

Willian Luís de Oliveira  
Katiane Silva Conceição

# Introdução ao R

São Carlos - SP  
Agosto/2015



UNIVERSIDADE DE SÃO PAULO  
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO  
DEPARTAMENTO DE MATEMÁTICA APLICADA E ESTATÍSTICA

# Introdução ao R

**Willian Luís de Oliveira**

Universidade Federal de São Carlos

Departamento de Estatística

Programa de Pós-Graduação em Estatística

e-mail: `willian26oliveira@gmail.com`

**Katiane Silva Conceição**

Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Departamento de Matemática Aplicada e Estatística

e-mail: `katiane@icmc.usp.br`

**Colaboradores**

**Prof. Dr. Mário de Castro**

**Prof. Dr. Marinho Gomes de Andrade Filho**

São Carlos - SP

Agosto/2015

# Prefácio

Esta apostila faz parte do material didático do minicurso “Introdução ao R”, ministrado no **XVIII Simpósio de Matemática para a Graduação** organizado pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, *campus* de São Carlos.

O principal objetivo deste minicurso é fazer uma breve apresentação do *software* R aos estudantes de graduação. Um incentivo ao uso do programa é que o mesmo é gratuito, possui uma linguagem simples e bastante poderosa.

Esperamos cumprir o nosso objetivo e estamos à disposição para qualquer crítica e/ou sugestão que venham contribuir para a melhoria deste trabalho.

Willian Oliveira e Katiane Conceição  
São Carlos, 2015.

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Ambiente R</b>	<b>2</b>
2.1	Como instalar? . . . . .	2
2.2	Uso do R . . . . .	2
2.3	Sistema de ajuda (help) . . . . .	4
2.4	Tinn-R . . . . .	4
<b>3</b>	<b>Operações Matemáticas</b>	<b>5</b>
<b>4</b>	<b>A Organização de Dados no R</b>	<b>11</b>
4.1	Vetor . . . . .	12
4.2	Matriz . . . . .	19
4.3	Ficha de dados ( <i>data frame</i> ) . . . . .	25
4.4	Lista . . . . .	26
4.5	Expressando dados faltantes . . . . .	27
<b>5</b>	<b>Manipulando Dados</b>	<b>28</b>
5.1	Extraindo elementos . . . . .	28
5.2	Operações com vetores e matrizes . . . . .	36
5.3	Leitura de dados em arquivos . . . . .	40
5.4	Escrita de dados em arquivo . . . . .	42
5.5	Função <code>attach</code> . . . . .	44
<b>6</b>	<b>Distribuições de Probabilidade</b>	<b>46</b>
6.1	Função <code>sample</code> . . . . .	51
<b>7</b>	<b>Funções Estatísticas Básicas</b>	<b>54</b>
7.1	Estatística descritiva . . . . .	54
7.2	Tabela de frequência . . . . .	56
7.2.1	Tabela de frequências a partir de fatores . . . . .	56
7.2.2	Tabela de frequências relativas . . . . .	58
7.2.3	Tabela de frequências por classe . . . . .	60

7.3	Operações repetidas: função <code>apply</code>	62
<b>8</b>	<b>Gráficos</b>	<b>63</b>
8.1	Histograma	63
8.2	Gráfico de caixa ( <i>box-plot</i> )	64
8.3	Série temporal	64
8.4	Gráfico de dispersão	65
8.5	Gráfico de dispersão de pares de variáveis	66
8.6	Gráfico de barras	66
8.7	Gráfico de setores ou pizza	66
8.8	Desenhando símbolos em um gráfico	67
8.9	Argumentos das funções gráficas	67
8.10	Funções <code>points</code> e <code>lines</code> : adicionando pontos ou linhas ao gráfico corrente	69
8.11	Funções <code>polygon</code> e <code>text</code> : adicionando polígono ou texto ao gráfico corrente	69
8.12	Função <code>abline</code> : adicionando uma linha ao gráfico corrente	70
8.13	Função <code>legend</code> : adicionando legenda ao gráfico corrente	70
8.14	Funções gráficas interativas	71
<b>9</b>	<b>Funções</b>	<b>72</b>
9.1	Sintaxe geral	72
9.2	Expressando condições	73
9.3	Iteração	74
9.3.1	for	74
9.3.2	while	75
9.3.3	repeat	75
<b>10</b>	<b>R na internet</b>	<b>76</b>

# Capítulo 1

## Introdução

A linguagem de programação R tornou-se nos últimos anos uma das mais populares no meio acadêmico, principalmente entre estudantes e pesquisadores da área de Estatística. Isso se deve primeiramente ao fato de o R ser um *software* livre e de fácil acesso. A linguagem de programação R no padrão estruturado torna fácil escrever um programa. O número de usuários do R cresceu rapidamente em todo mundo, aumentando assim os “pacotes” desenvolvidos e disponibilizados por estes usuários. Como consequência dessa característica, cresce constantemente a demanda por cursos introdutórios desse *software*.

Este material visa apresentar uma introdução da linguagem R para estudantes que estão iniciando seus estudos de graduação em qualquer área que exija o uso de estatística. Portanto, este minicurso tem como objetivo principal introduzir os estudantes na linguagem R através dos comandos básicos para uma análise estatística descritiva de uma massa de dados.

Iniciamos experimentando o *software* R para ter uma ideia de seus recursos e a sua forma de trabalhar. Para isto, executamos e estudamos os comandos e os seus resultados para nos familiarizar com o programa. Nas seções seguintes veremos com mais detalhes o uso do *software* R.

# Capítulo 2

## Ambiente R

O *software* R é um programa gratuito que permite fazer operações numéricas, vetoriais e matriciais, além de ser uma linguagem de programação. Possui características como manipulação e armazenamento dos dados, operações sobre variáveis indexadas, ferramentas para análise de dados e uma vasta biblioteca de funções gráficas.

### 2.1 Como instalar?

O programa pode ser obtido gratuitamente no *site* <http://www.r-project.org>. Neste *site* pode-se obter mais informações sobre o R, bem como pacotes, o projeto R e documentação (manuais e livros). Atualizações do sistema são feitas regularmente por pesquisadores de toda a parte do mundo. A *versão 3.2.2* disponível desde 14-08-2015 é a mais recente.

### 2.2 Uso do R

Como um primeiro contato, inicie o R em seu computador para ter uma ideia de seus recursos e sua forma de trabalhar (Figura 2.1). Este ambiente pode ser modificado de acordo com a preferência de cada usuário, seguindo os passos na barra de Menu do R:

Editar → Preferências da interface gráfica

Dessa forma, pode-se alterar a interface do R, como exemplos, o tipo de fonte, tamanho, cor, entre outras opções disponíveis.

Pode-se observar no ambiente da Figura 2.1 o símbolo “>”, que é um indicativo de que o R aguarda a entrada de comandos. Um desses comandos pode ser adicionar um objeto (que será descrito posteriormente), uma operação, leitura de dados (com extensão adequada), os quais serão discutidos com mais detalhes nos próximos capítulos. O R tenta interpretar tudo o que é digitado na linha de comando seguido de **ENTER**.

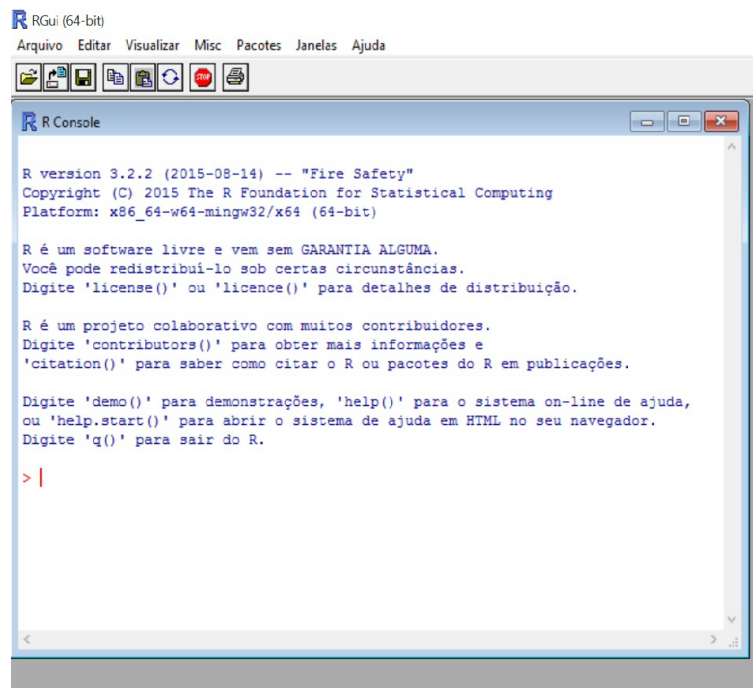


Figura 2.1: Ambiente R.

Abaixo listamos alguns comandos úteis que serão muito utilizados:

### Escrever comentários no programa:

```
> #
```

### Listar objetos:

```
> ls()
```

ou ainda,

```
> objects()
```

### Remover objetos:

- *Especificando o objeto a ser excluído:*

```
> rm("nome do objeto")
```

- *Excluindo todos os objetos simultaneamente:*

```
> rm(list = ls(all = TRUE))
```

### Limpar tela:

```
> ctrl + L
```

### Sair do R:

```
> q()
```

Esses comandos podem também ser encontrados na barra de Menu do R.



## 2.3 Sistema de ajuda (help)

O comando `help` serve para obter informação sobre uma função específica. Por exemplo, para obter informação sobre a função `sin` (seno) pode utilizar-se qualquer uma das opções:

```
> help(sin)
> help("sin")
> help('sin')
> ?sin
```

O comando `help.search()` permite pesquisar uma sequência de caracteres. Por exemplo,

```
> help.search("solve system")
```

Ao digitar no R o comando `help.start()`, sem a especificação de uma sequência de caracteres, uma página em seu navegador com informações diversas sobre o R será exibida (ver também o menu Help no ambiente de trabalho).

## 2.4 Tinn-R

Tinn-R é um substituto pequeno e simples, mas bastante eficaz, para o editor de código de base fornecido pelo Rgui.

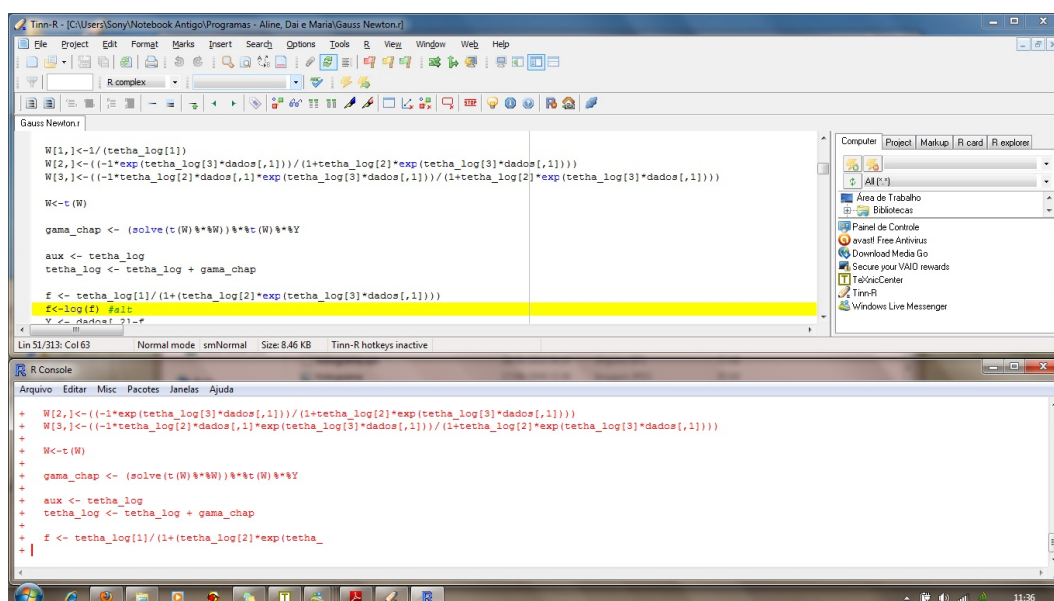


Figura 2.2: Tinn-R.

## Capítulo 3

# Operações Matemáticas

A maioria dos dados que analisamos são numéricos e estes são frequentemente manipulados usando funções e operadores matemáticos. Iniciaremos o uso do R apresentando os operadores básicos que podem ser utilizados. A Tabela 3.1 exibe os operadores matemáticos, lógicos e de comparação no R.

Tabela 3.1: Operações no R.

Operadores Aritméticos	
+	adição
-	subtração
*	produto escalar
/	divisão
^ ou **	potência
% %	resto da divisão
% / %	inteiro da divisão
Operadores de Comparação	
==	igual a
!=	não igual a ou "diferente de"
<	menor do que
>	maior do que
<=	menor do que ou igual a
>=	maior do que ou igual a
Operadores Lógicos	
!	negação
	ou
	ou sequencial (para avaliar condições)
&	e
&&	e sequencial (para avaliar condições)

Para ilustrar o uso dos operadores aritméticos, seguem abaixo alguns exemplos.

```
> 1 + 2 + 3      # soma estes números
[1] 6

> 5 - 3          # subtrai estes números
[1] 2

> 2 * 5          # multiplica estes números
[1] 10

> 17 / 5         # divide estes números
[1] 3.4

> pi / 2
[1] 1.570796

> 2 + 3 * 4      # prioridade de operação:  multiplicação
[1] 14

> (2 + 3) * 4    # prioridade de operação:  o que está entre ()
[1] 20

> 3 / 2 + 1      # prioridade de operação:  divisão
[1] 2.5

> 2 ** 3         # potências são indicadas por ** ou ^
[1] 8

> 4 * 3 ** 3     # prioridade de operação:  potência
[1] 108

> 17 %% 5        # resto da divisão
[1] 2

> 17 %/% 5       # inteiro da divisão
[1] 3
```

**Observação:** O separador decimal é um ponto (.). Exemplo:

```
> 2.5 - 3.8
[1] -1.3
```

Os operadores de comparação também operam em cada elemento, retornando valores lógicos TRUE (T) ou FALSE (F). Alguns exemplos:

```
> 5 > 2                # compara esses valores
[1] TRUE

> 12 <= 4
[1] FALSE

> 7 == 10
[1] FALSE

> 7 = 10
Erro em 7 = 10 :  lado esquerdo da atribuição inválida (do_set)

> 8 != 8
[1] FALSE

> 10 != 20 / 2
[1] FALSE
```

Os operadores lógicos são utilizados para combinar expressões que retornam valores lógicos, como aquelas formadas por operadores de comparação, e a expressão final retorna um valor lógico. Abaixo seguem alguns exemplos:

```
> ! (6 == 8 - 2)        # nega o resultado
[1] FALSE

> (5 > 2) & (7 < 10)     # verdadeiro E verdadeiro = verdadeiro
[1] TRUE

> (2 < 8) | (6 < 9)      # verdadeiro OU verdadeiro = verdadeiro
[1] TRUE

> (7 == 10) & (4 > 8 / 2) # falso E falso = falso
[1] FALSE

> (6 != 12 / 2) | (1 >= 4) # falso OU falso = falso
[1] FALSE

> (3 >= 1) & (4 < 2)      # verdadeiro E falso = falso
[1] FALSE

> (6 < 14) | (3 >= 5)     # verdadeiro OU falso = verdadeiro
[1] TRUE
```

Existem funções matemáticas no R que operam com quantidades numéricas. Estas funções fazem transformações numéricas ou operações matemáticas com os dados. Na Tabela

3.2 são apresentadas as funções matemáticas mais utilizadas.

Tabela 3.2: Funções matemáticas no R.

Funções Simples	
<code>sqrt</code>	raiz quadrada
<code>abs</code>	valor absoluto
<code>exp</code>	exponenciação
<code>log</code>	logaritmo natural
<code>log2</code>	logaritmo base 2
<code>log10</code>	logaritmo base 10
<code>factorial</code>	fatorial
<code>lfactorial</code>	logaritmo natural do fatorial
<code>choose</code>	combinação de x, k a k
<code>gamma</code>	gama
<code>beta</code>	beta
Funções de Arredondamento	
<code>ceiling</code>	menor inteiro maior do que ou igual ao número
<code>floor</code>	maior inteiro menor do que ou igual ao número
<code>trunc</code>	trunca o valor, retornando um inteiro
<code>round</code>	arredonda os valores no objeto numérico
Funções Trigonômicas	
<code>sin</code>	seno
<code>cos</code>	cosseno
<code>tan</code>	tangente
<code>asin</code>	arco seno
<code>acos</code>	arco cosseno
<code>atan</code>	arco tangente

Apresentamos a seguir exemplos das funções apresentadas na Tabela 3.2.

```
> abs(-58)
[1] 58

> sqrt(2)
[1] 1.414214

> exp(4)
[1] 54.59815

> log(5)                # logaritmo natural: base e
```

```
[1] 1.609438

> log2(5)                # logaritmo base 2
[1] 2.321928

> log10(5)               # logaritmo base 10
[1] 0.69897

> factorial(5)           # 5!
[1] 120

> lfactorial(3)          # logaritmo natural de 3!
[1] 1.791759

> choose(8,2)            # combinação de 8, 2 a 2
[1] 28

> gamma(4)               # função gama - n inteiro, gamma(n) = (n-1)!
[1] 6

> gamma(5.67)            # função gama - n contínuo, calculado pela integral
[1] 69.0305

> beta(4,2)              # função beta -  $n_1$  e  $n_2$  inteiro
[1] 0.05

> beta(12.9,3.51)        # função beta -  $n_1$  e  $n_2$  contínuo
[1] 0.0003092307

> ceiling(4.2)           # menor inteiro maior ou igual a 4.2
[1] 5

> ceiling(7)             # menor inteiro maior ou igual a 7
[1] 7

> floor(4.8)             # maior inteiro menor ou igual 4
[1] 4

> trunc(9.67)            # retorna apenas a parte inteira de 9.67
[1] 9

> round(76.3094827,2)    # arredondamento com duas casas decimais
[1] 76.31

> round(12.5786432,0)    # arredondamento com zero casa decimal
[1] 13
```

```
> sin(pi / 4)           # seno de  $\frac{\pi}{4}$ 
[1] 0.7071068

> acos(0.8)             # arco cujo cosseno é 0.8
[1] 0.6435011
```

**Observações:**

**1** - O logaritmo em qualquer base,  $\log_n(x)$ , pode ser calculado usando a função `log(x, base=n)` ou simplesmente `log(x, n)`.

**2** - Nas Funções trigonométricas, a informação de ângulo deverá ser em radianos, não em graus (i.e., um ângulo reto é  $\frac{\pi}{2}$ ).

**3** - Existem também no R funções trigonométricas hiperbólicas (`sinh`, `cosh`, `tanh`) e hiperbólicas inversas (`asinh`, `acosh`, `atanh`).

# Capítulo 4

## A Organização de Dados no R

O programa estatístico R dispõe de uma linguagem de programação orientada a objetos. Neste tipo de linguagem pode-se acessar os dados armazenados na memória. O R não dá acesso direto à memória do computador, mas oferece uma série de estruturas de dados especializadas, denominadas objetos. Esses objetos são referenciados por meio de símbolos ou variáveis. O resultado de uma expressão pode ser atribuído a um objeto usando um dos operadores de atribuição: “objeto = expressão” , “objeto <- expressão” ou “expressão -> objeto”. Exemplos:

```
> x = 15          # atribuindo ao objeto x o número 15
> x              # visualizar o objeto x
[1] 15

> 13 - 7 -> y     # atribuindo ao objeto y o resultado de 13-7
> y              # visualizar o objeto y
[1] 6

> z <- ((16 + 5) / 3) ^ 2
> z
[1] 49
```

**Observação:** O R diferencia letras minúsculas de maiúsculas. Logo, objetos criados como `x` e `X` serão diferentes.

Os elementos de um objeto podem ser dos seguintes modos:

`logical`: Modo binário, com valores representados por `TRUE` (T) ou `FALSE` (F).

`numeric`: Números reais.

`complex`: Números complexos (parte real e imaginária).

`character`: Caracteres.



O tipo de dado irá depender da sua natureza: uma simples sequência de números é mais facilmente representada como um vetor, enquanto dados de um planilha (colunas representando variáveis e linhas representando indivíduos) são melhor representados como uma matriz ou *data frame*. O *software* R possui uma grande variedade de tipos de dados. Concentramos nossa atenção em quatro tipos: vetor, matriz, ficha de dados (*data frame*) e lista. A seguir veremos mais detalhes de cada um dos tipos de dados e como criá-los no R.

## 4.1 Vetor

Um vetor é um conjunto de elementos em uma ordem específica. A ordem é especificada quando você cria o vetor (usualmente a ordem na qual você digita ou lê os dados) e isto é importante porque você pode se referir a um elemento unicamente pela sua posição no vetor. Todos os elementos de um vetor devem ser de um único modo. Os vetores numéricos são os mais usuais. Como exemplos, os números (vetor de comprimento unitário):

```
> x <- 10          # o objeto x é um vetor de dimensão 1
> y <- x / 2
> z <- x + y
```

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

```
> a <- -17 + 3i     # o objeto a é número complexo
> a
[1] -17+3i
```

Ainda podemos criar um vetor das seguintes maneiras:

a) `scan()`

Esta função coloca o R em modo de entrada de dados. O usuário deve digitar cada dado seguido da tecla **ENTER**. Para encerrar, digitar **ENTER** duas vezes. Exemplo:

```
> z <- scan()
1:  2
2: 45
3:  6 4: 78
5:
Read 4 items
> z          # imprime o vetor z
```

```
[1] 2 45 6 78
```

Você ainda pode criar um vetor de caracteres (como nomes, letras, siglas) e lógico usando a função `scan` com os argumentos `what = character()` e `what = logical()`, respectivamente. Exemplos:

```
> alunos <- scan(what = character())
```

```
1: João
```

```
2: Paulo
```

```
3: Ana
```

```
4: Maria
```

```
5:
```

```
Read 4 items
```

```
> alunos
```

```
[1] "João" "Paulo" "Ana" "Maria"
```

```
> resposta <- scan(what = logical())
```

```
1: F
```

```
2: F
```

```
3: T
```

```
4:
```

```
Read 3 items
```

```
> resposta
```

```
[1] FALSE FALSE TRUE
```

**Observação:** Ao utilizar a função `scan`, todos os elementos podem ser incluídos na mesma linha, separados por espaço. Exemplos:

```
> Z <- scan()
```

```
1: 3 15 7 0
```

```
5:
```

```
Read 4 items
```

```
> Z
```

```
[1] 3 15 7 0
```

```
> sexo <- scan(what = character())
```

```
1: F F M F M M
```

```
7:
```

```
Read 6 items
```

```
> sexo
```

```
[1] "F" "F" "M" "F" "M" "M"

> gabarito <- scan(what = logical())
1:  F F T F T F F
8:  T T F
11: Read 10 items
> gabarito
[1] FALSE FALSE TRUE FALSE TRUE FALSE FALSE TRUE TRUE FALSE
```

### b) c()

Outro modo de criar um vetor é usar a função `c()`, que concatena os números ou caracteres associados a um objeto. Exemplos:

```
> x <- c(92,4,55,6,4)      # vetor x
> x                        # imprime o vetor x
[1] 92 4 55 6 4

> alunos <- c("João","Paulo","Ana","Maria","Carlos")
> alunos
[1] "João" "Paulo" "Ana" "Maria" "Carlos"
```

### Observações:

- 1 - Podemos especificar cada elemento do vetor de caracteres por ' ', ao invés de " ".
- 2 - Podemos representar também dados categóricos no R, que são observações vindas de um número finito de categorias. Esses dados são representados no R por meio de fatores através da função `factor`. Exemplos:

```
> sexo <- factor(c("F","M","F","F","M"))
> sexo
[1] F  M  F  F  M
Levels:  F  M

> sucesso <- factor(c(0,1,1,0,1,0,0,0,1,0))
> sucesso
[1] 0 1 1 0 1 0 0 0 1 0
Levels:  0 1

> moeda <- factor(c('cara','coroa','coroa','cara','coroa'))
> moeda
[1] cara coroa coroa cara coroa
Levels:  cara coroa
```

c) `seq()` ou “ : ”

Em R existem várias funções para gerar sucessões ou sequências numéricas. Um exemplo mais simples, criar um vetor contendo números de 1 a 6:

```
> x <- 1:6
> x
[1] 1 2 3 4 5 6
```

O operador “ : ” tem prioridade máxima em uma expressão. Vejamos:

```
> 2 * 1:15      # o produto entre um número e um vetor = um vetor
[1] 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
```

A expressão `7:2` constrói uma sequência decrescente. Exemplo:

```
> y <- 7:2
> y
[1] 7 6 5 4 3 2
```

A função `seq` permite gerar sequências mais complexas e dispõe de cinco argumentos embora não se utilizem todos simultaneamente. Os dois primeiros argumentos podem ser especificados pelo nome, mediante a indicação `from = valor inicial` e `to = valor final`. O resultado é equivalente ao do operador “ : ”. Os dois argumentos seguintes são `by = incremento` e `length = comprimento`, que especificam o incremento entre dois valores sucessivos e o comprimento da sucessão, respectivamente. Deixando de especificar esses argumentos, a função `seq` segue como *default* `from`, `to`, `by`. Alguns exemplos:

```
> seq(from = 1, to = 6)
[1] 1 2 3 4 5 6

> seq(6,1)
[1] 6 5 4 3 2 1

> s <- seq(from = -5, to = 5, by = 0.5)
> s
[1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0 2.5
[17] 3.0 3.5 4.0 4.5 5.0

> s2 <- seq(-5,5,1.5)
> s2
[1] -5.0 -3.5 -2.0 -0.5 1.0 2.5 4.0
```

```
> v <- seq(from = 0, to = 45, length = 5)
> v
[1] 0.00 11.25 22.50 33.75 45.00
```

**Observação:** Especificando os argumentos da função `seq` pelo nome, torna-se irrelevante a sua ordem. Gerando a mesma sequência:

```
> v2 <- seq(length = 5, from = 0, to = 55)
> v2
[1] 0.00 11.25 22.50 33.75 45.00

> w <- seq(length = 15, from = -5, by = 0.2)
> w
[1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.0 -2.8 -2.6 -2.4 -2.2
```

O quinto argumento desta função é `along = vetor`, e se for usado, deve ser o único parâmetro especificado, e cria a sequência de mesmo comprimento que o vetor especificado ou uma sucessão vazia se o vetor especificado é vazio (o que pode acontecer).

```
> k <- c(2,5,3,7)

> seq(1, 9, along = k)      # sequência de 1 a 9 com mesmo comprimento de k
[1] 1.000000 3.666667 6.333333 9.000000

> k2 <- c(4,7,8,4,34,5)

> seq(1, 9, along = k2)     # sequência de 1 a 9 com mesmo comprimento de k2
[1] 1.0 2.6 4.2 5.8 7.4 9.0
```

#### d) `rep()`

Uma outra função relacionada com `seq` é a função `rep`, que pode ser usada para replicar um objeto de diversas maneiras. A forma mais simples é:

```
> t <- rep(4, times = 5)    # replica o número 4 cinco vezes
> t
[1] 4 4 4 4 4

> rep(1,5)                  # replica o número 1 cinco vezes
[1] 1 1 1 1 1

> rep(1:5,3)                # cria a sequência de 1 a 5, repetindo-a 3 vezes
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

> rep(c(1,2), c(3,5))
```

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

> rep(c("cara", "coroa"), c(2,4))
[1] "cara" "cara" "coroa" "coroa" "coroa" "coroa"

> rep(1:3, rep(5,3))
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3

> rep(1:5, each = 3)
[1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

e) `gl()`

Apesar de pouco utilizada, a função `gl` gerar fatores, especificando o padrão de seus níveis. Seus principais argumentos consiste em especificar o número de níveis (`n`), o número de replicações (`k`), o número especificando o comprimento (`length`, que por *default* `length=n*k`) e os nomes dos níveis dos fatores (`labels`), que deve ser de tamanho compatível ao número de níveis `n`. Alguns exemplos:

```
> grupo = gl(n = 2, k = 3, labels = c("Control", "Treat"))
> grupo
[1] Control Control Control Treat Treat Treat
Levels: Control Treat

> gl(2, 2, labels = c("Fem", "Masc"))
[1] Fem Fem Masc Masc
Levels: Fem Masc

> gl(2, 1, length = 20)
[1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Levels: 1 2

> gl(2, 2, 20)
[1] 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2
Levels: 1 2
```

### *Funções importantes definidas para vetores*

Na Tabela 4.1 listamos algumas funções existentes no R que são usadas em objetos vetoriais.

Tabela 4.1: Funções em vetores.

Funções	
<code>length</code>	comprimento de vetor
<code>mode</code>	modo da estrutura vetorial definida
<code>sum</code>	soma dos elementos
<code>prod</code>	produto dos elementos
<code>cumsum</code>	soma acumulada dos elementos
<code>cumprod</code>	produto acumulado dos elementos
<code>sort</code>	ordem crescente
<code>rev</code>	ordem inversa do vetor
<code>order</code> ou <code>sort.list</code>	retorna um vetor de inteiros contendo a permutação que irá ordenar o objeto de entrada em ordem crescente
<code>rank</code>	posto dos valores
<code>duplicated</code>	verifica repetições de valores
<code>unique</code>	retira os elementos iguais
<code>any</code>	verifica se pelo menos um elemento do vetor satisfaz a condição
<code>all</code>	verifica se todos os elementos do vetor satisfazem a condição

Para ilustrar as funções apresentadas na Tabela 4.1, considere o vetor `x`:

```
> x <- c(9,4,5,6,4,7,10)
```

Utilizando o objeto vetorial `x`, vejamos algumas aplicações dessas funções:

```
> length(x)
```

```
[1] 7
```

```
> mode(x)
```

```
[1] "numeric"
```

```
sum(x)
```

```
[1] 45
```

```
> prod(x)
```

```
[1] 302400
```

```
> cumsum(x)
```

```
[1] 9 13 18 24 28 35 45
```

```
> cumprod(x)
```

```
[1] 9 36 180 1080 4320 30240 302400

> sort(x)
[1] 4 4 5 6 7 9 10

> rev(x)
[1] 10 7 4 6 5 4 9

> order(x)
[1] 2 5 3 4 6 1 7

> sort.list(x)
[1] 2 5 3 4 6 1 7

> rank(x)
[1] 6.0 1.5 3.0 4.0 1.5 5.0 7.0

> duplicated(x)
[1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE

> unique(x)
[1] 9 4 5 6 7 10

> any(x>5)
[1] TRUE

> all(x>5)
[1] FALSE
```

Vale ressaltar que estas operações ainda podem ser atribuídas a um outro objeto:

```
> soma <- sum(x)
> soma
[1] 45
```

## 4.2 Matriz

Uma matriz é uma disposição bidimensional em linhas e colunas. É a maneira usual de se “enxergar os dados”. Geralmente as colunas são as variáveis e as linhas são os indivíduos. Os elementos de uma matriz no *software* R devem ser de um único modo (numérica, caracter, lógica ou complexa), sendo mais comum termos elementos numéricos.

Usamos a função `matrix` para converter dados em uma matriz, definindo alguns argu-



mentos específicos. A função `matrix` tem a seguinte estrutura:

```
matrix(D, nrow = n1, ncol = n2, byrow = FALSE),
```

em que:

`D` é um objeto que corresponde aos dados;

`n1` e `n2` são os números de linhas e colunas, respectivamente (compatíveis com o tamanho de `D`);

`byrow` tem valor lógico `FALSE` (F) para preenchimento por colunas e `TRUE` (T) para preenchimento por linhas.

Podemos criar uma matriz digitando seus elementos diretamente a partir da função `scan`, sem argumentos:

```
> notas.port <- matrix(scan(), ncol = 3, byrow = TRUE)
```

```
1:  2 5 0 8 9 5 8 4 6 7 4 5 6 3 6
```

```
16:
```

```
Read 15 items
```

```
> notas.port
```

	[,1]	[,2]	[,3]
[1,]	2	5	0
[2,]	8	9	5
[3,]	8	4	6
[4,]	7	4	5
[5,]	6	3	6

A função `c()` também pode ser utilizada para combinar os valores em uma matriz:

```
> M <- matrix(c(4,5,8,7), nrow = 2, ncol = 2, byrow = TRUE)
```

```
> M
```

	[,1]	[,2]
[1,]	4	5
[2,]	8	7

Para simplificar, podemos criar uma matriz sem precisar escrever o nome de cada argumento, seguindo a mesma ordem: `nrow`, `ncol`, `byrow`. Contudo, a não especificação do argumento `byrow` implicará a assumir o *default* (`FALSE` ou `F`). Vejamos o mesmo exemplo da matriz `M` descrita acima:

```
> M <- matrix(c(4,5,8,7), 2, 2, T)
```

```
> M
```

	[,1]	[,2]
[1,]	4	5

```
[2,]    8    7
```

**Observações:**

**1** - Ao criar uma matriz especificando o nome de cada argumento, estes não precisam seguir uma determinada ordem.

**2** - A criação de uma matriz pode ser feita especificando somente o número de linhas (**nrow**) ou colunas (**ncol**) e o argumento **byrow**.

Outros exemplos:

```
> N <- matrix(c(1,5,8,3), byrow = TRUE, ncol = 4, nrow = 1)
```

```
> N
```

```
      [,1] [,2] [,3] [,4]
[1,]    1    5    8    3
```

```
> O <- matrix(c(7,1,2), ncol = 1)
```

```
> O
```

```
      [,1]
[1,]    7
[2,]    1
[3,]    2
```

```
> P <- matrix(0:5, 2, byrow = T)
```

```
> P
```

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

```
> Q <- matrix(c("a","b","c","d","e","f"), 3)
```

```
> Q
```

```
      [,1] [,2]
[1,]  "a"  "d"
[2,]  "b"  "e"
[3,]  "c"  "f"
```

```
> R <- matrix(c("a","b","c","d","e","f"), ncol = 2)
```

```
> R
```

```
      [,1] [,2]
[1,]  "a"  "d"
[2,]  "b"  "e"
[3,]  "c"  "f"
```

Uma matriz também pode ser construída pela combinação de dois ou mais vetores de mesmo comprimento (`length`). Como exemplo, considere três vetores com as notas de cinco alunos em cada prova:

```
> notas.p1 <- c(9.5,9.0,9.8,9.9,8.7)
> notas.p2 <- c(9.4,8.9,9.5,9.7,8.6)
> notas.p3 <- c(9.6,9.0,9.9,10.0,8.8)
```

Usamos a função `cbind()` para combinar cada vetor como uma coluna da matriz `notas.mat`:

```
> notas.mat <- cbind(notas.p1,notas.p2,notas.p3)
> notas.mat
```

	notas.p1	notas.p2	notas.p3
[1,]	9.5	9.4	9.6
[2,]	9.0	8.9	9.0
[3,]	9.8	9.5	9.9
[4,]	9.9	9.7	10.0
[5,]	8.7	8.6	8.8

Suponha agora que temos cinco vetores com as notas das três provas de cada aluno:

```
> notas.a1 <- c(9.5,9.4,9.6)
> notas.a2 <- c(9.0,8.9,9.0)
> notas.a3 <- c(9.8,9.5,9.9)
> notas.a4 <- c(9.9,9.7,10.0)
> notas.a5 <- c(8.7,8.6,8.8)
```

Usamos a função `rbind()` para combinar cada vetor como uma linha da matriz `notas.mat2`:

```
> notas.mat2 <- rbind(notas.a1,notas.a2,notas.a3,notas.a4,notas.a5)
> notas.mat2
```

	[,1]	[,2]	[,3]
notas.a1	9.5	9.4	9.6
notas.a2	9.0	8.9	9.0
notas.a3	9.8	9.5	9.9
notas.a4	9.9	9.7	10.0
notas.a5	8.7	8.6	8.8

Também podemos transformar um vetor A já existente em uma matriz:

```
> A <- c(3,2,-4,0,-1,8)           # A é um vetor
> B <- matrix(A, byrow=T, nrow=2, ncol=3)  # transforma A em uma matriz 2 x 3
> B
      [,1] [,2] [,3]
[1,]   3   2  -4
[2,]   0  -1   8
```

### *Funções importantes definidas para matrizes*

Também existe no R funções específicas para objetos matriciais. Na Tabela 4.2 listamos algumas dessas funções.

Tabela 4.2: Funções matriciais.

Funções	
<code>dim</code>	dimensão de uma matriz ou converte um dado vetor em uma matriz com dimensões especificadas
<code>nrow</code>	devolve o número de linhas de uma matriz
<code>ncol</code>	devolve o número de colunas de uma matriz
<code>mode</code>	modo da estrutura matricial definida
<code>det</code>	determinante da matriz (quadrada)
<code>t</code>	transposta da matriz
<code>diag</code>	devolve a matriz identidade de ordem $n$ (se $n$ é um número) ou devolve uma matriz diagonal cujos elementos da diagonal principal são $z$ (se $z$ é um vetor) ou devolve um vetor com os elementos da diagonal principal de $w$ (se $w$ é uma matriz quadrada)
<code>solve</code>	devolve a inversa da matriz (quadrada) ou devolve a solução do(s) sistema(s) $Ax=b$ , em que $A$ é uma matriz quadrada e $b$ pode ser um vetor ou uma matriz
<code>eigen</code>	devolve os valores e vetores próprios da matriz (quadrada)
<code>isSymmetric</code>	verifica se a matriz (quadrada) é simétrica

Para ilustrar as funções matriciais apresentadas na Tabela 4.2, considere a matriz  $W$  e os vetores  $d$  e  $h$  apresentados a seguir:

```
> W <- matrix(c(8,5,3,7), 2)
> W
      [,1] [,2]
```

```
[1,] 8 3
[2,] 5 7

> d <- c(1,6)

> h <- c(7,0,1,3,6,4)
```

Vejamos agora alguns exemplos:

```
> dim(W)                # devolve o número de linhas e de colunas da matriz
[1] 2 2

> dim(h) <- c(2,3)      # transforma o vetor h em uma matriz 2 × 3
> h
      [,1] [,2] [,3]
[1,]  7    1    6
[2,]  0    3    4

> nrow(h)
[1] 2

> ncol(h)
[1] 3

> mode(W)
[1] "numeric"

> det(W)
[1] 41

> t(h)
      [,1] [,2]
[1,]  7    0
[2,]  1    3
[3,]  6    4

> diag(3)
      [,1] [,2] [,3]
[1,]  1    0    0
[2,]  0    1    0
[3,]  0    0    1

> diag(d)
      [,1] [,2]
[1,]  1    6
```

```
[1,] 1 0
[2,] 0 6

> diag(W)
[1] 8 7

> solve(W)
      [,1]      [,2]
[1,] 0.1707317 -0.07317073
[2,] -0.1219512 0.19512195

> solve(W,d) # solução de sistema
[1] -0.2682927 1.0487805

> eigen(W)
$values
[1] 11.405125 3.594875
$vectors
      [,1]      [,2]
[1,] 0.6610612 -0.5628892
[2,] 0.7503320 0.8265324

> isSymmetric(W)
[1] FALSE
```

### 4.3 Ficha de dados (*data frame*)

Frequentemente os dados contém variáveis numéricas (como idade), variáveis categóricas (como sexo) e caracteres (como nome do indivíduo). Uma tabela deste tipo não pode ser manipulada como uma matriz, já que mescla dados de diferentes modos, mas pode ser lida como um **data frame**.

Um **data frame** também é uma disposição bidimensional dos dados, podendo ter elementos de diferentes modos em diferentes colunas, desde que cada coluna tenha o mesmo tamanho. Assim, um **data frame** é uma estrutura tabular na qual as colunas representam variáveis e as linhas representam os indivíduos.

Um **data frame** pode ser criado combinando diferentes objetos de vários modos, como vetores de mesmo comprimento e matrizes com mesmo número de linhas. Exemplos:

```
> lixo <- data.frame(A = 1:4, B = c("a","b","c","d"))
> lixo
  A B
1 1 a
```

```

2 2 b
3 3 c
4 4 d

> nome <- c("Daniel", "Carlos", "Aline", "Julia", "Paulo")
> sexo <- c("M","M","F","F","M")
> idade <- c(18,16,15,17,16)
> notas.port <- matrix(c(2,5,0,8,9,5,8,4,6,7,4,5,6,3,6), ncol=3, byrow=T)
> banco.alunos <- data.frame(nome, sexo, idade, notas.port)
> banco.alunos
  nome      sexo  idade  X1  X2  X3
1 Daniel      F    18    2   5   0
2 Carlos      F    16    8   9   5
3 Aline       M    15    8   4   6
4 Julia       F    17    7   4   5
5 Paulo       M    10    6   3   6

```

## 4.4 Lista

A lista é o tipo mais flexível de objeto em R, pois seus componentes podem ser de qualquer modo, incluindo outras listas. Você pode combinar em uma lista, por exemplo, um vetor numérico com 10 valores e uma matriz  $5 \times 5$  de valores lógicos.

A lista é a escolha mais frequente para retornar valores nas análises. A saída do ajuste de um modelo de regressão linear retorna em uma lista valores numéricos como os coeficientes e os resíduos, que têm comprimentos diferentes, um valor lógico que indica se o intercepto foi ou não ajustado, dentre outros resultados.

Uma lista é criada com a função `list`, com os componentes separados por vírgula:

```
list(componente1, componente2, ..., componenteN)
```

Criaremos uma lista com os nomes de cinco alunos na primeira componente, com as idades dos professores na segunda componente e com o nome do reitor da universidade na terceira componente. Note que os componentes são numerados e podem ter um nome associado.

```

> alunos <- c("Juca", "Rosa", "Hugo", "Duda", "Mara")
> alunos
[1] "Juca" "Rosa" "Hugo" "Duda" "Mara"
> # criando a lista:
> notas.list <- list(alunos, c(54,67,39,55), reitor = "João Paulo da Silva")
> notas.list

```

```
[[1]]
[1] "Juca" "Rosa" "Hugo" "Duda" "Mara"
[[2]]
[1] 54 67 39 55
$reitor
[1] "João Paulo da Silva"
```

### Observações Gerais:

**1** - O nome das dimensões de uma matriz é uma lista com dois componentes: um com os nomes das linhas e o outro com os nomes das colunas.

**2** - A função `is.factor(x)` testa se o objeto `x` é do tipo fator. As funções `is.numeric`, `is.vector` e `is.matrix` testa se um dado objeto é do modo numérico, vetor e matriz, respectivamente.

**3** - A função `data.matrix` converte *data frame* em matriz: `data.matrix("nome do objeto data frame")`.

## 4.5 Expressando dados faltantes

Um valor faltante (*missing value*) é representado pelo símbolo `NA` (*Not Avaliable*), com letras maiúsculas. Por exemplo, se no vetor `notas` a nota do quarto aluno não está disponível, armazenamos `NA` na quarta posição do vetor:

```
> notas <- c(8.5,9.0,6.3,NA,7.9,5.7)
> notas
[1] 8.5 9.0 6.3 NA 7.9 5.7
```

Para saber se algum elemento do objeto tem valor `NA` utilizamos a função `is.na`, que retorna `TRUE` para elementos `NA` e `FALSE`, caso contrário:

```
> is.na(notas)
[1] FALSE FALSE FALSE TRUE FALSE FALSE
```



# Capítulo 5

## Manipulando Dados

Neste capítulo apresentaremos algumas formas de extrair e modificar elementos de um objeto no R.

### 5.1 Extrair elementos

#### a) Vetores

Considere o vetor `notas` com as notas de seis alunos:

```
> notas <- c(8.5,9.0,6.3,NA,7.9,5.7)
```

Podemos extrair um determinado elemento do vetor `notas` indicando a sua posição entre `[ ]`. Por exemplo, a nota do quinto aluno:

```
> notas[5]  
[1] 7.9
```

Suponha agora que se sabe a nota do quarto aluno que antes era desconhecida e assumia `NA`. A nota deste aluno foi 7,5 e deseja incluí-la no vetor `notas`. Para isso, fazemos:

```
> notas[4] = 7.5  
> notas  
[1] 8.5 9.0 6.3 7.5 7.9 5.7
```

Também podemos extrair elementos obedecendo uma determinada condição. Considere que a média de aprovação é 7. Suponha que queremos saber apenas as notas dos alunos aprovados, verificando quais das notas obedecem a esta condição ( $\text{nota} \geq 7$ ):

```
> notas >= 7
[1] TRUE TRUE FALSE TRUE TRUE FALSE
```

Até o momento, apenas foi verificado quais notas obedecem a condição. Agora vamos selecionar apenas as notas dos alunos aprovados

```
> notas[notas >= 7]
[1] 8.5 9.0 7.5 7.9
```

Uma outra maneira de fazer esse mesmo procedimento é criar objetos com as etapas. Veja o mesmo exemplo:

```
> notas.ap <- notas >= 7
> notas.ap
[1] TRUE TRUE FALSE TRUE TRUE FALSE
> notas.select <- notas[notas.ap]
> notas.select
[1] 8.5 9.0 7.5 7.9
```

Podemos extrair também uma sequência de elementos do vetor. Considere o vetor `y`:

```
> y <- c(3,15,9,7,2,3)
```

Vejamos alguns exemplos:

```
> y[2:3]
[1] 15 9

> y[-4]
[1] 3 15 9 2 3

> y[-c(2,5)]
[1] 3 9 7 3

> y[c(1,5)]
[1] 3 2

> y[y != 3]
[1] 15 9 7 2
```

## b) Matrizes

Considere a matriz `notas.port` com notas de cinco alunos nas três provas (linhas indicam cada aluno e as colunas indicam as diferentes provas):

```
> notas.port <- matrix(c(2,5,0,8,9,5,8,4,6,7,4,5,6,3,6), ncol = 3, byrow = T)
> notas.port
      [,1] [,2] [,3]
[1,]    2    5    0
[2,]    8    9    5
[3,]    8    4    6
[4,]    7    4    5
[5,]    6    3    6
```

Podemos extrair um determinado elemento da matriz `notas.port` indicando as posições da linha e coluna entre `[ ]` (`[linha,coluna]`). Por exemplo, a nota do quinto aluno na terceira prova:

```
> notas.port[5,3]
[1] 6
```

Para extrair uma linha inteira, basta fornecer o número da linha e deixar em branco o número da coluna. Assim, para obter as notas do segundo aluno nas três provas, devemos fazer:

```
> notas.port[2,]
[1] 8 9 5
```

Da mesma forma, podemos obter as notas de todos os alunos na primeira prova:

```
> notas.port[,1]
[1] 2 8 8 7 6
```

Também pode ser combinado índices de linhas ou colunas em vetores e usar esses vetores dentro do operador `[ ]`. Exemplos:

```
> notas.port[c(2,5),2:3]
      [,1] [,2]
[1,]    9    5
[2,]    3    6

> notas.port[c(1,3,5),]
      [,1] [,2] [,3]
[1,]    2    5    0
[3,]    8    4    6
[5,]    6    3    6
```

```

[1,] 2 5 0
[2,] 8 4 6
[3,] 6 3 6

> notas.port2 <- notas.port[-c(1,5),]
> notas.port2
      [,1] [,2] [,3]
[1,] 8 9 5
[1,] 8 4 6
[1,] 7 4 5

> prova2 <- notas.port[, -c(1,3)]
> prova2
[1] 5 9 4 4 3

> prova3 <- notas.port[, 3]
> prova3
[1] 0 5 6 5 6

> notas.port >= 7
      [,1] [,2] [,3]
[1,] FALSE FALSE FALSE
[2,] TRUE  TRUE FALSE
[3,] TRUE  FALSE FALSE
[4,] TRUE  FALSE FALSE
[5,] FALSE FALSE FALSE

> notas.port[notas.port[,1] >= 7,1]
[1] 8 8 7

> notas.port[2,notas.port[2,] >= 7]
[1] 8 9

```

Para substituir um ou mais elementos de uma matriz, basta fornecer a linha e a coluna do(s) elemento(s) a serem substituídos. Alguns exemplos:

```

> notas.port[1,3] = 7
> notas.port
      [,1] [,2] [,3]
[1,] 2 5 7
[2,] 8 9 5
[3,] 8 4 6

```

```
[4,] 7 4 5
[5,] 6 3 6

> notas.port[3,] = c(2,7,9)
> notas.port
      [,1] [,2] [,3]
[1,] 2    5    7
[2,] 8    9    5
[3,] 2    7    9
[4,] 7    4    5
[5,] 6    3    6
```

### c) Ficha de Dados

Os elementos de um *data frame* podem ser referidos (como em uma matriz) indicando a linha e a coluna entre `[ ]`. Considere a ficha de dados `banco.alunos`:

```
> nome <- c("Daniel", "Carlos", "Aline", "Julia")
> sexo <- c("M","M","F","F")
> idade <- c(18,16,15,17)
> banco.alunos <- data.frame(nome, sexo, idade)
> banco.alunos
  nome      sexo  idade
1 Daniel      F     18
2 Carlos      F     16
3 Aline       M     15
4 Julia       F     17
```

Vejamos alguns exemplos:

```
> banco.alunos[4,2]
[1] F
Levels:  F M

> banco.alunos[,3]
[1] 18 16 15 17

> banco.alunos[,2]
[1] M M F F
Levels:  F M

> banco.alunos[2]
```

```
      sexo
1      M
2      M
3      F
4      F

> banco.alunos[[1]]
[1] Daniel Carlos Aline Julia
Levels: Aline Carlos Daniel Julia

> banco.alunos[, "sexo"]
[1] M M F F
Levels: F M

> banco.alunos["idade"]
      idade
1      18
2      16
3      15
4      17
```

A notação "nome do data frame"\$"nome da componente" também pode ser usada:

```
> banco.alunos$sexo
[1] M M F F
Levels: F M

> banco.alunos$idade
[1] 18 16 15 17

> banco.alunos$idade[3]
[1] 15

> banco.alunos$nome[2:3]
[1] Carlos Aline
Levels: Aline Carlos Daniel Julia
```

Para substituir elementos de uma ficha de dados, o procedimento é o mesmo usado em matrizes. Devemos fornecer a linha e a coluna do elemento que desejamos substituir. Vejamos alguns exemplos:

```
> banco.alunos[1,3] = 17
> banco.alunos
  nome      sexo  idade
1 Daniel      M    17
2 Carlos      M    16
3 Aline       F    15
4 Julia       F    17

> banco.alunos$sexo[3] = "M"
> banco.alunos
  nome      sexo  idade
1 Daniel      M    17
2 Carlos      M    16
3 Aline       M    15
4 Julia       F    17

> banco.alunos[3,2] = "F"
> banco.alunos
  nome      sexo  idade
1 Daniel      M    17
2 Carlos      M    16
3 Aline       F    15
4 Julia       F    17

> banco.alunos[1,1] = "Carlos"
> banco.alunos[2,1] = "Daniel"
> banco.alunos
  nome      sexo  idade
1 Carlos      M    17
2 Daniel      M    16
3 Aline       F    15
4 Julia       F    17
```

**Observação:** Em ficha de dados, para dados categóricos que são considerados fatores, os elementos só podem ser alterados pelos níveis que o compõe. Por exemplo, substituir o nome “Aline” por “Alana”. Neste caso, “Alana” não pertence ao nível. Então, a troca não será permitida. Vejamos:

```
> banco.alunos$nome[3] = "Alana"
Warning message:
In `[<-.factor'('*tmp*', 3, value = "Alana") :
```

invalid factor level, NAs generated

#### d) Listas

Os componentes de uma lista são identificados pela sua posição na lista entre colchetes duplos `[[.]]`; se tiverem nome, também podem ser referidos pelo mesmo, `"nome da Lista"$"nome da componente"`. Considere a lista `notas.list`:

```
> alunos <- c("Juca", "Rosa", "Hugo", "Duda", "Mara")
> notas.list <- list(alunos, idade=c(54,67,39,55), reitor="João Paulo da Silva")
> notas.list
[[1]]
[1] "Juca" "Rosa" "Hugo" "Duda" "Mara"
$idade
[1] 54 67 39 55
$reitor
[1] "João Paulo da Silva"
```

Extraímos elementos de uma lista da forma:

```
> notas.list[[1]]
[1] "Juca" "Rosa" "Hugo" "Duda" "Mara"

> notas.list[1]
[[1]]
[1] "Juca" "Rosa" "Hugo" "Duda" "Mara"

> notas.list$reitor
[1] "João Paulo da Silva"

> notas.list$idade
[1] 54 67 39 55

> notas.list[[2]]
[1] 54 67 39 55

> notas.list[[2]][4]           # recuperando a quarta idade
[1] 55

> notas.list[[2]][c(1,3)]
[1] 54 39
```



Para substituir elementos de uma lista, vejamos alguns exemplos:

```
> notas.list$reitor = "José Silva"
> notas.list
[[1]]
[1] "Juca" "Rosa" "Hugo" "Duda" "Mara"
$idade
[1] 54 67 39 55
$reitor
[1] "José Silva"

> notas.list[[2]][c(1,3)] = c(48,32)
> notas.list$idade
[1] 48 67 32 55

> notas.list[[1]][2] = "Alana"
> notas.list[[1]]
[1] "Juca" "Alana" "Hugo" "Duda" "Mara"

> notas.list[[1]] = c("Jonas","Alana","Sheila","Juan")
> notas.list
[[1]]
[1] "Jonas" "Alana" "Sheila" "Juan"
$idade
[1] 48 67 32 55
$reitor
[1] "José Silva"
```

## 5.2 Operações com vetores e matrizes

A maioria dos dados que analisamos são numéricos e dados numéricos são frequentemente manipulados usando funções e operadores matemáticos. Operações matemáticas no R são vetorizadas, indicando que elas atuam no objeto de dados inteiro de uma só vez, elemento por elemento.

Em vetores podemos utilizar expressões aritméticas, caso em que as operações são realizadas em elemento a elemento. Dois vetores que se utilizem na mesma expressão não tem que, obrigatoriamente, ser do mesmo comprimento. Se não os são, o resultado é um vetor com o comprimento do vetor mais longo, sendo que o mais curto é utilizado ciclicamente, repetindo-se os elementos na sequência o quanto necessário, até que seu comprimento coincida com o do vetor do mais longo. Em particular, uma constante ou um vetor unitário será simplesmente repetido tantas vezes quanto o comprimento do vetor mais longo. Exemplo:

```
> x <- scan()
1:  4
2:  8
3:  3
4:  2
5:
Read 4 items
> y <- c(3,4,1)
> x + y
[1] 7 12 4 5
Warning message:
In x + y: longer object length is not a multiple of shorter object length
```

Os operadores aritméticos elementares são os habituais: soma (+), subtração (-), produto (\*), / e potência (^ ou \*\*). Para ilustração, considere os vetores **w** e **z**:

```
> w <- c(1,5,3,7)
> z <- c(2,9,1,0)
```

Vejamos alguns exemplos:

```
> w + z                # soma os dois vetores, elemento por elemento
[1] 3 14 4 7

> w - z                # subtrai os dois vetores, elemento por elemento
[1] -1 -4 2 7

> w * z                # multiplica os dois vetores, elemento por elemento
[1] 2 45 3 0

> z / w                # divide os dois vetores, elemento por elemento
[1] 2.0000000 1.8000000 0.3333333 0.0000000

> 10 * w               # multiplica por 10 todos os elementos do vetor w
[1] 10 50 30 70

> w - 2                # subtrai 2 de todos os elementos do vetor w
[1] -1 3 1 5

> z ^ 2                # eleva ao quadrado cada elemento do vetor z
[1] 4 81 1 0
```

Também estão disponíveis para vetores as funções aritméticas comuns: logaritmo (`log`), exponencial (`exp`), raiz quadrada (`sqrt`), seno (`sin`), cosseno (`cos`), tangente (`tan`), entre outros. Exemplos:

```
> exp(z)
[1] 7.389056 8103.083928 2.718282 1.000000

> log(w + z)
[1] 1.098612 2.639057 1.386294 1.945910
```

Com as matrizes, os operadores funcionam igualmente, aplicando a operação elemento por elemento da matriz. Considere as matrizes `M`, `Mt` e `V`:

```
> M <- matrix(5:8, 2, 2)
> M
      [1,] [2,]
[1,]    5    7
[2,]    6    8

> Mt <- t(M)           # transposta de M
> Mt
      [1,] [2,]
[1,]    5    6
[2,]    7    8

> V <- matrix(9:12, 2, 2)
> V
      [1,] [2,]
[1,]    9   11
[2,]   10   12
```

Como as matrizes `M`, `Mt` e `V` têm mesma dimensão, operações como soma, subtração, produto, divisão podem ser feitas com elemento por elemento. Alguns exemplos:

```
> Mt + V
      [1,] [2,]
[1,]   14   17
[2,]   17   20

> Mt * V
      [1,] [2,]
```

```
[1,] 45 66
[2,] 70 96
```

Vale ressaltar que a multiplicação matricial é feita usando-se o operador `%*%`. Exemplo:

```
> Mt %*% V
      [1,] [2,]
[1,] 105 127
[2,] 143 173
```

Outros exemplos:

```
> a <- c(3,5,4)
> b <-matrix (scan(), ncol = 2, byrow = TRUE)
1: 3
2: 4
3: 5
4: 6
5:
Read 4 items
> b
      [1,] [2,]
[1,] 3 4
[2,] 5 6

> b + 3
      [1,] [2,]
[1,] 6 7
[2,] 8 9

> a + b
      [1,] [2,]
[1,] 6 8
[2,] 10 9
Warning message:
In a + b: longer object length is not a multiple of shorter object length
```

## 5.3 Leitura de dados em arquivos

Considere os arquivos “temperatura.txt”, “carro.txt”, “notas.txt”, “peso.txt” e “saude.xls” salvos no computador nas pastas C → Curso R.

### a) Função `read.table`

Uma tabela de valores (ficheiro) em um arquivo pode ser atribuída a um *data frame*, através da função `read.table` e seus argumentos:

```
read.table(nome, header = FALSE, sep = " ", dec = ".", row.names, col.names,  
           as.is = FALSE, na.strings = "NA"),
```

em que:

**nome** é o nome do arquivo de dados; se este não se encontrar na pasta de trabalho, é necessário indicar o endereço completo (utilizando / ou \\ como separador de pastas);

**header** é uma variável lógica que indica se o ficheiro tem ou não os nomes das variáveis na primeira linha. Por omissão o valor é **FALSE** (ou **F**);

**sep** é o caractere que separa os valores em cada linha; por omissão é " " que corresponde a um ou mais espaços em branco;

**dec** é o caractere utilizado como separador decimal. Por omissão é o ponto;

**row.names** é um vetor com os nomes das linhas. Por omissão, 1, 2, 3, ...;

**col.names** é um vetor com os nomes das colunas. Por omissão, V1, V2, V3, ...;

**as.is** é uma variável lógica que controla a conversão das variáveis alfanuméricas em fatores (objetos do R para variáveis qualitativas – vetores de categorias). Se o seu valor é **TRUE** (ou **T**) a leitura mantém o tipo original dos dados;

**na.strings** é o valor usado para os dados desconhecidos no ficheiro e que será convertido em **NA** (valor faltante).

Abaixo segue duas maneiras diferentes de fazer a leitura de um mesmo conjunto de dados em um arquivo com extensão “.txt”:

```
> dados <- read.table("C:\\Curso R\\temperatura.txt", header = T, sep = " ")
```

```
> dados <- read.table("C:/Curso R/temperatura.txt", header = T, sep = " ")
```

**Observação:** O arquivo `temperatura.txt` deverá estar no caminho indicado.

### b) Função `scan`

A função `scan` pode ser utilizada para a leitura de dados de um arquivo na situação em que ficheiro não contém os nomes das variáveis.

Suponha que você tenha guardado ou anotado o volume de gasolina (em litros) que você adicionou a seu carro em 118 paradas para “completar o tanque”. Suponha também que

esses dados estão em um arquivo texto chamado `carro.txt`. Você pode criar o vetor numérico `consumo` utilizando uma das seguintes formas:

```
> consumo <- scan(file = "C:\\Curso R\\carro.txt")
> consumo <- scan(file = "C:/Curso R/carro.txt")
```

Podemos ainda omitir o argumento `file` e considerar para a leitura dos dados uma das seguintes formas:


```
> consumo <- scan("C:\\Curso R\\carro.txt")
> consumo <- scan("C:/Curso R/carro.txt")
```

Para ler matrizes a partir de conjuntos em dados em arquivos como `notas.txt`, podemos utilizar a função `matrix` juntamente com a função `scan`:

```
> notas.port <- matrix(scan("C:\\Curso R\\notas.txt"), ncol = 3, byrow = TRUE)
Read 15 items
> notas.port
      [,1] [,2] [,3]
[1,]    2    5    0
[2,]    8    9    5
[3,]    8    4    6
[4,]    7    4    5
[5,]    6    3    6
```

### c) clipboard nas funções `read.table` e `scan`

É possível ler o conjunto de dados diretamente de arquivos (até mesmo com extensão “.xls”) com a função `read.table` e `scan`. Para isso, o procedimento é o seguinte:

- (i) Com o arquivo aberto, selecione e copie ( + c) as células de interesse;
- (ii) Em seguida, para a leitura dos dados no R, execute a função `read.table` ou `scan` com as seguintes considerações:

- Para células selecionadas sem rótulos (nome das variáveis), a leitura de dados pode ser feita executando uma das linhas de comando:

```
> read.table("clipboard")
> scan("clipboard") # se a variável é numérica
```

```
> scan(what=character(),"clipboard")      # se a variável é caracteres
```

- Para células selecionadas com rótulos (nome das variáveis), a leitura de dados só poderá ser feita através da função `read.table`:

```
> read.table("clipboard", header = T)
```

**Observação:** A leitura de dados utilizando a função `scan` só é ideal quando temos uma variável, pois os dados selecionados são lidos no R como um único vetor.

Para ilustrar esse procedimento de leitura, considere o arquivo “saude.xls”. Com o arquivo aberto, selecione e copie (`ctrl` + `c`) todas as células que tem informações. Observe que no arquivo tem o rótulo de cada variável (SEXO; IDADE; ALT; PESO; CINT; TXPUL; SIST; DIAST; COL; IMC; PERNA; COTOV; PULSO; BRAÇO). No R, execute a linha de comando:

```
> dados.saude <- read.table("clipboard", header = T)
```

Considere agora o arquivo “peso.txt”. Abra o arquivo, selecione e copie (`ctrl` + `c`) os dados. No R, execute a linha de comando:

```
> dados.peso <- read.table("clipboard", header = T)
```

Um outro exemplo, considere o arquivo “notas.txt”. Abra o arquivo e observe que os dados estão dispostos uma única coluna. Para transformá-los em uma matriz, selecione e copie (`ctrl` + `c`) os dados e execute no R a linha de comando:

```
> notas.port <- matrix(scan("clipboard"), ncol = 3, byrow = T)
```

## 5.4 Escrita de dados em arquivo

Para escrever em um arquivo o conteúdo de um *data frame* ou de uma matriz usa-se a função `write.table` com os seus argumentos como descritos a seguir:

```
write.table(x, file = " ", quote = TRUE, sep = " ", na = "NA", dec = ".",  
            row.names = TRUE, col.names = TRUE)
```

em que:

`x` é o nome do objeto a guardar;

`file` é o nome do arquivo, incluindo o caminho de pastas;

`quote` é uma variável lógica que indica se as variáveis alfanuméricas são ou não escritas entre

aspas;

**sep** é o caractere que separa os valores em cada linha; por omissão é um espaço em branco;

**na** é a representação no arquivo dos NA's de **x**;

**dec** é o caractere utilizado como separador decimal. Por omissão é o ponto;

**row.names**, **col.names** ou são variáveis lógicas que indicam se o arquivo vai ou não conter os nomes das linhas/colunas de **x**, ou são vetores alfanuméricos com os nomes a atribuir às linhas/colunas no arquivo.

Vejamos um exemplo:

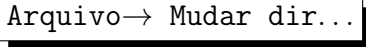

```
> dados1 <- rpois(10,0.5)
> dados2 <- rpois(10,1)
> write.table(cbind(dados1,dados2), file = "C:/Curso R/DADOS.txt", sep = " ",
              row.names = F, col.names = T)
```

Um outra forma de salvar objetos é utilizar a própria interface do R com a função **save**, criando arquivo com extensão “.Rdata”. Na função **save** o usuário deve fornecer os objetos que deseja guardar e com o argumento **file** indicar o local e o nome seguido da extensão “.Rdata” que o arquivo será salvo. Exemplo:

```
> amostra1 <- runif(1000,10,50)
> amostra2 <- rpois(1000,15)
> save(amostra1, amostra2, file = "C:/Curso R/AMOSTRAS.Rdata")
```

### Observações Gerais:

- 1 - Tanto na leitura de dados quanto na escrita de dados, podemos substituir " " por ' '.
- 2 - Algumas extensões que o R permite ler e salvar são: “.txt”, “.dat” e “.csv”.
- 3 - Podemos omitir o caminho de pasta no argumento **file** por optar em fazer a mudança de diretório na barra de Menu do R ou, equivalentemente, utilizar a função **setwd**. Para isso, deve-se utilizar uma das seguintes maneiras:

(i) Na barra de Menu do R, vá em . Uma janela “Procurar pasta” é aberta e nela o usuário deverá indicar o local em que se encontra o(s) arquivo(s) de dados (pastas **C** → **Curso R**) e por fim clicar em .

(ii) Ou, alternativamente, utiliza a função **setwd** para a mudança:

```
> setwd("caminho da pasta").
```



A vantagem em fazer a mudança de diretório é que ela precisa ser feita uma única vez e qualquer arquivo que esteja na pasta indicada pode ser lido. A seguir é ilustrado os exemplos anteriores (leitura e escrita) usando a função `setwd`:

```
> setwd("C:/Curso R")
> dados <- read.table("temperatura.txt", header = T, sep = " ")
> consumo <- scan("carro.txt")
> notas.port <- matrix(scan("notas.txt"), ncol = 3, byrow = T)

> dados1 <- rpois(10,0.5)
> dados2 <- rpois(10,1)
> write.table(cbind(dados1,dados2), file = "DADOS.txt", sep = " ", row.names = F,
              col.names = T)

> amostra1 <- runif(1000,10,50)
> amostra2 <- rpois(1000,15)
> save(amostra1, amostra2, file = "AMOSTRAS.Rdata")
```

## 5.5 Função `attach`

É vantajoso poder referir cada componente de uma lista ou ficha de dados como se tratasse de uma variável, com o nome que tem, sem ter necessidade de explicitamente indicar o nome do objeto correspondente a lista ou ficha de dados (`nome do objeto$nome do componente`). Para tal, usa-se a função `attach()`, tendo como argumento o nome do objeto, de modo a permitir acessar diretamente às suas componentes sem explicitar o objeto. Como exemplo, considere a lista de dados dada por `dados`:

```
> dados <- data.frame(alunos = c("Viviane","Anderson"), idades = c(20,29))
> dados      alunos      idades
1  Viviane        20
2  Anderson        29

> alunos
Erro: objeto 'alunos' não encontrado
```

O comando `attach` conecta os nomes das variáveis ao caminho de busca, de modo que, caso não haja outros objetos com o mesmo nome, as variáveis contidas lista passam a poder referir-se com os nomes “alunos” e “idades”:

```
> attach(dados)
> alunos
```

```
[1] Viviane Anderson
Levels:  Anderson Viviane
> idades
[1] 20 29
```

Entretanto, se for executado o comando `alunos <- idades`, não se substitui a variável “alunos” na lista; uma nova variável é criada, com o nome “alunos”, com prioridade sobre a variável `dados$alunos` no caminho de busca. Se o pretendido fosse mesmo atribuir à variável “alunos” da lista, dever-se-ia fazer `dados$alunos <- idades`. Para desagregar as variáveis da lista do caminho de busca, usa-se o comando:

```
> detach(dados)
```

Uma vez realizada esta função, deixarão de existir os objetos “alunos” e “idades” como tal, embora continuem a existir e estar disponíveis como componentes da lista.

# Capítulo 6

## Distribuições de Probabilidade

O R tem muitas funções para realizar cálculos de probabilidade, incluindo geração de números aleatórios de determinadas distribuições de probabilidade. Estas funções têm a seguinte forma geral:

`letranome (argumentos separados por vírgula)`

em que:

**letra:** indica a operação;

**nome:** indica a distribuição de probabilidade.

Na Tabela 6.1 fornecemos as letras, o que corresponde cada uma e os argumentos necessários.

Tabela 6.1: Funções R para geração de números aleatórios e cálculos de probabilidades.

Letras	Operação	Argumentos necessários
r	Gera números pseudo-aleatórios	Tamanho da amostra, parâmetros da distribuição
p	Calcula a função de probabilidade acumulada	Vetor de quantis, parâmetros da distribuição
q	Calcula o quantil correspondente a uma dada probabilidade	Vetor de probabilidades, parâmetros da distribuição
d	Calcula a densidade de probabilidade	Vetor de quantis, parâmetros da distribuição

Uma lista das distribuições mais conhecidas e que estão disponíveis no R pode ser vista na Tabela 6.2. Para mais detalhes de outras distribuições não listada, ver help do R.

Tabela 6.2: Distribuições de probabilidade em R.

Distribuição	Nome	Argumentos	Default
Poisson	<code>pois</code>	$\lambda$	—
Binomial	<code>binom</code>	$n, p$	—
Binomial negativa	<code>nbinom</code>	$n, p$	—
Geométrica	<code>geom</code>	$p$	—
Hipergeométrica	<code>hyper</code>	$m, n, k$	—
Uniforme	<code>unif</code>	$\min, \max$	0, 1
Exponencial	<code>exp</code>	$\lambda$	—
Normal	<code>norm</code>	$\mu, \sigma$	0, 1
t-Student	<code>t</code>	$df$	—
Beta	<code>beta</code>	$\alpha, \beta$	—
Gama	<code>gamma</code>	$\alpha, \beta$	—

Também podemos obter no R os argumentos necessários para geração de números aleatórios e cálculos de probabilidades. Para isso, usamos o comando `args`. Como exemplo, suponha que queremos saber quais os argumentos necessários para obter o quantil associado a um valor de probabilidade da distribuição binomial.

```
> args(qbinom)
function (p, size, prob, lower.tail = TRUE, log.p = FALSE)
NULL
```

**Observação:** Os argumentos `lower.tail` e `log.p` quando não especificados segue o *default*, assumindo `TRUE` e `FALSE`, respectivamente. O argumento `lower.tail`, quando `TRUE` (ou `T`), considera a função usando “ $\leq$ ”. Ou seja, no exemplo acima, deseja-se obter  $x$  tal que  $P(X \leq x) = p$ , em que  $p$  é dado e  $X$  é uma variável aleatória com distribuição binomial. Se `lower.tail` assume `FALSE` (ou `F`), considera o cálculo da probabilidade usando “ $>$ ”. Ou seja, obter  $x$  tal que  $P(X > x) = p$ . O argumento `log.p`, quando `FALSE` (ou `F`), retorna o valor da função na escala original ( $x$ ). Quando `TRUE` (ou `T`), retorna o logaritmo natural do valor ( $\log(x)$ ).

Outra forma de obter informações sobre as funções é utilizando o comando `help`, por exemplo:

```
> help(Binomial)
```

Abre-se uma janela contendo várias informações sobre as funções e seus argumentos,

como no exemplo:

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

Vejamos alguns exemplos:

```
> x <- 0:10
> fx <- dbinom(x, 10, 0.35)
> fx
[1] 1.346274e-02  7.249169e-02  1.756530e-01  2.522196e-01  2.376685e-01
[6] 1.535704e-01  6.890980e-02  2.120302e-02  4.281378e-03  5.123017e-04
[11] 2.758547e-05

> Fx <- pbinom(x, 10, 0.35)
> Fx
[1] 0.01346274  0.08595444  0.26160739  0.51382702  0.75149551  0.90506592
[7] 0.97397572  0.99517873  0.99946011  0.99997241  1.00000000

> dbinom(7, 10, 0.35)
[1] 0.02120302

> sum(dbinom(0:7, 10, 0.35))
[1] 0.9951787

> 1 - pbinom(7, 10, 0.35)
[1] 0.004821265

> pbinom(7, 10, 0.35, lower = FALSE)
[1] 0.004821265

> dnorm(-1)                # normal-padrão (média=0 e desvio-padrão=1)
[1] 0.2419707

> pnorm(-1)
[1] 0.1586553

> qnorm(0.975)
[1] 1.959964

> rnorm(5)
[1] -1.1662232 -2.7355749 -1.2279732 -0.1371890  1.3781193
```

**Observação:** Cada vez que o comando `rnorm` é chamado, diferentes elementos da amostra são produzidos porque a semente do gerador é modificada. Para gerar duas amostras idênticas deve-se usar o comando `set.seed` como ilustrado abaixo:

```
set.seed("número inteiro qualquer")
```

Vejamos alguns exemplos:

```
> set.seed(214)           # fixando uma semente inicial
> rnorm(6)                # amostra de 6 elementos
[1] -0.46774980  0.04088223  1.00335193  2.02522505  0.30640096  0.42577748

> rnorm(6)
[1] 0.7488927  0.4464515 -2.2051418  1.9818137 -2.6255525 -0.7230179

> rexp(5,0.3)
[1] 6.3169667  0.6450481  2.9735890  9.1089234  5.8656271

> rexp(5,0.3)
[1] 0.4842575  4.7864576  0.4717516  1.0937959  10.2059892

> runif(5)
[1] 0.6807363  0.1331188  0.4157805  0.8683657  0.6214230

> set.seed(214)           # retorna o valor da semente ao valor inicial
> rnorm(6)                # gera novamente a primeira amostra de 6 elementos
[1] -0.46774980  0.04088223  1.00335193  2.02522505  0.30640096  0.42577748
```

As funções relacionadas à distribuição normal tem (entre outros) os argumentos `mean` e `sd` para definir média e desvio padrão da distribuição que podem ser modificados como nos exemplos a seguir.

```
> qnorm(0.975, mean = 100, sd = 8)
[1] 115.6797

> qnorm(0.975, m = 100, s = 8)
[1] 115.6797

> qnorm(0.975, 100, 8)
[1] 115.6797
```

Estas funções aceitam também vetores em seus argumentos, como ilustrado nos exemplos abaixo:

```
> qnorm(c(0.05, 0.95))
[1] -1.644854  1.644854

> rnorm(4, mean = c(0, 10, 100, 1000))
[1] 0.7488927  10.4464515  97.7948582  1001.9818137

> rnorm(4, mean = c(10, 20, 30, 40), sd=c(2, 5))
[1] 4.748895  16.384911  28.233862  37.374148
```

Note que no último exemplo a lei da reciclagem foi utilizada no vetor de desvios padrão, isto é, os desvios padrão utilizados foram (2, 5, 2, 5).

Cálculos de probabilidades usuais, para os quais utilizávamos tabelas estatísticas, podem ser facilmente obtidos. Como exemplo, seja  $X$  uma v.a. com distribuição  $N(100, 100)$ . Calcular as probabilidades:

```
a)  $P(X < 95)$ 
> pnorm(95, 100, 10)
[1] 0.3085375

b)  $P(90 < X < 110)$ 
> pnorm(110, 100, 10) - pnorm(90, 100, 10)
[1] 0.6826895

c)  $P(X > 95)$ 
> 1 - pnorm(95, 100, 10)
[1] 0.6914625
```

Ou ainda,

```
> pnorm(95, 100, 10, lower = FALSE)
[1] 0.6914625
```

Note que a última probabilidade foi calculada de duas formas diferentes, sendo a segunda usando o argumento `lower` (mais estável numericamente).

Uma outra função importante que também encontramos no R é `ppoints`. Esta função cria valores uniformemente espaçados entre 0 e 1 (probabilidades), sendo muito útil para construir gráficos das distribuições. Exemplos:

```
> ppoints(5)
[1] 0.1190476 0.3095238 0.5000000 0.6904762 0.8809524

> ppoints(10)
[1] 0.06097561 0.15853659 0.25609756 0.35365854 0.45121951 0.54878049
```

```
[7] 0.64634146 0.74390244 0.84146341 0.93902439
```

### **Exercícios Resolvidos:**

1) Gere 10 valores de uma distribuição Normal com média = 25 e desvio padrão = 5.

```
> rnorm(10,25,5)
```

2) Seja  $Z \sim N(0, 1)$ . Calcule:

a)  $P(Z < -1.64)$

```
> pnorm(-1.64,0,1)
```

```
[1] 0.05050258
```

b)  $P(Z > 1.96)$

```
> 1 - pnorm(1.96,0,1)
```

```
[1] 0.02499790
```

3) Seja  $Z \sim N(0, 1)$ :

a) Encontre  $k$  tal que  $P(Z < k) = 0.05$

```
> qnorm(0.05,0,1)
```

```
[1] -1.644854
```

b) Encontre  $k$  tal que  $P(Z > k) = 0.025$

```
> qnorm(1-0.025,0,1)
```

```
[1] 1.959964
```

4) Calcule  $P(X = 5)$  onde  $X \sim \text{Bin}(10, 0.5)$ .

```
> dbinom(5,10,0.5)
```

```
[1] 0.2460938
```

## **6.1 Função sample**

A função `sample` é usada para tomar uma amostra aleatória (com ou sem reposição) de um objeto vetorial `x`, permitindo que o usuário determine as probabilidades de seleção de cada elemento e gerar uma permutação do vetor `x`. Esta função é dada por:

```
sample(x, size, replace = FALSE, prob = NULL)
```

em que:

`x` é o objeto vetorial de dados numéricos, complexos ou caracteres a ser amostrado ou permu-



tado. Se `x` é um inteiro positivo, uma amostra ou permutação será tomada da sequência `1:x`.  
`size` tamanho da amostra (default: comprimento de `x`);  
`replace = FALSE` amostragem sem reposição (*default*); se `TRUE` amostragem com reposição;  
`prob` vetor de probabilidade de seleção de cada elemento de `x` (*default*: probabilidades iguais).

Para ilustração, considere o vetor `W`:

```
> W <- seq(2,20,2)
> W
[1] 2 4 6 8 10 12 14 16 18 20
```

Vejamos alguns exemplos da função `sample` utilizando o objeto `W`:

```
> sample(W)                # permutação de W
[1] 6 8 10 14 4 16 18 12 20 2

> sample(W,5,TRUE)         # amostra de tamanho 5, com reposição
[1] 6 8 12 20 6

> sample(W,5)              # amostra de tamanho 5, sem reposição
[1] 6 8 10 14 4

> sample(10)               # permutação da sequência 1:10
[1] 3 4 5 7 2 8 9 6 10 1

> sample(10,5,TRUE)        # amostra de tamanho 5 da sequência 1:10, com reposição
[1] 3 4 6 10 3

> sample(10,5)             # amostra de tamanho 5 da sequência 1:10, sem reposição
[1] 3 4 5 7 2
```

Você pode definir uma função de probabilidade e gerar números desta distribuição usando a função `sample`:

```
> sample(4, 30, prob = c(0.1,0.2,0.3,0.4), replace = TRUE)
[1] 4 4 3 1 4 2 1 3 3 4 4 4 3 4 2 3 2 1 4 2 1 4 3 4 4 4 4 4 2 4
```

**Observação:** Para obter os resultados acima, basta fixar a semente em 1 para cada exemplo (`set.seed(1)`).

### Exercícios Resolvidos:

1) Uma urna tem 10 bolas verdes, 8 bolas amarelas, 6 bolas azuis e 4 bolas brancas. Retire,

aleatoriamente e sem reposição, 6 bolas desta urna.

```
> sample(rep(c("verde", "amarela", "azul", "branca"), c(10,8,6,4)), 6, FALSE)
```

ou ainda,

```
> sample(c("verde", "amarela", "azul", "branca"), 6, TRUE, c(10,8,6,4))
```

2) Use a função `sample` para gerar 10 números aleatórios de uma distribuição Bernoulli(0.6).

```
> sample(0:1,10, TRUE, c(0.4,0.6))
```

3) Seja a matriz:

$$lx = \begin{pmatrix} 1 & 10 \\ 2 & 20 \\ 3 & 30 \end{pmatrix}$$

Crie uma nova matriz, `lx2`, que é a matriz `lx` com as linhas permutadas, mantendo a correspondência entre as colunas (Ex: 2 e 20 têm que estar na mesma linha).

```
> lx <- matrix(c(1:3, 10 * (1:3)), ncol = 2)
```

```
> lx
```

	[1,]	[2,]
[1,]	1	10
[2,]	2	20
[3,]	3	30

```
> lx2 <- lx[sample(3),]
```

# Capítulo 7

## Funções Estatísticas Básicas

### 7.1 Estatística descritiva

Existe no R funções estatísticas que sumarizam os dados. A Tabela 7.1 apresenta algumas dessas funções.

Tabela 7.1: Funções do R mais comuns para resumo de dados.

Função	Descrição
<code>min</code>	retorna o menor elemento do objeto numérico
<code>max</code>	retorna o maior elemento do objeto numérico
<code>range</code>	devolve um vetor com $(\min(x), \max(x))$
<code>median</code>	calcula a mediana dos elementos do vetor (ou matriz) <code>x</code>
<code>quantile</code>	devolve o quantil de ordem $p$ dos elementos do vetor (ou matriz) <code>x</code> ; por omissão $p = \text{seq}(0, 1, 0.25)$ , isto é, devolve os extremos e os quartis de <code>x</code>
<code>mean</code>	calcula a média aritmética dos elementos do vetor (ou matriz) <code>x</code>
<code>var</code>	retorna a variância se o objeto é um vetor e a matriz de covariância se o objeto é uma matriz, considerando as colunas como variáveis
<code>cor</code>	retorna a matriz de correlação da matriz, considerando as colunas como variáveis
<code>sd</code>	devolve o desvio padrão dos elementos do vetor <code>x</code> ; é igual a $\sqrt{\text{var}(x)}$
<code>IQR</code>	devolve a amplitude inter-quartil dos elementos do vetor <code>x</code>
<code>summary</code>	devolve os extremos, os quartis e a média do vetor

O uso destas funções consiste basicamente em especificar a função desejada com o nome do objeto entre parênteses. Para ilustração, considere o objeto `x`:

```
> x <- c(5, 4, 2, 3, 3, 5, 5, 2, 4, 1, 4, 0, 1, 2, 9, 0, 3, 5, 1, 3)
```

Alguns exemplos:

```
> min(x)
[1] 0

> max(x)
[1] 9

> quantile(x, prob=0.25)
25%
1.75

> quantile(x, prob=c(0.25,0.75))
25% 75%
1.75 4.25

> mean(x)
[1] 3.1

> var(x)
[1] 4.621053

> summary(x)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.00   1.75   3.00   3.10   4.25   9.00
```

Vale ressaltar que, estas operações ainda podem ser atribuídas a um outro objeto:

```
> mediana <- median(x)
> mediana
[1] 3
```

Note que muitas das funções estatísticas anteriores podem ser usadas com um argumento adicional da seguinte forma:

`função(x, na.rm = FALSE)`

em que `na.rm` é um argumento que indica se os valores desconhecidos devem ser ou não ignorados. Caso o objeto `x` inclua elementos `NA` e se:

- `na.rm = FALSE` (o que ocorre por omissão) a função devolve `NA` ou uma mensagem de erro;
- `na.rm = TRUE` os valores `NA` são ignorados.

Exemplos:

```
> y <- c(NA, 53, 31, 15, 62)
> mean(y)
[1] NA

> mean(y, na.rm = TRUE)
[1] 40.25
```

## 7.2 Tabela de frequência

**Fatores:** um fator é um vetor que se usa para especificar uma classificação discreta em categorias dos componentes de outros vetores da mesma dimensão. Em R, existem fatores ordenados e não ordenados.

Como exemplo, suponha que se dispõe de uma amostra de 22 profissionais liberais de diversos estados do Brasil; o vetor `estados` contém as siglas do estado de cada um dos elementos desta amostra:

```
> estados <- c("SP", "RS", "RS", "RS", "MG", "RS", "RJ", "BA", "SP", "SP", "AM",
               "RS", "PR", "SP", "TO", "BA", "MG", "SP", "RJ", "TO", "RJ", "PA")
```

Para criar um fator a partir do objeto vetorial `estados`, usa-se a função `factor()`.

```
> festados <- factor(estados)
> festados
[1] SP RS RS RS MG RS RJ BA SP SP AM RS PR SP TO BA MG SP RJ TO RJ PA
Levels: AM BA MG PA PR RJ RS SP TO
```

Para obter as categorias de um fator usa-se a função `levels()`:

```
> levels(festados)
[1] "AM" "BA" "MG" "PA" "PR" "RJ" "RS" "SP" "TO"
```

### 7.2.1 Tabela de frequências a partir de fatores

Vimos que um fator define uma partição por categorias ou uma tabela de entrada simples. De modo semelhante, dois fatores definem uma tabela de dupla entrada, e assim sucessivamente. A função `table()` permite calcular tabelas de frequências a partir de fatores de igual comprimento.

Suponha, por exemplo, que `festados` é um fator de categorias que são as siglas dos

estados, associado a um vetor de dados. Para obtermos uma tabela de frequências do objeto `festados`, executamos no R as seguintes linhas de comandos:

```
> freqest <- table(festados)
> freqest
festados
AM BA MG PA PR RJ RS SP TO
 1  2  2  1  1  3  5  5  2
```

As frequências podem ser ordenadas e rotuladas pelos níveis ou categorias do fator usando a função `tapply`. Essa função rotula cada nível por um número inteiro (começando em 1 até o número total de níveis). Assim ao utilizar essa função, o número associado a cada nível é retornado. Exemplo:

```
> tapply(festados, festados)
[1] 8 7 7 7 3 7 6 2 8 8 1 7 5 8 9 2 3 8 6 9 6 4
```

Suponha agora um outro objeto `sexo`, sendo que o objeto `fsexo` é um fator que classifica ou agrupa os sexos por classes pré-definidas (“m” e “f”).

```
> sexo <-c("m", "f", "f", "f", "m", "m", "f", "f", "f", "m", "f", "m", "f",
           "f", "m", "m", "m", "f", "m", "f", "m", "m")
> fsexo <- factor(sexo)
> fsexo
[1] m f f f m m f f f m f m f f m m m f m f m m
Levels:  f m
```

Então, para calcular uma tabela de frequências de dupla entrada considerando os objetos `festados` e `fsexo` executamos as seguintes linhas de comando no R:

```
> table(festados,fsexo)
      fsexo
festados f  m
      AM 1  0
      BA 1  1
      MG 0  2
      PA 0  1
      PR 1  0
      RJ 1  2
```

```
RS 3 2
SP 3 2
TO 1 1
```

**Observações:**

- 1 - Podemos encontrar tabelas de frequências diretamente com os dados, sem precisar transformá-los em fatores.
- 2 - A extensão para tabelas de frequências de múltiplas entradas é imediata.

Outros exemplos com objetos vetoriais numéricos:

```
> n.filhos <- c(5, 4, 2, 3, 4, 1, 0, 1, 2, 2)
> curso <- c("mat","est","est","mat","eng","est","eng","eng","est","eng")

> table(n.filhos)
n.filhos
0 1 2 3 4 5 9
2 3 3 4 3 4 1

> table(curso)
curso
eng est mat
  4   4   2

> table(curso, n.filhos)
      n.filhos
curso 0 1 2 3 4 5
eng   1 1 1 0 1 0
est   0 1 2 0 1 0
mat   0 0 0 1 0 1
```

**7.2.2 Tabela de frequências relativas**

Existem diferentes maneiras de calcular a frequência relativa de um objeto, dependendo do tipo (vetor, matriz, ...) e dos elementos (numéricos, caracteres) definidos no objeto.

De forma geral uma tabela de frequência relativa pode ser obtida a partir de:

```
prop.table(x, margin = NULL)
```

em que:

`x` é uma tabela;

`margin` é um índice ou vetor de índices para gerar uma proporção marginal por linha (se

margin=1) e coluna (se margin=2).

Seguem alguns exemplos em que uma tabela de frequência relativa (proporção) pode ser obtida. Para isso, considere os objetos `sexo` e `estados`:

```
> sexo <-c("m", "f", "f", "f", "m", "m", "f", "f", "f", "m", "f", "m", "f",
           "f", "m", "m", "m", "f", "m", "f", "m", "m")
> estados <- c("SP", "RS", "RS", "RS", "MG", "RS", "RJ", "BA", "SP", "SP", "AM",
              "RS", "PR", "SP", "TO", "BA", "MG", "SP", "RJ", "TO", "RJ", "PA")

> prop.table(table(sexo), margin = NULL)
sexo
  f  m
0.5 0.5

> prop.table(table(estados), margin = NULL)
estados
      AM      BA      MG      PA      PR      RJ
0.04545455 0.09090909 0.09090909 0.04545455 0.04545455 0.13636364
      RS      SP      TO
0.22727273 0.22727273 0.09090909

> prop.table(table(estados,sexo), margin = NULL)
      sexo
estados  f      m
  AM 0.04545455 0.00000000
  BA 0.04545455 0.04545455
  MG 0.00000000 0.09090909
  PA 0.00000000 0.04545455
  PR 0.04545455 0.00000000
  RJ 0.04545455 0.09090909
  RS 0.13636364 0.09090909
  SP 0.13636364 0.09090909
  TO 0.04545455 0.04545455

> prop.table(table(estados,sexo), margin = 1)
      sexo
estados  f      m
  AM 1.0000000 0.0000000
  BA 0.5000000 0.5000000
  MG 0.0000000 1.0000000
```



```
PA 0.0000000 1.0000000
PR 1.0000000 0.0000000
RJ 0.3333333 0.6666667
RS 0.6000000 0.4000000
SP 0.6000000 0.4000000
TO 0.5000000 0.5000000
```

Outros Exemplos:

```
> chuva_mm <- matrix(c(80, 30, 100, 180, 60, 40, 90, 120), nrow = 2, ncol = 4,
  byrow = TRUE, dimnames = list(c("2007", "2008"), c("primavera",
    "verao", "outono", "inverno")))
> chuva_mm
      primavera verao outono inverno
2007         80    30    100     180
2008         60    40     90     120

> prop.table(chuva_mm, margin = 1)           # proporção por linha
      primavera      verao      outono  inverno
2007 0.2051282 0.07692308 0.2564103 0.4615385
2008 0.1935484 0.12903226 0.2903226 0.3870968

> prop.table(chuva_mm, margin = 2)           # proporção por coluna
      primavera      verao      outono  inverno
2007 0.5714286 0.4285714 0.5263158      0.6
2008 0.4285714 0.5714286 0.4736842      0.4
```

### 7.2.3 Tabela de frequências por classe

Utilizamos a função `cut` para construir tabelas de frequência por classe:

```
cut(x, breaks, labels = NULL, right = TRUE, diag.lab = 3, ...)
```

em que:

**x** é um vetor numérico que será convertido a um fator (classes) pelo comando;

**breaks** um vetor numérico de dois ou mais pontos de corte ou um único número (maior do que ou igual a 2), dando o número de intervalos em que **x** é para ser cortado;

**labels** nome para os níveis das categorias resultantes; **right** lógico, que indica se os intervalos devem ser fechados no lado direito (e aberto à esquerda) ou vice-versa;

**diag.lab** inteiro que é usado quando as etiquetas não são dadas. Determina o número dos dígitos usados na formação do corte.

A função `cut` divide a amplitude do objeto `x` em intervalos e codifica os valores de `x` de acordo com o intervalo que eles caem. Para ilustrar a construção dessa tabela, considere o conjunto de dados hipotético `peso`:

```
> set.seed(1)
> peso <- round(runif(50, 50, 90),1)
> c.peso <- cut(peso, breaks=7, right=TRUE)
> c.peso
 [1] (56.1,61.7] (61.7,67.3] (67.3,72.9] (84.1,89.7] (56.1,61.7] (84.1,89.7]
 [7] (84.1,89.7] (72.9,78.5] (72.9,78.5] (50.5,56.1] (56.1,61.7] (56.1,61.7]
[13] (72.9,78.5] (61.7,67.3] (78.5,84.1] (67.3,72.9] (78.5,84.1] (84.1,89.7]
[19] (61.7,67.3] (78.5,84.1] (84.1,89.7] (56.1,61.7] (72.9,78.5] (50.5,56.1]
[25] (56.1,61.7] (61.7,67.3] (50.5,56.1] (61.7,67.3] (84.1,89.7] (61.7,67.3]
[31] (67.3,72.9] (72.9,78.5] (67.3,72.9] (56.1,61.7] (78.5,84.1] (72.9,78.5]
[37] (78.5,84.1] (50.5,56.1] (78.5,84.1] (61.7,67.3] (78.5,84.1] (72.9,78.5]
[43] (78.5,84.1] (67.3,72.9] (67.3,72.9] (78.5,84.1] (50.5,56.1] (67.3,72.9]
[49] (78.5,84.1] (72.9,78.5]
7 Levels: (50.5,56.1] (56.1,61.7] (61.7,67.3] (67.3,72.9] ... (84.1,89.7]
```

Para o exemplo, a tabela de frequência por classe é confeccionada utilizando a função `table`:

```
> table(c.peso)
c.peso
(50.5,56.1] (56.1,61.7] (61.7,67.3] (67.3,72.9] (72.9,78.5] (78.5,84.1]
          5          7          7          7          8          10
(84.1,89.7]
          6
```

Podemos ainda especificar as classes desejadas especificando o argumento `breaks` como vetor. Vejamos o exemplo:

```
> c.peso2 <- cut(peso, breaks=seq(50,90,10), right=TRUE)
> c.peso2
 [1] (60,70] (60,70] (70,80] (80,90] (50,60] (80,90] (80,90] (70,80] (70,80]
[10] (50,60] (50,60] (50,60] (70,80] (60,70] (80,90] (60,70] (70,80] (80,90]
[19] (60,70] (80,90] (80,90] (50,60] (70,80] (50,60] (60,70] (60,70] (50,60]
[28] (60,70] (80,90] (60,70] (60,70] (70,80] (60,70] (50,60] (80,90] (70,80]
[37] (80,90] (50,60] (70,80] (60,70] (80,90] (70,80] (80,90] (70,80] (70,80]
[46] (80,90] (50,60] (60,70] (70,80] (70,80]
```

```
Levels: (50,60] (60,70] (70,80] (80,90]
> table(c.peso2)
c.peso2
(50,60] (60,70] (70,80] (80,90]
      10      13      14      13
```

### 7.3 Operações repetidas: função apply

Você pode usar a função `apply` para aplicar uma determinada função a todas as colunas ou linhas de uma matriz:

```
apply(matriz, MARGIN, FUN, ...)
```

em que:

**FUN** nome da função a ser aplicada;

**...** definição de argumentos opcionais da função **FUN**;

**MARGIN** 1 para aplicar a função **FUN** em cada linha da matriz e 2 para aplicar a função **FUN** em cada coluna da matriz.

Para ilustração, considere a matriz `notas.port` apresentada a seguir:

```
> notas.port <- matrix(c(2,5,0,8,9,5,8,4,6,7,4,5,6,3,6), ncol=3, byrow=T)
> notas.port
      [,1] [,2] [,3]
[1,]  2    5    0
[2,]  8    9    5
[3,]  8    4    6
[4,]  7    4    5
[5,]  6    3    6
```

Vamos criar um vetor que receba a média de cada linha do objeto `notas.port`:

```
> mean.notas <- apply(notas.port,1,mean)
> mean.notas
[1] 2.333333 7.333333 6.000000 5.333333 5.000000
```

# Capítulo 8

## Gráficos

Os gráficos (histogramas, *box-plots*, *scatter plots*, entre outros) são mostrados em janelas gráficas que devem ser abertas no menu **Tools/Grafic Device**. Entretanto, os comandos para execução dos gráficos devem ser feitos na linha de comando, como as funções. Por *default*, apenas um gráfico será desenhado na janela. Para obter mais que um gráfico em diferentes janelas deve-se usar o comando `x11()`. Se é desejado desenhar mais de um gráfico na mesma janela, usa-se antes de executar os comandos para desenhá-los a função `par(mfrow = c(nº de linhas, nº de colunas))`, imaginando-se a tela gráfica com uma matriz. Vale ressaltar que a tela gráfica continuará particionada desse modo até que se defina outra partição através do mesmo comando `par(mfrow = ...)`. Vejamos alguns gráficos úteis na análise exploratória dos dados:

### 8.1 Histograma

```
hist(x, probability = FALSE, ...)
```

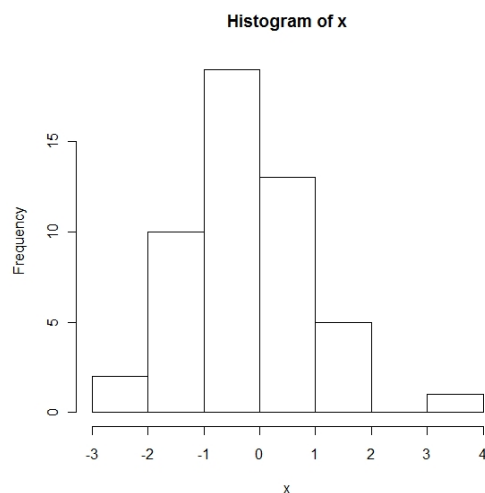
em que:

`x` é um vetor numérico de dados para o histograma;

`probability` se `TRUE` (ou `T`) a altura das barras do histograma serão as densidades de probabilidade; se `FALSE` (ou `F`), a altura das barras do histograma serão as contagens.

Exemplo:

```
> x <- rnorm(50,0,1)
> hist(x, probability = FALSE)
```



## 8.2 Gráfico de caixa (*box-plot*)

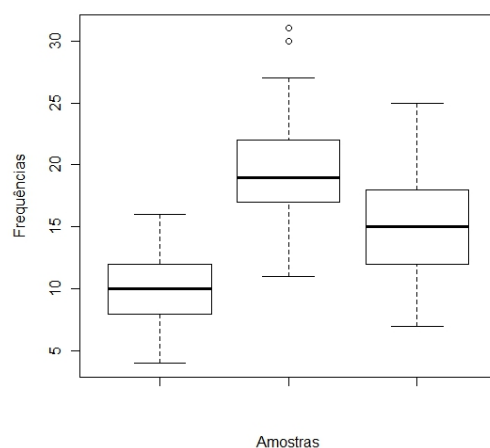
```
boxplot(x1, ..., xn, ...)
```

em que:

$x_1, \dots, x_n$  são objetos vetoriais com os dados. Produz  $n$  *box-plots* no mesmo gráfico.

Exemplo:

```
> boxplot(rpois(50,10), rpois(50,20), rpois(50,15), xlab = "Amostras",
          ylab = "Frequências")
```



## 8.3 Série temporal

```
ts.plot(tsplot(x1, ..., xn, ...))
```

em que:

$x_1, \dots, x_n$  são objetos vetoriais com os dados da série. Produz um gráfico com  $n$  séries temporais.

Exemplo:

```
> ts1 <- sin(seq(-pi, pi, len = 100)) + rnorm(100,0,0.1)
> ts2 <- cos(seq(-pi, pi, len = 100)) + rnorm(100,0,0.1)
```

Para convertendo os dados em série temporal, utilizamos a função `ts`:

```
> s1 <- ts (ts1)
> s2 <- ts (ts2)
> ts.plot(s1,s2, col = c("red","blue"))
```

## 8.4 Gráfico de dispersão

`plot(x,y,...)`

em que:

$x$  e  $y$  são objetos.

Algumas considerações:

- Se  $x$  e  $y$  forem objetos vetoriais, `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 (`plot(x)`) ou uma lista contendo os dois elementos  $x$  e  $y$  ou uma matriz de duas colunas;
- Se  $x$  é uma série de observações ao longo do tempo, este comando produz um gráfico da série temporal; se  $x$  é um vetor numérico, o comando cria um gráfico dos valores do vetor sobre os respectivos índices; se  $x$  é um vetor complexo, é produzido um gráfico da parte imaginária versus a parte real dos elementos;
- Se  $x$  é um `data.frame`, `plot(x)` criam gráficos das distribuições de todas as combinações das variáveis na ficha de dados.

Exemplo:

```
> x <- runif(30,1,50)
> y <- 2 + 0.5 * x + rnorm(30,0,1)
> plot(x,y)
```

## 8.5 Gráfico de dispersão de pares de variáveis

`pairs(x)`

em que:

`x` é uma matriz.

Exemplo:

```
> x1 <- runif(40,1,50)
> x2 <- runif(40,0,1)
> y <- 2 + 0.5 * x1 + 4 * x2 + rnorm(40,0,1)
> pairs(cbind(x1,x2,y))
```

## 8.6 Gráfico de barras

`barplot(x)`

em que:

`x` é um objeto vetorial.

Exemplo:

```
> x <- c(1,4,1,2,4,3,3,3,0,1,3,4,1,2,3)
> barplot(x)
> barplot(table(x))
```

## 8.7 Gráfico de setores ou pizza

`pie(x)`

em que:

`x` é um objeto vetorial.

Exemplo:

```
> x <- c(1,4,1,2,4,3,3,3,0,1,3,4,1,2,3)
> pie(x)
```

```
> pie(table(x))
```

## 8.8 Desenhando símbolos em um gráfico

Podemos fazer um gráfico de dados tridimensionais em duas dimensões codificando a terceira variável de acordo com o tamanho de um símbolo desenhado em cada localização x-y. Os símbolos podem ser círculos, quadrados, retângulos, *box-plots*, etc.

```
symbols(x, y, circles =, squares =, ...)
```

em que:

**x** e **y** são as coordenadas X e Y dos pontos (vetores).

**circles** = vetor contendo o raio dos círculos. **squares** = vetor contendo o comprimento do lado do quadrado.

Exatamente um dos argumentos **circles**, **squares**, dentre outros, deve ser dado. Como exemplo, vejamos as coordenadas de localização de certa espécie de árvore em um parque:

```
> x <- runif(20,0,100)
```

```
> y <- runif(20,0,100)
```

O Diâmetro (em centímetros) do tronco da árvore é dado por:

```
> z <- round(abs(rnorm(20,100,50)))
```

Como os dados, o mapa com a localização das árvores e representação de seu diâmetro são:

```
> symbols(x, y, circle = z, inches = 0.5)
```

## 8.9 Argumentos das funções gráficas

É possível definir uma série de argumentos para as funções gráficas, entre os quais:

**add = TRUE**: O gráfico criado será sobreposto ao gráfico atual, em vez de o apagar previamente (só está disponível para algumas funções).

**axes = FALSE**: Elimina os eixos. Esta opção é útil para que o usuário 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 em alguns 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"` ou `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.

`xlab = "string"` e `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 objetos 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.

Os gráficos do R são formados por pontos, linhas, texto e polígonos. Existem parâmetros gráficos que controlam como se desenhavam estes elementos gráficos, como 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 exata, 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, digite no R os seguintes comandos:

```
> plot(1, t = "n")  
> legend(locator(1), as.character(0:18), pch = 0:18)
```

aponte e dê um clique no topo do gráfico. Então, aparecerá uma lista de caracteres e com o seu respectivo código.

`lty`: É o tipo de linha. Embora alguns tipos de linhas não possam ser usadas em alguns dispositivos gráficos, o tipo 1 corresponde a uma linha sólida, e os tipos 2 e seguintes correspondem a linhas pontilhadas, tracejadas ou combinações destes tipos. Para obter os tipos de linhas, faça as ordens indicadas para `pch`, substituindo `pch` por `lty`.

`lwd`: Especifica a espessura da linha, medida em múltiplos da largura base. Afeta os eixos e as

linhas desenhadas com as funções `lines()`, etc.

**col:** Especifica a cor que se utiliza para os pontos, linha, texto, imagens e preenchimento de regiões. Casa 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. Note que `col = "blue"` também é válido. Para verificar as cores disponíveis no R basta utilizar o comando:

```
> colors()
```

**font:** 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:** Especificam a fonte a usar nos eixos.

**font.lab:** Especificam a fonte a usar nas etiquetas.

**font.main:** Especificam a fonte a usar no título principal.

**font.sub** - Especificam a fonte a usar no sub-título.

**cex:** 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, podendo especificar um número decimal.

## 8.10 Funções `points` e `lines`: adicionando pontos ou linhas ao gráfico corrente

Pode acontecer que as funções gráficas não produza exatamente o tipo de gráfico pretendido. Neste caso, podem usar-se algumas funções para adicionar informação adicional tal como pontos, linhas ou texto. Por exemplo:

`points(x, y)` - Adiciona pontos ao gráfico corrente

`lines(x, y)` - Adiciona pontos conectados com segmentos de linha ao gráfico corrente. A opção `type` da função `plot()` pode usar-se nesta função (os valores pré-definidos são "p" para `point()` e "l" para `lines()`).

## 8.11 Funções `polygon` e `text`: adicionando polígono ou texto ao gráfico corrente

Utilizamos a função `polygon` para desenhar uma linha poligonal no gráfico atual:

```
polygon(x, y,...)
```

em que:

$x$  e  $y$  são vértices (os pontos  $(x,y)$ ).

Opcionalmente pode sombrear ou preencher a figura com uma cor:

```
text(x, y, etiquetas,...)
```

que acrescenta texto aos pontos  $(x,y)$ . Geralmente `etiquetas` é um vetor de valores inteiros ou de caracteres, de modo a que `etiquetas[i]` é colocado no ponto  $(x[i],y[i])$ . O valor por *default* é `1:length(x)`.

Nota: Esta função é geralmente utilizada em comandos do tipo:

```
> plot(x, y, type = "n ")
```

```
> text(x, y, etiquetas)
```

## 8.12 Função `abline`: adicionando uma linha ao gráfico corrente

`abline(a, b)`: Acrescenta uma reta de declive  $b$  e ordenada na origem  $a$  ao gráfico atual.

`abline(h = y)`: Acrescenta uma linha horizontal à altura  $y$ .

`abline(v = x)`: Acrescenta uma linha vertical no ponto de abscissa  $x$ .

`abline(lm.obj)`: `lm.obj` refere-se a uma lista com uma componente designada coeficientes 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.

## 8.13 Função `legend`: adicionando legenda ao gráfico corrente

A função `legend` aplica a legenda ao gráfico atual, na posição especificada:

```
legend(x, y, legenda,...)
```

As fontes a usar, estilos de linha, cores, entre outros, são definidos no vetor `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.

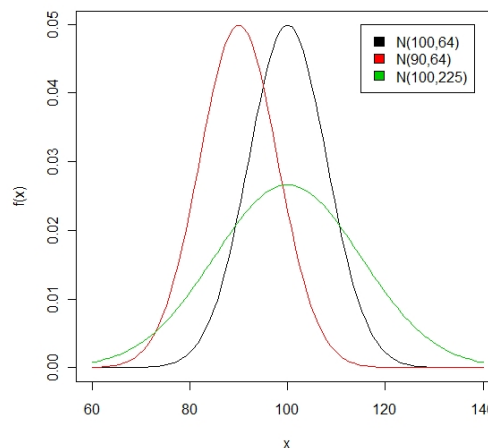
`legend(...,lty = v)`: Tipos de linha.

`legend(...,lwd = v)`: Espessura de linha.

`legend(..., pch = v)`: Caracteres para desenhar (vetor alfanumérico).

Exemplo:

```
> plot(function(x) dnorm(x, 100, 8), 60, 140, ylab = "f(x)")
> plot(function(x) dnorm(x, 90, 8), 60, 140, add = TRUE, col = 2)
> plot(function(x) dnorm(x, 100, 15), 60, 140, add = TRUE, col = 3)
> legend("topright", c("N(100,64)", "N(90,64)", "N(100,225)"), fill = 1:3)
```



## 8.14 Funções gráficas interativas

O R dispõe de funções que permitem extrair ou adicionar informação a um gráfico utilizando o *mouse*. A função mais fácil é a `locator()`. Essa função permite que o usuário selecione regiões do gráfico usando o botão esquerdo do *mouse*, até que se tenha selecionado um máximo de `n` pontos (por *default*, `n=512`) ou até pressionar o botão direito para terminar a seleção:

`locator(n, type)`

O argumento `type` permite acrescentar elementos ao gráfico. Por *default*, o argumento `type` está desativado. A função `locator()` devolve uma lista com as coordenadas (`x`, `y`) dos pontos selecionados.

Outra função é `identify`, que permite identificar os pontos pré-definidos utilizando o botão esquerdo do *mouse* e colocando a etiqueta definida pelo `label` junto ao ponto. Se o `label` não é definido, o ponto será identificado pelo seu índice:

`identify(x, y, labels)`

É útil quando existe um vetor de índices ou de etiquetas associado aos valores `x` e `y`, permitindo identificar pontos do gráfico com o índice ou etiqueta respectiva.

# Capítulo 9

## Funções

Uma das grandes vantagens do R é incluir uma linguagem de programação, que nos permite escrever nossas próprias funções. Isso o torna uma ferramenta poderosa para testar novas metodologias e realizar simulações. Você poderá construir uma função completamente nova no R (um novo estimador que você está propondo, por exemplo) ou apenas uma modificação personalizada de uma função já existente. Você pode ainda desejar usar as funções já existentes de modo repetido no seu conjunto de dados e seu trabalho será facilitado incorporando estas tarefas em uma única função. Veremos agora os conceitos básicos para a construção de funções no R e alguns exemplos simples.

### 9.1 Sintaxe geral

A sintaxe geral para definir uma nova função é a partir da linha de comando:

```
nome <- function(argumentos) {corpo}
```

em que:

**nome** nome que você escolhe para a função.

**argumentos** os argumentos necessários para a função, separados por vírgula.

**corpo** conjunto de comandos (tarefas) necessários para que a função retorne o valor desejado.

Como exemplo, apresentamos uma função que, dado um vetor de valores, retorne os valores padronizados:

```
> padroniza <- function(x) {  
+ z <- (x - mean(x)) / (sd(x))  
+ return(z)  
}
```

Usando a nova função **padroniza**:

```
> x <- 1:6
> Zx <- padroniza(x)
> Zx
[1] -1.3363062 -0.8017837 -0.2672612  0.2672612  0.8017837  1.3363062
```

Uma observação sobre a função `padroniza` é que `x` é o único argumento da função. Note que `z` não foi criado no diretório de dados (não é um objeto do R, e sim da função).

O último objeto que se faz referência antes de “fechar a função” com “`}`” é o objeto que será retornado pela função. Nesse caso o objeto da função `z`. Tendo visto esse exemplo bem simples, vamos generalizar algumas observações sobre os elementos da sintaxe geral:

1. Você pode especificar valores *default* para alguns ou todos os argumentos escrevendo na juntamente com o nome do argumento o valor (nome do argumento= valor). Os argumentos que não tiverem um valor *default* são os argumentos obrigatórios ao chamar na função.
2. Os objetos criados dentro do corpo da função são locais àquela função, ou seja, só existem na memória apenas durante a execução da função.
3. Você pode colocar comentários no corpo da função usando o símbolo `#` antes do comentário.
4. A função irá retornar apenas um objeto, que pode ser um vetor, uma matriz ou uma lista, etc. O nome do objeto a ser referenciado deve aparecer (sozinho) na última linha antes do “`}`” final. Se nenhum nome aparecer no final do corpo da função, será retornado valor da última expressão avaliada no corpo da função.

## 9.2 Expressando condições

Em algumas funções que construímos é necessário avaliar condições para que uma determinada tarefa seja realizada, especialmente dentro de *loops* (`for`, `while` e `repeat` descritos na próxima seção). Os cálculos condicionais são feitos através da conhecida construção `if-else`:

```
if (condição1) { tarefa a ser realizada se condição1 = TRUE}
else if (condição2) { tarefa a ser realizada se condição2 = TRUE}
else if (condição3) { tarefa a ser realizada se condição3 = TRUE}
...
else { tarefa a ser realizada se todas as condições = FALSE}
```

em que `condição` é uma expressão que resulta em um único valor lógico (`TRUE` ou `FALSE`).

As condições 1, 2, ..., N são mutuamente excludentes e uma certa condição será realizada se todas as outras forem **FALSE**. Mas podemos ter estruturas mais simples, como no exemplo a seguir.

Você pode usar uma expressão condicional **if** simples, sem **else**:

```
if(mean(x) > median(x)) {estimativa <- y}
```

ou colocar uma alternativa para o caso em que a condição seja falsa:

```
if(mean(x) > median(x)) {estimativa <- mean(x)}  
else {estimativa <- median(x)}
```

Podemos testar múltiplas condições usando os operadores **&** e **|** representam E e OU, respectivamente:

```
if (condição1 & condição2) {tarefa a ser realizada se as duas condições são TRUE}  
if (condição1 | condição2) {tarefa a ser realizada se ao menos uma das condições  
é TRUE}
```

Exemplo:

```
if (idade < 10) {idade2 <- 1}  
else if (idade >= 10 & idade < 20) {idade2 <- 2}  
else if (idade >= 20 & idade < 40) {idade2 <- 3}  
else if (idade >= 40 & idade < 60) {idade2 <- 4}  
else {idade2 <- 5}
```

## 9.3 Iteração

Voce pode desejar fazer uma série de tarefas repetidas dentro de uma função (ou até mesmo fora). As tarefas iterativas (*loops*) podem ser feitas através dos comandos **for**, **while** e **repeat**.

### 9.3.1 for

A estrutura **for** permite que uma tarefa seja repetida à medida que uma variável assume valores em uma seqüência específica:

```
for (variável in seqüência) {tarefas}
```

Como exemplo, vamos escrever uma função que gere um vetor com as somas acumuladas dos elementos de um vetor dado (tarefa da função **cumsum**).

```
> soma.fun <- function(x) {  
+ n <- length(x)  
+ soma <- NULL          # vetor onde serão armazenadas as somas  
+ y <- 0  
+   for (i in 1:n) {    # percorrendo cada elemento do vetor x  
+     y <- y + x[i]  
+     soma <- c(soma, y)  
+   }  
+ soma  
+ }  
> soma.fun(1:5)  
[1] 1 3 6 10 15
```

### 9.3.2 while

A estrutura **while** permite que uma tarefa seja repetida enquanto uma condição (expressão com resultado lógico) é verdadeira:

```
while (condição) {tarefas}
```

Exemplo:

```
> cont <- 1  
> while(cont < 10) {cont <- cont + 1}  
> cont  
[1] 10
```

### 9.3.3 repeat

A estrutura **repeat** permite que uma tarefa seja repetida indefinidamente, a não ser que a condição no comando **break** seja satisfeita:

```
repeat {tarefas (incluindo avaliação para break)}
```

Exemplo:

```
> repeat{  
+ numero <- rnorm(1,0,1)  
+   if (numero > 0) break  
+ }  
> numero  
[1] 1.329799
```



# Capítulo 10

## R na internet

O *software* R conta com um enorme acervo de arquivos de ajuda que são disponíveis no programa e na internet. Mais informações sobre o funcionamento do R podem ser encontradas nos endereços abaixo:

1. <http://www.r-project.org/>

Página principal do projeto R. Na seção Documentation na página do Projeto R há manuais e questões frequentemente perguntadas (FAQ's) e estes estão também disponíveis localmente quando R é instalado. Há também um boletim, páginas de ajuda e listas de publicações em revistas científicas. As listas de e-mail são fontes extremamente úteis de informações sobre como executar tarefas em R, além de dicas de como resolver problemas em que não há ajuda documentada.

2. <http://www.statmethods.net/index.html>

O principal objetivo deste *site* é ajudar usuários já familiarizado com métodos estatísticos mas que pouco utilizam a linguagem R em seus trabalhos. É uma referência de fácil acesso.

3. <http://www.rseek.org/>

Rseek.org torna mais fácil a busca de informações sobre o R filtrando *sites*. É como uma pesquisa no Google, mas restringe a pesquisa apenas aos *sites* conhecidos por conterem informações sobre o R. A lista dos *sites* pesquisados é mantida pelo criador de Rseek.org, Sasha Goodman, e um grupo de voluntários.

4. <http://bm2.genes.nig.ac.jp/RGM2/index.php>

Neste *site* é possível encontrar uma coleção de gráficos e todos os pacotes do R.

5. <http://www.sciviews.org/Tinn-R/>

*Site* com algumas informações sobre o editor Tinn-R e link para fazer o download.

6. <http://addictedtor.free.fr/graphiques/>

Neste *site* pode-se obter informações de gráficos no *software* R.

# Referências Bibliográficas

- R Development Core Team (2010). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL <http://www.R-project.org>.
- Reis, E. A. (1997). Noções Básicas de *S-PLUS* for Windows, 1<sup>a</sup> edição. Universidade Federal de Minas Gerais, Departamento de Estatística, Belo Horizonte. URL: <http://www.est.ufmg.br/portal/arquivos/rts/rte9703.pdf>
- Reisen, V. A. & Silva, A. N. (2011). O uso da linguagem R para cálculos de Estatística Básica, Editora da Universidade Federal do Espírito Santo, Vitória/ES.
- Ribeiro Jr., P. J. (2003). Curso sobre o Programa Computacional R. Universidade Federal do Paraná, Departamento de Estatística, Curitiba/PR.
- Venables, W. N.; Smith, D. M. (1992). An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics.