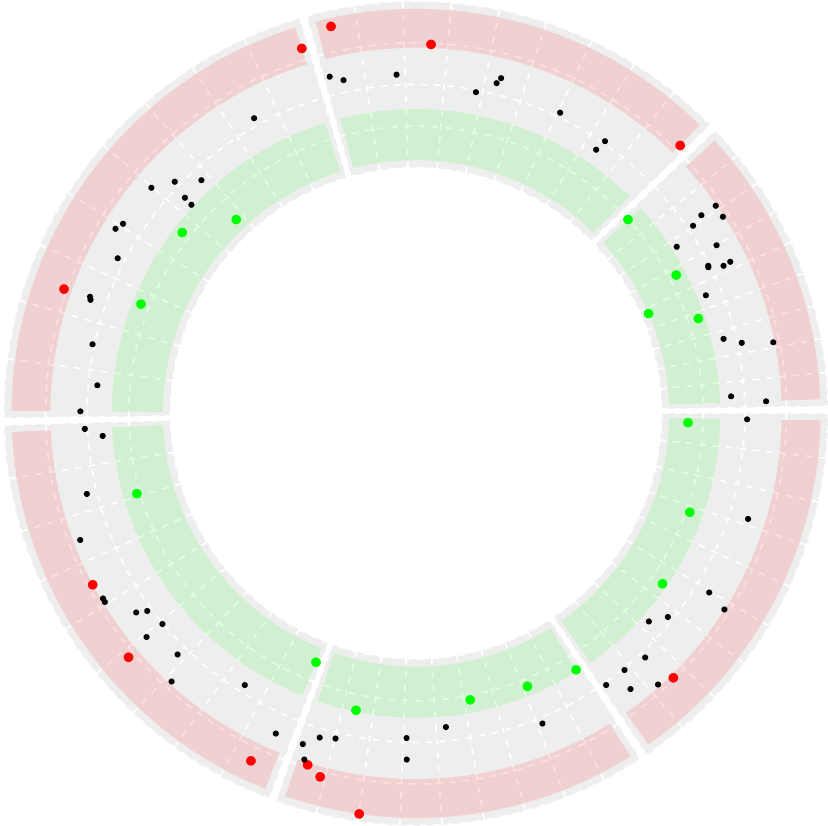

Figure 20: Combine low-level graphical functions to generate high-level graphic

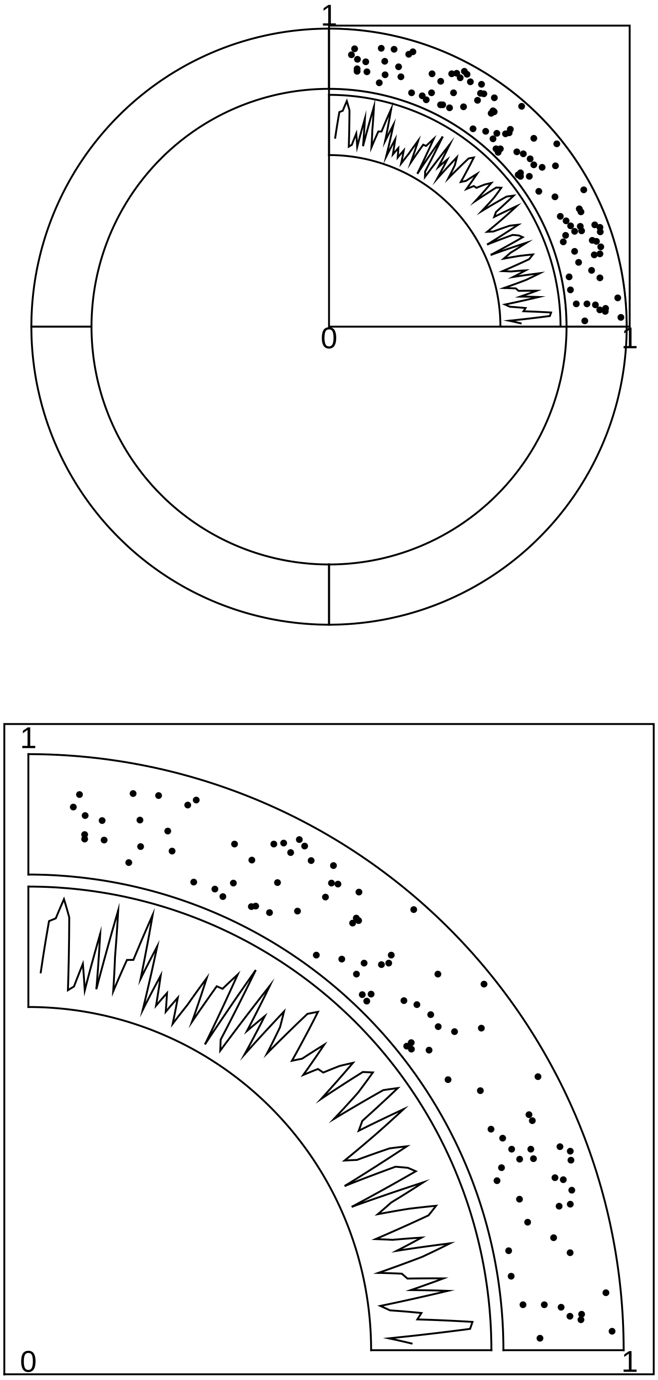

Figure 21: Part of the circle

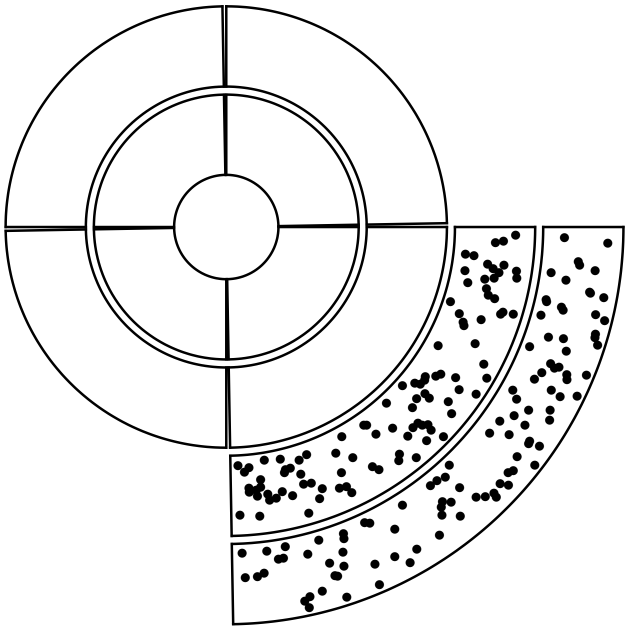

Figure 22: Part of the circular layout, situation 2.

```
> circos.trackPlotRegion(factors = factors, ylim = c(0, 1), bg.col = NA, bg.border = NA)
> circos.updatePlotRegion(sector.index = "a", bg.border = "black")
> x1 = runif(100)
> y1 = runif(100)
> circos.points(x1, y1, pch = 16, cex = 0.5)
> circos.trackPlotRegion(factors = factors, ylim = c(0, 1),bg.col = NA, bg.border = NA)
> circos.updatePlotRegion(sector.index = "a", bg.border = "black")
> x1 = runif(100)
> y1 = runif(100)
> circos.points(x1, y1, pch = 16, cex = 0.5)
> circos.trackPlotRegion(factors = factors, ylim = c(0, 1))
> circos.trackPlotRegion(factors = factors, ylim = c(0, 1))
> circos.clear()
```

5.2 Combine several parts of circular layouts

Since circular layout by *circlize* is finally plotted in an ordinary R plotting system. Two seperated circular layouts can be plotted together by some tricks. Here the secret is `par(new = TRUE)` which allows to draw a new figure on the previous canvas region. **Just remember the radius of the circos is always 1.**

The first example is to draw one outer circos plot and an inner circos plot (figure 23).

```
> library(circlize)
> par(mar = c(1, 1, 1, 1))
```

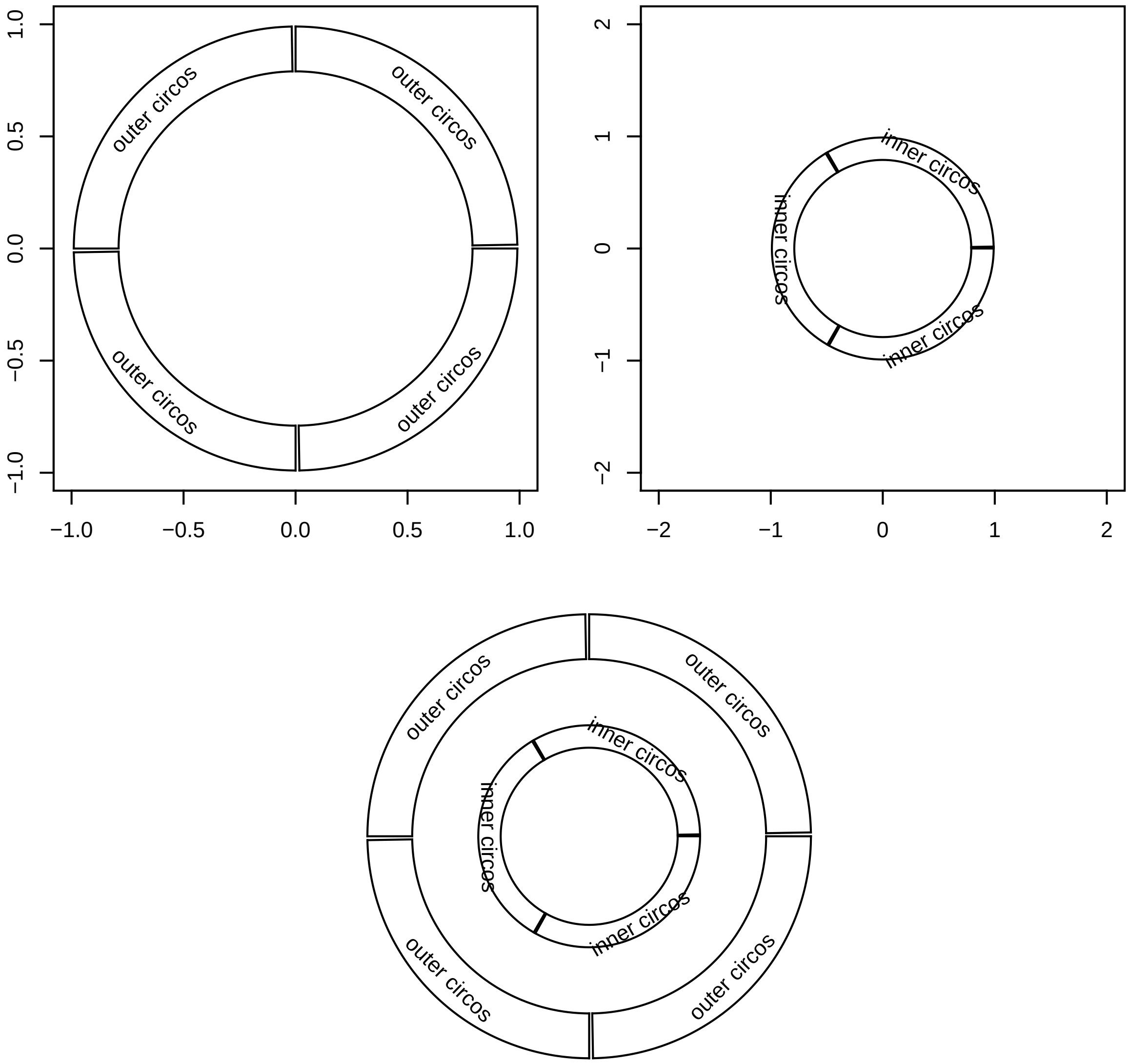

Figure 23: An outer and an inner circular layout

```
> factors = letters[1:4]
> circos.initialize(factors = factors, xlim = c(0, 1))
> circos.trackPlotRegion(ylim = c(0, 1), panel.fun = function(x, y) {
+     circos.text(0.5, 0.5, "outer circos")
+ })
> circos.clear()
> par(new = TRUE)
> circos.par("canvas.xlim" = c(-2, 2), "canvas.ylim" = c(-2, 2))
> factors = letters[1:3]
> circos.initialize(factors = factors, xlim = c(0, 1))
> circos.trackPlotRegion(ylim = c(0, 1), panel.fun = function(x, y) {
+     circos.text(0.5, 0.5, "inner circos")
+ })
> circos.clear()
```

The second example is drawing two seperated circular layouts in which every circos plot only contains a half (figure 24).

```
> library(circlize)
> par(mar = c(1, 1, 1, 1))
> factors = letters[1:4]
> circos.par("canvas.xlim" = c(-1, 1.5), "canvas.ylim" = c(-1, 1.5), start.degree = -45)
> circos.initialize(factors = factors, xlim = c(0, 1))
> circos.trackPlotRegion(ylim = c(0, 1), bg.col = NA, bg.border = NA)
> circos.updatePlotRegion(sector.index = "a")
> circos.text(0.5, 0.5, "first one")
> circos.updatePlotRegion(sector.index = "b")
> circos.text(0.5, 0.5, "first one")
> circos.clear()
> par(new = TRUE)
> circos.par("canvas.xlim" = c(-1.5, 1), "canvas.ylim" = c(-1.5, 1), start.degree = -45)
> circos.initialize(factors = factors, xlim = c(0, 1))
> circos.trackPlotRegion(ylim = c(0, 1), bg.col = NA, bg.border = NA)
> circos.updatePlotRegion(sector.index = "d")
> circos.text(0.5, 0.5, "second one")
> circos.updatePlotRegion(sector.index = "c")
> circos.text(0.5, 0.5, "second one")
> circos.clear()
```

The third example is to draw sectors with different radius (figure 25). In fact, it draws four circos graphs in which only one sector of each graphs is plotted. Note links can not be drawn in these different sectors because links can only be drawn in one circos plot.

```
> library(circlize)
> par(mar = c(1, 1, 1, 1))
> factors = letters[1:4]
> lim = c(1, 1.1, 1.2, 1.3)
> for(i in 1:4) {
+     circos.par("canvas.xlim" = c(-lim[i], lim[i]),
+         "canvas.ylim" = c(-lim[i], lim[i]), "default.track.height" = 0.4)
+     circos.initialize(factors = factors, xlim = c(0, 1))
+     circos.trackPlotRegion(ylim = c(0, 1), bg.border = NA)
+     circos.updatePlotRegion(sector.index = factors[i], bg.border = "black")
+     circos.points(runif(10), runif(10), pch = 16)
+     circos.clear()
+     par(new = TRUE)
+ }
> par(new = FALSE)
```

It is different from example in "Draw part of the circular layout" section. In that example, cells both visible and invisible all belong to a same track and they are in a same circos plot, so they should have same radius. But here, cells have different radius to the center of the circle and they belong to different circos plot (although only part of each circos plot is visible).

5.3 Draw outside and combine with canvas coordinate

Sometimes it is very useful to draw something outside the plotting region of cell. (You can think it is similar as `par(xpd = NA)` setting.) The following is a simple example to illustrate such circumstance (figure 26). The text is drawn outside the cell.

```
> library(circlize)
> set.seed(12345)
> par(mar = c(1, 1, 1, 1))
> factors = letters[1:4]
> circos.par("canvas.xlim" = c(-1.5, 1.5), "canvas.ylim" = c(-1.5, 1.5), "gap.degree" = 10)
> circos.initialize(factors = factors, xlim = c(0, 1))
```

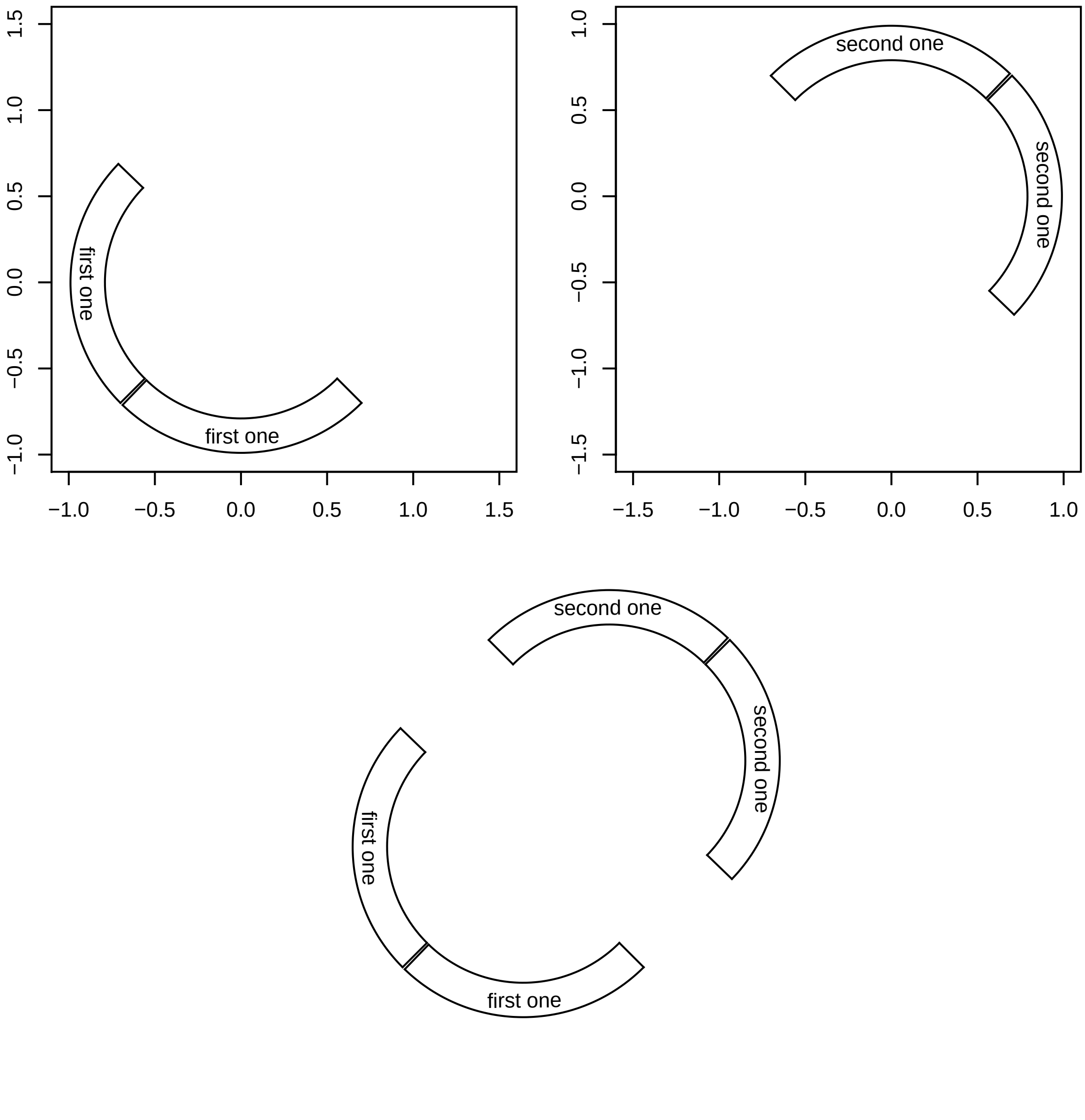

Figure 24: Two seperated circos layouts

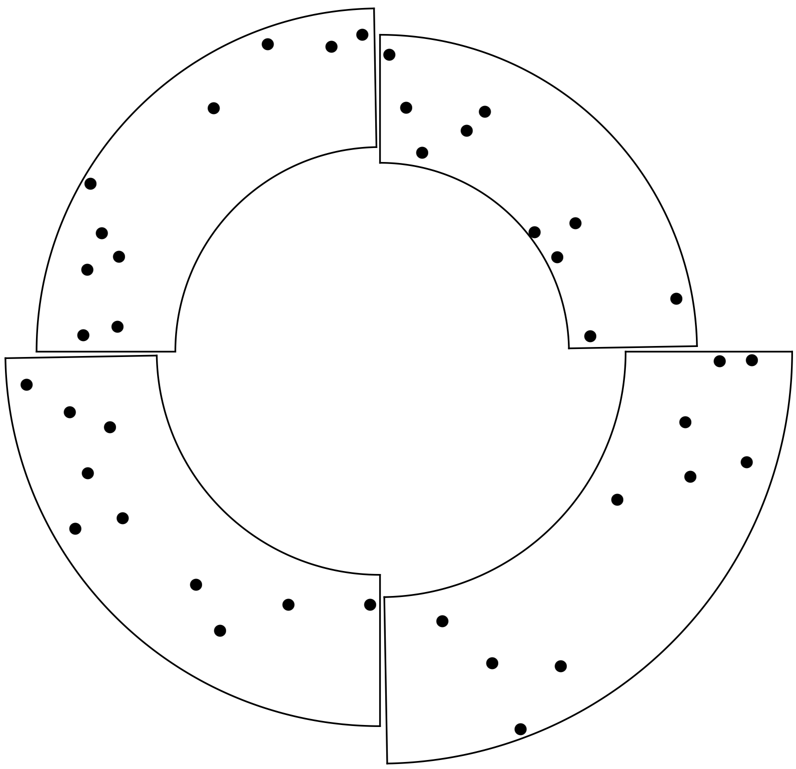

Figure 25: Sectors with different radius

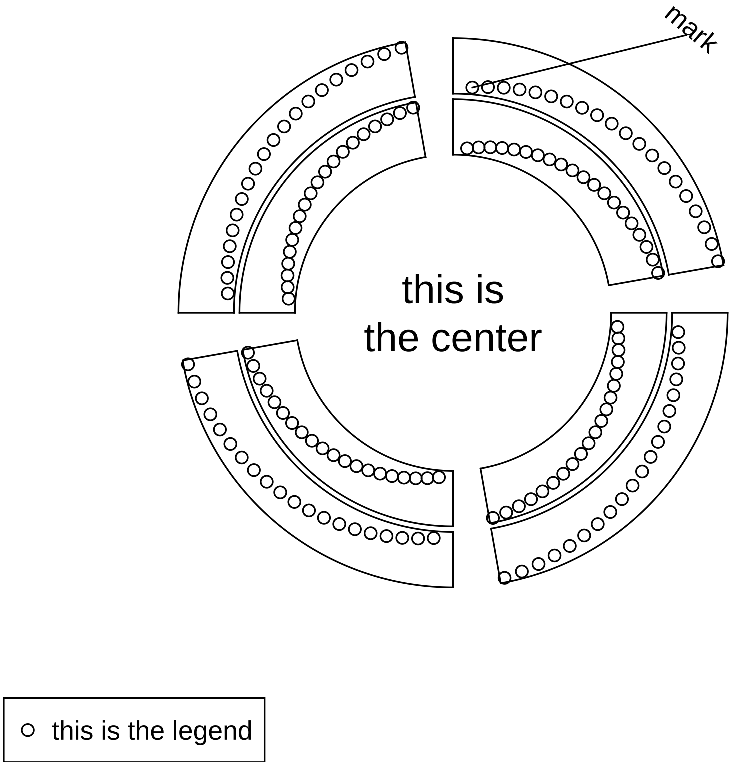

Figure 26: Draw outside the cell and combine with canvas coordinate

```
> circos.trackPlotRegion(ylim = c(0, 1), panel.fun = function(x, y) {
+     circos.points(1:20/20, 1:20/20)
+ })
> circos.lines(c(1/20, 0.5), c(1/20, 3), sector.index = "d", straight = TRUE)
> circos.text(0.5, 3, "mark", sector.index = "d", adj = c(0.5, 0))
> circos.trackPlotRegion(ylim = c(0, 1), panel.fun = function(x, y) {
+     circos.points(1:20/20, 1:20/20)
+ })
> text(0, 0, "this is\nthe center", cex = 1.5)
> legend("bottomleft", pch = 1, legend = "this is the legend")
> circos.clear()
```

Since the final graphic is drawn in an ordinary canvas plotting region, we can add additional graphics through the traditional way. It is useful to add some legends or marks on the figure. You can see how `text` and `legend` work in the example code.

5.4 Draw figures with `layout`

You can use `layout` to arrange multiple figures together (also it is available from `par(mfrow)` or `par(mfcol)`) (figure 27).

```
> library(circlize)
> set.seed(12345)
> rand_color = function() {
```

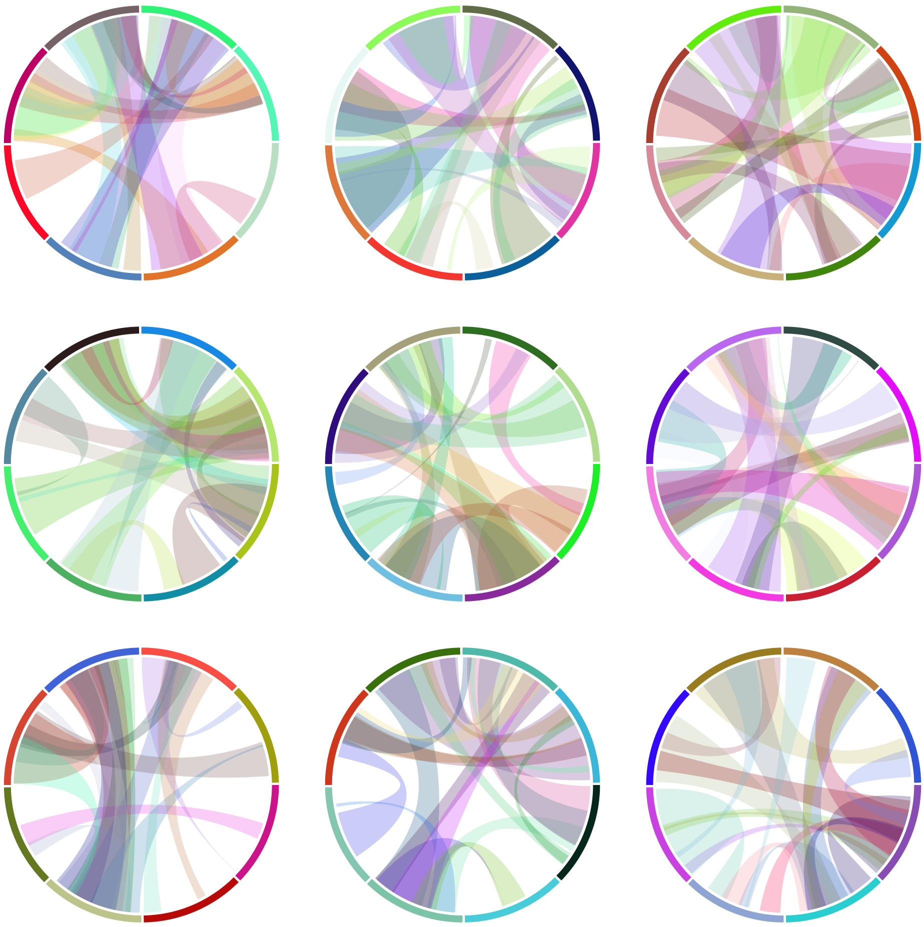

Figure 27: Draw multiple figures by `layout`

```
+     return(rgb(runif(1), runif(1), runif(1)))
+ }
> layout(matrix(1:9, 3, 3))
> for(i in 1:9) {
+     factors = 1:8
+     par(mar = c(0.5, 0.5, 0.5, 0.5))
+     circos.par(cell.padding = c(0, 0, 0, 0))
+     circos.initialize(factors, xlim = c(0, 1))
+     circos.trackPlotRegion(ylim = c(0, 1), track.height = 0.05,
+         bg.col = sapply(1:8, function(x) rand_color()), bg.border = NA)
+     for(i in 1:20) {
+         se = sample(1:8, 2)
+         col = rand_color()
+         col = paste(col, "40", sep = "")
+         circos.link(se[1], runif(2), se[2], runif(2), col = col)
+     }
+     circos.clear()
+ }
```