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

May 27, 2017

Contents

List of Figures

Chapter 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 the plot using `grid` so that the user may edit 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`, see Figure 1.1

```
> set.seed(110)
> x = runif(25)
> y = runif(25)
> plot(x,y, pch = 16)
> grid.echo()
```

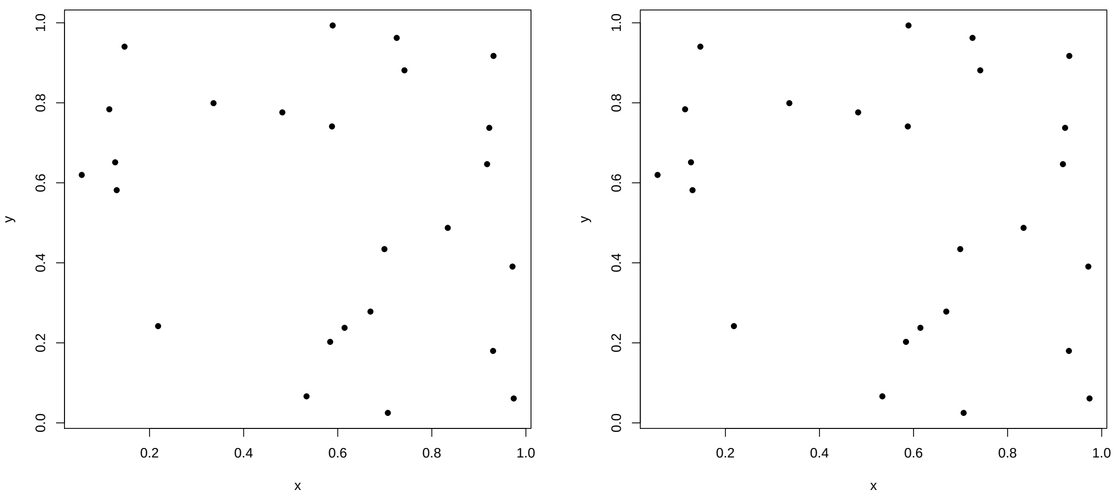

Figure 1.1: The left plot is drawn by using `plot()`; the Right plot is redrawn using `grid.echo()`. Two plots are identical to each other

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 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
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. See Figure 1.2

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"))
```

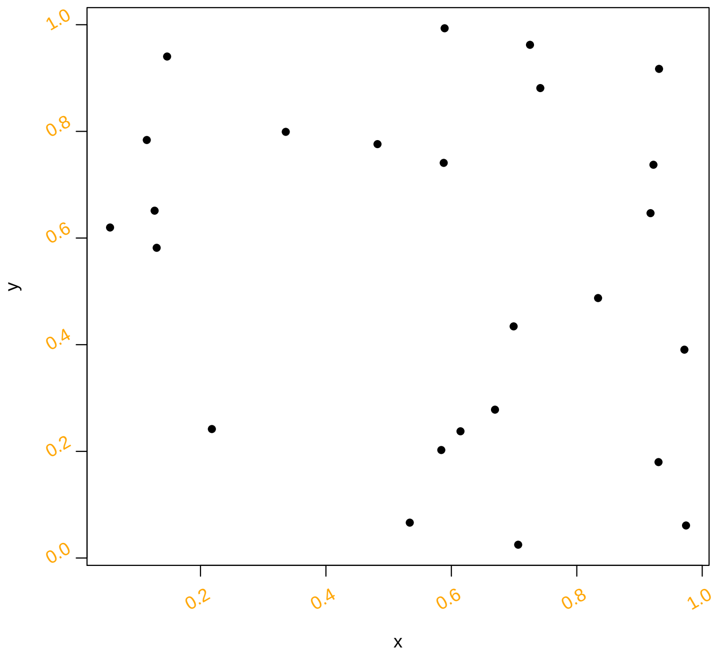

Figure 1.2: The angle and the color of the bottom and left axis of the previous plot have been changed by 30 degrees and orange

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. See Figure 1.3.

```
> 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()
```

1.4 Aim of this project

The purpose of this paper is emulate the Perspective Plots, `persp()` and Level (Contour) Plots, `filled.contour()` using the `grid` package. However, these two functions are written in C, as part of the core `R` source code. This means that a normal `R` user or developer cannot modify the code. Also, the `C` code is structured so that the normal `R` user or developer cannot separately call the `C` code. The solution of this paper as follows:

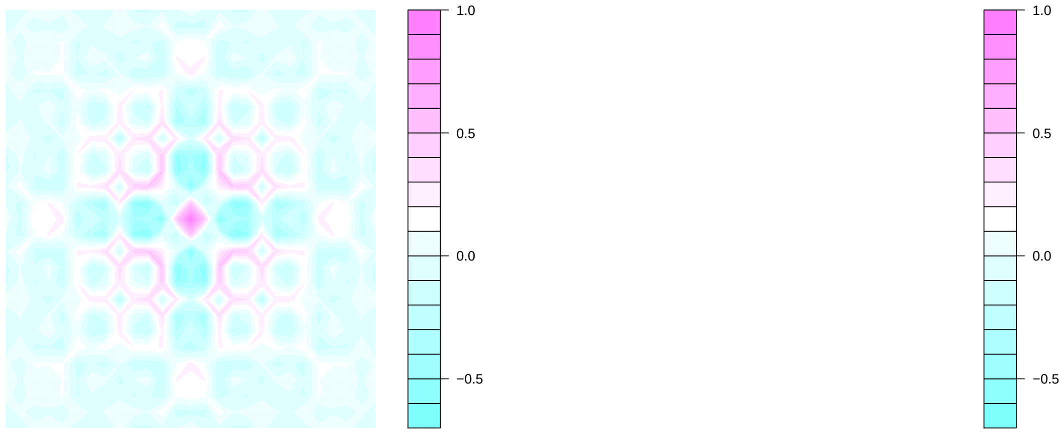

Figure 1.3: The left plot been drawn by using `filled.contour()` and the right plot been redrawn by calling `grid.echo()`. There is a "blank" page on the right plot because the grid.echo cannot emulate `filled.contour()` in this stage.

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. integrate the standalone to the `gridGraphics`
   (a) navigating to the correct viewport that has been set up by `gridGraphics`
   (b) Create a new viewport for setting up the correct x-scale and y-scale preparing for drawing, (if necessary) then draw the contents in the correct viewport.
   (c) Test the difference of the plots that been drawn by `graphics` and `grid`

Chapter 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. See Figure2.1.

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 `filled.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()[[1]][[4]][[2]][[2]]
> head(reco[[1]]) #x

[1] 4 4 7 7 8 9

> head(reco[[2]]) #y

[1]  2 10  4 22 16 10

> reco$xlab

[1] "cars$speed"

> reco$ylab

[1] "cars$dist"
```

The example demonstrates 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.

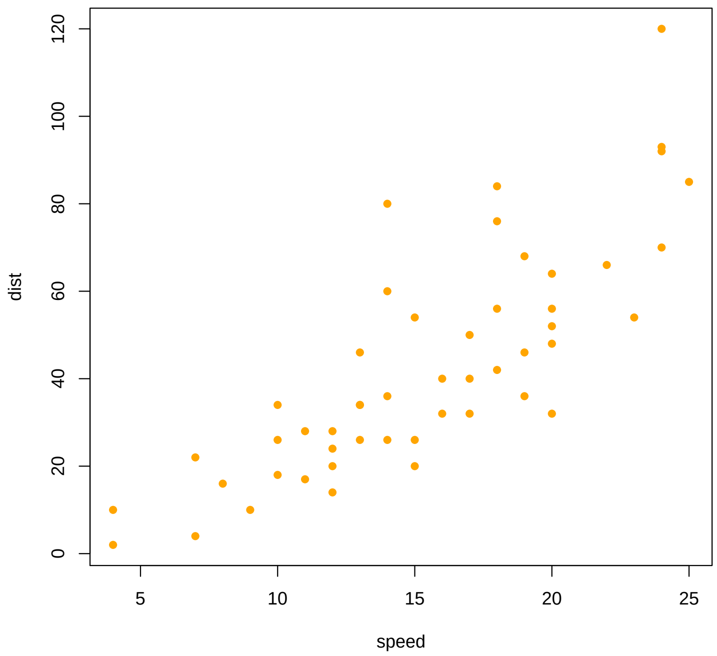

Figure 2.1: The details of the plot of dist vs speed displayed by the graphics engine display list

Chapter 3

Standalone

3.1 The Perspective Plots `persp()`

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. See figure 3.1

```
> x = y = seq(-10, 10, length= 40)
> 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')
```

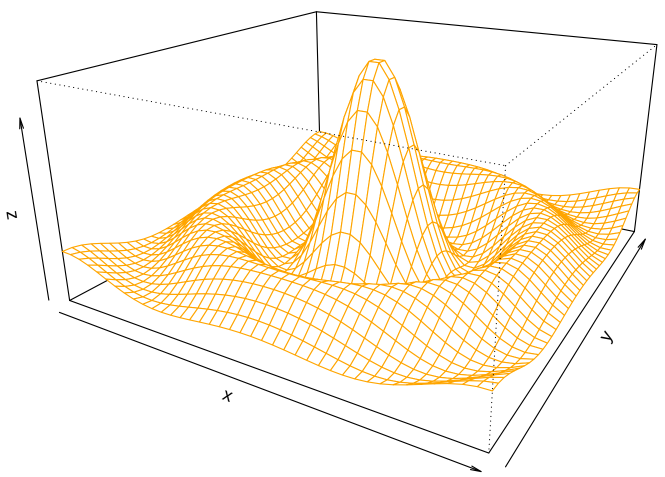

Figure 3.1: An example of Perspective Plot been drawn by `persp()`

It is clearly to see that the Perspective Plots is formed by a finite number of ”polygon”, each polygon has 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 some information from the graphics engine display list. However, the value of the vertices is not in the display list, therefore the plot cannot be reproduced directly. But we can access value of **x**, **y** and **z**, therefore we should re-do the calculation 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 " caught " from the display list.

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

[1] -10.000000  -9.487179  -8.974359  -8.461538  -7.948718  -7.435897

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

[1] -10.000000  -9.487179  -8.974359  -8.461538  -7.948718  -7.435897

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

            [,1]       [,2]
[1,]  0.70709805  0.6807616
[2,]  0.68076162  0.5602102
[3,]  0.56890062  0.3623630
[4,]  0.38799490  0.1144364
[5,]  0.16158290 -0.1521360
[6,] -0.08388565 -0.4067000
```

3.1.1 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 previous 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 **length(x)** $\times$ **length(y)**.

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

First step is the reduce the **Z** matrix into a one dimension vector which has length of **length(x)** $\times$ **length(y)**. It can be 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 previous section, we explained that the Perspective Plots is made by finite number of polygons. Each polygon has 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-right to top-right, lastly, connect top-right to top-left. Every polygon is being 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] -0.3929051 -0.3827005 -0.3720915 -0.3611435 -0.3499392 -0.3385634
```

```
> head(points3d$y)
```

```
[1] -0.1060481 -0.1099038 -0.1156894 -0.1230654 -0.1315269 -0.1404974
```

Because of we are drawing a 3-D surface in a 2-D plane, some polygons that stay 'behind' cannot 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) combining another 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] 1561 1562 1521 1563 1522 1564
```

Figure 3.2 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.

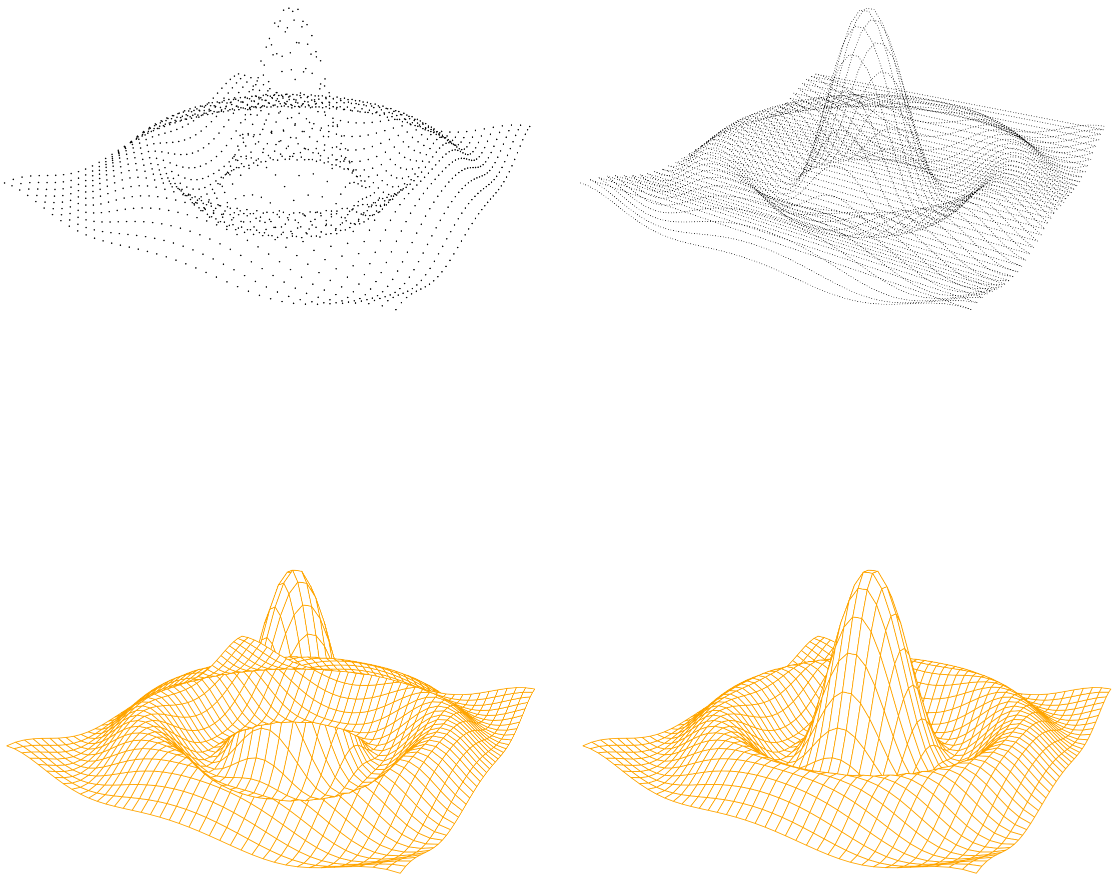

Figure 3.2: The top-left figure is only plotting the transformed 2-dimenstion points. The top-right figure is being drawn by connecting the points line-by-line. The top-right figure is drawn unorderly by using the grid.polygon. Finally, the bottom-left figure is drawn in a correct order.

3.1.2 Lighting

The other main benefit supported by `persp()` is the shading. It shades the surface by assuming the surface is being illuminated from a given direction. (light source)

In `persp()`, the main parameters that the user needs to specify for producing a shaded perspective plot is: *ltheta*, *lphi* and *shade*.

ltheta and *lphi* are used for setting up the direction of the light source. In particular, *ltheta* specifies the angle in the z direction, *lphi* specifies the angle in the x direction.

shade is the parameter that specifies the shade at each facets of the surface, and the shades will be computed as follows:

$$(\frac{1+d}{2})^{shade} \tag{3.1}$$

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

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

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 be close to 1. Therefore, it will look like non-shading plot. Similarly, if the shade gets larger, the term close to 0 and the plot gets darker.

```
> trans = persp(x, y, z, theta=30, phi = 20, expand = 0.5,
+  col = 'white', border = 'orange')
> trans = persp(x, y, z, theta=30, phi = 20, expand = 0.5,
+  col = 'white', border = 'orange',
+  shade = 0.8, ltheta = 30, lphi = 20)
```

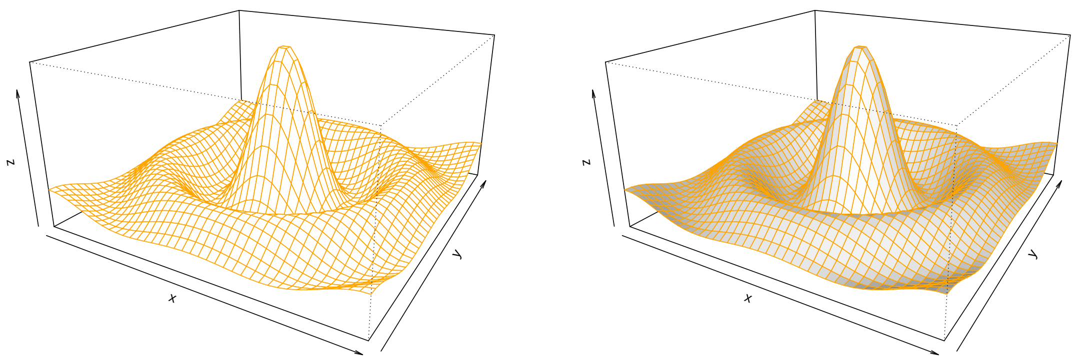

Figure 3.3: Adding a light source to the perspective plot from the same angel of view. The left plot is origin plot without shadding, where the right plot is been shaded.

3.1.3 Difference beween `C` and `R`

Many functions in `R` call `C` code to do a lot of the work. This is the case for `persp()` and `filled.contour()`. Although the structure of `C` code is quite similar to `R` code in some special cases, there are some `C` code structures which behave completely different to `R`, therefore translating `C` code to `R` code is not just "copy-and-paste".

Pointers

One main data structure in `C` is the pointer, which is a type of reference that records the address/location of a 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 multiplying the variable c and `s` for further calculation. In this case, the `c*` and `s*` are the pointer which will point to the machine memory of `s` and `c` and modify them.

However, this process cannot be reproduced on `R` because `R` does not have the pointer data structure. One possible solution will be rather than doing the Limit checking and multiply `s` and `c`, do the limit checking and return/assign the `s` and `c` as `xs` ad `ys` for further calculation.

```
LimitCheck = function ( lim ) {
    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]
...
```

Array

The other main difference is that `C` uses array data format but `R` uses matrix data format.

```
FindPolygonVertices(c[k - 1], c[k],
        x[i - 1], x[i],
        y[j - 1], y[j],
        z[i - 1 + (j - 1) * nx],
        z[i + (j - 1) * nx],
        z[i - 1 + j * nx],
        z[i + j * nx],
        px, py, pz, &npt);


out = lFindPolygonVertices(sc[k], sc[k + 1],
        x[i], x[i + 1],
        y[j], y[j + 1],
        z[i, j],
        z[i + 1, j],
        z[i, j + 1],
        z[i + 1, j + 1],
        px, py, pz, npt)
```

To get the same elements in the matrix as the elements in the array, one solution is to change the matrix data format into vector data format. However, `R` is farmillar with matrix data structure. Hence it is easiler and more understandable by `R` user and programmer. The `z[i - 1 + (j - 1) * nx]` in FindPolygonVertices() is selecting the i^{th} element from the j^{th} column, which the following `R` code provied the same result That is, `z[i, j]` in `R` will provide the same result as `C`

The top piece of codes is both calling the `FindPolygonVertices()` function by feeding parameter into it. However, the `z` is array in the first call as it written on `C` but the second is matrix as it written on `R`. the `- 1` on the `R` code because `C` starting at 0 index but `R` starting at 1.

3.1.4 Box and other features

One feather that `persp()` supported is whether draw a container (box) around the surface. (See figure 3.4). Therefore, it is necessary to find out whether the edge of the box in front of the surface or behind the surface.
Maybe rewrite this paragraph?

The solution will be that translates the `C` code to `R` code directly. The reason for doing this directly translation is that R is sensitive on drawing the dot lines. More specifically, it may cause difference if we connect two points with a dotted line in different direction. Due to the purpose of this paper, the plot should be drawn as identical as possible. Therefore, the direct translation is required.

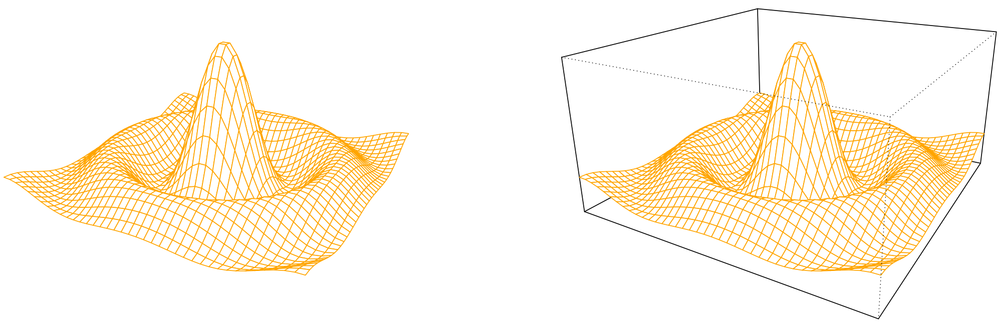

Figure 3.4: The surace been drawn by ignore the box in the left plot, right plot drawn the surface as well as box

Other feather that `persp` supported is the detail of the axis.(See figure 3.5) More specifily, the axis has three type, no axes, simple axes which only contain the label of axes, or showing the scale of each axes. These feathers are required to be reproduced by `grid`, The solution to this problem by translating the `C` code to `R` code directly.

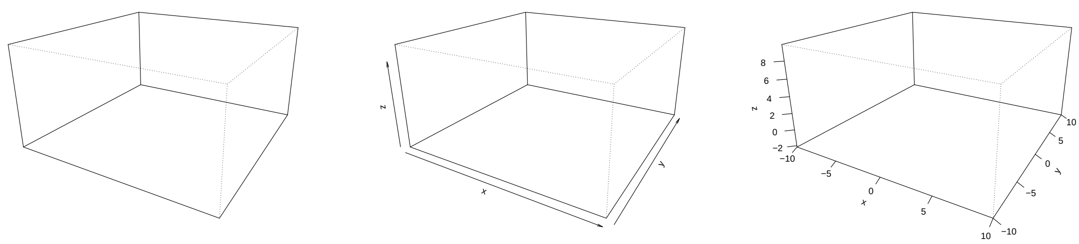

Figure 3.5: The Perspective surfaces are been ignored in this example, the left plot shows no axis been drawn, the simple axes been drawn in the middle plot and the right plot shows more detail for each axis.

3.2 The Filled Contour Plot

3.2.1 Direct translation from C to R

The other tasks of this paper is to emulate the Level (Contour) Plots (`filled.contour`) from **graphics** to **grid**. As previous section, the first step to emulate `filled.contour` is to access the information from the graphics engine display list. (See figure 3.6)

```
> x = 10*1:nrow(volcano)
> y = 10*1:ncol(volcano)
> filled.contour(x, y, volcano, color = terrain.colors,
+     plot.title = title(main = "The Topography of Maunga Whau",
+     xlab = "Meters North", ylab = "Meters West"),
+     plot.axes = { axis(1, seq(100, 800, by = 100))
+                   axis(2, seq(100, 600, by = 100)) },
+     key.title = title(main = "Height\n(meters)"),
+     key.axes = axis(4, seq(90, 190, by = 10)))
> xx = recordPlot()
> info = xx[[1]][[12]][[2]]
> head(info[[2]])  ## print the values of x

[1] 10 20 30 40 50 60

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

[1] 10 20 30 40 50 60

> dim(info[[4]])  ## print the dimension of z

[1] 87 61

> length(info[[5]])  ## print the length of s

[1] 22
```

The example shows the plot of topography of Maunga Whau and also the information from the `filled.contour` call in the graphics engine display list. Same problem as `persp()`, there is no way to reproduce this plot directly by only using the coordinates of x, y and z.

There is an algorithm to create this contour plot in the `graphics` package written by `C`. The first step of the solution will be translated the `C` code directly to maximize the accuracy.

```
static void
FindPolygonVertices(..., double *x, double *y, double *z, int *npt, ...)
{
  *npt = 0;
  FindCutPoints(low, high, x1, y1, z11, x2, y1, z21, x, y, z, npt);
  FindCutPoints(low, high, y1, x2, z21, y2, x2, z22, y, x, z, npt);
  FindCutPoints(low, high, x2, y2, z22, x1, y2, z12, x, y, z, npt);
  FindCutPoints(low, high, y2, x1, z12, y1, x1, z11, y, x, z, npt);
}
```

This piece of `C` code is the algorithm used for calculate the coordinates of the vertex of each polygon in the level contour plot. The parameters $*x$, $*y$, $*z$ are the array pointers which have length of 8 individually, $*npc$ is also a pointer has length of 1. If the `FindCutPoints` is called, the elements in the arrays of x, y, z will be modified. In general, we feed the location of memory of x, y, z and npt to `FindPolygonVertices()` and modify the values of x, y, z and npt within the `FindCutPoints`.

For example, the first call of `FindCutPoints()` modifies the elements in the pointer arrays of x, y, z. The location of elements in arrays been modified will depend on the parameter $*npt$. More specifically,

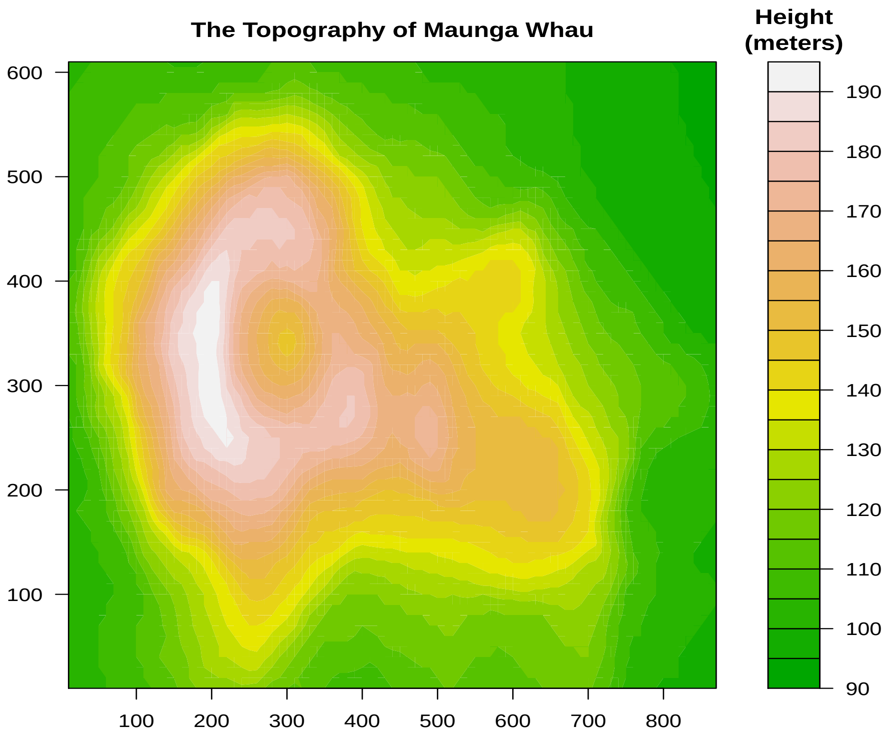

Figure 3.6: The topography of the Maunga Whau been drawn by using the `filled.contour`

the $*x$ as a function of x_1 and x_2, y as a function of y_1 and so on. The second `FindCutPoints()` is slightly different, x will depend on a function of x_2, y as a function of y_1 and y_2. In the third `FindCutPoints()` call, x will depend on a function of x_2 and x_1, y will depend on a function of y_2. Finally, x will depend on a function of x_1, y depend on the function of y_2 and y_1.

There is no pointer data structure in `R` hence we cannot produce the same action as `C`. One approximation to this action will be as follows:

```
lFindPolygonVertices = function(...)
{
    out = list(); npt = 0
    out1 = lFindCutPoints(...)
    x = y = z = numeric(8); npt = out1$npt
     ...
    out$x = out1$x + out2$y + out3$x + out4$y
    out$y = out1$y + out2$x + out3$y + out4$x
    out$npt = out4$npt
    out
}
```

Instead of mortify x, y, z and npt inside `FindCutPoints()`, record the values for x, y, z and npt outside the `lFindCutPoints()` call in `R` every time. At last, we combined every x and y together as the previous `C` code behave.

3.2.2 Vectorization

In `C`, the total iteration in the loops is equal to

$$Total = nx * ny * ns \tag{3.2}$$

Where:
$nx = length(x) - 1$,
$ny = length(y) - 1$,
$ns = length(levels) - 1$
It requires huge iteration. For example, In Figure 3.6, the Topography of Maunga Whau, the length of x is 87 and the length of y is 61, where the length of levels is 22. Therefore, there are at most 108360 polygons that we need to consider which it will slow down the software. Instead of calculate the coordinates of every polygons one by one, it is a good idea to vectorize the function to speed up the software.

```
> ## Time compartion
> system.time(filled.contour.volcano())
> # user  system elapsed
> # 0.01    0.05    0.07
>
> ## time for for loop version of filled.contout()
> system.time(grid.echo())
> # user  system elapsed
> # 10.03    0.23   10.32
>
>
> ## time for for vectorizetion version of filled.contout()
> system.time(grid.echo())
> # user  system elapsed
> # 1.28    0.53    1.82
```

The pervious code is comparing the time for drawing the filled contour plot of the Topography of Maunga Wha.
The previous code is used for comaring the time for directed translation from `C` (for loop vertion) and `R` optimize version (vectorization) of `filled.contour()`. The time for drawing the filled contour plot directly from `C` takes 0.07 second in total.
We redraw this plot on grid by using different method, and record the time spent for each method by using `system.time()`. The first record time is for the for loop version. It took more than 10 seconds to redraw the plot, which is very slow. The time of vectorization version recorded at the next, which only takes 1.82 second in total. It is much faster than the for loop version.

Chapter 4

Integrate to `gridGraphics` package

4.1 Some concept of `grid`

The previous section explained the internal calculation for `persp()` and `filled.contour()`. It works perfectly outside the `gridGraphics` package. However, `grid.echo()` still cannot emulate thest two kinds of plot because they are not been integrated to the package yet. Therefore, it requires more work.
The `gridGraphics` package provides the structure of viewports which act identical to the layout of plots in the plot region been drawn by `gridGraphics`. See figure 4.1.

```
> set.seed(110)
> par(mfrow = c(1,2))
> x = rnorm(1000)
> hist(x, probability = T)
> plot(density(x))
> grid.echo()
```

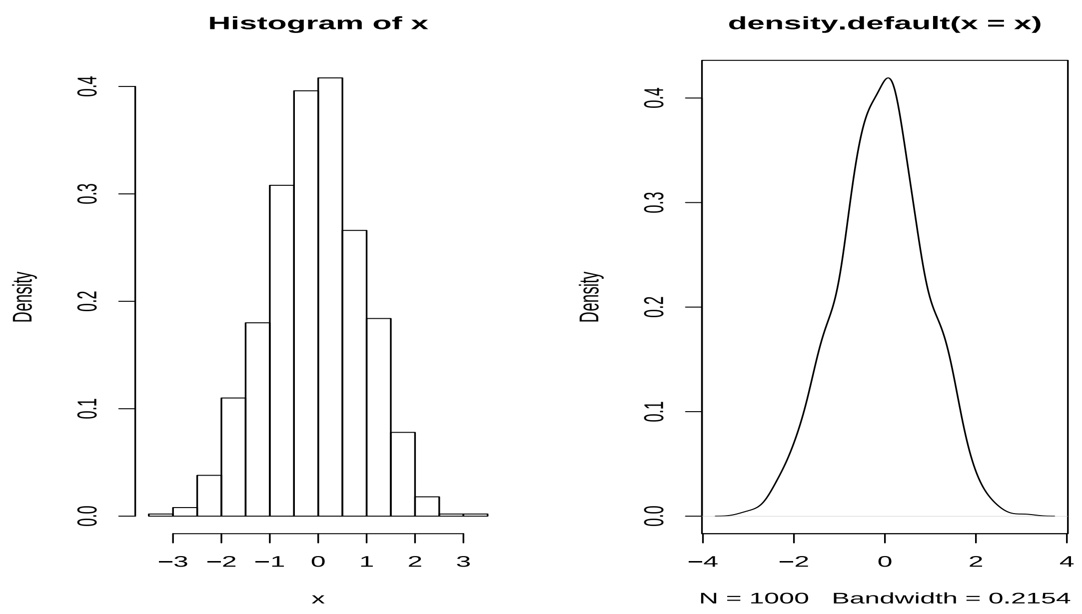

Figure 4.1: display the `grid` version of draw two plots into one overall graph by setting `par(mfrow())`. The left-plot is the histogram of observations generated by standard normal distribution, right-plot is the density plot of the observations

```
grid.ls(viewports = TRUE)

 graphics-root
   graphics-inner
     graphics-figure-1
       graphics-plot-1-clip
         graphics-window-1-1
           graphics-plot-1-rect-1
           5
 graphics-root
   graphics-inner
     graphics-figure-2
       graphics-plot-2-clip
         graphics-window-2-1
           graphics-plot-2-lines-1
           5
```

The previous code used the `graphics` package to plot a histogram on the left and a density plot of a thousands of random observation generated by standard normal distribution. Then redraw the plot on `grid`. The grobs object and the veiwports are created.

The grobs object and the veiwports and been listed by `grid.ls()`. In the previous example, we can see that the contents(rectangles) of histogram were drawn in the viewports of *graphics-inner::graphics-figure-1::graphics-plot-1-clip::graphics-window-1-1*. The density plot was drawn in the other viewports which is *graphics-inner::graphics-figure-2::graphics-plot-2-clip::graphics-window-2-1*. Although it is completely different structure of the plot that was drawn by `graphics`, they are identical to each other.

To reproduce the same plots as `graphics`, we need to modify the `grid` structure of the plot so that it behaves identically to the plot that was drawn by `graphics`. In this example, the viewports need to be set in the same location and the same size as the `graphics` plot region, and also, the x-scale and y-scale of the viewports in `grid` need to be set the same user coordinates as in `graphics`.

4.2 Integrate `persp()`

The core of `gridGraphics` package provides some basic viewport structure to support the perspective plots(`persp()`), based on the gerneral plots that been drawn by `graphics`. However, there are some specific details that `gridGraphics` not fully supported. The following problems need to be solved before integrating `persp()` into the package:

- The viewport that `persp()` needs to be on
- The xcale and yscale need to be calcualted
- whether the clipping happens for every component when drawing

It is not allowable to call `grid.newpage()` to create a new page for drawing a perspective plot since it will destory other feathers within the plot. For example, the points and lines added to perspective plot. These features will desappear when calling `grid.echo()` to reproduce the plot, since they are in a different viewport and the current graphics devices only contains the viewport that I created. Therefore, it is necessary to draw the perspective plot in the correct viewport.

The other problem that we may consider is the actual scale for the viewport, i.e. the x-scale and y-scale. Unfortunately, `gridGraphics` does no support the calculation for the actual limit of x and y since the other kinds of plot that `graphics` provides is in two dimension. The calculation of the limit of x and y is not as simple as `range(x)` or `range(y)`, because there is one more dimension of z. And, the plot is drawn in a two-dimension graphics devices.

The final problem will be wether the clipping happeneds. More specifically, the different components of the perspective plot should be drawn in clipped region or non-clipped region.

The first problem can be solved by ...

The limit of x and y will depend on the ratio of horizontal and verticle length of the current windows graphics devices. On `grid`, it is simple to track the actual length of the viewport in the current windows graphics devices.

The following exmple is calculating the verticle length and horizontal length of the viewport in current windows graphics devices. (See figure 4.2) The dotted rectangle region is the viewport region that we focus on. As result, the verticle length of the viewport in current windows graphics devices in my PC is 5.16 inches, where the horizontal length is 5.76 inches.

```
> plot(cars$speed, cars$dist, col = 'orange',
+      pch = 16, xlab = 'speed', ylab = 'dist')
> grid.echo()
> downViewport('graphics-plot-1')
> grid.rect(gp = gpar(col = 'red', lty = 12221, lwd = 2))
> convertX(unit(1.0, 'npc'), 'inches')

[1] 5.76inches

> convertY(unit(1.0, 'npc'), 'inches')

[1] 5.16inches
```

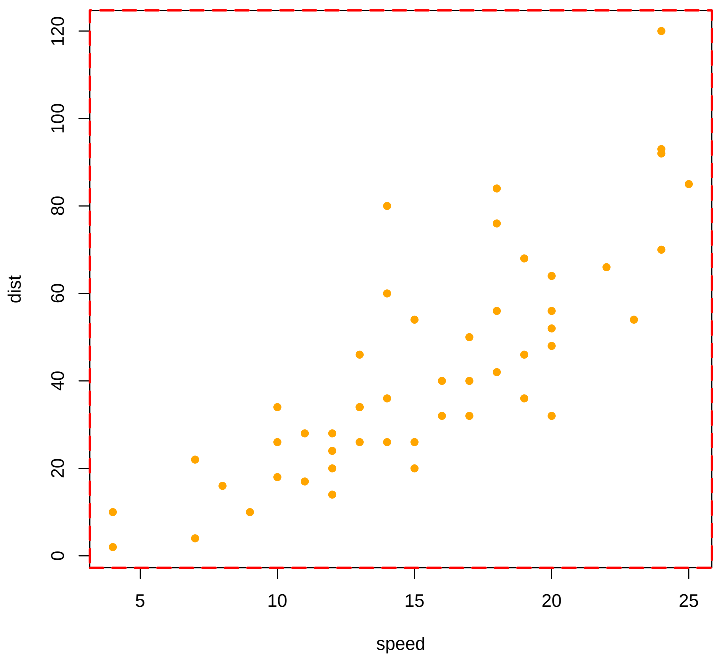

Figure 4.2: Used the example from chapter 2, calculate the actual verticle length and horizontal length of the viewport (the region of the red dotted line)

The idea of this example is that it is possible to track the actual vertical length and horizontal length by navigating to the specific viewport and record them. It leads to the solution of our second problem, first of all, we navigate to the viewport that has been drawn by `persp()`, calculate the limit of x and y base on the size of this viewport. Then we create another viewport (visible for other `gridGraphics` functions eg `lines()` and `points()`) that has the same location and the same size as the previous viewport. Then we modify the xscale and yscale from the new veiwport to be the limit of x and y that we calculated. Finally, the concepts of `persp()` will be drawn in this viewport.

```
> testPersp21(box = FALSE)
> usr = par('usr')
> rect(usr[1], usr[3], usr[2], usr[4], lty = 12221, lwd = 2, border = 'red')
> usr
> ## [1] -0.4555925  0.3807924 -0.5003499  0.3360350
> grid.echo()
> downViewport('graphics-window-1-0');
> grid.rect(gp = gpar(col = 'red', lty = 12221))
> c(current.viewport()$xscale, current.viewport()$yscale)
> ## [1] -0.04  1.04 -0.04  1.04
```

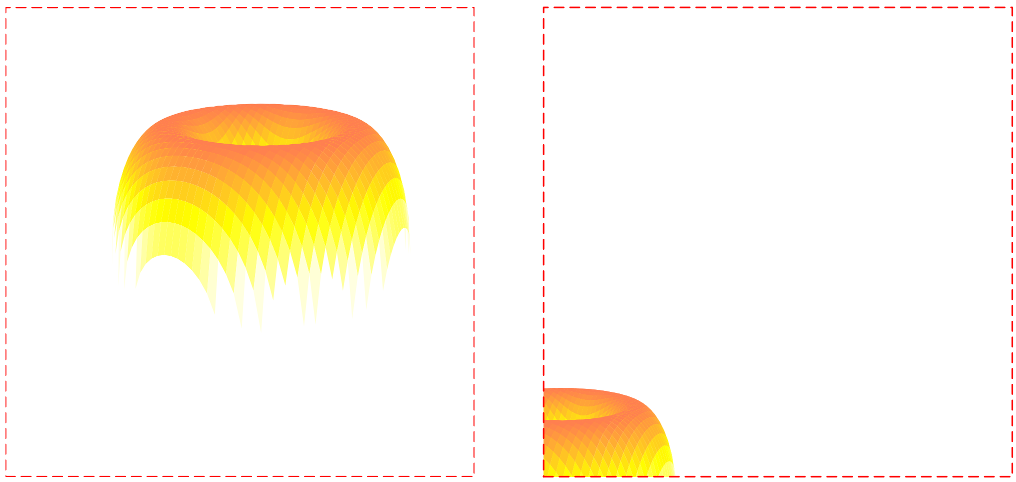

Figure 4.3: A Torus been drawn by `persp()` on the left plot, the right plot tried to reproduce the `persp()`, because of the scale of the viewport is different to the limit, the Torus is been drawn on the bottom-left corner.

Figure 4.3 shows a Torus drawn on the plot region. The red dotted rectangles indicate the plot region of `graphics`(left plot) and the viewport region for `grid`(right plot). Although the plot region is identical to the viewport region, the scales are different. The limit of x and y are $(-0.4555925, 0.3807924)$, $(-0.5003499, 0.3360350)$. The scale of x and y in the viewport region are $(-0.04, 1.04)$, $(-0.04, 1.04)$. Therefore, the scale needs to be moltified.

```
## navigate to the viewport
depth = gotovp(TRUE)
## calculate the limit of x and y
lim = PerspWindow(xr, yr, zr, trans, 'r')
## create another viewport by using the limit that we just calculated
vp = viewport(0.5, 0.5, 1, 1, default.units = 'npc',
              xscale = lim[1:2], yscale = lim[3:4])
## back to the Root viewport
upViewport(depth)
```

The code is creating a new temporary viewport which contains the true x scale and y scale prepared for drawing the perspective plot. The values for x scale and y scale are calculated by `PerspWindow()`. It will do the calculation by considering the 'actual' ratio of horizontal length and vertical length of the current graphics device, similar to the calculation of the `C` code does.
After we create the temporary viewport that contains the correct scale, then we added this viewport to the location of the tree which inside the viewport created by `gridGraphics`. That is, push temporary viewport inside the odd viewport. The Final step will be drawn the concepts within this viewport. To do that, we need to push a temporary viewport every time we drawn.
The following code is how does the surface of the plot is drawn internally.

```
## navigate to the viewport which has the true limit of x and y
depth = gotovp(FALSE)
pushViewport(vp)
## draw the surface inside the viewport
DrawFacets(...)
## back to the Root viewport
upViewport()
upViewport(depth)
```

The next problem will be the merge the temporary viewport into the `gridGraphics` viewport tree to make sure all the features (such as points and lines are added afther perspective plots is drawn) are drawn in the correct viewport. Although the scales have been fixed, other features have no information about the temporary viewport. In the other word, these features are drawn in the viewport that `gridGraphics` creates rather than drawn in the temporary viewport. Figure 4.4 shows a Rosette shape ring is drawn above the surface of tour(left-plot). The left-plot is tried to reproduce a `grid` version by `grid.echo()`. Although the tour is drawn in the correct location, the Rosette shape ring appear on the bottom-left region of the plot(right-plot).

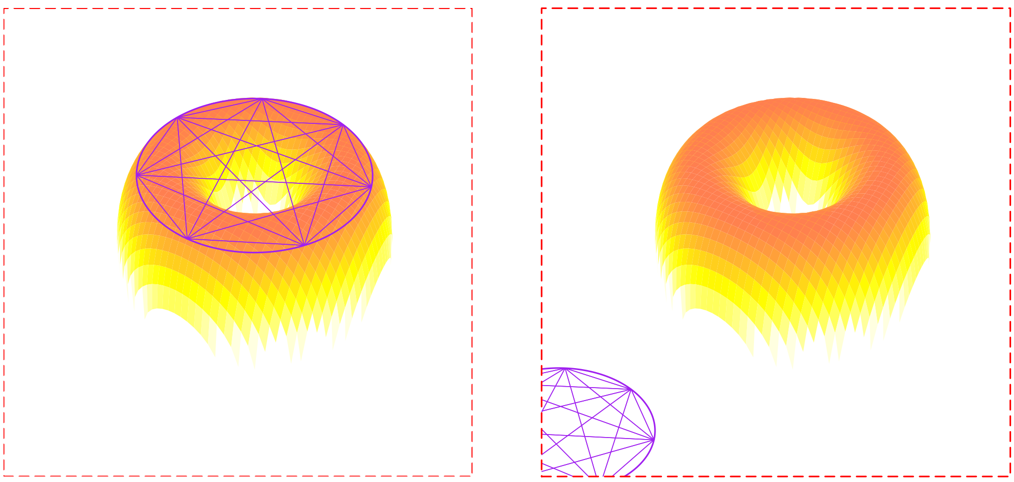

Figure 4.4: A Rosette shape ring added to the tour(left-plot) on `graphics`, the right-plot reproduced the left-plot by `grid,echo()`. Althought the tour are identical, the Rosette shape ring is on the incorrect location.

The final problem will be to decide whether allows the concepets are drawn outside the plot region or not . Figure 4.5 shows the previous tour surface draw over the limit of the box. The left-plot is drawn by `graphics` which is the behaviours that we need to reproduce on `grid`. In `persp()`, there are three concepts are been drawn, the surface of perspective plot, the box that contains the surface and the axes plus the labels of axes. By default, `graphics` draw the surface by setting `clipping = "on"`. On the other hand, the surface will not exceed the limit of the plot region. However, the box and the axes may be drawn outside the plot region if it is necessary. Alternatively, the right plot is drawn by `grid` which indicates compeletely result comparing with the left plot (the surface is drawn

over the plot region but the axes is not drawn outside the plot region). Due to the perpose of this paper, it is necessary to make sure the plots as identical as possible.
In `grid`, it is possible to define a viewport that either 'cut' the concepts if they are excess the limit of the viewprot, or continuous draw them outside the viewport by setting the `clip` equal to be `"on"` or `"off"`.
The solution will be distinguished the clipping region for the concpets. Alternatively, the surface cannot excess the limit of the plot region therefore we need to "cut" the surface if it excesses the limit. On the oter hand, we drawn the surface in the `clip = "on"`'s viewport. The box and the axes (include the labels and the units) are drawn in the `clip = "off"`'s viewport.
Since `gridGraphics` package already setted up the clipping region, therefore it can be solved by navigating to the specific viewport and draw the concept of `persp()`. The following code is the action of solving this problem.

```
...
## go to the viewport weather clip = 'off'
depth = gotovp(TRUE)
## draw the axes
PerspAxes(...)
upViewport(depth)
...
## go to the viewport weather clip = 'off'
depth = gotovp(TRUE)
## draw the Box
EdgeDone = PerspBox(0, xr, yr, zr, EdgeDone, trans, 1, lwd)
upViewport(depth)
...
## go to the viewport weather clip = 'on'
depth = gotovp(FALSE)
## draw the surface
DrawFacets(...)
upViewport(depth)
...
```

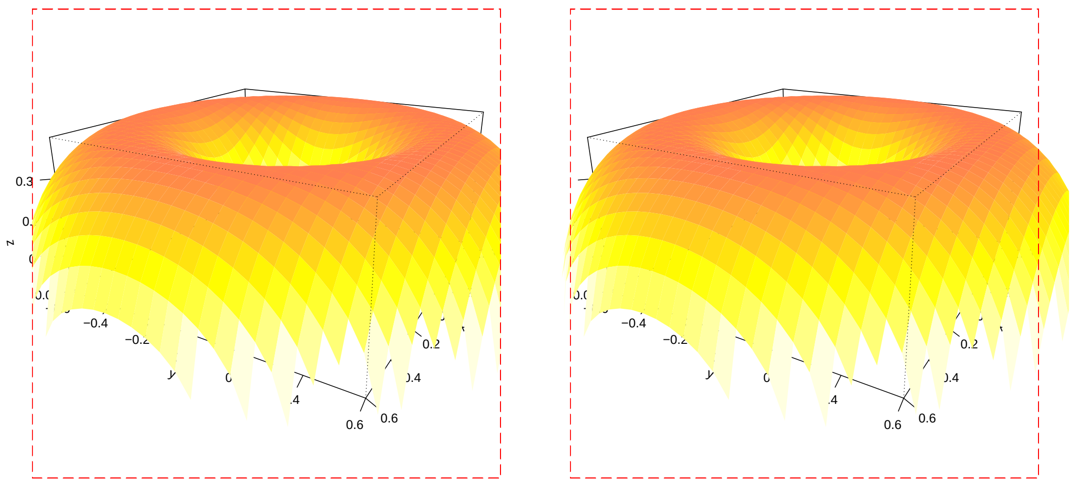

Figure 4.5: A `graphics persp()` is drawn at the left. The red regions are the plot region for `graphics`(left plot) and the viewport region for `grid`(right plot). Clearly, the right plot is tried to reproduce the left plot, but the surface is not allowed excess the limit of the viewport, but the axes label and units are needs to be drawn even excess the limit.

4.3 Integrate `filled.contour()`

Unlike `persp()`, `filled.contour()` is "made up" by multiple plots in one plot region. `gridGraphics` package will take care about most of the simple plots (such as the levels bar on the right-hand side of the plot, the titles, and the axes). (See figure 1.3 or figure 3.6).
`gridGraphics` fully convert the layout of `filled.contour()` to the viewport structure of `grid`, therefore we do not need to build or modify the viewports. However, it is necessary to "move" the filled contours into the correct location with the correct scale. On the other hand, we need to draw the filled contours in the correct viewport.
In section 3.2, we discussed how a Filled Contour Plot been drawn by the `filled.contour()`. More specifically, we only figure out who we drawn the main filled contour, but we still do not know how we display it. The next task is to display Filled Contour Plot in the correct location. Figure 4.6 shows the top-left is the contents of `filled.contour()`, which redrawn by using `grid` package. The red dotted rectangle is the viewport region. The next step is fill the blank region (top-right plot) by the top-left plot. The solution is similar to the first step of integrating `persp()`, that is, navigating to the correct viewport region and then drawn the filled contour. The following code is the solution in action.

```
## navigating to the correct viewport
depth = gotovp(TRUE)
## actual drawing
grid.polygon(...)
## reset
upViewport(depth)
```

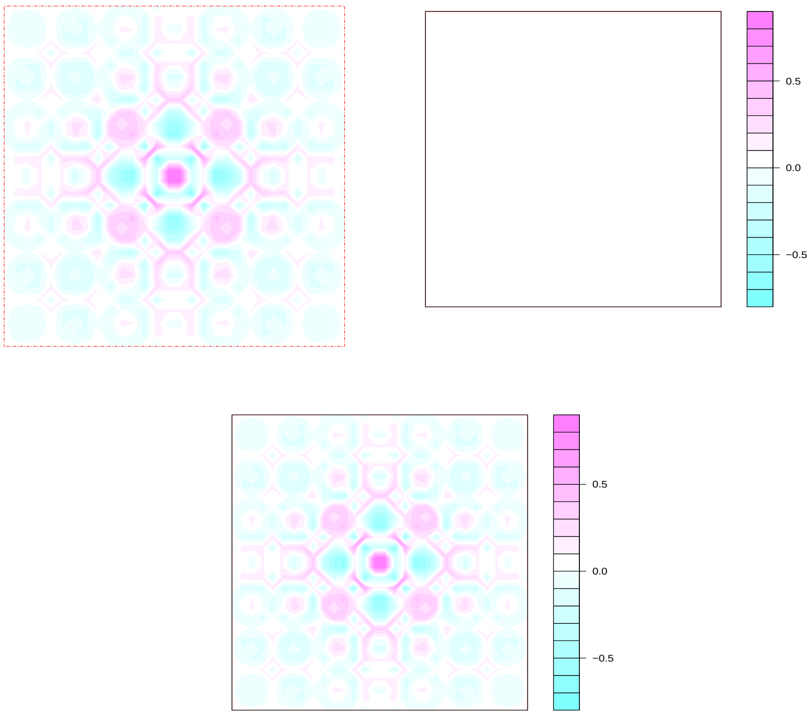

Figure 4.6: The top-left filled contour are plotted by `grid`, top-right plot is the failed reproduce the `filled.contour()` by `grid.echo()` at the original state, and the bottom-center plot is the success reproduce the `filled.contour()` by `grid.echo()` after integrated to the `gridGraphics` package.

Chapter 5

Testing

The aim of this paper is to redraw Perspective Plots and Level (Contour) Plots using the `grid` package with an identical result to the `graphics` package. Every plot drawn by `grid` graphics should be not only identical to `graphics` by human eye, but also the machine 'thinks' they are identical.

However, there are some tiny differences which cannot distinguish by human eye, for example, the color are differences at one pixel between two plots, or the colors of few area have one unit difference on red Channel distortion compare to the other plot. Therefore, we need software to detect those tiny differences.

ImageMagick is the software that can be used for the comparison in this paper, it can create, edit, compose, or convert bitmap images and read and write images in a variety of formats (over 200). The features used in this paper is `compare`, which is a program to mathematically and visually annotate the difference between an image and its reconstruction.

The following example (see figure 5.1) are drawn two rotated sinc function, where the colors of the surface are close to each other and cannot be distinguish by human eyes. However, there is a difference for the colors, the top-left plot ($rgb = 211, 182, 255$) has one color pixel higher on red channel than the right-plot ($rgb = 210, 182, 255$). The bottom plot shows the difference, the region filled by red color is the difference, which is true because the color of the surface is different, the box and labels which have lighter color because they are mathematically and visually identical to each other.

The following figure shows the test that built in the `gridGraphics` package. In this section, we only show the different plot. The `graphics` and `grid` plots are in ??.

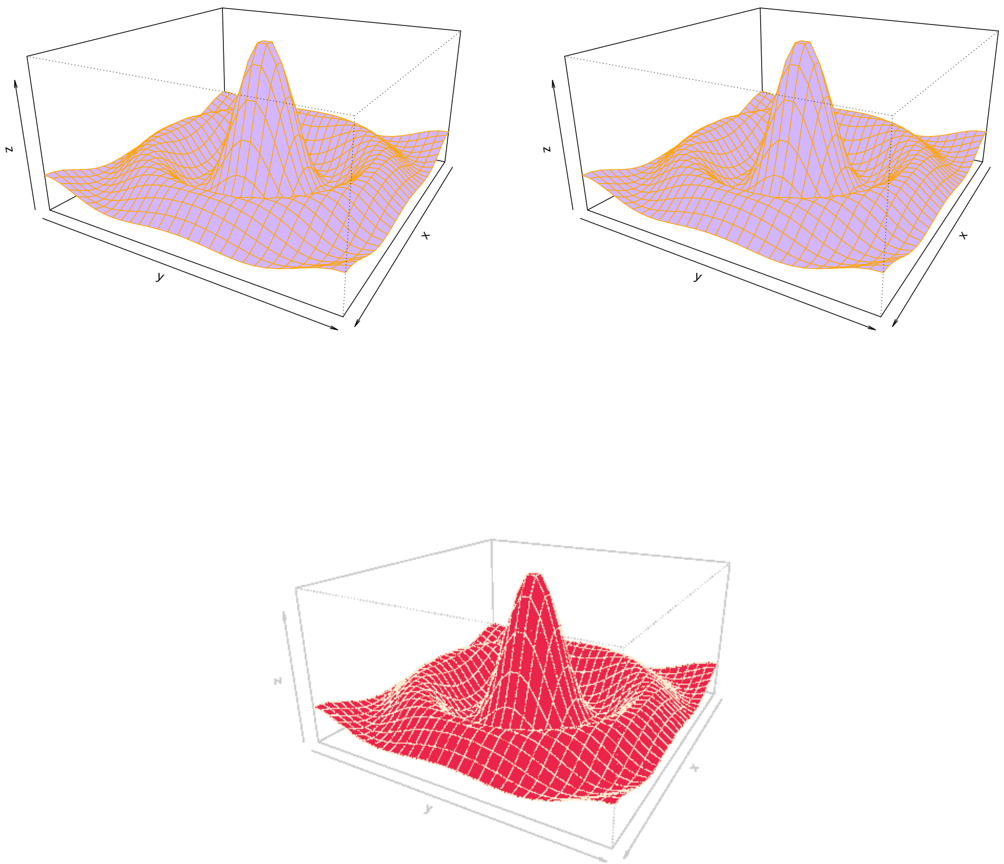

Figure 5.1: The top-left plot looks identical to the plot at top-right by human eyes, however, there is a tiny difference at the color of the surface, which been detected by the software, the red color indicated the difference.

Chapter 6

Example

persp() and filled.contour() are now supported by gridGraphics. This means most of the plots that drawn by persp() and filled.contour() by graphics package now is able to reproduce by grid.echo() on grid. The advantage of grid is grid is more flexible than graphics. For example, a plot drawn by grid can record the viewport structure and we may draw and edit any plot features within different viewport easily.

6.1 Example to solution

The persp() and filled.contour() are now can been edit more easily since they are integraded to gridGraphics. On the other hand, they are now available been drawn on grid.

An other Torus shape is drawn by graphics (See figure 6.1), it has been redrawn on grid by grid.echo() and then we can listing the grobs of this plot. Since we are only drawing the surface, and the surface is formed by polygons. Then we can modify the features of the polygons easily by using grid.edit. In this example, we changed the white color (with opacity is 0) to purple (with opacity is 0.3).

```
> Torus_shape(col = 'NA', border = 'gray', box = FALSE)
> grid.echo()
> grid.ls()
> newCol = rgb(160/255, 32/255, 240/255, alpha = 0.3)
> grid.edit('polygon', gp = gpar(fill = newCol))
```

```
> f8()
> grid.echo()
> poly.obj = grid.get(grid.ls(print = FALSE)$name[3])
> rect.obj = grid.get(grid.ls(print = FALSE)$name[1])
> colpoly = colorspace::desaturate(poly.obj$gp$fill)
> colrect = colorspace::desaturate(rect.obj$gp$fill)
> ## modify the colors
> grid.edit(grid.ls(print = FALSE)$name[3], gp = gpar(fill = colpoly))
> grid.edit(grid.ls(print = FALSE)$name[1], gp = gpar(fill = colrect))
```

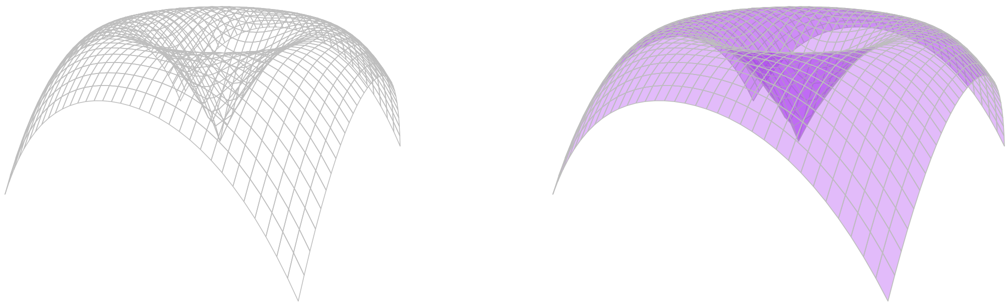

Figure 6.1: The left plot is drawn by `graphics`, where the right plot is been drawn by redrawn the left plot on grid (`grid.echo()`) and then moltify the colors by `grid.edit()`

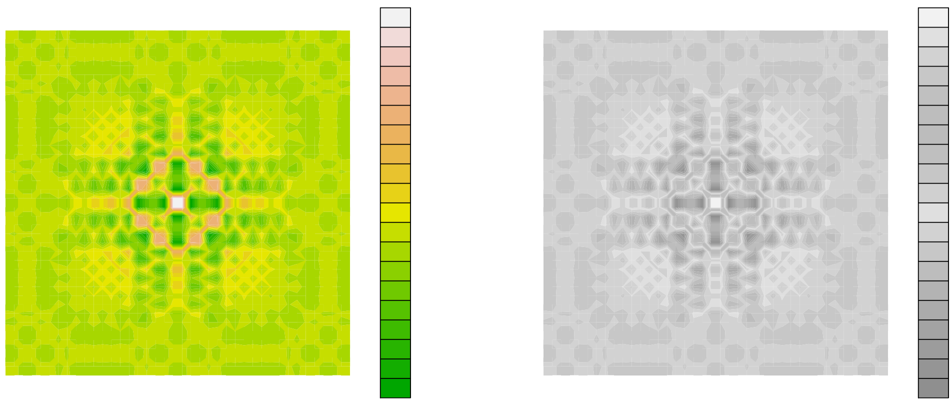

Figure 6.2: The left plot is drawn by `graphics`, where the right plot is been drawn by redrawn the left plot on grid (`grid.echo()`) and then modify the colors by `grid.edit()`

6.2 An advance example

In the `graphics` package it is very normal to draw two or more plots in the plot region. However, it is not allowable to combine the `filled.contour()` with any other kind of plots together within one plot region, because `filled.contour()` will create the layout itself when it is called. Therefore, even if we specify a layout before plotting, the `filled.contour()` will overwrite the layout.

The viewport structures of `grid` provides more flexibility than the layout of `graphics`, which allows us draw any other plots with `filled.contour()` within one page. The idea is to create a viewport before drawing any plots (said root viewport), then push a viewport within the Root viewport for drawing a grid version of `filled.contour()` so that the `filled.contour()` will only modify the structure of its own viewport but not overwrite the Root viewport.

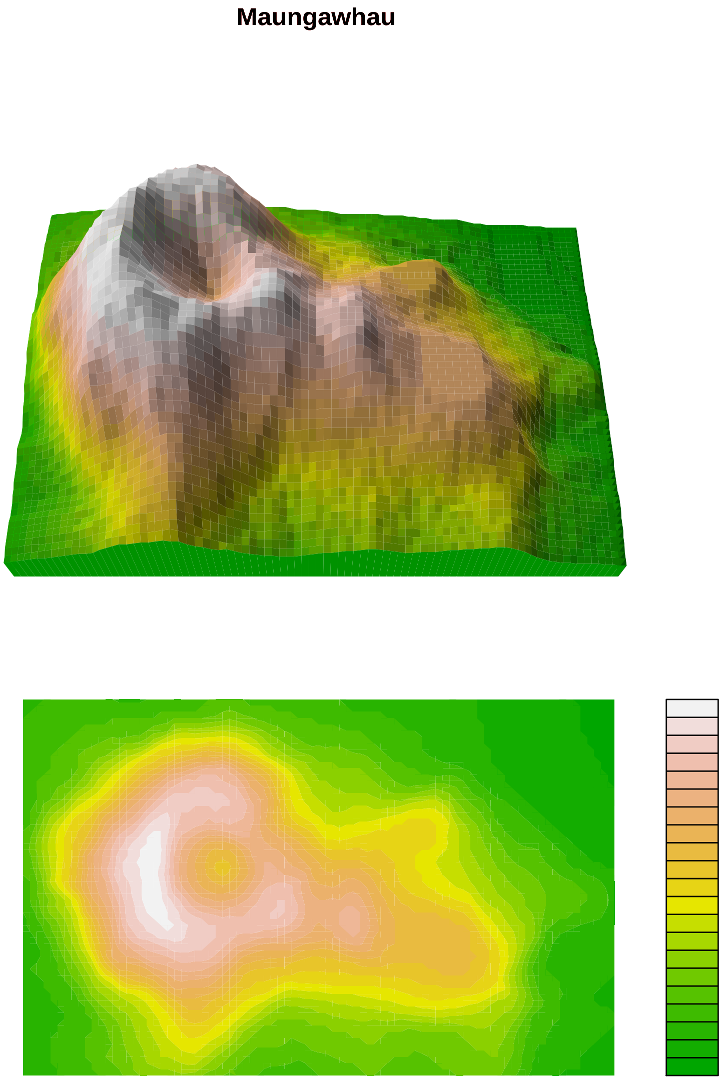

Figure 6.3: The top plot shows the shape of Maungawhau volcano, where the bottom figure shows the level contour (i.e. the height)

6.3 A more advance example

6.3.1 Basic concepts

Another benefit that `grid` provides is we can convert the graphics drawn with package `grid` to SVG format by the `gridSVG` package.
SVG (Scalable Vector Graphics) is a two-dimensional vector image format with XML-based. A SVG format graphics can support for interactivity and animation which most of the `R` graphics system is not supported or it is difficult achieve by `R` graphics system.

```
<circle id = "circle1"
        cx="50" cy="50" r="40"
        stroke="black" stroke-width="3"
         fill ="red" />
```

The pervious codes created the circle filled with red color (See figure 6.4), where the radius is 40units. As we see, the color of the circle is defined on the `circle` element which is the `fill = "red"` in it's attribute. Inorder to modity the color of the circle, the first step to find the `circle` element. The element contains a `id`, its value is `"circle1"`.

The following codes is coded with `JavaScript`. It interactives with the SVG image that changes the color of the circle from red to yellow, when a mouse moves into the circle region. The key step of this interactivity will be track the circle element and modify its color.

The method for tracking the `circle` element is to find the elements where its `id` attribute is `"circle1"`. Therefore, the `getElementById()` method is used. It will return the element that has the ID attribute with the specified value. After the circle element is tracked, we can simply add a `onmouseover` Even to it. That is, the color of the circle will change only if a mouse moving into the circle. The `Onmove` function is the action when the even is activated.

```
Onmove = function(){
     this.setAttribute('fill', "yellow");
}
var Oncircle;
Oncircle = document.getElementById("circle1");
Oncircle.onmouseover = Onmove;
```

Figure 6.4: The left circle is created with svg imgae, the right circle is changed color to yellow then the mouse enter the region of the circle.

6.3.2 The sinc surface

Since those action cannot been achieved by `R` graphics system. The first step is to export the plot into SVG image. To do that, the `gridSVG` is required. The following codes are used for export the SVG image (See figure 6.5). `surface()` is the 'wrap' function for drawing the sinc surface perspective plot. `addFeatures()` is creating the texts and the little boxes of color at the top-left region in the plot. It also creates some hidden objects which are all invisible unless the specific action been actived. A javascript is imported for supporting the animations by using `grid.script()`. Finally we exported the SVG image by `grid.export()`

```
## draw a surface
surface()
## add the button at the top−left of the plot prepare for animation
addFeatures()
## import the javascript file into the SVG image
grid.script( file =”example.js”)
## export the SVG image
grid.export(”example.svg”, strict = FALSE)
```

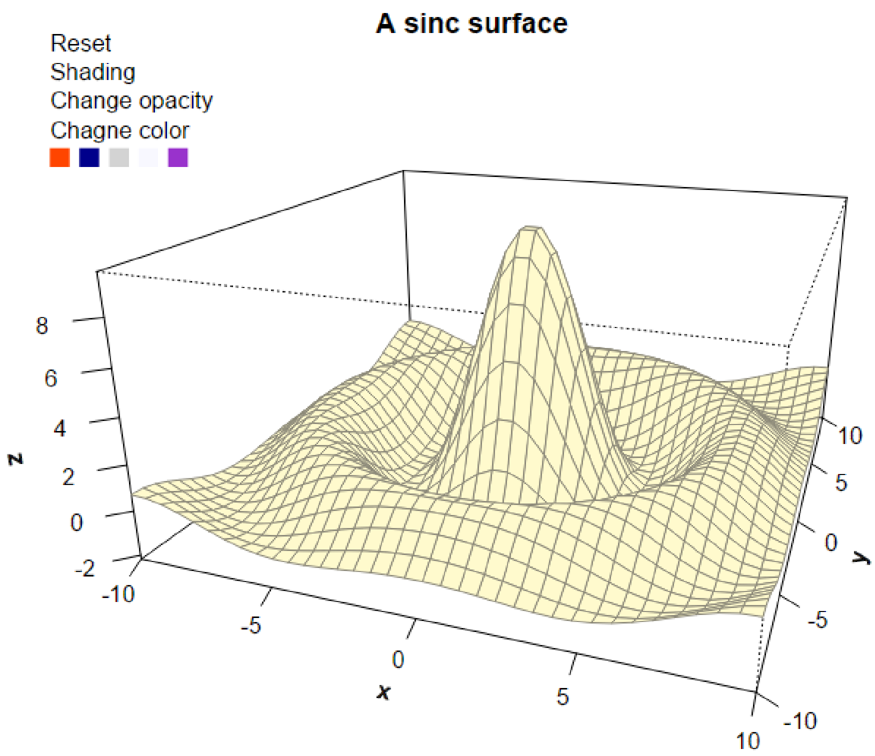

Figure 6.5: A SVG image of the sinc surface perspective plot

In this example, we are going through the animation of the following action.

- Changing the color of the surface
- Changing the opacity of the surface
- Shading the surface
- Highlighting the specific fragment of the surface and showing its z-value

Animation of Changing the color of the surface

Simliar to the circle example, we change the color of the surface by locating the elements that contain the surface then modify the color. However, the complicate task will be create the animation. The animation of `JavaScript` are done by programming gradual change in elements' style. The animation will look continuous when the time between every changes are small.
The following code is key step for creating the animation (See figure 6.6).The `animate()` can produce

all the action that we disscussed previously. The `frame()` in the `animate()` will change the color of the surface gradually every time it been called. The `setInterval(frame, 30)` will call the `frame()` every 30 milliseconds. On the other than, the color of the surface will be changed gradually every 30 milliseconds. This action will be actived when user clicks the colored boxes.

```
animate = function(...){
   ...
   setInterval(frame, 30)
   function frame() {
      ...
   }
  ...
}

color_fill = function(){
     reset();
     color_new = this.getAttribute('fill');
     animate(..., action = 'color', colors = color_new);
}
 ...
color_id = document.getElementById('color.2.' + 1)
color_id.onclick = color_fill
```

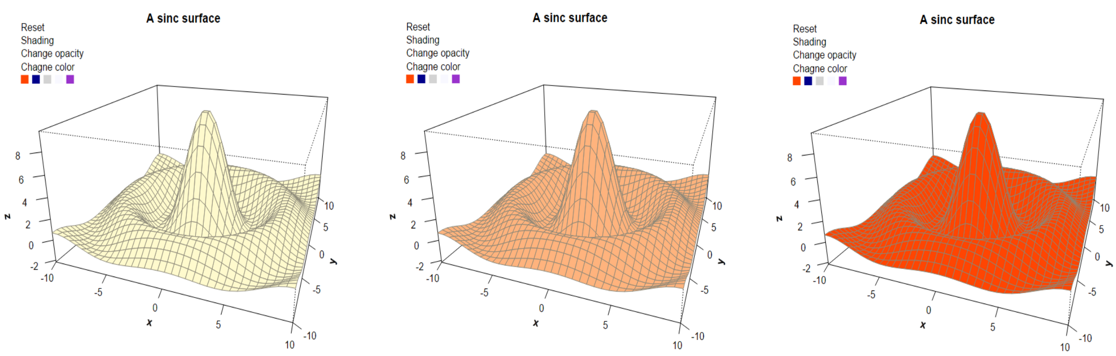

Figure 6.6: The plot shows the animation that changing the surface color from the left plot to right plot

Animation of Changing the opacity of the surface

Simliar to the animation of Changing the color of the surface, the opacity attribute of the surface will be change in this example. The animation will be actived when user clicks the "Change opacity" text on the top-left corner of figure 6.5. Once user clicked the text, the `addAlpha()` will call the `animate()` which it locates all the polygons and produces gradual change of the stroke opacity in every 30 milliseconds. (See figure 6.7)

```
animate = function(...){
  ...
  setInterval(frame, 30)
  function frame() {
     ...
     polygons_odd[i].setAttribute("stroke-opacity", ...);
  }
 ...
}
```

```
addAlpha = function(){
    animate(...,  action = 'alpha',  ...) ;
}
 ...
alpha = document.getElementById('alpha.1.1.text');
alpha.onclick  = addAlpha;
```

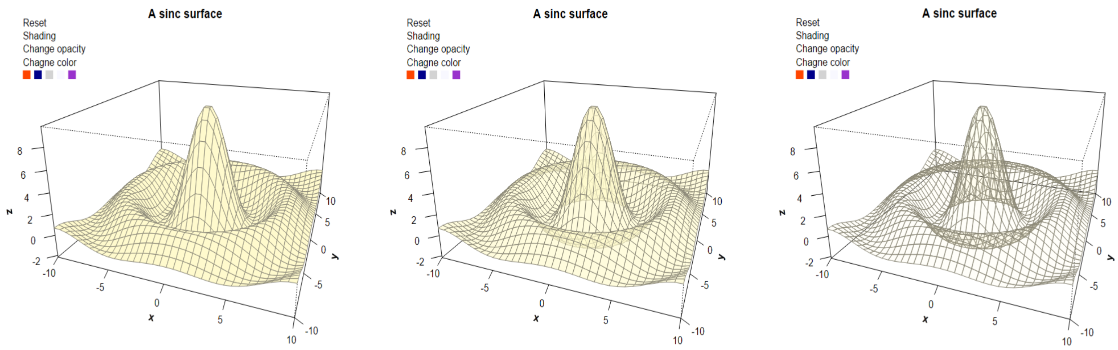

Figure 6.7: The plot shows the animation that decreasing the opacity of sinc surface from the left plot to right plot

Animation of shading the surface

The figure 6.8 shows the action of adding shade of the current sinc surface. The middle plot is captured during the duration.

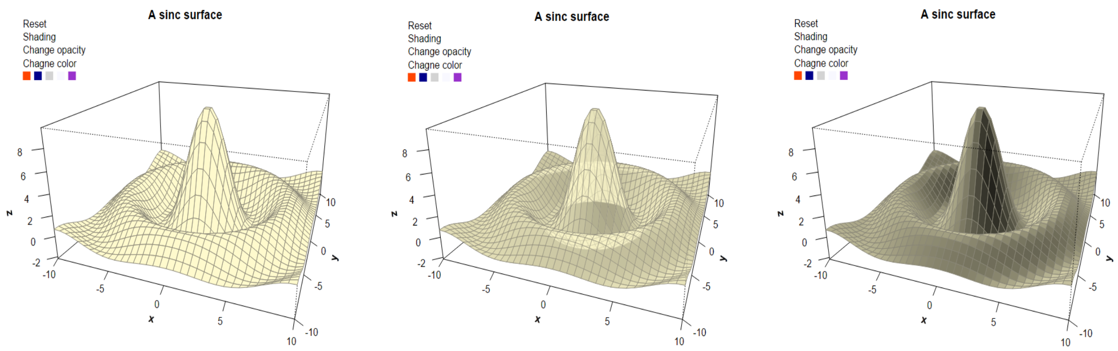

Figure 6.8: The shading animation

Highlighting the specific fragment of the surface and showing its z-value

The action of this example will be slightly difference to the previous example. There two actions need to be actived at the same time when user moves the mouse into a specific fragments of the sinc surface. One is highlight the fragment and the other one is display the z-value of itself. When user moves the mouse left the fragment, the color needs reset and the z-value needs to be hided.
The trick to perform this action is to display the correct z-value for the specific polygon. The svg we created is not only a sinc surface and the action text, but also a set of z values, located at the top-right corner of the box. Since their opacity are 0 hence they are invisible at the beginning.
First to all, we need to add the `onmouseover` and the `onmouseout` events to all fragments. That is, the functions `polygonon()` and `polygonout()` are actived when the mouse enter and leave the regon

of the specific fragments.
There are two action will happen when the mouse enter the fragments. One is highlighting the fragment and the other is display its z-value. The highlighting action can be done simply by changing the color of the fragments. The z-value is displayed by macthing the correct index. That is, the action funtion `polygonon()` will pull out the index of the id attribute from the specific polygon element, and then it will find the label element by matching this index. Finally the z-value will be diaplayed by setting the opacity of the label element to be 1.
When the mouse leave the fragment, it is necessary to reset the color of this fragment and hide the z-value. These actions will be done by `polygonout()`.

```
for(i = 1; i <= total; i++){
    obj = document.getElementById('polygon.' + 2 + '.' + i);
    obj.onmouseover = polygonon;
    obj.onmouseout = polygonout;
}
polygonon = function(){
    str = this.id;
    polygon_index = str.replace(/polygon.[0-9]./, '');
    // highlight the polygon
    this.setAttribute('fill', "rgb(255,100,100)");
    // show the 'value'
    label = document.getElementById('labels.' + '1.' + polygon_index + '.text');
    label.setAttribute("fill-opacity",1);
}
polygonout = function(){
    str = this.id;
    polygon_index = str.replace(/polygon.[0-9]./, '');
    // hide the label
    label = document.getElementById('labels.' + '1.' + polygon_index + '.text');
    label.setAttribute("fill-opacity",0);
    opacity = this.getAttribute('fill-opacity');
    // rest the color
    this.setAttribute("fill",window.new_rgb);
}
```

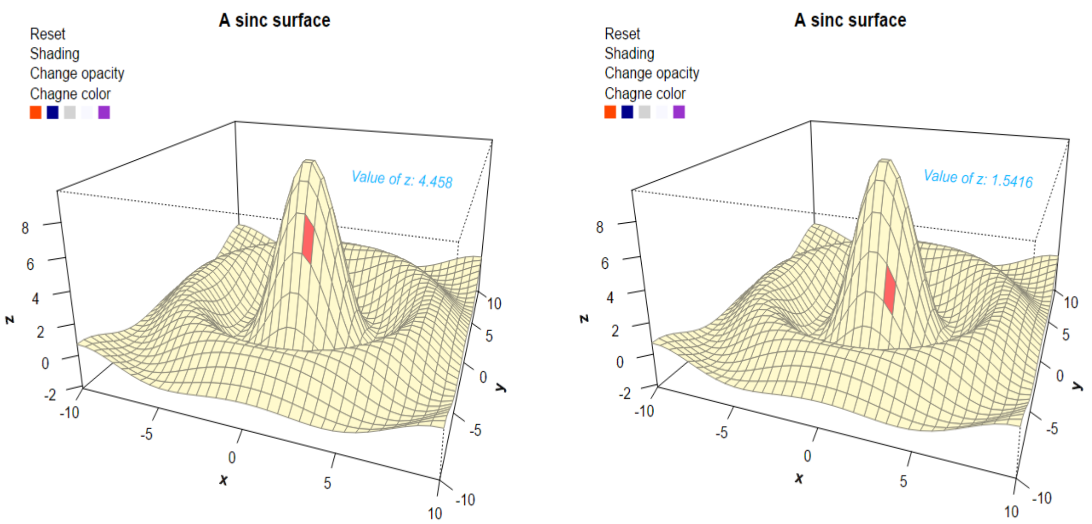

Figure 6.9: The fragment will be highlight when the mouse move into its area. Also its z-value will be appeared

Chapter 7

Conclusion

Chapter 8

Reference

Chapter 9

Appendix

9.1 persp.R

9.2 filled.contour.R