Faculdade de Economia

Universidade do Porto

Introdução à Programação em R

Luís Torgo
ltorgo@liacc.up.pt

Grupo de Matemática e Informática

Outubro de 2006

Conteúdo

1 Introdução ao Ambiente R **7**
- 1.1 Instalação do R . . . 7
- 1.2 Começar a usar o R . . . 7
- 1.3 Ajuda sobre o R . . . 9
- 1.4 "Packages" do R . . . 9

2 Fundamentos da Linguagem R **11**
- 2.1 Os objectos do R . . . 11
- 2.2 Vectores . . . 12
- 2.3 Operações com Vectores . . . 14
- 2.4 Factores . . . 15
- 2.5 Sequências . . . 18
- 2.6 Indexação . . . 20
- 2.7 Matrizes . . . 22
- 2.8 *Arrays* . . . 26
- 2.9 Listas . . . 27
- 2.10 *Data Frames* . . . 29
- 2.11 Séries Temporais . . . 31
  - 2.11.1 Séries Regulares . . . 31
  - 2.11.2 Séries Irregulares . . . 36

3 Programação em R **45**
- 3.1 Interacção com o Utilizador . . . 45
- 3.2 Estruturas de Controlo da Linguagem R . . . 46
  - 3.2.1 Instruções Condicionais . . . 46
  - 3.2.2 Instruções Iterativas . . . 48
  - 3.2.3 Evitando ciclos . . . 50
- 3.3 Funções . . . 52
  - 3.3.1 Criar funções . . . 52
  - 3.3.2 Ambientes e "scope" de variáveis . . . 53
  - 3.3.3 Argumentos de funções . . . 54
  - 3.3.4 *Lazy evaluation* . . . 56
  - 3.3.5 Algumas funções úteis . . . 56
- 3.4 Objectos, Classes e Métodos . . . 58
- 3.5 Depuração de Programas em R . . . 60

4 Manipulação de Dados **63**
- 4.1 Carregar dados para o R . . . 63
  - 4.1.1 De ficheiros de texto . . . 63
  - 4.1.2 Da Internet . . . 64
  - 4.1.3 Do *Excel* . . . 65
  - 4.1.4 De bases de dados . . . 65
- 4.2 Sumarização de dados . . . 67
- 4.3 Fórmulas . . . 70
- 4.4 Visualização de dados . . . 71
  - 4.4.1 Gráficos Univariados . . . 73
  - 4.4.2 Gráficos de 3 Variáveis . . . 74
  - 4.4.3 Gráficos Multivariados . . . 78
  - 4.4.4 Gráficos Condicionados . . . 81
  - 4.4.5 Interacção com Gráficos . . . 83
  - 4.4.6 Adicionar Informação a Gráficos Existentes . . . 84

4.4.7 Parâmetros de Gráficos 90
4.4.8 Dividir a Janela de Gráficos em Várias Áreas 91

Referências **95**

Índice **97**

Prefácio

Este documento tem como objectivo principal fornecer uma introdução à programação usando a linguagem R. A escolha desta linguagem como veículo para aprender a programar prende-se com os objectivos do curso em que esta disciplina se insere. Uma vez que este curso tem a análise de dados e o *data mining* como um dos assuntos centrais, o R sendo também um ambiente de análise exploratória de dados, permite uma melhor articulação com outras disciplinas.

A escolha do R tem ainda outros motivos relacionados com o facto de este ser um software gratuito e de código aberto. Estas características, em conjunto com as suas reconhecidas qualidades, fazem dele uma ferramenta quase ideal para aprender a programar dentro dum contexto de análise de dados e sistemas de apoio à decisão. Ao usarem uma ferramenta deste género como plataforma de implementação dos conceitos aprendidos, os alunos poderão facilmente, pelo seu carácter gratuito, levar o que aprenderam para o cenário profissional em que se enquadram ou venham a enquadrar. Além disso, as suas características de código aberto irão facilitar a eventual necessidade de adaptar algum dos muitos métodos disponíveis no R, para assim melhor os adequarem aos seus interesses/objectivos.

1 Introdução ao Ambiente R

O que é o R?

O R (R Development Core Team, 2006) é ao mesmo tempo uma linguagem de programação e um ambiente para computação estatística e gráficos. Trata-se de uma linguagem de programação especializada em computação com dados. Uma das suas principais características é o seu carácter gratuito e a sua disponibilidade para uma gama bastante variada de sistemas operativos. Neste documento iremos concentrar a nossa atenção na versão Windows, mas basicamente tudo o que aqui é descrito também se aplica às outras versões, dadas as muito reduzidas diferenças entre as versões para as diversas plataformas.

Apesar do seu carácter gratuito o R é uma ferramenta bastante poderosa com boas capacidades ao nível da programação e um conjunto bastante vasto (e em constante crescimento) de *packages* que acrescentam bastantes potencialidades à já poderosa versão base do R.

O R é uma variante da linguagem S com a qual John Chambers (Chambers, 1998) ganhou o prestigiado prémio de software da organização ACM.

1.1 Instalação do R

Embora o R esteja instalado no sistema informático da FEP, dado o seu carácter gratuito, os alunos poderão instalar este software nos seus computadores em casa. Para isso necessitarão simplesmente de ligar o computador à Internet e seguir os passos que descrevemos em seguida. Caso não pretenda realizar a instalação do R no seu computador, poderá avançar para a próxima secção.

Download do R

Para instalar o R os alunos deverão começar por ligar o seu computador à Internet. Após esta ligação deverá ser feito o *download* do R a partir do *site* desta aplicação, `http://www.r-project.org`. Neste *site* deverá ser seguido o *link* com o nome **CRAN** no menu disponível à esquerda. Depois de escolher um dos muitos locais espalhados pelo mundo para fazer o *download*, deverá seguir o *link Windows (95 and later)* disponível na secção *Precompiled Binary Distributions*. No écran seguinte, deve "entrar" na pasta **base** e fazer o *download* do ficheiro **R-2.3.1-win32.exe**[1].

Instalação do R

Uma vez efectuado o *download* do ficheiro mencionado acima, deverá proceder-se à instalação do R , bastando para isso executar o ficheiro (carregando duas vezes em cima dele no **Explorer**). De notar que em versões mais recentes do **Windows** esta instalação poderá ter de ser feita pelo utilizador **Administrador**, por questões de permissões.

1.2 Começar a usar o R

Iniciar o R

Para se executar o R basta usar ou o ícone que normalmente está disponível no *desktop* do **Windows**, ou então usar o respectivo item no menu "Start"[2].

A execução do R faz aparecer a janela desta aplicação de que se pode ver um exemplo na Figura 1.

O prompt do R

À parte os menus que iremos explorar mais tarde, esta janela apresenta o *prompt* do R (`>`), com o cursor à sua frente. É aqui que vamos introduzir os comandos que pretendemos que o R execute. Podemos começar por ver um pequeno exemplo do tipo de interacção que vamos ter com o R, escrevendo o seguinte comando no *prompt* e carregando em seguida na tecla ENTER (esta é a forma de mandar o R executar o comando que acabamos de escrever),

Executar comandos

```
> R.version
platform i386-pc-mingw32
arch     i386
os       mingw32
system   i386, mingw32
status
```

[1] O nome do ficheiro poderá variar em versões posteriores do R. Na altura da escrita deste texto a versão do R disponível era a 2.3.1 e daí o nome do ficheiro.

[2] Na FEP o R está disponível seguindo o percurso de menus: `Start->Programs->Aplications->R`.

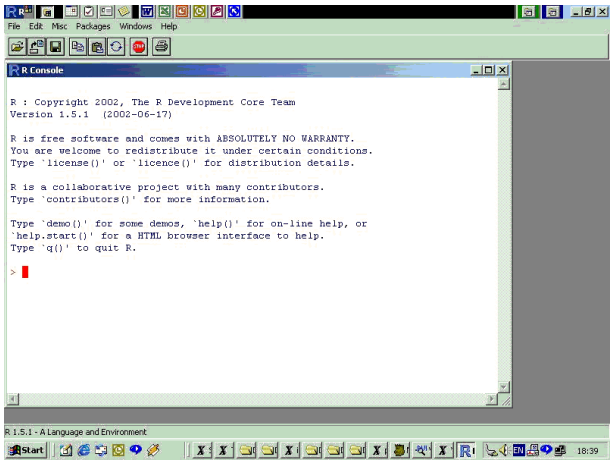


Figura 1: O ambiente R.

```
major    2
minor    3.1
year     2006
month    06
day      01
svn rev  38247
language R
version.string Version 2.3.1 (2006-06-01)
```

O comando `R.version` ao ser executado fez aparecer uma série de informação sobre a versão do R bem como sobre o computador onde o R está a ser executado.

Terminar o R

Para terminar a execução do R basta executar o seguinte comando,

```
> q()
```

Continuar o trabalho mais tarde

Ao executar este comando irá aparecer uma caixa de diálogo como a apresentada na Figura 2. Se respondermos *Yes* a esta pergunta o R vai guardar a informação contida na memória do computador num ficheiro, de modo a que da próxima vez que executarmos o R no local onde esse ficheiro é guardado, ele vai permitir-nos continuar o nosso trabalho exactamente de onde o estamos a abandonar ao executar o comando `q()`. A informação guardada consiste basicamente na história de comandos que executamos nesta sessão que agora terminamos, bem como os objectos que criamos na nossa sessão. O resultado será que o R vai criar 2 ficheiros: um chamado `.Rhistory` contendo a lista dos comandos que executamos, e outro chamado `.RData` contendo os objectos criados na sessão. Gravar a sessão só terá interesse se de facto pretendermos continuar o que estávamos a fazer mais tarde. Na maior parte das vezes vamos escolher a opção *No* que irá abandonar o R sem guardar o estado em que estávamos. Os ficheiros com o estado da sessão são sempre gravados no directório actual onde o R está a funcionar. Para saber o directório actual do R basta fazer no *prompt*,

```
> getwd()
```

Alterar o directório actual

Em resposta a este comando o R irá apresentar no écran o directório actual. Para o alterar poder-se-á ou usar a opção `Change dir...` do menu *File*, ou então usar a função `setwd()` conforme ilustrado no exemplo seguinte,

```
> setwd('U:\\My Documents\\AulasR')
```

Note a utilização de dois caracteres \\, em vez do único caracter como é costume em ambientes Windows[3].

[3] Acrescente-se a título de curiosidade que poderá ser usado em vez destes dois caracteres um único caracter "/".

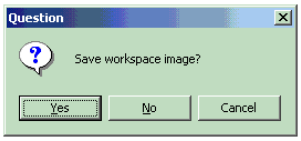


Figura 2: Abandonar o R.

1.3 Ajuda sobre o R

O R tem um sistema de ajuda bastante elaborado que lhe permitirá obter muita informação extra sobre a linguagem, bem como muitos outros aspectos que lhe estão associados. **Obter ajuda no R**

Nas versões Windows do R, a forma mais fácil de obter ajuda é através da utilização do menu *Help* disponível na janela da aplicação R. Através deste menu é possível por exemplo, escolher a opção *Html help* o que fará lançar um *browser* onde poderá tirar partido de uma série de manuais e outros tipo de ajuda disponíveis no R.

No entanto, se pretende simplesmente obter ajuda sobre uma função em particular do R, a forma mais simples será provavelmente usar a função `help()`,

```
> help(sqrt)
```

Este comando irá fazer aparecer uma pequena janela com todo um conjunto de informações úteis sobre a função escolhida, que vai da simples descrição dos seus argumentos até exemplos de utilização, bem como funções relacionadas. Em alternativa, poderá introduzir antes "`? sqrt`", produzindo exactamente o mesmo efeito.

Quando não sabemos o nome concreto da função sobre a qual preciamos de ajuda, podemos usar como alternativas as funções `apropos()` e `help.search()`. Genericamente, ambas produzem uma lista das funções do R que contêm referências ao texto que incluimos como argumento destas funções. Experimente por exemplo fazer, `apropos('model')`, ou `help.search('model')`.

Para dúvidas mais complexas poderá ainda consultar a muita documentação gratuita disponível no site do R (www.r-project.org), ou a *mailing list* de apoio disponível no mesmo site. Se optar por esta última alternativa recomenda-se que antes de colocar qualquer pergunta faça uma procura pelos arquivos da lista para evitar colocar questões já respondidas, o que nem sempre é bem recebido pelas pessoas que se voluntariam para ajudar.

Finalmente uma alternativa poderosa que junta várias destas formas de ajuda do R é utilizar no R a função `RSiteSearch()`. Esta função faz lançar um *browser* que irá mostrar o resultado da procura da "string" que passar como argumento à função. O resultado da procura envolve toda a ajuda de todas as funções do R, ajuda nas *mailing lists*, bem como em outros documentos. Por exemplo, se pretendo saber o que existe nestes locais sobre redes neuronais, poderia fazer a seguinte procura,

```
> RSiteSearch('neural networks')
```

1.4 "Packages" do R

Uma instalação do R vem já com um conjunto de *packages* instaladas. Estas *packages* não são mais do que agregações de funções que foram criadas por alguém que as disponibilizou à comunidade de forma gratuíta. Qualquer pessoa pode criar as suas *packages* e submetê-las ao portal do R para que sejam consideradas na lista de *packages* disponíveis aí. Quando se executa o R só algumas funções estão disponíveis de imediato. Essas são as funções incluídas nas *packages* que foram julgadas mais importantes ou de uso mais comum e que são automaticamente carregadas quando se executa o R. Em qualquer altura poderemos "carregar" uma *package* que contenha funções extra que necessitemos para o nosso trabalho. Para tal ser possível a *package* deverá estar instalada no nosso computador. Se tal não fôr o caso teremos que fazer o *download* da mesma e instalá-la. Este processo é simples, desde que o seu computador esteja ligado à Internet.

Instalar novas packages

Para instalar uma nova *package* podemos usar a função `install.packages()`, que leva como argumento o nome da *package* a instalar[4]. Depois de indicado um repositório de onde fazer o *download* da *package* o R irá encarregar-se de todo o processo, inclusivé da sua instalação no seu computador.

Usar packages

Para utilizar *packages* que estejam disponíveis no seu sistema basta usar a função `library()`, por exemplo,

```
library(rpart)
```

Esta instrução faz com que a partir de agora passem a estar disponíveis para utilização, todos os objectos (funções, dados, etc.) definidos nessa *package*. Cada *package* instalada tem uma ajuda específica que pode ser obtida no sistema de ajuda HTML do R que foi descrito na Secção 1.3.

[4]Na realidade podemos instalar várias indicando um vector (c.f. Secção 2.2) com nomes de *packages*.

2 Fundamentos da Linguagem R

2.1 Os objectos do R

O R é uma linguagem baseada em objectos. Isto quer dizer que tudo o que nós vamos usar no R está guardado na memória do computador sob a forma de um objecto. Todos os objectos em R têm um nome associado e podem armazenar diferentes tipos de coisas (números, texto, vectores, matrizes, expressões, chamadas a funções, etc.).

Objectos do R

Para armazenar algo num objecto usamos o operador de atribuição. Este operador consiste num sinal `<` seguido por um sinal `-`, como se vê no exemplo apresentado em seguida, em que guardamos o número 4.5 no objecto que resolvemos chamar **taxa.de.juro**,

Operador de atribuição

```
> taxa.de.juro <- 4.5
```

Para ver o conteúdo de um objecto (o que está guardado na memória sob um determinado nome) basta digitar o nome do objecto no *prompt* do R, carregando em seguida em Enter. Como resposta o R mostra-nos o conteúdo do objecto que lhe indicamos. Por exemplo, suponhamos que queremos confirmar o conteúdo do objecto **taxa.de.juro**,

Ver o conteúdo de um objecto

```
> taxa.de.juro

[1] 4.5
```

O estranho `[1]` que aparece antes do número guardado em **taxa.de.juro**, tem o significado "esta linha está a mostrar o conteúdo de **taxa.de.juro** a começar no elemento nº 1 do objecto". O significado disto tornar-se-á mais óbvio quando virmos objectos que podem armazenar mais do que um elemento, como por exemplo um vector contendo 100 números.

A operação de atribuição é *destrutiva* no sentido que ao atribuir um novo valor a um objecto já existente, vamos perder o conteúdo que ele estava a armazenar anteriormente.

```
> taxa.de.juro <- 3.9
> taxa.de.juro

[1] 3.9
```

Também podemos atribuir expressões numéricas a objectos. O resultado de tal operação é o de que o resultado do cálculo da expressão é colocado no objecto, e não a expressão propriamente dita,

Expressões numéricas

```
> z <- 5
> w <- z^2
> w

[1] 25

> i <- (z * 2 + 45)/2
> i

[1] 27.5
```

Sempre que pretendemos atribuir o resultado de uma expressão a um objecto e em seguida ver o seu conteúdo (como nos exemplos acima), podemos em alternativa usar a seguinte sintaxe que nos poupa algum tempo,

```
> (w <- z^2)

[1] 25
```

Na realidade não é necessário atribuir o resultado das expressões a objectos. Isto só fará sentido se pretendermos usar o resultado do cálculo mais tarde, por exemplo noutra expressão. Se pretendermos saber simplesmente o resultado do cálculo, podemos usar o R como uma espécie de calculadora,

O R como uma calculadora

```
> (34 + 90)/12.5

[1] 9.92
```

Sempre que criamos um novo objecto usando a instrução de atribuição, ele irá ficar na memória do computador. Como esta é limitada, poderemos apagar os objectos sempre que não precisarmos mais deles. Podemos ver quais os objectos actualmente na memória do computador usando as funções `ls()` ou `objects()`. Se já não necessitamos de algum dos objectos podemos então apagá-lo com a função `rm()` como nos exemplos apresentados em seguida,

Listar os objectos em memória

```
> ls()

[1] "i"            "taxa.de.juro" "w"            "z"

> rm(taxa.de.juro)
> rm(z, w)
> objects()

[1] "i"
```

Nomes válidos

O nome dos objectos pode ser formado por qualquer letra maiúscula ou minúscula, os dígitos 0 a 9 (excepto no início do nome), e também o ponto final ".". Os nomes dos objectos em R são sensíveis às letras maiúsculas / minúsculas. Isto quer dizer que **Cor** e **cor** são dois objectos diferentes para o R. Note também que não pode usar espaços nos nomes dos objectos. Por exemplo, se pretende ter um objecto que guarda uma determinada taxa de câmbio, poderia sentir-se tentado(a) a escolher o nome **taxa de câmbio** para o objecto, o que iria gerar um erro do R,

```
> taxa de câmbio <- 0.05
Error: syntax error
```

Em alternativa poderia usar o seguinte nome que já seria válido,

```
> taxa.de.câmbio <- 0.05
```

2.2 Vectores

O que é um vector

O objecto mais básico do R para guardar dados é o vector. Um vector é uma estrutura de dados que permite armazenar um conjunto de valores do mesmo tipo (por exemplo números) sob um mesmo nome. Esses elementos podem depois ser acedidos individualmente usando um esquema de indexação que iremos descrever mais à frente. Este tipo de estrutura de dados é bastante útil quando pretendemos armazenar várias coisas relacionadas na memória do computador. Por exemplo, suponhamos que pretendemos guardar os lucros obtidos pela nossa empresa ao longo dos 12 meses do ano anterior. Em vez de termos 12 objectos diferentes, cada um guardando o lucro em cada mês, uma vez que são números relacionados uns com os outros, faz mais sentido guardá-los todos numa mesma estrutura de dados, que neste caso será um vector com 12 números. Em R, mesmo quando atribuímos um único número a um objecto (por exemplo fazendo `x <- 45`), estamos de facto a criar um vector de números com um único elemento.

Modo e tamanho dos vectores

Todos os vectores em R têm um *modo* e um *tamanho*. O modo determina o tipo de valores guardado no vector. Em R podemos ter vectores com modo *character*, *logical*, *numeric* e *complex*. Ou seja, podemos ter vectores para armazenar os seguintes tipos de dados atómicos: conjuntos de caracteres, valores lógicos (**F** ou **T** ou **FALSE** ou

TRUE)[5], números inteiros ou reais, e números complexos. O tamanho de um vector é o número de elementos que ele contém, e pode ser obtido com a função `length()`.

Criar vectores

Na maioria das situações vamos criar vectores com mais do que um elemento. Para isso precisamos de usar a função `c()` para indicar ao R os elementos que formam o vector separando-os por vírgulas,

```
> v <- c(4, 7, 23.5, 76.2, 80)
> v

[1]  4.0  7.0 23.5 76.2 80.0

> length(v)

[1] 5

> mode(v)

[1] "numeric"
```

Coerção de tipos

Todos os elementos de um vector têm que ser do mesmo tipo (*modo*). Caso tentemos criar um vector com elementos de tipo diferente o R vai forçá-los a ser do mesmo tipo, alterando-os. Vejamos um exemplo disso,

```
> v <- c(4, 7, 23.5, 76.2, 80, "rrt")
> v

[1] "4"    "7"    "23.5" "76.2" "80"   "rrt"
```

Porque um dos elementos do vector era do tipo caracter, o R passou todos os outros (que eram números) para o tipo caracter, gerando assim um vector de conjuntos de caracteres (*strings*). Isto quer dizer que por exemplo o primeiro elemento desse vector é a *string* "4" e não o número 4, o que tem como consequência, por exemplo, não podermos usar este elemento numa expressão numérica.

As *strings* em R são conjuntos de caracteres englobados por aspas ou plicas,

```
> w <- c("rrt", "ola", "isto e uma string")
> w

[1] "rrt"               "ola"               "isto e uma string"
```

Valores desconhecidos

Todos os vectores podem ter um elemento especial que é o **NA**. Este valor representa um valor desconhecido. Por exemplo, se temos os lucros trimestrais de uma empresa guardados num vector, mas por alguma razão desconhecemos o seu valor no terceiro trimestre, poderíamos usar a seguinte instrução para criar esse vector,

```
> lucros <- c(234000, 245000, NA, 124500)
> lucros

[1] 234000 245000     NA 124500
```

Aceder a um elemento

Como já foi mencionado anteriormente, os elementos de um vector podem ser acedidos através de um índice. Na sua forma mais simples este índice é um número indicando o elemento que pretendemos aceder. Esse número é colocado entre parêntesis rectos a seguir ao nome do vector,

```
> lucros[2]

[1] 245000
```

Alterar um eleme

Usando esta forma de aceder aos elementos individuais de um vector podemos alterar o conteúdo de um elemento particular de um vector,

```
> lucros[3] <- 45000
> lucros

[1] 234000 245000  45000 124500
```

Vectores vazios

O R permite-nos criar vectores vazios usando a função `vector()`,

```
> k <- vector()
```

Alterar tamanho de um vector

O tamanho de um vector já existente pode ser alterado atribuindo mais elementos a índices até agora inexistentes,

```
> k[3] <- 45
> k

[1] NA NA 45
```

Repare que os dois primeiros elementos do vector **k**, que anteriormente era um vector vazio, ficaram com o valor **NA** ao colocarmos o valor 45 no terceiro elemento.

Para diminuirmos o tamanho de um vector podemos usar a instrução de atribuição. Por exemplo,

```
> v <- c(45, 243, 78, 343, 445, 645, 2, 44, 56, 77)
> v

 [1]  45 243  78 343 445 645   2  44  56  77

> v <- c(v[5], v[7])
> v

[1] 445   2
```

Através do uso de formas de indexação mais poderosas que iremos explorar na Secção 2.6 vamos conseguir eliminar mais facilmente elementos particulares de um vector.

2.3 Operações com Vectores

Operações com vectores

Um dos aspectos mais poderosos da linguagem R é a possibilidade de "vectorizar" a maioria das suas funções. Isto quer dizer que a maioria das funções disponíveis no R , ao serem aplicadas a um vector, produzem como resultado um vector de resultados, que é obtido aplicando a função a cada um dos elementos do vector. Vejamos um exemplo com a função `sqrt()` que serve para calcular raízes quadradas,

```
> v <- c(4, 7, 23.5, 76.2, 80)
> x <- sqrt(v)
> x

[1] 2.000000 2.645751 4.847680 8.729261 8.944272
```

Ao atribuir a **x** o resultado da aplicação da função ao vector **v**, estamos de facto a criar um vector com as raízes quadradas dos números contidos em **v**.

Esta característica do R pode ser usada para levar a cabo operações aritméticas envolvendo vectores,

```
> v1 <- c(4, 6, 87)
> v2 <- c(34, 32.4, 12)
> v1 + v2
```

```
[1] 38.0 38.4 99.0
```

O que acontece se tentamos realizar operações envolvendo vectores de tamanho diferente? O R vai usar um regra de *reciclagem* dos valores do vector mais curto até este atingir o tamanho do maior. Por exemplo,

gra da reciclagem

```
> v1 <- c(4, 6, 8, 24)
> v2 <- c(10, 2)
> v1 + v2

[1] 14  8 18 26
```

É como se o vector `c(10,2)` fosse de facto `c(10,2,10,2)`. Se os tamanhos não são múltiplos um do outro, o R imprime um aviso no écran,

```
> v1 <- c(4, 6, 8, 24)
> v2 <- c(10, 2, 4)
> v1 + v2

[1] 14  8 12 34
```

Repare-se que um aviso não é um erro, o que quer dizer que a operação foi levada a cabo.

Como foi mencionado anteriormente, um número é de facto armazenado em R como um vector de tamanho 1. Isto dá bastante jeito para operações como a seguinte,

```
> v1 <- c(4, 6, 8, 24)
> 2 * v1

[1]  8 12 16 48
```

Repare que o número 2 (de facto o vector `c(2)`!) foi reciclado até atingir o tamanho do vector **v1**, tendo como resultado a multiplicação dos elementos todos deste vector por 2. Como veremos mais tarde esta regra de reciclagem também se aplica a outros objectos, como por exemplo matrizes.

2.4 Factores

Os factores proporcionam uma forma fácil e compacta de lidar com dados categóricos (ou nominais). Este tipo de dados podem ser vistos como variáveis que podem tomar como valores possíveis um conjunto finito de etiquetas (por exemplo uma variável que armazene o estado civil de uma pessoa). Os factores têm associados a eles um conjunto de níveis que são os valores possíveis que podem tomar. Como veremos mais tarde várias funções do R (por exemplo funções ligadas à visualização de dados) tiram partido do facto de guardarmos informação categórica como factores em vez de usarmos *strings*.

Variáveis nominais (factores)

Vejamos como criar factores em R. Suponhamos que pretendemos guardar o sexo de 10 indivíduos num vector,

Criar factores

```
> s <- c("f", "m", "m", "m", "f", "m", "f", "m", "f", "f")
> s

 [1] "f" "m" "m" "m" "f" "m" "f" "m" "f" "f"
```

Ao transformarmos um vector de caracteres num factor, o R vai dar-nos a possibilidade de fazer coisas como por exemplo contar quantas vezes ocorre cada valor desse factor. Podemos transformar um vector de caracteres num factor da seguinte forma,

[5] Como o R é sensível às letras maiúsculas / minúsculas, **True**, por exemplo, não é um valor lógico válido.

```
> s <- factor(s)
> s

 [1] f m m m f m f m f f
Levels: f m
```

Repare que **s** deixou de ser um vector de caracteres[6]. Neste pequeno exemplo, o nosso factor tem dois níveis, 'f' e 'm'.

Suponha agora que temos 4 novos indivíduos cujo sexo também pretendemos armazenar. Imagine que por coincidência todos pertencem ao sexo masculino. Se pretendemos que o factor resultante mantenha os 2 níveis possíveis para o sexo de um indivíduo teremos que fazer,

```
> outro.s <- factor(c("m", "m", "m", "m"), levels = c("f", "m"))
> outro.s

[1] m m m m
Levels: f m
```

Sem o parâmetro extra, indicando os níveis a usar, a função `factor()` iria dar origem a um factor com um único nível ('m'), uma vez que este é o único valor que ocorre no vector de caracteres que estamos a transformar num factor.

Conforme mencionado anteriormente, uma das vantagens de transformar um vector de caracteres num factor, é a possibilidade de usar funções específicas para lidar com factores. Uma dessas funções permite-nos contar o número de ocorrências de cada valor (nível),

Contar ocorrências

```
> table(s)

s
f m
5 5

> table(outro.s)

outro.s
f m
0 4
```

Tabulações cruzadas

A função `table()` também pode ser usada para fazer tabulações cruzadas de dois factores, desde que estes tenham o mesmo tamanho. Imaginemos que temos um outro vector com a gama de idades dos indivíduos cujo sexo está armazenado em **s**. Podemos fazer uma tabulação cruzada da idade e do sexo dos 10 indivíduos, da seguinte forma,

```
> idade <- factor(c("adulto", "adulto", "jovem", "jovem", "adulto",
+     "adulto", "adulto", "jovem", "adulto", "jovem"))
> s

 [1] f m m m f m f m f f
Levels: f m

> idade

 [1] adulto adulto jovem  jovem  adulto adulto adulto jovem  adulto jovem
Levels: adulto jovem

> table(idade, s)
```

[6] De facto, **s** é um tipo especial de vector de números, uma vez que internamente, o R guarda os factores como vectores de números, com tantos valores quantos os níveis do factor. Isto pode ser confirmado fazendo `mode(s)`.

```
        s
idade    f m
  adulto 4 2
  jovem  1 3
```

Isto quer dizer que existem 4 indivíduos que são adultos do sexo feminino, 2 que são homens adultos, etc.

Repare como introduzimos a primeira instrução. Como era muito grande, mudamos de linha. Uma vez que o R descobre que a instrução ainda não está terminada, apresenta a linha seguinte com o *prompt* de continuação, que é um sinal mais (+).

Comandos espalhados por várias linhas

Também é possível obter facilmente frequências marginais e relativas. Por exemplo, para saber o número total de adultos e jovens faríamos,

Frequências marginais e relativas

```
> tabela.cruzada <- table(idade, s)
> margin.table(tabela.cruzada, 1)

idade
adulto  jovem
     6      4
```

Para obter a mesma informação relativamente ao sexo dos indivíduos, bastaria mudar o número 1, que indica que pretendemos os totais por linha (a primeira dimensão do objecto) da tabela cruzada, para um 2, para obtermos os totais por coluna da tabela cruzada,

```
> margin.table(tabela.cruzada, 2)

s
f m
5 5
```

Relativamente a frequências relativas, podemos usar a função `prop.table()`, de cujo uso apresentamos alguns exemplos,

```
> prop.table(tabela.cruzada, 1)

        s
idade            f         m
  adulto 0.6666667 0.3333333
  jovem  0.2500000 0.7500000

> prop.table(tabela.cruzada, 2)

        s
idade      f   m
  adulto 0.8 0.4
  jovem  0.2 0.6

> prop.table(tabela.cruzada)

        s
idade      f   m
  adulto 0.4 0.2
  jovem  0.1 0.3
```

No primeiro exemplo, as proporções são calculadas em relação ao total por linha. Assim, o número 0.6666667 é o resultado de dividir 4 (o número de adultos femininos), por 6 (o número total de adultos). No segundo exemplo, as proporções são relativamente aos totais por coluna, enquanto que no terceiro exemplo são relativas ao número total de indivíduos (isto é, ficamos a saber que o número de adultos masculinos representa 20% do total dos indivíduos). Caso pretendêssemos, obter os número em percentagem, poderíamos simplesmente multiplicar as chamadas às funções pelo número 100[7],

[7] Refira-se a título de curiosidade que isto é mais um exemplo da regra de reciclagem.

```
> 100 * prop.table(tabela.cruzada)

        s
idade     f  m
  adulto 40 20
  jovem  10 30
```

2.5 Sequências

Gerar sequências de inteiros

Podem-se gerar sequências em R de várias formas. Por exemplo, imaginemos que pretendemos criar um vector com os número de 1 a 1000. Em vez de os escrevermos todos, podemos usar,

```
> x <- 1:1000
```

o que cria um vector com os número inteiros de 1 a 1000.

Devemos ter algum cuidado com a precedência do operador ":", que é usado para gerar sequências, em relação aos operadores aritméticos. O exemplo seguinte ilustra os perigos que corremos,

```
> 10:15 - 1

[1]  9 10 11 12 13 14

> 10:(15 - 1)

[1] 10 11 12 13 14
```

Repare o que aconteceu no primeiro exemplo. Devido ao operador ":" ter maior precedência do que o "-", o R gerou um vector com os números de 10 a 15 e depois subtraiu a esse vector o número 1. Devido à regra da reciclagem isto correspondeu a subtrair 1 a todos os elementos do vector, levando ao resultado que vemos acima. Já no segundo exemplo, usamos os parêntesis para indicar ao R que o que pretendemos é um vector contendo os números desde 10 até ao resultado da operação 15-1, i.e. 14.

Sequências descendentes

O operador ":" também pode ser usado para gerar sequências descendentes,

```
> 5:0

[1] 5 4 3 2 1 0
```

Sequências de reais

Para gerar sequências com números reais podemos usar a função `seq()`,

```
> seq(-4, 1, 0.5)

 [1] -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0
```

No exemplo acima geramos uma sequência formada pelos números a começar em -4 até 1 de 0.5 em 0.5. Esta função tem várias outras possibilidades para explicitar a sequência que pretendemos. Alguns exemplos são dados a baixo. Poderá também explorar as potencialidades de ajuda do R, digitando o comando `? seq` que mostra o *help* relativamente à função `seq`.

```
> seq(from = 1, to = 5, length = 4)

[1] 1.000000 2.333333 3.666667 5.000000

> seq(from = 1, to = 5, length = 2)

[1] 1 5

> seq(length = 10, from = -2, by = 0.2)
```

```
 [1] -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2
```

mes dos
gumentos

Repare que existe uma diferença subtil na forma como usamos as função `seq()` nos exemplos acima. De facto, em vez de indicarmos os valores dos argumentos da função (como por exemplo em `seq(-4,1,0.5)`, indicamos o nome dos argumentos seguido de um sinal igual e o valor com o qual pretendemos chamar a função. As duas formas são equivalentes, no entanto a chamada por nome tem vantagens de legibilidade e também quando a função tem muitos argumentos permite-nos desrespeitar a sua ordem, ou seja, relativamente aos exemplos anteriores obteríamos o mesmo resultado com `seq(length=4,from=1,to=5)`. Iremos analisar com mais detalhe esta questão quando estudarmos o uso e criação de funções em R na Secção 3.3.1.

Sequências repetidas

Uma outra função bastante útil para gerar sequências é a função `rep()`,

```
> rep(5, 10)

 [1] 5 5 5 5 5 5 5 5 5 5

> rep("sim", 3)

[1] "sim" "sim" "sim"

> rep(1:3, 2)

[1] 1 2 3 1 2 3
```

Sequências com factores

A função `gl()` pode ser usada para gerar sequências envolvendo factores. A sintaxe desta função é `gl(k,n)`, em que `k` é o número de níveis do factor e `n` o número de repetições de cada nível. Vejamos dois exemplos,

```
> gl(3, 5)

 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3

> gl(2, 5, labels = c("nao", "sim"))

 [1] nao nao nao nao nao sim sim sim sim sim
Levels: nao sim
```

Sequências aleatórias

Finalmente, o R tem uma série de funções para gerar sequências aleatórias de acordo com uma série de funções de distribuição de probabilidade. Essas funções têm a forma genérica `r`*`func`*`(n, par1, par2, ...)`, em que `n` é o número de dados a gerar, e `par1, par2, ...` são valores de alguns parâmetros que a função específica a ser usada possa precisar. Por exemplo, se pretendemos 10 números gerados aleatoriamente de acordo com uma distribuição normal de média 0 e desvio padrão unitário, podemos fazer,

```
> rnorm(10)

 [1]  0.6068968  1.2468432  1.8680112  1.3694861 -2.7365075 -0.8887380
 [7]  0.2129164  0.8129556 -0.1786157  0.7735804
```

Se preferirmos 10 números provenientes de uma distribuição normal com média 10 e desvio padrão 3, faríamos

```
> rnorm(10, mean = 10, sd = 3)

 [1] 11.224101 12.993983  9.577280  8.083312  7.553354 13.085718  8.347373
 [8]  8.388759 11.020737  4.095438
```

De igual modo para obter 5 números obtidos de forma aleatória de uma distribuição t de Student com 10 graus de liberdade, fazemos

```
> rt(5, df = 10)

[1]  1.1357666 -2.7568824  0.7445925 -0.3963767  1.2472630
```

O R tem muitas mais funções para outras distribuições de probabilidade, bem como funções semelhantes para obter a densidade de probabilidade, as densidades acumuladas e os quartis das distribuições. No entanto, a exploração exaustiva destas funções sai fora do âmbito desta cadeira, podendo o aluno por exemplo obter mais informação no livro Dalgaard (2002), ou então consultando a ajuda das funções que contém exemplos e referências a funções relacionadas.

2.6 Indexação

Vectores de índices
Vector de índices lógicos

Na Secção 2.2 já vimos exemplos de como extrair um elemento de um vector indicando a sua posição entre parêntesis rectos. O R também nos permite usar vectores dentro desses parêntesis rectos. Estes vectores chamam-se vectores de índices. Existem vários tipos de vectores de índices. Os vectores de índices booleanos extraem de um vector os elementos correspondentes a posições verdadeiras. Vejamos um exemplo concreto,

```
> x <- c(0, -3, 4, -1, 45, 90, -5)
> x

[1]  0 -3  4 -1 45 90 -5

> x > 0

[1] FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE

> y <- x > 0
```

A terceira instrução do código mostrado acima é uma condição lógica envolvendo o vector **x**. Esta condição é uma comparação com o valor zero. Devido à regra de reciclagem, o resultado desta comparação é um vector de valores lógicos resultantes de fazer a comparação para cada elemento do vector **x**. Na instrução seguinte guardamos este vector lógico no objecto **y**. Usando este vector como um vector de índices, podemos obter os elementos de **x** que são maiores do que zero, da seguinte forma,

```
> x[y]

[1]  4 45 90
```

Como os elementos de **y** que são verdadeiros (TRUE) são o 3º, 5º e o 6º, isto corresponde a extrair esses elementos de **x**, que é o resultado que obtemos com a instrução mostrada acima. Poderíamos obter um resultado igual, evitando criar um novo vector, fazendo

```
> x[x > 0]

[1]  4 45 90
```

Tirando partido da gama de operadores lógicos disponíveis no R, podemos construir vectores de indexação lógicos mais complexos,

```
> x[x <= -2 | x > 5]

[1] -3 45 90 -5

> x[x > 40 & x < 100]

[1] 45 90
```

eradores lógicos

O operador "|" corresponde ao OU (disjunção) lógico, enquanto o operador "&" é o E (conjunção) lógica. Isto quer dizer que a primeira instrução dá como resultado os elementos de **x** que são menores ou iguais a -2, ou maiores do que 5. O segundo exemplo, por sua vez, mostra-nos os elementos de **x** que são maiores do que 40 e menores do que 100. Note que existem duas variantes destes dois operadores lógicos com um modo de funcionamento ligeiramente diferente. Eles são os operadores "||" e "&&". Ambos fazem as mesmas operações que os anteriores, o OU e o E lógicos. A diferença reside no facto de os operadores anteriores fazerem essas operações de forma vectorial, ou seja elemento a elemento quando os operandos contêm mais do que um elemento, o que permite por exemplo aplicar as operações sobre conjuntos de números, como nos exemplos mostrados em cima. Ou seja estes operadores produzem um conjunto de resultados (valores T ou F). Já as variantes com dois símbolos, só fazem a operação sobre o primeiro elemento, ou seja procedem da esquerda para a direita, produzindo um único resultado. Estas últimas variantes são essencialmente usadas em instruções de controlo da execução de programas, como iremos ver na Secção 3.2.1.

Vector de índices numéricos

O R também nos permite usar um vector de números inteiros como índice de um outro vector. Os números desse vector de índices correspondem aos elementos a extrair do outro vector,

```
> (v <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j"))

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

> v[c(4, 6)]

[1] "d" "f"

> v[1:3]

[1] "a" "b" "c"
```

Índices negativos

Podemos ainda usar um vector com números negativos, o que nos permite indicar quais os elementos a não obter como resultado da indexação,

```
> v[-1]

[1] "b" "c" "d" "e" "f" "g" "h" "i" "j"

> v[-c(4, 6)]

[1] "a" "b" "c" "e" "g" "h" "i" "j"

> v[-(1:3)]

[1] "d" "e" "f" "g" "h" "i" "j"
```

Note bem a importância dos parêntesis no último exemplo, devido à precedência do operador ":" já mencionada na Secção 2.5.

Índices com nomes

Um vector também pode ser indexado por um vector de *strings* tirando partido do facto de o R permitir dar nomes aos elementos de um vector através na função `names()`. Vectores com os elementos com nomes são por vezes preferíveis pois as suas "posições" são mais facilmente memorizáveis. Suponhamos que tínhamos um vector com as taxas de inflação de 5 países europeus. Poderíamos criar um vector com nomes da seguinte forma,

Dar nomes a elementos de vectores

```
> tx.infl <- c(2.5, 2, 2.1, 2.3, 1.9)
> names(tx.infl) <- c("Portugal", "França", "Espanha", "Itália",
+     "Alemanha")
> tx.infl
```

```
Portugal   França  Espanha   Itália Alemanha
     2.5      2.0      2.1      2.3      1.9
```

Os elementos do vector **tx.infl** podem agora ser acedidos usando os seus nomes,

```
> tx.infl["Portugal"]

Portugal
     2.5

> tx.infl[c("Espanha", "Alemanha")]

 Espanha Alemanha
     2.1      1.9
```

Índices vazios

Finalmente, os índices podem ser vazios, o que tem o significado que todos os elementos são seleccionados. Por exemplo, se pretendemos preencher todas as posições de um vector com zeros podemos fazer `x[] <- 0`. De notar, que isto é diferente de fazer `x <- 0`, que teria como efeito atribuir ao vector **x** um vector com um único elemento (zero), enquanto que a primeira alternativa, assumindo que o vector **x** já existia, iria substituir todos os seus elementos por zero. Tente ambas as alternativas.

2.7 Matrizes

O que é uma matriz

Por vezes poderemos estar interessados em armazenar a nossa informação em estruturas de dados com mais do que uma dimensão, como é o caso dos vectores. As matrizes arranjam a informação em duas dimensões. Em R, as matrizes não são mais do que vectores com uma propriedade especial que é a dimensão. Vejamos um exemplo. Suponhamos que temos doze números correspondentes às vendas trimestrais durante o último ano, em três lojas. As instruções seguintes permitem "organizar" esses números como uma matriz,

Criar uma matriz

```
> vendas <- c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23, 78, 90)
> vendas

 [1] 45 23 66 77 33 44 56 12 78 23 78 90

> dim(vendas) <- c(3, 4)
> vendas

     [,1] [,2] [,3] [,4]
[1,]   45   77   56   23
[2,]   23   33   12   78
[3,]   66   44   78   90
```

Repare como os números foram "espalhados" por uma matriz com 3 linhas e 4 colunas, que foi a dimensão que atribuímos ao vector **vendas** através da função `dim()`. Na realidade seria mais simples criar a matriz usando uma função específica para isso,

```
> vendas <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23, 78,
+     90), 3, 4)
```

Como eventualmente terá reparado os números foram "espalhados" pela matriz por coluna, i.e. primeiro foi preenchida a primeira coluna, depois a segunda, etc. Caso não seja isto o que pretendemos, poderemos preencher a matriz por linhas da seguinte forma,

```
> vendas <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23, 78,
+     90), 3, 4, byrow = T)
> vendas
```

```
     [,1] [,2] [,3] [,4]
[1,]   45   23   66   77
[2,]   33   44   56   12
[3,]   78   23   78   90
```

Nas matrizes também é possível dar nomes aos elementos para tornar a leitura da informação mais legível. Vejamos como fazer isso para a nossa matriz de vendas trimestrais nas 3 lojas,

Dar nomes às linhas e colunas

```
> rownames(vendas) <- c("loja1", "loja2", "loja3")
> colnames(vendas) <- c("1.trim", "2.trim", "3.trim", "4.trim")
> vendas

      1.trim 2.trim 3.trim 4.trim
loja1     45     23     66     77
loja2     33     44     56     12
loja3     78     23     78     90
```

Como a visualização das matrizes sugere, podemos aceder aos elementos individuais das matrizes usando um esquema de indexação semelhante ao dos vectores, mas desta vez com dois índices (as dimensões da matriz),

Aceder aos elementos da matriz

```
> vendas[2, 2]

[1] 44
```

Ou então, tirando partido dos nomes,

```
> vendas["loja2", "2.trim"]

[1] 44
```

De igual modo, podemos tirar partido dos esquemas de indexação discutidos na Secção 2.6 para seleccionar elementos das matrizes, como mostram os seguintes exemplos,

```
> vendas[-2, 2]

loja1 loja3
   23    23

> vendas[1, -c(2, 4)]

1.trim 3.trim
    45     66
```

Podemos mesmo omitir uma das dimensões das matrizes para deste modo obter todos os elementos da mesma (um índice vazio),

```
> vendas[1, ]

1.trim 2.trim 3.trim 4.trim
    45     23     66     77

> vendas[, "4.trim"]

loja1 loja2 loja3
   77    12    90
```

As funções `cbind()` e `rbind()` podem ser usadas para juntar dois ou mais vectores ou matrizes, por colunas ou por linhas, respectivamente. Os seguintes exemplos ilustram o seu uso,

```
> m1 <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5)
> m1

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23

> cbind(c(4, 76), m1[, 4])

     [,1] [,2]
[1,]    4   56
[2,]   76   12

> m2 <- matrix(rep(10, 50), 10, 5)
> m2

      [,1] [,2] [,3] [,4] [,5]
 [1,]   10   10   10   10   10
 [2,]   10   10   10   10   10
 [3,]   10   10   10   10   10
 [4,]   10   10   10   10   10
 [5,]   10   10   10   10   10
 [6,]   10   10   10   10   10
 [7,]   10   10   10   10   10
 [8,]   10   10   10   10   10
 [9,]   10   10   10   10   10
[10,]   10   10   10   10   10

> m3 <- rbind(m1[1, ], m2[5, ])
> m3

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   10   10   10   10   10
```

Reciclagem com matrizes

As regras aritméticas e de reciclagem que estudamos anteriormente, também se aplicam às matrizes. Vejamos uns pequenos exemplos,

```
> m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5)
> m

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23

> m * 3

     [,1] [,2] [,3] [,4] [,5]
[1,]  135  198   99  168  234
[2,]   69  231  132   36   69

> m1 <- matrix(c(45, 23, 66, 77, 33, 44), 2, 3)
> m1

     [,1] [,2] [,3]
[1,]   45   66   33
[2,]   23   77   44

> m2 <- matrix(c(12, 65, 32, 7, 4, 78), 2, 3)
> m2
```

```
     [,1] [,2] [,3]
[1,]   12   32    4
[2,]   65    7   78

> m1 + m2

     [,1] [,2] [,3]
[1,]   57   98   37
[2,]   88   84  122
```

A aplicação das operações a matrizes (como no exemplo "`> m*3`" apresentado acima), funciona elemento a elemento como no caso dos vectores. Isto significa que se um operando é menor ele é reciclado até perfazer o tamanho do maior. No entanto, o R também possui operadores especiais para as usuais operações da álgebra matricial. Por exemplo a multiplicação de duas matrizes pode ser feita da seguinte forma,

Multiplicação matricial

```
> m1 <- matrix(c(45, 23, 66, 77, 33, 44), 2, 3)
> m1

     [,1] [,2] [,3]
[1,]   45   66   33
[2,]   23   77   44

> m2 <- matrix(c(5, 3, 466, 54.5, 3.2, -34), 3, 2)
> m2

     [,1]  [,2]
[1,]    5  54.5
[2,]    3   3.2
[3,]  466 -34.0

> m1 %*% m2

      [,1]   [,2]
[1,] 15801 1541.7
[2,] 20850    3.9
```

Atente no operador especial (`%*%`) para simbolizar que se trata da multiplicação matricial e não a usual multiplicação. A multiplicação matricial tem, como é sabido, regras especiais no que concerne, por exemplo, à dimensão das matrizes envolvidas, pelo que não poderá ser usada com quaisquer matrizes.

Ainda no contexto da álgebra matricial, o R tem muitas outras funções, como por exemplo a função `t()` para obter a transposta de uma matriz quadrada,

Transposta de uma matriz

```
> t(m1)

     [,1] [,2]
[1,]   45   23
[2,]   66   77
[3,]   33   44
```

ou a função `det()` para calcular o determinante de uma matriz,

Determinante de uma matriz

```
> m <- matrix(c(34, -23, 43, 5), 2, 2)
> det(m)

[1] 1159
```

É também possível usar a função `solve()` para obter a inversa de uma matriz,

Inversa de uma matriz

```
> solve(m)

            [,1]        [,2]
[1,] 0.004314064 -0.03710095
[2,] 0.019844694  0.02933563
```

Resolver sistemas de equações

Finalmente, esta mesma função pode ser usada para resolver sistemas de equações lineares. Imaginemos o seguinte sistema de equações,

$$\begin{cases} -4x + 0.3y &= 12.3 \\ 54.3x - 4y &= 45 \end{cases}$$

Podemos resolvê-lo em R da seguinte forma,

```
> coefs <- matrix(c(-4, 0.3, 54.3, -4), 2, 2, byrow = T)
> ys <- c(12.3, 45)
> solve(coefs, ys)

[1]  216.2069 2923.7586
```

2.8 *Arrays*

Os *arrays* são extensões das matrizes para mais do que duas dimensões. Isto significa que podem ter vários índices. À parte esta pequena diferença, eles podem ser usados da mesma forma do que as matrizes. De modo semelhante à função `matrix`, existe uma função `array()` para facilitar a criação de *arrays*. Segue-se um pequeno exemplo,

Criar arrays

```
> a <- array(1:50, dim = c(2, 5, 5))
> a

, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   11   13   15   17   19
[2,]   12   14   16   18   20

, , 3

     [,1] [,2] [,3] [,4] [,5]
[1,]   21   23   25   27   29
[2,]   22   24   26   28   30

, , 4

     [,1] [,2] [,3] [,4] [,5]
[1,]   31   33   35   37   39
[2,]   32   34   36   38   40

, , 5

     [,1] [,2] [,3] [,4] [,5]
[1,]   41   43   45   47   49
[2,]   42   44   46   48   50
```

Indexar arrays

Os esquemas usuais de indexação que vimos para vectores e matrizes, podem também ser aplicados aos *arrays*.

2.9 Listas

Uma lista é uma colecção ordenada de objectos conhecidos como os componentes da lista. Esses componentes não necessitam de ser do mesmo tipo, modo ou tamanho. Os componentes de uma lista em R são sempre numerados e podem também ter um nome associados a eles. Vejamos um pequeno exemplo de criação de uma lista em R,

O que é uma lista

Criar uma lista

```
> estudante <- list(nro = 34453, nome = "Carlos Silva", notas = c(14.3,
+     12, 15, 19))
```

O objecto **estudante** é uma lista formada por três componentes. Um é um número e tem o nome **nro**, outro é uma *string* e tem o nome **nome**, e o terceiro componente é um vector de números com o nome **notas**.

Podemos extrair componentes específicos de uma lista através da seguinte sintaxe,

Extrair componentes de uma lista

```
> estudante[[1]]

[1] 34453

> estudante[[3]]

[1] 14.3 12.0 15.0 19.0
```

Como certamente terá reparado, usamos parêntesis rectos duplos. Se, por exemplo, tivéssemos indicado `estudante[1]`, teríamos obtido um resultado diferente. Neste último caso teríamos uma sub-lista formada pelo primeiro componente da lista **estudante**. Pelo contrário, `estudante[[1]]` obtém o conteúdo do primeiro componente da lista **estudante**, que neste caso é um número, e logo não é uma lista.

No caso de listas com componentes com nomes, como o caso da lista **estudante**, podemos extrair os componentes da lista usando uma forma alternativa,

```
> estudante$nro

[1] 34453
```

Podemos ver todo o conteúdo da lista (como aliás de qualquer objecto no R), escrevendo o seu nome,

Mostrar conteúdo da lista

```
> estudante

$nro
[1] 34453

$nome
[1] "Carlos Silva"

$notas
[1] 14.3 12.0 15.0 19.0
```

Os nomes dos componentes de uma lista são de facto uma propriedade da lista e portanto podemos manipulá-los como fizemos com os nomes dos elementos de um vector,

```
> names(estudante)

[1] "nro"   "nome"  "notas"

> names(estudante) <- c("número", "nome", "notas")
> estudante
```

```
$número
[1] 34453

$nome
[1] "Carlos Silva"

$notas
[1] 14.3 12.0 15.0 19.0
```

Extender listas

As listas podem ser extendidas acrescentando-lhes novos componentes,

```
> estudante$pais <- c("Ana Castro", "Miguel Silva")
> estudante

$número
[1] 34453

$nome
[1] "Carlos Silva"

$notas
[1] 14.3 12.0 15.0 19.0

$pais
[1] "Ana Castro"   "Miguel Silva"
```

Número de componentes de uma lista

Podemos saber quantos componentes tem uma lista usando a função `length()`,

```
> length(estudante)

[1] 4
```

Juntar listas

Podemos juntar duas ou mais listas numa só usando a função `c()`,

```
> resto <- list(idade = 19, sexo = "masculino")
> estudante <- c(estudante, resto)
> estudante

$número
[1] 34453

$nome
[1] "Carlos Silva"

$notas
[1] 14.3 12.0 15.0 19.0

$pais
[1] "Ana Castro"   "Miguel Silva"

$idade
[1] 19

$sexo
[1] "masculino"
```

2.10 *Data Frames*

que é um data me

Um *data frame* é um objecto do R que é normalmente usado para guardar tabelas de dados de um problema qualquer. Na sua forma, um *data frame*, é muito semelhante a uma matriz, mas as suas colunas têm nomes e podem conter dados de tipo diferente, contráriamente a uma matriz. Um *data frame* pode ser visto como uma tabela de uma base de dados, em que cada linha corresponde a um registo (linha) da tabela. Cada coluna corresponde às propriedades (campos) a serem armazenadas para cada registo da tabela.

Criar um data frame

Podemos criar um *data frame* da seguinte forma,

```
> notas.inform <- data.frame(nros = c(2355, 3456, 2334, 5456),
+     turma = c("tp1", "tp1", "tp2", "tp3"), notas = c(10.3, 9.3,
+         14.2, 15))
> notas.inform

  nros turma notas
1 2355   tp1  10.3
2 3456   tp1   9.3
3 2334   tp2  14.2
4 5456   tp3  15.0
```

Aceder aos elementos de um data frame

Os elementos de um *data frame* podem ser acedidos como uma matriz,

```
> notas.inform[2, 2]

[1] tp1
Levels: tp1 tp2 tp3
```

Atente no facto que a função `data.frame()` transformou a coluna **turma** num factor. Isto é feito sempre que os elementos de uma coluna sejam todos do tipo *string*, como é o caso.

Indexar data frames

Os esquemas de indexação descritos na Secção 2.6 também podem ser usados com os *data frames*. Adicionalmente, as colunas dos *data frames* podem ser acedidas na sua totalidade usando o seu nome,

```
> notas.inform$nros

[1] 2355 3456 2334 5456
```

Usando os esquemas de indexação envolvendo condições lógicas, que foram descritos na Secção 2.6, podemos fazer consultas mais complexas aos dados guardados num *data frame*. Prosseguindo a analogia com tabelas de bases de dados, os exemplos seguintes podem ser visto como *queries* a uma tabela,

```
> notas.inform[notas.inform$notas > 10, ]

  nros turma notas
1 2355   tp1  10.3
3 2334   tp2  14.2
4 5456   tp3  15.0

> notas.inform[notas.inform$notas > 14, "nros"]

[1] 2334 5456

> notas.inform[notas.inform$turma == "tp1", c("nros", "notas")]

  nros notas
1 2355  10.3
2 3456   9.3
```

As funções attach e detach()

As consultas a *data frames* apresentadas acima, podem ser simplificadas através do uso da função `attach()`. Sem entrar em detalhes técnicos, podemos dizer que esta função nos permite aceder directamente aos valores nas colunas de um *data frame* sem para isso necessitarmos de colocar o nome deste atrás do nome da coluna, como foi feito nos exemplos acima. Vejamos como isto funciona,

```
> attach(notas.inform)

        The following object(s) are masked from notas.inform ( position 4 ) :

         notas nros turma

> notas.inform[notas > 14, ]

  nros turma notas
3 2334   tp2  14.2
4 5456   tp3  15.0

> turma

[1] tp1 tp1 tp2 tp3
Levels: tp1 tp2 tp3
```

O efeito da função `attach()` é o mesmo que se obteria se criássemos uns objectos com o nome das colunas, contendo os dados das mesmas (por exemplo fazendo "`> notas <- notas.inform$notas`"). A função `detach()` tem o efeito inverso, e deve ser executada quando já não precisamos mais deste acesso "directo" às colunas,

```
> detach(notas.inform)
> turma
Error: Object "turma" not found
```

Repare que se a seguir a fazermos o *detach* tentarmos usar os nomes das colunas, como o fizemos anteriormente, vamos gerar um erro, como se vê no exemplo acima.

Acrescentar colunas a um data frame

É possível acrescentar novas colunas a um *data frame*,

```
> notas.inform$resultado <- c("aprovado", "oral", "aprovado", "aprovado")
> notas.inform

  nros turma notas resultado
1 2355   tp1  10.3  aprovado
2 3456   tp1   9.3      oral
3 2334   tp2  14.2  aprovado
4 5456   tp3  15.0  aprovado
```

Número de linhas e colunas

A única restrição a este tipo de acrescentos é a de que a nova coluna deverá ter tantos elementos quantas as linhas do *data frame*. Podemos saber o número de linhas e colunas de um *data frame* da seguinte forma,

```
> nrow(notas.inform)

[1] 4

> ncol(notas.inform)

[1] 4
```

Introduzir dados num data frame

Como veremos em secções posteriores deste texto de apoio, na maioria dos casos não iremos escrever os dados a guardar num *data frame* "à mão" usando a função `data.frame()`, como fizemos anteriormente. De facto, na maioria das situações iremos buscar os nossos dados ou a uma base de dados, ou a um ficheiro de texto, ou mesmo a uma fonte de dados disponível na Internet. De qualquer modo, sempre que pretendemos introduzir nós próprios os dados podemos usar um interface tipo folha de cálculo que o R possui, para facilitar a nossa tarefa. Vejamos como,

```
> outras.notas <- data.frame()
> outras.notas <- edit(outras.notas)
```

A primeira instrução serve para dizer ao R que o objecto **outras.notas** é um *data frame* (vazio de dados, neste momento). A segunda instrução invoca o referido interface tipo folha de cálculo com esse objecto, e coloca o resultado da nossa edição nesse mesmo objecto, ou seja as alterações que fizermos na folha de cálculo serão armazenadas no objecto **outras.notas**. O interface da função `edit()` permite-nos acrescentar facilmente valores nas colunas do *data frame* bastando para isso *clicar* em cima de uma célula e começar a escrever o seu valor, bem como dar nomes às colunas do *data frame*, *clicando* em cima do seu nome. Experimente.

2.11 Séries Temporais

Existem inúmeras aplicações onde os dados que são recolhidos estão etiquetados pelo tempo. Quer isto dizer que existe uma ordem temporal entre as diferentes observações de uma ou mais variáveis. Estes tipos de dados são normalmente conhecidos como séries temporais.

O que é uma série temporal?

Uma série temporal (univariada) é um conjunto de observações de uma variável Y ao longo do tempo, $y_1, y_2, \ldots, y_{t-1}, y_t$. Conforme seria previsível o R tem várias facilidades para lidar com este tipo de dados. Na nossa exposição vamos distinguir dois tipos de séries: regulares e irregulares. Este tipo de séries vão ser armazenados em objectos de diferente tipo em R, conforme vamos ver.

2.11.1 Séries Regulares

Séries regulares

Numa série temporal regular todas as observações estão igualmente espaçadas no tempo. Quer isto dizer que a diferença temporal entre quaisquer duas observações é sempre a mesma.

Para este tipo de dados o R disponibiliza os objectos da classe `ts` ou `mts`, para séries multivariadas. Este tipo de objectos permite guardar dados referentes a séries temporais regulares. Vejamos exemplos da criação deste tipo de objectos:

Criar séries regulares

```
> ts(rnorm(10), start = 1990, frequency = 1)

Time Series:
Start = 1990
End = 1999
Frequency = 1
 [1] -0.007145366  0.286113853 -0.178509296  1.110593387 -0.537384054
 [6]  0.235266885 -0.347034238  0.186546117 -0.560699688 -0.939207908

> ts(rnorm(10), frequency = 4, start = c(1959, 2))

           Qtr1        Qtr2        Qtr3        Qtr4
1959              -0.06177427 -0.39010380  0.52328884
1960 -0.52024659  0.79999330  0.11223208  1.08645298
1961  2.64958972  0.27428917  1.44606713

> ts(rnorm(10), frequency = 12, start = c(1959, 2))

             Feb         Mar         Apr         May         Jun         Jul
1959  1.86124434 -1.21152214 -1.25001426 -0.65898372  0.87284088 -1.32159856
             Aug         Sep         Oct         Nov
1959 -1.82540080 -0.02475364  1.51389043 -1.20735747
```

Como vemos, a função principal para criar este tipo de objectos é a função `ts()`. Esta função recebe no primeiro argumento as observações da série e depois tem um conjunto de argumentos que podem ser usados para explicitar os intervalos de tempo regulares a que

se observaram esses valores. Assim no primeiro exemplo, indicamos que o primeiro valor é observado no tempo 1990 e que entre cada unidade de tempo (i.e. 1990, 1991, 1992, etc., uma vez que é assumido um incremento de 1 entre as unidades de tempo) a variável é amostrada uma única vez. Já no segundo exemplo, indicamos que a série começa no segundo período de amostragem da unidade de tempo 1959, e que a série é amostrada 4 vezes entre cada unidade. Isto leva o R a interpretar esta série como uma série de valores trimestrais, o que por sua vez determina a escolha de uma forma adequada para representar esta série. No último exemplo, indicamos que a série é amostrada 12 vezes entre cada unidade de tempo o que leva o R a assumir amostragens mensais entre cada ano, e mais uma vez determina uma maneira adequada de a representar.

Séries multivariadas

No caso de séries multivariadas podemos criá-las de igual forma:

```
> (m <- ts(matrix(rnorm(30), 10, 3), start = c(1961, 6), frequency = 12))

            Series 1   Series 2   Series 3
Jun 1961 -0.59624245  1.0808275  0.5293899
Jul 1961  0.29513818 -0.9746566 -0.7295948
Aug 1961  0.61927156 -0.1127870  1.2087357
Sep 1961 -0.04721077  1.0725438  1.1728149
Oct 1961 -1.20085735  0.1319664  0.4304150
Nov 1961 -0.85436753  0.3550235 -0.9931418
Dec 1961 -1.21451369 -1.6505330 -1.1396180
Jan 1962 -0.44169035  0.4167947  0.6371993
Feb 1962  0.91459593  0.1500598 -0.6730498
Mar 1962 -0.36624088 -0.2471749  0.6283073
```

Repare como foram "tratadas" de maneira diferente, em termos de apresentação, pelo R.

Gráficos de séries

Relativamente à representação gráfica das séries temporais, podemos usar de igual modo a muitos objectos, a função `plot()`, como vemos no seguinte exemplo,

```
> k <- ts(rnorm(100), frequency = 4, start = c(1959, 2))
> plot(k)
```

que produz o gráfico apresentado na Figura 3.

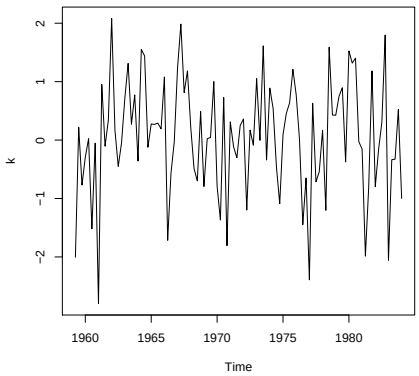


Figura 3: Gráfico de uma série temporal univariada.

No caso de o objecto em cause ser uma série multivariada a função `plot()` produz gráficos separados para cada série como vemos na Figura 4.

```
> m <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 6), frequency = 12)
> plot(m)
```

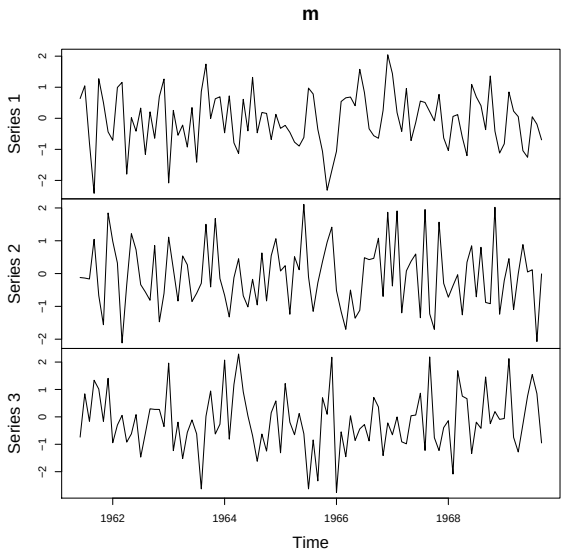


Figura 4: Gráfico de uma série temporal multivariada.

Através do parâmetro "plot.type" podemos indicar que pretendemos um só gráfico com todas as séries,

```
> m <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 6), frequency = 12)
> plot(m, plot.type = "single", col = 1:3)
> legend("topright", legend = colnames(m), col = 1:3, lty = 1)
```

o que leva ao gráfico na Figura 5.

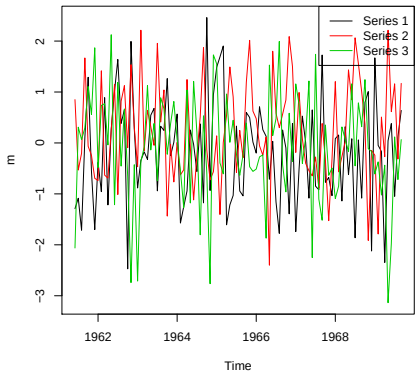


Figura 5: Gráfico único de uma série temporal multivariada.

Existem uma série de funções para manipular os objectos criados pela funçao `ts()` que podem ter grande utilidade. Vejamos alguns exemplos,

Algumas funções úteis

```
> x <- ts(rnorm(10), frequency = 4, start = c(1959, 2))
> start(x)

[1] 1959    2

> end(x)
```

```
[1] 1961    3

> window(x, start = c(1959, 5))

            Qtr1        Qtr2        Qtr3        Qtr4
1960  0.96364972 -1.42112904 -0.64183730 -0.24208549
1961  1.39783879  0.24219084 -0.05084689

> window(x, end = c(1959, 7))

           Qtr1       Qtr2       Qtr3       Qtr4
1959             0.9245848 -0.0970050  0.8055003
1960  0.9636497 -1.4211290 -0.6418373

> window(x, start = c(1959, 4), end = c(1959, 9))

           Qtr1       Qtr2       Qtr3       Qtr4
1959                                   0.8055003
1960  0.9636497 -1.4211290 -0.6418373 -0.2420855
1961  1.3978388
```

A função `start()` permite-nos obter a etiqueta temporal do início da série, enquanto que a função `end()` faz o contrário. Quanto à função `window()`, ela permite-nos extraír uma janela temporal da série, explicitando o início e o fim dessa janela, ou omitindo um desses tempos o que faz com que seja assumido o início ou o final de toda a série, respectivamente.

Lags e diferenças

```
> (x <- ts(rnorm(10), frequency = 4, start = c(1959, 2)))

            Qtr1        Qtr2        Qtr3        Qtr4
1959             -0.53646433 -0.67589743 -0.43825372
1960  0.60812619  0.26228902 -0.03652399  0.43490010
1961  1.47686955 -1.24983261  1.44151984

> lag(x)

            Qtr1        Qtr2        Qtr3        Qtr4
1959 -0.53646433 -0.67589743 -0.43825372  0.60812619
1960  0.26228902 -0.03652399  0.43490010  1.47686955
1961 -1.24983261  1.44151984

> lag(x, 4)

            Qtr1        Qtr2        Qtr3        Qtr4
1958             -0.53646433 -0.67589743 -0.43825372
1959  0.60812619  0.26228902 -0.03652399  0.43490010
1960  1.47686955 -1.24983261  1.44151984

> diff(x)

           Qtr1       Qtr2       Qtr3       Qtr4
1959                       -0.1394331  0.2376437
1960  1.0463799 -0.3458372 -0.2988130  0.4714241
1961  1.0419695 -2.7267022  2.6913525

> diff(x, differences = 2)

            Qtr1        Qtr2        Qtr3        Qtr4
1959                                    0.37707682
1960  0.80873619 -1.39221706  0.04702415  0.77023710
1961  0.57054536 -3.76867161  5.41805462
```

```
> diff(x, lag = 2)

            Qtr1        Qtr2        Qtr3        Qtr4
1959                                        0.09821061
1960  1.28402362  0.70054274 -0.64465017  0.17261108
1961  1.51339354 -1.68473271 -0.03534971
```

A função `lag()` permite-nos fazer um "shift" da série para trás (por defeito) ou para a frente x unidades de tempo (o segundo argumento da função). Quanto à função `diff()` ela permite-nos calcular diferenças de vária ordem (parâmetro "differences") entre valores sucessivos, ou mesmo valores distanciados mais unidades de tempo (parâmetro "lag").

Juntar séries numa só

Por vezes estamos interessados em juntar várias séries num só objecto, isto é numa série multivariada. A maior dificuldade desta tarefa decorre da possibilidade de não haver total concordância das etiquetas temporais das séries que pretendemos juntar. Ao utlizarmos objectos da classe `ts` o R vai encarregar-se de resolver este problema por nós! Vejamos um exemplo:

```
> (x <- ts(rnorm(10), frequency = 4, start = c(1959, 2)))

            Qtr1        Qtr2        Qtr3        Qtr4
1959              1.01532292  1.50409802 -0.59415289
1960  0.28531497  0.45710867  0.28391136  0.16622712
1961  0.03461353 -0.72061564  0.08729295

> (y <- ts(rnorm(10), frequency = 4, start = c(1960, 1)))

             Qtr1         Qtr2         Qtr3         Qtr4
1960 -0.041664330 -0.058680340 -1.360239359  0.897174604
1961  0.008411543  0.279581887 -0.478224920 -0.370200800
1962  1.162627682  0.134785203

> cbind(x, y)

                  x           y
1959 Q2  1.01532292          NA
1959 Q3  1.50409802          NA
1959 Q4 -0.59415289          NA
1960 Q1  0.28531497 -0.041664330
1960 Q2  0.45710867 -0.058680340
1960 Q3  0.28391136 -1.360239359
1960 Q4  0.16622712  0.897174604
1961 Q1  0.03461353  0.008411543
1961 Q2 -0.72061564  0.279581887
1961 Q3  0.08729295 -0.478224920
1961 Q4          NA -0.370200800
1962 Q1          NA  1.162627682
1962 Q2          NA  0.134785203
```

A função `cbind()`, quando aplicada a séries temporais vai fazer uma junção delas olhando para as respectivas etiquetas temporais e preenchendo os valores não concordantes com NA's.

"Embed's" temporais

A função `embed()`, por sua vez, permite gerar diferentes colunas formadas pela série temporal deslizada diferentes passos temporais para trás. Vejamos um exemplo,

```
> (x <- ts(rnorm(10), frequency = 4, start = c(1959, 2)))

            Qtr1        Qtr2        Qtr3        Qtr4
1959             -1.26304477  0.52757493 -1.66384494
1960  1.79545393  0.07105220 -0.08895523 -1.77403572
1961  0.82165185  0.14742042  1.33359531
```

```
> embed(x, 3)

            [,1]        [,2]        [,3]
[1,] -1.66384494  0.52757493 -1.26304477
[2,]  1.79545393 -1.66384494  0.52757493
[3,]  0.07105220  1.79545393 -1.66384494
[4,] -0.08895523  0.07105220  1.79545393
[5,] -1.77403572 -0.08895523  0.07105220
[6,]  0.82165185 -1.77403572 -0.08895523
[7,]  0.14742042  0.82165185 -1.77403572
[8,]  1.33359531  0.14742042  0.82165185
```

Note, em primeiro lugar que o resultado é uma matriz e não uma série multivariada. Na matriz resultante, a linha i tem os valores $Y_{i+e-1}, Y_{i+e-2}, \cdots, Y_i$ da série temporal, em que e é o tamanho do "embed" indicado no segundo argumento da função.

Um efeito semelhante, mas em que o resultado é uma série multivariada, pode ser obtido da seguinte forma,

```
> x

            Qtr1        Qtr2        Qtr3        Qtr4
1959              -1.26304477  0.52757493 -1.66384494
1960  1.79545393  0.07105220 -0.08895523 -1.77403572
1961  0.82165185  0.14742042  1.33359531

> na.omit(cbind(lag(x, 2), lag(x), x))

          lag(x, 2)      lag(x)           x
1959 Q2 -1.66384494  0.52757493 -1.26304477
1959 Q3  1.79545393 -1.66384494  0.52757493
1959 Q4  0.07105220  1.79545393 -1.66384494
1960 Q1 -0.08895523  0.07105220  1.79545393
1960 Q2 -1.77403572 -0.08895523  0.07105220
1960 Q3  0.82165185 -1.77403572 -0.08895523
1960 Q4  0.14742042  0.82165185 -1.77403572
1961 Q1  1.33359531  0.14742042  0.82165185
```

Note como a função `na.omit()` foi usada para retirar do resultado do `cbind()` as "linhas" em que existiam NA's.

2.11.2 Séries Irregulares

Nas séries irregulares os valores não têm necessariamente que ser espalhados de forma igual ao longo do tempo. Isto quer dizer que as etiquetas temporais não têm necessariamente que ter um regularidade como acontece com as séries regulares que vimos anteriormente.

O R tem várias packages que definem objectos específicos para armazenar este tipo de dados. A lista seguinte apresenta as principais:

- A package `its` que define os objectos "its".
- A package `tseries` que define os objectos "irts".
- A package `fBasics` que define os objectos "timeSeries".
- A package `zoo` que define os objectos "zoo".

Criar uma série irregular

No nosso estudo vamos usar a package "zoo". Vejamos como criar um objecto "zoo" em R:

```
> library(zoo)
> (x <- zoo(rnorm(5), c("2006-3-21", "2006-3-24", "2006-6-21",
+     "2006-6-23", "2006-09-21")))

 2006-09-21   2006-3-21   2006-3-24   2006-6-21   2006-6-23
-2.45827056 -1.13546300  0.10284559 -0.57846242 -0.04002960

> index(x)

[1] "2006-09-21" "2006-3-21"  "2006-3-24"  "2006-6-21"  "2006-6-23"

> coredata(x)

[1] -2.45827056 -1.13546300  0.10284559 -0.57846242 -0.04002960
```

O segundo argumento da função `zoo()`, as etiquetas temporais dos dados, pode ser de qualquer tipo (!), desde que faça sentido ordenar os seus valores (isto é desde que se possa aplicar a função `order()` aos valores). Fora isto, não há qualquer limitação o que torna este tipo de objectos bastante flexíveis.

A maior parte das vezes as etiquetas das nossas séries temporais vão ser datas/horas. Nesse contexto, vamos agora ver maneiras alternativas de lidar com datas e/ou horas em R.

Datas/horas em R

O R tem vários tipos de objectos para lidar com datas e tempo em geral. Vejamos alguns exemplos:

1. Os objectos do tipo "Date"

   Objectos do tipo "Date"

   Neste tipo de objectos as datas são representadas internamente como o número de dias que passaram desde 1970-01-01.

   ```
   > Sys.Date()

   [1] "2006-10-21"

   > hoje <- Sys.Date()
   > format(hoje, "%d %b %Y")

   [1] "21 Out 2006"

   > (dezsemanas <- seq(hoje, len = 10, by = "1 week"))

    [1] "2006-10-21" "2006-10-28" "2006-11-04" "2006-11-11" "2006-11-18"
    [6] "2006-11-25" "2006-12-02" "2006-12-09" "2006-12-16" "2006-12-23"

   > weekdays(hoje)

   [1] "sábado"

   > months(hoje)

   [1] "Outubro"
   ```

   A função `Sys.Date()` permite-nos obter a data de hoje num objecto do tipo "Date". A função `format()` permite mostrar um objecto deste tipo de muitas formas, através de um segundo parâmetro que tem inúmeras possibilidades para extraír e mostrar partes de uma data no écran. Consulte a ajuda desta função para saber mais sobre estas variantes. A função `seq()`, que já estudamos anteriormente, quando aplicada a objectos do tipo "Date" permite gerar sequências de datas. A função possui valores próprios no parâmetro "by" que podem ser usados para produzir a sequência desejada[8]. A função `weekdays()` permite-nos obter o dia da semana correspondente à data em causa, enquanto que a função `months()` faz o mesmo para o nome do mês.

[8] Consulte a ajuda da função `seq.Date` para obter mais alternativas.

```
> as.Date("2006-9-23") - as.Date("2003-04-30")

Time difference of 1242 days

> ISOdate(2001, 1, 1) - ISOdate(2000, 6, 14)

Time difference of 201 days

> cut(ISOdate(2001, 1, 1) + 70 * 86400 * runif(10), "weeks")

 [1] 2001-02-12 2001-01-15 2001-01-29 2001-01-29 2001-01-08 2001-02-26
 [7] 2001-01-29 2001-01-01 2001-01-08 2001-01-01
9 Levels: 2001-01-01 2001-01-08 2001-01-15 2001-01-22 ... 2001-02-26

> table(cut(seq(ISOdate(2006, 1, 1), to = ISOdate(2006, 12, 31),
+     by = "day"), "month"))

2006-01-01 2006-02-01 2006-03-01 2006-04-01 2006-05-01 2006-06-01 2006-07-01
        31         28         31         30         31         30         31
2006-08-01 2006-09-01 2006-10-01 2006-11-01 2006-12-01
        31         30         31         30         31
```

Nestes novos exemplos vemos a função `as.Date()`, que permite converter uma string para um objecto do tipo "Date". Vemos também a função `ISOdate()` que nos possibilita uma forma alternativa de criar um objecto do tipo "Date" indicando o ano, mês e dia pelos seus números. Vemos ainda a possibilidade que o R nos dá de usar alguns operadores aritméticos com objectos do tipo "Date", obtendo resultados em termos da diferença temporal entre as datas. Por fim, vemos dois exemplos de utilização da função `cut()` que, quando aplicada a objectos do tipo "Date", gera um factor em que cada valor resulta de uma discretização das datas fornecidas no primeiro argumento do `cut()`, para um conjunto de intervalos definidos pelo segundo argumento. No primeiro exemplo os intervalos são definidos dividindo as datas por semanas, enquanto que no segundo é por mês. No segundo exemplo, vemos uma utilidade deste tipo de discretização, aplicando a função `table()` ao factor resultante, obtendo deste modo uma contagem do número de datas em cada intervalo (cada mês neste exemplo). Na realidade, neste segundo exemplo, se repararmos atentamente nas datas que são fornecidas ao `cut()`, o que vamos obter é o número de dias em cada mês do ano de 2006.

Objectos do tipo "POSIXt"

2. Objectos do tipo POSIXt

   Este tipo de objectos permite guardar tempos que contêm informação não só sobre a data, como os anteriores, mas também sobre as horas. Com este tipo de objectos podemos guardar tempos até ao segundo. Na realidade existem dois tipos de "sub-objectos":

   (a) POSIXct que representam as datas como o número de segundos que passaram desde 1970.

   (b) POSIXlt que representam as datas como uma lista com várias componentes, como: "sec"; "min"; "hour"; "mday"; "mon"; "year"; etc.

   Vejamos exemplos destes tipo de objectos:

```
> (z <- Sys.time())

[1] "2006-10-21 00:59:09 Hora de Verão de GMT"

> as.POSIXlt(Sys.time(), "GMT")

[1] "2006-10-20 23:59:09 GMT"
```

```
> as.POSIXct("2006-12-23 12:45") - as.POSIXct("2006-12-21 21:54")

Time difference of 1.61875 days

> format(Sys.time(), "%a %b %d %X %Y %Z")

[1] "sáb Out 21 0:59:09 2006 Hora de Verão de GMT"
```

A função `Sys.time()` obtém a data/hora actual no computador num objecto do tipo POSIXt. A função `as.POSIXlt()` pode ser usada para converter diferentes objectos para a classe POSIXlt, e neste caso aproveitamos também para mostrar como, durante essa conversão, resolvemos mudar de fuso horário, do nosso para GMT. Vemos também um exemplo de como fazer aritmética com este tipo de objectos, além de um outro exemplo do uso da função `format()` que já vimos acima.

```
> x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
> strptime(x, "%d%b%Y")

[1] "1960-01-01" "1960-01-02" "1960-03-31" "1960-07-30"

> dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
> times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
> x <- paste(dates, times)
> strptime(x, "%m/%d/%y %H:%M:%S")

[1] "1992-02-27 23:03:20" "1992-02-27 22:29:56" "1992-01-14 01:03:30"
[4] "1992-02-28 18:21:03" "1992-02-01 16:56:26"
```

Nestes novos exemplos vemos ilustrações de uma operação bastante frequente, em particular se vamos importar dados que involvem datas, de outras aplicações. Normalmente recebemos essas datas como strings, e portanto temos depois que extraír dessas strings as nossas datas/horas. O problema é que não existe uma única norma seguida por todos para representar datas. Aqui entra a função `strptime()` que pode ser usada para extraír datas/horas de strings indicando quar o formato em que essas datas/horas estão representadas nas strings. Isso é feito através do segundo argumento da função que usa um conjunto de códigos com vário significado, muito ao estilo do que vimos na função `format()`. Para mais detalhes sobre estes códigos o melhor é consultar a ajuda da função.

Após esta incursão pelas datas e horas em R voltemos aos nossos objectos "zoo". Começemos por ver exemplos de visualização deste tipo de objectos. Os gráficos da Figura 6 mostram-nos uma série univariada e também uma multivariada, guardadas em objectos "zoo".

Séries multivariadas irregulares

```
> z <- zoo(rnorm(100), sort(ISOdate(2001, 1, 1) + 70 * 86400 *
+     runif(100)))
> z.mtx <- zoo(matrix(rnorm(100), 50, 2), sort(ISOdate(2001, 1,
+     1) + 70 * 86400 * runif(50)))
> par(mfrow = c(1, 2))
> plot(z)
> plot(z.mtx, plot.type = "single", col = 1:2)
> legend("topright", c("1ªcol", "2ªcol"), lty = 1, col = 1:2)
> par(mfrow = c(1, 1))
```

Vejamos agora um conjunto de funções úteis para diversos tipos de manipulação de objectos "zoo".

Início e fim de séries "zoo"

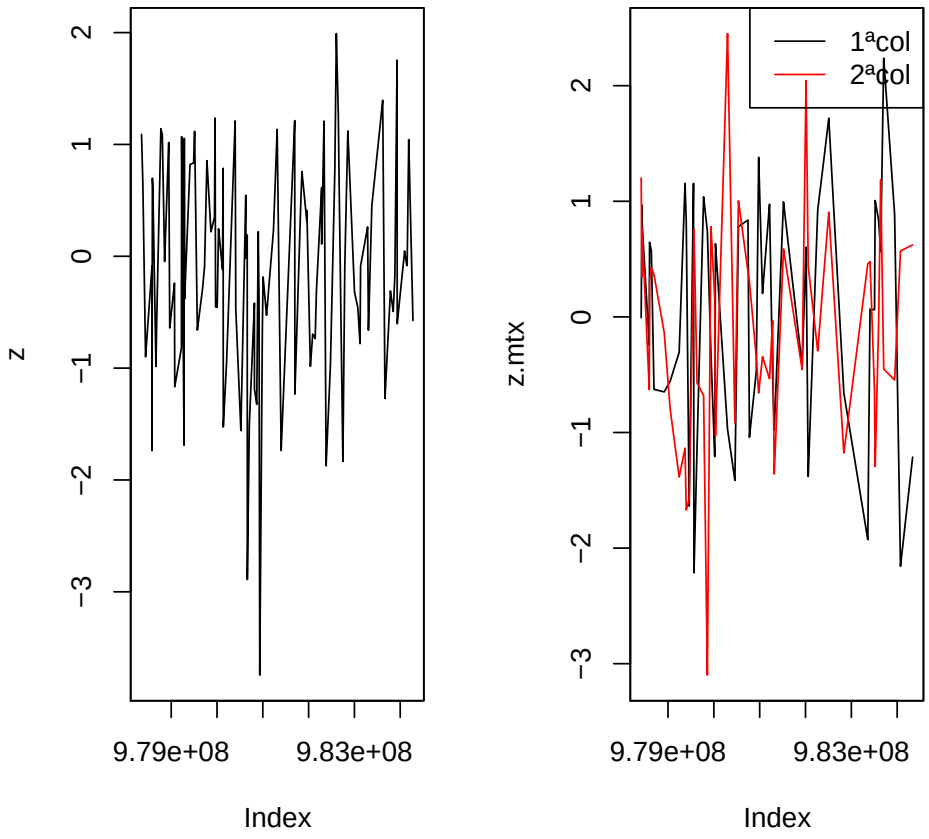


Figura 6: Gráficos de objectos “zoo”.

```
> (x <- zoo(rnorm(15), seq(as.Date("2006-09-01"), length = 15,
+     by = "days")))

  2006-09-01   2006-09-02   2006-09-03   2006-09-04   2006-09-05   2006-09-06
 0.367653423  1.446087183  1.342642534  2.005092105 -0.310355469 -0.001024538
  2006-09-07   2006-09-08   2006-09-09   2006-09-10   2006-09-11   2006-09-12
 0.209334800  1.250436925  0.161235127 -0.270383756  2.978076138 -1.512003802
  2006-09-13   2006-09-14   2006-09-15
-0.172072092  0.237434668 -1.095001597

> start(x)

[1] "2006-09-01"

> end(x)

[1] "2006-09-15"

> x[3:6]

  2006-09-03   2006-09-04   2006-09-05   2006-09-06
 1.342642534  2.005092105 -0.310355469 -0.001024538

> x[coredata(x) > 0]

2006-09-01 2006-09-02 2006-09-03 2006-09-04 2006-09-07 2006-09-08 2006-09-09
 0.3676534  1.4460872  1.3426425  2.0050921  0.2093348  1.2504369  0.1612351
2006-09-11 2006-09-14
 2.9780761  0.2374347
```

As funções `start()` e `end()` funcionam da mesma forma do que com objectos "ts". COmo vemos a indexação funciona da mesma forma como se de um vector "normal" se trata-se, mantendo-se as respectivas etiquetas temporais. De qualquer modo convém não esquecer que se trata, não de um vector normal, mas de um objecto mais complexo que contém para além dos valores da série, as etiquetas. É importante lembrar-mo-nos disso quando pretendemos escolher um subconjunto de valores de uma série impondo uma condição aos seus valores, como é ilustrado no último exemplo. Nesses casos temos que nos lembrar que a condição a ser imposta é sobre os valores (e daí o uso da função `coredata()`) e não sobre o objecto como um todo.

```
> x[as.Date("2006-09-14")]

2006-09-14
 0.2374347

> window(x, start = as.Date("2006-09-10"), end = as.Date("2006-09-15"))

2006-09-10 2006-09-11 2006-09-12 2006-09-13 2006-09-14 2006-09-15
-0.2703838  2.9780761 -1.5120038 -0.1720721  0.2374347 -1.0950016

> (y <- zoo(rnorm(10), seq(as.Date("2006-09-10"), length = 10,
+     by = "days")))

 2006-09-10  2006-09-11  2006-09-12  2006-09-13  2006-09-14  2006-09-15
 0.67339402  1.75291084 -0.13133473 -0.01916885  0.50119663 -1.07051968
 2006-09-16  2006-09-17  2006-09-18  2006-09-19
 1.27971776  0.79160867  0.64387366 -0.64057581

> x - y
```

```
 2006-09-10  2006-09-11  2006-09-12  2006-09-13  2006-09-14  2006-09-15
-0.94377777  1.22516530 -1.38066907 -0.15290324 -0.26376196 -0.02448191
```

Janelas temporais de objectos "zoo

No primeiro exemplo vemos como podemos indexar um objecto "zoo" por valores das etiquetas temporais. No segundo vemos a função `window()` aplicada a um objecto "zoo". Em seguida vemos outros exemplos envolvendo sequências de datas e operações aritméticas com objectos com etiquetas temporais não totalmente coincidentes.

Nos exemplos seguintes vemos como juntar séries "zoo".

```
> x <- zoo(rnorm(7), seq(as.Date("2006-09-01"), length = 7, by = "days"))
> y <- zoo(rnorm(7), seq(as.Date("2006-09-06"), length = 7, by = "days"))
> cbind(x, y)

                   x          y
2006-09-01  0.072992109         NA
2006-09-02 -0.302664430         NA
2006-09-03  0.597683194         NA
2006-09-04  0.572150915         NA
2006-09-05 -0.002487241         NA
2006-09-06 -0.898433693 -0.9438510
2006-09-07  0.753787841  1.3302728
2006-09-08           NA  1.1581772
2006-09-09           NA -0.4568366
2006-09-10           NA -0.7295473
2006-09-11           NA -0.6993896
2006-09-12           NA -0.6021335

> merge(x, y, all = FALSE)

                  x         y
2006-09-06 -0.8984337 -0.943851
2006-09-07  0.7537878  1.330273
```

Juntar objectos "zoo"

A função `cbind()` pode ser usada para juntar duas séries univariadas numa série multivariada, encarregando-se o R de tratar de valores com etiquetas não concordantes (os valores da outra série ficam com NA), enquanto que a junção usando a função `merge()` permite, em alternativa e através do parâmetro "all", indicar que só queremos na junção os valores concordantes.

```
> xx <- x[1:6]
> lag(xx)

  2006-09-01   2006-09-02   2006-09-03   2006-09-04   2006-09-05
-0.302664430  0.597683194  0.572150915 -0.002487241 -0.898433693

> merge(xx, lag(xx))

                   xx      lag(xx)
2006-09-01  0.072992109 -0.302664430
2006-09-02 -0.302664430  0.597683194
2006-09-03  0.597683194  0.572150915
2006-09-04  0.572150915 -0.002487241
2006-09-05 -0.002487241 -0.898433693
2006-09-06 -0.898433693           NA

> diff(xx)

 2006-09-02  2006-09-03  2006-09-04  2006-09-05  2006-09-06
-0.37565654  0.90034762 -0.02553228 -0.57463816 -0.89594645
```

nções, diferenças
ags de objectos
oo"

No código anterior vemos a utilização da função `lag()` sobre objectos "zoo", bem como mais um exemplo de junção de séries através da função `merge()`. Já a função `diff()` permite-nos obter diferenças entre valores sucessivos da série.

Vejamos agora alguns exemplos de formas de lidar com valores desconhecidos em séries temporais guardadas em objectos "zoo".

Valores desconhecidos em objectos "zoo"

```
> x

  2006-09-01   2006-09-02   2006-09-03   2006-09-04   2006-09-05   2006-09-06
 0.072992109 -0.302664430  0.597683194  0.572150915 -0.002487241 -0.898433693
  2006-09-07
 0.753787841

> x[sample(1:length(x), 3)] <- NA
> x

2006-09-01 2006-09-02 2006-09-03 2006-09-04 2006-09-05 2006-09-06 2006-09-07
        NA -0.3026644         NA  0.5721509         NA -0.8984337  0.7537878

> na.omit(x)

2006-09-02 2006-09-04 2006-09-06 2006-09-07
-0.3026644  0.5721509 -0.8984337  0.7537878

> na.contiguous(x)

2006-09-06 2006-09-07
-0.8984337  0.7537878

> na.approx(x)

2006-09-02 2006-09-03 2006-09-04 2006-09-05 2006-09-06 2006-09-07
-0.3026644  0.1347432  0.5721509 -0.1631414 -0.8984337  0.7537878

> na.locf(x)

2006-09-02 2006-09-03 2006-09-04 2006-09-05 2006-09-06 2006-09-07
-0.3026644 -0.3026644  0.5721509  0.5721509 -0.8984337  0.7537878
```

A função `na.omit()` permite eliminar os pontos em que o valor é NA. A função `na.contiguous()` extrai a mais longa sequência sem valores NA. A função `na.approx()` usa interpolação linear para preencher os valores NA, enquanto que a função `na.locf()` preenche cada valor NA com o valor não desconhecido mais recente (i.e. anterior ao NA).

Por vezes temos interesse em aplicar uma qualquer função a uma janela de tamanho x da nossa série. Ainda mais interessante é fazer essa janela deslizar pela série toda. Para isso podemos usar as seguintes funções:

Funções deslizantes

```
> (x <- zoo(rnorm(7), seq(as.Date("2006-09-01"), length = 7, by = "days")))

  2006-09-01   2006-09-02   2006-09-03   2006-09-04   2006-09-05   2006-09-06
-0.961754723  0.008705073 -0.991048555  1.438521921  0.650619612  0.312436234
  2006-09-07
-1.376853779

> rapply(x, 4, mean)

2006-09-02 2006-09-03 2006-09-04 2006-09-05
-0.1263941  0.2766995  0.3526323  0.2561810

> rapply(x, 4, mean, align = "right")
```

```
2006-09-04 2006-09-05 2006-09-06 2006-09-07
-0.1263941  0.2766995  0.3526323  0.2561810

> rollmean(x, 3)

2006-09-02 2006-09-03 2006-09-04 2006-09-05 2006-09-06
-0.6480327  0.1520595  0.3660310  0.8005259 -0.1379326

> rollmean(x, 3, na.pad = T)

2006-09-01 2006-09-02 2006-09-03 2006-09-04 2006-09-05 2006-09-06 2006-09-07
        NA -0.6480327  0.1520595  0.3660310  0.8005259 -0.1379326         NA

> rollmean(x, 3, na.pad = T, align = "right")

2006-09-01 2006-09-02 2006-09-03 2006-09-04 2006-09-05 2006-09-06 2006-09-07
        NA         NA -0.6480327  0.1520595  0.3660310  0.8005259 -0.1379326

> rollmean(x, 3, align = "right")

2006-09-03 2006-09-04 2006-09-05 2006-09-06 2006-09-07
-0.6480327  0.1520595  0.3660310  0.8005259 -0.1379326
```

A função `rapply()` permite-nos aplicar uma função qualquer a uma janela deslizante de um certo tamanho (2º argumento), de uma série "zoo". A aplicação da função pode ser feita centrando a janela em cada ponto (valor por defeito), ou alinhando a janela à direita ou à esquerda, usando para isso o parâmetro "align". A função `rollmean()` não é mais do que um "envelope" à chamada da função `rapply()` com a função "mean". Ambas as funções (rapply e rollmean) têm um parâmetro "na.pad" que permite indicar se pretendemos que as etiquetas temporais para as quais não é possível calcular o valor da função, por inexistância de valores suficientes para preencher a janela deslizante, deverão ou não ser preenchidos por NA's.

3 Programação em R

3.1 Interacção com o Utilizador

O R possui diversas funções que têm como objectivo ou escrever conteúdos no écran, ou ler coisas introduzidas pelo utilizador no teclado, isto é para gerir a interacção com o utilizador.

Escrever no écran

A função `print()`, por exemplo, pode ser usada para escrever o conteúdo de qualquer objecto[9].

Por vezes, quando os objectos são muito grandes (p. ex. grandes listas) é mais conveniente usar a função `str()`,

```
> str(list(nros = rnorm(100), dados = matrix(rnorm(10000), 100,
+     100)))

List of 2
 $ nros : num [1:100] -0.543  0.654  0.207  0.454 -1.125 ...
 $ dados: num [1:100, 1:100] -2.210  0.511  1.681  0.358  0.656 ...
```

A função `cat()` também pode ser usada para escrever objectos ou constantes. Funciona com um número qualquer de argumentos, e o que faz é transformar os seus argumentos em "strings", concatená-los, e só depois os escreve no écran,

```
> x <- 34
> cat("x tem o valor de ", x, "\t o que é estranho!\n")

x tem o valor de  34          o que é estranho!
```

Nesta instrução, certos caracteres contidos em "strings" têm significado especial. Por exemplo "`\t`" significa o caracter `Tab`, enquanto "`\n`" significa o caracter `Newline` (isto é, muda de linha).

Leitura de dados

Relativamente a leitura de dados introduzidos pelo utilizador também existem várias hipóteses em R. A função `scan()`, por exemplo, permite ler dados de vário tipo,

```
> x <- scan(n=5)
1: 45 21.4 56 66.54 45.8787
Read 5 items
> x
[1] 45.0000 21.4000 56.0000 66.5400 45.8787
```

Ainda um outro exemplo,

```
> x <- scan()
1: 45 66 34.2
4: 456.7 7
6: 12.2
7:
Read 6 items
> x
[1]  45.0  66.0  34.2 456.7   7.0  12.2
```

E finalmente um outro exemplo com dados de outro tipo,

```
> scan(what=character())
1: 'erer' 'fdf'
3: '233' 44.5
5:
Read 4 items
[1] "erer" "fdf"  "233"  "44.5"
```

A função tem ainda muitas outras possíveis formas de ser usada que podem ser consultadas na respectiva ajuda.

[9] Note que este é mais um exemplo de uma função genérica (c.f. Secção 3.4), existindo diversos métodos para diferentes classes de objectos.

3.2 Estruturas de Controlo da Linguagem R

A linguagem R, como a qualquer linguagem de programação, possui várias instruções destinadas a alterar o curso sequencial normal de execução dos programas.

3.2.1 Instruções Condicionais

As instruções condicionais permitem ao programador explicitar diferentes alternativas a serem executadas dependendo de alguma condição a ser testada na altura da execução das instruções.

A instrução if

A instrução `if`, por exemplo, permite explicitar uma condição booleana e dois conjuntos de instruções alternativos que são executados dependendo do valor da condição. Vejamos um pequeno exemplo,

```
> if (x > 0) y <- z / x else y <- z
```

Esta instrução pode ser facilmente explicada dizendo que no caso de a variável **x** ser superior a 0, a instrução que é executada é a atribuição da expressão **z/x** a **y**. Caso contrário, o R efectua a outra atribuição que vem à frente da palavra `else`. Em termos genéricos, a semântica associada à instrução `if` pode ser descrita da seguinte forma: se a condição booleana incluída entre parenteses à frente da palavra `if` fôr verdadeira, o R executa o conjunto de instruções que aparecem entre esta condição e a palavra `else`; se a condição fôr falsa, é executado o conjunto de instruções que aparece à frente da palavra `else`.

Blocos de instruções

No sentido de passarmos do exemplo apresentado acima, para um exemplo mais genérico envolvendo um conjunto de instruções, temos que introduzir a noção de conjunto ou bloco de instruções na linguagem R. Em R um bloco de instruções é normalmente indicado fornecendo essas instruções em linhas separadas e seguidas, delimitadas por chavetas. Por exemplo,

```
> if (x > 0) {
    cat('x é positivo.\n')
    y <- z / x
} else  {
    cat('x não é positivo!\n')
    y <- z
}
```

Note-se que neste pequeno exemplo usamos a identação das instruções em cada um dos 2 blocos de instruções "contidos" na instrução `if`. Embora não obrigatória, esta forma de proceder é sempre aconselhável, por facilitar a "leitura" da instrução.

A instrução `if` tem ainda outras variantes que passamos a descrever. Por exemplo, a claúsula `else` é opcional, podendo nós usar `if's` sem alternativa para o caso de a condição ser falsa. Nesse caso, o R não fará nada como resultado da instrução `if`.

IF's aninhados

Podemos ainda usar várias instruções `if` aninhadas, como no exemplo seguinte:

```
> if (idade < 18) {
    grupo <- 1
} else if (idade < 35) {
    grupo <- 2
} else if (idade < 65) {
    grupo <- 3
} else {
    grupo <- 4
}
```

Note-se que embora só exista uma instrução em cada bloco deste exemplo, colocamos todas as instruções entre chavetas como se tratassem de conjuntos de instruções. Neste exemplo concreto, não colocar as chavetas iria originar um erro de sintaxe. A explicação

de tal comportamento, aparentemente estranho, tem a ver com o facto de o R ser uma linguagem interpretada e sai um pouco do âmbito desta disciplina. De qualquer modo, uma boa regra para evitar problemas, é usar sempre as chavetas para delimitar as duas partes de uma instrução `if`. Mais detalhes e explicações sobre este comportamento podem ser obtidas na secção 3.2.1 do manual "R Language Definition" que vem com o R e está disponível no sistema de ajuda do R.

A função "ifelse()"

Uma função relacionada com a instrução `if` é a função `ifelse()`. Esta função permite trabalhar com vectores de condições boleanas. O seu significado poderá ser melhor apreendido vom um pequeno exemplo,

```
> x <- rnorm(5, sd = 10)
> x

[1] -5.9357417 -4.5964847  0.5452268 -2.9091063 -9.2760717

> sig <- ifelse(x < 0, "-", "+")
> sig

[1] "-" "-" "+" "-" "-"
```

A função "switch()"

A instrução `switch()` pode também ser usada para escolher uma de várias alternativas. Ela consiste numa série de argumentos em que dependendo do valor do primeiro argumento, o resultado do `switch` é um dos outros argumentos. Vejamos em primeiro lugar um exemplo em que o primeiro argumento é um número. Neste caso, o valor desse número determina qual o argumento do `switch` que é avaliado. Por exemplo,

```
> op <- 2
> vs <- rnorm(10)
> switch(op, mean(vs), median(vs))

[1] -0.1936888
```

O valor obtido resulta de aplicar a função `median()` ao vector **vs**, uma vez que o valor de **op** é 2, e logo é escolhido o segundo argumento a seguir ao teste contido no primeiro argumento do `switch`. Na realidade, todos os argumentos a seguir ao primeiro são tomados como uma lista (c.f. Secção 2.9) que pode ter nomes associados a cada componente como vamos ver no exemplo a seguir. No caso de não ter nomes, como no exemplo acima, é escolhida a componente da lista pelo número indicado no primeiro argumento do `switch`. No caso de o número ser superior ao tamanho da lista, o resultado do `switch` é `NULL`.

Ao dar nomes às componentes da lista, passamos a poder usar valores textuais no primeiro argumento, como podemos ver no exemplo seguinte,

```
> semaforo <- "verde"
> switch(semaforo, verde = "continua", amarelo = "acelera", vermelho = "pára")

[1] "continua"
```

Por estes exemplos podemos ver que a instrução `switch` tem a forma genérica de

`switch(valor,lista)`

e que o resultado da instrução não é mais do que usar o conteúdo do primeiro argumento para aceder a uma das componentes da lista cujo conteúdo das componentes é indicado nos argumentos seguintes da instrução. Se o primeiro argumento fôr um número então o resultado é a componente com esse número de ordem, se o primeiro argumento fôr um nome, então é extraída a componente com esse nome.

3.2.2 Instruções Iterativas

O R tem várias instruções iterativas ("ciclos") que nos permitem repetir blocos de instruções. Para além destas instruções o R possui ainda duas instruções que podem ser usadas para controlar a execução dos ciclos.

Instruções iterativ

A instrução `while` tem a seguinte estrutura genérica,

O ciclo while

```
while (<condição boolena>)
   <bloco de instruções>
```

A sua semântica pode ser descrita por: enquanto a condição booleana fôr verdadeira, repetir o bloco de instruções no "corpo" do ciclo. Vejamos um pequeno exemplo,

```
> x <- rnorm(1)
> while (x < -0.3) {
+     cat("x=", x, "\t")
+     x <- rnorm(1)
+ }
> x

[1] 1.516122
```

Note-se que as instruções no bloco dentro do ciclo podem nunca ser executadas, bastando para isso que a condição seja falsa da primeira vez que o R "chega" ao ciclo. No exemplo acima, se o primeiro número "sorteado" pela função `rnorm()` fôr superior ou igual a -0.3, as instruções do ciclo não são executadas nenhuma vez. Em sintese podemos dizer que as instruções no bloco do ciclo `while` podem ser executadas zero ou mais vezes.

Atente-se que é um erro relativamente frequente escrever blocos de instruções que levam a que os ciclos nunca mais terminem. Por exemplo, se no exemplo anterior nos tivessemos enganado e, dentro do ciclo, tivessemos escrito "`y <- rnorm(1)`", em vez da atribuição que aí se encontra, caso o R "entrasse" no ciclo, nunca mais o terminaria[10]. Como regra geral podemos dizer que se deve escrever os ciclos de tal forma que exista sempre possibilidade de que a(s) variável(eis) que controla(m) a execução dos mesmos (no nosso exemplo a variável **x**), possa sempre ver o seu valor alterado nas instruções que formam o bloco contido no ciclo, de forma a que a condição possa vir a ser falsa, ou seja de forma a que o ciclo possa vir a terminar.

O ciclo repeat

A instrução `repeat` permite mandar o R executar um bloco de instruções uma ou mais vezes. A sua forma genérica é,

```
repeat
   <bloco de instruções>
```

Repare que uma vez que não existe qualquer condição lógica a governar a execução repetida do bloco de instruções, como no caso do ciclo `while`, o R socorre-se de outras instruções que permitem parar a execução de um processo iterativo. Em concreto a instrução `break` se executada dentro do ciclo faz o R terminar a repetição da execução do bloco de instruções. Mais uma vez, para evitar ciclos "infinitos" convirá garantir que tal instruções é passível de ser executada no bloco de instruções que forma o corpo do ciclo. Vejamos um pequeno exemplo que lê frases introduzidas pelo utilizador até este introduzir uma frase vazia,

Saír de um bloco/ciclo

```
texto <- c()
repeat {
  cat('Introduza uma frase ? (frase vazia termina) ')
  fr <- readLines(n=1)
  if (fr == '') break else texto <- c(texto,fr)
}
```

[10] Se por acaso isso lhe acontecer, pode pressionar a tecla Esc para fazer o R abortar a computação do ciclo.

Conforme deverá ser óbvio, este ciclo só irá terminar quando a variável **fr**, que contém a frase que o utilizador acabou de introduzir, fôr vazia. Se tal não acontecer, a instrução `break` não é executada e como tal o ciclo é repetido, ou seja nova frase é pedida.

O próximo exemplo ilustra o uso de outra instrução que pode ser usada para controlar o que é feito num ciclo, a instrução `next`,

Avançar para a próxima iteração

```
repeat {
  cat('Introduza um nro positivo ? (zero termina) ')
  nro <- scan(n=1)
  if (nro < 0) next
  if (nro == 0) break
  pos <- c(pos,nro)
}
```

Neste exemplo pretende-se ler um conjunto de números positivos que são coleccionados num vector. O utilizador deverá introduzir o número zero para terminar o ciclo. No caso de o utilizador introduzir um número negativo, pretende-se pedir novo número sem que o número negativo seja acrescentado ao vector com os números lidos. Conseguimos esse efeito com a instrução `next`. Quando o R encontra esta instrução, salta imediatamente para o início do ciclo que esteja a executar, não executando portanto qualquer instrução que viesse a seguir ao `next` dentro do ciclo.

O ciclo for

Finalmente o R tem ainda a instrução `for` que permite controlar o número de vezes que um ciclo é executado através de uma variável de controlo que vai tomar uma série de valores pré-definidos em cada iteração do ciclo. A sua sintaxe genérica é,

```
for(<var> in <conjunto>)
   <bloco de instruções>
```

Vejamos um pequeno exemplo de utilização destes ciclos,

```
> x <- rnorm(10)
> k <- 0
> for (v in x) {
+     if (v > 0)
+         y <- v
+     else y <- 0
+     k <- k + y
+ }
```

Este pequeno ciclo começa por obter 10 número aleatórios de uma distribuição normal, e depois obtém a soma daqueles que são positivos. Para isso é usado um ciclo `for` em que a variável de controlo, neste caso chamada **v**, vai tomar, em cada iteração, um dos valores do conjunto **x**, isto é os 10 números sorteados. Isto quer dizer que na primeira iteração o valor de **v** é igual a **x**[**1**], na segunda é igual a **x**[**2**] e assim sucessivamente para todos os elementos de **x**. Note que este é um simples exemplo ilustrativo do uso do `for`, uma vez que em termos de estilo de programação "à R" a solução para este problema poderia ser obtida de forma muito mais simples, sem recurso a qualquer ciclo,

```
> k <- sum(x[x > 0])
```

Eficiência dos ciclos

Note-se ainda que esta segunda forma seria muito mais eficiente em termos de tempo de execução, embora para um vector com este tamanho a diferença seja irrelevante. O mesmo já não se poderá dizer para vectores maiores como se pode constatar nesta pequena experiência,

```
> x <- rnorm(1e+05)
> t <- Sys.time()
> k <- 0
```

```
> for (v in x) {
+     if (v > 0)
+         y <- v
+     else y <- 0
+     k <- k + y
+ }
> Sys.time() - t

Time difference of 3.188 secs

> t <- Sys.time()
> k <- sum(x[x > 0])
> Sys.time() - t

Time difference of 0.07799983 secs
```

Tempos de execução

A função `Sys.time()` permite-nos aceder ao relógio do computador, conseguindo dessa forma temporizar a execução de uma parte do nosso código como vemos no exemplo. O que também podemos ver é a diferença muito significativa dos tempos de execução envolvendo o ciclo `for` e sem esse ciclo. Ou seja para obter o mesmo resultado demoramos cerca de 30 vezes mais! De facto, uma lição a aprender deste pequeno exemplo é que os ciclos são estruturas computacionalmente pesadas na linguagem R [11] e que muitas vezes os podemos evitar usando a vectorização das operações que o R disponibiliza.

3.2.3 Evitando ciclos

O R possui, como a maioria das linguagens, objectos que permitem armazenar conjuntos de valores (p. ex. vectores, matrizes, etc.). Existem inúmeros problemas onde para os resolver, precisamos de percorrer todos os elementos desses objectos. Na maioria das linguagens isso faz-se com o recurso a ciclos. No R, temos felizmentes várias alternativas bastante mais práticas e sintéticas, para além de muito mais eficientes como vimos no último exemplo da secção anterior.

Métodos para evitar ciclos

Uma das formas mais comuns de evitar os ciclos consiste em tirar partido do facto de muitas funções do R serem vectorizáveis. Por exemplo, se pretendemos sumar todos os elementos de uma matriz **m** em vez de andarmos a percorrê-los, usando para isso dois ciclos *for* aninhados um dentro do outro (para permitir percorrer todas as colunas, para cada linha), como é comum em muitas lingaugens de programação, em R podemos simplesmente fazer `s <- sum(m)`. Compare a simplicidade disto com a versão dos ciclos,

```
> m <- matrix(rnorm(10), 5, 2)
> s <- 0
> for (c in 1:ncol(m)) for (l in 1:nrow(m)) s <- s + m[l, c]
```

Para além da maior simplicidade nunca é demais realçar que a versão com ciclos demorar consideravelmente mais tempo a ser executada, o que pode ser critico em matrizes muito grandes.

O R tem ainda outras funções que podem ser explicitamente usadas para vectorizar operações. Suponhamos que pretendemos saber o valor mínimo de cada coluna de uma matriz. No fundo o que pretendemos é aplicar a função `min()` a cada uma das colunas da matriz. Em R podemos implementar essa ideia através da função `apply()`, como vemos em seguida,

A função apply

```
> m <- matrix(rnorm(100), 10, 10)
> apply(m, 2, min)

 [1] -0.2740504 -1.4357644 -2.3413347 -1.7631999 -1.8275833 -1.5714361
 [7] -1.3689081 -0.9579934 -2.9972401 -0.4734547
```

[11] Note-se no entanto que isto tem vindo a melhorar de forma significativa nas versões mais recentes.

A função `apply()` permite-nos aplicar uma qualquer função a uma das dimensões de uma matriz ou *array*. Neste caso estamos a aplicar a função `min()` à 2ª dimensão (as colunas) da matriz **m**. O resultado é um vector de números, os mínimos de cada coluna. Se pretendessemos o mesmo resultado sobre as linhas bastaria colocar um 1 no segundo argumento da função, indicando desta forma que a função deveria ser aplicada à 1ª dimensão do objecto.[12]

A função tapply

A funçãoo `tapply()` permite executar operações semelhantes mas em sub-grupos dos dados, determinados por valores de factores. Vejamos um exemplo usando um conjunto de dados que vem com o R para efeitos ilustrativos,

```
> data(warpbreaks)
> head(warpbreaks)
```

```
  breaks wool tension
1     26    A       L
2     30    A       L
3     54    A       L
4     25    A       L
5     70    A       L
6     52    A       L
```

```
> tapply(warpbreaks$breaks, warpbreaks[, -1], sum)
```

```
    tension
wool   L   M   H
   A 401 216 221
   B 254 259 169
```

Como vemos o conjunto de dados **warpbreaks** possui 2 colunas que são factores (**wool** e **tension**). A chamada à função `tapply()` calcula a soma da coluna **breaks** deste conjunto de dados, mas para todos os sub-grupos formados pelas combinações dos valores dos dois factores. Assim por exemplo, 401 é a soma do valor de **breaks** para as linhas do conjunto de dados onde **wool** tem o valor *A* e **tension** o valor *L*.

A função sapply

Uma outra função interessante é a função `sapply()`, que permite aplicar uma função a todos os elementos de um vector ou lista. Vejamos dois exemplos ilustrativos,

```
> x <- sample(10)
> sapply(x, function(y) (y - mean(x))/sd(x))
```

```
 [1]  1.1560120 -0.8257228  0.8257228  0.4954337 -0.1651446  1.4863011
 [7] -1.1560120  0.1651446 -0.4954337 -1.4863011
```

```
> l <- list(f1 = sample(20), f2 = c(2.3, 1.3, 4.5, 2.4), f3 = rnorm(100))
> sapply(l, quantile)
```

```
        f1    f2         f3
0%    1.00 1.300 -3.2453520
25%   5.75 2.050 -0.6011432
50%  10.50 2.350  0.1545809
75%  15.25 2.925  0.6604947
100% 20.00 4.500  2.3431545
```

No primeiro exemplo, aplicamos uma função a um vector. Note, que a função não necessita de existir no R e pode mesmo ser definida na própria chamada ao `sapply()`, como é o caso deste exemplo. Esta função pode ser vista como temporária, uma vez que só existe durante a execução do `sapply()`. No segundo exemplo, vemos a função `quartile()` a ser aplicada a uma lista. Disto resultada a aplicação a cada elemento da lista, originado neste caso uma matriz de resultados, com nomes apropriados aos mesmos.

Para finalizar diga-se que a função `sapply()` é uma versão amigável da função `lapply()`, cuja ajuda pode ser consultada para mais informações.

[12] Na realidade o segundo argumento da função pode ser um vector de dimensões, mas esse tipo de utilização sai fora do âmbito deste texto.

3.3 Funções

Na linguagem R as funções são também objectos e podem ser manipuladas de forma semelhante aos outros objectos que estudamos até agora. Uma função tem três características principais: uma lista de argumentos, o corpo da função e o ambiente onde ela é definida. A lista de argumentos é uma lista de símbolos (os argumentos) separada por vírgulas, que podem ter valores por defeito. Existe ainda a possibilidade de um argumento ser o conjunto de caracteres "...", cujo significado iremos descrever mais à frente. O corpo de uma função é formado por um conjunto de instruções da linguagem R, sendo que normalmente é um bloco de instruções. Quanto ao ambiente da função, trata-se do ambiente activo quando ela foi criada e isso determina que tipo de objectos são visíveis pela função conforme vermos mais à frente.

3.3.1 Criar funções

Criar funções

A criação de uma função consiste na atribuição do conteúdo de uma função a um nome, como qualquer outro objecto do R. Esse conteúdo é a lista dos seus argumentos e as instrução que formam o corpo da função. Vejamos um exemplo simples de uma função que recebe como argumento uma temperatura em graus Celsius e a converte para graus Fahrenheit,

```
> cel2far <- function(cel) {
+     res <- 9/5 * cel + 32
+     res
+ }
> cel2far(27.4)

[1] 81.32

> cel2far(c(0, -34.2, 35.6, 43.2))

[1]  32.00 -29.56  96.08 109.76
```

Note como, tirando partido da vectorização das operações aritméticas do R (c.f. Secção 2.3), podemos usar a nossa função com um vector de números como argumento e não só com um único valor de temperatura. Repare ainda que o corpo desta função está desnecessariamente complexo, podendo nós em alternativa definir a função como,

```
> cel2far <- function(cel) 9/5 * cel + 32
```

Em resumo, a criação de uma função é uma atribuição com a forma genérica,

```
<nome da função> <- function(<lista de argumentos>) <corpo da função>
```

Valor retornado

O valor retornado como resultado da função é o resultado da última computação efectuada na função. No exemplo acima essa última computação é o conteúdo do objecto **res** e portanto esse é o resultado da função. Existe uma função específica, a função `return()`, que permite retornar um determinado valor numa função. Isso quer dizer que essa função se executada no corpo de uma função vai fazer com que a execução da mesma termine aí imediatamente. Vejamos um exemplo,

```
> calculo <- function(x) {
+     if (x < 0)
+         return(NULL)
+     y <- 34 * sqrt(x)/5.2
+     y
+ }
```

Se esta função for chamada com um número negativo a condição do `if` é verdadeira e portanto a função `return` é executada. O seu efeito é terminar imediatamente a execução da função `calculo`, e retornar o valor **NULL** como resultado dessa execução. Se a função fôr chamada com um número não negativo, então a execução segue o curso normal sendo nesse caso o conteúdo do objecto **y** que é retornado como resultado da função.

3.3.2 Ambientes e "scope" de variáveis

Quando começamos a nossa interacção com o R, por exemplo criando novos objectos (vectores, matrizes, funções, etc.), estamos a fazê-lo num ambiente chamado o *workspace*, ou o ambiente de topo. Sempre que fazemos `ls()` por exemplo, para saber os objectos que temos na memória, o que obtemos é a lista de objectos neste ambiente de topo. Como vamos ver, existem situações que levam à criação de novos ambientes, levando a uma hierarquia de ambientes com a forma de uma árvore invertida, aninhados uns nos outros, até ao ambiente de topo. A importância destas noções reside nas regras de "scope" ou visibilidade dos objectos em R que dependem do ambiente em que eles são definidos.

Workspace

Scoping

Quando criamos uma função ela, como objecto que é, fica associada ao ambiente onde foi criada. Este é normalmente o ambiente de topo, o tal *workspace* do R. No entanto, quando chamamos essa função o R vai executá-la num novo ambiente "por baixo" do ambiente onde ela é chamada. Quer isto dizer que o ambiente onde são executadas as instruções que formam o corpo de uma função não é o mesmo ambiente onde a função é chamada. Isto tem impacto nos objectos que são visíveis em cada um destes dois ambientes, o ambiente de onde chamamos a função, e o ambiente onde as instruções da função são executadas (que está em baixo do primeiro em termos da hierarquia de ambientes do R). Vejamos um exemplo das implicações e importância desta informação. Consideremos o seguinte código, que inclui a definição de uma função, e que assumimos ser introduzido no ambiente de topo,

```
> x <- 2
> z <- 56.3
> f <- function(x) {
+   a <- 34
+   y <- x / 4 * a * z
+   y
+ }
> f(21)
[1] 10049.55
> a
Error: Object "a" not found
```

Este pequeno exemplo ilustra várias questões importantes ligadas à questão dos ambientes e também do "scope" ou visibilidade das variáveis. Analisemos cada uma delas. O objecto **a**, por exemplo, é criado dentro da função **f**, ou seja a instrução de atribuição que cria o objecto **a**, é executada no ambiente da função **f** que é diferente do ambiente de topo, concretamente está abaixo deste. Essa é a razão pela qual, quando no ambiente de topo, pedimos o conteúdo do objecto **a**, obtemos um erro, indicando que não foi encontrado nenhum objecto com esse nome. O erro tem lógica uma vez que no ambiente de topo não existe de facto, qualquer objecto com o nome **a**. Por outro lado, se antentarmos de novo no código da função, verificaremos que existe uma atribuição que usa o objecto **z**, objecto esse que não existe no ambiente da função **f**. No entanto, ele existe no ambiente acima deste na hierarquia de ambientes. E isto leva-nos à regra fundamental da avaliação de instruções na linguagem R, relativamente à visibilidade do objectos. Esta regra, conhecida por *lexical scoping*, diz-nos que:

A regra de lexical scoping

> Quando um objecto é necessário numa avaliação, ele é procurado pelo R no ambiente em que foi pedido. Caso seja encontrado um objecto com esse nome nesse ambiente, ele é usado. Caso tal não aconteça o R procura-o no ambiente acima, e assim sucessivamente até ao ambiente de topo. Se neste processo for encontrado um objecto com esse nome ele é usado. De contrário o R dá um erro dizendo que tal objecto não existe.

Assim, no ambiente da execução da função **f**, o objecto **z** não existe, no entanto, no ambiente acima, o ambiente de topo, existe um objecto com o nome **z** e por isso o seu valor (56.3), é usado na atribuição dentro da função **f**. Por outro lado, na mesma

instrução de atribuição dentro da função **f**, é referido um objecto com o nome **x**. O valor que é usado é o valor do objecto com esse nome que existe na própria função (que por sinal é o seu único argumento), e não o valor do objecto com o nome **x** que existe no ambiente de topo. Assim, no exemplo de chamada da função mostrado acima, o valor de **x** usado na atribuição é o valor 21 e não o valor 2.

Estas são provavelmente as noções principais ligadas a ambientes e "scoping" da linguagem R. No entanto, existe muito mais por detrás destas importantes noções, como por exemplo funções específicas para lidar com ambientes, ou para aceder a objectos em ambientes específicos. Tais assuntos saem no entanto fora do âmbito deste texto. O leitor interessado poderá consultar por exemplo o livro *S Programming* por Venables and Ripley (2000).

3.3.3 Argumentos de funções

Valores por defeito

Quando criamos uma função indicamos normalmente a sua lista de argumentos. Nesta altura a linguagem R permite-nos também explicitar, caso queiramos, um valor por defeito para cada um desses argumentos. O uso desta facilidade, vai permitir ao utilizador das nossas funções, evitar incluir um valor para esses argumentos caso pretenda usar o valor por defeito. Isto é particularmente útil em funções com muitos argumentos, alguns dos quais só usados em situações muito particulares, sendo que os seus valores por defeito fazem sentido na maioria das utilizações dessas funções. Por exemplo, a função `mean()` serve para calcular a média dos valores que lhe são fornecidos,

```
> mean(c(21, 45.3, 342.4, 54.3, 65.3, 1000.2))

[1] 254.75
```

Valores por defeito

A maioria das vezes iremos usar esta função deste modo. No entanto, se consultarmos a ajuda desta função iremos observar que ela tem outros dois argumentos, *trim* e *na.rm*, cada um deles com um valor por defeito, 0 e *FALSE*, respectivamente. O facto de eles terem um valor por defeito permite-nos fazer chamadas à função como a apresentada em cima, sem que obtenhamos um erro por não termos indicado o valor destes argumentos. Ao não indicarmos estes valores o R vai assumir os valores por defeito indicados pelo criador da função nos cálculos executados dentro da mesma. Se não pretendessemos usar estes valores, teriamos que o explicitar na chamada à função, como no exemplo a seguir,

```
> mean(c(21, 45.3, 342.4, 54.3, 65.3, 1000.2), trim = 0.2)

[1] 126.825
```

Use a ajuda da função para ver se entende o que foi calculado com este valor do parâmetro *trim*.

Em resumo, ao criarmos uma função podemos indicar valores por defeito para alguns dos seus parâmetros, bastanto para isso usar o sinal igual seguido do valor à frente do nome do argumento, conforme ilustrado neste pequeno exemplo,

```
> valor.central <- function(x, estat = "mean") {
+     if (estat == "mean")
+         return(mean(x))
+     else if (estat == "median")
+         return(median(x))
+     else return(NULL)
+ }
> x <- rnorm(10)
> valor.central(x)

[1] 0.4182037

> valor.central(x, estat = "median")
```

```
[1] 0.4513049
```

mero variável de râmetros

Uma outra facilidade bastante conveniente da linguagem R é a possibilidade de criar funções com número variável de parâmetros. Isso é conseguido por um parâmetro especial que é indicado por "...". Este parâmetro especial é de facto uma lista que pode agregar um número qualquer de parâmetros usados na chamada da função. Uma utilização frequente desta facilidade é na criação de funções que chamam outras funções e ao fazê-lo pretendem passar-lhes parâmetros que só a elas lhes interessam. Por exemplo, no exemplo da função **valor.central** mostrada acima, não previmos a possibilidade de o utilizador ao calcular o valor central usando a função `mean()` pretender fazê-lo usando um média truncada dos 20% valores extremos. Tal é possível através do parâmetro *trim* da função `mean()`, mas da forma como criamos a função `valor.central()`, o utilizador não tem forma de usar a função **mean** deste modo. Do mesmo modo a função `median()`, que também pode ser chamada da função que criamos, também tem parâmetros próprios que são inacessíveis ao utilizador da nossa função. Uma forma de dar a volta a este problema reside exactamente na utilização do parâmetro "...", como vemos nesta definição alternativa de `valor.central()`,

```
> valor.central <- function(x, estat = "mean", ...) {
+     if (estat == "mean")
+         return(mean(x, ...))
+     else if (estat == "median")
+         return(median(x, ...))
+     else return(NULL)
+ }
> x <- rnorm(10)
> valor.central(x)

[1] 0.1530552

> valor.central(x, trim = 0.2)

[1] 0.2216488

> valor.central(x, estat = "median", na.rm = T)

[1] 0.1385397
```

Desta forma, tudo o que não sejam parâmetros específicos da função `valor.central()` são agregados no tal parâmetro especial "...", que por sua vez é passado para as chamadas das funções **mean** e **median**, com qualquer que seja o seu conteúdo. Desta forma conseguimos, por exemplo, obter um valor central calculado com a função `mean()`, mas que seja uma média truncada dos 20% valores extremos, como vemos no exemplo acima.

Existem ainda outras utilizações deste parâmetro especial, mas mais uma vez estão fora do âmbito deste texto, pelo que o leitor interessado deverá consultar bibliografia mais avançada sobre a linguagem R.

Parâmetros formais e actuais

Para além das questões ligadas aos parâmetros formais (os parâmetros usados na definição da função), que descrevemos até agora, existem ainda questões ligadas aos parâmetros actuais (os usados nas chamadas às funções), que agora detalhamos.

Em particular, como já vem acontecendo em diversos exemplos fornecidos ao longo deste texto, existem duas formas de indicar os valores dos parâmetros com os quais pretendemos chamar uma qualquer função: através de posição, ou por nome.

Uma chamada por posição ocorre quando pretendemos que o valor indicado na posição x da lista de argumentos da função seja associado ao argumento formal na mesma posição. Por exemplo, ao executar `seq(10,23)` o R vai assumir que o valor 10 é para ser atribuído ao primeiro argumento formal da função `seq()` (o argumento *from* no caso, conforme pode confirma na ajuda da função), e o valor 23 no segundo argumento formal (o argumento *to*). Isto leva a obter uma sequência de números de 10 a 23, de 1 em 1.

Já a chamada `seq(10,length=23)` tem um resultado completmente diferente, uma vez que o segundo valor é indicado por nome e não por posição como no exemplo anterior. Isto quer dizer que o número 23 não é atribuído ao argumento na segunda posição, como anteriormente, mas sim ao argumento formal com o nome *length*.

A utilização da chamada por nome, além de mais clara, é por vezes indispensável, nomeadamente em funções com muitos argumentos, a maioria com valores por defeito, em que o utilizador pretende usar um valor diferente num dos argumentos do fim da lista de argumentos formais. Sem esta facilidade o utilizador iria ter que "preencher" todos os outros argumentos até "chegar" ao argumento que pretende alterar. Com a chamada por nome, basta-lhe proceder como vimos no exemplo de cima, indicando o nome do argumento formal e o valor que pretende usar.

3.3.4 *Lazy evaluation*

O que é a *lazy evaluation*?

Conforme já foi mencionado na Secção 3.3.2, sempre que é chamada uma função o R cria um novo ambiente por baixo do ambiente onde a chamda foi efectuada. Nessa altura as expressões dos argumentos actuais são verificadas sintaticamente, mas não são avaliadas. A este procedimento chama-se *lazy evaluation*. Os argumentos actuais só são avaliados quando são necessários pela primeira vez no corpo da função. Esta regra pode ter por vezes implicações inexperadas conforme ilustrado nestes pequenos exemplos:

```
> f1 <- function(a1, a2 = sqrt(a1)) {
+     a1 <- a1^2
+     a2
+ }
> f1(4)

[1] 4

> f2 <- function(a1, a2 = sqrt(a1)) {
+     z <- a2/a1
+     a1 <- a1^2
+     a2
+ }
> f2(4)

[1] 2
```

Na primeira função, embora o argumento *a2* seja definido com uma valor por defeito igual à raíz quadrada do primeiro argumento, quando chamamos a função com o valor 4, poderíamos estar à espera que o argumento *a2* (que por sinal é o valor retornado pela função), tomasse o valor 2. Todavia, devido à *lazy evaluation*, a expressão `sqrt(a1)` só é calculada quando necessário no corpo da função, e isso só acontece na última instrução. Acontece que nessa altura *a1* já não é 4 mas sim 16, devido à primeira instrução do corpo da função, e portanto `sqrt(a1)` vai dar 4 e não 2 como poderíamos esperar.

Já na segunda função, porque *a2* é necessário na primeira instrução, logo aí é inicializado com o valor `sqrt(a1)`, que nessa altura é igual a 2. Daí o resultado diferente desta segunda função.

3.3.5 Algumas funções úteis

O R possui uma enorme quantidade de funções. Nesta secção apresentamos uma brevíssima resenha de algumas funções úteis, agrupadas pelo seu tipo de funcionalidade. Esta lista está obviamente muito longe de ser exaustiva.

Algumas estatísticas básicas

`sum(x)`	Soma dos elementos do vector `x`.
`max(x)`	Máximo dos elementos de `x`.
`min(x)`	Mínimo dos elementos de `x`.
`which.max(x)`	O índice do maior valor em `x`.
`which.min(x)`	O índice do menor valor em `x`.
`range(x)`	O *range* de valores em `x` (produz o mesmo resultado que `c(min(x),max(x))`).
`length(x)`	O número de elementos em `x`.
`mean(x)`	A média dos valores em `x`.
`median(x)`	A mediana dos valores em `x`.
`sd(x)`	O desvio padrão dos elementos em `x`.
`var(x)`	A variância dos elementos em `x`.
`quantile(x)`	Os quartis de `x`.
`scale(x)`	Normaliza os elementos em `x`, *i.e.* subtrai a média e divide pelo desvio-padrão, resultando num vector de números com média zero e desvio-padrão unitário. Também funciona com *data frames* (só as colunas numéricas, obviamente).

Algumas operações vectoriais e matemáticas

`sort(x)`	Elementos de `x` ordenados.
`rev(x)`	Inverte a ordem dos elementos em `x`.
`rank(x)`	Faz o *ranking* dos elementos de `x`.
`log(x,base)`	Calcula o logaritmo na base "`base`" de todos os elementos de `x`.
`exp(x)`	Calcula o exponencial dos elementos de `x`.
`sqrt(x)`	Raiz quadrada dos elementos de `x`.
`abs(x)`	Valor absoluto dos elementos de `x`.
`round(x,n)`	Arredonda os valores em `x` para `n` casas decimais.
`cumsum(x)`	Obtém um vector em que o elemento i é a soma dos elementos $x[1]$ até $x[i]$.
`cumprod(x)`	O mesmo para o produto.
`match(x,s)`	Obtém um vector com o mesmo tamanho de `x`, contendo os elementos de `x` que pertencem a `s`. Os elementos que não pertencem a `s` aparecem no resultado assinalados com o valor NA.
`union(x,y)`	Obtém um vector com a união dos vectores `x` e `y`.
`intersect(x,y)`	Obtém um vector com a intersecção dos vectores `x` e `y`.
`setdiff(x,y)`	Obtém um vector resultante de retirar os elementos de `y` do vector `x`.
`is.element(x,y)`	Retorna o valor TRUE se `x` está contido no vector `y`.
`choose(n,k)`	Calcula o número de combinações `k`, `n` a `n`.

Álgebra Matricial

`diag(x,nrow,ncol)`	Constrói uma matriz diagonal com `nrow` linhas e `ncol` colunas, usando o número `x`. Também pode ser usada para extrair ou substituir os elementos na diagonal de uma matriz (ver exemplos fazendo "`? diag`").
`t(x)`	A matriz transposta de `x`.
`nrow(x)`	Número de linhas de `x`.
`ncol(x)`	Número de colunas de `x`.
`A %*% B`	Multiplicação matricial de `A` por `B`.
`solve(A,b)`	Resolve o sistema de equações lineares $Ax = b$. Com um único argumento (uma matriz) (*e.g.* `solve(A)`) calcula a sua inversa.
`qr(x)`	Decomposição QR da matriz `x`.
`svd(x)`	Decomposição SVD (*singular value decomposition*) da matriz `x`.
`eigen(x)`	Valores e vectores próprios da matriz quadrada `x`.
`det(x)`	O determinante da matriz quadrada `x`.

3.4 Objectos, Classes e Métodos

Esta secção inclui informação que sai um pouco fora do âmbito introdutório deste texto. De facto, existe muito para dizer sobre estes tópicos no contexto da linguagem R. Todavia, mesmo tratando-se de uma introdução, há questões relacionadas com estes tópicos que qualquer utilizador do R deve conhecer.

A maioria dos objectos que vimos até agora tem uma estrutura relativamente simples. Todavia, é possível em R construír novos tipos (classes) de objectos, usando estes tipos elementares que temos vindo a descrever. Por exemplo, existem funções em R que retornam como resultado um modelo de um conjunto de dados (por exemplo uma rede neuronal). Tal objecto tem uma complexidade bem maior do que os que estudamos até agora, não deixando no entanto de ser um objecto como outro qualquer do ponto de vista do R. Isto quer dizer que por exemplo pode fazer sentido perguntar ao R qual o conteúdo deste objecto complexo. Para tal ser possível, e de forma transparente para o utilizador, é conveniente que a pessoa que decidiu criar tais objectos complexos, indique também ao R como mostrá-los ao utilizador. Traduzido de outro modo, o utilizador deverá indicar ao R um método para mostrar um objecto da classe que acabou de criar.

Muita desta aparente complexidade, pode ser "escondida" do utilizador comum. Por exemplo, a função `summary()` produz um sumário do conteúdo do objecto que lhe é fornecido como argumento. O significado deste sumário, bem como a forma como ele é obtido, está em grande parte dependente do tipo de objecto de que queremos o sumário. No entanto o utilizador não precisa de saber isso. Vejamos um exemplo concreto deste comportamento,

```
> x <- rnorm(10)
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.3790 -0.2195 -0.1066  0.1157  0.8568  1.3760

> summary(matrix(x, 5, 2))

       V1                 V2
 Min.   :-0.6689   Min.   :-1.37895
 1st Qu.:-0.1069   1st Qu.:-0.25349
 Median :-0.1062   Median :-0.11738
 Mean   : 0.3167   Mean   :-0.08521
 3rd Qu.: 1.0894   3rd Qu.: 0.15905
 Max.   : 1.3758   Max.   : 1.16472
```

Repare como duas chamadas à mesma função (na realidade só aparentemente a mesma como veremos), produzem resultados diferentes dependendo da classe do objecto dado como argumento, embora conceptualmente ambos sejam sumários desse argumento. De facto, a função `summary()` é uma função genérica, que basicamente o que faz é ver a classe do objecto que lhe é passado como argumento, e depois "despacha" a tarefa de produzir o sumário para funções especializadas nessa classe de objectos. Para isso, obviamente é preciso que alguém tenha escrito essas funções especializadas, ou seja que alguém tenha fornecido uma método **summary** para a classe de objectos em causa (no caso do exemplo um vector e uma matriz). Podemos rapidamente confirmar isto que se está a passar "por detrás" da função `summary()` executando o seguinte,

Funções genéricas

Métodos

```
> methods(summary)
 [1] summary.aov            summary.aovlist        summary.connection
 [4] summary.data.frame     summary.Date           summary.default
 [7] summary.ecdf*          summary.factor         summary.glm
[10] summary.infl           summary.lm             summary.loess*
[13] summary.manova         summary.matrix         summary.mlm
[16] summary.nls*           summary.packageStatus* summary.POSIXct
[19] summary.POSIXlt        summary.ppr*           summary.prcomp*
[22] summary.princomp*      summary.stepfun        summary.stl*
[25] summary.table          summary.tukeysmooth*

   Non-visible functions are asterisked
```

A função `methods()` mostra a lista de métodos (isto é funções especializadas em classes particulares) que existem para a função genérica `summary()`. Podemos ver, por exemplo que existe uma função `summary.matrix()`, que como o nome indica, é a função usada para produzir sumários para objectos da classe "matrix".

Esta característica da linguagem R, e das linguagens orientadas aos objectos genericamente, é bastante conveniente, uma vez que permite uma interacção com o utilizador mais transparente. A este, basta-lhe saber que a função `summary()` produz sumários de objectos, sejam eles de que tipo forem, não necessitando por exemplo de conhecer todas as funções específicas que existem para produzir os sumários para cada classe de objecto.

A função `class()` permite-nos saber qual a classe de um qualquer objecto,

Classes

```
> x <- rnorm(10)
> class(x)

[1] "numeric"

> class(matrix(x, 5, 2))

[1] "matrix"

> class(as.POSIXct("2006-9-23"))

[1] "POSIXt"  "POSIXct"
```

Para criarmos uma nova classe basta-nos criar uma função que possa ser usada para criar essa classe de objectos e depois podemos escrever funções que implementem alguns métodos que sejam úteis para objectos dessa nova classe.

Criar um nova classe

Vejamos um exemplo deste processo criando uma nova classe de objectos que vise armazenar dados sobre pessoas. Para simplificar vamos assumir que um objecto da classe "pessoa" é formado por um nome, os nomes dos pais, a data de nascimento e o sexo da pessoa.

```
> pessoa <- function(n, p, d, s) {
+     p <- list(nome = n, pais = p, nascimento = d, sexo = s)
+     class(p) <- "pessoa"
```

```
+     p
+ }
> chefe <- pessoa("Ana", list(pai = "Carlos", mae = "Joana"), as.Date("1970-03-3"),
+     "f")
> class(chefe)

[1] "pessoa"
```

Repare como, depois de criar a função `pessoa()`, podemos usá-la para criar objectos da classe "pessoa". É este o caso do objecto **chefe** que como vemos tem essa classe.

Em seguida podemos ver como criar uma função para implementar o método de sumarização de objectos da classe "pessoa".

```
> summary.pessoa <- function(p) cat("O meu nome é ", p$nome, " e sou ",
+     ifelse(p$sexo == "f", "filha", "filho"), " de ", p$pais$pai,
+     " e ", p$pais$mae, " tendo nascido a ", format(p$nascimento,
+         "%d %b %Y"), "\n")
> summary(chefe)

O meu nome é  Ana  e sou  filha  de  Carlos  e  Joana  tendo nascido a  03 Mar 1970
```

Após a criação deste método passamos a poder usar a função `summary()` aplicada a objectos da classe "pessoa". Uma vez que criamos a função `summary.pessoa()`, quando chamamos `summary(chefe)`, o R vai começar por ver qual a classe do objecto **chefe** e, uma vez que essa é "pessoa", vai procurar uma função com o nome `summary.pessoa()`. Assim, se pretendemos criar uma método específico para a classe "pessoa" para uma função genérica qualquer "xpto", o que temos que fazer é criar a função "xpto.pessoa". Por exemplo, se pretendessemos aplicar a função genérica `plot()` a objectos da classe "pessoa", precisariamos de criar uma função chamada `plot.pessoa()`.

3.5 Depuração de Programas em R

O que é?

A depuração de programas em qualquer linguagem (*debugging*) de programação tem como objectivo principal encontrar a fonte de erros ou comportamentos imprevistos nos programas que desenvolvemos.

O R possui várias funções que podem ser utilizadas para ajudar nesta tarefa. Uma delas é a função `traceback()` que permite ao utilizador obter a sequência de funções que foram chamadas até o erro ocorrer.

Se tal não fôr suficiente para identificar a fonte do erro, ou então se localizamos a sua possível origem mas não entendemos o porquê do erro, poderemos também socorrer-mo-nos da função `browser()`. Esta função se introduzida no código de uma função onde suspeitamos existir algo de errado, fará com que a partir da sua posição, todas as instruções seguintes sejam executadas passo a passo e unicamente sob a nossa ordem. Ou seja, depois de encontrar `browser()` o R vai parar e só executará a instrução seguinte quando nós mandarmos, parando logo em seguida de novo. Em cada uma destas paragens o R vai mostrar um *prompt* especial onde o utilizador poderá fazer quase tudo o que pode fazer no *prompt* normal do R (por exemplo ver o conteúdo de objectos da função a ser depurada), mas onde existem ainda sertos comandos especiais ligados à função `browser()`.

Vejamos um pequeno exemplo deste tipo de depuração. Suponhamos que temos a seguinte função, que aparenta estar sem problemas,

```
> er <- function(n,x2) {
+   s <- 0
+   for(i in 1:n) {
+     s <- s + sqrt(x2)
+     x2 <- x2-1
+   }
```

```
+   s
+ }
> er(2,4)
[1] 3.732051
```

Suponhamos agora que fazemos a seguinte chamada que inesperadamente produz um aviso sobre um erro numérico (**NaN** significa *Not A Number*) na execução da função,

```
> er(3,1)
[1] NaN
Warning message:
NaNs produced in: sqrt(x2)
```

Vamos incluir uma chamada à função `browser()` no corpo da função antes do local que suspeitamos ser a fonte do aviso. Voltamos a fazer a mesma chamada e usamos então as facilidades da execução passo a passo, como vemos a seguir,

```
> er <- function(n,x2) {
+   s <- 0
+   browser()
+   for(i in 1:n) {
+     s <- s + sqrt(x2)
+     x2 <- x2-1
+   }
+   s
+ }
> er(3,1)
Called from: er(3, 1)
Browse[1]> x2
[1] 1
Browse[1]> print(n)
[1] 3
Browse[1]> n
debug: for (i in 1:n) {
    s <- s + sqrt(x2)
    x2 <- x2 - 1
}
Browse[1]> n
debug: i
Browse[1]> i
NULL
Browse[1]> n
debug: s <- s + sqrt(x2)
Browse[1]> i
[1] 1
Browse[1]> s
[1] 0
Browse[1]> sqrt(x2)
[1] 1
Browse[1]>
debug: x2 <- x2 - 1
Browse[1]> s
[1] 1
Browse[1]>
debug: i
Browse[1]> x2
[1] 0
Browse[1]>
```

```
debug: s <- s + sqrt(x2)
Browse[1]> i
[1] 2
Browse[1]> sqrt(x2)
[1] 0
Browse[1]>
debug: x2 <- x2 - 1
Browse[1]>
debug: i
Browse[1]>
debug: s <- s + sqrt(x2)
Browse[1]> sqrt(x2)
[1] NaN
Warning message:
NaNs produced in: sqrt(x2)
Browse[1]> x2
[1] -1
Browse[1]> Q
>
```

Quando chamamos de novo a função o R pára quando encontra a instrução `browser()` fazendo aparecer um *prompt* com a forma genérica `Browse[i]>`. A partir daqui está nas nossas mãos o que pretendemos fazer. Tipicamente começamos por inspeccionar o valor dos objectos da função para ver se econtramos algo suspeito que possa estar a orginar o erro. Nesta sessão exemplo começamos por interrogar o R sobre o valor dos parâmetros, *x2* e *n*. No caso deste último, repare que usamos a construção `print(n)` e não simplesmente o nome do objecto como fizemos para o *x2*. A explicação para isto reside no facto de neste ambiente de *debugging* o N seguido de ENTER tem um significado especial que é dizer ao R, "podes avançar para a próxima instrução", que foi o que fizemos a seguir. O mesmo efeito pode ser conseguido carregando unicamente na tecla ENTER. E por aí continuamos até que reparamos que o erro está a ser originado por uma chamada `sqrt(x2)`, e quando perguntamos o valor de *x2* na altura da chamada que deu o erro, verificamos que ele é -1, e daí o erro uma vez que o R não sabe calcular raízes quadradas de números negativos. Ou seja, a nossa função vai decrementando o valor de *x2*, dentro do ciclo e se este fôr executado bastantes vezes o *x2* acabará por ficar negativo, em particular se fôr inferior a *n* que é o paràmetro que controla o número de execuções do ciclo. Descoberto o erro podemos abortar este ambiente de *debugging* usando o comando Q. Um outro comando deste ambiente de *debugging* que por vezes é útil é o comando C. Se executado quando o *debugger* está dentro de um ciclo, faz com que o R deixe execute o resto das iterações do mesmo até ao fim, sem qualquer interrupção só voltando a aparecer o *prompt* no fim destas.

Em vez de alterar a função que pretendemos depurar, podemos simplesmente fazer `debug(er)` e depois fazer a chamada problemática, `er(3,1)`. A intereacção que se segue é em tudo idêntica à que vimos acima. Note-se que até fazermos `undebug(er)`, sempre que voltamos a chamar a função `er()` o R entra em modo de *debugging*. A vantagem da função `browser()` em relação a esta alternativa é que podemos iniciar o *debugging* no ponto da função que queremos, enquanto que com esta alternativa ele começa logo deste a primeira instrução da função.

4 Manipulação de Dados

O R tem vários objectos onde podemos armazenar diversos tipos de dados. No entanto, conforme já mencionado, são os *data frames* que se revelam como os mais adequados para armazenar tabelas de dados de um problema qualquer. Na Secção 2.10 vimos já o que é um *data frame*, como os criar, bem como algumas operações essenciais para a sua manipulação. Nesta secção iremos analisar mais algumas questões importante para analisar dados guardados em *data frames* usando as potencialidades do R. Nomeadamente iremos ver: como carregar dados de várias fontes para um *data frame*, como obter sumários das propriedades principais dos dados, e também técnicas para visualizar esses dados.

Dados que vêm com o R

Antes de abordarmos estes tópicos só uma pequena nota sobre o facto de o R ter já disponíveis diversas tabelas de dados que poderemos usar para por exemplo testar as nossas capacidades com a linguagem. Além disso, a maioria das *packages* extra que vamos instalando vem também com mais conjuntos de dados para ilustrar algumas das suas funcionalidades. Para sabermos os dados que estão actualmente (isto é com as *packages* neste momento carregadas) disponíveis, podemos fazer:

```
> data()
```

Como resultado é-nos apresentada um lista com os nomes e breves descrições dos conjuntos de dados disponíveis. Podemos obter mais informação sobre qualquer um deles, usando as funções de ajuda aplicadas ao nome do conjunto de dados, como se de uma função se tratasse.

Se pretendermos usar algum dos conjuntos de dados podemos usar a mesma função,

```
> data(cars)
```

O efeito disto é criar um *data frame* com o nome **cars** contendo este conjunto de dados.

4.1 Carregar dados para o R

Nesta secção iremos analisar formas de clocar em *data frames*, dados que se encontram armazenados noutras plataformas e que, pela sua dimensão, não faz sentido introduzi-los de novo "à mão" no R. Dito de outra forma, iremos ver como importar dados para o R.

4.1.1 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.

Formato CSV

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()`. Esta função tem inúmeros parâmetros que permitem ajustar a importação aos detalhes particulares do formato dos dados no ficheiro que pretendemos carregar. Entre eles, está obviamente uma parâmetro que permite explicitar o separador dos valores que é usado (o parâmetro *sep*) 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. Todavia, por várias destas alternativas serem muito comuns, existem 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()*. Para

A função read.table

ver todos os detalhes das pequenas diferenças entre elas aconselhamos a leitura da ajuda dessas funções que, pela sua semelhança, está toda junta.

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
...
```

Olhando para o ficheiro com atenção poderemos reparar que os valores são separados por ponto e vírgula, os números reais usando o ponto como separador das casas decimais, e o ficheiro inclui o nome das colunas na primeira linha. Todas estas particularidades têm correspondência em parâmetros das função de leitura que mencionamos acima. Se pretendessemos usar a função `read.table()` para ler este ficheiro, deveriamos fazer:

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

O parâmetro *header* permite-nos indicar se o ficheiro tem, ou não, o nome das colunas na primeira linha de dados. O parâmetro *sep* permite conforme já mencionado, indicar o separador de valores usado no ficheiro. Finalmente o parâmetro *dec* permite indicar o caracter usado como separador de casas decimais dos números reais. Note-se que este formato é bastante vulgar, sendo que 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')
```

Ler ficheiros muito grandes

Estas são as funções principais para ler ficheiros de texto no R. Existem ainda outras possibilidades, embora menos usadas, das quais destacaríamos só a função `read.fwf()`, que permite ler ficheiros que usem um formato de tamanho fixo. Deixamos ainda 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.

Ler ficheiros muito grandes

4.1.2 Da Internet

Por vezes existem dados que nos interessam que estão acessíveis em páginas da Internet. Para sabermos qual a melhor forma de os importar para o R temos que analisar com cuidado o formato em que são fornecidos. Existem aqui várias possibilidades. 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 na Secção 4.1.1. 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()`. Vejamos,

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

O primeiro argumento indica o URL, e o segundo o nome do ficheiro local no nosso computador onde os dados devem ser guardados. Note que se indicar um caminho para o ficheiro, como no exemplo acima, deverá separar o nome das pastas por dois caracteres "\\" e não um só como estaria à espera.

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.

4.1.3 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. Um consiste em gravar os dados do *Excel* para um ficheiro CSV, com as facilidades que o *Excel* tem para isso, e depois usar alguma das funções da Secção 4.1.1.

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*[13], 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 <-
```

4.1.4 De bases de dados

As bases de dados são as infraestruturas por excelência para armazenar dados, particularmente quando de grande dimensão, no contexto das organizações. Neste contexto, não será surpreendente descobrir que 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, sendo o seu objectivo exactamente tornar o código em R independente (ou pelo menos o mais possível) desse software. A menção ao SGBD a usar é feita unicamente quando se faz a ligação entre o R e esse software, sendo tudo o resto independente de qualquer que seja o SGBD usado. Para tornar tudo 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**. A ideia geral desta arquitectura de comunicação entre o R e os SGBD's pode ser melhor descrita pela Figura 7.

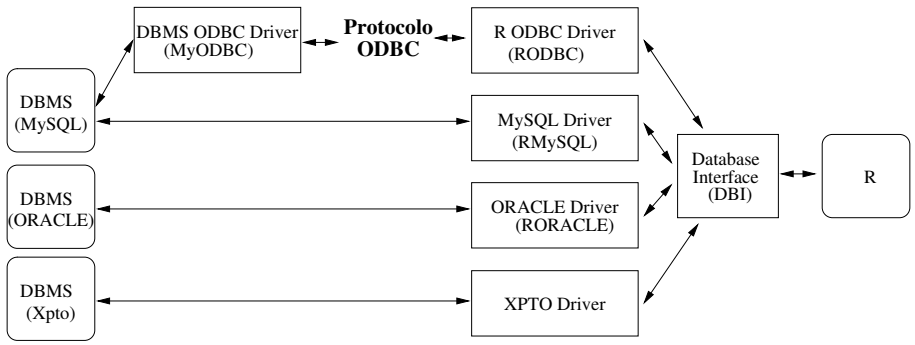


Figura 7: A arquitectura de comunicação entre o R e diferentes SGBD's.

[13]Note que esta estratégia é espcífica a versões *Windows* do R.

Ou seja a maioria do tempo o utilizador só "vê" as funções da *package* **DBI**, e portanto para ele é tudo praticamente igual independentemente do SGBD que se está a usar como fonte de dados.

Neste texto 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**. A ideia de usar o protocolo ODBC está descrita na Figura 7. Temos o *package* genérico **DBI** que vai "falar" com o *package* **RODBC** que também precisamos de instalar. Este por sua vez vai falar com as bases de dados no **MySQL** através de um *driver* que precisamos instalar no *Windows* que de uma forma simplista traduz a linguagem do protocolo ODBC para algo entendível pelo SGBD **MySQL**.

Da primeira vez que pretendemos fazer o interface a uma base de dados no **MySQL** usando o protocolo ODBC, são necessários alguns passos extra. Estes passos só são efectuados da primeira vez. Em particular precisamos de instalar o driver ODBC do **MySQL** que podemos obter no *site* deste SGBD (`http://www.mysql.com`). Depois de termos este *driver* (myodbc é o seu nome) instalado, já podemos fazer ligações ao **MySQL** usando o protocolo ODBC. De acordo com este protocolo cada ligação a uma base de dados tem um nome (o *Data Source Name*, ou *DSN* na linguagem ODBC). Este nome vai ser usado para nos referirmos à base de dados quando pretendermos importar dados do lado do R. Para criar uma ligação ODBC em *Windows* temos de usar um programa chamado *ODBC data sources* que está disponível no *Control Panel* do *Windows*. Depois de excutar este programa 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, que voltamos a referir só é feito a primeira vez que pretendemos criar uma ligação a um determinada base de dados, estamos prontos a, desde o R, establecer ligações à base de dados usando o protocolo ODBC. O exemplo que mostramos em seguida, establece 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)
```

As primeiras duas instruções carregam as *packages* necessárias para importar os dados. As duas seguintes establecem a ligação à base de dados, sendo usado o *DSN* que criamos no programa *ODBC Data Sources*, bem como introduzido o nosso nome de utilizador e respectiva *password* para podermos aceder à base de dados do **MySQL**. Em seguida vem a instrução principal para irmos importar dados de qualquer tabela da nossa base de dados[14]. Neste caso estamos a importar todos as linhas e colunas de uma tabela chamada "tabela" da nossa base de dados. Note-se que para conseguirmos lidar com este tipo de bases de dados relacionais vamos precisar de saber SQL, que é a linguagem de consulta por excelência deste tipo de bases de dados. O resultado da função `dbQuery()` é um *data frame*.

Finalmente, as últimas instruções fecham a ligação à base de dados. Note-se que a *package* **DBI** define ainda várias outras funções que permitem, por exemplo, fazer a

[14]Note-se que esta instrução já é completamente independente do SGBD que estamos a usar e seria portanto igual para qualquer outro sistema que não o **MySQL**.

operação inversa, ou seja enviar dados de um *data frame* do R para uma tabela de uma base de dados.

4.2 Sumarização de dados

Assim que temos os nossos dados no R, usando algum dos processos indicados, podemos começar a analisá-los. Nesta secção descrevemos algumas funções que indicam algumas características dos nossos conjuntos de dados. Para efeitos de ilustração destas funções vamos carregar o conjunto de dados chamado *iris* que vem com o R,

```
> data(iris)
```

Este conjunto de dados descreve em cada linha uma planta através de algumas das suas biometrias, além da espécie a que pertence.

Uma das primeiras coisas que podemos querer saber são as dimensões dos nossos dados,

```
> nrow(iris)

[1] 150

> ncol(iris)

[1] 5
```

Estatísticas descritivas

A função `summary()` fornece-nos algumas estatísticas descritivas básicas,

```
> 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
```

Análise por sub-grupos

Por vezes interessa-nos ter este tipo de análise descritiva por 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
```

Por vezes pretendemos unicamente ver os dados, ou parte deles. Neste tipo de tarefas, nem sempre é prático escrevermos o nome do *data frame* devido à dimensão dos dados. Nestas situações, existem algumas funções úteis como por exemplo as seguintes,

```
> 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
```

```
> 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 1 ...
```

As primeiras duas funções mostram as primeiras e últimas linhas do *data frame*, enquanto que a função `str()` dá-nos um sumário global do *data frame*.

Finalmente, os esquemas de indexação que estudamos na Secção 2.6 podem ser usados para obter sub-grupos de dados, aos quais podemos aplicar alguma função de sumarização se desejarmos, como os seguintes exemplos ilustram,

```
> summary(iris[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.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, c(Petal.Width, Species)))

  Petal.Width          Species
 Min.   :1.800   setosa    :0
 1st Qu.:2.000   versicolor:0
 Median :2.100   virginica :9
 Mean   :2.122
 3rd Qu.:2.300
 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
```

Note que as duas primeiras instruções produzem exactamente os mesmo resultados. A segunda usa a função `subset()`, que é uma forma alternativa de fazermos indexação de estruturas como os *data frames*, mas que pode revelar-se mais prática em certas situações. As instruções seguintes mostram mais alguns exemplos do uso desta função.

Por vezes estamos interessados em produzir sumários particulares dos dados que melhor se adequem às nossas necessidades. Por exemplo, poderiamos pretender ter um tabela que para cada coluna de um *data frame* nos mostrasse o seu valor médio, o máximo, o mínimo, o desvio padrão, a variância e o número de valores desconhecidos. Uma forma simples de conseguir tais efeitos é através da criação de funções próprias que produzam os resultados pretendidos. Na Secção 3.3.1 iremos ver em detalhe como criar funções, no entanto aqui deixamos uma ilustração de como usar essas funções criadas para obter os sumários que pretendemos.

```
> 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
```

A função `apply()`, que já mencionamos na Secção 3.2.3, permite-nos aplicar a todas as colunas de um *data frame* uma função qualquer, produzindo desta forma um conjunto de resultados para cada coluna. Neste caso aplicamos a função que criamos para obter as estatísticas que pretendiamos.

4.3 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. Começemos por exemplos de uso de fórmulas no contexto da obtenção de modelos. O exemplo seguinte obtém um modelo de regressão linear que relaciona a variável "Petal.Length" com a variável "Sepal.Width", usando os dados do conjunto de dados "iris".

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

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

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

Os exemplos seguintes obtêm modelos do mesmo tipo que relacionam a variável "Petal.Length" com a todas as interacções entre "Sepal.Width" e os valores de "Species" no caso do primeiro, enquanto o segundo exemplo faz o mesmo com todas as combinações possíveis entre "Sepal.Width" e os valores de "Species".

```
> 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
```

Na Figura 8 podemos ver um exemplo de um gráfico (neste caso um "boxplot") que nos dá uma ideia da distribuição da variável "Petal.Width" para os sub-conjuntos dos dados correspondentes a cada valor da variável "Species", o que é produzido pelo seguinte código,

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

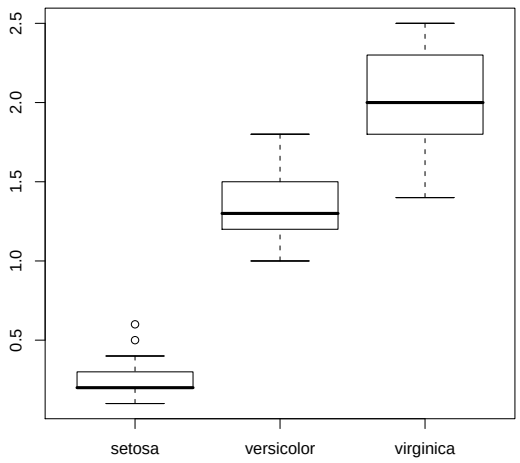


Figura 8: Um gráfico obtido com uma fórmula.

4.4 Visualização de dados

O R tem como um dos seus pontos mais fortes a flexibilidade e poder em termos de visualização de dados. De facto, para além de uma série de funções de alto nível que produzem os gráficos mais comuns neste tipo de análises, o R possibilita ao utilizador,

através de um conjunto de funções de mais baixo nível, alterar e produzir gráficos que melhor se adequem ao seu problema. Neste documento iremos analisar alguns dos aspectos mais básicos deste vasto assunto. Para informação mais detalhada pode ser consultado o interessante livro *R Graphics* por Murrell (2006).

Sistemas de gráficos

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:

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

***Devices* gráficos**

Independentemente do tipo de gráficos que produzimos eles "vão parar" a um determinado *device* gráfico. Por defeito, e na maioria das situações, esse *device* vai ser o écran. No entanto, existem situações em que podemos crer mudar o *device* de saída dos gráficos, como por exemplo se pretedermos colocar o gráfico num ficheiro PDF. A escolha do *device* define não só o local onde o gráfico é produzido, mas também o tipo de *output*.

Por exemplo, se pretendemos que um determinado gráfico vá parar a um ficheiro em formato PDF, podemos fazer algo do género,

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

Se quisessemos 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.

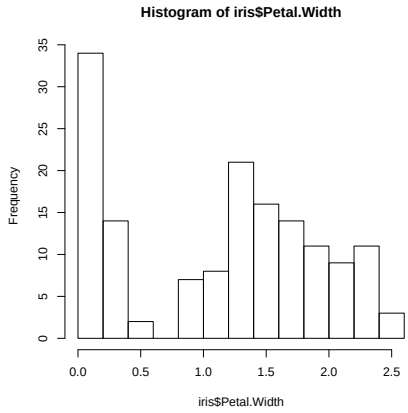


Figura 9: Um exemplo de um histograma.

4.4.1 Gráficos Univariados

Quando pretendemos visualizar um única variável, em particular se ela é contínua, temos ao nosso dispôr várias funções de alto nível. Por exemplo, podemos obter um histograma da variável:

stogramas

```
> hist(iris$Petal.Width)
```

O resultado desta instrução pode ser visto na Figura 9.

A Figura 10) mostra uma outra variante de um histograma obtida manipulando alguns dos parâmetros da função `hist()`.

```
> hist(iris$Petal.Width, main = "Histograma de Petal.Width", xlab = "",
+     ylab = "Probabilidade", prob = T)
```

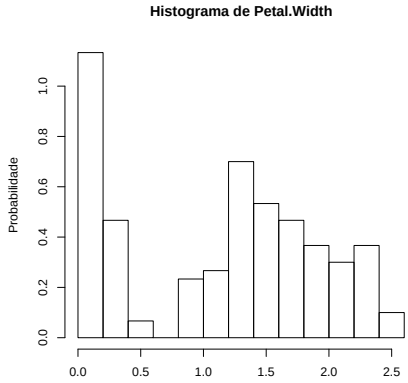


Figura 10: Ainda outro histograma.

Os gráficos de caixa de bigodes (*box plots*) são outros gráficos univariados bastante usados. Podem ser obtidos em R usando a função `boxplot()` da seguinte forma (ver o resultado na Figura 11), **Boxplots**

```
> boxplot(iris$Petal.Length, main = "Petal.Length")
```

A função `barplot()` pode ser usada para obter gráficos de barras, conforme ilustrado no seguinte exemplo (ver Figura 12), criado para nos mostrar quantas plantas existem de cada espécie dentro do subconjunto que tem $Petal.Width > 1$, **Gráficos de barras**

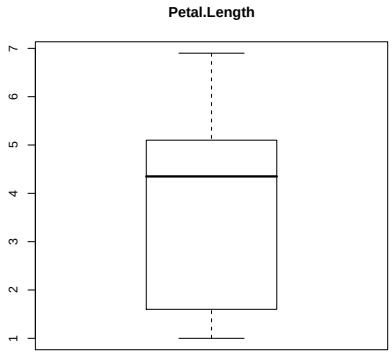


Figura 11: Um gráfico de caixa de bigodes.

```
> barplot(table(subset(iris, Petal.Width > 1)$Species))
```

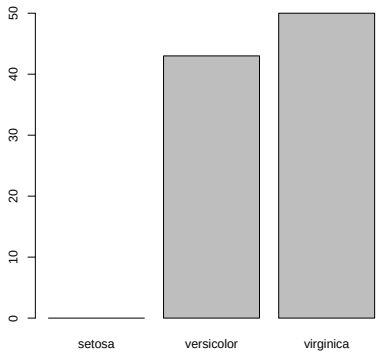


Figura 12: Um gráfico de barras.

4.4.2 Gráficos de 3 Variáveis

As funções principais para obter gráficos com dados que envolvem 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.

Gráficos tri-dimensionais

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")
```

O resultado das instruções apresentadas acima, que pode ser visto na Figura 13, é uma representação tri-dimensional da função $10 \times \frac{\sin(\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$. Para isso criamos uma função (f()) que calcula o valor da referida expressão, para quaisquer valores de x e y. Depois, usamos a função `outer()` para obter os valores dessa função para todas as combinações de um conjunto de valores de x e y.

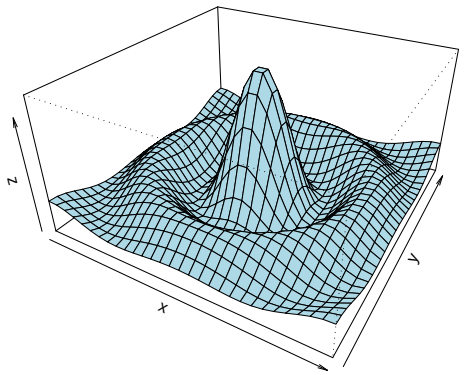


Figura 13: Um gráfico tri-dimensional com a função "persp".

A Figura 14 mostra-nos um outro exemplo de utilização da função `persp()`, usando outras parametrizações, bem como um outro conjunto de dados disponível no R.

```
> 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)
```

Gráficos de curva de nível

Por vezes estamos interessados em obter representações bi-dimensionais de superfícies tri-dimensionais. Isto pode ser conseguido de várias formas. Uma delas consiste em usar curvas de nível que possam transmitir a ideia do valor da terceira dimensão. No R tal efeito pode ser obtido com a função `contour()`. O código seguinte mostra um exemplo da sua utilização com os dados volcano, cujo resultado pode ser visualizado na Figura 15.

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

Gráficos com níveis de cor

Uma alternativa às curvas de nível é representar a terceira dimensão através de várias gradações de cor. Isto mesmo pode ser obtido com a função `image()` do modo seguinte, podendo o resultado ser visto na Figura 16.

```
> image(volcano)
```

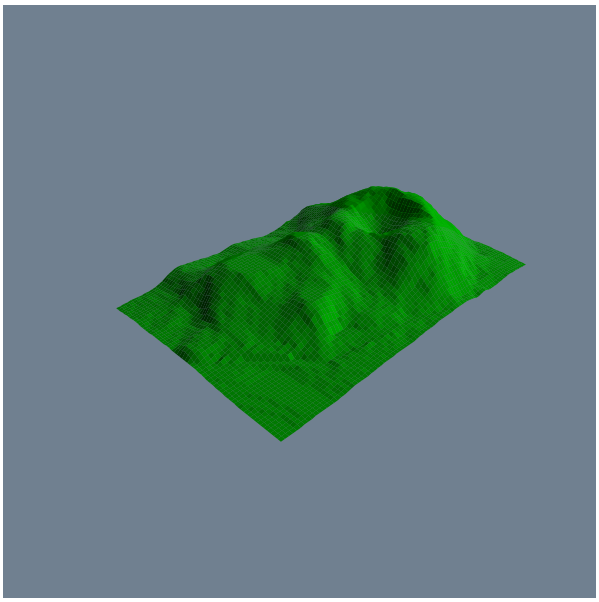


Figura 14: Um outro gráfico tri-dimensional com a função “persp”.

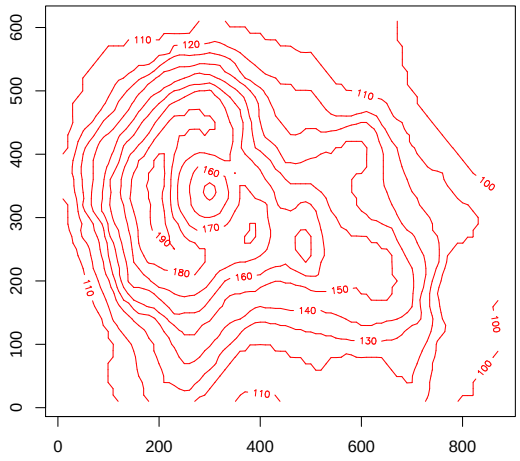


Figura 15: Um gráfico de curvas de nível.

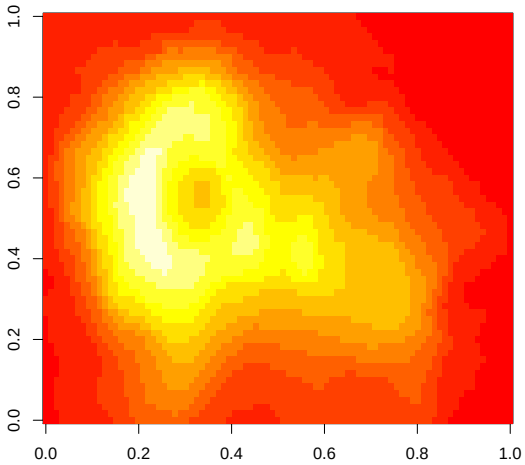


Figura 16: Um gráfico com gradações de cor.

Por fim, na Figura 17 podemos ver uma conjugação das duas ideias: as curvas de nível e as cores:

```
> 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)
```

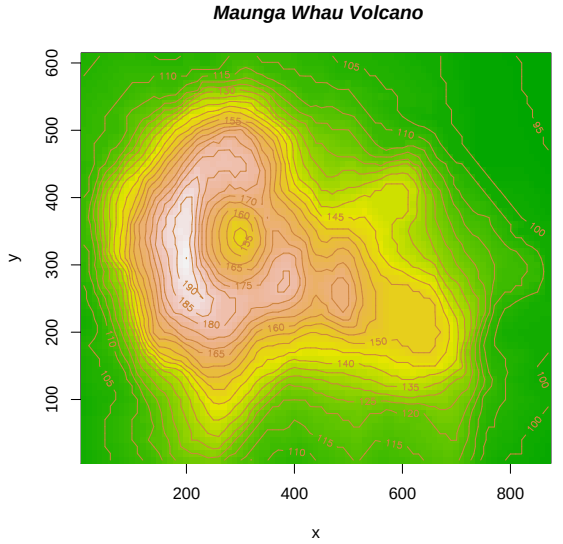


Figura 17: Um gráfico com curvas de nível e gradações de cor.

4.4.3 Gráficos Multivariados

Por vezes pretendemos obter gráficos com dados referentes a mais do que uma variável. função `matplot()`, por exemplo, permite-nos desenhar séries de dados, guardadas numa matriz em que cada coluna tem uma série de dados. Vejamos um exemplo cujo resultado é apresentado na Figura 18,

```
> m <- matrix(rnorm(100), 20, 5)
> op <- par(mfrow = c(1, 2), mar = c(2, 3, 0, 1))
> matplot(m)
> matplot(m, type = "l")
> par(op)
```

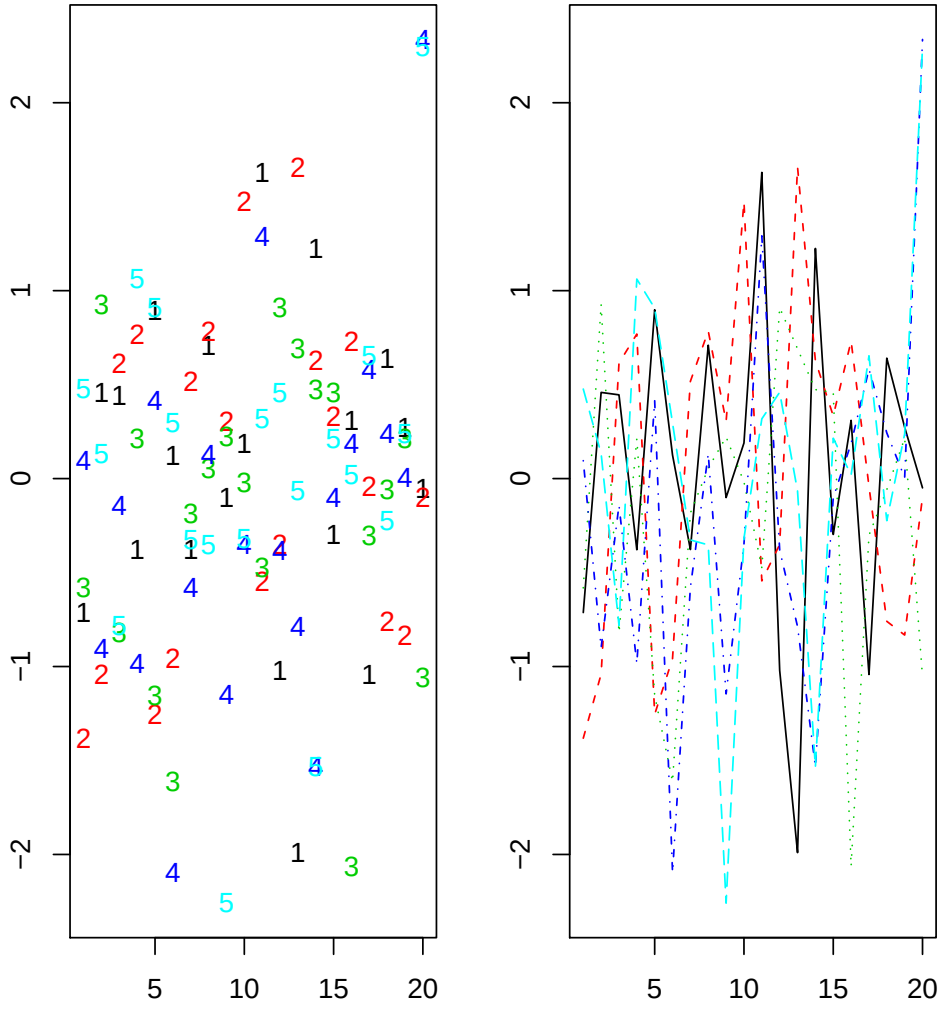


Figura 18: Gráficos de várias séries de dados.

Em R é também possível encontrar diversas funções que são capazes de lidar com *data frames* formados por várias colunas. Obviamente, este tipo de gráficos, muitas vezes resulta em figuras demasiado "cheias" com pouca compreensibilidade, devido à dimensão dos dados. No entanto, em muitas situações, estas são boas ferramentas de análise visual de dados.

Gráficos bivariados

A função `pairs()` é provavelmente a mais comum para este tipo de dados. Ela produz uma matriz simétrica de gráficos bivariados, como podemos ver na Figura 19,

```
> pairs(iris[,1:4])
```

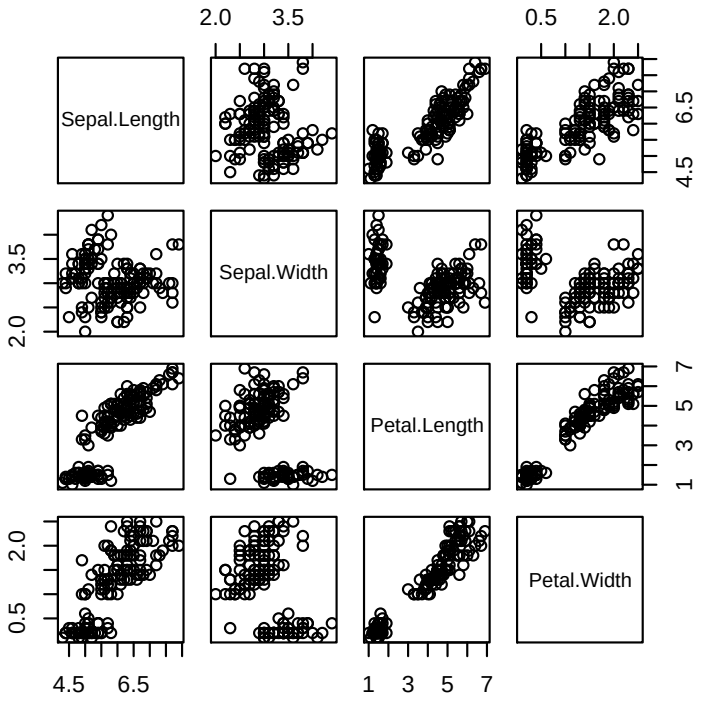


Figura 19: Uma matriz de gráficos bivariados.

Tirando partido da flexibilidade da programação em R é possível modificar grandemente este tipo de funções, fazendo por exemplo o R incluir histogramas de cada uma das variáveis na diagonal da matriz de gráficos da Figura 19, ou mesmo substituir a parte superior da matriz (tirando partido dela ser simétrica), por outros gráficos. Vejamos alguns exemplos deste tipo de modificações.

Na Figura 20 vemos um gráfico do tipo "pairs" onde, na diagonal, acrescentamos histogramas de cada uma das variáveis, e na parte superior direita colocamos os valores da correlação, numa fonte tanto maior quanto maior a correlação. Para isso, criamos duas funções: `panel.hist()` e `panel.cor()` que produzem esses objectivos e que são passadas depois em certos argumentos da função `pairs()`.

```
> 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)
```

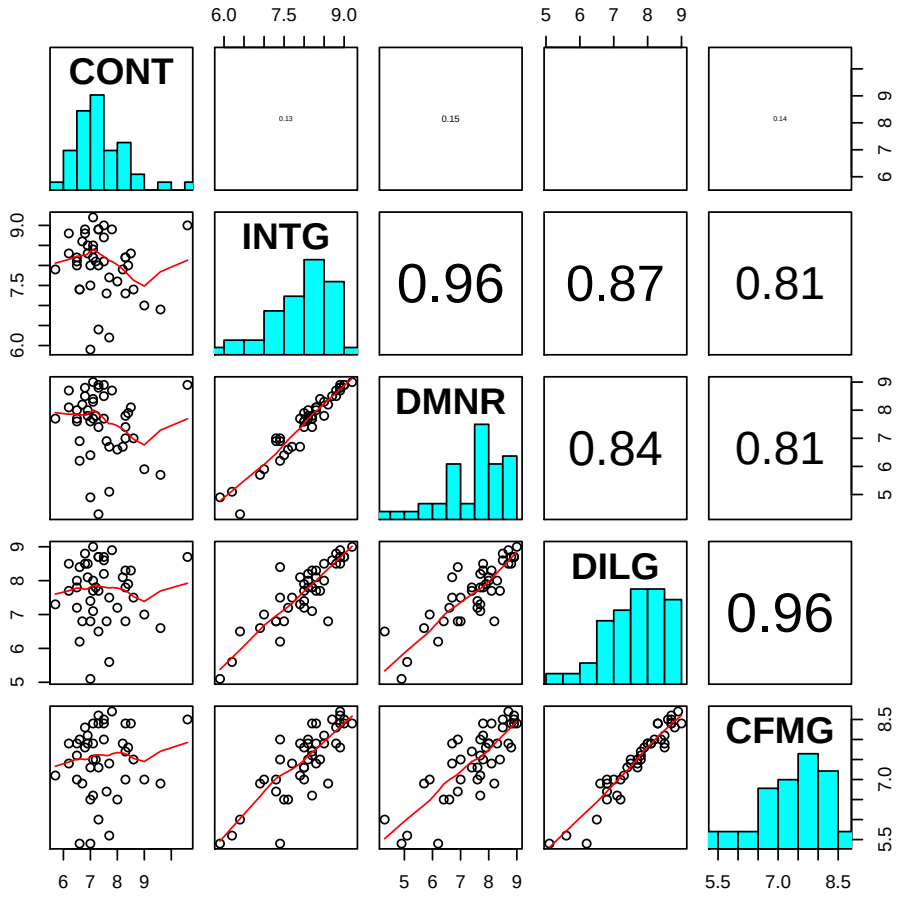


Figura 20: Um gráfico "pairs" com histogramas no meio e valores da correlação no canto superior direito.

Gráficos *stars*

Um outro tipo de gráficos que por vezes é muito interessante uma vez que nos permite comparar observações descritas por várias variáveis contínuas, são os gráficos do tipo *stars*. Estes gráficos mostram cada observação de um *data frame* por uma espécie de estrela, com tantos raios quantas as variáveis que descrevem as observações. O tamanho de um raio representa a diferença do valor nessa variável na observação em causa relativamente ao valor médio em todas as observações (depois de normalizados todos os valores). A primeira variável é representada pelo raio na posição horizontal, à direita. As variáveis seguintes seguem o sentido contrário aos ponteiros de relógio. Vejamos um exemplo com o conjunto de dados "mtcars" usando para isso a função `stars()`,

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

Note que este tipo de gráficos é de difícil visualização quando o número de observações é grande, podendo nós, nessas situações, focar-mo-nos em sub-conjuntos mais interessantes dos dados.

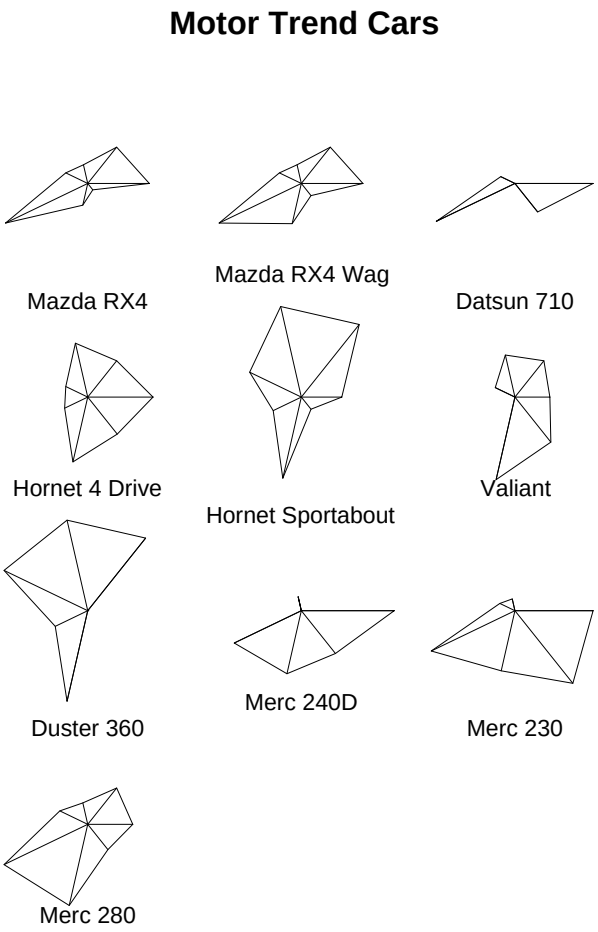


Figura 21: Um gráfico "stars".

Gráficos *mosaic*

Finalmente, vamos ver um tipo de gráficos bastante útil para variáveis nominais que são os gráficos do tipo *mosaic*. Eles usam as áreas relativas de diversos rectângulos para representar as diferenças entre as proporções de casos que possuem determinados valores das variáveis nominais em estudo. Vejamos um exemplo com o conjunto de dados "Titanic" que vem com o R,

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

O resultado desta chamada à função `mosaicplot()` pode ser visto na Figura 22.

4.4.4 Gráficos Condicionados

Gráficos condicionados

Por vezes pretendemos obter certo tipo de gráficos para diferentes sub-grupos dos nossos dados, de tal modo que seja fácil comparar cada sub-grupo. Por exemplo, poderiamos querer obter um gráfico de caixa de bigodes da variável *Petal.Width* para cada espécie de planta (ver Figura 23). Em R isso é fácil obter, uma vez que a própria função `boxplot()` permite explicitar este tipo de condicionamentos usando as fórmulas que estudamos na Secção 4.3

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

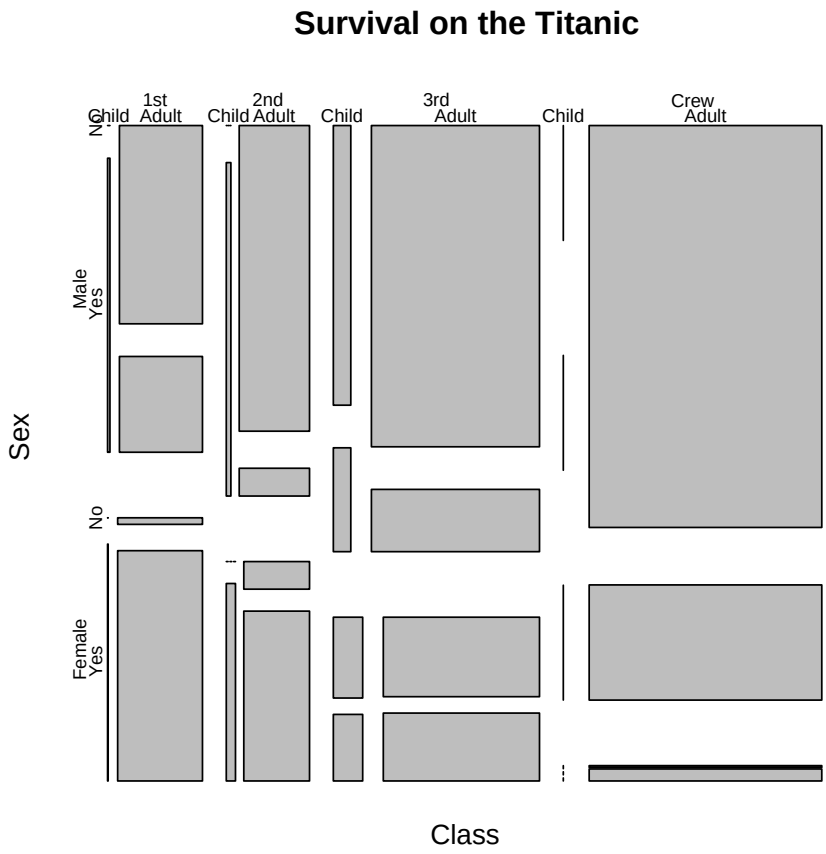


Figura 22: Um gráfico “mosaic”.

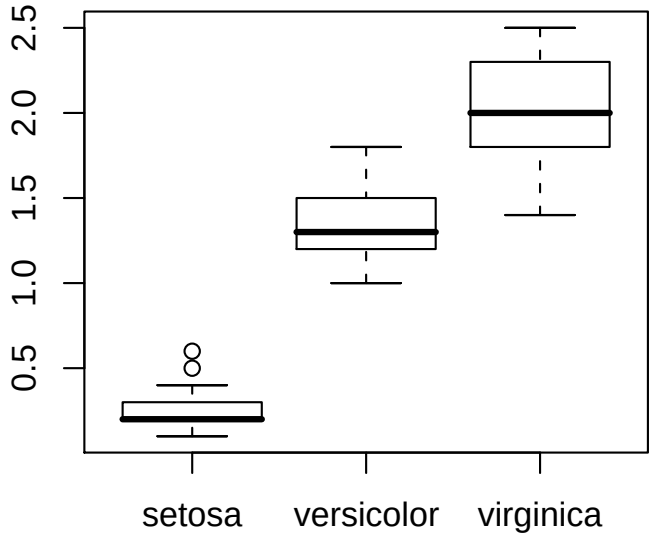
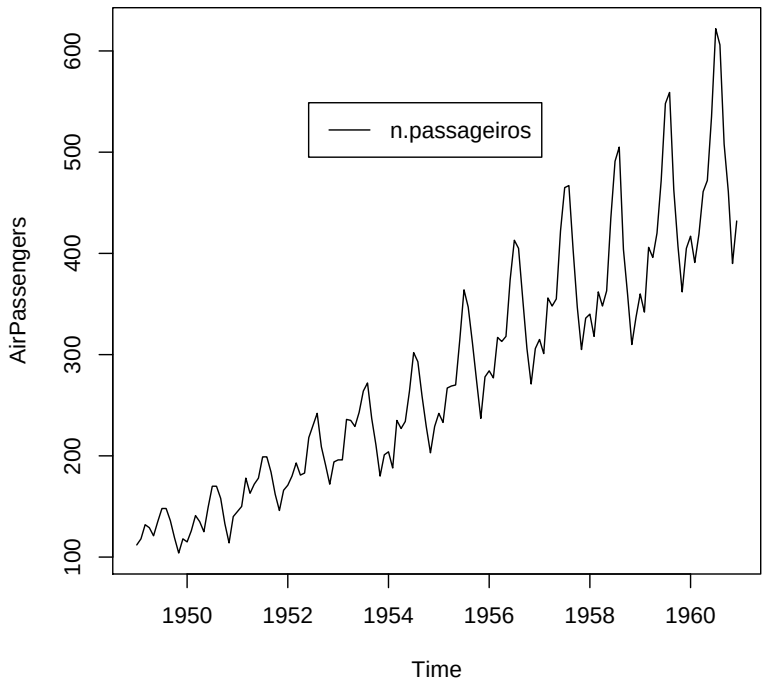


Figura 23: Um gráfico condicionado.

4.4.5 Interacção com Gráficos

Por vezes pretendemos ter alguma forma de interagir com os gráficos desenhados no écran. O R tem algumas funções que podem ser usadas neste contexto. A função `locator()`, por exemplo, pode ser usada para obter as coordenadas dos pontos "cliquados" com o rato numa janela de gráficos. Uma exemplo de uma situação em que isto pode dar jeito é para interactivamente decidir a localização da legenda de um gráfico. Isto pode ser feito da seguinte forma:

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



A função `legend()` permite acrescentar uma legenda numas determinadas coordenadas X-Y do gráfico. Na Secção 4.4.6 iremos ver esta e outras funções mais em detalhe. No exemplo acima, em vez de explicitar as coordenadas X-Y onde queremos colocar a legenda, usamos a função `locator()`. Ao encontrar esta função o R vai ficar em modo interactivo, o que quer dizer que fica à espera que o utilizador clique num qualquer local da janela gráfica. O resultado da função são as coordenadas X-Y do local onde o utilizador cliquar. O argumento da função indica quantos cliques esperamos do utilizador.

Uma outra função interactiva é a função `identify()` que serve para identificar pontos específicos de um gráfico e por exemplo escrever perto os respectivos valores da variável.

Experimente esta função com os seguintes exemplos:

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

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

4.4.6 Adicionar Informação a Gráficos Existentes

O R tem várias funções que podem ser usadas para acrescentar diversos tipos de informação a gráficos já existentes.

Acrescentar novos dados

Por exemplo, a função `points()` pode ser usada para acrescentar novos dados a um gráfico já desenhado:

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

O resultado pode ser observado na Figura 24 onde os pontos a vermelho foram acrescentados à posteriori, através da chamada à função `points()`.

O R tem também algumas funções que podem ser usadas para acrescentar mais alguns elementos a gráficos que já existem. Vejamos um exemplo, ainda com o mesmo histograma (resultado na Figura 26),

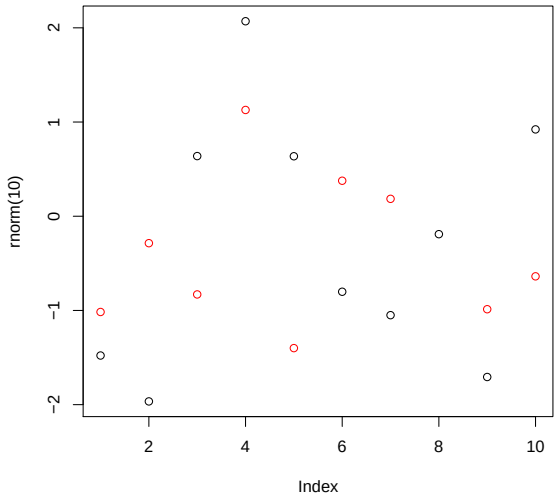


Figura 24: O uso da função "points".

Por vezes os novos dados que pretendemos acrescentar devem ser desenhados como linhas. Para isso a função `lines()` é mais indicada, e o seguinte exemplo ilustra a sua utilização:

Acrescentar linhas

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

O resultado pode ser observado na Figura 25.

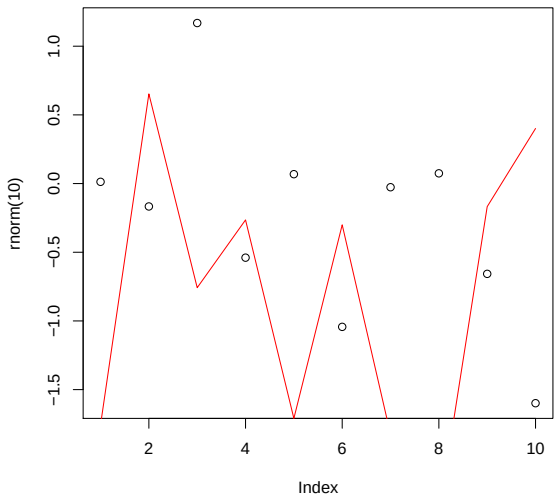


Figura 25: O uso da função "lines".

O exemplo seguinte mostra uma utilização da função `lines()` noutro contexto.

```
> hist(iris$Petal.Width, main = "Histograma de Petal.Width", xlab = "",
+     ylab = "Probabilidade", prob = T)
```

```
> lines(density(iris$Petal.Width))
> rug(jitter(iris$Petal.Width))
```

Neste exemplo concreto, cujo resultado é apresentado na Figura 26, passamos à função como argumento o resultado produzido pela função `density()`. Esta função produz uma estimativa usando uma aproximação tipo "kernel" à densidade da distribuição de uma variável contínua. Finalmente, usamos a função `rug()` para desenhar pequenos traços nos valores concretos da variável, perto do eixo onde está a escala do gráfico. Para evitar demasiadas sobreposições, causadas por valores muito iguais, usamos ainda a função `jitter()` que pega num vector de números e causa-lhes pequenas perturbações aleatórias.

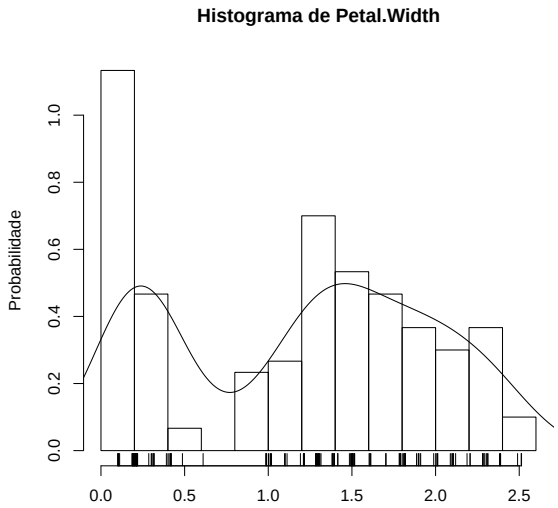


Figura 26: Um histograma com elementos extra.

Também relacionada com o desenho de linhas num gráfico já existente, está a função `abline()`. Esta função permite desenhar linhas horizontais, verticais, ou mesmo linhas inclinadas através da indicação do valor de Y quando X é zero e da inclinação da recta. Vejamos exemplos destas utilizações, cujo resultado pode ser observado na Figura 27,

```
> 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

Em R também é possível acrescentar texto a gráficos existentes. A função `text()`, por exemplo, pode ser usada para acrescentar qualquer texto em determinadas coordenadas de um gráfico. Vejamos um exemplo,

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

Neste exemplo (ver Figura 28), escrevemos o texto "pos" ou "neg" em cima de cada ponto desenhado, dependendo de o seu valor de Y ser positivo ou negativo, respectivamente.

A função `mtext()`, por sua vez, pode ser usada para acrescentar qualquer texto às margens de um gráfico. A margem a usar é determinada pelo parâmetro "side" desta função. Vejamos um exemplo cujo resultado aparece na Figura 29,

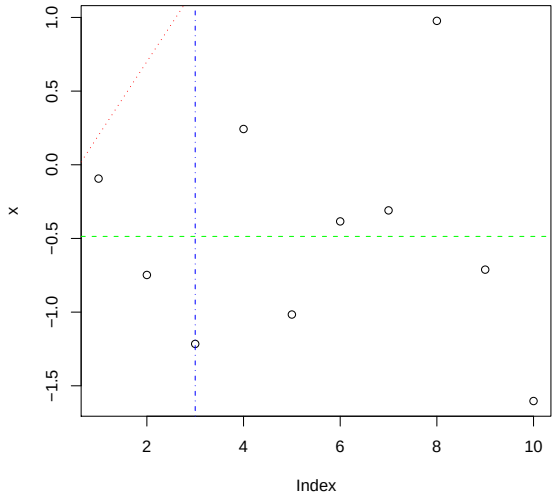


Figura 27: O uso da função “abline”.

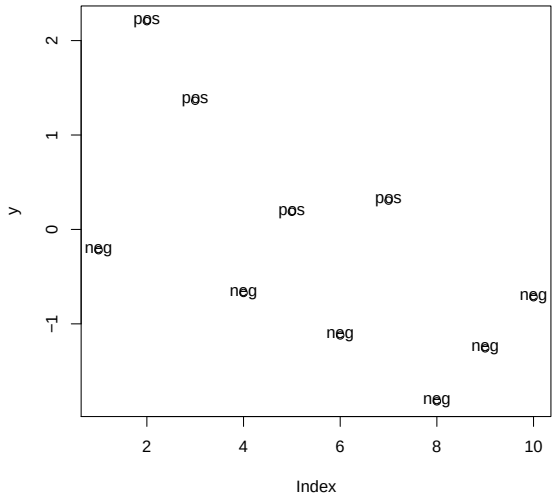


Figura 28: A função “text”.

```
> 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)
```

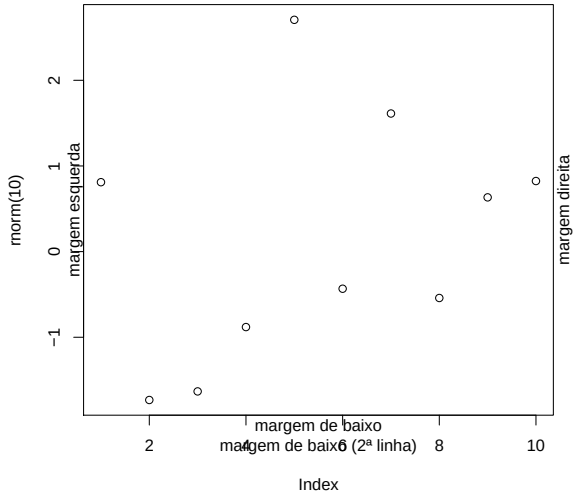


Figura 29: Escrever nas margens.

Acrescentar setas

A função `arrows()`, pode ser usada para acrescentar setas a um gráfico. Vejamos um exemplo cujo resultado aparece na Figura 30,

```
> 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)
```

A função tem como parâmetros principais as coordenadas X-Y dos pontos origem e destino da seta.

Acrescentar títulos e legendas

A função `title()`, permite acrescentar títulos a gráficos, embora a maioria das funções de alto nível tenha um parâmetro próprio (normalmente "main") onde se pode explicitar directamente o título do gráfico em causa. Já a função `legend()` permite-nos acrescentar uma legenda ao gráfico. O código seguinte ilustra o uso destas duas funções, aparecendo o resultado na Figura 31

```
> 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)
```

A função `legend()` pode receber nos dois primeiros argumentos as coordenadas X-Y do canto superior direito onde deve ficar colocada a legenda ou, em alternativa, podemos usar uma "string" que indique a posição relativa da legenda, como vemos no exemplo acima. Veja a ajuda da função para conhecer mais alternativas.

Acrescentar fórmulas

Finalmente, no R é também possível acrescentar fórmulas matemáticas a um gráfico. Para indicar as fórmulas temos que usar uma sintaxe um pouco complexa, cuja explicação sai fora do âmbito deste texto. Pode fazer `demo(plotmath)` no *prompt* do R para ver mais exemplos da sua utilização. Entretanto vejamos um exemplo concreto,

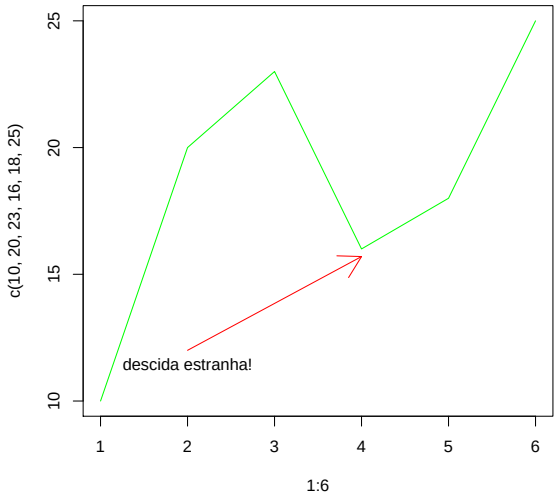


Figura 30: Desenhar setas.

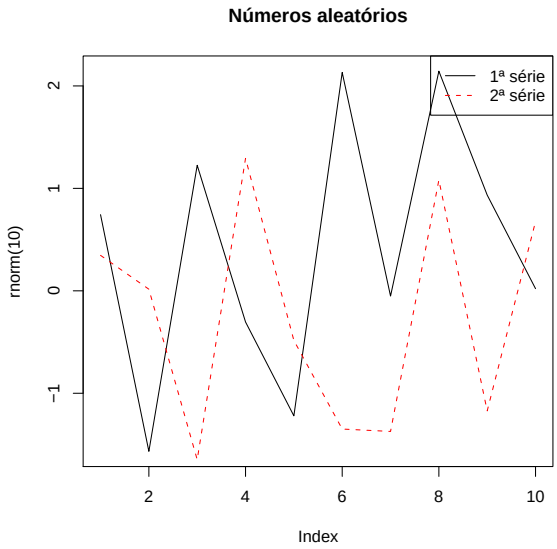


Figura 31: Títulos e legendas.

```
> 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))))
```

O resultado pode ser observado na Figura 32

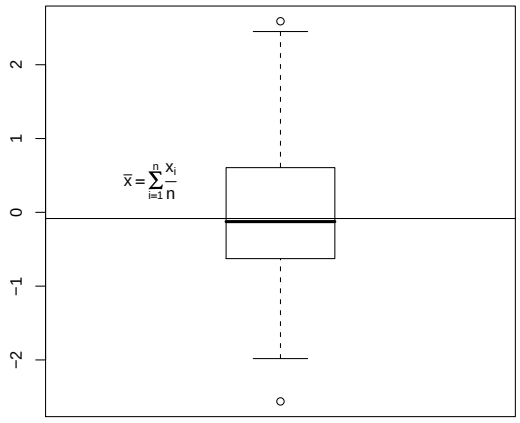


Figura 32: Fórmulas matemáticas.

4.4.7 Parâmetros de Gráficos

A estrutura genérica de um gráfico em R pode ser descrita pela Figura 33. Assim, temos uma área com o gráfico propriamente dito, e uma região circundante que forma as margens do gráfico. Ao conjunto chama-se normalmente uma figura.

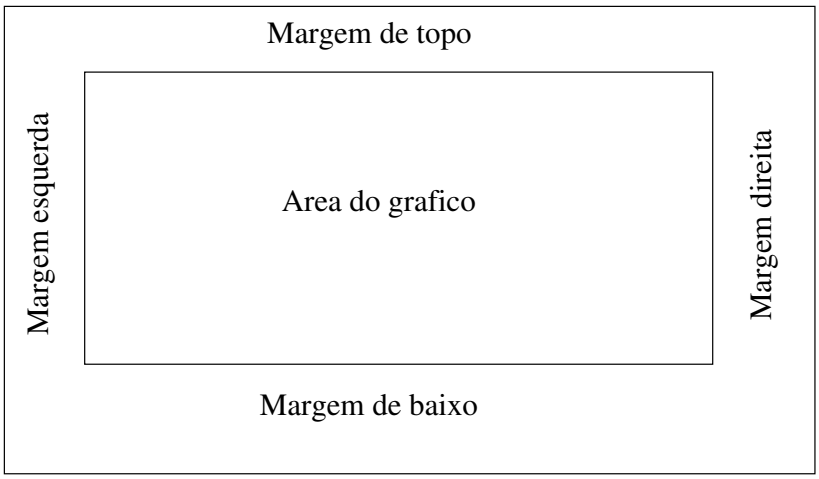


Figura 33: A estrutura genérica dos gráficos em R.

O espaço ocupado por cada um destes elementos que formam uma figura é totalmente configurável pela função `par()`. Esta função permite fazer o *setting* "permanente" de vários parâmetros gráficos para o *device* activo actual. Descrever os inúmeros parâmetros

que podem ser modificados com esta função sai fora do âmbito deste texto. A ajuda da função fornece uma descrição exaustiva dos mesmos.

Muitos dos parâmetros que podem ser modificados pela função `par()`, também o podem ser através de parâmetros da maioria das funções de alto nível para produzir gráficos. Todavia, usar esta alternativa faz com que um determinado valor para o parâmetro só seja usado no gráfico em causa e não em todos os gráficos produzidos no *device* activo, como acontece quando usamos a função `par()`.

Vejamos alguns dos parâmetros mais comuns e que aparecem em quase todas as funções de alto nível para produzir gráficos:

- `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.

4.4.8 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 das formas mais comuns:

- Gráficos de tamanho igual.

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

  O parâmetro "mfrow" permite explicitar uma divisão igual da janela de gráficos em x linhas e y colunas. Assim, ao colocarmos nesse parâmetro o valor `c(2,1)` por exemplo, iriamos dividir o *device* em duas linhas com uma coluna de gráficos, ou seja dois gráficos.

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

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

  A função `layout()` recebe uma matriz como parâmetro principal. As dimensões da matriz definem a forma como *device* é "dividido". Assim, no exemplo anterior ele vai ser dividido em duas linhas e duas colunas. Os valores na matriz determinam a ordem em que são usadas cada uma das áreas quando se produzem gráficos. A função `layout.show()` pode ser usada para exemplificar visualmente esta informação. O argumento desta função é o número de gráficos que "cabem" no *device*.

  O próximo exemplo é um pouco mais sofisticado. Ele usa a seguinte matriz como argumento,

  ```
  > matrix(1:2, 2, 2)
  ```

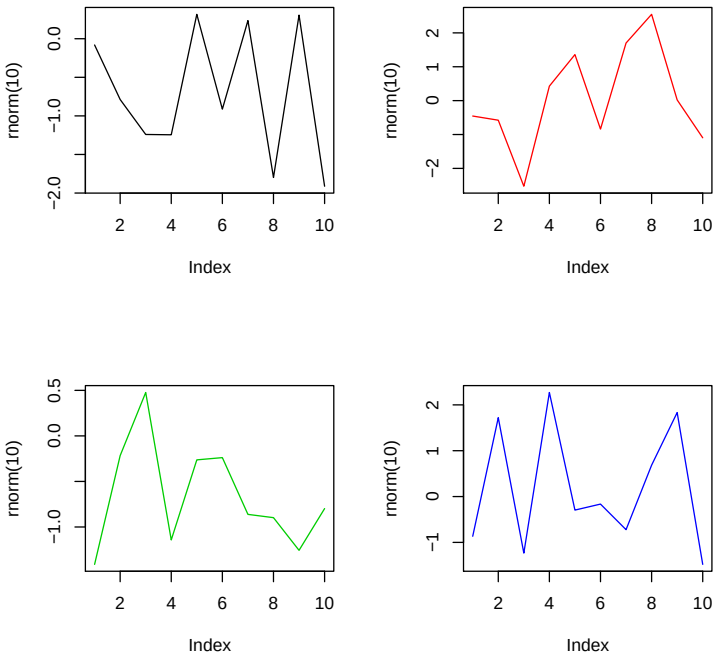


Figura 34: Vários gráficos de tamanho igual.

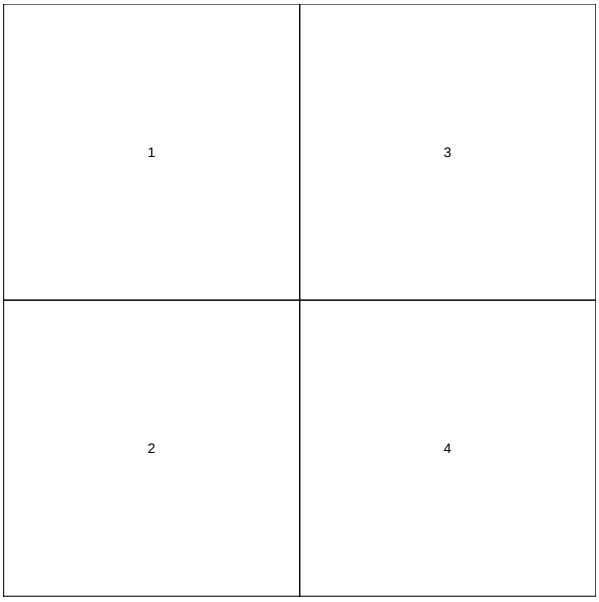


Figura 35: O *layout* criado.

```
     [,1] [,2]
[1,]    1    1
[2,]    2    2
```

O significado desta matriz é que o primeiro gráfico vai ocupar as duas primeiras áreas da matriz (toda a primeira linha), enquanto o segundo ocupa toda a segunda linha. Isto quer dizer que na realidade só vão "caber" 2 gráficos no nosso *device*.

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

O argumento "heights" serve para explicitar diferentes tamanhos entre as linhas da matriz que define as áreas em que é dividido o *device*. Neste caso, e uma vez que a matriz tem duas linhas, os dois números indicam que os gráficos na primeira linha vão ocupar 2/3 da área do *device*, enquanto que a segunda linha ocupa 1/3. O resultado desta divisão pode ser observado na Figura 36. É também possível explicitar tamanhos absolutos usando a função `lcm()` conforme poderá confirmar na ajuda da função `layout()`. É também possível explicitar os tamanhos usando o parâmetro "widths", ou ambos.

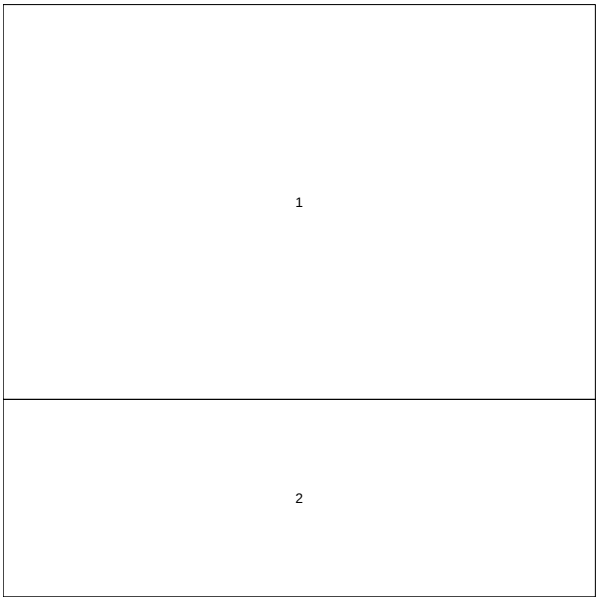


Figura 36: Um *layout* com diferentes tamanhos.

O nosso exemplo final usa a matriz,

```
> rbind(c(1, 2), c(3, 3))
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    3
```

como argumento, o que quer dizer que vamos ter 3 gráficos, sendo que o terceiro ocupa toda a linha de baixo da matriz de áreas em que é dividido o *device*.

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

O resultado deste *layout* pode ser observado na Figura 37.

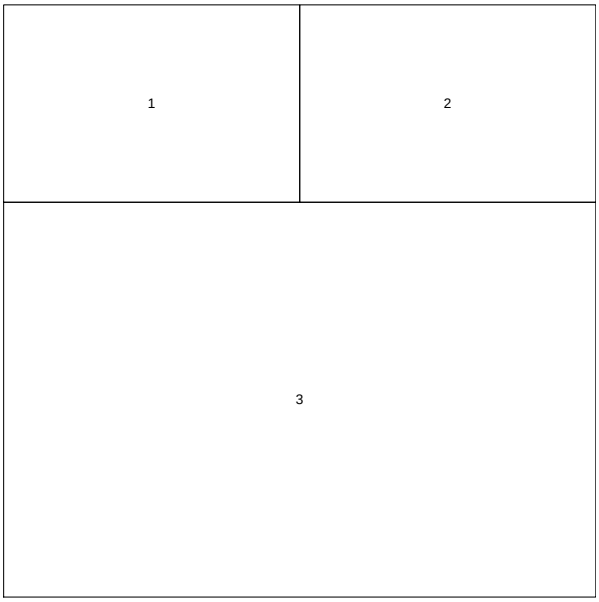


Figura 37: Mais um *layout* com diferentes tamanhos.

Referências

Chambers, J. (1998). *Programming with Data*. Springer.

Dalgaard, P. (2002). *Introductory Statistics with R*. Springer.

Murrell, P. (2006). *R Graphics*. Chapman & Hall/CRC.

R Development Core Team (2006). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Venables, W. and Ripley, B. (2000). *S Programming*. Springer.

Índice

álgebra matricial, 25, 57
  determinante, 25
  inversa, 25
  multiplicação, 25
  transposta, 25

ambientes
  workspace, 53
arrays, 26
atribuição, 11, 14

blocos de instruções, 46

coerção de tipos, 13

data frames, 29
  aceder às colunas, 29
  aceder aos elementos, 29
  acrescentar colunas, 30
  criar, 29
  número de colunas, 30
  número de linhas, 30
  preencher, 30
datas/horas, 37
  discretizar, 38
  extraír, 39
  mostrar, 37, 39
  objectos Date, 37
  objectos POSIXt, 38
  sequências, 37
debugging, 60
  debug, 62
  traceback, 60
  undebug, 62

expressões numéricas, 11

factores, 15, 29
  criar, 15
  frequências marginais, 17
  frequências relativas, 17
  níveis, 15
    tabelar, 16
  tabulação cruzada, 16
Funções
  argumentos, 54
    actuais, 55, 56
    formais, 55
    lazy evaluation, 56
    número variável, 55
    parâmetro ..., 55
    valores por defeito, 54
  criar, 52
  valor, 52

funções, 56
  abline, 86
  apply, 50, 69
  apropos, 9
  argumentos, 19
  array, 26
  arrows, 88
  as.Date, 38
  as.POSIXlt, 39
  attach, 30
  barplot, 73
  boxplot, 73, 81
  break, 48
  browser, 60
  by, 67
  c, 13, 28
  cat, 45
  cbind, 23, 35, 42
  class, 59
  colnames, 23
  contour, 75
  coredata, 41
  cut, 38
  data, 63
  data.frame, 29
  dbQuery, 66
  debug, 62
  density, 86
  det, 25
  detach, 30
  dev.cur, 72
  dev.list, 72
  dev.set, 72
  diff, 35, 43
  dim, 22
  download.file, 64
  edit, 30
  embed, 35
  end, 34, 41
  factor, 15
  for, 49
  format, 37
  getwd, 8
  gl, 19
  head, 68
  help, 9
  help.search, 9
  hist, 73
  identify, 84
  if, 46
  ifelse, 47
  image, 75
  install.packages, 10

ISOdate, 38
jitter, 86
jpeg, 72
lag, 35, 43
lapply, 51
layout, 91
layout.show, 91
legend, 84, 88
length, 13, 28
library, 10
lines, 85
locator, 83
ls, 12
margin.table, 17
matplot, 78
matrix, 22
mean, 54
median, 47
merge, 42
methods, 59
min, 50
months, 37
mosaicplot, 81
mtext, 86
na.approx, 43
na.contiguous, 43
na.locf, 43
na.omit, 36, 43
names, 21, 27
ncol, 30, 67
next, 49
nrow, 30, 67
objects, 12
outer, 75
pairs, 78
par, 90
pdf, 72
persp, 74
plot, 32
points, 84
print, 45
prop.table, 17
q, 8
quartile, 51
rapply, 44
rbind, 23
read.csv, 63
read.csv2, 63
read.delim, 63
read.delim2, 63
read.fwf, 64
read.table, 63
read.xls, 65
rep, 19
repeat, 48
return, 52
rm, 12
rnorm, 19, 48
rollmean, 44
rownames, 23
RSiteSearch, 9
rt, 19
rug, 86
sapply, 51
scan, 45, 64
seq, 18, 37, 55
setwd, 8
solve, 25
sqrt, 14
stars, 80
start, 34, 41
str, 45, 68
strptime, 39
subset, 69
summary, 67
switch, 47
Sys.Date, 37
Sys.time, 39, 50
t, 25
table, 16
tail, 68
tapply, 51
text, 86
title, 88
traceback, 60
ts, 31
undebug, 62
vector, 14
weekdays, 37
while, 48
window, 34, 42
windows, 72
zoo, 37

indexação, 13, 20
  índices negativos, 21
  arrays, 26
  data frames, 29
  matrizes, 23
  vectores, 20
    inteiros, 21
    lógicos, 20
    strings, 21
    vazios, 22
instruções iterativas, 48

listas, 27
  componentes, 27
    extraír, 27
  contar componentes, 28
  extender, 28
  juntar, 28
  mostrar, 27

matrizes, 22
  criar, 22
  indexar, *ver* matrizes,indexação
  nomes dimensões, 23

NA, 13

objectos
  atribuição, *ver* atribuição
  definição, 11
  listar, 12
  nomes, 12
  remover, 12
  ver conteúdo, 11
operadores lógicos, 20

packages, 9
  instalar, 10
  usar, 10

R
  ajuda, 9
  executar, 7
  executar comandos, 7
  executar comandos
    várias linhas, 17
  gravar estado, 8
  instalação, 7
  janela de aplicação, 7
  packages, *ver* packages
  prompt, 7
  site, 7
  terminar, 8
reciclagem, 15, 18, 20, 24

séries temporais
  diferenças, 35, 43
  embed, 35
  fim, 34, 41
  funções deslizantes, 43
  gráficos, 32
  início, 34, 41
  irregulares, 36
    criar, 36
    funções úteis, 39
    packages, 36
  janela temporal, 34, 42
  juntar, 35, 42
  lagging, 35
  regulares, 31
    criar, 31
    funções úteis, 33
  valores desconhecidos, 43
scoping, 53
  lexical scoping, 53
sequências, 18
  aleatórias, 19
  descendentes, 18
  factores, 19
  números reais, 18
  operador :, 18, 21
  repetidas, 19
sistemas de equações, 26
strings, 13

variáveis nominais, *ver* factores
vectores, 12
  alterar elemento, 14
  alterar tamanho, 14
  criar, 13
  dar nomes aos elementos, 21
  modo, 12
  operações aritméticas, 14
  reciclagem, *ver* reciclagem
  tamanho, 12
  vazios, 14
vectorização, 14, 25