Data Science with R

Part VIII: Graphics

Raphael Schleutker

The essential test of design is how well it assists the understanding of the content, not how stylish it is. — Edward Tufte

Table of contents

General info

R has two graphics systems: The traditional system provided by the `graphics` package and grid graphics provided by the `grid` package that is, for instance, used by `lattice` and `ggplot2`. Today we will focus on traditional graphics.

We will first see how graphics are composed and later see some high-level functions.

Basics

For a proper understanding of traditional graphics we have to know that graphics functions usually do not return a value. If at all they return the plotted data invisibly. Instead the graphical output is redirected to something that R calls devices.

Such a device can be the screen (default). In this case a window is opened to which the graphical output goes. But a device can also be a file like PDF, PNG, or TIFF.

When we open, for instance, a PDF device, all the graphical output of our graphics functions goes to that device (the file). Therefor we do not see the result on screen.

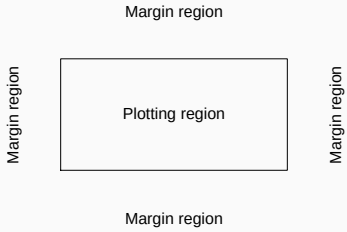
Basics

For the beginning we will stay with the default device. This allows us to see the output directly on the screen.

When producing a new plot the first function to call is `plot.new`. If we have already produced a plot, this causes completion of that plot and creates a new plot. That plot only consists of an empty canvas.

This canvas is divided in two (yet invisible) regions:

- The plotting region, in which the actual plotted data appears.
- The margin region, in which ticks and axis labels appear.



Basics

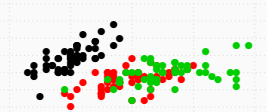
Initially, this plot lacks any coordinate system. We have to set it up using `plot.window`, which takes the limits for the x and y-axis and if the axis should be log-scaled. This allows us to put content in the next step cause now R knows were to place that data on the canvas. For instance we can add a grid for a better orientation.

```
plot.new()
plot.window(xlim = c(0,1), ylim = c(0,1))

grid() # Draw grid in plotting region.
```

Basics

R has many low-level graphical functions. We can add, for example, points for the sepal length and width of the iris dataset.

```
plot.new()
plot.window(xlim = c(4, 8), ylim = c(2,5))

grid() # Draw grid in plotting region.
points(iris$Sepal.Length, iris$Sepal.Width, pch = 16, col = iris$Species)
```

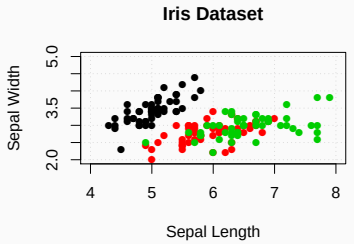


There are more low-level functions like `points`, e.g. `lines`, `polygon`, and `rect`. All of them take parameters that further specify their look.

For instance, the `col` argument of `points` takes a vector the same length as the x- and y-coordinates. The first element is matched with the first pair of coordinates and the point colored accordingly. The `pch` argument specifies the plotting character, i.e. the shape of the point. `16` means filled circle.

We can now proceed and add common components of a plot.

```
plot.new()
plot.window(xlim = c(4, 8), ylim = c(2,5))

grid() # Draw grid in plotting region.
points(iris$Sepal.Length, iris$Sepal.Width, pch = 16, col = iris$Species)
axis(1) # x-axis
axis(2) # y-axis
box() # Border around plotting region.
title(xlab = "Sepal Length", ylab = "Sepal Width", main = "Iris Dataset")
```



Using such low-level functions gives a very good idea about how traditional graphics are composed in R and using them is a very mighty tool to build beautiful plots.

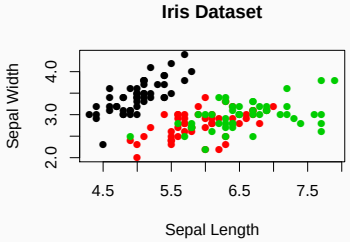
However, working with low-level functions is tedious. Therefor, R has many functions that bundle all of the previous things.

High-level functions

High-level functions

The previous plot can easily be created without calling all the low-level functions on your own.

```
plot(
    iris$Sepal.Length,
    iris$Sepal.Width,
    col = iris$Species,
    xlab = "Sepal Length", ylab = "Sepal Width",
    main = "Iris Dataset",
    pch = 16
)
```

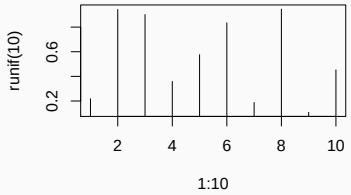


The `plot` function detects that it gets two numeric arguments and interprets them as coordinates. By default, it plots points. But we can also take another plotting type.

High-level functions

The previous plot can easily be created without calling all the low-level functions on your own.

```
plot(1:10, runif(10), type = "h")
```

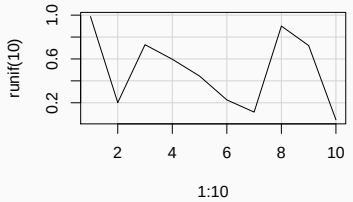


The generic axis labels are usually not very useful.

High-level functions

By default, the plot lacks a grid. We can add a grid with `grid()`, of course, but this puts the grid over the plot.

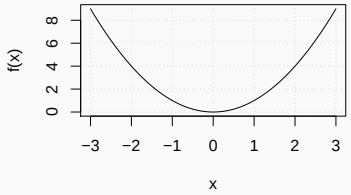
```
plot(
    1:10,
    runif(10),
    type = "l",
    panel.first = grid(lty = "solid") # Calls grid before plot is generated.
)
```



High-level functions

There are also even more abstract high-level functions like `curve`.

```
f <- function(x) x^2

curve(f, -3, 3, panel.first = grid())
```

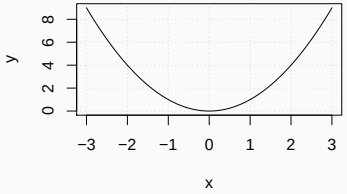


`curve` takes a function and creates a specified number of evenly spaced values for the definition interval. It then calls the function with these values and uses `lines` to create the function plot.

High-level functions

This can also be done manually.

```
x <- seq(-3, 3, length.out = 101) # Create 101 evenly spaced values.
y <- f(x) # Create the function values.

plot(x, y, type = "l", panel.first = grid())
```

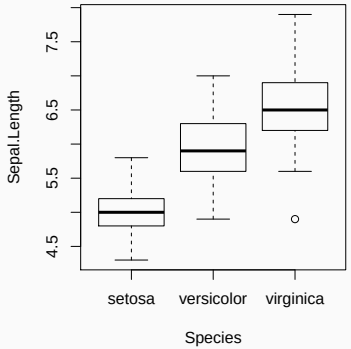


High-level functions

The graphics functions often work with the formula interface.

```
par(mar = c(5.1, 4.1, 1.1, 2.1))
```

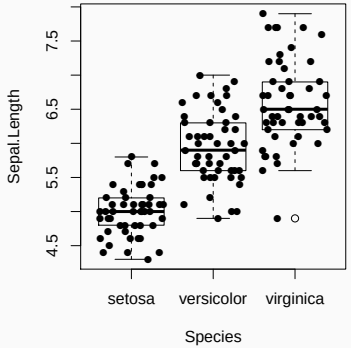
```
boxplot(Sepal.Length ~ Species, data = iris)
```



High-level functions

High-level functions create new plots, i.e. they call `plot.new`. Sometime, they take the argument `add` that can be set to `TRUE`. Then the output is added to the existing plot. Low-level functions are added by default.

```
x <- rep(c(1, 2, 3), each = 50) + runif(150, -0.4, 0.4)
boxplot(Sepal.Length ~ Species, data = iris)
points(x, iris$Sepal.Length, pch = 16)
```



Colors

Colors

R has a default color palette that we can view by using the `palette` function.

```
palette()

# [1] "black"   "red"     "green3"  "blue"
# [5] "cyan"    "magenta" "yellow"  "gray"
```

These colors are not very nice even though they will improve with the release of R 4.0.0.

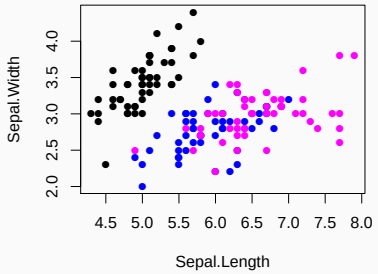
Colors

When using colors in a plot R uses the value handed over to the `col` argument. If it is a string, it will use different colors for different strings. If it is numeric, it will use the value as an index for the different colors, i.e. 3 corresponds to `green3` in the above palette.

Colors

We can modify the palette by giving the `palette` function a character string with either color names or hexadecimal representations of colors.

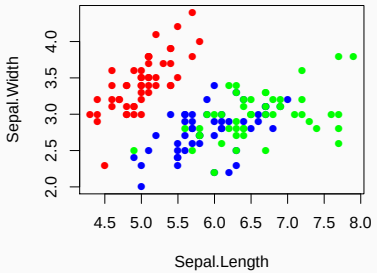
```
palette(c("black", "blue", "magenta"))
palette()

# [1] "black"   "blue"    "magenta"

plot(Sepal.Width ~ Sepal.Length, data = iris, col = Species, pch = 16)
```



Colors

Alternatively, we can hand over the colors for the plot manually. In this case, R will not use the standard palette by uses the colors directly.

```
plot(
    Sepal.Width ~ Sepal.Length, data = iris,
    col = rep(c("red", "blue", "green"), each = 50),
    pch = 16
)
```



Colors

At the time of writing, R knows 657 different colors by name. You can watch them using `colors()`.

Setting the palette to nice colors is fine for categorical values but sometimes we have numerical values and want to color them according to their values.

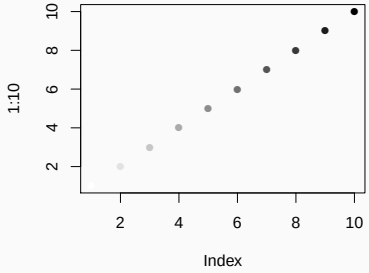
Colors

For this, R has two functions called `colorRamp` and `colorRampPalette`. Both take a character vector with colors and return a function, that interpolates colors (yes, functions can return other functions). The function returned by `colorRampPalette` takes an integer and returns the same amount of colors interpolated from the given ones. The function returned by `colorRamp` takes a value between 0 and 1 and returns the according color value between the specified ones.

Colors

For example, we want to create shades of gray between white and black.

```
greyRamp <- colorRampPalette(c("white", "black")) # Create the function.
greyRamp(10) # Call the function returned by colorRampPalette.


#  [1] "#FFFFFF" "#E2E2E2" "#C6C6C6" "#AAAAAA"
#  [5] "#8D8D8D" "#717171" "#555555" "#383838"
#  [9] "#1C1C1C" "#000000"


plot(1:10, pch = 16, col = greyRamp(10))
```



In the above example, we specified the color for each point by position, i.e. the first point becomes white and the last one black. If we want to color a point according to a value, we have to use `colorRamp` but the usage is slightly more difficult.

```
greyRamp <- colorRamp(c("white", "black"))
greyRamp(0.5)


#       [,1]  [,2]  [,3]
# [1,] 127.5 127.5 127.5
```

Colors

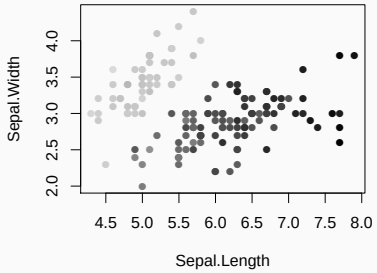
The function returned by `colorRamp` takes a value between 0 and 1 and returnes RGB values between 0 and 255. For 0.5 these are the RGB values for a grey in the center between white and black. `rgb` converts these values to a string that we can use.

```
rgb(greyRamp(0.5), maxColorValue = 255)

# [1] "#7F7F7F"
```

Let's see how we can use this for a plot.

Colors

```
plot(
  Sepal.Width ~ Sepal.Length, data = iris,
  col = rgb(
    greyRamp(with(iris, Petal.Length / max(Petal.Length))),
    maxColorValue = 255
  ),
  pch = 16
)
```



Colors

We can also provide several colors and interpolate color palettes from them. Some nice initial colors are provided by the package `RColorBrewer`.

```
library(RColorBrewer)

brewer.pal(5, "Spectral")

# [1] "#D7191C" "#FDAE61" "#FFFFBF" "#ABDDA4"
# [5] "#2B83BA"
```

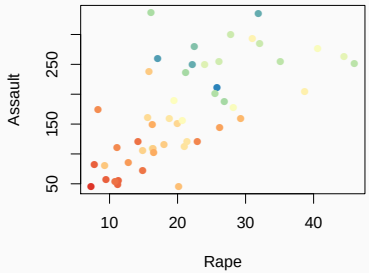
Colors

We can also provide several colors and interpolate color palettes from them. Some nice initial colors are provided by the package `RColorBrewer`.

```
library(RColorBrewer)

brewer.pal(5, "Spectral")

# [1] "#D7191C" "#FDAE61" "#FFFFBF" "#ABDDA4"
# [5] "#2B83BA"

spectralRamp <- colorRamp(brewer.pal(5, "Spectral"))
```

Colors

```
normalized_murder <- with(USArrests, Murder / max(Murder))

plot(
  Assault ~ Rape, data = USArrests,
  col = rgb(spectralRamp(normalized_murder), maxColorValue = 255),
  pch = 16
)
```



Apparently, the number of rapes, assaults and murders correlate with each other.

Graphical parameters

Graphical parameters

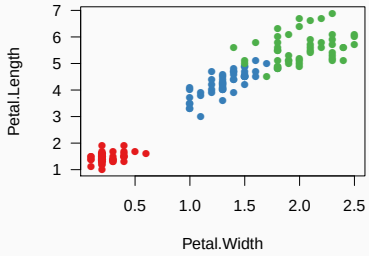
R has a global list of graphical parameters that defines how a plot looks. For example, we can define the size of the margins, the size of text, the orientation of tick labels, etc. The interface to these parameters is provided by `par`. When giving a string with the name of a graphical parameter, it returns the current value. But we can also use it to modify these values.

```
par("mar") # Size of margins in lines of text.

# [1] 5.1 4.1 4.1 2.1

par(mar = c(5.1, 4.1, 1.1, 1.1))
par("mar")

# [1] 5.1 4.1 1.1 1.1
```

Graphical parameters

There are dozens of different parameters that we can modify. Read the help page for `par` to get an idea. In the following we will see some common ones.

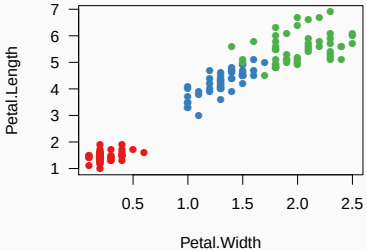
```
palette(brewer.pal(3, "Set1")) # Select a palette from RColorBrewer
par(las = 1) # Make all tick labels horizontal

plot(Petal.Length ~ Petal.Width, data = iris, col = Species, pch = 16)
```



Graphical parameters

Using par to set graphical parameters affects all plots. For many parameters it is possible to provide them for the plotting function (this is, what the ... argument is used for). This only affects the current plot.

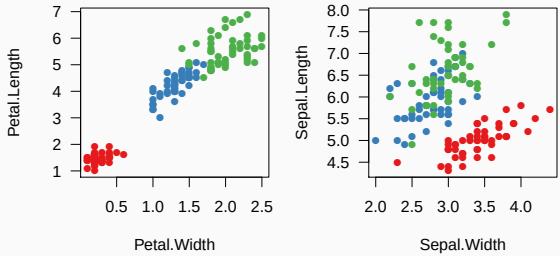
```r
par(las = 0) # Tick labels parallel to axis.

plot(
  Petal.Length ~ Petal.Width, data = iris,
  col = Species,
  pch = 16,
  las = 1
)
```



Graphical parameters

A very useful parameter is `mfrow`, which allows us to create several plots on one page.

```
par(las = 1, mfrow = c(1, 2)) # 1 row and 2 columns

plot(Petal.Length ~ Petal.Width, data = iris, col = Species, pch = 16)
plot(Sepal.Length ~ Sepal.Width, data = iris, col = Species, pch = 16)
```



Please note that the page is always split equally and each subplot is a complete plot with margin and plotting regions. Therefor, it is not possible to create facets as you may know them from ggplot2.

Devices

Devices

By default R outputs the graphical output to a window, in which we can then see the result. Usually, we want to save the plot as a file for later use. In this case we have to open a new device that is not a window but a file. For instance:

```
pdf("./output/iris.pdf", pointsize = 9, width = 100/25, height = 70 / 25)

par(mar = c(5.1, 4.1, 2.1, 2.1))

plot(
  Sepal.Length ~ Sepal.Width,
  data = iris,
  col = Species,
  pch = 16, las = 1
)

dev.off() # IMPORTANT! Close device.


# pdf
#   2
```

Devices

Since the graphical output is sent to the file we do not see it in the window.

Plots stored as PDFs are stored as vector graphics. When saving them as raster graphics we can also provide the resolution.

Devices

```
tiff(
  "./output/iris.tiff",
  pointsize = 9,
  width = 100, height = 70, units = "mm",
  res = 300  # 300 dpi
)

par(mar = c(5.1, 4.1, 2.1, 2.1))

plot(
  Sepal.Length ~ Sepal.Width,
  data = iris,
  col = Species,
  pch = 16, las = 1
)

dev.off() # IMPORTANT! Close device.


# pdf
#   2
```

Devices

How can you make sure that the text in your plots always appear as the same size in your reports?

1. Determine the final width and height of your figure on the paper in mm.
2. Open a graphics device and specify your measured width and height. Set the pointsize to 8 or 9.
3. Specify the dpi as required by the journal.

Never increase width or height to get more pixels in your final file. This makes the text appearing smaller and you would have to adjust that again.