An introduction to `circlize` package

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

July 5, 2013

1 Introduction

Circos layout (`http://circos.ca`) is very useful to represent complicated information, especially for genomic data. It is not only a way to visualize data, but also enhances the representation of scientific results into a level of aesthetics. The `circlize` package implements the circos layout in R. The advantage is that R is natural 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 for drawing 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, similar as `points`.
- `circos.lines`: draw lines in a cell, similar as `lines`.
- `circos.rect`: draw rectangle in a cell, similar as `rect`.
- `circos.polygon`: draw polygon in a cell, similar as `polygon`.
- `circos.text`: draw text in a cell, similar as `text`.
- `circos.axis`: draw 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 drawing 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.

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.updatePlotRegion`: update an existed 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 four 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 there are detailed explainations of circos parameters, coordinates and low-level functions. Finally it would tell you some tricks for drawing more complicated circos plot.

2 A quick glance

Following is an example to show the basic feature and usage of `circlize` package. 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 = 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 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.

```
> 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 1, 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. X-lims for cells in the track have already been defined in 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 a cell immediately. 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` and `track.index` argument). In drawing 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. 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. (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 1, top right). There are histograms among the track. The `circos.trackHist` can also create a new track because drawing histogram is really high-level. 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 1, middle left). Different background colors for cells can be assigned. So it may highlight some features 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` in `circos.updatePlotRegion`. 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 (figure 1, 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 fouth track (figure 1, bottom left). 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 (figure 1, 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, 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)
> circos.clear()
```

The final figure looks like figure 1.

3 Details

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

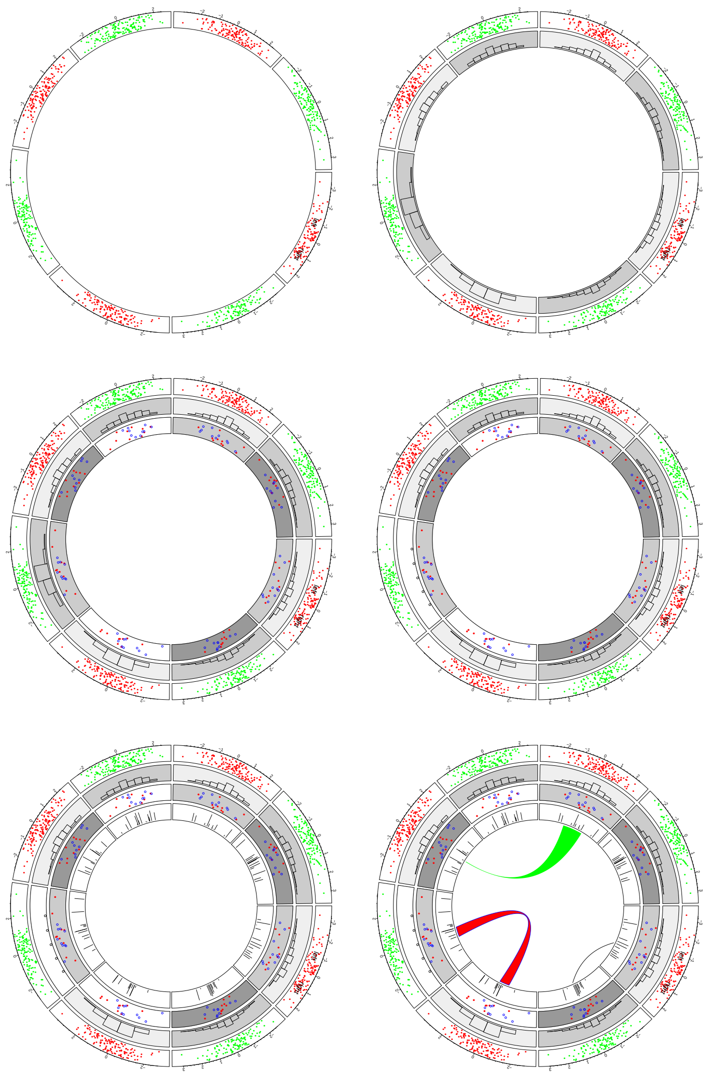


Figure 1: An example for circos layout

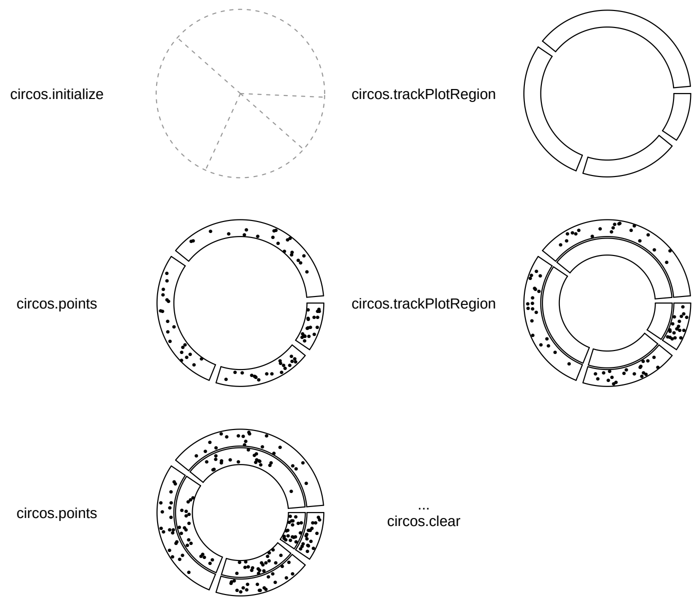


Figure 2: Order of drawing circos layout

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" (figure 2). See following:

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 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 graphs on it. There are three ways to do the plotting job.
   (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 and also it cannot draw complicated graphs.

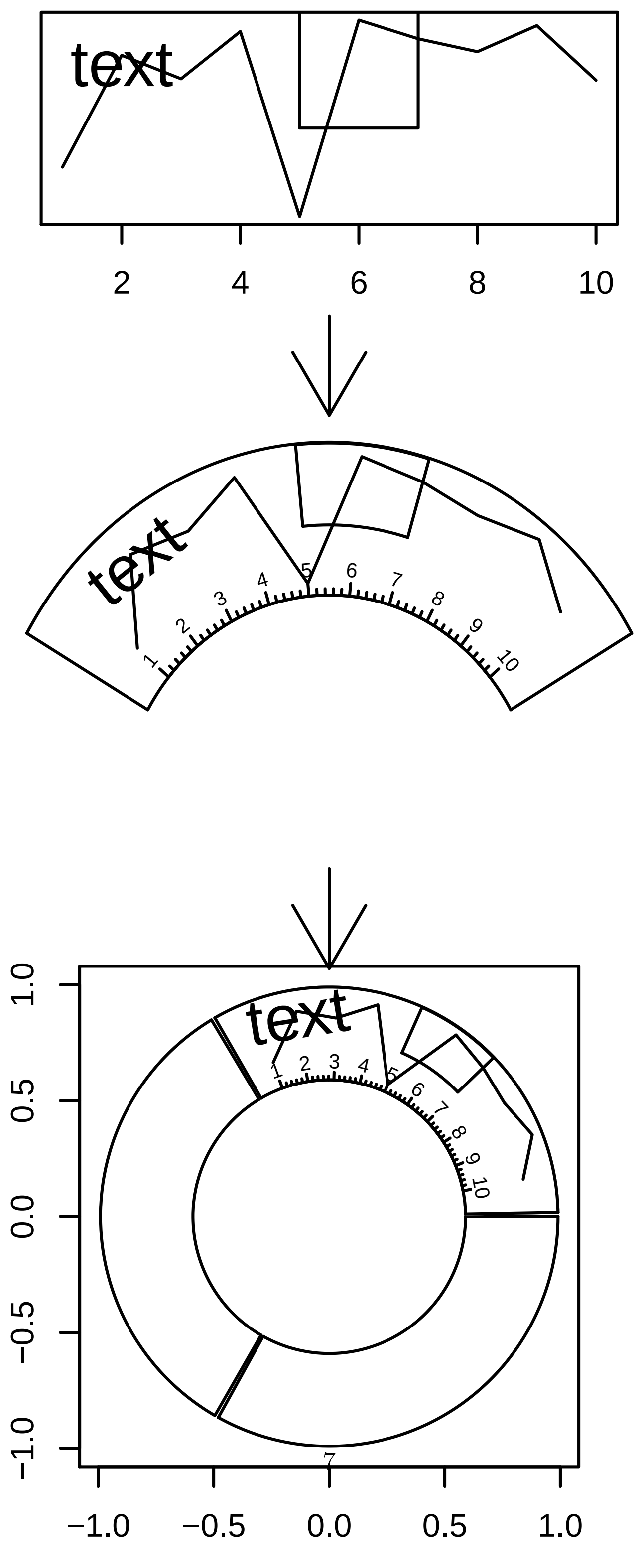


Figure 3: Transformation between different coordinates. Top: data coordinate; Middle: polar coordinate; Bottom: canvas coordinate.

(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 of data 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 rules would look like (pseudo code):

```
> 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)
+ }
```

or like following:

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

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

```
> circos.initialize(factors, xlim)
> 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, 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 is just drawn in the most nearly created cell. Note again, only `circos.trackPlotRegion` and `circos.updatePlotRegion` would reset the current track index and sector index.

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

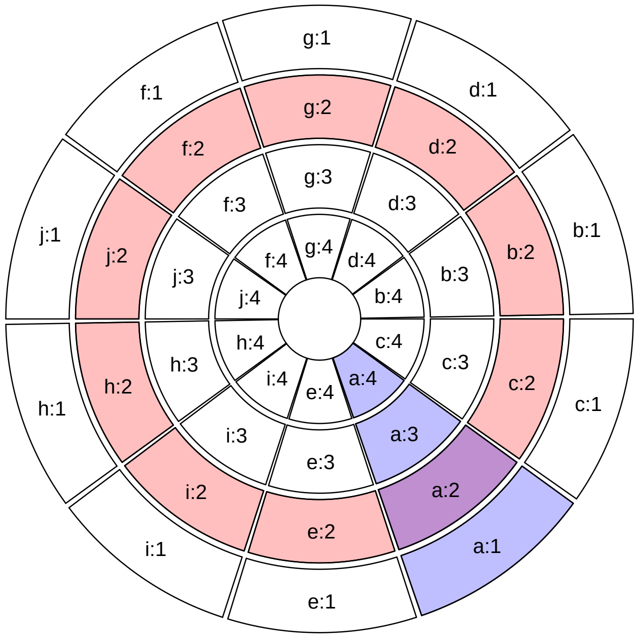


Figure 4: Coordinate in circos layout

3.2 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 different cells on a circle and a **canvas coordinate** which really draws the figures (figure 3). The package would first transform 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 circos layout is allways (or mostly except you want to draw something out of the plotting region) 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 the transformation.

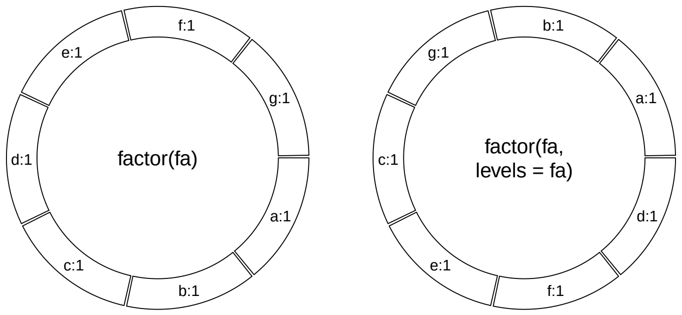


Figure 5: Different `factor` order in the initialization step.

3.3 Sectors and tracks

A circos layout is composed of sectors and tracks, as illustrated in figure 4. 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 a certain category (data coordinate).

Sectors are first allocated and 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. The following codes would generate different figures (figure 5):

```
> 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 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 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` argument to specify the range of y-values. There is also a `force.ylim` argument to sepcify whether all cells in one 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 finally if users just create cells in part of sectors in the track (not all sectors), 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 paramters are as follows, note some parameters can only be assigned before the initialization of the circos layout.

- `start.degree`: The starting degree at which the circle begin to draw. Note this degree is measured in the standard polar coordinate which means it is always reverse clockwise. See figure 7.

- `gap.degree`: Gap between two neighbour sectors. It can be a single value which means all gaps sharing same degree, or a vector which has same length as levels of the factors. See figure 7 and figure 6.

- `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 6.

- `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 paramter 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

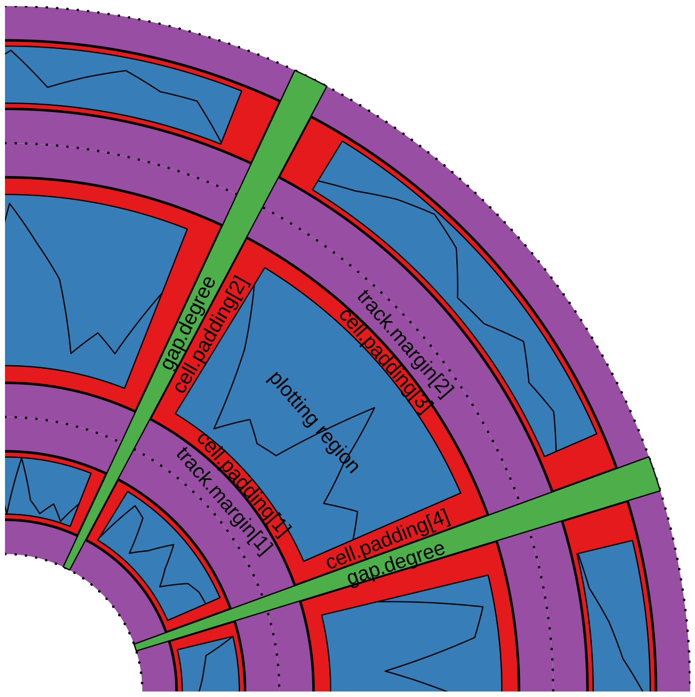
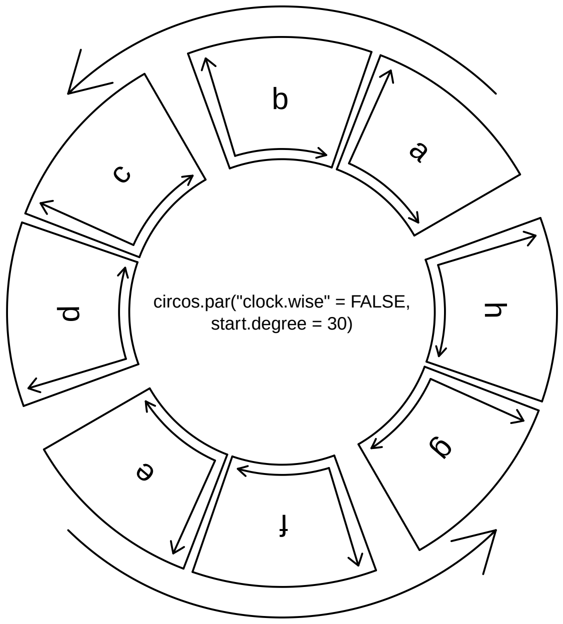


Figure 6: Regions for a cell



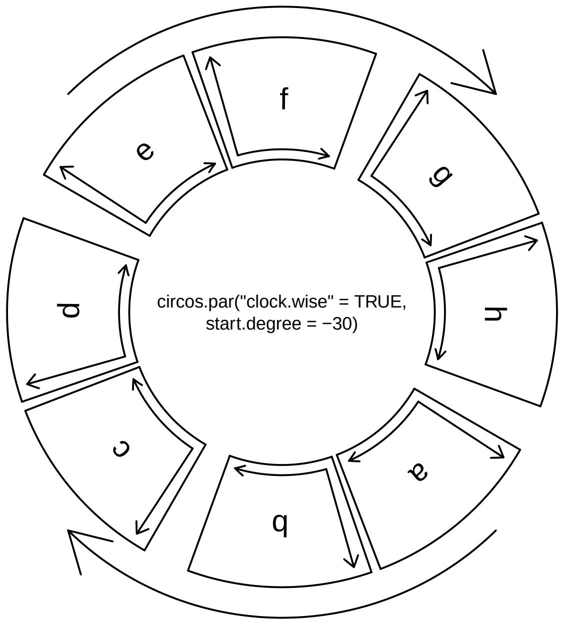


Figure 7: Sector directions. Sector orders are `a`, ..., `h`.

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 6.

- `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`.
- `default.track.height`: The default height of tracks. It is the percentage according to the radius of the unit circle (`1`). 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, 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. The package is forced to draw unit circle, so the `xlim` and `ylim` 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 clockwise (figure 7). 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.

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 the modification is effectiveless.

3.5 Points

Drawing 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)
```

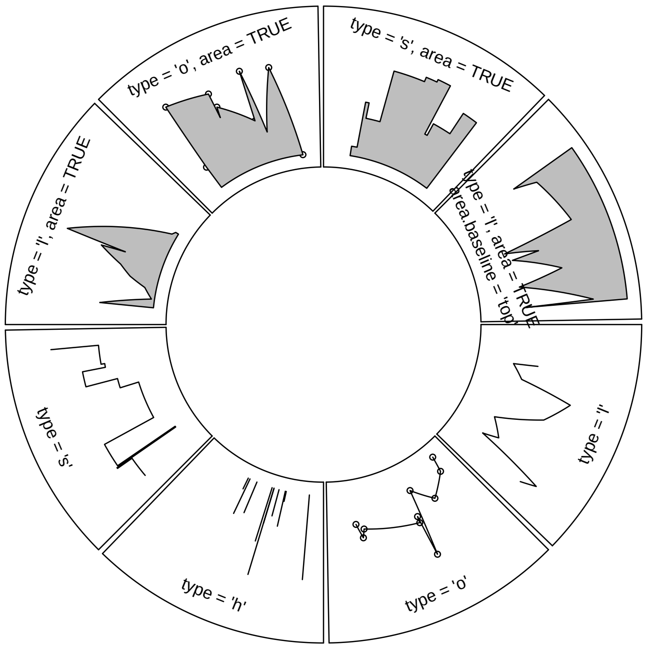


Figure 8: Line style settings

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 nearest `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.

3.6 Lines

Parameters for drawing lines by `circos.lines` are similar to `lines` function, as illustrated in figure 8. One additional feature is that the areas under/above lines can be specified by `area` argument which can help you identifying the direction of y-axis. Also the base line for the area can be set by `area.baseline`. `area.baseline` can be pre-defined string of `bottom` or `top`, or numeric values.

Straight lines will be transformed to curves when mapping to the circos layout. 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 as pdf format, especially for huge genomic data. So, 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 (such as radical lines), you can set `straight` argument to `TRUE`.

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, 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.

3.7 Text

Only the direction of text by `circos.text` should be noted, as illustrated in figure 9. Only six directions of text are allowed which are pre-defined in `c("default", "default2", "vertical_left", "vertical_right", "horizontal", "arc")`.

- `default`: direction of the tangent, facing bottom at 90° position.
- `default2`: direction of the tangent, facing bottom at −90° position.
- `vertical_left`: direction of radius, facing left at 90° position.
- `vertical_right`: direction of radius, facing right at 90° position.

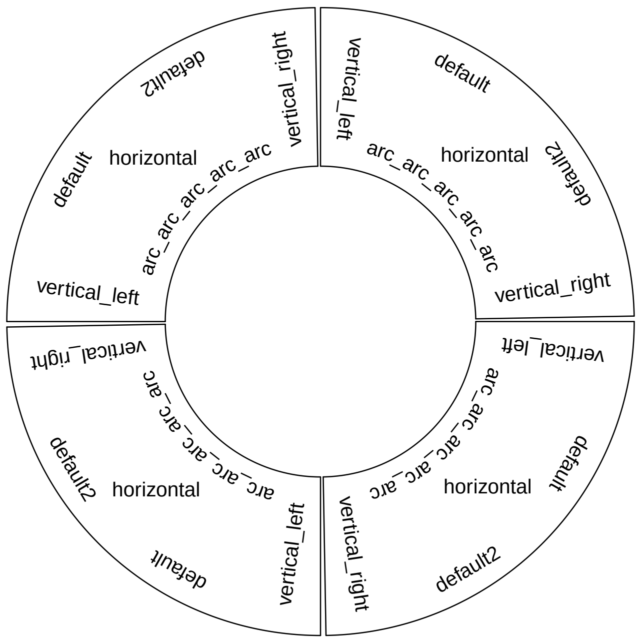


Figure 9: Text direction settings

- `horizontal`: horizontal direction in the canvas coordinate.
- `arc`: direction of the arc (not straight).

`srt` in `text` has been degenerated as `direction` in `circos.text` which only support only six types of rotation. 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, direction, adj, cex, col, font)
```

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

3.8 Rectangle

If you imagin the plotting region in a cell as Cartesian coordinate, then it draws rectangles. In the polar coordinate, the up and bottom edge become two arcs. Usage is similar as `rect`:

```
> 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)
```

3.9 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 10, 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.)

3.10 Axis

Because there may be no space to draw y-axis, only drawing x-axis for each cell is supported by `circos.axis`, as illustrated in figure 11. 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 differnet label directions to ensure the start/end position of the string is located near the major tick.

In figure 11, axis styles in different sectors are :

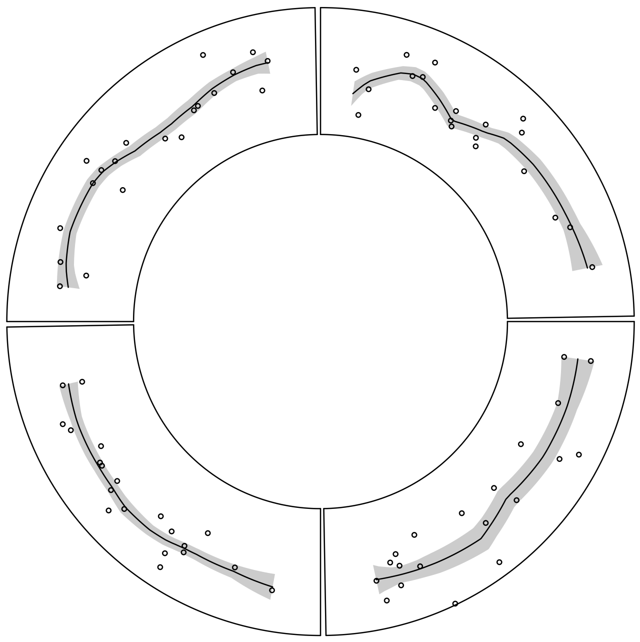


Figure 10: Area of standard deviation of the smooth line

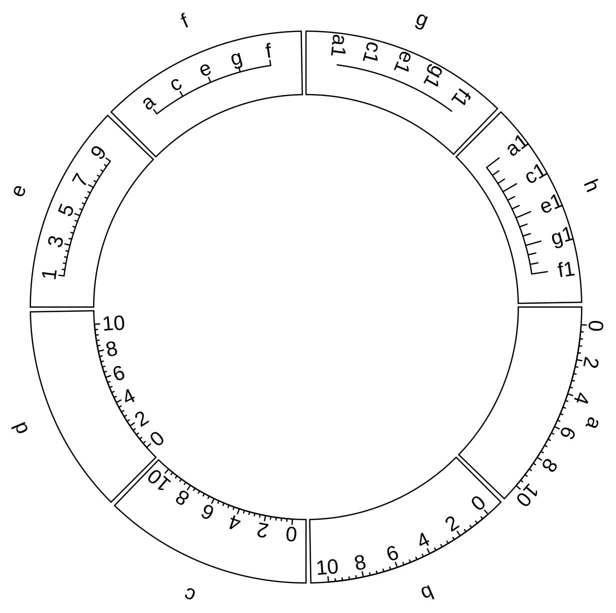


Figure 11: Axis settings

- 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 `default2`.
- 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 `vertical_left`.
- 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 `vertical_left`.
- 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 `vertical_right`.

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.direction, labels.away.percentage)
> circos.axis(h, major.at, labels, major.tick, minor.ticks,
+             major.tick.percentage, lwd)
```

3.11 Links

Links can be drawn by `circos.link` from points and intervals (figure 12, top). 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 curve. Links do not hold any position of track. So links can be overlapping with tracks.

The position of the 'root' of the link is controlled by `rou` argument. By default, it is the end position of the most recently created track. So normally, you don't need to care about this setting.

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 (maximum of the link height), see figure 12, bottom. The default height looks well from my view point, so you don't need to change this value.

Possible usage for `circos.link` is:

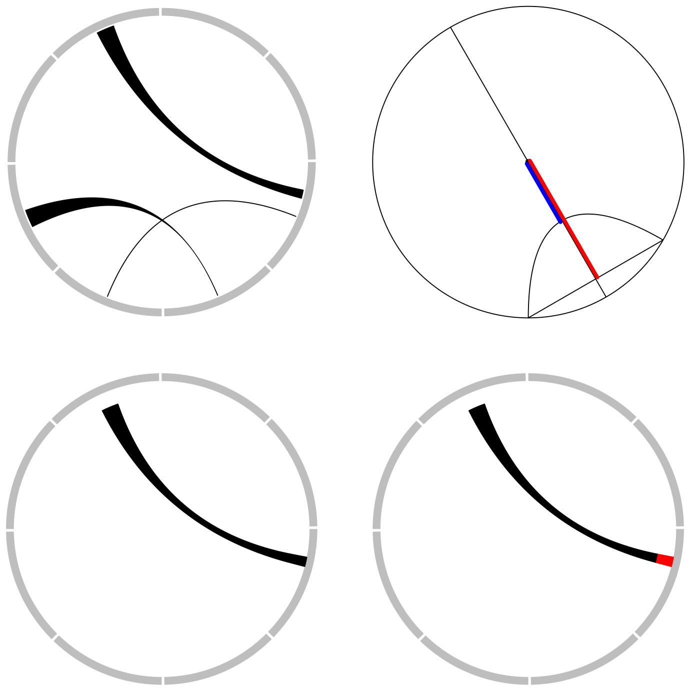


Figure 12: Drawing links

```
> 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, 0, sector.index2, 0, rou, top.ratio)
> circos.link(sector.index1, c(0, 1), sector.index2, 0,
+             col, lwd, lty, border)
```

circlize draws links as quadratic curves of which the symmetry axes pass through the center of the circle, so the two ends of the link must locate in a same circle. But sometimes drawing ends of links with different height is useful to represent the direction of links. Anyway, if the difference of the height of two ends are not too large, there is a way to approximate. The solution is add a rectangle at one end of a link (see figure 12). Following is the code:

```
> par(mar = c(1, 1, 1, 1))
> factors = letters[1:8]
> circos.initialize(factors = factors, xlim = c(0, 10))
>
> circos.par(track.margin = c(0.1, 0))
> circos.trackPlotRegion(factors = factors, ylim = c(0, 1), bg.col = "grey",
+    bg.border = NA, track.height = 0.05)
+
> circos.link("a", c(2, 3), "f", c(4, 6), border = "black")
>
> # the following codes calculate the position for the 'little rectangle'
> cell.ylim = get.cell.meta.data("cell.ylim", "a")
> d1 = circlize(2, cell.ylim[1], "a")
> theta1 = d1[1, 1]
> rou1 = d1[1, 2]
>
> d2 = circlize(3, cell.ylim[1], "a")
> theta2 = d2[1, 1]
> rou2 = d2[1, 2] - circos.par("track.margin")[1]
>
> # draw the 'little rectangle'
> draw.sector(start.degree = theta1, end.degree = theta2, rou1 = rou1,
+    rou2 = rou2, col = "black", border = "black")
>
> circos.clear()
```

Anyway, there are always some problems when implementing the link function. 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. I did some optimization to control such

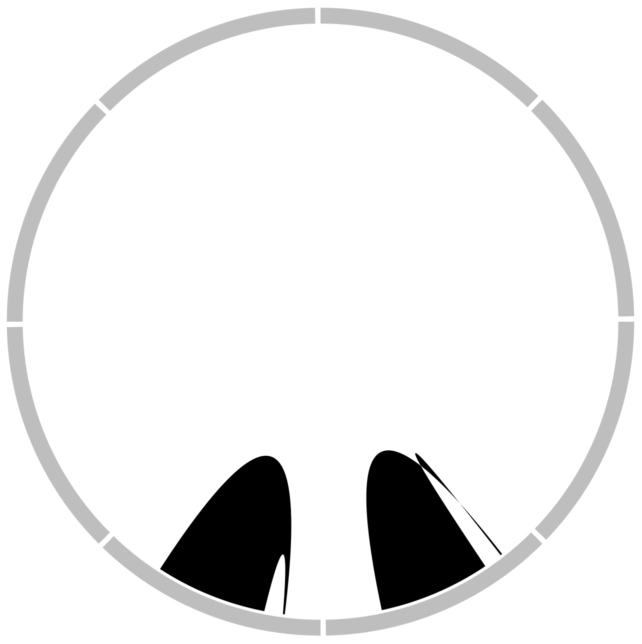


Figure 13: Adjust the links: right link is under default settings and the height of the low border of left link is adjusted by hand.

circumstance but it is not valid for all cases. However, I think such situation will not happen too often in real applications, but users can adjust the height of the low border by hand if they really meet it. The not-so-good argument `top.ratio.low` can control the height of the low border such as:

```
> circos.link("c", 1, "c", c(2, 8),
+     top.ratio = 0.5, top.ratio.low = 0.8)
```

The value of `top.ratio.low` must be larger than `top.ratio`. Example can be found in figure 13. The design of the function is not friendly and I am sorry for that. I would try to figure out a good solution in the future.

3.12 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 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 circos 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, the package would set the sector index and track index of the cell as the 'current' index for sector and track. When the cell is created, `panel.fun` would be exceeded afterward immediately. Without specifying `sector.index` and `track.index`, the 'current' one would be used.

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

3.13 High level plotting functions

With those low-level function such as `circos.points`, `circos.lines`, more high-level functions can be easily written. The package provides a high-level function `circos.trackHist` which draws histograms or the density distributions of data (figure 14). So users would know how to implement other high-level function to support graphs such as barplot, heatmap, ... accroding to the source code of `circos.trackHist`.

In figure 14, 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 15 you would see heatmaps and cluster dendrograms in a circos layout. Heatmaps are 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` in the source code of this vignette. If you want to draw heatmap or dendrogram (like

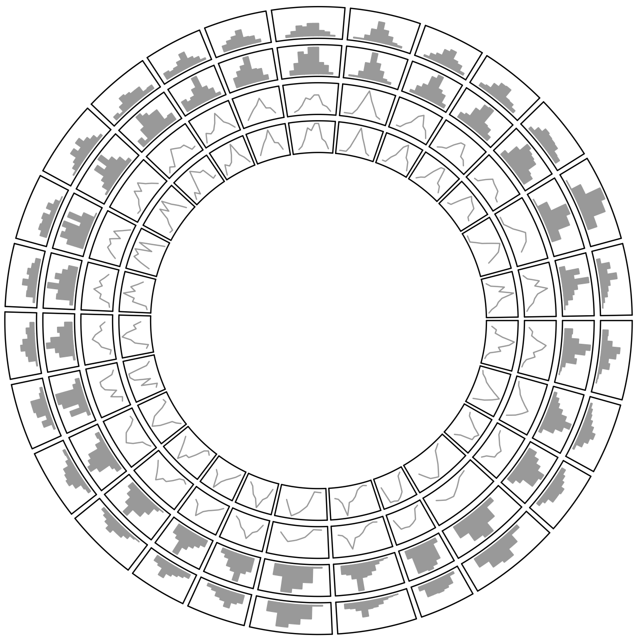


Figure 14: Histograms

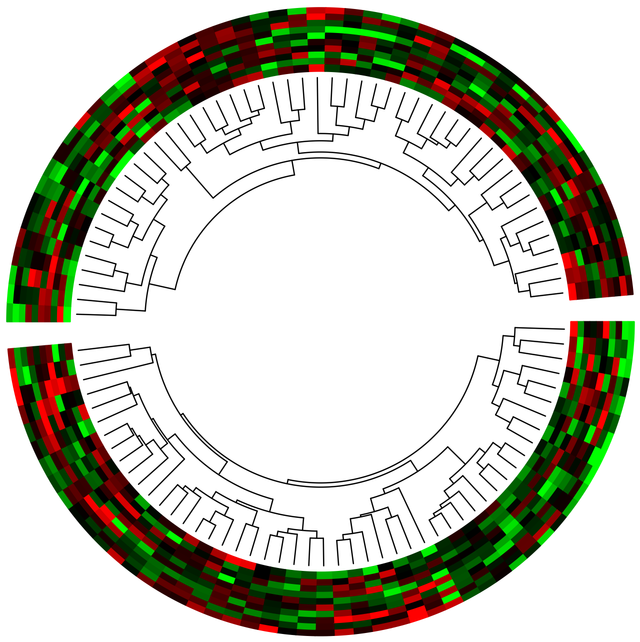


Figure 15: Heatmap with clustering

evolution tree) by your own, this may be helpful for you. (type `help(package = "circlize")` in your R terminal, click link of "Overview of user guides and package vignettes" in the browser, click link of "R code" corresponding to "circlize::circlize", and search for code chunk of "figheatmap".)

3.14 Other functions

`draw.sector` can be used to draw sectors or part of a ring. This is useful if you want to hightlight 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. 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 16 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 circos 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 usually, 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
- `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
- `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]`.

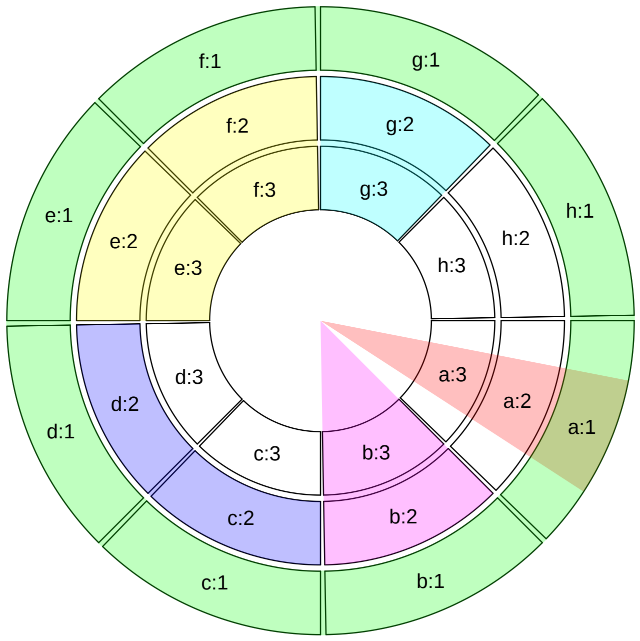


Figure 16: Hightlight sectors

- `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

3.15 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 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. In the example, only sectors between 0° to 90° are plotted (figure 17). First, four sectors with the same width are initialized. Then only the first sector is drawn with points and lines. From figure 17, we in fact created the whole circle, but only a quarter of the circle is in the canvas region. Codes are as follows.

```
> 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)
```

In the second situation, you don't need some sectors or cells but still y-ou 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 graphs on it (figure 18).

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

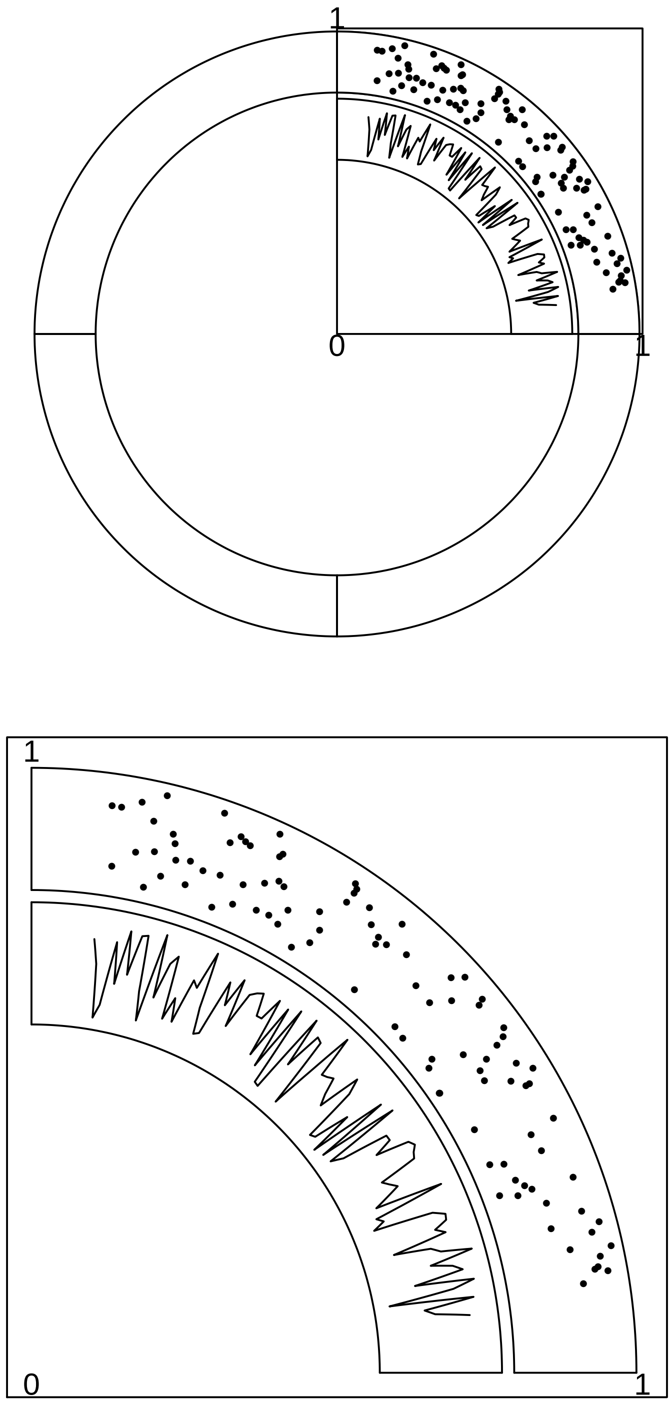


Figure 17: Part of the circos layout

```
> 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()
```

4.2 Combine several parts of circos layouts

Since the circos layout by `circlize` is finally plotted in an ordinary R plotting system. Two seperated 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 canvas 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 19).

```
> 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()
> 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 circos layouts in which every circos only contains a half (figure 20).

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

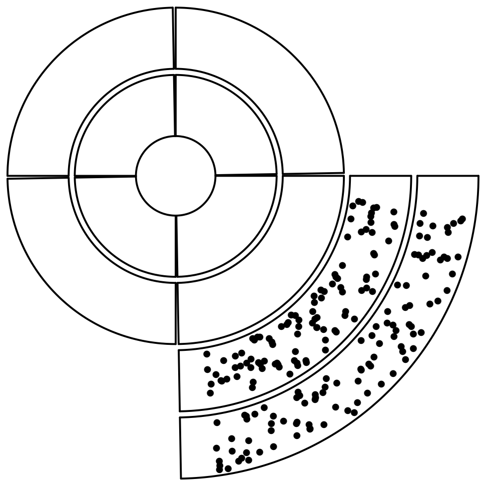


Figure 18: Part of the circos layout, situation 2.

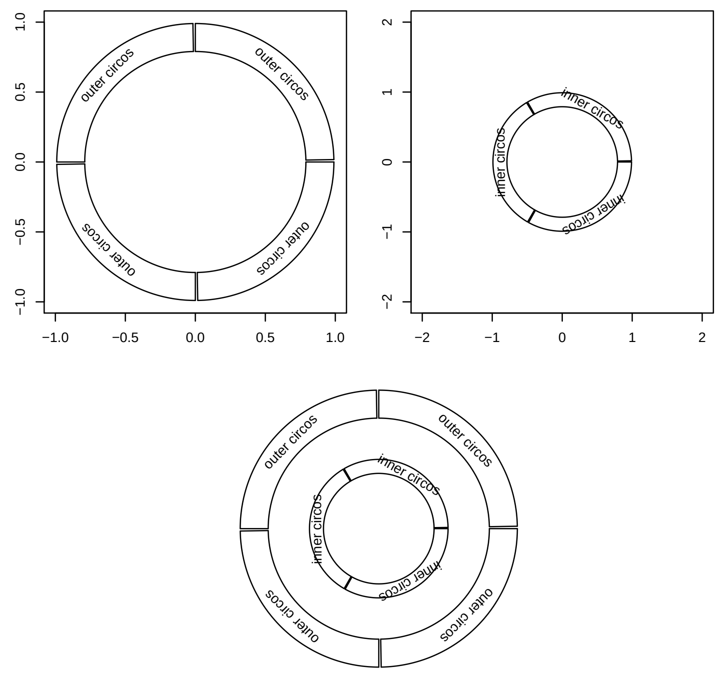


Figure 19: An outer and an inner circos layout

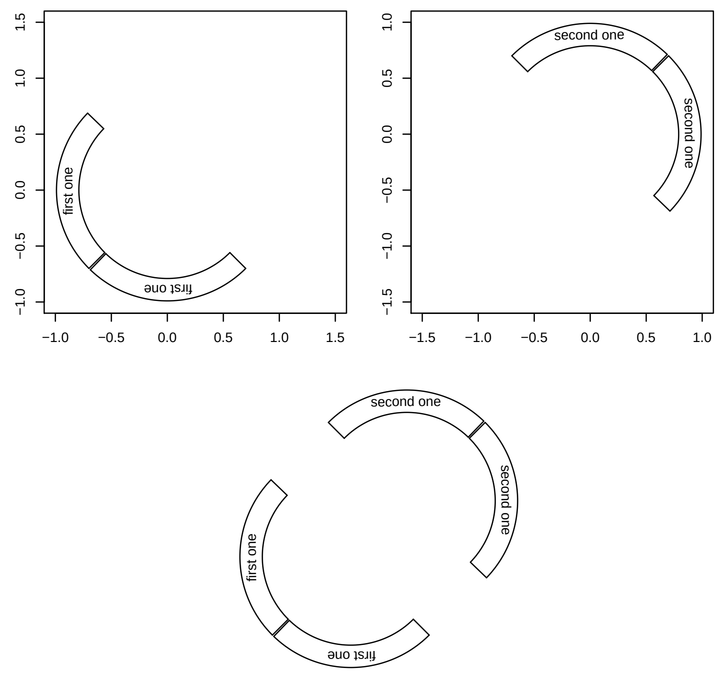


Figure 20: Two seperated circos layouts

```
> 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 21). 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 layout.

It is different from example in "Draw part of the circos layout" section. In that example, cells both visible and invisible are all in 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).

```
> 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)
```

4.3 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 22). The text is drawn outside the cell.

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

```
> 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))
> 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) {
```

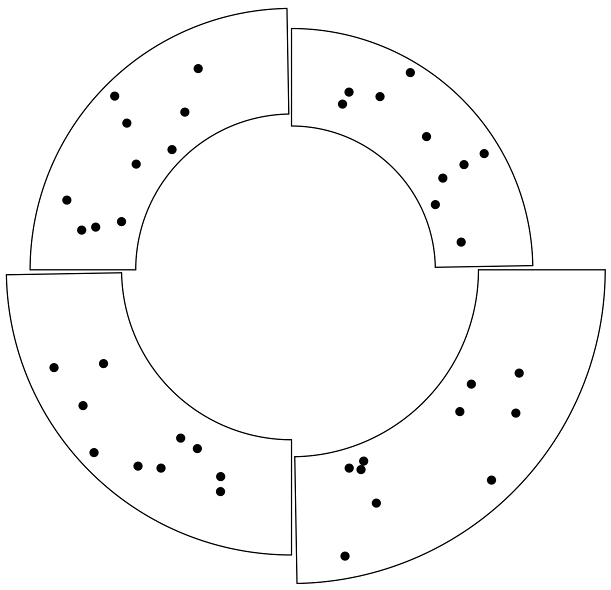


Figure 21: Sectors with different radius

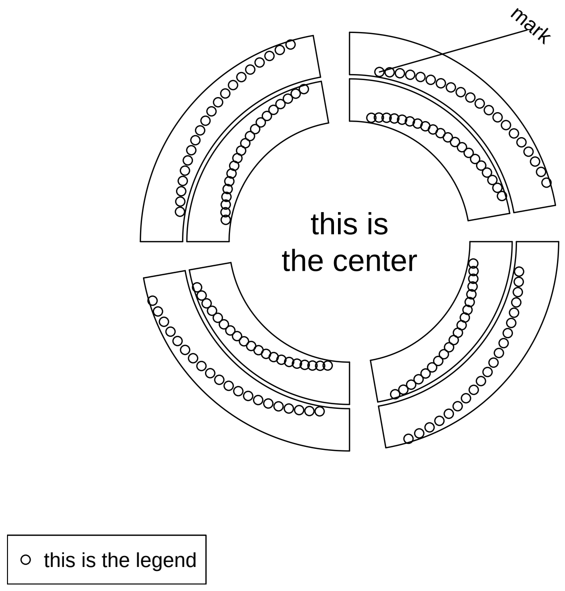


Figure 22: Draw outside the cell and combine with canvas coordinate

```
+     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()
```

4.4 Draw figures with `layout`

You can use `layout` to arrange multiple figures together (also it is available from `par(mfrow)` or `par(mfcol)`) (figure 23).

```
> library(circlize)
> set.seed(12345)
> rand_color = function() {
+     return(rgb(runif(1), runif(1), runif(1)))
+ }
> layout(matrix(1:9, 3, 3))
> for(i in 1:9) {
```

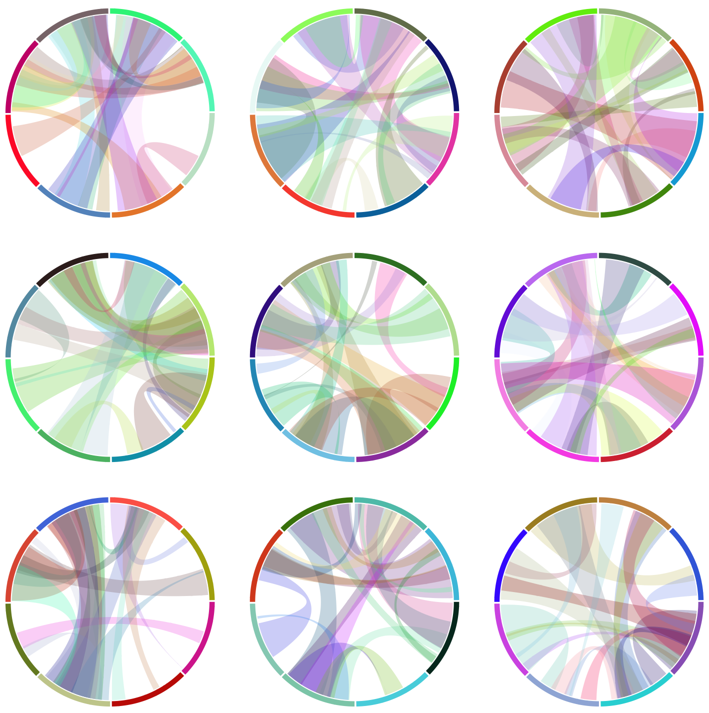


Figure 23: Draw multiple figures by `layout`

```
+     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()
+ }
```