Circular Visualization in R

Zuguang Gu

last revised on 2017-04-27

Contents

About **7**

I General Functionality **9**

1 Introduction **11**
1.1 Principle of design . . . 11
1.2 A quick glance . . . 12

2 Circular layout **19**
2.1 Coordinate transformation . . . 19
2.2 Rules for making the circular plot . . . 19
2.3 Sectors and tracks . . . 22
2.4 Graphic parameters . . . 24
2.5 Create plotting regions . . . 26
2.6 Update plotting regions . . . 26
2.7 `panel.fun` argument . . . 27
2.8 Other utilities . . . 29

3 Graphics **35**
3.1 Points . . . 35
3.2 Lines . . . 36
3.3 Segments . . . 37
3.4 Text . . . 37
3.5 Rectangles and polygons . . . 38
3.6 Axes . . . 40
3.7 Links . . . 41
3.8 Highlight sectors and tracks . . . 45
3.9 Work together with the base graphic system . . . 49

4 Legends **51**

5 Implement high-level circular plots **57**
5.1 Circular barplots . . . 57
5.2 Histograms . . . 58
5.3 Phylogenetic trees . . . 59
5.4 Heatmaps . . . 61

6 Advanced layout **63**
6.1 Zooming of sectors . . . 63
6.2 Visualize part of the circle . . . 65
6.3 Combine multiple circular plots . . . 66
6.4 Arrange multiple plots . . . 69

II Applications in Genomics 71

7 Introduction 73
7.1 Input data . . . 73

8 Initialize with genomic data 75
8.1 Initialize with cytoband data . . . 75
8.2 Customize chromosome track . . . 78
8.3 Initialize with general genomic category . . . 80
8.4 Zooming chromosomes . . . 81

9 Create plotting regions for genomic data 85
9.1 Points . . . 86
9.2 Lines . . . 87
9.3 Text . . . 87
9.4 Rectangles . . . 88
9.5 Links . . . 88
9.6 Mixed use of general circlize functions . . . 88

10 modes for `circos.genomicTrack()` 91
10.1 Normal mode . . . 91
10.2 Stack mode . . . 92
10.3 Applications . . . 93

11 High-level genomic functions 99
11.1 Ideograms . . . 99
11.2 Heatmaps . . . 99
11.3 Labels . . . 100
11.4 Genomic density and Rainfall plot . . . 101

12 Nested zooming 105
12.1 Basic idea . . . 105
12.2 Visualization of DMRs from tagmentation-based WGBS . . . 109

III Visualize Relations 113

13 The `chordDiagram()` function 115
13.1 Basic usage . . . 116
13.2 Adjust by `circos.par()` . . . 117
13.3 Colors . . . 120
13.4 Link border . . . 123
13.5 Highlight links . . . 125
13.6 Orders of links . . . 129
13.7 Self-links . . . 130
13.8 Symmetric matrix . . . 130
13.9 Directional relations . . . 130
13.10Reduce . . . 135

14 Advanced usage of `chordDiagram()` 139
14.1 Organization of tracks . . . 139
14.2 Customize sector labels . . . 140
14.3 Customize sector axes . . . 142
14.4 Put horizontally or vertically symmetric . . . 145
14.5 Compare two Chord diagrams . . . 145

14.6 Multiple-group Chord diagram 147

15 A complex example of Chord diagram **151**

IV Others **159**

16 Make fun of the package **161**
16.1 A clock 161
16.2 A dartboard 161
16.3 Ba-Gua and Tai-Ji 163
16.4 Circular doodle 164

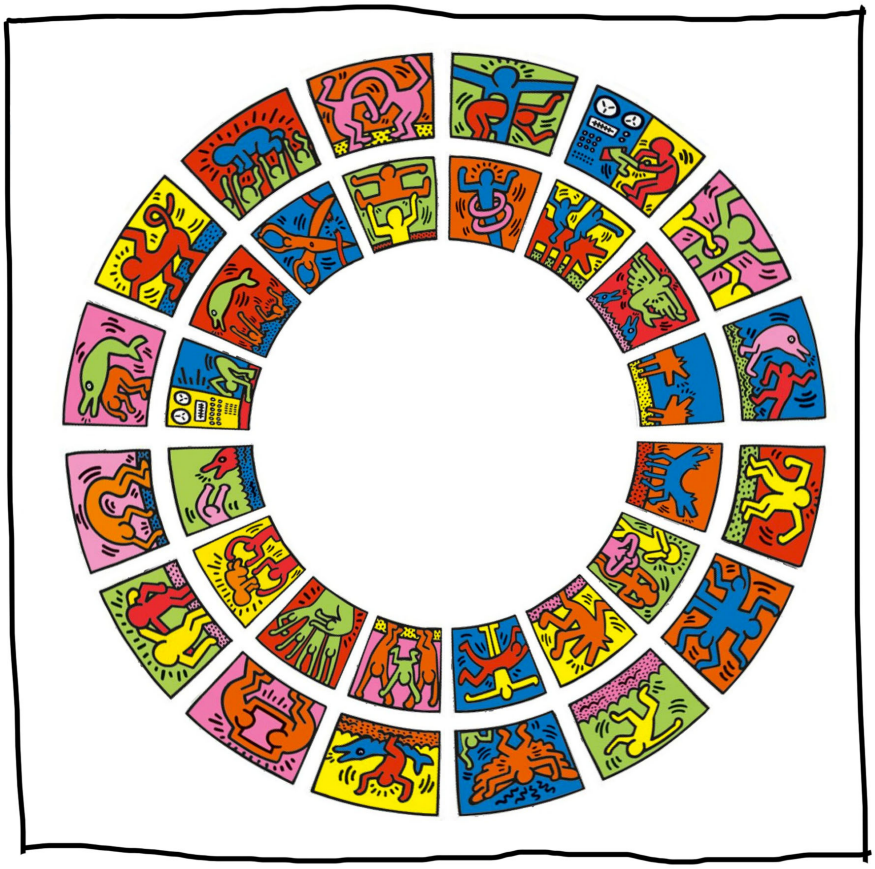
About

This is the documentation of the **circlize** package. Examples in the book are generated under version 0.4.0.

If you use **circlize** in your publications, I would be appreciated if you can cite:

Gu, Z. (2014) circlize implements and enhances circular visualization in R. Bioinformatics. DOI: 10.1093/bioinformatics/btu393

CIRCULAR VISUALIZATION IN R



Zuguang Gu

Part I

General Functionality

Chapter 1

Introduction

Circular layout is very useful to represent complicated information. First, it elegantly represents information with long axes or a large amount of categories; second, it intuitively shows data with multiple tracks focusing on the same object; third, it easily demonstrates relations between elements. It provides an efficient way to arrange information on the circle and it is beautiful.

Circos is a pioneer tool widely used for circular layout representations implemented in *Perl*. It greatly enhances the visualization of scientific results (especially in Genomics field). Thus, plots with circular layout are normally named as **"circos plot"**. Here the **circlize** package aims to implement **Circos** in R. One important advantage for the implementation in R is that R is an ideal environment which provides seamless connection between data analysis and data visualization. **circlize** is not a front-end wrapper to generate configuration files for **Circos**, while completely coded in R style by using R's elegant statistical and graphic engine. We aim to keep the flexibility and configurability of **Circos**, but also make the package more straightforward to use and enhance it to support more types of graphics.

In this book, chapters in Part I give detailed overviews of the general **circlize** functionalities. Part II introduces functions specifically designed for visualizing genomic datasets. Part III gives comprehensive guilds on visualizing relationships by Chord diagram.

1.1 Principle of design

A circular layout is composed of sectors and tracks. For data in different categories, they are allocated into different sectors and for multiple measurements of the same category, they are represented as stacked tracks from outside of the circle to the inside. The intersection of a sector and a track is called a cell (or a grid, a panel), which is the basic unit in a circular layout. It is an imaginary plotting region for data points in a certain category.

Since most of the figures are composed of simple graphics, such as points, lines, polygon, **circlize** implements low-level graphic functions for adding graphics in the circular plotting regions, so that more complicated graphics can be easily generated by different combinations of low-level graphic functions. This principle ensures the generality that types of high-level graphics are not restricted by the software itself and high-level packages focusing on specific interests can be built on it.

Currently there are following low-level graphic functions that can be used for adding graphics. The usage is very similar to the functions without `circos.` prefix from the base graphic engine, except there are some enhancement specifically designed for circular visualization.

- `circos.points()`: adds points in a cell.
- `circos.lines()`: adds lines in a cell.
- `circos.segments()`: adds segments in a cell.

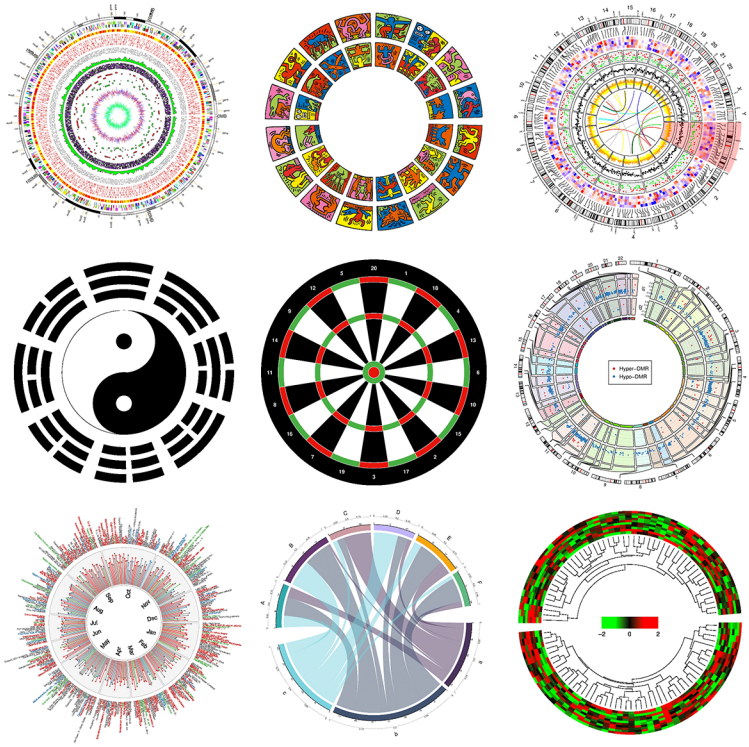


Figure 1.1: Examples by circlize

- `circos.rect()`: adds rectangles in a cell.
- `circos.polygon()`: adds polygons in a cell.
- `circos.text()`: adds text in a cell.
- `circos.axis()` ands `circos.yaxis()`: add axis in a cell.

Following functions arrange the circular layout.

- `circos.initialize()`: allocates sectors on the circle.
- `circos.track()`: creates plotting regions for cells in one single track.
- `circos.update()`: updates an existed cell.
- `circos.par()`: graphic parameters.
- `circos.info()`: prints general parameters of current circular plot.
- `circos.clear()`: resets graphic parameters and internal variables.

Thus, theoretically, you are able to draw most kinds of circular figures by the above functionalities. Figure 1.1 lists several complex circular plots made by **circlize**. After going through this book, you will definitely be able to implement yours.

1.2 A quick glance

Before we go too deep into the details, I first demonstrate a simple example with using basic functionalities in **circlize** package to help you to get a basic idea of how the package works.

First let's generate some random data. There needs a character vector to represent categories, a numeric vector of x values and a vectoe of y values.

```
set.seed(999)
n = 1000
df = data.frame(factors = sample(letters[1:8], n, replace = TRUE),
    x = rnorm(n), y = runif(n))
```

First we initialize the circular layout. The circle is split into sectors based on the data range on x-axes in each category. In following code, `df$x` is split by `df$factors` and the width of sectors are automatically calculated based on data ranges in each category. Be default, sectors are positioned started from $\theta = 0$ (in the polar coordinate system) and go along the circle clock-wisely. You may not see anything after running following code because no track has been added yet.

```
library(circlize)
circos.par("track.height" = 0.1)
circos.initialize(factors = df$factors, x = df$x)
```

We set a global parameter `track.height` to 0.1 by the option function `circis.par()` so that all tracks which will be added have a default height of 0.1. The circle used by **circlize** always has a radius of 1, so a height of 0.1 means 10% of the circle radius.

Note that the allocation of sectors only needs values on x direction (or on the circular direction), the values on y direction (radical direction) will be used in the step of creating tracks.

After the circular layout is initialized, graphics can be added to the plot in a track-by-track manner. Before drawing anything, we need to know that all tracks should be first created by `circos.trackPlotRegion()` or, for short, `circos.track()`, then the low-level functions can be added afterwards. Just think in the base R graphic engine, you need first call `plot()` then you can use functions such as `points()` and `lines()` to add graphics. Since x ranges for cells in the track have already been defined in the initialization step, here we only need to specify the y ranges for each cell. The y ranges can be specified by `y` argument as a numeric vector (so that y ranges will be automatically extracted and calculated in each cell) or `ylim` argument as a vector of length two. In principle, y ranges should be same for all cells in a same track. (See Figure 1.2)

```
circos.track(factors = df$factors, y = df$y,
    panel.fun = function(x, y) {
        circos.text(CELL_META$xcenter, CELL_META$cell.ylim[2] + uy(5, "mm"),
            CELL_META$sector.index)
        circos.axis(labels.cex = 0.6)
})
col = rep(c("#FF0000", "#00FF00"), 4)
circos.trackPoints(df$factors, df$x, df$y, col = col, pch = 16, cex = 0.5)
circos.text(-1, 0.5, "text", sector.index = "a", track.index = 1)
```

Axes for the circular plot are normally drawn on the most outside of the circle. Here we add axes in the first track by putting `circos.axis()` inside the self-defined function `panel.fun` (see the code above). `circos.track()` creates plotting region in a cell-by-cell manner and the `panel.fun` is actually executed immediately after the plotting region for a certain cell is created. Thus, `panel.fun` actually means adding graphics in the "current cell" (Usage of `panel.fun` is further discussed in Section 2.7). Without specifying any arguments, `circos.axis()` draws x-axes on the top of each cell (or the outside of each cell).

Also, we add sector name outside the first track by using `circos.text()`. `CELL_META` provides "meta information" for the current cell. There are several parameters which can be retrieved by `CELL_META`. All its usage is explained in Section 2.7. In above code, the sector names are drawn outside the cells and you may see warning messages saying data points exceeding the plotting regions. That is total fine and no worry about it. You can also add sector names by creating an empty track without borders as the first track and add sector names in it (like what `circos.initializeWithIdeogram()` and `chordDiagram()` do, after you go through following chapters).

When specifying the position of text on the y direction, an offset of `uy(5, "mm")` is added to the y position

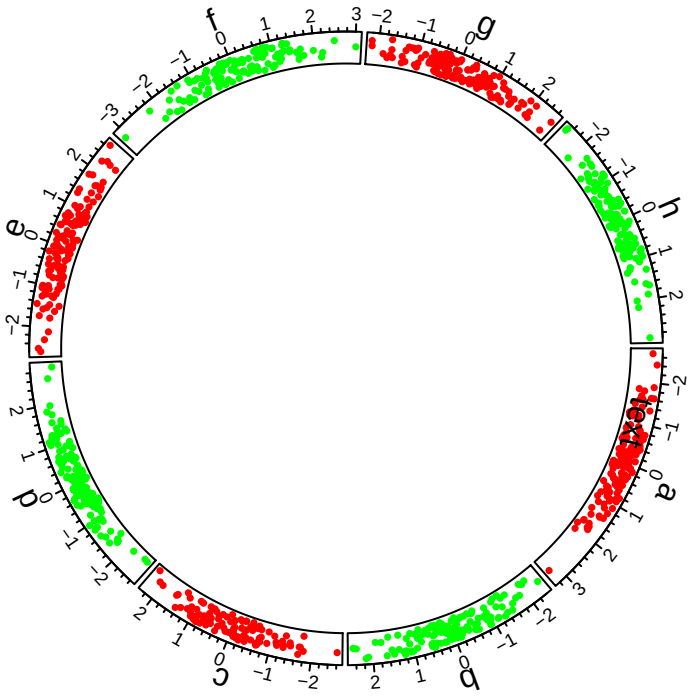


Figure 1.2: First example of circlize, add the first track.

of the text. In `circos.text()`, x and y values are measured in the data coordinate (the coordinate in cell), and `uy()` function (or `ux()` which is measured on x direction) converts absolute units to corresponding values in data coordinate. Section 2.8.2 provides more information of converting units in different coordinates.

After the track is created, points are added to the first track by `circos.trackPoints()`. `circos.trackPoints()` simply adds points in all cells simultaneously. As further explained in Section 3.1, it can be replaced by putting `circos.text()` in `panel.fun`, however, `circos.trackPoints()` would be more convenient if only the points are needed to put in the cells. It is quite straightforward to understand that this function needs a categorical variable (`df$factors`), values on x direction and y direction (`df$x` and `df$y`).

Low-level functions such as `circos.text()` can also be used outside `panel.fun` as shown in above code. If so, `sector.index` and `track.index` need to be specified explicitly because the "current" sector and "current" track may not be what you want. If the graphics are directly added to the track which are most recently created, `track.index` can be ommitted because this track is just marked as the "current" track.

OK, now we add histograms to the second track. Here `circos.trackHist()` is a high- level function which means it creates a new track (as you can imagin `hist()` is also a high-level function). `bin.size` is explicitly set so that the bin size for histograms in all cells are the same and can be compared to each other. (See Figure 1.3)

```
bgcol = rep(c("#EFEFEF", "#CCCCCC"), 4)
circos.trackHist(df$factors, df$x, bin.size = 0.2, bg.col = bgcol, col = NA)
```

In the third track and in `panel.fun`, we randomly picked 10 data points in each cell, sort them and connect them with lines. In following code, when `factors`, `x` and `y` arguments are set in `circos.track()`, x values and y values are split by `df$factors` and corresponding subset of x and y values are sent to `panel.fun` through `panel.fun`'s `x` and `y` arguments. Thus, `x` an `y` in `panel.fun` are exactly the values in the "current" cell. (See Figure 1.4)

```
circos.track(factors = df$factors, x = df$x, y = df$y,
    panel.fun = function(x, y) {
```

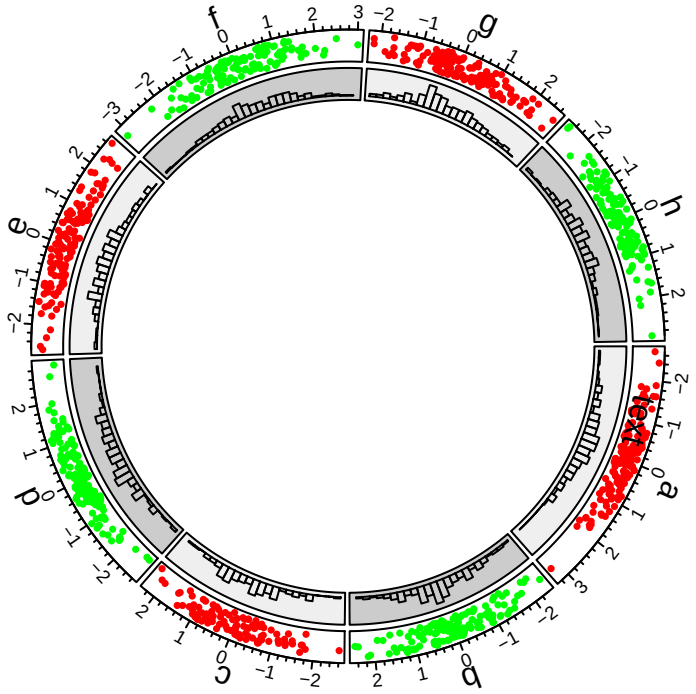


Figure 1.3: First example of circlize, add the second track.

```
        ind = sample(length(x), 10)
        x2 = x[ind]
        y2 = y[ind]
        od = order(x2)
        circos.lines(x2[od], y2[od])
})
```

Now we go back to the second track and update the cell in sector "d". This is done by `circos.updatePlotRegion()` or the short version `circos.update()`. The function erases graphics which have been added. `circos.update()` can not modify the `xlim` and `ylim` of the cell as well as other settings related to the position of the cell. `circos.update()` needs to explicitly specify the sector index and track index unless the "current" cell is what you want to update. After the calling of `circos.update()`, the "current" cell is redirected to the cell you just specified and you can use low-level graphic functions to add graphics directly into it. (See Figure 1.5)

```
circos.update(sector.index = "d", track.index = 2,
    bg.col = "#FF8080", bg.border = "black")
circos.points(x = -2:2, y = rep(0.5, 5), col = "white")
circos.text(CELL_META$xcenter, CELL_META$ycenter, "updated", col = "white")
```

Next we continue to create new tracks. Although we have gone back to the second track, when creating a new track, the new track is still created after the track which is most inside. In this new track, we add heatmaps by `circos.rect()`. Note here we haven't set the input data, while simply set `ylim` argument because heatmaps just fill the whole cell from the most left to right and from bottom to top. Also the exact value of `ylim` is not important and `x`, `y` in `panel.fun()` are not used (actually they are both `NULL`). (See Figure 1.6)

```
circos.track(ylim = c(0, 1), panel.fun = function(x, y) {
    xlim = CELL_META$xlim
    ylim = CELL_META$ylim
```

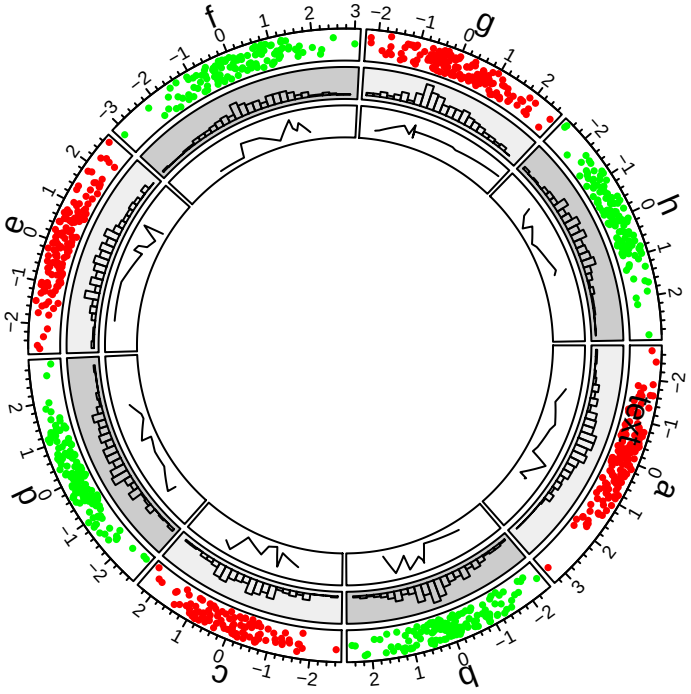


Figure 1.4: First example of circlize, add the third track.

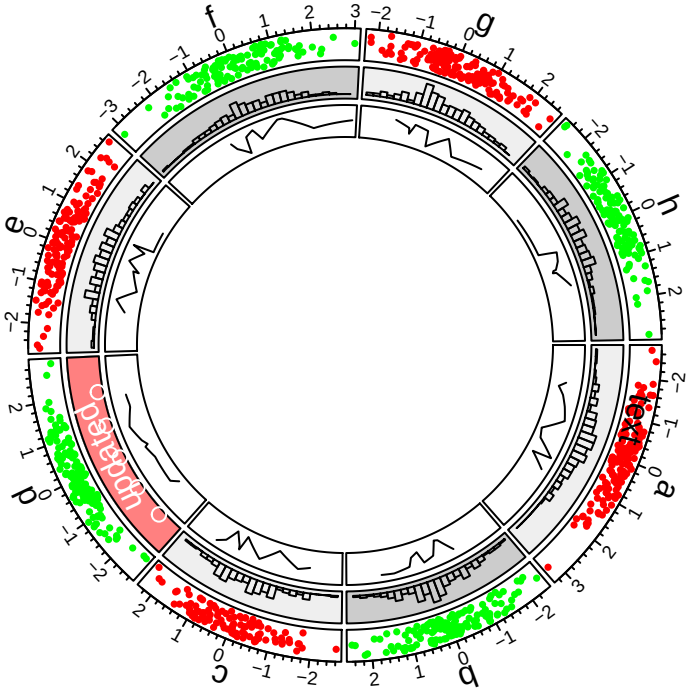


Figure 1.5: First example of circlize, update the second track.

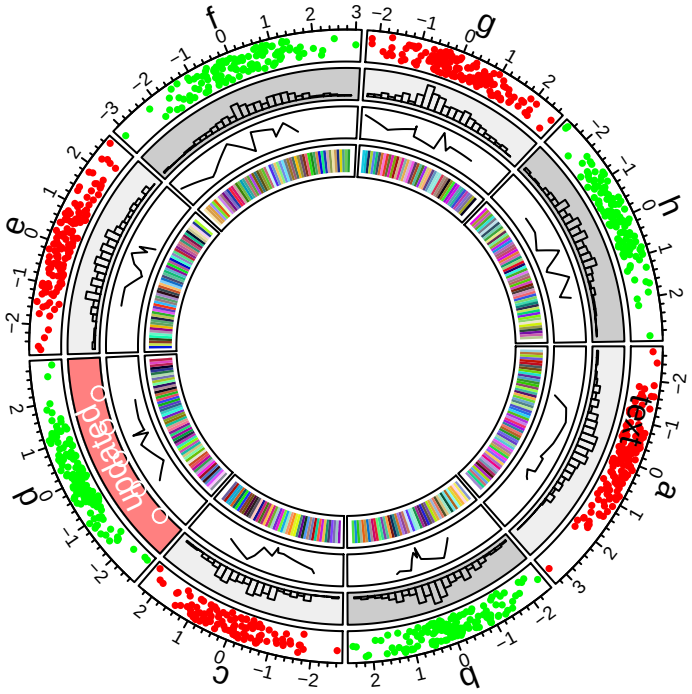


Figure 1.6: First example of circlize, add the fourth track.

```
    breaks = seq(xlim[1], xlim[2], by = 0.1)
    n_breaks = length(breaks)
    circos.rect(breaks[-n_breaks], rep(ylim[1], n_breaks - 1),
                breaks[-1], rep(ylim[2], n_breaks - 1),
                col = rand_color(n_breaks), border = NA)
})
```

In the most inside of the circle, links or ribbons are added. There can be links from single point to point, point to interval or interval to interval. Section 3.7 gives detailed usage of links. (See Figure 1.7)

```
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", border = "black", lwd = 2, lty = 2)
```

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

```
circos.clear()
```

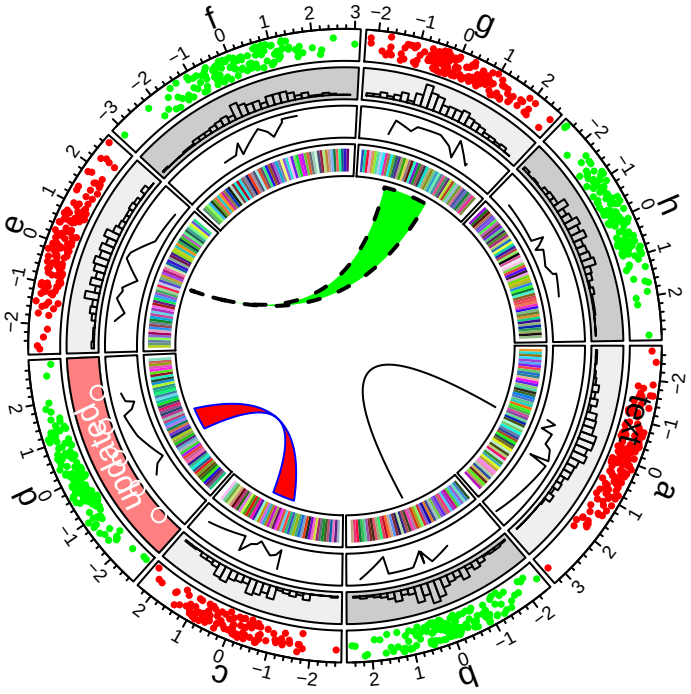


Figure 1.7: First example of circlize, add links.

Chapter 2

Circular layout

2.1 Coordinate transformation

To map graphics onto the circle, there exist transformations from several coordinate systems. First, there are **data coordinate systems** in which ranges for x-axes and y-axes are the ranges of original data. Second, there is a **polar coordinate system** in which these coordinates are mapped onto a circle. Finally, there is a **canvas coordinate system** in which graphics are really drawn on the graphical device (figure 2.1). Each cell has its own data coordinate and they are independent. **circlize** first transforms coordinates from data coordinate system to polar coordinate system and finally transforms into canvas coordinate system. 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 that cell. **circlize** knows which cell you are in and does all the transformations automatically.

The final canvas coordinate is in fact an ordinary coordinate in the base R graphic system with x range in `(-1, 1)` and y range in `(-1, 1)` by default. It should be noted that **the circular plot is always drawn inside the circle which has radius of 1 (which means it is always a unit circle), and from outside to inside**.

2.2 Rules for making the circular plot

The rule for making the circular plot is rather simple. It follows the sequence of `initialize layout -> create track -> add graphics -> create track -> add graphics - ... -> clear`. Graphics can be added at any time as long as the tracks are created. Details are shown in Figure 2.2 and as follows:

1. Initialize the layout using `circos.initialize()`. Since circular layout in fact visualizes data which is in categories, there must be at least a categorical variable. Ranges of x values on each category can be specified as a vector or the range itself. See Section 2.3.
2. Create plotting regions for the new track and add graphics. The new track is created just inside the previously created one. Only after the creation of the track can you add other graphics on it. There are three ways to add graphics in cells.
    - 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 and you need to subset the data by the categorical variable manually.
    - Use `circos.trackPoints()`, `circos.trackLines()`, … to add simple graphics through all cells simultaneously.

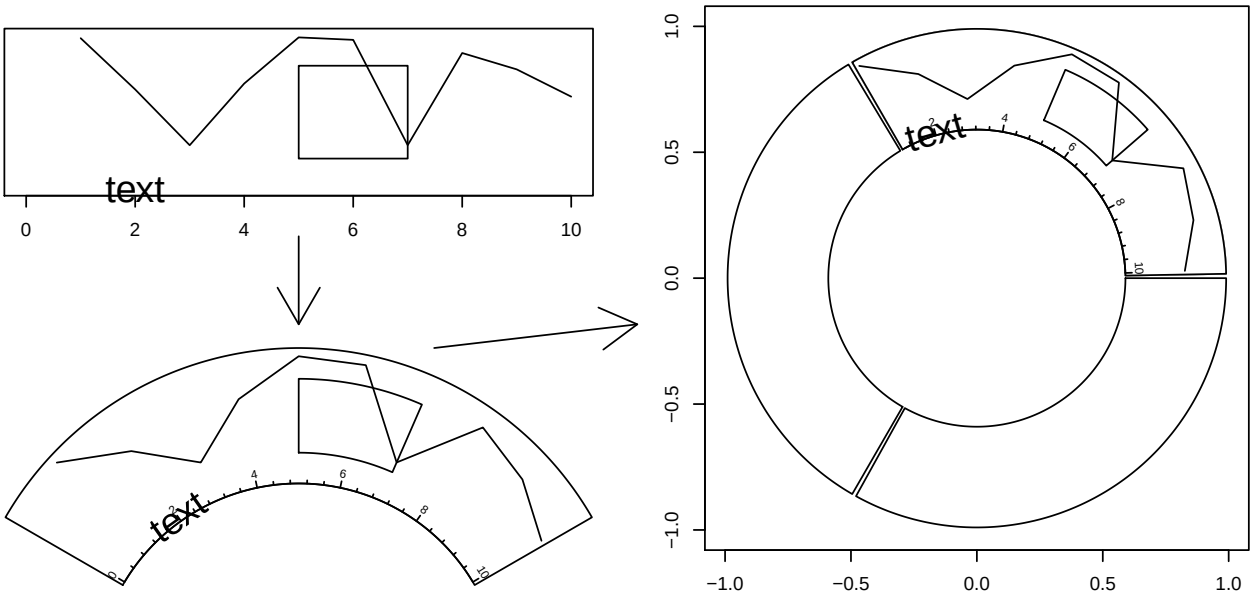


Figure 2.1: Transformation between different coordinates

- Use `panel.fun` argument in `circos.track()` to add 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 are in the current cell. This subset operation is applied automatically. This is the most recommended way. Section 2.7 gives detailed explanation of using `panel.fun` argument.

3. Repeat step 2 to add more tracks on the circle unless it reaches the center of the circle.
4. Call `circos.clear()` to clean up.

As mentioned above, there are three ways to add graphics on a track.

1. Create plotting regions for the whole track first and then add graphics by specifying `sector.index`. In the following pseudo code, `x1`, `y1` are data points in a given cell, which means you need to do data subsetting manually.

   In following code, `circos.points()` and `circos.lines()` are used separatedly from `circos.track()`, thus, the index for the sector needs to be explicitly specified by `sector.index` argument. There is also a `track.index` argument for both functions, however, the default value is the "current" track index and as the two functions are used just after `circos.track()`, the "current" track index is what the two functions expect and it can be ommited when calling the two functions.

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

2. Add graphics in a batch mode. In following code, `circos.trackPoints()` and `circos.trackLines()` need a categorical variable, a vector of x values and a vector of y values. X and y values will be split by the categorical variable and sent to corresponding cell to add the graphics. Internally, this is done by using `circos.points()` or `circos.lines()` in a `for` loop. This way to add graphics would be convenient if users only want to add a specific type of simple graphics (e.g. only points) to the track, but it is not recommended for making complex graphics.

   `circos.trackPoints()` and `circos.trackLines()` need a `track.index` to specify which track to add

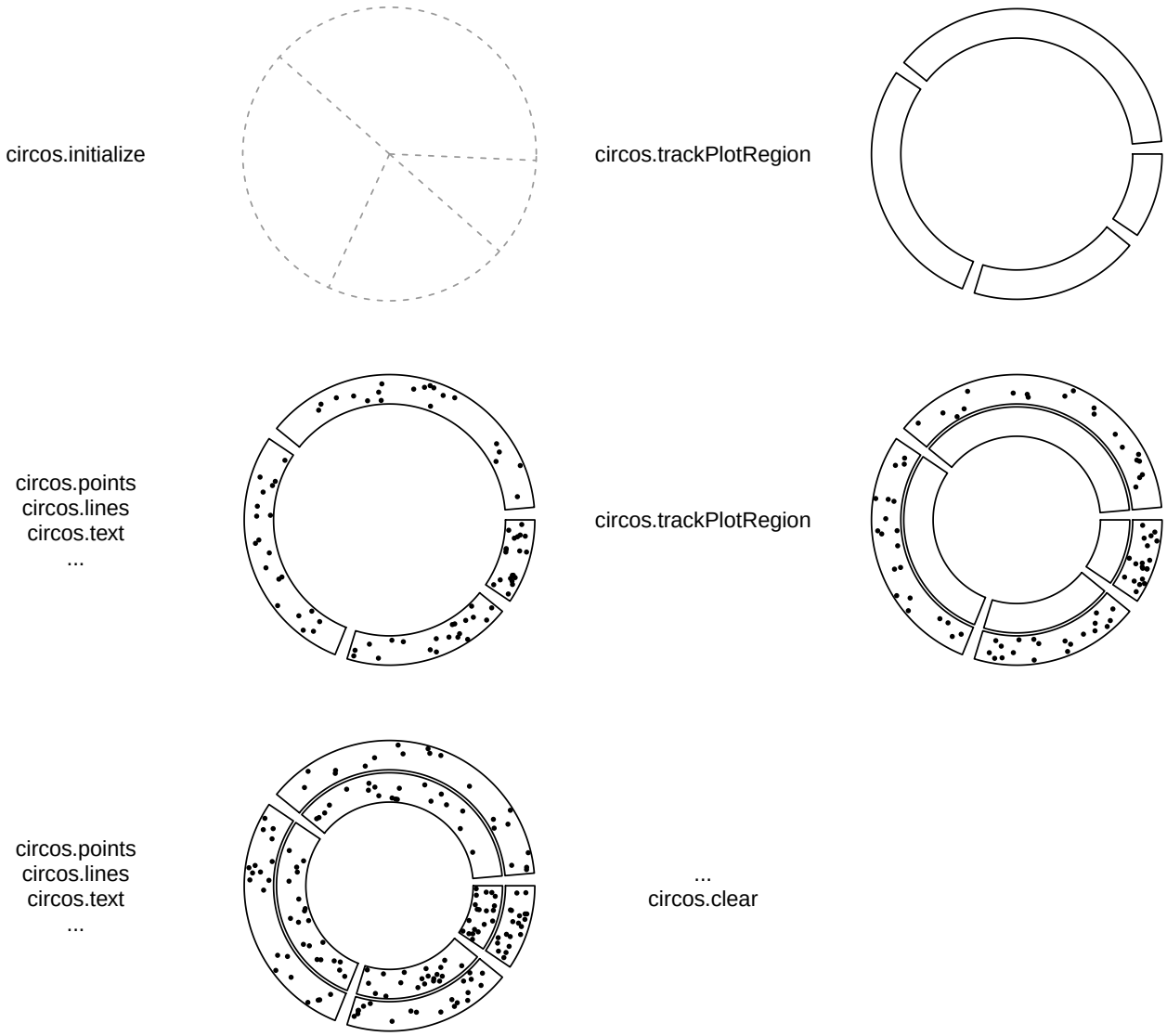


Figure 2.2: Order of drawing circular layout.

the graphics. Similarly, since these two are called just after `circos.track()`, the graphics are added in the newly created track right away.

```
circos.initialize(factors, xlim)
circos.track(factors, ylim)
circos.trackPoints(factors, x, y)
circos.trackLines(factors, x, y)
```

3. Use a panel function to add self-defined graphics as soon as the cell has been created. This is the way recommended and you can find most of the code in this book uses `panel.fun`. `circos.track()` creates cells one by one and after the creation of a cell, and `panel.fun` is executed on this cell immediately. In this case, the "current" sector and "current" track are marked to this cell that you can directly use low-level functions without specifying sector index and track index.

   If you look at following code, you will find the code inside `panel.fun` is as natural as using `points()` or `lines()` in the normal R graphic system. This is a way to help you think a cell is an "imaginary rectangular plotting region".

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

There are several internal variables keeping tracing of the current sector and track when applying `circos.track()` and `circos.update()`. Thus, although functions like `circos.points()`, `circos.lines()` need to specify the index of sector and track, they will take the current one 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 added in the most recently created or updated cell.

2.3 Sectors and tracks

A circular layout is composed of sectors and tracks. As illustrated in Figure 2.3, the red circle is one track and the blue represents one 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 this section, we introduce how to set data ranges on x and y directions in cells.

Sectors are first allocated on the circle by `circos.initialize()`. There must be a categorical variable (say `factors`) that on the circle, each sector corresponds to one category. The width of sectors (measured by degree) are proportional to the data range in sectors on x direction (or the circular direction). The data range can be specified as a numeric vector `x` which has same length as `factors`, then `x` is split by `factors` and data ranges are calculated for each sector internally.

Data ranges can also be specified directly by `xlim` argument. The valid value for `xilm` is a two-column matrix with same number of rows as number of sectors that each row in `xlim` corresponds to one sector. If `xlim` has row names which already cover sector names, row order of `xlim` is automatically adjusted. If `xlim` is a vector of length two, all sectors have the same x range.

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

After the initialization of the layout, you may not see anything drawn or only an empty graphical device is opened. That is because no track has been created yet, however, the layout has already been recorded internally.

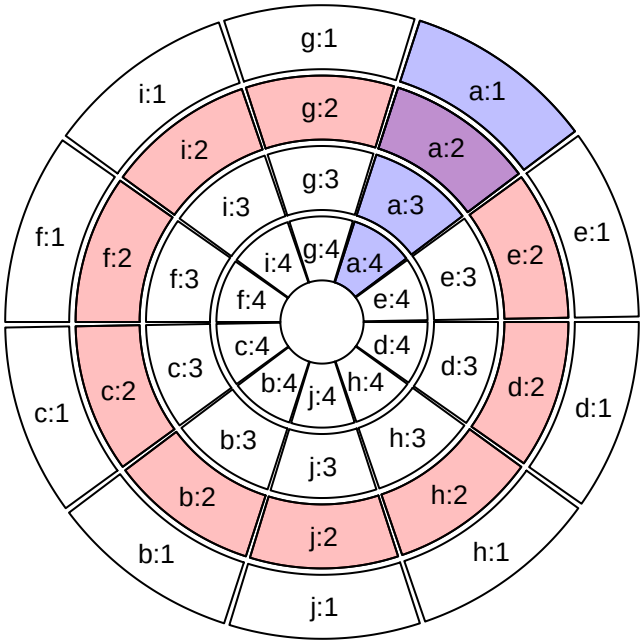


Figure 2.3: Sectors and tracks in circular layout.

In the initialization 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 input factor**. If the value for `factors` is not a factor, the order of sectors is `unique(factors)`. Thus, if you want to change the order of sectors, you can just change of the level of `factors` variable. The following code generates plots with different sector orders (Figure 2.4).

```
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))
```

In different tracks, cells in the same sector share the same data range on x-axes. Then, for each track, we only need to specify the data range on y direction (or the radical direction) for cells. Similar as `circos.initialize()`, `circos.track()` also receives either `y` or `ylim` argument to specify the range of y-values. Since all cells in a same track shares a same y range, `ylim` is just a vector of length two if it is specified.

`x` can also be specified in `circos.track()`, but it is only used to send to `panel.fun`. In Section 2.7, we will introduce how `x` and `y` are sent to each cell and how the graphics are added.

```
circos.track(factors, y = y)
circos.track(factors, ylim = c(0, 1))
circos.track(factors, x = x, y = y)
```

In the track creation step, since all sectors have already been allocated in the circle, if `factors` argument is not set, `circos.track()` 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 users only create cells for a subset of sectors in the track (not all sectors), in fact, cells in remaining unspecified sectors are created as well, but with no borders (pretending they are not created).

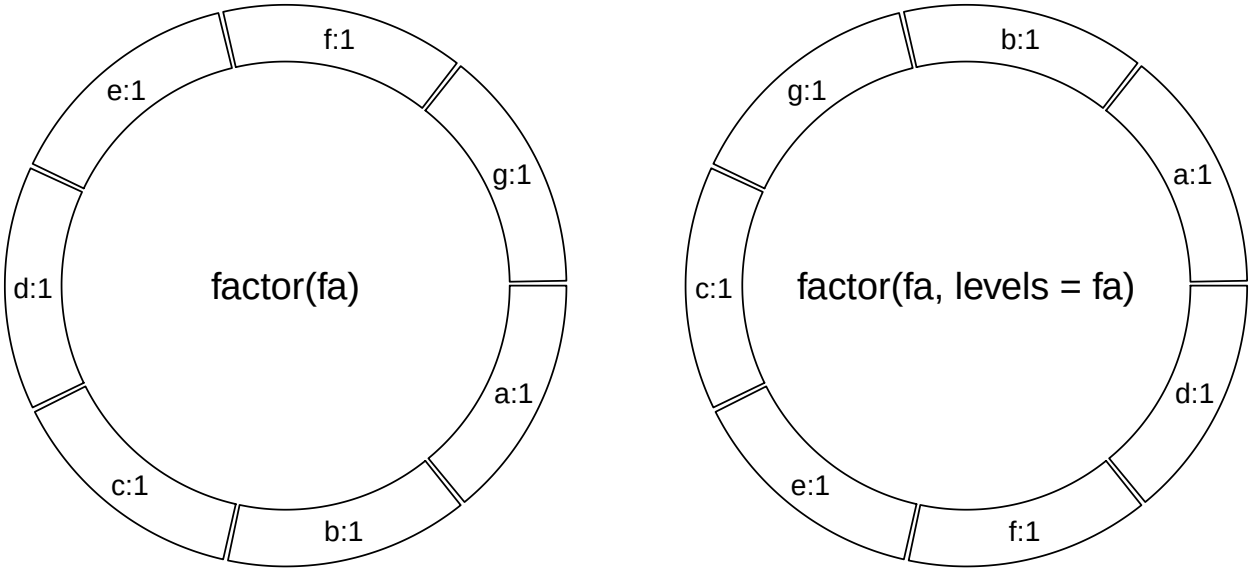


Figure 2.4: Different sector orders.

```
# assume `factors` only covers a subset of sectors
# You will only see cells that are covered in `factors` have borders
circos.track(factors, y = y)
# You will see all cells have borders
circos.track(ylim = ranges(y))
```

Cells are basic units in the circular plot and are independent from each other. After the creation of cells, they have self-contained meta values of x-lim and y-lim (data range measured in data coordinate). 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 normal rectangle region with its own coordinate.

2.4 Graphic parameters

Some basic parameters for the circular layout can be set by `circos.par()`. These parameters are listed as follows. Note some parameters can only be modified before the initialization of the circular layout.

- `start.degree`: The starting degree where the first sector is put. Note this degree is measured in the standard polar coordinate system which means it is always reverse clockwise. E.g. if it is set to 90, sectors start from the top center of the circle. See Figure 2.5.
- `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 number as sectors. **Note the first gap is after the first sector.** See Figure 2.5 and figure 2.6.
- `gap.after`: Same as `gap.degree`, but more understandable. Modifying values of `gap.after` will also modify `gap.degree` and vice versa.
- `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.after`, only bottom and top margin need to be set. The value for `track.margin` is the percentage to the radius of the unit circle. The value can also be set by `convert_height()` or the short version `uh()` function with absolute units. See figure 2.6.
- `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 parameter has four values, which control

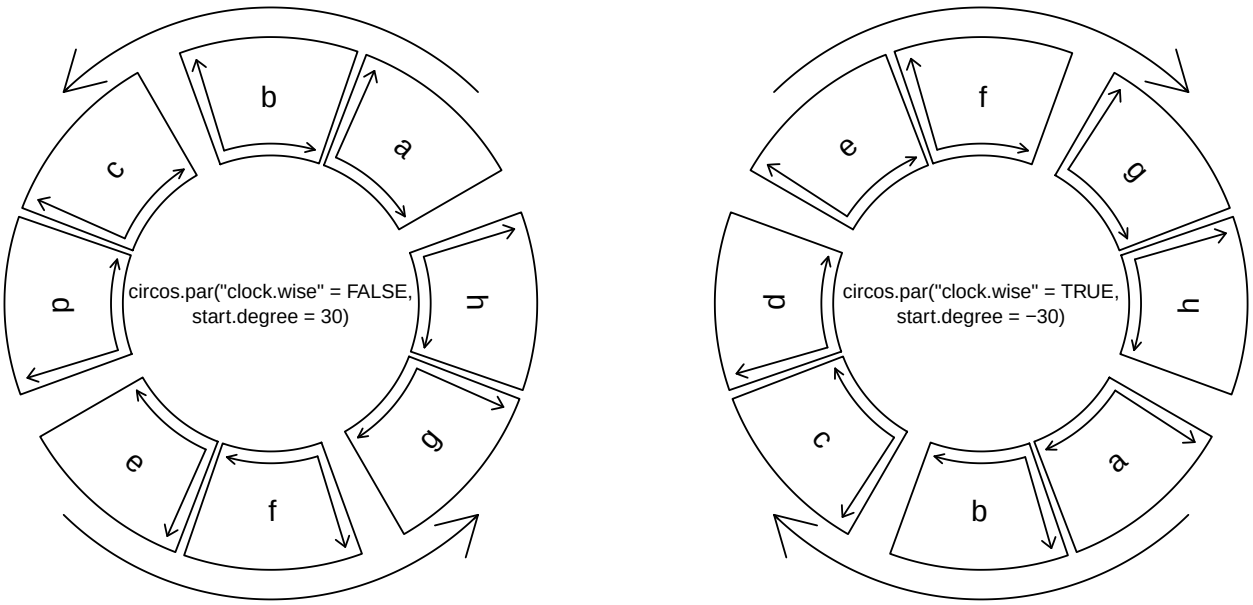


Figure 2.5: Sector directions.

the bottom, left, top and right padding respectively. The first and the third padding values are the percentages to the radius of the unit circle, and the second and fourth values are the degrees. The first and the third value can be set by `uh()` with absolute units. See figure 2.6.

- `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 (2π) divided by `unit.circle.segments`. More segments means better approximation for the curves, while generate larger file size if figures are in PDF format. See explanantion in Section 3.2.
- `track.height`: The default height of tracks. It is the percentage to the radius of the unit circle. The height includes the top and bottom cell paddings but not the margins. The value can be set by `uh()` with absolute units.
- `points.overflow.warning`: Since each cell is in fact not a real plotting region but only an ordinary rectangle (or more precisely, a circular rectangle), it does not remove points that are plotted outside of the region. So if some points (or lines, text) are out of the plotting region, by default, the package would continue drawing the points but with warning messages. However, in some circumstances, drawing something out of the plotting region is useful, such as adding some text annotations (like the first track in Figure 1.2). Set this value to `FALSE` to turn off the warnings.
- `canvas.xlim`: The ranges in the canvas coordinate in x direction. **circlize** is forced to put everything inside the unit circle, so `canvas.xlim` and `canvas.ylim` is `c(-1, 1)` by default. However, you can set it to a more broad interval if you want to leave more spaces out of the circle. By choosing proper `canvas.xlim` and `canvas.ylim`, actually you can customize the circle. E.g. setting `canvas.xlim` to `c(0, 1)` and `canvas.ylim` to `c(0, 1)` would only draw 1/4 of the circle.
- `canvas.ylim`: The ranges in the canvas coordinate in y direction.
- `clock.wise`: The order for drawing sectors. Default is `TRUE` which means clockwise (figure 2.5. **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 listed in following table.

`start.degree`	`0`
`gap.degree/gap.after`	`1`
`track.margin`	`c(0.01, 0.01)`

cell.padding	c(0.02, 1.00, 0.02, 1.00)
unit.circle.segments	500
track.height	0.2
points.overflow.warning	TRUE
canvas.xlim	c(-1, 1)
canvas.ylim	c(-1, 1)
clock.wise	TRUE

Parameters related to the allocation of sectors cannot be changed after the initialization of the circular layout. Thus, `start.degree`, `gap.degree`/`gap.after`, `canvas.xlim`, `canvas.ylim` and `clock.wise` can only be modified before `circos.initialize()`. The second and the fourth values of `cell.padding` (left and right paddings) can not be modified neither (or will be ignored).

Similar reason, since some of the parameters are defined before the initialization of the circular layout, after making each plot, you need to call `circos.clear()` to manually reset all the parameters.

2.5 Create plotting regions

As described above, only after creating the plotting region can you add low- level graphics on it. The minimal set of arguments for `circos.track()` is to set either `y` or `ylim` which assigns range of y values for this track. `circos.track()` creates tracks for all sectors although in some case only parts of them are visible.

If `factors` is not specified, all cells in the track will be created with the same settings. If `factors`, `x` and `y` are set, they need to be vectors with the same length. Proper values of `x` and `y` that correspond to current cell will be passed to `panel.fun` by subsetting `factors` internally. Section 2.7 explains the usage of `panel.fun`.

Graphic arguments such as `bg.border` and `bg.col` can either be a scalar or a vector. If it is a vector, the length must be equal to the number of sectors and the order corresponds to the order of sectors. Thus, 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 inside `panel.fun`. `get.cell.meta.data("cell.xlim")` and `get.cell.meta.data("cell.ylim")` give you dimensions of the plotting region and you can customize plot regions directly by e.g. `circos.rect(col = "#FF000040", border = 1)`.

`circos.track()` provides `track.margin` and `cell.padding` arguments that they only control track margins and cell paddings for the current track. Of course the second and fourth value in `cell.padding` are ignored.

2.6 Update plotting regions

`circos.track()` creates new tracks, however, if `track.index` argument is set to a track which already exists, `circos.track()` actually **re-creates** this track. In this case, coordinates on y directions can be re-defined, but settings related to the positions of the track such as the height of the track can not be modified.

```
circos.track(factors, ylim = c(0, 1), track.index = 1, ...)
```

For a single cell, `circos.update()` can be used to erase all graphics that have been already added in the cell. However, the data coordinate in the cell keeps unchanged.

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

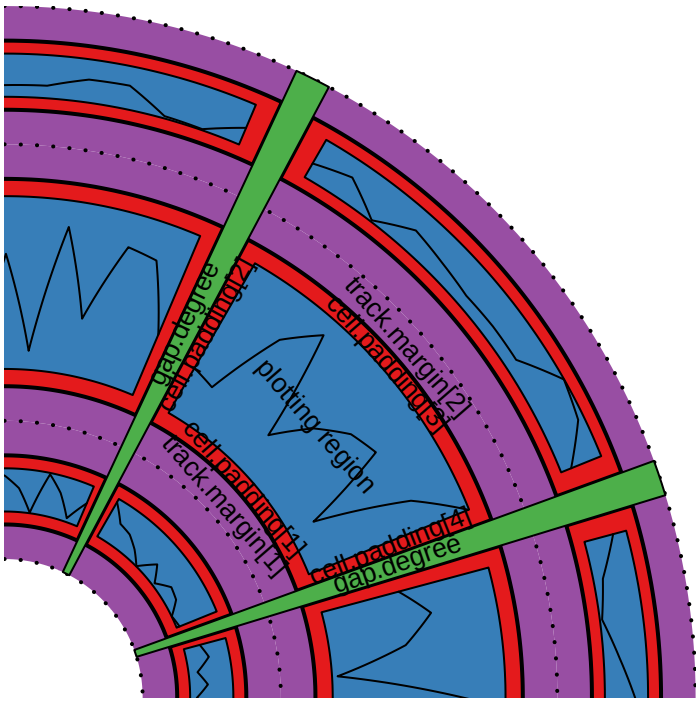


Figure 2.6: Regions in a cell.

2.7 panel.fun argument

`panel.fun` argument in `circos.track()` is extremely 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 `x` and `y` are automatically extracted from `x` and `y` in `circos.track()` according to the category defined in `factors`. In the following example, inside `panel.fun`, in sector `a`, the value of `x` is `1:3` and in sector `b`, value of `x` is `4:5`. If `x` or `y` in `circos.track()` 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.track(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.track()`, 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. When the cell is created, `panel.fun` is executed immediately. Without specifying `sector.index` and `track.index`, the 'current' ones are used and that's exactly what you need.

The advantage of `panel.fun` is that it makes you feel you are using graphic functions in the base graphic engine (You can see it is almost the same of using `circos.points(x, y)` and `points(x, y)`). It will be much easier for users to understand and customize new graphics.

Inside `panel.fun`, 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.

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

Information that can be extracted by `get.cell.meta.data()` are:

- `sector.index`: The name 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.
- `xcenter`: mean of `xlim`.
- `ycenter`: mean of `ylim`.
- `xrange`: defined as `xlim[2] - xlim[1]`.
- `yrange`: defined as `ylim[2] - ylim[1]`.
- `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`: Degree of right and left borders in the plotting region. The first element corresponds to the start point of values on x-axis and the second element corresponds to the end point of values on x-axis Since x-axis in data coordinate in cells are always clockwise, `xplot[1]` is larger than `xplot[2]`.
- `yplot`: Radius of bottom and top radius in the plotting region.
- `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 of the cell.
- `cell.padding`: Paddings of the cell.

Following example code uses `get.cell.meta.data()` to add sector index in the center of each cell.

```
circos.track(ylim = ylim, panel.fun = function(x, y) {
    sector.index = get.cell.meta.data("sector.index")
    xcenter = get.cell.meta.data("xcenter")
    ycenter = get.cell.meta.data("ycenter")
    circos.text(xcenter, ycenter, sector.index)
})
```

`get.cell.meta.data()` can also be used outside `panel.fun`, but you need to explictly specify `sector.index` and `track.index` arguments unless the current index is what you want.

There is a companion variable `CELL_META` which is identical to `get.cell.meta.data()` to get cell meta information, but easier and shorter to write. Actually, the value of `CELL_META` itself is meaningless, but e.g. `CELL_META$sector.index` is automatically redirected to `get.cell.meta.data("sector.index")`. Following code rewrites above example code with `CELL_META`.

```
circos.track(ylim = ylim, panel.fun = function(x, y) {
    circos.text(CELL_META$xcenter, CELL_META$ycenter,
        CELL_META$sector.index)
})
```

Please note `CELL_META` only extracts information for the "current" cell, thus, it is recommended to use only in `panel.fun`.

Nevertheless, if you have several lines of code which need to be executed out of `panel.fun`, you can flag the specified cell as the "current" cell by `set.current.cell()`, which can save you from typing too many `sector.index = ..., track.index = ...`. E.g. following code

```
circos.text(get.cell.meta.data("xcenter", sector.index, track.index),
            get.cell.meta.data("ycenter", sector.index, track.index),
```

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

can be simplified to:

```
set.current.cell(sector.index, track.index)
circos.text(get.cell.meta.data("xcenter"),
            get.cell.meta.data("ycenter"),
            get.cell.meta.data("sector.index"))
# or more simple
circos.text(CELL_META$xcenter, CELL_META$ycenter, CELL_META$sector.index)
```

2.8 Other utilities

2.8.1 `circlize()` and `reverse.circlize()`

circlize transform data points in several coordinate systems and it is basically done by the core function `circlize()`. The function transforms from data coordinate (coordinate in the cell) to the polar coordinate and its companion `reverse.circlize()` transforms from polar coordinate to a specified data coordinate. The default transformation is applied in the `current` cell.

```
factors = c("a", "b")
circos.initialize(factors, xlim = c(0, 1))
circos.track(ylim = c(0, 1))
# x = 0.5, y = 0.5 in sector a and track 1
circlize(0.5, 0.5, sector.index = "a", track.index = 1)
```

```
##      theta  rou
## [1,] 270.5 0.89
```

```
# theta = 90, rou = 0.9 in the polar coordinate
reverse.circlize(90, 0.9, sector.index = "a", track.index = 1)
```

```
##             x    y
## [1,] 1.519774 0.56
```

```
reverse.circlize(90, 0.9, sector.index = "b", track.index = 1)
```

```
##              x    y
## [1,] 0.5028249 0.56
```

You can see the results are different for two `reverse.circlize()` calls although it is the same points in the polar coordinate, because they are mapped to different cells.

`circlize()` and `reverse.circlize()` can be used to connect two circular plots if they are drawn on a same page. This provides a way to build more complex plots. Basically, the two circular plots share a same polar coordiante, then, the manipulation of `circlize->reverse.circlize->circlize` can transform coordinate for data points from the first circular plot to the second. In Chapter 12, we use this technique to combine two circular plots where one zooms subset of regions in the other one.

The transformation between polar coordinate and canvas coordinate is simple. **circlize** has a `circlize:::polar2Cartesian()` function but this function is not exported.

Following example (Figure 2.7) adds raster image to the circular plot. The raster image is added by `rasterImage()` which is applied in the canvas coordinate. Note how we change coordinate from data coordinate to canvas coordinate by using `circlize()` and `circlize:::polar2Cartesian()`.

```
library(yaml)
data = yaml.load_file("https://raw.githubusercontent.com/Templarian/slack-emoji-pokemon/master/pokemon.
set.seed(123)
pokemon_list = data$emojis[sample(length(data$emojis), 40)]
pokemon_name = sapply(pokemon_list, function(x) x$name)
pokemon_src = sapply(pokemon_list, function(x) x$src)

library(EBImage)
circos.par("points.overflow.warning" = FALSE)
circos.initialize(pokemon_name, xlim = c(0, 1))
circos.track(ylim = c(0, 1), panel.fun = function(x, y) {
    pos = circlize:::polar2Cartesian(circlize(CELL_META$xcenter, CELL_META$ycenter))
    image = EBImage::readImage(pokemon_src[CELL_META$sector.numeric.index])
    circos.text(CELL_META$xcenter, CELL_META$cell.ylim[1] - uy(2, "mm"),
        CELL_META$sector.index, facing = "clockwise", niceFacing = TRUE,
        adj = c(1, 0.5), cex = 0.6)
    rasterImage(image,
        xleft = pos[1, 1] - 0.05, ybottom = pos[1, 2] - 0.05,
        xright = pos[1, 1] + 0.05, ytop = pos[1, 2]+ 0.05)
}, bg.border = 1, track.height = 0.15)
```

```
circos.clear()
```

2.8.2 The convert functions

For the functions in **circlize** package, they needs arguments which are lengths measured either in the canvas coordinate or in the data coordinate. E.g. `track.height` argument in `circos.track()` corresponds to percent of radius of the unit circle. **circlize** package is built in the R base graphic system which is not straightforward to define a length with absolute units (e.g. a line of length 2 cm). To solve this problem, **circlize** provides three functions which convert absolute units to the canvas coordinate or the data coordinate accordingly.

`convert_length()` converts absolute units to the canvas coordinate. Since the aspect ratio for canvas coordinate is always set to 1, it doesn't matter whether to convert units in the x direction or in the y direction. The usage of `convert_length()` is straightforward, supported units are `mm`, `cm` and `inches`. If users want to convert a string height or width to the canvas coordinate, directly use `strheight()` or `strwidth()` functions.

```
convert_length(2, "mm")
```

Since `convert_length()` is mostly used to define heights on the radical direction, e.g. track height or height of track margins, the function has another name `convert_height()`, or the short name `uh()` (stands for *unit height*).

`convert_x()` and `convert_y()`, or the short version `ux()` and `uy()` (*unit x* and *unit y*), convert absolute units to the data coordinate. By default, the conversion is applied in the "current" cell, but it can still be used in other cells by specifying `sector.index` and `track.index` arguments. Since the width of the cell is not identical from the top to the bottom in the cell, for `convert_x()` or `ux()` function, the position on y direction where the convert is applied needs to be specified. By default it is at the middle point on y-axis.

Following plot (Figure 2.8) is an example of setting absolute units.

```
fa = letters[1:10]
circos.par(cell.padding = c(0, 0, 0, 0), track.margin = c(0, 0))
circos.initialize(fa, xlim = cbind(rep(0, 10), runif(10, 0.5, 1.5)))
```

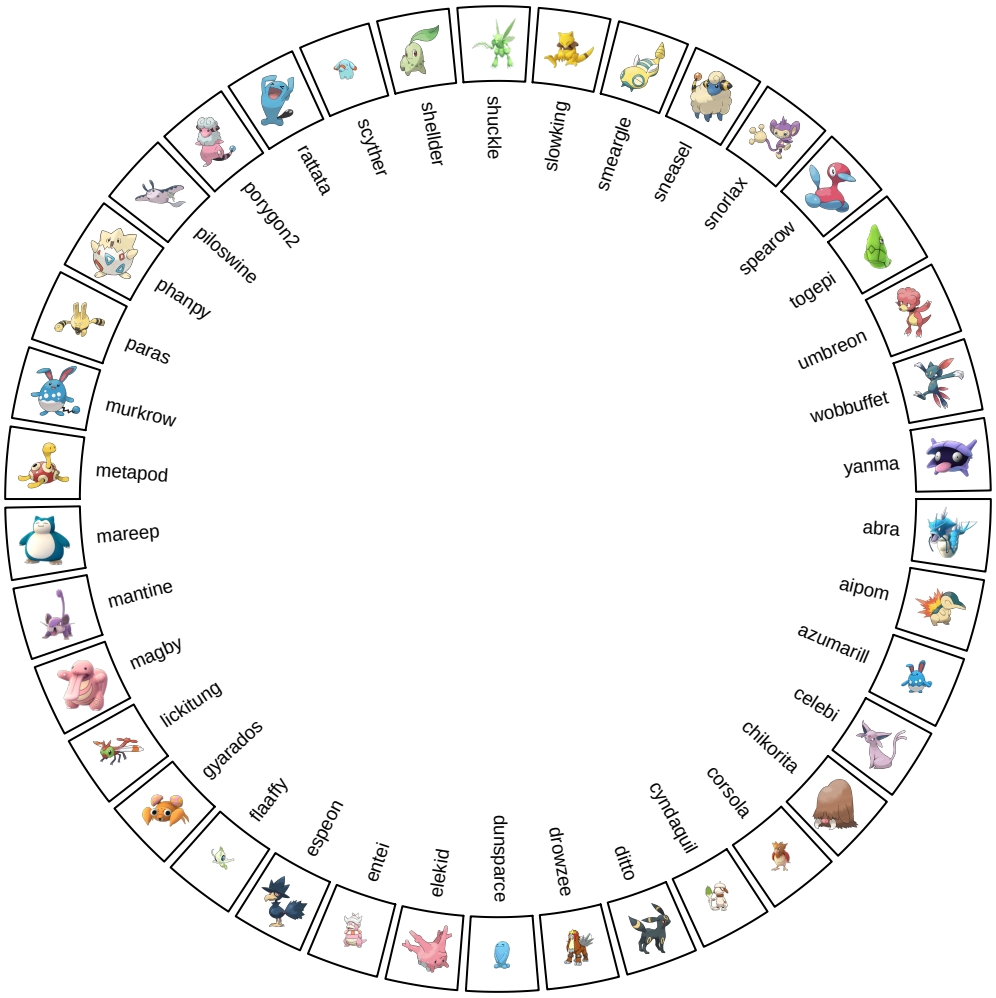


Figure 2.7: Add raster image to the circular plot.

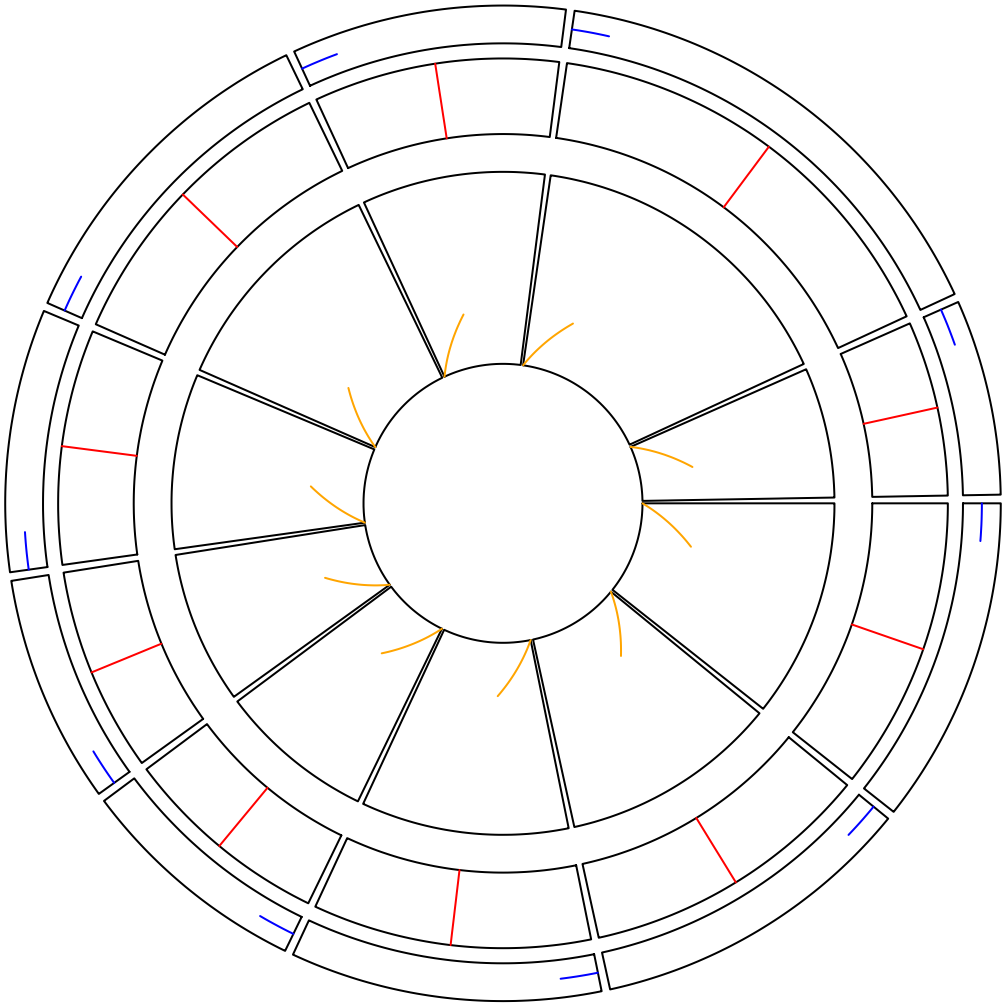


Figure 2.8: Setting absolute units

```
circos.track(ylim = c(0, 1), track.height = uh(5, "mm"),
    panel.fun = function(x, y) {
        circos.lines(c(0, 0 + ux(5, "mm")), c(0.5, 0.5), col = "blue")
    })
circos.track(ylim = c(0, 1), track.height = uh(1, "cm"),
    track.margin = c(0, uh(2, "mm")),
    panel.fun = function(x, y) {
        xcenter = get.cell.meta.data("xcenter")
        circos.lines(c(xcenter, xcenter), c(0, uy(1, "cm")), col = "red")
    })
circos.track(ylim = c(0, 1), track.height = uh(1, "inches"),
    track.margin = c(0, uh(5, "mm")),
    panel.fun = function(x, y) {
        line_length_on_x = ux(1*sqrt(2)/2, "cm")
        line_length_on_y = uy(1*sqrt(2)/2, "cm")
        circos.lines(c(0, line_length_on_x), c(0, line_length_on_y), col = "orange")
    })
```

```
circos.clear()
```

2.8.3 circos.info() and circos.clear()

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

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

```
## All your sectors:
## [1] "a" "b" "c"
##
## No track has been created
```

```
circos.track(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.xlim: [0.991453, 2.008547]
## cell.ylim: [-0.1, 1.1]
## xplot (degree): [0, 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
```

```
circos.clear()
```

It can also add labels to all cells by `circos.info(plot = TRUE)`.

You should always call `circos.clear()` at the end of every circular plot. There are several parameters for circular plot which can only be set before `circos.initialize()`, thus, before you draw the next circular plot, you need to reset all these parameters.

Chapter 3

Graphics

In this chapter, we will introduce low-level functions that add graphics to the circle. Usages of most of these functions are similar as normal graphic functions (e.g. `points()`, `lines()`). Combination use of these functions can generate very complex circular plots.

All low-level functions accept `sector.index` and `track.index` arguments which indicate which cell the graphics are added in. By default the graphics are added in the "current" sector and "current" track, so it is recommended to use them directly inside `panel.fun` function. However, they can also be used in other places with explicitly specifying sector and track index. Following code shows an example of using `ciros.points()`.

```
circos.track(..., panel.fun = function(x, y) {
    circos.points(x, y)
})
circos.points(x, y, sector.index, track.index)
```

In this chapter, we will also discuss how to customize links and how to highlight regions in the circle.

3.1 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)
```

There is a companion function `circos.trackPoints()` which adds points to all sectors in a same track simultaneously. The input of `circos.trackPoints()` must contain a vector of categorical factors, a vector of x values and a vector of y values. X values and y values are split by the categorical variable and corresponding subset of x and y values are internally sent to `circos.points()`. `circos.trackPoints()` adds points to the "current" track by default which is the most recently created track. Other tracks can also be selected by explictly setting `track.index` argument.

```
circos.track(...)
circos.trackPoints(fa, x, y)
```

`circos.trackPoints()` is simply implemented by `circos.points()` with a `for` loop. However, it is more recommended to directly use `circos.points()` and `panel.fun` which provides great more flexibility. Actually following code is identical to above code.

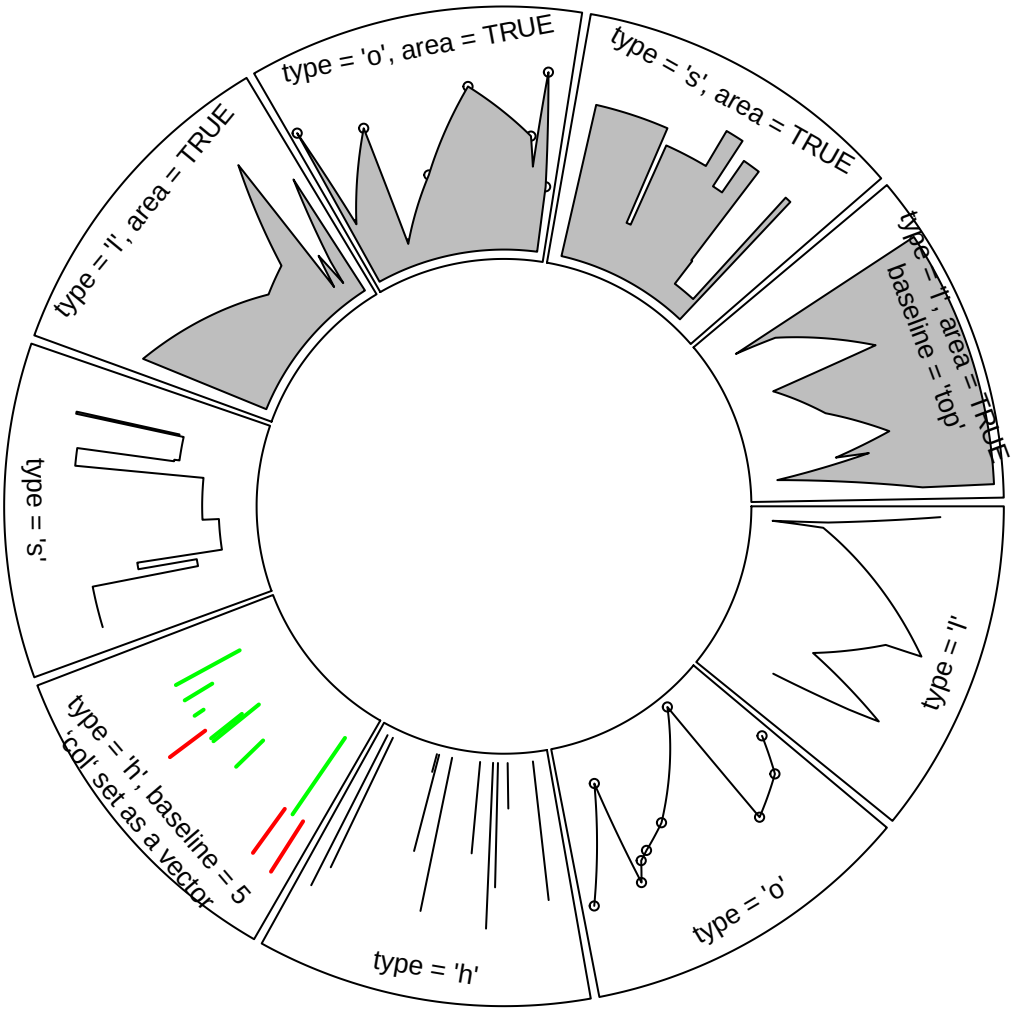


Figure 3.1: Line styles and areas supported in 'circos.lines()'

```
circos.track(fa, x, y, panel.fun = function(x, y) {
    circos.points(x, y)
})
```

Other low-level functions also have their companion `circos.track*()` function. The usage is same as `circos.trackPoints()` and they will not be further discussed in following sections.

3.2 Lines

Adding lines by `circos.lines()` is similar as `lines()` function. One additional feature is that the areas under or above the lines can be filled by specifing `area` argument to `TRUE`. Position of the baseline can be set to a pre-defined string of `bottom` or `top`, or a numeric value which is the position on y-axis. When `area` is set to `TRUE`, `col` controls the filled color and `border` controls the color for the borders.

`baseline` argument is also workable when `lty` is set to `"h"`. Note when `lty` is set to `"h"`, graphic parameters such as `col` can be set as a vector with same length as `x`. Figure 3.1 illustrates supported `lty` settings and `area`/`baseline` settings.

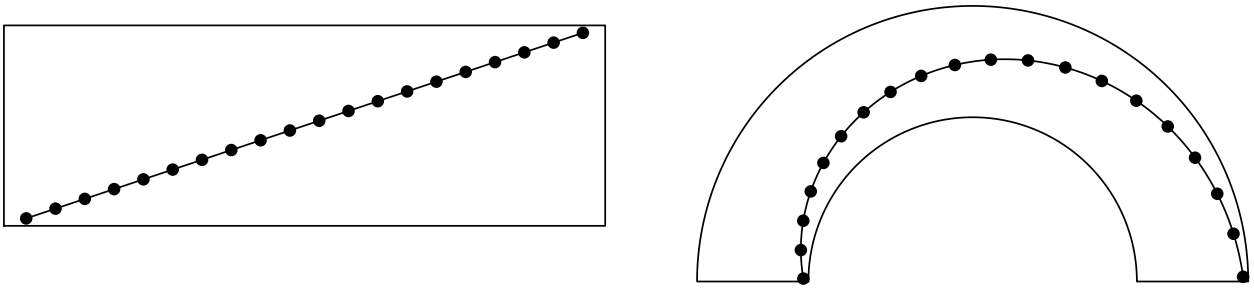


Figure 3.2: Transformation of straight lines into curves in the circle.

Straight lines are transformed to curves when mapping to the circular layout (Figure 3.2). Normally, curves are approximated by a series of segments of straight lines. With more and shorter segments, there is better approximation, but with larger size if the figures are generated into e.g. PDF files, especially for huge dataset. Default length of segments in **circlize** is a balance between the quality and size of the figure. You can set the length of the unit segment by `unit.circle.segments` option in `circos.par()`. The length of the segment is calculated as the length of the unit circle (2π) divided by `unit.circle.segments`. In some scenarios, actually you don't need to segment the lines such as radical lines, then you can set `straight` argument to `TRUE` to get rid of unnecessary segmentations.

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)
```

3.3 Segments

Line segments can be added by `circos.segments()` function. The usage is similar as `segments()`. Radical segments can be added by setting `straight` to `TRUE`.

```
circos.segments(x0, y0, x1, y1)
circos.segments(x0, y0, x1, y1, straight)
```

3.4 Text

Adding text by `circos.text()` is similar as `text()` function. Text is added on the plot for human reading, thus, when putting the text on the circle, the facing of text is very important. `circos.text()` supports seven facing options which are `inside`, `outside`, `clockwise`, `reverse.clockwise`, `downward`, `bending.inside` and `bending.outside`. Please note for `bending.inside` and `bending.outside`, currently, single line text is only supported. If you want to put bended text into two lines, you need to split text into two lines and add each line by `circos.text()` separately. The different facings are illustrated in figure 3.3.

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, niceFacing, adj, cex, col, font)
```

If, e.g., `facing` is set to `inside`, text which is on the bottom half of the circle is still facing to the top and hard to read. To make text more easy to read and not to hurt readers' neck too much, `circos.text()`

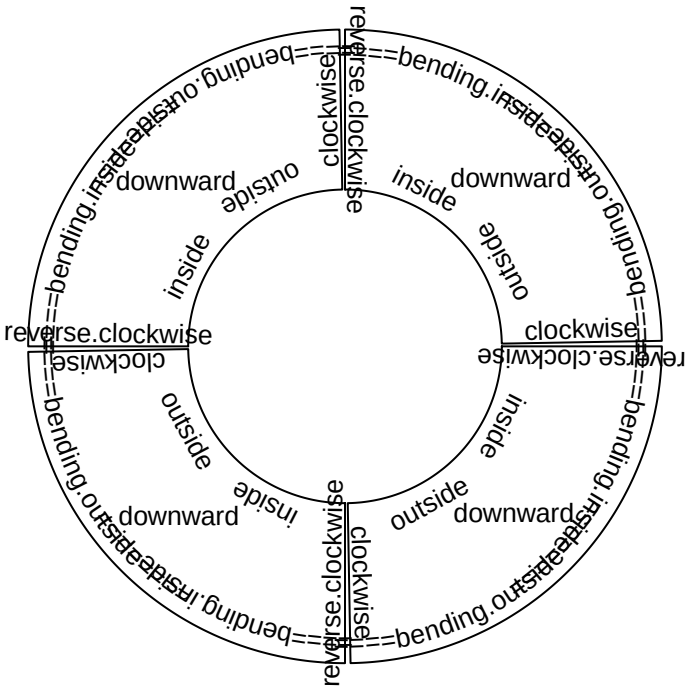


Figure 3.3: Text facings.

provides `niceFacing` option which automatically adjust text facing according to their positions in the circle. `niceFacing` only works for `facing` value of `inside`, `outside`, `clockwise`, `reverse.clockwise`, `bending.inside` and `bending.outside`.

When `niceFacing` is on, `adj` is also adjusted according to the corresponding facings. Figure 3.4 illustrates text positions under different settings of `adj` and `facing`. The red dots are the positions of the texts.

`adj` is internally passed to `text()`, thus, it actually adjusts text positions either horizontally or vertically (in the canvas coordinate). If the direction of the offset is circular, the offset value can be set as degrees that the position of the text is adjusted by wrapping the offset by `degree()`.

```
circos.text(x, y, labels, adj = c(0, degree(5)), facing = "clockwise")
```

As `circos.text()` is applied in the data coordiante, offset can be directly added to `x` or/and `y` as a value measured in the data coordinate. An absolute offset can be set by using `ux()` (in x direction) and `uy()` (in y direction).

```
circos.text(x + ux(2, "mm"), y + uy(2, "mm"), labels)
```

3.5 Rectangles and polygons

Theoretically, circular rectangles and polygons are all polygons. If you imagine the plotting region in a cell as Cartesian coordinate, then `circos.rect()` draws rectangles. In the circle, the up and bottom edge become two arcs. Note this function can be vectorized.

```
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)
```

`circos.polygon()` draws a polygon through a series of points in a cell. Please note the first data point must

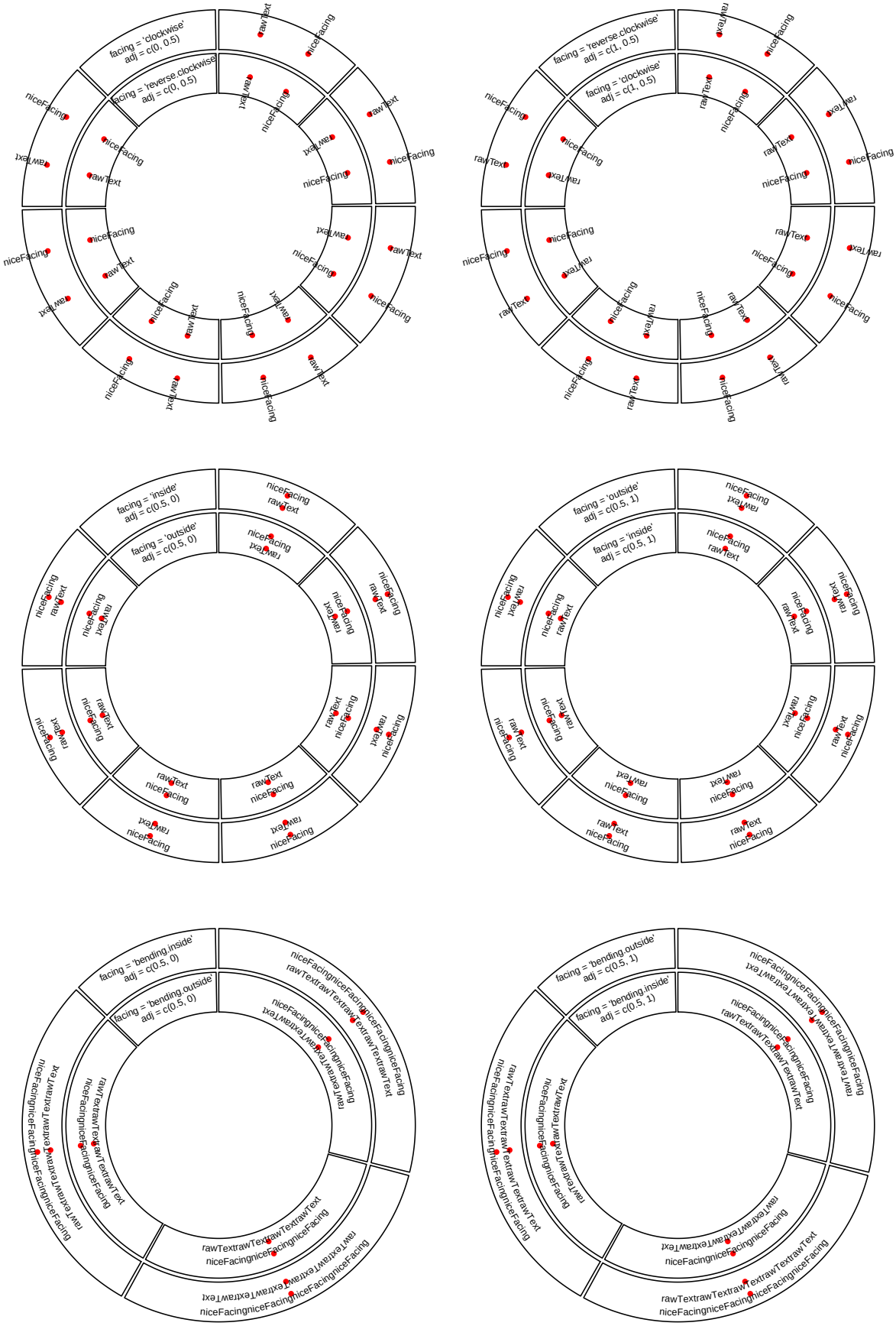


Figure 3.4: Human easy text facing.

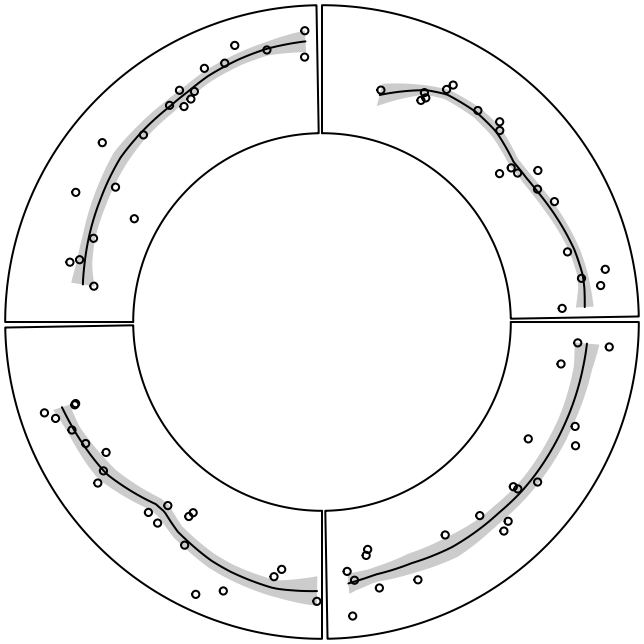


Figure 3.5: Area of standard deviation of the smoothed line.

overlap to the last data point.

```
circos.polygon(x, y)
circos.polygon(x, y, col, border, lty, lwd)
```

In Figure 3.5, the area of standard deviation of the smoothed line is drawn by `circos.polygon()`. Source code can be found in the **Examples** section of the `circos.polygon()` help page.

3.6 Axes

Mostly, we only draw x-axes on the circle. `circos.axis()` or `circos.xaxis()` privides options to customize x-axes which are on the circular direction. It supports basic functionalities as `axis()` such as defining the breaks and corresponding labels. Besides that, the function also supports to put x-axes to a specified position on y direction, to position the x-axes facing the center of the circle or outside of the circle, and to customize the axes ticks. The `at` and `labels` arguments can be set to a long vector that the parts which exceed the maximal value in the corresponding cell are removed automatically. The facing of labels text can be optimized by `labels.niceFacing` (by default it is `TRUE`).

Figure 3.6 illustrates different settings of x-axes. The explanations are as follows:

- a: Major ticks are calculated automatically, other settings are defaults.
- b: Ticks are pointing to inside of the circle, facing of tick labels is set to `outside`.
- c: Position of x-axis is `bottom` in the cell.
- d: Ticks are pointing to the inside of the circle, facing of tick labels is set to `reverse.clockwise`.
- e: manually set major ticks and also set the position of x-axis.
- f: replace numeric labels to characters, with no minor ticks.
- g: No ticks for both major and minor, facing of tick labels is set to `reverse.clockwise`.
- h: Number of minor ticks between two major ticks is set to 2. Length of ticks is longer. Facing of tick labels is set to `clockwise`.

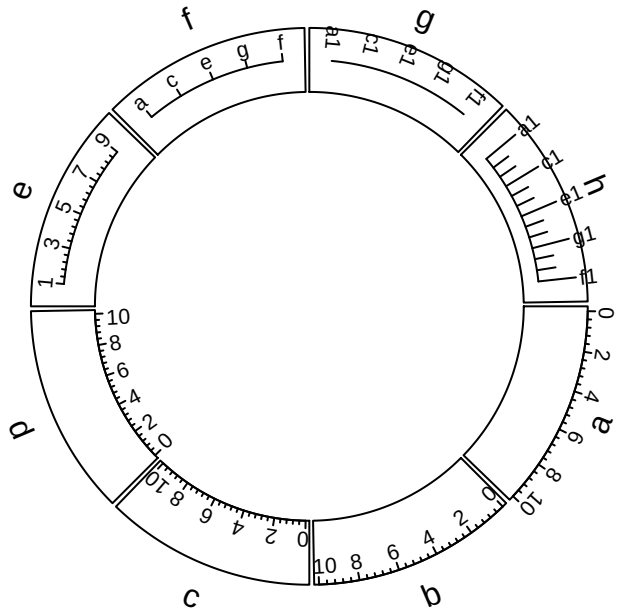


Figure 3.6: X-axes

As you may notice in the above figure, when the first and last axis labels exceed data ranges on x-axis in the corresponding cell, their positions are automatically adjusted to be shifted inwards in the cell.

Possible usage of `circos.axis()` is as follows. Note `h` can be `bottom`, `top` or a numeric value.

```
circos.axis(h)
circos.axis(h, sector.index, track.index)
circos.axis(h, major.at, labels, major.tick, direction)
circos.axis(h, major.at, labels, major.tick, labels.font, labels.cex,
            labels.facing, labels.niceFacing)
circos.axis(h, major.at, labels, major.tick, minor.ticks,
            major.tick.length, lwd)
```

Y-axis is also supported by `circos.yaxis()`. The usage is similar as `circos.axis()` One thing that needs to be note is users need to manually adjust `gap.degree` in `circos.par()` to make sure there are enough spaces for y-axes. (Figure 3.7)

```
circos.yaxis(side)
circos.yaxis(at, labels, sector.index, track.index)
```

3.7 Links

Links or ribbons are important part for the circular visualization. They are used to represent relations or interactions between sectors. In **circlize**, `circos.link()` draws links between single points and intervals. There are four mandatory arguments which are index for the first sector, positions on the first sector, index for the second sector and positions on the second sector. If the positions on the two sectors are all single points, the link represents as a line. If the positions on the two sectors are intervals, the link represents as a robbon (Figure 3.8). Possible usage for `circos.link()` is as follows.

```
circos.link(sector.index1, 0, sector.index2, 0)
circos.link(sector.index1, c(0, 1), sector.index2, 0)
```

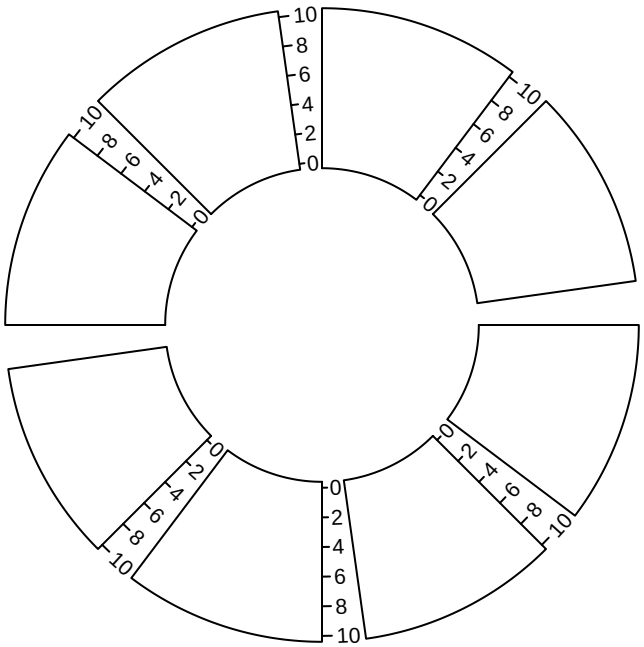


Figure 3.7: Y-axes

```
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 link end is controlled by `rou`. By default, it is the bottom of the most inside track and normally, you don't need to care about this setting. The two ends of the link are located in a same circle by default. The positions of two ends can be adjusted with different values for `rou1` and `rou2` arguments. See Figure 3.9.

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

The height of the link is controlled by `h` argument. In most cases, you don't need to care about the value of `h` because they are internally calculated based on the width of each link. However, when the link represents as a ribbon (i.e. link from point to interval or from interval to interval), It can not always ensure that one border is always below or above the other, which means, in some extreme cases, the two borders are intersected and the link would be messed up. 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 3.10).

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

When there are many links, the height of all links can be systematically adjusted by `h.ratio` (Figure 3.11). The value is between 0 and 1.

The border of link (if it is a ribbon) or the link itself (if it is a line) is in fact a quadratic Bezier curve, thus you can control the shape of the link by `w` and `w2` (`w2` controls the shape of bottom border). See Figure 3.12 for examples. 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)
```

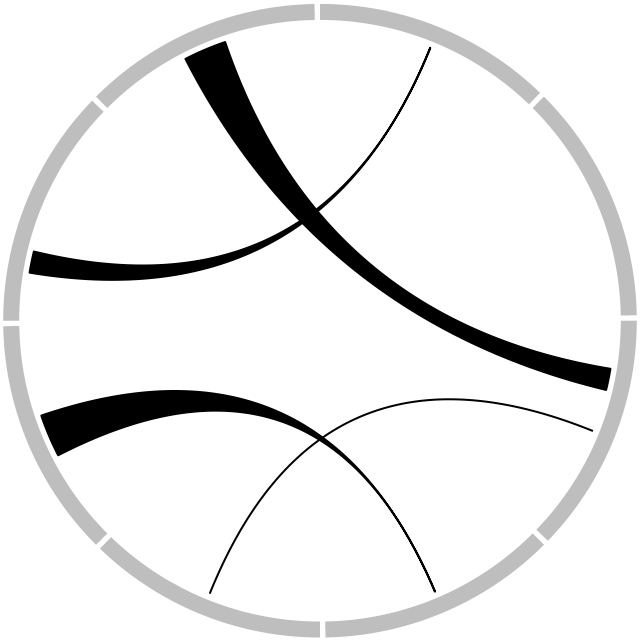


Figure 3.8: Different types of links.

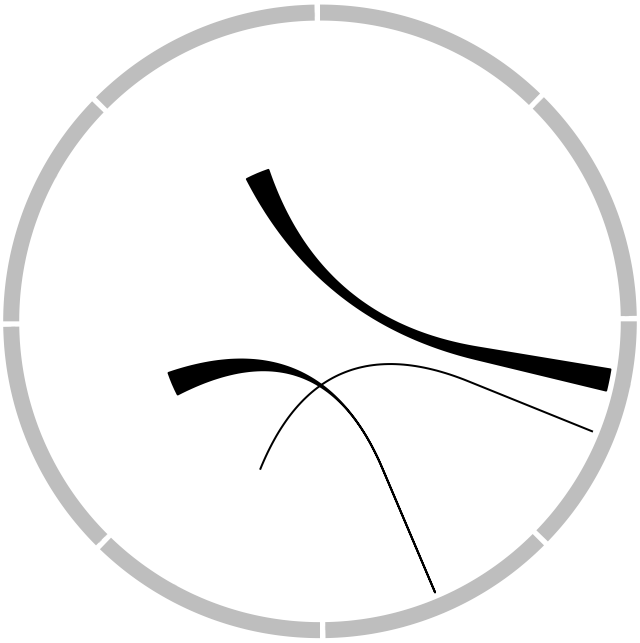


Figure 3.9: Positions of link ends.

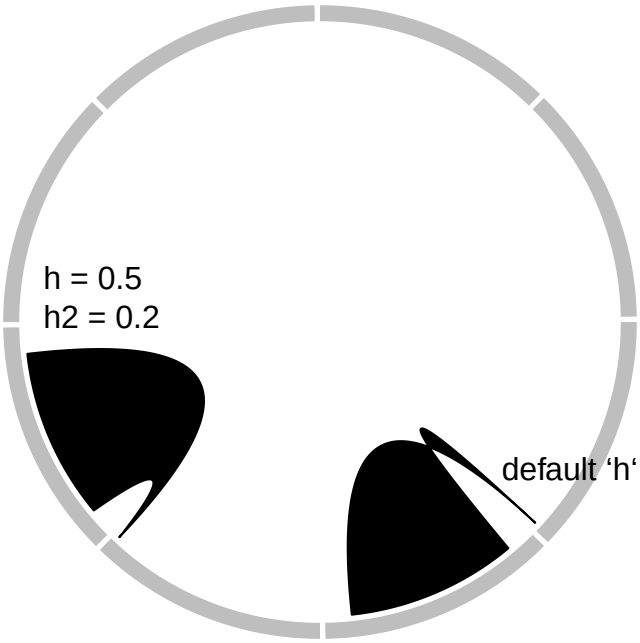


Figure 3.10: Adjust link heights.

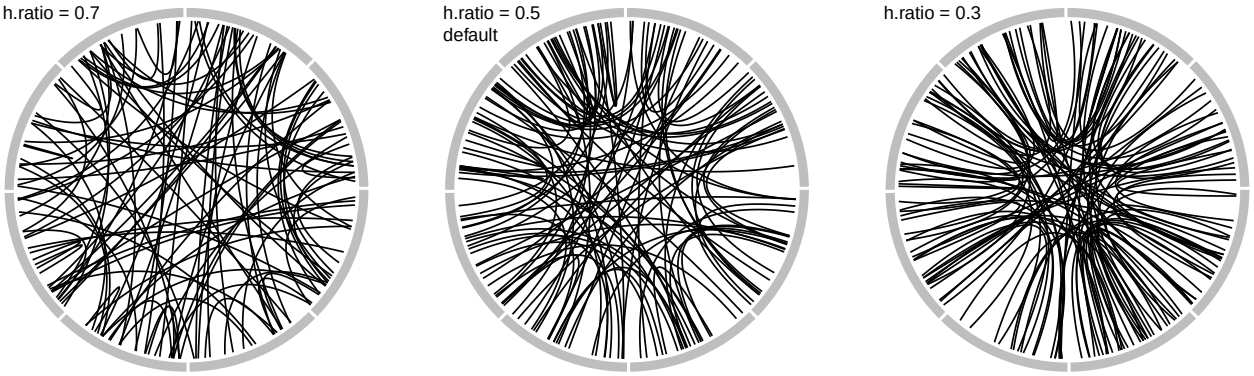


Figure 3.11: Adjust link heights by ‘h.ratio‘.

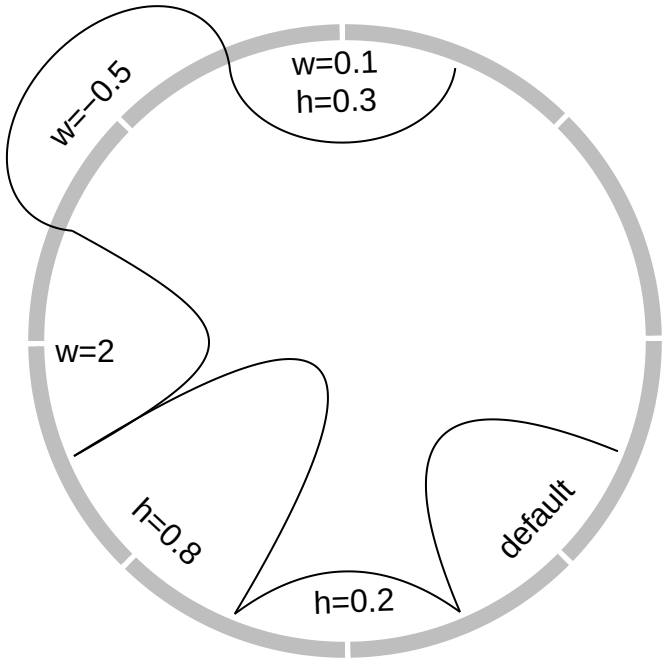


Figure 3.12: Different link shapes.

When the links represent as ribbons and the two ends overlap, the links will be de-generated as a 'hill' (Figure 3.13).

Links can have arrows to represent the directions. The `directional` argument controls how to add arrows. A value of `0` means there is no direction, `1` means the direction is from end 1 to end 2, `-1` means the direction is from end 2 to end 1, and `2` means bi-direction. If the link represents as a ribbon, a line with arrow will be added in the center of the link to represent directions. See Figure 3.14.

Type of arrows is controlled by `arr.type` argument and it is actually passed to `Arrowhead()` defined in **shape** package. Besides the arrow types supported in **shape** package, there is an additional arrow type `big.arrow` which turns the robbon into a big arrow (Figure 3.14).

Unequal height of the link ends can also represent directions which we will discuss more with the `chordDiagram()` function.

```
circos.link(sector.index1, 0, sector.index2, 0, directional = 1)
circos.link(sector.index1, c(0, 1), sector.index2, c(0, 1), directional = -1)
```

3.8 Highlight sectors and tracks

`draw.sector()` draws sectors, rings or their parts. This function is useful if you want to highlight some parts of your circular plot. it needs arguments of the position of circle center (by default `c(0, 0)`), the start degree and the end degree for sectors, and radius for two edges (or one edge) which are up or bottom borders. `draw.sector()` is independent from the circular plot.

Possible usage of `draw.sector()` is as follows.

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

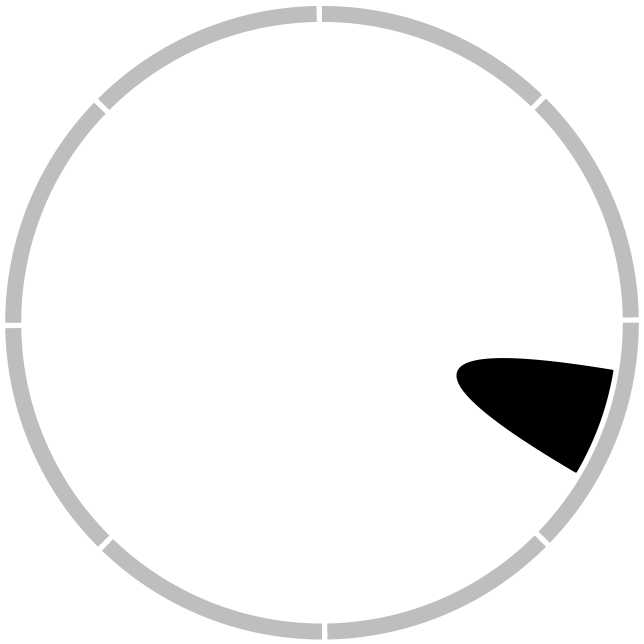


Figure 3.13: Link as a hill.

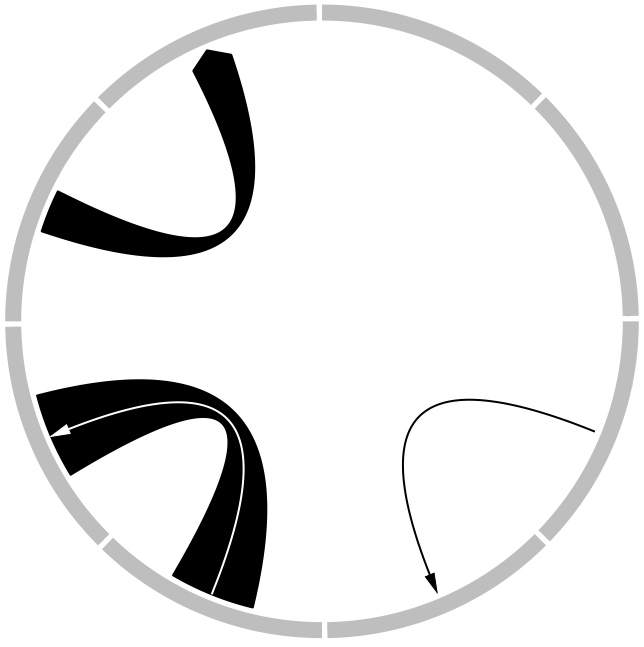


Figure 3.14: Link with arrows.

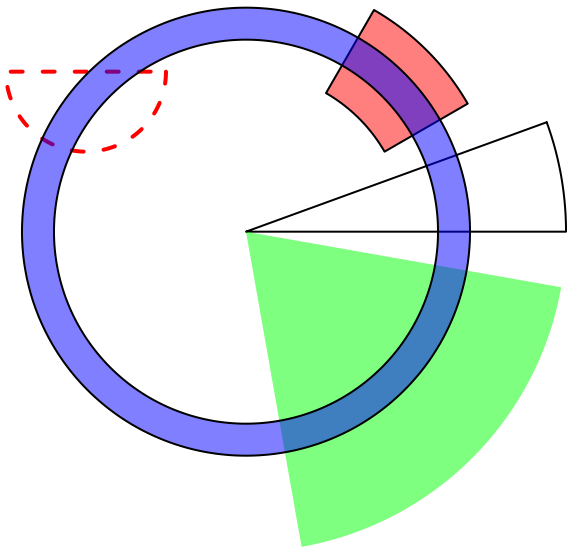


Figure 3.15: General usage of 'draw.sector()'.

Directions from `start.degree` and `end.degree` is important for drawing sectors. By default, it is clock wise.

```
draw.sector(start.degree, end.degree, clock.wise = FALSE)
```

Following code shows examples of `draw.sector()` (Figure 3.15).

```
par(mar = c(1, 1, 1, 1))
plot(c(-1, 1), c(-1, 1), type = "n", axes = FALSE, ann = FALSE, asp = 1)
draw.sector(20, 0)
draw.sector(30, 60, rou1 = 0.8, rou2 = 0.5, clock.wise = FALSE, col = "#FF000080")
draw.sector(350, 1000, col = "#00FF0080", border = NA)
draw.sector(0, 180, rou1 = 0.25, center = c(-0.5, 0.5), border = 2, lwd = 2, lty = 2)
draw.sector(0, 360, rou1 = 0.7, rou2 = 0.6, col = "#0000FF80")
```

In order to highlight cells in the circular plot, we can use `get.cell.meta.data()` to get the information of positions of cells. 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 can be obtained through `cell.top.radius` and `cell.bottom.radius`. Following code shows several examples to highlight sectors and tracks.

First we create a circular plot with eight sectors and three tracks.

```
factors = letters[1:8]
circos.initialize(factors, xlim = c(0, 1))
for(i in 1:3) {
    circos.track(ylim = c(0, 1))
}
circos.info(plot = TRUE)
```

If we want to highlight sector a (Figure 3.16):

```
draw.sector(get.cell.meta.data("cell.start.degree", sector.index = "a"),
            get.cell.meta.data("cell.end.degree", sector.index = "a"),
            rou1 = get.cell.meta.data("cell.top.radius", track.index = 1),
            col = "#FF000040")
```

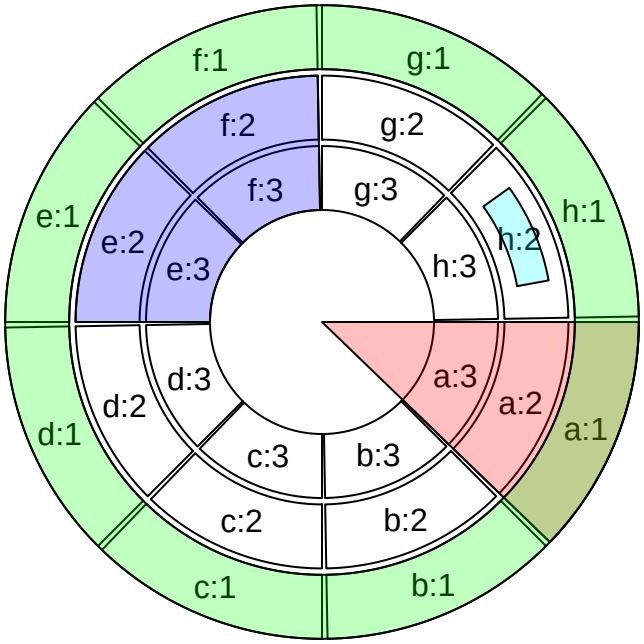


Figure 3.16: Highlight sectors and tracks.

If we want to highlight track 1 (Figure 3.16):

```
draw.sector(0, 360,
    rou1 = get.cell.meta.data("cell.top.radius", track.index = 1),
    rou2 = get.cell.meta.data("cell.bottom.radius", track.index = 1),
    col = "#00FF0040")
```

If we want to highlight track 2 and 3 in sector e and f (Figure 3.16):

```
draw.sector(get.cell.meta.data("cell.start.degree", sector.index = "e"),
            get.cell.meta.data("cell.end.degree", sector.index = "f"),
            rou1 = get.cell.meta.data("cell.top.radius", track.index = 2),
            rou2 = get.cell.meta.data("cell.bottom.radius", track.index = 3),
            col = "#0000FF40")
```

If we want to highlight specific regions such as a small region inside cell `h:2`, we can use `circlize()` to calculate the positions in the polar coordinate. But always keep in mind that x-axis in the cell are always clock wise. See Figure 3.16.

```
pos = circlize(c(0.2, 0.8), c(0.2, 0.8), sector.index = "h", track.index = 2)
draw.sector(pos[1, "theta"], pos[2, "theta"], pos[1, "rou"], pos[2, "rou"],
    clock.wise = TRUE, col = "#00FFFF40")
circos.clear()
```

If the purpose is to simply highlight complete cells, there is a helper function `highlight.sector()` for which you only need to specify index for sectors and tracks that you want to to highlight. Paddings of the highligted regions can be set by `padding` argument which should contain four values representing ratios of the width or height of the highlighted region (Figure 3.17).

One advantage of `highlight.sector()` is that it supports to add text in the highlighted regions. By default, the text is drawn at that center of the highlighted region. The position on the radical direction can be set by `text.vjust` argument either by a numeric value or a string in form of `"2 inches"`` or“-1.2cm”.

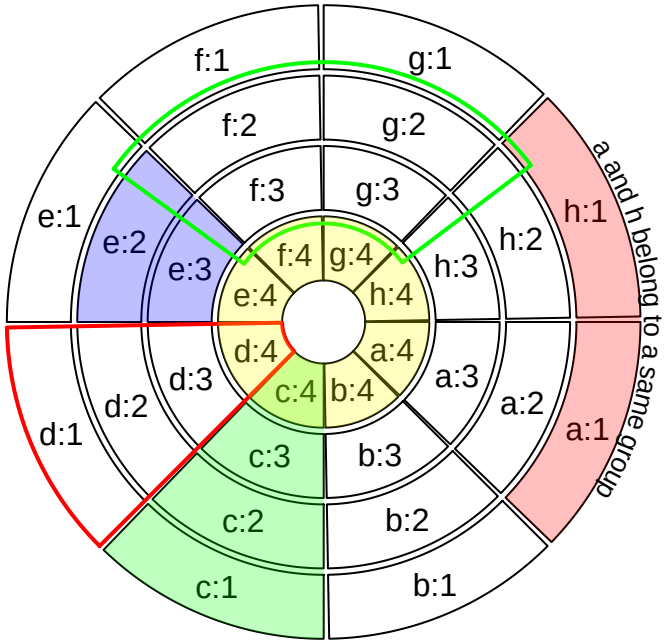


Figure 3.17: Highlight sectors.

```
factors = letters[1:8]
circos.initialize(factors, xlim = c(0, 1))
for(i in 1:4) {
    circos.track(ylim = c(0, 1))
}
circos.info(plot = TRUE)

highlight.sector(c("a", "h"), track.index = 1, text = "a and h belong to a same group",
    facing = "bending.inside", niceFacing = TRUE, text.vjust = "6mm", cex = 0.8)
highlight.sector("c", col = "#00FF0040")
highlight.sector("d", col = NA, border = "red", lwd = 2)
highlight.sector("e", col = "#0000FF40", track.index = c(2, 3))
highlight.sector(c("f", "g"), col = NA, border = "green",
    lwd = 2, track.index = c(2, 3), padding = c(0.1, 0.1, 0.1, 0.1))
highlight.sector(factors, col = "#FFFF0040", track.index = 4)
```

```
circos.clear()
```

3.9 Work together with the base graphic system

circlize is built on the base R graphic system, then, of course the base graphic functions can be used in combination with circlize functions. On the other hand, `circlize()` converts data points from the data coordinates to the canvas coordinates where the base graphic function can be directly applied.

Normally, the base functions such as `title()`, `text()`, `legend()` can be used to add extra information on the plot (Figure 3.18).

Sometimes, when the text or other graphics are far from the circle, you may set `par(xpd = NA)` so that the plotting is not clipped.

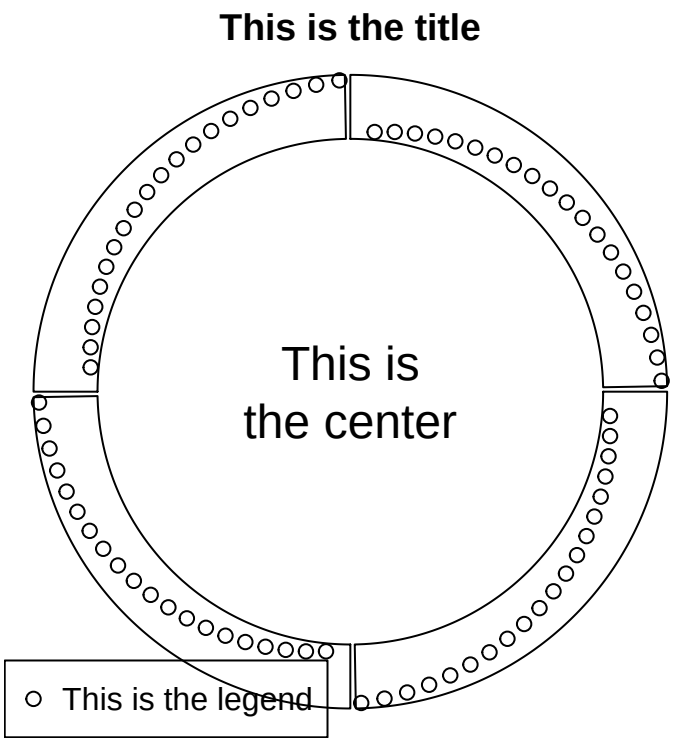


Figure 3.18: Work with base graphic functions.

```
factors = letters[1:4]
circos.initialize(factors = factors, xlim = c(0, 1))
circos.track(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")
title("This is the title")
```

```
circos.clear()
```

Chapter 4

Legends

circlize provides complete freedom for users to design their own graphics by implementing the self-defined function `panel.fun`. However one drawback arises that **circlize** is completely blind to users' data so that one important thing is missing for the visualization which is the legend.

Although legends cannot be automatically generated by **circlize**, by using functionality from other R packages, it is just a few more lines to really implement it. Here I will demonstrate how to customize legends and arrange to the circular plot.

As an example, a circular plot which contains two tracks and links inside the circle is generated. The first track will have a legend that contains points, the second track will have a legend that contains lines, and the links correspond to a continuous color mapping. The code is wrapped into a function so that it can be used repeatedly.

```
library(circlize)

col_fun = colorRamp2(c(-2, 0, 2), c("green", "yellow", "red"))
circlize_plot = function() {
    set.seed(12345)
    fa = letters[1:10]
    circos.initialize(fa, xlim = c(0, 1))
    circos.track(ylim = c(0, 1), panel.fun = function(x, y) {
        circos.points(runif(20), runif(20), cex = 0.5, pch = 16, col = 2)
        circos.points(runif(20), runif(20), cex = 0.5, pch = 16, col = 3)
    })
    circos.track(ylim = c(0, 1), panel.fun = function(x, y) {
        circos.lines(sort(runif(20)), runif(20), col = 4)
        circos.lines(sort(runif(20)), runif(20), col = 5)
    })

    for(i in 1:10) {
        circos.link(sample(fa, 1), sort(runif(10))[1:2],
                    sample(fa, 1), sort(runif(10))[1:2],
                    col = add_transparency(col_fun(rnorm(1))))
    }
    circos.clear()
}
```

In **ComplexHeatmap** package with version higher than 1.13.2, there is a `Legend()` function which customizes legends with various styles. In following code, legends for the two tracks and links are constructed.

In the end the three legends are packed vertically by `packLegend()`. For more detailed usage of `Legend()` and `packLegend()`, please refer to their help pages.

```
library(ComplexHeatmap)
# discrete
lgd_points = Legend(at = c("label1", "label2"), type = "points",
    legend_gp = gpar(col = 2:3), title_position = "topleft",
    title = "Track1")
# discrete
lgd_lines = Legend(at = c("label3", "label4"), type = "lines",
    legend_gp = gpar(col = 4:5, lwd = 2), title_position = "topleft",
    title = "Track2")
# continuous
lgd_links = Legend(at = c(-2, -1, 0, 1, 2), col_fun = col_fun,
    title_position = "topleft", title = "Links")

lgd_list_vertical = packLegend(lgd_points, lgd_lines, lgd_links)
lgd_list_vertical
```

```
## frame[GRID.frame.65]
```

`lgd_points`, `lgd_lines`, `lgd_links` and `lgd_list_vertical` are all `grob` objects (graphical objects) defined by **grid** package, which you can think as boxes which contain all graphical elements for legends and they can be added to the plot by `grid.draw()`.

circlize is implemented in the base graphic system while **ComplexHeatmap** is implemented by **grid** graphic system. However, these two systems can be mixed somehow. We can directly add grid graphics to the base graphics. (Actually they are two independent layers but drawn on a same graphic device.)

```
circlize_plot()
# next the grid graphics are added directly to the plot
# where circlize has created.
pushViewport(viewport(x = unit(2, "mm"), y = unit(4, "mm"),
    width = grobWidth(lgd_list_vertical),
    height = grobHeight(lgd_list_vertical),
    just = c("left", "bottom")))
grid.draw(lgd_list_vertical)
upViewport()
```

In Figure 4.1, the whole image region corresponds to the circular plot and the legend layer is drawn just on top of it. Actually you can see that one big problem is when there are many legends that the size for the legends is too big, they may overap to the circle. One solution is to split the legends into several parts and add each part to different corners in the plot (Figure 4.2).

```
lgd_list_vertical2 = packLegend(lgd_points, lgd_lines)
circlize_plot()
# next the grid graphics are added directly to the plot
# where circlize has created.
pushViewport(viewport(x = unit(2, "mm"), y = unit(2, "mm"),
    width = grobWidth(lgd_list_vertical2),
    height = grobHeight(lgd_list_vertical2),
    just = c("left", "bottom")))
grid.draw(lgd_list_vertical2)
upViewport()
pushViewport(viewport(x = unit(1, "npc") - unit(2, "mm"), y = unit(2, "mm"),
    width = grobWidth(lgd_links),
    height = grobHeight(lgd_links),
```

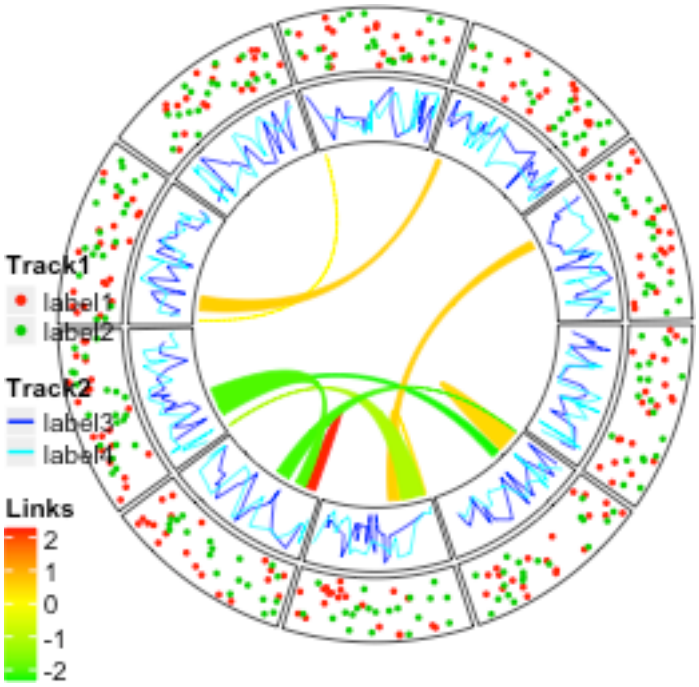


Figure 4.1: Directly add grid graphics.

```
    just = c("right", "bottom")))
grid.draw(lgd_links)
upViewport()
```

Still it can not solve the problem and sometimes it even makes the plot so messed up. One better way is to split the image region into two parts where one part only for the circular plot and the other part for legends.

To mix grid graphics and base graphics, ther are two important packages to use: the **grid** package and **gridBase** package. **grid** is the base for making grid graphics as well as arranging plotting regions (or, *viewports* in **grid** term), and **gridBase** makes it easy to integrate base graphics into grid system.

Following code is straightforward to understand. Only one line needs to be noticed: `par(omi = gridOMI(), new = TRUE)` that `gridOMI()` calculates the outer margins for the base graphics so that the base graphics can be put at the correct place and `new = TRUE` to ensure the base graphics are added to current graphic device instead of opening a new one.

Here I use `plot.new()` to open a new graphic device. In interactive session, it seems ok if you also use `grid.newpage()`, but `grid.newpage()` gives error when building a **knitr** document.

```
library(gridBase)
plot.new()
circle_size = unit(1, "snpc") # snpc unit gives you a square region

pushViewport(viewport(x = 0, y = 0.5, width = circle_size, height = circle_size,
    just = c("left", "center")))
par(omi = gridOMI(), new = TRUE)
circlize_plot()
upViewport()

pushViewport(viewport(x = circle_size, y = 0.5, width = grobWidth(lgd_list_vertical),
    height = grobHeight(lgd_list_vertical), just = c("left", "center")))
```

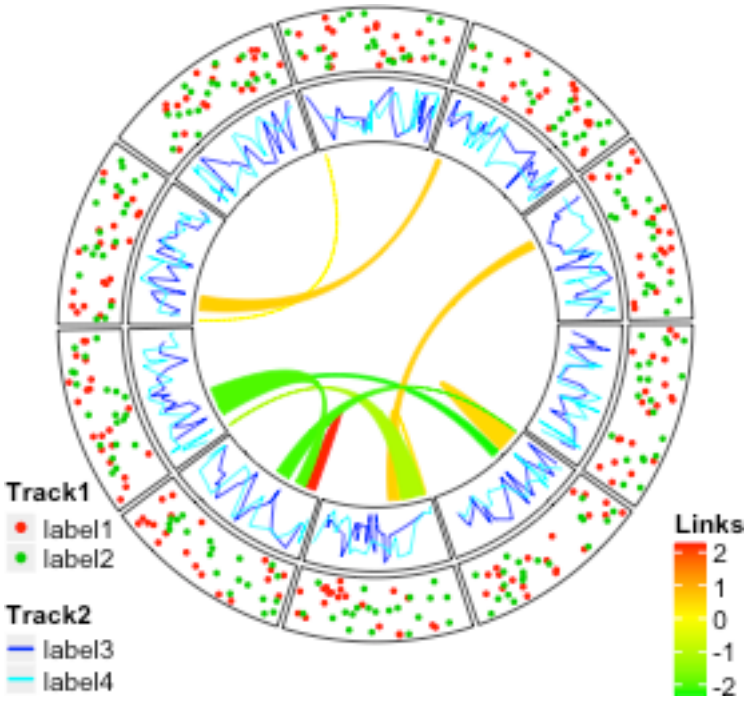
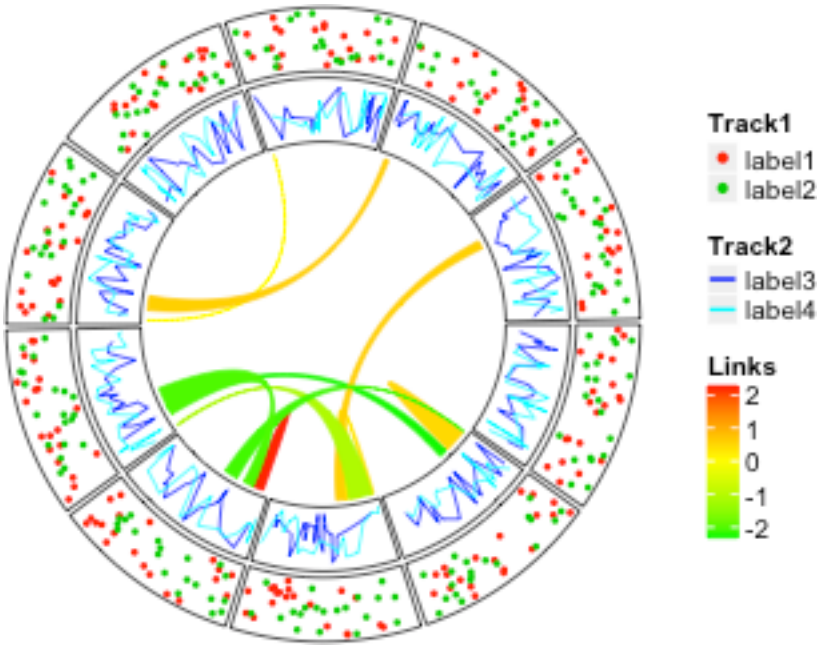


Figure 4.2: Split into two legends.

```
grid.draw(lgd_list_vertical)
upViewport()
```

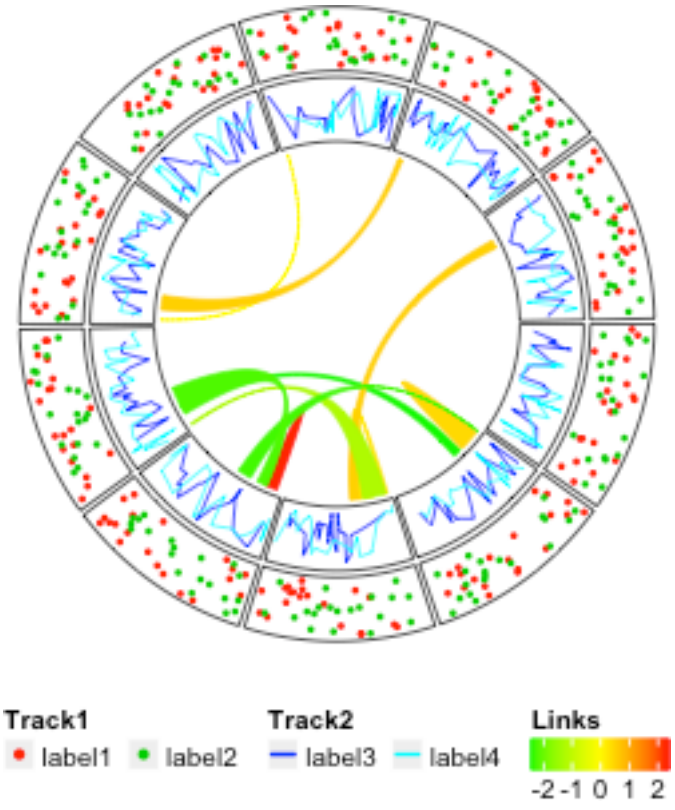


The legends can also be put at the bottom of the circular plot and it is just a matter how users arrange the grid viewports. In this case, all legends are changed to horizontal style, and three legends are packed horizontally as well.

```
lgd_points = Legend(at = c("label1", "label2"), type = "points",
    legend_gp = gpar(col = 2:3), title_position = "topleft",
    title = "Track1", nrow = 1)

lgd_lines = Legend(at = c("label3", "label4"), type = "lines",
    legend_gp = gpar(col = 4:5, lwd = 2), title_position = "topleft",
    title = "Track2", nrow = 1)

lgd_links = Legend(at = c(-2, -1, 0, 1, 2), col_fun = col_fun,
    title_position = "topleft", title = "Links", direction = "horizontal")

lgd_list_horizontal = packLegend(lgd_points, lgd_lines, lgd_links,
    direction = "horizontal")
```

Similar code to arrange viewports.

```
plot.new()
pushViewport(viewport(x = 0.5, y = 1, width = circle_size, height = circle_size,
    just = c("center", "top")))
par(omi = gridOMI(), new = TRUE)
circlize_plot()
upViewport()

pushViewport(viewport(x = 0.5, y = unit(1, "npc") - circle_size,
    width = grobWidth(lgd_list_horizontal), height = grobHeight(lgd_list_horizontal),
    just = c("center", "top")))
grid.draw(lgd_list_horizontal)
upViewport()
```



Chapter 5

Implement high-level circular plots

In this chapter, we show several examples which combine low-level graphic functions to construct complicated graphics for specific purposes.

5.1 Circular barplots

In the following code, we put all the nine bars in one track and one sector. You can also put them into 9 tracks, but the code would be very similar. See Figure 5.1.

```
category = paste0("category", "_", 1:9)
percent = sort(sample(40:80, 9))
color = rev(rainbow(length(percent)))

library(circlize)
circos.par("start.degree" = 90, cell.padding = c(0, 0, 0, 0))
circos.initialize("a", xlim = c(0, 100)) # 'a` just means there is one sector
circos.track(ylim = c(0.5, length(percent)+0.5), track.height = 0.8,
    bg.border = NA, panel.fun = function(x, y) {
        xlim = CELL_META$xlim
        circos.segments(rep(xlim[1], 9), 1:9,
                        rep(xlim[2], 9), 1:9,
                        col = "#CCCCCC")
        circos.rect(rep(0, 9), 1:9 - 0.45, percent, 1:9 + 0.45,
            col = color, border = "white")
        circos.text(rep(xlim[1], 9), 1:9,
            paste(category, " - ", percent, "%"),
            facing = "downward", adj = c(1.05, 0.5), cex = 0.8)
        breaks = seq(0, 85, by = 5)
        circos.axis(h = "top", major.at = breaks, labels = paste0(breaks, "%"),
            labels.cex = 0.6)
})
```

```
circos.clear()
```

When adding text by `circos.text()`, `adj` is specified to `c(1.05, 0.5)` which means text is aligned to the right and there is also offset between the text and the anchor points. We can also use `ux()` to set the offset to absolute units. Conversion on x direction in a circular coordinate is affected by the position on y axis, here we must set the `h` argument. Following code can be used to replace the `circos.text()` in above example.

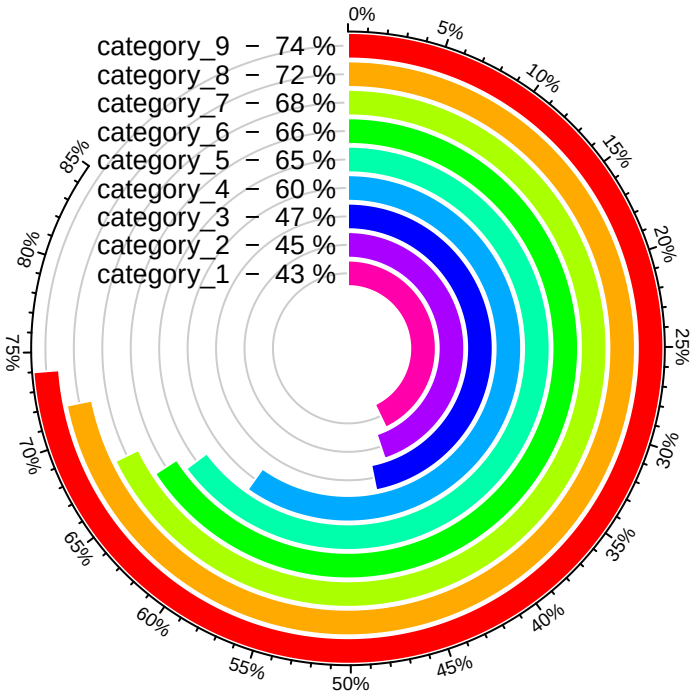


Figure 5.1: A circular barplot.

```
circos.text(xlim[1] - ux(2, "mm", h = 1:9), 1:9,
    paste(category, " - ", percent, "%"),
    facing = "downward", adj = c(1, 0.5), cex = 0.8)
```

5.2 Histograms

circlize ships a `circos.trackHist()` function which draws histograms in cells. This function is a high-level function which caculates data ranges on y axes and creates a new track. The implement of this function is simple, that it first calculates the histogram in each cell by `hist()` function, then draws histogram by using `circos.rect()`.

Users can choose to visualize data distributions by density lines by setting `draw.density = TRUE`.

Figure 5.2 shows a histogram track under default settings, a histogram track with specified `bin.size` and a track with density lines. By default, bin size of histogram in each cell is calculated separatedly and they will be different between cells, which makes it not consistent to compare. Manually setting `bin.size` in all cells to a same value helps to compare the distributions between cells.

```
x = rnorm(1600)
factors = sample(letters[1:16], 1600, replace = TRUE)
circos.initialize(factors = factors, x = x)
circos.trackHist(factors = factors, x = x, col = "#999999",
    border = "#999999")
circos.trackHist(factors = factors, x = x, bin.size = 0.1,
    col = "#999999", border = "#999999")
circos.trackHist(factors = factors, x = x, draw.density = TRUE,
    col = "#999999", border = "#999999")
```

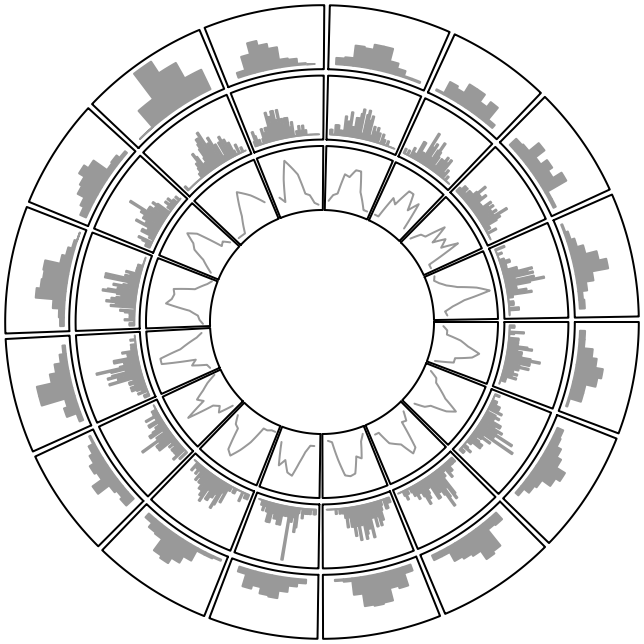


Figure 5.2: Histograms on circular layout.

```
circos.clear()
```

5.3 Phylogenetic trees

Circular dendrograms have many applications, one of which is to visualize phylogenetic trees. Basically, a phylogenetic tree is a dendrogram which is a combination of lines. In R, there are several classes that describe such type of tree such as `hclust`, `dendrogram` and `phylo`. In this example, we will demonstrate how to draw the tree from the `dendrogram` class. Nevertheless, other classes can be converted to `dendrogram` without too much difficulty.

The `bird.orders` data we are using here is from **ape** package. This data set is related to species of birds.

```
library(ape)
data(bird.orders)
hc = as.hclust(bird.orders)
```

We split the tree into six sub trees by `cutree()` and convert the data into a `dendrogram` object.

```
labels = hc$labels  # name of birds
ct = cutree(hc, 6)  # cut tree into 6 pieces
n = length(labels)  # number of bird species
dend = as.dendrogram(hc)
```

As we mentioned before, the x-value for the phylogenetic tree is in fact index. Thus, the x-lim is just the minimum and maximum index of labels in the tree. Since there is only one phylogenetic tree, we only need one "big" sector.

In the first track, we plot the name of each bird, with different colors to represent different sub trees.

```
circos.par(cell.padding = c(0, 0, 0, 0))
circos.initialize(factors = "a", xlim = c(0, n)) # only one sector
```

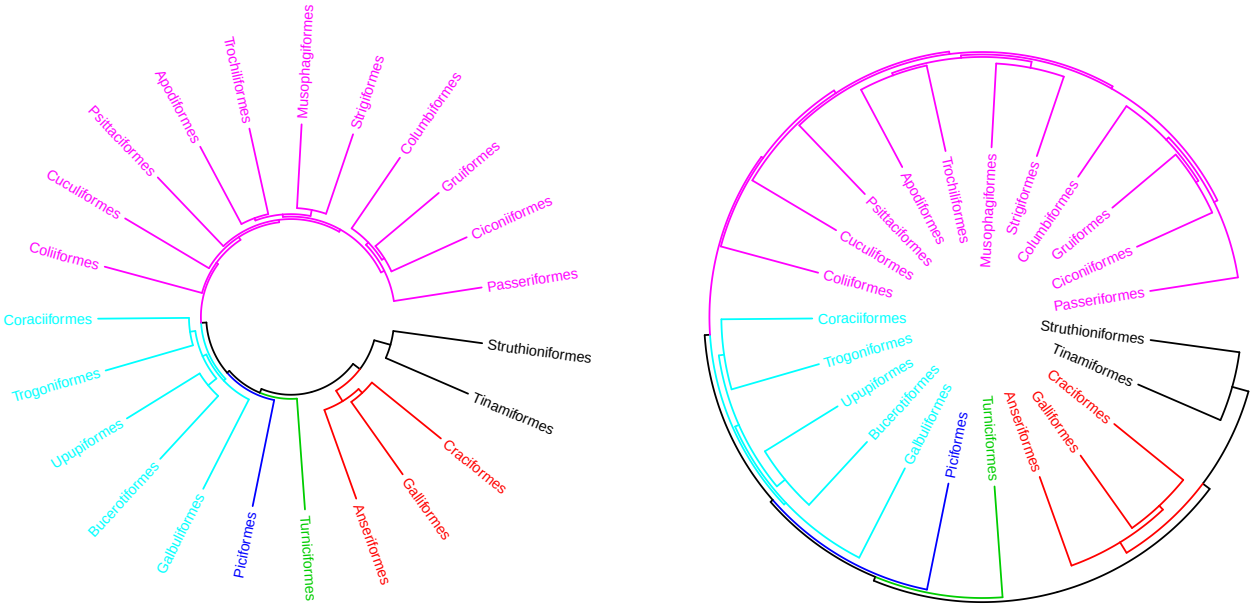


Figure 5.3: A circular phylogenetic tree.

```
circos.track(ylim = c(0, 1), bg.border = NA, track.height = 0.3,
    panel.fun = function(x, y) {
        for(i in seq_len(n)) {
            circos.text(i-0.5, 0, labels[i], adj = c(0, 0.5),
                facing = "clockwise", niceFacing = TRUE,
                col = ct[labels[i]], cex = 0.5)
        }
})
```

In the above code, setting `xlim` to `c(0, n)` is very important because the leaves of the dendrogram are drawn at `x = seq(0.5, n - 0.5)`.

In the second track, we plot the circular dendrogram by `circos.dendrogram()` (Figure 5.3 left). You can render the dendrogram by **dendextend** package.

```
suppressPackageStartupMessages(library(dendextend))
dend = color_branches(dend, k = 6, col = 1:6)
dend_height = attr(dend, "height")
circos.track(ylim = c(0, dend_height), bg.border = NA,
    track.height = 0.4, panel.fun = function(x, y) {
        circos.dendrogram(dend)
})
circos.clear()
```

By default, dendrograms are facing outside of the circle (so that the labels should also be added outside the dendrogram). In `circos.dendrogram()`, you can set `facing` argument to `inside` to make them facing inside. In this case, dendrogram track is added first and labels are added later (Figure 5.3 right).

```
circos.dendrogram(dend, facing = "inside")
```

If you look at the souce code of `circos.dendrogram()` and replace `circos.lines()` to `lines()`, actually the function can correctly make a dendrogram in the normal coordinate.

With the flexibility of **circlize** package, it is easy to add more tracks if you want to add more corresponded

information for the dendrogram to the plot.

5.4 Heatmaps

Heatmaps, and sometimes combined with dendrograms are frequently used to visualize e.g. gene expression. Heatmaps are basically composed by rectangles, thus, they can be implemented by `circos.rect()`.

In following example, we make a circular plot with two heatmaps. First we generate the two matrix and perform clustring on the two matrix.

```
mat = matrix(rnorm(100*10), nrow = 10, ncol = 100)
col_fun = colorRamp2(c(-2, 0, 2), c("green", "black", "red"))
factors = rep(letters[1:2], times = c(30, 70))
mat_list = list(a = mat[, factors == "a"],
                b = mat[, factors == "b"])
dend_list = list(a = as.dendrogram(hclust(dist(t(mat_list[["a"]])))),
                 b = as.dendrogram(hclust(dist(t(mat_list[["b"]])))))
```

In the first track, columns in the matrix are adjusted by the clustering. Also note we use `circos.rect()` in a vectorized way.

```
circos.par(cell.padding = c(0, 0, 0, 0), gap.degree = 5)
circos.initialize(factors, xlim = cbind(c(0, 0), table(factors)))
circos.track(ylim = c(0, 10), bg.border = NA, panel.fun = function(x, y) {
    sector.index = CELL_META$sector.index
    m = mat_list[[sector.index]]
    dend = dend_list[[sector.index]]

    m2 = m[, order.dendrogram(dend)]
    col_mat = col_fun(m2)
    nr = nrow(m2)
    nc = ncol(m2)
    for(i in 1:nr) {
        circos.rect(1:nc - 1, rep(nr - i, nc),
            1:nc, rep(nr - i + 1, nc),
            border = col_mat[i, ], col = col_mat[i, ])
    }
})
```

Since there are two dendrograms, it is important to make the height of both dendrogram in a same scale. We calculate the maximum height of the two dendrograms and set it to `ylim` of the second track (Figure 5.4).

```
max_height = max(sapply(dend_list, function(x) attr(x, "height")))
circos.track(ylim = c(0, max_height), bg.border = NA, track.height = 0.3,
    panel.fun = function(x, y) {

        sector.index = get.cell.meta.data("sector.index")
        dend = dend_list[[sector.index]]
        circos.dendrogram(dend, max_height = max_height)
})
circos.clear()
```

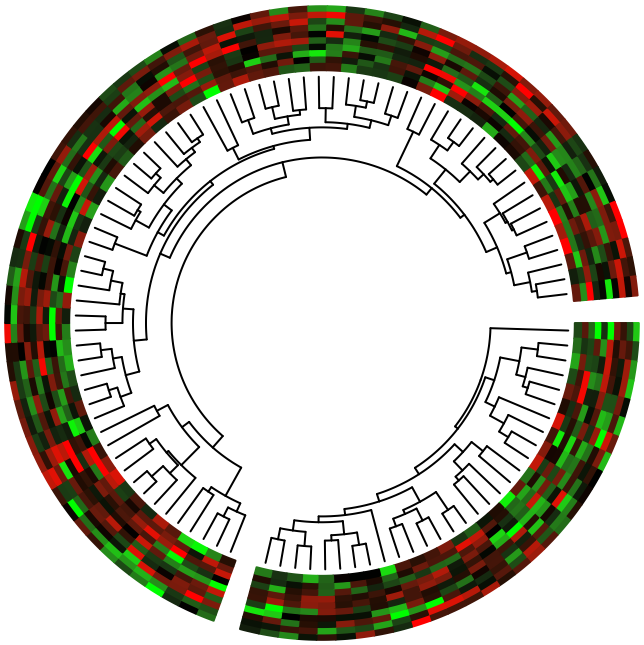


Figure 5.4: Circular heatmaps.

Chapter 6

Advanced layout

6.1 Zooming of sectors

In this section, we will introduce how to zoom sectors and put the zoomed sectors at the same track as the original sectors.

Under the default settings, width of sectors are calculated according to the data range in corresponding categories. Normally it is not a good idea to manually modify the default sector width since it reflects useful information of your data. However, sometimes manually modifying the width of sectors can make more advanced plots, e.g. zoomings.

The basic idea for zooming is to put original sectors on part of the circle and put the zoomed sectors on the other part, so that in the original sectors, widths are still proportional to their data ranges, and in the zoomed sectors, the widths are also proportional to the data ranges in the zoomed sectors.

This type of zooming is rather simple to implement. All we need to do is to copy the data which corresponds to the zoomed sectors, assign new category names to them and append to the original data. The good thing is since the data in the zoomed sectors is exactly the same as the original sectors, if you treat them as normal categories, the graphics will be exactly the same as in the original sectors, but with x direction zoomed.

Following example shows more clearly the basic idea of this "horizontal" zooming.

We first generate a data frame with six categories.

```
set.seed(123)
df = data.frame(
    factors = sample(letters[1:6], 400, replace = TRUE),
    x = rnorm(400),
    y = rnorm(400),
    stringsAsFactors = FALSE
)
```

We want to zoom sector a and the first 10 points in sector b. First we extract these data and format as a new data frame.

```
zoom_df_a = df[df$factors == "a", ]
zoom_df_b = df[df$factors == "b", ]
zoom_df_b = zoom_df_b[order(zoom_df_b[, 2])[1:10], ]
zoom_df = rbind(zoom_df_a, zoom_df_b)
```

Then we need to change the sector names in the zoomed data frame. Here we just simply add "zoom_" prefix to the original names to show that they are "zoomed" sectors. After that, it is attached to the original

data frame.

```
zoom_df$factors = paste0("zoom_", zoom_df$factors)
df2 = rbind(df, zoom_df)
```

In this example, we will put the original cells in the left half of the circle and the zoomed sectors in the right. As we have already mentioned before, we simply normalize the width of normal sectors and normalize the width of zoomed sectors separately. Note now the sum of the sector width for the original sectors is 1 and the sum of sector width for the zoomed sectors is 1, which means these two types of sectors have their own half circle.

You may notice the sum of the `sector.width` is not idential to 1. This is fine, they will be further normalized to 1 internally.

Strictly speaking, since the gaps between sectors are not taken into consideration, the width of the original sectors are not exactly 180 degree, but the real value is quite close to it.

```
xrange = tapply(df2$x, df2$factors, function(x) max(x) - min(x))
normal_sector_index = unique(df$factors)
zoomed_sector_index = unique(zoom_df$factors)
sector.width = c(xrange[normal_sector_index] / sum(xrange[normal_sector_index]),
                 xrange[zoomed_sector_index] / sum(xrange[zoomed_sector_index]))
sector.width
```

```
##         b         e         c         f         a         d    zoom_a
## 0.1685162 0.1765301 0.1732999 0.1705148 0.1518915 0.1592475 0.8097123
##    zoom_b
## 0.1902877
```

What to do next is just to make the circular plot in the normal way. All the graphics in sector a and b will be automatically zoomed to sector "zoom_a" and "zoom_b".

In following code, since the sector names are added outside the first track, `points.overflow.warning` is set to `FALSE` to turn off the warning messages.

```
circos.par(start.degree = 90, points.overflow.warning = FALSE)
circos.initialize(df2$factors, x = df2$x, sector.width = sector.width)
circos.track(df2$factors, x = df2$x, y = df2$y,
    panel.fun = function(x, y) {
    circos.points(x, y, col = "red", pch = 16, cex = 0.5)
    circos.text(CELL_META$xcenter, CELL_META$cell.ylim[2] + uy(2, "mm"),
        CELL_META$sector.index, niceFacing = TRUE)
})
```

Adding links from original sectors to zoomed sectors is a good idea to show where the zooming occurs (Figure 6.1). Notice that we manually adjust the position of one end of the sector b link.

```
circos.link("a", get.cell.meta.data("cell.xlim", sector.index = "a"),
    "zoom_a", get.cell.meta.data("cell.xlim", sector.index = "zoom_a"),
    border = NA, col = "#00000020")
circos.link("b", c(zoom_df_b[1, 2], zoom_df_b[10, 2]),
    "zoom_b", get.cell.meta.data("cell.xlim", sector.index = "zoom_b"),
    rou1 = get.cell.meta.data("cell.top.radius", sector.index = "b"),
    border = NA, col = "#00000020")
circos.clear()
```

Chapter 12 introduces another type of zooming by combining two circular plots.

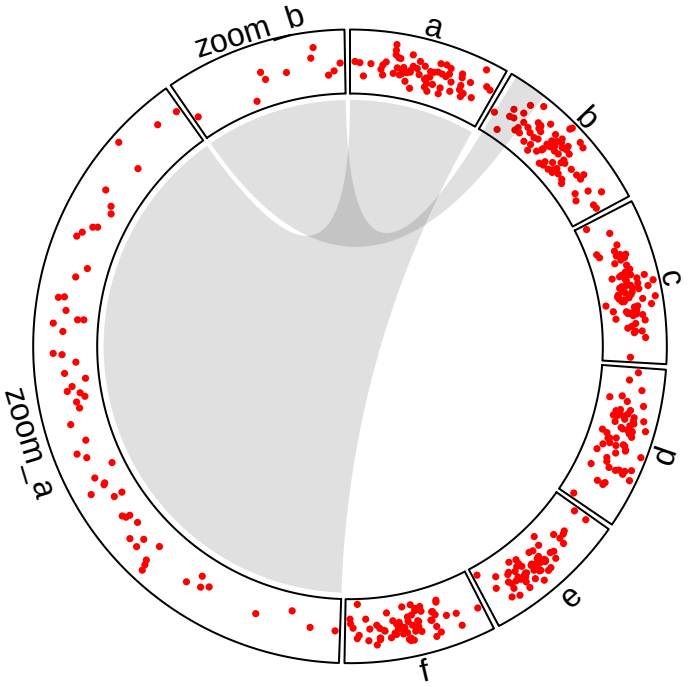


Figure 6.1: Zoom sectors.

6.2 Visualize part of the circle

`canvas.xlim` and `canvas.ylim` parameters in `circos.par()` are useful to generate plots only in part of the circle. As mentioned in previews chapters, the circular plot is always drawn in a canvas where x values range from -1 to 1 and y values range from -1 to 1. Thus, if `canvas.xlim` and `canvas.ylim` are all set to `c(0, 1)`, which means, the canvas is restricted to the right top part, then only sectors between 0 to 90 degree are visible (Figure 6.2).

To make the right plot in Figure 6.2, we only need to set one sector in the layout and set `gap.after` to 270. (One sector with `gap.after` of 270 degree means the width of this sector is exactly 90 degree.)

```
circos.par("canvas.xlim" = c(0, 1), "canvas.ylim" = c(0, 1),
    "start.degree" = 90, "gap.after" = 270)
factors = "a" # this is the name of your sector
circos.initialize(factors = factors, xlim = ...)
...
```

Similar idea can be applied to the circle where in some tracks, only a subset of cells are needed. Gererally there are two ways. The first way is to create the track and add graphics with subset of data that only corresponds to the cells that are needed. And the second way is to create an empty track first and customize the cells by `circos.update()`. Following code illustrates the two methods (Figure 6.3).

```
factors = letters[1:4]
circos.initialize(factors = factors, xlim = c(0, 1))

# directly specify the subset of data
df = data.frame(factors = rep("a", 100),
                x = runif(100),
                y = runif(100))
circos.track(df$factors, x = df$x, y = df$y,
```

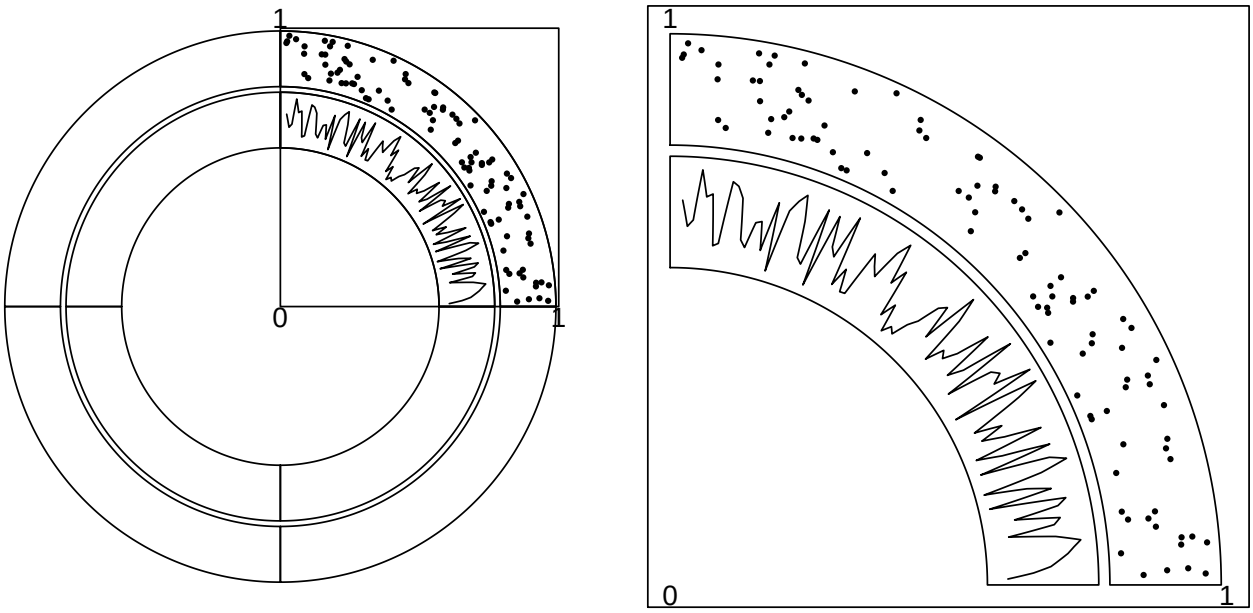


Figure 6.2: One quarter of the circle.

```
    panel.fun = function(x, y) {
        circos.points(x, y, pch = 16, cex = 0.5)
})

# create empty track first then fill graphics in the cell
circos.track(ylim = range(df$y), bg.border = NA)
circos.update(sector.index = "a", bg.border = "black")
circos.points(df$x, df$y, pch = 16, cex = 0.5)

circos.track(factors = factors, ylim = c(0, 1))
circos.track(factors = factors, ylim = c(0, 1))
```

```
circos.clear()
```

6.3 Combine multiple circular plots

circlize finally makes the circular plot in the base R graphic system. Seperated circular plots actually can be put in a same page by some tricks from the base graphic system. Here the key is `par(new = TRUE)` which allows to draw a new figure as a new layer directly on the previous canvas region. By setting different `canvas.xlim` and `canvas.ylim`, it allows to make more complex plots which include more than one circular plots.

Folowing code shows how the two independent circualr plots are added and nested. Figure 6.4 illustrates the invisible canvas coordinate and how the two circular plots overlap.

```
factors = letters[1:4]
circos.initialize(factors = factors, xlim = c(0, 1))
circos.track(ylim = c(0, 1), panel.fun = function(x, y) {
    circos.text(0.5, 0.5, "outer circos", niceFacing = TRUE)
})
```

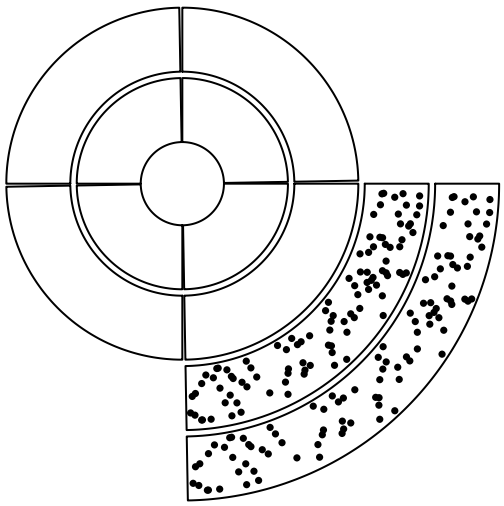


Figure 6.3: Show subset of cells in tracks.

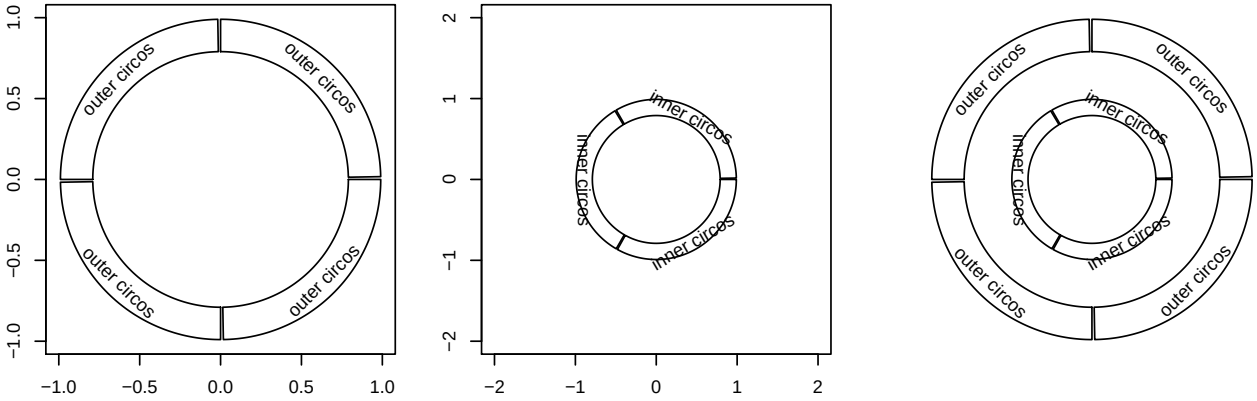


Figure 6.4: Nested circular plots.

```
circos.clear()

par(new = TRUE) # <- magic
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.track(ylim = c(0, 1), panel.fun = function(x, y) {
    circos.text(0.5, 0.5, "inner circos", niceFacing = TRUE)
})
circos.clear()
```

The second example (Figure 6.5) makes a plot where two circular plots separate from each other. You can use technique introduced in Section 6.2 to only show part of the circle, select proper `canvas.xlim` and `canvas.ylim`, and finally arrange the two plots into one page. The source code for generating Figure 6.5 is at https://github.com/jokergoo/circlize_book/blob/master/src/intro-20-separated.R.

The third example is to draw cells with different radius (Figure 6.6). In fact, it makes four circular plots where only one sector for each plot is plotted.

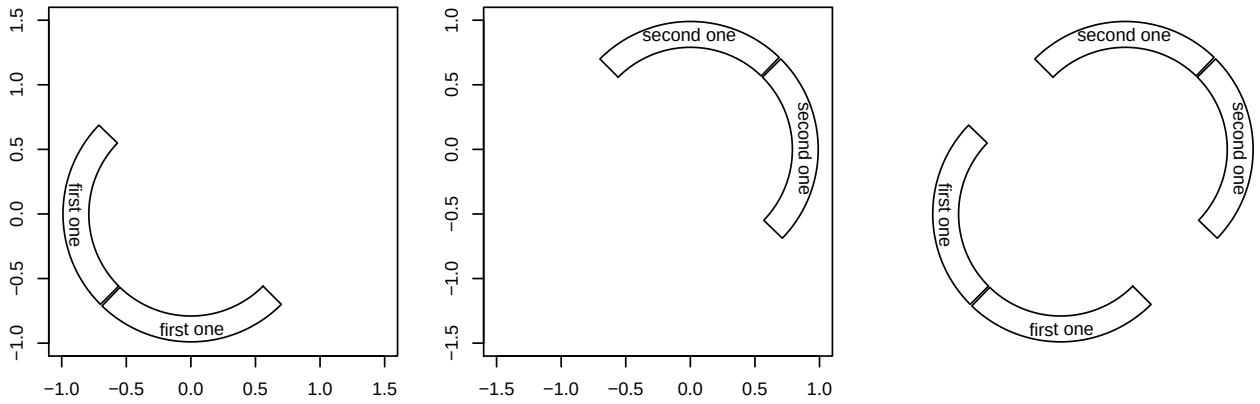


Figure 6.5: Two separated circular plots

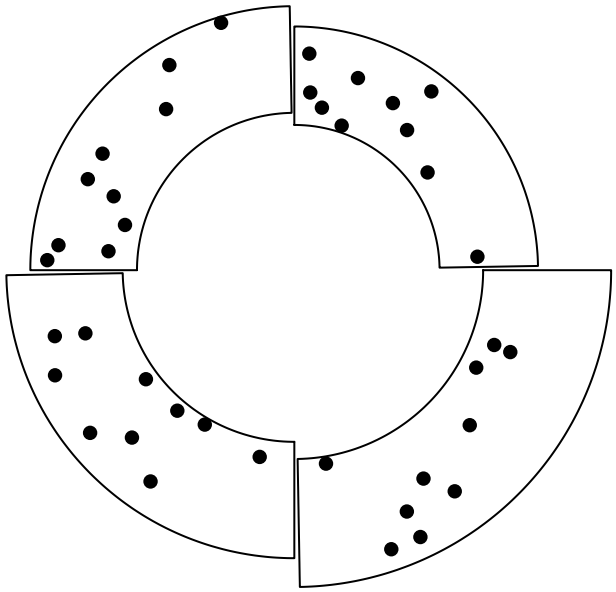


Figure 6.6: Cells with differnet radius.

```
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]),
        "track.height" = 0.4)
    circos.initialize(factors = factors, xlim = c(0, 1))
    circos.track(ylim = c(0, 1), bg.border = NA)
    circos.update(sector.index = factors[i], bg.border = "black")
    circos.points(runif(10), runif(10), pch = 16)
    circos.clear()
    par(new = TRUE)
}
par(new = FALSE)
```

Note above plot is different from the example in Figure 6.3. In Figure 6.3, cells both visible and invisible all

belong to a same track and they are in a same circular plot, thus they should have same radius. But for the example here, cells have different radius and they belong to different circular plot.

In chapter 12, we use this technique to implement zoomings by combining two circular plots.

6.4 Arrange multiple plots

circlize is implemented in the base R graphic system, thus, you can use `layout()` or `par(mforw, mfcol)` to arrange multiple circular plots in one page (Figure 6.7).

```
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.track(ylim = c(0, 1), track.height = 0.05,
        bg.col = rand_color(8), bg.border = NA)
    for(i in 1:20) {
        se = sample(1:8, 2)
        circos.link(se[1], runif(2), se[2], runif(2),
            col = rand_color(1, transparency = 0.4), border = NA)
    }
    circos.clear()
}
```

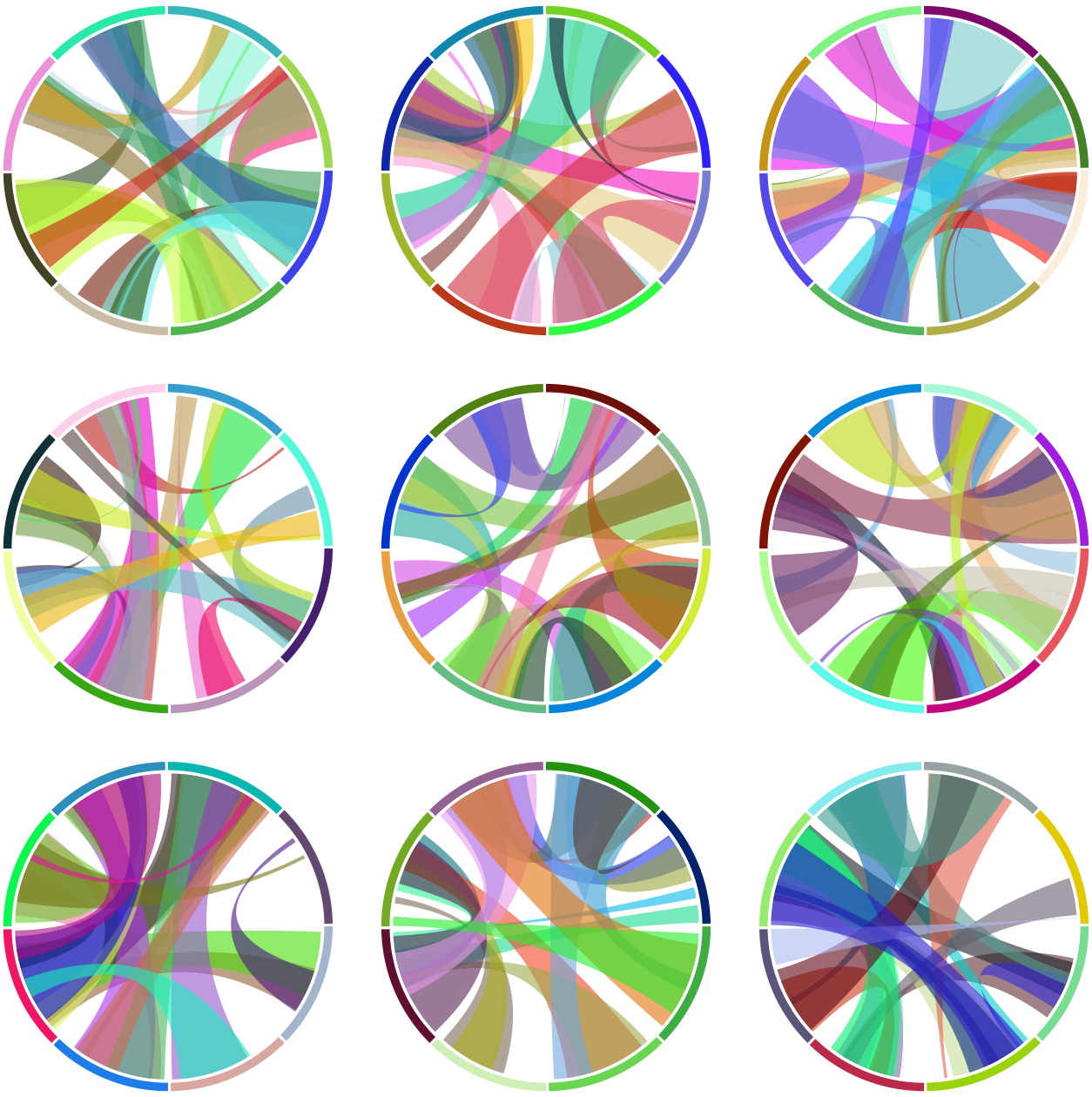


Figure 6.7: Arrange multiple circular plots.

Part II

Applications in Genomics

Chapter 7

Introduction

Circular visualization is popular in Genomics and related omics fields. It is efficient in revealing associations in high dimensional genomic data. In genomic plots, categories are usually chromosomes and data on x axes are genomic positions, but it can also be any kind of general genomic categories.

To make is easy for Genomics analysis, **circlize** package particularly provides functions which focus on genomic plots. These functions are synonymous to the basic graphic functions but expect special format of input data:

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

The genomic functions are implemented by basic circlize functions (e.g. `circos.track()`, `circos.points()`), thus, the use of genomic functions can be mixed with the basic circlize functions.

7.1 Input data

Genomic data is usually stored as a table where the first three columns define the genomic regions and following columns are values associated with the corresponding regions. Each genomic region is composed by three elements: genomic category (in most case, it is the chromosome), start position on the genomic category and the end position. Such data structure is known as *BED* format and is broadly used in genomic research.

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

```
set.seed(999)
bed = generateRandomBed()
head(bed)
```

```
##    chr   start     end     value1
## 1 chr1   39485  159163 -0.1887635
## 2 chr1  897195 1041959 -0.2435220
```

```
## 3 chr1 1161957 1177159  0.3749953
## 4 chr1 1201513 1481406 -0.2600839
## 5 chr1 1487402 1531773 -0.4633990
## 6 chr1 1769949 2736215 -0.8159909
```

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

```
## [1] 205
```

```
bed = generateRandomBed(nc = 2, fun = function(k) sample(letters, k, replace = TRUE))
head(bed)
```

```
##    chr   start     end value1 value2
## 1 chr1   98740  566688      e      e
## 2 chr1  769960  887938      b      q
## 3 chr1  906851  933021      u      o
## 4 chr1 1241911 1243537      k      f
## 5 chr1 1385344 1410947      v      x
## 6 chr1 1498302 1585389      u      v
```

All genomic functions in **circlize** expect input variable as a data frame which contains genomic data or a list of data frames which contains genomic data in different conditions.

Chapter 8

Initialize with genomic data

circlize is quite flexible to initialize the circular plot not only by chromosomes, but also by any type of general genomic categories.

8.1 Initialize with cytoband data

Cytoband data is an ideal data source to initialize genomic plots. It contains length of chromosomes as well as so called "chromosome band" annotation to help to identify positions on chromosomes.

8.1.1 Basic usage

If you work on human genome, the most straightforward way is to directly use `circos.initializeWithIdeogram()` (Figure 8.1). By default, the function creates a track with chromosome name and axes, and a track of ideograms.

Although chromosome names added to the plot are pure numeric, actually the internally names have the "chr" index. When you adding more tracks, the chromosome names should also have "chr" index.

```
circos.initializeWithIdeogram()
text(0, 0, "default", cex = 1)
```

```
circos.info()
```

```
## All your sectors:
##  [1] "chr1"  "chr2"  "chr3"  "chr4"  "chr5"  "chr6"  "chr7"  "chr8"
##  [9] "chr9"  "chr10" "chr11" "chr12" "chr13" "chr14" "chr15" "chr16"
## [17] "chr17" "chr18" "chr19" "chr20" "chr21" "chr22" "chrX"  "chrY"
##
## All your tracks:
## [1] 1 2
##
## Your current sector.index is chrY
## Your current track.index is 2
```

```
circos.clear()
```

By default, `circos.initializeWithIdeogram()` initializes the plot with cytoband data of human genome `hg19`. Users can also initialize with other species by specifying `species` argument and it will automatically download cytoband files for corresponding species.

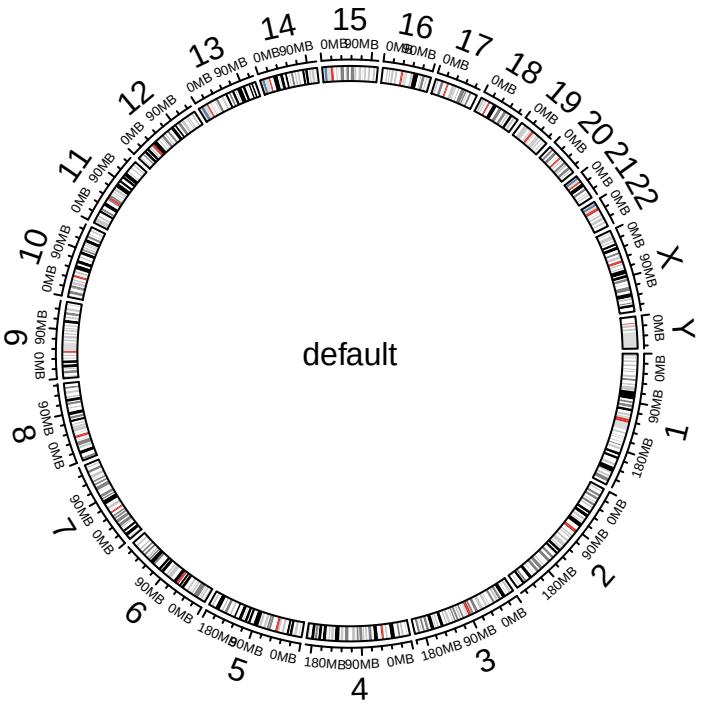


Figure 8.1: Initialize genomic plot, default.

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

When you are dealing rare species and there is no cytoband data available yet, `circos.initializeWithIdeogram()` will try to continue to download the "chromInfo" file form UCSC, which also contains lengths of chromosomes, but of course, there is no ideogram track on the plot.

In some cases, when there is no internet connection for downloading or there is no corresponding data avaiable on UCSC yet. You can manually construct a data frame which contains ranges of chromosomes or a file path if it is stored in a file, and sent to `circos.initializeWithIdeogram()`.

```
cytoband.file = system.file(package = "circlize", "extdata", "cytoBand.txt")
circos.initializeWithIdeogram(cytoband.file)

cytoband.df = read.table(cytoband.file, colClasses = c("character", "numeric",
    "numeric", "character", "character"), sep = "\t")
circos.initializeWithIdeogram(cytoband.df)
```

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

By default, `circos.intializeWithIdeogram()` uses all chromosomes which are available in cytoband data to initialize the circular plot. Users can choose a subset of chromosomes by specifying `chromosome.index`. This argument is also for ordering chromosomes (Figure 8.2).

```
circos.initializeWithIdeogram(chromosome.index = paste0("chr", c(3,5,2,8)))
text(0, 0, "subset of chromosomes", cex = 1)
```

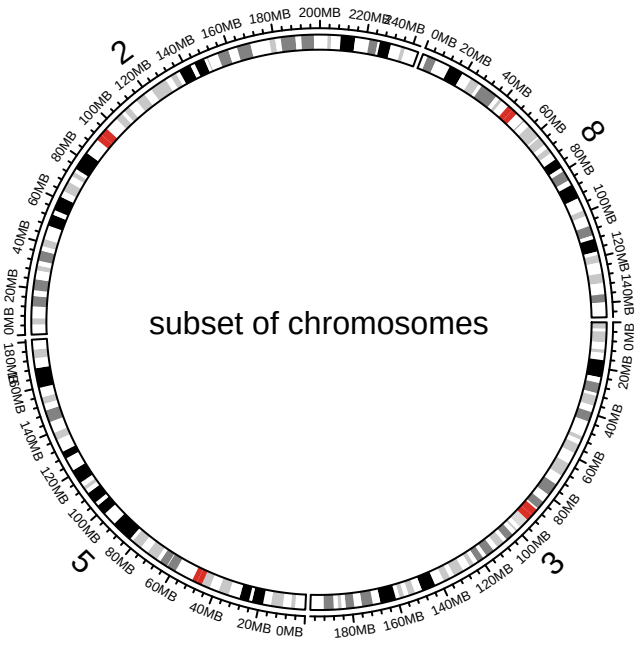


Figure 8.2: Initialize genomic plot, subset chromosomes.

```
circos.clear()
```

When there is no cytoband data for the specified species, and when chromInfo data is used instead, there may be many many extra short contigs. `chromosome.index` can also be useful to remove unnecessary contigs.

8.1.2 Pre-defined tracks

After the initialization of the circular plot, `circos.initializeWithIdeogram()` additionally creates a track where there are genomic axes and chromosome names, and create another track where there is an ideogram (depends on whether cytoband data is available). `plotType` argument is used to control which type of tracks to add. (figure Figure 8.3).

```
circos.initializeWithIdeogram(plotType = c("axis", "labels"))
text(0, 0, "plotType = c('axis', 'labels')", cex = 1)
circos.clear()

circos.initializeWithIdeogram(plotType = NULL)
text(0, 0, "plotType = NULL", cex = 1)
```

```
circos.clear()
```

8.1.3 Other general settings

Similar as general circular plot, the parameters for the layout can be controlled by `circos.par()` (Figure 8.4). Do remember when you explicitly set `circos.par()`, you need to call `circos.clear()` to finish the plotting.

```
circos.par("start.degree" = 90)
circos.initializeWithIdeogram()
```

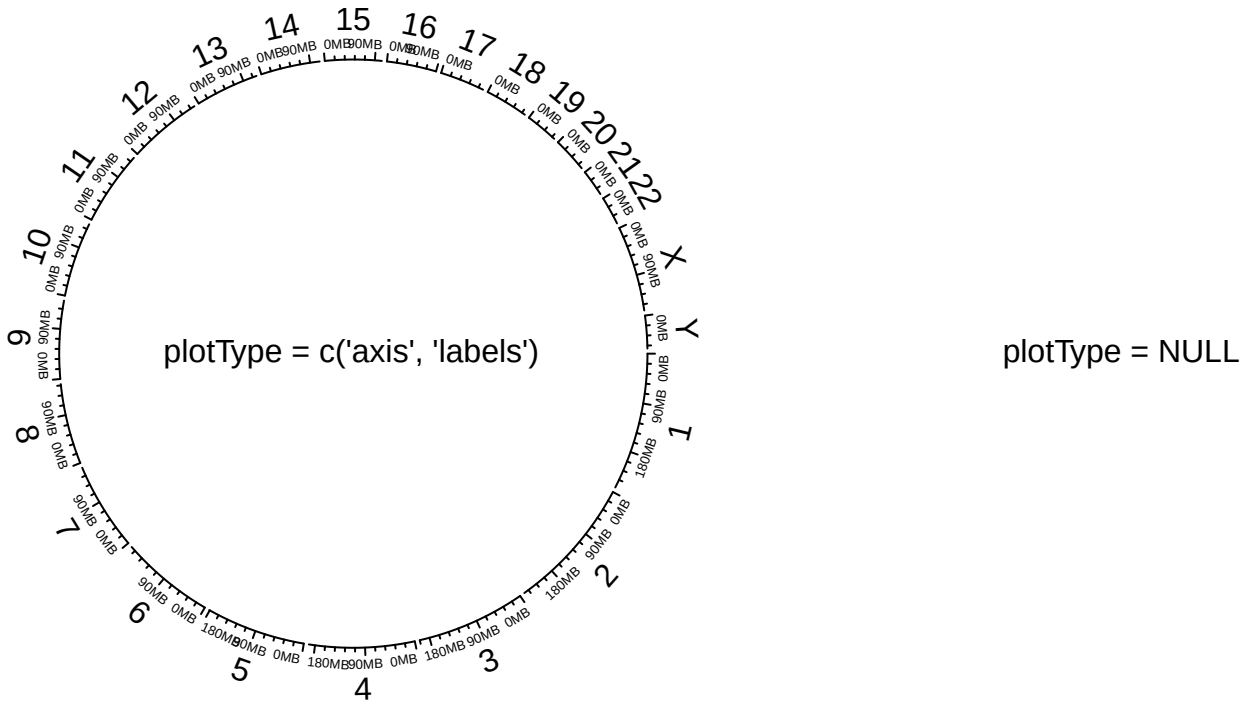


Figure 8.3: Initialize genomic plot, control tracks.

```
circos.clear()
text(0, 0, "'start.degree' = 90", cex = 1)

circos.par("gap.degree" = rep(c(2, 4), 12))
circos.initializeWithIdeogram()
circos.clear()
text(0, 0, "'gap.degree' = rep(c(2, 4), 12)", cex = 1)
```

8.2 Customize chromosome track

By default `circos.initializeWithIdeogram()` initializes the layout and adds two tracks. When `plotType` argument is set to `NULL`, the circular layout is only initialized but nothing is added. This makes it possible for users to completely design their own style of chromosome track.

In the following example, we use different colors to represent chromosomes and put chromosome names in the center of each cell (Figure 8.5).

```
set.seed(123)
circos.initializeWithIdeogram(plotType = NULL)
circos.track(ylim = c(0, 1), panel.fun = function(x, y) {
    chr = CELL_META$sector.index
    xlim = CELL_META$xlim
    ylim = CELL_META$ylim
    circos.rect(xlim[1], 0, xlim[2], 1, col = rand_color(1))
    circos.text(mean(xlim), mean(ylim), chr, cex = 0.7, col = "white",
        facing = "inside", niceFacing = TRUE)
}, track.height = 0.15, bg.border = NA)
```

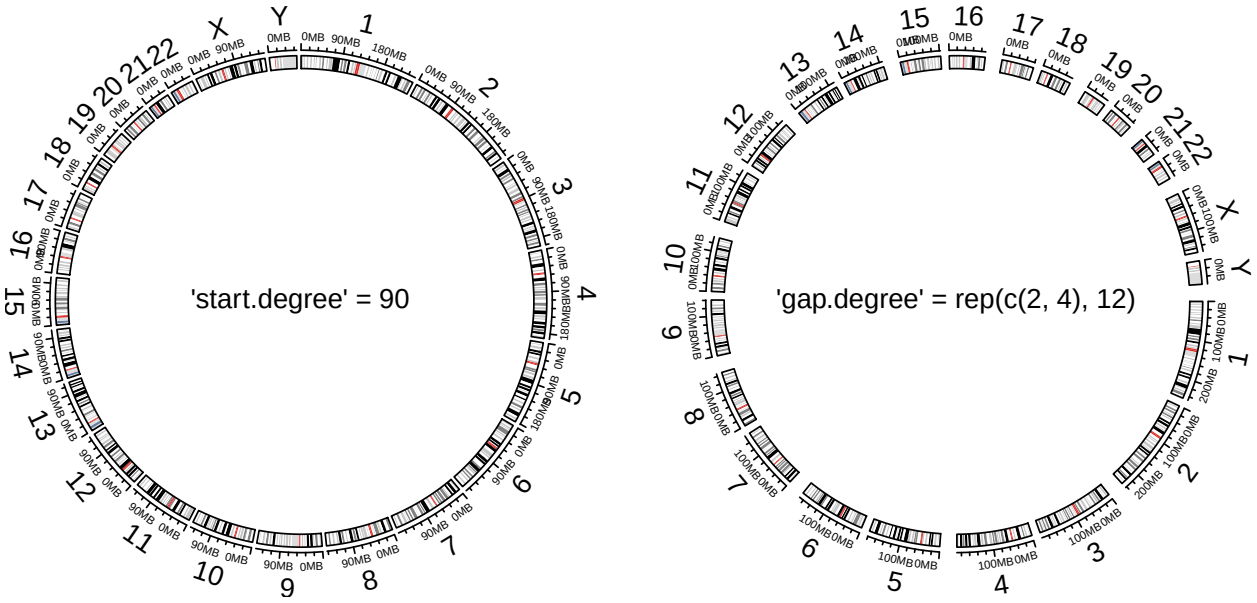


Figure 8.4: Initialize genomic plot, control layout.

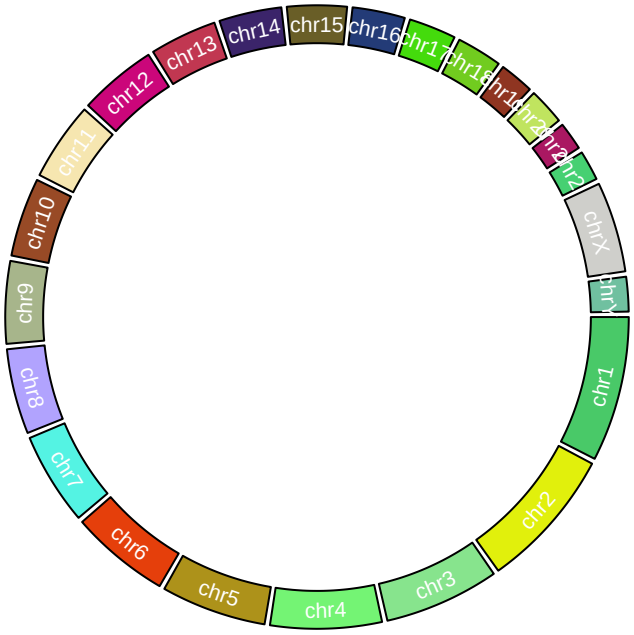


Figure 8.5: Customize chromosome track.

```
circos.clear()
```

8.3 Initialize with general genomic category

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

In the following example, a circular plot is initialized with three genes.

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

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

In following example, we plot the transcripts for TP53, TP63 and TP73 in a circular layout (Figure 8.6).

```
tp_family = readRDS(system.file(package = "circlize", "extdata", "tp_family_df.rds"))
head(tp_family)
```

```
##   gene   start     end        transcript exon
## 1 TP53 7565097 7565332 ENST00000413465.2    7
## 2 TP53 7577499 7577608 ENST00000413465.2    6
## 3 TP53 7578177 7578289 ENST00000413465.2    5
## 4 TP53 7578371 7578554 ENST00000413465.2    4
## 5 TP53 7579312 7579590 ENST00000413465.2    3
## 6 TP53 7579700 7579721 ENST00000413465.2    2
```

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

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

Next, we put transcripts one after the other for each gene. It is simply to add lines and rectangles. The usage of `circos.genomicTrack()` and `circos.genomicRect()` will be discussed in Chapter 9.

```
n = max(tapply(tp_family$transcript, tp_family$gene, function(x) length(unique(x))))
circos.genomicTrack(tp_family, ylim = c(0.5, n + 0.5),
    panel.fun = function(region, value, ...) {
        all_tx = unique(value$transcript)
        for(i in seq_along(all_tx)) {
            l = value$transcript == all_tx[i]
            # for each transcript
            current_tx_start = min(region[l, 1])
            current_tx_end = max(region[l, 2])
            circos.lines(c(current_tx_start, current_tx_end),
                c(n - i + 1, n - i + 1), col = "#CCCCCC")
            circos.genomicRect(region[l, , drop = FALSE], ytop = n - i + 1 + 0.4,
```

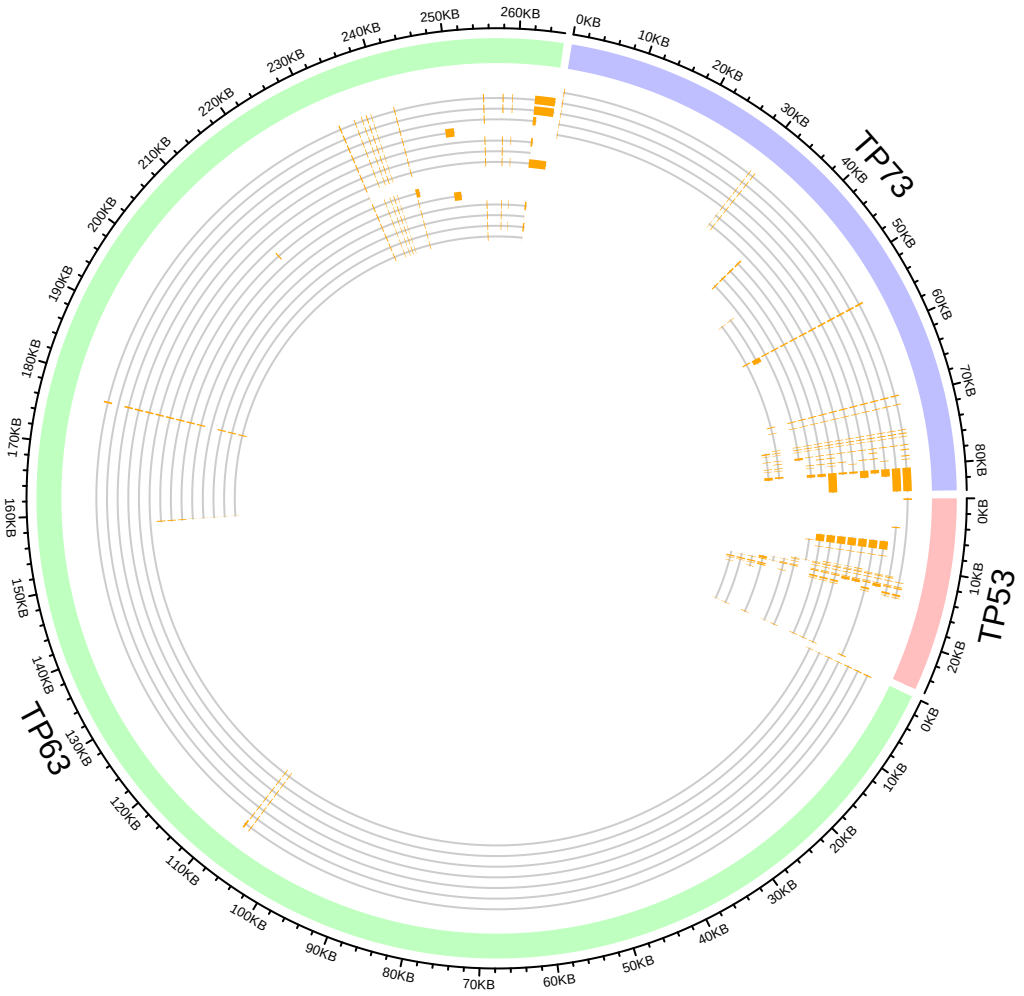


Figure 8.6: Circular representation of alternative transcripts for genes.

```
                ybottom = n - i + 1 - 0.4, col = "orange", border = NA)
        }
}, bg.border = NA, track.height = 0.4)
circos.clear()
```

In Figure 8.6, you may notice the start of axes becomes "0KB" while not the original values. It is just an adjustment of the axes labels to reflect the relative distance to the start of each gene, while the coordinate in the cells are still using the original values. Set `tickLabelsStartFromZero` to `FALSE` to recover axes labels to the original values.

8.4 Zooming chromosomes

The strategy is the same as introduced in Section 6.1. We first define a function `extend_chromosomes()` which copy data in subset of chromosomes into the original data frame.

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

```
    rbind(bed, zoom_bed)
}
```

We use `read.cytoband()` to download and read cytoband data from UCSC. In following, x ranges for normal chromosomes and zoomed chromosomes are normalized separetely.

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

xrange = c(cytoband$chr.len, cytoband$chr.len[c("chr1", "chr2")])
normal_chr_index = 1:24
zoomed_chr_index = 25:26

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

The extended cytoband data which is in form of a data frame is sent to `circos.initializeWithIdeogram()`. You can see the ideograms for chromosome 1 and 2 are zoomed (Figure 8.7).

```
circos.par(start.degree = 90)
circos.initializeWithIdeogram(extend_chromosomes(cytoband_df, c("chr1", "chr2")),
    sector.width = sector.width)
```

Add a new track.

```
bed = generateRandomBed(500)
circos.genomicTrack(extend_chromosomes(bed, c("chr1", "chr2")),
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, pch = 16, cex = 0.3)
})
```

Add a link from original chromosome to the zoomed chromosome (Figure 8.7).

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

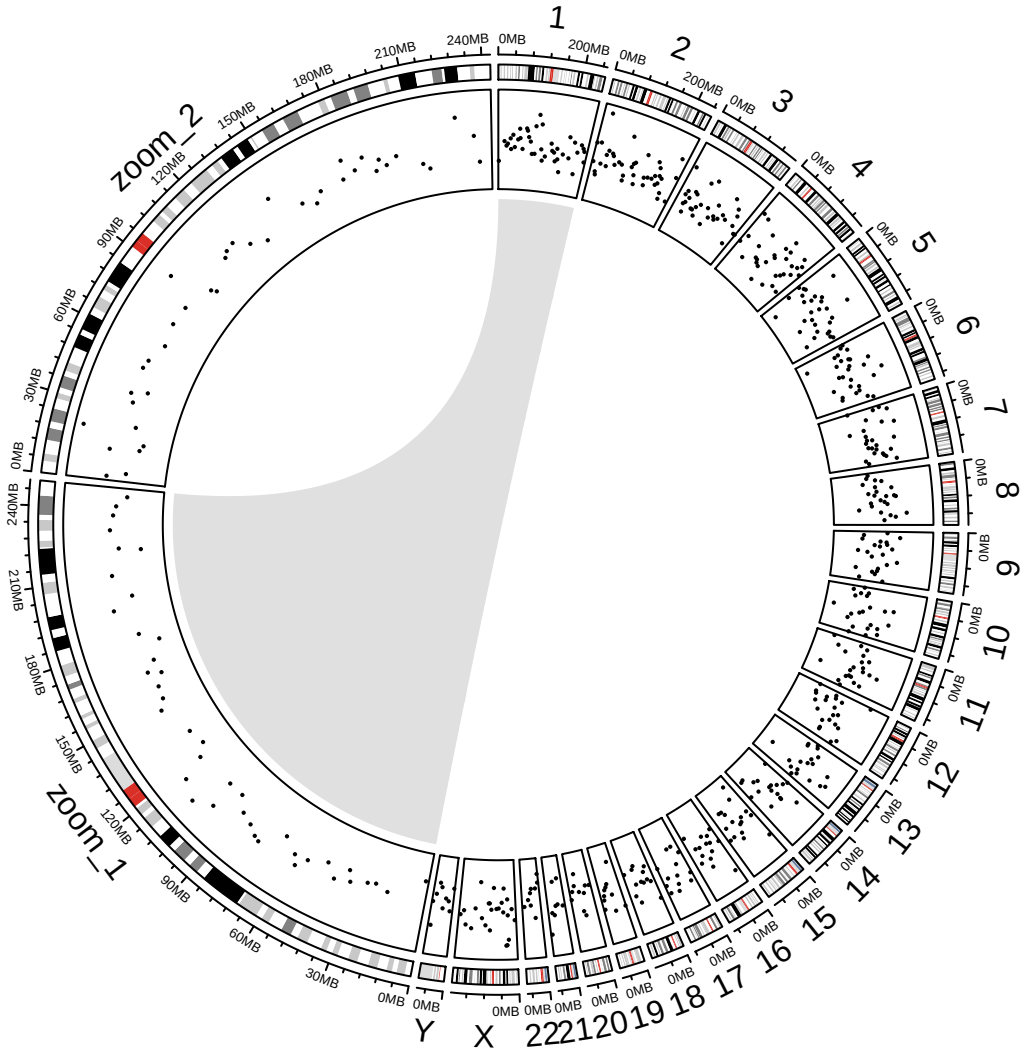


Figure 8.7: Zoom chromosomes.

Chapter 9

Create plotting regions for genomic data

Tracks are created and graphics are added by `circos.genomicTrackPlotRegions()`, or the short version `circos.genomicTrack()`. In following examples, chromosome will be used as the genomic category, and we assume `data` is simply a data frame in *BED* format (where the first column is the chromosome name, the second and third column are start and end positions, and the following columns are associated values). For more complex form of `data` and behaviour of the functions, we will introduce in Chapter 10.

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

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

Inside `panel.fun`, users can use low-level graphic functions to add basic graphics in each cell. `panel.fun` expects two arguments `region` and `value` which are automatically processed and passed from `circos.genomicTrack()`. `region` is a two-column data frame which only contains start position and end position in the current chromosome. `value` is also a data frame which contains other columns (start for the fourth column, if it exists). Thus, basically, `region` can be thought as values on x axes and `value` as values on y axes.

There should be a third arguments `...` which is mandatory and is used to pass user- invisible variables to inner functions and make magics (explained in Chapter 10). So whenever you use `panel.fun` in `circos.genomicTrack()`, please add it to the end of your function.

Following code demonstrates the values for `region` and `value` when used inside `panel.fun`.

```
bed = generateRandomBed(nc = 2)
head(bed, n = 2)
```

```
##    chr  start    end     value1      value2
## 1 chr1  39913 293918  0.6188539  0.07115116
## 2 chr1 333125 387791 -1.0251465 -1.01541848
```

```
circos.initializeWithIdeogram(plotType = NULL)
circos.genomicTrackPlotRegion(bed, panel.fun = function(region, value, ...) {
    if(CELL_META$sector.index == "chr1") {
        print(head(region, n = 2))
        print(head(value, n = 2))
```

```
    }
})
```

```
##    start    end
## 1  39913 293918
## 2 333125 387791
##       value1      value2
## 1  0.6188539  0.07115116
## 2 -1.0251465 -1.01541848
```

Since `circos.genomicTrack()` creates a new track, it needs values to calculate data ranges on y direction. Users can either specify the index of numeric columns in `data` by `numeric.column` (named index or numeric index, it can also be a vector with more than one columns) or directly set `ylim`. If none of them are set, the function will try to look for all numeric columns in `data` (of course, excluding the first three columns), and set them as `numeric.column`.

```
circos.genomicTrackPlotRegion(data, ylim = c(0, 1),
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, ...)
})
circos.genomicTrackPlotRegion(data, numeric.column = c("value1", "value2"),
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, ...)
})
```

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

As you already see in previous examples, **circlize** also provides low-level graphic functions specifically designed for genomic data. They are all implemented by corresponding normal circlize functions, but needs input variables with special format.

In this chapter, we introduce the basic usage of `circos.genomicTrack()` and low-level `circos.genomic*()`. In Chapter 10, we will introduce more usages of these functions, which are especially designed for genomic regions measured at multiple conditions. Example plots are shown together in Chapter 10.

9.1 Points

Usage of `circos.genomicPoints()` is similar as `circos.points()`. `circos.genomicPoints()` expects a two-column data frame which contains genomic regions and a data frame containing corresponding values. Points are always drawn at the middle of each region. The data column of the y values for plotting should be specified by `numeric.column`. If `numeric.column` has length larger than one, all the specified columns will be used for adding points.

If the function is called inside `circos.genomicTrack()` and users have been already set `numeric.column` in `circos.genomicTrack()`, proper value of `numeric.column` will be passed to `circos.genomicPoints()` through `...` in `panel.fun`, which means, you must add `...` as the final argument in `circos.genomicPoints()` to get such information. If `numeric.column` is not set in both places, `circos.genomicPoints()` will use all numeric columns detected in `value`.

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

Possible usages of `circos.genomicPoints()` are as follows.

```
circos.genomicPoints(region, value, numeric.column = c(1, 2))
circos.genomicPoints(region, value, cex, pch)
circos.genomicPoints(region, value, sector.index, track.index)
circos.genomicTrack(data, numeric.column = 4,
    panel.fun = function(region, value, ...) {
        # numeric.column is automatically passed to `circos.genomicPoints()`
        circos.genomicPoints(region, value, ...)
})
```

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

`circos.genomicPoints()` is simply implemented by `circos.points()`. The basic idea of the implementation is shown as following code, so, if you don't like the `circos.genomic*()` functions, it would not be difficult to directly use the `circos.*()` functions.

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

9.2 Lines

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

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

circlize additionally provides a new option `segment` for `lty` by which each genomic regions represent as 'horizontal' lines at y positions (see Figure 10.2, track H).

```
circos.genomicLines(region, value, lwd, lty = "segment")
```

9.3 Text

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

```
circos.genomicText(region, value, ...)
circos.genomicText(region, value, y = 1, labels)
circos.genomicText(region, value, numeric.column, labels.column)
circos.genomicText(region, value, facing, niceFacing, adj)
circos.genomicText(region, value, sector.index, track.index)
```

9.4 Rectangles

For `circos.genomicRect()`, Since the left and right of the rectangles are already determined by the start and end of the genomic regions, we only need to set the positions of top and bottom of the rectangles by specifying `ytop`, `ybottom` or `ytop.column`, `ybottom.column`.

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

9.5 Links

`circos.genomicLink()` expects two data frames and it adds links from genomic regions in the first data frame to corresponding genomic regions in the second data frame. All additional arguments are passed to `circos.link()`.

```
set.seed(123)
bed1 = generateRandomBed(nr = 100)
bed1 = bed1[sample(nrow(bed1), 20), ]
bed2 = generateRandomBed(nr = 100)
bed2 = bed2[sample(nrow(bed2), 20), ]

circos.initializeWithIdeogram()
circos.genomicLink(bed1, bed2, col = rand_color(nrow(bed1), transparency = 0.5),
    border = NA)
```

```
circos.clear()
```

9.6 Mixed use of general circlize functions

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

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

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

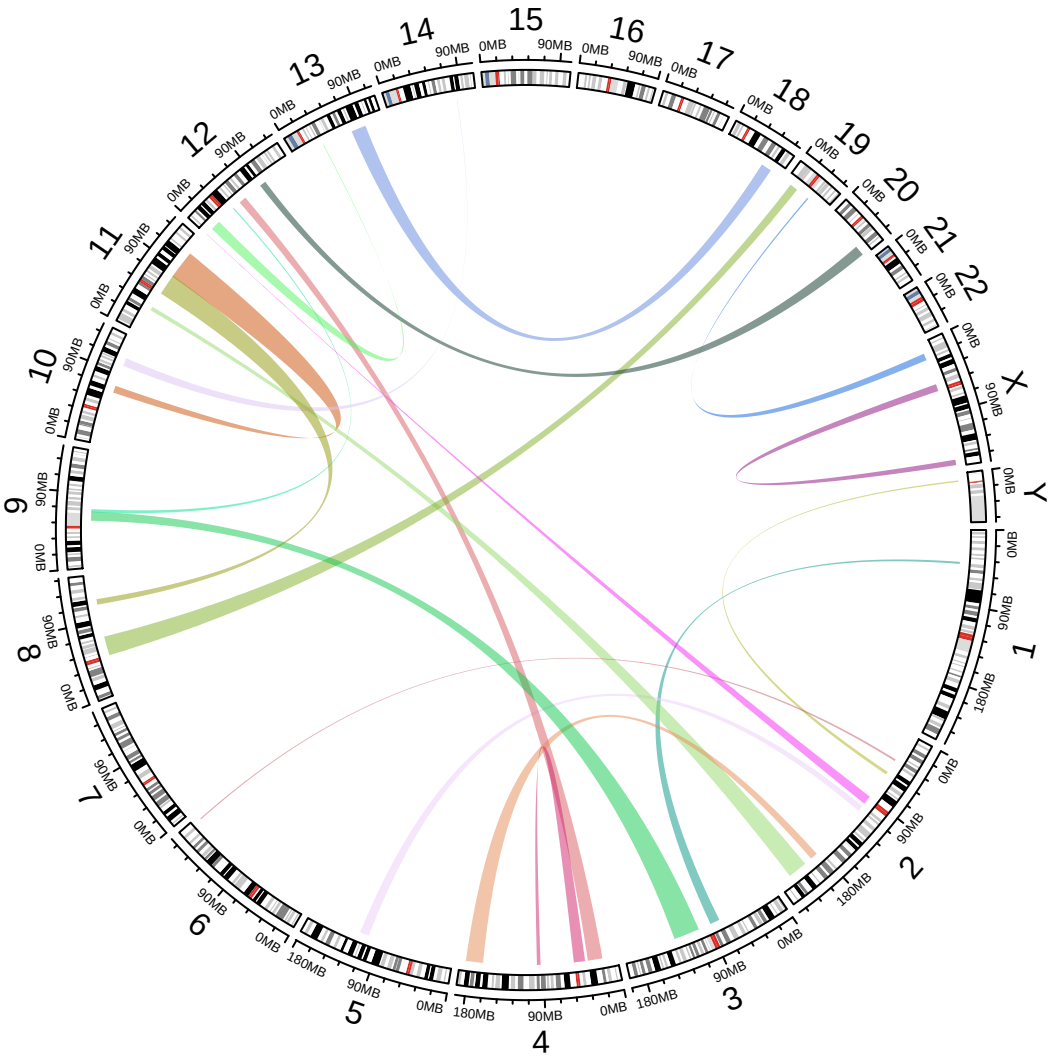


Figure 9.1: Add links from two sets of genomic regions.

Chapter 10

modes for `circos.genomicTrack()`

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

10.1 Normal mode

10.1.1 Input is a data frame

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

If there are more than one numeric columns, graphics are added for each column repeatedly (with same genomic positions).

```
data = generateRandomBed(nc = 2)
circos.genomicTrack(data, numeric.column = 4,
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, ...)
        circos.genomicPoints(region, value)
        # 1st column in `value` while 4th column in `data`
        circos.genomicPoints(region, value, numeric.column = 1)
})
```

10.1.2 Input is a list of data frames

If input data is a list of data frames, `panel.fun` is applied on each data frame iteratively to the current cell. Under such condition, `region` and `value` will contain corresponding data in the current data frame and in the current chromosome. The index for the current data frame can be get by `getI(...)`. Note `getI(...)` can only be used inside `panel.fun` and `...` argument is mandatory.

When `numeric.column` is specified in `circos.genomicTrack()`, the length of `numeric.column` can only be one or the number of data frames, which means, there is only one numeric column that will be used in each data frame. If it is not specified, the first numeric column in each data frame is used.

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

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

10.2 Stack mode

`circos.genomicTrack()` also supports a `stack` mode by setting `stack = TRUE`. Under `stack` mode, `ylim` is re-defined inside the function and the y-axis is splitted into several bins with equal height and graphics are put onto "horizontal" bins (with position `y = 1, 2, ...`).

10.2.1 Input is a data frame

Under `stack` mode, when input data is a single data frame containing one or more numeric columns, each numeric column defined in `numeric.column` will be treated as a single unit (recall that when `numeric.column` is not specified, all numeric columns are used). `ylim` is re-defined to `c(0.5, n+0.5)` in which `n` is number of numeric columns specified. `panel.fun` is applied iteratively on each numeric column and add graphics to the horizontal line `y = i`. In this case, actually `value` in e.g. `circos.genomicPoints()` doesn't used for mapping the y positions, while replaced with `y = i` internally.

In each iteration, in `panel.fun`, `region` is still the genomic regions in current chromosome, but `value` only contains current numeric column plus all non-numeric columns. The value of the index of "current" numeric column can be obtained by `getI(...)`.

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

10.2.2 Input is a list of data frames

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

```
bed_list = list(generateRandomBed(), generateRandomBed())
circos.genomicTrack(bed_list, stack = TRUE,
    panel.fun = function(region, value, ...) {
```

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

Under `stack` mode, if using a data frame with multiple numeric columns, graphics on all horizontal bins share the same genomic positions while if using a list of data frames, the genomic positions can be different.

10.3 Applications

In this section, we will show several real examples of adding genomic graphics under different modes. Again, if you are not happy with these functionalities, you can simply re-implement your plot with the basic circlize functions.

10.3.1 Points

To make plots more clear to look at, we only add graphics in the first quarter of the circle and initialize the plot only with chromosome 1.

```
set.seed(999)
circos.par("track.height" = 0.1, start.degree = 90,
    canvas.xlim = c(0, 1), canvas.ylim = c(0, 1), gap.degree = 270)
circos.initializeWithIdeogram(chromosome.index = "chr1", plotType = NULL)
```

In the example figure (Figure 10.1) below, each track contains points under different modes.

In track A, it is the most normal way to add points. Here `bed` only contains one numeric column and points are added at the middle points of regions.

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

In track B, if it is specified as `stack` mode, points are added in a horizontal line (or visually, a circular line).

```
circos.genomicTrack(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        circos.genomicPoints(region, value, pch = 16, cex = 0.5,...)
        i = getI(...)
        circos.lines(CELL_META$cell.xlim, c(i, i), lty = 2, col = "#00000040")
})
```

In track C, the input data is a list of two data frames. `panel.fun` is applied iterately on each data frame. The index of "current" index can be obtained by `getI(...)`.

```
bed1 = generateRandomBed(nr = 300)
bed2 = generateRandomBed(nr = 300)
bed_list = list(bed1, bed2)
circos.genomicTrack(bed_list,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicPoints(region, value, pch = 16, cex = 0.5, col = i, ...)
})
```

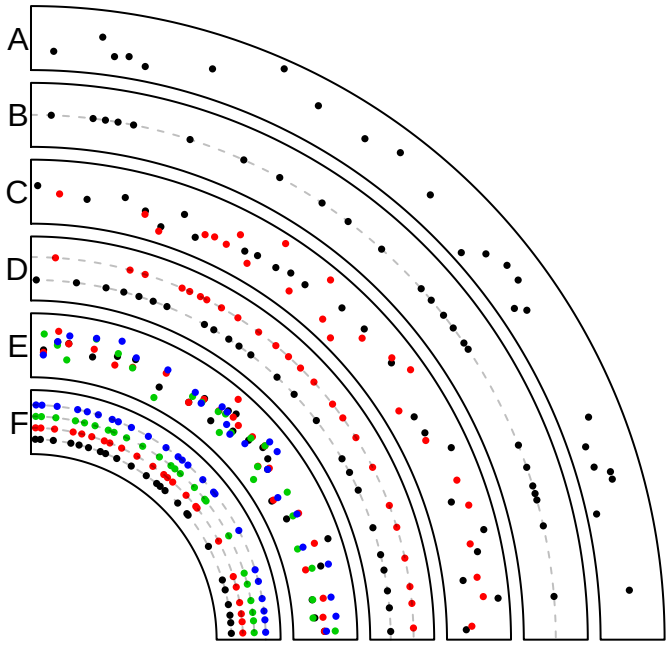


Figure 10.1: Add points under different modes.

In track D, the list of data frames is plotted under `stack` mode. Graphics corresponding to each data frame are added to a horizontal line.

```
circos.genomicTrack(bed_list, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicPoints(region, value, pch = 16, cex = 0.5, col = i, ...)
        circos.lines(CELL_META$cell.xlim, c(i, i), lty = 2, col = "#00000040")
})
```

In track E, the data frame has four numeric columns. Under normal mode, all the four columns are used with the same genomic coordinates.

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

In track F, the data frame has four columns but is plotted under `stack` mode. Graphics for each column are added to a horizontal line. Current column can be obtained by `getI(...)`. Note here `value` in `panel.fun` is a data frame with only one column (which is the current numeric column).

```
bed = generateRandomBed(nr = 300, nc = 4)
circos.genomicTrack(bed, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicPoints(region, value, pch = 16, cex = 0.5, col = i, ...)
        circos.lines(CELL_META$cell.xlim, c(i, i), lty = 2, col = "#00000040")
})
circos.clear()
```

10.3.2 Lines

Similar as previous figure, only the first quarter in the circle is visualized. Examples are shown in Figure 10.2.

```
circos.par("track.height" = 0.08, start.degree = 90,
    canvas.xlim = c(0, 1), canvas.ylim = c(0, 1), gap.degree = 270,
    cell.padding = c(0, 0, 0, 0))
circos.initializeWithIdeogram(chromosome.index = "chr1", plotType = NULL)
```

In track A, it is the most simple way to add lines. Middle points of regions are used as the values on x-axes.

```
bed = generateRandomBed(nr = 500)
circos.genomicTrack(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicLines(region, value)
})
```

`circos.genomicLines()` is implemented by `circos.lines()`, thus, arguments supported in `circos.lines()` can also be in `circos.genomicLines()`. In track B, the area under the line is filled with color and in track C, type of the line is set to `h`.

```
circos.genomicTrack(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicLines(region, value, area = TRUE)
})
circos.genomicTrack(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicLines(region, value, type = "h")
})
```

In track D, the input is a list of data frames. `panel.fun` is applied to each data frame iterately.

```
bed1 = generateRandomBed(nr = 500)
bed2 = generateRandomBed(nr = 500)
bed_list = list(bed1, bed2)
circos.genomicTrack(bed_list,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicLines(region, value, col = i, ...)
})
```

In track E, the input is a list of data frames and is drawn under `stack` mode. Each genomic region is drawn as a horizontal segment and is put on a horizontal line where the width of the segment corresponds to the width of the genomc region. Under `stack` mode, for `circos.genomicLines()`, type of lines is only restricted to segments.

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

In track F, the input is a data frame with four numeric columns. Each column is drawn under the normal mode where the same genomic coordinates are shared.

```
bed = generateRandomBed(nr = 500, nc = 4)
circos.genomicTrack(bed,
```

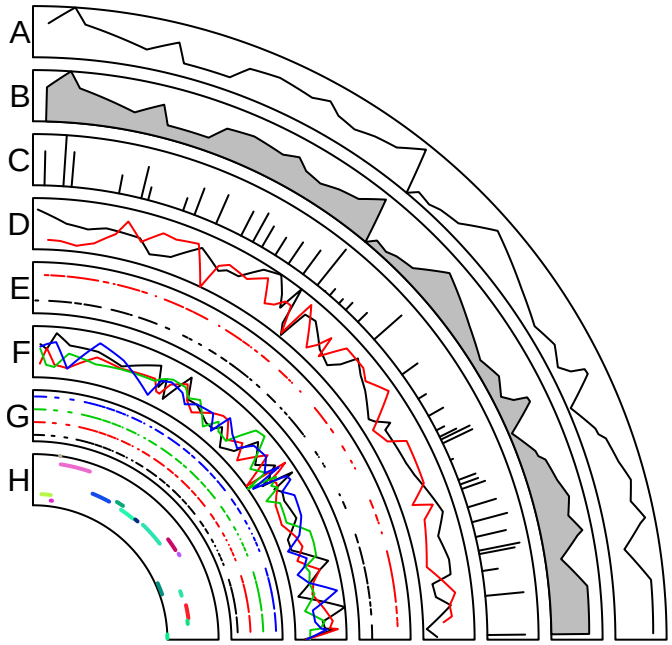


Figure 10.2: Add lines under different modes.

```
    panel.fun = function(region, value, ...) {
        circos.genomicLines(region, value, col = 1:4, ...)
})
```

In track G, the data frame with four numeric columns are drawn under `stack` mode. All the four columns are drawn to four horizontal lines.

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

In track H, we specify `type` to `segment` and set different colors for segments. Note each segment is located at the y position defined in the numeric column.

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

10.3.3 Rectangles

Again, only the first quarter of the circle is initialized. For rectangles, the filled colors are always used to represent numeric values. Here we define a color mapping function `col_fun` to map values to colors. Examples are in Figure 10.3.

```
circos.par("track.height" = 0.15, start.degree = 90,
    canvas.xlim = c(0, 1), canvas.ylim = c(0, 1), gap.degree = 270)
circos.initializeWithIdeogram(chromosome.index = "chr1", plotType = NULL)
col_fun = colorRamp2(breaks = c(-1, 0, 1), colors = c("green", "black", "red"))
```

To draw heatmaps, you probably want to use the `stack` mode. In track A, `bed` has four numeric columns and `stack` mode is used to arrange the heatmap. You can see rectangles are stacked for a certain genomic region.

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

In track B, the input is a list of data frames. Under `stack` mode, each data frame is added to a horizontal line. Since genomic positions for different data frames can be different, you may see in the figure, positions for the two sets of rectangles are different.

Under `stack` mode, by default, the height of rectangles is internally set to make them completely fill the cell in the vertical direction. `ytop` and `ybottom` can be used to adjust the height of rectangles. Note each line of rectangles is at `y = i` and the default height of rectangles are 1.

```
bed1 = generateRandomBed(nr = 100)
bed2 = generateRandomBed(nr = 100)
bed_list = list(bed1, bed2)
circos.genomicTrack(bed_list, stack = TRUE,
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicRect(region, value, ytop = i + 0.3, ybottom = i - 0.3,
            col = col_fun(value[[1]]), ...)
})
```

In track C, we implement same graphics as in track B, but with the normal mode. Under `stack` mode, data range on y axes and positions of rectangles are adjusted internally. Here we explicitly adjust it under the normal mode.

```
circos.genomicTrack(bed_list, ylim = c(0.5, 2.5),
    panel.fun = function(region, value, ...) {
        i = getI(...)
        circos.genomicRect(region, value, ytop = i + 0.3, ybottom = i - 0.3,
            col = col_fun(value[[1]]), ...)
})
```

In track D, rectangles are used to make barplots. We specify the position of the top of bars by `ytop.column` (1 means the first column in `value`).

```
bed = generateRandomBed(nr = 200)
circos.genomicTrack(bed,
    panel.fun = function(region, value, ...) {
        circos.genomicRect(region, value, ytop.column = 1, ybottom = 0,
            col = ifelse(value[[1]] > 0, "red", "green"), ...)
        circos.lines(CELL_META$cell.xlim, c(0, 0), lty = 2, col = "#00000040")
})
circos.clear()
```

Figure 10.3: Add rectangles under different modes.

Chapter 11

High-level genomic functions

In this chapter, several high-level functions which create tracks are introduced.

11.1 Ideograms

`circos.initializeWithIdeogram()` initializes the circular plot and adds ideogram track if the cytoband data is available. Actually, the ideograms are drawn by `circos.genomicIdeogram()`. `circos.genomicIdeogram()` creates a small track of ideograms and can be used anywhere in the circle. By default it adds ideograms for human genome hg19 (Figure 11.1).

```
circos.initializeWithIdeogram(plotType = c("labels", "axis"))
circos.track(ylim = c(0, 1))
circos.genomicIdeogram() # put ideogram as the third track
circos.genomicIdeogram(track.height = 0.2)
```

11.2 Heatmaps

Matrix which corresponds to genomic regions can be visualized as heatmaps. Heatmaps completely fill the track and there are connection lines connecting heatmaps and original positions in the genome. 'circos.genomicHeamtap()" draws connection lines and heatmaps as two tracks and combines them as an integrated track.

Generally, all numeric columns (excluding the first three columns) in the input data frame are used to make the heatmap. Columns can also be specified by `numeric.column` which is either an numeric vector or a character vector. Colors can be specfied as a color matrix or a color mapping function generated by `colorRamp2()`.

The height of the connection line track and the heatmap track can be set by `connection_height` and `heatmap_height` arguments. Also parameters for the styles of lines and rectangle borders can be adjusted, please check the help page of `circos.genomicHeatmap()`.

```
circos.initializeWithIdeogram()
bed = generateRandomBed(nr = 100, nc = 4)
col_fun = colorRamp2(c(-1, 0, 1), c("green", "black", "red"))
circos.genomicHeatmap(bed, col = col_fun, side = "inside", border = "white")
circos.clear()
```

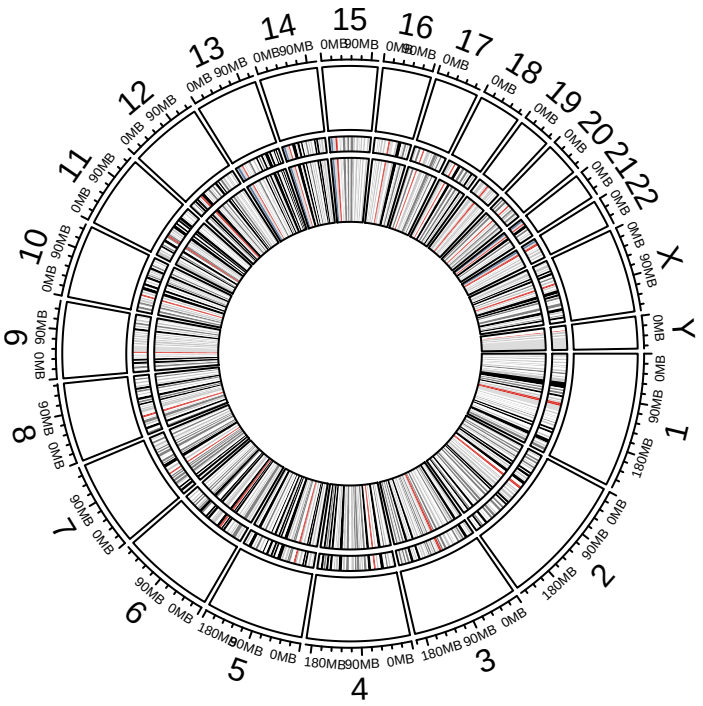


Figure 11.1: Circular ideograms.

In the left figure in Figure 11.2, the heatmaps are put inside the normal genomic track. Heatmaps are also be put outside the normal genomic track by setting `side = "outside"` (Figure 11.2, right).

```
circos.initializeWithIdeogram(plotType = NULL)
circos.genomicHeatmap(bed, col = col_fun, side = "outside",
    line_col = as.numeric(factor(bed[[1]])))
circos.genomicIdeogram()
circos.clear()
```

11.3 Labels

`circos.genomicLabels()` adds text labels for regions that are specified. Positions of labels are automatically adjusted so that they do not overlap to each other.

Similar as `circos.genomicHeatmap()`, `circos.genomicLabels()` also creates two tracks where one for the connection lines and one for the labels. You can set the height of the labels track to be the maximum width of labels by `labels_height = max(strwidth(labels))`. `padding` argument controls the gap between two neighbouring labels.

```
circos.initializeWithIdeogram()
bed = generateRandomBed(nr = 50, fun = function(k) sample(letters, k, replace = TRUE))
bed[1, 4] = "aaaaa"
circos.genomicLabels(bed, labels.column = 4, side = "inside")
circos.clear()
```

Similarlly, labels can be put outside of the normal genomic track (Figure 11.3 right).

```
circos.initializeWithIdeogram(plotType = NULL)
circos.genomicLabels(bed, labels.column = 4, side = "outside",
```

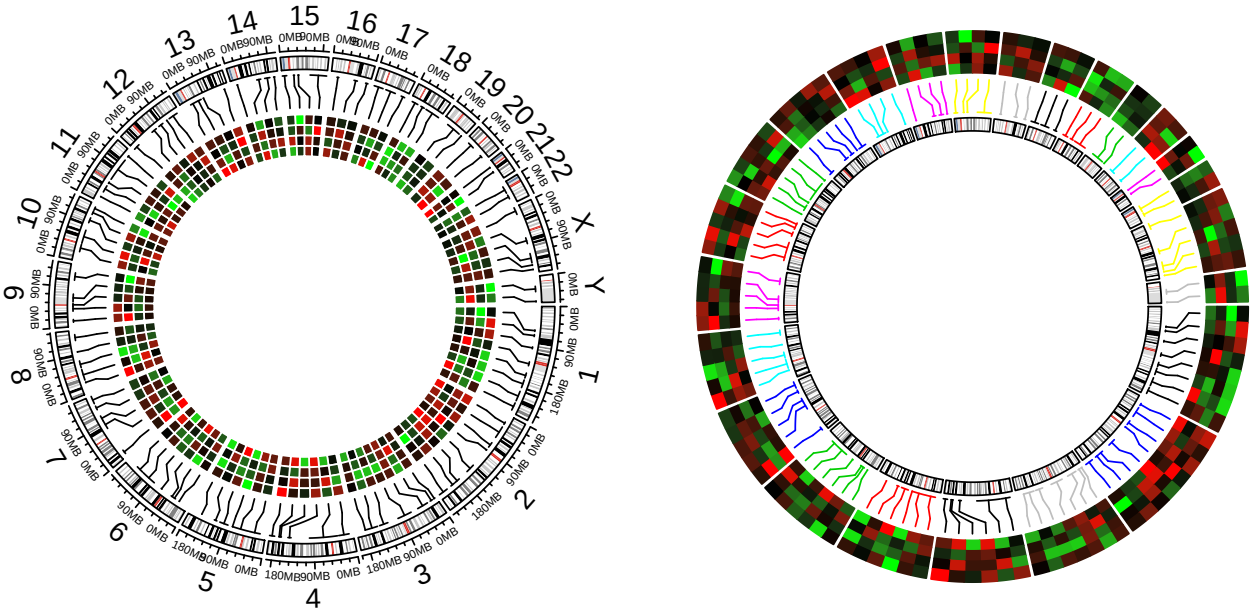


Figure 11.2: Genomic heamtaps.

```
    col = as.numeric(factor(bed[[1]])), line_col = as.numeric(factor(bed[[1]])))
circos.genomicIdeogram()
circos.clear()
```

11.4 Genomic density and Rainfall plot

Rainfall plots are used to visualize the distribution of genomic regions in the genome. Rainfall plots are particularly useful to identify clusters of regions. In the rainfall plot, each dot represents a region. The x-axis corresponds to the genomic coordinate, and the y-axis corresponds to the minimal distance (log10 transformed) of the region to its two neighbouring regions. A cluster of regions will appear as a "rainfall" in the plot.

`circos.genomicRainfall()` calculates neighbouring distance for each region and draw as points on the plot. Since `circos.genomicRainfall()` generates data on y-direction (`log10(distance)`), it is actually a high-level function which creates a new track.

The input data can be a single data frame or a list of data frames.

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

However, if the amount of regions in a cluster is high, dots will overlap, and direct assessment of the number and density of regions in the cluster will be impossible. To overcome this limitation, additional tracks are added which visualize the genomic density of regions (defined as the fraction of a genomic window that is covered by genomic regions).

`circos.genomicDensity()` calculates how much a genomic window is covered by regions in `bed`. It is also a high-level function and creates a new track.

The input data can be a single data frame or a list of data frames.

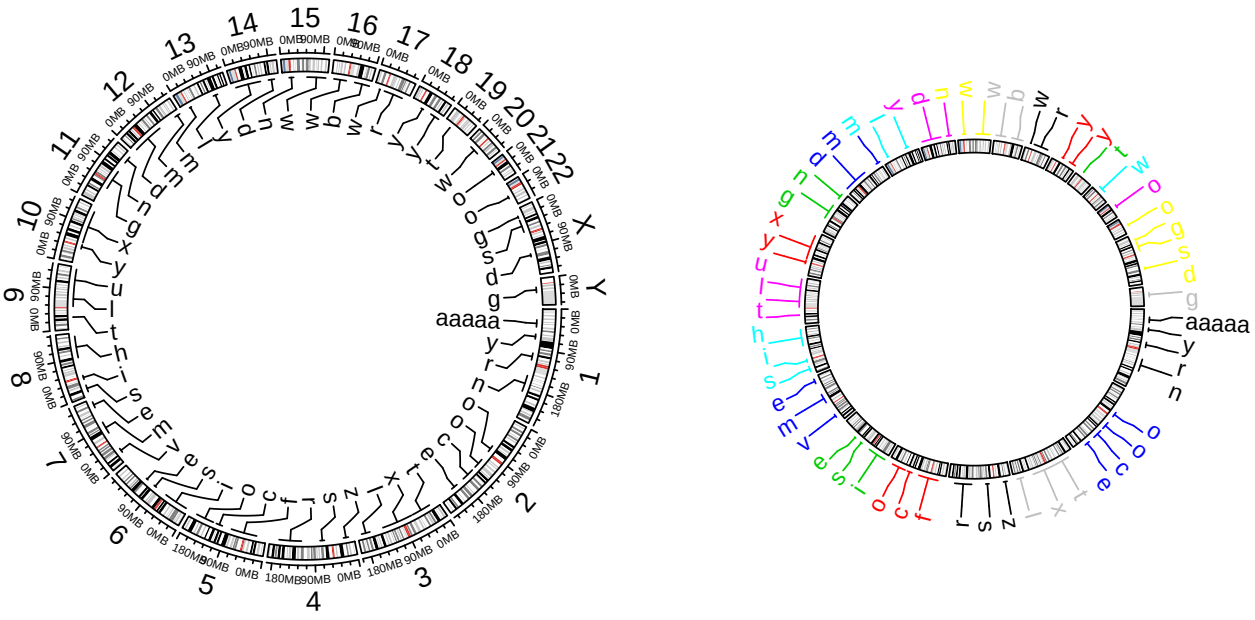


Figure 11.3: Genomic labels.

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

Following example makes a rainfall plot for differentially methylated regions (DMR) and their genomic densities. In the plot, red corresponds to hyper-methylated DMRs (gain of methylation) and blue corresponds to hypo-methylated DMRs (loss of methylation). You may see how the combination of rainfall track and genomic density track helps to get a more precise inference of the distribution of DMRs on genome (Figure 11.4).

```
load(system.file(package = "circlize", "extdata", "DMR.RData"))
circos.initializeWithIdeogram(chromosome.index = paste0("chr", 1:22))

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

```
circos.clear()
```

Internally, `rainfallTransform()` and `genomicDensity()` are used to the neighbrouing distance and the genomic density values.

```
head(rainfallTransform(DMR_hyper))
```

```
##       chr   start     end  dist
## 70   chr1  933445  934443 35323
## 104  chr1  969766  970362  4909
## 105  chr1  975271  976767  4909
## 154  chr1 1108819 1109923 31522
## 155  chr1 1141445 1142405 31522
## 157  chr1 1181550 1182782 39145
```

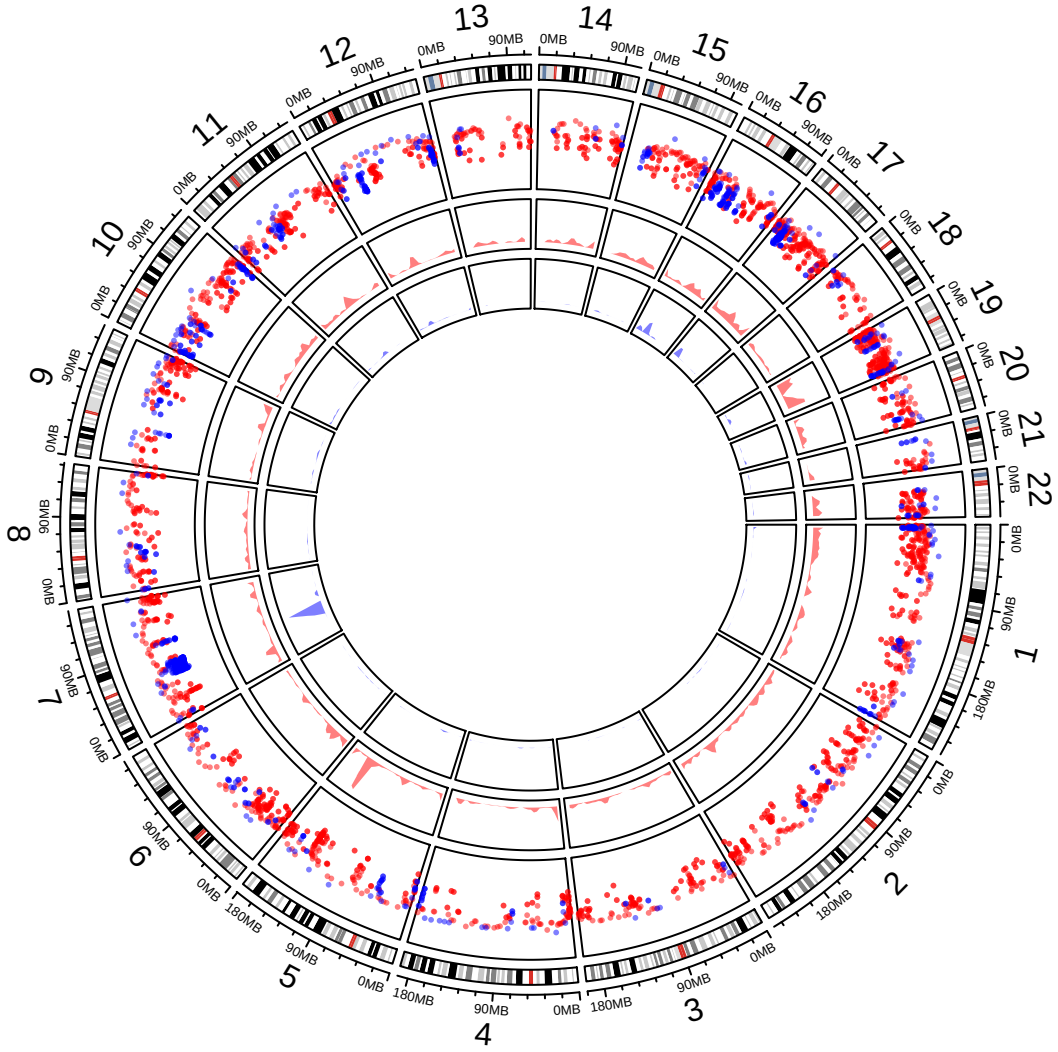


Figure 11.4: Genomic rainfall plot and densities.

```
head(genomicDensity(DMR_hyper, window.size = 1e6))
```

```
##    chr   start     end      pct
## 1 chr1       1 1000000 0.003093
## 2 chr1  500001 1500000 0.007592
## 3 chr1 1000001 2000000 0.008848
## 4 chr1 1500001 2500000 0.010155
## 5 chr1 2000001 3000000 0.011674
## 6 chr1 2500001 3500000 0.007783
```

Chapter 12

Nested zooming

12.1 Basic idea

In Section 6.1 we introduced how to zoom sectors to the same circle in the same track. This works fine if there are only a few regions that need to be zoomed. However, when the regions that need to be zoomed is too many, the method will not work efficiently. In this chapter, we introduce another zooming method which puts zoomed regions in a different circular plot.

To illustrate the basic idea, we first generate a random data set.

```
set.seed(123)
df = data.frame(cate = sample(letters[1:8], 400, replace = TRUE),
                x = runif(400),
                y = runif(400),
                stringsAsFactors = FALSE)
df = df[order(df[[1]], df[[2]]), ]
rownames(df) = NULL
df$interval_x = as.character(cut(df$x, c(0, 0.2, 0.4, 0.6, 0.8, 1.0)))
df$name = paste(df$cate, df$interval_x, sep = ":")
df$start = as.numeric(gsub("^\\((\\d(\\.\\d)?).*(\\d(\\.\\d)?)]", "\\1", df$interval_x))
df$end = as.numeric(gsub("^\\((\\d(\\.\\d)?),(\\d(\\.\\d)?)]$", "\\3", df$interval_x))
nm = sample(unique(df$name), 20)
df2 = df[df$name %in% nm, ]

correspondance = unique(df2[, c("cate", "start", "end", "name", "start", "end")])
zoom_sector = unique(df2[, c("name", "start", "end", "cate")])
zoom_data = df2[, c("name", "x", "y")]

data = df[, 1:3]
sector = data.frame(cate = letters[1:8], start = 0, end = 1, stringsAsFactors = FALSE)

sector_col = structure(rand_color(8, transparency = 0.5), names = letters[1:8])
```

Following variables are used for downstream visualization. `sector` contains sector names and coordinate at the x direction:

```
head(sector, n = 4)
```

```
##   cate start end
## 1    a     0   1
```

```
## 2    b     0   1
## 3    c     0   1
## 4    d     0   1
```

`data` contains data points for a track.

```
head(data, n = 4)
```

```
##   cate          x         y
## 1    a 0.02552670 0.5546818
## 2    a 0.04210805 0.8112373
## 3    a 0.04953844 0.3809037
## 4    a 0.05819180 0.7151335
```

In `sector`, we randomly sampled several intervals which will be used for zooming. The zoomed intervals are stored in `zoom_sector`. In the zooming track, each interval is treated as an independent sector, thus, the name for each zoomed interval uses combination of the original sector name and the interval itself, just for easily reading.

```
head(zoom_sector, n = 4)
```

```
##           name start end cate
## 16 a:(0.4,0.6]   0.4 0.6    a
## 37   a:(0.8,1]   0.8 1.0    a
## 43   b:(0,0.2]   0.0 0.2    b
## 64 b:(0.4,0.6]   0.4 0.6    b
```

And the subset of data which are in the zoomed intervals.

```
head(zoom_data, n = 4)
```

```
##           name         x          y
## 16 a:(0.4,0.6] 0.4122831 0.28695762
## 17 a:(0.4,0.6] 0.4286131 0.28869011
## 18 a:(0.4,0.6] 0.4361300 0.04236068
## 19 a:(0.4,0.6] 0.4498025 0.96864117
```

The correspondance between the original sectors and the zoomed intervals are in `correspondance`. The value is a data frame with six columns. The value is the position of the intervals in the second circular plot in the first plot.

```
head(correspondance, n = 4)
```

```
##    cate start end        name start.1 end.1
## 16    a   0.4 0.6 a:(0.4,0.6]     0.4   0.6
## 37    a   0.8 1.0   a:(0.8,1]     0.8   1.0
## 43    b   0.0 0.2   b:(0,0.2]     0.0   0.2
## 64    b   0.4 0.6 b:(0.4,0.6]     0.4   0.6
```

The zooming is actually composed of two circulat plots where one for the original track and one for the zoomed intervals. There is an additional connection track which identifies which intervals that are zoomed belong to which sector. The `circos.nested()` function in **circlize** puts the two circular plots together, arranges them and automatically draws the connection lines.

To define "the circular plot", the code for generating the plot needs to be wrapped into a function.

```
f1 = function() {
    circos.par(gap.degree = 10)
    circos.initialize(sector[, 1], xlim = sector[, 2:3])
    circos.track(data[[1]], x = data[[2]], y = data[[3]], ylim = c(0, 1),
```

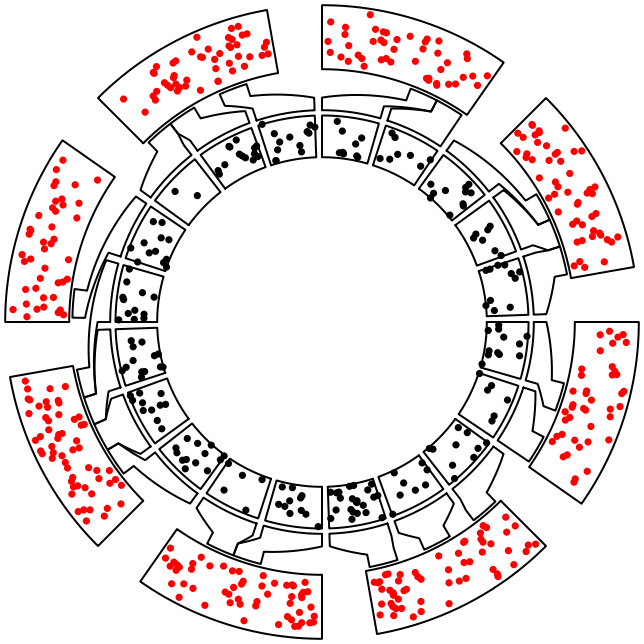


Figure 12.1: Nested zooming between two circular plots.

```
        panel.fun = function(x, y) {
            circos.points(x, y, pch = 16, cex = 0.5, col = "red")
    })
}

f2 = function() {
    circos.par(gap.degree = 2, cell.padding = c(0, 0, 0, 0))
    circos.initialize(zoom_sector[[1]], xlim = as.matrix(zoom_sector[, 2:3]))
    circos.track(zoom_data[[1]], x = zoom_data[[2]], y = zoom_data[[3]],
        panel.fun = function(x, y) {
            circos.points(x, y, pch = 16, cex = 0.5)
        })
}
```

In above, `f1()` is the code for generating the original plot and `f2()` is the code for generating the zoomed plot.

To combine the two plots, simply put `f1()`, `f2()` and `corresponance` to `circos.nested()` (Figure 12.1).

```
circos.nested(f1, f2, correspondance)
```

In the plot, the zoomed circle is put inside the original circle and the start degree for the second plot is automatically adjusted.

Zoomed circle can also be put outside by swtiching `f1()` and `f2()`. Actually, for `circos.nested()`, It doesn't care which one is zoomed or not, they are just two circular plots and a correspondance (Figure 12.2).

```
circos.nested(f2, f1, correspondance[, c(4:6, 1:3)])
```

There are some points that need to be noted while doing nested zoomings:

1. It can only be applied to the full circle.

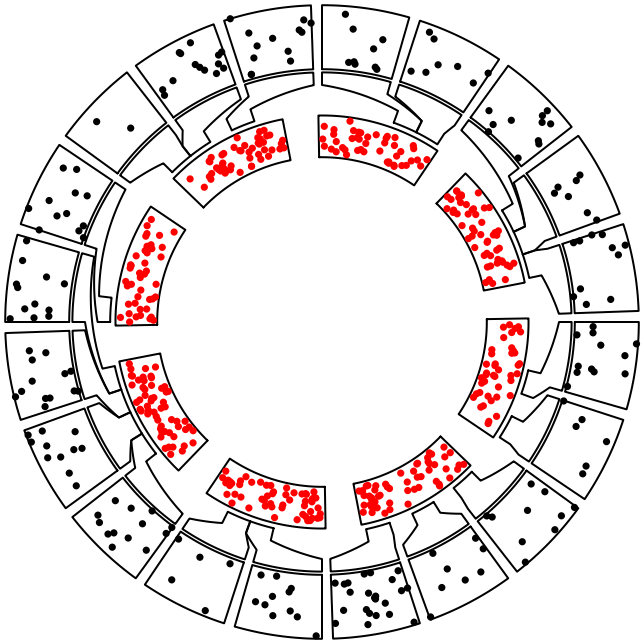


Figure 12.2: Nested zooming between two circular plots, zoomed plot is put outside.

2. If `canvas.xlim` and `canvas.ylim` are adjusted in the first plot, same value should be set to the second plot.
3. By default the start degree of the second plot is automatically adjusted to make the difference between the original positions and zoomed sectors to a minimal. However, users can also manually adjusted the start degree for the second plot by setting `circos.par("start.degree" = ...)` and `adjust_start_degree` must be set to `TRUE` in `circos.nested()`.
4. Since the function needs to know some information for the two circular plots, do not put `circos.clear()` at the end of each plot. They will be added internally.

`f1()` and `f2()` are just normal code for implementing circular plot. There is no problem to make it more complex (Figure 12.3).

```
sector_col = structure(rand_color(8, transparency = 0.5), names = letters[1:8])

f1 = function() {
    circos.par(gap.degree = 10)
    circos.initialize(sector[, 1], xlim = sector[, 2:3])
    circos.track(data[[1]], x = data[[2]], y = data[[3]], ylim = c(0, 1),
        panel.fun = function(x, y) {
            l = correspondance[[1]] == CELL_META$sector.index
            if(sum(l)) {
                for(i in which(l)) {
                    circos.rect(correspondance[i, 2], CELL_META$cell.ylim[1],
                                correspondance[i, 3], CELL_META$cell.ylim[2],
                                col = sector_col[CELL_META$sector.index],
                                border = sector_col[CELL_META$sector.index])
                }
            }
            circos.points(x, y, pch = 16, cex = 0.5)
            circos.text(CELL_META$xcenter, CELL_META$ylim[2] + uy(2, "mm"),
                CELL_META$sector.index, niceFacing = TRUE, adj = c(0.5, 0))
```

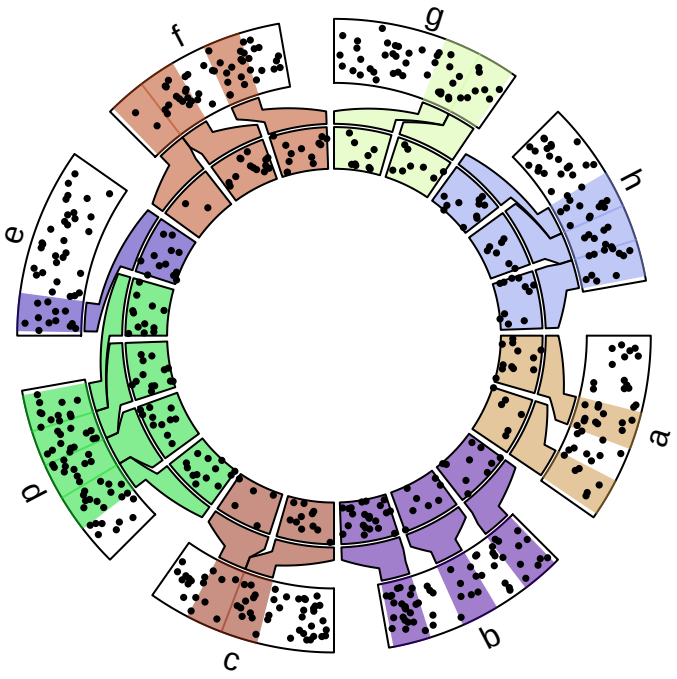


Figure 12.3: Nested zooming between two circular plots, slightly complex plots.

```
    })
}

f2 = function() {
    circos.par(gap.degree = 2, cell.padding = c(0, 0, 0, 0))
    circos.initialize(zoom_sector[[1]], xlim = as.matrix(zoom_sector[, 2:3]))
    circos.track(zoom_data[[1]], x = zoom_data[[2]], y = zoom_data[[3]],
        panel.fun = function(x, y) {
            circos.points(x, y, pch = 16, cex = 0.5)
        }, bg.col = sector_col[zoom_sector$cate],
        track.margin = c(0, 0))
}
circos.nested(f1, f2, correspondance, connection_col = sector_col[correspondance[[1]]])
```

12.2 Visualization of DMRs from tagmentation-based WGBS

Tagmentation-based whole-genome bisulfite sequencing (T-WGBS) is a technology which can examine only a minor fraction of methylome of interest. In this section, we demonstrate how to visualize DMRs which are detected from T-WGBS data in a circular plot by **circlize** package.

In the example data loaded, `tagments` contains regions which are sequenced, `DMR1` contains DMRs for one patient detectd in tagment regions. Correspondance between tagment regions and original genome is stored in `correspondance`.

```
load(system.file(package = "circlize", "extdata", "tagments_WGBS_DMR.RData"))
head(tagments, n = 4)
```

```
##                  tagments     start       end  chr
```

```
## 1   chr1-44876009-45016546  44876009  45016546 chr1
## 2   chr1-90460304-90761641  90460304  90761641 chr1
## 3 chr1-211666507-211692757 211666507 211692757 chr1
## 4   chr2-46387184-46477385  46387184  46477385 chr2
```

```
head(DMR1, n = 4)
```

```
##                      chr    start      end   methDiff
## 1 chr1-44876009-45016546 44894352 44894643 -0.2812889
## 2 chr1-44876009-45016546 44902069 44902966 -0.3331170
## 3 chr1-90460304-90761641 90535428 90536046 -0.3550701
## 4 chr1-90460304-90761641 90546991 90547262 -0.4310808
```

```
head(correspondance, n = 4)
```

```
##    chr     start       end                 tagments   start.1     end.1
## 1 chr1  44876009  45016546   chr1-44876009-45016546  44876009  45016546
## 2 chr1  90460304  90761641   chr1-90460304-90761641  90460304  90761641
## 3 chr1 211666507 211692757 chr1-211666507-211692757 211666507 211692757
## 4 chr2  46387184  46477385   chr2-46387184-46477385  46387184  46477385
```

In Figure 12.4, the tagment regions are actually zoomed from the genome. In following code, `f1()` only makes a circular plot with whole genome and `f2()` makes circular plot for the tagment regions.

```
chr_bg_color = rand_color(22, transparency = 0.8)
names(chr_bg_color) = paste0("chr", 1:22)

f1 = function() {
    circos.par(gap.after = 2, start.degree = 90)
    circos.initializeWithIdeogram(chromosome.index = paste0("chr", 1:22),
        plotType = c("ideogram", "labels"), ideogram.height = 0.03)
}

f2 = function() {
    circos.par(cell.padding = c(0, 0, 0, 0), gap.after = c(rep(1, nrow(tagments)-1), 10))
    circos.genomicInitialize(tagments, plotType = NULL)
    circos.genomicTrack(DMR1, ylim = c(-0.6, 0.6),
        panel.fun = function(region, value, ...) {
            for(h in seq(-0.6, 0.6, by = 0.2)) {
                circos.lines(CELL_META$cell.xlim, c(h, h), lty = 3, col = "#AAAAAA")
            }
            circos.lines(CELL_META$cell.xlim, c(0, 0), lty = 3, col = "#888888")

            circos.genomicPoints(region, value,
                col = ifelse(value[[1]] > 0, "#E41A1C", "#377EB8"),
                pch = 16, cex = 0.5)
    }, bg.col = chr_bg_color[tagments$chr], track.margin = c(0.02, 0))
    circos.yaxis(side = "left", at = seq(-0.6, 0.6, by = 0.3),
        sector.index = get.all.sector.index()[1], labels.cex = 0.4)
    circos.track(ylim = c(0, 1), track.height = uh(2, "mm"),
        bg.col = add_transparency(chr_bg_color[tagments$chr], 0))
}

circos.nested(f1, f2, correspondance, connection_col = chr_bg_color[correspondance[[1]]])
```

Figure 12.4: Visualization of DMRs.

Part III

Visualize Relations

Chapter 13

The `chordDiagram()` function

One unique feature of circular layout is the circular visualization of relations between objects by links. See examples in http://circos.ca/intro/tabular_visualization/. The name of such plot is called Chord diagram. In **circlize**, it is easy to plot Chord diagram in a straightforward or in a highly customized way.

There are two data formats that represent relations, either adjacency matrix or adjacency list. In adjacency matrix, value in i^{th} row and j^{th} column represents the relation from object in the i^{th} row and the object in the j^{th} column where the absolute value measures the strength of the relation. In adjacency list, relations are represented as a three-column data frame in which relations come from the first column and to the second column, and the third column represents the strength of the relation.

Following code shows an example of an adjacency matrix.

```
mat = matrix(1:9, 3)
rownames(mat) = letters[1:3]
colnames(mat) = LETTERS[1:3]
mat
```

```
##   A B C
## a 1 4 7
## b 2 5 8
## c 3 6 9
```

And the code in below is an example of a adjacency list.

```
df = data.frame(from = letters[1:3], to = LETTERS[1:3], value = 1:3)
df
```

```
##   from to value
## 1    a  A     1
## 2    b  B     2
## 3    c  C     3
```

Actually, it is not difficult to convert between these two formats. There are also R packages and functions do the conversion such as in **reshape2** package, `melt()` converts a matrix to a data frame and `dcast()` converts the data frame back to the matrix.

Chord diagram shows the information of the relation from several levels. 1. the links are straightforward to show the relations between objects; 2. width of links are proportional to the strength of the relation which is more illustrative than other graphic mappings; 3. colors of links can be another graphic mapping for relations; 4. width of sectors represents total strength for an object which connects to other objects or is connected from other objects. You can find an interesting example of using Chord diagram to visualize leagues system of players clubs by their national team from https://gjabel.wordpress.com/2014/06/05/

world-cup-players-representation-by-league-system/ and the adapted code is at http://jokergoo.github.io/circlize/example/wc2014.html.

In **circlize** package, there is a `chordDiagram()` function that supports both adjacency matrix and adjacency list. For different formats of input, the corresponding format of graphic parameters will be different either. E.g. when the input is a matrix, since information of the links in the Chord diagram is stored in the matrix, corresponding graphics for the links sometimes should also be specified as a matrix, while if the input is a data frame, the graphic parameters for links only need to be specified as an additional column to the data frame. However, in many cases, adjacency matrix is directly generated from upstream analysis and converting it into a adjacency list does not make sense, e.g. converting a correlation matrix to a adjacency list is obviously a bad idea. Thus, in this chapter, we will show usage for both adjacency matrix and list.

Since the usage for the two types of inputs are highly similar, in this chapter, we mainly show figures generated from matrix, but still keep the code which uses adjacency list runable.

13.1 Basic usage

First let's generate a random matrix and the corresponding adjacency list.

```
set.seed(999)
mat = matrix(sample(18, 18), 3, 6)
rownames(mat) = paste0("S", 1:3)
colnames(mat) = paste0("E", 1:6)
mat
```

```
##    E1 E2 E3 E4 E5 E6
## S1  8 13 18  6 11 14
## S2 10 12  1  3  5  7
## S3  2 16  4 17  9 15
```

```
df = data.frame(from = rep(rownames(mat), times = ncol(mat)),
    to = rep(colnames(mat), each = nrow(mat)),
    value = as.vector(mat),
    stringsAsFactors = FALSE)
df
```

```
##    from to value
## 1    S1 E1     8
## 2    S2 E1    10
## 3    S3 E1     2
## 4    S1 E2    13
## 5    S2 E2    12
## 6    S3 E2    16
## 7    S1 E3    18
## 8    S2 E3     1
## 9    S3 E3     4
## 10   S1 E4     6
## 11   S2 E4     3
## 12   S3 E4    17
## 13   S1 E5    11
## 14   S2 E5     5
## 15   S3 E5     9
## 16   S1 E6    14
## 17   S2 E6     7
## 18   S3 E6    15
```

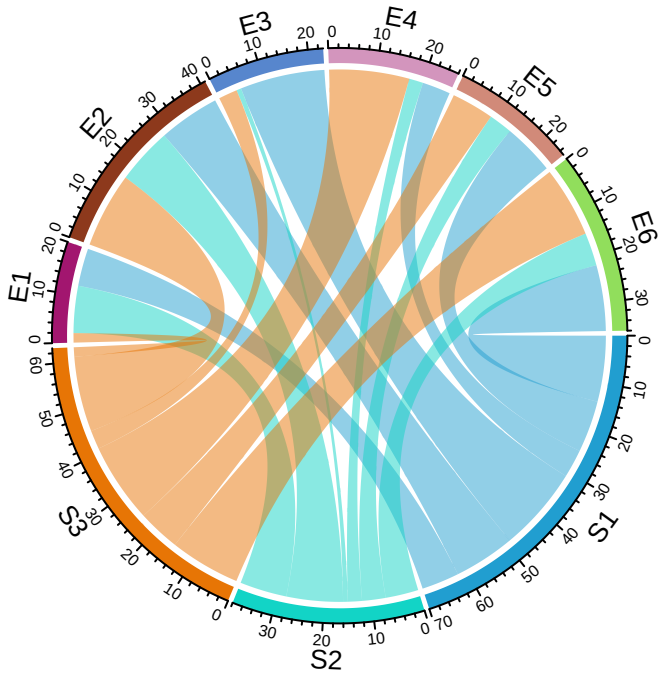


Figure 13.1: Basic usages of 'chordDiagram()'.

The most simple usage is just calling `chordDiagram()` with `mat` (Figure 13.1).

```
chordDiagram(mat)
```

```
circos.clear()
```

or call with `df`:

```
chordDiagram(df)
circos.clear()
```

The default Chord Diagram consists of a track of labels, a track of grids (or you call it lines) with axes, and links. Sectors which correspond to rows in the matrix locate at the bottom half of the circle. The order of sectors is the order of `union(rownames(mat), colnames(mat))` or `union(df[[1]], df[[2]])` if input is a data frame. The order of sectors can be controlled by `order` argument (Figure 13.2 right). Of course, the length of `order` vector should be same as the number of sectors.

```
chordDiagram(mat, order = c("S1", "E1", "E2", "S2", "E3", "E4", "S3", "E5", "E6"))
```

```
circos.clear()
```

Under default settings, the grid colors which represent sectors are randomly generated, and the link colors are same as grid colors which correspond to rows (or the first column if the input is an adjacency list) but with 50% transparency.

13.2 Adjust by `circos.par()`

Since Chord Diagram is implemented by basic circlize functions, like normal circular plot, the layout can be customized by `circos.par()`.

The gaps between sectors can be set by `circos.par(gap.after = ...)` (Figure 13.3). It is useful when

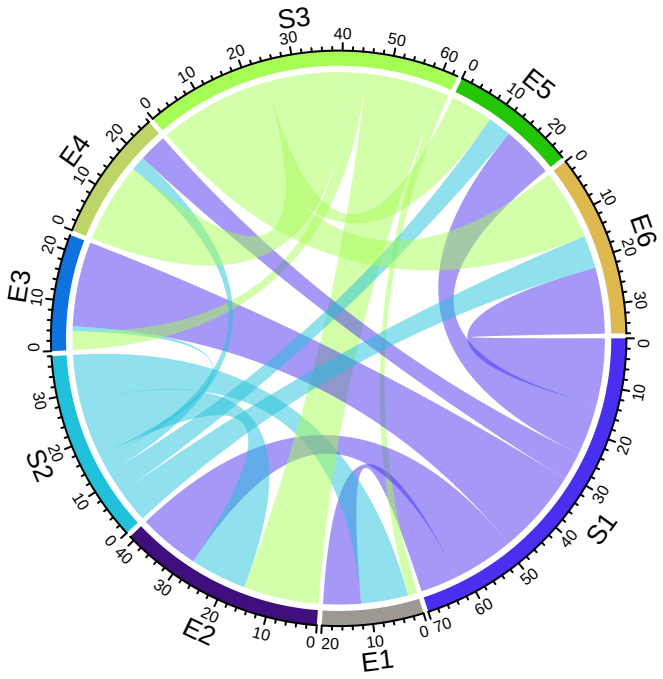


Figure 13.2: Adjust sector orders in Chord diagram.

you want to distinguish sectors between rows and columns. Please note since you change default graphical settings, you need to use `circos.clear()` in the end of the plot to reset it.

```
circos.par(gap.after = c(rep(5, nrow(mat)-1), 15, rep(5, ncol(mat)-1), 15))
chordDiagram(mat)
```

```
circos.clear()
```

If the input is a data frame:

```
circos.par(gap.after = c(rep(5, length(unique(df[[1]]))-1), 15,
                         rep(5, length(unique(df[[2]]))-1), 15))
chordDiagram(df)
circos.clear()
```

Similar to a normal circular plot, the first sector (which is the first row in the adjacency matrix or the first row in the adjacency list) starts from right center of the circle and sectors are arranged clock-wisely. The start degree for the first sector can be set by `circos.par(start.degree = ...)` and the direction can be set by `circos.par(clock.wise = ...)` (Figure 13.4).

```
circos.par(start.degree = 90, clock.wise = FALSE)
chordDiagram(mat)
```

```
circos.clear()
```

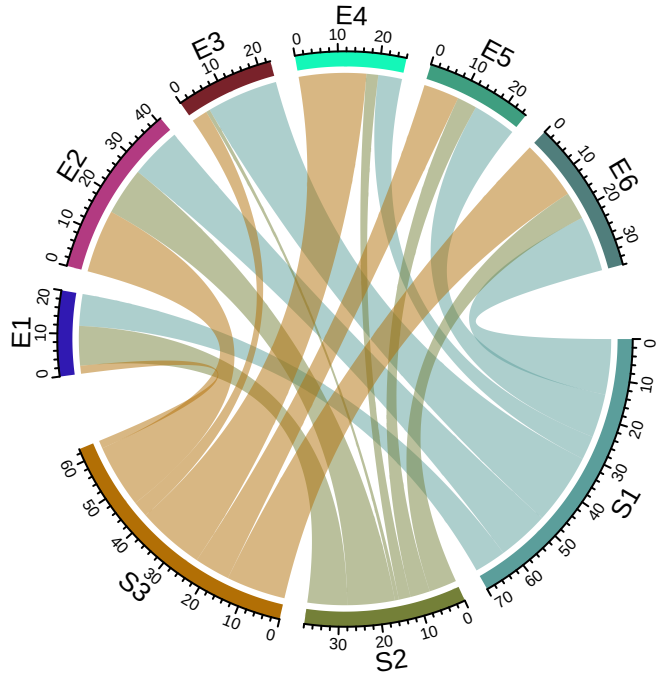


Figure 13.3: Set gaps in Chord diagram.

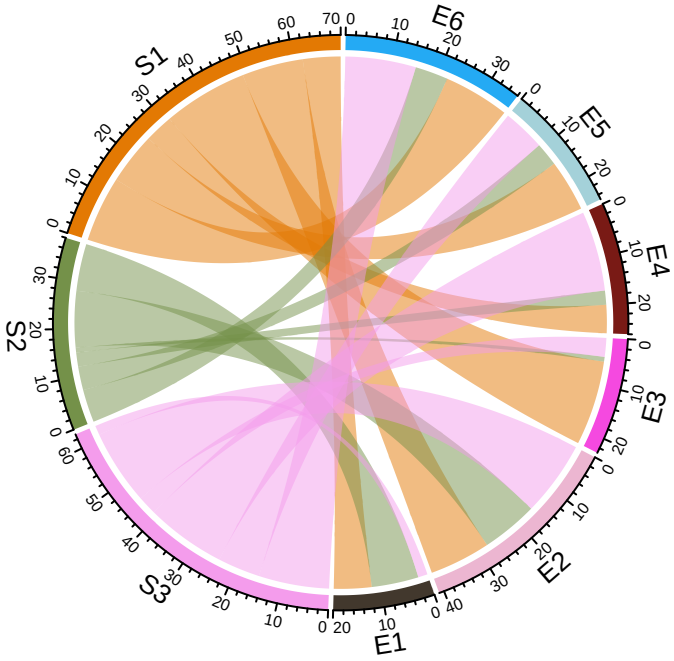


Figure 13.4: Change position and orientation of Chord diagram.

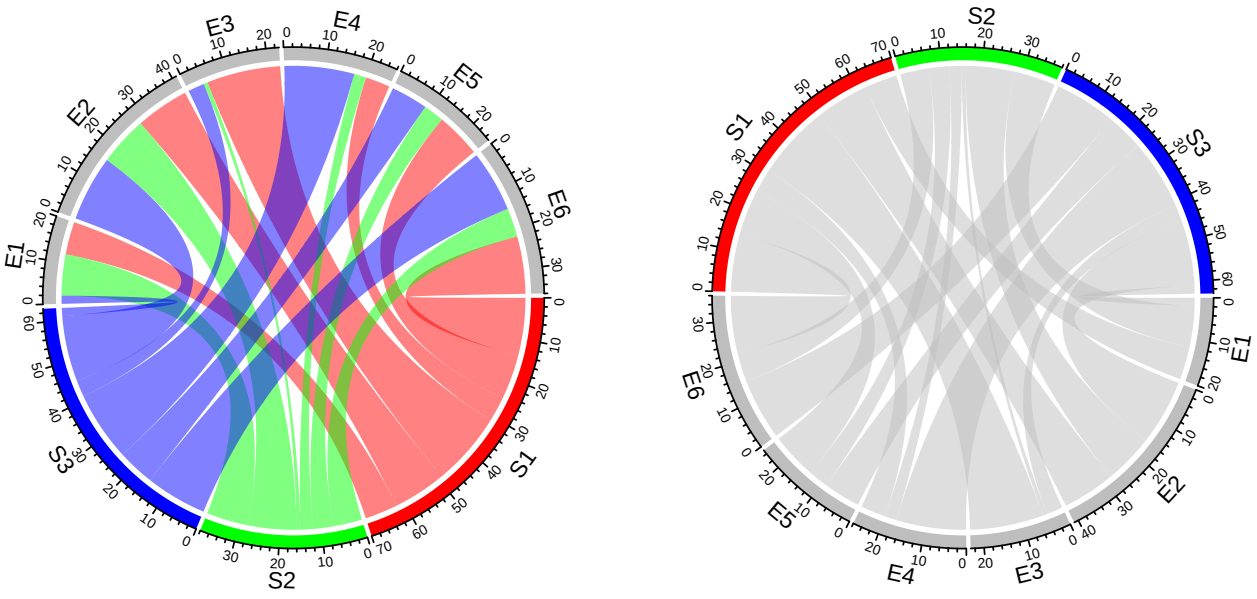


Figure 13.5: Set grid colors in Chord diagram.

13.3 Colors

13.3.1 Set grid colors

Grids have different colors to represent different sectors. Generally, sectors are divided into two groups. One contains sectors defined in rows of the matrix or the first column in the data frame, and the other contains sectors defined in columns of the matrix or the second column in the data frame. Thus, links connect objects in the two groups. By default, link colors are same as the color for the corresponding sectors in the first group.

Changing colors of grids may change the colors of links as well. Colors for grids can be set through `grid.col` argument. Values of `grid.col` better be a named vector of which names correspond to sector names. If it is has no name index, the order of `grid.col` is assumed to have the same order as sectors (Figure 13.5).

```
grid.col = c(S1 = "red", S2 = "green", S3 = "blue",
    E1 = "grey", E2 = "grey", E3 = "grey", E4 = "grey", E5 = "grey", E6 = "grey")
chordDiagram(mat, grid.col = grid.col)
chordDiagram(t(mat), grid.col = grid.col)
```

13.3.2 Set link colors

Transparency of link colors can be set by `transparency` argument (Figure 13.6). The value should between 0 to 1 in which 0 means no transparency and 1 means full transparency. Default transparency is 0.5.

```
chordDiagram(mat, grid.col = grid.col, transparency = 0)
```

For adjacecy matrix, colors for links can be customized by providing a matrix of colors. In the following example, we use `rand_color()` to generate a random color matrix. Note since `col_mat` already contains transparency, `transparency` will be ignored if it is set (Figure 13.7).

```
col_mat = rand_color(length(mat), transparency = 0.5)
dim(col_mat) = dim(mat)  # to make sure it is a matrix
```

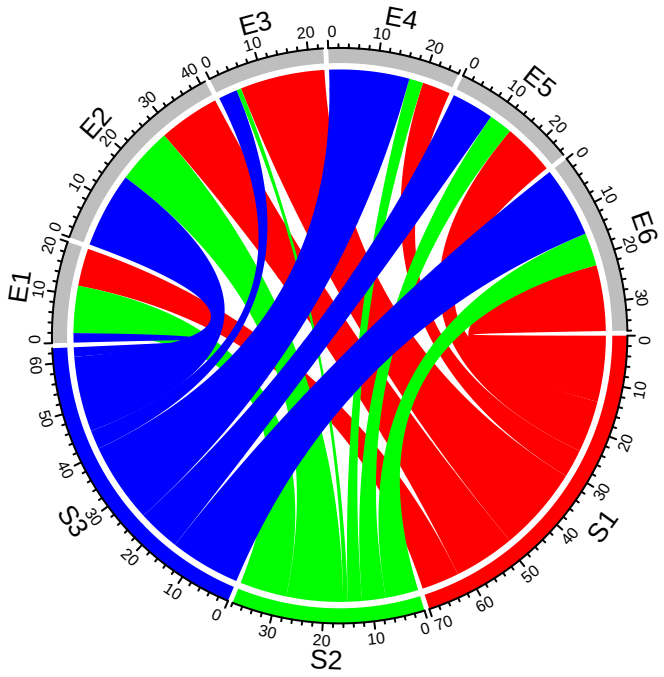


Figure 13.6: Transparency for links in Chord diagram.

```
chordDiagram(mat, grid.col = grid.col, col = col_mat)
```

While for ajacency list, colors for links can be customized as a vector.

```
col = rand_color(nrow(df))
chordDiagram(df, grid.col = grid.col, col = col)
```

When the strength of the relation (e.g. correlations) represents as continuous values, `col` can also be specified as a self-defined color mapping function. `chordDiagram()` accepts a color mapping generated by `colorRamp2()` (Figure 13.8).

```
col_fun = colorRamp2(range(mat), c("#FFEEEE", "#FF0000"), transparency = 0.5)
chordDiagram(mat, grid.col = grid.col, col = col_fun)
```

The color mapping function also works for adjacency list, but it will be applied to the third column in the data frame, so you need to make sure the third column has the proper values.

```
chordDiagram(df, grid.col = grid.col, col = col_fun)
```

When the input is a matrix, sometimes you don't need to generate the whole color matrix. You can just provide colors which correspond to rows or columns so that links from a same row/column will have the same color (Figure 13.9). Here note values of colors can be set as numbers, color names or hex code, same as in the base R graphics.

```
chordDiagram(mat, grid.col = grid.col, row.col = 1:3)
chordDiagram(mat, grid.col = grid.col, column.col = 1:6)
```

`row.col` and `column.col` is specifically designed for matrix. There is no similar settings for ajacency list.

To emphasize again, transparency of links can be included in `col`, `row.col` or `column.col`, if transparency is already set there, `transparency` argument will be ignored.

In Section 13.5, we will introduce how to highlight subset of links by only assigning colors to them.

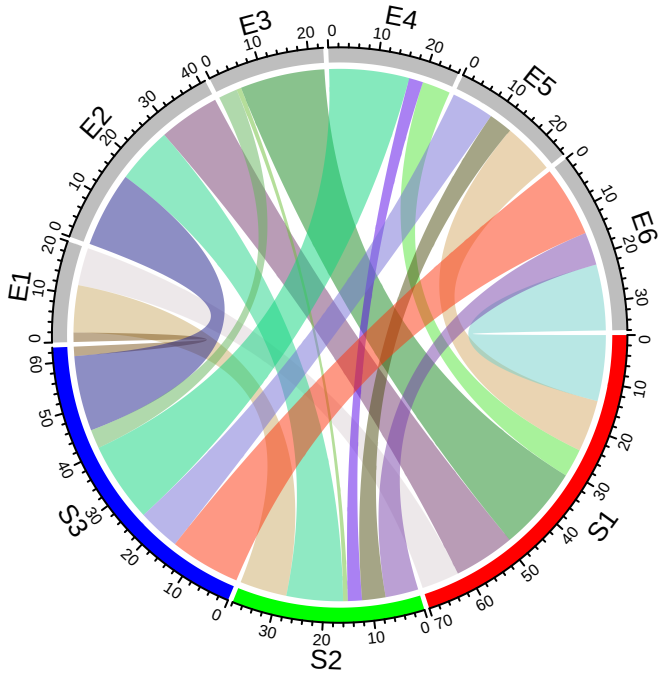


Figure 13.7: Set a color matrix for links in Chord diagram.

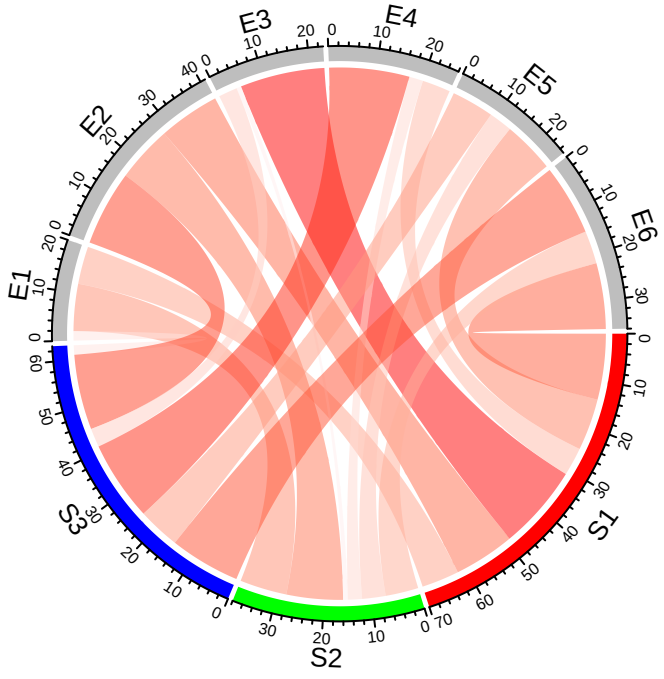


Figure 13.8: Set a color mapping function for links in Chord diagram.

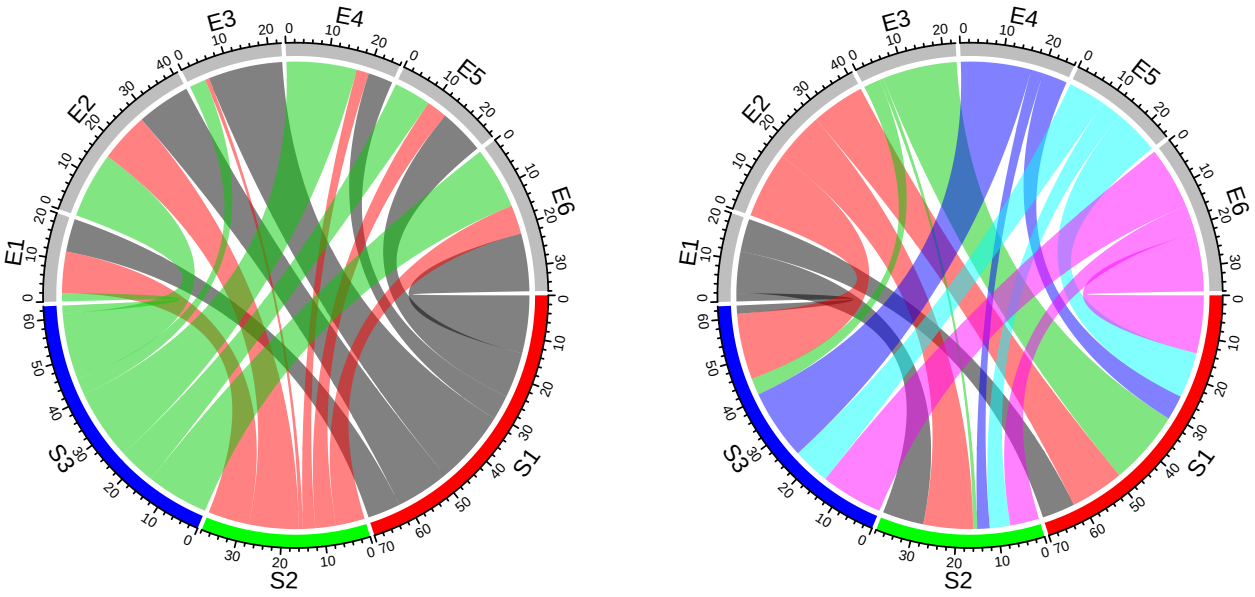


Figure 13.9: Set link colors same as row sectors or column sectors in Chord diagram.

13.4 Link border

`link.lwd`, `link.lty` and `link.border` control the line width, the line style and the color of the link border. All these three parameters can be set either a single scalar or a matrix if the input is adjacency matrix.

If it is set as a single scalar, it means all links share the same style (Figure 13.10).

```
chordDiagram(mat, grid.col = grid.col, link.lwd = 2, link.lty = 2, link.border = "red")
```

If it is set as a matrix, it should have same dimension as `mat` (Figure 13.11).

```
lwd_mat = matrix(1, nrow = nrow(mat), ncol = ncol(mat))
lwd_mat[mat > 12] = 2
border_mat = matrix(NA, nrow = nrow(mat), ncol = ncol(mat))
border_mat[mat > 12] = "red"
chordDiagram(mat, grid.col = grid.col, link.lwd = lwd_mat, link.border = border_mat)
```

The matrix is not necessary to have same dimensions as in `mat`. It can also be a sub matrix (Figure 13.12). For rows or columns of which the corresponding values are not specified in the matrix, default values are filled in. It must have row names and column names so that the settings can be mapped to the correct links.

```
border_mat2 = matrix("black", nrow = 1, ncol = ncol(mat))
rownames(border_mat2) = rownames(mat)[2]
colnames(border_mat2) = colnames(mat)
chordDiagram(mat, grid.col = grid.col, link.lwd = 2, link.border = border_mat2)
```

To be more convenient, graphic parameters can be set as a three-column data frame in which the first two columns correspond to row names and column names in the matrix, and the third column corresponds to the graphic parameters (Figure 13.13).

```
lty_df = data.frame(c("S1", "S2", "S3"), c("E5", "E6", "E4"), c(1, 2, 3))
lwd_df = data.frame(c("S1", "S2", "S3"), c("E5", "E6", "E4"), c(2, 2, 2))
border_df = data.frame(c("S1", "S2", "S3"), c("E5", "E6", "E4"), c(1, 1, 1))
chordDiagram(mat, grid.col = grid.col, link.lty = lty_df, link.lwd = lwd_df,
    link.border = border_df)
```

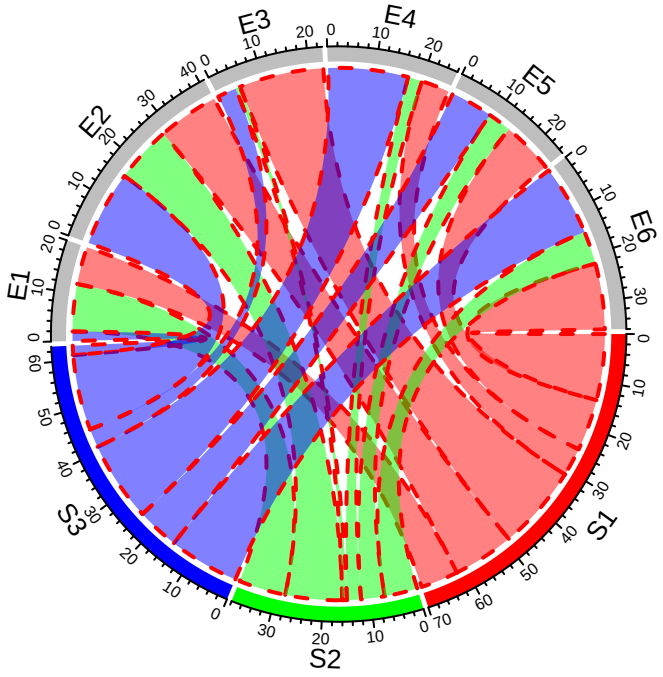


Figure 13.10: Line style for Chord diagram.

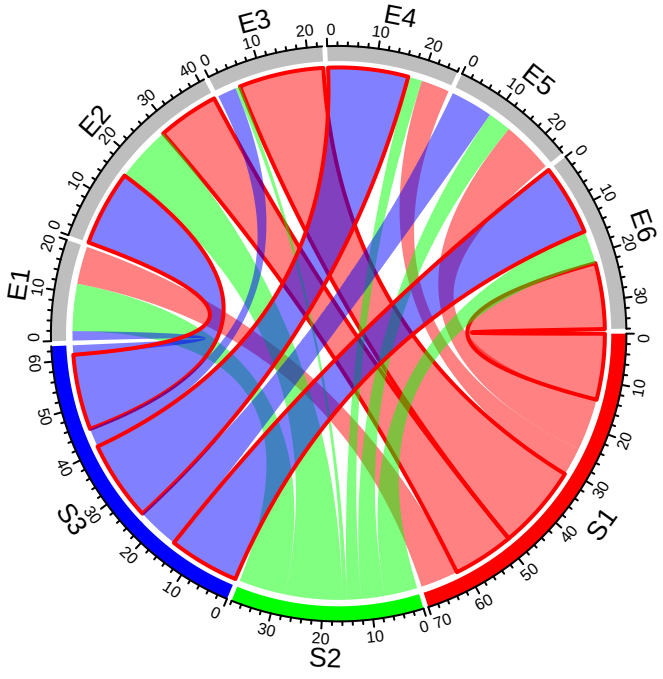


Figure 13.11: Set line style as a matrix.

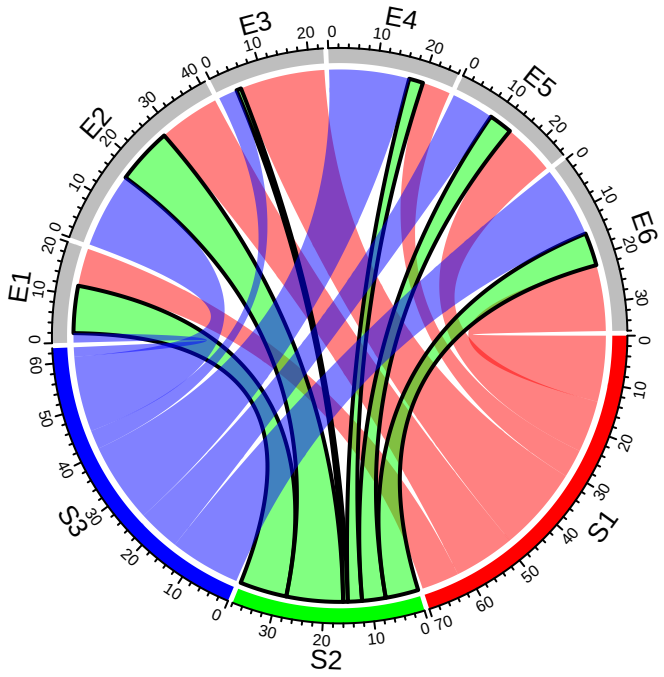


Figure 13.12: Set line style as a sub matrix.

It is much easier if the input is a data frame, you only need to set graphic settings as a vecotr.

```
chordDiagram(df, grid.col = grid.col, link.lty = sample(1:3, nrow(df), replace = TRUE),
    link.lwd = runif(nrow(df))*2, link.border = sample(0:1, nrow(df), replace = TRUE))
```

13.5 Highlight links

Sometimes we want to highlight some links to emphasize the importance of such relations. Highlighting by different border styles are introduced in Section 13.4 and here we focus on highlighting by colors.

THere are two ways to highlight links, one is to set different transparency to different links and the other is to only draw links that needs to be highlighted. Based on this rule and ways to assign colors to links (introduced in Section 13.3.2), we can highlight links which come from a same sector by assigning colors with different transparency by `row.col` argument (Figure 13.14).

```
chordDiagram(mat, grid.col = grid.col, row.col = c("#FF000080", "#00FF0010", "#0000FF10"))
```

We can also highlight links with values larger than a cutoff. There are at least three ways to do it. First, construct a color matrix and set links with small values to full transparency.

Since link colors can be specified as a matrix, we can set the transparency of those links to a high value or even set to full transparency (Figure 13.15). In following example, links with value less than 12 is set to `#00000000`.

```
col_mat[mat < 12] = "#00000000"
chordDiagram(mat, grid.col = grid.col, col = col_mat)
```

Following code demonstrates using a color mapping function to map values to different transparency. Note this is also workable for adjacency list.

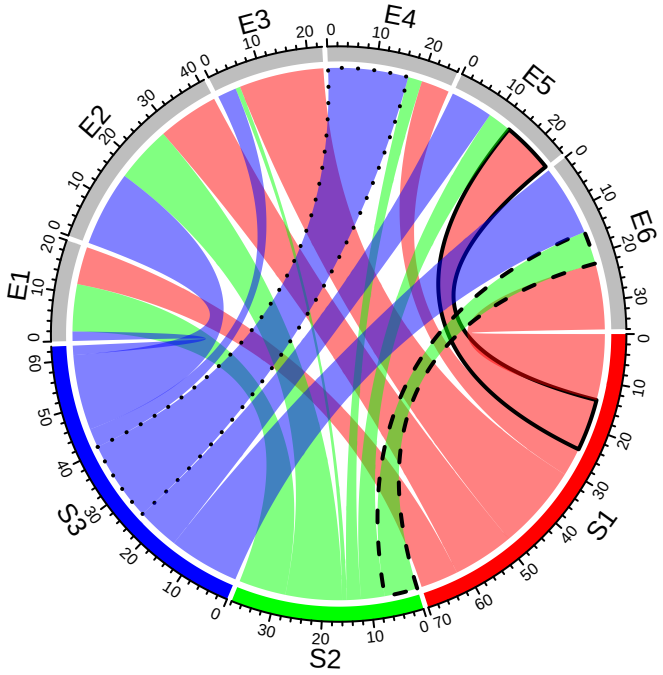


Figure 13.13: Set line style as a data frame.

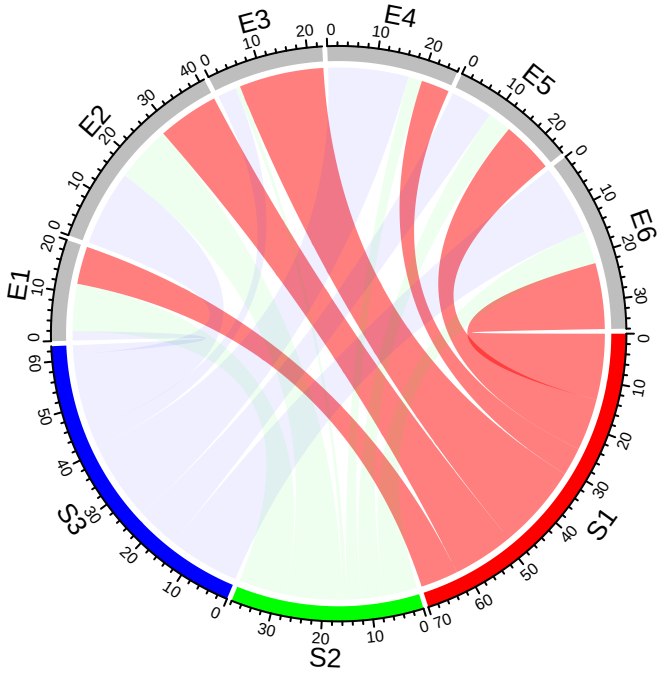


Figure 13.14: Highlight links by transparency.

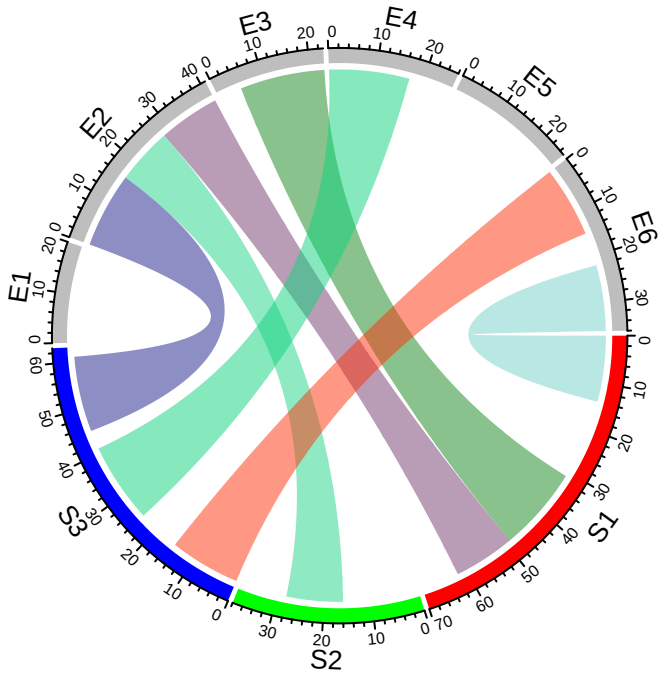


Figure 13.15: Highlight links by color matrix.

```
col_fun = function(x) ifelse(x < 12, "#00000000", "#FF000080")
chordDiagram(mat, grid.col = grid.col, col = col_fun)
```

For both color matrix and color mapping function, actually all links are all drawn and the reason why you cannot see some of them is they are assigned with full transparency. If a three-column data frame is used to assign colors to links of interest, links which are not defined in `col_df` are not drawn (Figure 13.16).

```
col_df = data.frame(c("S1", "S2", "S3"), c("E5", "E6", "E4"),
    c("#FF000080", "#00FF0080", "#0000FF80"))
chordDiagram(mat, grid.col = grid.col, col = col_df)
```

Highlighting links is relatively simple for adjacency list that you only need to construct a vector of colors according to what links you want to highlight.

```
col = rand_color(nrow(df))
col[df[[3]] < 10] = "#00000000"
chordDiagram(df, grid.col = grid.col, col = col)
```

The `link.visible` argument is recently introduced to **circlize** package which may provide a simple to control the visibility of links. The value can be set as an logical matrix for adjacency matrix or a logical vector for adjacency list (Figure 13.17).

```
col = rand_color(nrow(df))
chordDiagram(df, grid.col = grid.col, link.visible = df[[3]] >= 10)
```

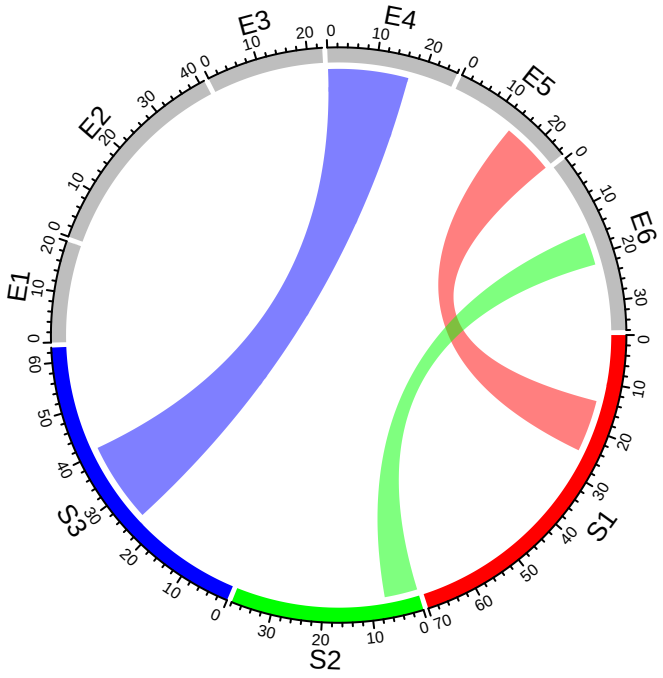


Figure 13.16: Highlight links by data frame.

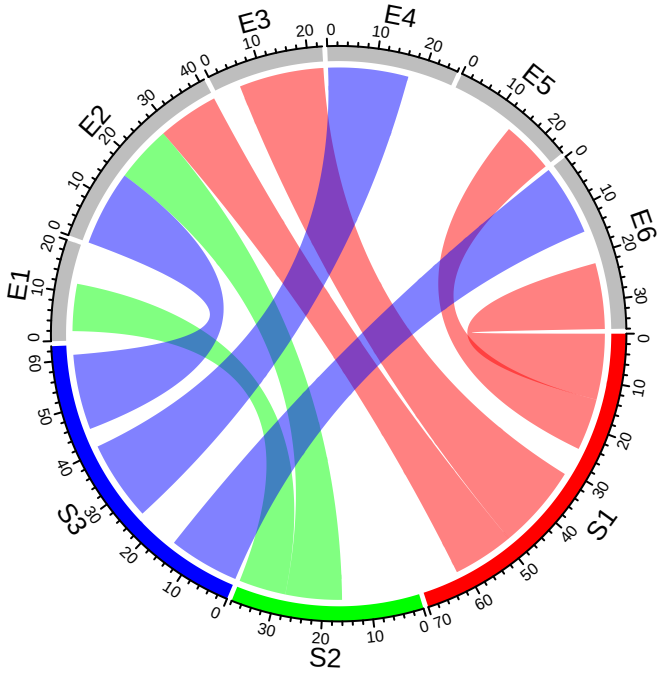


Figure 13.17: Highlight links by setting 'link.visible'.

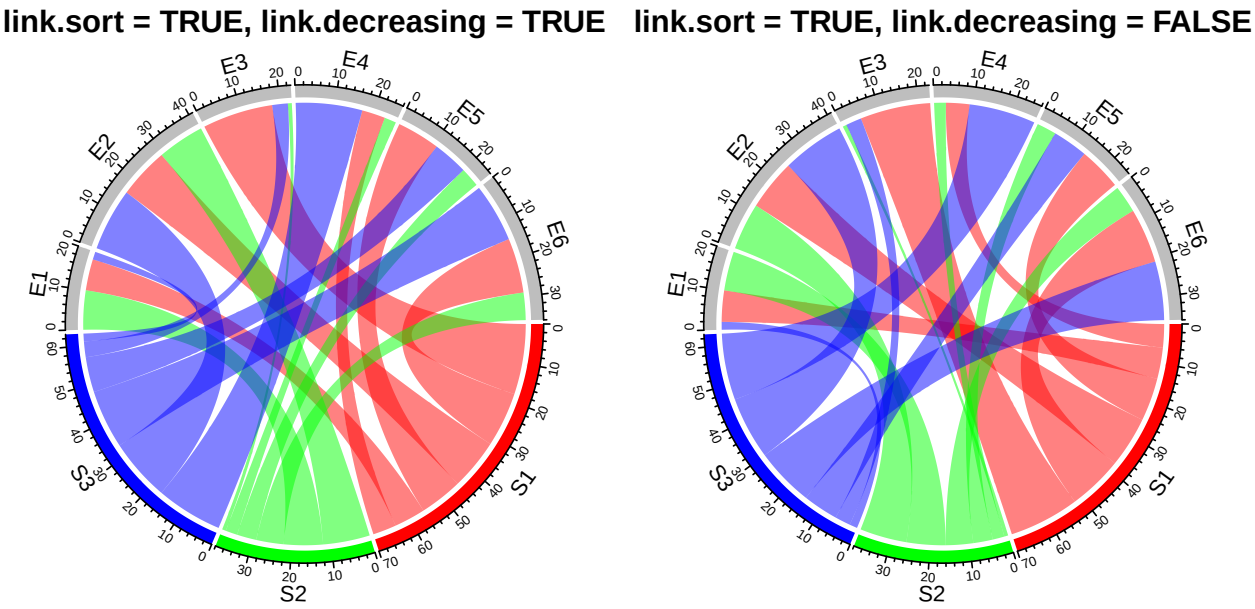


Figure 13.18: Order of positioning links on sectors.

13.6 Orders of links

13.6.1 Orders of positions on sectors

Orders of links on every sector are adjusted automatically to make them look nice. But sometimes sorting links according to the width on the sector is useful for detecting potential features. `link.sort` and `link.decreasing` can be set to control the order of positioning links on sectors (Figure 13.18).

```
chordDiagram(mat, grid.col = grid.col, link.sort = TRUE, link.decreasing = TRUE)
title("link.sort = TRUE, link.decreasing = TRUE", cex = 0.8)
chordDiagram(mat, grid.col = grid.col, link.sort = TRUE, link.decreasing = FALSE)
title("link.sort = TRUE, link.decreasing = FALSE", cex = 0.8)
```

13.6.2 Order of adding links

The default order of adding links to the plot is based on their order in the matrix or in the data frame. Normally, transparency should be set to the link colors so that they will not overlap to each other. In most cases, this looks fine, but sometimes, it improves the visualization to put wide links more forward and to put small links more backward in the plot. This can be set by `link.rank` argument which defines the order of adding links. Larger value means the corresponding link is added later (Figure 13.19).

```
chordDiagram(mat, grid.col = grid.col, transparency = 0)
chordDiagram(mat, grid.col = grid.col, transparency = 0, link.rank = rank(mat))
```

Similar code if the input is a data frame.

```
chordDiagram(df, grid.col = grid.col, transparency = 0, link.rank = rank(df[[3]]))
```

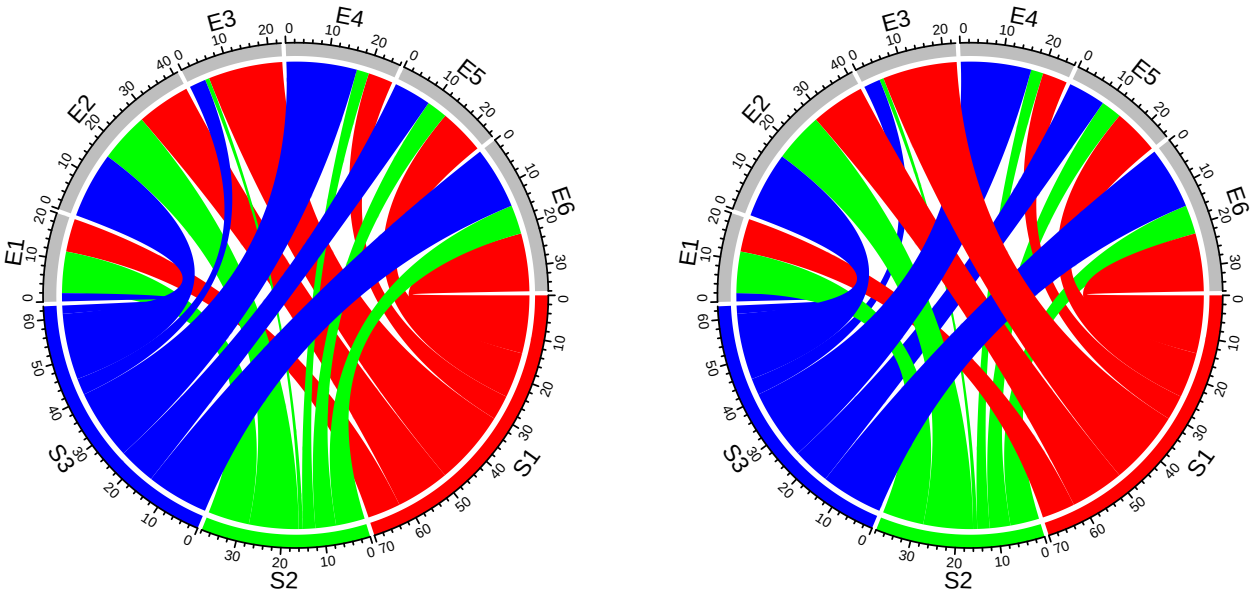


Figure 13.19: Order of adding links.

13.7 Self-links

How to set self links dependends on whether the information needs to be duplicated. The `self.link` argument can be set to 1 or 2 for the two different scenarios. Check the difference in Figure 13.20.

```
df2 = data.frame(start = c("a", "b", "c", "a"), end = c("a", "a", "b", "c"))
chordDiagram(df2, grid.col = 1:3, self.link = 1)
title("self.link = 1")
chordDiagram(df2, grid.col = 1:3, self.link = 2)
title("self.link = 2")
```

13.8 Symmetric matrix

When the matrix is symmetric, by setting `symmetric = TRUE`, only lower triangular matrix without the diagonal will be used (Figure 13.21).

```
mat3 = matrix(rnorm(25), 5)
colnames(mat3) = letters[1:5]
cor_mat = cor(mat3)
col_fun = colorRamp2(c(-1, 0, 1), c("green", "white", "red"))
chordDiagram(cor_mat, grid.col = 1:5, symmetric = TRUE, col = col_fun)
title("symmetric = TRUE")
chordDiagram(cor_mat, grid.col = 1:5, col = col_fun)
title("symmetric = FALSE")
```

13.9 Directional relations

In some cases, when the input is a matrix, rows and columns represent directions, or when the input is a data frame, the first column and second column represent directions. Argument `directional` is used to

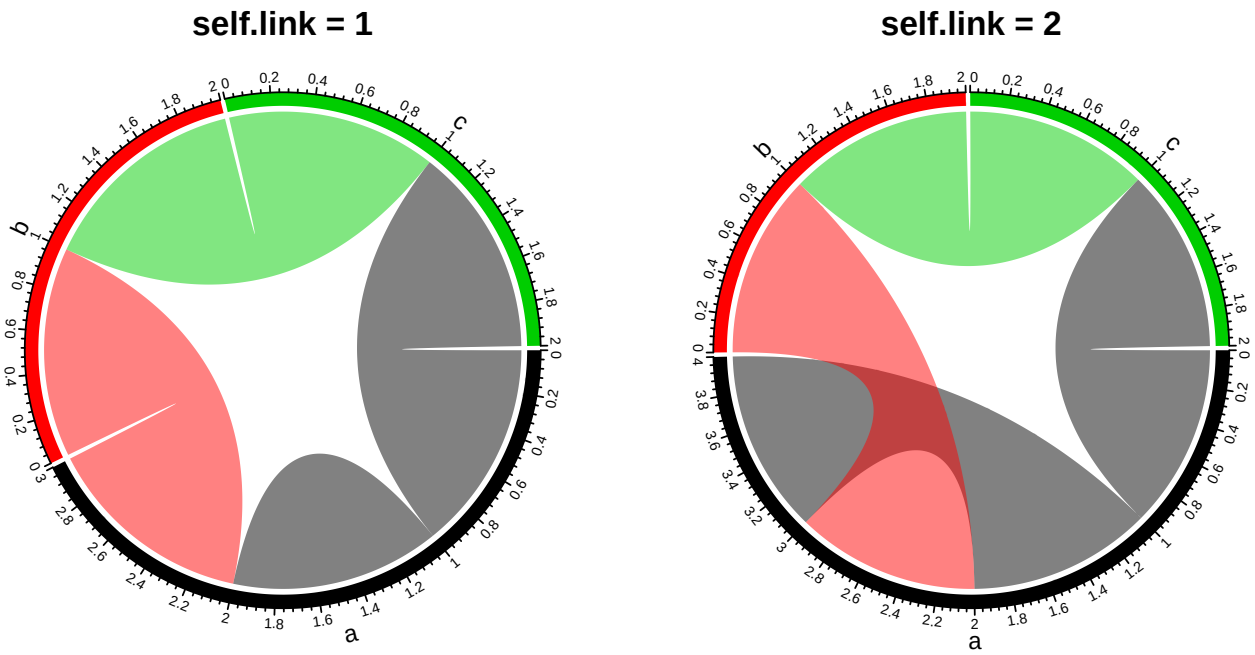


Figure 13.20: Self-links in Chord diagram.

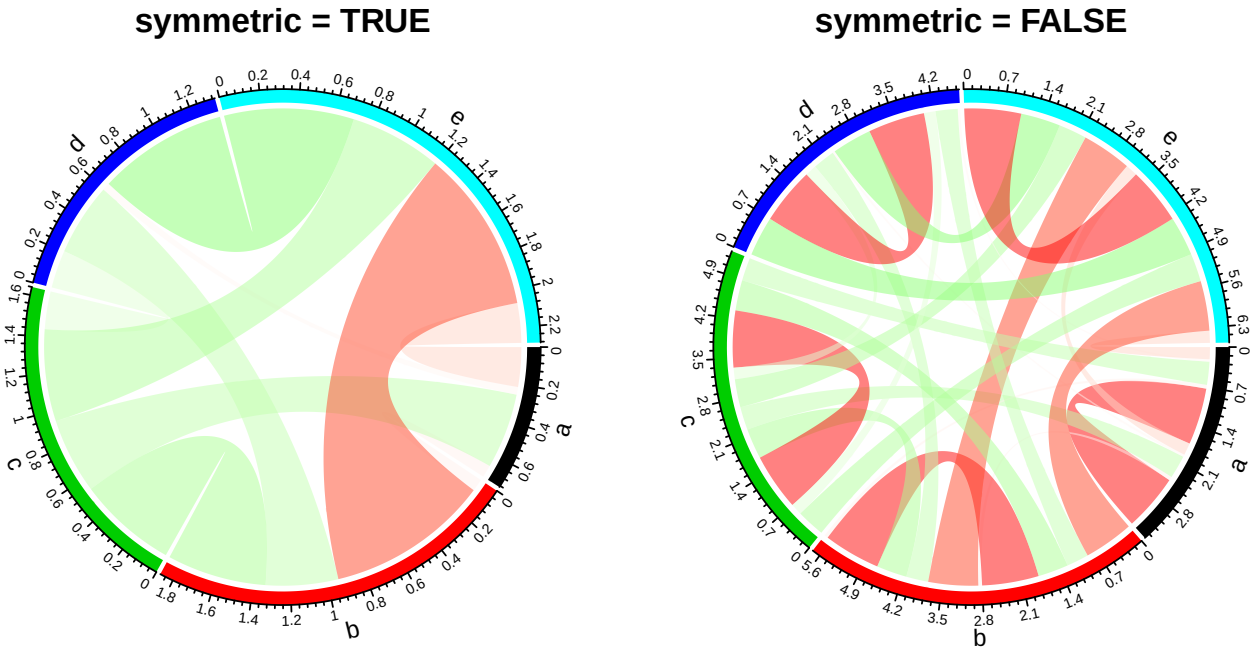


Figure 13.21: Symmetric matrix for Chord diagram.

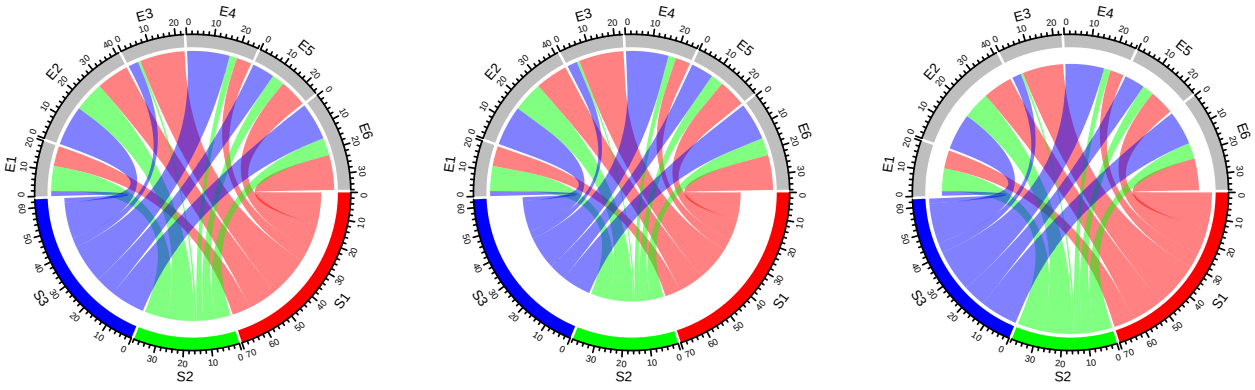


Figure 13.22: Represent directions by different height of link ends.

illustrate such direction on the plot. `directional` with value `1` means the direction is from rows to columns (or from the first column to the second column for the adjacency list) while `-1` means the direction is from columns to rows (or from the second column to the first column for the adjacency list). A value of `2` means bi-directional.

By default, the two ends of links have unequal height (Figure 13.22) to represent the directions. The position of starting end of the link is shorter than the other end to give users the feeling that the link is moving out. If this is not what your correct feeling, you can set `diffHeight` to a negative value.

```
par(mfrow = c(1, 3))
chordDiagram(mat, grid.col = grid.col, directional = 1)
chordDiagram(mat, grid.col = grid.col, directional = 1, diffHeight = uh(5, "mm"))
chordDiagram(mat, grid.col = grid.col, directional = -1)
```

Row names and column names in `mat` can also overlap. In this case, showing direction of the link is important to distinguish them (Figure 13.23).

```
mat2 = matrix(sample(100, 35), nrow = 5)
rownames(mat2) = letters[1:5]
colnames(mat2) = letters[1:7]
mat2
```

```
##    a  b  c  d  e  f  g
## a 55 20 84 16 14 97 57
## b 82 44 78 45 54 63 31
## c 77  3 99 76 86  8 18
## d 70 40  6 43 39 67 60
## e 79 12 25 17 10 93 30
```

```
chordDiagram(mat2, grid.col = 1:7, directional = 1, row.col = 1:5)
```

If you do not need self-link for which two ends of a link are in a same sector, just set corresponding values to 0 in the matrix (Figure 13.24).

```
mat3 = mat2
for(cn in intersect(rownames(mat3), colnames(mat3))) {
    mat3[cn, cn] = 0
}
mat3
```

```
##    a  b  c  d  e  f  g
## a  0 20 84 16 14 97 57
```

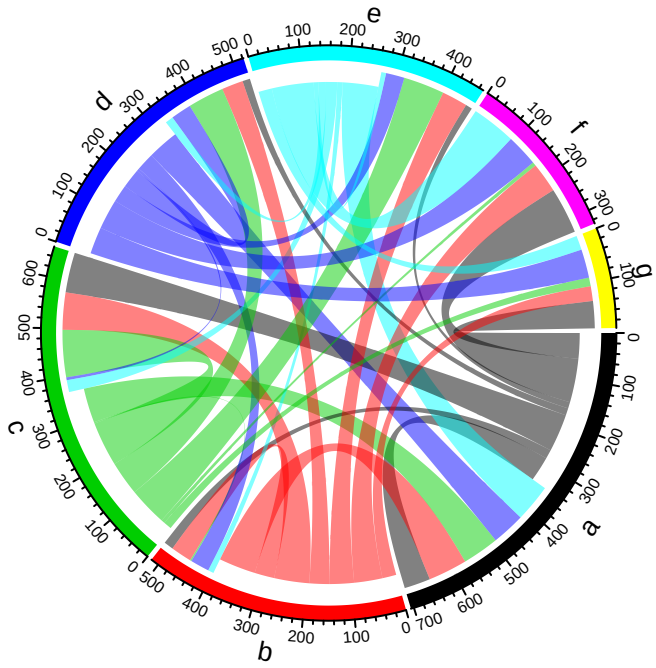


Figure 13.23: CHord diagram where row names and column names overlap.

```
## b 82  0 78 45 54 63 31
## c 77  3  0 76 86  8 18
## d 70 40  6  0 39 67 60
## e 79 12 25 17  0 93 30
```

```
chordDiagram(mat3, grid.col = 1:7, directional = 1, row.col = 1:5)
```

Links can have arrows to represent directions (Figure 13.25). When `direction.type` is set to `arrows`, Arrows are added at the center of links. Similar as other graphics parameters for links, the parameters for drawing arrows such as arrow color and line type can either be a scalar, a matrix, or a three-column data frame.

If `link.arr.col` is set as a data frame, only links specified in the data frame will have arrows. Pleast note this is the only way to draw arrows to subset of links.

```
arr.col = data.frame(c("S1", "S2", "S3"), c("E5", "E6", "E4"),
    c("black", "black", "black"))
chordDiagram(mat, grid.col = grid.col, directional = 1, direction.type = "arrows",
    link.arr.col = arr.col, link.arr.length = 0.2)
```

If combining both `arrows` and `diffHeight`, it will give you better visualization (Figure 13.26).

```
arr.col = data.frame(c("S1", "S2", "S3"), c("E5", "E6", "E4"),
    c("black", "black", "black"))
chordDiagram(mat, grid.col = grid.col, directional = 1,
    direction.type = c("diffHeight", "arrows"),
    link.arr.col = arr.col, link.arr.length = 0.2)
```

There is another arrow type: `big.arrow` which is efficient to visualize arrows when there are too many links (Figure 13.27).

```
matx = matrix(rnorm(64), 8)
chordDiagram(matx, directional = 1, direction.type = c("diffHeight", "arrows"),
```

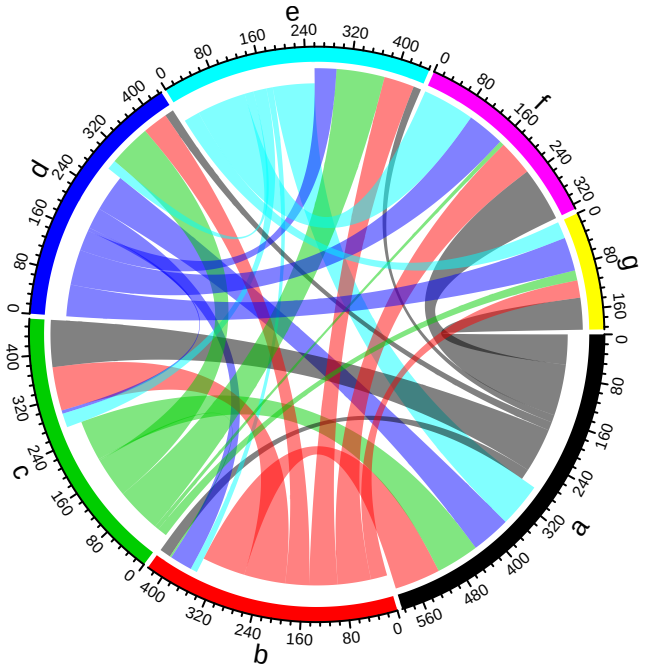


Figure 13.24: Directional Chord diagram without self links.

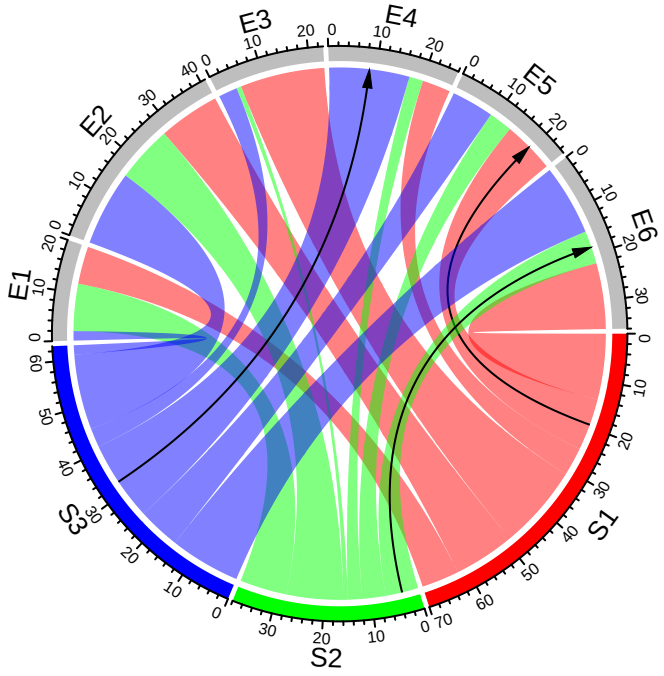


Figure 13.25: Use arrows to represent directions in Chord diagram.

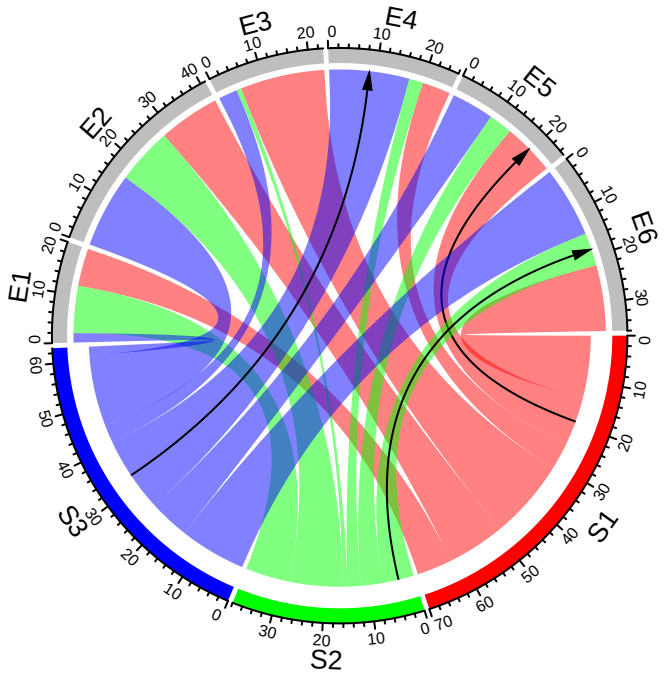


Figure 13.26: Use both arrows and link height to represent directions in Chord diagram.

```
    link.arr.type = "big.arrow")
```

If `diffHeight` is set to a negative value, the start ends are longer than the other ends (Figure 13.28).

```
chordDiagram(matx, directional = 1, direction.type = c("diffHeight", "arrows"),
    link.arr.type = "big.arrow", diffHeight = -uh(2, "mm"))
```

It is almost the same to visualize directional Chord diagram form a adjacency list.

```
chordDiagram(df, directional = 1)
```

13.10 Reduce

If a sector in Chord Diagram is too small, it will be removed from the original matrix, since basically it can be ignored visually from the plot. In the following matrix, the second row and third column contain tiny values.

```
mat = matrix(rnorm(36), 6, 6)
rownames(mat) = paste0("R", 1:6)
colnames(mat) = paste0("C", 1:6)
mat[2, ] = 1e-10
mat[, 3] = 1e-10
```

In the Chord Diagram, categories corresponding to the second row and the third column will be removed.

```
chordDiagram(mat)
circos.info()
```

```
## All your sectors:
##  [1] "R1" "R3" "R4" "R5" "R6" "C1" "C2" "C4" "C5" "C6"
```

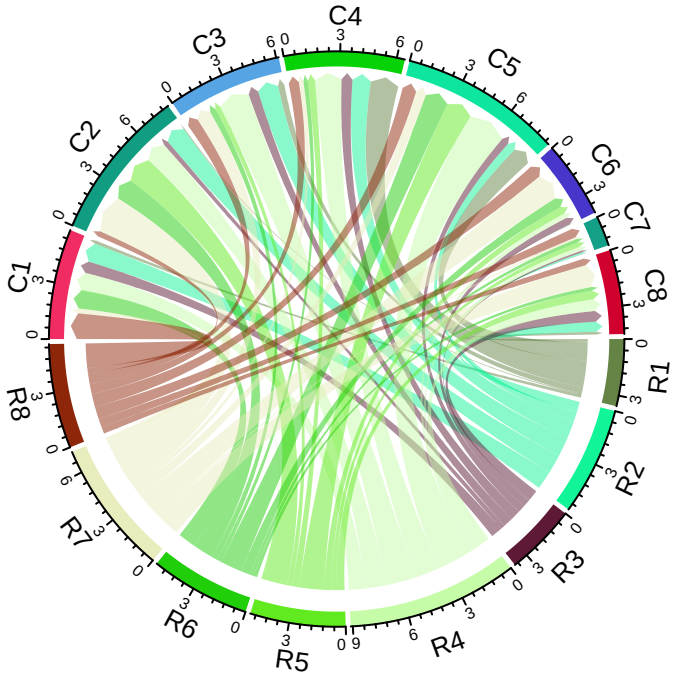


Figure 13.27: Use big arrows to represent directions in Chord diagram.

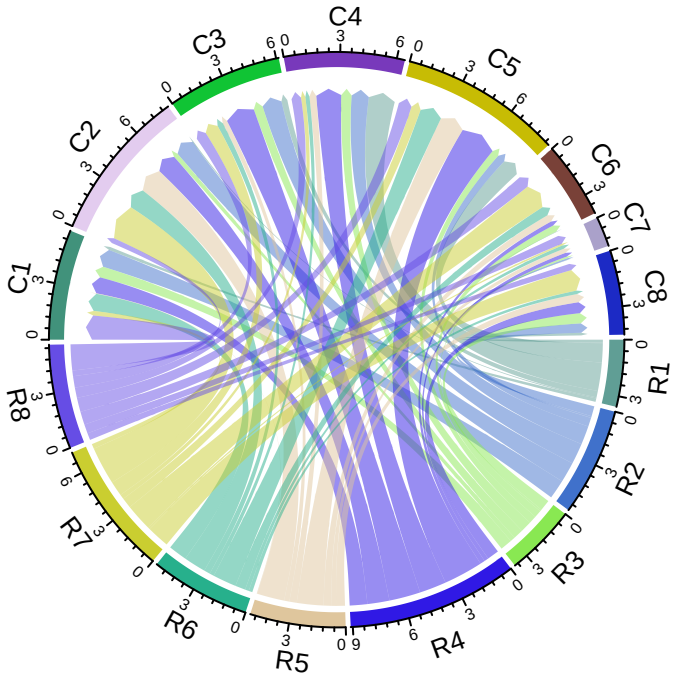


Figure 13.28: Use big arrows to represent directions in Chord diagram.

```
##
## All your tracks:
## [1] 1 2
##
## Your current sector.index is C6
## Your current track.index is 2
```

The `reduce` argument controls the size of sectors to be removed. The value is a percent of the size of a sector to the total size of all sectors.

You can also explictly remove sectors by assigning corresponding values to 0.

```
mat[2, ] = 0
mat[, 3] = 0
```

All parameters for sectors such as colors or gaps between sectors are also reduced accordingly by the function.

Chapter 14

Advanced usage of `chordDiagram()`

The default style of `chordDiagram()` is somehow enough for most visualization tasks, still you can have more configurations on the plot.

The usage is same for both ajacency matrix and ajacency list, so we only demonstrate with the matrix.

14.1 Organization of tracks

By default, `chordDiagram()` creates two tracks, one track for labels and one track for grids with axes.

```
chordDiagram(mat)
circos.info()
```

```
## All your sectors:
## [1] "S1" "S2" "S3" "E1" "E2" "E3" "E4" "E5" "E6"
##
## All your tracks:
## [1] 1 2
##
## Your current sector.index is E6
## Your current track.index is 2
```

These two tracks can be controlled by `annotationTrack` argument. Available values for this argument are `grid`, `name` and `axis`. The height of annotation tracks can be set by `annotationTrackHeight` which is the percentage to the radius of unit circle and can be set by `uh()` function with an absolute unit. Axes are only added if `grid` is set in `annotationTrack` (Figure 14.1).

```
chordDiagram(mat, grid.col = grid.col, annotationTrack = "grid")
chordDiagram(mat, grid.col = grid.col, annotationTrack = c("name", "grid"),
    annotationTrackHeight = c(0.03, 0.01))
chordDiagram(mat, grid.col = grid.col, annotationTrack = NULL)
```

Several empty tracks can be allocated before Chord diagram is drawn. Then self-defined graphics can be added to these empty tracks afterwards. The number of pre-allocated tracks can be set through `preAllocateTracks`.

```
chordDiagram(mat, preAllocateTracks = 2)
circos.info()
```

```
## All your sectors:
```

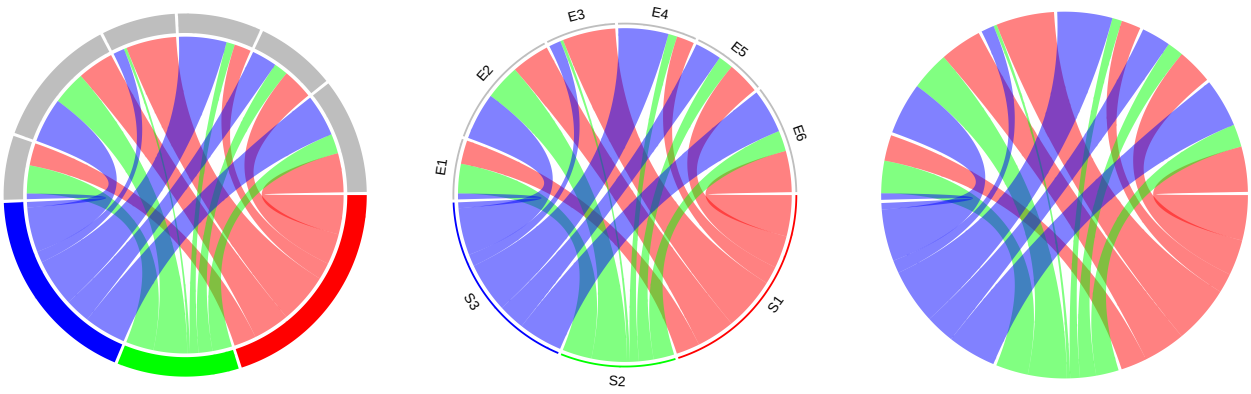


Figure 14.1: Track organization in 'chordDiagram()'.

```
## [1] "S1" "S2" "S3" "E1" "E2" "E3" "E4" "E5" "E6"
##
## All your tracks:
## [1] 1 2 3 4
##
## Your current sector.index is E6
## Your current track.index is 4
```

The default settings for pre-allocated tracks are:

```
list(ylim = c(0, 1),
     track.height = circos.par("track.height"),
     bg.col = NA,
     bg.border = NA,
     bg.lty = par("lty"),
     bg.lwd = par("lwd"))
```

The default settings for pre-allocated tracks can be overwritten by specifying `preAllocateTracks` as a list.

```
chordDiagram(mat, annotationTrack = NULL,
    preAllocateTracks = list(track.height = 0.3))
```

If more than one tracks need to be pre-allocated, just specify `preAllocateTracks` as a list which contains settings for each track:

```
chordDiagram(mat, annotationTrack = NULL,
    preAllocateTracks = list(list(track.height = 0.1),
                             list(bg.border = "black")))
```

By default `chordDiagram()` provides poor supports for customization of sector labels and axes, but with `preAllocateTracks` it is rather easy to customize them. Such customization will be introduced in next section.

14.2 Customize sector labels

In `chordDiagram()`, there is no argument to control the style of sector labels, but this can be done by first pre-allocating an empty track and customizing the labels in it later. In the following example, one track is firstly allocated and a Chord diagram is added without label track and axes. Later, the first track is updated with adding labels with clockwise facings (Figure 14.2).

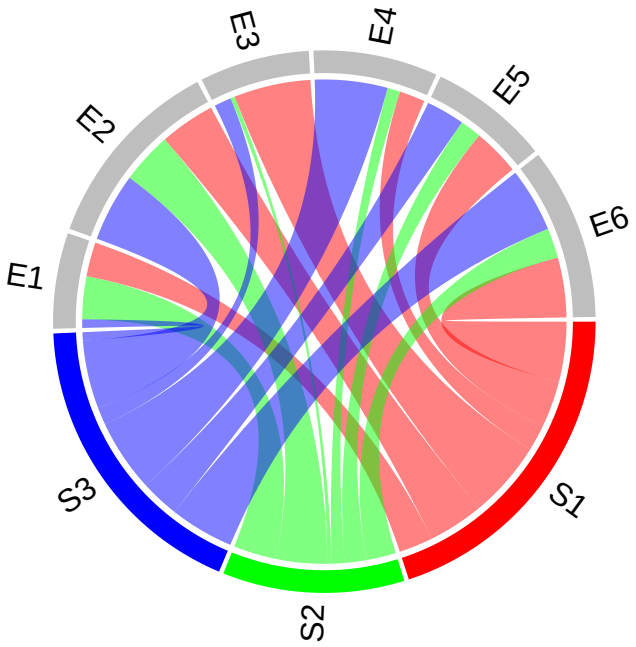


Figure 14.2: Change label directions.

```
chordDiagram(mat, grid.col = grid.col, annotationTrack = "grid",
    preAllocateTracks = list(track.height = max(strwidth(unlist(dimnames(mat))))))
# we go back to the first track and customize sector labels
circos.track(track.index = 1, panel.fun = function(x, y) {
    circos.text(CELL_META$xcenter, CELL_META$ylim[1], CELL_META$sector.index,
        facing = "clockwise", niceFacing = TRUE, adj = c(0, 0.5))
}, bg.border = NA) # here set bg.border to NA is important
```

In the following example, the labels are put on the grids (Figure 14.3). Please note `circos.text()` and `get.cell.meta.data()` can be used outside `panel.fun` if the sector index and track index are specified explicitly.

```
chordDiagram(mat, grid.col = grid.col,
    annotationTrack = c("grid", "axis"), annotationTrackHeight = uh(5, "mm"))
for(si in get.all.sector.index()) {
    xlim = get.cell.meta.data("xlim", sector.index = si, track.index = 1)
    ylim = get.cell.meta.data("ylim", sector.index = si, track.index = 1)
    circos.text(mean(xlim), mean(ylim), si, sector.index = si, track.index = 1,
        facing = "bending.inside", niceFacing = TRUE, col = "white")
}
```

For the last example in this section, if the width of the sector is less than 20 degree, the labels are added in the radical direction (Figure 14.4).

```
set.seed(123)
mat2 = matrix(rnorm(100), 10)
chordDiagram(mat2, annotationTrack = "grid",
    preAllocateTracks = list(track.height = max(strwidth(unlist(dimnames(mat))))))
circos.track(track.index = 1, panel.fun = function(x, y) {
    xlim = get.cell.meta.data("xlim")
    xplot = get.cell.meta.data("xplot")
```

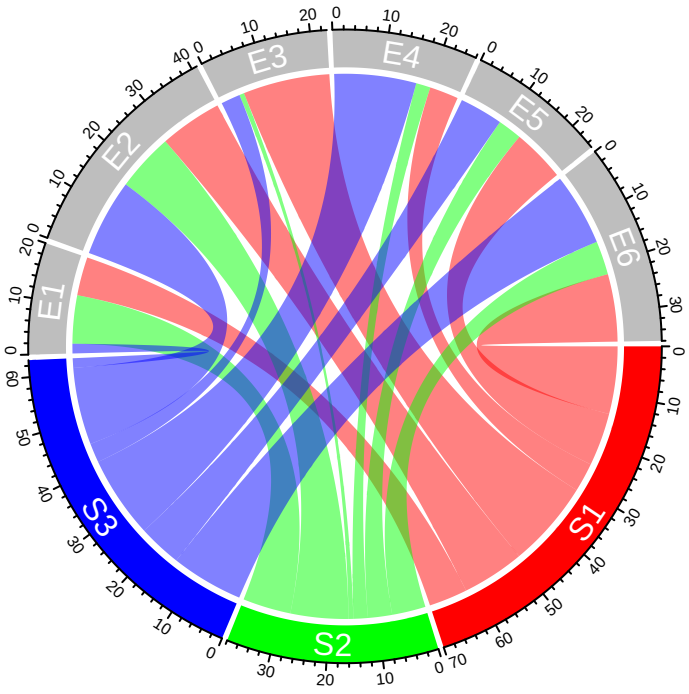


Figure 14.3: Put sector labels to the grid.

```
    ylim = get.cell.meta.data("ylim")
    sector.name = get.cell.meta.data("sector.index")

    if(abs(xplot[2] - xplot[1]) < 20) {
        circos.text(mean(xlim), ylim[1], sector.name, facing = "clockwise",
            niceFacing = TRUE, adj = c(0, 0.5), col = "red")
    } else {
        circos.text(mean(xlim), ylim[1], sector.name, facing = "inside",
            niceFacing = TRUE, adj = c(0.5, 0), col= "blue")
    }
}, bg.border = NA)
```

When you set direction of sector labels as radical (`clockwise` or `reverse.clockwise`), if the labels are too long and exceed your figure region, you can either decrease the size of the font or set `canvas.xlim` and `canvas.ylim` parameters in `circos.par()` to wider intervals.

14.3 Customize sector axes

Axes are helpful to correspond to the absolute values of links. By default `chordDiagram()` adds axes on the grid track. But it is easy to customize one with self-defined code.

In following example code, we draw another type of axes which show relative percent on sectors. We first pre-allocate an empty track by `preAllocateTracks` and come back to this track to add axes later.

You may see we add the first axes to the top of second track. You can also put them to the bottom of the first track.

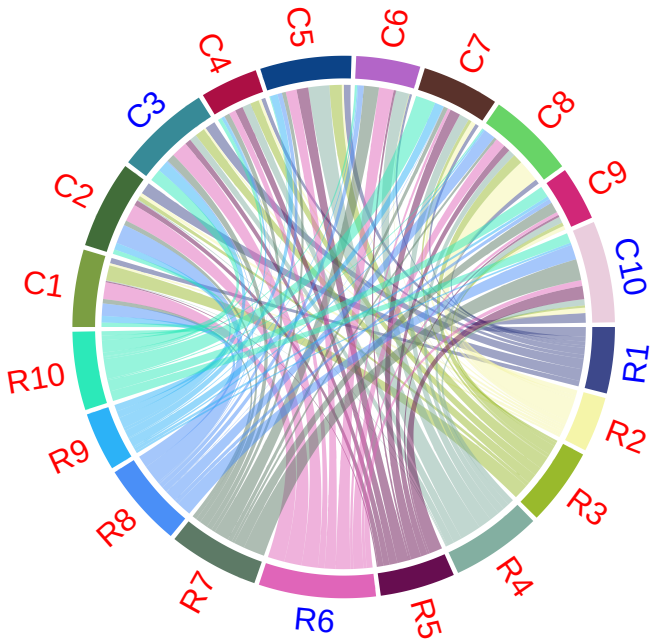


Figure 14.4: Adjust label direction according to the width of sectors.

```
# similar as the previous example, but we only plot the grid track
chordDiagram(mat, grid.col = grid.col, annotationTrack = "grid",
    preAllocateTracks = list(track.height = uh(5, "mm")))
for(si in get.all.sector.index()) {
    circos.axis(h = "top", labels.cex = 0.3, sector.index = si, track.index = 2)
}
```

Now we go back to the first track to add the second type of axes and sector names. In `panel.fun`, if the sector is less than 30 degree, the break for the axis is set to 0.5 (Figure 14.5).

```
# the second axis as well as the sector labels are added in this track
circos.track(track.index = 1, panel.fun = function(x, y) {
    xlim = get.cell.meta.data("xlim")
    ylim = get.cell.meta.data("ylim")
    sector.name = get.cell.meta.data("sector.index")
    xplot = get.cell.meta.data("xplot")

    circos.lines(xlim, c(mean(ylim), mean(ylim)), lty = 3) # dotted line
    by = ifelse(abs(xplot[2] - xplot[1]) > 30, 0.2, 0.5)
    for(p in seq(by, 1, by = by)) {
        circos.text(p*(xlim[2] - xlim[1]) + xlim[1], mean(ylim) + 0.1,
            paste0(p*100, "%"), cex = 0.3, adj = c(0.5, 0), niceFacing = TRUE)
    }

    circos.text(mean(xlim), 1, sector.name, niceFacing = TRUE, adj = c(0.5, 0))
}, bg.border = NA)
circos.clear()
```

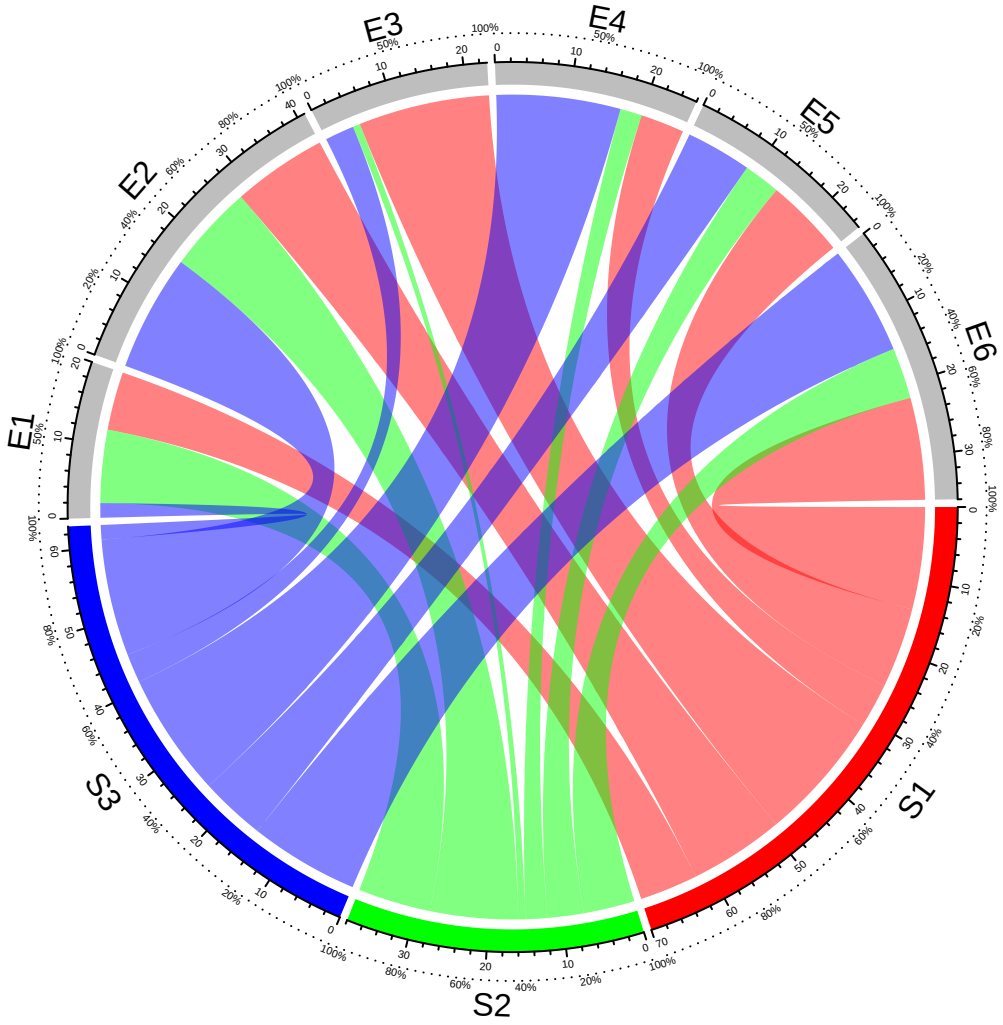


Figure 14.5: Customize sector axes for Chord diagram.

14.4 Put horizontally or vertically symmetric

In Chord diagram, when there are two groups (which correspond to rows and columns if the input is an adjacency matrix), it is always visually beautiful to rotate the diagram to be symmetric on horizontal direction or vertical direction. Actually it is quite easy to calculate a proper degree that needs to be rotated for the circle.

In the Chord diagram, the total width of row sectors corresponds to the sum of row sum of the matrix with absolute values and so is for the column sectors.

```
row_sum = sum(rowSums(abs(mat)))
col_sum = sum(colSums(abs(mat)))
```

Assume small gaps between sectors are set to 1 degree and big gaps between row and column sectors are set to 20 degree.

```
small_gap = 1
big_gap = 20
```

In the circle, there are regions which are covered by small gaps, big gaps and sectors. Since the width of sectors are proportional to the row sums and/or column sums of the matrix, it is easy to calculate how much degrees are hold by the row sectors:

```
nr = nrow(mat)
nc = ncol(mat)
n_sector = nr + nc
row_sector_degree = (360 - small_gap*(n_sector - 2) - big_gap*2) * (row_sum/(row_sum + col_sum)) +
                    small_gap*(nr-1)
```

If the row sectors are put in the right of the circle, we can calculate the "start degree" for the circle. Note `chordDiagram()` always draw row sectors first and by default the circle goes clock-wisely.

```
start_degree = 90 - (180 - row_sector_degree)/2
```

When there are small gaps and big gaps between sectors, the `gap.after` in `circos.par()` should be set as a vector. We also added a vertical line which assists to see the symmetry (Figure 14.6 left).

```
gaps = c(rep(small_gap, nrow(mat) - 1), big_gap, rep(small_gap, ncol(mat) - 1), big_gap)
circos.par(gap.after = gaps, start.degree = start_degree)
chordDiagram(mat, grid.col = grid.col)
circos.clear()
abline(v = 0, lty = 2, col = "#00000080")
```

Similarly we can adjust the "start degree" to let the circle looks horizonally symmetric (Figure 14.6 right).

```
start_degree = 0 - (180 - row_sector_degree)/2
gaps = c(rep(small_gap, nrow(mat) - 1), big_gap, rep(small_gap, ncol(mat) - 1), big_gap)
circos.par(gap.after = gaps, start.degree = start_degree)
chordDiagram(mat, grid.col = grid.col)
circos.clear()
abline(h = 0, lty = 2, col = "#00000080")
```

14.5 Compare two Chord diagrams

Normally, in Chord diagram, values in `mat` are normalized to the summation of the absolute values in the matrix, which means the width for links only represents relative values. Then, when comparing two Chord

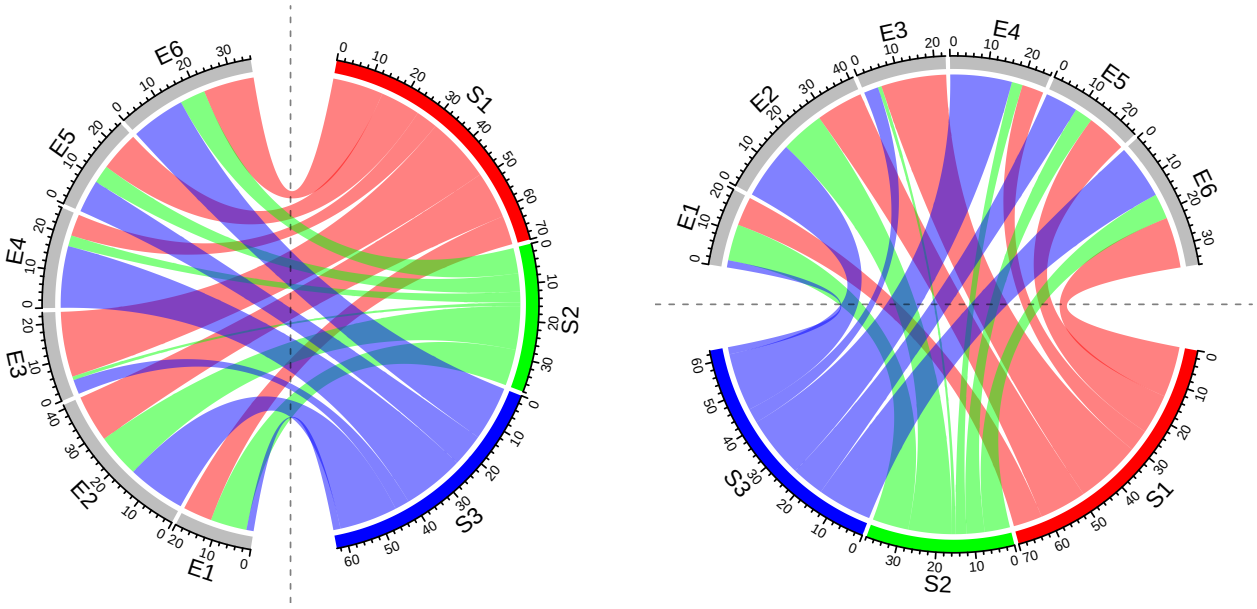


Figure 14.6: Rotate Chord diagram.

diagrams, it is necessary that unit width of links in the two plots should be represented in a same scale. This problem can be solved by adding more blank gaps to the Chord diagram which has smaller matrix.

First, let's plot a Chord diagram. In this Chord diagram, we set larger gaps between rows and columns for better visualization. Axis on the grid illustrates scale of the values.

```
mat1 = matrix(sample(20, 25, replace = TRUE), 5)
gap.after = c(rep(2, 4), 10, rep(2, 4), 10)
circos.par(gap.after = gap.after, start.degree = -10/2)
chordDiagram(mat1, directional = 1, grid.col = rep(1:5, 2))
circos.clear()
```

The second matrix only has half the values in `mat1`.

```
mat2 = mat1 / 2
```

If the second Chord diagram is plotted in the way as the first one, the two diagrams will looks exactly the same (except the axes) which makes the comparison impossible. However, we can adjust the gaps between sectors to make the scale of the two plots same.

First we calculate the percentage of `mat2` in `mat1`. And then we calculate the degree which corresponds to the difference. In the following code, `360 - sum(gap.after)` is the total degree for values in `mat1` (excluding the gaps) and `blank.degree` corresponds the difference between `mat1` and `mat2`.

```
percent = sum(abs(mat2)) / sum(abs(mat1))
blank.degree = (360 - sum(gap.after)) * (1 - percent)
```

Since now we have the additional blank gap, we can set it to `circos.par()` and plot the second Chord Diagram.

```
big.gap = (blank.degree - sum(rep(2, 8)))/2
gap.after = c(rep(2, 4), big.gap, rep(2, 4), big.gap)
circos.par(gap.after = gap.after, start.degree = -big.gap/2)
chordDiagram(mat2, directional = 1, grid.col = rep(1:5, 2), transparency = 0.5)
circos.clear()
```

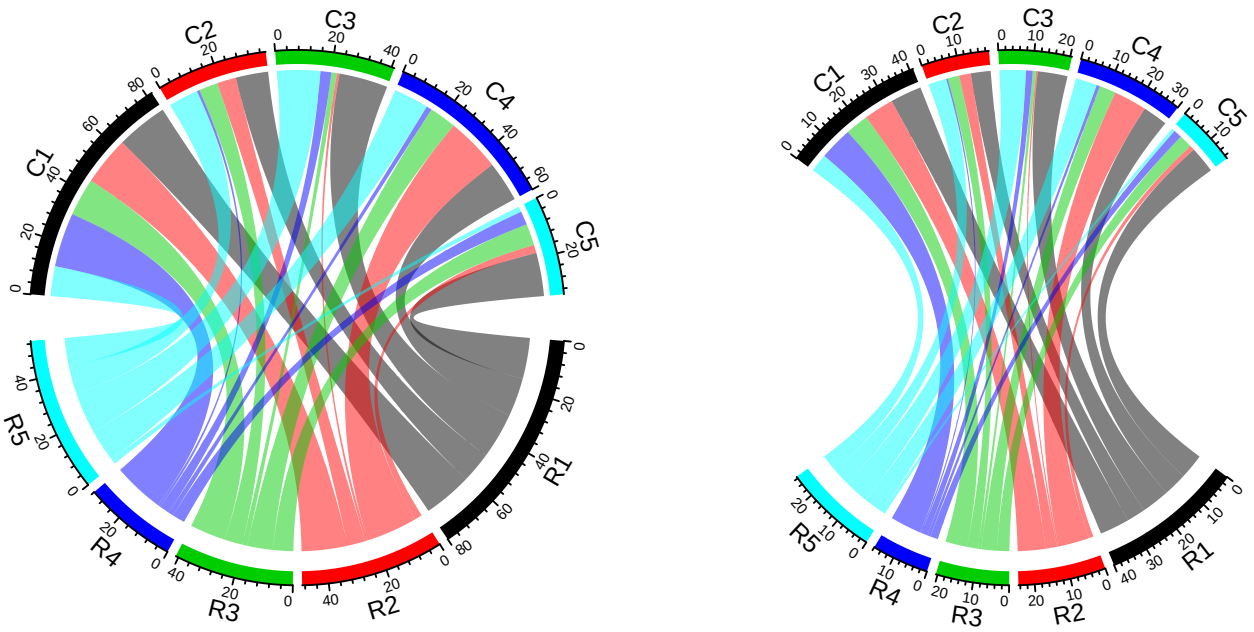


Figure 14.7: Compare two Chord Diagrams in a same scale.

Now the scale of the two Chord diagrams (Figure 14.7) are the same if you compare the scale of axes in the two diagrams.

14.6 Multiple-group Chord diagram

Generally `chordDiagram()` function visualizes relations between two groups (i.e. from rows to columns if the input is an adjacency matrix or from column 1 to column 2 if the input is an adjacency list), however, for `chordDiagram()`, it actually doesn't need any grouping information. The visual effect of grouping is just enhanced by setting different gap degrees. In this case, it is easy to make a Chord diagram with more than two groups.

First let's generate three matrix which contain pairwise relations from three groups:

```
options(digits = 2)
mat1 = matrix(rnorm(25), nrow = 5)
rownames(mat1) = paste0("A", 1:5)
colnames(mat1) = paste0("B", 1:5)

mat2 = matrix(rnorm(25), nrow = 5)
rownames(mat2) = paste0("A", 1:5)
colnames(mat2) = paste0("C", 1:5)

mat3 = matrix(rnorm(25), nrow = 5)
rownames(mat3) = paste0("B", 1:5)
colnames(mat3) = paste0("C", 1:5)
```

Since `chordDiagram()` only accepts one single matrix, here the three matrix are merged into one big matrix.

```
mat = matrix(0, nrow = 10, ncol = 10)
rownames(mat) = c(rownames(mat2), rownames(mat3))
colnames(mat) = c(colnames(mat1), colnames(mat2))
```

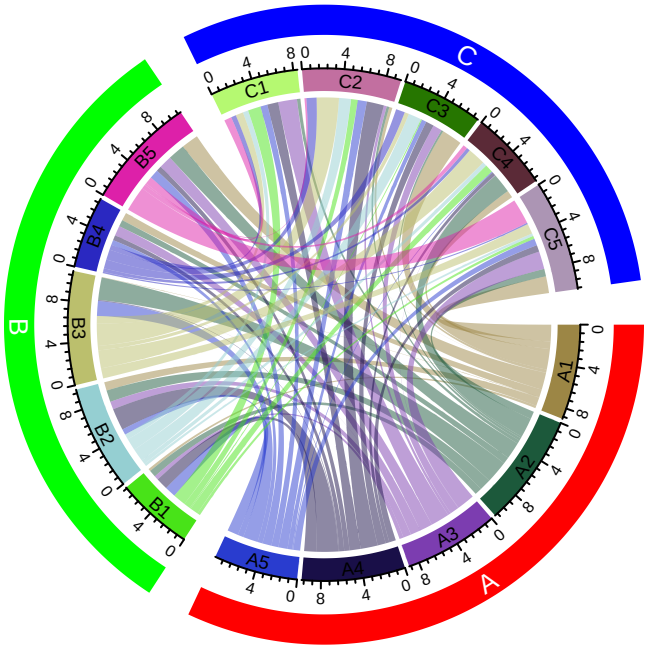
```
mat[rownames(mat1), colnames(mat1)] = mat1
mat[rownames(mat2), colnames(mat2)] = mat2
mat[rownames(mat3), colnames(mat3)] = mat3
mat
```

```
##       B1    B2     B3    B4    B5     C1    C2      C3     C4    C5
## A1 -0.26  0.90 -0.048  0.85 -1.83 -0.019 -0.33  2.1960  0.786  1.80
## A2 -0.75 -1.26 -2.399 -0.71 -1.81  0.362 -0.28 -0.2047 -2.102 -0.81
## A3  0.44  0.84 -0.019  1.07  1.37  2.011  0.31  0.9751 -0.042  1.90
## A4 -1.28 -2.35 -0.089 -0.54 -0.56 -1.178  1.84 -0.8676 -0.405  0.71
## A5  1.18  0.61 -1.595  0.54  0.97 -0.755 -0.98 -0.5012 -0.113  0.74
## B1  0.00  0.00  0.000  0.00  0.00  1.366 -0.71 -0.1262 -0.870  0.32
## B2  0.00  0.00  0.000  0.00  0.00 -0.576 -1.28 -1.3884 -0.463 -0.28
## B3  0.00  0.00  0.000  0.00  0.00 -0.805  2.38  0.4699 -1.911  1.09
## B4  0.00  0.00  0.000  0.00  0.00 -0.535 -1.09  0.9604  0.370  0.16
## B5  0.00  0.00  0.000  0.00  0.00  0.792  0.19 -0.0051 -0.462  2.71
```

When making the chord diagram, we set larger gaps between groups to identify different groups. Here we manually adjust `gap.after` in `circos.par()`.

Also we add an additional track in which we add lines to enhance the visual effect of different groups.

```
library(circlize)
circos.par(gap.after = rep(c(rep(1, 4), 8), 3))
chordDiagram(mat, annotationTrack = c("grid", "axis"),
    preAllocateTracks = list(
        track.height = uh(4, "mm"),
        track.margin = c(uh(4, "mm"), 0)
))
circos.track(track.index = 2, panel.fun = function(x, y) {
    sector.index = get.cell.meta.data("sector.index")
    xlim = get.cell.meta.data("xlim")
    ylim = get.cell.meta.data("ylim")
    circos.text(mean(xlim), mean(ylim), sector.index, cex = 0.6, niceFacing = TRUE)
}, bg.border = NA)

highlight.sector(rownames(mat1), track.index = 1, col = "red",
    text = "A", cex = 0.8, text.col = "white", niceFacing = TRUE)
highlight.sector(colnames(mat1), track.index = 1, col = "green",
    text = "B", cex = 0.8, text.col = "white", niceFacing = TRUE)
highlight.sector(colnames(mat2), track.index = 1, col = "blue",
    text = "C", cex = 0.8, text.col = "white", niceFacing = TRUE)
```




```
circos.clear()
```

If row names and column names in the big matrix are not grouped, the sector order can be manually adjusted by `order` argument.

```
chordDiagram(mat, order = c(paset0("A", 1:5), paset0("B", 1:5), paset0("C", 1:5)))
```

It is similar way to construct a multiple-group Chord diagram with data frame as input.

```
library(reshape2)
df2 = do.call("rbind", list(melt(mat1), melt(mat2), melt(mat3)))
chordDiagram(df2, order = c(paste0("A", 1:5), paste0("B", 1:5), paste0("C", 1:5)))
```

Chapter 15

A complex example of Chord diagram

In this chapter, we demonstrate how to make a complex Chord diagram and how to customize additional tracks by visualizing chromatin state transitions as well as methylation changes. A chromatin state transition matrix shows how much a chromatin state in the genome has been changed from e.g. one group of samples to the other. The genomic regions for which the chromatin states change also have variable methylation patterns which may show interesting correspondance to chromatin states change.

The data used in this post is processed from Roadmap dataset. The chromatin states are learned from five core chromatin marks. Roadmap samples are classified into two groups based on expression profile. In each group, a chromatin state is assigned to the corresponding genomic bin if it is recurrent in at least half of the samples in each group.

The processed data can be set from http://jokergoo.github.io/data/chromatin_transition.RData.

```
load("data/chromatin_transition.RData")
```

In the RData file, there are three matrix: `mat`, `meth_mat_1` and `meth_mat_2` which are:

- `mat`: chromatin state transition matrix. Rows correspond to states in group 1 and columns correspond to group 2. The value in the matrix are total base pairs that transite from one state to the other. E.g. `mat["TssA", "TxFlnk"]` is the total base pairs that have "TssA" state in samples in group 1 and transites to "TxFlnk" state in samples in group 2. On the diagonal are the regions where the states have not been changed in the two groups.
- `meth_mat_1`: mean methylation for each set of regions in group 1. E.g. `meth_mat_1["TssA", "TxFlnk"]` is the mean methylation for the regions in **group 1** that have "TssA" state in group 1 and "TxFlnk" state in group 2.
- `meth_mat_2`: mean methylation for each set of regions in group 2. E.g. `meth_mat_2["TssA", "TxFlnk"]` is the mean methylation for the regions in **group 2** that have "TssA" state in group 1 and "TxFlnk" state in group 2.

```
mat[1:4, 1:4]
```

```
##            TssA TssAFlnk TxFlnk     Tx
## TssA     497200    79600  13400   1800
## TssAFlnk  56400   233200   5000    800
## TxFlnk        0      400  43000   1800
## Tx          800      200    200 166400
```

```
meth_mat_1[1:4, 1:4]
```

```
##               TssA  TssAFlnk    TxFlnk        Tx
## TssA     0.1647232 0.1580874 0.1917435 0.2690045
```

```
## TssAFlnk 0.2591677 0.2689880 0.3616242 0.3411387
## TxFlnk          NA 0.3697514 0.3360386 0.4752722
## Tx       0.8268626 0.7822987 0.5799682 0.6595322
```

Normally, in majority in the genome, chromatin states of regions are not changed in the two groups, thus, we should only look at the regions in which the states are changed.

```
# proportion of the unchanges states in the genome
sum(diag(mat))/sum(mat)
```

```
## [1] 0.6192262
```

```
# remove the unchanged states
diag(mat) = 0
```

When making the plot, actually rows and columns are different (because one is from group 1 and the other is from group 2), thus we give them different names and the original names are stored in `all_states`.

```
all_states = rownames(mat)
n_states = nrow(mat)

rownames(mat) = paste0("R_", seq_len(n_states))
colnames(mat) = paste0("C_", seq_len(n_states))

dimnames(meth_mat_1) = dimnames(mat)
dimnames(meth_mat_2) = dimnames(mat)
```

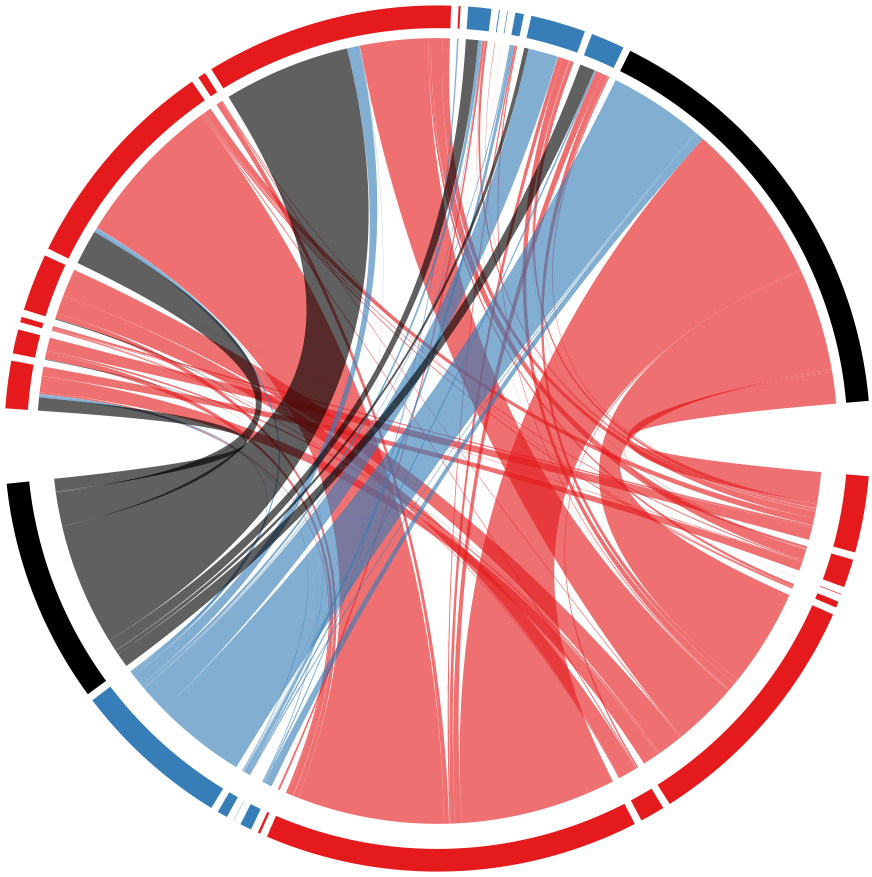
Next we set the colors. `colmat` is the color of the links and the colors are represent as hexadecimal code. Links have more transparent (`A0`) if they contain few transitions (< 70th percentile) because we don't want it to disturb the visualization of the major transitions.

```
state_col = c("TssA" = "#E41A1C",     "TssAFlnk" = "#E41A1C",
              "TxFlnk" = "#E41A1C",   "Tx" = "#E41A1C",
              "TxWk" = "#E41A1C",     "EnhG" = "#E41A1C",
              "Enh" = "#E41A1C",      "ZNF/Rpts" = "#E41A1C",
              "Het" = "#377EB8",      "TssBiv" = "#377EB8",
              "BivFlnk" = "#377EB8", "EnhBiv" = "#377EB8",
              "ReprPC" = "#377EB8",   "ReprPCWk" = "#377EB8",
              "Quies" = "black")

# one for rows and one for columns
state_col2 = c(state_col, state_col)
names(state_col2) = c(rownames(mat), colnames(mat))

colmat = rep(state_col2[rownames(mat)], n_states)
colmat = rgb(t(col2rgb(colmat)), maxColorValue = 255)

qati = quantile(mat, 0.7)
colmat[mat > qati] = paste0(colmat[mat > qati], "A0")
colmat[mat <= qati] = paste0(colmat[mat <= qati], "20")
dim(colmat) = dim(mat)
```

Now we can use `chordDiagram()` function to make the plot. Here we set one pre-allocated track in which the methylation information will be added later. Also we only set `annotationTrack` to `grid` and the axes and sector labels will be customized in later code.

`chordDiagram()` returns a data frame which contains coordinates for all links.

```
circos.par(start.degree = -5, gap.after = c(rep(1, n_states-1), 10, rep(1, n_states-1), 10),
    cell.padding = c(0, 0, 0, 0), points.overflow.warning = FALSE)

cdm_res = chordDiagram(mat, col = colmat, grid.col = state_col2,
    directional = TRUE, annotationTrack = "grid",
    preAllocateTracks = list(track.height = 0.1))
```



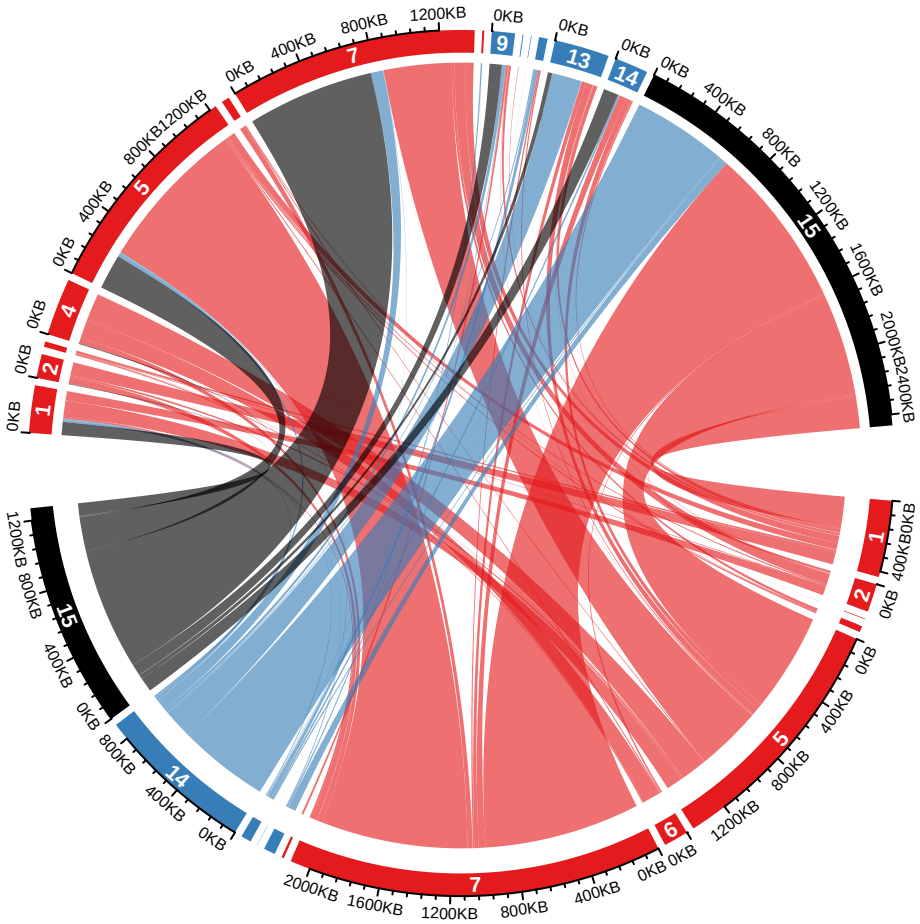
```
head(cdm_res)
```

```
##    rn  cn value o1 o2      x1     x2       col
## 1 R_1 C_1     0 15 13  431200 267200 #E41A1C20
## 2 R_2 C_1 56400 15 12  159800 267200 #E41A1CA0
## 3 R_3 C_1     0 15 11    3600 210800 #E41A1C20
## 4 R_4 C_1   800 15 10   34600 210800 #E41A1C20
## 5 R_5 C_1 98200 15  9 1411600 210000 #E41A1CA0
## 6 R_6 C_1     0 15  8  139800 111800 #E41A1C20
```

Now the axes are added in the second track, also, the index of states are added at the center of the grids in the second track, if the degree for a sector is larger than 3 degrees. Note since there is already one pre-allocated track, the circular rectangles are in the second track (`track.index = 2`).

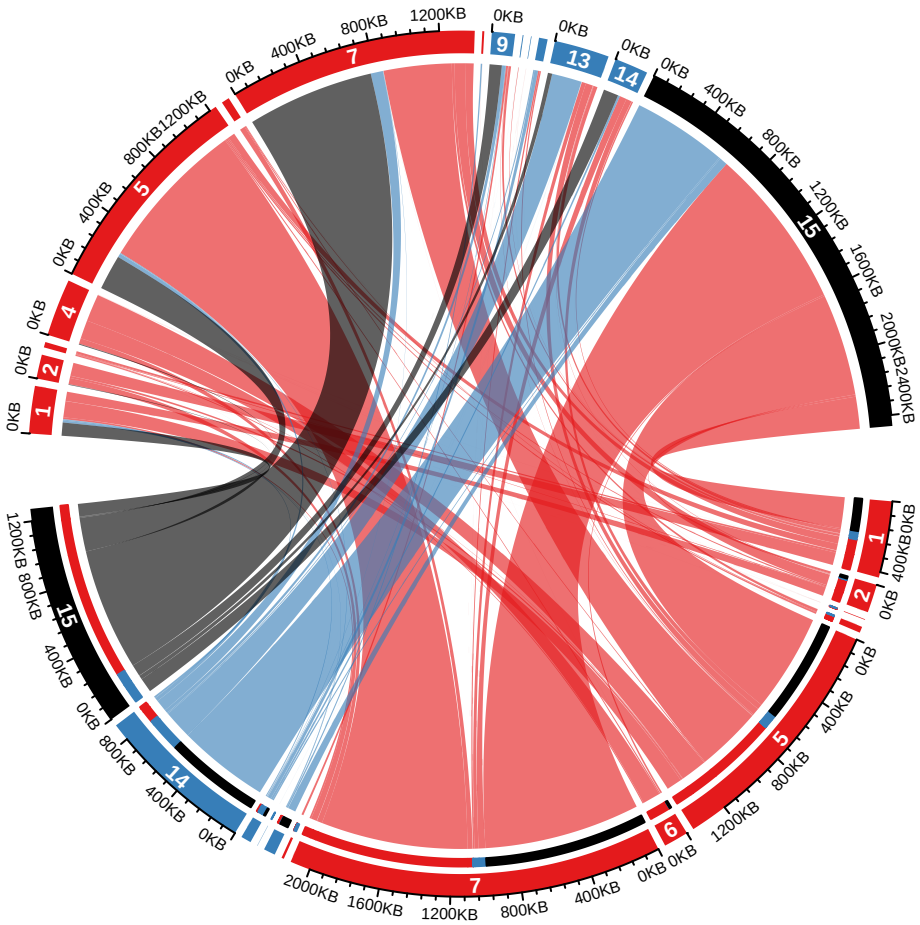
```
circos.track(track.index = 2, panel.fun = function(x, y) {
    if(abs(CELL_META$cell.start.degree - CELL_META$cell.end.degree) > 3) {
        sn = CELL_META$sector.index
        i_state = as.numeric(gsub("(C|R)_", "", sn))
        circos.text(CELL_META$xcenter, CELL_META$ycenter, i_state, col = "white",
            font = 2, cex = 0.7, adj = c(0.5, 0.5), niceFacing = TRUE)
```

```
        xlim = CELL_META$xlim
        breaks = seq(0, xlim[2], by = 4e5)
        circos.axis(major.at = breaks, labels = paste0(breaks/1000, "KB"), labels.cex = 0.5)
    }
}, bg.border = NA)
```



On the top half, it is easy to see the proportion of different transitions in group 1 that come to every state in group 2. However, it is not straightforward for the states in the bottom half to see the proportion of different states in group 2 they transite to. This can be solved by adding small circular rectangles to represent the proportions. In following example, the newly added circular rectangles in the bottom half show e.g. how much the state 15 in group 1 has been transited to different states in group 2.

```
for(i in seq_len(nrow(cdm_res))) {
    if(cdm_res$value[i] > 0) {
        circos.rect(cdm_res[i, "x1"], -uy(1, "mm"),
            cdm_res[i, "x1"] - abs(cdm_res[i, "value"]), -uy(2, "mm"),
            col = state_col2[cdm_res$cn[i]], border = state_col2[cdm_res$cn[i]],
            sector.index = cdm_res$rn[i], track.index = 2)
    }
}
```

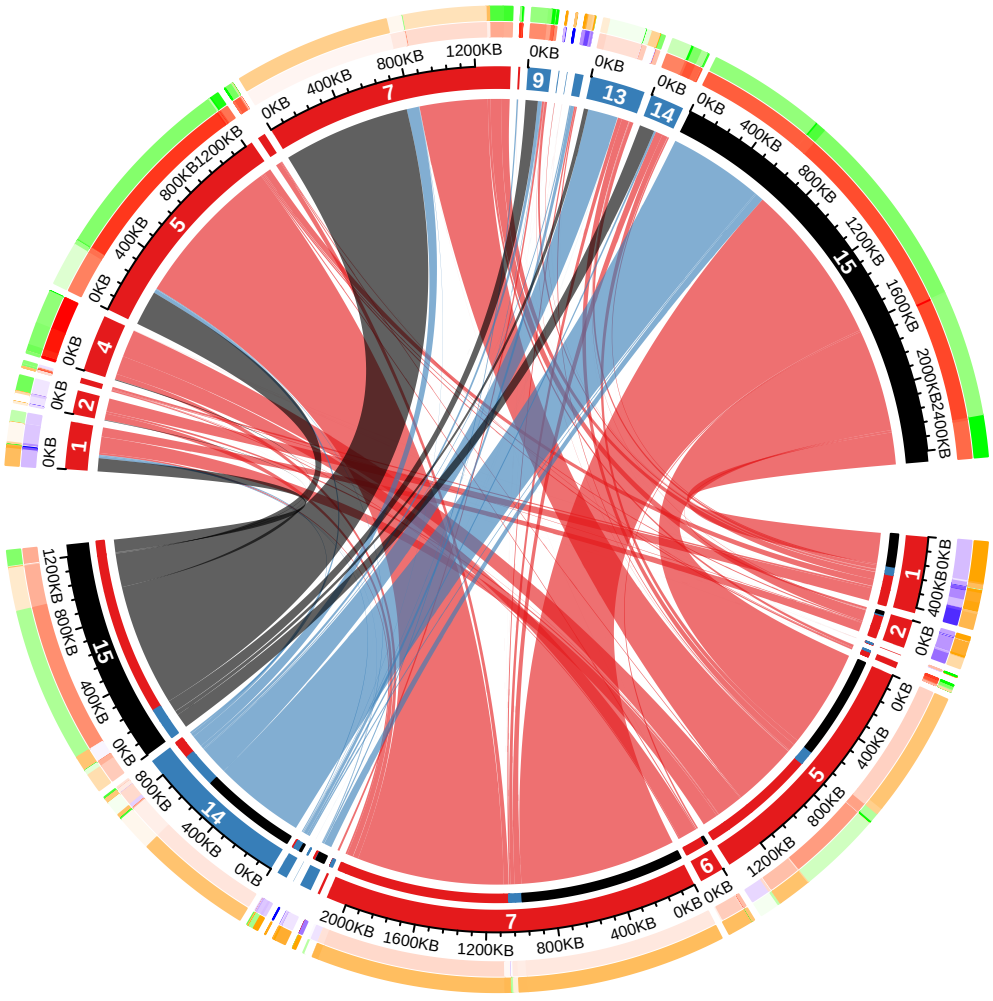


Methylation in each category is put on the most outside of the circle. On this track, we will put two paralle rectangles which are mean methylation and methylation difference between group 1 and group 2. Basically, on the bottom, we show `meth_mat_2 - meth_mat_1` and on the top we show `meth_mat_1 - meth_mat_2`.

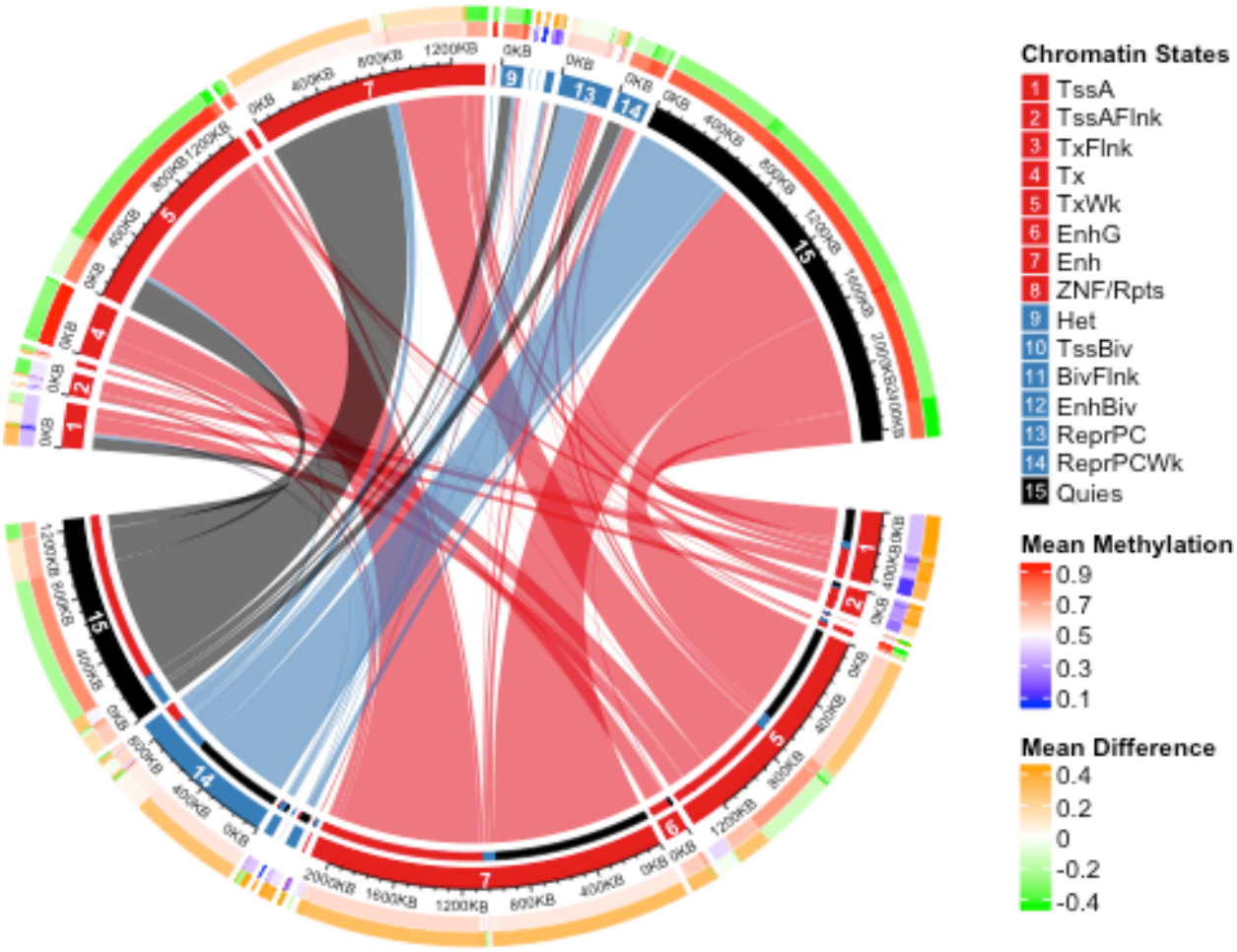
The logic of following code is simple that it just simply adds rectangles repeatedly.

```
abs_max = quantile(abs(c(meth_mat_1, meth_mat_2) - 0.5), 0.95, na.rm = TRUE)
col_fun = colorRamp2(c(0.5 - abs_max, 0.5, 0.5 + abs_max), c("blue", "white", "red"))
col_fun2 = colorRamp2(c(-abs_max, 0, abs_max), c("green", "white", "orange"))

ylim = get.cell.meta.data("ylim", sector.index = rownames(mat)[1], track.index = 1)
y1 = ylim[1] + (ylim[2] - ylim[1])*0.4
y2 = ylim[2]
for(i in seq_len(nrow(cdm_res))) {
    if(cdm_res$value[i] > 0) {
        circos.rect(cdm_res[i, "x1"], y1, cdm_res[i, "x1"] - abs(cdm_res[i, "value"]), y1 + (y2-y1)*0.45
            col = col_fun(meth_mat_1[cdm_res$rn[i], cdm_res$cn[i]]),
            border = col_fun(meth_mat_1[cdm_res$rn[i], cdm_res$cn[i]]),
            sector.index = cdm_res$rn[i], track.index = 1)

        circos.rect(cdm_res[i, "x1"], y1 + (y2-y1)*0.55, cdm_res[i, "x1"] - abs(cdm_res[i, "value"]), y1
            col = col_fun2(meth_mat_2[cdm_res$rn[i], cdm_res$cn[i]] - meth_mat_1[cdm_res$rn[i], cdm_res
            border = col_fun2(meth_mat_2[cdm_res$rn[i], cdm_res$cn[i]] - meth_mat_1[cdm_res$rn[i], cdm_
            sector.index = cdm_res$rn[i], track.index = 1)
```

```
        circos.rect(cdm_res[i, "x2"], y1, cdm_res[i, "x2"] - abs(cdm_res[i, "value"]), y1 + (y2-y1)*0.45
            col = col_fun(meth_mat_2[cdm_res$rn[i], cdm_res$cn[i]]),
            border = col_fun(meth_mat_2[cdm_res$rn[i], cdm_res$cn[i]]),
            sector.index = cdm_res$cn[i], track.index = 1)

        circos.rect(cdm_res[i, "x2"], y1 + (y2-y1)*0.55, cdm_res[i, "x2"] - abs(cdm_res[i, "value"]), y1
            col = col_fun2(meth_mat_1[cdm_res$rn[i], cdm_res$cn[i]] - meth_mat_2[cdm_res$rn[i], cdm_res$
            border = col_fun2(meth_mat_1[cdm_res$rn[i], cdm_res$cn[i]] - meth_mat_2[cdm_res$rn[i], cdm_r
            sector.index = cdm_res$cn[i], track.index = 1)
    }
}
circos.clear()
```



Legends can be added according to instructions discussed in Section 4.


Chromatin States
1 TssA
2 TssAFlnk
3 TxFlnk
4 Tx
5 TxWk
6 EnhG
7 Enh
8 ZNF/Rpts
9 Het
10 TssBiv
11 BivFlnk
12 EnhBiv
13 ReprPC
14 ReprPCWk
15 Quies
Mean Methylation
0.9
0.7
0.5
0.3
0.1
Mean Difference
0.4
0.2
0
-0.2
-0.4

Part IV

Others

Chapter 16

Make fun of the package

16.1 A clock

The first example is a clock. The key function here is `circos.axis()` (Figure 16.1). In the example, the whole circle only contains one sector in which major tick at 0 is overlapping with major tick at 12.

Later we calculate the positions of the hour hand, the minute hand and the second hand based on current time when this Chapter is generated. The hands are drawn by `arraws()` function in the canvas coordinate. A real-time clock can be found at the **Examples** section in the help page of `circos.axis()`.

```
circos.par(gap.degree = 0, cell.padding = c(0, 0, 0, 0), start.degree = 90)
circos.initialize(factors = "a", xlim = c(0, 12))
circos.track(ylim = c(0, 1), bg.border = NA)
circos.axis(major.at = 0:12, labels = NULL, direction = "inside",
    major.tick.length = uy(2, "mm"))
circos.text(1:12, rep(1, 12) - uy(6, "mm"), 1:12, facing = "downward")

current.time = as.POSIXlt(Sys.time())
sec = ceiling(current.time$sec)
min = current.time$min
hour = current.time$hour

sec.degree = 90 - sec/60 * 360
arrows(0, 0, cos(sec.degree/180*pi)*0.8, sin(sec.degree/180*pi)*0.8)

min.degree = 90 - min/60 * 360
arrows(0, 0, cos(min.degree/180*pi)*0.7, sin(min.degree/180*pi)*0.7, lwd = 2)

hour.degree = 90 - hour/12 * 360 - min/60 * 360/12
arrows(0, 0, cos(hour.degree/180*pi)*0.4, sin(hour.degree/180*pi)*0.4, lwd = 2)
```

```
circos.clear()
```

16.2 A dartboard

The second example is a dartboard. In Figure 16.2, tracks are assigned with different heights and each cell is initialized with different colors. The most inside green ring and red circle are plotted by `draw.sector()`.

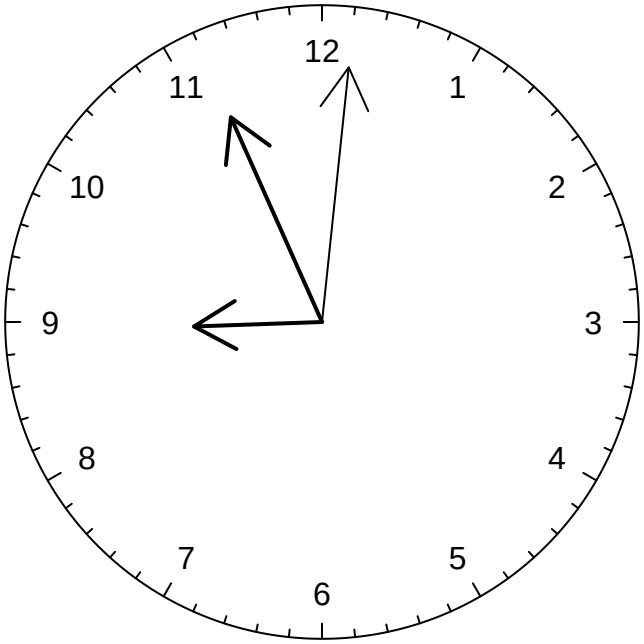


Figure 16.1: A clock.

We use `circos.trackText()` to add labels because we know the global order of the labels.

Now you can project the plot on your door and begin to play!

```
factors = 1:20  # just indicate there are 20 sectors
circos.par(gap.degree = 0, cell.padding = c(0, 0, 0, 0),
    start.degree = 360/20/2, track.margin = c(0, 0), clock.wise = FALSE)
circos.initialize(factors = factors, xlim = c(0, 1))

circos.track(ylim = c(0, 1), factors = factors, bg.col = "black", track.height = 0.15)
circos.trackText(x = rep(0.5, 20), y = rep(0.5, 20),
    labels = c(13, 4, 18, 1, 20, 5, 12, 9, 14, 11, 8, 16, 7, 19, 3, 17, 2, 15, 10, 6),
    cex = 0.8, factors = factors, col = "#EEEEEE", font = 2, facing = "downward")
circos.track(ylim = c(0, 1), factors = factors,
    bg.col = rep(c("#E41A1C", "#4DAF4A"), 10), bg.border = "#EEEEEE", track.height = 0.05)
circos.track(ylim = c(0, 1), factors = factors,
    bg.col = rep(c("black", "white"), 10), bg.border = "#EEEEEE", track.height = 0.275)
circos.track(ylim = c(0, 1), factors = factors,
    bg.col = rep(c("#E41A1C", "#4DAF4A"), 10), bg.border = "#EEEEEE", track.height = 0.05)
circos.track(ylim = c(0, 1), factors = factors,
    bg.col = rep(c("black", "white"), 10), bg.border = "#EEEEEE", track.height = 0.375)

draw.sector(center = c(0, 0), start.degree = 0, end.degree = 360,
    rou1 = 0.1, col = "#4DAF4A", border = "#EEEEEE")
draw.sector(center = c(0, 0), start.degree = 0, end.degree = 360,
    rou1 = 0.05, col = "#E41A1C", border = "#EEEEEE")
```

```
circos.clear()
```

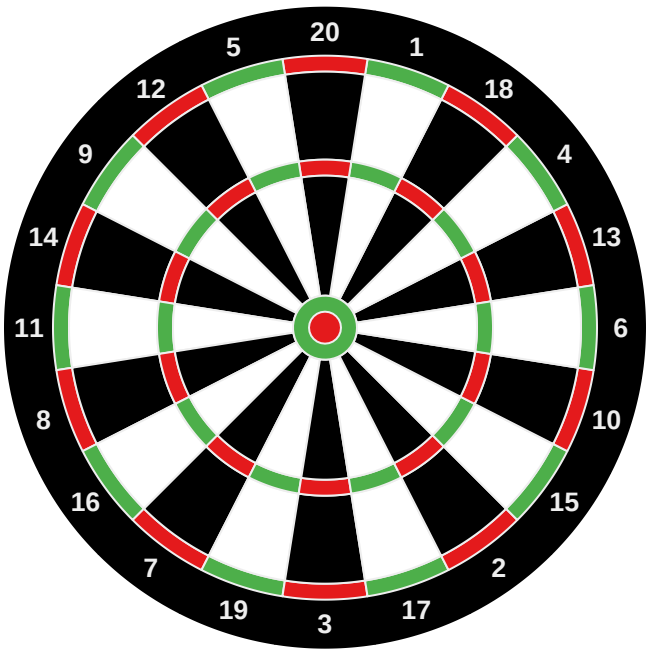


Figure 16.2: A dartboard.

16.3 Ba-Gua and Tai-Ji

The third example is Ba-Gua. The key functions are `circos.rect()` and `draw.sector()` (Figure 16.3).

Bagua was originated several thousands years ago in China. It is the source of almost all ancient Chinese philosophy. It abstracts the rule of universe into base signs and combination of the two basic signs generates the whole system of the universe.

Inside Ba-Gua, these is the Tai-Ji. Tai-Ji refers to the most original state at the creation of the universe. In ancient Chinese philosophy system, at the very beginning, the whole world is a huge mass of air (chaos). Then the lighter air floated up and created sky while heavier sinked down and created ground. The upper world is called Yang and the bottom world is called Ying. And that is Tai-Ji.

Looking at Tai-Ji, you can see there are two states interacting with each other. The white one and the black one gradually transformed into each other at the end. And in the center of white and black, the opposite color is generated. In real world, Taiji can represent all phenomenon that is of dualism. Such as male and female, correct and wrong. However things would change, good thing would become bad thing as time goes by, and bad thing would also turn into good according to how you look at the world.

```
factors = 1:8
circos.par(start.degree = 22.5, gap.degree = 6)
circos.initialize(factors = factors, xlim = c(0, 1))

# yang yao is __ (a long segment)
add_yang_yao = function() {
    circos.rect(0,0,1,1, col = "black")
}

# yin yao is -- (two short segments)
add_yin_yao = function() {
    circos.rect(0,0,0.45,1, col = "black")
    circos.rect(0.55,0,1,1, col = "black")
```

```
}
circos.track(ylim = c(0, 1), factors = factors, bg.border = NA,
    panel.fun = function(x, y) {
        i = get.cell.meta.data("sector.numeric.index")
        if(i %in% c(2, 5, 7, 8)) add_yang_yao() else add_yin_yao()
}, track.height = 0.1)

circos.track(ylim = c(0, 1), factors = factors, bg.border = NA,
    panel.fun = function(x, y) {
        i = get.cell.meta.data("sector.numeric.index")
        if(i %in% c(1, 6, 7, 8)) add_yang_yao() else add_yin_yao()
    }, track.height = 0.1)

circos.track(ylim = c(0, 1), factors = factors, bg.border = NA,
    panel.fun = function(x, y) {
        i = get.cell.meta.data("sector.numeric.index")
        if(i %in% c(4, 5, 6, 7)) add_yang_yao() else add_yin_yao()
    }, track.height = 0.1)

# the bottom of the most recent track
r = get.cell.meta.data("cell.bottom.radius") - 0.1
# draw taiji, note default order is clock wise for `draw.sector`
draw.sector(center = c(0, 0), start.degree = 90, end.degree = -90,
    rou1 = r, col = "black", border = "black")
draw.sector(center = c(0, 0), start.degree = 270, end.degree = 90,
    rou1 = r, col = "white", border = "black")
draw.sector(center = c(0, r/2), start.degree = 0, end.degree = 360,
    rou1 = r/2, col = "white", border = "white")
draw.sector(center = c(0, -r/2), start.degree = 0, end.degree = 360,
    rou1 = r/2, col = "black", border = "black")
draw.sector(center = c(0, r/2), start.degree = 0, end.degree = 360,
    rou1 = r/8, col = "black", border = "black")
draw.sector(center = c(0, -r/2), start.degree = 0, end.degree = 360,
    rou1 = r/8, col = "white", border = "white")
```

```
circos.clear()
```

16.4 Circular doodle

Figure 16.4 is a circular style of Keith Haring's doodle. The circular transformation is as follows: 1. use **jpeg** package to read RGB values in the original figure; 2. use `circos.rect()` to draw every pixel as tiny rectangles into the circle. Source code for generating the figure can be found at http://jokergoo.github.io/circlize/example/doodle.html (note this implementation is quite slow to run).

Figure 16.3: Ba-Gua and Tai-Ji.

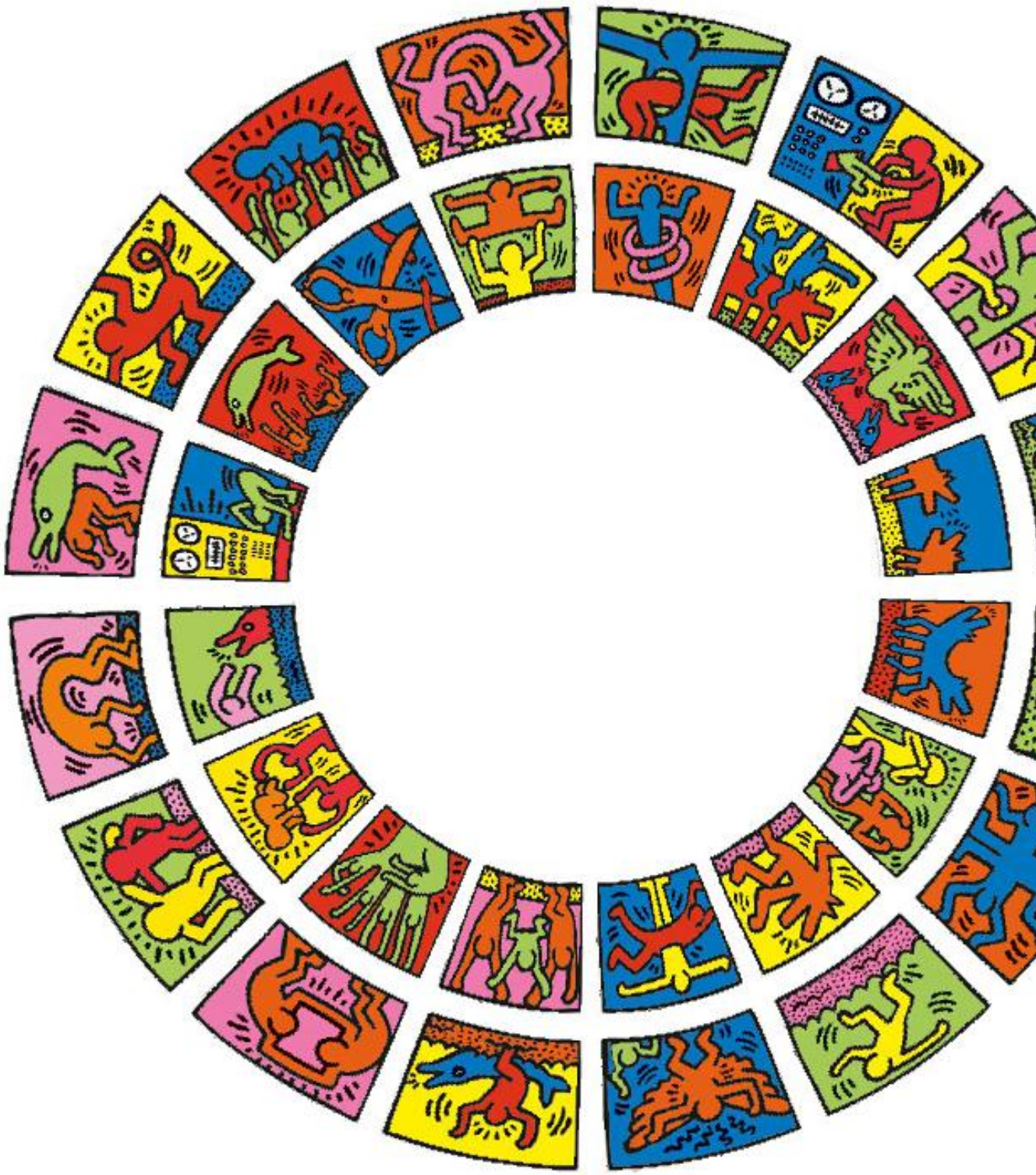


Figure 16.4: Keith Haring doodle in circular layout.

Bibliography