# Introdução a R

Notas sobre R: Um ambiente de programação para Análise de Dados e Gráficos

Versão 1.1.0 (11 Julho 2000)

**R Development Core Team**

# Índice Geral

# Prefácio

Esta introdução ao R vem na sequência de um conjunto inicial de notas descrevendo os ambientes S e S-Plus escritas por Bill Venables e Dave Smith. Nós fizemos um conjunto de pequenas alterações para evidenciar as diferenças entre os programas R e S, e desenvolvemos alguns temas.

R é um projecto em evolução e as suas capacidades actuais não coincidem com as do ambiente S. Nestas notas adoptámos a convenção de que qualquer característica que se vá a implementar é especificada como tal no início da secção onde tal melhoria é descrita. Os utilizadores podem contribuir para o projecto implementando tais evoluções ainda não desenvolvidas.

Gostaríamos de apresentar um forte agradecimento a Bill Venables por permitir distribuir esta versão modificada das suas notas originais, e por ser um defensor do projecto R desde o seu início.

Comentários e correcções são sempre bem-vindos. Por favor enviar a correspondência para o seguinte endereço de e-mail: R-core@r-project.org.

## Sugestões ao leitor

A primeira relação com R deveria começar com a sessão introdutória no Apêndice A. Foi escrita de modo a que o leitor ganhe alguma familiaridade com o estilo das sessões R e mais importante, ganhará imediatamente algum feedback sobre o que acontece.

Muitos utilizadores elegem R pelas suas capacidades gráficas. Neste caso, o Capítulo 12 [Rotinas gráficas], pág. 79 na sessão sobre as capacidades gráficas pode ser lido em qualquer momento sem necessidade de esperar pelo estudo de todas as secções precedentes.

# 1 Introdução e preliminares

## 1.1 O ambiente R

R é um conjunto integrado de programas para manipulação de dados, cálculo e gráficos. Entre outras características permite:

- manipulação e armazenamento efectivo dos dados,
- operadores para cálculo sobre variáveis indexadas e cálculo matricial,
- uma vasta, coerente e integrada colecção de ferramentas para análise de dados,
- capacidades gráficas para análise exploratória de dados, que permitem a visualização directamente no écran ou obter cópias impressas,
- uma linguagem de programação bem desenvolvida, simples e eficiente, que inclui estruturas condicionais, estruturas cíclicas, funções recursivas, e capacidades de entrada e saída de dados. (Refira-se que muitas das funções oferecidas foram desenvolvidas na própria linguagem R).

O termo "ambiente" caracteriza R como um sistema completamente planeado e coerente, e não apenas como um conjunto ampliado de ferramentas muito específicas e inflexíveis, como é frequentemente o caso de outros programas de análise de dados.

R é em grande parte um veículo para o desenvolvimento de novos métodos interactivos de análise de dados. Como tal, é muito dinâmico e as diferentes versões nem sempre são completamente compatíveis com as anteriores. Se alguns utilizadores preferem as alterações pelos novos métodos e tecnologias que acompanham as novas versões, outros pelo contrário, ficam desiludidos porque os seus códigos-fonte deixaram de funcionar. Embora R possa ser entendido como tratando-se de uma linguagem de programação, os programas escritos em R devem considerar-se essencialmente efémeros.

## 1.2 Programas relacionados. Documentação

R pode definir-se como uma nova implementação da linguagem S desenvolvida por Rick Becker, John Chambers e Allan Wilks nos Laboratórios AT&T. Muitos dos manuais e livros sobre S são úteis para R.

A referência base é o livro *The New S Language: A Programming Environment for Data Analysis and Graphics*, de Richard A. Becker, John M. Chambers e Allan R. Wilks. As novas características da versão S de Agosto de 1991 (S versão 3) são descritas no livro Satistical Models in S editado por John M. Chambers e Trevor J. Hastie. Veja-se o Apêndice F [Referências], pág. 110, sobre a lista de referências.

## 1.3 Estatística com R

Na introdução ao R não se mencionou a palavra *estatística*, muito embora muitas pessoas utilizem R como um sistema estatístico. Nós preferimos descrevê-lo como um ambiente sobre o

qual se implementaram muitas metodologias estatísticas, tanto clássicas como modernas. Muitas destas fazem parte do ambiente base de R, e outras acompanham R sob a forma de *bibliotecas* ou '*packages*' (a distinção entre ambos os conceitos á fundamentalmente uma questão histórica). Conjuntamente com R são incluídas oito bibliotecas (designadas por bibliotecas standard), embora muitas outras estejam disponíveis no site de CRAN (http://cran.r-roject.org).

Tal como referido, muitas das metodologias estatísticas, quer clássicas quer modernas, estão disponíveis em R, embora os utilizadores necessitem de estar dispostos a trabalhar um pouco para encontrá-las.

Existe uma diferença fundamental entre a filosofia subjacente a R (e S) e os restantes sistemas estatísticos. Em R, uma análise estatística é realizada numa série de passos, em que os resultados intermédios vão sendo armazenados sob a forma de objectos, que por sua vez serão a entrada para análises subsequentes, obtendo-se no final um conjunto minimizado de resultados, enquanto que em outros sistemas estatísticos, tais como SAS ou SPSS, se obtém de imediato uma extensa lista de output para qualquer análise, por exemplo uma regressão linear ou análise discriminante.

## 1.4 R em ambiente de janelas

A forma mais prática e conveniente de usar R é numa estação de trabalho em ambiente de janelas. Estas notas estão escritas pensando que os utilizadores dispõem de tais características. Ocasionalmente referiremos em particular à utilização de R num ambiente X-windows, embora na sua maior parte as notas se possam aplicar genericamente a qualquer implementação do ambiente R.

Muitos utilizadores necessitam de, ocasionalmente, inter-actuar directamente com o sistema operativo. Nestas notas é considerada em particular a interacção com o sistema operativo UNIX. Se utiliza R em ambiente Windows é provável que necessite de realizar alguns pequenos ajustes.

A instalação do sistema operativo e do programa de modo a obter um máximo rendimento das capacidades parametrizadas de R é uma tarefa interessante, embora muito fastidiosa, e está fora do âmbito destas notas. Si tiver dificuldades de instalação, procure o especialista der sistemas da sua área.

## 1.5 Uso interactivo de R

Quando R espera a entrada de ordens, apresenta um símbolo indicativo de que aguarda a entrada. O símbolo predeterminado é "$>$", que em UNIX pode coincidir com o indicativo ou prompt do sistema, pelo que pode inicialmente propiciar alguma confusão e parecer que nada está a acontecer. Se for este o seu caso, é possível modificar o indicativo para um que lhe seja mais sugestivo. Nestas notas assume-se que o prompt do sistema operativo UNIX é "$".

Para utilizar R pela primeira vez, em ambiente UNIX, o procedimento recomendado é o seguinte:

1. Crie um subdirectório, por exemplo com o nome "*trabalho*", para guardar os arquivos de dados que vai a utilizar com R. Este será o directório de trabalho cada vez que utilize R para este problema concreto.

   ```
   $ mkdir trabalho
   $ cd trabalho
   ```

2. Inicie R com a ordem ou comando

   ```
   $ R
   ```

3. A partir deste momento está em condições de usar os comandos de R (como se verá em seguida)

4. Para sair de R o comando é

   ```
   > q()
   ```

   O programa perguntará se pretende guardar os dados desta sessão. Pode responder *yes* (sim), *no* (não) ou *cancel* (cancelar) primindo as teclas **y**, **n** ou **c**, de modo a que guarde os dados, não guarde os dados antes de sair ou voltar ao ambiente R, respectivamente. Se optar por guardar os dados, estes estarão disponíveis para a sessão seguinte.

As sessões seguintes são mais fáceis:

1. Mude para o directório de trabalho:

   ```
   $ cd trabalho
   $ R
   ```

2. Use os comandos pretendidos de R, e termine a sessão com `q()`, guardando ou não os dados, conforme pretender.

O procedimento para usar R em ambiente Windows é basicamente o mesmo. Crie uma pasta ou directório para directório de trabalho (por exemplo *c:\programas\R\trabalho*) e defina este directório no campo "*Iniciar em*" do atalho para R no *Ambiente de Trabalho*. Para iniciar R, baste fazer duplo clique no ícone.

## 1.6 Uma sessão de introdução

Recomenda-se aos utilizadores principiantes que desejem fazer uma abordagem prévia ao estilo de funcionamento de R, que realizem a sessão de introdução apresentada no Apêndice A [Um exemplo de sessão], pág. 95.

## 1.7 Ajuda sobre funções e capacidades

R dispõe de uma rotina de ajuda similar ao comando ***man*** do UNIX. Para obter informação sobre uma função concreta, por exemplo ***solve***, o comando é:

```
> help(solve)
```

ou, alternativamente:

```
> ?solve
```

Com as funções e capacidades especificadas por caracteres especiais, o argumento deverá ser escrito entre aspas, formando uma "*cadeia de caracteres*":

```
>help("[[")
```

Tanto se podem usar aspas ("*texto*") como apóstrofos ('*texto*'). Por uniformização, nestas notas usar-se-ão aspas. Se houver necessidade de utilizar aspas dentre de uma frase, recomenda-se que se use aspas no exterior e apóstrofos no interior da frase, como no exemplo:

```
> print("Disse 'bom dia' e foi-se")
```

Em muitas versões de R está disponível ajuda em formato HTML, executando o comando:

```
> help.start()
```

que iniciará um browser Web (*netscape* em UNIX) que permite a leitura de páginas com hipertexto. Em UNIX, as ordens de ajuda posteriores serão enviadas para o sistema de ajuda em formato HTML.

As versões R em Windows dispõem de outros sistemas opcionais de ajuda. Utilize:

```
> ?help
```

para obter informações adicionais.

## 1.8 Comandos de R. Maiúsculas e minúsculas

Tecnicamente, R é uma *linguagem de expressões* com regras de sintaxe muito simples. Faz a distinção entre maiúsculas e minúsculas, como todos os sistemas desenvolvidos em UNIX, de modo que os caracteres A e a são entendidos como sendo símbolos diferentes, referindo-se portanto a variáveis diferentes.

Os comandos ou ordens elementares consistem de *expressões* ou de *assignações*. Se uma ordem ou comando é uma expressão, o seu valor é calculado e visualizado, perdendo-se de seguida. Uma assignação ou atribuição pelo contrário, calcula a expressão e atribui ou assigna o resultado (que não é mostrado automaticamente) a uma variável.

Os comandos são separados por ponto e vírgula (;), ou são entrados em nova linha. Podem agrupar-se, dentro de chavetas ('{' ... '}'), vários comandos elementares numa expressão mais complexa. Podem inserir-se comentários, em qualquer comando[1], começando com o caracter cardinal ('#'). Se ao terminar uma linha, o comando não está sintacticamente completo, R mostra o símbolo de continuação de comando, que por defeito é o símbolo:

```
+
```

na linha seguinte e nas sucessivas e continua a ler até que ordem esteja sintacticamente completa. Este símbolo de continuação pode ser alterado. Por convenção nestas notas será omitido o símbolo e a continuação do comando é indicada pelo avanço da linha.

## 1.9 Recuperação e correcção de comandos anteriores

Em ambiente Windows e em muitas versões sob UNIX, R permite recuperar e executar os comandos anteriores. As setas verticais do teclado podem usar-se para percorrer o *histórico dos*

[1] Os comentários não podem inserir-se dentro de cadeias de caracteres, nem no interior da lista de argumentos de uma função.

*comandos* executados. Quando se tiver recuperado o comando pretendido, podem usar-se as setas horizontais para deslocar o cursor ao longo da linha de comando, podem eliminar-se caracteres com a tecla <DEL>, ou adicionar mais caracteres. No Apêndice C [O editor de comandos], pág. 105, serão dados mais pormenores.

A recuperação de comandos e as capacidades de edição em ambiente UNIX são facilmente configuráveis. Pode obter mais informação sobre este assunto consultando o manual de UNIX sobre ***readline*** (`$ man readline`)

Também pode utilizar o editor de texto ***emacs*** (via "ESS", *Emacs Speaks Statistics*), para trabalhar mais comodamente de modo interactivo com R. Veja a secção "*R and Emacs*" em "*The R statistical sistem FAQ*".

## 1.10 Execução de um ficheiro de comandos e re-direccionamento da saída

Se os comandos estão guardados num ficheiro externo, por exemplo '*commands.R*' localizado no directório de trabalho, podem ser executados numa sessão de R com o comando

```
> source("commands.R")
```

Em ambiente Windows, o comando ***source*** está disponível dentro do menu **File**, opção **Source R code**.

A função ***sink***

```
> sink("record.list")
```

redirecciona todas as saídas da consola subsequentes para o arquivo externo '*record.list*'. O comando:

```
> sink()
```

redirecciona novamente a saída novamente para a consola.

## 1.11 Guardar e eliminar de objectos

As entidades criadas e manuseadas por R designam-se por *objectos*. Estes podem ser variáveis, vectores ou matrizes de números, cadeias de caracteres, funções, ou mais genericamente estruturas mais complexas construídas a partir de destes elementos mais simples.

Durante uma sessão de R, os objectos são criados e guardados por nomes (este assunto será discutido mais em pormenor na próxima sessão). O comando:

```
> objects()
```

dá a lista dos nomes dos objectos presentemente guardados por R. Para o mesmo efeito também se pode usar o comando:

```
> ls()
```

O conjunto de objectos actualmente guardados por R designa-se por *espaço de trabalho* ('*workspace*').

Para eliminar objectos usa-se o comando ***rm***, como por exemplo:

```
> rm(x, y, z, tinta, chaparro, temporal, barra)
```

que elimina os objectos designados pelos nomes x, y, z, tinta, chaparro, temporal, barra.

Todos os objectos criados numa sessão de trabalho em R podem ser definitivamente guardados num arquivo, a fim de serem usados em sessões futuras. No final de cada sessão, ao dar a ordem de encerrar (comando `q()`), é dada a oportunidade para guardar todos os objectos actualmente disponíveis. Caso o utilizador opte por guardar a sessão, os objectos são guardados num ficheiro com o nome *'.Rdata'*[2] no directório corrente.

Quando R é iniciado posteriormente, o espaço de trabalho guardado é recuperado, tornando disponíveis os objectos e o historial de comandos guardados nesse ficheiro.

É recomendável que se utilizem directórios de trabalho distintos para as diversas análises efectuadas em R. É frequente que criar objectos com os nomes genéricos tais como x, y, z, etc., durante uma sessão. Estes nomes podem ser sugestivos durante a sessão onde são criados, mas será extremamente difícil associar nomes deste tipo a objectos quando se realizem várias análises no mesmo directório.

[2] Se o nome de arquivo começa por ponto, este fica invisível para a listagem normal em UNIX.

# 2 Cálculos simples. Números e vectores

## 2.1 Vectores numéricos. Assignação

R utiliza diferentes *estruturas de dados*. A estrutura mais simples é o *vector numérico* (na presente sessão iremos utilizar a expressão vector como referindo-se a vector numérico), que é um conjunto ordenado de números. Para criar um vector, por exemplo com o nome x, constituído por cinco números, por exemplo 10.4, 5.6, 3.1, 6.4 e 21.7, usa-se o comando:

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

Esta ordem é uma assignação ou atribuição, em que se utiliza a função ***c()*** que, neste contexto, pode ter um número arbitrário de vectores como argumento, e cujo resultado é o vector constituído pela concatenação ou junção sequencial de todos eles[1].

Um número, por si mesmo, é considerado um vector de comprimento unitário.

Note-se que o operador de assignação ou atribuição à esquerda (‘**<-**‘) não é o operador igualdade (‘=’) usual, que se reserva para outro propósito. É constituído por dois caracteres ‘menor que’ (‘<’) e ‘menos’ (‘-‘) que obrigatoriamente devem ir unidos e ‘apontam’ para o objecto que recebe o valor da expressão[2].

A operação de assignação também pode ser efectuada com a função ***assign()***. Uma maneira equivalente de fazer a assignação anterior é:

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

O operador assignação usual (‘<-‘) pode ser entendido como um atalho ou abreviatura da função assign().

A assignação também se pode fazer ‘à direita’, trocando obviamente o símbolo de assignação pelo símbolo ‘maior que’ que aponta a direcção à direita(isto é, ‘->’). A mesma operação de assignação pode assim ser feita do seguinte modo:

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

Se uma expressão é usada como comando tal qual, sem assignação, o seu valor é calculado, visualizado no écran, e perde-se[3]. Assim, o comando:

```
> 1/x
```

simplesmente calcula os inversos dos cinco valores anteriores (atribuídos ao vector x), e imprime o resultado no écran (e o valor de x, obviamente, não é alterado).

---

[1] Com argumentos de outro tipo, por exemplo list, a acção da função ***c()*** pode ser diferente. Veja-se a Secção 6.2.1 [Concatenação de listas], pág. 34.

[2] O caracter de sublinhado (‘_’) é um sinónimo do operador assignação à esquerda (‘<-‘), mas não se aconselha a sua utilização pois resulta num código menos legível.

[3] O valor é guardado na variável ***.Last.value*** que o guarda até que seja executado outra ordem.

A assignação:

```
> y <- c(x, 0, x)
```

cria o vector y com 11 elementos, constituídos por duas cópias de x com o valor 0 (zero) entre ambas, isto é, 10.4, 5.6, 3.1, 6.4, 21.7, 0, 10.4, 5.6, 3.1, 6.4, 21.7.

## 2.2 Aritmética sobre vectores

Os vectores podem usar-se em expressões aritméticas, caso em que as operações se realizam elemento a elemento. Dois vectores que se utilizem na mesma expressão não tem que, obrigatoriamente, ser do mesmo comprimento. Se o não são, o resultado é um vector com o comprimento do vector mais longo, e o mais curto é utilizado ciclicamente, repetindo-se tantas vezes quantas as necessárias (pode acontecer que se repita um número não inteiro de vezes), até que coincida com o comprimento do mais longo. Em particular, uma constante ou um vector unitário será simplesmente repetida tantas vezes quanto o comprimento do vector mais longo. Deste modo, e sendo x e y os vectores atrás definidos, a ordem:

```
> v <- 2*x + y + 1
```

cria um novo vector, v, de 11 elementos, em que cada um deles é o resultado da soma do dobro de cada elemento do vector x, repetido 2.2 vezes (as necessárias para igualar o comprimento de y) , com o elemento respectivo do vector y, repetido uma só vez, e com o valor 1, repetido 11 vezes, isto é, os elementos de v são: 32.2, 17.8, 10.3, 20.2, 66.1, 21.8, 22.6, 12.8, 16.9, 50.8, 43.5.

Os operadores aritméticos elementares são os habituais `+`, `-`, `*`, `/` e `^` para potências. Também estão disponíveis as funções aritméticas comuns: ***log***, ***exp***, ***sin***, ***cos***, ***tan***, ***sqrt*** têm o seu significado usual. As funções ***min*** e ***max*** obtêm os valores mínimo e máximo de um vector, respectivamente. A função ***range*** dá como resultado um vector de comprimento 2, e cujos elementos são c(min(x), max(x)); ***length***(x) dá o número de elementos ou comprimento do vector x; ***sum***(x) calcula a soma de todos os valores contidos no vector x, e ***prod***(x) o respectivo produto.

Duas funções estatísticas são ***mean***(x), que calcula a média, isto é:

```
> sum(x)/length(x)
```

e ***var***(x) que calcula a variância da amostra, isto é:

```
> sum((x-mean(x))^2)/(length(x)-1)
```

Se o argumento de var() é uma matriz $n \times p$, o resultado é a matriz $p \times p$ de variância-covariância correspondente a interpretar as linhas como p vectores amostrais independentes.

A função ***sort***(x) origina um vector da mesma dimensão de x, em que os elementos estão ordenados por ordem crescente. Para o mesmo efeito também se dispõe das funções ***order***() e ***sort.list***(), mais flexíveis, que produzem a permutação de x correspondente à ordenação.

Note-se que max() e min() seleccionam os valores máximo e mínimo nos argumentos especificados, mesmo tratando-se de vários vectores. As funções paralelas para máximo e mínimo, respectivamente ***pmax***() e ***pmin***), dão como resultado um vector (de comprimento igual ao de maior dimensão especificado em argumento), em que cada elemento é o elemento

máximo (ou mínimo) na posição respectiva em qualquer dos vectores especificados em argumento.

Na maioria dos casos o utilizador não deve preocupar-se se os 'números' num vector numérico são valores inteiros, reais ou mesmo complexos. Internamente os cálculos são executados como números de dupla precisão, reais ou imaginários, consoante os dados entrados.

Para trabalhar com números complexos, deve-se indicar explicitamente a parte complexa. Assim:

```
> sqrt(-17)
```

dará como resultado ***NaN*** ('*Not a Number*') e uma mensagem de advertência; mas:

```
> sqrt(-17+0i)
```

realiza correctamente o cálculo da raiz quadrada deste número complexo.

## 2.3 Gerar sequências regulares

Em R existem várias funções para gerar sucessões ou sequências numéricas. Por exemplo, `1:30` é o vector c(1, 2, ... , 29, 30). O operador '`:`' (dois pontos) tem prioridade máxima numa expressão onde seja usado; assim ,por exemplo, 2*1:15 é o vector c(2, 4, ... , 28, 30). Faça o comando `n <- 10` e compare as sequências `1:n-1` e `1:(n-1)`.

A expressão `30:1` pode usar-se para construir a sequência decrescente.

A função ***seq***() permite gerar sequências mais complexas. Dispõe de cinco argumentos, embora não se utilizem todos em simultâneo. Os dois primeiros argumentos, se especificados, indicam o início e o fim da série e se estes são os únicos argumentos, o resultado é equivalente ao do operador 'dois pontos'. Isto é, o resultado de seq(2,10) é o mesmo de 2:10.

Os parâmetros para a função seq(), bem como para outras funções, podem especificar-se sequencialmente (isto é, na ordem é que devem ser interpretados), ou então pelo *nome do argumento*, sendo neste caso a sua ordem irrelevante.

No caso da função seq(), os dois primeiros parâmetros podem ser especificados pelo nome, mediante a indicação ***from=valor_inicial*** e ***to=valor_final***; assim, seq(from=1, to=30) seq(1,30), seq(to=30, from=1) originam a mesma sequência idêntica à obtida com 1:30.

Os dois argumentos seguintes são ***by=incremento*** e ***length=valor***, que especificam o incremento entre dois valores sucessivos e o comprimento da sucessão, respectivamente. Se nenhum destes argumentos é especificado, o valor por defeito do incremento é a unidade (isto é, by=1).

Por exemplo:

```
> seq(-5,5, by=.2) -> s3
```

cria o vector s3 cujos elementos são c(-5.0, -4.8, -4.6, ... , 4.6, 4.8, 5.0). De modo similar:

```
> s4 <- seq(length=51, from=-5, by=.2)
```

gera o vector s4 cujos elementos são os mesmos do vector s3.

O quinto argumento desta função é ***along=vector***, e se se usa deve ser o único parâmetro especificado, e cria a sequência 1, 2, ... , length(vector), ou uma sucessão vazia se o vector é vazio (o que pode acontecer).

Uma função relacionada com seq() é a função ***rep***() que pode ser usada para replicar um objecto de diversas maneiras. A forma mais simples é:

```
> s5 <- rep(x, times=5)
```

que coloca sequencialmente 5 cópias de x no vector s5.

## 2.4 Vectores lógicos

Tal como vectores numéricos, R manipula igualmente valores lógicos. Os elementos de um vector lógico são um dos dois valores possíveis: **FALSE** (Falso) e **TRUE** (Verdadeiro). Estes valores são geralmente abreviados para **F** e **T**, respectivamente.

Os vectores lógicos são o resultado da avaliação de *condições*. Por exemplo:

```
> temp <- x > 13
```

cria o vector temp com o comprimento do vector x, e cujos elementos são FALSE correspondentes aos valores de x que não satisfaçam a condição, ou TRUE para os elementos de x que a cumpram a condição de ser superior a 13.

Os operadores lógicos são <, <=, >, >=, == para a igualdade exacta e != para a desigualdade. Além destes operadores, e sendo c1 e c2 duas expressões lógicas, então c1 & c2 é a sua intercepção ("and"), c1 | c2 é a sua reunião ("or") e !c1 é a negação de c1.

Os operadores lógicos podem ser usados nas operações aritméticas ordinárias, caso em que se transformam em vectores numéricos, sendo FALSE substituído por 0 (zero) e TRUE por 1 (um). Contudo, há situações em que os vectores lógicos e as suas transformações numéricas correspondentes não são equivalentes, como se verá na próxima sub-secção.

## 2.5 Valores em falta

Em alguns casos pode acontecer não se conhecerem todos os elementos de um vector. Quando um elemento ou valor não está disponível, ou está em falta ("*missing value*", no sentido estatístico), é-lhe atribuído o valor especial ***NA*** (do inglês, "*Not Available*"). De um modo geral, qualquer operação envolvendo um valor NA origina um outro valor NA. A justificação para esta regra é simplesmente que a especificação de uma operação não está completa, o resultado desta não pode ser conhecido, não estando disponível para operações subsequentes.

A função ***is.na(x)*** origina um vector lógico com a dimensão de x, com o valor TRUE se e só se o correspondente elemento de x é NA, e FALSE no caso contrário. Veja-se o seguinte exemplo, em que é criado o vector z, em que os primeiros três valores são os valores 1, 2, 3, e o quarto valor é um "missing value"; o vector ind tem os primeiros 3 elementos com o valor FALSE e o quarto valor é TRUE:

```
> z <- c(1:3,NA); ind <- is.na(z)
```

Note-se que a expressão lógica x == NA é diferente de is.na(x), pois NA não é na realidade um valor mas sim um indicador de um valor que não está disponível. Deste modo, x == NA dá origem a um vector com a dimensão de x, cujos elementos são todos NA pois a expressão lógica está incompleta e, por consequência, irresolúvel.

Refira-se ainda que há um segundo tipo de "missing values" que são originados por cálculos indeterminados, designados por valores ***NaN*** ("*Not a Number*"). Alguns exemplos de valores NaN são os produzidos por expressões indeterminadas do tipo:

```
> 0/0
> Inf - Inf
```

Em conclusão, is.na(x) origina o valor TRUE quer com valores NA como NaN; a função ***is.nan(x)*** dá TRUE apenas com valores NaN.

## 2.6 Vectores alfanuméricos

Valores alfanuméricos e vectores de caracteres são usados frequentemente em R, como por exemplo para as *etiquetas* ("*labels*") dos gráficos. Quando são necessários, os valores alfanuméricos são definidos como cadeias de caracteres delimitadas por aspas duplas, por exemplo "valores x", "Resultado de nova iteração".

As cadeias de caracteres podem ser concatenadas para um vector alfanumérico usando a função c(); é frequente o uso desta operação.

A função ***paste***() toma como entrada um número variável de argumentos e adiciona-os um a um sequencialmente numa cadeia de caracteres. Quaisquer valores numéricos dados como argumento da função paste() são convertidos em cadeias de caracteres, da mesma maneira como aconteceria ao serem impressos. Os argumentos são, por defeito, separados no vector resultante por um espaço em branco, mas este caracter de separação pode ser definido, usando o parâmetro ***sep="separador"***, em que a expressão "separador" define o separador a usar (pode inclusivamente ser nulo).

Por exemplo:

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

guarda em labs o seguinte vector de caracteres:

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Caso não se houvesse definido um separador nulo, o vector de caracteres seria:

```
c("X 1","Y 2","X 3","Y 4","X 5","Y 6","X 7","Y 8","X 9","Y 10")
```

Note-se que o vector c("X","Y"), com apenas dois valores, é repetido cinco vezes até perfazer o comprimento da sequência 1:10[4].

---

[4] paste(..., collapse="separador") permite colapsar os argumentos da função numa única cadeia de caracteres. Existem outras funções para manipular caracteres, tais como ***sub***() e ***substring***(). Ver ajuda.

## 2.7 Vectores indexados. Selecção e modificação de sub-vectores

Sub-conjuntos de elementos de um vector podem ser seleccionados dando ao nome do vector um vector de índices definidos entre parêntesis rectos. Mais genericamente, podem seleccionar-se sub-conjuntos de elementos de qualquer expressão que seja calculada e atribuída a um vector, através da indicação de um vector de índices imediatamente após a expressão.

Tais vectores de índices podem pertencer a uma das seguintes quatro categorias:

1. **Vector lógico**. Neste caso, o vector de índices deve ser da mesma dimensão do vector do qual se seleccionam os elementos. Os elementos correspondentes a TRUE são seleccionados e os que correspondem a FALSE são omitidos. Por exemplo:

   ```
   > y <- x[!is.na(x)]
   ```

   cria o objecto y que contém os valores definidos de x, na mesma sequência. Note-se que se x contém "missing values" y será de comprimento inferior a x. Do mesmo modo:

   ```
   > (x+1)[(!is.na(x)) & x > 0] -> z
   ```

   cria o objecto z onde coloca os valores de x+1 para os quais o respectivo valor de x esteja definido e seja positivo.

2. **Vector de valores inteiros positivos**. Neste caso, os valores do vector índice devem pertencer ao conjunto {1, 2, 3, ... , length(x)} . Os elementos correspondentes do vector são seleccionados e concatenados, nesta ordem, no vector resultante. O vector de índices pode ser de qualquer dimensão e o resultado é da mesma dimensão do vector de índices. Por exemplo, x[6] é o sexto elemento de x e

   ```
   > x[1:10]
   ```

   selecciona os primeiros 10 elementos de x (assumindo que length(x) não é inferior a 10). De modo idêntico:

   ```
   > c("x", "y")[rep(c(1,2,2,1), times=4)]
   ```

   origina uma cadeia de caracteres de comprimento 16, constituída pela sequência "x" "y" "y" "x" repetida quatro vezes.

3. **Vector de valores inteiros negativos**. Um vector de índices deste tipo especifica que os elementos devem ser excluídos, e não seleccionados. Então:

   ```
   > y <- x[-(1:5)]
   ```

   selecciona para o vector y desde o 6º até ao último elemento de x.

4. **Vector alfanumérico**. Esta opção só pode realizar-se aplicada a um objecto com o atributo names definido para identificar os seus componentes. Neste caso, um sub-vector do vector de nomes pode ser usado do mesmo modo que o vector de valores inteiros atrás descrito.

   ```
   > fruta <- c(5, 10, 1, 20)
   > names(fruta) <- c("laranja", "banana", "maçã", "pêra")
   > jantar <- fruta[c("maçã", "laranja")]
   ```

   O vector jantar contém os valores 1 e 5.

A vantagem dos índices alfanuméricos é que são mais fáceis de recordar que os vectores de índices numéricos. Esta opção é particularmente útil quando associada a "*data frames*" (***folhas de dados***), como se verá posteriormente.

Numa ordem de assignação, também se pode indexar a variável ou vector à qual irão ser assignados valores; neste caso, a atribuição de valores realiza-se apenas aos elementos indexados. A expressão deve ser da forma vector[vector_índice] já que a utilização de uma expressão arbitrária em vez do nome do vector não faria sentido neste contexto.

O vector ao qual se irão atribuir valores deve ser da mesma dimensão do vector de índices, e no caso de se tratar de um vector indexado lógico tem de ser do mesmo comprimento do vector que indexa. Por exemplo:

```
> x[is.na(x)] <- 0
```

substitui qualquer elemento de x não definido pelo valor 0, e:

```
> y[y < 0] <- -y[y < 0]
```

faz o mesmo que:

```
> y <- abs(y)
```

## 2.8 Classes de objectos

Os vectores são o tipo mais importante de objectos em R, mas há vários outros tipos de objectos aos quais nos referiremos de maneira mais formal nos próximos capítulos.

- ***Matrizes*** ou, mais genericamente***, variáveis indexadas*** ('*Arrays*') são a generalização multi-dimensional dos vectores. De facto, são vectores que podem ser indexados por dois ou mais índices, que correspondem a outras tantas dimensões, e que serão visualizadas de modo especial. Veja-se Capítulo 5 [Variáveis indexadas. Matrizes], pág. 22.
- Os ***factores*** são estruturas de dados que servem para representar dados categóricos. Veja-se Capítulo 4 [Factores], pág. 19.
- As ***listas*** são formas gerais de vectores em que os diversos elementos não necessitam de ser do mesmo tipo, e que muitas vezes são, por sua vez, vectores ou listas. As listas possibilitam um modo conveniente de apresentar os resultados de cálculos estatísticos. Veja-se Secção 6.1 [Listas], pág. 32.
- As ***folhas de dados*** ("*data frames*") são estruturas em forma de tabela do tipo matricial, nas quais as colunas podem ser de diferentes tipos. As tabelas de dados são apropriadas para representar matrizes de dados, em que cada linha se refere a uma unidade de observação ou indivíduo e as colunas representam as variáveis observadas, e que podem ser numéricas ou alfanuméricas. Muitos resultados experimentais são facilmente representados numa tabela de dados: os tratamentos são variáveis categóricas e as variáveis resposta são variáveis numéricas. Secção 6.2 [Tabelas de dados], pág. 34.
- As ***funções*** são elas próprias consideradas em R como objectos, que podem ser guardados no espaço de trabalho. Deste modo, o utilizador pode desenvolver e guardar

as suas próprias funções, ampliando as capacidades de R. Veja-se Capítulo 10 [Defina as suas próprias funções], pág. 52.

# 3 Objectos: modos e atributos

## 3.1 Atributos intrínsecos: modo e dimensão

As entidades que R manipula designam-se por *objectos*. Exemplos de objectos são vectores de valores reais, vectores de números complexos, vectores de valores lógicos e vectores de cadeias de caracteres (ou vectores alfanuméricos). Estes objectos são designados por estruturas 'atómicas', pois os seus elementos são todos do mesmo tipo, ou *modo*, nomeadamente *numérico*[1], *complexo*, *lógico* ou *alfanumérico*, respectivamente.

Os vectores devem ter *todos os seus valores do mesmo modo*, *lógico*, *numérico*, *complexo* ou *alfanumérico*. Isto é, um vector não pode ser ambíguo em relação ao tipo de valores que contém. A única excepção a esta regra é que todos os tipos de vectores podem conter o valor especial ***NA*** para elementos não definidos. Mesmo que um vector esteja vazio, tem na mesma um modo. Por exemplo, um vector alfanumérico vazio aparece como ***character(0)*** e um vector numérico vazio como ***numeric(0)***.

As listas manuseadas por R são do modo ***list***. Estas são sequências ordenadas de objectos, cada qual pode ser de modo distinto. As listas são designadas por estruturas 'recursivas', em vez de 'atómicas', pois os seus elementos podem ser outras listas.

As outras estruturas 'recursivas' são as funções cujo modo é "***function***" e as expressões com modo "***expression***". O modo função engloba as funções que fazem parte do sistema R bem como as funções definidas pelo utilizador, que serão discutidas posteriormente. Os objectos cujo modo é expressão constituem um módulo avançado de R, que não será abordado nestas notas, aparte do mínimo necessário ao tratamento de fórmulas na descrição de modelos estatísticos.

Com o modo de um objecto designa-se o tipo fundamental de dos seus elementos constituintes. O modo é um caso particular dos *atributos* de um objecto. Os *atributos* fornecem informação específica acerca do objecto. Outro atributo é a dimensão comprimento ("*length*") de um objecto. As funções ***mode(objecto)*** e ***length(objecto)*** usam-se para saber qual o modo e o comprimento de qualquer estrutura definida.

Podem atribuir-se outros propriedades a um objecto com a função ***attributes(object)***, como se verá na Secção 3.3[Obter e definir atributos], pág. 17. Por isso, mode e lenght são designados por atributos intrínsecos do objecto.

Por exemplo, se z é um vector de dimensão 100 de números complexos, então o resultado da função ***mode(z)*** é "***complex***" e o de ***length(z)*** é o valor 100.

R provoca a alteração do modo de um objecto sempre que considere necessário esta alteração (e mesmo em situações em que o não é). Por exemplo, com o vector:

```
> z <- 0:9
```

[1] O modo numérico consiste na realidade de dois modos distintos: *inteiro* e *dupla precisão*.

pode definir-se:

```
> digitos <- as.character(z)
```

após o que digitos passa a ser o vector alfanumérico c("0", "1", "2", ... , "9"). Uma provocação mais para alterar o modo, e re-constroi-se de novo um vector numérico:

```
> d <- as.integer(digitos)
```

Neste momento, d e z são o mesmo vector[2]. Há um vasto leque de funções da forma ***as.****something()* para provocar a alteração de modo, ou para investir um objecto com algum atributo de que não disponha. O utilizador deve consultar os diversos arquivos de ajuda para se familiarizar com estas funções.

## 3.2 Alterar a dimensão de um objecto

Um objecto vazio (isto é, sem elementos) tem um modo. Por exemplo:

```
> e <- numeric()
```

define a estrutura e como um vector numérico vazio. De modo similar, ***character()*** define um vector alfanumérico vazio. Uma vez que o objecto de qualquer dimensão tenha sido criado, novos elementos podem ser-lhe atribuídos simplesmente dando um valor de índice fora da sua actual dimensão. Assim:

```
> e[3] <- 17
```

provoca que agora o vector e tenha dimensão3 (os dois primeiros elementos são neste momento ***NA***). Esta regra aplica-se a qualquer estrutura, desde que o modo dos elementos adicionais seja concordante com o modo do objecto.

Este ajustamento automático da dimensão de um objecto é usado frequentemente, por exemplo com a função ***scan()*** para fazer a entrada de valores. (Veja Secção 7.2 [A função scan()], pág. 39).

De modo semelhante, para truncar a dimensão de um objecto requer apenas um comando de assignação. Se *alfa* é um objecto de dimensão 10, então:

```
> alfa <- alfa[2*1:5]
```

transforma alfa num objecto de dimensão 5, constituído apenas pelos elementos de índice par. Os elementos de índice ímpar não são retidos.

## 3.3 Obter e definir atributos

A função ***attributes(objecto***) dá a lista de todos os atributos não intrínsecos actualmente definidos para o objecto em causa. A função ***attr(objecto, atributo)*** pode ser usada para seleccionar um atributo específico. Estas funções só raramente são usadas, excepto em circunstâncias bastante especiais, quando um novo atributo tem de ser definido com uma

---

[2] Geralmente a forçagem de alteração de modo numérico para alfanumérico, e de novo para numérico não é exactamente reversível, devido aos problemas de arredondamento dos dígitos dos valores.

finalidade específica, tal como por exemplo associar a data de criação ou um operador com um objecto. O conceito é, contudo, muito importante.

Deve ser dado muito cuidado quando se definem ou eliminam atributos, pois eles fazem parte integral do objecto usado em R.

Quando a função ***attr()*** é usada no lado esquerdo de um comando de assignação, pode ser usada quer para associar um novo atributo quer para alterar um atributo existente. Por exemplo:

```
> attr(z,"dim") <- c(10,10)
```

permite que R trate z como uma matriz de 10 linhas por 10 colunas.

## 3.4 Classes de objectos

Um atributo especial designado por *class* (classe) do objecto é usado para vocacionar esse objecto para o estilo de programação em R.

Por exemplo se um objecto é da classe "*data.frame*", será visualizado segundo uma determinada forma, a função ***plot()*** visualizá-lo-á de determinada forma, e qualquer outra função de uso genérico tal como ***summary()*** reagirão perante este objecto de uma maneira especificamente orientada para esta classe.

Para remover temporariamente os efeitos de classe de um objecto, usa-se a função ***unclass()***. Por exemplo, se inverno é da classe "*data.frame*" então:

```
> inverno
```

visualiza este objecto na forma de tabela de dados, que é semelhante a uma matriz, enquanto que:

```
> unclass(winter)
```

o visualizará como uma lista normal. Somente em situações muito especiais vai sentir a necessidade de usar esta capacidade, mas estamos no ponto em que está a familiarizar-se com os conceitos de classe e funções.

As funções e as classes serão posteriormente discutidas de modo muito resumido na Secção 10.9 [Orientação para objectos], pág. 63.

# 4 Factores

Um ***factor*** é um vector que se usa para especificar uma classificação discreta em categorias dos componentes de outros vectores da mesma dimensão. Em R existem *factores ordenados* e *não ordenados*. Se bem que as aplicações práticas de factores seja na definição de fórmulas de modelos estatísticos (veja Secção 11.1.1 [Contrastes], pág. 67), aqui limitar-nos-emos a apresentar alguns exemplos.

## 4.1 Um exemplo específico

Suponha que se dispõem de uma amostra de 30 profissionais liberais de diversos distritos do continente; o vector provincia contém as iniciais da província[1] de cada um dos elementos desta amostra:

```
> provincia <- c("tmd", "bl", "min", "rib" , "rib", "ba", "alt",
  "alt", "min", "alg", "rib", "alg", "min", "min", "bl", "tmd",
  "bl", "ba", "alt", "alg", "min", "rib", "rib", "alt", "bl",
  "bb", "rib", "alg", "alg", "bb")
```

Para criar um *factor* a partir do vector provincia, usa-se a função ***factor()***:

```
> fprovincia <- factor(provincia)
```

A função ***print()*** manuseia os factores de modo específico, e a saída é a seguinte:

```
> fprovincia
```

[1] tmd bl min rib rib ba alt alt min alg rib alg min min bl tmd bl ba alt

[20] alg min rib rib alt bl bb rib alg alg bb

Levels: alg alt ba bb bl min rib tmd

Para obter as categorias de um factor usa-se a função ***levels()***:

```
> levels(fprovincia)
```

[1] "alg" "alt" "ba" "bb" "bl" "min" "rib" "tmd"

## 4.2 A função tapply() e variáveis indexadas desiguais

Em continuação do exemplo anterior, suponhamos que dispomos de outro vector contendo os rendimentos desses contabilistas (medidos numa unidade monetária apropriada):

```
> rendimento <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70,
                  42, 56, 61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48,
                  52, 46, 59, 46, 58, 43)
```

Para calcular a média amostral de cada uma das categorias (neste caso, de cada província), podemos usar a função ***tapply()***:

```
> rendamedia <- tapply(rendimento, fprovincia, mean)
```

---

[1] Estes códigos representam: min: Minho; tmd: Trás-os-Montes; ba: Beira Alta; bb: Beira Baixa; bl: Beira Litoral; alt: Alentejo; alg: Algarve; rib: Ribatejo.

que calcula o vector de médias de cada uma das categorias:

```
> rendamedia
```

| alg | alt | ba | bb | bl | min | rib | tmd |
|---|---|---|---|---|---|---|---|
| 56.00000 | 52.25000 | 55.50000 | 44.50000 | 55.00000 | 53.60000 | 57.33333 | 60.50000 |

A função ***tapply()*** aplica uma função, neste caso a função ***mean()***, a cada grupo de elementos do primeiro argumento (vector rendimento), definidos pelos níveis ou categorias do segundo argumento (vector fprovincia), como se cada grupo fosse um vector por si só. O resultado é uma estrutura cujo comprimento é o número de categorias do factor. Veja a ajuda associada a *tapply()* para mais detalhes.

Suponha agora que pretendemos calcular os erros-padrão da média dos rendimentos por província. Para tal, necessitamos de definir uma função que calcule o erro-padrão da média de um vector numérico. Sendo ***var()*** a função que calcula a variância amostral, então a função para calcular o erro-padrão da média[2] pode ser definida pela expressão[3]:

```
> erropadrao <- function(x)
              {sqrt(var(x)/length(x))}
```

Agora basta aplicar a função *erropadrao()* como argumento à função ***tapply()***:

```
> errop <- tapply(rendimento, fprovincia, erropadrao)
```

Os valores calculados são:

```
> errop
```

| alg | alt | ba | bb | bl | min | rib | tmd |
|---|---|---|---|---|---|---|---|
| 5.244044 | 2.657536 | 4.500000 | 1.500000 | 2.738613 | 4.106093 | 4.310195 | 0.500000 |

Como exercício, pode calcular o intervalo de confiança a 95% para a média dos rendimentos por província. Para tal, pode usar a função ***tapply()***, a função ***length()*** para calcular os tamanhos amostrais, e a função ***qt()*** para obter os quantis das distribuições t de Student correspondentes.

A função ***tapply()*** pode usar-se para aplicar uma função a um vector indexado por diferentes categorias simultaneamente. Por exemplo, pode interessar dividir a amostra por estado ou por sexo. Os elementos do vector vão ser agrupados em sub-amostras correspondentes às distintas categorias ou níveis, e a função é aplicada a cada uma destas sub-amostras. O resultado é uma variável indexada etiquetada com os níveis de cada categoria.

A combinação de um vector com um factor a etiquetá-lo é um exemplo do que se designa por *variável indexada desigual* ("*ragged array*"), pois possivelmente os tamanhos das sub-classes são diferentes. Quando estes tamanhos são iguais, a indexação pode fazer-se implicitamente e mais eficientemente, como se verá adiante.

---

[2] Erro-padrão da média: $s_{\bar{x}} = \sqrt{s^2/n}$

[3] A definição de funções será tratado no Capítulo 10 [Defina as suas próprias funções], pág. 52

## 4.3 Factores ordenados

Os níveis dos factores são guardados por ordem alfabética (tal como no exemplo anterior), ou na ordem em que se especificaram explicitamente na função ***factor()***.

Por vezes há uma ordenação natural nos níveis de um factor, que pretendemos ter em conta para as análises estatísticas subsequentes. A função ***ordered()*** permite criar este tipo de factores e o seu uso é idêntico ao da função ***factor()***. Os factores criados pela função ***factor()*** são designados factores nominais ou, não havendo risco de confusão, simplesmente por factores; os que são criados com a função ***ordered()*** são designados factores ordenados. Na maior parte das vezes, a única diferença entre ambos os tipos consiste em que os ordenados são impressos indicando a ordem dos níveis

```
> ordered(provincia)
```

[1] tmd bl  min rib rib ba  alt alt min alg rib alg min min bl  tmd bl  ba  alt

[20] alg min rib rib alt bl  bb  rib alg alg bb

Levels: alg < alt < ba < bb < bl < min < rib < tmd

Além disso, os contrastes gerados pelos dois tipos de factores ao ajustar modelos lineares são distintos.

# 5 Variáveis indexadas. Matrizes

## 5.1 Variáveis indexadas

Uma variável indexada pode considerar-se como uma colecção de dados, indexada por vários índices. R permite criar e manipular variáveis indexadas em geral, e matrizes em particular.

Um vector de dimensões é um vector de números inteiros. Se o seu comprimento é k, então a variável indexada correspondente é k-dimensional. Os elementos do vector de dimensões indicam os limites superiores dos k índices. Os limites inferiores valem sempre 1 (um). Uma matriz é uma variável indexada com k=2.

Um vector pode transforma-se numa variável indexada quando se assigna um vector de dimensões ao atributo *dim*. Suponhamos, por exemplo, que z é um vector de 1500 elementos. A assignação:

```
> dim(z) <- c(3,5,100)
```

faz com que R considere z como uma matriz de $3\times5\times100$ elementos.

Existem outras funções, como ***matrix()*** e ***array()***, que permitem assignações mais simples e naturais, como se verá na Secção 5.4 [A função array()], pág. 24.

Os elementos do vector passam a formar parte da variável indexada seguindo a regra de prioridade máxima à coluna, também usada na linguagem FORTRAN, na qual o primeiro índice é o que se move mais rápido e o último é o mais lento[1].

Por exemplo, se se define a variável indexada a, com vector de dimensões c(3,4,2), esta matriz terá $3\times4\times2=24$ elementos que obedecem à sequência a[1,1,1], a[2,1,1],...,a[2,4,2], a[3,4,2].

## 5.2 Elementos de uma variável indexada

Os elementos individuais de uma variável indexada podem ser referidos com o nome da variável seguido, entre parêntesis rectos, dos índices respectivos separados por vírgula.

Em geral, pode referir-se qualquer sub-secção de uma variável indexada, mediante uma sucessão de *vectores-índices*, tendo em conta que se um elemento do *vector-índice é vazio, equivale a utilizar toda a amplitude de valores para o dito índice.*

---

[1] Experimente com os seguintes comandos:

```
> z <- c(1:100)
```

Veja o vector z; de seguida, faça:

```
> dim(z) <- c(10,10)
```

e veja como está organizada a matriz z.

Assim, no exemplo da variável indexada anterior, designada com o nome a, a sub-secção a[2, , ] é uma variável com as dimensões $4\times2$, com o vector de dimensões c(4,2), contendo os seguintes elementos da variável a, na ordem indicada:

c(a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1],

a[2,1,2], a[2,2,2], a[2,2,2], a[2,4,2]

A variável a[ , , ], que consiste em omitir todos os índices, equivale à variável a completa.

Para qualquer variável indexada, por exemplo z, o vector de dimensões pode referir-se explicitamente mediante a função ***dim(z)*** (pode usar-se em qualquer dos lados da assignação).

Se se especifica uma variável indexada com um só índice, somente são especificados os elementos correspondentes ao vector de dados, e o vector de dimensões é ignorado. No caso de o índice não ser um vector, mas uma variável indexada, o tratamento é diferente, como se verá na próxima secção.

## 5.3 Uso de variáveis indexadas como índices

Uma variável indexada pode utilizar não apenas um vector de índices, mas também uma variável indexada de índices, quer para assignar um vector a uma colecção irregular de elementos de uma variável indexada, quer para extrair uma colecção irregular de elementos para um vector.

Vejamos um exemplo sobre uma matriz, a fim de tornar mais clara a exposição. No caso de uma matriz, que é uma variável indexada com dois índices, pode construir-se uma matriz de índices constituída por duas colunas e várias linhas. As entradas da matriz de índices identificam as linhas e as colunas. Suponhamos que x é uma matriz $4\times5$ e que se desejam efectuar as seguintes tarefas:

- Extrair os elementos x[1,3], x[2,2] e x[3,1] para um objecto com estrutura de vector;
- Substituir estes elementos de x por zeros.

  Para tal, pode usar-se uma matriz de índices de $3\times2$ elementos. A matriz x pode ser gerada com o seguinte comando:

```
> x <- array(1:20, dim=c(4,5))      # gera a matriz x
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20

> i <- array(c(1:3,3:1), dim=c(3,2))  # i é uma matriz de índices 3x2
> i
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
   > x[i]
   [1] 9  6  3
   > x[i] <- 0              # substitui os elementos por zeros
   > x
   [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
```

Um exemplo algo mais complexo consiste em gerar a matriz de desenho de um delineamento por blocos definido por dois factores, *bloco* (com **b** níveis) e *variedade* (com **v** níveis), sendo n o número de parcelas. Pode fazer-se do seguinte modo (nota: b, v e n são valores numéricos):

```
> xb <- matrix(0, n, b)
> xv <- matrix(0, n, v)
> ib <- cbind(1:n, bloco)
> iv <- cbind(1:n, variedade)
> xb[ib] <- 1
> xv[iv] <- 1
> x <- cbind(xb, xv)
```

A matriz de incidência N pode construir-se com:

```
> N <- crossprod(xb, xv)
```

Contudo, uma maneira mais simples de construir esta matriz é usar a função ***table()***:

```
> N <- table(bloco, variedade)
```

## 5.4 A função array()

Uma variável indexada pode construir-se modificando o atributo *dim* de um vector, ou directamente, utilizando a função array() que tem a forma:

```
> z <- array(vector_de_dados, vector_de_dimensões)
```

Por exemplo, se o vector h contém 24 (ou menos) valores numéricos, a ordem:

```
> z <- array(h, dim=c(3,4,2))
```

armazena o vector h na variável indexada z, de dimensão $3\times4\times2$. Se o tamanho do vector h é exactamente 24, o resultado do comando anterior coincide com:

```
> dim(z) <- c(3,4,2)
```

Contudo, se h tem menos que 24 elementos, os seus valores repetem-se desde o princípio tantas vezes quantas as necessárias para perfazer os 24 elementos de z (veja-se Secção 5.4.1 [Reciclagem], pág. 25). O caso extremo, muito frequente, corresponde a um vector de comprimento 1, como no seguinte exemplo:

```
> z <- array(0, c(3,4,2))
```

em que z é uma variável indexada cujos elementos são todos zero.

A expressão ***dim(z)*** refere-se ao vector de dimensões c(3,4,2), **z[1:24]** refere-se ao vector de dados semelhante ao vector h, e **z[]** com o índice vazio (ou apenas z) refere-se a todo a variável indexada z.

As variáveis indexadas podem usar-se em expressões aritméticas e o resultado é uma variável indexada formada a partir das operações elemento a elemento dos vectores subjacentes. Os atributos *dim* dos operando, regra geral, devem ser iguais e coincidirem com o vector de dimensões do resultado. Assim, se A, B e C são variáveis indexadas similares, então:

```
> D <- 2*A*B + C + 1
```

origina uma variável indexada similar, D, cujo vector de dados é o resultado das operações aritméticas indicadas sobre os vectores de dados subjacentes a A, B e C. As regras exactas correspondentes aos cálculos em que se misturam variáveis indexadas e vectores devem ser atentamente estudadas.

### 5.4.1 Operações com variáveis indexadas e vectores. Reciclagem

As regras precisas que afectam as operações sobre elementos em que se misturam variáveis indexadas e vectores raramente são explicitamente referenciadas. Pela experiência, consideramos que as seguintes fiáveis as seguintes regras:

- A expressão é analisada e avaliada da esquerda para a direita.
- Se a expressão envolve um vector operando mais curto que os demais, é extendido reciclando os seus valores até igualar o comprimento dos restantes operandos.
- Se só há variáveis indexadas e vectores mais curtos, as variáveis indexadas devem ter o mesmo atributo *dim*, ou ocorrerá um erro.
- Se existe um vector operando mais extenso que uma variável indexada ou matriz ocorrerá erro.
- Se estão envolvidas variáveis indexadas e não ocorre erro, o resultado da expressão origina uma variável indexada com o mesmo atributo *dim* das que intervêm na operação.

## 5.5 Produto externo de duas variáveis indexadas

Uma operação fundamental com variáveis indexadas é o *produto externo*. Se a e b são duas variáveis indexadas numéricas, o seu produto externo é uma variável indexada cujo vector de dimensão é a concatenação dos correspondentes aos operandos, na ordem da operação, e cujo

vector de dados é obtido mediante todos os possíveis produtos dos elementos dos vectores subjacentes de a e b. A operação 'produto exterior' é indicada pelo operador ***%o%***:

```
> ab <- a %o% b
```

ou então, com a função ***outer()***:

```
> ab <- outer(a, b, "*")
```

Sejam a <- c(1:5) e b <- c(6:10). Então ab é:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    6    7    8    9   10
[2,]   12   14   16   18   20
[3,]   18   21   24   27   30
[4,]   24   28   32   36   40
[5,]   30   35   40   45   50
```

Usando a função outer(), a operação multiplicação pode ser substituída por qualquer outra operação aritmética ou função de duas variáveis. Por exemplo, para calcular a função $f(x,y)=\cos(y)/(1+x^2)$ sobre a retícula formada por todos os pontos que se obtêm pelas ordenadas e abcissas definidas pelos elementos dos vectores x e y respectivamente, pode definir-se a seguinte função (a definição de funções em R será abordada no Capítulo 10 [Defina as suas próprias funções], pág. 53) :

```
> f <- function(x,y)
      {cos(y)/(1+x^2)}
> z <- outer(x,y,f)
```

Em particular, o produto externo de dois vectores é uma variável indexada com dois índices (isto é, uma matriz de ordem 1, pelo menos). Deve ter-se em conta que o produto externo não é comutativo.

### Exemplo: Distribuição dos determinantes de uma matriz de dígitos $2\times2$

Um exemplo pertinente da utilização da função outer() é o cálculo do determinante de uma matriz $2\times2$, [a, b; c, d], em que cada um dos seus elementos é um número natural entre 0 e 9 (isto é, um dígito). O problema consiste em calcular os determinantes, $ad-bc$, d e todas as matrizes deste tipo, e representar graficamente a sua distribuição, supondo que cada dígito é seleccionada aleatoriamente de uma distribuição uniforme.

Para tal, pode utilizar a função outer() duas vezes:

```
> d <- outer(0:9,0:9)
> fr <- table(outer(d, d, "-"))
> plot(as.numeric(names(fr)), fr, type="h",
      xlab="Determinante", ylab="Frequência")
```

Note-se como se há forçado como numérico o atributo *names* da tabela de frequências, de modo a recuperar a amplitude dos valores dos determinantes. A forma aparentemente "óbvia"

de resolver este problema com o uso de ciclos ***for***, que se discutirão no Capítulo 9 [Ciclos. expressões condicionais], pág. 51, é tão ineficaz que se torna impraticável.

Ao observar o resultado, é talvez surpreendente como aproximadamente 1 em cada 20 destas matrizes é singular.

## 5.6 Transposta generalizada de uma variável indexada

A função ***aperm(a, perm)*** pode usar-se para permutar a variável indexada a. O argumento ***perm*** deve ser uma permutação dos valores inteiros {1, ..., k}, sendo k o número de índices de a. O resultado é uma variável indexada do mesmo tamanho que a, em que a dimensão que na variável original era ***perm[j]*** passa agora a ser a dimensão j. Se A é uma matriz (no sentido matemático), então:

```
> B <- aperm(A, c(2,1)
```

origina a matriz B, que é a matriz transposta de A. No caso de matrizes (no sentido matemático), é mais fácil usar a função ***t()***, e o comando é simplesmente `B <- t(A)`.

## 5.7 Operações com matrizes

Como já anteriormente se definiu, uma matriz é simplesmente uma variável indexada com dois índices. Face à sua importância, necessitam ser tratadas numa secção à parte. R dispõe de muitos operadores e funções específicas para matrizes, Por exemplo, acabámos de referir que ***t(x)*** é a matriz transposta de x. As funções ***nrow(x)*** e ***ncol(x)*** indicam o número de linhas e de colunas de uma matriz x.

### 5.7.1 Produto matricial. Matriz inversa. Resolução de sistemas lineares

O operador ***%*%*** realiza o produto matricial. Uma matriz $n \times 1$ ou $1 \times n$ pode ser usada, caso seja necessário, como um vector n-dimensional. Analogamente, R pode usar automaticamente um vector numa operação matricial, convertendo-o para uma matriz-fila ou matriz-coluna, quando tal seja possível (por vezes, esta conversão pode resultar ambígua, como se verá).

Se, por exemplo, A e B são matrizes quadradas com o mesmo tamanho, então:

```
> A * B
```

dá como resultado uma matriz com o produto, elemento a elemento, das duas matrizes, enquanto que:

```
> A %*% B
```

origina o produto matricial de A por B. Se x é um vector, então:

```
> x %*% A %*% x
```

cria uma forma quadrática[2].

---

[2] A expressão x %*% é ambígua, pois tanto pode significar $x'x$ como $xx'$, em que x é um vector coluna. Neste tipo de casos, a interpretação corresponde à matriz de menor tamanho,

A função ***crossprod()*** calcula o produto cruzado de duas matrizes, isto é, `crossprod(X,y)` é o mesmo que `t(X) %*% y`, mas a função é mais eficiente. Se se omite o segundo argumento da função ***crossprod()***, é assumido igual ao primeiro.

O resultado da função ***diag(v)*** depende do argumento. Se v é um vector, ***diag(v)*** dá uma matriz diagonal, em que os elementos da diagonal principal são os elementos do vector v. Por outro lado, se M é uma matriz, ***diag(M)*** dá um vector com os elementos da diagonal principal de M (esta é convenção usada pelo programa MATLAB para a função ***diag()***). Por último, se k é um único valor numérico natural, ***diag(k)*** origina a matriz identidade $k \times k$ .

### 5.7.2 Autovalores e autovectores

A função ***eigen(Sm)*** calcula os valores próprios ou autovalores, e os vectores próprios ou autovectores, de uma matriz simétrica Sm. O resultado é uma lista com duas componentes, cujo nome é respectivamente values e vectors. A assignação:

```
> ev <- eigen(Sm)
```

cria a lista *ev*, em que *ev$val* se refere ao vector de valores próprios e *ev$vec* é a matriz com os vectores próprios. Se só necessitamos de calcular os autovalores, pode fazer-se:

```
> autoval <- eigen(Sm)$values
```

e autoval conterá o vector de valores próprios, sendo a segunda componente descartada. Se a expressão:

```
> eigen(Sm)
```

é usada como comando, as duas componentes são visualizadas com os respectivos nomes.

### 5.7.3 Decomposição em valores singulares. Determinantes

A função ***svd()*** admite como argumento uma matriz qualquer, M, e calcula a sua decomposição em valores singulares, que consiste em obter três matrizes U, D e V, tais que a primeira é uma matriz de colunas ortogonais com o mesmo espaço de colunas que M, a segunda é uma matriz diagonal de números não negativos e a terceira é uma coluna de colunas ortogonais com o mesmo espaço de linhas que M, tais que M=U%*%D%*%t(V). D é apresentado sob a forma de vector formado pelos elementos diagonais. O resultado de ***svd()*** é uma lista com três componentes, cujos nomes são d, u e v, correspondentes às matrizes descritas.

Se M é uma matriz quadrada, é fácil ver que:

```
> AbsDetM <- prod(svd(M)$d)
```

calcula o valor absoluto do determinante de M. Se precisa deste cálculo frequentemente pode defini-lo como uma nova função de R:

---

pelo que o resultado é neste caso o escalar $x'x$ . A matriz $xx'$ pode calcular-se fazendo cbind(x) %*% x ou x %*% rbind(x), pois o resultado de rbind() ou de cbind() é sempre uma matriz.

```
> AbsDet <- function(M)
      {prod(svd(M)$d)}
```

com a qual poderá usar AbsDet() como qualquer outra função. Deixa-se como exercício, trivial embora útil, o cálculo de uma função, ***tr()***, que calcula o traço de uma matriz quadrada. Tenha em conta que não necessita realizar nenhuma iteração; estude atentamente o código da função anterior.

### 5.7.4 Ajustamento por mínimos quadrados. Decomposição QR

A função ***lsfit()*** calcula uma lista que contém os resultados de um ajustamento pelo método dos mínimos quadrados. Uma assignação da forma:

```
> minquad <- lsfit(X, y)
```

guarda os resultados do *ajustamento por mínimos quadrados* de um vector de observações, y, e uma matriz de desenho, X. Veja a ajuda sobre esta função para mais detalhes, bem como para a função ***ls.diag()*** subsequente que, entre outras coisas, permite diagnosticar a regressão. Note que o termo independente é automaticamente incluído, não sendo necessário incluí-lo explicitamente como uma coluna de X.

Outra função estreitamente relacionada é a função ***qr()*** e suas similares. Considere as seguintes assignações:

```
> xplus <- qr(x)
> b <- qr.coef(xplus, y)
> fit <- qr.fitted(xplus, y)
> res <- qr.resid(xplus, y)
```

que calculam a projecção ortogonal de y sobre x,, guardando os resultados em fit, a projecção sobre o complemento ortogonal em res e o vector de coeficientes para a projecção em b (isto é, b é essencialmente o resultado do operador "backslash" do MATLAB).

Não é necessário assumir que x seja de ordem completa. As redundâncias são detectadas e logo removidas .

Esta era a metodologia antiga, de 'baixo nível', de efectuar o ajustamento pelo método dos mínimos quadrados. Embora continue a ser útil em determinados contextos, é actualmente substituída pelas potencialidades dos modelos estatísticos, como se verá no Capítulo 11 [Modelos estatísticos em R], pág. 64.

## 5.8 Partições de uma matriz. Funções cbind() e rbind()

Como já anteriormente visto de modo informal, as matrizes podem ser reconstruídas pela junção de vectores ou outras matrizes. Genericamente, ***cbind()*** faz a união horizontal (modo coluna) e ***rbind()*** faz a união vertical (modo linha) de matrizes.

Na assignação:

```
> x <- cbind(arg_1, arg_2, arg_3, ...)
```

os argumentos de cbind() devem ser vectores (com qualquer comprimento) ou matrizes com o mesmo número de linhas. O resultado é uma matriz que resulta da concatenação lateral dos argumentos arg_1, arg_2, ... aumentando o número de colunas.

Se alguns dos argumentos de ***cbind()*** são vectores, estes não podem ser mais extensos que o tamanho das colunas das matrizes envolvidas; se os vectores são de menor comprimento, são estendidos ciclicamente até igualarem o comprimento das colunas das matrizes (ou o comprimento do vector mais extenso, se não existe nenhuma matriz).

A função ***rbind()*** faz a correspondente concatenação vertical. Neste caso, qualquer vector especificado em argumento, provavelmente ciclicamente estendido, é tomado como vector fila (ou linha).

Suponhamos que X1 e X2 têm o mesmo número de linhas. Para combinar estas matrizes horizontalmente, numa matriz X, cuja primeira coluna se pretende que seja de 1's, faz-se o comando:

```
> X <- cbind(1, X1, X2)
```

O resultado de ***cbind()*** ou ***rbind()*** tem sempre a estrutura de matriz. Estas funções são assim o modo mais expedito de tratar o vector x como matriz coluna ou matriz linha, respectivamente.

## 5.9 A função concatenação c() com matrizes

Enquanto que as funções ***cbind()*** e ***rbind()*** são funções de concatenação que respeitam o atributo *dim*, a função ***c()*** não o respeita, antes pelo contrário retira os atributos *dim* ou *dimnames* aos objectos numéricos, o que por certo, é útil em determinadas situações.

A forma *oficial* de transformar uma variável indexada no seu vector subjacente é utilizar a função ***as.vector()***:

```
> vec <- as.vector(X)
```

O mesmo resultado é obtido utilizando a função ***c()***, devido ao efeito colateral mencionado:

```
> vec <-c(X)
```

Existem umas diferenças subtis entre as duas alternativas, porém a opção entre ambas é fundamentalmente uma questão de estilo (preferencialmente use a maneira formal).

## 5.10 Tabelas de frequências a partir de factores

Vimos que um factor define uma partição por categorias, ou uma tabela de entrada simples. De modo semelhante, dois factores definem uma tabela de dupla entrada, e assim sucessivamente. A função ***table()*** permite calcular tabelas de frequências a partir de factores de igual comprimento. Se existem k categorias, o resultado será uma variável k-indexada contendo as frequências de cada categoria.

Suponhamos, por exemplo, que fprovincia é um factor de categorias que são as iniciais das províncias[3], associado a um vector de dados. A assignação:

```
> freqprov <- table(fprovincia)
```

cria em freqprov uma tabela de frequências de cada província na amostra. As frequência são ordenadas e etiquetadas pelos níveis ou categorias do factor. Esta ordem é equivalente, e mais fácil que:

```
> freqprov <- tapply(fprovincia, fprovincia, comprimento)
```

Suponha agora que frenda é um factor que classifica ou agrupa os rendimentos por classes pré-definidas, por exemplo com a função ***cut()***:

```
> factor(cut(rendimento, breaks=35+10*(0:7))) -> frenda
```

Então, para calcular uma tabela de frequências de dupla entrada:

```
> table(frenda, fprovincia)
```

| | fprovincia | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| frenda | alg | alt | ba | bb | bl | min | rib | tmd |
| (35,45] | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| (45,55] | 1 | 3 | 1 | 1 | 2 | 1 | 1 | 0 |
| (55,65] | 2 | 1 | 1 | 0 | 2 | 3 | 3 | 2 |
| (65,75] | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

A extensão para tabelas de frequência de múltiplas entradas é imediata.

[3] Referimo-nos ao exemplo usado na Secção 4.1 e 4.2, pág.19.

# 6 Listas e folhas de dados

## 6.1 Listas

Uma *lista* em R é um objecto constituído por uma colecção ordenada de objectos, conhecidos como as suas *componentes*.

Não é necessário que os objectos sejam do mesmo modo ou tipo; assim, uma lista pode ser constituída, por exemplo, por um vector numérico, um valor lógico, uma matriz, um vector complexo, uma variável indexada alfanumérica, e uma função. De seguida apresenta-se um exemplo de uma lista:

```
> lst <- list(nome="José", esposa="Maria",
              n.filhos=3, idade.filhos=c(4,7,9))
```

O aspecto da visualização desta lista é o seguinte:

```
> lst
$nome
[1] "José"
$esposa
[1] "Maria"
$n.filhos
[1] 3
$idade.filhos
[1] 4 7 9
```

As componentes estão sempre *numeradas* e podem ser identificadas pelo respectivo número. Na lista lst anterior, com 4 componentes, cada uma delas pode ser referido por lst[[1]], lst[[2]], lst[[3]], lst[[4]]. Como a última componente lst$idade.filhos ou lst[[4]] é um vector, a expressão lst[[4]][1] identifica o valor 4 (idade do primeiro filho):

```
> lst$idade.filhos
[1] 4 7 9
> lst[[4]]
[1] 4 7 9
> lst[[4]][1]
[1] 4
```

A função ***length()*** aplicada a uma lista devolve o número de componentes dessa lista.

As componentes de uma lista podem ter *nome*, caso em que podem ser identificadas por esse nome, com uma expressão do tipo:

```
> nome_da_lista$nome_da_componente
```

Esta convenção permite a obtenção de uma componente sem o emprego do respectivo número. No exemplo anterior:

**lst$nome** coincide com **lst[[1]]**, cujo conteúdo é "José",

**lst$esposa** coincide com **lst[[2]]**, cujo conteúdo é "Maria",

**lst$n.filhos** coincide com **lst[[3]]**, cujo conteúdo é 3,

**lst$idade.filhos** coincide com **lst[[4]]**, cujo conteúdo é o vector c(4, 7, 9),

**lst$idade.filhos[1]** é o mesmo **lst[[4]][1]** e tem o valor 4.

Também é possível utilizar o nome das componentes entre aspas, como por exemplo **lst[["nome"]]**, que coincide com **lst$nome**. Esta opção é muito útil no caso em que o nome das componentes se guarda noutra variável, tal como:

```
>  x <- "nome" ; lst[[x]]
```

[1] "José"

É muito importante distinguir entre lst[[1]] e lst[1]. O operador '[[ **...** ]]' é usado para seleccionar uma só componente de uma lista, enquanto que '[ **...** ]' é o operador genérico para variáveis indexadas. Isto é, lst[[4]] é o quarto objecto da lista lst, e se é uma lista com nomes, o nome não está incluído. Por outro lado, lst[4] é uma sub-lista da lista lst, constituída pela sua quarta componente; se a lista tem nome, este passa também para a sub-lista:

```
> lst[[4]]
```

[1] 4 7 9

```
> lst[4]
```

$idade.filhos

[1] 4 7 9

Os nomes das componentes podem abreviar-se até ao mínimo de caracteres necessários para identificá-las de modo exacto é único, sem possibilidade de confusão. Assim:

```
> lista <- list(coeficientes=c(1.3, 4), covariancia=0.87)
```

**lista$coeficientes** pode especificar-se apenas por **lista$coe**, bem como **lista$covariancia** pode resumir-se a **lista$cov**. O vector de nomes é um atributo da lista, e como os restantes atributos, pode ser manipulado com a função ***names()***, que também pode ser usado sobre outros objectos:

```
> names(lst) <- c("NOME","ESPOSA","FILHOS","IDADEF")
> lst
```

$NOME

[1] "José"

$ESPOSA

[1] "Maria"

$FILHOS

[1] 3

```
$IDADEF
[1] 4 7 9
```

## 6.2 Construção e modificação de listas

A função ***list()*** permite criar listas a partir de objectos já existentes. Uma assignação da forma :

```
> lista <- list(nome_1=objecto_1, ... , nome_n=objecto_n)
```

guarda em lista uma lista de n componentes que são objecto_1, ... , objecto_n, aos quais são atribuídos os nomes nome_1, ... , nome_n, que podem ser quaisquer. Se os nomes são omitidos, as componentes apenas ficam numeradas. Os objectos existentes usados para construir uma lista são copiados para a nova lista e os originais não são modificados.

As listas, tal como todos os objectos indexados, podem ampliar-se especificando componentes adicionais. Por exemplo:

```
> lst[[5]] <- list(nacionalidade="Portuguesa")
```

acrescenta um objecto chamado 'nacionalidade' à lista lst atrás usada.

### 6.2.1 Concatenação de listas

Se se especificam listas como argumentos da função ***c()***, o resultado é um objecto cujo modo é list (isto é, é uma lista) cujas componentes são todas as listas indicadas em argumento, unidas sequencialmente.

```
> lista.ABC <- c(lista.A, lista.B, lista.C)
```

Recorde que quando os argumentos são vectores, a função ***c()*** une-os a todos num único vector. Neste caso, os restantes atributos, tal como *dim*, são perdidos.

## 6.3 Folhas de dados

Uma *folha de dados*[1] é uma lista de classe "*data.frame*". Há algumas restrições sobre que listas podem pertencer a esta classe, nomeadamente:

- As componentes devem ser vectores (numéricos, alfanuméricos ou lógicos), factores, matrizes numéricas, listas ou outras folhas de dados.
- As matrizes, listas e folhas de dados contribuem para a nova folha de dados com tantas novas variáveis quantas as colunas, elementos ou variáveis que contenham, respectivamente.
- Os vectores numéricos e factores são incluídos sem quaisquer modificações; vectores não numéricos (alfanuméricos ou lógicos) são transformados em factores, cujas categorias são valores únicos contidos no vector.

[1] Tradução livre da expressão 'data frame', por analogia com uma folha de cálculo.

- Os vectores que constituem a folha de dados devem ter o mesmo comprimento, e as matrizes devem ter o mesmo tamanho em linha.

Uma folha de dados pode ser encarada, em muitos sentidos, como uma matriz cujas colunas podem ter modos e atributos distintos. Podem visualizar-se em forma de matriz, e as suas linhas e colunas podem seleccionar-se usando as convenções das variáveis indexadas.

### 6.3.1 Criação de uma folha de dados

Os objectos que satisfaçam as restrições impostas às colunas podem agregar-se numa folha de dados usando a função ***data.frame()***:

```
> contab <- data.frame(domicilio=fprovincia,
                       rendimento=renda, classe=frenda)
```

Uma lista cujas componentes cumpram as restrições impostas pode ser transformada em folha de dados com a função ***as.data.frame()***.

A maneira mais fácil de construir uma folha de dados é usar a função ***read.table()*** para importar um ficheiro externo a R. Este assunto é abordado no Capítulo 7 [Importação de ficheiros], pág. 38.

### 6.3.2 Funções attach() e detach()

A notação **$** usada com listas, como opor exemplo **contab$domicilio**, nem sempre é a mais conveniente. Por vezes é vantajoso poder referir cada componente de uma lista ou folha de dados como se se tratasse de uma variável, com o nome que tem, sem ter necessidade de explicitamente indicar o nome da lista ou folha de dados.

Para tal, usa-se a função ***attach()***, tendo como argumento o nome de uma lista ou de uma folha de dados, de modo a permitir aceder directamente às suas componentes sem explicitar o nome da lista. Suponhamos que *lentilhas* é uma folha de dados, com três colunas (ou variáveis), designadas por *lentilhas$u*, *lentilhas$v* e *lentilhas$w*. O comando:

```
> attach(lentilhas)
```

conecta os nomes das variáveis ao caminho de busca, de modo que, caso não haja outros objectos com os mesmo nome, as variáveis contidas na folha de dados passam a poder referir-se com os nomes u, v e w. Entretanto, se fizer o comando:

```
> u <- v + w
```

não se substitui a variável u da folha de dados pela soma das outras duas variáveis; é criada uma nova variável, com o nome u, com prioridade sobre a variável lentilhas$u no caminho de busca. Se o pretendido fosse mesmo atribuir a soma à variável u da folha de dados, dever-se-ia fazer:

```
> lentilhas$u <- v + w
```

Porém, esta variável recém-assignada não é visível enquanto não se proceder à libertação e posterior ligação das variáveis da folha ao caminho de busca.

Para desagregar uma folha de dados do caminho de busca, faz-se o comando:

```
> detach(lentilhas)
```

Uma vez realizada esta função, deixarão de existir as variáveis u, v, w como tal, embora continuem a existir e estar disponíveis como componentes da folha de dados lentilhas. Como argumentos das funções ***attach()*** e ***detach()*** podem indicar-se, não os nomes das listas, mas a ordem que ocupam no caminho de busca, embora menos claro e propenso a erros, de modo que se aconselha o uso dos nomes.

> **Nota**: A actual versão de R permite definir até 20 itens no caminho de busca, pelo que deve evitar ligar um objecto mais que uma vez. Desligue os objectos do caminho de busca sempre que já não necessitar de ter as suas componentes directamente acessíveis. Não é possível assignar valores a listas ou folhas de dados que estejam ligados com ***attach()*** ao caminho de busca (são, de certa forma, estáticas).

### 6.3.3 Trabalhar com folhas de dados

Uma metodologia que permite tratar diferentes problemas utilizando o mesmo directório de trabalho é a seguinte:

- Reuna todas as variáveis de um mesmo problema numa mesma folha de dados, e dê-lhe um nome sugestivo.
- Para tratar um determinado problema conecte, com a função ***attach()***, a folha de dados correspondente (que fica com prioridade 2 no caminho de buscas) e utilize o directório de trabalho para as variáveis temporárias (cuja prioridade de busca será 1).
- Antes de terminar uma análise, assigne as variáveis temporárias que deseja conservar à folha de dados, utilizando a forma *folha_dados$nome_variavel*, e desconecte a folha de dados com ***detach()***.
- Para finalizar, elimine, com o comando ***rm()***, do directório de trabalho as variáveis temporárias que não deseje conservar, de modo a mantê-lo o mais limpo e desocupado possível.

Deste modo poderá utilizar o mesmo directório para analisar diferentes problemas, podendo acontecer, sem perigo de confusão, que haja variáveis com os mesmos nomes em diversas folhas de dados.

### 6.3.4 Conecção de objectos variados

A função ***attach()*** é uma função genérica que permite ligar ao caminho de busca não apenas directórios e folhas de dados, mas também outros tipos de objectos, tais como listas:

```
> attach(nome_lista)
```

Posteriormente poderá desligar os objectos com a função ***detach()***, utilizando como argumento o respectivo número de posição no caminho de busca, ou preferivelmente o nome do objecto a desligar.

### 6.3.5 Gestão do caminho de busca

A função ***search()*** indica a trajectória de busca actual, sendo a melhor maneira de saber quais são as folhas de dados, listas ou bibliotecas que foram conectadas ou desconectadas. Se não realizou nenhuma conecção ou desconecção, o seu valor é:

```
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

onde ".GlobalEnv", que ocupa a posição 1 do caminho de busca, corresponde ao espaço de trabalho[2].

Uma vez ligada a folha de dados lentilhas teríamos:

```
> search()
[1] ".GlobalEnv"   "lentilhas"    "Autoloads"    "package:base"
> ls(2)
[1] "u" "v" "w"
```

e, como se vê, pode usar-se ls(valor) para saber quais as variáveis ligadas na posição valor do caminho de busca (no caso, 2 corresponde à folha de dados lentilhas).

Finalmente, desconecta-se a ligação de busca à folha de dados, e confirma-se que foi removida do caminho de busca:

```
> detach(lentilhas)
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

[2] Consulte a ajuda sobre autoload para a descrição deste termo.

# 7 Importação de ficheiros externos

Se a quantidade de dados a introduzir é extensa, é mais prático proceder à importação destes dados a partir de um ficheiro externo, em vez de os introduzir directamente a partir do teclado. Em R, as capacidades de leitura de ficheiros externos são simples, e os requisitos a que estes devem obedecer são restritas e inflexíveis. Pressupõe-se que o utilizador está habilitado a editar e modificar os arquivos de dados com outras aplicações, tais como editores de texto[1] e folhas de cálculo, de modo a ajustá-los às exigências de R. Geralmente estas tarefas são extremamente simples.

A função ***read.fwf()*** pode usar-se para ler arquivos com campos de largura fixa não delimitados por separador (esta função utiliza uma rotina perl que converte o ficheiro num cuja estrutura está adaptada à leitura com ***read.table()***). A função ***count.fields()*** conta o número de campos por linha de um ficheiro com campos delimitados. Estas duas funções podem resolver alguns problemas de importação de ficheiros, mas de um modo geral é mais aconselhável ajustar o ficheiro de dados aos requisitos de R antes de começar a sessão de trabalho.

Se os dados lidos vão ser guardados em variáveis numa folha de dados, como se recomenda, podem ler-se os dados directamente com a função ***read.table()***. Dispõe-se também da função ***scan()***, mais genérica, e que pode ser usada directamente.

## 7.1 A função read.table()

Para ler na íntegra uma folha de dados directamente, o ficheiro externo deve reunir os requisitos necessários:

- A primeira linha do arquivo deve conter os nomes dos campos ou variáveis.
- Em cada uma das linhas seguintes, o primeiro elemento é a etiqueta da linha, seguido dos valores das restantes variáveis.

Esta disposição pressupõe que a primeira linha tem menos um dado que as restantes (a coluna das etiquetas das linhas não tem nome). Apresenta-se de seguida um exemplo das primeiras linhas de um ficheiro (casas.dat, supostamente localizado no directório de trabalho) adaptado à sua importação com a função ***read.table()***.

Estrutura do ficheiro externo com nomes das variáveis e etiquetas de linha

| | Preco | Superficie | Area | Divisoes | Anos | Calef |
|---|---|---|---|---|---|---|
| 01 | 52.00 | 111.0 | 830 | 5 | 6.2 | não |
| 02 | 54.75 | 128.0 | 710 | 5 | 7.5 | não |
| 03 | 57.50 | 101.0 | 1000 | 5 | 4.2 | não |
| 04 | 57.50 | 131.0 | 690 | 6 | 8.8 | não |
| 05 | 59.75 | 93.0 | 900 | 5 | 1.9 | sim |
| ... | ... | ... | ... | ... | ... | ... |

[1] Em ambiente UNIX pode usar as aplicações sed ou awk.

Por defeito, os campos numéricos (excepto as etiquetas de linha) são lidos como variáveis numéricas, e campos não-numéricos (tal como Calef) são lidos como factores. Esta regra pode alterar-se, caso seja necessário.

A função ***read.table()*** importa este arquivo directamente para uma folha de dados:

```
> PrecoCasas <- read.table("casas.dat")
```

Frequentemente omitem-se as etiquetas de linha no ficheiro externo, e usam-se as etiquetas que, por defeito, R assume. Neste caso, o arquivo de dados deve ter a seguinte estrutura:

Estrutura do ficheiro externo com nomes das variáveis e sem etiquetas de linha

| Preco | Superficie | Area | Divisoes | Anos | Calef |
|---|---|---|---|---|---|
| 52.00 | 111.0 | 830 | 5 | 6.2 | não |
| 54.75 | 128.0 | 710 | 5 | 7.5 | não |
| 57.50 | 101.0 | 1000 | 5 | 4.2 | não |
| 57.50 | 131.0 | 690 | 6 | 8.8 | não |
| 59.75 | 93.0 | 900 | 5 | 1.9 | sim |
| ... | ... | ... | ... | ... | ... |

e será importado com comando:

```
> PrecoCasas <- read.table("casas.dat", header=T)
```

onde o parâmetro adicional ***header=TRUE*** (ou simplesmente ***header=T***) indica que a primeira linha contém os nomes das variáveis e não existem etiquetas de linha.

## 7.2 A função scan()

Suponhamos que pretendemos importar o ficheiro input.dat, supostamente localizado no directório corrente, que contém em colunas os dados correspondentes a três vectores, todos com o mesmo comprimento, sendo o primeiro alfanumérico e os outros dois numéricos. O primeiro passo consiste em ler os três vectores do ficheiro, com a função ***scan()***:

```
> entrada <- scan("input.dat", list(" ",0,0))
```

O segundo argumento usado na função é uma estrutura de controlo, destinada a definir o modo como os três vectores serão lidos. O resultado é guardado na lista designada com o nome entrada, cujas componentes são os três vectores. De seguida, podemos referir-nos a cada uma destas componentes, assignando-lhes nomes:

```
> etiqueta <- entrada[[1]]; x <- entrada[[2]]; y <- entrada[[3]]
```

Esta assignação de nomes às componentes da lista poderia ter sido executada ao importar os dados:

```
> entrada <- scan("input.dat", list(etiqueta=" ", x=0, y=0))
```

Pretendendo aceder directamente às variáveis, podem re-assignar-se os nomes:

```
> etiqueta <- entrada$etiqueta; x <- entrada$x; y <- entrada$y
```

ou usar a função attach() sobre a lista entrada a fim de ligar as variáveis na posição 2 do caminho de busca (veja Secção 6.3.4 [Conecção de objectos variados], pág. 36).

Se todas as componentes do arquivo a importar são do mesmo modo (numéricas, alfanuméricas, etc.), pode usar-se como segundo argumento na função scan() um único valor (de modo idêntico às componentes) e não uma lista:

```
> X <- matrix(scan("light.dat",0), ncol=5, byrow=TRUE)
```

A função scan() permite realizar importações mais complexas, como pode consultar na ajuda.

## 7.3 Acesso a dados internos

Conjuntamente com R são fornecidos mais de cinquenta conjuntos de dados, e outros mais estão disponíveis nas bibliotecas (incluindo as bibliotecas standard que acompanham o programa). Para poder utilizar estes dados, têm de carregar-se explicitamente, usando a função ***data()***. Para obter a lista das conjuntos de dados existentes, use o comando:

```
> data()
```

e para carregar um desses conjuntos de dados, deve indicar-se o nome como argumento da função:

```
> data(infert)
```

Normalmente um comando deste tipo carrega um objecto com o mesmo nome, que deve ser uma folha de dados. Em determinados casos, pode acontecer que sejam carregados vários objectos, pelo que numa tal situação deverá consultar a ajuda disponível sobre o objecto em concreto para saber qual será o resultado do comando.

### 7.3.1 Acesso a dados de uma biblioteca

Para aceder aos dados incluídos numa biblioteca, basta especificar o nome da biblioteca em argumento da função ***data()***. Por exemplo:

```
> data(package="nls")
> data(Puromycin, package="nls")
```

Se uma biblioteca foi ligado pela função ***library()***, os conjuntos de dados nela contidos foram automaticamente incluídos no caminho de busca, e não é necessário usar o argumento *package*. A seguinte sequência de comandos:

```
> library(nls)
> data()
> data(Puromycin)
```

liga a biblioteca ***nls***, apresenta uma listagem de todos os conjuntos de dados ligados actualmente (pelo menos, as bibliotecas ***base*** e ***nls***) e carrega conjunto de dados Puromycin da biblioteca ***nls*** (ou da primeira biblioteca que contenha um o conjunto de dados com este nome).

As bibliotecas criadas pelos utilizadores de R são uma valiosa fonte de dados. As notas do Dr. Venables, origem desta introdução, contêm um conjunto de dados disponível em CRAN na biblioteca ***Rnotes***.

## 7.4 Edição de dados

Uma vez carregada uma estrutura de dados, a função ***data.entry()***, disponível nalgumas versões de R, permite modificá-la. A ordem:

```
> xnovo <- data.entry(xvelho)
```

edita xvelho usando um ambiente similar a uma folha de cálculo. Ao finalizar, o resultado é guardado em xnovo. Os objectos xvelho, e consequentemente xnovo, podem ser matrizes , vectores, folhas de dados ou objectos atómicos.

Se se utiliza a função sem argumentos:

```
> xnovo <- data.entry()
```

abre uma folha vazia, permitindo a introdução de dados[2].

## 7.5 Importação de dados

Em muitos casos pode ser necessário importar os dados a partir de bases de dados, ou genericamente, desde ficheiros criados com outros programas.. Estão a desenvolver-se diversas rotinas para a importação de dados de fontes externas a R. Presentemente existe a biblioteca ***stataread*** que lê e escreve ficheiros Stata, e uma versão experimental da biblioteca ***foreign***, planeada para permitir a leitura de ficheiros SAS, Minitab e SPSS. Outras bibliotecas permitem o acesso a bases de dados que suportam **SQL**, e está quase pronta a rotina **ROBDC** para aceder a bases de dados **ODBC** (tais como Access e Microsoft Windows).

---

[2] Em ambiente Windows pode usar o editor notepad:

> edit()

e ao sair, guardar os dados introduzidos num ficheiro que depois poderá importar com ***read.table()***.

# 8 Distribuições de probabilidades

## 8.1 Funções estatísticas

R dispõe de um amplio conjunto de tabelas estatísticas. Para cada uma das distribuições suportadas, dispõem-se de funções que permitem calcular a função de distribuição, $F(x) = \Pr(X \leq x)$, a função de distribuição inversa, a função densidade e a geração de números pseudo-aleatórios. As distribuições disponíveis são as seguintes:

| Distribuição | nome da função | Argumentos adicionais |
|---|---|---|
| beta | beta | shape1, shape2, ncp |
| binomial | binom | size, prob |
| Cauchy | cauchy | location, scale |
| qui-quadrado | chisq | df, ncp |
| exponencial | exp | rate |
| F de Snedecor | f | df1, df2, ncp |
| gamma | gamma | shape, scale |
| geométrica | geom | prob |
| hipergeométrica | hyper | m, n, k |
| log-normal | lnorm | meanlog, sdlog |
| logística | logis | location, scale |
| binomial negativa | nbinom | size, prob |
| normal | norm | mean, sd |
| Poisson | pois | lambda |
| t de Student | t | df, ncp |
| uniforme | unif | min, max |
| Weibull | weibull | shape, scale |
| Wilcoxon | wilcox | m, n |

Para construir o nome de cada função, utilize o nome da função, precedido do prefixo "**d**" para a função densidade, "**p**" para a função de distribuição, "**q**" para a função de distribuição inversa, e "**r**" para a função geradora de números pseudo-aleatórios. O primeiro argumento é x para a função de densidade, q para a função de distribuição, p para a função de distribuição inversa e n para a função geradora de números pseudo-aleatórios (excepto no caso de ***rhyper()*** e ***rwilcox()***, para as quais é **nn**). No momento em que este manual foi elaborado, o parâmetro de não-centralidade **ncp** apenas está disponível para as funções de distribuição e algumas outras funções. Consulte os tópicos de ajuda para cada um dos casos.

As funções ***pxxx()*** (função de distribuição) e ***qxxx()*** (função de distribuição inversa) têm os argumentos lógicos **lower.tail** e **log.p**; as funções ***dxxx()*** (função densidade) têm o argumento

lógico **log**. O uso destes argumentos permite, por exemplo, obter a função 'hazard' cumulativa, $H(t) = -\log\left(1 - F(t)\right)$, através da expressão:

```
-pxxx(t, ..., lower.tail = FALSE, log.p = TRUE)
```

ou mais correctamente, as funções log-verosimilhança (fazendo `dxxx(... , log = TRUE)` directamente.

Além das funções anteriores, estão também disponíveis as funções ***ptukey()*** e ***qtuckey()*** para a distribuição do "studentized range" de uma amostra proveniente da distribuição normal.

Os seguintes exemplos exemplificam algumas destas funções:

```
> ## Valor da probabilidade das caudas (p-value) da distribuição t-
  Student
> 2*pt(-2.43, df=13)
[1] 0.0303309
> ## Percentil 1% superior de uma distribuição F(2,7)
> qf(0.99,2,7)
[1] 9.546578
```

## 8.2 Análise da distribuição de uma amostra de dados

Dada uma amostra uni-dimensional de dados, pode abordar-se o estudo da sua distribuição de diversas maneiras. A mais fácil consiste em calcular um resumo estatístico, com qualquer das funções ***summary()*** ou ***fivenum()***; também se pode construir um diagrama de caule-e-folhas com a função ***stem()***:

```
> data(faithful)
> attach(faithful)
> summary(eruptions)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.600   2.163   4.000   3.488   4.454   5.100
> fivenum(eruptions)
[1] 1.6000 2.1585 4.0000 4.4585 5.1000
> stem(eruptions)

  The decimal point is 1 digit(s) to the left of the |

  16 | 070355555588
  18 | 000022233333335577777777888822335777888
  20 | 00002223378800035778
  22 | 0002335578023578
  24 | 00228
  26 | 23
  28 | 080
  30 | 7
  32 | 2337
  34 | 250077
```

36 | 0000823577

38 | 2333335582225577

40 | 000000335778888800233555577778

42 | 033355557788002333335555777778

44 | 022223355577800000000233333577788888

46 | 0000233357700000023578

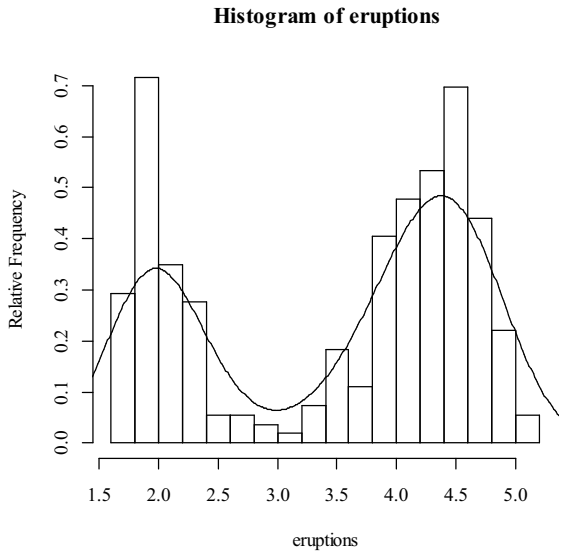
48 | 00000022335800333

50 | 0370

Em vez do diagrama de caule-e-folhas, pode construir-se um histograma com a função ***hist()***:

```
> hist(eruptions)
> ## Definir os intervalos menores, e sobrepor a função de densidade
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE)
> lines(density(eruptions, bw=0.1))
> rug(eruptions) # Mostra os pontos
```

A função ***density()*** permite realizar gráficos da função de densidade, e utilizámo-la para sobrepor este gráfico ao histograma previamente construído. O factor de suavização, **bw**, foi

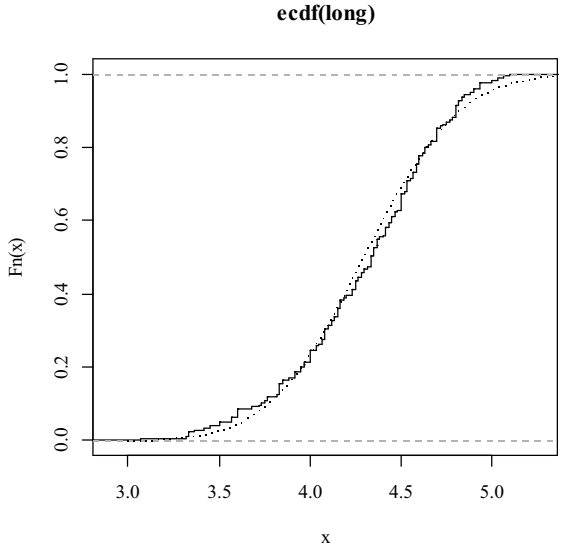



seleccionada por tentativas, de modo a que gráfico resulte mais elucidativo, pois o seu valor por defeito apresenta uma linha de densidades extremamente suavizada (as bibliotecas **Mass** e **KernSmooth** têm disponíveis métodos automáticos para seleccionar o factor de suavização).

A função ***ecdf()***, disponível na biblioteca standard **stepfun**, permite representar a função de distribuição cumulativa empírica:

```
> # Carregar a biblioteca stepfun
> library(stepfun)
```

```
> plot(ecdf(eruptions), do.poits=FALSE, verticals=TRUE)
```

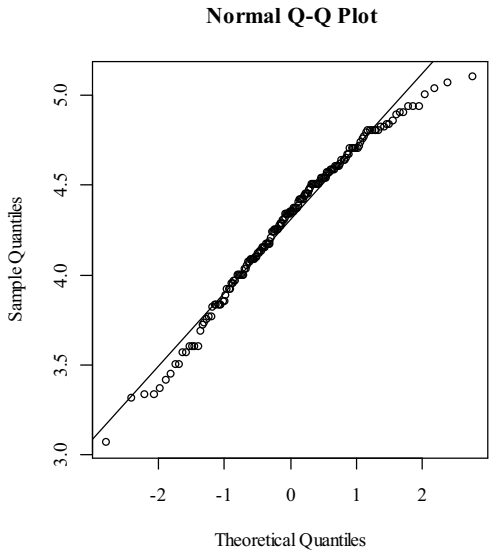



Esta distribuição, obviamente, não se parece com nenhuma das distribuições standard. Porém podemos analisar o que se passa com as erupções com mais de 3 minutos: Vamos seleccioná-las, e ajustar-lhes uma distribuição normal.

```
> long <- eruptions[eruptions > 3]
> plot(ecdf(long), do.points=FALSE, verticals=TRUE)
> x <- seq(3, 5.4, 0.01)
> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)
```

Os gráficos **Q-Q** (quantil-quantil) podem ser úteis para analisar os dados mais cuidadosamente:

```
> par(pty="s")
> qqnorm(long); qqline(long)
```



que mostra um ajustamento razoável, embora com a cauda direita mais curta do que teoricamente se deveria esperar numa distribuição normal. Vamos compará-la com uma amostra de dados pseudo-aleatórios gerados a partir de uma distribuição t-Student com 5 graus de liberdade ($t_5$ ) e com o mesmo tamanho amostral, n=250:

```
> x<-rt(250,df=5)
> qqnorm(x); qqline(x)
```

que na maioria das vezes (recorde que é uma amostra pseudo-aleatória) terá caudas mais longas do que teoricamente seria de esperar numa distribuição normal. Podemos realizar um gráfico Q-Q destes dados contra uma distribuição $t_5$, mediante:

```
> qqplot(qt(ppoints(250), df=5), x, xlab="Gráfico Q-Q de t_5")
> qqline(x)
```

Por último, realizemos um contraste de hipóteses para comprovar a normalidade. A biblioteca **ctest** contém uma rotina para realizar o teste de Shapiro-Wilk:

```
> library(ctest)
> shapiro.test(long)
```

Shapiro-Wilk normality test

data: long

W = 0.9793, p-value = 0.01052

e o teste de Kolmogorov-Smirnov:

```
> ks.test(long, "pnorm", mean=mean(long), sd=sqrt(var(long)))
```

One-sample Kolmogorov-Smirnov test

data: long

D = 0.0661, p-value = 0.4284

alternative hypothesis: two.sided

(Note que a teoria de distribuição não é válida neste teste, já que estimámos os parâmetros da distribuição normal a partir da mesma amostra).

## 8.3 Contrastes de duas amostras

Até agora, limitámo-nos a ajustar uma amostra à distribuição normal. Uma metodologia estatística mais comum é comparar duas amostras. Considere as seguintes duas amostras, obtidas em Rice (1995, pág. 490), referentes ao calor latente na fusão de gelo em cal/gm:

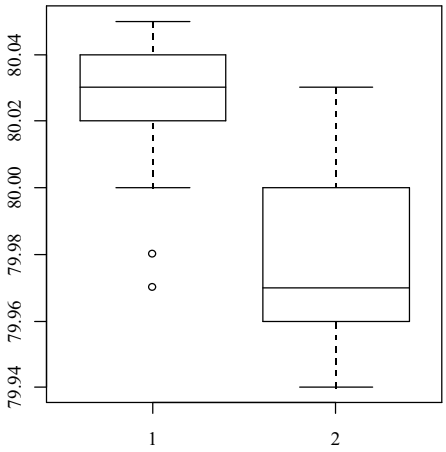
Método A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02

Método B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97

Após introduzir os dados, podemos comparar graficamente as duas amostras, mediante um diagrama de extremos-e-quartis:

```
> A <- scan()
```

```
1: 79.98  80.04  80.02  80.04  80.03  80.03  80.04  79.97
9: 80.05  80.03  80.02  80.00  80.02
Read 13 items
> B <- scan()
1: 80.02  79.94  79.98  79.97  79.97  80.03  79.95  79.97
Read 8 items
> boxplot(A,B)
```




que mostra claramente que o método A tem tendência a dar valores médios mais elevados:

Para contrastar a igualdade de médias das duas populações de onde se retiraram as amostras, usa-se um contraste de hipóteses t-Student para duas amostras independentes:

```
> t.test(A,B)
        Welch Two Sample t-test
data:  A and B
t = 3.2499, df = 12.027, p-value = 0.00694
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01385526 0.07018320
sample estimates:
mean of x mean of y
 80.02077  79.97875
```

que revela uma diferença significativa entre as duas médias, sob o pressuposto da independência e normalidade das amostras. A função ***t.test()*** de R não pressupõe a igualdade das variâncias

(em contraste com a equivalente função t.test do programa S-PLUS). Podemos testar a igualdade de variâncias usando a função ***var.test()*** disponível na biblioteca **ctest**:

```
> library(ctest)     # Não é necessário, se previamente carregada
> var.test(A, B)
```

F test to compare two variances

data: A and B

F = 0.5837, num df = 12, denom df = 7, p-value = 0.3938

alternative hypothesis: true ratio of variances is not equal to 1

95 percent confidence interval:

0.1251097 2.1052687

sample estimates:

ratio of variances

0.5837405

que não revela evidência de que as variâncias sejam significativamente diferentes; agora podemos realizar o teste t clássico, assumindo variâncias iguais:

```
> t.test(A, B, var.equal=TRUE)
```

Two Sample t-test

data: A and B

t = 3.4722, df = 19, p-value = 0.002551

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

0.01669058 0.06734788

sample estimates:

mean of x mean of y

80.02077 79.97875

Tal como referido, uma das condições de aplicação dos testes anteriores é a normalidade das amostras. Se esta não se verifica, pode utilizar-se o teste de Wilcoxon (ou teste de Mann-Whitney), que apenas exige que, sob o pressuposto da hipótese nula, a distribuição seja contínua. A função ***wilcox.test()*** está disponível na biblioteca **ctest**.

```
> library(ctest)     # Não é necessário, se previamente carregada
> wilcox.test(A, B)
```

Wilcoxon rank sum test with continuity correction

data: A and B

W = 89, p-value = 0.007497

alternative hypothesis: true mu is not equal to 0

Warning message:

Cannot compute exact p-value with ties in: wilcox.test(A, B)

O resultado do teste contém uma nota de advertência, sugerindo que, havendo valores repetidos em cada amostra (provavelmente devido a arredondamentos), seja provável que os dados não procedam de uma amostra contínua.

Além do diagrama de extremos-e-quartis, R dispõe de outras funções (disponíveis na biblioteca **stepfun**) que permitem comparar graficamente duas amostras. As funções:

```
> library(stepfun)   # Não é necessário, se previamente carregada
> plot(ecdf(A), do.points=FALSE, verticals=TRUE, xlim=range(A,B))
> plot(ecdf(B), do.points=FALSE, verticals=TRUE, add=TRUE)
```

representam graficamente as duas funções de distribuição empíricas; a função ***qqplot()*** constrói um gráfico Q-Q das duas amostras.

O teste de Kolmogorov-Smirnov, que apenas exige que a distribuição comum das duas amostras sejam contínuas, calcula a distância máxima entre as duas funções de distribuição:

```
> ks.test(A, B)
```

[1] -1.250000e-01 -2.500000e-01 -5.480769e-01 -5.961538e-01 -5.192308e-01

[6] -4.134615e-01 -3.076923e-01 -7.692308e-02  5.551115e-17

Two-sample Kolmogorov-Smirnov test

data:  A and B

D = 0.5962, p-value = 0.05919

alternative hypothesis: two.sided

Warning message:

cannot compute correct p-values with ties in: ks.test(A, B)

fazendo a mesma advertência do teste Wilcoxon.

# 9 Ciclos. Expressões condicionais

## 9.1 Expressões agrupadas

R é uma linguagem de expressões, no sentido que os comandos de que dispõe são funções ou expressões que devolvem um resultado. Mesmo a função de assignação é uma expressão, cujo resultado é o valor assignado e que pode utilizar-se em qualquer sítio em que se possa usar uma expressão. Em particular, é possível realizar assignações múltiplas.

As ordens podem a grupar-se entre chavetas, **{expres_1; ... ; expres_m}** sendo separadas por ponto-e-vírgula ';'. O resultado deste grupo de expressões é o resultado da última expressão do grupo que seja calculada. Como um tal grupo de expressões é também uma expressão, pode incluir-se entre parêntesis, e ser usado como parte de uma expressão maior.

## 9.2 Ordens de controlo

### 9.2.1 Execução condicional. A ordem if

A linguagem R dispõe de ordens condicionais da forma:

```
> if (expres_1) expres_2 else expres_3
```

onde o resultado de *expres_1* deve ser um valor lógico; se este é verdadeiro (T ou TRUE), é calculada a expressão *expres_2*; caso contrário, e se a ordem contém a expressão **else**, será executada a expressão *expres_3*.

Os operadores lógicos **&&** (AND ou E) e || (OR ou OU) podem utilizar-se como condições de uma expressão **if**. Enquanto que os operadores **&** e | se aplicam a todos os elementos de um vector, **&&** e || aplicam-se a vectores de comprimento unitário e só calculam o segundo argumento se é necessário, isto é, se o valor da expressão completa não se deduz do primeiro argumento.

Existe uma versão vectorizada da construção **if/then**, que é a função **ifelse**, cuja forma é `ifelse(condição, a, b)`, e cujo resultado é um vector com o comprimento do maior dos seus argumentos, e cujo i.ésimo valor é a[i] se a condição[i] é verdadeira, ou b[i] em caso contrário.

### 9.2.2 Ciclos. As ordens for, repeat, while

Os ciclos repetitivos for são da forma:

```
> for (nome in expres_1) expres_2
```

onde *nome* representa uma variável de controlo das iterações, *expres_1* é um vector (geralmente uma sequência do tipo **1:n**, em que n é um número natural), e *expres_2* é uma expressão, frequentemente agrupada, em cujas sub-expressões pode aparecer a variável de controlo; esta expressão é calculada repetidamente à medida que a variável de controlo nome percorre os valores da *expres_1*.

Por exemplo, suponhamos que ind é um vector de indicadores de classes, e pretendem construir-se gráficos de dispersão (x,y) separados por classe. Uma possibilidade é usar a função ***coplot()***, que será analisada adiante, e que produz uma matriz de gráficos correspondentes a cada nível do factor. Outra maneira de construir estes gráficos é usar uma estrutura repetitiva:

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc))
      {
       plot(xc[[i]], yc[[i]]);
       abline(lsfit(xc[[i]], yc[[i]]))
      }
```

A função ***split()*** produz uma lista de vectores dividindo um vector de acordo com as classes especificadas por um factor. Esta função é muito útil, nomeadamente quando usada conjuntamente com diagramas de extremos-e-quartis. Consulte a ajuda para mais pormenores.

**Nota**: Em R a função ***for()*** é utilizada menos frequentemente que em outras linguagens tradicionais, pois R trabalha com os objectos inteiros, tirando vantagens do uso da estrutura dos objectos.

Outras estruturas repetitivas são:

```
> repeat expressão
```

e:

```
> while (condição) expressão
```

A função ***break()*** usa-se para terminar qualquer ciclo. Esta é a única forma de quebrar um ciclo repeat (a não ser que ocorra um erro).

A função ***next()*** usa-se para deixar de executar um ciclo, e passar à expressão seguinte.

As ordens de controlo são habitualmente usadas na construção de funções, que serão tratadas no Capítulo 10 [Defina as suas próprias funções], pág. 52, onde serão abordados vários exemplos.

# 10 Defina as suas próprias funções

Como já foi referido diversas vezes até agora, R permite construir objectos do modo *function*, que constituem novas funções que se podem usar por sua vez em expressões posteriores. Neste contexto, a linguagem R ganha consideravelmente em potência, comodidade e elegância, e aprender a escrever funções úteis é uma das formas de conseguir que o uso da linguagem R seja cómodo e produtivo.

Deve realçar-se que muitas das funções que estão disponíveis em R, tais como ***mean()***, ***var()***, ***postscipt()***, estão escritas com a própria linguagem R, não diferindo substancialmente das novas funções que o utilizador possa criar.

Para definir uma função deve fazer-se uma assignação da forma:

```
> Nome_da_Função <- function(arg_1, arg_2, ...) expressão
```

onde expressão significa a expressão de R (geralmente uma expressão agrupada), que usa os argumentos *arg_i*, para calcular um valor. O valor dessa expressão é o valor devolvido pela função.

O uso da função é geralmente da forma `Nome_da_Função(arg_1, arg_2, ...)` e pode utilizar-se em qualquer altura que o uso de uma função seja correcto.

## 10.1 Alguns exemplos simples

Como um primeiro exemplo, considere a função para calcular a estatística t-Student de duas amostras, mostrando todos os passos. Este exemplo é muito artificial, já que existem, tal como vimos, outros modos mais simples de obter o mesmo resultado.

A função (chamemos-lhe *teste.t*) pode ser definida da seguinte forma:

```
> teste.t <- function(y1, y2)
            {
              n1 <- length(y1); n2 <- length(y2)
              yb1 <- mean(y1);  yb2 <- mean(y2)
              s1 <- var(y1);    s2 <- var(y2)
              s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
              tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
              tst
            }
```

Uma vez escrita esta função, pode usar-se para realizar um contraste de t-Student para as médias de duas amostras (para exemplificar, usemos os dados dos vectores A e B, definidos na Secção 8.3 [Contrastes de duas amostras], pág. 52), sendo usada da seguinte forma:

```
> teste.t(A, B)
```

[1] 3.472245

Como um segundo exemplo, pretende-se escrever uma função para emular directamente a função "backslash" do programa MATLAB, que calcula os coeficientes da projecção ortogonal

do vector y sobre o espaço das colunas da matriz X (isto é, os geralmente designados estimadores de mínimos quadrados dos coeficientes de regressão). Esta estimativa é normalmente executada com a função ***qr()***; porém, dada a sua complexidade, é mais cómodo dispor de uma função com a seguinte forma, que permita usá-la directamente.

Dados um vector $y_{n\times 1}$ e uma matriz $X_{n\times p}$, então define-se $X \setminus y = (X'X)^{-1}X'y$ onde $(X'X)^{-1}$ é a matriz inversa generalizada de $(X'X)$.

A função ***backslash()*** pode ser definida da seguinte forma:

```
> backslash <- function(X, y)
                {
                   X <- qr(X)
                   qr.coef.(X,y)
                }
```

Uma vez escrita a função, pode usar-se sobre a matriz M e o vector v:

```
> CoefReg <- backslash(M, v)
```

A função ***lsfit()*** realiza este cálculo, e muito mais[1]. Também utiliza as funções ***qr()*** e ***qr.coef()*** na mesma forma e para os mesmos cálculos que são usadas na função atrás escrita. Contudo, pode ser vantajoso ter esta parte isolada, se se pretendem calcular frequentemente os coeficientes de regressão isoladamente dos restantes resultados obtidos com ***lsfit()***. Nesta linha de ideias, seria igualmente cómodo ter um operador binário matricial.

## 10.2 Definição de um operador binário

Se houvéssemos dado à função ***backslash()*** um nome delimitado pelos caracteres **%**, com a forma:

```
%nome%
```

este poderia utilizar-se como um *operador binário*, em vez de usar-se na sua forma funcional. Suponhamos que usamos o caracter **!** para o nome a dar a este operador, escrevendo-o entre os símbolos de percentagem. A definição da função deve começar da seguinte forma:

```
> "%!%" <- function(X, y)
        {
        ... ...
        }
```

Chamamos a atenção para o facto de se escrever o nome entre aspas, pois os caracteres **%** são especiais. Uma vez definida a função, a sua utilização é da forma `X %!% y`.

Os operadores de produto matricial, **%*%**, e produto externo, **%o%**, são exemplos de operadores binários definidos desta forma.

---

[1] Veja também os métodos descritos no Capítulo 11 [Modelos estatísticos em R], pág. 63

## 10.3 Argumentos com nome. Argumentos pré-determinados

Já anteriormente vimos (Secção 2.3 [Gerar sequências regulares], pág. 10) que quando os argumentos são definidos por nome, da forma "*argumento=valor*", a ordem em que são introduzidos nas funções é irrelevante. Aliás, podem usar-se simultaneamente as duas formas de especificar os argumentos: podem começar-se a especificar os argumentos pela ordem específica, e de seguida especificar outros argumentos pelo nome.

Assim, se a função ***fun1()*** está definida como:

```
> fun1 <- function(dados, folha.dados, graph, limite)
      {
      (Aqui são escritas as expressões que constituem a função)
      }
```

as seguintes chamadas da função são equivalentes:

```
> resultado <- fun1(d, fd, T, 20)
> resultado <- fun1(d, fd, graph=TRUE, limite=20)
> resultado <- fun1(dados=d, limite=20, graph=T, folha.dados=fd)
```

Em determinados casos pode definir-se um valor pré-determinado para alguns dos argumentos de uma função; neste caso, ao executar-se essa função, este argumento pode omitir-se se o valor pré-determinado é o valor apropriado. Por exemplo, se a função ***fun1()*** estivesse definida como:

```
> fun1 <- function(dados, folha.dados, graph=TRUE, limite=20)
      {
      (Aqui são escritas as expressões que constituem a função)
      }
```

a execução da função pode ser da forma:

```
> resultado <- fun1(d, hd)
```

que seria equivalente às três chamadas anteriormente ilustradas. Caso haja necessidade de alterar o valor de um argumento pré-determinado, a chamada da função seria:

```
> resultado <- fun1(d, hd, limite=10)
```

É importante realçar que os valores pré-determinados podem ser expressões arbitrárias, que podem inclusivamente envolver outros argumentos da mesma função, e não é obrigatório que sejam constantes como no exemplo anterior.

## 10.4 O argumento "..."

Frequentemente há necessidade de que uma função possa passar os valores dos seus argumentos a outra função. Por exemplo, muitas funções gráficas, como ***plot()***, utilizam a função ***par()***, e permitem ao utilizador passar os parâmetros gráficos a ***par()*** a fim de controlar o resultado gráfico. (Veja Secção 12.4.1 [A função par()], pág. 86, para mais detalhes sobre a função ***par()***). Esta transferência de argumentos pode realizar-se incluindo um argumento adicional, definido por "**...**", na função, que pode ser ultrapassado. De seguida apresenta-se o esboço de um exemplo:

```
> fun1 <- function(dados, folha.dados, graph=TRUE, limite=20)
      {
      (Aqui são escritas algumas expressões que constituem a
      função)
      if (graph)
            par(pch="*", ...)
      (Aqui são escritas mais expressões da função)
      }
```

## 10.5 Assignações dentro de uma função

É fundamental ter em conta que *qualquer assignação ordinária realizada no interior de uma função é local e temporário, sendo perdido após sair da função.* Potanto, as assignações do tipo `X <- qr(X)`, tal como foi feito na definição da função ***backslash()***, não afectam o valor do argumento da função em que se utiliza; reportando-nos ainda ao exemplo da função ***backslash()***, veja-se que a expressão seguinte faz uso da matriz X, e não do resultado desta assignação `X <- qr(X)` local.

Para uma completa compreensão das regras que regem o âmbito das assignações em R, o utilizador deverá estar familiarizado com a noção de estrutura de cálculo. Este é um tópico avançado, e com relativo grau de complexidade, que não será abordado neste manual.

Se se pretendem realizar assignações globais e permanentes no interior de uma função, deve usar-se o operador de "super-assignação" **<<-** ou a função ***assign()***. Veja a ajuda sobre este tópico para mais detalhes. O operador **<<-** em R tem uma semântica diferente de S-PLUS. Este tema será abordado na Secção 10.7 [Âmbito], pág. 58.

## 10.6 Alguns exemplos mais complexos

### 10.6.1 Eficiência dos factores num desenho em blocos

Analisemos agora um exemplo mais complexo: o cálculo da eficiência dos factores num desenho em blocos (alguns aspectos deste problema foram já previamente tratados na Secção 5.3 [Uso de variáveis indexadas como índices], pág.23).

Um desenho em blocos é definido por dois factores, por exemplo factor **bloco** (**b** níveis) e factor **variedade** (**v** níveis). Sendo $R_{v\times v}$ e $K_{b\times b}$ as matrizes de repetições e de tamanho dos blocos, respectivamente, e $N_{b\times v}$ a matriz de incidência, as eficiências dos factores são definidas como sendo os valores próprios da matriz:

$$E = I_v - R^{-1/2}N'K^{-1}NR^{-1/2} = I_v - A'A$$

sendo $A = K^{-1/2}N'R^{-1/2}$. Uma função para calcular as eficiências dos factores poderia ser definida como:

```
> efbloc <- function(bloco, variedade)
      {
      bloco <- as.factor(bloco)          # uma pequena precaução
```

```
b <- length(levels(bloco))
variedade <- as.factor(variedade)   # uma pequena precaução
v <- length(levels(variedade))
K <- as.vector(table(bloco))        # elimina atributo dim
R <- as.vector(table(variedade))    # elimina atributo dim
N <- table(bloco, variedade)
A <- 1/sqrt(K)*N*rep(1/sqrt(R), rep(b ,v))
sv <- svd(A)
list(eficiencia=1-sv$d^2, cvblocos=sv$u, cvvariedades=sv$v)
}
```

Do ponto de vista numérico, é preferível trabalhar com a função de decomposição ***svd()*** em vez de com a função de autovalores.

O resultado desta função é uma lista que contém como primeira componente as eficiências dos factores, e os contrastes canónicos para o factor bloco e para o factor variedade, pois estes elementos fornecem informação adicional útil.

### 10.6.2 Eliminar os nomes ao imprimir uma variável indexada

Para visualizar grandes matrizes ou variáveis indexadas, pode ser cómodo faze-lo de forma compacta, sem os nomes das variáveis. A simples eliminação do atributo *dimnames* não é suficiente; a solução consiste em assignar cadeias vazias de texto a este atributo. Por exemplo, para visualizar a matriz X:

```
> temp <- X
> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X)))
> temp
> rm(temp)
```

O resultado desta sequência comandos pode obter-se pela definição de uma função ***sem.nomes()***, que utiliza uns pequenos truques para alcançar o mesmo objectivo. Esta função ilustra como algumas funções úteis e eficazes podem ter um código tão reduzido.

```
> sem.nomes <- function(a)
      {
      # Remove os nomes das variáveis para visualizar matrizes
      d <- list()
      l <- 0
      for (i in dim(a))
            {
            d[[l <- l+1]] <- rep(“”, i)
            }
      dimnames(a) <- d
      a
      }
```

Uma vez definida a função, para visualizar a matriz X de forma compacta, sem nomes de variáveis e etiquetas de linha, basta fazer:

```
> sem.nomes(X)
```

Esta função é particularmente útil para matrizes inteiras muito extensas, onde interessa mais tentar descobrir o padrão de comportamento que os valores em si.

### 10.6.3 Integração numérica recursiva

As funções podem ser recursivas e, inclusivamente, podem definir-se outras funções no seu interior. Note-se, contudo, que estas funções interiores, que na verdade são como que variáveis, não são disponibilizadas para outras funções, como o seriam se fossem definidas fora da definição de uma função, isto é, directamente na linha de comando.

O exemplo seguinte mostra uma forma trivial de realizar recursivamente uma integração numérica uni-dimensional. O integrando é calculado nos extremos e no centro do intervalo. Se o resultado de aplicar a regra do trapézio a um só intervalo é bastante próximo ao resultado de aplicá-la a dois intervalos, então este valor é considerado como sendo o resultado. Caso contrário, aplica-se o procedimento a cada um dos dois intervalos. O resultado é um processo de integração modificado que localiza os cálculos da função nas regiões onde é menos linear. Este método consome, contudo, grande quantidade de recursos de cálculo, e a função só é competitiva com outros algoritmos quando o integrando é difícil de calcular. O exemplo é ao mesmo tempo, um pequeno puzle de programação em R.

```
> area <- function(f, a, b, eps=1.0e-06, lim=10)
{
fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun)
   {
   ## A função fun1 só é visível dentro da função area
   d <- (a + b)/2
   h <- (b - a)/4
   fd <- f(d)
   a1 <- h * (fa + fd)
   a2 <- h * (fd + fb)
   if (abs(a0 - a1 - a2) < eps || lim==0)
   return(a1 + a2)
   else
         {
         return(fun(f, a , d, fa, fd, a1, eps, lim -1, fun) +
               fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
         }
   }
fa <- f(a)
fb <- f(b)
a0 <- ((fa + fb) * (b - a))/2
```

```
fun1( f, a, b, fa, fb, a0, eps, lim, fun1)
}
```

Para testar a função, calcule o integral da função ***sin(x)*** no intervalo [0, 1]:

```
> area(sin, 0, 1)
```

[1] 0.4596929

Experimente a definir uma função do tipo $f1 = 2 \times x^2 + 5$, e calcule $\int_0^5 2 \times x^2 + 5.dx$:

```
> f1 <- function(x)
        {
        2*x^2+5
        }
> area(f1, 0, 5)
```

[1] 108.3334

## 10.7 Âmbito

A discussão nesta sessão é mais técnica que nos restantes capítulos deste manual. Pretende-se ver em pormenor uma das maiores diferenças entre R e S-PLUS.

Os símbolos que aparecem no interior do corpo de uma função dividem-se em três classes: parâmetros formais, variáveis locais e variáveis livres. Os parâmetros formais são os que aparecem na lista de argumentos da função e os seus valores são definidos no processo de assignação dos argumentos da função aos parâmetros formais. As varáveis locais são aquelas cujos valores são calculados pala avaliação das expressões no interior das funções. As variáveis que não são parâmetros formais nem variáveis locais são as designadas variáveis livres. As variáveis livres transformam-se em variáveis locais se se lhes atribuem valores. Para aclarar estes conceitos, consideremos a seguinte função:

```
> f <- function(x)
      {
      y <- 2*x
      print(x)
      print(y)
      print(z)
      }
```

Nesta função, x é um parâmetro formal, y é uma variável local e z é uma variável livre.

Em R a ligação de um valor a uma variável livre é realizada consultando o ambiente no qual a função foi criada, que se designa por *âmbito léxico*. Definamos a função ***cubo()***:

```
> cubo <- function(n)
      {
      sq <- function() {n*n}
      n*sq()
      }
```

A variável n não é um argumento da função ***sq()***. Portanto, é uma variável livre, e utilizam-se as regras do âmbito léxico para determinar o seu valor. Em âmbito estático (como em S-PLUS) o valor é associado a uma variável global chamada **n**. Em âmbito léxico (como em R), é um parâmetro para a função ***cubo()***, pois há uma assignação activa para a variável n no momento em que se define a função ***sq()***. A diferença de avaliação entre R e S-PLUS é que S-PLUS tenta encontrar uma variável global chamada **n**, enquanto que R em tenta em primeiro lugar encontrar uma variável n no ambiente criado quando se activa a função ***cubo()***.

```
## Primeiro cálculo em S-PLUS
> cubo(2)
Error in sq(): Object "n" not found
Dumped
## Segundo cálculo em S-LUS, após assignar valor a n
> n<-3
> cubo(2)
[1] 18
## A mesma função calculada em R
> cubo(2)
[1] 8
```

O âmbito léxico pode usar-se para conceder às funções um *estado cambiante*. No exemplo seguinte ilustra-se como pode utilizar-se R para simular uma conta bancária. Uma conta bancária necessita de ter um balanço ou total, uma função para realizar depósitos, outra para retirar fundos e ainda outra para calcular o balanço. Vamos criar uma função chamada ***conta.banco()***, que contém três funções e que devolve uma lista como resultado. Quando se executa a função ***conta.banco()***, esta assume como argumento um valor numérico, total, e devolve uma lista que contém uma lista que contém as três funções internas. Já que estas estão definidas no interior de um ambiente que contém a variável total, estas têm acesso ao seu valor.

O operador de assignação especial, **<<-**, utiliza-se para mudar o valor associado à variável total. Este operador procura nos ambientes envolventes um ambiente que contenha o símbolo total, e quando encontra este objecto altera o seu valor, nesse ambiente, para o valor do lado direito da expressão de assignação. Se se alcança o ambiente de topo ou global sem encontrar o objecto total, então esta variável é criada e é-lhe atribuído o valor. Na maioria dos casos, o operador **<<-** cria uma variável global e atribui-lhe o valor do lado direito da expressão de assignação[2]. Somente nos casos em que **<<-** é utilizado numa função que é resultado de outra função é que acontecerá a situação especial atrás descrita.

```
> conta.banco <- function(total)
 {
  list(
```

[2] De certo modo, este funcionamento é semelhante ao do S-PLUS, já que neste programa o operador **<<-** cria ou assigna sempre valores a uma variável global.

```
  deposito = function(quantia)
        {
        if (quantia <=0)
        stop("Os depósitos devem ser positivos! \n")
        total <<- total + quantia
        cat("Depositado ", quantia, ". O total é ", total, "\n\n")
        },
  saque = function(quantia)
        {
        if (quantia > total)
        stop("Não tem cobertura! \n")
  total <<- total - quantia
        cat("Levantado ", quantia, ". O total é ", total, "\n\n")
        },
  balanco = function()
        {
        cat("O total é ", total, "\n\n")
        }
  )
}
```

```
> Antonio <- conta.banco(100)
> Roberto <- conta.banco(200)
> Antonio$deposito(30)
Depositado  30 . O total é  130
> Antonio$balanco()
O total é  130
> Roberto$balanco()
O total é  200
> Roberto$saque(100)
Levantado  100 . O total é  100
> Antonio$deposito(50)
Depositado  50 . O total é  180
> Antonio$balanco()
O total é  180
> Antonio$saque(500)
Error in Antonio$saque(500) : Não tem cobertura!
```

## 10.8 Personalização do ambiente

Os utilizadores de R podem personalizar o ambiente de trabalho, adaptando-o às suas necessidades, de diversos modos. Existe um ficheiro de inicialização, e cada directório pode ter o

seu próprio ficheiro de inicialização específico. Finalmente, podem usar-se as funções especiais **.First** e **.Last**.

O ficheiro de inicialização denomina-se "***Rprofile***" e encontra-se no sub-directório *library* do directório de instalação de R. As ordens contidas neste arquivo são executadas cada vez que se inicia uma sessão de R. Existe um segundo ficheiro de configuração pessoal, denominado "***.Rprofile***"[3], que pode existir em qualquer directório de trabalho. Se inicia a sessão de R a partir do directório que contém este ficheiro, as ordens nele contidas são executadas. Este arquivo permite a cada utilizador ter controlo sobre o seu espaço de trabalho, tal como permite dispor de diferentes modos de inicialização para diferentes directórios de trabalho. Se no directório a partir do qual se inicia a sessão não contém o ficheiro "***.Rprofile***", então R procurará este ficheiro no directório inicial do utilizador e, caso exista, utiliza-o.

A função **.First()** pode existir em qualquer destes dois ficheiros de configuração ou no ficheiro de imagem "***.Rdata***". Esta função é automaticamente executada no início da sessão, e pode usar-se para iniciar o ambiente de trabalho. No exemplo seguinte, a função **.First()** altera o indicativo de sistema para o símbolo **$**, e define outras características de funcionamento para a sessão de trabalho.

Em resumo, a sequência em que se executam estes ficheiros de personalização do ambiente é "***Rprofile***", "***.Rprofile***", "***.Rdata***" e por último a função "***.First***". Qualquer definição levada a cabo nos últimos ficheiros pode mascarar as definições efectuadas nos ficheiros anteriores.

```
> .First <- function()
      {
      #Altera o prompt para $
      options(prompt="$", continue="+\t")
      #Personaliza números e resultado
      options(digits=5, length=999)
      #Abre uma janela gráfica
      x11()
      #Define caracter para gráficos
      par(pch = "+")
      #Executa o ficheiro mystff.R
      source(file.path(getenv("HOME", "R", "mystuff.R")
      #Conecta a biblioteca stepfun
      library(stepfun)
      }
```

De modo análogo, existe a função ***.Last()*** que, caso esteja definida, é executada no final de cada sessão. A seguir apresenta-se um exemplo:

```
> .Last <- function()
      {
```

[3] Em UNIX este é um ficheiro escondido, pois o nome começa por ponto.

```
#Uma pequena medida de segurança
graphics.off()
# Hora de terminar!
cat(paste(system.date(), "\nAdeus\n"))
}
```

## 10.9 Classes. Funções genéricas. Orientação para objectos

A classe de um objecto determina o modo como será tratado pelas designadas funções genéricas. Dizendo de outra maneira, uma função é designada genérica se as acções que realiza sobre os seus argumentos são específicas da classe dos argumentos. Se o argumento não tem o atributo de classe, ou é de classe não contemplada especificamente para a função genérica em questão, então é executada uma acção pré-determinada.

O mecanismo do atributo de classe permite ao utilizador a possibilidade de definir e escrever funções para fins específicos. Entre outras funções específicas, citam-se as funções ***plot()*** para visualizar graficamente objectos, a função ***summary()*** para realizar uma análise descritiva, e ***anova()*** para realizar a análise de variância.

É enorme o número de funções genéricas que podem tratar uma classe de objectos de modo específico. Por exemplo, entre as funções que podem tratar de modo específico objectos da classe "*data.frame*", citam-se :

| | | | |
|---|---|---|---|
| [ | [[<- | any | as.matrix |
| [<- | model | plot | summary |

Pode obter-se uma lista completa usando a função ***methods()***:

```
> methods(class="data.frame")
```

O número de classes de objectos que uma função genérica pode manusear pode também ser grande. Por exemplo, a função ***plot()*** tem diversas variantes para classes de objectos:

| | | | |
|---|---|---|---|
| data.frame | default | density | factor |

Para obter a lista de classes que uma função pode manusear, usa-se também a função ***methods()***:

```
> methods(plot)
```

Veja na documentação mais informações sobre o tema.

# 11 Modelos estatísticos em R

Este capítulo pressupõe que o utilizador está familiarizado com a terminologia estatística, em particular com análise de modelos de regressão e análise de variância. Posteriormente, faremos umas assunções mais ambiciosas, em particular o conhecimento de modelos lineares generalizados e regressão não linear.

Os requisitos para o ajustamento de modelos estatísticos estão suficientemente bem conhecidos para permitir a construção de ferramentas genéricas de aplicação a um amplo espectro de problemas

R dispõe de um conjunto de capacidades que tornam muito simples o ajustamento de modelos estatísticos. Como foi referido no início, as saídas são mínimas pode ser necessário utilizar funções de extracção para obter resultados mais detalhados.

## 11.1 Definição de modelos estatísticos. Formulário

Um exemplo elementar de modelo estatístico é o modelo de regressão linear com erros independentes e variância constante:

$$y_i = \sum_{j=0}^{p} \beta_j x_{ij} + e_i, \quad e_i \sim NID(0, \sigma^2), \quad i = 1, ..., n$$

Em notação matricial pode escrever-se:

$$y = X\beta + e$$

onde **y** é o vector da variável resposta, **X** é *matriz do modelo* ou a *matriz do desenho*, formada pelas colunas $x_0, x_1, ..., x_p$, que são as variáveis independentes. Geralmente $x_0$ será uma coluna de 1 que define o *termo independente* ou intercepção.

### Exemplos

Antes de dar uma definição formal, alguns exemplos ajudarão a clarificar as ideias.

Suponhamos que $y, x_0, x_1, ..., x_p$ são variáveis numéricas, X é uma matriz e A, B, C, ... são factores. As fórmulas que aparecem na tabela seguinte especificam os modelos descritos na coluna direita:

| | |
|---|---|
| $y \sim x$ | |
| $y \sim 1 + x$ | Ambos definem o mesmo modelo de regressão linear de y sobre x. O temo independente está implícito no primeiro modelo e explícito no segundo. |
| $y \sim 0 + x$ | |
| $y \sim -1 + x$ | |
| $y \sim x - 1$ | Todos estes modelos definem a regressão linear de y sobre x, sem termo independente, isto é, o modelo passa pela origem das coordenadas. |

| | |
|---|---|
| $\log(y) \sim x1 + x2$ | Regressão múltipla da variável transformada log(y) sobre x1 e x2, com um termo independente implícito. |
| $y \sim poly(x1, x2)$<br>$y \sim 1 + x + I(x \,\hat{}\, 2)$ | Regressão polinomial de segundo grau de y sobre x . A primeira forma utiliza polinómios ortogonais e segunda forma utiliza potências de modo explícito |
| $y \sim X + poly(x, 2)$ | Regressão múltipla de y com um modelo matricial constituído pela matriz X e termos polinomiais em x de segundo grau. |
| $y \sim A$ | Análise de variância de entrada simples em y, com as classes definidas pelo factor A. |
| $y \sim A + x$ | Análise de co-variância de entrada simples em y, com as classes definias pelo factor A e covariável x. |
| $y \sim A * B$<br>$y \sim A + B + A : B$<br>$y \sim B\%in\%A$<br>$y \sim A / B$ | Modelo não aditivo de dois factores de y sobre A e B. Os dois primeiros modelos especificam a mesma classificação cruzada e os dois últimos especificam a mesma classificação aninhada. Em termos genéricos, os quatro modelos especificam o mesmo sub-espaço de modelos. |
| $y \sim (A + B + C) \,\hat{}\, 2$<br>$y \sim A * B * C - A : B : C$ | Modelo com três factores, que contém efeitos principais e interacções dos factores dois a dois. Ambas as expressões especificam o mesmo modelo. |
| $y \sim A * x$<br>$y \sim A / x$<br>$y \sim A / (1 + x) - 1$ | Modelos de regressão linear simples de y sobre x, separados para cada nível do factor A. A última expressão produz estimativas explícitas de tantos termos independentes e dependestes quantos os níveis de A. |
| $y \sim A * B + Error(C)$ | Delineamento com dois factores de tratamentos, A e B, e termo de erro estratificado pelos níveis do factor C. Por exemplo, um delineamento em "split-plot" completo (contendo todos os blocos e sub-blocos) definido pelo factor C. |

O operador **~** usado para definir a *fórmula* ou *expressão formal* do modelo em R. A expressão genérica, para um modelo linear ordinário é:

$$resposta \sim op\_1\ termo\_1 op\_2\ termo\_2\ op\_3\ termo\_3 ...$$

em que:

*resposta* vector ou matriz (o uma expressão que origina um vector ou uma matriz) que define as variáveis resposta em análise.

*o_i* é um operador, seja + ou -, implicando a inclusão ou exclusão de um termo no modelo (o primeiro operador é opcional).

*termo_i* pode ser:

- um vector, uma matriz, ou o valor 1;
- um factor
- uma *expressão formal* ou *fórmula* constituída por factores, vectores ou matrizes, ligados por *operadores formais*.

Em todos os casos, cada termo define um conjunto de colunas (variáveis) que podem ser incluídas ou eliminadas da matriz do modelo. O valor 1 significa um termo independente e é incluído por defeito, a não ser que explicitamente se exclua.

Os *operadores formais* são similares à notação usada por Wilkinson e Rogers nos programas GLIM e GENSTAT. Uma diferença inevitável é que o operador "**.**" é substituído por "**:**", já que o ponto é m caracter válido para nomes de objectos em R. De seguida apresenta-se um resumo da notação (baseada em Chambers & Hastie, 1992, pág. 29):

| | |
|---|---|
| $Y \sim M$ | Y é modelado por M. |
| $M\_1 + M\_2$ | Inclui M_1 e M_2. |
| $M\_1 - M\_2$ | Inclui M_1 e não inclui os termos M_2. |
| $M\_1 : M\_2$ | Produto tensorial de M_1 e M_2. Se ambos são factores, corresponde ao factor de sub-classes. |
| $M\_1 \%in\% M\_2$ | Similar a $M\_1 : M\_2$, apenas com diferente codificação. |
| $M\_1 * M\_2$ | Corresponde a $M\_1 + M\_2 + M\_1 : M\_2$. |
| $M\_1 / M\_2$ | Corresponde $M\_1 + M\_2 \%in\% M\_1$. |
| $M \wedge n$ | Todos os termos em M e respectivas interacções até à ordem n. |
| $I(M)$ | Isola M. Dentro de M todos os operadores têm o seu sentido aritmético habitual e esse termo aparece na matriz do modelo. |

Note que, dentro de parêntesis que geralmente envolvem os argumentos de uma função, todos os operadores têm o seu significado aritmético habitual. A função ***I()*** é a função identidade, utilizada somente para poder introduzir termos nas expressões, definindo-os mediante operadores aritméticos.

Em particular, quando as expressões formais especificam *colunas da matriz do modelo*, a especificação dos parâmetros é implícita. Não é este o caso em outros contextos, por exemplo na especificação de modelos não lineares.

### 11.1.1 Contrastes

É necessário ter uma ideia, ainda que geral, do modo como as expressões do modelo especificam as colunas da matriz do modelo. Esta noção é fácil se as variáveis são contínuas, em que cada uma constitui uma coluna da referida matriz. Do mesmo modo, se o modelo contém um termo independente, a matriz contém uma coluna de 1.

No caso de um factor A, com k níveis, a resposta depende de se o factor é nominal ou ordinal. No caso de um *factor nominal*, geram-se k-1 colunas correspondentes aos segundo, terceiro, etc., até ao k-ésimo nível do factor. (Portanto, a parametrização implícita consiste em contrastar a resposta do primeiro nível com cada um dos restantes níveis). No caso de *factores ordinais*, as k-1 colunas são os polinómios ortogonais sobre 1, 2, ..., k, omitindo o termo constante.

Esta situação pode parecer complicada, mas ainda há mais. Em primeiro lugar, se o termo independente é omitido num modelo que contém algum termo de tipo factor, o primeiro dos ditos termos é codificado em k colunas correspondentes aos indicadores de todos os níveis do factor. Em segundo lugar, todo este comportamento se pode alterar mediante o argumento ***contrasts*** da função ***options()***. Os valores pré-definidos são:

```
> options(contrasts=c("contr.treatment",  "contr.poly"))
```

A razão porque se referem estes valores é porque os valores pré-determinados em R são distintos de S-PLUS no caso de factores nominais, que utiliza os contrastes de Helmert. Portanto, para obter os mesmos resultados que em S-PLUS, deverá alterar estes valores pré-definidos:

```
> options(contrasts=c("contr.helmert", "contr.poly"))
```

Esta diferença é deliberada, já que entendemos que os contrastes pré-definidos de R são mais fáceis de interpretar para os principiantes.

Há ainda outras possibilidades, pois o esquema de contrastes a utilizar pode fixar-se para cada termo do modelo, utilizando as funções ***contrasts()*** e ***c()***.

Ainda não se consideraram os termos de interacção, que geram os produtos das colunas introduzidas pelos termos dos seus componentes.

Embora os detalhes sejam complicados, as fórmulas dos modelos em R geram habitualmente os modelos que um estatístico poderia esperar, desde que se preserve a marginalidade. Por exemplo, o ajustamento de um modelo com interacção, mas sem os correspondentes efeitos principais conduzirá a resultados surpreendentes, cuja interpretação se deixa para os especialistas.

## 11.2 Modelos lineares

A função elementar para ajustar modelos múltiplos ordinários é a função ***lm()***, e uma versão resumida da sua utilização é:

```
> modelo.ajustado <- lm(expressão_do_modelo,data=folha de dados)
```

Por exemplo:

```
> fm2 <- lm(y ~ x1 + x2, data = producao)
```

ajustará um modelo de regressão múltipla de y sobre x1 e x2 (com termo independente implícito). O termo *data=producao*, embora seja tecnicamente opcional, é importante e especifica que qualquer variável para a construção do modelo deve provir prioritariamente da folha de dados producao, *independentemente de que tenha ou não sido conectada ao caminho de busca com a função* ***attach()***.

## 11.3 Funções genéricas para extrair informação do modelo

O resultado da função ***lm()*** é um objecto do modelo ajustado, que consiste numa lista de resultados de classe *lm*. A informação acerca do modelo ajustado pode imprimir-se ou visualizar-se graficamente, usando funções genéricas orientadas para objectos desta classe, entre as quais se destacam[1]:

| | | | | | |
|---|---|---|---|---|---|
| add1 | coef | effects | kappa | predict | residuals |
| alias | deviance | family | labels | print | step |
| anova | drop1 | formula | plot | proj | summary |

De seguida apresenta-se uma breve descrição das mais usadas.

anova(objecto_1, objecto_2)

Compara um sub-modelo com um modelo externo e produz uma tabela de análise de variância.

coeficient(objecto) Calcula a matriz de coeficientes de regressão.

Forma reduzida: ***coef(objecto)***

deviance(objecto) Soma dos quadrados dos resíduos, ponderados se for caso disso.

formula(objecto) Obtém a fórmula do modelo.

plot(objecto) Cria quatro gráficos, mostrando os resíduos, os valores ajustados, e alguns diagnósticos.

predict(objecto, newdata=folha de dados)

O resultado é um vector ou matriz de valores preditos pelo modelo, correspondentes aos valores da folha de dados. Esta nova folha de dados usada deve conter variáveis cujas etiquetas coincidam com as da original.

print(objecto) Visualiza uma versão resumida do objecto. Geralmente o seu uso é implícito.

---

[1] Pode usar a função

```
> methods(class="lm")
```

para obter a listagem de todas as funções relacionadas.

| | |
|---|---|
| residuals(objecto) | Extrai a matriz de resíduos, ponderados se for caso disso.<br>Forma reduzida: ***resid(objecto)*** |
| step(objecto) | Selecciona um modelo adequado, adicionando ou eliminando termos e preservando hierarquias. O modelo final é o que tiver valor máximo do coeficiente AIC (Akaike's Information Criterion). |
| summary(objecto) | Visualiza um resumo estatístico completo dos resultados da análise de regressão. |

## 11.4 Análise de variância. Comparação de modelos.

A função de ajustamento de modelo ***aov(formula, data=folha de dados)*** opera no seu modo mais simples de maneira similar à função ***lm()*** e a muitas das funções genéricas listadas na Secção 11.3 [Funções genéricas para extrair informação do modelo], pág. 67 também se lhe aplicam.

Destaca-se ainda que a função ***aov()*** realiza uma análise de modelos estratificados de erros múltiplos, tais como delineamentos *"split plot"*, ou delineamentos em blocos incompletos balanceados com recuperação de informação inter-blcos. A expressão do modelo:

$$resposta \sim formula.media + Error(formula.estratos)$$

especifica um modelo multi-estrato com os termos de erro definidos pela expressão *formula.estratos*. No caso mais simples, *formula.estratos* é um factor, quando define um delineamento com dois níveis, isto é, com termos entre e dentro dos níveis do factor.

Por exemplo, se todas as variáveis independentes são factores, um modelo como o seguinte:

```
> fm <- aov(producao ~ v+n*p*k + Error(exploracao/blocos),
      data = explora)
```

seria usado tipicamente para descrever um modelo com o termo médio `v + n*p*k` e três níveis de erro, nomeadamente "entre explorações, "dentro de explorações, entre blocos" e "dentro de blocos".

### 11.4.1 Tabela da ANOVA

Refira-se que a tabela da análise de variância corresponde a uma sucessão de modelos ajustados. As somas de quadrados que aparecem na tabela anova correspondem a diminuições nas somas de quadrados residuais como resultado da inclusão de um *termo concreto* num *lugar concreto* da sucessão. Portanto a ordem de inclusão somente será irrelevante em delineamentos ortogonais.

Para delineamentos com vários estratos o procedimento consiste, em primeiro lugar, em projectar a resposta sobre os estratos do erro, mais uma vez em sequência, e de seguida ajustar o modelo a cada projecção. Para mais detalhes, consulte Chambers & Hastie (1992).

Uma alternativa mais flexível da tabela completa da anova consiste em comparar dois ou mais modelos directamente utilizando a função ***anova()***:

```
> anova(modelo_ajustado_1, modelo_ajsutado_2, ....)
```

O resultado é uma tabela de análise de variância que mostra as diferenças entre os modelos quando ajustados nesta sequência. Os modelos ajustados em comparação constituem uma sucessão hierárquica. Este resultado não fornece informação diferente da obtida pela função pré-definida, porém facilita a sua compreensão e controlo.

## 11.5 Actualização de modelos ajustados

A função ***update()*** utiliza-se frequentemente para ajustar um modelo que difere de outro ajustado previamente por alguns termos adicionados ou removidos. A sua expressão é:

```
> novo.modelo <- update(modelo.anterior, nova.formula)
```

Na expressão "*nova.formula*" utiliza-se o ponto "**.**" para referir a toda a parte correspondente ao modelo anterior. Por exemplo:

```
> fm05 <- lm(y ~x1 + x2 + x3 + x4 + x5, data=producao)
> fm06 <- lm(fm05, . ~ . + x6)
> sfm06 <- update(fm06, sqrt(.) ~ . )
```

ajusta um modelo de regressão múltipla com 5 variáveis independentes, contidas na folha de dados *producao*, de seguida ajusta um modelo adicional incluindo uma sexta variável independente, e finalmente ajusta outro modelo alternativo a este, em que a variável dependente é substituía pela sua raiz quadrada.

Note-se que, tendo definido a origem dos dados com *data=producao* no modelo inicial, esta informação é transferida automaticamente para os modelos seguintes.

O nome "**.**" Pode ser usado noutros contextos, porém com significados ligeiramente distintos. Por exemplo:

```
> fmfull <- lm(y ~ . , data=producao)
```

ajustará um modelo com variável resposta y, e *com todas as variáveis independentes existentes na folha de dados producao.*

Outras funções que permitem explorar modelos de forma sequencial são as funções ***add1()***, ***drop1()*** e ***step()***. Os nomes são sugestivos, mas deverá consultar a ajuda disponível para estas funções.

## 11.6 Modelos lineares generalizados

Os modelos lineares generalizados são uma extensão dos modelos lineares, com o objectivo de ter em consideração tanto distribuições de respostas não normais, como transformações para obter a linearidade de uma forma directa. Um modelo linear generalizado pode descrever-se de acordo com as seguintes suposições:

- Existe uma variável resposta y, e diversas variáveis causais x1, x2, ..., cujos valores influem na distribuição da variável resposta.
- As variáveis causais influenciam a distribuição de y mediante uma *função linear simples*. Esta função linear recebe o nome de *predictor linear*, e escreve-se geralmente:

$$\eta = \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$$

Isto é, $x_i$ não influencia na distribuição de y se e só se $\beta_i = 0$.

- A distribuição de y é da forma

$$f_y(y;\mu,\varphi) = \exp\left[\frac{A}{\varphi}\{y\lambda(\mu) - \gamma(\lambda(\mu))\} + \tau(y,\varphi)\right]$$

onde φ é um *parâmetro de escala* (possivelmente conhecido) que permanece constante para todas as observações, A representa um factor de ponderação *a priori* que se supõe conhecido, mas que pode variar com as observações, e μ é o valor médio de y. Portanto, supõe-se que a distribuição de y fica determinada pela sua média e possivelmente por um parâmetro de escala.

- A média, μ, é uma função invertível do predictor linear:

$$\mu = m(\eta), \qquad \eta = m^{-1}(\mu) = \ell(\mu)$$

e esta função inversa, $\ell()$, denomina-se *função de enlace*.

Estes pressupostos são suficientemente abrangentes para abarcar uma ampla classe de modelos úteis na estatística aplicada e, simultaneamente, suficientemente restritivos para o desenvolvimento de uma metodologia unificada de estimação e inferência, pelo menos assimptoticamente. Os interessados em aprofundar este tema podem consultar qualquer das obras de referência sobre o assunto, tal como McCullagh & Nelder (1989) ou Dobson (1990).

### 11.6.1 Famílias

A classe de modelos lineares generalizados que podem ser tratados em R inclui as variáveis resposta de distribuições *gaussiana* (*normal*), *binomial*, *poisson*, *gaussiana inversa* (*normal inversa*) e *gamma*, bem como os modelos de *quasi-verosimilhança* em que a distribuição da variável resposta não está explicitamente definida. Nesta última situação, deve especificar-se a *função de variância* como uma função da média; nos restantes casos, esta função está implícita na distribuição da variável resposta.

Cada distribuição da variável resposta admite diversas funções de enlace que relacionam a média com o predictor linear. De seguida apresentam-se as que automaticamente estão disponíveis:

| Nome da família | Função de enlace |
|---|---|
| binomial | logit, probit, cloglog |
| gaussian | identity |
| gama | identity, inverse, log |
| inverse.gausian | 1/mu^2 |
| poisson | identity, log, sqrt |
| qausi | logit, probit, cloglog, identity, inverse log, sqrt, 1/mu^2 |

A combinação de uma distribuição da variável resposta, uma função de enlace e outras informações que são necessárias para levar a cabo a modelização, denomina-se *família* do modelo linear generalizado.

### 11.6.2 A função glm()

Dado que a distribuição da variável resposta depende das variáveis predictoras através de uma função linear simples, pode utilizar-se o mesmo mecanismo dos modelos lineares para especificar a parte linear do modelo linear generalizado. Contudo, a família deve especificar-se de modo distinto.

A função de R que permite ajustar um modelo linear generalizado é ***glm()***, que é da seguinte forma:

```
> modelo.ajustado <- glm(formula,
      family=familia.geradora, data=folha.dados)
```

A única característica nova é o parâmetro *famlia.geradora* através do qual se descreve a família do modelo linear generalizado. É o nome de uma função que gera uma lista de funções e expressões que, juntas, definem e controlam o modelo e o processo de estimação. Embora possa parecer complicado à primeira vista, a sua utilização é muito simples.

Os nomes das funções standard geradoras de famílias que acompanham R são listadas na tabela apresentada na Secção 11.6.1 [Famílias], pág. 70, com a designação de "Nome da família". Caso haja necessidade de seleccionar uma função de enlace, deve indicar-se como um parâmetro, entre parêntesis, do nome da família. No caso da família "*quasi*", a função de variância pode especificar-se do mesmo modo.

Vejamos alguns exemplos.

#### Família gaussian

Uma expressão da forma:

```
> fm <- glm( y ~ x1 + x2, family = gaussian, data=vendas)
```

obtém o mesmo resultado de:

```
> fm <- lm( y ~ x1 + x2, data=vendas)
```

mas com menor eficiência. Note que a família gaussian não dispõe automaticamente de uma série de funções de enlace, pelo que não admite parâmetros. Se uma determinada análise necessita de usar a família gaussian com um enlace não-standard, a solução passa pelo uso da família *quasi*, como se verá posteriormente.

#### Família binomial

Consideremos o seguinte exemplo artificial (Silvey, 1970).

Os homens da ilha de Kalythos, no Mar Egeu, sofrem de uma doença ocular congénita cujos efeitos se agravam com a idade. Tomou-se uma amostra de vários homens desta ilha, de idades diferentes, cujos resultados se seguem:

| | | | | | |
|---|---|---|---|---|---|
| Idade: | 20 | 35 | 45 | 55 | 70 |
| Nº de homens: | 50 | 50 | 50 | 50 | 50 |
| Nº de cegos: | 6 | 17 | 26 | 37 | 44 |

Consideremos o problema de ajustar um modelo logístico e um modelo probit a estes dados, e estimar em cada modelo o parâmetro LD50, correspondente à idade em que a probabilidade de cegueira é se 50%.

Se y é o número de cegos com idade x, e n é o número de sujeitos estudados, ambos os modelos são da forma:

$$y \sim B(n, F(\beta_0 + \beta_1 x))$$

em que, para o modelo probit, $F(z) = \Phi(z)$ é a função de distribuição normal (0,1), e no modelo logit (modelo por defeito), $F(z) = e^z/(1+e^z)$. Em ambos os casos, LD50 é definido como:

$$LD50 = -\beta_0/\beta_1$$

isto é, o ponto em que o argumento da função de distribuição é zero.

Em primeiro lugar, vamos introduzir os dados para uma folha de dados:

```
> kalythos <- data.frame(x=c(20,35,45,55,70),
                n=rep(50,5), y=c(6,17,26,37,44))
```

Para ajustar um modelo binomial utilizando ***glm()***, existem duas possibilidades para a resposta:

- Se a resposta é um *vector*, então deve corresponder a *dados binários*, e portanto só deve conter 0 (zero) e 1 (um).
- Se a resposta é uma *matriz de duas colunas*, a primeira coluna deve conter o número de sucessos e a segunda o número de insucessos.

Vamos a usar a segunda convenção, pelo que devemos adicionar uma matriz à nossa folha de dados:

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n-kalythos$y)
```

Para ajustar os dois modelos utilizamos:

```
> fmp <- glm(Ymat ~ x, family=binomial(link=probit),data= kalythos)
> fml <- glm(Ymat ~ x, family = binomial, data = kalythos)
```

Já que a função de enlace é, por defeito, a função *logit*, este parâmetro pode ser omitido tal como se fez na segunda expressão. Para ver os resultados de cada um dos ajustamentos, faz-se:

```
> summary(fmp)
> summary(fml)
```

Ambos os modelos se ajustam (demasiado bem). Para estimar LD50 podemos definir a seguinte função:

```
> ld50 <- function(b)
        {
```

```
        -b[1]/b[2]
        }
```

e de seguida calcular o seu valor, com os resultados fmp e fml anteriores:

```
> ldp <- ld50(coef(fmp))
> ldl <- ld50(coef(fml))
> c(ldp, ldl)
(Intercept) (Intercept)
  43.66335    43.60119
```

obtendo-se as estimativas de 43.663 anos e 43.601 anos, respectivamente.

### Modelos poisson

Para a família poisson a função de enlace pré-definida é ***log***, e o uso que fundamentalmente se faz na prática desta família é para ajustar modelos log-lineares de Poisson a dados de frequências cuja distribuição é geralmente multi-nomial. Este é um extenso e importante tema, que constitui uma parte fundamental da utilização de modelos generalizados não-gaussianos, que não será aqui desenvolvido.

Ocasionalmente surgem dados cuja distribuição é na realidade Poisson, que no passado se analisavam como dados gaussianos, após lhes aplicar uma transformação logarítmica ou raiz quadrada. Como alternativa a esta última transformação, pode ajustar-se um modelo linear generalizado de Poisson, como no seguinte exemplo:

```
> fmod <- glm(y ~ A + B + x, family = poisson(link = sqrt),
              data = worm.counts))
```

### Modelos quasi (quasi-verosimilhança)

Em todas as famílias, a variância da variável resposta depende da sua média, e tem um parâmetro de escala multiplicativo. A forma de dependência da variância em relação à média é uma característica da distribuição da variável resposta; por exemplo, para a distribuição Poisson será $\mathrm{var}[y] = \mu$.

Para a estimação e inferência da família *quasi-verosimilhança*, a distribuição exacta da variável resposta não está especificada, mas apenas uma função de enlace e a forma como a variância depende da média. Já que a estimação quase-verosimilhança utiliza formalmente as mesmas técnicas da distribuição gaussiana, esta família permite ajustar modelos gaussianos com funções de enlace ou inclusivamente com funções de variância não-standard.

Por exemplo, consideremos a seguinte regressão não linear:

$$y = \frac{\theta_1 z_1}{z_2 - \theta_2} + e$$

que pode escrever-se como:

$$y = \frac{1}{\beta_1 x_1 + \beta_2 x_2} + e$$

onde $x_1 = z_2/z_1$, $x_2 = -1/z_1$, $\beta_1 = 1/\theta_1$, $\beta_2 = \theta_2/\theta_1$. Considerando que existe uma folha de dados apropriada, bioquimica, podemos ajustar o modelo mediante:

```
> nlfit <- glm(y ~ x1 + x2 - 1,
        family =  quasi(link = inverse, variance = constant),
        data = bioquimica)
```

Consulte o manual e a ajuda disponível para mais detalhes.

## 11.7 Modelos de mínimos quadrados não lineares e de máxima verosimilhança

Algumas formas de modelos não lineares podem ajustar-se usando modelos lineares generalizados (função ***glm()***), mas na maior parte dos casos é necessário utilizar optimização não linear. A função de R para a realizar é ***nlm()***, que substitui as funções ***ms()*** e ***nlmin()*** de S-PLUS. Procuramos os valores dos parâmetros que minimizam algum índice de falta de ajustamento e ***nlm()*** resolve-o testando vários parâmetros iterativamente. Ao contrário da regressão linear, por exemplo, não há qualquer garantia de que o procedimento seja convergente para estimadores satisfatórios. A função ***nlm()*** necessita de uma valores de partida para os parâmetros a estimar e a convergência depende criticamente da qualidade desses valores iniciais.

### 11.7.1 Mínimos quadrados

Uma forma de ajustar um modelo não linear é minimizando a soma dos quadrados dos resíduos (SSE). este método faz sentido se os erros observados seguem satisfatoriamente uma distribuição normal.

Apresenta-se um exemplo retirado de Bates & Watts (1988), pág. 51. Os dados são:

```
> x <- c(0.02,0.02,0.06,0.06,0.11,0.11,0.22,0.22,0.56,
        0.56,1.10,1.10)
> y <- c(76,47,97,107,123,139,159,152,191,201,207,200)
```

O modelo que se pretende ajustar é:

```
> fn <- function(p)
        {
        sum((y-(p[1]*x)/(p[2]+x))^2)
        }
```

Para iniciar o ajustamento, necessitamos de uns valores iniciais para os parâmetros a estimar. Uma forma de encontrar uns valores de partida apropriados é representar graficamente os dados, tentar adivinhar os valores desses parâmetros, e desenhar sobre o gráfico a curva correspondente a estes valores.

```
> plot(x, y)
> xajustado <- seq(0.02, 1.1, 0.05)
> yajustado <- 200*xajustado / (0.1 + xajustado)
> lines(spline(xajustado, yajustado))
```

Embora se pudesse tentar encontrar valores melhor ajustados, os valores 200 e 0.1 parecem adequados. Passemos agora à estimação:

```
> resultado <- nlm(fn, p = c(200, .1), hessian = TRUE)
```

Após preceder ao ajustamento, *resultado$minimum* contém a soma dos quadrados dos resíduos (SSE), e *resultado$estimates* contém os estimadores de mínimos quadrados dos parâmetros. Para obter os erros-padrão aproximados (SE) dos estimadores faz-se:

```
> sqrt(diag(2*resultado$minimum/(length(y) - 2) *
        solve(resultado$hessian)))
```

O valor 2 na expressão anterior representa o número de parâmetros estimados. Um intervalo de confiança a 95% será parâmetro estimado ± 1.96 SE. Podemos representar o modelo ajustado sobre o gráfico dos valores:

```
> plot(x,y)
> xajustado <- seq(0.02, 1.1, 0.05)
> yajustado <- 212.68384222 * xajustado / (0.06412146 + xajustado)
> lines(spline(xajustado, yajustado))
```

A biblioteca **nls** dispõe de muitas outras capacidades para ajustamento de modelos não lineares pelo método dos mínimos quadrados. O modelo que acabámos de ajustar é o modelo de Michaelis-Menten, cuja função na biblioteca nls é Ssmicmen():

```
> library(nls)
> df <- data.frame(x=x, y=y)
> modelo <- nls(y ~ SSmicmen(x, Vm, k), df)
> modelo
```

Nonlinear regression model

model: y ~ SSmicmen(x, Vm, k)

data: df

Vm k

212.68370740 0.06412123

residual sum-of-squares: 1195.449

```
> summary(modelo)
```

Formula: y ~ SSmicmen(x, Vm, k)

Parameters:

Estimate Std. Error t value Pr(>|t|)

Vm 2.127e+02 6.947e+00 30.615 3.24e-11 ***

k 6.412e-02 8.281e-03 7.743 1.57e-05 ***

---

Signif. codes: 0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Residual standard error: 10.93 on 10 degrees of freedom

Correlation of Parameter Estimates:

Vm

k 0.7651

### 11.7.2 Máxima verosimilhança

O método de máxima verosimilhança é um método de ajustamento que se aplica mesmo que os erros não tenham distribuição normal. O método estima os valores dos parâmetros que maximizam a função log-verosimilhança ou, o que é equivalente, minimizam o valor -log-verosimilhança. Apresenta-se um exemplo retirado de Dobson (1990), pág. 108-111, em que se ajusta um modelo *logit* aos dados (dose, resposta), os quais evidentemente também poderiam ser modelados com ***glm()***:

```
> x <- c(1.6907,1.7242,1.7552,1.7842,1.8113,1.8369,1.8610,1.8839)
> y <- c(6, 13, 18, 28, 52, 53, 61, 60)
> n <- c(59, 60, 62, 56, 63, 59, 62, 60)
```

A função –log-verosimilhança a minimizar é:

```
> lv <- function(p)
      {
      sum( -(y * ( p[1]+p[2] * x)
            - n * log(1 + exp( p[1] + p[2] * x))
            + log(choose(n,y))))
      }
```

Tendo seleccionado uns valores iniciais (no caso, -50 e 20), procede-se ao ajustamento:

```
> resultado <- nlm(lv, p=c(-50, 20), hessian = TRUE)
```

Após o que, *resultado$minimum* contém o valor de –log-verosimilhança, e *resultado$estimates* contém os estimadores de máxima verosimilhança. Para obter os erros-padrão dos parâmetros faz-se:

```
> sqrt(diag(solve(resultado$hessian)))
```

e, tal como anteriormente, um intervalo de confiança a 95% para os parâmetros estimados será estimativa ± 1.96 SE.

## 11.8 Alguns modelos não-standard

Concluimos este capítulo com uma breve referência a outras capacidades de R para modelos especiais de regressão e análise de dados.

- **Modelos mistos**. A biblioteca **nlme** criada com o contributo dos utilizadores contém as funções ***lme()*** e ***nlme()*** para modelos lineares e não-lineares de efeitos mistos, isto é, modelos de regressão linear e não-linear em que alguns coeficientes correspondem a efeitos aleatórios. Estas funções fazem um intenso uso de fórmulas para especificar os modelos.

- **Regressão com aproximação local**. A função ***loess()*** ajusta um modelo de regressão não paramétrica utilizando regressão polinomial localmente ponderada. Este tipo de regressão é útil para realçar tendências em dados confusos ou para reduzir os dados e obter alguns padrões da estrutura de grandes amostras de dados.

  Esta função está disponível na biblioteca **modreg**.

- **Regressão robusta**. Existem várias funções para ajustar modelos de regressão resistentes à influência de valores anómalos (outliers). A função ***lqs()***, disponível na biblioteca **lqs**, contém os algoritmos mais recentes para ajustamentos resistentes. Outras funções menos resistentes mas mais eficientes estatisticamente podem encontrar-se noutras bibliotecas criadas pelos utilizadores, tal como a função ***rlm()*** da biblioteca **MASS**.

- **Modelos aditivos**. Esta técnica tenta construir uma função de regressão a partir de funções aditivas suavizadas das variáveis predictoras, geralmente uma por cada variável preditora. As funções ***avas()*** e ***ace()*** da biblioteca **acepack** e as funções ***bruto()*** e ***mars()*** na biblioteca **mda** são exemplos de técnicas deste tipo, disponíveis em bibliotecas desenvolvidas pelos utilizadores de R.

- **Modelos em árvore**. Em vez de buscar um modelo global explícito para a predição ou interpretação, os modelos baseados em árvores tentam separar recursivamente os dados, em pontos críticos das variáveis preditoras, com a finalidade de conseguir uma partição dos dados em grupos tão homogéneos quanto possível dentro dos grupos e heterogéneos entre os grupos. Os resultados conduzem geralmente a uma compreensão dos dados que outros métodos não conseguem. Os modelos são especificados em forma de modelos lineares ordinários. A função de ajustamento é ***tree()***, e muitas funções genéricas, tais como ***plot()*** e ***text()*** podem visualizar graficamente os resultados destes modelos. Pode encontrar estes modelos nas bibliotecas **rpart** e **tree**, desenvolvidas pelos utilizadores.

# 12 Rotinas gráficas

As capacidades gráficas são uma componente muito importante e extremamente versátil do ambiente R. É possível utilizar estas possibilidades para criar uma grande variedade de gráficos estatísticos, e também para definir novos tipos de gráficos.

As capacidades gráficas podem usar-se de modo interactivo, ou em processamento por lotes, mas na maioria dos casos, o modo interactivo é mais produtivo. O modo interactivo é também mais fácil, porque no arranque, R inicia um *dispositivo gráfico* que abre uma *janela gráfica* para visualizar os gráficos. Embora este procedimento seja automático, é útil saber que o comando utilizado é **X11()** em ambiente UNIX; em ambiente Windows, podem usar-se os comandos **X11()** ou **windows()**.

Após ter iniciado o dispositivo gráfico, os comandos para criar gráficos podem usar-se para representar graficamente dados ou para definir novos tipos de gráficos.

Os comandos sobre gráficos podem agrupar-se em três categorias:

- Funções gráficas de **alto nível,** que criam novos gráficos na janela gráfica, definindo os eixos, etiquetas, títulos, etc.
- Funções gráfica de **baixo nível**, que permitem adicionar novas informações a gráficos já criados, tal como novos dados, linhas, etiquetas.
- Funções gráficas **interactivas** que permitem adicionar ou remover interactivamente informação aos gráficos existentes, utilizando um dispositivo apontador, como por exemplo o rato.

R mantém automaticamente uma lista dos *parâmetros gráficos* que pode ser manipulada para personalizar os gráficos.

## 12.1 Funções gráficas de alto nível

As funções gráficas de alto nível estão desenhadas para gerar um gráfico completo a partir dos dados que são passados como argumento para a função. Quando for possível, os eixos, etiquetas e títulos são gerados automaticamente (a não ser que se especifique o contrário). As funções gráficas de alto nível iniciam sempre um novo gráfico, eliminando o gráfico actual se tal for necessário.

### 12.1.1 A função plot()

Uma das funções gráficas mais usada em R é a função ***plot()***, que é uma função genérica: o tipo de gráfico que é criado depende do tipo ou da classe do primeiro argumento dado à função.

`plot(x , y)`

`plot(xy)` Se x e y são vectores, ***plot(x,y)*** cria um gráfico de pontos ou diagrama de dispersão de y em função de x. O mesmo efeito é obtido dando apenas um

argumento (como na segunda forma) ou uma lista contendo os dois elementos x e y ou uma matriz de duas colunas.

`plot(x)` Se x é uma série de observações ao longo do tempo, este comando produz um gráfico da série temporal; se x é um vector numérico, o comando cria um gráfico dos valores do vector sobre os respectivos índices; se x é um vector complexo, é produzido um gráfico da parte imaginária versus a parte real dos elementos.

`plot(f)`

`plot(f, y)` em que f é um factor e y um vector numérico. A primeira forma cria um gráfico de barras das classes de f; a segunda forma produz um diagrama de extremos-e-quartis (ou caixa-com-bigodes) de y para cada nível de f.

`plot(df)`

`plot(~expr)`

`plot(y~expr)` em que df é uma folha de dados, y é um objecto, *expr* é uma lista de nomes de objectos separados por "+" (por exemplo, **a+b+c**). As primeiras duas formas criam gráficos das distribuições de todas as combinações das variáveis na folha de dados (primeira forma) ou dos objectos definidos na expressão (segundo caso). A terceira forma produz os gráficos de y versus todos os objectos definidos na expressão.

### 12.1.2 Gráficos de dados multivariados

R dispõe de duas funções muito úteis para representar graficamente dados multivariados. Se X é uma matriz numérica ou uma folha de dados numéricos, o comando:

```
> pairs(X)
```

cria uma matriz de gráficos de dispersão de todas as combinações de pares de variáveis definidas pelas colunas de X, isto é, cada coluna de X é representada graficamente versus cada uma das outras colunas de X; os n(n-1) gráficos resultantes são dispostos numa matriz de gráficos, com as escalas constantes ao longo das linhas e colunas da matriz.

Quando se dispõe de três ou quatro variáveis, a função ***coplot()*** pode ser mais elucidativa. Se a e b são vectores numéricos e c é um vector numérico ou um factor (todos com o mesmo comprimento), o comando:

```
> coplot(a ~ b | c)
```

produz os diagramas de dispersão de a sobre b para cada valor de c, ou para cada categoria de c, se esta for um factor. Se c é um vector numérico, este é dividido em intervalos, e são produzidos os diagramas de dispersão de a sobre b para cada intervalo de c. O número e limites dos intervalos podem ser controlados usando o argumento given.values na função ***coplot()***; a função ***co.intervals()*** também pode ser útil para definir os intervalos. Podem usar-se duas variáveis condicionantes num comando da forma:

```
> coplot(a ~b | c + d)
```

que cria os diagramas de dispersão de a sobre b para todos os intervalos definidos conjuntamente por c e d.

Ambas as funções ***pairs()*** e ***coplot()*** admitem o argumento *panel* para personalizar o tipo de gráfico que é criado em cada painel. Por defeito, a função para criar diagramas de dispersão é a função ***points()***, mas se se usam algumas funções de baixo nível sobre dois vectores numéricos x e y como argumento de panel, pode criar-se o tipo de gráfico que se pretenda. Um exemplo de uma função para definir os painéis de ***coplot()*** é a função ***panel.smooth()***.

### 12.1.3 Outras funções gráficas

R dispõe de outras funções gráficas de alto nível para a criar diferentes tipos de gráficos. Algumas destas funções são:

`qqnorm(x)`
`qqline(x)`
`qqplot(x, y)` Gráficos para a comparação de distribuições. A primeira função cria um gráfico do vector numérico x versus os valores normais estandardizados. A segunda função acrescenta uma linha recta ao gráfico de ***qqnorm()***, que passa pelos quartis da distribuição dos valores. A terceira forma cria um gráfico dos quantis de x *versus* os quantis de y, permitindo comparar as duas distribuições.

`hist(x)`
`hist(x, nclass=n)`
`hist(x, breaks=b, ...)` Produz um histograma do vector numérico x. Geralmente o número de classes é calculado automática e correctamente, mas pode alterar-se o número de classes com o parâmetro *nclass=n*; Alternativamente, podem definir-se os limites das classes com o argumento *breaks*. Usando o argumento *probability=TRUE*, é representado o histograma das frequências relativas.

`dotplot(x, ...)` Constrói um gráfico de pontos para x. Neste tipo de gráficos, o eixo y etiqueta os dados (pela sua posição no vector) e o eixo x representa os valores. Por exemplo, permite seleccionar visualmente de todos os valores que ficam num determinado intervalo.

`image(x, y, z, ...)`
`contour(x, y, z, ...)`
`persp(x, y, z, ...)` Permitem criar gráficos tri-dimensionais. A função ***image()*** representa uma retícula de rectângulos com cores diferentes segundo o valor de z; ***contour()*** representa curvas de níveis com os valores z; ***persp()*** representa uma superfície tri-dimensional.

### 12.1.4 Argumentos das funções gráficas de alto nível

É possível definir uma série de argumentos para as funções gráficas de alto nível, entre os quais:

*add=TRUE* Obriga a portar-se como se se tratasse de uma função de baixo nível, de modo que o gráfico criado será sobreposto ao gráfico actual, em vez de o apagar previamente (só está disponível para algumas funções).

*axes=FALSE* Elimina os eixos. Esta opção é útil para que o utilizador defina e personalize os eixos com a função ***axis()***. Por defeito a opção é *axes=TRUE* que define automaticamente os eixos.

*log="x"*

*log="y"*

*log="xy"* Transforma o eixo x, o eixo y ou ambos, em escala logarítmica. Não funciona nalguns tipos de gráficos.

*type=* Este argumento controla o tipo de gráfico produzido, de acordo com as seguintes especificações:

- *type = "p"* Representa os pontos individualmente (por defeito)
- *type = "l"* Gráfico de linhas
- *type = "b"* Pontos unidos por linhas
- *type = "o"* Pontos e linhas, com estas sobrepostas aos pontos
- *type = "h"* Representa linhas verticais desde os pontos ao eixo x=0
- *type = "s"*
- *type = "S"* Gráficos em escada; na primeira opção (*type="s"*), os pontos são definidos pelo topo da linha vertical; na segunda opção, os pontos são a base da linha vertical.
- *type = "n"* Não se produz qualquer gráfico; são apenas desenhados os eixos (por defeito) e são representadas as coordenadas de acordo com os dados. Ideal para definir de seguida um gráfico com funções de baixo nível.

*xlab = string*

*ylab = string* Definem os nomes para os eixos x e y, respectivamente, para substituição dos nomes definidos por defeito, que normalmente são os nomes dos objectos utilizados para a criação do gráfico.

*main = string* Define o título do gráfico, colocando-o no topo, em letras de tamanho grande.

*sub = string* Define o sub-título do gráfico, colocando-o abaixo do eixo dos x em letras de tamanho pequeno.

## 12.2 Funções gráficas de baixo nível

Pode acontecer que, por vezes, as funções gráficas de alto nível não produzam exactamente o tipo de gráfico pretendido. Neste caso, os comandos de baixo nível podem usar-se para adicionar informação adicional (tal como pontos, linhas ou texto) ao gráfico actual.

Algumas das funções de baixo nível mais usadas são:

`points(x,y)`

`lines(x,y)` Acrescenta pontos ou linhas ao gráfico actual. A opção *type* da função ***plot()*** pode usar-se nesta função (os valores pré-definidos são *"p"* para ***points()*** e *"l"* para ***lines()***).

`text(x, y, etiquetas, ...)` Acrescenta texto aos pontos (x,y). Geralmente, etiquetas é um vector de valores inteiros ou de caracteres, de modo a que etiquetas[i] é colocado no ponto (x[i], y[i]). O valor por defeito é 1:*length(x)*.

**Nota**: Esta função é geralmente utilizada num comando do tipo:

```
> plot(x, y, type="n") ; text(x, y, etiquetas)
```

O parâmetro *type="n"* suprime os pontos, mas são definidos os eixos; a função ***text()*** aplica os caracteres definidos em etiquetas no sítio dos pontos.

```
abline(a, b)
abline(h = y)
abline(v = x)
```

`abline(lm.obj)` Acrescenta uma recta de declive b e ordenada na origem a ao gráfico actual. A opção $h = y$ representa uma linha horizontal à altura y; a opção $v = x$ representa uma linha vertical no ponto de abcissa x. Na última forma, *lm.obj* refere-se a uma lista com uma componente designada *coefficients* de dimensão 2 (por exemplo, o resultado de uma função de ajustamento de um modelo de regressão), que são assumidos para ordenada na origem e declive, nesta ordem.

`polygon(x, y, ...)` Desenha uma linha poligonal no gráfico actual, cujos vértices são os pontos (x,y). Opcionalmente pode sombrear ou preencher a figura com uma cor.

`legend(x, y, legenda, ...)` Aplica a legenda ao gráfico actual, na posição especificada. As fontes a usar, estilos de linha, cores, etc., são definidos no vector legenda. Deve definir-se pelo menos mais um argumento v (com o mesmo comprimento de legenda), especificando algumas características, tal como se segue:

*legend(... , fill = v)* Cores de preenchimento;

*legend(... , col = v)* Cores para as linhas ou pontos;

*legend(..., lty = v)* Tipos de linha;

*legend(..., lwd = v)* Espessura de linha;

*legend(..., pch = v)* Caracteres para desenhar (vector alfanumérico).

`title(main, sub)` Aplica o título principal, *main*, na parte superior do gráfico, em caracteres grandes, e o sub-título, *sub*, na parte inferior, em fontes menores.

`axis(side, ...)` Acrescenta um eixo ao gráfico actual, no lado especificado pelo primeiro argumento (*side* especifica uma das posições de 1 a 4, contando no sentido dos ponteiros do relógio a partir do fundo). Outros argumentos especificam a posição do eixo, dentro ou fora do gráfico, as marcas e etiquetas do eixo. Esta função é útil para definir os eixos, se na função ***plot()*** se eliminaram os eixos com o argumento *axes=FALSE*.

As funções gráficas de baixo nível geralmente requerem alguma informação de posicionamento, como as coordenadas (x,y) para definir o local dos novos elementos acrescentados ao gráfico. As coordenadas são especificadas em termos de *coordenadas do utilizador*, as quais são definidas pelas funções de alto nível prévias, e são seleccionadas em função dos dados usados na construção do gráfico.

Quando os argumentos x e y são necessários, podem substitui-se por um objecto de classe *list*, que deve conter duas componentes chamadas x e y, ou por uma matriz de duas colunas. Deste modo, funções como ***locator()*** (que veremos de seguida) podem usar-se para especificar interactivamente as posições no gráfico.

### 12.2.1 Anotações matemáticas

Em muitas situações, é conveniente acrescentar símbolos matemáticos e fórmulas ao gráfico criado. Esta tarefa realiza-se em R especificando uma expressão, em vez de uma cadeia de caracteres, em qualquer das funções ***text()***, ***mtext()***, ***axis()*** ou ***title()***. Por exemplo, a seguinte ordem define a fórmula da distribuição binomial:

```
> text(x, y, expression(paste(bgroup("(", atop(n,x),")"),
            p^x, q^{n-x})))
```

Mais informações, incluindo una lista completa das capacidades disponíveis acerca deste tema, pode obter-se com os comandos:

```
> help(plotmath)
> example(plotmath)
```

### 12.2.2 Fontes vectoriais Hershey

É possível escrever texto utilizando as fontes vectoriais Hershey nas funções ***text()*** e ***contour()***. São três as razões para utilizar estas fontes:

- Produzem melhores resultados, especialmente no monitor, com texto rodado ou de pequeno tamanho;
- Contêm símbolos que podem não estar disponíveis nas fontes correntes, tais como os signos do zodíaco, símbolos cartográficos ou astronómicos;
- Contêm caracteres cirílicos e japoneses (Kana e Kanji).

A informação detalhada, incluindo as tabelas de caracteres, pode obter-se com as ordens:

```
> help(Hershey)
> example(Hershey)
> help(Japanese)
> examples(Japanese)
```

## 12.3 Funções gráficas interactivas

R dispõe de funções que permitem extrair ou adicionar informação a um gráfico utilizando o rato. A mais fácil é a função ***locator()***:

`locator(n, type)` Permite que o utilizador seleccione regiões do gráfico usando o botão esquerdo do rato, até que se tenha seleccionado um máximo de n (por defeito, n=512) pontos, ou até pressionar o botão direito para terminar a selecção. O argumento type permite acrescentar elementos ao gráfico (utiliza algumas da opções deste argumento das funções de alto nível). Por defeito, o argumento type está desactivado. A função ***locator()*** devolve uma lista com as coordenadas (x,y) dos pontos seleccionados.

Geralmente evoca-se a função ***locator()*** sem qualquer argumento. É particularmente útil para seleccionar interactivamente posições do gráfico para a colocação de elementos adicionais, tal como etiquetas, quando de outro modo seria difícil calcular previamente onde colocá-las. Por exemplo, para colocar a anotação "Anómalo" junto a um ponto, faz-se:

```
> text(locator(1), "Anómalo", adj=0)
```

e aponta-se com o rato para a vizinhança do ponto pretendido. Não dispondo de rato, a função ***locator()*** também se pode usar; neste caso, o utilizador deverá especificar as coordenadas (x,y).

`identify(x, y, labels)` Permite identificar os pontos definidos por x e y, utilizando o botão esquerdo do rato, e colocando a etiqueta definida por *label* junto ao ponto; se não se definir *label*, o ponto será identificado pelo seu índice. É útil quando existe um vector de índices ou de etiquetas associado aos valores x e y, permitindo identificar pontos do gráfico com o índice ou etiqueta respectiva.

Por vezes interessa identificar pontos particulares num gráfico, e não apenas a sua posição. Por exemplo, pode pretender-se seleccionar uma observação de interesse no gráfico, para posteriormente manipular de alguma forma. Dado um conjunto de coordenadas (x,y) em dois vectores numéricos x e y, podemos usar a função ***identify()*** do seguinte modo para seleccionar estes pontos:

```
> plot(x, y)
> identify(x, y)
```

A função ***identify()*** não faz automaticamente nenhuma acção por si só; antes permite ao utilizador mover o rato e pressionar o botão esquerdo na proximidade dos pontos. O ponto mais próximo ao ponteiro (se está suficientemente próximo) é identificado com o seu número índice, isto é, a posição que ocupa nos vectores x e y. Alternativamente poderia usar-se um vector de

etiquetas (contendo, por exemplo, os nomes das observações) para a identificação, utilizando este vector como argumento *labels* da função. O argumento *plot=FALSE* inibe a identificação dos pontos sobre o gráfico, e a função ***identify()*** dá uma lista dos índices das observações apontadas, que permitem a identificação dessas observações nos vectores x e y.

## 12.4 Uso de parâmetros gráficos

Quando se criam gráficos, especialmente se se destinam a apresentação ou publicação, é possível que R não produza de modo automático a aparência pretendida. O utilizador pode personalizar cada aspecto do gráfico utilizando *parâmetros gráficos*. R dispõe de uma vasta colecção de parâmetros gráficos que permitem controlar aspectos tais como o estilo de linha, cores, disposição das figuras, alinhamento do texto, etc. Cada parâmetro gráfico é identificado por um nome (por exemplo, "col" para a cor) e toma um valor (por exemplo, *col="blue"* para definir a cor azul).

A cada dispositivo gráfico activo corresponde uma lista dos parâmetros gráficos, e cada dispositivo gráfico dispõe de um conjunto pré-determinado de parâmetros definidos ao iniciar. Os parâmetros gráficos podem indicar-se de dois modos: de modo permanente, que afectarão todas as funções gráficas que acedam a esse dispositivo; de modo temporário, que só afectam a função gráfica que o utiliza nesse momento.

### 12.4.1 Definição de parâmetros gráficos de modo permanente. A função par()

A função ***par()*** é usada para aceder e modificar a lista dos parâmetros gráficos do dispositivo gráfico actual.

`par()` Sem argumentos, devolve uma lista de todos os parâmetros e respectivos valores do dispositivo gráfico actual.

`par(c("col", "lty"))` Com um vector de caracteres (contendo nomes válidos de argumentos), devolve os valores dos parâmetros especificados.

`par(col=4, lty=2)` Especificando os valores dos parâmetros, estes são alterados para os valores indicados. Devolve, de modo invisível, uma lista dos valores originais.

Ao modificar qualquer parâmetro com a função ***par()***, a modificação é *permanente*, no sentido que as funções gráficas chamadas de seguida (no dispositivo gráfico actual) serão afectadas pelos novos valores. Pode pensar-se que este tipo de assignação equivale a modificar os valores pré-determinados dos parâmetros que utilizarão as funções gráficas, a não ser que outros valores sejam especificados.

A utilização da função ***par()*** afecta sempre os valores globais dos parâmetros gráficos, mesmo que ***par()*** seja chamada dentro de outra função. Este comportamento pode ser indesejável: usualmente pretendemos definir alguns parâmetros gráficos, criar alguns gráficos, e retomar os valores originais, sem afectar o resto da sessão de trabalho em R. Podemos recuperar os parâmetros iniciais, guardando-os ao proceder à sua alteração para posteriormente os recuperar (recorde que ao alterar os parâmetros, é devolvida uma lista com os valores originais):

```
> oldpar <- par(col = 4, lty =2)
      ... funções gráficas ...
> par(oldpar)
```

### 12.4.2 Alterações temporárias. Argumentos das funções gráficas

Os parâmetros gráficos também podem passar-se a praticamente todas as funções gráficas como argumentos com nome, o que tem o mesmo efeito que utilizá-los na função ***par()***, excepto que as alterações só actuam na função em que são definidos. Por exemplo:

```
> plot(x, y, pch="+")
```

realiza um diagrama de dispersão utilizando o símbolo "+" para representar os pontos, sem alterar o valor pré-definido para os gráficos posteriores.

## 12.5 Parâmetros gráficos habituais

De seguida apresentam-se muitos dos parâmetros gráficos habitualmente usados. A ajuda disponível para a função ***par()*** contém um resumo mais conciso.

Os parâmetros gráficos serão apresentados no seguinte formato:

*nome = valor* — Descrição do efeito do parâmetro. A palavra *nome* significa o nome do parâmetro, isto é, o nome do argumento que deve usar-se na função ***par()*** ou outra função gráfica; ***valor*** é um valor típico do parâmetro.

### 12.5.1 Elementos gráficos

Os gráficos de R são formados por pontos, linhas, texto e polígonos (regiões preenchidas). Existem parâmetros gráficos que controlam como se desenham estes *elementos gráficos*, como por exemplo:

*pch = "+"* — Caracter a ser usado para desenhar os pontos. O valor pré-determinado varia entre dispositivos gráficos, mas normalmente é *"o"*. Os pontos tendem a aparecer em posição ligeiramente distinta da exacta, salvo se se usa ".", que produz pontos centrados.

*pch = 4* — O argumento pch pode ser especificado por um valor inteiro entre 0 e 18 (ambos incluídos). Para saber os símbolos que correspondem a cada código, utilize as seguintes ordens:

```
> plot(1,  t =  "n")
> legend(locator(1), as.character(0:18), pch=0:18)
```

e aponte para o topo do gráfico. Aparece uma lista de caracteres e respectivo código.

*lty = 2* — É o tipo de linha. Embora alguns tipos de linhas não possam ser usadas nalguns dispositivos gráficos, o tipo 1 corresponde a uma linha sólida, e os tipos 2 e seguintes correspondem a linhas ponteadas, tracejadas ou combinações destes tipos. Para obter os tipos de linhas, faça as ordens indicadas para *pch*, substituindo pch por *lty*.

*lwd = 2* Especifica a espessura da linha, medida em múltiplos da largura base. Afecta os eixos e a linhas desenhadas com as funções ***lines()***, etc.

*col = 2* Especifica a cor que se utiliza para os pontos, linhas, texto, imagens e preenchimento de regiões. Cada um destes elementos gráficos admite uma lista de cores possíveis e o valor deste parâmetro é um índice dessa lista. Obviamente este parâmetro só aplicável em alguns dispositivos.

*font = 2* Valor inteiro que especifica a fonte que se utilizará para o texto. Se tal for possível, os dispositivos gráficos usam o valor 1 para texto normal, 2 para texto em negrito, 3 em itálico e 4 itálico negrito.

*font.axis*

*font.lab*

*font.main*

*font.sub* Especificam a fonte a usar nos eixos, etiquetas, título principal e sub-título, respectivamente.

*adj = -0.1* Indica como deve alinhar-se o texto em relação à posição de desenho. O valor 0 indica alinhamento à esquerda, 1 indica alinhamento à direita, e o.5 indica texto centrado. Pode usar qualquer outro valor que indicará a proporção de texto que aparece à esquerda da posição de desenho; assim, o valor –0.1 deixará 10% da extensão de texto entre si e a posição de desenho.

*cex = 1.5* Define a expansão do texto. O valor indica a proporção de aumento ou diminuição do texto (incluindo os caracteres de desenho) em relação ao tamanho pré-definido.

### 12.5.2 Eixos e marcas de escala

A maior parte dos gráficos criados em R têm eixos, mas é sempre possível definir os eixos com a função gráfica de baixo nível ***axis()***. Os eixos têm três componentes principais: a linha do eixo, cujo estilo é controlado pelo parâmetro *lty*, as marcas de escala, que indicam as unidades de divisão do eixo, e as etiquetas de escala, que indicam as unidades das marcas de escala. Estas componentes podem modificar-se com os seguintes parâmetros gráficos:

*lab = c(5, 7, 12)* Os primeiros dois argumentos especificam o número de marcas ou intervalos no eixo x e y, respectivamente; o terceiro argumento é o comprimento das etiquetas dos eixos, em caracteres (incluindo o ponto decimal). Se se escolhe um valor demasiado pequeno para este parâmetro, pode acontecer que todas as etiquetas se arredondem para o mesmo número.

*las = 1* Corresponde à orientação das etiquetas dos eixos. O valor 0 indica sempre paralelo ao eixo, 1 indica sempre horizontal e 2 indica perpendicular ao eixo.

*mpg = c(3, 1, 0)* São as posições das componentes dos eixos. O primeiro argumento indica a distância, medida em linhas de texto, entre o eixo e as etiquetas. O

segundo é a distância para as etiquetas das marcas, e o último é a distância entre a posição do eixo e a linha do eixo (normalmente 0). Os valores positivos indicam que as componentes estão fora da zona do gráfico e valores negativos indicam que as componentes ficam dentro da área de desenho.

*tck = 0.01* Comprimento das marcas de divisão, definida como uma fracção do tamanho da zona de desenho. Quando o valor de *tck* é pequeno (menos de 0.5) as marcas de divisão de ambos os eixos são do mesmo tamanho. Um valor de 1 faz aparecer uma grelha no gráfico. Se o valor é negativo, as marcas de divisão fazem-se na parte exterior da zona de desenho. Utilize *tck = 0.01* e *mgp = c(1, -1.5, 0)* para obter marcas de divisão internas.

*xaxs = "s"*

*yaxs = "d"* Estilo dos eixos x e y respectivamente. Com os estilos *"s"* (de standard) e *"e"* (de extensão) tanto a marca maior como a marca menor ficam fora do intervalo dos dados. Os eixos extendidos podem ampliar-se ligeiramente se existe algum ponto muito próximo do bordo do gráfico. Este estilo de eixos pode deixar por vezes grandes zonas vazias nas proximidades dos bordos do gráfico. Com os estilos *"i"* (interno) e *"r"* (pré-definido) as marcas de divisão ficam sempre dentro do intervalo dos dados; o estilo *"r"* deixa um pequeno espaço nos bordos.

Seleccionando o estilo *"d"* (directo) os eixos actuais ficam bloqueados e são usados para os gráficos seguintes até que este parâmetro seja alterado para outro valor. Este procedimento é útil para gerar séries de gráficos com escalas fixas.

### 12.5.3 Margens das figuras

Um gráfico único de R designa-se por *figura* e compreende uma zona de desenho rodeada de margens (onde são colocadas os títulos, etiquetas dos eixos, etc.)e, geralmente, delimitada pelos próprios eixos.

Uma figura típica é ilustrada na página seguinte.

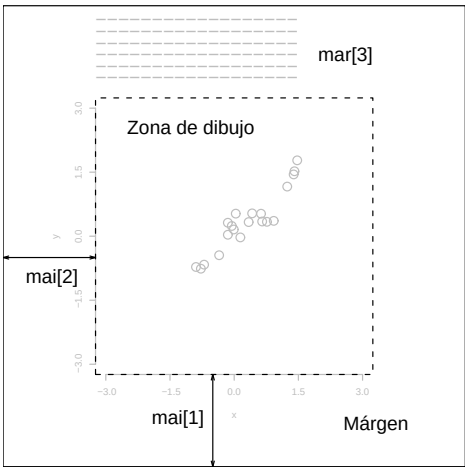
Os parâmetros gráficos que controlam a disposição da figura são:

*mai = c(1, 0.5, 0.5, 0)* Largura das margens inferior, esquerda, superior e direita, respectivamente, medidas em polegadas.

*mar = c(4, 2, 2, 1)* Similar a mai, mas em linhas de texto.

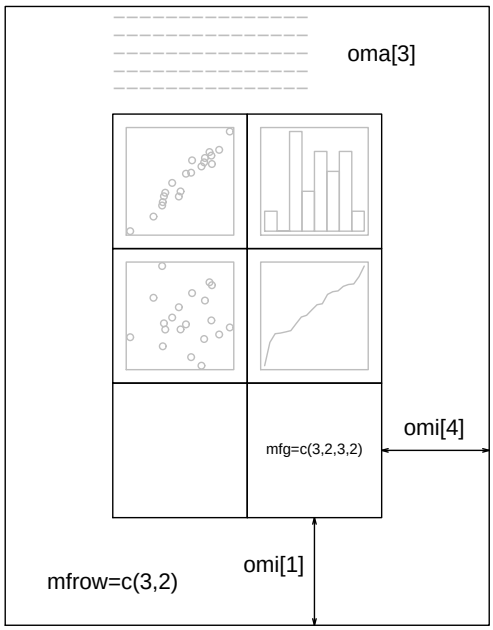
Os parâmetros *mar* e *mai* estão relacionados no sentido de que uma alteração num se reflecte no outro. Os valores pré-determinados são geralmente demasiado grandes, a margem direita raramente é necessária, tal como a superior, se não se inclui título. As margens inferior e esquerda devem ter o tamanho suficiente para incluir o eixo e as etiquetas das marcas de escala. Além disso, o valor pré-determinado não tem em conta a superfície do dispositivo gráfico. Assim, se utiliza o dispositivo gráfico ***postscript()*** com o argumento *height =4* obterá um gráfico em que metade do mesmo são margens, salvo se explicitamente mudar mar ou mai.

Quando há figuras múltiplas, como se verá de seguida, as margens são reduzidas a metade; contudo, esta redução pode não ser suficiente quando várias figuras compartem a mesma página.




### 12.5.4 Figuras múltiplas

R permite a criação de uma matriz de $n \times m$ figuras numa só página. Cada figura tem as suas próprias margens, e a matriz de figuras pode estar opcionalmente rodeada de uma *margem exterior*, tal como se mostra na figura seguinte:




Os parâmetros gráficos relacionados com as figuras múltiplas são:

*mfcol = c(3, 2)*

*mfrow = c(2, 4)* Definem o tamanho da matriz de figuras múltiplas. Em ambos os casos, o primeiro valor é o número e linhas e o segundo o número de colunas. A diferença entre os dois, é que com a primeira forma, *mfcol*, a matriz é preenchida por colunas, enquanto que com a segunda forma, mfrow, o

preenchimento é feito por linhas. A distribuição na figura de exemplo foi criada com *mfrow=c(3,2)* e mostra a página após terem sido criados os primeiros quatro gráficos.

*mfg = c(2, 2, 3, 2)* Definem a posição da figura actual dentro da matriz de figuras múltiplas. Os primeiros dois valores indicam a linha e a coluna da figura actual; os dois últimos valores são o número de linhas e colunas da matriz de figuras múltiplas. Utilize estes parâmetros para seleccionar cada uma das diferentes figuras da matriz. Os dois últimos valores podem ser diferentes dos verdadeiros valores, a fim de poder obter figuras de tamanhos distintos mesma página.

*fig = c(4, 9, 1, 4)/10* Definem a posição da figura actual na página. Os valores são as posições das margens esquerda, direita, inferior e superior, respectivamente, expressas em proporção da página desde o canto inferior esquerdo. O exemplo corresponde a uma figura na parte inferior direita da página. este parâmetro permite colocar uma figura em qualquer lugar da página.

*oma = c(2, 0, 3, 0)*

*omi = c(0, 0, 0.8, 0)* Definem o tamanho das margens exteriores. De modo similar a *mar* e a *mai*, o primeiro está expresso em linhas de texto e o segundo em polegadas, e correspondem às margens inferior, esquerda, superior e direita, respectivamente.

As margens exteriores são particularmente úteis para ajustar convenientemente os títulos e etiquetas. Pode acrescentar texto nestas margens com a função ***mtext()***, usando o argumento *outer=TRUE*. Por defeito, não são criadas margens exteriores, tendo de ser definidas explicitamente com os argumentos oma ou omi.

É possível criar disposições mais complexas de figuras múltiplas, usando as funções ***split.screen()*** e ***layout()***.

## 12.6 Dispositivos gráficos

É possível criar gráficos com R (com níveis de qualidade diversos) em quase todos os tipos de monitores ou de impressoras. Contudo, é necessário definir previamente do tipo de dispositivo de que se trata. Esta definição é realizada iniciando um *controlador do dispositivo gráfico*. A finalidade do controlador do dispositivo é converter as instruções gráficas de R (por exemplo, "desenha uma linha") numa forma que o dispositivo em particular entenda.

Os controladores de dispositivo são iniciados chamando uma função do controlador. Existe uma função para cada controlador, e a lista completa pode obter-se com o comando `help(Devices)`. Por exemplo, a ordem:

```
> postscript()
```

direcciona qualquer saída gráfica para uma impressora com formato PostScript. Alguns controladores de dispositivos gráficos habituais são:

`x11()` Para usar com o ambiente de janelas X11 e Microsoft Windows.

`postscript()` Para imprimir em impressoras PostScript ou criar ficheiros com este formato.

`pictex()` Cria um ficheiro LATEX.

Ao terminar de utilizar um dispositivo, assegure-se de finalizar o respectivo controlador com o comando:

```
> dev.off()
```

Esta ordem assegura que o dispositivo encerra correctamente; por exemplo no caso de uma impressora, assegura que cada página é completamente composta e enviada para a impressora.

### 12.6.1 Inclusão de gráficos PostScript em documentos

Utilizando o argumento file na função ***postscript()*** pode guardar os gráficos, em formato PostScript, no arquivo que deseje. O gráfico terá a orientação horizontal, a não ser que se especifique o argumento *horizontal=FALSE*. O tamanho do gráfico é controlado com ao argumentos *width* (largura) e *height* (altura) (o gráfico será re-dimensionado de modo a ajustar-se correctamente às dimensões especificadas). Por exemplo, o comando:

```
>postscript("grafico.ps",horizontal=FALSE,height=5,pointsize=10)
```

criará um arquivo que contém o código PostScript para uma figura com 5 polegadas de altura, e que poderá ser incluído num documento. Tenha em atenção que se o ficheiro já existe, o seu conteúdo será sobrescrito pelo actual comando. Isto ocorrerá mesmo que o arquivo tenha sido criado na sessão actual.

A maior parte das criação de ficheiros PostScript destina-se à inclusão de figuras em documentos. Esta tarefa pode resultar melhor utilizando formato EPS (*Ecapsulated PostScript*): o arquivo produzido por R é sempre deste formato, embora só marque o arquivo como sendo EPS se se utiliza o argumento *onefile=FALSE*. Esta notação é consequência da compatibilidade com S, e indica que a saída é constituída por uma única página (que é uma especificação do formato EPS). Para criar um gráfico que possa incluir num documento sem qualquer problema, deverá utilizar o comando:

```
> postscript("grafico.eps", horizontal=FALSE, onefile=FALSE,
      height = 8, width = 6, poitsize = 10)
```

### 12.6.2 Dispositivos gráficos múltiplos

Na utilização avançada de R é geralmente necessário dispor de diversos dispositivos gráficos em simultâneo. Naturalmente apenas um dos dispositivos gráficos aceitará as ordens gráficas em cada momento, que é designado por *dispositivo actual*. Quando se abrem vários dispositivos, formam uma sequência numerada cujos nomes determinam o tipo de dispositivo em cada posição.

Os principais comandos relacionados com dispositivos gráficos múltiplos, e o respectivo significado, são os seguintes:

`X11()` Abre uma janela gráfica em UNIX e em Microsoft Windows

`windows()` Abre uma janela gráfica em Microsoft Windows

`postscript()`

`pictex()` Cada chamada a uma função de controlador de dispositivo abre um novo dispositivo gráfico, e, portanto, acrescenta um elemento à lista de dispositivos, ao mesmo tempo que o último chamado passa a ser o dispositivo actual, para o qual serão enviados os resultados gráficos. (Nalgumas plataformas é possível que existam outros dispositivos disponíveis).

`dev.list()` Informa o número e o nome de todos os dispositivos activos. O dispositivo na posição 1 desta lista é sempre um *dispositivo nulo* que não aceita qualquer ordem gráfica.

`dev.next()`

`dev.prev()` Informa qual o nome e o número do dispositivo gráfico seguinte e prévio em relação ao dispositivo actual.

`dev.set(which = k)` Pode usar-se para mudar o dispositivo gráfico para o que está na k.ésima posição da lista de dispositivos. Informa qual o nome e o número desse dispositivo.

`dev.off(k)` Encerra o dispositivo gráfico que está na k.ésima posição da lista de dispositivos. Para alguns dispositivos, como os postscript, ou finalizará o gráfico, imprimindo-o de seguida, ou terminará a gravação em ficheiro EPS para posterior impressão, dependendo de como o dispositivo foi iniciado.

`dev.copy(device, ..., which=k)`

`dev.print(device, …, which=k)` Realiza uma cópia do dispositivo k. Aqui, a expressão *device* é uma função do dispositivo, como *postscript*, com argumentos adicionais se tal for necessário, especificados por ... . A função ***dev.print()*** é similar, mas o dispositivo copiado é fechado imediatamente, o que finaliza as acções pendentes, que se realizam imediatamente.

`graphics.off()` Encerra todos os dispositivos gráficos, excepto o dispositivo nulo.

## 12.7 Gráficos dinâmicos

R não dispõe (actualmente) de nenhuma função de gráficos dinâmicos, por exemplo para rodar uma nuvem de pontos, ou activar e desactivar pontos interactivamente. Contudo, muitas capacidades gráficas dinâmicas estão disponíveis no sistema **Xgobi** de Swayne, Cook e Buja, disponível em

http://www.research.att.com/areas/stat/xgobi/

às quais se pode aceder desde R através da biblioteca **xgobi**.

**Xgobi** está actualmente disponível para o ambiente X-Windows, tanto em UNIX como em Microsoft Windows, e existem interfaces com R disponíveis em ambos os sistemas.

# Apêndice A Um exemplo de sessão

A sessão seguinte pretende apresentar, usando-os, alguns aspectos do ambiente R. Muitos destes aspectos talvez sejam desconhecidos e, provavelmente, enigmáticos ao princípio, mas essa sensação não tardará em desaparecer. A sessão está elaborada para o sistema UNIX, sendo provável que os utilizadores em ambiente Microsoft Windows tenha de proceder a alguns pequenos ajustes.

Ligue o terminal e inicie o ambiente de janelas. Deverá ter o ficheiro '**morley.tab**' no directório de trabalho; caso não o tenha, deve copiá-lo (do directório base/data na estrutura de directórios de R) antes de iniciar a sessão de trabalho.

`$ R` Inicia o programa R, e aparece a mensagem inicial.

(Por comodidade e para evitar confusão, dentro de R não mostraremos o símbolo do sistema, na parte esquerda dos comandos)

`help.start()` Inicia o interface HTML para a ajuda sobre os comandos (utilizando o navegador WEB disponível no computador). Deverá fazer uma breve exploração das capacidades desta utilidade. Minimize a janela de ajuda e continue a sessão.

```
x <- rnorm(50)
```

`y <- rnorm(x)` Gera dois vectores aleatórios, contendo cada um deles 50 valores pseudo-aleatórios obtidos de uma distribuição normal (0,1), armazenando estes valores nos vectores x e y.

`plot(x,y)` Gera um gráfico de pontos no plano (x,y). Aparece automaticamente uma janela gráfica com o diagrama de dispersão.

`ls()` Apresenta a listagem dos nomes dos objectos existentes no momento actual no espaço de trabalho de R.

`rm(x,y)` Elimina os objectos x e y.

`x <- 1: 20` Cria o vector x com a sequência (1, 2,..., 20).

`w <- 1 + sqrt(x)/2` Cria o vector w com os desvios típicos ponderados.

```
dummy <- data.frame(x=x, y=x + rnorm(x)*w)
```

`dummy` Cria a folha de dados dummy com duas colunas, x e y, e visualiza-a.

```
fm <- lm(y ~ x, data=dummy)
```

`summary(fm)` Calcula a regressão linear simples de y sobre x e apresenta o resultado.

```
fm1 <- lm(y ~ x, data=dummy, weight=1/w^2)
```

`summary(fm1)` Já que se conhecem os desvios típicos, pode realizar-se uma regressão ponderada.

`attach(dummy)` Conecta a folha de dados, de modo que as suas colunas aparecem listadas como variáveis.

`rlf <- lowess(x,y)` Calcula uma regressão local não paramétrica.

`plot(x, y)` Diagrama de dispersão standard.

`lines(x, lrf$y)` Acrescenta a linha de regressão local não paramétrica ao gráfico.

`abline(0, 1, lty=3)` Acrescenta a verdadeira linha de regressão (intercepçao=0, declive=1) ao gráfico (lty=3: linha do tipo ponteado).

`abline(coef(fm))` Acrescenta a recta de regressão linear simples.

`abline(coef(fm1), col="red")` Acrescenta a recta de regressão ponderada (linha de cor vermelha).

`detach()` Remove a folha de dados do caminho de busca.

`plot(fitted(fm), resid(fm), xlab="Valores estimados", ylab="Resíduos", main="Resíduos versus Valores estimados")`
Um gráfico de diagnóstico da regressão para investigar a possível heteroscedasticidade. Acrescenta título e nomes aos eixos do gráfico.

`qqnorm(resid(fm), main=""Resíduos por quantil")`
Gráfico em papel probabilístico normal para comprovar assimetria, achatamento e dados anómalos (não é muito útil neste caso).

`rm(fm, fm1, lrf, x, dummy)` Elimina os objectos descriminados.

Na próxima sessão irão usar-se os dados clássicos de Michaelson e Morley para medir a velocidade da luz.

`file.show("morley.tab")` Visualiza o conteúdo do ficheiro. Opcional.

`mm <- read.table("morley.tab")`

`mm` Lê os dados do ficheiro "morley.tab" para uma folha de dados com o nome mm; visualiza o conteúdo de mm. Existem cinco experiências (coluna Expt) e cada uma contém 20 observações (identificados na coluna Run); a coluna Speed contém a velocidade da luz medida em cada caso, codificada numa unidade apropriada.

`mm$Expt <- factor(mm$Expt)`

`mm$Run <- factor(mm$Run)` Transforma as colunas Expt e Run em factores.

`attach(mm)` Conecta a folha de dados mm à posição 2 (por defeito) do caminho de busca.

`plot(Expt, Speed, main="Velocidade da luz", xlab="Experiencia No.")`
Compara as cinco experiências usando diagramas de extremos-e-quartis.

`fm <- aov(Speed~Run+Expt, data=mm)`

`summary(fm)` Analisa os dados como um delineamento em blocos aleatórios, considerando as experiências e as séries (número das observações) como os factores do ensaio.

`fm0 <- update(fm, .~. - Run)`

`anova(fm0, fm)` Ajusta um sub-modelo, omitindo 'Runs', e compara os dois modelos usando uma análise de variância formal.

`detach()`

`rm(fm, fm0)` Desconecta a folha de dados e elimina os objectos fm, fm0.

De seguida iremos apresentar algumas capacidades gráficas: gráficos do tipo contour image.

```
x <- seq(-pi, pi, len=50)
```

`y <- x` x e y são dois vectores cujos valores estão igualmente espaçados no intervalo $-\pi \le x \le \pi$.

```
f <- outer(x, y, function(x, y) cos(y)/(1+x^2))
```

f é uma matriz quadrada, com as linhas e as colunas indexadas por x e y, respectivamente, dos valores da função $\cos(y)/(1+x^2)$

```
oldpar <- par(no.readonly=TRUE)
```

`par(pty="s")` Guarda os argumentos gráficos na lista oldpar, e modifica o parâmetro pty (zona de desenho) para o formato "s" (quadrado).

```
contour(x, y, f)
contour(x, y, f, nlevels=15, add=TRUE)
```

Cria um gráfico de contorno (ou curvas de nível) da matriz f. Adiciona mais linhas para maior detalhe.

`fa <- (f -t(f))/2` fa é a 'parte assimétrica' de f. (t(f) é a transposta de f).

`contour(x, y, fa, nint=15)` Desenha um mapa de curvas de nível...

`par(oldpar)` ... e recupera os parâmetros gráficos originais.

```
image(x, y, f)
```

`image (x, y, fa)` Cria gráficos de alta densidade.

```
objects()
```

`rm(x, y, f, fa)` Listagem dos objectos existentes no espaço de trabalho; remove os objectos especificados antes de prosseguir.

Em R podem efectuar-se operações aritméticas com números complexos. 1i é a parte imaginária i.

```
th <- seq(-pi, pi, len=100)
z <- exp(1i*th)
par(pty="s")
```

`plot(z, type="l")` A representação gráfica de um número complexo consiste em representar a parte imaginária versus a parte real do valor. Neste caso, obtém-se um círculo.

```
w <- rnorm(100) + rnorm(100)*1i
```

Suponha que pretende gerar pontos pseudo-aleatórios dentro do círculo unitário. Uma primeira tentativa consiste em gerar valores complexos cujas partes real e imaginária, respectivamente, procedam de uma distribuição normal (0,1) ...

```
w <- ifelse(Mod(w)>1, 1/w,w)
```

... e de seguida, substituir os pontos que caiem fora do círculo pelos seus inversos.

```
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")
```

`lines(z)` Todos os pontos estão dentro do círculo unitário, mas a sua distribuição não é uniforme.

```
w <- sqrt(runif(100))*exp(2*pi*runif(100)*1i)
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")
```

`lines(z)` Este segundo método utiliza a distribuição uniforme. Neste caso, os pontos apresentam uma distribuição mais uniformemente espaçada dentro do círculo.

`rm(th, w, z)` De novo se eliminam os objectos.

`q()` Termina o programa R. O programa pergunta se pretende guardar o espaço de trabalho. Já que se trata apenas de uma sessão de apresentação, provavelmente responderá que não.

# Apêndice B Execução de R

## B.1 Execução de R em ambiente Unix

A ordem 'R' utiliza-se para executar o programa R, com possibilidade de dar várias opções complementares, da forma:

```
R [opções] [<entrada] [>saida]
```

(em que *entrada* e *saida* se referem aos nomes dos ficheiros, opcionais, de entrada e de saída), ou através do interface **R CMD**, para aceder a várias ferramentas de R (por exemplo, para processar arquivos com formato de documentação de R ou para manipular bibliotecas) que não estão desenvolvidas para serem usadas 'directamente'.

Muitas opções controlam o que ocorre ao iniciar e ao terminar uma sessão de R. O mecanismo de iniciação (utilize '***help(Startup)***' para mais informações) é o seguinte:

- A não ser que se especifique a opção '`--no-environ`', R procura o arquivo '***.Renviron***' no directório actual; se não o encontra, busca o arquivo que se especifica na variável de ambiente **R_ENVIRON**, e se esta variável não existe, procura o arquivo '***.Renviron***' no directório de entrada (*home*) do utilizador. O primeiro destes arquivos que for encontrado é executado, definindo as variáveis de ambiente. As variáveis são exportadas automaticamente, desde que sejam especificadas uma por linha, em linhas do tipo 'nome=valor'. As variáveis que se podem definir incluem **R_PAPERSIZE** (tamanho por defeito para o papel), **R_PRIMTCMD** (o comando de impressão por defeito), **R_LIB**S (para especificar o conjunto de directórios contendo as bibliotecas disponíveis), e **R_VSIZE** e **R_NSIZE** (veja adiante).
- De seguida, R procura o ficheiro que define o perfil de iniciação global, a não ser que na linha de comando se dê a opção '`--no-site-file`'. O nome deste ficheiro é definido pela variável **R_PROFILE**. Se esta variável não estiver definida, é assumido o ficheiro '**$R-HOME/etc./Rprofile**'.
- Seguidamente, a não ser que se especifique a opção '`--no-init-file`', R procura um ficheiro chamado '***.Rprofile***' no directório corrente, ou no directório de entrada do utilizador (nesta ordem), e executa-o.
- Se existe o ficheiro '***.Rdata***' no directório corrente, é executado (salvo se tiver dado a opção '`--no-restore`').
- Finalmente, se existe uma função designada '***.First()***', executa-a. Esta função, tal como a função '***.Last()***' que é executada ao sair do programa) pode ser definida nos ficheiros de entrada ou no ficheiro '***.RData***' atrás mencionados.

Além destas especificações, existem opções para controlar a memória disponível para a sessão de R (veja '***help(Memory)***' para mais informação). R utiliza um modelo de memória estático. Isto significa que, ao iniciar, o sistema operativo reserva uma quantidade fixa de

memória, que não pode alterar-se durante a execução. Assim, pode acontecer que não exista memória suficiente em determinado momento da sessão, por exemplo para carregar um ficheiro enorme de dados. As opções '`--nsize`' e '`--vsize`' (ou as variáveis de ambiente '**R_NSIZE**' e '**R_VSIZE**') podem usar-se para controlar a quantidade de memória disponível para objectos de tamanho fixo e variável.

As opções que se podem especificar na linha de comando são:

`--help`

`-h` Mostra uma pequena mensagem de ajuda e continua.

`--version` Mostra a informação da versão e continua.

`RHOME` Mostra o trajecto do directório inicial (home) de R e continua. Exceptuando os ficheiros de ajuda e o arquivo executável de chamada do programa, a instalação de R coloca todos os outros ficheiros (executáveis, bibliotecas, etc.) neste directório.

`--save`

`--no-save` Especifica se deve guardar ou não o espaço de trabalho ao terminar a sessão. Em modo interactivo, se nada se especificar, o programa pergunta se se pretende guardar ou não este espaço. Em processamento por lote, é obrigatório especificar uma destas opções.

`--no-environ` Não procura qualquer dos arquivos atrás mencionados para definir as variáveis de ambiente.

`--no-site-file` Não carrega o perfil global ao iniciar o programa.

`--no-init-file` Não carrega o perfil do utilizador ao iniciar o programa.

`--restore`

`--no-restore` Especifica se deve ou não recuperar o espaço de trabalho previamente guardado no arquivo '.Rdata' no directório corrente. Por defeito, é recuperado.

`--vanilla` Combina as opções '`--no-save`', '`--no-environ`', '`--no-site-file`', '`--no-init-file`' e '`--no-restore`'.

`--no-readline` Desactiva a edição de comandos através de readline. Esta opção deve utilizar-se quando se executa R em conjunto com Emacs utilizando a biblioteca **ESS** ("Emacs Speaks Statistics"). Veja Apêndice C [Editor de comandos], pág. 102, para mais informação.

`--v-size=N` Especifica a quantidade de memória reservada para objectos de tamanho variável, definindo o tamanho do 'vector heap' para N bytes. N deve ser um valor inteiro ou um valor inteiro terminando em 'M', 'K' ou 'k', que significam respectivamente 'Mega' (2^20), 'Kilo' (2^10) ou 'kilo' (1000) bytes.

`--n-size=N` Especifica a quantidade de memória reservada para objectos de tamanho fixo. São válidas as considerações feitas para '`--n-v-size`'.

`--quiet`
`--silent`

`-q` Não é mostrada a mensagem inicial de 'copyright'.

`--slave` Executa R com o mínimo de saídas possíveis. Esta opção é útil quando se utiliza R para efectuar cálculos cujos resultados são as entradas para outros programas.

`--verbose` Mostra o máximo de saídas possíveis, e além disso, coloca a opção *verbose=TRUE*. R utiliza esta opção para controlar se deve apresentar mensagens de diagnóstico.

`--debugger=depurador`

`- d depurador` Executa R a partir do programa de depuração 'depurador'. Caso existam outras opções na linha de comando, são ignoradas; Qualquer outra opção, se necessária, deve dar-se quando se inicia R a partir do programa de depuração.

`--gui=tipo` Utiliza 'tipo' como o interface gráfico (note-se que também inclui os gráficos interactivos). Os valores possíveis para 'tipo' são **X11** (por defeito) e **GNOME**, desde que este suporte esteja disponível.

Note que é possível re-direccionar a entrada (`<entrada`) e a saída (`>saida`) .

R CMD permite utilizar diversas ferramentas que são úteis em utilização conjunta com R, mas que não estão concebidas para serem usadas directamente a partir da linha de comando. A forma geral de usá-las é:

```
R CMD comando argumentos
```

onde '*comando*' é o nome da ferramenta ou aplicação e '*argumentos*' são os argumentos que se pretendem passar a essa aplicação.

As ferramentas disponíveis são:

`BATCH` Executa R em processamento por lotes.

`COMPILE` Compila arquivos para usar com R.

`SHLIB` Constrói bibliotecas partilhadas do sistema operativo para carregamento dinâmico.

`INSTALL` Instala bibliotecas.

`REMOVE` Remove bibliotecas.

`build` Constrói bibliotecas.

`check` Verifica bibliotecas.

`Rdconv` Converte ficheiros do formato Rd para outros formatos, incluindo HTML, Nroff, LATEX, texto ASCII sem formato, e formato S.

| | |
|---|---|
| `Rd2dvi` | Converte ficheiros do formato Rd para o formato DVI/PDF. |
| `Rd2txt` | Converte ficheiros do formato Rd para o formato texto. |
| `Rdindex` | Extrai a informação para os índices dos ficheiros Rd. |
| `Sd2Rd` | Converte ficheiros de formato S em formato Rd. |

As primeiras cinco aplicações (BATCH, COMPILE, SHLIB, INSTALL, REMOVE) podem ser executadas directamente sem usar a opção CMD, isto é, na forma:

```
R comando argumentos
```

Faça o comando:

```
R CMD comando --help
```

para obter mais informações relacionadas com cada uma destas ferramentas.

## B.2 Execução de R em ambiente Microsoft Windows

O procedimento de inicio em Microsoft Windows é muito similar ao descrito para ambiente UNIX, mas não necessariamente idêntico. Existem duas versões de R para Windows: uma baseada em janelas do tipo MDI (cujo programa executável é **Rgui.exe**) e outra versão destinada a correr em modo terminal ou janela de DOS (cujo programa executável é **Rterm.exe**, mais vocacionada para processamento por lotes).

Existem várias opções para controlar o que ocorre ao iniciar e ao terminar uma sessão de R. O mecanismo de arranque (utilize '***help(Startup)***' para informação mais detalhada) é descrito em seguida. As referências ao '*directório inicial*' (*home*) devem ser clarificadas, pois esta noção nem sempre está definida em Windows. Se a variável de ambiente **R_USER** está definida, esta define qual o directório inicial. Caso contrário, este é definido pela variável de ambiente **HOME**, se está definida. Se não, será definido pelas variáveis **HOMEDRIVE** e **HOMEPATH** (em ambiente Windows NT). Se nenhuma destas variáveis estiver definida, então o directório inicial é o directório a partir do qual se iniciar o programa.

- A não ser que se especifique a opção '`--no-environ`', R procura o ficheiro '`.Renviron`' no directório actual; não o encontrando, procura-o no directório inicial do utilizador. Se encontra algum destes arquivos, executa-o e define as variáveis de ambiente. As variáveis são exportadas automaticamente, desde que sejam especificadas uma por linha, em linhas do tipo 'nome=valor'. As variáveis que se podem definir incluem **R_PAPERSIZE** (tamanho por defeito para o papel), **R_PRIMTCMD** (o comando de impressão por defeito), **R_LIBS** (para especificar o conjunto de directórios contendo as bibliotecas disponíveis), e **R_VSIZE** e **R_NSIZE** (veja adiante).
- De seguida, R procura o ficheiro que define o perfil de iniciação global, a não ser que na linha de comando se dê a opção '`--no-site-file`'. O nome deste ficheiro é definido pela variável **R_PROFILE**. Se esta variável não estiver definida, é assumido o ficheiro '**$R-HOME/etc./Rprofile**'.
- Seguidamente, a não ser que se especifique a opção '`--no-init-file`', R procura um ficheiro chamado '**.Rprofile**' no directório corrente, ou no directório de entrada do utilizador (nesta ordem), e executa-o.
- Se existe o ficheiro '**.Rdata**' no directório corrente, é executado (salvo se tiver dado a opção '`--no-restore`').
- Finalmente, se existe uma função designada '**.First()**', executa-a. Esta função, tal como a função '**.Last()**' que é executada ao sair do programa) pode ser definida nos ficheiros de entrada ou no ficheiro '**.RData**' atrás mencionados.

Além destas especificações, existem opções para controlar a memória disponível para a sessão de R (veja '***help(Memory)***' para mais informação). R utiliza um modelo de memória estático. Isto significa que, ao iniciar, o sistema operativo reserva uma quantidade fixa de memória, que não pode alterar-se durante a execução. Assim, pode acontecer que não exista

memória suficiente em determinado momento da sessão, por exemplo para carregar um ficheiro enorme de dados. As opções '`--nsize`' e '`--vsize`' (ou as variáveis de ambiente '**R_NSIZE**' e '**R_VSIZE**') podem usar-se para controlar a quantidade de memória disponível para objectos de tamanho fixo e variável.

As opções que se podem especificar na linha de comando são:

`--version` Mostra a informação da versão e continua.

`--mdi`
`--sdi`

`--no-mdi` Controla se Rgui será executado como um programa MDI (por defeito), onde cada nova janela aberta está contida dentro da janela principal, ou como um programa SDI, em que cada janela (consola, gráficos e resultados) aparece de modo independente no escritório.

`--save`

`--no-save` Especifica se deve guardar ou não o espaço de trabalho ao terminar a sessão. Em modo interactivo, se nada se especificar, o programa pergunta se se pretende guardar ou não este espaço. Em processamento por lote, é obrigatório especificar uma destas opções.

`--restore`

`--no-restore` Especifica se deve ou não recuperar o espaço de trabalho previamente guardado no arquivo '**.Rdata**' no directório corrente. Por defeito, é recuperado.

`--no-site-file` Não carrega o perfil global ao iniciar o programa.

`--no-init-file` Não carrega o perfil do utilizador ao iniciar o programa.

`--no-environ` Não procura qualquer dos arquivos atrás mencionados para definir as variáveis de ambiente.

`--vanilla` Combina as opções '`--no-save`', '`--no-environ`', '`--no-site-file`', '`--no-init-file`' e '`--no-restore`'.

`-q`
`--quiet`

`--silent` Não é mostrada a mensagem inicial de 'copyright'.

`--slave` Executa R com o mínimo de saídas possíveis.

`--verbose` Mostra o máximo de saídas possíveis.

`--ess` Prepara Rterm para uso em modo R-inferior em ESS.

# Apêndice C Editor de comandos

## C.1 Preliminares

Se a biblioteca de GNU **readline** está disponível quando se instala e configura R em ambiente UNIX, fica disponível um editor de comandos interno que permite recuperar, editar e voltar a executar as ordens previamente utilizadas.

Este editor pode desactivar-se com a opção '`--no-readline`' ao iniciar o programa (o que permite utilizar ESS[1]).

A versão para Microsoft Windows dispõe de um editor de comandos mais fácil; veja o tópico '**Console**' no menu '**Help**' do programa **Rgui**.

Quando se utiliza R com as capacidades de edição de readline, as opções descritas de seguida ficam disponíveis.

Muitas das ordens do editor de comandos utilizam caracteres *Control* e *Meta*. Os caracteres Control, tais como Control-m, obtêm-se mantendo carregada a tecla <CTRL> enquanto se carrega a tecla <m>, e de seguida será representado pela notação C-m. Os caracteres Meta, tais como Meta-b, obtêm-se carregando a tecla <META> e de seguida (após soltar) a tecla <b>, e será representado pela notação M-b. Se o teclado não tem a tecla <META> podem obter-se os caracteres Meta com a sequência de duas teclas que começa com a tecla ESC. Isto é, para obter M-b, deverá fazer <ESC> <b>.As sequências ESC também podem realizar-se nos teclados com a tecla <META>. Deve ter-se em atenção que os caracteres Meta distinguem entre minúsculas e maiúsculas.

## C.2 Acções de edição

O programa R conserva o historial de comandos que se executam, incluindo as linhas de erro, o que permite recuperar as linhas de comandos anteriores, modificá-las se tal for necessário, e tornar a executá-las como novas ordens. No estilo de edição emacs qualquer caracter que se digite é inserido na posição do cursor, arrastando os caracteres à direita do cursos. No estilo de edição vi o modo de inserção de caracteres é iniciado pela sequência M-i ou M-a, seguindo-se a inserção de caracteres; o modo de inserção é terminado carregando a tecla <ESC>.

Quando se carrega a tecla <RET>, a ordem em edição é executada.

De seguida resumem-se algumas das acções possíveis com o editor de comandos. É pena que não se consigam mostrar algumas capacidades, tais como o arrastamento do cursor com as teclas direccionais.

---

[1] Abreviatura do editor de texto 'Emacs Speaks Statistics'. Veja a direcção URL http://ess.stat.wisc.edu/

## C.3 Resumo do editor de linha de comandos

### Recuperação dos comandos anteriores e deslocamentos verticais

| | |
|---|---|
| C-p | Recupera o comando anterior (retrocede no histórico de comandos). |
| C-n | Recupera o comando posterior (avança no histórico de comandos). |
| C-r texto | Recupera o último comando que contém 'texto'. |

Na maior parte dos terminais, é possível utilizar as teclas direccionais verticais 'seta para cima' e 'seta para baixo' em vez das sequências C-p e C-n, respectivamente.

### Movimentos laterais do cursor

| | |
|---|---|
| C-a | Vai para o início da linha. |
| C-e | Vai para o fim da linha. |
| M-b | Retrocede uma palavra. |
| M-f | Avança uma palavra. |
| C-b | Retrocede um caracter. |
| C-f | Avança um caracter. |

Na maior parte dos terminais, é possível utilizar as teclas direccionais horizontais 'seta para a direita' e 'seta para esquerda' em vez das sequências C-b e C-f, respectivamente.

### Edição

| | |
|---|---|
| texto | Insere 'texto' na posição do cursor. |
| C-f texto | Insere 'texto à frente do cursor. |
| <DEL> | Elimina o caracter antes (à esquerda) do cursor. |
| C-d | Elimina o caracter na posição do cursor. |
| M-d | Elimina o resto da palavra desde a posição do cursor, e guarda a parte eliminada. |
| C-k | Elimina o resto da linha desde a posição do cursor, e guarda a parte eliminada. |
| C-y | Insere o último texto guardado. |
| C-t | Troca o caracter na posição do cursor com o seguinte. |
| M-l | Substitui o resto da palavra por minúsculas. |
| M-c | Substitui o resto da palavra por maiúsculas. |
| <RET> | Executa o comando em edição.<br>Ao carregar <RET> termina-se a edição da linha de comando. |

# Apêndice D Índice de funções e variáveis

# Apêndice E Índice de conceitos

# Apêndice F Referências

D.M.Bates e D.G.Watts (1988), *Nonlinear Regression Analysis and Its Applications*. John Wiley & Sons, New York.

Richard A. Becker, John M. Chambers e Allan R. Wilks (1988), *The New S Language*. Chapman & Hall, New York. (Este livro é frequentemente designado por "Blue Book").

John M. Chambers e Trevor J. Hastie, Editores (1992), *Statistical Models in S*. Chapman & Hall, New York. (Este livro é frequentemente designado por "White Book").

Annette J. Dobson (1990), *An Introduction to Generalized Linear Models*. Chapman & Hall, London.

Peter McCullagh e John A. Nelder (1989), *Generalized Linear Models*. Second edition, Chapman & Hall, Londosn.

John A. Rice (1995), *Mathematical Statistics and Data Analysis*. Second edition, Duxbury Press, Belmont, CA.

S. D. Silvey (1970), *Statistical Inference*. Penguin, London.