Manipulação de Dados em R

Luís Torgo
FEP, Universidade do Porto
ltorgo@liacc.up.pt

13 de Dezembro de 2006

O objectivo principal deste material é introduzir os alunos a diversas formas de manipular dados em R. Nomeadamente, iremos aprender formas de importar dados de outras aplicações para o R, sumariar dados e também técnicas de visualização de dados em R.

1. Importar Dados para o R

1.1. Importar de ficheiros de texto

Os ficheiros de texto são uma das formas mais comum de armazenar dados.

Para além disso, existem ainda diversas plataformas que permitem exportar dados armazenados nas suas estruturas próprias, para ficheiros de texto em formatos particulares, como por exemplo o formato CSV.

Esta é muitas vezes a forma mais simples de importar para o R os dados de uma aplicação qualquer, que use um formato mais fora de vulgar.

O formato mais comum de armazenar uma tabela de dados num ficheiro de texto consiste em pôr cada linha da tabela de dados numa linha separada do ficheiro, e fazer separar os valores de cada coluna de uma linha, por um qualquer caracter separador.

Relativamente a este último, escolhas comuns são a vírgula (levando ao formato CSV, *Comma Separated Values*)), o ponto e vírgula, ou o caracter *tab*.

A função principal do R para ler este tipo de ficheiros é a função `read.table()`.

Um dos parâmetros principais é o parâmetro *sep* usado para indicar o separador dos valores no ficheiro. Por defeito, o seu valor é *white space*, que inclui um ou mais caracteres de espaço e *tab's*.

É possível explicitar outro separador, como por exemplo a vírgula.

Várias destas alternativas são muito comuns. Por isso o R tem outras funções que são essencialmente iguais à `read.table()`, sendo que a diferença fundamental está no separador por defeito que é assumido bem como noutros pequenos detalhes relacionados com os *defaults*. É esse o caso das funções `read.csv()`, `read.csv2()`, `read.delim()` e `read.delim2()`.

Vejamos um pequeno exemplo de como usarmos uma destas funções. Suponhamos que temos um ficheiro de texto com dados, de que mostramos as primeiras linhas,

```
ID;Nome;Nota
434;Carlos;13,2
523;Ana;15,1
874;Susana;4,8
103;Joaquim;15,9
...
```

Se pretendessemos usar a função `read.table()` para ler este ficheiro, deveriamos fazer:

```
dados <- read.table('ficheiro.txt',header=T,sep=';',dec=',')
```

Como este formato é bastante vulgar, uma das funções mencionadas usa estes valores como valores por defeito, e portanto seria mais prático, neste caso particular, fazer,

```
dados <- read.csv2('ficheiro.txt')
```

Estas são as funções principais para ler ficheiros de texto no R. Existem ainda outras possibilidades, como a função `read.fwf()`, que permite ler ficheiros que usem um formato de tamanho fixo.

Uma breve nota sobra a leitura de ficheiros muito grandes (dezenas de milhares de linhas e centenas de colunas). Nestas situações, e assumindo que os dados cabem na memória do computador que se está a usar, as funções do tipo `read.table()` poderão mostrar-se demasiado lentas. Nestas situações recomenda-se que se considere a utilização da função `scan()`, de uso menos intuitivo, mas bastante mais eficiente em termos computacionais.

1.2. Importar da Internet

Para sabermos qual a melhor forma de importar dados da Internet para o R temos que analisar com cuidado o formato em que são fornecidos.

Se os dados estão fornecidos como um ficheiro de texto com um formato qualquer (por exemplo CSV), e o que nos dão é o URL para esse local (por exemplo `http://www.blabla.com/dados.txt`), então a melhor forma de proceder é fazer o *download* dos dados para um ficheiro local, e depois proceder de alguma das formas indicadas atrás.

Para fazer o *download* do ficheiro, poderemos ou abrir um *browser* e usar alguma opção do género *Save As...*, ou então fazer tudo "dentro" do R. Para isso poderemos usar a função `download.file()`,

```
> download.file('http://www.blabla.com/dados.txt','c:\\My Documents\\dados.txt')
```

Se o URL apontar para uma página Web normal (que é escrita na linguagem HTML), não é tão fácil importar os dados para o R. Por exemplo, a página poderia mostrar uma tabela, no meio de outra informação, cujo conteúdo gostariamos de importar para o R. Nestes casos é necessário importar toda a página (em HTML) e depois interpretar o conteúdo dessa página para daí extraír a informação da tabela. Realizar essa tarefa é possível em R, nomeadamente tirando partido de funções existentes na *package* **XML**, mas no entanto sai fora do âmbito deste texto.

1.3. Do Excel

Em muitas organizações é comum usar o *Excel* para armazenar tabelas de dados.
Existem várias possibilidades para importar os dados para o R.

Uma consiste em gravar os dados do *Excel* para um ficheiro CSV, com as facilidades que o *Excel* tem para isso, e depois usar as funções que aprendemos.

Uma outra possibilidade é utilizar a função `read.xls()` disponível na *package* **gdata**. Esta função permite explicitar o nome da folha de cálculo e o número da *worksheet* contendo os dados que pretendemos importar,

```
dados <- read.xls('dados.xls', sheet=3)
```

Finalmente, é possível usar o *clipboard* do *Windows*, para fazer a importação dos dados. Em concreto, podemos seleccionar a tabela de dados no *Excel*, fazer *Edit+Copy*, e depois, já no R, fazer:

```
dados <- read.table('clipboard',header=T)
```

1.4. De bases de dados

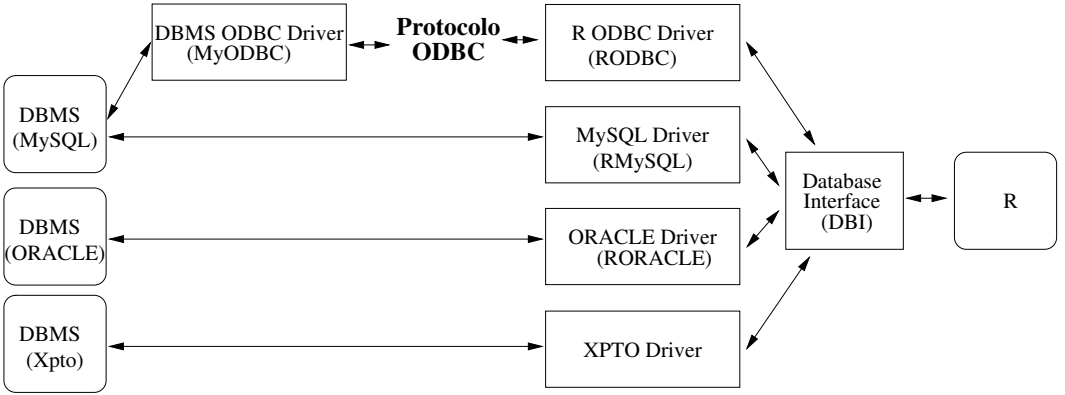
As bases de dados são as infraestruturas por excelência para armazenar dados.

O R tem várias facilidades para fazer o *interface* com este tipo de software, tanto para importar dados como para exportar dados do R para uma base de dados.

O R tem uma *package* chamada **DBI** que implementa uma série de funções de *interface* com bases de dados. Estas funções são independentes do sistema de gestão de bases de dados (SGBD) que estamos a usar.

Para tornar isto possível, para além da *package* **DBI**, precisamos ainda de outras *packages* associadas a cada um dos SGBD's que pretendemos de facto usar.

Por exemplo, se pretendemos fazer o *interface* com uma base de dados guardada em `Oracle`, temos que instalar também a *package* **ROracle**.



Vamos ver um exemplo concreto usando o SGBD **MySQL**, um excelente SGBD de carácter gratuíto como o R.

A melhor forma de fazermos o interface com este SGBD depende da plataforma onde estamos a trabalhar com o R. No *Windows* é melhor usar o protocolo ODBC, enquanto que noutras plataformas é mais fácil usar o *package* **RMySQL**.

Da primeira vez que pretendemos fazer o interface a uma base de dados no **MySQL** usando o protocolo ODBC, são necessários alguns passos extra.

1. Instalar o driver ODBC do **MySQL** (chamado myodbc)que podemos obter no *site* deste SGBD (`http://www.m`
2. Em ODBC cada ligação a uma base de dados tem um nome (o *Data Source Name*, ou *DSN* na linguagem ODBC). Para criar uma ligação ODBC em *Windows* temos de usar um programa chamado *ODBC data sources* que está disponível no *Control Panel* do *Windows*. Temos que criar uma nova *User Data Source* que use o *MySQL ODBC driver* (myodbc) que instalamos anteriormente.
   Durante este processo de criação vão-nos ser perguntadas várias coisas como por exemplo o endereço do servido **MySQL** (tipicamente `localhost` se o servidor está a ser executado no seu computador), o nome da base de dados que pretendemos aceder, e também o nome que pretendemos dar a esta ligação (o tal nome que depois vamos usar no R).

Assim que este processo esteja completo, estamos prontos a establecer ligações à base de dados usando o protocolo ODBC. O exemplo que mostramos em seguida, faz uma ligação a um base de dados, para a qual anteriormente criamos uma ligação ODBC (DSN) com o nome "teste",

```
> library(RODBC)
> library(DBI)
> drv <- dbDriver('ODBC')
> ch <- dbConnect(drv,'teste','xpto','passwordxpto')
> dados <- dbGetQuery(ch,'select * from tabela')
> dbDisconnect(ch)
> dbUnloadDriver(drv)
```

2. Sumarizar Dados em R

Temos já visto vários exemplos das potencialidades do R quando toca a sumarizar as propriedades básicas de dados.

A função principal para este efeito é a função `summary()`,

```
> data(iris)
> summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

Muitas vezes pretendemos ter só uma pequena ideia sobre a forma dos dados, e dada a sua dimensão, é impraticável mostrar todo o *data frame*. Nessas situações estas funções são úteis,

```
> str(iris)

'data.frame':        150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 ...
```

```
> head(iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
> tail(iris)

    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

2.1. Sumários sobre sub-grupos dos dados

Quando os sub-grupos são definidos por factores, podemos usar a função `by()`,

```
> by(iris[, -5], iris$Species, summary)

iris$Species: setosa
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.300   Min.   :1.000   Min.   :0.100
 1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200
 Median :5.000   Median :3.400   Median :1.500   Median :0.200
 Mean   :5.006   Mean   :3.428   Mean   :1.462   Mean   :0.246
 3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300
 Max.   :5.800   Max.   :4.400   Max.   :1.900   Max.   :0.600
------------------------------------------------------------
iris$Species: versicolor
  Sepal.Length    Sepal.Width     Petal.Length   Petal.Width
 Min.   :4.900   Min.   :2.000   Min.   :3.00   Min.   :1.000
 1st Qu.:5.600   1st Qu.:2.525   1st Qu.:4.00   1st Qu.:1.200
 Median :5.900   Median :2.800   Median :4.35   Median :1.300
 Mean   :5.936   Mean   :2.770   Mean   :4.26   Mean   :1.326
 3rd Qu.:6.300   3rd Qu.:3.000   3rd Qu.:4.60   3rd Qu.:1.500
 Max.   :7.000   Max.   :3.400   Max.   :5.10   Max.   :1.800
------------------------------------------------------------
iris$Species: virginica
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.900   Min.   :2.200   Min.   :4.500   Min.   :1.400
 1st Qu.:6.225   1st Qu.:2.800   1st Qu.:5.100   1st Qu.:1.800
 Median :6.500   Median :3.000   Median :5.550   Median :2.000
 Mean   :6.588   Mean   :2.974   Mean   :5.552   Mean   :2.026
 3rd Qu.:6.900   3rd Qu.:3.175   3rd Qu.:5.875   3rd Qu.:2.300
 Max.   :7.900   Max.   :3.800   Max.   :6.900   Max.   :2.500
```

O segundo argumento pode ter uma lista de factores,

```
> data(warpbreaks)
> attach(warpbreaks)
> by(warpbreaks[, 1], list(wool = wool, tension = tension), summary)
wool: A
tension: L
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  25.00   26.00   51.00   44.56   54.00   70.00
------------------------------------------------------------
wool: B
tension: L
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.00   20.00   29.00   28.22   31.00   44.00
------------------------------------------------------------
wool: A
tension: M
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     12      18      21      24      30      36
------------------------------------------------------------
wool: B
tension: M
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  16.00   21.00   28.00   28.78   39.00   42.00
------------------------------------------------------------
wool: A
tension: H
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  10.00   18.00   24.00   24.56   28.00   43.00
------------------------------------------------------------
wool: B
tension: H
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.00   15.00   17.00   18.78   21.00   28.00
```

A função a aplicar pode ser quão "complicada" quisermos,

```
> by(warpbreaks, tension, function(x) lm(breaks ~ wool, data = x))

tension: L

Call:
lm(formula = breaks ~ wool, data = x)

Coefficients:
(Intercept)        woolB
      44.56       -16.33

------------------------------------------------------------
tension: M

Call:
lm(formula = breaks ~ wool, data = x)

Coefficients:
(Intercept)        woolB
     24.000        4.778

------------------------------------------------------------
tension: H

Call:
lm(formula = breaks ~ wool, data = x)

Coefficients:
(Intercept)        woolB
     24.556       -5.778
```

A função `aggregate()` produz resultados semelhantes mas numa estrutura de dados mais facilmente re-utilizável,

```
> aggregate(warpbreaks[, 1], list(wool = wool, tension = tension),
+     mean)

  wool tension        x
1    A       L 44.55556
2    B       L 28.22222
3    A       M 24.00000
4    B       M 28.77778
5    A       H 24.55556
6    B       H 18.77778
```

Note todavia que a função a aplicar aos sub-grupos tem que produzir um escalar, contrariamente ao caso da função `by()`. Isto quer dizer, por exemplo, que não poderiamos aplicar a função `summary()` com esta função `aggregate()`.

Por vezes os sub-grupos não são mais do que um sub-conjunto das colunas. A função `apply()` é bastante útil quando queremos fazer "coisas" sobre as colunas (ou linhas) de um *data frame*,

```
> meuSumario <- function(x) {
+     s <- c(mean(x, na.rm = T), min(x, na.rm = T), max(x, na.rm = T),
+         sd(x, na.rm = T), var(x, na.rm = T), length(which(is.na(x))))
+     names(s) <- c("média", "mín", "máx", "desvioPadrão",
+         "variância", "N.desc.")
+     s
+ }
> apply(iris[, 1:4], 2, meuSumario)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
média	5.8433333	3.0573333	3.758000	1.1993333
mín	4.3000000	2.0000000	1.000000	0.1000000
máx	7.9000000	4.4000000	6.900000	2.5000000
desvioPadrão	0.8280661	0.4358663	1.765298	0.7622377
variância	0.6856935	0.1899794	3.116278	0.5810063
N.desc.	0.0000000	0.0000000	0.000000	0.0000000

Neste exemplo criamos uma função específica para obter o sumário que pretendiamos.

Quando se tratam de sub-grupos das linhas, muitas vezes estes são definidos por condições lógicas nos valores das colunas.

Os mecanimos de indexação que aprendemos são muito úteis para seleccionar estes sub-grupos e depois aplicar a função de sumarização que pretendemos.

A função `subset()` por vezes é mais prática do que a indexação,

```
> summary(subset(iris, Petal.Length > 6))
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :7.200   Min.   :2.600   Min.   :6.100   Min.   :1.800   setosa    :0
 1st Qu.:7.400   1st Qu.:2.800   1st Qu.:6.100   1st Qu.:2.000   versicolor:0
 Median :7.700   Median :3.000   Median :6.400   Median :2.100   virginica :9
 Mean   :7.578   Mean   :3.144   Mean   :6.433   Mean   :2.122
 3rd Qu.:7.700   3rd Qu.:3.600   3rd Qu.:6.700   3rd Qu.:2.300
 Max.   :7.900   Max.   :3.800   Max.   :6.900   Max.   :2.500
```

```
> summary(subset(iris, Petal.Length > 6, Sepal.Width:Petal.Width))
```

```
  Sepal.Width     Petal.Length    Petal.Width
 Min.   :2.600   Min.   :6.100   Min.   :1.800
 1st Qu.:2.800   1st Qu.:6.100   1st Qu.:2.000
 Median :3.000   Median :6.400   Median :2.100
 Mean   :3.144   Mean   :6.433   Mean   :2.122
 3rd Qu.:3.600   3rd Qu.:6.700   3rd Qu.:2.300
 Max.   :3.800   Max.   :6.900   Max.   :2.500
```

3. Fórmulas

As fórmulas são objectos da linguagem R que permitem explicitar uma forma genérica de um sub-conjunto de dados.

Elas são muito usadas para indicar a estrutura genérica de um modelo de dados que pretendemos obter com uma qualquer função.

São também bastante usadas para obter gráficos que mostrem uma relação específica entre um sub-conjunto das variáveis dos nossos dados.

Uma fórmula tem a estrutura genérica

`variável ~ expressão`

em que `variável` representa a variável dependente, e `expressão` é uma expressão que indica de que variáveis depende `variável`.

Vejamos um exemplo com os dados "iris",

```
> data(iris)
> names(iris)

[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

No contexto destes dados, a fórmula "`Sepal.Length  Petal.Length + Species`", indica que pretendemos obter algo (um modelo, um gráfico, etc.) que relacione a variável `Sepal.Length` com as variáveis `Petal.Length` e `Species`.

As fórmulas também podem incluir transformações dos dados, como ilustrado nesta fórmula "`Petal.Length ~ log(Petal.Width)`".

As expressões a seguir ao símbolo "~" podem incluír:

- O sinal "+" significando inclusão.
- O sinal "−" significando exclusão.
- O sinal "." significando incluir todas as variáveis.
- O sinal "*" aplicado a factores representa todas as combinações dos seus valores.

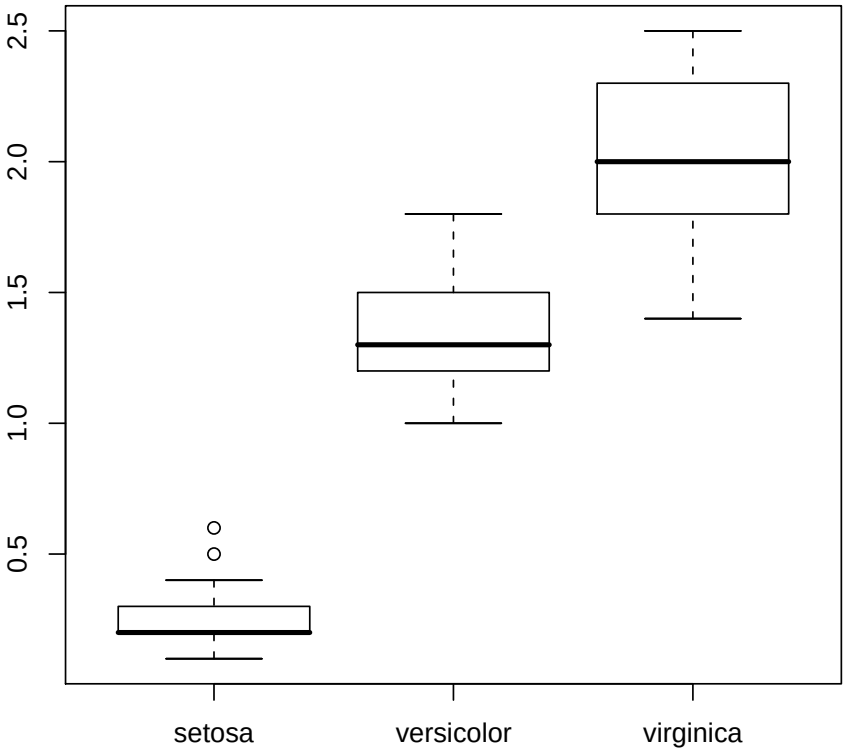
- A função "I()" permite usar os operadores no seu verdadeiro sentido aritmético.
- O sinal ":" gera todas as interacções com os valores de um factor.

Vejamos alguns exemplos do uso de fórmulas:

```
> lm(Petal.Length ~ Sepal.Width, data = iris)


Call:
lm(formula = Petal.Length ~ Sepal.Width, data = iris)

Coefficients:
(Intercept)  Sepal.Width
      9.063       -1.735
```

```
> lm(Petal.Length ~ Sepal.Width:Species, data = iris)


Call:
lm(formula = Petal.Length ~ Sepal.Width:Species, data = iris)

Coefficients:
                  (Intercept)      Sepal.Width:Speciessetosa
                       2.1887                         -0.2085
Sepal.Width:Speciesversicolor   Sepal.Width:Speciesvirginica
                       0.7489                          1.1258



> lm(Petal.Length ~ Sepal.Width * Species, data = iris)


Call:
lm(formula = Petal.Length ~ Sepal.Width * Species, data = iris)

Coefficients:
                  (Intercept)                   Sepal.Width
                      1.18292                       0.08141
           Speciesversicolor              Speciesvirginica
                      0.75200                       2.32798
Sepal.Width:Speciesversicolor   Sepal.Width:Speciesvirginica
                      0.75797                       0.60490
```

```
> boxplot(Petal.Width ~ Species, data = iris)
```



4. Visualizar Dados em R

Os gráficos em R estão organizados em 2 tipos de sistemas de gráficos:

1. O sistema tradicional implementado na package "graphics".
2. O sistema de gráficos Trellis implementado com base na package "grid" e disponibilizado na package "lattice".

As funções disponibilizadas em cada uma destas packages podem ser divididas em 3 classes de funções:

1. Funções de alto-nível que produzem gráficos completos.
2. Funções de baixo-nível que permitem adicionar elementos a gráficos já desenhados.
3. Funções para trabalhar de forma interactiva com gráficos já desenhados.

4.1. Os devices gráficos

Um *device* gráfico é o "local" onde é desenhado o gráfico.

Por defeito este local é o écran. Quando faço `plot(rnorm(10))` vou ver o resultado no écran.

Todavia, o R permite facilmente direccionar o mesmo gráfico para outro *device*. A escolha do *device* define não só o local onde o gráfico é produzido, mas também o tipo de *output*.

Por exemplo, se quisesse o gráfico apresentado acima num ficheiro PDF, bastaria fazer:

```
> pdf(file = "exp.pdf")
> plot(rnorm(10))
> dev.off()
```

Se quisesse o mesmo em formato JPEG,

```
> jpeg(file = "exp.pdf")
> plot(rnorm(10))
> dev.off()
```

Qualquer destes (e de muitos outros) *devices* existentes no R, tem uma série de parâmetros que permitem controlar o *output*.

Em R também é possível abrir vários *devices* ao mesmo tempo, embora só um deles possa estar activo (para onde vão os gráficos).

Isto é útil, por exemplo, para ter vários gráficos no écran ao mesmo tempo.

A função `windows()` permite abrir mais uma janela de gráficos em Windows,

```
> plot(rnorm(10)
> windows()
> plot(rnorm(20)
```

O segundo gráfico vai surgir numa outra janela, o que permite ao utilizador ver os dois ao mesmo tempo se quiser.

Note que posteriores gráficos que venhamos a realizar vão ser desenhados na segunda janela de gráficos (o *device* actualmente activo).

As funções `dev.cur()`, `dev.set()`, e `dev.list()` são úteis para saber qual o *device* activo actualmente, mudar o *device* activo, e listar os *devices* activos.

4.2. Os gráficos tradicionais

Estes gráficos são proporcionados pela package "graphics" que é automaticamente carregada quando se executa o R.

4.2.1. Gráficos de uma ou duas variáveis

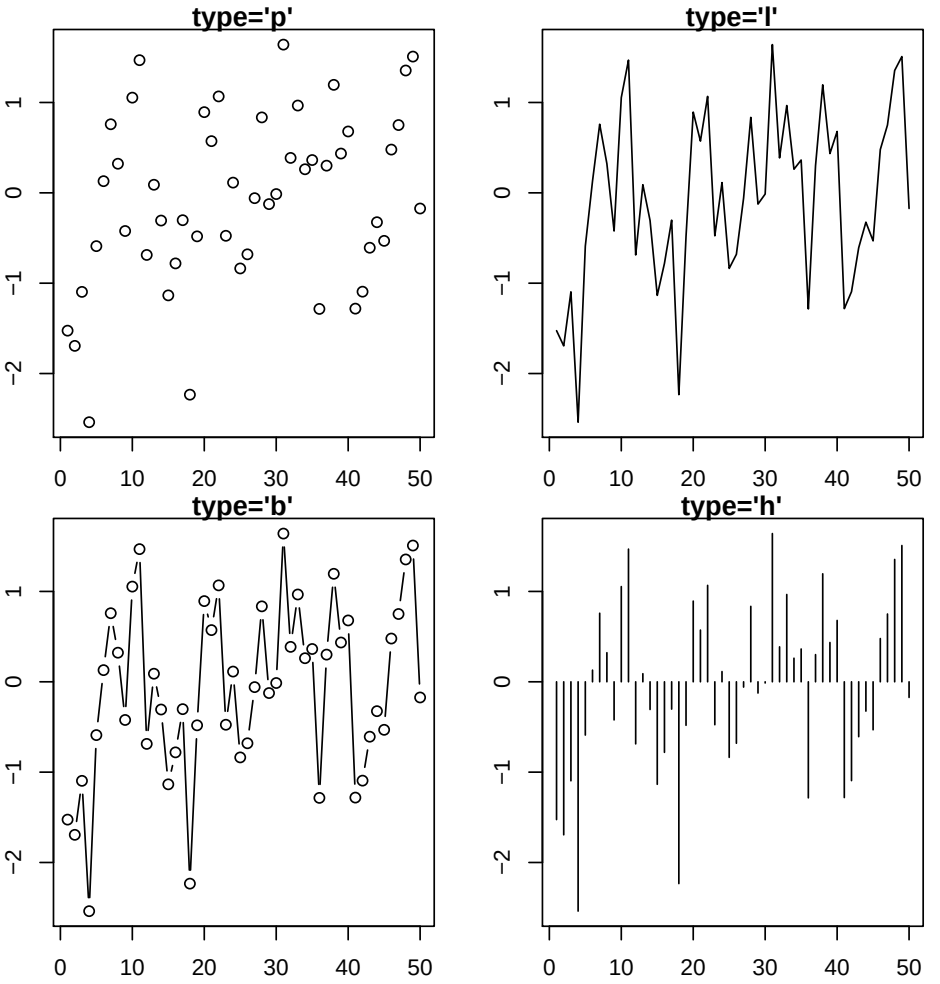
1. A função `plot()`

   Esta função pode ser usada para desenhar símbolos num conjunto de coordenadas x, y, muitas vezes conhecidos como *scatterplots*.

   A função pode também ser usada para desenhar linhas unindo estes pontos, bastando para isso mudar o valor do parâmetro `type`.

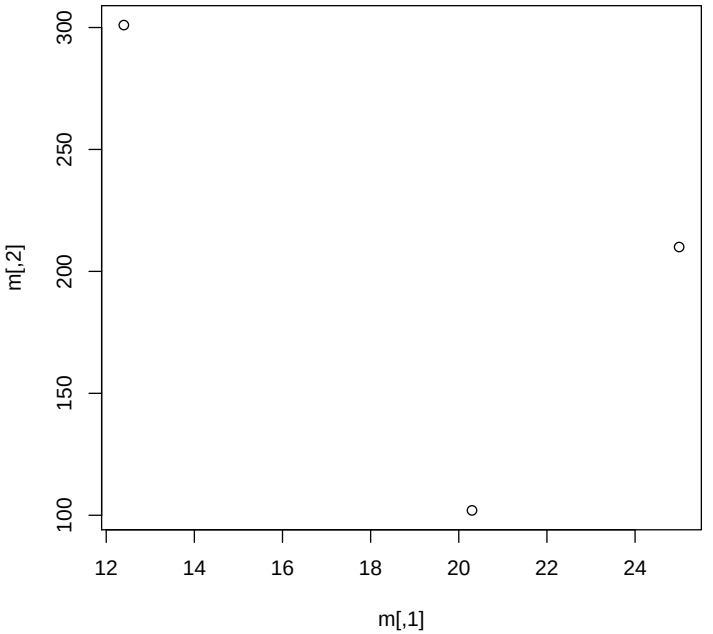
   A função `plot()` é uma função genérica, existindo muitos métodos para objectos de classes específicas. Aqui vamos descrever o seu método *default*.

```
> x <- rnorm(50)
> plot(x, type = "p")
> plot(x, type = "l")
> plot(x, type = "b")
> plot(x, type = "h")
```


type='p'
type='l'
type='b'
type='h'
1
0
−1
−2
0
10
20
30
40
50

Quando fornecemos à função `plot()` uma matriz com duas colunas (ou mesmo um *data frame*), ela vai usar cada coluna como sendo as coordenadas (x e y dos pontos a desenhar,

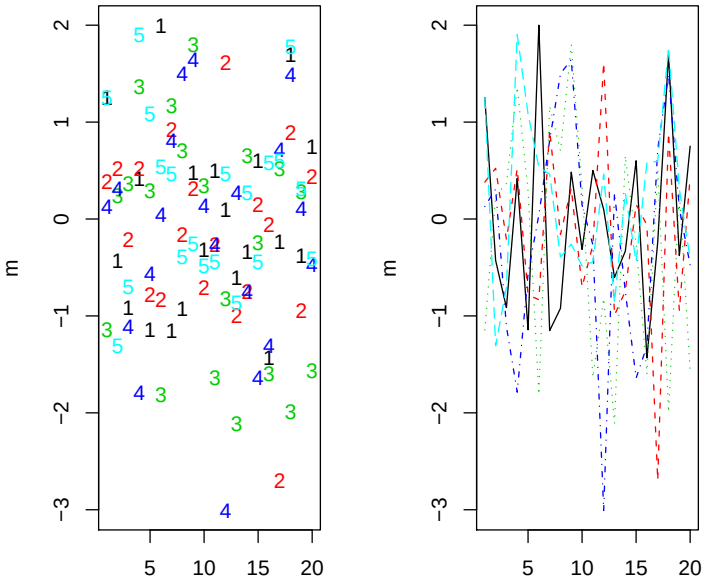
```
> m <- matrix(c(20.3, 12.4, 25, 102, 301, 210), 3, 2)
> plot(m)
```



2. A função `matplot()`

Esta função pode ser usada para desenhar séries de dados, guardadas numa matriz em que cada coluna tem uma série de dados.

```
> m <- matrix(rnorm(100), 20, 5)
> op <- par(mfrow = c(1, 2))
> matplot(m)
> matplot(m, type = "l")
> par(op)
```



4.2.2. Gráficos de mais do que duas variáveis

As funções principais para gráficos com 3 variáveis são:

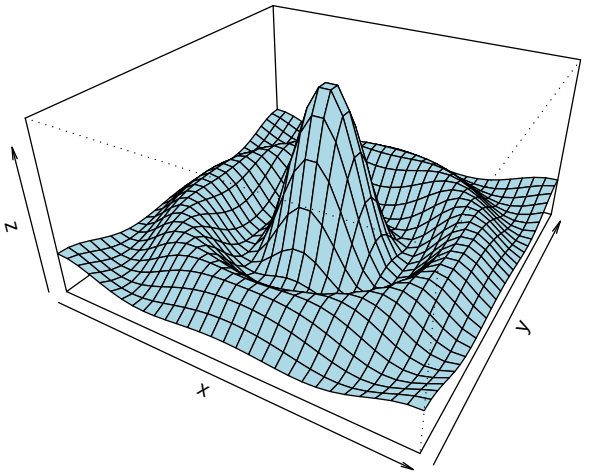
- `persp()` para produzir superfícies tridimensionais.
- `countour()` para produzir gráficos com curvas de nível representando uma superfície tridimensional.
- `image()` para produzir uma representação bidimensional de uma superfície tridimensional, usando cores para representar a 3ª dimensão.

Para dados com mais do que 3 variáveis, temos:

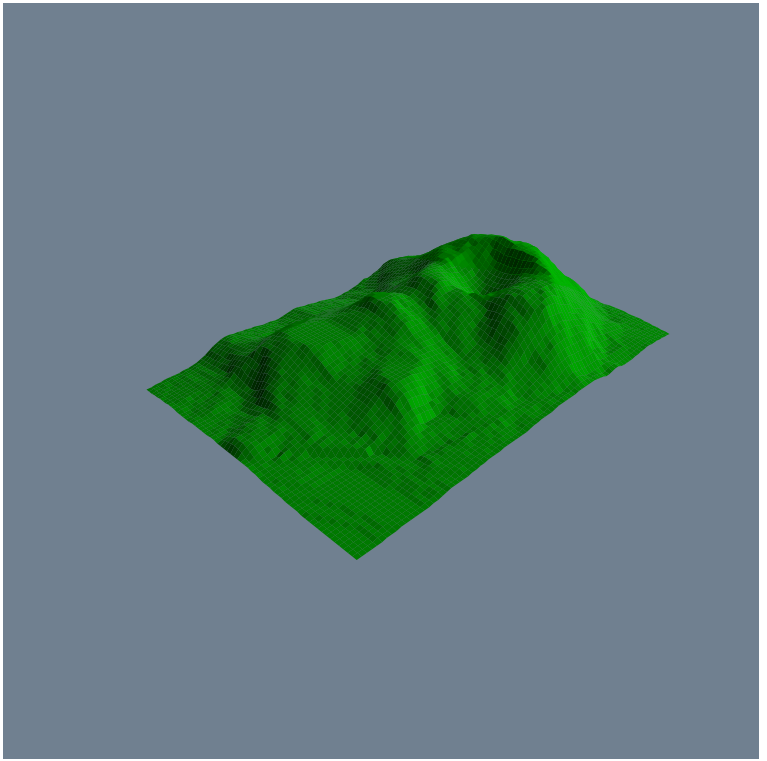
- `pairs()` para produzir uma matriz de *scatterplots* para todas os pares de combinações de variáveis.
- `stars()` para produzir gráficos do tipo *stars* para variáveis contínuas.
- `mosaicplot()` para produzir gráficos do tipo *mosaic* para variáveis nominais.

Vejamos alguns exemplos da sua utilização

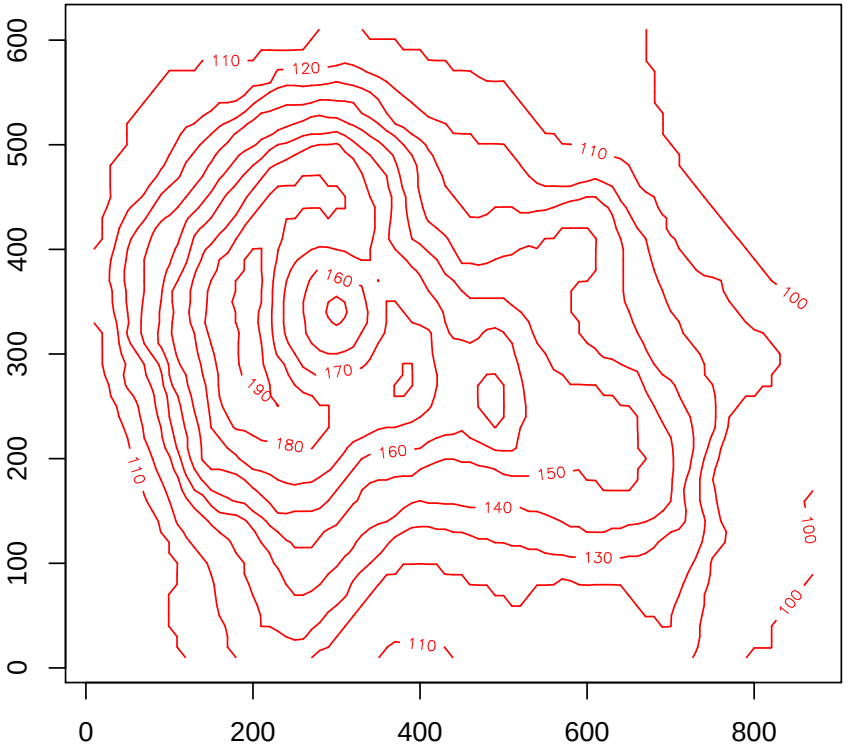
```
> x <- seq(-10, 10, length = 30)
> y <- x
> f <- function(x, y) {
+     r <- sqrt(x^2 + y^2)
+     10 * sin(r)/r
+ }
> z <- outer(x, y, f)
> z[is.na(z)] <- 1
> persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
```



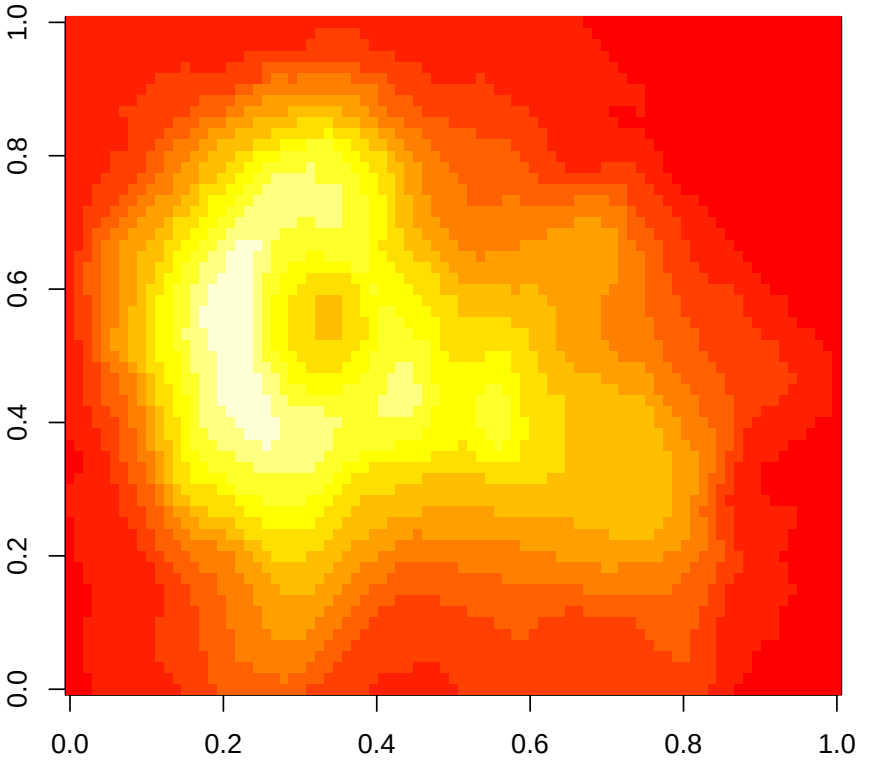
```
> z <- 2 * volcano
> x <- 10 * (1:nrow(z))
> y <- 10 * (1:ncol(z))
> op <- par(bg = "slategray")
> persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE,
+     ltheta = -120, shade = 0.75, border = NA, box = FALSE)
> par(op)
```



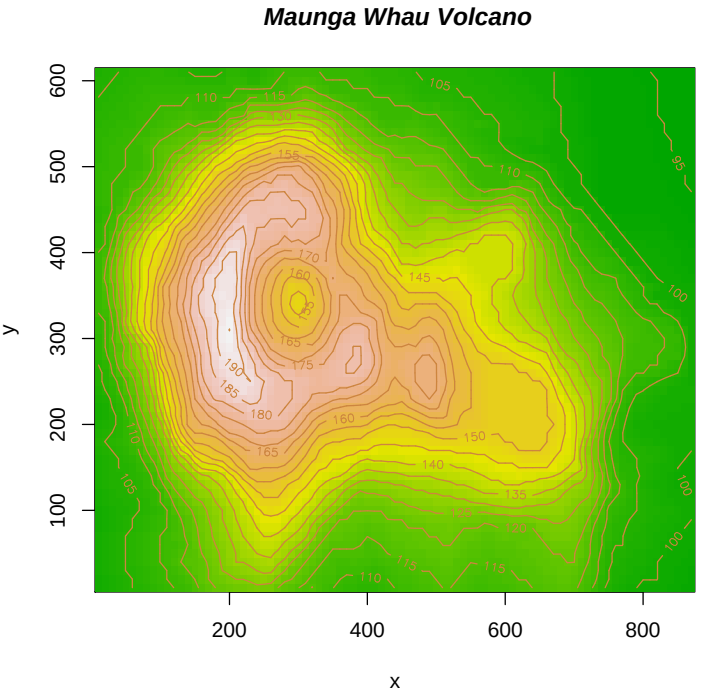
```
> x <- 10 * 1:nrow(volcano)
> y <- 10 * 1:ncol(volcano)
> contour(x, y, volcano, col = "red", lty = "solid")
```



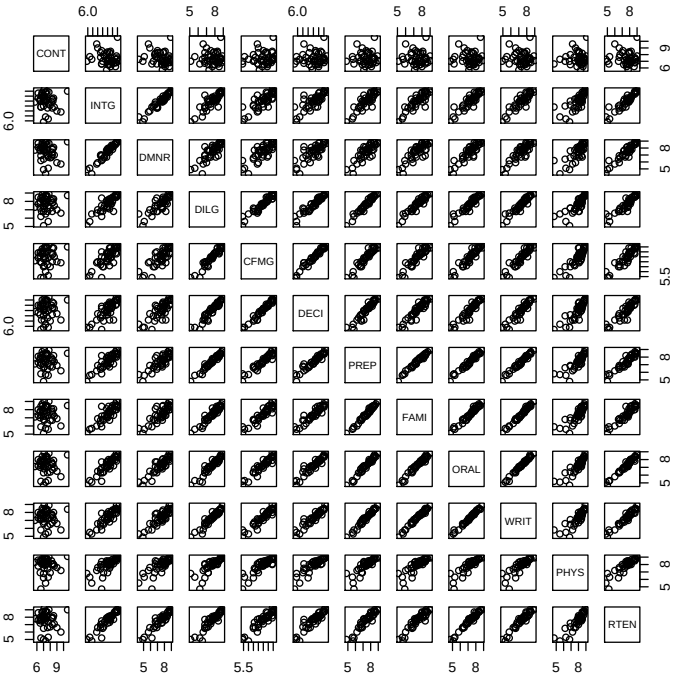
```
> image(volcano)
```



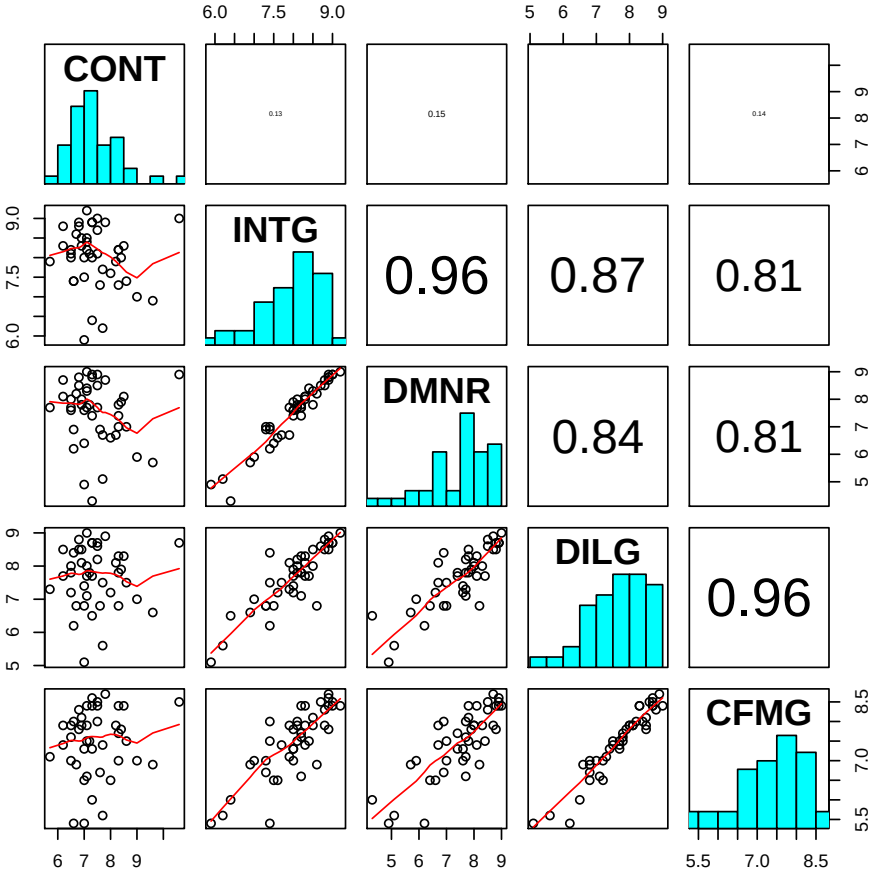
```
> x <- 10 * (1:nrow(volcano))
> y <- 10 * (1:ncol(volcano))
> image(x, y, volcano, col = terrain.colors(100))
> contour(x, y, volcano, levels = seq(90, 200, by = 5), add = TRUE,
+     col = "peru")
> title(main = "Maunga Whau Volcano", font.main = 4)
```



```
> pairs(USJudgeRatings)
```

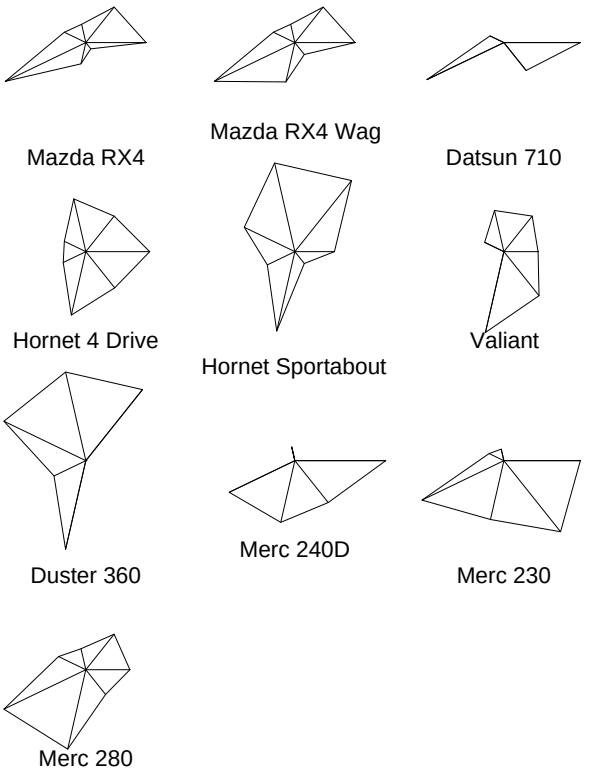


```
> panel.hist <- function(x, ...) {
+     usr <- par("usr")
+     on.exit(par(usr))
+     par(usr = c(usr[1:2], 0, 1.5))
+     h <- hist(x, plot = FALSE)
+     breaks <- h$breaks
+     nB <- length(breaks)
+     y <- h$counts
+     y <- y/max(y)
+     rect(breaks[-nB], 0, breaks[-1], y, col = "cyan", ...)
+ }
> panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor) {
+     usr <- par("usr")
+     on.exit(par(usr))
+     par(usr = c(0, 1, 0, 1))
+     r <- abs(cor(x, y))
+     txt <- format(c(r, 0.123456789), digits = digits)[1]
+     txt <- paste(prefix, txt, sep = "")
+     if (missing(cex.cor))
+         cex <- 0.8/strwidth(txt)
+     text(0.5, 0.5, txt, cex = cex * r)
+ }
> pairs(USJudgeRatings[1:5], lower.panel = panel.smooth, upper.panel = panel.cor,
+     diag.panel = panel.hist, cex.labels = 2, font.labels = 2)
```

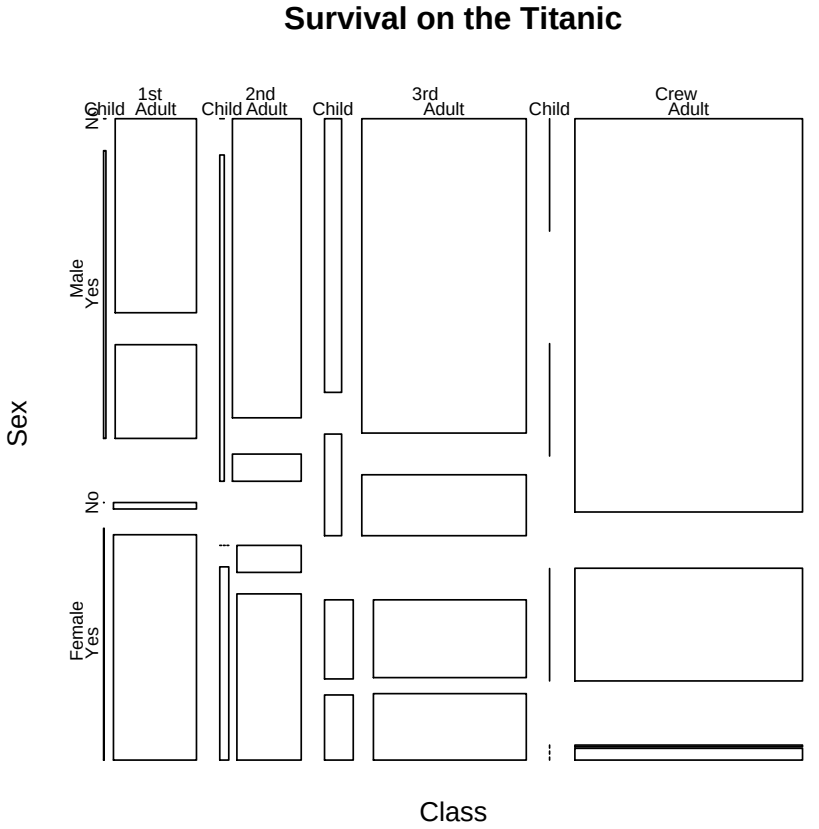

CONT
INTG
DMNR
DILG
CFMG
0.96
0.87
0.81
0.84
0.81
0.96
0.13
0.15
0.14
6.0 7.5 9.0
5 6 7 8 9
6.0 7.5 9.0
5 6 7 8 9
5 6 7 8 9
5.5 7.0 8.5
5.5 7.0 8.5
6 7 8 9
5 6 7 8 9

```
> stars(mtcars[1:10, 1:7], main = "Motor Trend Cars")
```

Motor Trend Cars



```
> mosaicplot(Titanic, main = "Survival on the Titanic")
```

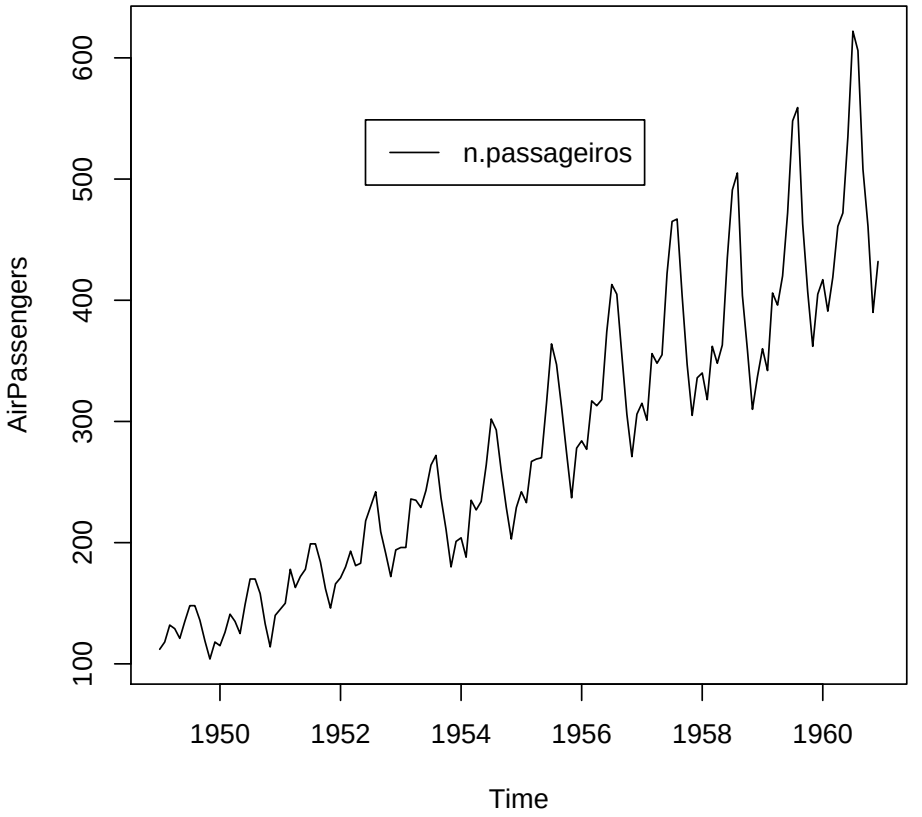


4.2.3. Interacção com Gráficos

- A função `locator()`
  Serve para obter as coordenadas dos pontos "cliquados" com o rato numa janela de gráficos.

  Um exemplo de aplicação para posicionar a legenda de um gráfico:

```
> plot(AirPassengers)
> legend(locator(1), "n.passageiros", lty = 1)
```


n.passageiros
AirPassengers
100
200
300
400
500
600
1950
1952
1954
1956
1958
1960
Time

- A função `identify()`
  Serve para identificar pontos específicos de um gráfico e por exemplo escrever perto os respectivos valores da variável.

  Um exemplo

```
> plot(CO2$uptake)
> identify(CO2$uptake)
```

  ou então

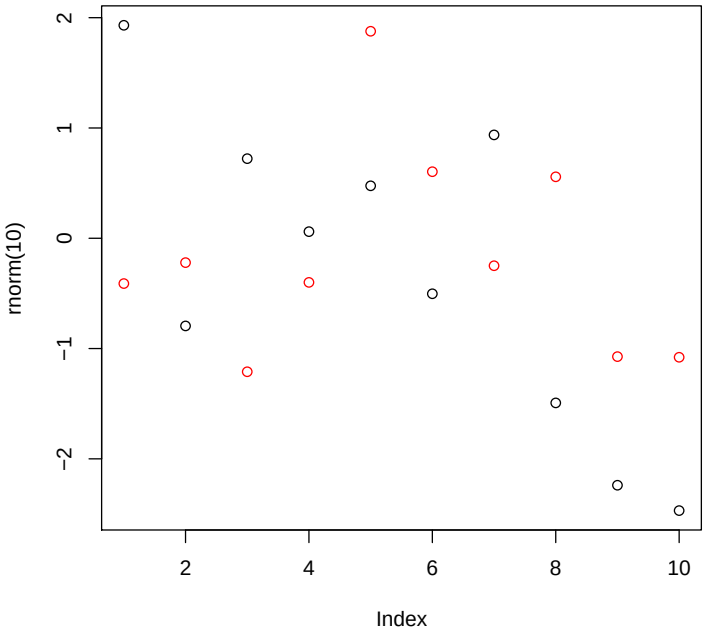
```
> plot(CO2$uptake)
> identify(CO2$uptake, labels = CO2$Plant)
```

4.2.4. Adicionar Informação a Gráficos Existentes

- Novos "pontos" num gráfico (função `points()`)

  Esta função permite acrescentar novos dados a um gráfico já desenhado:

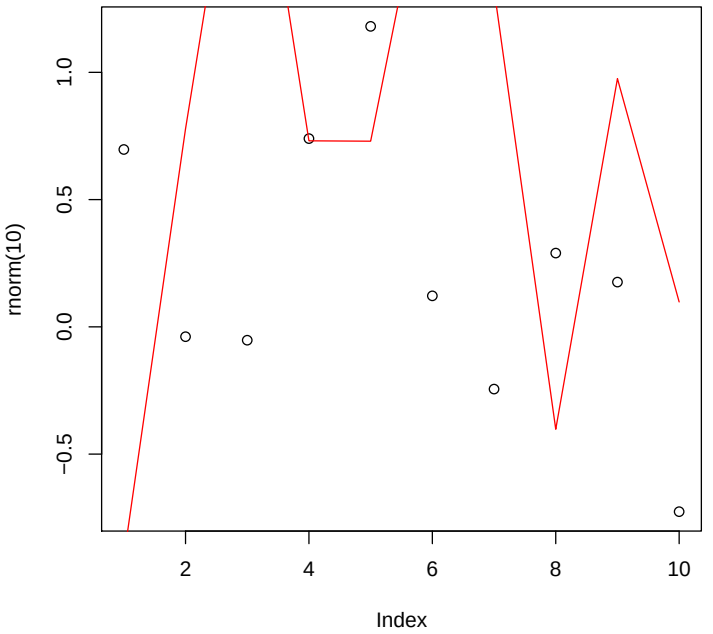
```
> plot(rnorm(10))
> points(rnorm(10), col = "red")
```



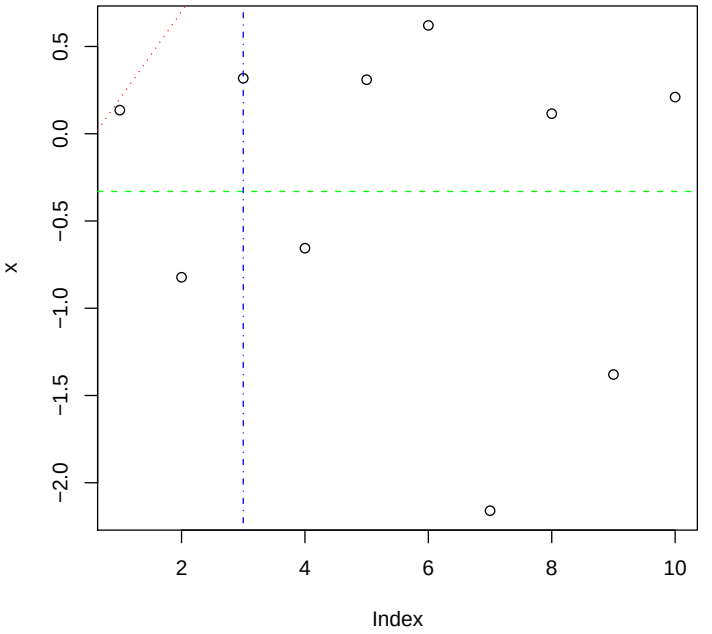
- Novas linhas num gráfico (funções `lines()` e `abline()`)

  Esta função permite acrescentar novos dados, desenhados sob a forma de uma linha, a um gráfico já desenhado:

```
> plot(rnorm(10))
> lines(rnorm(10), col = "red")
```



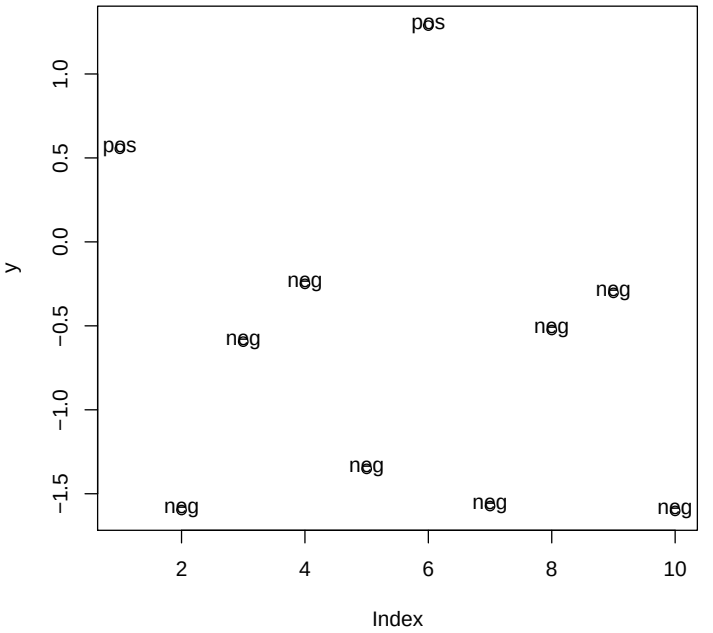
```
> x <- rnorm(10)
> plot(x)
> abline(h = mean(x), col = "green", lty = 2)
> abline(v = 3, col = "blue", lty = 4)
> abline(-0.3, 0.5, col = "red", lty = 3)
```



- Acrescentar texto a um gráfico (função `text()`)

  Esta função permite acrescentar texto em qualquer ponto de um gráfico já desenhado:

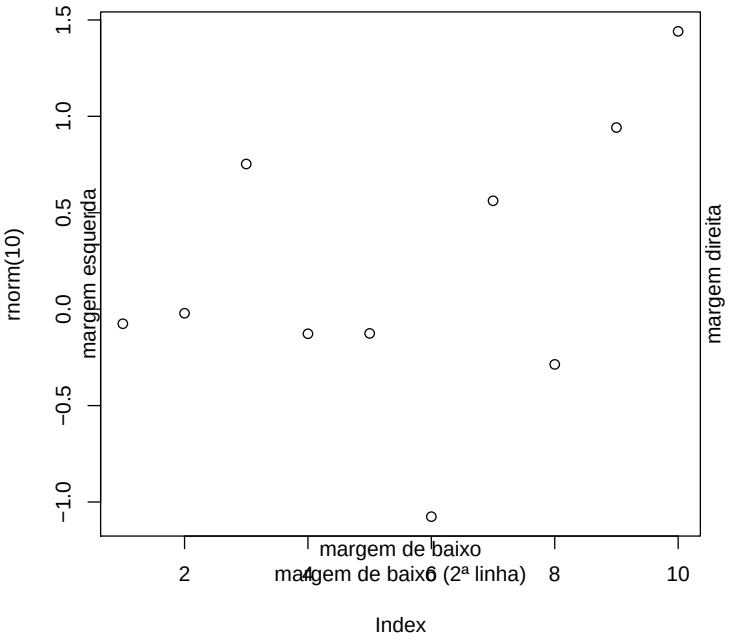
```
> y <- rnorm(10)
> plot(y)
> text(1:10, y, ifelse(y > 0, "pos", "neg"))
```



- Acrescentar texto nas margens de um gráfico (função `mtext()`)

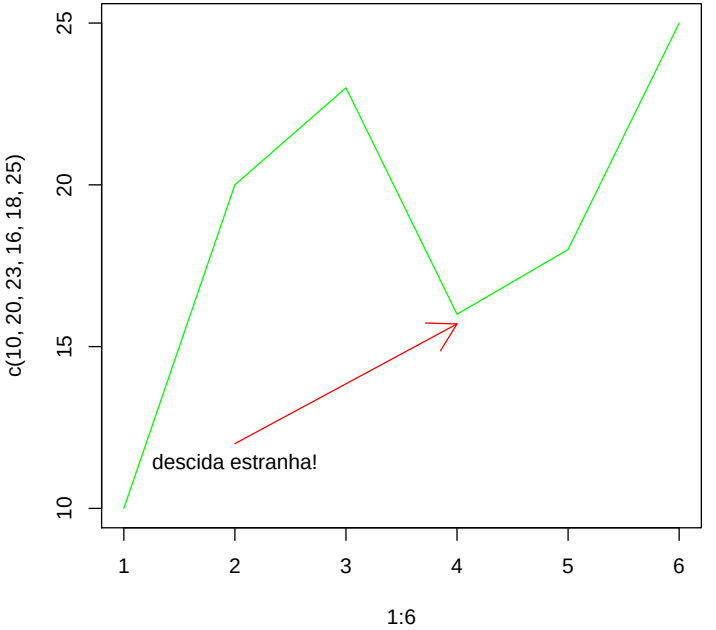
  Esta função permite acrescentar texto em qualquer das margens de um gráfico já desenhado:

```
> plot(rnorm(10))
> mtext("margem de baixo", side = 1)
> mtext("margem de baixo (2ª linha)", side = 1, line = 1)
> mtext("margem esquerda", side = 2)
> mtext("margem direita", side = 4)
```



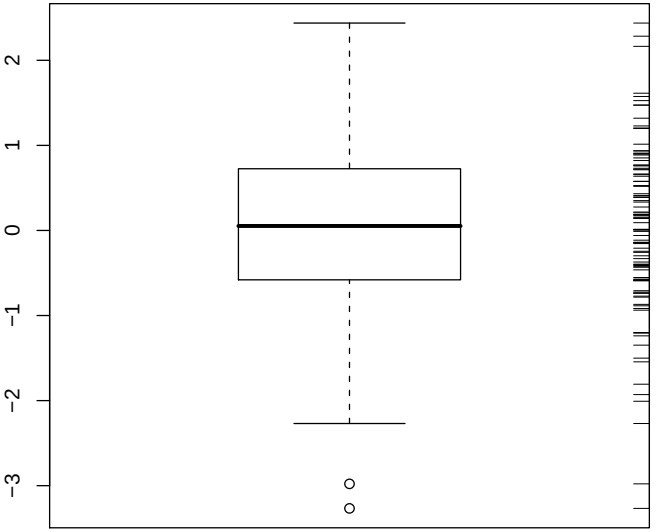
- Acrescentar setas a um gráfico (função `arrows()`)

```
> plot(1:6, c(10, 20, 23, 16, 18, 25), type = "l", col = "green")
> arrows(2, 12, 4, 15.7, col = "red")
> text(2, 12, "descida estranha!", pos = 1)
```



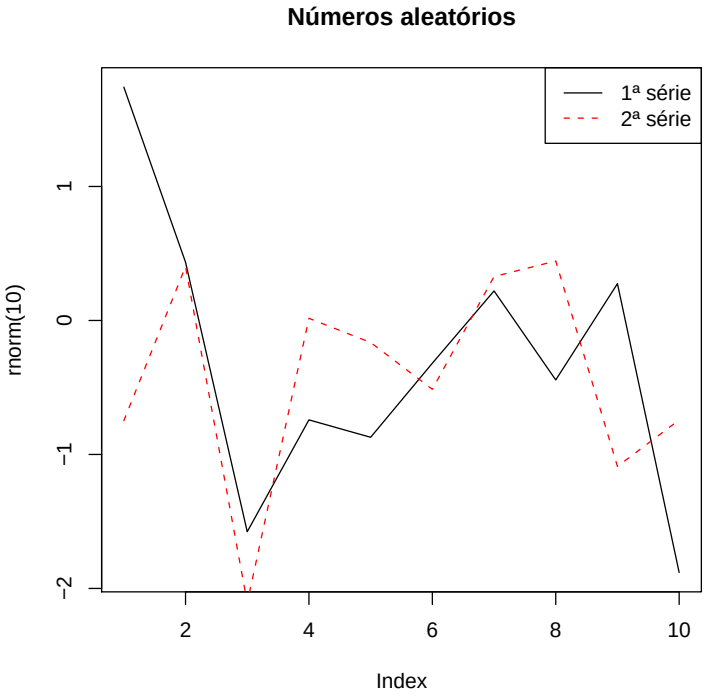
- Acrescentar traços sobre os dados reais... (função `rug()`)

```
> x <- rnorm(100)
> boxplot(x)
> rug(x, side = 4)
```



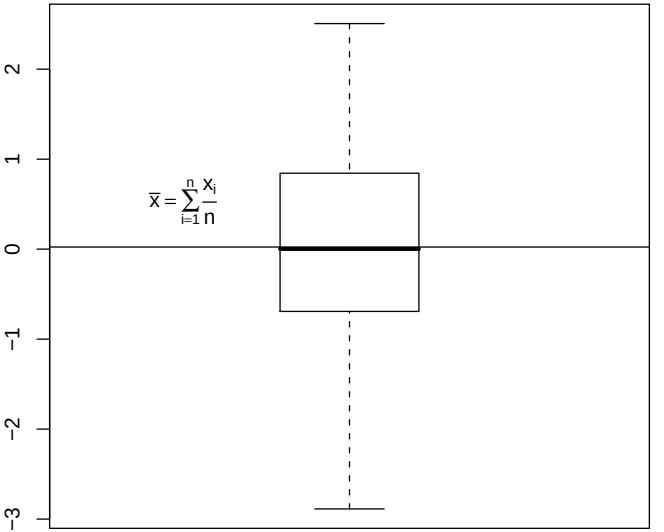
- Acrescentar títulos e legendas (funções `title()` e `legend()`)

```
> plot(rnorm(10), type = "l")
> lines(rnorm(10), col = "red", lty = 2)
> title("Números aleatórios")
> legend("topright", c("1ª série", "2ª série"), lty = 1:2,
+     col = 1:2)
```



- Acrescentar fórmulas matemáticas

```
> x <- rnorm(100)
> boxplot(x, boxwex = 0.5)
> abline(h = mean(x))
> text(0.7, mean(x) + 0.5, substitute(paste(bar(x) == sum(frac(x[i],
+     n), i == 1, n))))
```



Fazer `demo(plotmath)` para mais exemplos.

4.3. Parâmetros de Gráficos

Os gráficos do R são altamente parametrizáveis.
Qualquer função tem inúmeros parâmetros que podem ser consultados na respectiva ajuda.
A função `par()` permite fazer o *setting* "permanente" de vários parâmetros gráficos.
Vejamos alguns dos parâmetros mais comuns e que aparecem em quase todas as funções:

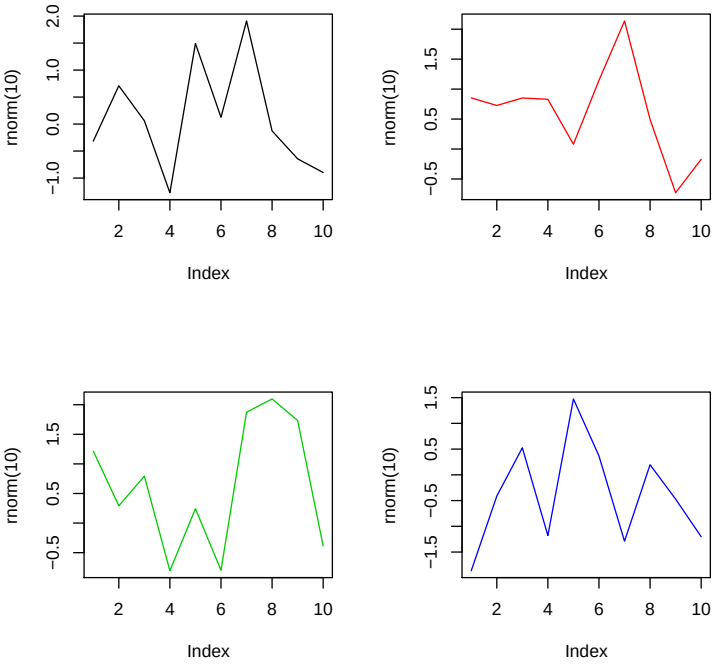
- `col` - permite indicar a cor de elementos do gráfico.
- `main` - permite dar um título ao gráfico.
- `xlab` - permite dar um título ao eixo dos X's.
- `ylab` - permite dar um título ao eixo dos Y's.
- `xlim` - permite indicar um vector com 2 números que serão usados como o *range* de valores no eixo dos X's.
- `ylim` - permite indicar um vector com 2 números que serão usados como o *range* de valores no eixo dos Y's.
- `lty` - permite indicar o tipo de linhas que vão ser usadas nos gráficos.
- `cex` - permite indicar um tamanho relativo do texto usado nos gráficos.
- `pch` - os símbolos usados para desenhar os pontos no gráfico.

4.4. Dividir a Janela de Gráficos em Várias Áreas

Há várias maneiras de usar a área da janela de gráficos para desenhar vários gráficos ao mesmo tempo. Vejamos duas formas:

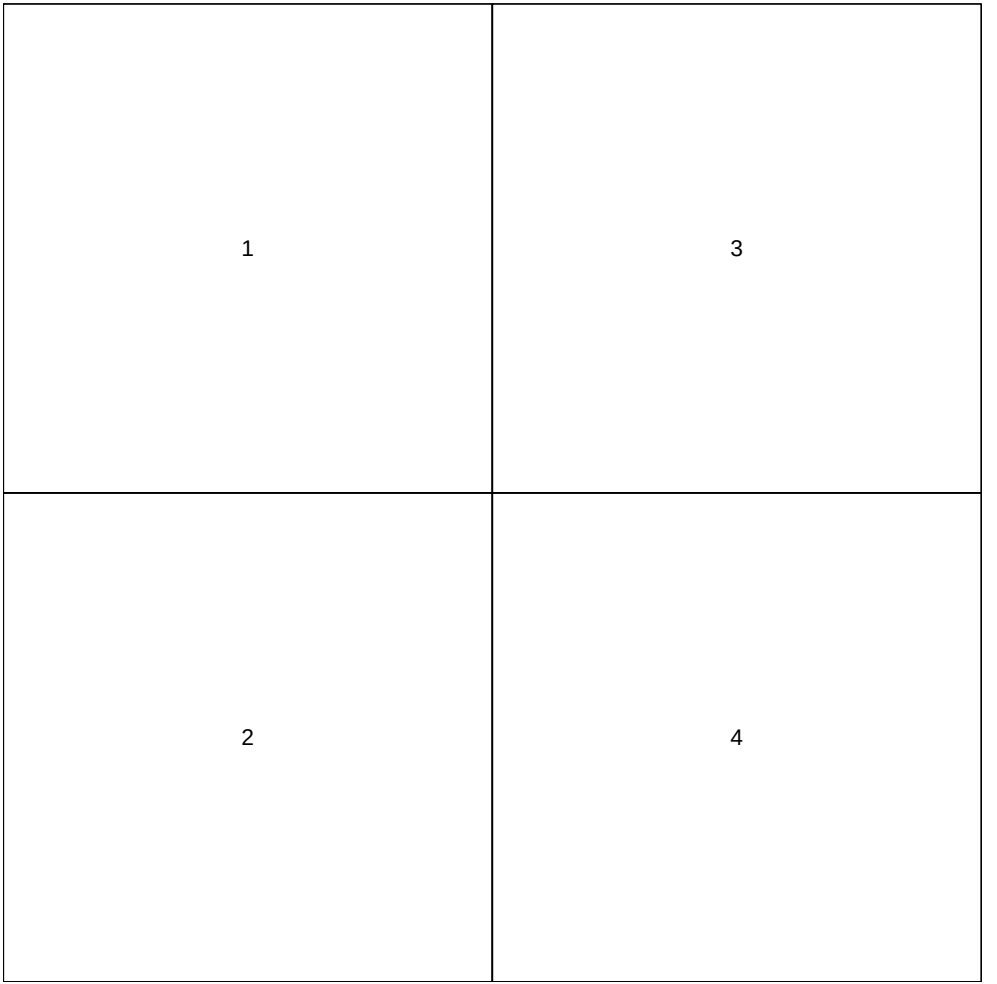
- Gráficos de tamanho igual.

```
> op <- par(mfrow = c(2, 2))
> for (i in 1:4) plot(rnorm(10), col = i, type = "l")
> par(op)
```

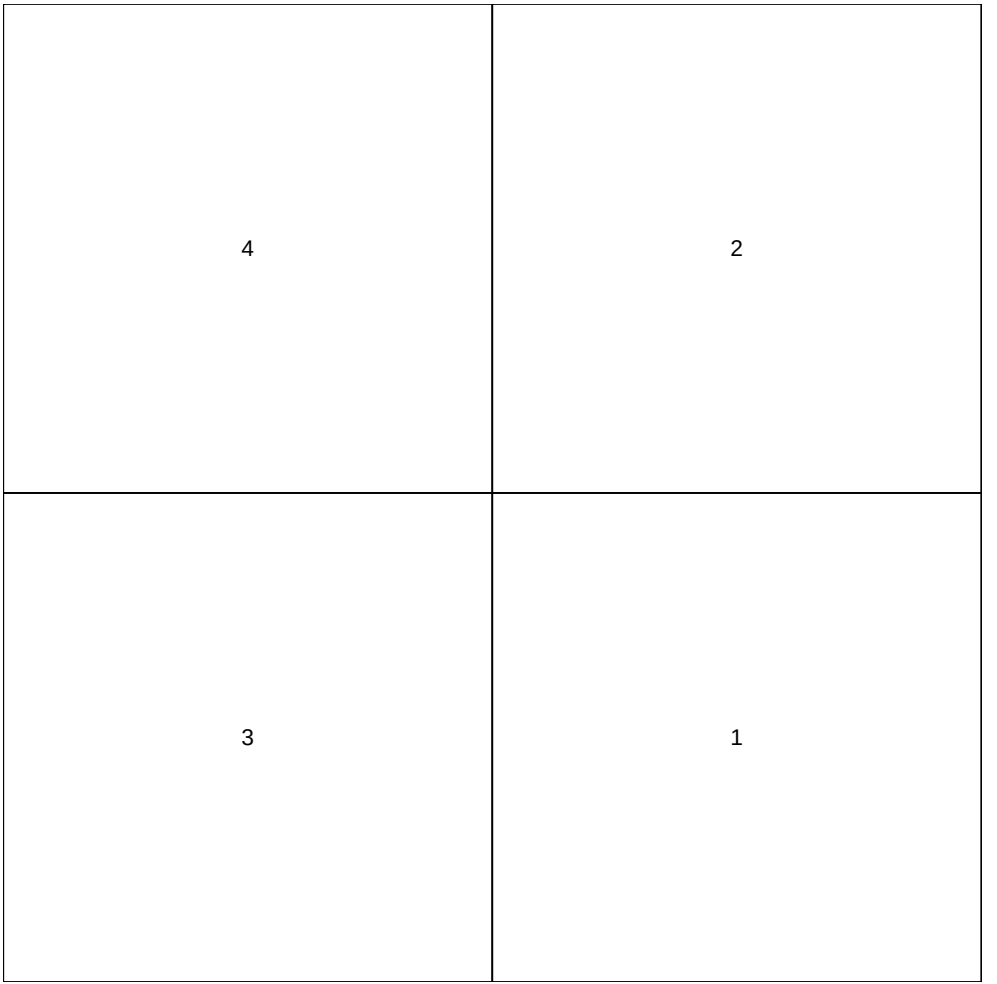


- Gráficos de diferente tamanho - a função `layout()`.

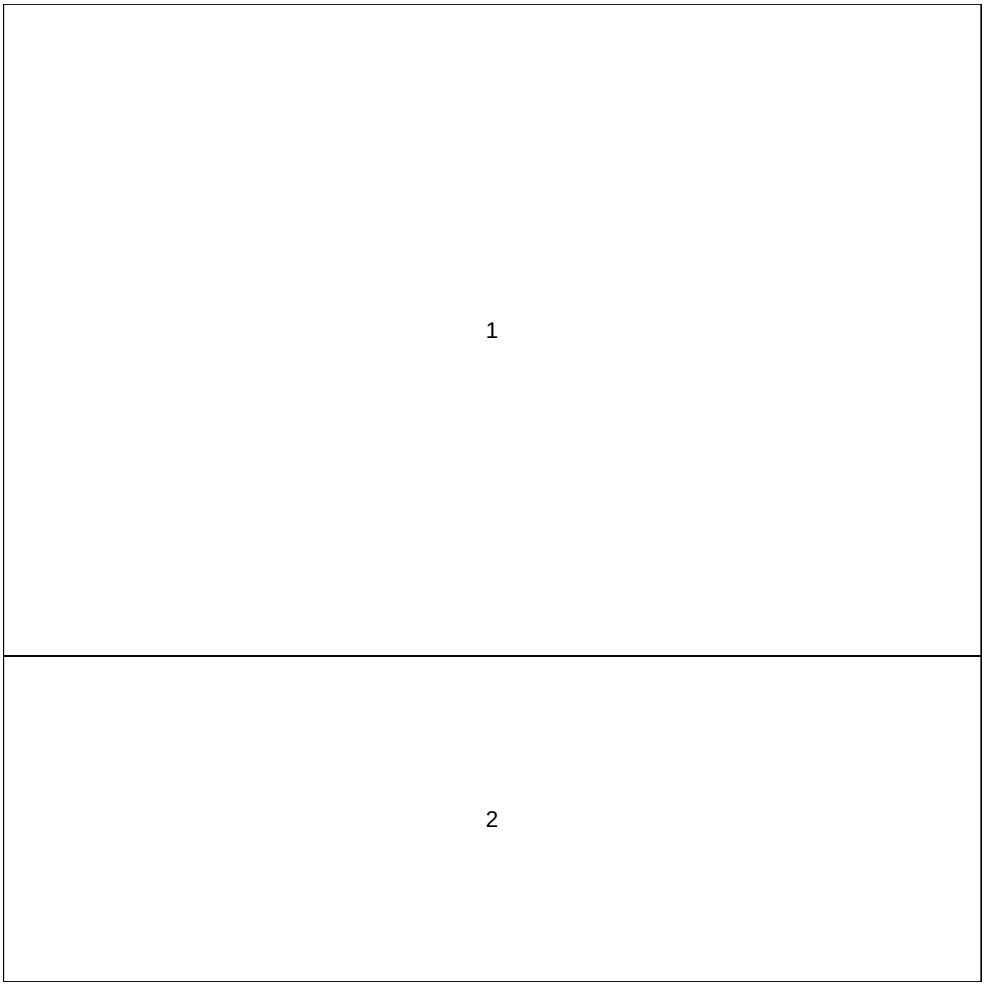
```
> layout(matrix(1:4, 2, 2))
> layout.show(4)
```



```
> layout(matrix(4:1, 2, 2))
> layout.show(4)
```



```
> layout(matrix(1:2, 2, 2), heights = c(2, 1))
> layout.show(2)
```



```
> layout(rbind(c(1, 2), c(3, 3)), height = c(1, 2))
> layout.show(3)
```