

# An introduction of the `circlize` package

Zuguang Gu <z.gu@dkfz.de>  
German Cancer Research Center,  
Heidelberg, Germany

May 26, 2013

## 1 Introduction

Circos layout (<http://circos.ca>) is very useful to represent complicated informations, especially for genomic data. The `circlize` package implements the circos layout in R. The advantage is that R is nature-born to draw statistical graphs, thus, types of plottings are not restricted by the package but by user's inspiration. The `circlize` package allocates and illustrates data which is from a certain category into a cell inside a circle and makes you felling that you are plotting figures in a normal plotting coordinate.

Since most of the figures are composed of simple graphs, such as points, lines, polygon (for filled color) *et al*, so we just need to implement those low level functions to draw figures in circos layout.

Currently there are following functions that can be used for plotting, they are similar to the functions without `circos.` prefix from the traditional graph engine:

- `circos.points`: draw points in a cell
- `circos.lines`: draw lines in a cell
- `circos.rect`: draw rectangle in a cell
- `circos.polygon`: draw polygon in a cell
- `circos.text`: draw text in a cell
- `circos.axis`: draw axis in a cell
- `circos.link`: this maybe the unique feature for circos layout to represent relationships between elements.

For drawing points, lines and text 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.

Also, the function drawing histograms in the whole track is available:

- `circos.trackHist`

Functions to arrange the circos layout:

- `circos.trackPlotRegion`: create plotting regions of cells in one track
- `circos.updatPlotRegion`: update one specified cell
- `circos.par`: circos parameters
- `circos.clear`: reset circos parameters and internal variables

Theoretically, you are able to draw most kinds of circos figures by the above functions. As you can see, all figures in the three 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 draw a circos layout. Then it tells you the basic principle (or the order of using the circos functions) for drawing. After that a detailed explanation of circos parameters, coordinates and low level functions. Finally it would tell some tricks for drawing more complicated circos plot.

## 2 A quick glance

Following is an example. First generate some data. There needs to have a factor to represent categories, values on x-axis, and values on y-axis.

```
> set.seed(12345)
> n = 10000
> a = data.frame(factor = sample(letters[1:8], n, replace = TRUE),
+   x = rnorm(n), y = runif(n))
```

Initialize the layout. In this step, the function allocates sectors along 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. The order of the sectors in the circle is the order of levels of **factors**.

```

> 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. 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. X-lims for cells in the track have been already defined by the initialization step, so here we only need to specify the y-lims for each cell, either by `y` or `ylim` argument.

We also draw axis for each cell in the first track, The axis for each cell is drawn by `panel.fun` argument. `circos.trackPlotRegion` creates plotting region cell by cell and the `panel.fun` is actually executed after the creation of the plotting region for the cell. So `panel.fun` actually means drawing graphs in the “current cell”. After that, draw points through the whole track by `circos.trackPoints`. Finally, add two texts in a certain cell (the cell is specified by `sector.index` argument).

Here what should be note is that the first track has a index number of 1. Then 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 assigned internally. So if `track.index` is not specified, it means the track index is the current track index (it would be explained in the following sections).

```

> circos.trackPlotRegion(factors = a$factor, y = a$y,
+   panel.fun = function(x, y) {
+     circos.axis()
+   })
> circos.trackPoints(a$factor, a$x, a$y, col = col,
+   pch = 16, cex = 0.5)
> circos.text(-1, 0.5, "left", sector.index = "a")
> circos.text(1, 0.5, "right", sector.index = "a")

```

Draw the second track. It is histograms among the track. The `circos.trackHist` can also create a new track because drawing histogram is really high level. So the track index for this track is 2.

```

> circos.trackHist(a$factor, a$x, bg.col = bgcol, col = NA)

```

Draw the third track. Different background colors for cells can be assigned. So it may highlight some feature of the `circlize` package. 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`. The function erases graphs which have been drawn. Here we erase graphs in one cell in track 2, sector d and re-draw some points.

```
> circos.updatePlotRegion(sector.index = "d", track.index = 2)
> circos.points(x = runif(100), y = runif(100))
```

Draw the fourth track. Here you can choose different line types.

```
> 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. 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, top.ratio = 0.9)
> circos.link("c", c(-0.5, 0.5), "d", c(-0.5,0.5), col = "red",
+   border = "blue", top.ratio = 0.2)
> circos.link("e", 0, "g", c(-1,1), col = "green", lwd = 2, lty = 2)
```

The final figure looks like figure 1.

## 3 Details

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

### 3.1 Rules to draw the circos layout

The rules for drawing the circos layout is rather simple. It follows the sequence of “initialize - create track - draw graphs - create track - draw graphs - ... - clear”.

1. Initialize the layout using `circos.initialize`. Since circos layout in fact visualizes data which is in categories, there should be a factor and a x-range to allocate categories into sectors.
2. Create plotting regions for the new track and apply plottings. The new track is created just inside the previous created one and the index of the track is added by 1 automatically. Only after the creation of the track can you add other graphs on it. There are three ways to do the plotting job.

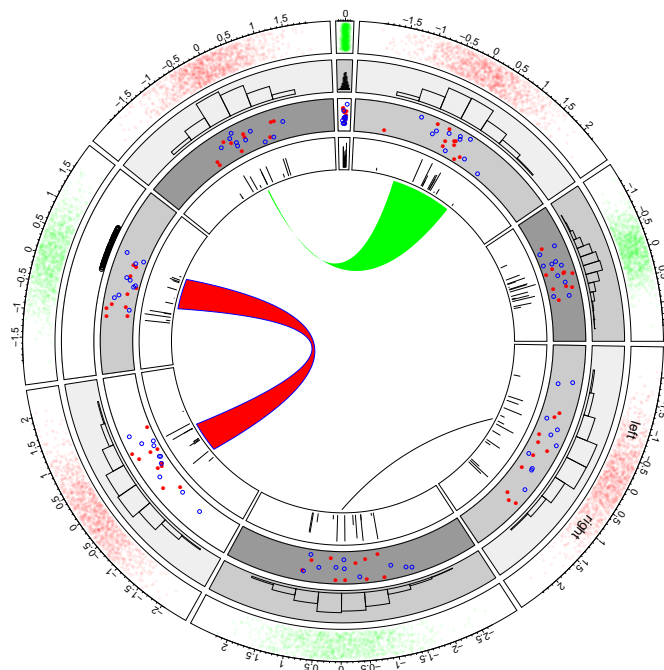


Figure 1: An example for circos layout

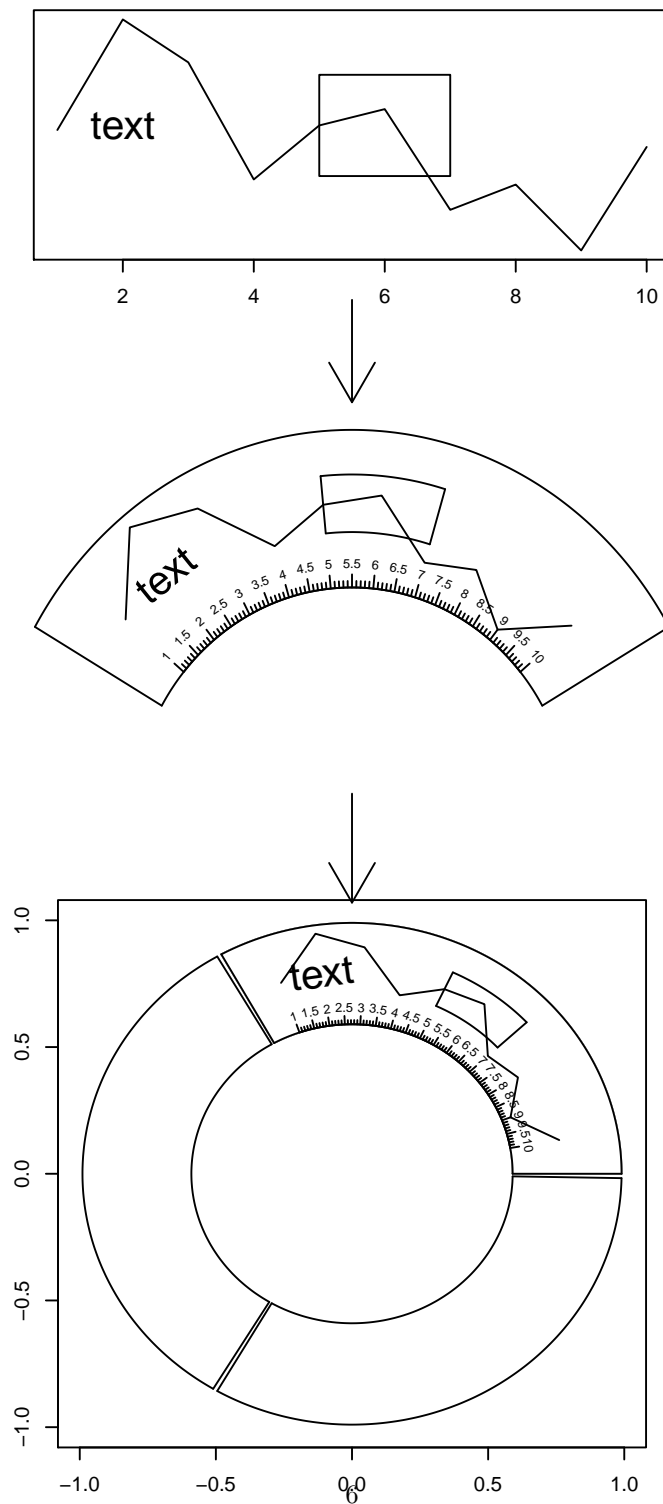


Figure 2: Transformation between different coordinates

- (a) After the creation of the track. use low level function like `circos.points`, `circos.lines`, ... to draw graphs cell by cell. It allways involves a `for` loop.
- (b) Use `circis.trackPoints`, `circos.trackLines`, ... to draw same style of graphs through all cells simultaneously. However, it is not recommended because it would make you a little confused.
- (c) Use `panel.fun` argument in `circos.trackPlotRegion` to draw graphs immediately after the creation of 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 internally.

Plotting regions for cells that have been created can be updated by `circos.updatePlotRegion`. `circos.updatePlotRegion` will erase every that you have already plotted in the plotting region of the cell.

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

### 3. Call `circos.clear` to do cleanings.

Codes for the `circos` layout drawing rule would looks like (pseudo code):

```
> circos.initialize(factors, xlim)
> circos.trackPlotRegion(factors, ylim)
> for(sector.index in all.sector.index) {
+   circos.points(x, y, sector.index)
+   circos.lines(x, y, sector.index)
+ }
```

or like following:

```
> circos.trackPlotRegion(factors, ylim)
> circos.trackPoints(factors, x, y)
> circos.trackLines(factors, x, y)
```

or like following. This the most natural way I feel.

```
> circos.trackPlotRegion(factors, x, 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, the values for sector index and the track index, by default, are taken as the current calculated ones. 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 is just drawn in the most nearly created cell.

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

## 3.2 Coordinate transformation

There is a user coordinate in which the range for x-axis and y-axis is the range of data, a polar coordinate to allocates different cells and a the canvas coordinate which really draw the figures (figure 2). The package would first transform the user 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 defaulte.

**It should be noted that the circos layout is always 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 (user coordinate) in which x-lim and y-lim are ranges of data in that category respectively. The `circlize` package would know which cell you are drawing in.

## 3.3 Sectors and tracks

A circos layout is composed of sectors and tracks, as illustrated in figure 3. 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 values in certain category.

Sectors are first allocated and is determined by `circos.initialize` and track allocation is then 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 each sector on the circle would be 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. If `x` which is the x-value corresponding to `factors` is specified, the range for x-value in different category 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



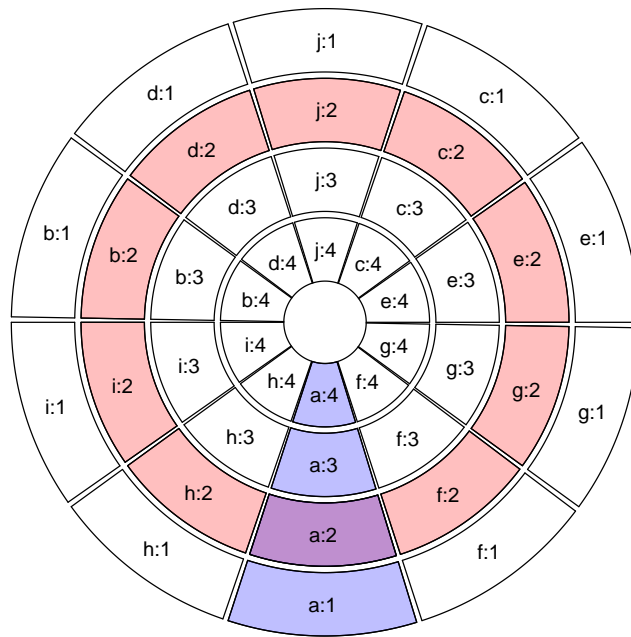


Figure 3: Coordinate in circo layout

to the x-ranges of a category and the order of rows in `xlim` corresponds to the order of levels of `factors`.

Since all cells in one sector 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` arguments to specify the range of y-values. There is also a `force.ylim` argument to specify whether all cells in one track should share the same y-ranges

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

In track creation step, since all the 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 been determined. If `factors` is just a vector, it would be converted to factor automatically. And finally if users just create cells in part of sectors in the track, in fact, the cells in remaining unspecified sectors would also be created, but with no borders (pretending they are not created).

### 3.4 Circos parameters

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

- **start.degree**: The starting degree which the circle begins to draw. Note this degree is measured in the standard polar coordinate which means it is anti-clockwise.
- **gap.degree**: Gap between two neighbour sectors. Note a gap is allocated first in front of the first sector. See figure 4.
- **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 4.
- **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 controls the bottom, left, top and right padding respectively. The four values are all percentages in which the first and the third padding values are the percentages according to the range of values on y-axis and the second and fourth values are the percentages according to the range of values on x-axis. See figure 4.

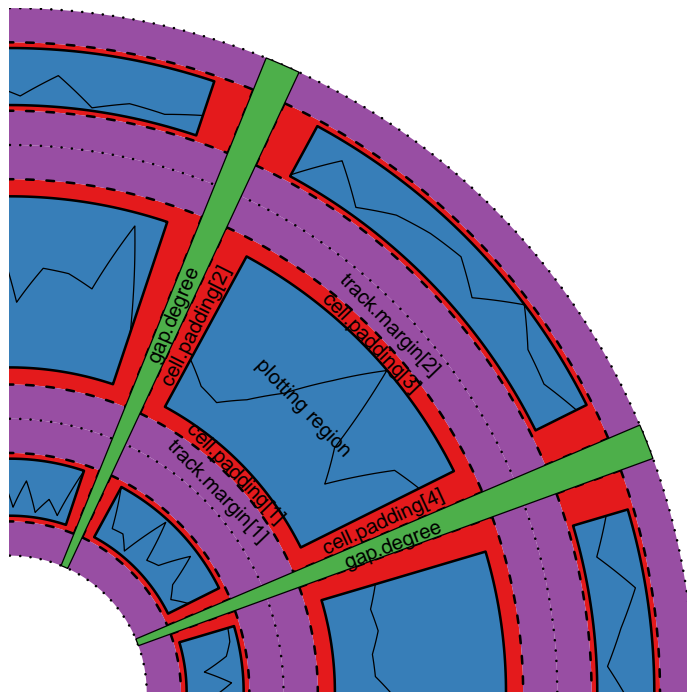


Figure 4: Regions for a cell

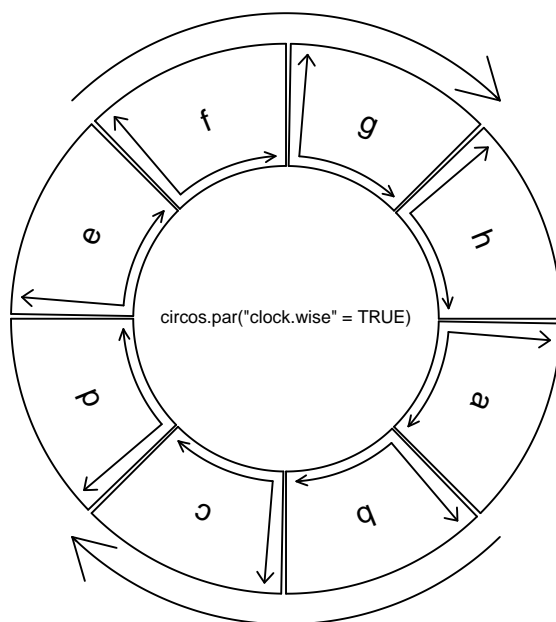
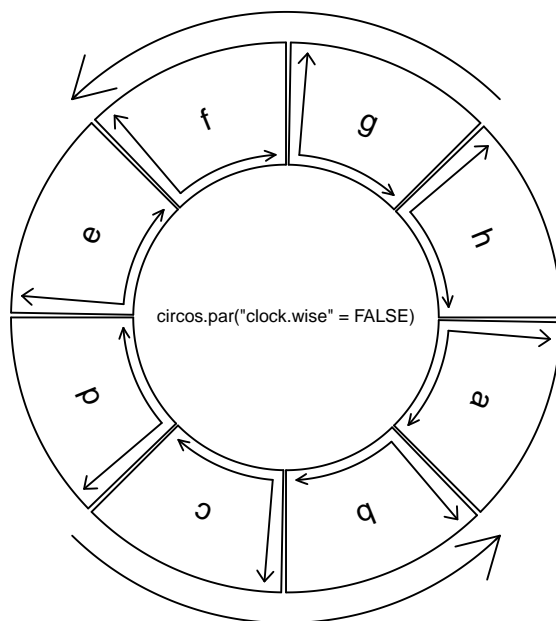


Figure 5: Sector directions

- `unit.circoe.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 segmentation is the length of the unit circle divided by `unit.circoe.segments`.
- `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. However, the definition would be changed in future version because I think it would be more reasonable to include the margins in the track.
- `points.overflow.warning`: Since each cell is in fact not a plotting region but only an ordinary rectangle, it does not eliminate points that are plotted out 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. By default, the package draws unit circle, so the xlim for the canvas would be `c(-1, 1)`. 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 anticlockwise (figure 5). But note that inside each cell, the direction of x-axis is always anticlockwise.

Parameters related to the allocation of sectors can be changes after the initialization of the 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` can not be modified either.

### 3.5 Points

Drawing points is similar as `points` function.

### 3.6 Lines

Parameters for drawing lines by `circos.lines` are similar to `lines` function, as illustrated in figure 6. One additional feature is that the areas under lines

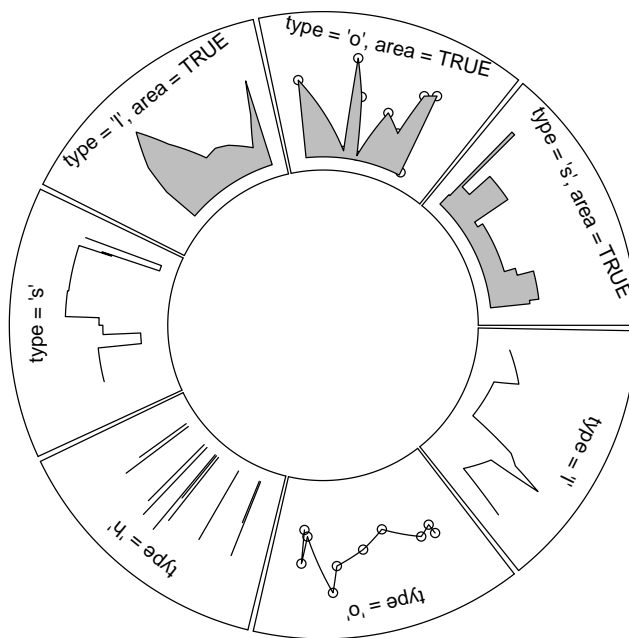


Figure 6: Line style settings

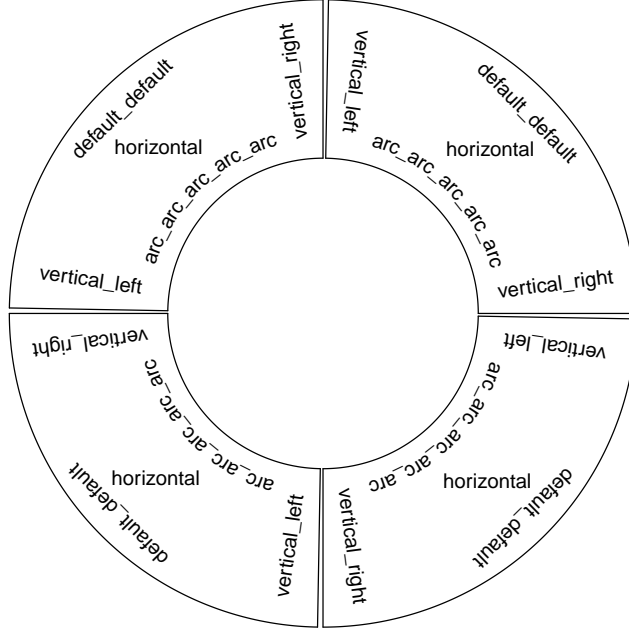


Figure 7: Text direction settings

can be specified by **area** argument. Also the base line for the area can be set by **area.baseline**.

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

### 3.7 Text

Only the direction of text by `circos.text` should be noted, as illustrated in figure 7. Only five direction of text is allowed which are in `c("default", "vertical_left", "vertical_right", "horizontal", "arc")`.

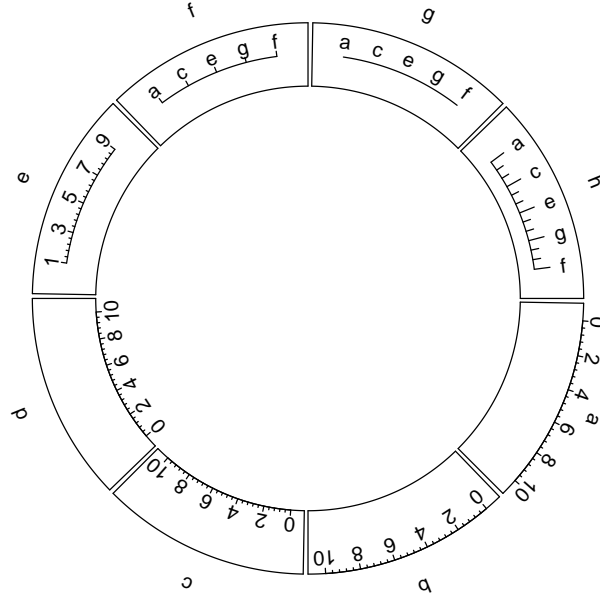


Figure 8: Axis settings

- **default**: direction of the tangent.
- **vertical\_left**: direction of radius, facing left at  $90^\circ$  position.
- **vertical\_right**: direction of radius, facing right at  $90^\circ$  position.
- **horizontal**: horizontal direction in the canvas coordinate.
- **arc**: direction of the arc.

### 3.8 Axis

Because there may be no space to draw y-axis, only x-axis for each cell is supported by `circos.axis`, as illustrated in figure 8. 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.



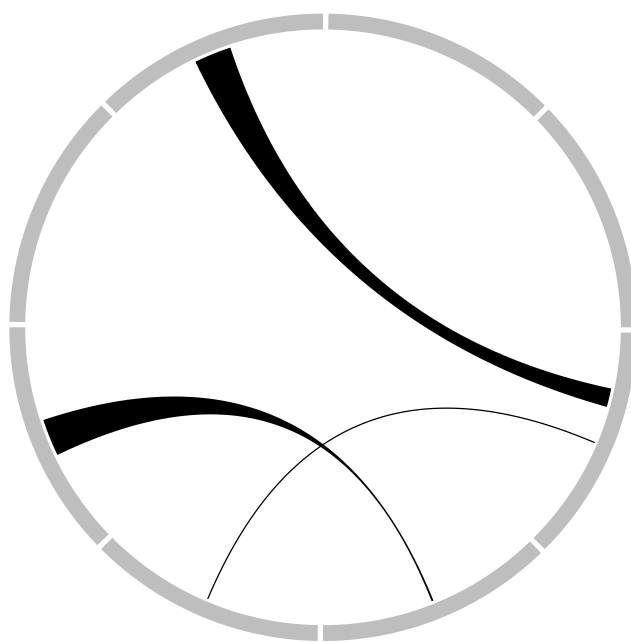


Figure 9: Link settings

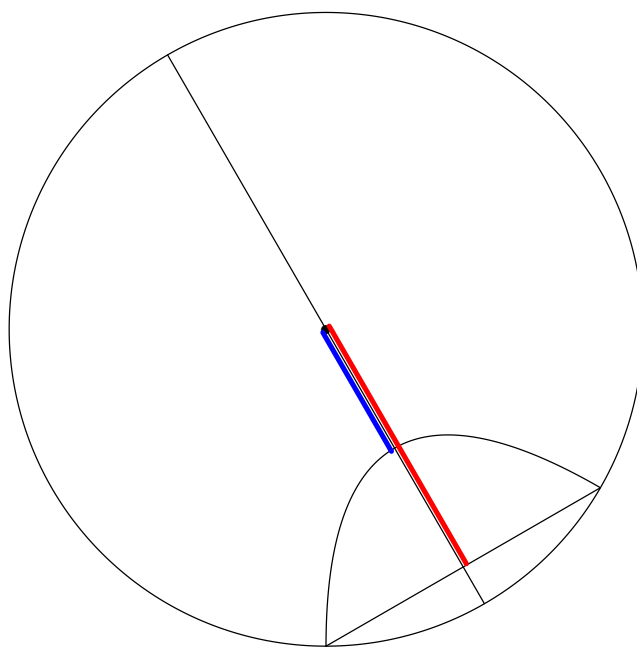


Figure 10: Additional settings for links

### 3.9 Links

Links can be drawn by `circos.link` from points and intervals (figure 9). 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. The link is in fact a quadratic curve.

The height of the link can be controlled by `top.ratio` argument in `circos.link` which is the ratio between the length of blue line and the red line (figure 10). Links do not hold any position of track.

### 3.10 The `panel.fun` argument in `circos.trackPlotRegion`

`panel.fun` argument in `circos.trackPlotRegion` is useful to apply plottings as soon as the cell has been created. This self-defined function need two arguments `x` and `y` which are data points in the cell. The value for such values are 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.

```
> 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)
+   })
```

### 3.11 High level plotting functions

With those low level function such as `circos.lines`, `circos.lines`, more high level functions can be easily written. The package provides a high level function `circos.trackHist` which draws the histogram or the density distribution of data (figure 11). So users would know how to implement other high level function to support graphs such as barplot, heatmap, et al. according to the source code of `circos.trackHist`.

In figure 11, the first track is histograms in which all the `ylim` are 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 different.

### 3.12 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 need argument of the position of circle center, the start degree and the end degree for the sector, and radius for two edges (or one edge) of the arc. An example is figure 12.

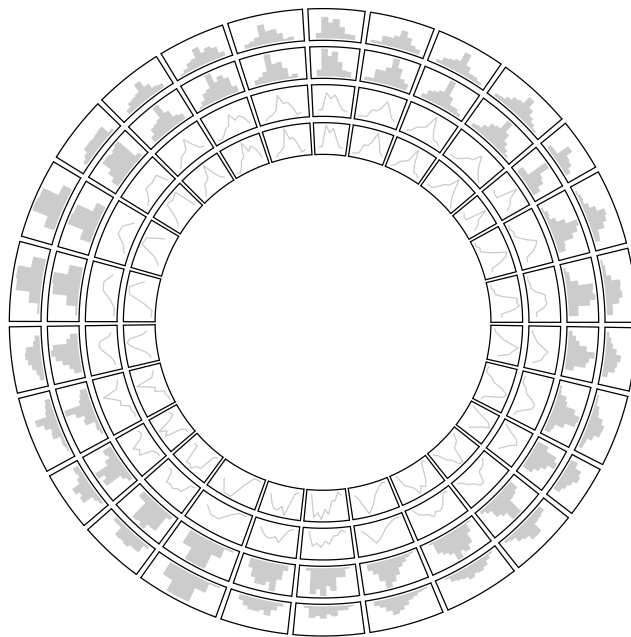


Figure 11: Drawing histograms

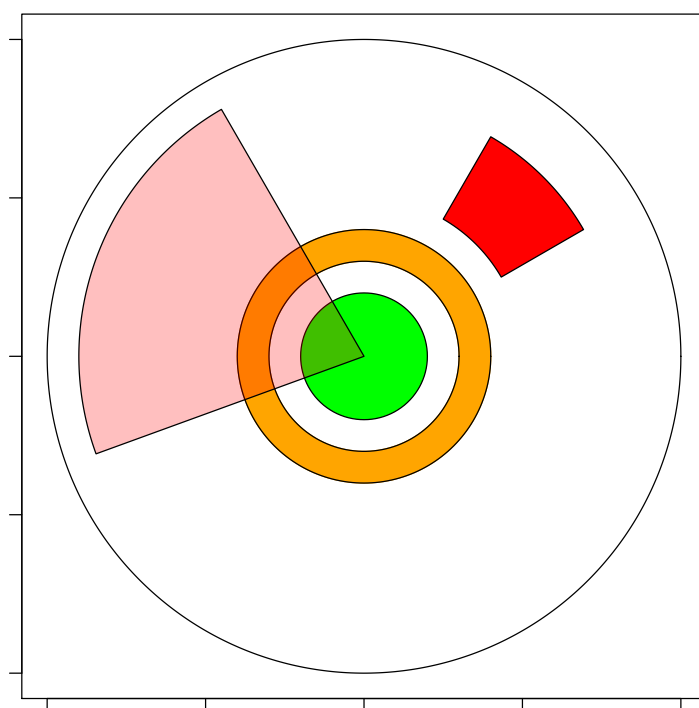


Figure 12: Hightlight sectors

### 3.13 Do not forget `circos.clear`

You should call `circos.clear` to complete end the `circos` plottings. Because there are several global variables tracing the status of the plot such as the index and position for the newest track. Such variables should be reset before drawing any new `circos` figures.

## 4 Advanced plottings

### 4.1 Draw part of the `circos` layout

`canvas.xlim` and `canvas.ylim` in `circos.par` is useful to draw only part of circle. Here only sector between  $0^\circ$  to  $90^\circ$  are plotted (figure 13). Codes are as follows. First, four sectors with the same length are initialized. Then only the first sector is drawn with points and lines.

```
> par(mar = c(1, 1, 1, 1))
> circos.par("canvas.xlim" = c(0, 1), "canvas.ylim" = c(0, 1),
+ "clock.wise" = FALSE)
> 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")
> circos.points(runif(100), runif(100), 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, runif(100), pch = 16, cex = 0.5)
> circos.clear()
```

### 4.2 Combine two parts of `circos` layouts

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

The first example is to draw one outer `circos` and an inner `circos` (figure 14).

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

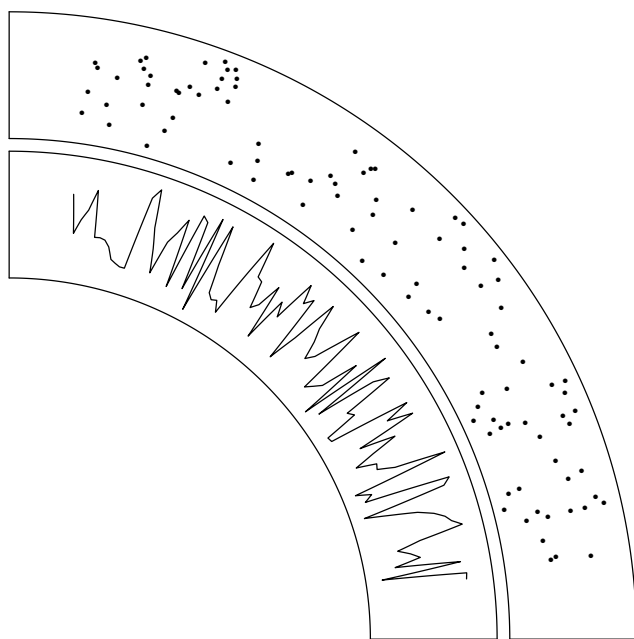


Figure 13: Part of the circos layout

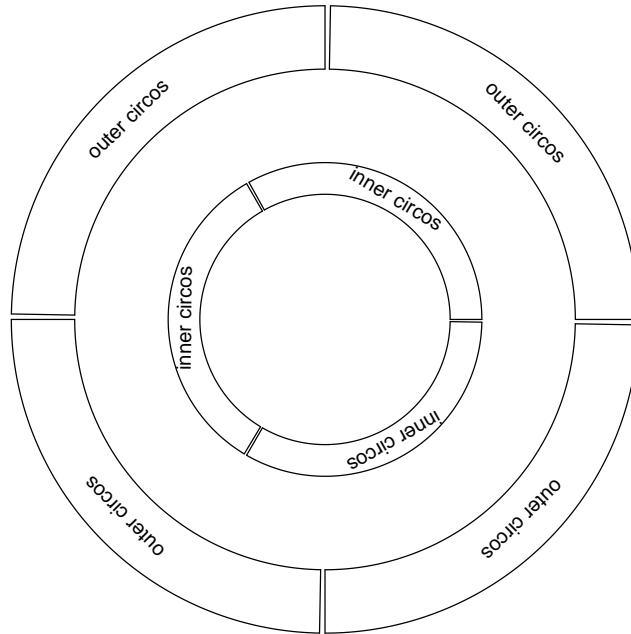


Figure 14: An outer and an inner circos layout

```
> 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 to draw to separated circos layouts in which every circos only contains a half (figure 15).

```
> library(circlize)
> par(mar = c(1, 1, 1, 1))
> factors = letters[1:4]
> circos.par("canvas.xlim" = c(-1, 1.2), "canvas.ylim" = c(-1, 1.2),
+   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(-1.2, 1), "canvas.ylim" = c(-1.2, 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()

```

### 4.3 Drawing outside and combine with canvas coordinate

Sometimes it is very useful to draw something outside the plotting region of cell. The following is a simple example to illustrate such circumstance (figure 16). The text is drawn outside the cell.

```

> 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))
> 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()

```

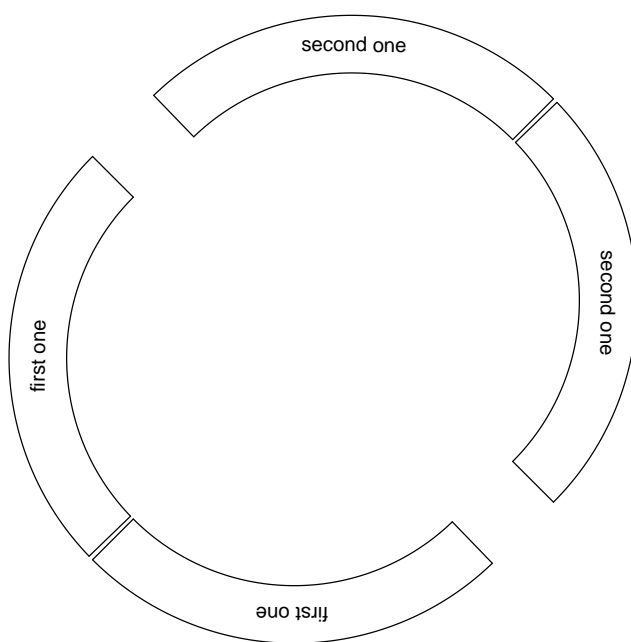


Figure 15: Two seperated circos layouts

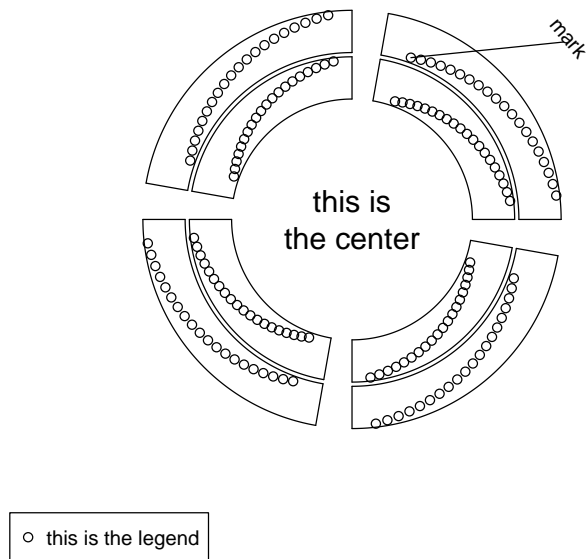


Figure 16: Drawing outside the cell and combine with canvas coordinate

Since the final graph is drawn in the ordinary canvas plotting region, we can add additional graphs through the traditional way. You can also see how **text** and **legend** work in the above example code.