Emulate `persp` plot and `filled.contour` plot on `gridGraphics`

April 4, 2017

0.1 Introduction

0.1.1 Background

The core graphics system in R can been divided in to two main packages. The first package is the graphics package. It is older and it provides the original GRZ graphics system from S, sometimes referred to as "traditional" graphics. It is relatively fast and many other R packages build on top of it. The newer package is the grid package. It is actually slower but is has more flexibility and additional features compared to the graphics package.
A graph that is drawn using grid can been edited in many more ways than a graph that has been drawn using the basic graphics package. However, there is a new package, called `gridGraphics`, which allows us to convert a plot that has been drawn by the `graphics` package to an equivalent plot drawn by `grid` graphics. This means that the additional flexibility and features of grid become available for any plot drawn using the `graphics` package.

0.1.2 The `gridGraphics` package

`gridGraphics` is like a 'translator' that translates a plot that has been drawn using the basic graphics package to a plot that has been drawn using the grid package.
The `gridGraphic` package has a main function called `grid.echo()`, which takes a recorded plot as an argument (or NULL for the current plot of the current graphics device). The grid.echo() replicates the plot using grid so that the user may edited the plot in more ways than they can with the original plot drawn by basic graphic package.
The following code provides a quick example. We generate 25 random numbers for x and y. First, we draw a scatter plot using the function `plot()` from the basic graphics package, then we redraw it using `grid.echo()` from the `gridGraphics` package with grid.

```
> pdf("figure/report_basic_demo_%0d.pdf", onefile=FALSE)
> dev.control("enable")
> set.seed(110)
> x = runif(25)
> y = runif(25)
> plot(x,y, pch = 16)
> grid.echo()
```

One example that shows the advantage of drawing the plot using grid rather than basic graphics is that there are objects, called grid grobs, which recorded a list of the details of each components of the plot that has been drawn. The list of grobs can been seen by calling the function `grid.ls()`.

```
> grid.ls()

graphics-plot-1-points-1
graphics-plot-1-bottom-axis-line-1
graphics-plot-1-bottom-axis-ticks-1
```

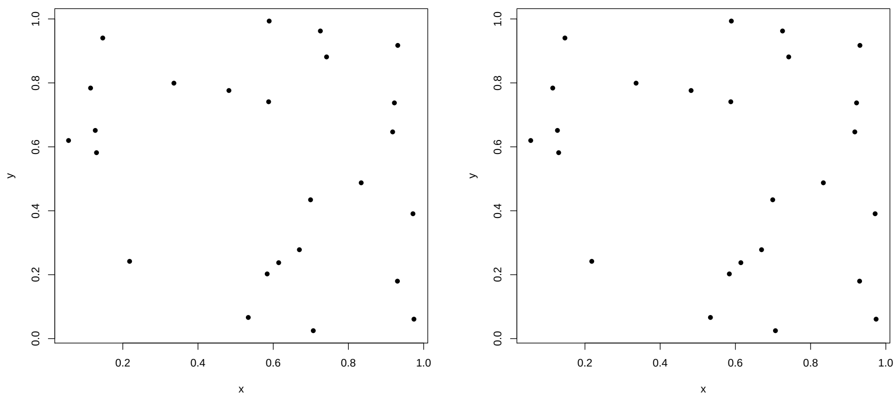

Figure 1: The left plot is drawn by using plot(); the Right plot is redrawn using grid.echo(). Overall, two plots are identical to each other

```
graphics-plot-1-bottom-axis-labels-1
graphics-plot-1-left-axis-line-1
graphics-plot-1-left-axis-ticks-1
graphics-plot-1-left-axis-labels-1
graphics-plot-1-box-1
graphics-plot-1-xlab-1
graphics-plot-1-ylab-1
```

As we see, the `grid.ls()` function returns a list of grid grobs for the previous plot that has been redrawn by `grid`. There is one element called "graphics-plot-1-bottom-axis-labels-1" which represents the labels of the bottom axis. In `grid`, there are several functions that can be used to manipulate this grob. For example, if the user wants to rotate the labels of the bottom axis by 30 degrees and changes the color from default to orange, then the following code performs these changes.

```
> grid.edit("graphics-plot-1-bottom-axis-labels-1",
+           rot=30, gp=gpar(col="orange"))
> grid.edit("graphics-plot-1-left-axis-labels-1",
+           rot=30, gp=gpar(col="orange"))
```

0.1.3 The problem

The `grid.echo()` function can replicate most plots that are drawn by the graphics package. However, there are a few functions in the graphics package that `grid.echo()` cannot replicate. One such function is persp() which draws 3-dimemtional surfaces, the other one is the `filled.contour()`. If we can draw a plot with `persp()` or `filled.countour()`, the result from calling `grid.echo()` is a (mostly) blank screen.

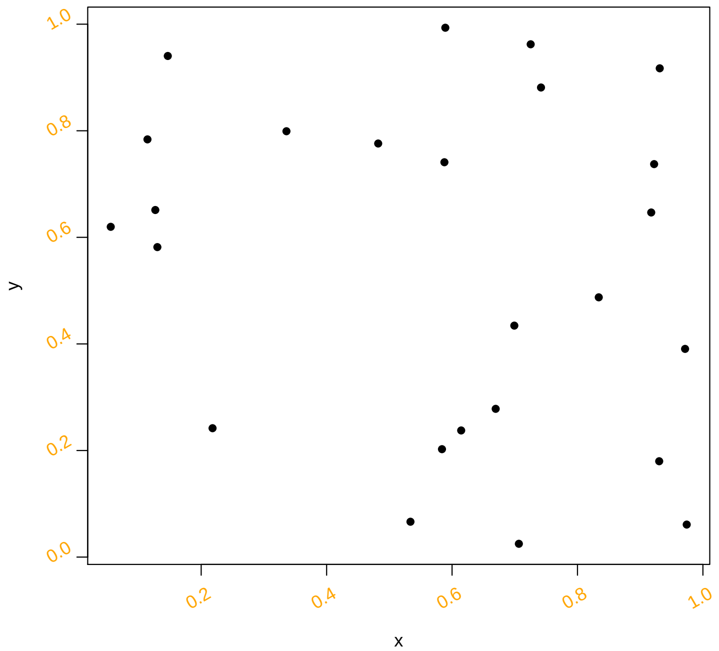

Figure 2: The angle and the color of the bottom and left axis of the previous plot have been changed by 30 degrees and orange

```
> x <- y <- seq(-4*pi, 4*pi, len = 27)
> r <- sqrt(outer(x^2, y^2, "+"))
> filled.contour(cos(r^2)*exp(-r/(2*pi)), frame.plot = FALSE, plot.axes = {})
> grid.echo()
```

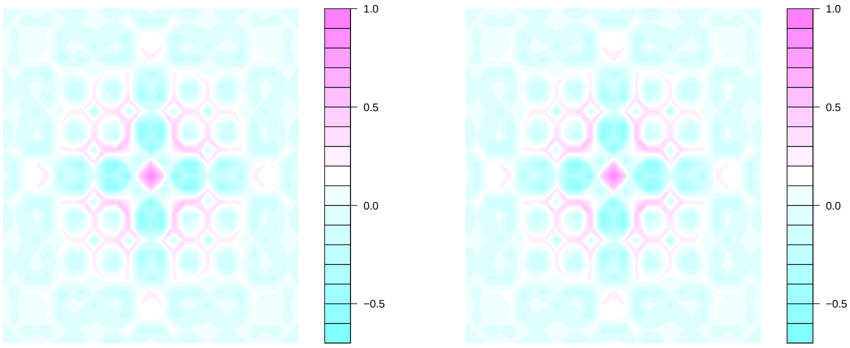

Figure 3: The left plot been drawn by using `filled.contour` and the right plot been redrawn by calling `grid.echo()`. There is a "blank" page on the right plot because the grid.echo cannot emulate filled.contour()

0.1.4 Aim of this project

The purpose of this paper is emulate the Perspective Plots, `persp()` and Level (Contour) Plots, `filled.contour()` into `grid` package. However, these two functions are written by C, which is difficult for debugging and tracking. The solution of this paper as follows:

1. emulate the `persp()` function on `grid` separate from the `gridGraphics` package (standalone):
   (a) Extract the information from the graphics engine display list.
   (b) Understanding and translating the calculation that been done by C code from the `graphics` package to R code
   (c) Draw the Perspective Plot on `grid`.
2. Connect the standalone to the `gridGraphics`

0.2 The graphics engine display list

The information for every plot drawn by R can be recorded. For example, In the simple `plot()` function, it is possible to obtain the parameters for x and y, even the label of the x-axis and y-axis.
This information is called the graphics engine display list. In this paper, we use this graphics engine display list to replicate the `persp()` plot and texttfilled.contour() plot using grid.
The `recordPlot()` function can be used to access the graphics engine display list, the `recordPlot()` function been used. This function saves the plot in an R object.

```
> plot(cars$speed[1:25], cars$dist[1:25], col = 'orange',
+       pch = 16, xlab = 'speed', ylab = 'dist')
> reco = recordPlot()
> reco[[1]][[4]][[2]][[2]]

$x
 [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14 15 15

$y
 [1]  2 10  4 22 16 10 18 26 34 17 28 14 20 24 28 26 34 34 46 26 36 60 80 20 26

$xlab
[1] "cars$speed[1:25]"

$ylab
[1] "cars$dist[1:25]"
```

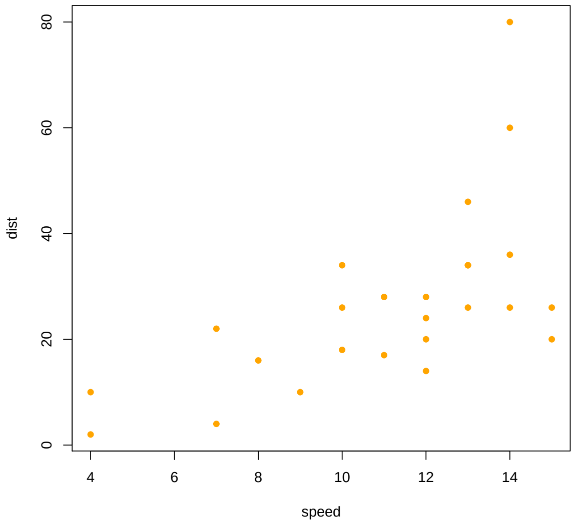

Figure 4: The details of the plot of dist vs speed displayed by the graphics engine display list

The example demonstrate how to access the graphics engine display list of a plot drawn by `plot`. The values of x and y, the labels of x-axis and y-axis been displayed.

0.3 standalone

The Perspective Plots `persp()` is used to draw a surface over the x-y plane. Usually, it has three main argument, **x, y, z**. Where **x** and **y** are the locations of grid line which the value z been measured, **z** is a matrix which containing the values that been used to plot, or it is the matrix that been calculated by a specific function, such as 3-D mathematical functions. The following example shows how to draw a obligatory mathematical surface rotated sinc function on Perspective Plot.

```
> x = y = seq(-10, 10, length= 60)
> f <- function(x, y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
> z <- outer(x, y, f)
> z[is.na(z)] <- 1
> trans = persp(x, y, z, theta=30, phi = 20, expand = 0.5,
+                col = 'White', border = 'orange')
```

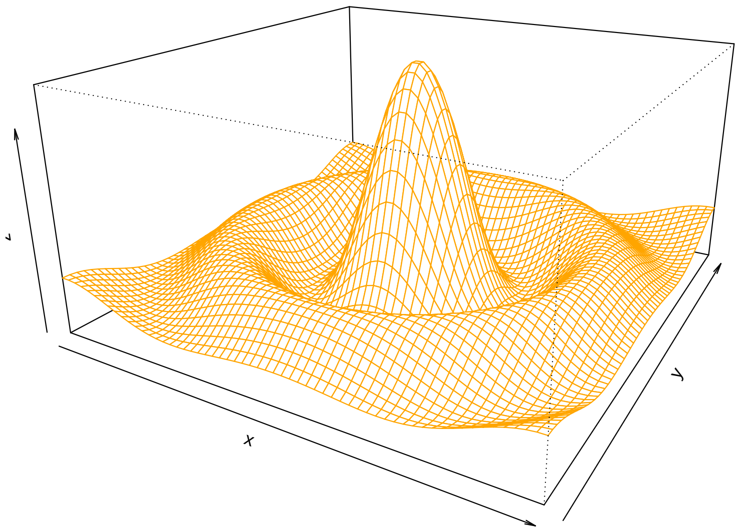

Figure 5: An example of Perspective Plot been drawn by `persp()`

From the pervious example, It is clearly to see that the Perspective Plots is formed by a finite number of "polygon", each polygons have 4 Vertices. If we can access the values for each Vertices of the polygon, then we can reproduce this polygon. If we can access all the values of Vertices of all polygons, then we can reproduce the Perspective Plot.

Inorder to emulate this plot, we need to access the some information from the graphics engine display list. However, the value of the vertices are not in the display list, therefore the plot can not been reproduce directly. But we can access value of **x**, **y** and **z**, therefore we have to re-do the calculation in order to get values of all vertices. The following codes show that the value of **x**, **y** and **z** which inputted by the user can been "catched" from the display list.

```
> reco = recordPlot()
> info = reco[[1]][[3]][[2]]
> ## print the values of x
> head(info[[2]])
```

```
[1] -10.000000  -9.661017  -9.322034  -8.983051  -8.644068  -8.305085
```

```
> ## print the values of y
> head(info[[3]])
```

```
[1] -10.000000  -9.661017  -9.322034  -8.983051  -8.644068  -8.305085
```

```
> ## print the values of z
> info[[4]][1:6, 1:2]
```

```
          [,1]      [,2]
[1,] 0.7070981 0.6998135
[2,] 0.6998135 0.6510811
[3,] 0.6534671 0.5639162
[4,] 0.5714305 0.4439461
[5,] 0.4589249 0.2984302
[6,] 0.3225432 0.1356653
```

0.3.1 The translation from 3-D points into 2-D points

The values of **x**, **y** and **z** can been recored from the display list, which been explained by the pervious section, the next task is to use this information to reproduce the vertics in 3-D.

As we know, the matrix, **z** is computed by a specific functions, given two inputs, **x** and **y**, or the expression of z can been written as: $z = f(x, y)$, it contains all the values for all combination of x and **y** and the dimenstion of **z** is $\mathbf{dim(x)} \times \mathbf{dim(y)}$.

One 3-dimenstions points contains a set values of (x, y, z), but **z** is $\mathbf{dim(x)} \times \mathbf{dim(y)}$ matrix, **x** is a vector which has length of **length(x)** and **y** is a vector which has length of **length(y)**. Inorder to produce the points, the D of **x**, **y** and **z** need to be matched and also in a right order.

First step is the reduce the **z** $\mathbf{dim(x)} \times \mathbf{dim(y)}$ matrix into a one D vector which has length of $\mathbf{dim(x)} \times \mathbf{dim(y)}$. It can been reduced by either along x direction or y direction. In this paper, we reduced along the x direction. The second step is repeat the vector x and y until the same length of **z**. Since **z** is reduced along the x direction say z_p, hence we repeat x until the length of y say x_p, and we repeat each y by the length of **x**, say y_p. At last, the combination of $_p$, y_p, z_p is the 3-D points which prepare for computing the vertices.

```
> xTmp = rep(x, length(y))
> yTmp = rep(y,each = length(x))
> zTmp = as.numeric(z)
> length(xTmp) == length(zTmp) & length(yTmp) == length(zTmp)

[1] TRUE
```

The idea of transform the points into vertices is repeating the points in a right order. From pervious section, we explained that the Perspective Plots is made by finite number of polygons. Each polygon have 4 vertices. The total number of polygons are required to be drawn is depend on the length of input **x** and the length of input **y**, that is, **total = (length(x) - 1)** $\times$ **(length(y) - 1)**. The polygons been drawn by connecting 4 points in a specific order. The algorithm of the drawing as follows: starting from bottom-left, first connect bottom-left to bottom-right, second connect from bottom-rigth to top-right, lastly, connect top-right to top-left. Every polygon are been drawn in this order. The surface of Perspective Plots is formed until all the polygons are been drawn.

Before drawing the surface, the transformation of 3-D vertices into 2-D vertices is required. This transformation required two main variables, the 3-D vertices and 4×4 viewing transformation matrix **p**. The 3-dimenstion vertices are computed, the matrix **p** can been recored from the `persp()` call. This transformation can be done easily on R by using the `trans3d()` function.

```
> points3d = trans3d(xTmp, yTmp, zTmp, trans)
> head(points3d$x)

[1] -0.3928108 -0.3861145 -0.3792354 -0.3721881 -0.3649927 -0.3576724

> head(points3d$y)

[1] -0.1066821 -0.1090048 -0.1121947 -0.1161865 -0.1208728 -0.1261142
```

Because of we are drawing a 3-D surface in a 2-D plane, some polygons that stay 'behind' can not been seen, It is necessary to draw the polygons in a right order. The order defined by using the **x** and **y** coordinate of the 3-D vertices (but ignore the **z** coordinate) combinding an other column **1**, then do the matrix multiplication with the viewing transformation **p**. The fourth column from this multiplication is the drawing order of the polygons.

```
> orderTemp = cbind(xTmp, yTmp, 0, 1) %*% trans
> zdepth = orderTemp[, 4]
> ## the zdepth of a set of 4 points of each polygon
> a = order(zdepth, decreasing = TRUE)
> head(a)
```

```
[1] 3541 3542 3481 3543 3482 3544
```

The following figures shows how does this paper approximate to the solution. The top-left figure is drawn by plotting the transformed 2-dimenstion points, the shape of the Perspective Plots been presented. The top-right figure is drawn by connecting the points line-by-line, the shape become more obvious. The bottom-left figure is drawn by using the `grid.polygon`. By default, the origin order of the polygons is drawn along x-axis, then along y-axis. Clearly this is not the correct order. Finally, the bottom-right figure shows the true Perspective Plots by fixing the order.

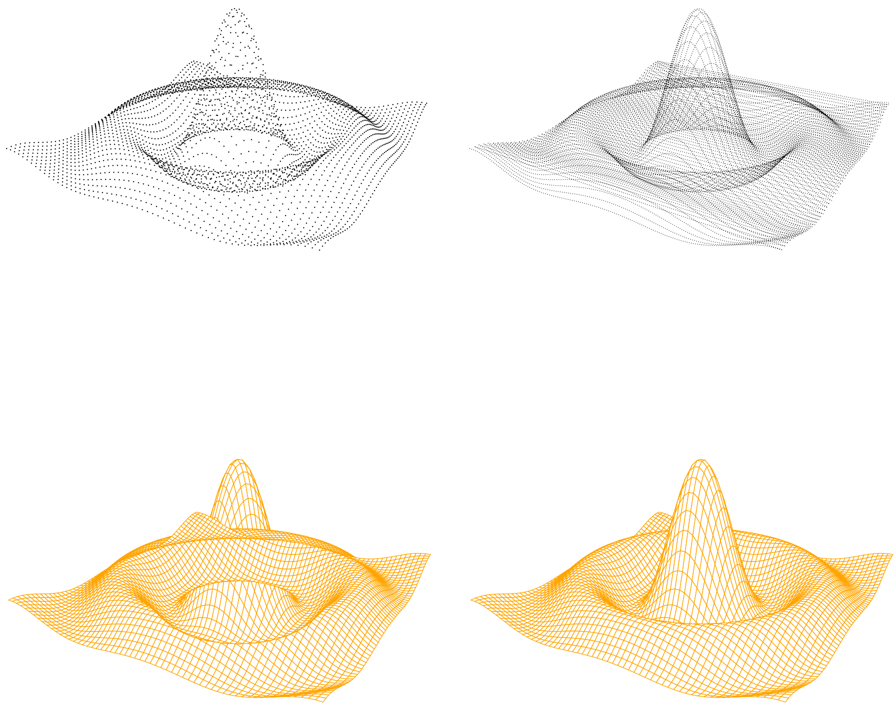

Figure 6: The top-left figure is only plotting the transformed 2-dimenstion points. The top-right figure is been drawn by connecting the points line-by-line. The top-right figure is drawn unorderly by using the grid.polygon. Finally, the bottom-left figure is drawn in a correct order.

0.3.2 lighting

The other main benifit supported by `persp()` is that it can shade the surface by assuming the surface being illuminate from a given direction..... In `persp()`, the main parameters that user need to specify for produce a shaded perspective plots are: **ltheta**, **lphi** and shade **ltheta** and **lphi** are used for setting up the direction of the light source. In particular, **ltheta** specified the angle in z direction, **lphi** specified the angle in x direction. **shade** is the parameter that specified the shade at each facets of the surface, the shades will compute as follows:

$$(\frac{1+d}{2})^{shade} \tag{1}$$

Where **d** is the dot product of the unit vector normal to each of the facet(u) and the unit vector of the direction of the light(v).

The color of each facet will been calculated by the color that recored from the graphics engine display list multiply by the **shade**. Finally, the surface been drawn by filling the colors for every facets.

If the normal vector is perpendicular to the direction of the light source, then $d = 0$ and the term $\left(\frac{1+d}{2}\right)^{shade}$ will be close to 0, therefore the corrosponding facets will become darker, the brightness and darkness will depend on the value of the **shade** if shade close to 0, the term $\left(\frac{1+d}{2}\right)^{shade}$ will close to 1. Therefore it will looks similar to non-shading plot. Simiarlly, if shade gets larger, the term close to 0 and the plot gets more darker.

```
> trans = persp(x, y, z, theta=30, phi = 20, expand = 0.5,
+  col = 'White', border = 'NA', shade = 0.5, ltheta = 30, lphi = 20)
> grid.echo()
```

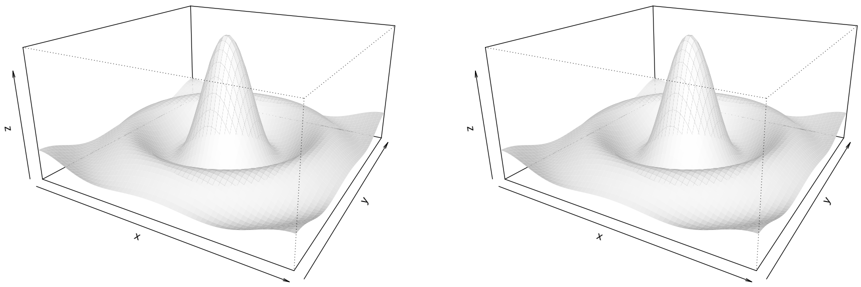

Figure 7: Adding a light source to the perspective plot from the same angel of view. The left figure been drawn by `graphics` and the right figure been drawn by `grid`, they are identical to each other.

0.3.3 Difference beween `C` and `R`

As we know, most base functions of `R` are been written by `C` include `persp` and `fill.contour`. Although the structure of `C` code are quite simliar to `R` code in some special case, there are some `C` code structures which behave compeletely different to `R`, therefore translate `C` code to `R` code is not just "copy-and-paste", even just doing direct translation.

Pointers

One main data structure in `C` is the pointers, which is a type of reference that records the address/location of an global object or a local variable in a function. Pointers can be manipulated by using assignment or pointer arithmetic.

```
> # static int LimitCheck(double *lim, double *c, double *s)
> # {
> #     if (!R_FINITE(lim[0]) || !R_FINITE(lim[1]) || lim[0] >= lim[1])
> #         return 0;
> #     *s = 0.5 * fabs(lim[1] - lim[0]);
> #     *c = 0.5 * (lim[1] + lim[0]);
> #     return 1;
> # }
```

The top piece of code is used for checking the Limit for the `persp()` function. It also multifying the variable c and `s` for further calculation. In this case, the `*c` and `s*` are the pointer which will pointing to the mechine memory of `s` and `c` and multify them. However, this process cannot be reproduce on `R` because `R` does not have the pointer data structure. One possible solution will be rather than doing the Limit checking and multify `s` and `c`, do the limit checking and return/assign the `s` and `c` as xs ad ys for further calculation.

```
> # LimitCheck = function ( lim ) {
> #     ## not finished yet...
> #     s = 0.5 * abs(lim[2] - lim[1])
> #     c = 0.5 * (lim[2] + lim[1])
> #     c(s, c)
> # }
> # xs = LimitCheck(xr)[1]
> # xc = LimitCheck(xr)[2]
> # ...
```

The best way to reproduce this behaviours on `grid` is by translating the `C` code to `R` code directly. However, it is not as simliar as ”copy-and-paste” since the structure of `C` is quite different from `R`. The following example shows the different structure between `C` and `R`.

`C` allows programmer to access the address of memory location for every variable, but we have no permission on `R`, hence we need to do a bit more work. The functions on `C` (`SetToIdentity`, `XRotate` and `ZRotate`) are all accessing the memory of the variable `VT` and eidting it. The last part of `R` code is to approch the setp as `C` doee, that is, updating the matrix `VT` by keep multifying with other rotation matrix.

0.3.4 Box and axes