Visualize Relations by Chord Diagram

Zuguang Gu <z.gu@dkfz.de>

November 25, 2015

One unique feature of circular layout is the links (or connectors) to represent relations between elements (http://circos.ca/intro/tabular_visualization/). The name of such plot is sometimes called Chord Diagram (http://en.wikipedia.org/wiki/Chord_diagram). In **circlize**, it is easy to plot it in a straightforward or 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 element in the i^{th} row and the element in the j^{th} column. 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 end at the second column, the third column represents the strength of the relation.

In following code, there is a connection from `b` to `B` and the strength of the connection is 5.

```
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
```

While in following code, there is a connection from `b` to `B` with the strength 2.

```
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
```

In **circlize** package, there is a `chordDiagram` function that supports both adjacency matrix and adjacency list. The function will dispatch to `chordDiagramFromMatrix` or `chordDiagramFromDataFrame` according to whether the input is a matrix or a data frame. Internally, the adjacency matrix is converted to a adjacency list and makes the final Chord diagram by `chordDiagramFromDataFrame`, but the adjacency matrix still has some unique features such visualizing a sysmmetric matrix. Users are encouraged to use `chordDiagram` directly instead of using the two dispatched functions.

1 The `chordDiagram` function

Because for adjacency matrix and adjacency list, most of the settings are the same, we only generate figures with using the matrix. But still the code for using the adjacency list is demonstrated without executing and the figures are generated only if necessary.

1.1 Basic usage

First let's generate a random matrix and the corresponding adjacency list. We also eliminate several rows in the data frame for users to see the difference.

```
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 = df[sample(18, 10), ]
df
```

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

The most simple usage is just calling `chordDiagram` with `mat` (figure 1 A).

```
chordDiagram(mat)
circos.clear()
```

or call with `df`:

```
# code is not run when building the vignette
chordDiagram(df)
circos.clear()
```

The default Chord Diagram consists of a track of labels, a track of grids with axes added, and links. Under default settings, the grid colors are randomly generated, and the link colors are same as grid colors which correspond to rows but with 50% transparency. 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.

Since Chord Diagram is implemented by **circlize** package, like normal circos plot, there are a lot of settings that can be configured.

The gaps between sectors can be set through `circos.par` (figure 1 B). It is useful when rows and columns are different measurements (as in `mat`). Please note since you change default circos graphical settings, you need to use `circos.clear` in the end to reset it.

```
circos.par(gap.degree = c(rep(2, nrow(mat)-1), 10, rep(2, ncol(mat)-1), 10))
chordDiagram(mat)
circos.clear()
```

For the data frame:

```
# code is not run when building the vignette
circos.par(gap.degree = c(rep(2, length(unique(df[[1]]))-1), 10,
                          rep(2, length(unique(df[[2]]))-1), 10))
chordDiagram(df)
circos.clear()
```

Similarly, the start degree for the first sector can also be set through `circos.par` (figure 1 C). Since this setting is the same for using adjacency matrix and adjacency list, we will not show the code with using the adjacency list and so is in the remaining part of this vignette.

```
circos.par(start.degree = 90)
chordDiagram(mat)
circos.clear()
```

The order of sectors can be controlled by `order` argument (figure 1 D). 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"))
```

1.2 Colors

Setting colors is flexible. Colors for grids can be set through `grid.col`. Values of `grid.col` should be a named vector of which names correspond to sector names. If `grid.col` has no name index, the order of `grid.col` corresponds to the order of sector names. As explained before, the default link colors are same as grids which correspond to rows or the first column for the data frame(figure 2 A).

```
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)
```

Transparency of link colors can be set through `transparency` (figure 2 B). The value should between 0 to 1 in which 0 means no transparency and 1 means full transparency. Default transparency is 0.5. A value of `NA` means just to ignore the transparency setting.

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

For adjacecy matrix, colors for links can be customized by providing a matrix of colors which correspond to `mat`. In the following example, we use `rand_color` which is shipped by **circlize** package to generate a random color matrix. Note since `col_mat` already contains transparency, `transparency` will be ignored if it is set (figure 2 C).

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

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

```
# code is not run when building the vignette
col = rand_color(nrow(df))
chordDiagram(df, grid.col = grid.col, col = col)
```

`col` argument can also be a self-defined function which maps values to colors. Here we use `colorRamp2` which is available in **circlize** to generate a function with a list of break points and corresponding colors (figure 2 D). This functionality also works for adjacency list.

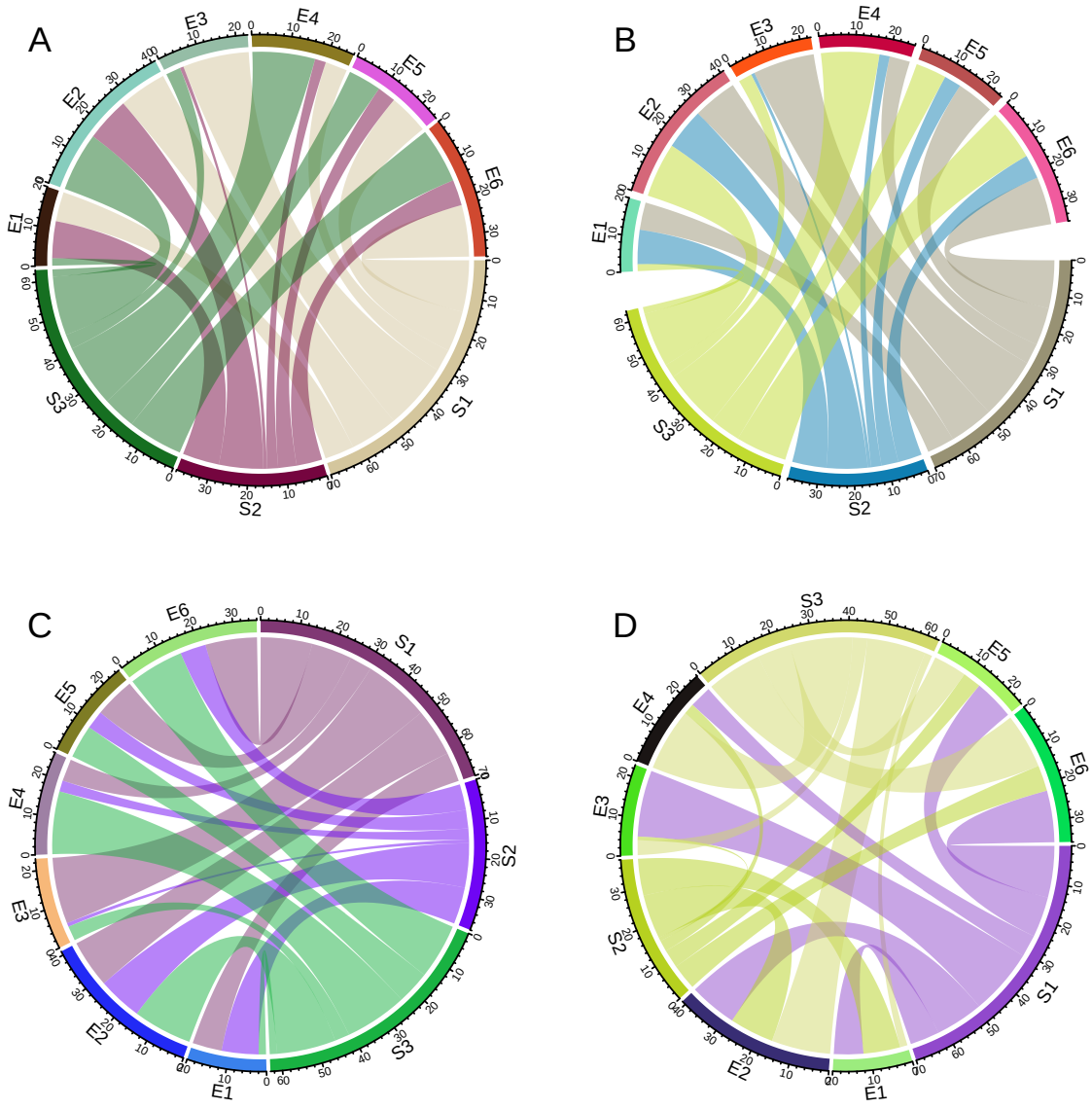

Figure 1: Basic usages of `chordDiagram`. A) default style; B) set `gap.degree`; C) set `start.degree`; D) set orders of sectors.

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

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 2 E, F). Here note values of colors can be set as numbers, color names or hex code, same as in the traditional R graphics.

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

There is no similar settings as `row.col` for ajacency list.

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

In **Highlight links** section, we will introduce how to enhance subset of links by only setting colors to them.

1.3 Other graphic settings for links

`link.lwd`, `link.lty` and `link.border` control the style of link border. All these three parameters can be set either a single scalar or a matrix with names.

If it is set as a single scale, it means all links share the same style (figure 3 A).

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

If it is set as a matrix, it should correspond to rows and columns in `mat` (figure 3 B).

```
lwd_mat = matrix(1, nrow = nrow(mat), ncol = ncol(mat))
rownames(lwd_mat) = rownames(mat)
colnames(lwd_mat) = colnames(mat)
lwd_mat[mat > 12] = 2

border_mat = matrix(NA, nrow = nrow(mat), ncol = ncol(mat))
rownames(border_mat) = rownames(mat)
colnames(border_mat) = colnames(mat)
border_mat[mat > 12] = "black"

chordDiagram(mat, grid.col = grid.col, link.lwd = lwd_mat, link.border = border_mat)
```

The matrix is not necessary to have same number of rows and columns as in `mat`. It can also be a sub matrix (figure 3 C). For rows or columns of which the corresponding values are not specified in the matrix, default values are filled in. But no matter it is a full matrix or a sub matrix, 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, and the third column corresponds to the graphic parameters (figure 3 D).

```
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)
```

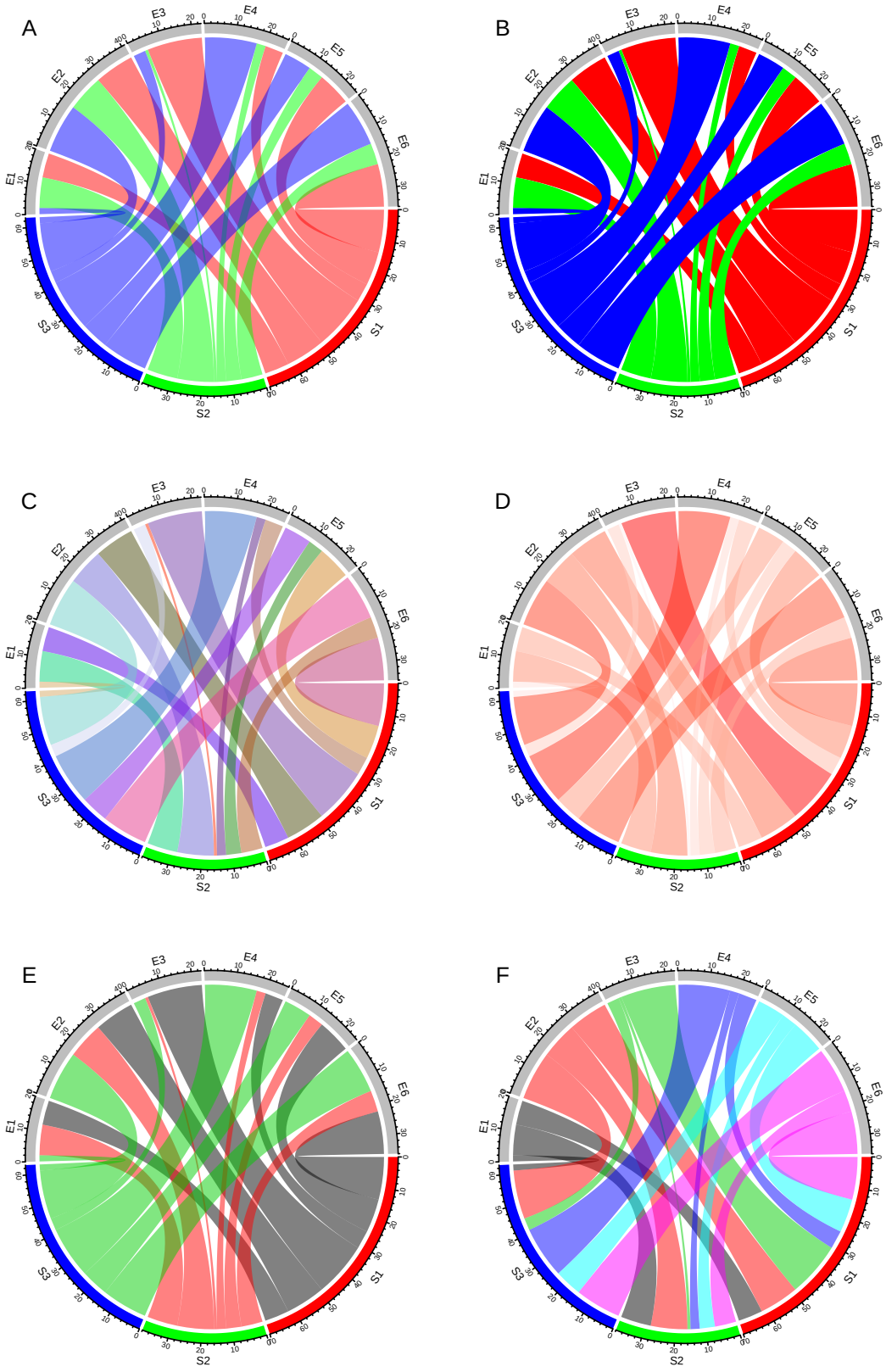

Figure 2: Color settings in `chordDiagram`. A) set `grid.col`; B) set `transparency`; C) set `col` as a matrix; D) set `col` as a function; E) set `row.col`; F) set `column.col`.

For adjacency list, it is much easier, you only need to set these graphical settings as a vecotr.

```
# code is not run when building the vignette
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))
```

1.4 Highlight links by colors

Sometimes we want to highlight some links to emphasize the importance of such connections. Highlighting by different link styles are introduced in previous section and here we focus on highlighting by colors.

The key point for highlighting by colors is to set different color transparency to different links. For example, if we want to highlight links which correspond to state "S1", we can set `row.col` with different transparency (figure 4 A).

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

We can also highlight links with values larger than a cutoff (figure 4 B, C). There are at least three ways to do it. First, construct a color matrix and set links with small values to full transparency.

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

Second, use a color function to generate different colors with different values. Note this is also workable for adjacency list.

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

Third, use a three-column data frame to assign colors to links of interest (figure 4 D).

```
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)
```

Just remember if you want to highlight links by colors, make sure to set high or full transparency to the links that you want to ignore.

Again, for adjacency list, you only need to construct a vector of colors according to what links you want to highlight.

```
# code is not run when building the vignette
col = rand_color(nrow(df))
col[df[[3]] < 10] = "#00000000"
chordDiagram(df, grid.col = grid.col, col = col)
```

1.5 Sort links 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 link orders (figure 5).

```
chordDiagram(mat, grid.col = grid.col, link.border = 1,
text(-0.9, 0.9, "A", cex = 1.5)
```

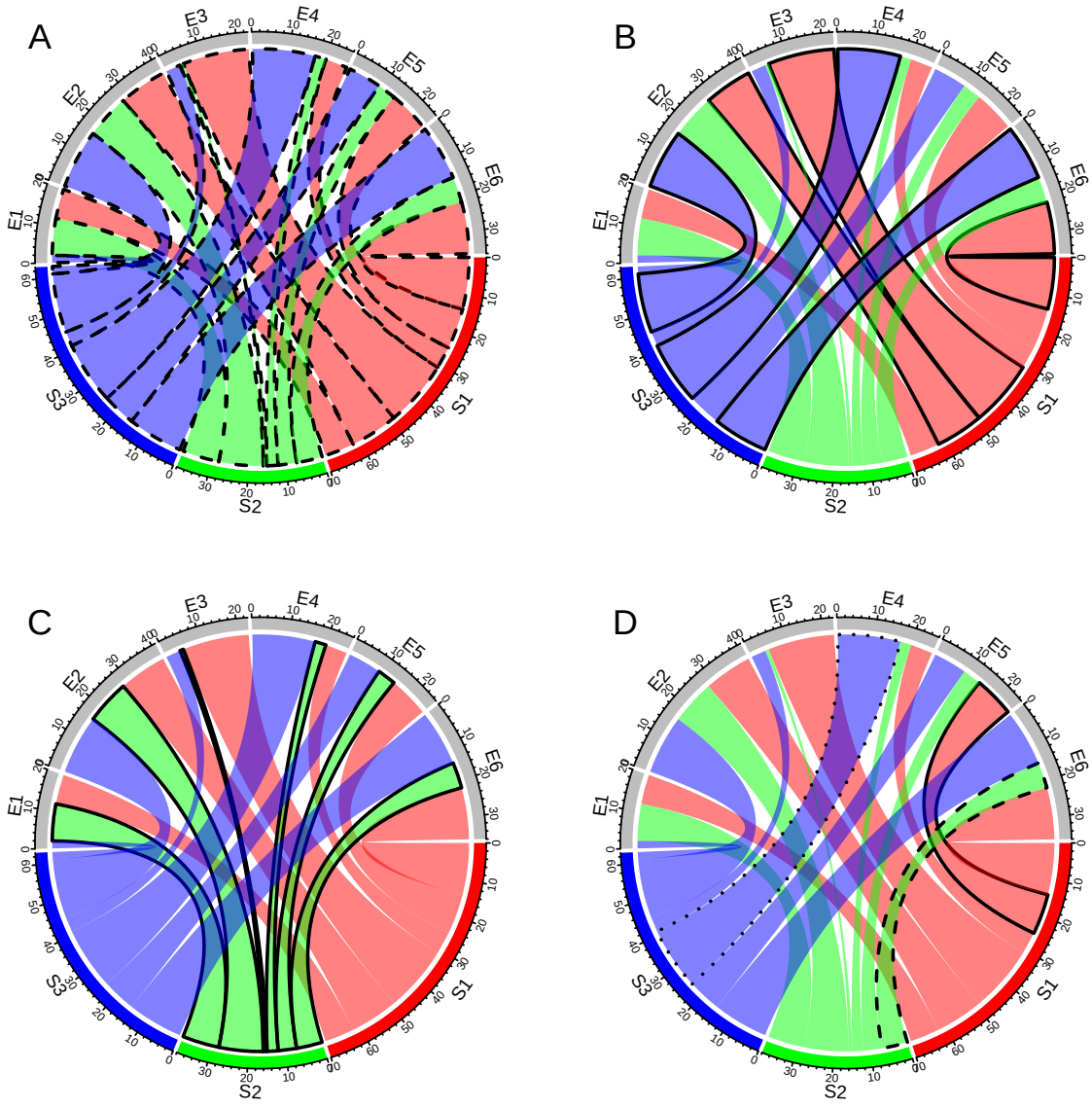

Figure 3: Link style settings in `chordDiagram`. A) graphic parameters set as scalar; B) graphic parameters set as matrix; C) graphic parameters set as sub matrix. D) graphic parameters set as a three-column data frame.

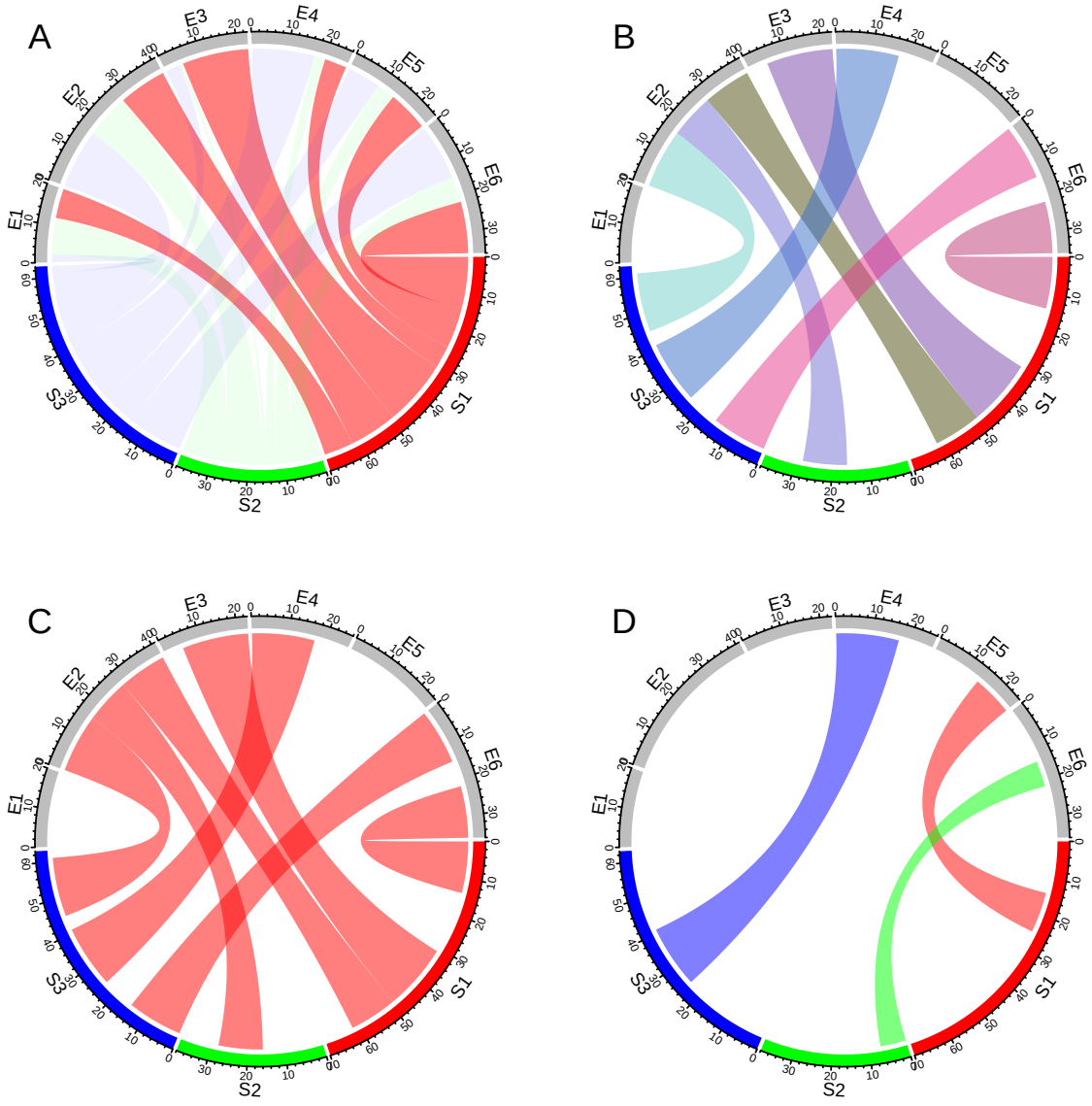

Figure 4: Highlight links by colors. A) set `row.col`; B) set by matrix; C) set by color function; D) set by a three-column data frame.

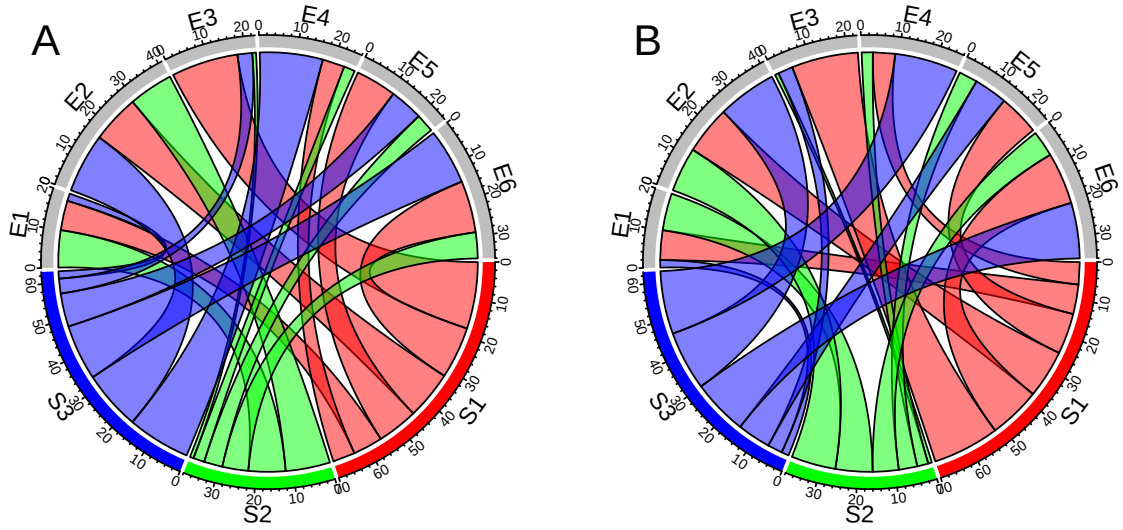

Figure 5: Orders of links. A) set `self.decreasing` to `TRUE`; B) set `self.decreasing` to `FALSE`.

1.6 Directional connections

In some cases, rows and columns represent information of direction (This is also natural for adjacency list). Argument `directional` is used to illustrate such direction. `directional` with value 1 means the direction is from rows to columns (from the first column to the second column for the adjacency list) while -1 means the direction is from columns to rows (from the second column to the first column for the adjacency list). By default, the roots of links will have unequal height (figure 6 A, B, C) to represent the directions, and the starting foot of the link is shorter than the ending foot to give people the feeling that the link is moving out. If this is not what you feel, you can set `diffHeight` to a negative value.

```
chordDiagram(mat, grid.col = grid.col, directional = 1)
chordDiagram(mat, grid.col = grid.col, directional = 1, diffHeight = 0.08)
chordDiagram(mat, grid.col = grid.col, directional = -1)
```

Row names and column names in `mat` can also overlap. In this case, showing directions of the link is quite important to distinguish them (figure 6 D).

```
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 86 21  5 67   1 94 27
## b 85 39 35 95  64 90  2
## c 28 84 61 11  48 20 74
## d 12 55 62 78 100 18 57
## e 37  3 19 65  70 72 97
```

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

If you don't need self-loop for which two roots of a link are in a same sector, just set corresponding values to 0 in `mat2` (figure 6 E).

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

```
##    a  b  c  d   e  f  g
## a  0 21  5 67   1 94 27
## b 85  0 35 95  64 90  2
## c 28 84  0 11  48 20 74
## d 12 55 62  0 100 18 57
## e 37  3 19 65   0 72 97
```

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

If the amount of links is not too large, links with arrows are more intuitionistic to represent directions (figure 6 F). In this case, `direction.type` can be set to `arrows`. Similar as other graphics parameters for links, the parameters for drawing arrows can either be a scalar, a matrix with dimension names, 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 effect (figure 6 G).

```
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 also another arrow type: `big.arrow` which is efficient to visualize arrows if the width of links are too small (figure 6 H)

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

If `diffHeight` is set to a negative value, the start roots are longer than the end roots (figure 6 I)

```
chordDiagram(matx, directional = 1, direction.type = c("diffHeight", "arrows"),
    link.arr.type = "big.arrow", diffHeight = -0.04)
```

Settings are similar for adjacency list, except by default, connection is from the first column in the data frame to the second one.

```
# code is not run when building the vignette
chordDiagram(df, directional = 1)
```

1.7 Self-links

How to set self links dependends on specific scenarios. If it is a self link, the two roots will be on a same sector. If the link is plotted in the normal way, the value for this link would be counted twice

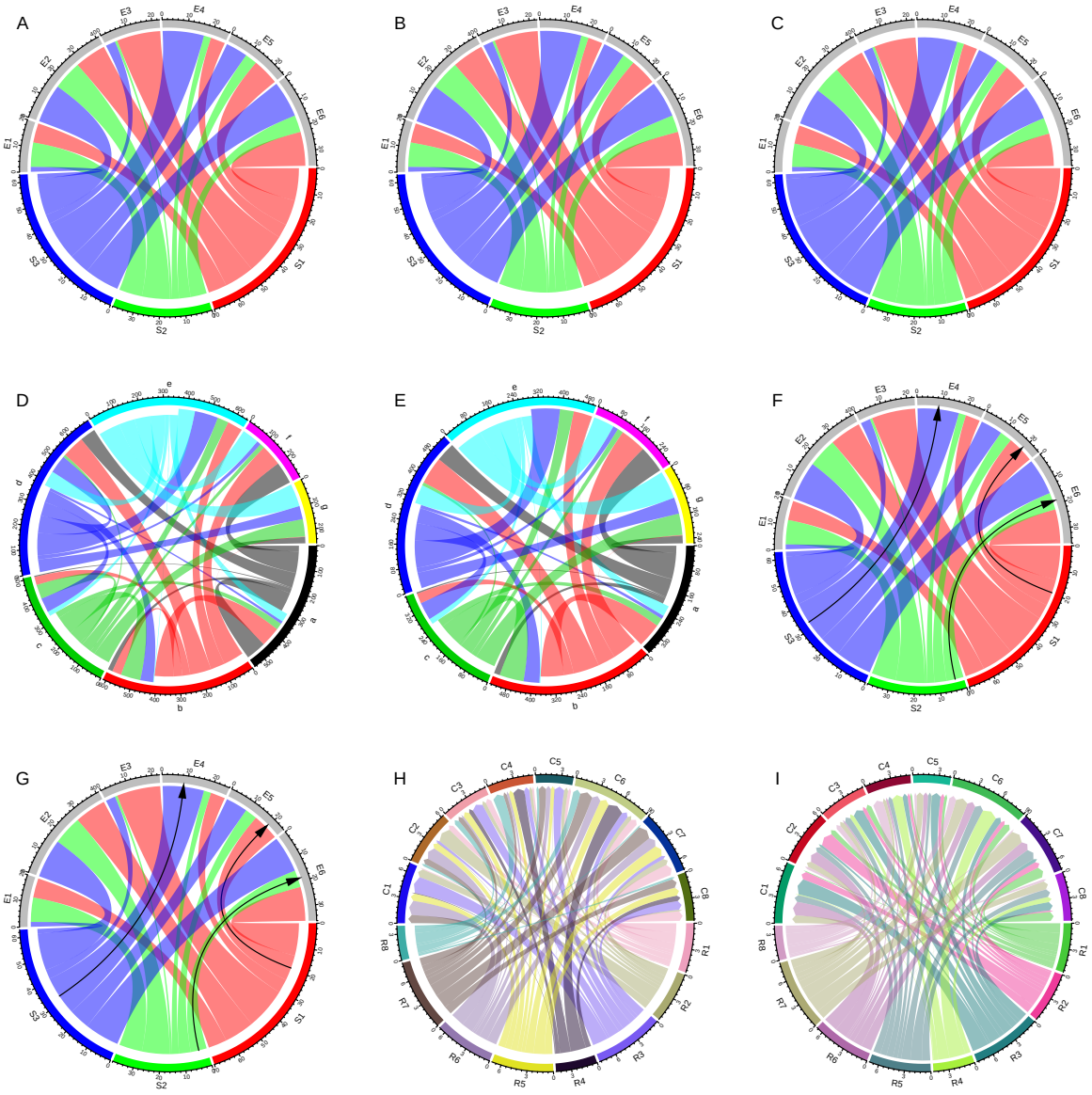

Figure 6: Visualization of directional matrix. A) with default settings; B) set difference of two feet of links; C) set the starting feet; D, E) row names and column names have overlaps; F, G, H, I) directions are represented by arrows.

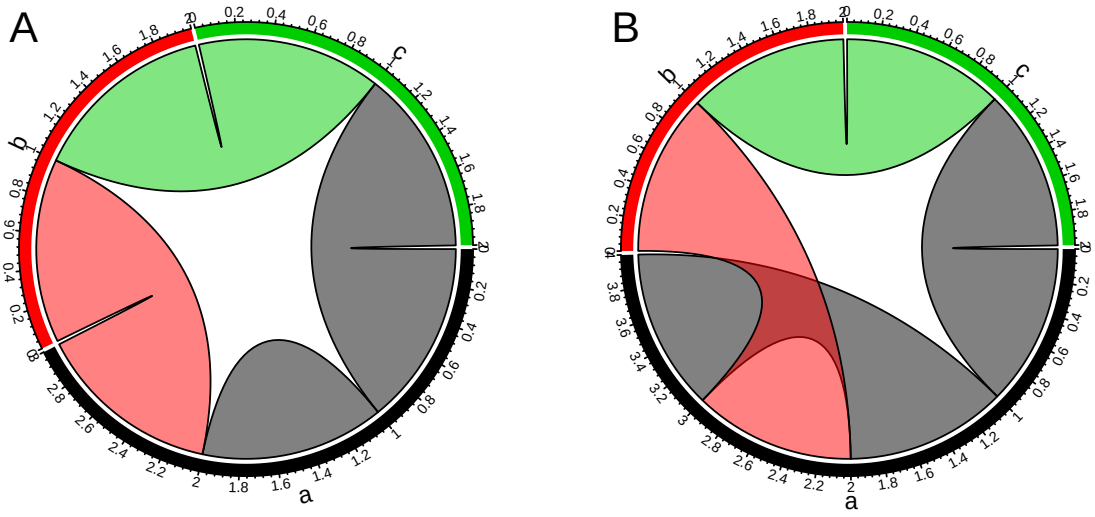

Figure 7: Deal with self links. A) set `self.link` to 1; B) set `self.link` to 2.

and sometimes it causes problems. If users think the self-link represent as a unchanged state, the link should be degenerated as a 'mountain' with the width corresponding to singular value. While in another scenario, suppose the connection represents the migration between countries, then the link should be plotted in the normal way because the two roots contain information of direction and are two different roots.

The `self.link` argument can be set to 1 or 2 for these two different scenarios. You may see the difference in figure 7.

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

1.8 Symmetric matrix

`chordDiagram` can also be used to visualize symmetric matrix. If `symmetric` is set to `TRUE`, only lower triangular matrix without the diagonal will be used. Of course, your matrix should be symmetric. In figure 8, you can see the difference with specifying `symmetric` or not when visualizing a symmetric matrix.

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

2 Advanced usage

Although the default style of `chordDiagram` is 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.

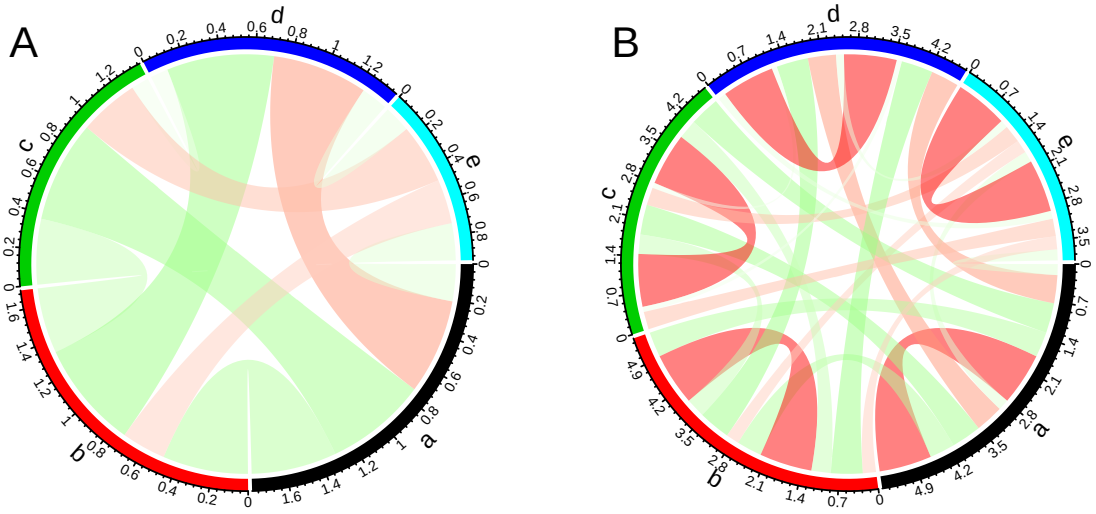

Figure 8: Visualization of symmetric matrix. A) set `symmetric` to `TRUE`; B) set `symmetric` to `FALSE`.

2.1 Organization of tracks

By default, `chordDiagram` which utilizes circos graphics functions will create two tracks, one track for labels and one track for grids (or plus 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`. Available values for this argument are `grid` and `name`. The height of annotation tracks can be set through `annotationTrackHeight` which corresponds to values in `annotationTrack` (figure 9 A, B, C). The value in `annotationTrackHeight` is the percentage to the radius of unit circle. If `grid` is specified in `annotationTrack`, there can also be an additional option `axis` to set whether add axes on this track.

```
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:
```

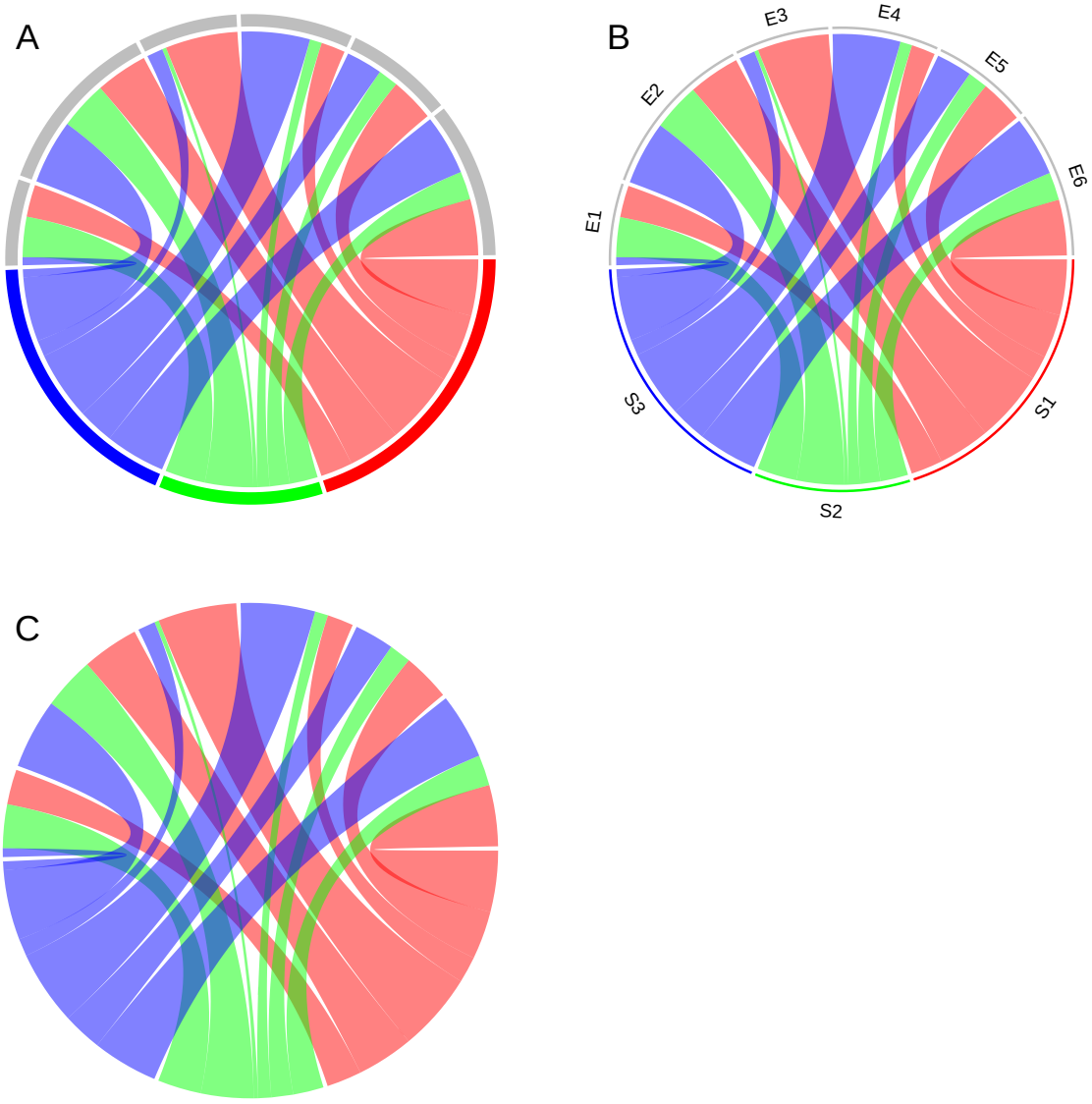

Figure 9: Track organization in `chordDiagram`. A) only show the grid track; B) set label track and grid track with heights; C) do not add label track or grid track.

```
## [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))
circos.info(sector.index = "S1", track.index = 1)
```

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 support 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.

2.2 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. Later, the first track is updated with setting `facing` of labels (figure 10 A).

```
chordDiagram(mat, grid.col = grid.col, annotationTrack = "grid",
    preAllocateTracks = list(track.height = 0.3))
# we go back to the first track and customize sector labels
circos.trackPlotRegion(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")
    circos.text(mean(xlim), ylim[1], sector.name, 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 inside the grids (figure 10 B). Please note `get.cell.meta.data` and `circos.text` are used outside of `panel.fun`, so `track.index` and `sector.index` should be specified explicitly.

```
chordDiagram(mat, grid.col = grid.col,
    annotationTrack = "grid", annotationTrackHeight = 0.15)
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", col = "white")
}
```

For the last example, we add the sector labels with different style. If the width of the sector is less than 20 degree, the labels are added in the radical direction.

```
mat2 = matrix(rnorm(100), 10)
chordDiagram(mat2, annotationTrack = "grid", preAllocateTracks = list(track.height = 0.1))
circos.trackPlotRegion(track.index = 1, panel.fun = function(x, y) {
    xlim = get.cell.meta.data("xlim")
    xplot = get.cell.meta.data("xplot")
    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))
    } else {
        circos.text(mean(xlim), ylim[1], sector.name, facing = "inside",
            niceFacing = TRUE, adj = c(0.5, 0))
    }
}, bg.border = NA)
```

One last thing, 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` to wider intervals.

2.3 Customize sector axes

Axes are helpful to visualize the absolute values of links. By default `chordDiagram` add axes on the grid track. But it is easy to customize a better one with self-defined code.

Another type of axes which show relative percent is also helpful for visualizing Chord diagram. Here we pre-allocate an empty track by `preAllocateTracks` and come back to this track to add axes later. In following example, a major tick is put every 25% in each sector. And the axes are only added if the sector width is larger than 20 degree (figure 11).

```
# similar as the previous example, but we only plot the grid track
chordDiagram(mat, grid.col = grid.col, annotationTrack = "grid",
    preAllocateTracks = list(track.height = 0.1))
for(si in get.all.sector.index()) {
    circos.axis(h = "top", labels.cex = 0.3, major.tick.percentage = 0.2,
        sector.index = si, track.index = 2)
}

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

    if(abs(xplot[2] - xplot[1]) > 20) {
```

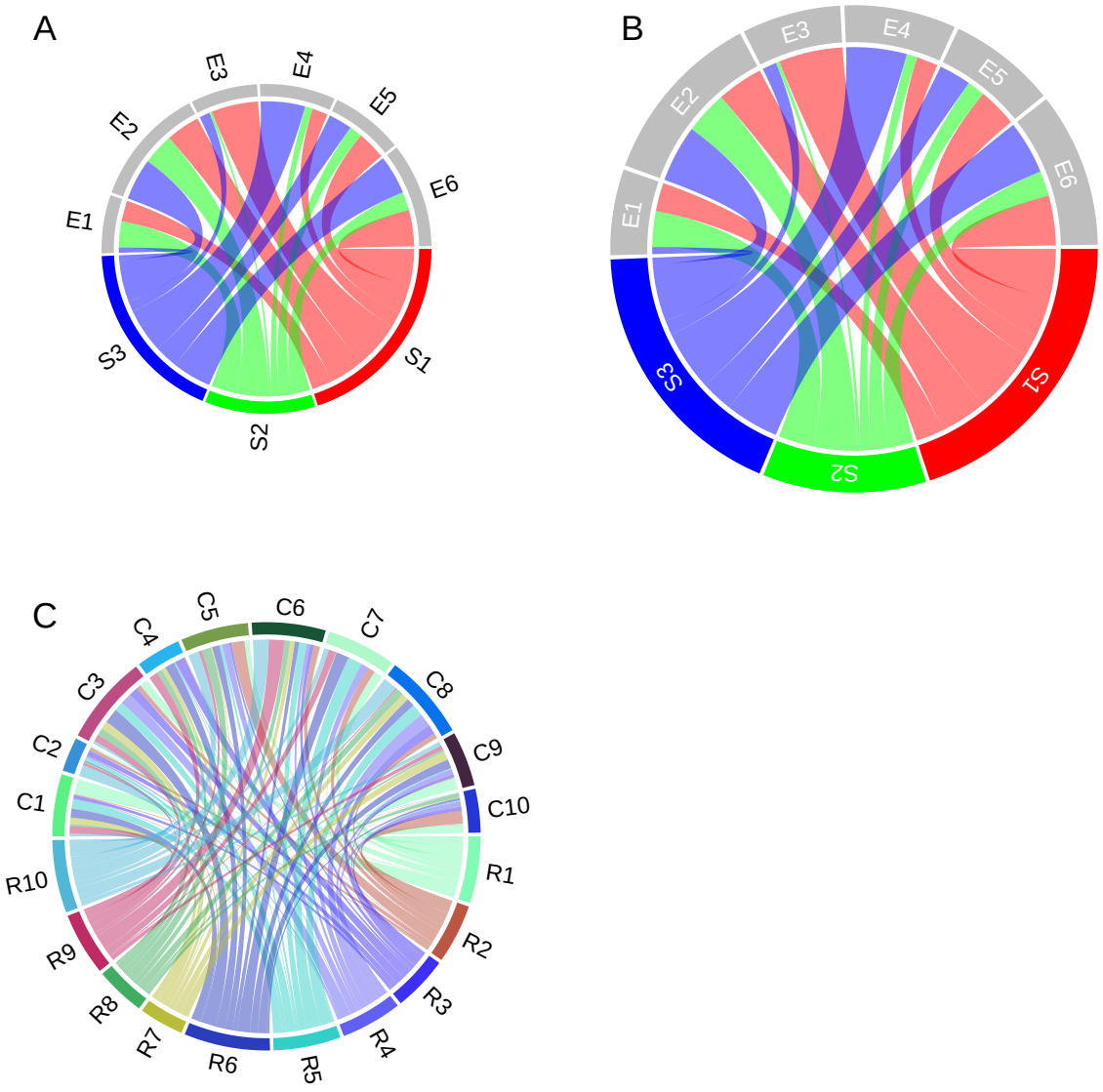

Figure 10: Customize sector labels. A) put sector labels in radical direction; B) sector labels are put inside grids; C) sector labels are put in different direction according the width of sectors.

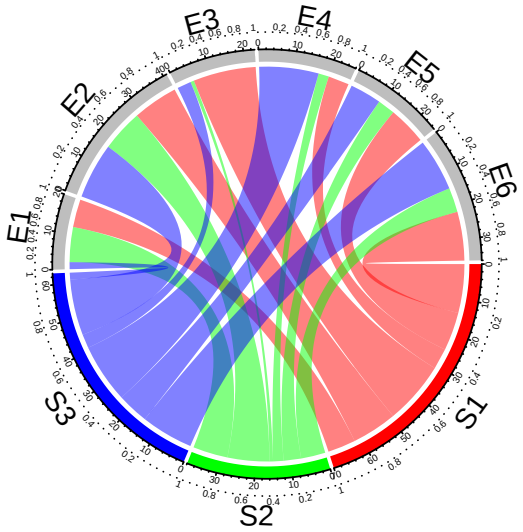

Figure 11: Customize sector axes

```
        circos.lines(xlim, c(mean(ylim), mean(ylim)), lty = 3) # dotted line
        for(p in seq(0.2, 1, by = 0.2)) {
            circos.text(p*(xlim[2] - xlim[1]) + xlim[1], mean(ylim) + 0.1,
                p, 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()
```

2.4 Compare two Chord diagrams

Normally, in Chord diagram, values in `mat` are normalized to the summation (of the absolute values) and each value is put to the circle according to its percentage, which means the width for each link only represents kind of relative value. However, when comparing two Chord 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 values.

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.degree = c(rep(2, 4), 10, rep(2, 4), 10)
circos.clear()
circos.par(gap.degree = gap.degree, 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 which makes the comparison impossible. What we want to compare between two

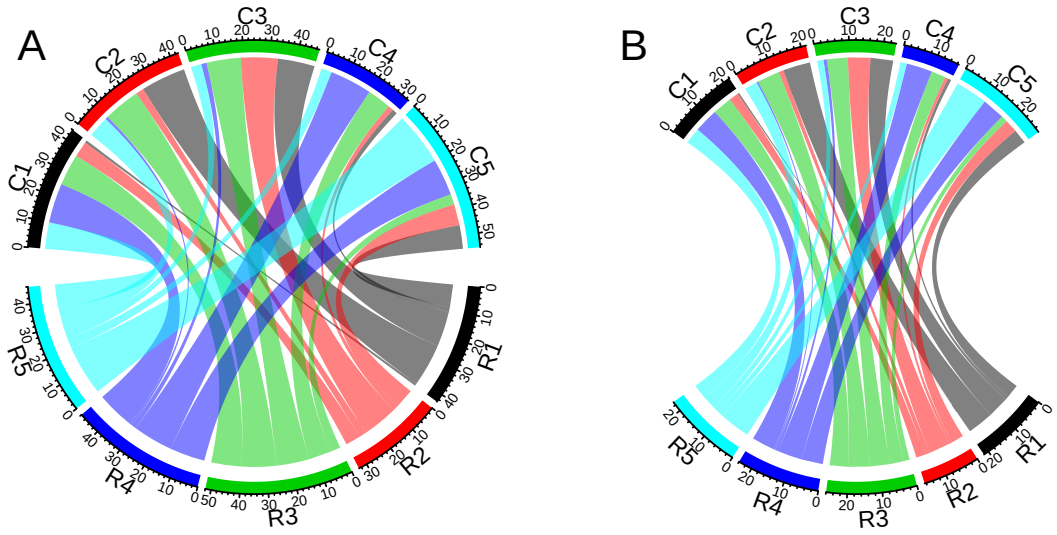

Figure 12: Compare two Chord Diagrams and make them in same scale. bottom matrix has half the values as in the upper matrix.

diagrams is the absolute values. For example, if the matrix contains the amount of transitions from one state to another state, then the interest is to see which diagram has more transitions.

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

Now the scale of the two Chord diagrams (figure 12) are the same if you look at the scale of axes in the two diagrams.

3 Misc

If a sector in Chord Diagram is too small, it will be removed from the original matrix. In the following matrix, the second row and third column contain tiny numbers.

```
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 (figure 13 A).

```
chordDiagram(mat)
```

If you set `row.col`, `column.col` or `col` to specify the colors of the links, colors corresponding to the second row and the third column will also be removed (figure 13 B).

```
chordDiagram(mat, row.col = rep(c("red", "blue"), 3))
```

`grid.col` is reduced if it is set as a vector which has the same length as categories which are from the unreduced matrix (figure 13 C).

```
chordDiagram(mat, grid.col = rep(c("red", "blue"), 6))
circos.clear()
```

`circos.par("gap.degree")` will be reduced as well (figure 13 D).

```
circos.par("gap.degree" = rep(c(2, 10), 6))
chordDiagram(mat)
circos.clear()
```

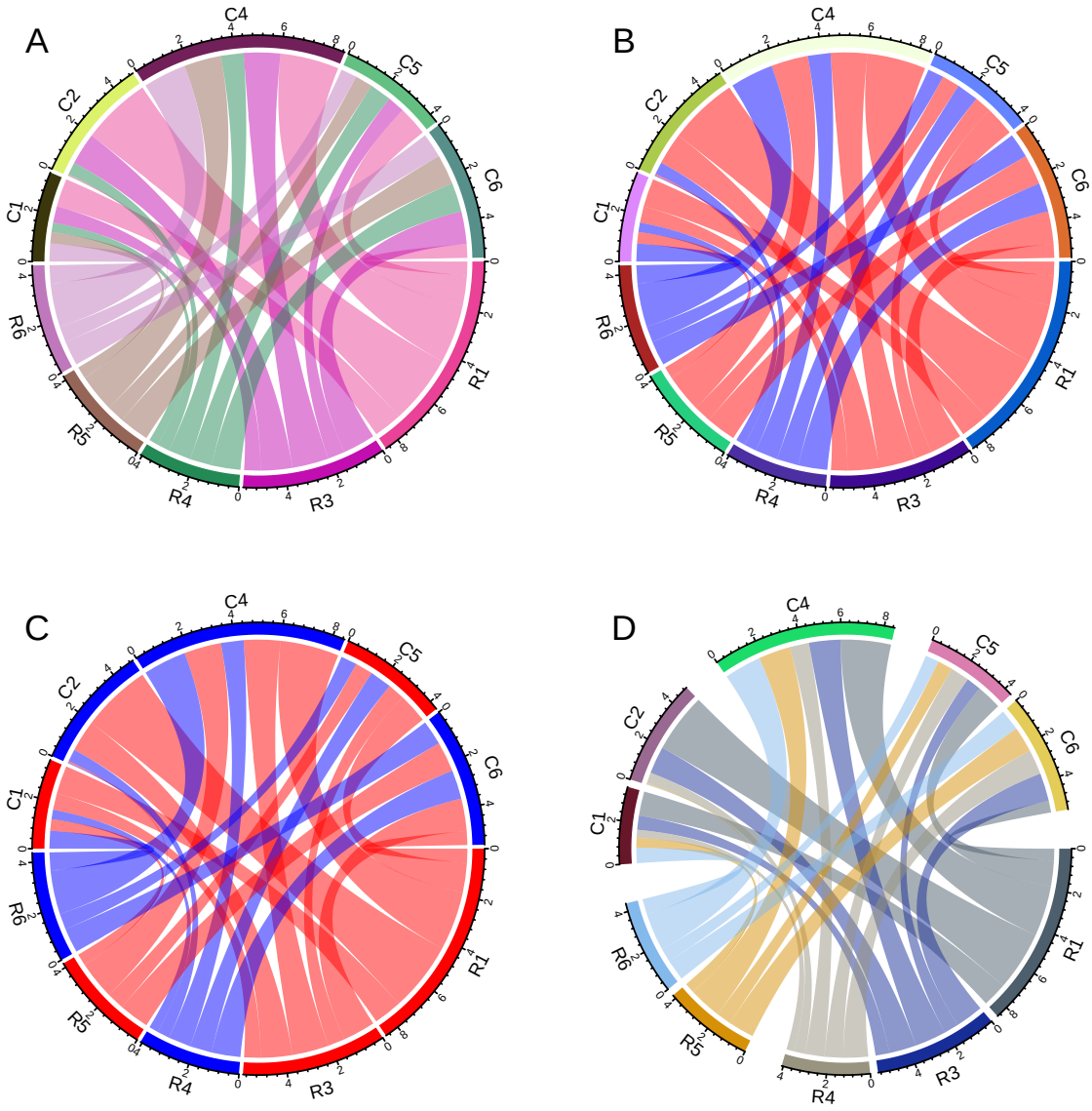

Figure 13: Reduced Chord Diagram with removing tiny sectors. A) notice how sector labels are reduced; B) notice how link colors are reduced; C) notice how grid colors are reduced; D) notice how gap degrees are reduced.