How to Use Grid Package to Draw a Forest Plot

PharmaSUG China 2023 – Paper DV-153

Han Cheng, BeiGene, Inc., Beijing

Abstract

Visualization is always a significant method to monitor the clinical trial process and review data. For that purpose, many kinds of figures are widely used by pharmaceutical companies, like box plot, swimlane plot, waterfall plot, forest plot and so on. Instead of SAS, most of them could be plotted conveniently and fancily by ggplot, which is almost the most strong and popular plotting package in R Studio. However, for forest plot, it seems that ggplot doesn't work out well since it has a table-like structure. This paper demonstrates how to use another package grid, which has been a base package recently, to generate a basic forest plot, and shows the advantage of using grid package when plotting.

Introduction

Functions in the ggplot2 package allow extensive customization of many plotting elements. For example, users can use ggplot2 functions to change the theme of a plot, to customize the colors used within the plot, and to create multiple faceted graphs.

Comparing to ggplot2, Grid graphics is a low-level system for plotting within R and is as a separate system from base R graphics. "low-level" means that grid graphics functions are typically used to modify very specific elements of a plot, rather than being functions to use for a one-line plotting call. For example, if users want to quickly draw a scatter plot, ggplot2 would always be the first choice. However, if users want to create a plot with an inset plot with tilted axis labels, or just create a table-like plot, then grid package is supposed to be selected with the higher priority.

Even though it does take more time and code to create plots using grid graphics comparing to plot with ggplot2, it is usually worth using grid graphics when producing a very unusual plot that cannot be created by ggplot2 directly. Now the grid package is a base package, which means it is installed automatically when installing R.

Data preparation

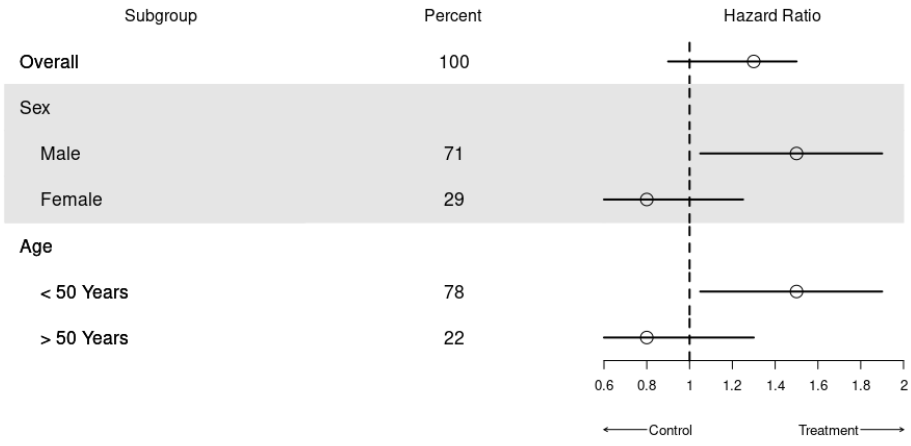


Figure 1: Figure 1: Forest plot

To produce a figure in *Figure 1*, the first step is to dummy the input data frame at first, which has the general structure used in our daily work, and the header would like to be shown in the output.

```
df <- data.frame(subgroup =
                   c("Overall", "Sex", "Male", "Female", "Age", "< 50 Years", "> 50 Years"),
                 percent = c("100", "", "71", "29", "", "78", "22"),
                 low = c(0.9, NA, 1.05, 0.6, NA, 1.05, 0.6),
                 high = c(1.5, NA, 1.9, 1.25, NA, 1.9, 1.3),
                 mean = c(1.3, NA, 1.5, 0.8, NA, 1.5, 0.8),
                 id = c(1, 1, 2, 2, 1, 2, 2),
                 ref = c(NA, 1, 1, 1, NA, NA, NA))

block_headers = c("Subgroup", "Percent", "Hazard Ratio")
```

Then several additional steps are needed to preprocess the data, like adding spaces in front of the sub-category, defining the column position of the point-range plot to be shown, and deriving some key variables for building coordinates. At the following, point range plot is defined on the third column.

```
dt <- df %>%
  dplyr::mutate(label = ifelse(id == 1, as.character(subgroup), paste0("    ", subgroup)))

ci_column <- 3
ref_line <- 1
xrange<-range(c(min(dt$low, na.rm = T), max(dt$high, na.rm = T)))
xlim <- pretty(xrange)
```

Main part of plotting

Firstly, **grid.newpage** creates a new page in case of any other interference factors. In general, plotting by grid will start from generating a viewport. In this case a viewport is established as the parent viewport, named "top", and pushed forward. In R, a viewport is a drawing context that can be used to create a plot within a plot, and by using **pushViewport**, we could push the viewport we just defined to the current position.

In this step we can define the width and height of the diagram either. Moreover, it is given additional rows to **n_row** because generally, it is needed to have extra spaces for other objects to create a more informative figure, like headers on the top and arrows at the bottom in *Figure 1*.

```
n_row <- nrow(dt)+3
n_col <- 3

grid.newpage()
pushViewport(viewport(width = unit(9, "inches"),
                      height = unit(4.6, "inches"),
                      xscale = c(0, n_col),
                      yscale = c(0, n_row),
                      name = "top"))
```

For the first column, which shows the category of each section, users can re-generate a new viewport in a small single cell and push it to the current position. As a supplement, two of the most commonly used units in the grid are **native** (Locations and dimensions are relative to the viewport's x scale and y scale) and **npc** (Normalized Parent Coordinates. Default unit, the origin of the viewport is (0, 0) and the viewport has a width and height of 1 unit). Here we choose **native** as our unit because it is more convenient for us to control the whole viewport tree here if the x scale and y scale have been defined in the parent viewport in advance.

One of the most critical concept of grid graphics is the concept of grobs. Grobs are graphical objects that you can make and change with grid graphics functions. They are like elements of the graphics and once you have created one or more of these grobs, you can add them to or take them away from larger grid graphics objects. Once you have created a grob object, you can use the **grid.draw** function to plot it to the current position.

Here we use **grid.rect**, which is equivalent to **grid.draw(rectGrob(..))** to draw a rectangular filled with gray color, as a gray background to distinguish different categories. Then we use **grid.text** to show the contents of variable "label". After that, **upViewport** is used to navigate to the parent of the current **vp** viewport, without removing **vp** from the viewport tree. The final step is to have all of these in a loop, so the texts could be drawn row by row.

```
for(i in 1:nrow(dt)){
    vp<-viewport(x = unit(0,"native"),
                 y = unit(nrow(dt)+2.5-i, "native"),
                 width = unit(1, "native"),
                 height = unit(1, "native"))
    pushViewport(vp)

    if(!is.na(dt$ref[i])){
      grid.rect(gp = gpar(col = "gray90", fill = "gray90"))
    }

    grid.text(as.character(dt[i, "label"]),
              0.05, 0.5,
              gp = gpar(cex=1.1),
              just = "left")

    upViewport()
}
```

For the second column "percent", it could be produced with almost the same code since it only includes text information either.

```
for(i in 1:nrow(dt)){
    vp<-viewport(x = unit(1,"native"),
                 y = unit(nrow(dt)+2.5-i,"native"),
                 width = unit(1, "native"),
                 height = unit(1, "native"))
    pushViewport(vp)

    if(!is.na(dt$ref[i])){
      grid.rect(gp = gpar(col = "gray90", fill = "gray90"))
    }

    grid.text(as.character(dt[i, "percent"]),
              0.5, 0.5,
              gp = gpar(cex=1.1),
              just = "center")

    upViewport()
}
```

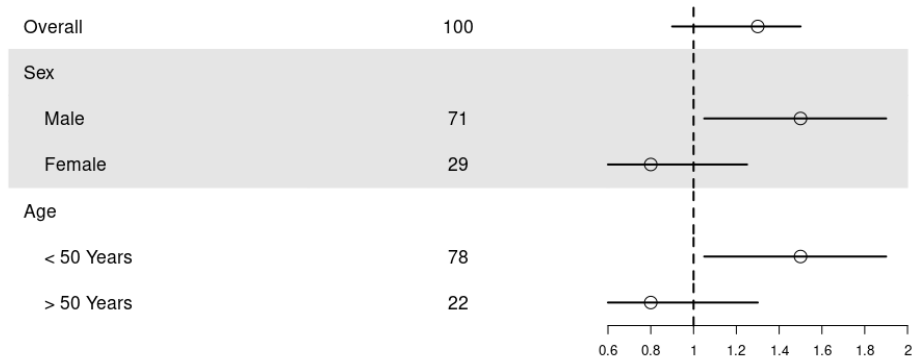
Overall	100
Sex	
Male	71
Female	29
Age	
< 50 Years	78
> 50 Years	22

As for the point-range plot part, some updates are needed. The first one is, considering to reflect the true value of mean, upper limit and lower limit, it is needed to add x scale and y scale options when generating separated children viewport to keep them consistent to the parent viewport. Besides that, **grid.lines** and **grid.points** are used for plotting confidence intervals and the points of mean value. When using the Grob functions, users can adjust the format as they want by **gp** option. Aesthetics that can be set by specifying a **gpar** object for the **gp** parameter of a grob include color (**col**), fill (**fill**), transparency (**alpha**), line type (**lty**), line width (**lwd**), font elements (**fontsize**, **fontface**, **fontfamily**) and so on.

Also, X axis can be drawn directly at last round of loop by applying **grid.xaxis**.

```
for(i in 1:nrow(dt)){
  vp<-viewport(x = unit(ci_column-1,"native"),
               y = unit(nrow(dt)+2.5-i,"native"),
               width = unit(1, "native"),
               height = unit(1, "native"),
               xscale = c(min(xlim), max(xlim)),
               yscale = c(0,1))
  pushViewport(vp)

  if(!is.na(dt$ref[i])){
    grid.rect(gp = gpar(col = "gray90", fill = "gray90"))
  }

  grid.points(x = unit(dt$mean[i], "native"),
              y = 0.5,
              gp = gpar(pch = 5))

  grid.lines(x = unit(c(dt$low[i], dt$high[i]), "native"),
             y = 0.5,
             gp = gpar(lwd = 2))

  grid.lines(x = unit(ref_line, "native"),
             y = c(0, 1),
             gp = gpar(lwd = 2, lty = "dashed"))

  if(i == max(nrow(dt))){
    grid.xaxis(at = xlim,
               gp = gpar(fontsize = 10))
```

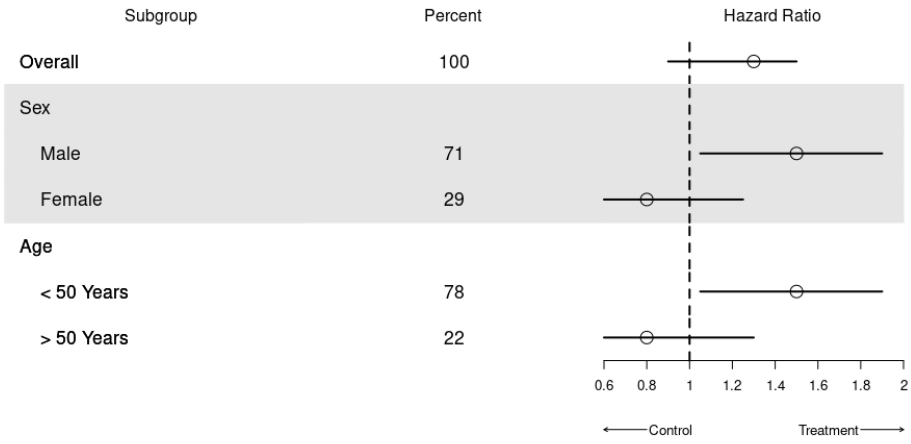
```
  }

  upViewport()
}
```



From now, the main body of the forest plot has been finished, and generally, users still need some more details on it. We can use different Grob functions to add the header that was defined at the beginning on the top of the graph, and lines with arrow at the bottom. Just like the following.

```
for(i in 1:n_col){
  vp <- viewport(x = unit(i-1, "native"),
                 y = unit(n_row-0.5, "native"),
                 width = unit(1, "native"),
                 height = unit(1, "native"))
  pushViewport(vp)

  grid.text(block_headers[i], 0.4, 0.5, just = c("left"))

  upViewport()
}

vp <- viewport(x = unit(ci_column-1, "native"),
               y = unit(0.5, "native"),
               width = unit(1, "native"),
               height = unit(1, "native"),
               xscale = c(min(xlim), max(xlim)),
               yscale = c(0,1))
pushViewport(vp)

grid.lines(x = unit(c(min(xlim), ref_line-0.2), "native"),
           y = 0.5,
           arrow = arrow(angle = 30,
                         length = unit(0.2, "lines"),
                         ends = "first"))

grid.lines(x = unit(c(max(xlim)-0.2, max(xlim)), "native"),
           y = 0.5,
```

```
           arrow = arrow(angle = 30,
                         length = unit(0.2, "lines"),
                         ends = "last"))

grid.text("Control",
          0.15, 0.5,
          just = c("left"),
          gp = gpar(fontsize = 10))

grid.text("Treatment",
          0.85, 0.5,
          just = c("right"),
          gp = gpar(fontsize = 10))

upViewport()
```



Conclusion

In all, the forest plot can be produced with grid package in a style that is similar to the figure generated by SAS. In most cases, I would recommend ggplot2 if producing graphical output directly. However, the grid package provides a set of functions and classes that represent graphical objects or grobs, that can be manipulated like any other R objects, which means grid package provides you the greatest flexibility when drawing all kinds of diagrams.

Grid graphics provide an extensive graphics system that can allow you to create almost any plot you can imagine in R. In this paper, we have only scraped the surface of the grid graphics system. It is very helpful for you to study grid graphics in greater depth, especially when facing a situation that need to create very tailored, unusual graphs.

References

- Mastering Software Development in R. *Roger D. Peng, Sean Kross, and Brooke Anderson*
- R Graphics. *Paul Murrell*

Contact Information

Your comments and questions are valued and encouraged. Contact the author at:

- Name: Han Cheng
- Email: han.cheng@beigene.com