UNDIFINED

February 11, 2017

1 Introduction

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.

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 theplot using grid so that the user may edite the plot in more ways than they can with the original plot drawn by bacis 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()
```

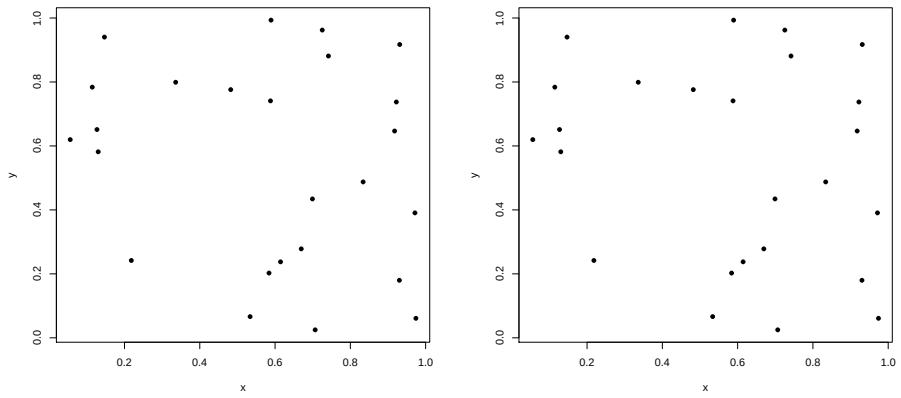

Figure 1: The left plot is drawn by using plot(); the Right plot is redrawn using grid.echo(). Overall, two plots are identical to each other

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 recored 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
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 mainpulate 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"))
```

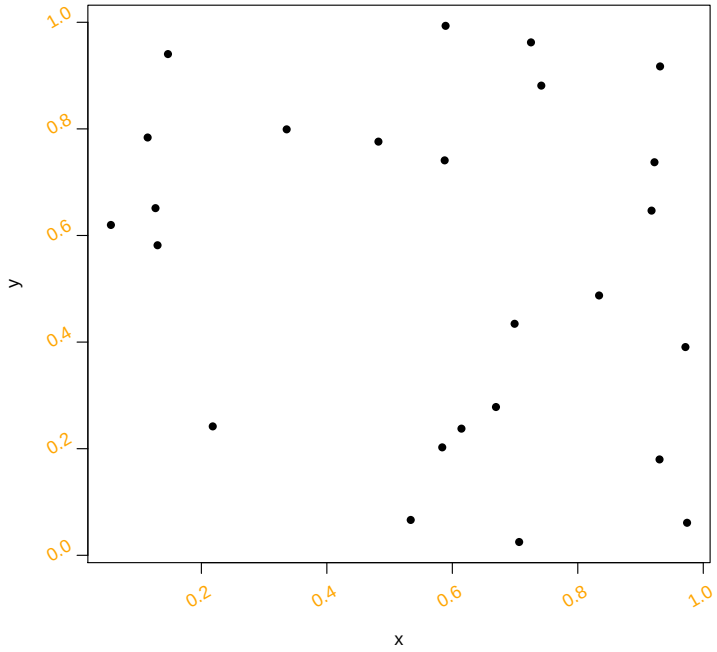

Figure 2: The angle and the color of the bottom and left axis of the previous plot have been changed by 30 degrees and orange

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.

```
> 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()
```

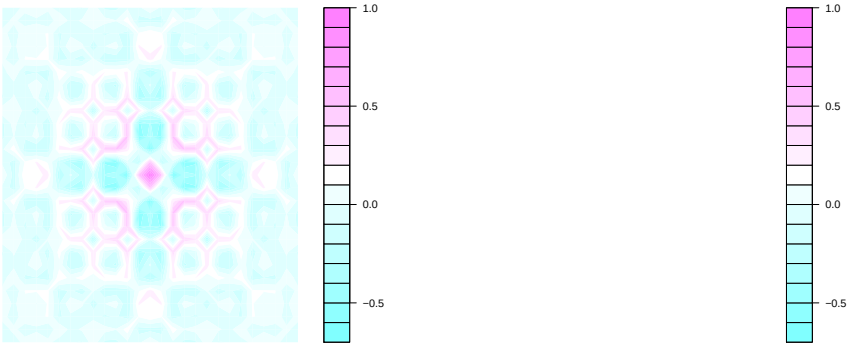

Figure 3: The left plot been drawn by using `filled.contour` and the right plot been redrawn by calling `grid.echo()`. There is a "blank" page on the right plot because the grid.echo cannot emulate filled.contour()

1.4 Aim of this project

The functions `persp()` and `filled.contour()` are written in C. However, it is very hard to debug and track the C code. One possible solution will be:

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`

2 comment

NOTE to Jason: explain how gridGraphics works first: graphics display list; gridGraphics implements an R version of each low-level C function on the display list (e.g., for C_plot_xy there is an R function called C_plot_xy in the gridGraphics package). THEN maybe write about 3D to 2D transformations, but only maybe.

3 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 texttt-tfilled.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$speed, cars$dist, col = 'orange',
+       pch = 16, xlab = 'speed', ylab = 'dist')
> reco = recordPlot()
> ## Displays the inputs
> 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
[26] 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24 24 24 24 25

$y
 [1]   2  10   4  22  16  10  18  26  34  17  28  14  20  24  28  26  34  34  46
[20]  26  36  60  80  20  26  54  32  40  32  40  50  42  56  76  84  36  46  68
[39]  32  48  52  56  64  66  54  70  92  93 120  85

$xlab
[1] "cars$speed"

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

This example shows that: suppose we have a data set called `cars`, which contain two columns, the speed of the cars and the distance of travel. We have a plot which plotted the speed againist to the distance of travel. The `recordPlot()` will save this plot as an R object. As result, we can access the information of this plot. The last line of the code will access the x-coordinate and the y-coordinate, or the x-label and the y-label.

There are many way for solving this problem, one possible solution will be translate the C code to R code such that they are as simliar as possible. The reason for doing this direct translation include:

1. It is hard to debug and track the C code.
2. It is very simple to debug the R code. If the R code is almost identical to the C code, then we can debug the R code to ensure that the R code can also provide the same result.

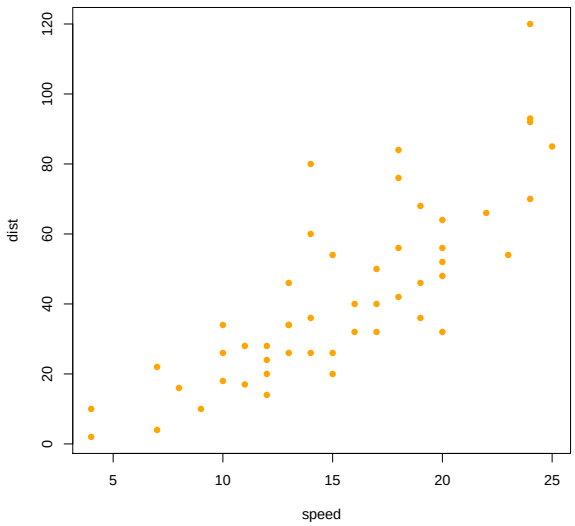

Figure 4: dist vs speed

4 standalone

The Perspective Plots is used to draw a surface over the x-y plane. Usually, it has three main argument, *x, y z.* 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= 30)
> 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 = 30, expand = 0.5, col = "yellow")
```

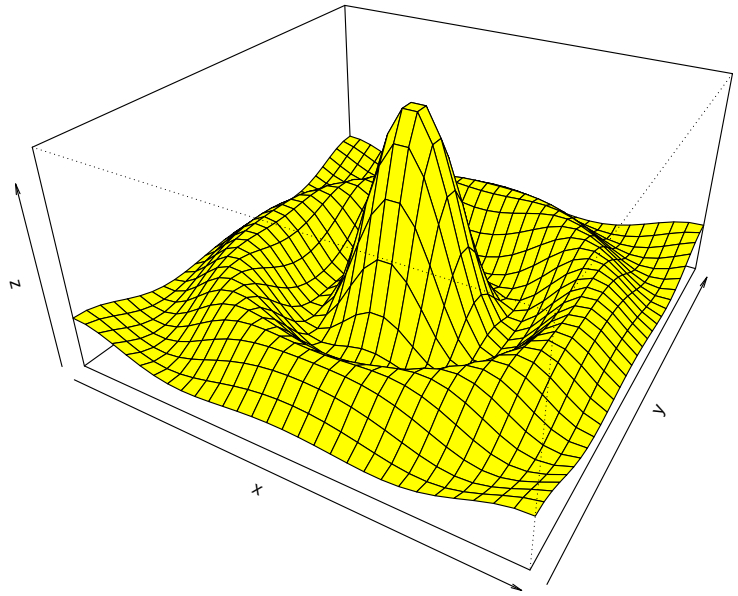

Figure 5: An example of Perspective Plot been drawn by `persp()`

From the pervious example, we discovered 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 inorder to get values of all vertices. The folowing 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  -9.310345  -8.620690  -7.931034  -7.241379  -6.551724

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

```
[1] -10.000000  -9.310345  -8.620690  -7.931034  -7.241379  -6.551724
```

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

```
           [,1]       [,2]
[1,]  0.7070981  0.6512071
[2,]  0.6512071  0.4291166
[3,]  0.4502042  0.0960367
[4,]  0.1532827 -0.2695253
[5,] -0.1766027 -0.5910827
[6,] -0.4800328 -0.8127730
```

4.1 The translation from 3-dimension points into 2-dimension 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-dimensions.

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 $dim(x) \times dim(y)$.

```
> 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
```

One 3-dimenstions points contains a set values of (x, y, z), but z is $dim(x) \times dim(y)$ 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 dimension of x, y and z need to be matched and also in a right order.

First step is the reduce the z $dim(x) \times dim(y)$ matrix into a one dimension vector which has length of $dim(x) \times dim(y)$. 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 x_p, y_p, z_p is the 3-dimensions points which preper for computing the vertics.

The idea of transform the points into vertics 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 polygons have 4 vertics. The total number of polygons are required to be drawn is depent 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 polygons are been drawn in this order. The surface of Perspective Plots is been formed until all the polygons are been drawn.

Before drawing the surface, the transformation of 3-dimensions vertics into 2-dimension vertics is required. This transformation required two main variables, the 3-dimension vertics and 4×4 viewing tranformation matrix p. The 3-dimenstion vertics 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] -0.3855721 -0.3715974 -0.3567724 -0.3413650 -0.3256723 -0.3099360

> head(points3d$y)

[1] -0.1141316 -0.1210975 -0.1310004 -0.1428878 -0.1555315 -0.1677817
```

Because of we are drawing a 3-dimension surface in a 2-dimension 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 vertics (but ignore the z coordinate) combinding an other column **1**, then do the matrix multification with the viewing transformation p. The fourth column from this multification 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] 871 872 841 873 842 874

> graphics:::plot(points3d$x, points3d$y, type = 'l', lty = '1212', asp = 1, axes = FALSE,
```