Estatística Computacional com R

Fernando P. Mayer Wagner H. Bonat

Laboratório de Estatística e Geoinformação (LEG)
Departamento de Estatística (DEST)
Universidade Federal do Paraná (UFPR)

2018-07-24
Última atualização: 2018-07-25

Sumário

Prefácio

Essa apostila é destinada inicialmente aos alunos da disciplina CE083 - Estatística Computacional I, do curso de Estatística da UFPR.

Informação de sessão

Abaixo seguem as informações do ambiente em que o documento foi gerado pela última vez.

```
Wednesday, 25 July, 2018, 02:05
---------------------------------------
R version 3.5.0 (2017-01-27)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 14.04.5 LTS

Matrix products: default
BLAS: /home/travis/R-bin/lib/R/lib/libRblas.so
LAPACK: /home/travis/R-bin/lib/R/lib/libRlapack.so

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] MASS_7.3-50  gdata_2.18.0 knitr_1.20

loaded via a namespace (and not attached):
 [1] Rcpp_0.12.18    bookdown_0.7.15 gtools_3.8.1    digest_0.6.15
 [5] rprojroot_1.3-2 backports_1.1.2 magrittr_1.5    evaluate_0.11
 [9] highr_0.7       stringi_1.2.4   rstudioapi_0.7  rmarkdown_1.10
[13] tools_3.5.0     stringr_1.3.1   xfun_0.3        yaml_2.1.19
[17] compiler_3.5.0  htmltools_0.3.6
```

Parte I

R básico

Capítulo 1

Computação científica e interação com o R

1.1 Interagindo com o computador

O que significa este ícone?

- É um documento do Microsoft Excel?
- É um arquivo de **texto pleno**, separado por vírgulas (CSV *comma separated values*)
- De fato, o nome do arquivo é `final.csv` e não `final`
- O Excel pode sim abrir este arquivo... assim como milhares de outros programas!

O que está acontecendo?

- O computador (leia-se, nesse caso, o sistema operacional Windows) "proteje" o usuário dos detalhes sujos
- Isso é ruim? **Sim!**
- O usuário se acostuma com o computador ditando as regras
- É importante lembrar que é você quem deve dizer o que o computador deve fazer (nesse caso, com qual programa abrir certo arquivo)

O que deve acontecer?

- Para a maioria dos usuários, a interação com o computador se limita a clicar em links, selecionar menus e caixas de diálogo
- O problema com essa abordagem é que parece que o usuário é controlado pelo computador
- A verdade deve ser o oposto!
- É o usuário que possui o controle e deve dizer para o computador exatamente o que fazer
- Escrever código ainda tem a vantagem de deixar registrado tudo o que foi feito

1.2 Editores de texto

Uma característica importante de códigos de programação é que eles são em **texto puro**, por isso precisamos de um bom **editor de textos**

Características de um bom editor:

- **Identação automática**
- **Complementação de parênteses**
- **Destaque de sintaxe** (*syntax highlighting*)
- **Numeração de linhas**
- **Auto completar comandos**

1.2.1 Editores para R

Windows:

- Interface padrão: pouco recomendado
- Tinn-R

Linux:

- Vim-R-plugin
- Gedit-R-plugin

Todas as plataformas:

- Rstudio: recomendado para iniciantes
- Emacs + ESS: altamente recomendado

1.3 R

> *"The statistical software should help, by supporting each step from user to programmer, with as few intrusive barriers as possible."*
>
> *"... to turn ideas into software, quickly and faithfully."*
>
> — John M. Chambers

O R é um dialeto do S e:

- *Ambiente* estatístico para análise de dados e produção de gráficos
- Uma completa linguagem de programação:
  - Interpretada (contrário de compilada)
  - Orientada a objetos:

  Tudo no R é um objeto...
- Livre distribuição (código-aberto)
- Mais de 10000 pacotes adicionais

Pequeno histórico:

- 1980: Linguagem S: desenvolvida por R. Becker, J. Chambers e A. Wilks (AT&T Bell Laboratories)
- 1980: Versão comercial: S-Plus (Insightful Corporation)
- 1996: Versão livre: R desenvolvido por R. Ihaka e R. Gentleman (Universidade de Auckland)
- 1997: R Development Core Team
- Hoje: 20 desenvolvedores principais e muitos outros colaboradores em todo o mundo
- Estatísticos, matemáticos e programadores

1.3.1 Configuração inicial

- O **diretório de trabalho** é uma pasta onde o R será direcionado. Todos os arquivos que serão importados (base de dados, ...) ou exportados (base de dados, gráficos, ...) por ele ficarão nesta pasta.
- Existem duas maneiras de configurar o diretório de trabalho (suponha que vamos usar a pasta `~/estatcomp1`):
- 1) Utilizando a função `setwd()` dentro do R:

```
setwd("~/estatcomp1")
```

- 2) Pelo menu do RStudio em `Session > Set Working Directory > Choose Directory...` Confira o diretório que está trabalhando com a função

```
getwd()
```

1.3.2 O R como uma calculadora

O símbolo > indica que o R está pronto para receber um comando:

```
> 2 + 2
[1] 4
```

O símbolo > muda para + se o comando estiver incompleto:

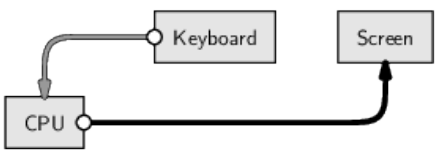
```
> 2 *
+ 2
[1] 4
```

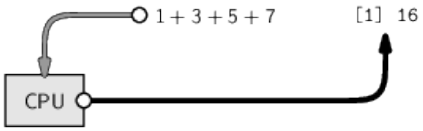
Espaços entre os números não fazem diferença:

```
> 2+          2
[1] 4
```

1.3.3 Para onde vão os resultados?

```
> 1 + 3 + 5 + 7
[1] 16
```





- Note que o resultado é apenas mostrado na tela, nada é salvo na memória (por enquanto)

1.3.4 O editor de scripts

- Para criar rotinas computacionais é necessário utilizar um editor de scripts.
- Clique em `File > New file > R script`. Salve com a extensão `.R`.
- Para enviar comandos diretamente para o console, selecione-os e aperte `Ctrl + <Enter>`.
- Para adicionar comentários ao script, utiliza-se o símbolo # antes do texto e/ou comandos. O que estiver depois do símbolo não será interpretado pelo R. Portanto:

```
2 + 2     # esta linha será executada
# 2 + 2     esta linha não será executada
```

1.3.5 Operadores aritméticos

Operador	Significado
`+`	adição
`-`	subtração
`*`	multiplicação
`/`	divisão
`^`	potência
`exp()`	exponencial
`sqrt()`	raíz quadrada
`factorial()`	fatorial
`log(); log2(); log10()`	logaritmos

1.3.6 Ordens de execução

As operações são realizadas sempre seguindo as prioridades:

1. De dentro para fora de parênteses `()`
2. Multiplicação e divisão
3. Adição e subtração

```
> 5 * 2 - 10 + 7
[1] 7
> 5 * 2 - (10 + 7)
[1] -7
> 5 * (2 - 10 + 7)
[1] -5
> 5 * (2 - (10 + 7))
[1] -75
```

Exercícios

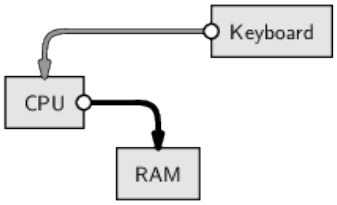
1. Calcule a seguinte equação: $32 + 16^2 - 25^3$
2. Divida o resultado por 345
3. Qual o resultado da expressão $\frac{e^{-2}2^4-1}{4!}$?
4. E do logaritmo desta expressão?

1.3.7 "Salvando" resultados

Do exercício anterior

```
> x <- 32 + 16^2 - 25^3
> x
[1] -15337
> x/345
[1] -44.45507
> (y <- (exp(-2) * 2^4 - 1)/factorial(4))
[1] 0.04855686
> log(y)
[1] -3.02502
```

Quando criamos uma variável (x, y), ela fica armazenada **temporariamente** na memória RAM.



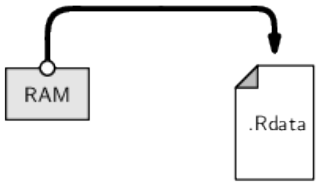
Para saber quais objetos estão criados, usamos a **função** `ls()`

```
> ls()
[1] "x" "y"
```

Estas variáveis ficam armazenadas no chamado *workspace* do R

- O *workspace* consiste de tudo que or criado durante uma sessão do R, armazenado na memória RAM

Para efetivamente salvar esas variáveis, podemos armazenar esse *workspace* do R em disco, em um arquivo chamdo `.Rdata`



- Quando o R é iniciado em um diretório com um arquivo `.Rdata`, as variáveis salvas são automaticamente carregadas
- No entanto, é sempre melhor salvar os dados e o **script**, assim é possível gerar os resultados novamente, sem salvar nada sem necessidade
- Veremos mais pra frente como salvar variáveis específicas, por exemplo, resultados de uma análise que leva muito tempo para ser executada
- O mais importante é salvar o **código**, assim sabemos **como** chegamos a determinado resultado, e podemos recriá-lo depois

1.3.8 Finalizando o programa

A qualquer momento durante uma sessão você pode usar o comando

```
> save.image()
```

No RStudio:

- `File > Save As...`
- Na janela que abrir, digite o nome do arquivo (por exemplo `script_aula1`) e salve
- Automaticamente o script será salvo com a extensão `.R` (nesse caso `script_aula1.R`) no diretório de trabalho que você configurou no início

Alternativamente, você pode também salvar toda sua área de trabalho, clicando em `Workspace > Save As Default Workspace`. Este processo irá gerar dois arquivos:

- `.Rdata`: contém todos os objetos criados durante uma sessão. Não é necessário (e nem recomendado) dar um nome antes do ponto. Dessa forma, a próxima vez que o programa for iniciado neste diretório, a área de trabalho será carregada automaticamente.
- `.Rhistory`: um arquivo texto que contém todos os comandos que foram digitados no console.

Referências

- Leek, J. The Elements of Data Analytic Style. Leanpub, 2015.
- Murrell, P. Introduction to data technologies. Boca Raton: Chapman & Hall/CRC, 2009.
- Peng, RD. R programming for data science. Leanpub, 2015.

Capítulo 2

Objetos e classes

2.1 Funções e argumentos

As funções no R são definidas como:

```
nome(argumento1, argumento2, ...)
```

Exemplo: função `runif()` (para gerar valores aleatórios de uma distribuição uniforme):

```
runif(n, min = 0, max = 1)
```

```
runif(10, 1, 100)
 [1] 31.468845 26.509578 55.679921  6.581932 47.386379 48.893303 81.427859
 [8] 37.661733 55.109301 17.855943
```

Argumentos que já possuem um valor especificado (como `max` e `min`) podem ser omitidos:

```
runif(10)
```

Se os argumentos forem nomeados, a ordem deles dentro da função não tem mais importância:

```
runif(min = 1, max = 100, n = 10)
```

Argumentos nomeados e não nomeados podem ser utilizados, desde que os não nomeados estejam na posição correta:

```
runif(10, max = 100, min = 1)
```

2.1.1 Outros tipos de argumentos

Exemplo: função `sample()`:

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

- `x` e `size` devem ser obrigatoriamente especificados
- `replace` é lógico: `TRUE` (`T`) ou `FALSE` (`F`)
- `prob` é um argumento vazio ou ausente ("opcional")

Exemplo: função `plot()`:

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

- "`...`" permite especificar argumentos de outras funções (por exemplo `par()`)

Para ver todos os argumentos disponíveis de uma função, podemos usar a função `args()`

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

2.2 Mecanismos de ajuda

Argumentos e detalhes do funcionamento das funções:

```
?runif
```

ou

```
help(runif)
```

A documentação contém os campos:

- **Description:** breve descrição
- **Usage:** função e todos seus argumentos
- **Arguments:** lista descrevendo cada argumento
- **Details:** descrição detalhada
- **Value:** o que a função retorna
- **References:** bibliografia relacionada
- **See Also:** funções relacionadas
- **Examples:** exemplos práticos

Procura por nomes de funções que contenham algum termo:

```
apropos("mod")
apropos("model")
```

Procura por funções que contenham `palavra` em qualquer parte de sua documentação:

```
help.search("palavra")
```

Ajuda através do navegador (também contém manuais, . . .):

```
help.start()
```

Sites para busca na documentação dos diversos pacotes:

- RDocumentation https://www.rdocumentation.org/
- R Package Documentation https://rdrr.io/
- R Contributed Documentation (várias línguas) https://cran.r-project.org/other-docs.html

Os pacotes do R contém funções específicas para determinadas tarefas, e estendem a instalação básica do R. Atualmente existem mais de 10000 pacotes disponíveis no CRAN, além de diversos outros hospedados em sites como Github, por exemplo.

Ao instalar o R, os seguintes pacotes já vêm instalados (fazem parte do chamado "R core"):

```
NULL
```

No entanto, nem todos são carregados na inicialização do R. Por padrão, apenas os seguintes pacotes são carregados automaticamente:

```
[1] "knitr"     "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
```

Para listar os pacotes carregados, use a função

```
search()
```

Note que o primeiro elemento, `.GlobalEnv`, será sempre carregado pois ele é o *ambiente* queirá armazenar (e deixar disponível) os objetos criados pelo usuário. Para carregar um pacote instalado, usamos a função `library()`, por exemplo

```
library(lattice)
search()
```

Isso tornará todas as funções do pacote `lattice` disponíveis para uso.

Para instalar um pacote usamos a função `install.packages()`. Sabendo o nome do pacote, por exemplo, `mvtnorm`, fazemos

```
install.packages("mvtnorm")
```

Se o diretório padrão de instalação de um pacote for de acesso restrito (root por exemplo), o R irá perguntar se você gostaria de instalar o pacote em uma biblioteca pessoal, e sugerirá um diretório que possui as permissões necessárias. Você pode se antecipar e já definir e criar um diretório na sua pasta pessoal, e instalar os pacotes sempre nesse local. Por exemplo, defina `~/R/library` como sua biblioteca pessoal. Para instalar os pacotes sempre nesse diretório faça:

```
install.packages("mvtnorm", lib = "~/R/library")
```

Para verificar as bibliotecas disponíveis e se existem pacotes para ser atualizados, use

```
packageStatus()
```

Para atualizar automaticamente todos os pacotes faça

```
update.packages(ask = FALSE)
```

2.3 Criando uma função

A ideia original do R é transformar usuários em programadores

> *"... to turn ideas into software, quickly and faithfully."*
>
> – John M. Chambers

Criar funções para realizar trabalhos específicos é um dos grandes poderes do R

Por exemplo, podemos criar a famosa função

```
ola.mundo <- function(){
    writeLines("Olá mundo")
}
```

E chama-la através de

```
ola.mundo()
Olá mundo
```

A função acima não permite alterar o resultado de saída. Podemos fazer isso incluindo um **argumento**

```
ola.mundo <- function(texto){
    writeLines(texto)
}
```

E fazer por exemplo

```
ola.mundo("Funções são legais")
Funções são legais
```

(Veremos detalhes de funções mais adiante)

Exercícios

1. Usando a função `runif()` gere 30 números aleatórios entre:
   - 0 e 1
   - -5 e 5
   - 10 e 500

alternando a posição dos argumentos da função.

2. Veja o help da função (?) `"+"`
3. Crie uma função para fazer a soma de dois números: `x` e `y`
4. Crie uma função para simular a jogada de um dado.
5. Crie uma função para simular a jogada de dois dados.

2.4 Objetos

O que é um objeto?

- Um **símbolo** ou uma **variável** capaz de armazenar qualquer valor ou estrutura de dados

Por quê objetos?

- Uma maneira simples de acessar os dados armazenados na memória (o R não permite acesso direto à memória)

Programação:

- Objetos ⇒ Classes ⇒ Métodos

  "Tudo no R é um objeto."

  "Todo objeto no R tem uma classe"

- **Classe:** é a definição de um objeto. Descreve a forma do objeto e como ele será manipulado pelas diferentes funções
- **Método:** são **funções genéricas** que executam suas tarefas de acordo com cada classe. Duas das funções genéricas mais importantes são:
  - `summary()`
  - `plot()`

Veja o resultado de

```
methods(summary)
methods(plot)
```

(Veremos mais detalhes adiante).

A variável `x` recebe o valor 2 (tornando-se um objeto dentro do R):

```
x <- 2
```

O símbolo `<-` é chamado de **operador de atribuição**. Ele serve para atribuir valores a objetos, e é formado pelos símbolos `<` e `-`, obrigatoriamente **sem espaços**.

Para ver o conteúdo do objeto:

```
x
[1] 2
```

Observação: O símbolo = pode ser usado no lugar de <- mas não é recomendado.

Quando você faz

```
x <- 2
```

está fazendo uma **declaração**, ou seja, declarando que a variável x irá agora se tornar um objeto que armazena o número 2. As declarações podem ser feitas uma em cada linha

```
x <- 2
y <- 4
```

ou separadas por ;

```
x <- 2; y <- 4
```

Operações matemáticas em objetos:

```
x + x
[1] 4
```

Objetos podem armazenar diferentes estruturas de dados:

```
y <- runif(10)
y
 [1] 0.6249965 0.8821655 0.2803538 0.3984879 0.7625511 0.6690217 0.2046122
 [8] 0.3575249 0.3594751 0.6902905
```

Note que cada objeto só pode armazenar uma estrutura (um número ou uma sequência de valores) de cada vez! (Aqui, o valor 4 que estava armazenado em y foi sobrescrito pelos valores acima.)

2.4.1 Nomes de objetos

- Podem ser formados por letras, números, "_", e "."
- Não podem começar com número e/ou ponto
- Não podem conter espaços
- Evite usar acentos
- Evite usar nomes de funções como:

`c q t C D F I T diff df data var pt`

- O R é *case-sensitive*, portanto:

`dados` ≠ `Dados` ≠ `DADOS`

2.4.2 Gerenciando a área de trabalho

Liste os objetos criados com a função `ls()`:

```
ls()
```

Para remover apenas um objeto:

```
rm(x)
```

Para remover outros objetos:

```
rm(x, y)
```

Para remover todos os objetos:

```
rm(list = ls())
```

Cuidado! O comando acima apaga todos os objetos na sua área de trabalho sem perguntar. Depois só é possível recuperar os objetos ao rodar os script novamente.

Exercícios

1. Armazene o resultado da equação $32 + 16^2 - 25^3$ no objeto `x`
2. Divida `x` por 345 e armazene em `y`
3. Crie um objeto (com o nome que você quiser) para armazenar 30 valores aleatórios de uma distribuição uniforme entre 10 e 50
4. Remova o objeto `y`
5. Remova os demais objetos de uma única vez
6. Procure a função utilizada para gerar numeros aleatórios de uma distribuição de Poisson, e gere 100 valores para a VA $X \sim \text{Poisson}(5)$.

2.5 Tipos e classes de objetos

Para saber como trabalhar com dados no R, é fundamental entender as possíveis estruturas (ou tipos) de dados possíveis. O formato mais básico de dados são os vetores, e a partir deles, outras estruturas mais complexas podem ser construídas. O R possui dois tipos básicos de vetores:

- **Vetores atômicos**: existem seis tipos básicos:
- `double`
- `integer`
- `character`
- `logical`
- `complex`
- `raw`

Os tipos `integer` e `double` são chamados conjuntamente de `numeric`. - **Listas**: também chamadas de *vetores recursivos* pois listas podem conter outras listas.

A principal diferença entre vetores atômicos e listas é que o primeiro é **homogêneo** (cada vetor só pode conter um tipo), enquanto que o segundo pode ser **heterogêneo** (cada vetor pode conter mais de um tipo).

Um vetor atômico só pode conter elementos de um mesmo tipo

Um vetor, como o próprio nome diz, é uma estrutura unidimensional, mas na maioria das vezes iremos trabalhar com estruturas de dados bidimensionais (linhas e colunas). Portanto diferentes estruturas (com diferentes dimensões) podem ser criadas a partir dos vetores atômicos. Quando isso acontece, temos o que é chamado de **classe** de um objeto. Embora os vetores atômicos só possuam seis tipos básicos, existe um número muito grande de classes, e novas são inventadas todos os dias. E mesmo que um objeto seja de qualquer classe, ele sempre será de um dos seis tipos básicos (ou uma lista).

Para verificar o tipo de um objeto, usamos a função `typeof()`, enquanto que a classe é verificada com a função `class()`. Vejamos alguns exemplos:

```
## double
x <- c(2, 4, 6)
typeof(x)
[1] "double"
class(x)
[1] "numeric"
## integer
x <- c(2L, 4L, 6L)
```

```
typeof(x)
[1] "integer"
class(x)
[1] "integer"
## character
x <- c("a", "b", "c")
typeof(x)
[1] "character"
class(x)
[1] "character"
## logical
x <- c(TRUE, FALSE, TRUE)
typeof(x)
[1] "logical"
class(x)
[1] "logical"
## complex
x <- c(2 + 1i, 4 + 1i, 6 + 1i)
typeof(x)
[1] "complex"
class(x)
[1] "complex"
## raw
x <- raw(3)
typeof(x)
[1] "raw"
class(x)
[1] "raw"
```

2.5.1 Vetores numéricos

Características:

- Coleção ordenada de valores
- Estrutura unidimensional

Usando a função `c()` para criar vetores:

```
num <- c(10, 5, 2, 4, 8, 9)
num
[1] 10  5  2  4  8  9
typeof(num)
[1] "double"
class(num)
[1] "numeric"
```

Por que `numeric` e não `integer`?

```
x <- c(10L, 5L, 2L, 4L, 8L, 9L)
x
[1] 10  5  2  4  8  9
typeof(x)
[1] "integer"
class(x)
[1] "integer"
```

Para forçar a representação de um número para inteiro é necessário usar o sufixo `L`.

Note que a diferença entre `numeric` e `integer` também possui impacto computacional, pois o armazenamento de números inteiros ocupa menos espaço na memória. Dessa forma, esperamos que o vetor `x` acima ocupe menos espaço na memória do que o vetor `num`, embora sejam aparentemente idênticos. Veja:

```
object.size(num)
[1] 96 bytes
object.size(x)
[1] 80 bytes
```

A diferença pode parecer pequena, mas pode ter um grande impacto computacional quando os vetores são formados por milhares ou milhões de números.

2.5.1.1 Representação numérica dentro do R

Os números que aparecem na tela do console do R são apenas representações simplificadas do número real armazenado na memória. Por exemplo,

```
x <- runif(10)
x
 [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673 0.0455565 0.5281055
 [8] 0.8924190 0.5514350 0.4566147
```

O objeto `x` contém números como 0.2875775, 0.7883051, etc, que possuem 7 casas decimais, que é o padrão do R. O número de casas decimais é controlado pelo argumento `digits` da função `options()`. Para visualizar essa opção, use

```
getOption("digits")
[1] 7
```

Note que esse valor de 7 é o número de **dígitos significativos**, e pode variar conforme a sequência de números. Por exemplo,

```
y <- runif(10)
y
 [1] 0.069360916 0.817775199 0.942621732 0.269381876 0.169348123
 [6] 0.033895622 0.178785004 0.641665366 0.022877743 0.008324827
```

possui valores com 9 casas decimais. Isto é apenas a representação do número que aparece na tela. Internamente, cada número é armazenado com uma precisão de 64 bits. Como consequência, cada número possui uma acurácia de até 16 dígitos significativos. Isso pode introduzir algum tipo de erro, por exemplo:

```
sqrt(2)^2 - 2
[1] 4.440892e-16
print(sqrt(2)^2, digits = 22)
[1] 2.000000000000000444089
```

não é exatamente zero, pois a raíz quadrada de 2 não pode ser armazenada com toda precisão com "apenas" 16 dígitos significativos. Esse tipo de erro é chamado de **erro de ponto flutuante**, e as operações nessas condições são chamadas de **aritmética de ponto flutuante**. Para mais informações sobre esse assunto veja What Every Computer Scientist Should Know About Floating-Point Arithmetic e Why doesn't R think these numbers are equal?.

No R os números podem ser representados com até 22 casas decimais. Você pode ver o número com toda sua precisão usando a função `print()` e especificando o número de casas decimais com o argumento `digits` (de 1 a 22)

```
print(x, digits = 1)
 [1] 0.29 0.79 0.41 0.88 0.94 0.05 0.53 0.89 0.55 0.46
```

```
print(x, digits = 7) # padrão
 [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673 0.0455565 0.5281055
 [8] 0.8924190 0.5514350 0.4566147
print(x, digits = 22)
 [1] 0.2875775201246142387390 1 0.7883051354438066482544
 [3] 0.4089769218116998672485 4 0.8830174040049314498901 4
 [5] 0.9404672842938452959061 7 0.0455564993899315595626 8
 [7] 0.5281054880470037460327 1 0.8924190443940460681915 3
 [9] 0.5514350144658237695694 0 0.4566147353034466505050 7
```

Também é possível alterar a representação na tela para o formato científico, usando a função `format()`

```
format(x, scientific = TRUE)
 [1] "2.875775e-01" "7.883051e-01" "4.089769e-01" "8.830174e-01"
 [5] "9.404673e-01" "4.555650e-02" "5.281055e-01" "8.924190e-01"
 [9] "5.514350e-01" "4.566147e-01"
```

Nessa representação, o valor 2.875775e-01 = $2.875775 \times 10^{-01} = 0.2875775$.

2.5.1.2 Sequências de números

Usando a função `seq()`

```
seq(1, 10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Ou `1:10` gera o mesmo resultado. Para a sequência variar em 2

```
seq(from = 1, to = 10, by = 2)
[1] 1 3 5 7 9
```

Para obter 15 valores entre 1 e 10

```
seq(from = 1, to = 10, length.out = 15)
 [1]  1.000000  1.642857  2.285714  2.928571  3.571429  4.214286  4.857143
 [8]  5.500000  6.142857  6.785714  7.428571  8.071429  8.714286  9.357143
[15] 10.000000
```

Usando a função `rep()`

```
rep(1, 10)
 [1] 1 1 1 1 1 1 1 1 1 1
```

Para gerar um sequência várias vezes

```
rep(c(1, 2, 3), times = 5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Para repetir um número da sequência várias vezes

```
rep(c(1, 2, 3), each = 5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

2.5.1.3 Operações matemáticas em vetores numéricos

Operações podem ser feitas entre um vetor e um número:

```
num * 2
[1] 20 10  4  8 16 18
```

E também entre vetores de mesmo comprimento ou com comprimentos múltiplos:

```
num * num
[1] 100  25   4  16  64  81
num + c(2, 4, 1)
[1] 12  9  3  6 12 10
```

2.5.1.4 A Regra da Reciclagem

Original		Expandido		Resposta
num	c(2,4,1)	num	c(2,4,1)	num + c(2,4,1)
10	2	10	2	12
5	4	5	4	9
2	1	2	1	3
4		4	2	6
8		8	4	12
9		9	1	10

Agora tente:

```
num + c(2, 4, 1, 3)
```

2.5.2 Outros tipos de vetores

Vetores também podem ter outros tipos:

- Vetor de caracteres:

```
caracter <- c("brava", "joaquina", "armação")
caracter
[1] "brava"    "joaquina" "armação"
typeof(caracter)
[1] "character"
class(caracter)
[1] "character"
```

- Vetor lógico:

```
logico <- caracter == "armação"
logico
[1] FALSE FALSE  TRUE
typeof(logico)
[1] "logical"
class(logico)
[1] "logical"
```

ou

```
logico <- num > 4
logico
[1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE
```

No exemplo anterior, a condição `num > 4` é uma **expressão condicional**, e o símbolo > um **operador lógico**. Os operadores lógicos utilizados no R são:

Operador	Sintaxe	Teste
<	`a < b`	a é menor que b?
<=	`a <= b`	a é menor ou igual a b?
>	`a > b`	a é maior que b
>=	`a >= b`	a é maior ou igual a b?
==	`a == b`	a é igual a b?
!=	`a != b`	a é diferente de b?
%in%	`a %in% c(a, b)`	a está contido no vetor `c(a, b)`?

2.5.3 Misturando classes de objetos

Algumas vezes isso acontece por acidente, mas também pode acontecer de propósito.

O que acontece aqui?

```
w <- c(5L, "a")
x <- c(1.7, "a")
y <- c(TRUE, 2)
z <- c("a", T)
```

Lembre-se da regra:

Um vetor só pode conter elementos do mesmo tipo

Quando objetos de diferentes tipos são misturados, ocorre a **coerção**, para que cada elemento possua a mesma classe.

Nos exemplos acima, nós vemos o efeito da **coerção implícita**, quando o R tenta representar todos os objetos de uma única forma.

Nós podemos forçar um objeto a mudar de classe, através da **coerção explícita**, realizada pelas funções `as.*`:

```
x <- 0:6
typeof(x)
[1] "integer"
class(x)
[1] "integer"
as.numeric(x)
[1] 0 1 2 3 4 5 6
as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
as.factor(x)
[1] 0 1 2 3 4 5 6
Levels: 0 1 2 3 4 5 6
```

De `?logical`:

```
Logical vectors are coerced to integer vectors in contexts where a
numerical value is required, with 'TRUE' being mapped to '1L',
'FALSE' to '0L' and 'NA' to 'NA_integer_'.
```

```
(x <- c(FALSE, TRUE))
[1] FALSE  TRUE
class(x)
[1] "logical"
```

```
as.numeric(x)
[1] 0 1
```

Algumas vezes não é possível fazer a coerção, então:

```
x <- c("a", "b", "c")
as.numeric(x)
Warning: NAs introduced by coercion
[1] NA NA NA
as.logical(x)
[1] NA NA NA
```

2.5.4 Valores perdidos e especiais

Valores perdidos devem ser definidos como `NA` (*not available*):

```
perd <- c(3, 5, NA, 2)
perd
[1]  3  5 NA  2
class(perd)
[1] "numeric"
```

Podemos testar a presença de `NA`s com a função `is.na()`:

```
is.na(perd)
[1] FALSE FALSE  TRUE FALSE
```

Ou:

```
any(is.na(perd))
[1] TRUE
```

Outros valores especiais são:

- `NaN` (*not a number*) - exemplo: `0/0`
- `-Inf` e `Inf` - exemplo: `1/0`

A função `is.na()` também testa a presença de `NaN`s:

```
perd <- c(-1,0,1)/0
perd
[1] -Inf  NaN  Inf
is.na(perd)
[1] FALSE  TRUE FALSE
```

A função `is.infinite()` testa se há valores infinitos

```
is.infinite(perd)
[1]  TRUE FALSE  TRUE
```

Exercícios

1. Crie um objeto com os valores 54, 0, 17, 94, 12.5, 2, 0.9, 15.
   a. Some o objeto acima com os valores 5, 6, e depois com os valores 5, 6, 7.
2. Construa um único objeto com as letras: `A`, `B`, e `C`, repetidas cada uma 15, 12, e 8 vezes, respectivamente.
   a. Mostre na tela, em forma de verdadeiro ou falso, onde estão as letras `B` nesse objeto.

   b. Veja a página de ajuda da função `sum()` e descubra como fazer para contar o número de letras `B` neste vetor (usando `sum()`).
3. Crie um objeto com 100 valores aleatórios de uma distribuição uniforme $U(0,1)$. Conte quantas vezes aparecem valores maiores ou iguais a 0,5.
4. Calcule as 50 primeiras potências de 2, ou seja, $2, 2^2, 2^3, \ldots, 2^{50}$.
   a. Calcule o quadrado dos números inteiros de 1 a 50, ou seja, $1^2, 2^2, 3^2, \ldots, 50^2$.
   b. Quais pares são iguais, ou seja, quais números inteiros dos dois exercícios anteriores satisfazem a condição $2^n = n^2$?
   c. Quantos pares existem?
5. Calcule o seno, coseno e a tangente para os números variando de 0 a 2π, com distância de 0.1 entre eles. (Use as funções `sin()`, `cos()`, `tan()`).
   a. Calcule a tangente usando a relação $\tan(x) = \sin(x)/\cos(x)$.
   b. Calcule as diferenças das tangentes calculadas pela função do R e pela razão acima.
   c. Quais valores são exatamente iguais?
   d. Qual a diferença máxima (em módulo) entre eles? Qual é a causa dessa diferença?

2.6 Outras classes

Como mencionado na seção anterior, o R possui 6 tipos básicos de estrutura de dados, mas diversas classes podem ser construídas a partir destes tipos básicos. Abaixo, veremos algumas das mais importantes.

2.6.1 Fator

Os fatores são parecidos com caracteres no R, mas são armazenados e tratados de maneira diferente.

Características:

- Coleção de categorias ou **níveis** (*levels*)
- Estrutura unidimensional

Utilizando as funções `factor()` e `c()`:

```
fator <- factor(c("alta","baixa","baixa","media",
                  "alta","media","baixa","media","media"))
fator
[1] alta  baixa baixa media alta  media baixa media media
Levels: alta baixa media
class(fator)
[1] "factor"
typeof(fator)
[1] "integer"
```

Note que o objeto é da classe `factor`, mas seu tipo básico é `integer`! Isso significa que cada categoria única é identificada internamente por um número, e isso faz com que os fatores possuam uma ordenação, de acordo com as categorias únicas. Por isso existe a identificação dos `Levels` (níveis) de um fator.

Veja o que acontece quando "remover a classe" desse objeto

```
unclass(fator)
[1] 1 2 2 3 1 3 2 3 3
attr(,"levels")
[1] "alta"  "baixa" "media"
```

Fatores podem ser convertidos para caracteres, e **também** para números inteiros

```
as.character(fator)
[1] "alta"  "baixa" "baixa" "media" "alta"  "media" "baixa" "media" "media"
as.integer(fator)
[1] 1 2 2 3 1 3 2 3 3
```

Caso haja uma hierarquia, os níveis dos fatores podem ser ordenados explicitamente através do argumento `levels`:

```
fator <- factor(c("alta","baixa","baixa","media",
                  "alta","media","baixa","media","media"),
                levels = c("alta","media","baixa"))
fator
[1] alta  baixa baixa media alta  media baixa media media
Levels: alta media baixa
typeof(fator)
[1] "integer"
class(fator)
[1] "factor"
```

Além disso, os níveis dos fatores podem também ser explicitamente ordenados

```
fator <- factor(c("alta","baixa","baixa","media",
                  "alta","media","baixa","media","media"),
                levels = c("baixa", "media", "alta"),
                ordered = TRUE)
fator
[1] alta  baixa baixa media alta  media baixa media media
Levels: baixa < media < alta
typeof(fator)
[1] "integer"
class(fator)
[1] "ordered" "factor"
```

(Veja que um objeto pode ter mais de uma classe). Isso geralmente só será útil em casos especificos.

As seguintes funções são úteis para verificar os níveis e o número de níveis de um fator:

```
levels(fator)
[1] "baixa" "media" "alta"
nlevels(fator)
[1] 3
```

2.6.2 Matriz

Matrizes são vetores que podem ser dispostos em duas dimensões.

Características:

- Podem conter apenas um tipo de informação (números, caracteres)
- Estrutura bidimensional

Utilizando a função `matrix()`:

```
matriz <- matrix(1:12, nrow = 3, ncol = 4)
matriz
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
class(matriz)
[1] "matrix"
typeof(matriz)
[1] "integer"
```

Alterando a ordem de preenchimento da matriz (por linhas):

```
matriz <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
matriz
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Para verificar a dimensão da matriz:

```
dim(matriz)
[1] 3 4
```

Adicionando colunas com `cbind()`

```
cbind(matriz, rep(99, 3))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4   99
[2,]    5    6    7    8   99
[3,]    9   10   11   12   99
```

Adicionando linhas com `rbind()`

```
rbind(matriz, rep(99, 4))
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   99   99   99   99
```

Matrizes também podem ser criadas a partir de vetores adicionando um **atributo** de dimensão

```
m <- 1:10
m
 [1]  1  2  3  4  5  6  7  8  9 10
class(m)
[1] "integer"
dim(m)
NULL
dim(m) <- c(2, 5)
m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
class(m)
[1] "matrix"
typeof(m)
[1] "integer"
```

2.6.2.1 Operações matemáticas em matrizes

Matriz multiplicada por um escalar

```
matriz * 2
     [,1] [,2] [,3] [,4]
[1,]    2    4    6    8
[2,]   10   12   14   16
[3,]   18   20   22   24
```

Multiplicação de matrizes (observe as dimensões!)

```
matriz2 <- matrix(1, nrow = 4, ncol = 3)
matriz %*% matriz2
     [,1] [,2] [,3]
[1,]   10   10   10
[2,]   26   26   26
[3,]   42   42   42
```

2.6.3 Array

Um array é a forma mais geral de uma matriz, pois pode ter n dimensões.

Características:

- Estrutura n-dimensional
- Assim como as matrizes, podem conter apenas um tipo de informação (números, caracteres)

Para criar um array, usamos a função `array()`, passando como primeiro argumento um vetor atômico, e especificamos a dimensão com o argumento `dim`. Por exemplo, para criar um objeto com 3 dimensões $2 \times 2 \times 3$, fazemos

```
ar <- array(1:12, dim = c(2, 2, 3))
ar
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

Similarmente, um array de 2 dimensões $3 \times 2 \times 2$ é obtido com

```
ar <- array(1:12, dim = c(3, 2, 2))
ar
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

2.6.4 Lista

Como já vimos, uma lista não é uma "classe" propriamente dita, mas sim um tipo de estrutura de dados básico, ao lado dos vetores atômicos. E, assim como os vetores atômicos, listas são estruturas unidimensionais. A grande diferença é que listas agrupam objetos de diferentes tipos, inclusive outras listas.

Características:

- Pode combinar uma coleção de objetos de diferentes tipos ou classes (é um tipo básico de vetor, assim como os vetores atômicos)
- Estrutura "unidimensional": apenas o número de elementos na lista é contado

Ppor exemplo, podemos criar uma lista com uma sequência de números, um caracter e outra lista

```
lista <- list(1:30, "R", list(TRUE, FALSE))
lista
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
[24] 24 25 26 27 28 29 30

[[2]]
[1] "R"

[[3]]
[[3]][[1]]
[1] TRUE

[[3]][[2]]
[1] FALSE
class(lista)
[1] "list"
typeof(lista)
[1] "list"
```

Para melhor visualizar a estrutura dessa lista (ou de qualquer outro objeto) poddemos usar a função `str()`

```
str(lista)
List of 3
 $ : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
 $ : chr "R"
 $ :List of 2
  ..$ : logi TRUE
  ..$ : logi FALSE
```

Note que de fato é uma estrutura unidimensional

```
dim(lista)
NULL
```

```
length(lista)
[1] 3
```

Listas podem armazenar objetos de diferentes classes e dimensões, por exemplo, usando objetos criados anteriormente

```
lista <- list(fator, matriz)
lista
[[1]]
[1] alta  baixa baixa media alta  media baixa media media
Levels: baixa < media < alta

[[2]]
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
length(lista)
[1] 2
```

2.6.5 Data frame

Data frame é a versão bidimensional de uma lista. Data frames **são** listas, mas onde cada componente dever ter obrigatoriamente o mesmo comprimento. Cada vetor da lista vira uma coluna em um data frame, permitindo então que as "colunas" sejam de diferentes tipos.

Os data frames são as estruturas mais comuns para se trabalhar com dados no R.

Características:

- Uma lista de vetores e/ou fatores, de **mesmo comprimento**
- Pode conter diferentes tipos de dados (numérico, fator, ...)
- Estrutura bidimensional

Utilizando a função `data.frame()`:

```
da <- data.frame(nome = c("João", "José", "Maria"),
                 sexo = c("M", "M", "F"),
                 idade = c(32, 34, 30))
da
   nome sexo idade
1  João    M    32
2  José    M    34
3 Maria    F    30
class(da)
[1] "data.frame"
typeof(da)
[1] "list"
dim(da)
[1] 3 3
```

Veja os detalhes com `str()`

```
str(da)
'data.frame':   3 obs. of  3 variables:
 $ nome : Factor w/ 3 levels "João","José",..: 1 2 3
 $ sexo : Factor w/ 2 levels "F","M": 2 2 1
 $ idade: num  32 34 30
```

Note que a função `data.frame()` converte caracteres para fator automaticamente. Para que isso não aconteça, use o argumento `stringsAsFactors = FALSE`

```
da <- data.frame(nome = c("João", "José", "Maria"),
                 sexo = c("M", "M", "F"),
                 idade = c(32, 34, 30),
                 stringsAsFactors = FALSE)
da
   nome sexo idade
1  João    M    32
2  José    M    34
3 Maria    F    30
str(da)
'data.frame':   3 obs. of  3 variables:
 $ nome : chr  "João" "José" "Maria"
 $ sexo : chr  "M" "M" "F"
 $ idade: num  32 34 30
```

Data frames podem ser formados com objetos criados anteriormente, desde que tenham o mesmo comprimento:

```
length(num)
[1] 6
length(fator)
[1] 9
db <- data.frame(numerico = c(num, NA, NA, NA),
                 fator = fator)
db
  numerico fator
1       10  alta
2        5 baixa
3        2 baixa
4        4 media
5        8  alta
6        9 media
7       NA baixa
8       NA media
9       NA media
str(db)
'data.frame':   9 obs. of  2 variables:
 $ numerico: num  10 5 2 4 8 9 NA NA NA
 $ fator   : Ord.factor w/ 3 levels "baixa"<"media"<..: 3 1 1 2 3 2 1 2 2
```

Algumas vezes pode ser necessário converter um data frame para uma matriz. Existem duas opções:

```
as.matrix(db)
      numerico fator
 [1,] "10"     "alta"
 [2,] " 5"     "baixa"
 [3,] " 2"     "baixa"
 [4,] " 4"     "media"
 [5,] " 8"     "alta"
 [6,] " 9"     "media"
 [7,] NA       "baixa"
 [8,] NA       "media"
 [9,] NA       "media"
data.matrix(db)
```

```
      numerico fator
 [1,]       10     3
 [2,]        5     1
 [3,]        2     1
 [4,]        4     2
 [5,]        8     3
 [6,]        9     2
 [7,]       NA     1
 [8,]       NA     2
 [9,]       NA     2
```

Geralmente é o resultado de `data.matrix()` o que você está procurando.

Lembre que os níveis de um fator são armazenados internamente como números: 1° nível = 1, 2° nível = 2, . . .

```
fator
[1] alta  baixa baixa media alta  media baixa media media
Levels: baixa < media < alta
str(fator)
 Ord.factor w/ 3 levels "baixa"<"media"<..: 3 1 1 2 3 2 1 2 2
as.numeric(fator)
[1] 3 1 1 2 3 2 1 2 2
```

2.7 Atributos de objetos

Um atributo é um pedaço de informação que pode ser "anexado" à qualquer objeto, e não irá interferir nos valores daquele objeto. Os atributos podem ser vistos como "metadados", alguma descrição associada à um objeto. Os principais atributos são:

- `names`
- `dimnames`
- `dim`
- `class`

Alguns atributos também podem ser visualizados de uma só vez através da função `attributes()`.

Por exemplo, considere o seguinte vetor

```
x <- 1:6
attributes(x)
NULL
```

Mostra que o objeto `x` não possui nenhum atributo. Mas podemos definir nomes, por exemplo, para cada componente desse vetor

```
names(x)
NULL
names(x) <- c("um", "dois", "tres", "quatro", "cinco", "seis")
names(x)
[1] "um"    "dois"  "tres"  "quatro" "cinco"  "seis"
attributes(x)
$names
[1] "um"    "dois"  "tres"  "quatro" "cinco"  "seis"
```

Nesse caso específico, o R irá mostrar os nomes acima dos componentes, mas isso não altera como as oprações serão realizadas

```
x
    um   dois   tres quatro  cinco   seis
     1      2      3      4      5      6
x + 2
    um   dois   tres quatro  cinco   seis
     3      4      5      6      7      8
```

Os nomes então podem ser definidos através da função *auxiliar* `names()`, sendo assim, também podemos remover esse atributo declarando ele como nulo

```
names(x) <- NULL
attributes(x)
NULL
x
[1] 1 2 3 4 5 6
```

Outros atributos também podem ser definidos de maneira similar. Veja os exemplos abaixo:

```
length(x)
[1] 6
## Altera o comprimento (preenche com NA)
length(x) <- 10
x
 [1]  1  2  3  4  5  6 NA NA NA NA
## Altera a dimensão
length(x) <- 6
dim(x)
NULL
dim(x) <- c(3, 2)
x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
attributes(x)
$dim
[1] 3 2
## Remove dimensão
dim(x) <- NULL
x
[1] 1 2 3 4 5 6
```

Assim como vimos em data frames, listas também podem ter nomes

```
x <- list(Curitiba = 1, Paraná = 2, Brasil = 3)
x
$Curitiba
[1] 1

$Paraná
[1] 2

$Brasil
[1] 3
names(x)
[1] "Curitiba" "Paraná"   "Brasil"
```

Podemos também associar nomes às *linhas* e *colunas* de uma matriz:

```
matriz
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
attributes(matriz)
$dim
[1] 3 4
rownames(matriz) <- c("A","B","C")
colnames(matriz) <- c("T1","T2","T3","T4")
matriz
  T1 T2 T3 T4
A  1  2  3  4
B  5  6  7  8
C  9 10 11 12
attributes(matriz)
$dim
[1] 3 4

$dimnames
$dimnames[[1]]
[1] "A" "B" "C"

$dimnames[[2]]
[1] "T1" "T2" "T3" "T4"
```

Para data frames existe uma função especial para os nomes de linhas, `row.names()`. Data frames também não possuem nomes de colunas, apenas nomes, já que é um caso particular de lista. Então para verificar/alterar nomes de colunas de um data frame também use `names()`.

```
da
   nome sexo idade
1  João    M    32
2  José    M    34
3 Maria    F    30
attributes(da)
$names
[1] "nome"  "sexo"  "idade"

$class
[1] "data.frame"

$row.names
[1] 1 2 3
names(da)
[1] "nome"  "sexo"  "idade"
row.names(da)
[1] "1" "2" "3"
```

Um resumo das funções para alterar/acessar nomes de linhas e colunas em matrizes e data frames.

Classe	Nomes de colunas	Nomes de linhas
`data.frame`	`names()`	`row.names()`
`matrix`	`colnames()`	`rownames()`

Exercícios

1. Crie um objeto para armazenar a seguinte matriz

$$\begin{bmatrix} 2 & 8 & 4 \\ 0 & 4 & 1 \\ 9 & 7 & 5 \end{bmatrix}$$

2. Atribua nomes para as linhas e colunas dessa matriz.
3. Crie uma lista (**não nomeada**) com dois componentes: (1) um vetor com as letras `A`, `B`, e `C`, repetidas 2, 5, e 4 vezes respectivamente; e (2) a matriz do exemplo anterior.
4. Atribua nomes para estes dois componentes da lista.
5. Inclua mais um componente nesta lista, com o nome de `fator`, e que seja um vetor da classe `factor`, idêntico ao objeto `caracter` criado acima (que possui apenas os nomes `brava`, `joaquina`, `armação`).
6. Crie um data frame para armazenar duas variáveis: local (`A`, `B`, `C`, `D`), e contagem (42, 34, 59 e 18).
7. Crie um data frame com as seguintes colunas:

- Nome
- Sobrenome
- Se possui animal de estimação
- Caso possua, dizer o número de animais (caso contrário, colocar 0)

Para criar o data frame, a primeira linha deve ser preenchida com as suas próprias informação (use a função `data.frame()`). Depois, pergunte essas mesmas informações para dois colegas ao seu lado, e adicione as informações deles à esse data frame (use `rbind()`). Acresente mais uma coluna com o nome do time de futebol de cada um.

Referências

Para mais detalhes e exemplos dos assuntos abordados aqui, veja Grolemund (2014). Uma abordagem mais avançada e detalhada sobre programação orientada a objetos no R pode ser consultada em Wickham (2015).

Grolemund, Garrett. 2014. *Hands-On Programming with R - Write Your Own Functions and Simulations*. O'Reily Media. http://shop.oreilly.com/product/0636920028574.do.

Wickham, Hadley. 2015. *Advanced R*. CRC Press.

Capítulo 3

Indexação e seleção condicional

3.1 Indexação

Existem 6 maneiras diferentes de indexar valores no R. Podemos indexar usando:

- Inteiros positivos
- Inteiros negativos
- Zero
- Espaço em branco
- Nomes
- Valores lógicos

Existem três tipos de operadores que podem ser usados para indexar (e selecionar) **sub-conjuntos** (*subsets*) de objetos no R:

- O operador `[ ]` sempre retorna um objeto da mesma classe que o original. Pode ser usado para selecionar múltiplos elementos de um objeto
- O operador`[[ ]]` é usado para extrair elementos de uma **lista** ou **data frame**. Pode ser usado para extrair um único elemento, e a classe do objeto retornado não precisa necessariamente ser uma lista ou data frame.
- O operador `$` é usado para extrair elementos **nomeados** de uma lista ou data frame. É similar ao operador `[[ ]]`.

3.1.1 Indexação de vetores

Observe o seguinte vetor de contagens

```
cont <- c(8, 4, NA, 9, 6, 1, 7, 9)
cont
[1]  8  4 NA  9  6  1  7  9
```

Para acessar o valor que está na posição 4, faça:

```
cont[4]
[1] 9
```

Os colchetes `[ ]` são utilizados para **extração** (seleção de um intervalo de dados) ou **substituição** de elementos. O valor dentro dos colchetes é chamado de **índice**.

Para acessar os valores nas posições 1, 4 e 8 é necessário o uso da função `c()`:

```
cont[c(1, 4, 8)]
[1] 8 9 9
```

Ou:

```
ind <- c(1, 4, 8)
cont[ind]
[1] 8 9 9
```

Note que os índices no R começam em 1, e não em 0, como algumas outras linguagens.

Para selecionar todos os valores, **excluindo** aquele da posição 4, usamos um índice negativo

```
cont[-4]
[1]  8  4 NA  6  1  7  9
```

Da mesma forma se quiséssemos todos os valores, menos aqueles das posições 1, 4 e 8:

```
cont[-c(1, 4, 8)]
[1]  4 NA  6  1  7
```

Também é possível selecionar uma sequência de elementos (com qualquer uma das funções de gerar sequências que já vimos antes):

```
## Seleciona os elementos de 1 a 5
cont[1:5]
[1]  8  4 NA  9  6
## Seleciona os elementos nas posições ímpar
cont[seq(1, 8, by = 2)]
[1]  8 NA  6  7
```

Mas note que para selecionar todos menos aqueles de uma sequência, precisamos colocá-la entre parênteses

```
cont[-1:5]
Error in cont[-1:5]: only 0's may be mixed with negative subscripts
cont[-(1:5)]
[1] 1 7 9
```

Para selecionar todos os elementos que sejam `NA`, ou todos menos os `NA`s, precisamos usar a função `is.na()`

```
## Para selecionar os NAs
cont[is.na(cont)]
[1] NA
## Para selecionar todos menos os NAs
cont[!is.na(cont)]
[1] 8 4 9 6 1 7 9
```

Para substituir os `NA`s por algum valor (*e.g.* 0):

```
cont[is.na(cont)] <- 0
cont
[1] 8 4 0 9 6 1 7 9
```

E para especificar `NA` para algum valor

```
is.na(cont) <- 3
cont
[1]  8  4 NA  9  6  1  7  9
## Mais seguro do que
# cont[3] <- NA
```

Note que se utilizarmos o operador de atribuição `<-` em conjunto com uma indexação, estaremos **substituindo** os valores selecionados pelos valores do lado direito do operador de atribuição.

Podemos também utilizar mais duas formas de indexação no R: espaços em branco e zero:

```
cont[0]
numeric(0)
cont[]
[1]  8  4 NA  9  6  1  7  9
```

Note que o índice zero não existe no R, mas ao utilizá-lo é retornado um vetor "vazio", ou um vetor de comprimento zero. Essa forma de indexação é raramente utilizada no R.

Ao deixar um espaço em branco, estamos simplesmente informando que queremos todos os valores daquele vetor. Essa forma de indexação será particularmente útil para objetos que possuem duas ou mais dimensões, como matrizes e data frames.

Exercícios

1. Crie um vetor com os valores: `88, 5, 12, 13`
2. Selecione o elemento na posição 3
3. Selecione o valor 88
4. Selecione os valores 13 e 5
5. Selecione todos os valores, menos o 88 e o 13
6. Insira o valor 168 **entre** os valores 12 e 13, criando um novo objeto

3.1.1.1 Vetores nomeados

Quando vetores possuem seus componentes **nomeados**, a indexação pode ser realizada pelos nomes destes componentes.

```
## Atribui as letras "a", "b", ..., "h" para os componentes de cont
names(cont) <- letters[1:length(cont)]
## Acessando o quarto elemento
cont["d"]
d
9
## Acessando o sexto e o primeiro elemento
cont[c("f", "a")]
f a
1 8
```

Dica: veja `?letters`

3.1.2 Indexação de matrizes

Considere a seguinte matriz

```
mat <- matrix(1:9, nrow = 3)
mat
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Acesse o valor que está na linha 2 da coluna 3:

```
mat[2, 3]
[1] 8
```

A indexação de matrizes sempre segue a ordem `[linha, coluna]`

Para acessar todas as linhas da coluna 1, deixamos o espaço em branco (que também é uma forma de seleção, como vimos) na posição das linhas e especificamos a coluna desejada

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

Para acessar todas as colunas da linha 1, usamos a mesma lógica

```
mat[1, ]
[1] 1 4 7
```

Note que o resultado destas extrações trazem os valores internos das matrizes, mas com a dimensão reduzida (nestes casos para uma dimensão). Se o objetivo for, por exemplo, extrair uma parte da matriz, mas mantendo as duas dimensões, então precisamos usar o argumento `drop` da "função" `[` (sim, também é uma função; veja `"?["`). Veja as diferenças:

```
mat[3, 2]
[1] 6
mat[3, 2, drop = FALSE]
     [,1]
[1,]    6
mat[1, ]
[1] 1 4 7
mat[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    4    7
```

Para acessar as linhas 1 e 3 das colunas 2 e 3:

```
mat[c(1, 3), c(2, 3)]
     [,1] [,2]
[1,]    4    7
[2,]    6    9
```

E note que aqui a dimensão já é 2 pois naturalmente o resultado preisa ser representado em duas dimensões.

3.1.2.1 Matrizes nomeadas

Se as matrizes tiverem linhas e/ou colunas nomeados, a indexação também pode ser feita com os nomes.

```
##---------------------------------------------------------------------
## Nomes das colunas
colnames(mat) <- LETTERS[1:3]
## Todas as linhas da coluna B
mat[, "B"]
[1] 4 5 6
## Elemento da linha 1 e coluna C
mat[1, "C"]
C
7
##---------------------------------------------------------------------
## Nomes das linhas
rownames(mat) <- LETTERS[24:26]
## Todas as colunas da linha X
mat["X", ]
```

```
A B C
1 4 7
## Elemento da linha Y e coluna A
mat["Y", "A"]
[1] 2
```

3.1.3 Indexação de listas

Considere a seguinte lista:

```
lis <- list(c(3, 8, 7, 4), mat, 5:0)
lis
[[1]]
[1] 3 8 7 4

[[2]]
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9

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

Podemos acessar um componente da lista da maneira usual

```
lis[1]
[[1]]
[1] 3 8 7 4
```

Mas note que esse objeto continua sendo uma lista

```
class(lis[1])
[1] "list"
```

Geralmente o que queremos é acessar os elementos que estão **contidos** nos componentes da lista, e para isso precisamos usar `[[` no lugar de `[`:

```
lis[[1]]
[1] 3 8 7 4
class(lis[[1]])
[1] "numeric"
```

Isso é importante, por exemplo, se quisessemos aplicar uma função qualquer a um componente da lista. No primeiro caso isso não é possível pois o conteúdo continua dentro de uma lista, enquanto que no segundo caso os valores retornados são os próprios números:

```
mean(lis[1])
Warning in mean.default(lis[1]): argument is not numeric or logical:
returning NA
[1] NA
mean(lis[[1]])
[1] 5.5
```

Para acessar o segundo **componente** da lista:

```
lis[[2]]
  A B C
X 1 4 7
```

```
Y 2 5 8
Z 3 6 9
```

Para acessar o terceiro valor do primeiro componente:

```
lis[[1]][3]
[1] 7
```

Note que o segundo elemento da lista é uma matriz, portanto a indexação da matriz *dentro da lista* também segue o mesmo padrão

```
lis[[2]][2, 3]
[1] 8
```

Se a lista tiver componentes **nomeados**, eles podem ser acessados com o operador $:

```
lis <- list(vetor1 = c(3, 8, 7, 4), mat = mat, vetor2 = 5:0)
## Ou
## names(lis) <- c("vetor1", "mat", "vetor2")
lis
$vetor1
[1] 3 8 7 4

$mat
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9

$vetor2
[1] 5 4 3 2 1 0
```

```
## Acesando o segundo componente
lis$mat
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9
## Linha 2 e coluna 3
lis$mat[2, 3]
[1] 8
## Terceiro elemento do primeiro componente
lis$vetor1[3]
[1] 7
```

Ou então

```
lis[["mat"]]
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9
lis[["vetor1"]][3]
[1] 7
```

O símbolo $ é utilizado para acessar componentes **nomeados** de listas ou data frames.

3.1.4 Indexação de data frames

Considere o seguinte data frame

```
da <- data.frame(A = 4:1, B = c(2, NA, 5, 8))
da
  A  B
1 4  2
2 3 NA
3 2  5
4 1  8
```

Para acessar o segundo elemento da primeira coluna (segue a mesma lógica da indexação de matrizes pois também possui duas dimensões):

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

Acesse todas as linhas da coluna B:

```
da[, 2]
[1]  2 NA  5  8
```

Ou usando o nome da coluna:

```
da[,"B"]
[1]  2 NA  5  8
```

Todas as colunas da linha 1:

```
da[1, ]
  A B
1 4 2
```

Ou usando o "nome" da linha:

```
da["1", ]
  A B
1 4 2
```

Note que o argumento `drop=` também é válido se quiser manter a dimensão do objeto

```
da[, "B"]
[1]  2 NA  5  8
da[, "B", drop = FALSE]
   B
1  2
2 NA
3  5
4  8
```

Como o data frame é um caso particular de uma lista (onde todos os componentes tem o mesmo comprimento), as colunas de um data frame também podem ser acessadas com `$`:

```
da$A
[1] 4 3 2 1
```

Para acessar o terceiro elemento da coluna B:

```
da$B[3]
[1] 5
```

Para acessar os elementos nas posições 2 e 4 da coluna B:

```
da$B[c(2, 4)]
[1] NA  8
```

Assim como nas listas, podemos indexar um data frame usando apenas um índice, que nesse caso retorna a coluna (componente) do data frame:

```
da[1]
  A
1 4
2 3
3 2
4 1
class(da[1])
[1] "data.frame"
```

Note que dessa forma a classe é mantida. Também podemos indexar os data frames usando `[[` da mesma forma como em listas

```
da[[1]]
[1] 4 3 2 1
da[["A"]]
[1] 4 3 2 1
da[["A"]][2:3]
[1] 3 2
```

Para lidar com NAs em data frames, podemos também usar a função `is.na()`

```
da[is.na(da), ] # nao retorna o resultado esperado
    A  B
NA NA NA
## Deve ser feito por coluna
da[is.na(da$A), ]
[1] A B
<0 rows> (or 0-length row.names)
da[is.na(da$B), ]
  A  B
2 3 NA
## De maneira geral
is.na(da)
         A     B
[1,] FALSE FALSE
[2,] FALSE  TRUE
[3,] FALSE FALSE
[4,] FALSE FALSE
is.na(da$A)
[1] FALSE FALSE FALSE FALSE
is.na(da$B)
[1] FALSE  TRUE FALSE FALSE
```

Para remover as linhas que possuem NA, note que será necessário remover todas as colunas daquela linha, pois data frames não podem ter colunas de comprimentos diferentes

```
da[!is.na(da$B), ]
  A B
1 4 2
3 2 5
4 1 8
```

A função `complete.cases()` facilita esse processo. Veja o resultado

```
complete.cases(da)
[1]  TRUE FALSE  TRUE  TRUE
da[complete.cases(da), ]
  A B
1 4 2
3 2 5
4 1 8
```

3.1.4.1 A função `with()`

Para evitar fazer muitas indexações de um mesmo data frame, por exemplo, podemos utilizar a função `with()`

```
with(da, A)
[1] 4 3 2 1
```

é o mesmo que

```
da$A
[1] 4 3 2 1
```

Também é útil para acessar elementos específicos dentro de data frames. Por exemplo, o terceiro elemento da coluna B

```
with(da, B[3])
[1] 5
```

E também para aplicar funções nas colunas do data frame

```
with(da, mean(A))
[1] 2.5
```

Exercícios

1. Crie a seguinte matriz
$$\begin{bmatrix} 4 & 16 & 2 \\ 10 & 5 & 11 \\ 9 & 9 & 5 \\ 2 & 0 & NA \end{bmatrix}$$
2. Acesse o elemento na quarta linha e na segunda coluna
3. Acesse todos os elementos, com exceção da segunda coluna e da terceira linha
4. Crie uma lista (nomeada) com 3 componentes: um vetor numérico de comprimento 10, um vetor de caracteres de comprimento 7, e a matriz do exercício anterior
5. Acesse os elementos nas posições de 5 a 3 do primeiro componente da lista
6. Acesse os caracteres de todas as posições, menos na 2 e 6
7. Acesse todas as linhas da coluna 3 da matriz dentro desta lista
8. "Crie" um novo componente nessa lista (usando `$`), contendo 30 valores aleatórios de uma distribuição normal $\mathrm{N}(12, 4)$ (veja `?rnorm`)
9. Crie um data frame que contenha duas colunas: a primeira com as letras de "A" até "J", e outra com o resultado de uma chamada da função `runif(7, 1, 5)`
10. Extraia as duas primeiras linhas desse data frame
11. Extraia as duas últimas linhas desse data frame
12. Qual é o valor que está na linha 6 e coluna 1?
13. Qual é o valor que está na linha 4 da coluna 2?
14. Como você faria para contar quantos valores perdidos (`NA`s) existem nesse data frame?

15. Elimine os NAs deste data frame.
16. Crie uma nova coluna neste data frame, com valores numéricos (quaisquer)
17. Crie mais um componente na lista anterior, que será também uma lista com dois componentes: A com os valores 1:5, e B com as letras de "a" a "f"
18. Acesse o número 4 de A
19. Acesse a letra "c" de B

3.2 Seleção condicional

3.2.1 Seleção condicional em vetores

A **seleção condicional** serve para extrair dados que satisfaçam algum critério, usando **expressões condicionais** e **operadores lógicos**.

Considere o seguinte vetor

```
dados <- c(5, 15, 42, 28, 79, 4, 7, 14)
```

Selecione apenas os valores maiores do que 15:

```
dados[dados > 15]
[1] 42 28 79
```

Selecione os valores maiores que 15 E menores ou iguais a 35:

```
dados[dados > 15 & dados <= 35]
[1] 28
```

Para entender como funciona a seleção condicional, observe apenas o resultado da condição dentro do colchetes:

```
## Usando & (e)
dados > 15 & dados <= 35
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## Usando | (ou)
dados > 15 | dados <= 35
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Os valores selecionados serão aqueles em que a condição for TRUE, no primeiro caso apenas o quarto elemento do vetor dados.

A seleção condicional também é muito útil para selecionar elementos de um vetor, baseado em uma condição de outro vetor.

Considere o seguinte vetor de caracteres

```
cara <- letters[1:length(dados)]
```

Considere que de alguma forma, os objetos dados e cara possuem alguma relação. Sendo assim, podemos selecionar elementos de dados, baseados em alguma condição de cara

```
## Elemento de dados onde cara é igual a "c"
dados[cara == "c"]
[1] 42
```

Se quisermos selecionar mais de um elemento de dados, baseado em uma condição de cara?

```
## Elementos de dados onde cara é igual a "a" e "c"
cara == "a" & cara == "c" # porque não funciona?
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
cara == "a" | cara == "c"
```

```
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
dados[cara == "a" | cara == "c"]
[1]  5 42
```

Uma solução melhor seria se pudessemos usar

```
dados[cara == c("a", "c")]
[1] 5
```

mas nesse caso só temos o primeiro elemento. Um operador muito útil nestes casos é o `%in%`

```
dados[cara %in% c("a", "c")]
[1]  5 42
cara %in% c("a", "c")
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Veja a diferença

```
cara == c("a", "c")
[1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
cara %in% c("a", "c")
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Também é possível fazer a seleção de `cara`, baseado em uma condição em `dados`

```
## Elemento de cara onde dados é igual a 15
cara[dados == 15]
[1] "b"
## Elemento de cara onde dados for maior do que 30
cara[dados > 30]
[1] "c" "e"
## Elemento de cara onde dados for igual a 4 ou 14
cara[dados %in% c(4, 14)]
[1] "f" "h"
```

3.2.2 A função `which()`

Até agora vimos seleções condicionais que nos retornavam o resultado de uma expressão condicional em vetores. No entanto, muitas vezes estamos interessados em saber a **posição** do resultado de uma expressão condicional, ao invés do resultado em si.

A fução `which()` retorna as posições dos elementos que retornarem `TRUE` em uma expressão condicional.

```
## Elementos maiores de 15
dados[dados > 15]
[1] 42 28 79
which(dados > 15)
[1] 3 4 5
## Elementos maiores de 15 e menores ou iguais a 35
dados[dados > 15 & dados <= 35]
[1] 28
which(dados > 15 & dados <= 35)
[1] 4
## Elementos de dados onde cara igual a "c"
dados[cara == "c"]
[1] 42
which(cara == "c")
[1] 3
```

```
## Elementos de dadaos onde cara igual a "a" ou "c"
dados[cara %in% c("a", "c")]
[1]  5 42
which(cara %in% c("a", "c"))
[1] 1 3
```

Exercícios

1. Crie um vetor (x) com os valores 3, 8, 10, 4, 9, 7, 1, 9, 2, 4.
2. Selecione os elementos maiores ou iguais a 5
3. Selecione todos os elementos menos o 4
4. Selecione os elementos maiores que 4 e menores que 8
5. Crie um vetor (a) com as letras de A até J
6. Selecione os elementos de x onde a for igual a "F"
7. Selecione os elementos de x onde a for igual a "B", "D", e "H"
8. Qual a posição do número 10 em x?
9. Quais as posições dos valores maiores ou iguais a 8 e menores ou iguais a 10 em x?
10. Quais as posições das letras "A", "B", "D" em a?

3.2.3 Seleção condicional em data frames

Considere o seguinte data frame

```
dados <- data.frame(ano = c(2001, 2002, 2003, 2004, 2005),
                    captura = c(26, 18, 25, 32, NA),
                    porto = c("SP", "RS", "SC", "SC", "RN"))
```

Extraia deste objeto apenas a linha correspondente ao ano 2004:

```
dados[dados$ano == 2004, ]
   ano captura porto
4 2004      32    SC
```

Mostre as linhas apenas do porto "SC":

```
dados[dados$porto == "SC", ]
   ano captura porto
3 2003      25    SC
4 2004      32    SC
```

Observe as linhas onde a captura seja maior que 20, selecionando apenas a coluna `captura`:

```
dados[dados$captura > 20, "captura"]
[1] 26 25 32 NA
```

Também exclua as linhas com `NA`s (agora com todas as colunas):

```
dados[dados$captura > 20 & !is.na(dados$captura), ]
   ano captura porto
1 2001      26    SP
3 2003      25    SC
4 2004      32    SC
dados[dados$captura > 20 & complete.cases(dados), ]
   ano captura porto
1 2001      26    SP
3 2003      25    SC
4 2004      32    SC
```

A condição pode ser feita com diferentes colunas:

```
dados[dados$captura > 25 & dados$porto == "SP", ]
   ano captura porto
1 2001      26    SP
```

A função `subset()` serve para os mesmos propósitos, e facilita todo o processo de seleção condicional:

```
dados[dados$porto == "SC", ]
   ano captura porto
3 2003      25    SC
4 2004      32    SC
subset(dados, porto == "SC")
   ano captura porto
3 2003      25    SC
4 2004      32    SC
dados[dados$captura > 20, ]
    ano captura porto
1  2001      26    SP
3  2003      25    SC
4  2004      32    SC
NA   NA      NA  <NA>
subset(dados, captura > 20)
   ano captura porto
1 2001      26    SP
3 2003      25    SC
4 2004      32    SC
dados[dados$captura > 20 & !is.na(dados$captura), ]
   ano captura porto
1 2001      26    SP
3 2003      25    SC
4 2004      32    SC
dados[dados$captura > 20, "captura"]
[1] 26 25 32 NA
subset(dados, captura > 20, select = captura)
  captura
1      26
3      25
4      32
```

A grande vantagem é que a função `subset()` já lida com os `NA`s (se isso for o que você precisa).

Exercícios

1. Você contou 42 caranguejos na Joaquina, 34 no Campeche, 59 na Armação, e 18 na Praia Mole. Crie um data frame para armazenar estas informações (número de caranguejos observados e local).
2. Com o data frame criado no exercício anterior, mostre qual a praia onde foram coletadas menos de 30 caranguejos (usando seleção condicional!).
3. Crie uma nova coluna (região) neste data frame indicando que Joaquina e Praia Mole estão localizadas no leste da ilha (leste), e Campeche e Armação estão no sul (sul).
4. Selecione as praias de região leste que possuem menos de 20 caranguejos contados.
5. Você está interessado em saber em qual das duas praias do sul, o número de caranguejos contados foi maior do que 40. Usando a seleção condicional, mostre essa informação na tela.
6. Qual região possui praias com mais de 50 caranguejos contados?

Capítulo 4

Entrada e saída de dados no R

A entrada de dados no R pode ser realizada de diferentes formas. O formato mais adequado vai depender do tamanho do conjunto de dados, e se os dados já existem em outro formato para serem importados ou se serão digitados diretamente no R.

A seguir são descritas as formas de entrada de dados com indicação de quando cada uma das formas deve ser usada. Os três primeiros casos são adequados para entrada de dados diretamente no R, os seguintes descrevem como importar dados já disponíveis eletronicamentede um arquivo texto, em outro sistema ou no próprio R.

Posteriormente também será mostrado como fazer para exportar bases de dados geradas e/ou alteradas dentro do R.

4.1 Entrada de dados

4.1.1 Entrada de dados diretamente no R

4.1.1.1 Vetores

A forma mais básica de entrada de dados no R é através da função `c()` (como já vimos). A partir dela pode se criar os outros tipos de objetos como listas e data frames.

As funções básicas de entrada de dados são:

- `c()`
- `rep()`
- `seq()` ou `:`

A partir destas funções básicas podemos criar objetos de classes mais específicas com

- `matrix()`
- `list()`
- `data.frame()`

4.1.2 Entrada via teclado

4.1.2.1 Usando a função `scan()`

Esta função lê dados diretamento do console, isto é, coloca o R em modo *prompt* onde o usuário deve digitar cada dado seguido da tecla Enter. Para encerrar a entrada de dados basta digitar Enter duas vezes consecutivas.

Veja o seguinte resultado:

```
y <- scan()

1: 11
2: 24
3: 35
4: 29
5: 39
6: 47
7:
Read 6 items
```

```
y
[1] 11 24 35 29 39 47
```

Os dados também podem ser digitados em sequência, desde que separados por um espaço,

```
y <- scan()

1: 11 24
3: 35 29
5: 39 47
7:
Read 6 items
```

```
y
[1] 11 24 35 29 39 47
```

Este formato é mais ágil que o anterior (com `c()`, por exemplo) e é conveniente para digitar vetores longos. Esta função pode também ser usada para ler dados de um arquivo ou conexão, aceitando inclusive endereços de URLs (endereços da *web*) o que iremos mencionar em detalhes mais adiante.

Por padrão, a função `scan()` aceita apenas valores numéricos como entrada (lembre-se que vetores só podem ter elementos da mesma classe). Para alterar a classe de objeto de entrada, precisamos especificar o argumento `what` de `scan()`. Por exemplo, para entrar com um vetor de caracteres, fazemos

```
x <- scan(what = "character")

1: a
2: b
3: c
4:
Read 3 items
```

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

Outras classe possíveis para o argumento `what` são: `logical`, `integer`, `numeric`, `complex`, `character`, `raw` e `list`.

Exercícios

1. Usando a função `scan()` crie objetos para armazenar os seguintes valores:
   a. 19, 13, 19, 23, 18, 20, 25, 14, 20, 18, 22, 18, 23, 14, 19
   b. `joaquina`, `armação`, `praia brava`, `praia mole`, `morro das pedras`
   c. `TRUE`, `TRUE`, `FALSE`, `FALSE`, `TRUE`

4.1.2.2 Usando a função `readLines()`

Esta função é particularmente útil para ler entradas na forma de texto (*strings*). Por exemplo, para ler uma linha a ser digitada na tela do R, siga o comando abaixo e digite o texto indicado. Ao terminar pressione a tecla Enter e o texto será armazanado no objeto texto.

```
texto <- readLines(n = 1)
```

```
Estou digitando no console
```

```
texto
[1] "Estou digitando no console"
```

Um possível uso é dentro que funções que solicitem que o usuário responda e/ou entre com informações na medida que são solicitadas. Experimente definir e rodar o função a seguir.

```
fn.ex <- function() {
    cat("Digite o nome do time de futebol de sua preferência (em letras minúsculas)\n")
    time <- readLines(n = 1)
    if (time == "atletico-pr")
        cat("BOA ESCOLHA!!!\n")
    else cat("Ihh, tá mal de escolha...\n")
    return(invisible())
}
```

```
fn.ex()
```

Nesse exemplo, `readLines()` foi utilizada para efetuar a leitura via teclado, mas a função permite ainda entrada de dados por conexões com outros dispositivos de *input*. Por exemplo, pode ser utilizada para ler texto de um arquivo. Consulte a documentação da função para maiores detalhes e exemplos.

4.1.3 Entrada de dados em arquivos texto

Se os dados já estão disponíveis em formato eletrônico, isto é, já foram digitados em outro programa, voce pode importar os dados para o R sem a necessidade de digitá-los novamente.

A forma mais fácil de fazer isto é usar dados em formato texto (arquivo do tipo ASCII). Por exemplo, se seus dados estão disponíveis em uma planilha eletrônica como LibreOffice Calc, MS Excel ou similar, você pode escolher a opção Salvar como... e gravar os dados em um arquivo em formato texto. Os dois prncipais formatos de texto são:

- `txt`: arquivo de texto puro, onde as colunas são separadas geralmente por uma tabulação (Tab) ou espaço (Spc)
- `csv`: arquivo de texto, onde as colunas são geralmente separadas por vírgula (*comma separated value*), ou ponto-e-vírgula.

No R usa-se `scan()` mencionada anteriormente, ou então a função mais flexível `read.table()` para ler os dados de um arquivo texto e armazenar no formato de um data frame.

Antes de importar para o R:

- Se houverem valores perdidos, preencha com `NA`
- A matriz de dados deve formar um bloco só. Se houverem colunas de diferentes comprimentos, preencha com `NA`
- Salve o arquivo como "valores separados por vírgula" (`.csv`), mas atenção:
  - Se o separador de decimal for `,`, o separador de campos será `;` automaticamente (o que é mais comum nos sistemas em português).

4.1.3.1 A função `read.table()`

O método mais comum de importação de dados para o R, é utilizando a função `read.table()`. Como exemplo, baixe o arquivo `crabs.csv` disponível aqui, e salve em um diretório chamado `dados` no seu diretório de trabalho.

Para importar um arquivo `.csv` faça:

```
dados <- read.table("dados/crabs.csv", header = TRUE,
                    sep = ";", dec = ",")
```

Argumentos:

- `"crabs.csv"`: nome do arquivo. (Considerando que o arquivo `crabs.csv` está dentro do diretório `dados`).
- `header = TRUE`: significa que a primeira linha do arquivo deve ser interpretada como os nomes das colunas
- `sep = ";"`: o separador de colunas (também pode ser `","`, `"\t"` para tabulação e `" "` para espaços)
- `dec = ","`: o separador de decimais (também pode ser `"."`)

As funções `read.csv()` e `read.csv2()` são chamadas de *wrappers* (envelopes) que tornam o usa da função `read.table()` um pouco mais direta, alterando alguns argumentos. Por exemplo, o comando acima poderia ser substituído por

```
dados <- read.csv2("dados/crabs.csv")
```

O objeto criado com as funções `read.*()` sempre serão da classe `data.frame`, e quando houverem colunas com caracteres, estas colunas sempre serão da classe `factor`. Você pode alterar esse padrão usando o argumento `stringAsFactors = FALSE`

```
dados2 <- read.csv2("dados/crabs.csv", stringsAsFactors = FALSE)
```

Para conferir a estrutura dos dados importados, usamos a função `str()` que serve para demonstrar a estrutura de um objeto, como o nome das colunas e suas classes:

```
str(dados)
'data.frame':   156 obs. of  7 variables:
 $ especie: Factor w/ 2 levels "azul","laranja": 1 1 1 1 1 1 1 1 1 1 ...
 $ sexo   : Factor w/ 2 levels "F","M": 2 2 2 2 2 2 2 2 2 2 ...
 $ FL     : num  8.1 8.8 9.2 9.6 10.8 11.6 11.8 12.3 12.6 12.8 ...
 $ RW     : num  6.7 7.7 7.8 7.9 9 9.1 10.5 11 10 10.9 ...
 $ CL     : num  16.1 18.1 19 20.1 23 24.5 25.2 26.8 27.7 27.4 ...
 $ CW     : num  19 20.8 22.4 23.1 26.5 28.4 29.3 31.5 31.7 31.5 ...
 $ BD     : num  7 7.4 7.7 8.2 9.8 10.4 10.3 11.4 11.4 11 ...
str(dados2)
'data.frame':   156 obs. of  7 variables:
 $ especie: chr  "azul" "azul" "azul" "azul" ...
 $ sexo   : chr  "M" "M" "M" "M" ...
 $ FL     : num  8.1 8.8 9.2 9.6 10.8 11.6 11.8 12.3 12.6 12.8 ...
 $ RW     : num  6.7 7.7 7.8 7.9 9 9.1 10.5 11 10 10.9 ...
 $ CL     : num  16.1 18.1 19 20.1 23 24.5 25.2 26.8 27.7 27.4 ...
 $ CW     : num  19 20.8 22.4 23.1 26.5 28.4 29.3 31.5 31.7 31.5 ...
 $ BD     : num  7 7.4 7.7 8.2 9.8 10.4 10.3 11.4 11.4 11 ...
```

Podemos também visualizar algumas linhas iniciais e finais do objeto importado através de duas funções auxiliares:

```
head(dados)
  especie sexo   FL  RW   CL   CW   BD
1    azul    M  8.1 6.7 16.1 19.0  7.0
```

```
2    azul    M  8.8 7.7 18.1 20.8  7.4
3    azul    M  9.2 7.8 19.0 22.4  7.7
4    azul    M  9.6 7.9 20.1 23.1  8.2
5    azul    M 10.8 9.0 23.0 26.5  9.8
6    azul    M 11.6 9.1 24.5 28.4 10.4
tail(dados)
    especie sexo   FL   RW   CL   CW   BD
151 laranja    F 21.3 18.4 43.8 48.4 20.0
152 laranja    F 21.4 18.0 41.2 46.2 18.7
153 laranja    F 21.7 17.1 41.7 47.2 19.6
154 laranja    F 21.9 17.2 42.6 47.4 19.5
155 laranja    F 22.5 17.2 43.0 48.7 19.8
156 laranja    F 23.1 20.2 46.2 52.5 21.1
```

As funções permitem ainda ler dados diretamente disponíveis na *web*. Por exemplo, os dados do exemplo poderiam ser lidos diretamente com o comando a seguir, sem a necessidade de copiar primeiro os dados para algum local no computador do usuário:

```
dados <- read.csv2("http://www.leg.ufpr.br/~fernandomayer/data/crabs.csv")
```

Para maiores informações consulte a documentação desta função com `?read.table()`. Embora `read.table()` seja provavelmente a função mais utilizada existem outras que podem ser úteis e determinadas situações:

- `read.fwf()` é conveniente para ler *fixed width formats*
- `read.fortran()` é semelhante à anterior porém usando o estilo Fortran de especificação das colunas
- `read.csv()`, `read.csv2()`, `read.delim()` e `read.delim2()`: estas funções são praticamente iguais a `read.table()` porém com diferentes opções padrão. Em geral (mas não sempre) dados em formato `csv` usado no Brasil são lidos diretamente com `read.csv2()`.

Exercícios

1. Baixe os arquivos a seguir e coloque os arquivos em um local apropriado (de preferência no mesmo diretório de trabalho que voce definiu no início da sessão), faça a importação usando a função `read.table()`, e confira a estrutura dos dados com `str()`.
   a. prb0519.dat
   b. tab0303.dat
   c. tab1208.dat
   d. ReadMe.txt
   e. montgomery_6-26.csv
   f. montgomery_14-12.txt
   g. montgomery_ex6-2.csv
   h. ipea_habitacao.csv
   i. stratford.csv
2. Faça a leitura dos dados do exercício anterior, mas agora utilize o endereço *web* dos arquivos.

4.1.4 Entrada de dados através da área de transferência

Um mecanismos comum para copiar dados de um programa para o outro é usando a **área de transferência** (ou *clipboard*). Tipicamente isto é feito com o mecanismo de copia-e-cola, ou seja-se, marca-se os dados desejados em algum aplicativo (editor, planilha, página web, etc), usa-se o mecanismo de COPIAR (opção no menu do programa que muitas vezes corresponde o teclar Ctrl + c), o que transfere os dados para a área de transferência. Funções como `scan()`, `read.table()` e

outras podem ser usades apra ler os dados diretamente da área de transferência passando-se a opção "clipboard" ao primeiro argumento. Por exemplo, os seguintes dados:

```
ID  Grupo  Gasto  Ano
23      A  25,4   11
12      B  12,3   09
23      A  19,8   07
```

podem ser marcados e copiados para área de transferência e lidos diretamente com

```
dados.clip <- read.table("clipboard", header = TRUE, dec = ",")
```

```
str(dados.clip)
'data.frame':   3 obs. of  4 variables:
 $ ID   : int  23 12 23
 $ Grupo: Factor w/ 2 levels "A","B": 1 2 1
 $ Gasto: num  25.4 12.3 19.8
 $ Ano  : int  11 9 7
```

4.1.5 Importando dados diretamente de planilhas

Existem alguns pacotes disponíveis que podem ler dados diretamente de planilhas do MS Excel. No entanto, estes pacotes geralmente possuem particularidades quanto ao sistema operacional e demais dependências para funcionar corretamente.

Um destes pacotes, é o **gdata**, que funciona em diversos sistemas operacionais mas depende da linguagem Perl estar instalada. Por exemplo, para ler o conjuto de dados crabs armazenado em uma planilha do Excel (disponível aqui), podemos usar

```
## Carrega o pacote
library(gdata)
## Leitura diretamente do Excel
dados.xls <- read.xls("dados/crabs.xls", sheet = "Plan1",
                      header = TRUE, dec = ",")
## Estrutura
str(dados.xls)
'data.frame':   156 obs. of  7 variables:
 $ especie: Factor w/ 2 levels "azul","laranja": 1 1 1 1 1 1 1 1 1 1 ...
 $ sexo   : Factor w/ 2 levels "F","M": 2 2 2 2 2 2 2 2 2 2 ...
 $ FL     : num  8.1 8.8 9.2 9.6 10.8 11.6 11.8 12.3 12.6 12.8 ...
 $ RW     : num  6.7 7.7 7.8 7.9 9 9.1 10.5 11 10 10.9 ...
 $ CL     : num  16.1 18.1 19 20.1 23 24.5 25.2 26.8 27.7 27.4 ...
 $ CW     : num  19 20.8 22.4 23.1 26.5 28.4 29.3 31.5 31.7 31.5 ...
 $ BD     : num  7 7.4 7.7 8.2 9.8 10.4 10.3 11.4 11.4 11 ...
```

Outros pacotes que possuem funções similares são: **openxlsx**, **xlsx**, e **XLConnect**.

Estruturas de dados mais complexas são tipicamente armazenadas nos chamados DBMS (*database management system*) ou RDBMS (*relational database management system*). Aguns exemplos são Oracle, Microsoft SQL server, MySQL, PostgreSQL, Microsoft Access, dentre outros. O R possui ferramentas implementadas em pacotes para acesso a estes sistemas gerenciadores.

Para mais detalhes consulte o manual R Data Import/Export e a documentação dos pacotes que implementam tal funcionalidade. Alguns destes pacotes disponíveis são: **RODBC**, **DBI**, **RMySQL**, **RPostgreSQL**, **ROracle**, **RNetCDF**, **RSQLite**, dentre outros.

4.1.6 Carregando dados já disponíveis no R

O R já possui alguns conjuntos de dados que estão disponíeis logo após a instalação. Estes dados são também objetos que precisam ser carregados para ficarem disponiveis para o usuário. Normalmente, estes conjuntos de dados são para uso de exemplo de funções.

Para carregar conjuntos de dados que são disponibilizados com o R, use o comando `data()`. Por exemplo, abaixo mostramos como carregar o conjunto `mtcars` que está no pacote **datasets**.

```
## Objetos criados até o momento nesta seção
ls()
[1] "dados"      "dados.clip" "dados.xls"  "dados2"     "fn.ex"
[6] "texto"      "x"          "y"
## Carrega a base de dados mtcars
data(mtcars)
## Note como agora o objeto mtcars fica disponível na sua área de
## trabalho
ls()
[1] "dados"      "dados.clip" "dados.xls"  "dados2"     "fn.ex"
[6] "mtcars"     "texto"      "x"          "y"
## Estrutura e visualização do objeto
str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

As bases de dados também possuem páginas de documentação para explicar o que são os dados e as colunas correspondentes. Para ver o que são os dados do `mtcars` por exemplo, veja `?mtcars`.

O conjunto `mtcars` é disponibilizado prontamente pois faz parte do pacote **datasets**, que por padrão é sempre carregado na inicialização do R. No entanto, existem outros conjuntos de dados, disponibilizados por outros pacotes, que precisam ser carregados para que os dados possam ser disponibilizados. Por exemplo, os dados do objeto `topo` são do pacote **MASS**. Se tentarmos fazer

```
data(topo)
Warning in data(topo): data set 'topo' not found
```

Portanto, precisamos primeiro carregar o pacote **MASS** com

```
library(MASS)
```

e agora podemos carregar o objeto `topo` com

```
data(topo)
## O objeto fica disponível na sua área de trabalho
ls()
 [1] "dados"      "dados.clip" "dados.xls"  "dados2"     "fn.ex"
 [6] "mtcars"     "texto"      "topo"       "x"          "y"
## Confere a estrutura
str(topo)
'data.frame':   52 obs. of  3 variables:
 $ x: num  0.3 1.4 2.4 3.6 5.7 1.6 2.9 3.4 3.4 4.8 ...
 $ y: num  6.1 6.2 6.1 6.2 6.2 5.2 5.1 5.3 5.7 5.6 ...
 $ z: int  870 793 755 690 800 800 730 728 710 780 ...
```

A função `data()` pode ainda ser usada para listar os conjutos de dados disponíveis.

```
data()
```

e também pode ser útil para listar os conjuntos de dados disponíveis para um pacote específico, por exemplo

```
data(package = "nlme")
```

4.1.7 Importando dados de outros programas

É possível ler dados diretamente de outros formatos que não seja texto (ASCII). Isto em geral é mais eficiente e requer menos memória do que converter para formato texto. Há funções para importar dados diretamente de EpiInfo, Minitab, S-PLUS, SAS, SPSS, Stata, Systat e Octave. Além disto é comum surgir a necessidade de importar dados de planilhas eletrônicas. Muitas funções que permitem a importação de dados de outros programas são implementadas no pacote **foreign**.

A seguir listamos algumas (não todas!) destas funções:

- `read.dbf()` para arquivos DBASE
- `read.epiinfo()` para arquivos .REC do Epi-Info
- `read.mtp()` para arquivos "Minitab Portable Worksheet"
- `read.S()` para arquivos do S-PLUS, e `restore.data()` para "dumps" do S-PLUS
- `read.spss()` para dados do SPSS
- `read.systat()` para dados do SYSTAT
- `read.dta()` para dados do STATA
- `read.octave()` para dados do OCTAVE (um clone do MATLAB)
- Para dados do SAS há ao menos duas alternativas:
  - O pacote **foreign** disponibiliza `read.xport()` para ler do formato TRANSPORT do SAS e `read.ssd()` pode escrever dados permanentes do SAS (`.ssd` ou `.sas7bdat`) no formato TRANSPORT, se o SAS estiver disponível no seu sistema e depois usa internamente `read.xport()` para ler os dados no R.
  - O pacote **Hmisc** disponibiliza `sas.get()` que também requer o SAS no sistema.

Para mais detalhes consulte a documentação de cada função e/ou o manual R Data Import/Export.

4.2 Saída de dados do R

4.2.1 Usando a função `write.table()`

Para exportar objetos do R, usamos a função `write.table()`, que possui argumentos parecidos com aqueles da função `read.table()`.

A função `write.table()` é capaz de criar um arquivo de texto no formato `txt` ou `csv`, com as especificações definidas pelos argumentos.

Para ilustrar o uso desta função, considerer o conjunto de dados `iris`

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

Podemos exportar esse data frame com

```
write.table(iris, file = "dados/iris.csv")
```

Por padrão, o arquivo resultante tem colunas separadas por espaço, o separador de decimal é ponto, e os nomes das linhas são também incluídos (o que geralmente é desnecessário). Para alterar essa configuração podemos fazer

```
write.table(iris, file = "dados/iris.csv", row.names = FALSE,
            sep = ";", dec = ",")
```

Os argumentos são

- `iris`: o nome do objeto a ser exportado (matriz ou data frame)
- `"iris.csv"`: nome do arquivo a ser gerado. (Considerando que o arquivo `iris.csv` será criado dentro do diretório `dados`).
- `row.names = FALSE`: para eliminar o nome das linhas do objeto (geralmente desnecessário), como retornado por `row.names()`
- `sep = ";"`: o separador de colunas (também pode ser `","`, `"\t"` para tabulação e `" "` para espaços)
- `dec = ","`: o separador de decimais (também pode ser `"."`)

Note que o objeto a ser exportado (nesse caso `iris`) deve ser em formato tabular, ou seja, uma matriz ou data frame. Outras classes de objetos podem ser exportadas, mas haverá uma coerção para data frame, o que pode fazer com que o resultado final não seja o esperado.

Assim como `read.table()` possui as funções `read.csv()` e `read.csv2()`, a função `write.table()` possui as funções `write.table()` e `write.table2()` como *wrappers*. O comando acima também poderia ser executado como

```
write.csv2(iris, file = "dados/iris.csv", row.names = FALSE)
```

Note que `row.names = FALSE` ainda é necessário para eliminar os nomes das linhas.

O pacote **foreign** também possui funções para exportar para uma variedade de formatos. Veja a documentação em `help(package = "foreign")`. Os pacotes para ler dados diretamente de arquivos do MS Excel mencionados acima também possuem funções para exportar diretamente para esse formato.

Exercícios

1. Considere a tabela abaixo com o resultado de uma pesquisa que avaliou o número de fumates e não fumantes por sexo.
   Sexo
   Condição
   Masculino
   Feminino

Fumante
49
54
64
61
37
79
52
64
68
29
Não fumante
27
40
58
39
52
44
41
34
30
44

2. Digite estes dados em uma planilha eletrônica em um formato apropriado para um data frame do R, e salve em um arquivo `csv`.
3. Importe esse arquivo para o R com `read.table()`.
4. Crie uma nova coluna no objeto que contém estes dados, sendo a coluna com o número de pessoas multiplicada por 2.
5. Exporte esse novo objeto usando a função `write.table()`.
6. Tente criar esse mesmo conjunto de dados usando comandos do R (ex.: `c()`, `rep()`, `data.frame()`, etc.)

4.2.2 Usando os formatos textual e binário para ler/escrever dados

As formas mais comuns de entrada de dados no R são através da entrada direta pelo teclado (e.g. `c()` ou `scan()`), ou pela importação de arquivos de texto (e.g. `read.table()`). No etanto, ainda existem mais dois formatos par armazenar dados para leitura no R: o textual e o binário.

O **formato binário** é aquele armazenado em um arquivo binário, ou seja, um arquivo que contém apenas 0s e 1s, e possui um formato específico que só pode ser lido por determinado *software* ou função. É o oposto de um arquivo de texto, por exemplo, que podemos abrir e editar em qualquer prorama que edite texto puro.

O **formato textual** é o intermediário entre o texto puro e o binário. Os dados em formato textual são apresentados como texto puro, mas contém informações adicionais, chamados de **metadados**, que preservam toda a estrutura dos dados, como as classes de cada coluna de um data frame.

4.2.2.1 Formato textual

O formato textual é muito útil para compartilhar conjuntos de dados que não são muito grandes, e onde a formatação (leia-se: classes de objetos) precisa ser mantida.

Para criar um conjunto de dados no formato textual, usamos a função `dput()`. Vamos criar um data frame de exemplo e ver o resultado da chamada dessa função:

```
da <- data.frame(A = c(1, 2), B = c("a", "b"))
dput(da)
```

```
structure(list(A = c(1, 2), B = structure(1:2, .Label = c("a",
"b"), class = "factor")), class = "data.frame", row.names = c(NA,
-2L))
```

Note que o resultado de `dput()` é no formato do R, e preserva metadados como as classes do objeto e de cada coluna, e os nomes das linhas e colunas.

Outas classes de objetos são facilmente preservadas quando armazenadas com o resultado de `dput()`. Por exemplo, uma matriz:

```
ma <- matrix(1:9, ncol = 3)
dput(ma)
structure(1:9, .Dim = c(3L, 3L))
```

E uma lista:

```
la <- list(da, ma)
dput(la)
list(structure(list(A = c(1, 2), B = structure(1:2, .Label = c("a",
"b"), class = "factor")), class = "data.frame", row.names = c(NA,
-2L)), structure(1:9, .Dim = c(3L, 3L)))
```

A saída da função `dput()` pode ser copiada para um script do R, para garantir que qualquer pessoa que venha usar o código (incluindo você no futuro), usará os dados no formato correto (esperado). Isso é muito importante para a **pesquisa reproduzível**!

A saída de `dput()` também pode ser salva diretamente em um arquivo de script do R, por exemplo,

```
dput(da, file = "da.R")
```

irá criar o arquivo `da.R` com o resultado da função. Para importar os dados salvos dessa forma, usamos a função `dget()`,

```
da2 <- dget(file = "da.R")
da2
  A B
1 1 a
2 2 b
```

Múltiplos objetos podem ser armazenados em formato textual usando a função `dump()`.

```
dump(c("da", "ma", "la"), file = "dados.R")
```

Note que os objetos são passados como um vetor de caracteres, e um arquivo chamado `dados.R` é criado com todos os objetos no formato textual. Para importar estes objetos para uma sessão do R, usamos a função `source()`,

```
source("dados.R")
```

que já cria os objetos na sua área de trabalho com os mesmos nomes e atributos como foram armazenados.

4.2.2.2 Formato binário

Armazenar dados em formato binário é vantajoso quando não há uma forma "fácil" de armazenar os dados em formato de texto puro ou textual. Além disso, algumas vezes o formato binário possui maior eficiência em termos de velocidade de leitura/escrita, dependendo dos dados. Outra vantagem é que valores numéricos geralmente perdem precisão quando armazenados em texto ou textual, enquanto que o formato binário preserva toda a precisão (embora essa perda de precisão seja desprezível na maioria dos casos).

Para salvar um objeto contendo dados no R, usamos a função `save()`. Por exemplo, para armazenar o objeto da criado acima, fazemos

```
save(da, file = "dados.rda")
```

Esse comando irá criar o arquivo (binário) `dados.rda`. Note que a extensão `.rda` é comumente utilizada para dados binários do R, mas não é única.

Paa salvar mais de um objeto no mesmo arquivo, basta passar os nomes na mesma função

```
save(da, ma, file = "dados.rda")
```

A função `save.image()` pode ser utilizada se a intenção é salvar **todos** os objetos criados na sua área de trabalho (isso inclui qualquer objeto, não só os conjuntos de dados). Nesse caso, podemos fazer

```
save.image(file = "workspace.RData")
```

Note que quando foi utilizada a função `save()`, a extensão do arquivo foi `rda`, e com `save.image()` foi `RData`. Isso é uma convenção comum de arquivos binários do R, mas não é obrigatório. Qualquer uma das extensões funciona em ambas as funções.

Para carregar os conjuntos de dados (ou de forma mais geral, os objetos) armazenados em formato binário, usamos a função `load()`

```
load("dados.rda")
load("workspace.RData")
```

Dessa forma, os objetos já estarão disponíveis na sua área de trabalho.

4.3 Informações sobre diretórios e arquivos

O R possui uma variedade de funções para mostrar informações sobre arquivos e diretórios. Alguns exemplos são:

- `file.info()` mostra o tamanho do arquivo, data de criação, ...
- `dir()` mostra todos os arquivos presentes em um diretório (tente com `recursive = TRUE`)
- `file.exists()` retorna `TRUE` ou `FALSE` para a presença de um arquivo
- `getwd()` e `setwd()` para verificar e altear o diretório de trabalho

Veja `?files` para uma lista copleta de funções úteis para manipular arquivos de dentro do R.

Capítulo 5

Programando com dados

Por quê programar?

- Evitar repetições desnecessárias de análises ou cálculos que são repetidos com frequência.
- Fica documentado as etapas que você realizou para chegar a um resultado.
- Fácil recuperação e modificação de programas.

Como programar?

- Criando programas! (Scripts, rotinas, **algoritmos**).
- Crie uma sequência lógica de comandos que devem ser executados em ordem.
- Utilize as ferramentas básicas da programação: **estruturas de repetição** (`for()`) e **estruturas de seleção** (`if()`).

5.1 Estrutura de repetição `for()`

Serve para repetir um ou mais comandos diversas vezes. Para ver como funciona, considere o seguinte exemplo:

```
for(i in 1:10){
    print(i)
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

O resultado é a chamada do comando `print()` para cada valor que o índice `i` recebe (nesse caso `i` recebe os valores de 1 a 10).

A sintaxe será sempre nesse formato:

```
for(<índice> in <valores>){
    <comandos>
}
```

Veja outro exemplo em como podemos aplicar o índice:

```
x <- 100:200
for(j in 1:10){
    print(x[j])
}
[1] 100
[1] 101
[1] 102
[1] 103
[1] 104
[1] 105
[1] 106
[1] 107
[1] 108
[1] 109
```

Veja que o índice não precisa ser `i`, na verdade pode ser qualquer letra ou palavra. Nesse caso, veja que utilizamos os valores como índice para selecionar elementos de `x` naquelas posições específicas.

Um outro exemplo seria se quisessemos imprimir o quadrado de alguns números (não necessariamente em sequência):

```
for(i in c(2, 9, 4, 6)){
    print(i^2)
}
[1] 4
[1] 81
[1] 16
[1] 36
```

Ou mesmo imprimir caracteres a partir de um vetor de caracteres:

```
for(veiculos in c("carro", "ônibus", "trem", "bicicleta")){
    print(veiculos)
}
[1] "carro"
[1] "ônibus"
[1] "trem"
[1] "bicicleta"
```

Exemplo: cálculo de notas de uma disciplina.

```
## Importa os dados
url <- "http://leg.ufpr.br/~fernandomayer/data/notas.csv"
notas <- read.table(url, header = TRUE, sep = ";", dec = ",")
## Analisa a estrutura dos dados
str(notas)
'data.frame':   30 obs. of  4 variables:
 $ nome  : Factor w/ 30 levels "Aluno_1","Aluno_10",..: 1 12 23 25 26 27 28 29 30 2 ...
 $ prova1: int  8 2 9 1 7 10 1 5 5 10 ...
 $ prova2: int  4 7 2 10 6 0 8 9 6 2 ...
 $ prova3: int  1 6 4 9 8 3 0 7 1 3 ...
head(notas)
     nome prova1 prova2 prova3
1 Aluno_1      8      4      1
2 Aluno_2      2      7      6
3 Aluno_3      9      2      4
4 Aluno_4      1     10      9
```

```
5 Aluno_5      7      6      8
6 Aluno_6     10      0      3
summary(notas)
       nome        prova1          prova2          prova3
 Aluno_1 : 1   Min.   : 0.000   Min.   : 0.000   Min.   :0.0
 Aluno_10: 1   1st Qu.: 2.000   1st Qu.: 3.000   1st Qu.:3.0
 Aluno_11: 1   Median : 4.000   Median : 6.000   Median :6.5
 Aluno_12: 1   Mean   : 4.433   Mean   : 5.433   Mean   :5.4
 Aluno_13: 1   3rd Qu.: 6.750   3rd Qu.: 8.000   3rd Qu.:8.0
 Aluno_14: 1   Max.   :10.000   Max.   :10.000   Max.   :9.0
 (Other) :24
```

Antes de seguir adiante, veja o resultado de

```
for(i in 1:30){
    print(notas[i, c("prova1", "prova2", "prova3")])
}
```

Para calcular as médias das 3 provas, precisamos inicialmente de um vetor para armazenar os resultados. Esse vetor pode ser um novo objeto ou uma nova coluna no dataframe

```
## Aqui vamos criar uma nova coluna no dataframe, contendo apenas o
## valor 0
notas$media <- 0 # note que aqui será usada a regra da reciclagem, ou
                 # seja, o valor zero será repetido até completar todas
                 # as linhas do dataframe
## Estrutura de repetição para calcular a média
for(i in 1:30){
    ## Aqui, cada linha i da coluna media sera substituida pelo
    ## respectivo valor da media caculada
    notas$media[i] <- sum(notas[i, c("prova1", "prova2", "prova3")])/3
}

## Confere os resultados
head(notas)
     nome prova1 prova2 prova3    media
1 Aluno_1      8      4      1 4.333333
2 Aluno_2      2      7      6 5.000000
3 Aluno_3      9      2      4 5.000000
4 Aluno_4      1     10      9 6.666667
5 Aluno_5      7      6      8 7.000000
6 Aluno_6     10      0      3 4.333333
```

Agora podemos melhorar o código, tornando-o mais **genérico**. Dessa forma fica mais fácil fazer alterações e procurar erros. Uma forma de melhorar o código acima é generalizando alguns passos.

```
## Armazenamos o número de linhas no dataframe
nlinhas <- nrow(notas)
## Identificamos as colunas de interesse no cálculo da média, e
## armazenamos em um objeto separado
provas <- c("prova1", "prova2", "prova3")
## Sabendo o número de provas, fica mais fácil dividir pelo total no
## cálculo da média
nprovas <- length(provas)
## Cria uma nova coluna apenas para comparar o cálculo com o anterior
notas$media2 <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
```

```
    notas$media2[i] <- sum(notas[i, provas])/nprovas
}

## Confere
head(notas)
     nome prova1 prova2 prova3    media   media2
1 Aluno_1      8      4      1 4.333333 4.333333
2 Aluno_2      2      7      6 5.000000 5.000000
3 Aluno_3      9      2      4 5.000000 5.000000
4 Aluno_4      1     10      9 6.666667 6.666667
5 Aluno_5      7      6      8 7.000000 7.000000
6 Aluno_6     10      0      3 4.333333 4.333333
identical(notas$media, notas$media2)
[1] TRUE
```

Ainda podemos melhorar (leia-se: **otimizar**) o código, se utilizarmos funções prontas do R. No caso da média isso é possível pois a função `mean()` já existe. Em seguida veremos como fazer quando o cálculo que estamos utilizando não está implementado em nenhuma função pronta do R.

```
## Cria uma nova coluna apenas para comparação
notas$media3 <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    notas$media3[i] <- mean(as.numeric(notas[i, provas]))
}

## Confere
head(notas)
     nome prova1 prova2 prova3    media   media2   media3
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333

## A única diferença é que aqui precisamos transformar cada linha em um
## vetor de números com as.numeric(), pois
notas[1, provas]
  prova1 prova2 prova3
1      8      4      1
## é um data.frame:
class(notas[1, provas])
[1] "data.frame"
```

No caso acima vimos que não era necessário calcular a média através de `soma/total` porque existe uma função pronta no R para fazer esse cálculo. Mas, e se quisessemos, por exemplo, calcular a Coeficiente de Variação (CV) entre as notas das três provas de cada aluno? Uma busca por

```
help.search("coefficient of variation")
```

não retorna nenhuma função (dos pacotes básicos) para fazer esse cálculo. O motivo é simples: como é uma conta simples de fazer não há necessidade de se criar uma função extra dentro dos pacotes. No entanto, nós podemos criar uma função que calcule o CV, e usá-la para o nosso propósito

```
cv <- function(x){
    desv.pad <- sd(x)
```

```
    med <- mean(x)
    cv <- desv.pad/med
    return(cv)
}
```

NOTA: na função criada acima o único argumento que usamos foi `x`, que neste caso deve ser um vetor de números para o cálculo do CV. Os argumentos colocados dentro de `function()` devem ser apropriados para o propósito de cada função.

Antes de aplicar a função dentro de um `for()` devemos testá-la para ver se ela está funcioanando de maneira correta. Por exemplo, o CV para as notas do primeiro aluno pode ser calculado "manualmente" por

```
sd(as.numeric(notas[1, provas]))/mean(as.numeric(notas[1, provas]))
[1] 0.8104349
```

E através da função, o resultado é

```
cv(as.numeric(notas[1, provas]))
[1] 0.8104349
```

o que mostra que a função está funcionando corretamente, e podemos aplicá-la em todas as linhas usando a repetição

```
## Cria uma nova coluna para o CV
notas$CV <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    notas$CV[i] <- cv(as.numeric(notas[i, provas]))
}

## Confere
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157
```

Podemos agora querer calcular as médias ponderadas para as provas. Por exemplo:

- Prova 1: peso 3
- Prova 2: peso 3
- Prova 3: peso 4

Usando a fórmula:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{n} x_i \cdot w_i$$

onde w_i são os pesos, e $N = \sum_{i=1}^{n} w_i$ é a soma dos pesos. Como já vimos que criar uma função é uma forma mais prática (e elegante) de executar determinada tarefa, vamos criar uma função que calcule as médias ponderadas.

```
med.pond <- function(notas, pesos){
    ## Multiplica o valor de cada prova pelo seu peso
    pond <- notas * pesos
    ## Calcula o valor total dos pesos
```

```
    peso.total <- sum(pesos)
    ## Calcula a soma da ponderação
    sum.pond <- sum(pond)
    ## Finalmente calcula a média ponderada
    saida <- sum.pond/peso.total
    return(saida)
}
```

Antes de aplicar a função para o caso geral, sempre é importante testar e conferir o resultado em um caso menor. Podemos verificar o resultado da média ponderada para o primeiro aluno

```
sum(notas[1, provas] * c(3, 3, 4))/10
[1] 4
```

e testar a função para o mesmo caso

```
med.pond(notas = notas[1, provas], pesos = c(3, 3, 4))
[1] 4
```

Como o resultado é o mesmo podemos aplicar a função para todas as linhas através do `for()`

```
## Cria uma nova coluna para a média ponderada
notas$MP <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    notas$MP[i] <- med.pond(notas = notas[i, provas], pesos = c(3, 3, 4))
}

## Confere
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
```

NOTA: uma função para calcular a média ponderada já existe implementada no R. Veja `?weighted.mean()` e confira os resultados obtidos aqui

Repare na construção da função acima: agora usamos dois argumentos, `notas` e `pesos`, pois precisamos dos doiss vetores para calcular a média ponderada. Repare também que ambos argumentos não possuem um valor padrão. Poderíamos, por exemplo, assumir valores padrão para os pesos, e deixar para que o usuário mude apenas se achar necessário.

```
## Atribuindo pesos iguais para as provas como padrão
med.pond <- function(notas, pesos = rep(1, length(notas))){
    ## Multiplica o valor de cada prova pelo seu peso
    pond <- notas * pesos
    ## Calcula o valor total dos pesos
    peso.total <- sum(pesos)
    ## Calcula a soma da ponderação
    sum.pond <- sum(pond)
    ## Finalmente calcula a média ponderada
    saida <- sum.pond/peso.total
    return(saida)
}
```

Repare que neste caso, como os pesos são iguais, a chamada da função sem alterar o argumento

pesos gera o mesmo resultado do cálculo da média comum.

```
## Cria uma nova coluna para a média ponderada para comparação
notas$MP2 <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    notas$MP2[i] <- med.pond(notas = notas[i, provas])
}

## Confere
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2
1 4.333333
2 5.000000
3 5.000000
4 6.666667
5 7.000000
6 4.333333
```

5.2 Estrutura de seleção `if()`

Uma estrutura de seleção serve para executar algum comando apenas se alguma condição (em forma de **expressão condicional**) seja satisfeita. Geralmente é utilizada dentro de um `for()`.

No exemplo inicial poderíamos querer imprimir um resultado caso satisfaça determinada condição. Por exemplo, se o valor de x for menor ou igual a 105, então imprima um texto informando isso.

```
x <- 100:200
for(j in 1:10){
    if(x[j] <= 105){
        print("Menor ou igual a 105")
    }
}
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
```

Mas também podemos considerar o que aconteceria caso contrário. Por exemplo, se o valor de xfor maior do que 105, então imprima outro texto.

```
x <- 100:200
for(j in 1:10){
    if(x[j] <= 105){
        print("Menor ou igual a 105")
    } else{
        print("Maior do que 105")
```

```
    }
}
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Menor ou igual a 105"
[1] "Maior do que 105"
[1] "Maior do que 105"
[1] "Maior do que 105"
[1] "Maior do que 105"
```

A sintaxe será sempre no formato:

```
if(<condição>){
    <comandos que satisfazem a condição>
} else{
   <comandos que não satisfazem a condição>
}
```

Como vimos acima, a especificação do `else{}` não é obrigatória.

Voltando ao exemplo das notas, podemos adicionar uma coluna com a condição do aluno: `aprovado` ou `reprovado` de acordo com a sua nota. Para isso precisamos criar uma condição (nesse caso se a nota é maior do que 7), e verificar se ela é verdadeira.

```
## Nova coluna para armazenar a situacao
notas$situacao <- NA # aqui usamos NA porque o resultado será um
                     # caracter
## Estrutura de repetição
for(i in 1:nlinhas){
    ## Estrutura de seleção (usando a média ponderada)
    if(notas$MP[i] >= 7){
        notas$situacao[i] <- "aprovado"
    } else{
        notas$situacao[i] <- "reprovado"
    }
}

## Confere
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2  situacao
1 4.333333 reprovado
2 5.000000 reprovado
3 5.000000 reprovado
4 6.666667 reprovado
5 7.000000  aprovado
6 4.333333 reprovado
```

5.3 O modo R: vetorização

As funções vetorizadas do R, além de facilitar e resumir a execução de tarefas repetitivas, também são computacionalmente mais eficientes, *i.e.* o tempo de execução das rotinas é muito mais rápido.

Já vimos que a **regra da reciclagem** é uma forma de vetorizar cálculos no R. Os cálculos feitos com funções vetorizadas (ou usando a regra de reciclagem) são muito mais eficientes (e preferíveis) no R. Por exemplo, podemos criar um vetor muito grande de números e querer calcular o quadrado de cada número. Se pensássemos em usar uma estrutura de repetição, o cálculo seria o seguinte:

```
## Vetor com uma sequência de 1 a 1.000.000
x <- 1:1000000
## Calcula o quadrado de cada número da sequência em x usando for()
y1 <- numeric(length(x)) # vetor de mesmo comprimento de x que vai
                         # receber os resultados
for(i in 1:length(x)){
    y1[i] <- x[i]^2
}
```

Mas, da forma vetorial e usando a regra da reciclagem, a mesma operação pode ser feita apenas com

```
## Calcula o quadrado de cada número da sequência em x usando a regra da
## reciclagem
y2 <- x^2
## Confere os resultados
identical(y1, y2)
[1] TRUE
```

Note que os resultados são exatamente iguais, mas então porque se prefere o formato vetorial? Primeiro porque é muito mais simples de escrever, e segundo (e principalmente) porque a forma vetorizada é muito mais **eficiente computacionalmente**. A eficiência computacional pode ser medida de várias formas (alocação de memória, tempo de execução, etc), mas apenas para comparação, vamos medir o tempo de execução destas mesmas operações usando o `for()` e usando a regra da reciclagem.

```
## Tempo de execução usando for()
y1 <- numeric(length(x))
st1 <- system.time(
    for(i in 1:length(x)){
        y1[i] <- x[i]^2
    }
)
st1
   user  system elapsed
  0.104   0.004   0.110

## Tempo de execução usando a regra da reciclagem
st2 <- system.time(
    y2 <- x^2
)
st2
   user  system elapsed
  0.000   0.000   0.002
```

Olhando o resultado de `elapsed`, que é o tempo total de execução de uma função medido por `system.time()`, notamos que usando a regra da reciclagem, o cálculo é aproximadamente $0.11/0.002 = 55$ vezes mais rápido. Claramente esse é só um exemplo de um cálculo muito simples.

Mas em situações mais complexas, a diferença entro o tempo de execução das duas formas pode ser muito maior.

Uma nota de precaução

Existem duas formas básicas de tornar um loop `for` no R mais rápido:

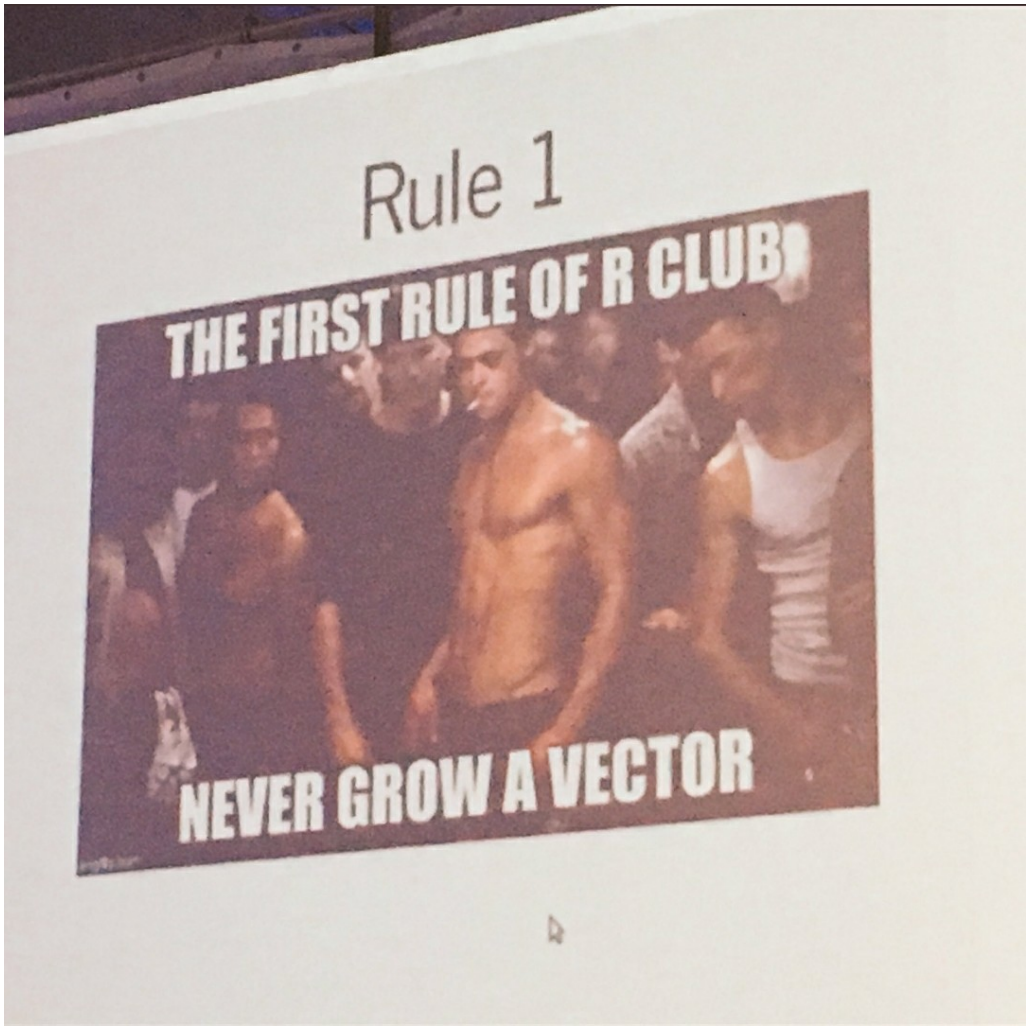
1. Faça o máximo possível fora do loop
2. Crie um objeto com tamanho suficiente para armazenar *todos* os resultados do loop **antes** de executá-lo

Veja este exemplo:

```
## Vetor com uma sequência de 1 a 1.000.000
x <- 1:1000000

## Cria um objeto de armazenamento com o mesmo tamanho do resultado
st1 <- system.time({
    out <- numeric(length(x))
    for(i in 1:length(x)){
        out[i] <- x[i]^2
    }
})
st1
   user  system elapsed
  0.104   0.004   0.108

## Cria um objeto de tamanho "zero" e vai "crescendo" esse vetor
st2 <- system.time({
    out <- numeric(0)
    for(i in 1:length(x)){
        out[i] <- x[i]^2
    }
})
st2
   user  system elapsed
  0.328   0.012   0.343
```

Essa simples diferença gera um aumento de tempo de execução da segunda forma em aproximadamente 0.343/0.108 = 3.18 vezes. Isso acontece porque, da segunda forma, o vetor `out` precisa ter seu tamanho aumentado com um elemento a cada iteração. Para fazer isso, o R precisa encontrar um espaço na memória que possa armazenar o objeto maior. É necessário então copiar o vetor de saída e apagar sua versão anterior antes de seguir para o próximo loop. Ao final, foi necessário escrever um milhão de vezes na memória do computador.

Já no primeiro caso, o tamanho do vetor de armazenamento nunca muda, e a memória para esse vetor já foi alocada previamente, de uma única vez.



Voltando ao exemplo das notas, por exemplo, o cálculo da média simples poderia ser feita diretamente com a função `apply()`

```
notas$media.apply <- apply(X = notas[, provas], MARGIN = 1, FUN = mean)
head(notas)
```

```
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2  situacao media.apply
1 4.333333 reprovado    4.333333
2 5.000000 reprovado    5.000000
3 5.000000 reprovado    5.000000
4 6.666667 reprovado    6.666667
5 7.000000  aprovado    7.000000
6 4.333333 reprovado    4.333333
```

As médias ponderadas poderiam ser calculadas da mesma forma, e usando a função que criamos anteriormente

```
notas$MP.apply <- apply(X = notas[, provas], MARGIN = 1, FUN = med.pond)
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2  situacao media.apply MP.apply
1 4.333333 reprovado    4.333333 4.333333
2 5.000000 reprovado    5.000000 5.000000
3 5.000000 reprovado    5.000000 5.000000
4 6.666667 reprovado    6.666667 6.666667
5 7.000000  aprovado    7.000000 7.000000
6 4.333333 reprovado    4.333333 4.333333
```

Mas note que como temos o argumento pesos especificado com um padrão, devemos alterar na própria função `apply()`

```
notas$MP.apply <- apply(X = notas[, provas], MARGIN = 1,
                        FUN = med.pond, pesos = c(3, 3, 4))
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2  situacao media.apply MP.apply
1 4.333333 reprovado    4.333333      4.0
2 5.000000 reprovado    5.000000      5.1
3 5.000000 reprovado    5.000000      4.9
4 6.666667 reprovado    6.666667      6.9
5 7.000000  aprovado    7.000000      7.1
6 4.333333 reprovado    4.333333      4.2
```

NOTA: veja que isso é possível devido à presença do argumento `...` na função `apply()`, que permite passar argumentos de outras funções dentro dela.

Também poderíamos usar a função `weighted.mean()` implementada no R

```
notas$MP2.apply <- apply(X = notas[, provas], MARGIN = 1,
                         FUN = weighted.mean, w = c(3, 3, 4))
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2  situacao media.apply MP.apply MP2.apply
1 4.333333 reprovado    4.333333      4.0       4.0
2 5.000000 reprovado    5.000000      5.1       5.1
3 5.000000 reprovado    5.000000      4.9       4.9
```

```
4 6.666667 reprovado    6.666667      6.9       6.9
5 7.000000  aprovado    7.000000      7.1       7.1
6 4.333333 reprovado    4.333333      4.2       4.2
```

O Coeficiente de Variação poderia ser calculado usando nossa função `cv()`

```
notas$CV.apply <- apply(X = notas[, provas], MARGIN = 1, FUN = cv)
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2  situacao media.apply MP.apply MP2.apply  CV.apply
1 4.333333 reprovado    4.333333      4.0       4.0 0.8104349
2 5.000000 reprovado    5.000000      5.1       5.1 0.5291503
3 5.000000 reprovado    5.000000      4.9       4.9 0.7211103
4 6.666667 reprovado    6.666667      6.9       6.9 0.7399324
5 7.000000  aprovado    7.000000      7.1       7.1 0.1428571
6 4.333333 reprovado    4.333333      4.2       4.2 1.1842157
```

Finalmente, a estrutura de repetição `if()` também possui uma forma vetorizada através da função `ifelse()`. Essa função funciona da seguinte forma:

```
ifelse(<condição>, <valor se verdadeiro>, <valor se falso>)
```

Dessa forma, a atribuição da situação dos alunos poderia ser feita da seguinte forma:

```
notas$situacao2 <- ifelse(notas$MP >= 7, "aprovado", "reprovado")
head(notas)
     nome prova1 prova2 prova3    media   media2   media3        CV  MP
1 Aluno_1      8      4      1 4.333333 4.333333 4.333333 0.8104349 4.0
2 Aluno_2      2      7      6 5.000000 5.000000 5.000000 0.5291503 5.1
3 Aluno_3      9      2      4 5.000000 5.000000 5.000000 0.7211103 4.9
4 Aluno_4      1     10      9 6.666667 6.666667 6.666667 0.7399324 6.9
5 Aluno_5      7      6      8 7.000000 7.000000 7.000000 0.1428571 7.1
6 Aluno_6     10      0      3 4.333333 4.333333 4.333333 1.1842157 4.2
       MP2  situacao media.apply MP.apply MP2.apply  CV.apply situacao2
1 4.333333 reprovado    4.333333      4.0       4.0 0.8104349 reprovado
2 5.000000 reprovado    5.000000      5.1       5.1 0.5291503 reprovado
3 5.000000 reprovado    5.000000      4.9       4.9 0.7211103 reprovado
4 6.666667 reprovado    6.666667      6.9       6.9 0.7399324 reprovado
5 7.000000  aprovado    7.000000      7.1       7.1 0.1428571  aprovado
6 4.333333 reprovado    4.333333      4.2       4.2 1.1842157 reprovado
```

5.4 Outras estruturas: while e repeat

O `while` executa comandos enquanto uma determinada condição permanece verdadeira.

```
## Calcule a soma em 1,2,3... até que o soma seja maior do que 1000
n <- 0
soma <- 0
while(soma <= 1000){
    n <- n + 1
```

```
    soma <- soma + n
}
soma
[1] 1035
```

O `repeat` é ainda mais básico, e irá executar comandos até que você explicitamente pare a execução com o comando `break`.

```
## Mesmo exemplo
n <- 0
soma <- 0
repeat{
    n <- n + 1
    soma <- soma + n
    if(soma > 1000) break
}
soma
[1] 1035
```

Parte II

Estatística

Capítulo 6

Análise exploratória de dados

Nesta sessão vamos ver alguns (mas não todos!) comandos do R para fazer uma análise exploratória descritiva de um conjunto de dados.

Uma boa forma de iniciar uma análise descritiva adequada é verificar os tipos de de variáveis disponíveis. Variáveis podem ser classificadas da seguinte forma:

- Qualitativas
  - Nominais
  - Ordinais
- Quantitativas
  - Discretas
  - Contínuas

e podem ser resumidas por tabelas, gráficos e/ou medidas.

6.1 O conjunto de dados `milsa`

O livro "Estatística Básica" de W. O. Bussab e P. A. Morettin traz no segundo capítulo um conjunto de dados hipotético de atributos de 36 funcionários da companhia "Milsa". Os dados estão reproduzidos na tabela abaixo. Consulte o livro para mais detalhes sobre este dados.

Funcionario	Est.civil	Inst	Filhos	Salario	Anos	Meses	Regiao
1	solteiro	1o Grau	NA	4.00	26	3	interior
2	casado	1o Grau	1	4.56	32	10	capital
3	casado	1o Grau	2	5.25	36	5	capital
4	solteiro	2o Grau	NA	5.73	20	10	outro
5	solteiro	1o Grau	NA	6.26	40	7	outro
6	casado	1o Grau	0	6.66	28	0	interior
7	solteiro	1o Grau	NA	6.86	41	0	interior
8	solteiro	1o Grau	NA	7.39	43	4	capital
9	casado	2o Grau	1	7.59	34	10	capital
10	solteiro	2o Grau	NA	7.44	23	6	outro
11	casado	2o Grau	2	8.12	33	6	interior
12	solteiro	1o Grau	NA	8.46	27	11	capital
13	solteiro	2o Grau	NA	8.74	37	5	outro
14	casado	1o Grau	3	8.95	44	2	outro
15	casado	2o Grau	0	9.13	30	5	interior
16	solteiro	2o Grau	NA	9.35	38	8	outro
17	casado	2o Grau	1	9.77	31	7	capital
18	casado	1o Grau	2	9.80	39	7	outro
19	solteiro	Superior	NA	10.53	25	8	interior
20	solteiro	2o Grau	NA	10.76	37	4	interior
21	casado	2o Grau	1	11.06	30	9	outro
22	solteiro	2o Grau	NA	11.59	34	2	capital
23	solteiro	1o Grau	NA	12.00	41	0	outro
24	casado	Superior	0	12.79	26	1	outro
25	casado	2o Grau	2	13.23	32	5	interior
26	casado	2o Grau	2	13.60	35	0	outro
27	solteiro	1o Grau	NA	13.85	46	7	outro
28	casado	2o Grau	0	14.69	29	8	interior
29	casado	2o Grau	5	14.71	40	6	interior
30	casado	2o Grau	2	15.99	35	10	capital
31	solteiro	Superior	NA	16.22	31	5	outro
32	casado	2o Grau	1	16.61	36	4	interior
33	casado	Superior	3	17.26	43	7	capital
34	solteiro	Superior	NA	18.75	33	7	capital
35	casado	2o Grau	2	19.40	48	11	capital
36	casado	Superior	3	23.30	42	2	interior

Estes dados estão disponíveis em um arquivo csv no endereço http://www.leg.ufpr.br/~fernandomayer/data/milsa.csv.

O nosso objetivo é, através do R,

- Entrar com os dados
- Fazer uma análise descritiva

Estes são dados no "estilo planilha", com variáveis de diferentes tipos: categóricas e numéricas (qualitativas e quantitativas). Portanto o formato ideal de armazenamento destes dados no R é o `data.frame`.

Para importar os dados do endereço acima diretamente para o R, usamos

```
url <- "http://www.leg.ufpr.br/~fernandomayer/data/milsa.csv"
milsa <- read.csv(url)
```

E para conferir a estrutura dos dados podemos usar algumas funções como:

```
str(milsa)
'data.frame':   36 obs. of  8 variables:
 $ Funcionario: int  1 2 3 4 5 6 7 8 9 10 ...
 $ Est.civil  : Factor w/ 2 levels "casado","solteiro": 2 1 1 2 2 1 2 2 1 2 ...
 $ Inst       : Factor w/ 3 levels "1o Grau","2o Grau",..: 1 1 1 2 1 1 1 1 2 2 ...
 $ Filhos     : int  NA 1 2 NA NA 0 NA NA 1 NA ...
 $ Salario    : num  4 4.56 5.25 5.73 6.26 6.66 6.86 7.39 7.59 7.44 ...
 $ Anos       : int  26 32 36 20 40 28 41 43 34 23 ...
 $ Meses      : int  3 10 5 10 7 0 0 4 10 6 ...
 $ Regiao     : Factor w/ 3 levels "capital","interior",..: 2 1 1 3 3 2 2 1 1 3 ...
head(milsa)
  Funcionario Est.civil    Inst Filhos Salario Anos Meses   Regiao
1           1  solteiro 1o Grau     NA    4.00   26     3 interior
2           2    casado 1o Grau      1    4.56   32    10  capital
3           3    casado 1o Grau      2    5.25   36     5  capital
4           4  solteiro 2o Grau     NA    5.73   20    10    outro
5           5  solteiro 1o Grau     NA    6.26   40     7    outro
6           6    casado 1o Grau      0    6.66   28     0 interior
tail(milsa)
   Funcionario Est.civil     Inst Filhos Salario Anos Meses   Regiao
31          31  solteiro Superior     NA   16.22   31     5    outro
32          32    casado  2o Grau      1   16.61   36     4 interior
33          33    casado Superior      3   17.26   43     7  capital
34          34  solteiro Superior     NA   18.75   33     7  capital
35          35    casado  2o Grau      2   19.40   48    11  capital
36          36    casado Superior      3   23.30   42     2 interior
```

Podemos classificar todas a svariáveis desse conjunto de dados como:

Variável	Classificação
`Funcionario`	Quantitativa discreta
`Est.civil`	Qualitativa nominal
`Inst`	Qualitativa ordinal
`Filhos`	Quantitativa discreta
`Salario`	Quantitativa contínua
`Anos`	Quantitativa contínua
`Meses`	Quantitativa contínua
`Regiao`	Qualitativa nominal

Como a variável `Inst` é qualitativa ordinal, podemos indicar para o R que ela deve ser tratada como ordinal. Se observarmos os níveis desse fator:

```
levels(milsa$Inst)
[1] "1o Grau"  "2o Grau"  "Superior"
```

já notamos que a ordenação está correta (da esquerda para a direita), pois sabemos que a classificação interna dos níveis é por ordem alfabética, e nesse caso, por coincidência, a ordem já está na sequência correta. Mesmo assim, podemos indicar que este fator é ordinal, usando o argumento `ordered` da função `factor()`

```
milsa$Inst <- factor(milsa$Inst, ordered = TRUE)
```

Note agora a modificação na classe dessa coluna, e a representação dos níveis:

```
class(milsa$Inst)
[1] "ordered" "factor"
```

```
milsa$Inst
 [1] 1o Grau  1o Grau  1o Grau  2o Grau  1o Grau  1o Grau  1o Grau
 [8] 1o Grau  2o Grau  2o Grau  2o Grau  1o Grau  2o Grau  1o Grau
[15] 2o Grau  2o Grau  2o Grau  1o Grau  Superior 2o Grau  2o Grau
[22] 2o Grau  1o Grau  Superior 2o Grau  2o Grau  1o Grau  2o Grau
[29] 2o Grau  2o Grau  Superior 2o Grau  Superior Superior 2o Grau
[36] Superior
Levels: 1o Grau < 2o Grau < Superior
```

A coluna continua sendo um `factor`, mas agora também é `ordered` (sim, um objeto pode ter mais de uma classe, se elas foram compatíveis e/ou complementares). Os níveis agora são representados por

```
1o Grau < 2o Grau < Superior
```

para indicar explicitamente que existe uma ordem nos níveis desse fator.

Podemos ainda definir uma nova variável, chamada `Idade`, a partir das variáveis `Anos` e `Meses`:

```
milsa$Idade <- milsa$Anos + milsa$Meses/12
```

Os dois comandos acima (de modificação da classe de uma variável, e a criação de uma nova variável) poderiam ser facilmente executadas de uma única vez através do comando `transform()`

```
milsa <- transform(milsa,
                   Inst = factor(Inst, ordered = TRUE),
                   Idade = Anos + Meses/12)
```

Agora que os dados estão prontos podemos começar a análise descritiva. A seguir mostramos como fazer análises descritivas uni e bi-variadas. Inspecione os comandos mostrados a seguir e os resultados por elas produzidos. Sugerimos ainda que o leitor use o R para reproduzir os resultados mostrados no texto dos capítulos 1 a 3 do livro de Bussab & Morettin, relacionados com este exemplo. Veja os scripts do livro aqui.

6.2 Análise univariada

A análise univariada consiste basicamente em, para cada uma das variáveis individualmente:

- Classificar a variável quanto a seu tipo: qualitativa (nominal ou ordinal) ou quantitativa (discreta ou contínua)
- Obter tabelas, gráficos e/ou medidas que resumam a variável

A partir destes resultados pode-se montar um resumo geral dos dados.

A seguir vamos mostrar como obter tabelas, gráficos e medidas com o R. Para isto vamos selecionar uma variável de cada tipo para que o leitor possa, por analogia, obter resultados para as demais.

6.2.1 Variável Qualitativa Nominal

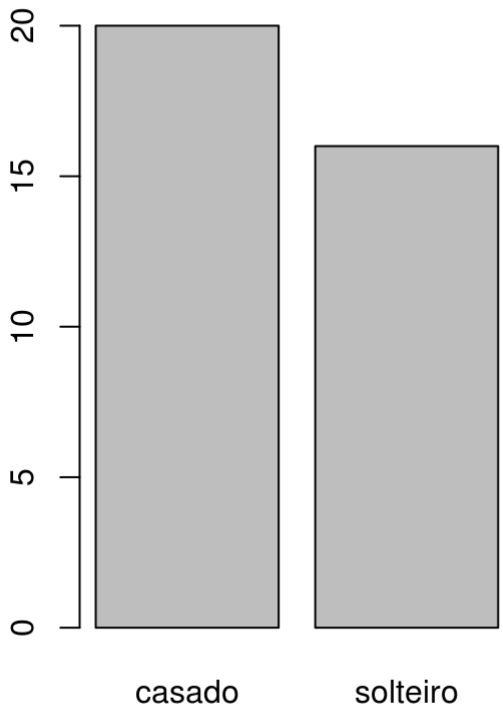
A variável `Est.civil` é uma qualitativa nominal. Desta forma podemos obter: (i) uma tabela de frequências (absolutas e/ou relativas), (ii) um gráfico de setores, (iii) a "moda", *i.e.* o valor que ocorre com maior frequência.

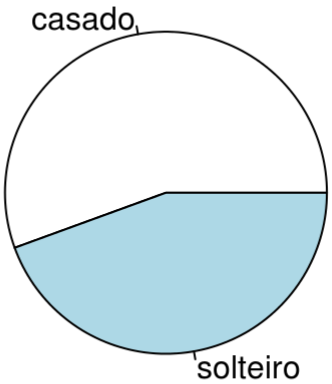
Já vimos, através do resultado da função `str()` acima, que esta variável é um fator. A seguir obtemos frequências absolutas e relativas (note duas formas diferentes de obter as frequências relativas).

```
## Frequência absoluta
civil.tb <- table(milsa$Est.civil)
civil.tb

  casado solteiro
      20       16
## Frequência relativa, calculando manualmente
civil.tb/length(milsa$Est.civil)

   casado  solteiro
0.5555556 0.4444444
## Frequência relativa, com a função prop.table()
prop.table(civil.tb)

   casado  solteiro
0.5555556 0.4444444
```

Os gráficos de barras e de setores são adequados para representar esta variável. Os comandos `barplot()` e `pie()` usam o resultado da função `table()` para gerar os gráficos:

```
par(mfrow = c(1,2))
barplot(civil.tb)
pie(civil.tb)
par(mfrow = c(1,1))
```





A *moda* de qualquer variável aleatória é definida como o valor mais frequente encontrado na amostra. No R não há uma função pronta para "calcular" a moda, pois ela pode ser obtida facilmente através do uso de funções básicas. Uma opção seria usar os comandos abaixo:

```
names(civil.tb)[which.max(civil.tb)]
[1] "casado"
```

Deixamos a cargo do leitor entender e interpretar esse comando.

6.2.2 Variável Qualitativa Ordinal

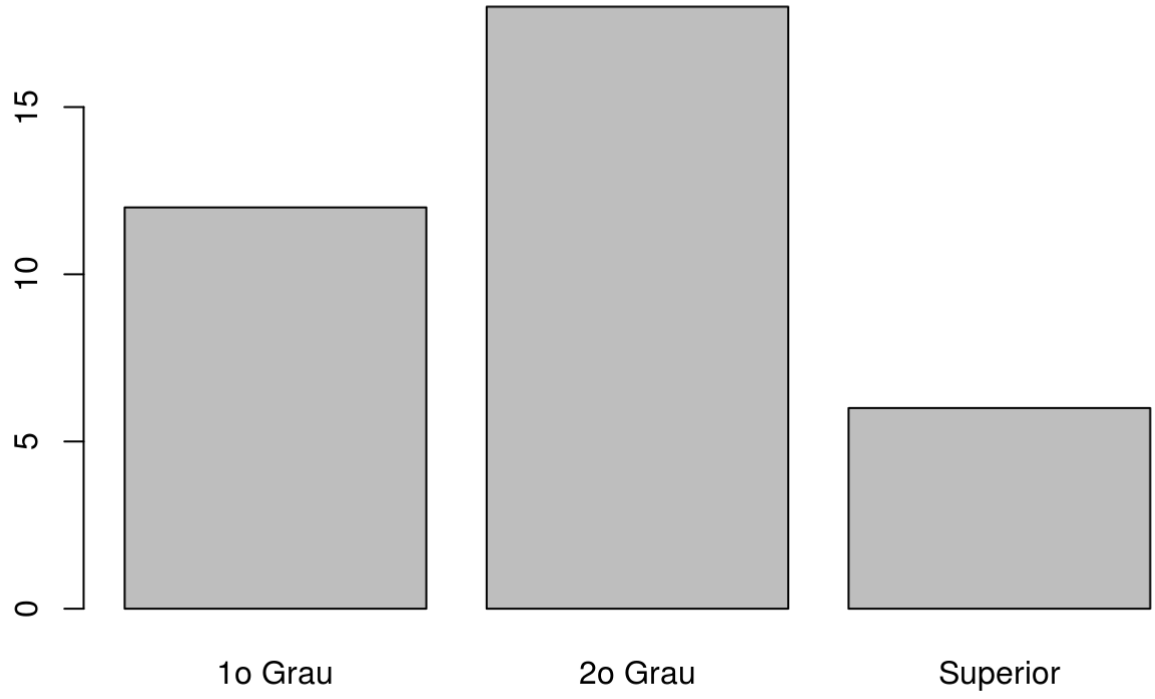
Para exemplificar como obter análises para uma variável qualitativa ordinal vamos selecionar a variável `Inst`.

As tabelas de frequências são obtidas de forma semelhante à mostrada anteriormente.

```
## Frequência absoluta
inst.tb <- table(milsa$Inst)
inst.tb

 1o Grau  2o Grau Superior
      12       18        6
## Frequência relativa
prop.table(inst.tb)

  1o Grau   2o Grau  Superior
0.3333333 0.5000000 0.1666667
```

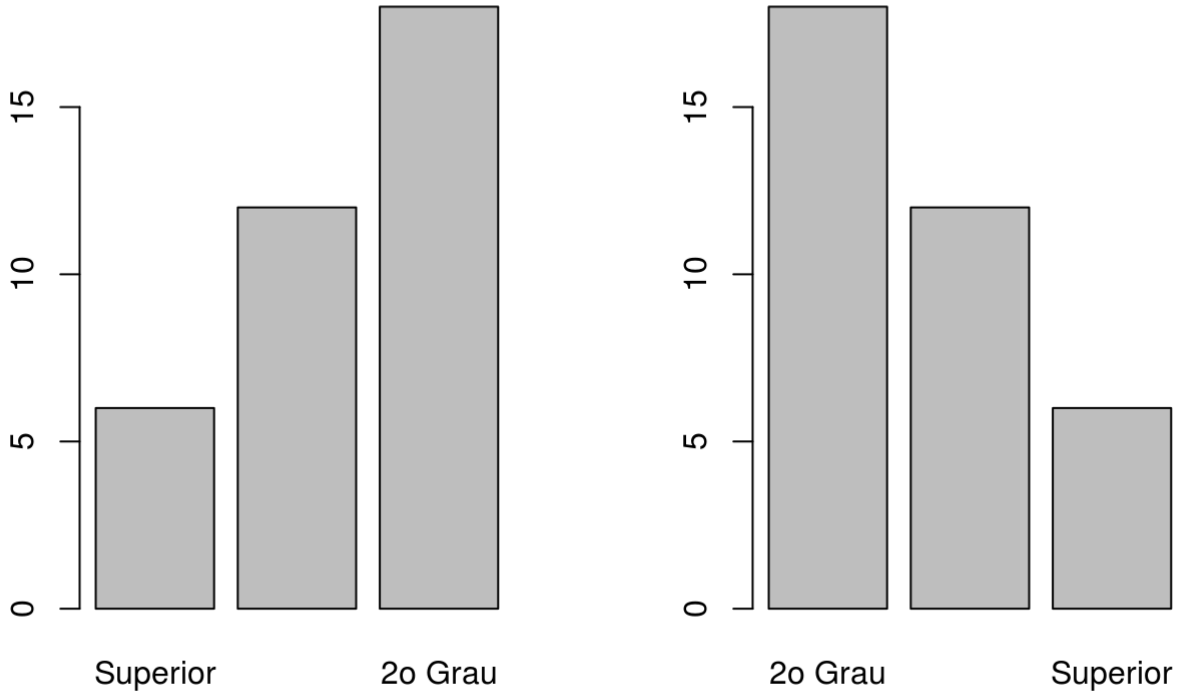
O gráfico de setores não é adequado para este tipo de variável por não expressar a ordem dos possíveis valores. Usamos então apenas um gráfico de barras conforme mostrado abaixo

```
barplot(inst.tb)
```



Em alguns casos podemos querer mostrar o gráfico de barras com as barras classificadas da menor para a maior, ou vice-versa, independente da ordem dos níveis. Para isso podemos usar a função `sort()` para ordenar os valores da tabela e fazer o gráfico

```
par(mfrow = c(1,2))
## Menor para maior
barplot(sort(inst.tb))
## Maior para menor
barplot(sort(inst.tb, decreasing = TRUE))
par(mfrow = c(1,1))
```



Para uma variável ordinal, além da moda podemos também calcular outras medidas, tais como a mediana conforme exemplificado a seguir. Note que o comando `median()` não funciona com variáveis não numéricas, e por isso usamos o comando seguinte.

```
## Moda
names(inst.tb)[which.max(inst.tb)]
[1] "2o Grau"
## Mediana
median(milsa$Inst) # só funciona para variáveis numéricas
Error in median.default(milsa$Inst): need numeric data
median(as.numeric(milsa$Inst)) # traz a mediana da codificação do nível
[1] 2
levels(milsa$Inst)[median(as.numeric(milsa$Inst))] # valor correto
[1] "2o Grau"
```

6.2.3 Variável quantitativa discreta

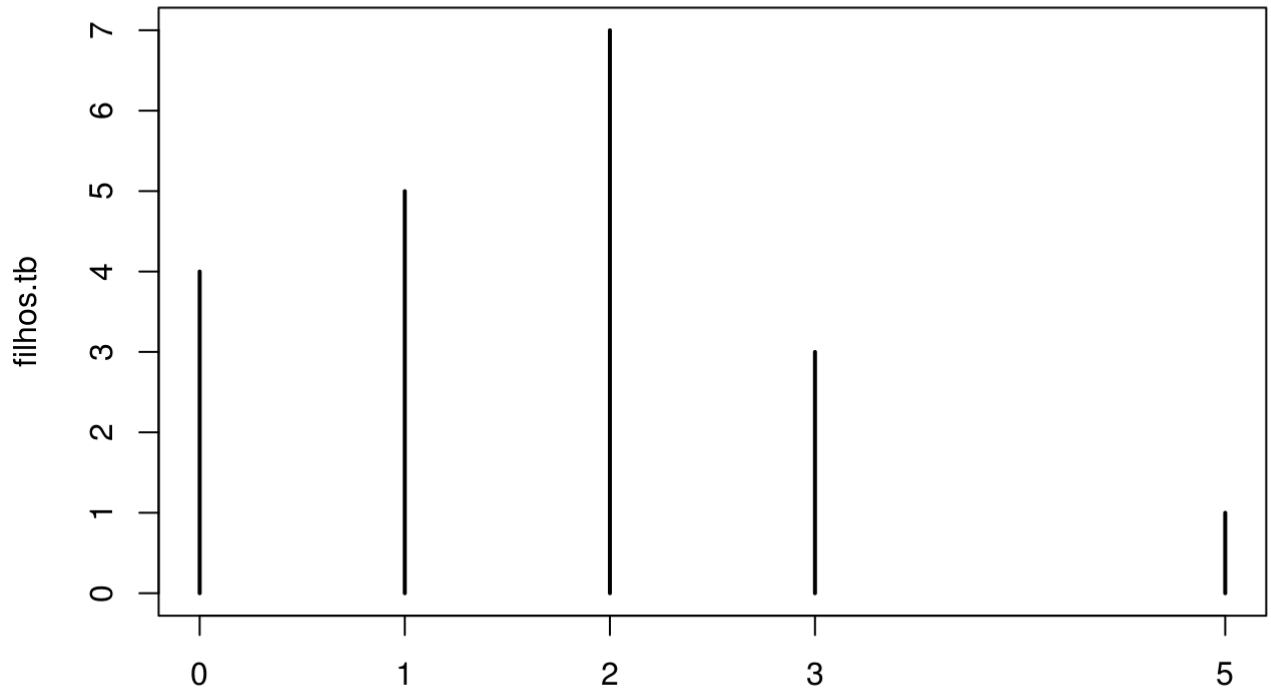
Vamos agora usar a variável `Filhos` (número de filhos) para ilustrar algumas análises que podem ser feitas com uma quantitativa discreta.

Frequências absolutas e relativas são obtidas como anteriormente. Também vamos calcular a frequência acumulada, onde a frequência em uma classe é a soma das frequências das classes anteriores. Para isso usamos a função `cumsum()`, que já faz a soma acumulada.

```
## Frequência absoluta
filhos.tb <- table(milsa$Filhos)
filhos.tb

0 1 2 3 5
4 5 7 3 1
## Frequência relativa
filhos.tbr <- prop.table(filhos.tb)
filhos.tbr

   0    1    2    3    5
0.20 0.25 0.35 0.15 0.05
## Frequência acumulada
filhos.tba <- cumsum(filhos.tbr)
filhos.tba
   0    1    2    3    5
0.20 0.45 0.80 0.95 1.00
```

O gráfico adequado para frequências absolutas de uma variável discreta é parecido com um gráfico de barras, mas nesse caso, as frequências são indicadas por linhas. Usando a função `plot()` em um objeto resultado da função `table()`, o gráfico adequado já é selecionado:

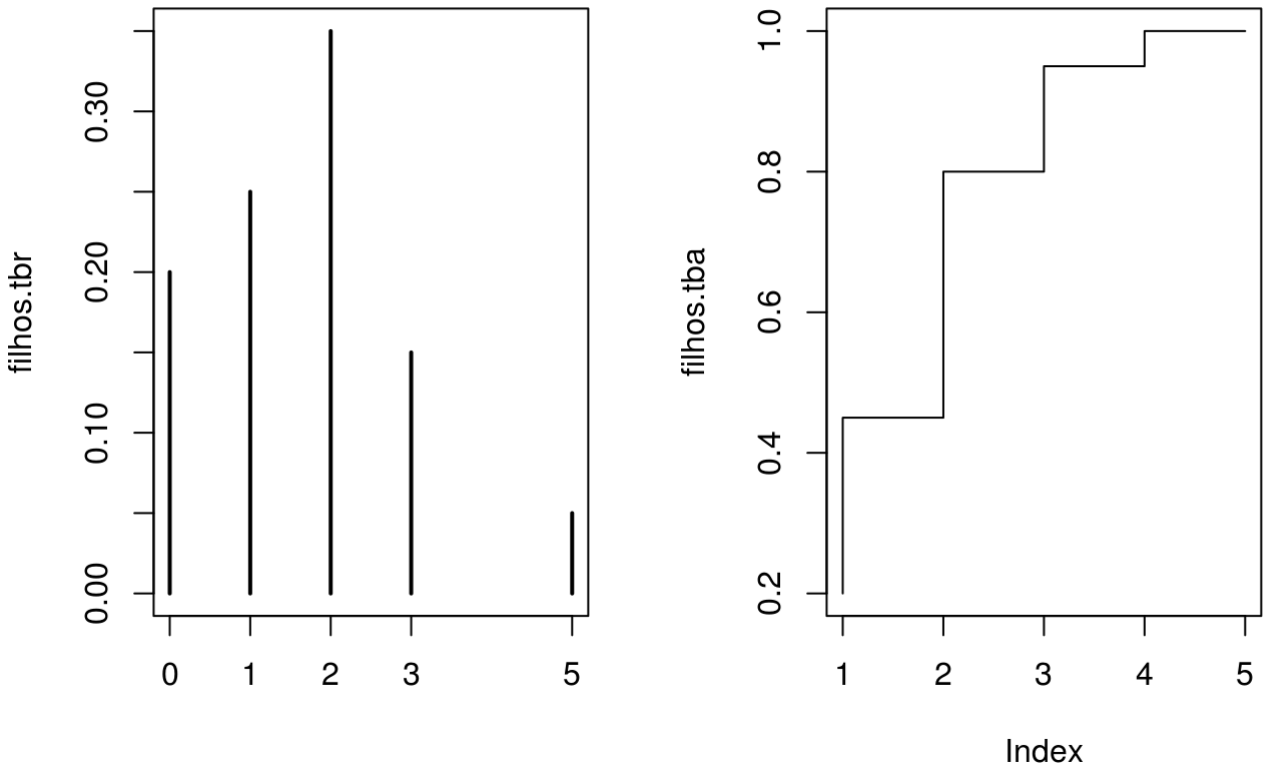
```
plot(filhos.tb)
```



Outra possibilidade seria fazer gráficos de frequências relativas e de frequências acumuladas

conforme mostrado na

```
par(mfrow = c(1,2))
## Frequência relativa
plot(filhos.tbr)
## Frequência relativa acumulada
plot(filhos.tba, type = "S") # tipo step (escada)
par(mfrow = c(1,1))
```



Sendo a variável numérica há uma maior diversidade de medidas estatísticas que podem ser calculadas.

A seguir mostramos como obter algumas medidas de posição: moda, mediana, média e média aparada. Note que o argumento `na.rm = TRUE` é necessário porque não há informação sobre número de filhos para alguns indivíduos (`NA`). Para calcular a média aparada, usamos o argumento `trim = 0.1` que indica que a média deve ser calculada excluindo-se 10% dos menores e 10% dos maiores valores do vetor de dados. Ao final mostramos como obter os quartis, incluido o mínimo e o máximo.

```
## Moda
names(filhos.tb)[which.max(filhos.tb)]
[1] "2"
## Mediana
median(milsa$Filhos, na.rm = TRUE)
[1] 2
## Média
```

```
mean(milsa$Filhos, na.rm = TRUE)
[1] 1.65
## Média aparada
mean(milsa$Filhos, trim = 0.1, na.rm = TRUE)
[1] 1.5625
## Quartis
quantile(milsa$Filhos, na.rm = TRUE)
  0%  25%  50%  75% 100%
   0    1    2    2    5
```

Passando agora para medidas de dispersão, vejamos como obter máximo e mínimo, e com isso a amplitude, além da variância, desvio padrão, e coeficiente de variação. Também obtemos os quartis para calcular a amplitude interquartílica.

```
## Máximo e mínimo
max(milsa$Filhos, na.rm = TRUE)
[1] 5
min(milsa$Filhos, na.rm = TRUE)
[1] 0
## As duas informações juntas
range(milsa$Filhos, na.rm = TRUE)
[1] 0 5
## Amplitude é a diferença entre máximo e mínimo
diff(range(milsa$Filhos, na.rm = TRUE))
[1] 5
## Variância
var(milsa$Filhos, na.rm = TRUE)
[1] 1.607895
## Desvio-padrão
sd(milsa$Filhos, na.rm = TRUE)
[1] 1.268028
## Coeficiente de variação
sd(milsa$Filhos, na.rm = TRUE)/mean(milsa$Filhos, na.rm = TRUE)
[1] 0.7685018
## Quartis
(filhos.qt <- quantile(milsa$Filhos, na.rm = TRUE))
  0%  25%  50%  75% 100%
   0    1    2    2    5
## Amplitude interquartílica
filhos.qt[4] - filhos.qt[2]
75%
  1
```

Finalmente, podemos usar a função **genérica** `summary()` para resumir od dados de uma só vez

```
summary(milsa$Filhos)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00    1.00    2.00    1.65    2.00    5.00      16
```

6.2.4 Variável quantitativa contínua

Para concluir os exemplos para análise univariada vamos considerar a variável quantitativa contínua `Salario`.

Para se fazer uma tabela de frequências de uma VA contínua, é preciso primeiro agrupar os dados em classes. Nos comandos mostrados a seguir verificamos inicialmente os valores máximo e mínimo dos dados, depois usamos o critério de Sturges para definir o número de classes. Usamos

a função `cut()` para agrupar os dados em classes e finalmente obtemos as frequências absolutas e relativas.

```
## Máximo e mínimo
range(milsa$Salario)
[1]  4.0 23.3
## Número de classes estimado, com base no critério de Sturges. Veja
## outras opções em ?nclass
nclass.Sturges(milsa$Salario)
[1] 7
## Criando as classes com a função cut(), usando os valores mínimos e
## máximos dados em range()
cut(milsa$Salario, breaks = seq(4, 23.3, length.out = 8))
 [1] <NA>        (4,6.76]    (4,6.76]    (4,6.76]    (4,6.76]
 [6] (4,6.76]    (6.76,9.51] (6.76,9.51] (6.76,9.51] (6.76,9.51]
[11] (6.76,9.51] (6.76,9.51] (6.76,9.51] (6.76,9.51] (6.76,9.51]
[16] (6.76,9.51] (9.51,12.3] (9.51,12.3] (9.51,12.3] (9.51,12.3]
[21] (9.51,12.3] (9.51,12.3] (9.51,12.3] (12.3,15]   (12.3,15]
[26] (12.3,15]   (12.3,15]   (12.3,15]   (12.3,15]   (15,17.8]
[31] (15,17.8]   (15,17.8]   (15,17.8]   (17.8,20.5] (17.8,20.5]
[36] (20.5,23.3]
7 Levels: (4,6.76] (6.76,9.51] (9.51,12.3] (12.3,15] ... (20.5,23.3]
```

Note que uma das classes é `NA`. Isso ocorre pela definição das classes, que por padrão é no formato `(a,b]`, ou seja, o intervalo é aberto em a (não inclui a) e fechado em b (inclui b). Podemos alterar esse padrão usando o argumento `include.lowest = TRUE`,

```
cut(milsa$Salario, breaks = seq(4, 23.3, length.out = 8),
    include.lowest = TRUE)
 [1] [4,6.76]    [4,6.76]    [4,6.76]    [4,6.76]    [4,6.76]
 [6] [4,6.76]    (6.76,9.51] (6.76,9.51] (6.76,9.51] (6.76,9.51]
[11] (6.76,9.51] (6.76,9.51] (6.76,9.51] (6.76,9.51] (6.76,9.51]
[16] (6.76,9.51] (9.51,12.3] (9.51,12.3] (9.51,12.3] (9.51,12.3]
[21] (9.51,12.3] (9.51,12.3] (9.51,12.3] (12.3,15]   (12.3,15]
[26] (12.3,15]   (12.3,15]   (12.3,15]   (12.3,15]   (15,17.8]
[31] (15,17.8]   (15,17.8]   (15,17.8]   (17.8,20.5] (17.8,20.5]
[36] (20.5,23.3]
7 Levels: [4,6.76] (6.76,9.51] (9.51,12.3] (12.3,15] ... (20.5,23.3]
```

E note que agora a primeira classe fica `[a,b]`, ou seja, fechada (incluindo) os dois lados. Para que o intervalo seja fechado à esquerda, usamos o argumento `right = FALSE`. As combinações possíveis para esses dois argumentos, e as classes resultantes são apresentadas na tabela abaixo:

Argumentos	Resultado
`include.lowest = T, right = T`	`[a,b], ..., (y,z]`
`include.lowest = F, right = T`	`(a,b], ..., (y,z]`
`include.lowest = F, right = F`	`[a,b), ..., [y,z)`
`include.lowest = T, right = F`	`[a,b), ..., [y,z]`

Outra opção para "acomodar" todos os extremos dentro das classes, seria naturalmente atribuir valores um pouco menores que o mínimo, e um pouco maiores que o máximo. Abaixo, usamos essa abordagem e fazemos uma tabela com as frequências absolutas e relativas.

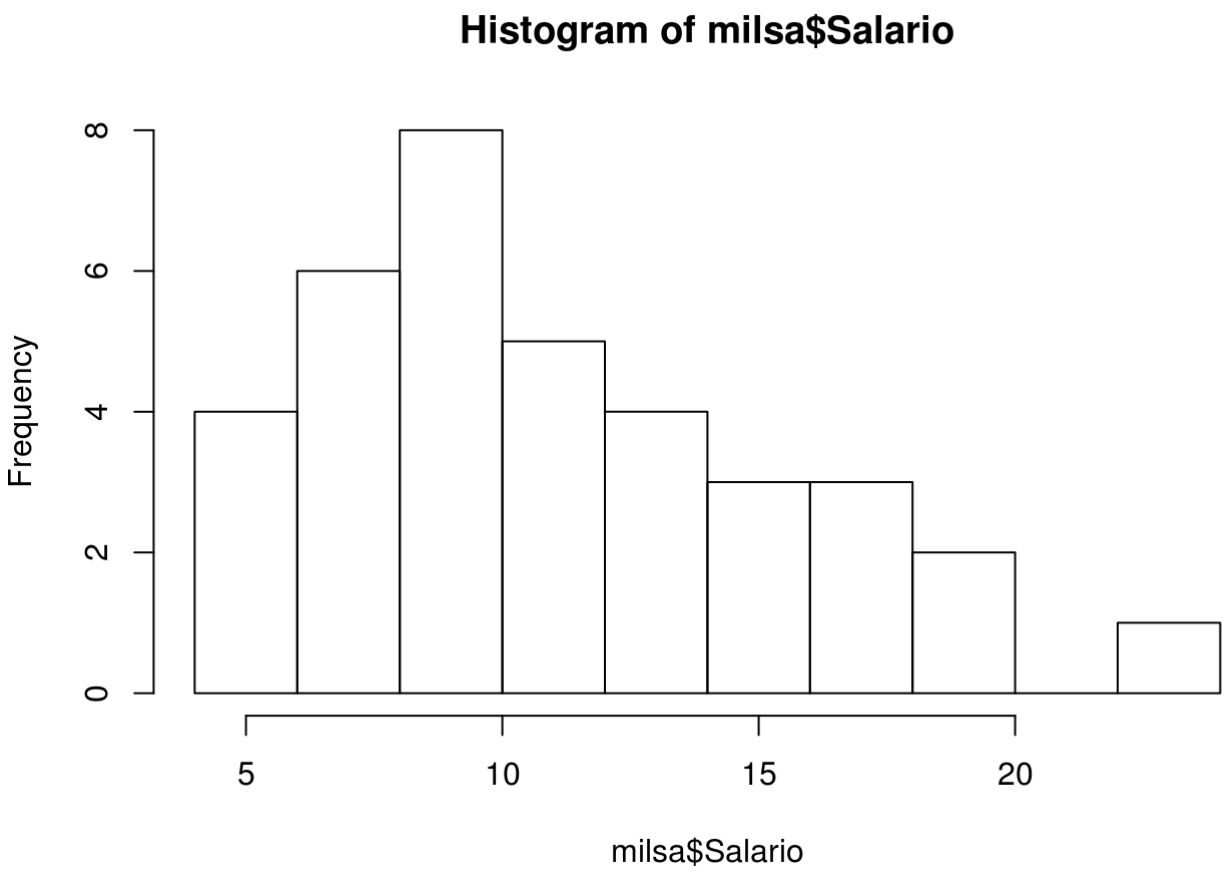
```
salario.cut <- cut(milsa$Salario,
                   breaks = seq(3.5, 23.5, length.out = 8))
salario.cut
```

```
 [1] (3.5,6.36]  (3.5,6.36]  (3.5,6.36]  (3.5,6.36]  (3.5,6.36]
 [6] (6.36,9.21] (6.36,9.21] (6.36,9.21] (6.36,9.21] (6.36,9.21]
[11] (6.36,9.21] (6.36,9.21] (6.36,9.21] (6.36,9.21] (6.36,9.21]
[16] (9.21,12.1] (9.21,12.1] (9.21,12.1] (9.21,12.1] (9.21,12.1]
[21] (9.21,12.1] (9.21,12.1] (9.21,12.1] (12.1,14.9] (12.1,14.9]
[26] (12.1,14.9] (12.1,14.9] (12.1,14.9] (12.1,14.9] (14.9,17.8]
[31] (14.9,17.8] (14.9,17.8] (14.9,17.8] (17.8,20.6] (17.8,20.6]
[36] (20.6,23.5]
7 Levels: (3.5,6.36] (6.36,9.21] (9.21,12.1] (12.1,14.9] ... (20.6,23.5]
## Tabela com as frequencias absolutas por classe
salario.tb <- table(salario.cut)
salario.tb
salario.cut
 (3.5,6.36] (6.36,9.21] (9.21,12.1] (12.1,14.9] (14.9,17.8] (17.8,20.6]
          5          10           8           6           4           2
(20.6,23.5]
          1
## Tabela com as frequências relativas
prop.table(salario.tb)
salario.cut
 (3.5,6.36] (6.36,9.21] (9.21,12.1] (12.1,14.9] (14.9,17.8] (17.8,20.6]
 0.13888889  0.27777778  0.22222222  0.16666667  0.11111111  0.05555556
(20.6,23.5]
 0.02777778
```

Na sequência vamos mostrar dois possíveis gráficos para variáveis contínuas: o histograma e o *box-plot*.

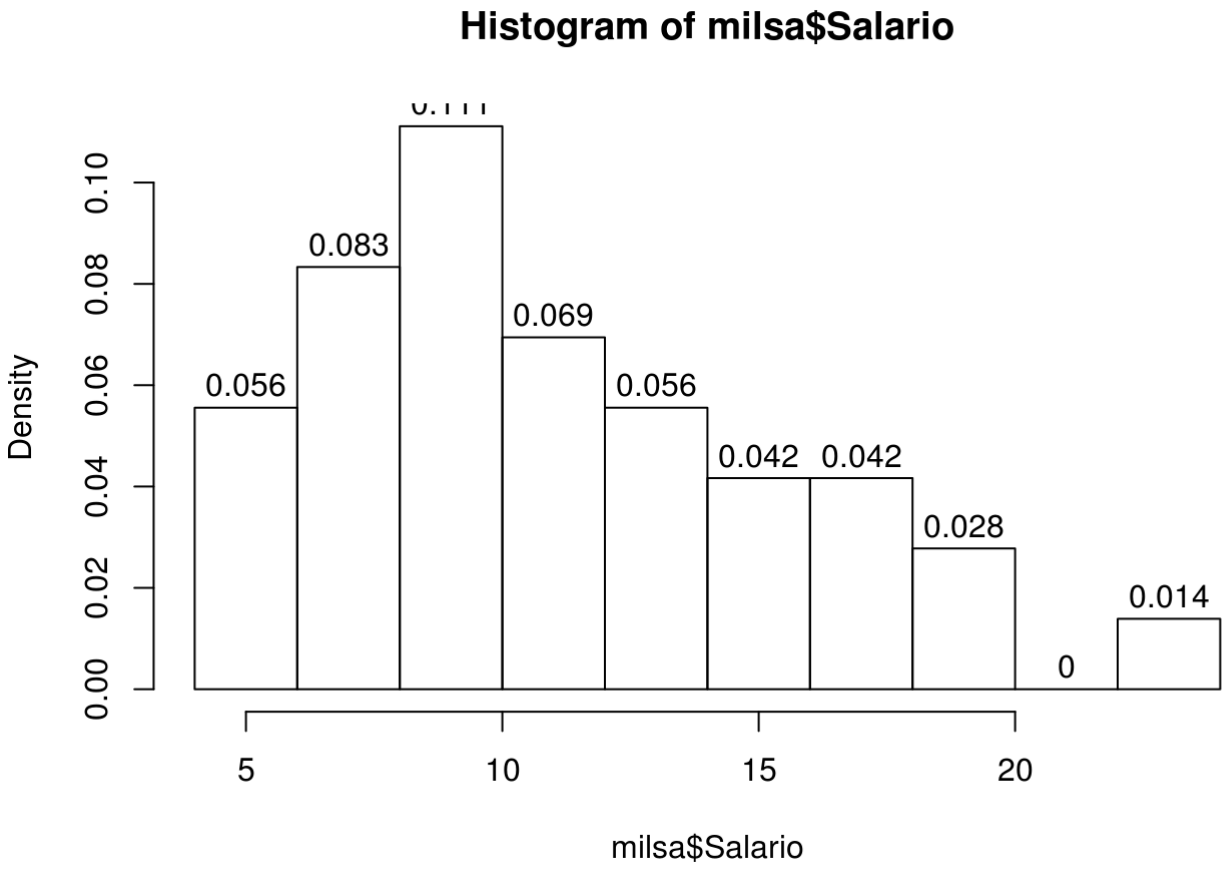
Para fazer um histograma usamos a função `hist()`, por exemplo,

```
hist(milsa$Salario)
```



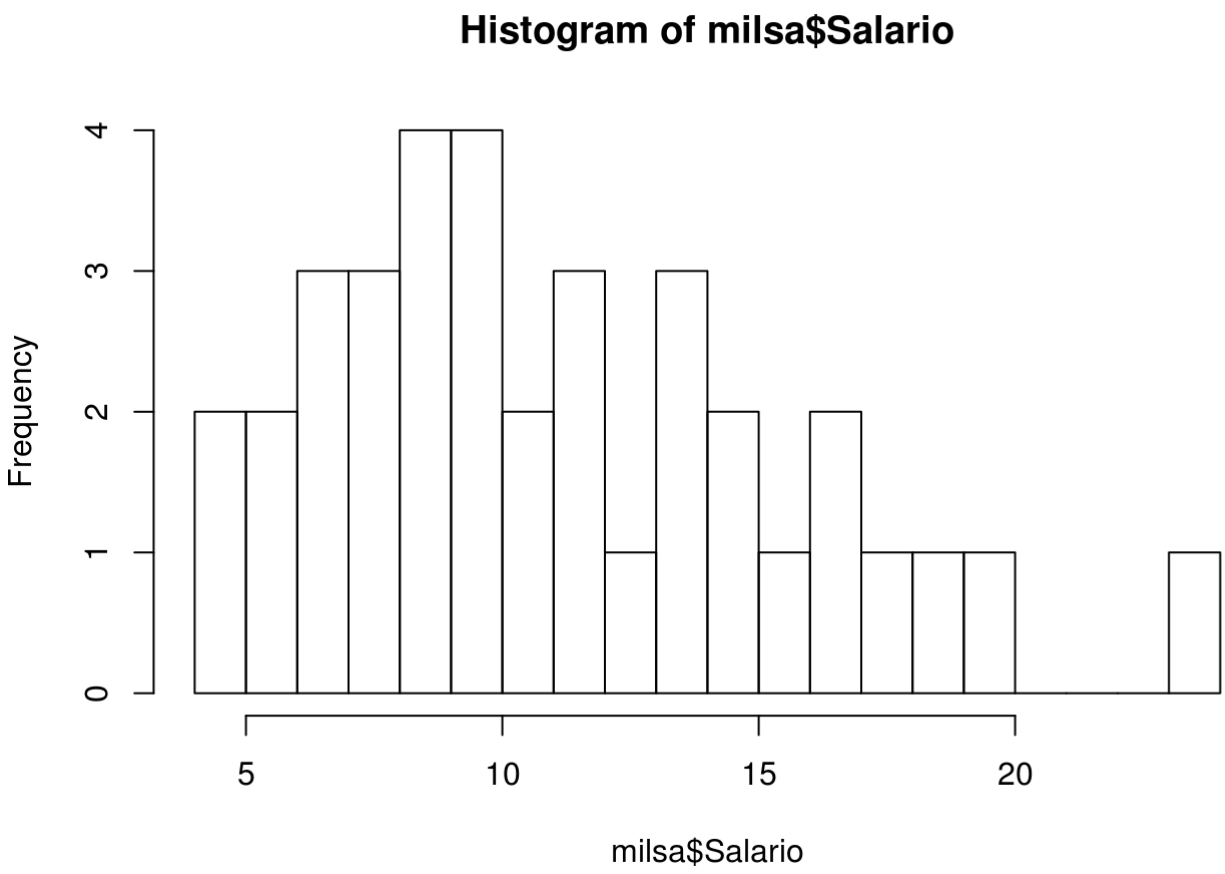
A função `hist()` possui vários argumentos para alterar o comportamento da saída do gráfico. Por exemplo, com `labels = TRUE` as frequências são mostradas acima de cada barra. Com `freq = FALSE`, o gráfico é feito com as frequências relativas.

```
hist(milsa$Salario, freq = FALSE, labels = TRUE)
```



Por padrão, a função `hist()` calcula automaticamente o número de classes e os valores limites de cada classe. No entanto, isto pode ser alterado com o argumento `breaks`, que pode receber um vetor definindo os limites das classes, uma função para definir as quebras, um nome de critério (por exemplo, `"Sturges"`), ou um único escalar definido o número de classes. As últimas três opções são apenas sugestões utilizadas pela função. O argumento `nclass` também funciona dessa forma, recebendo apenas um valor com o número de classes (como sugestão).

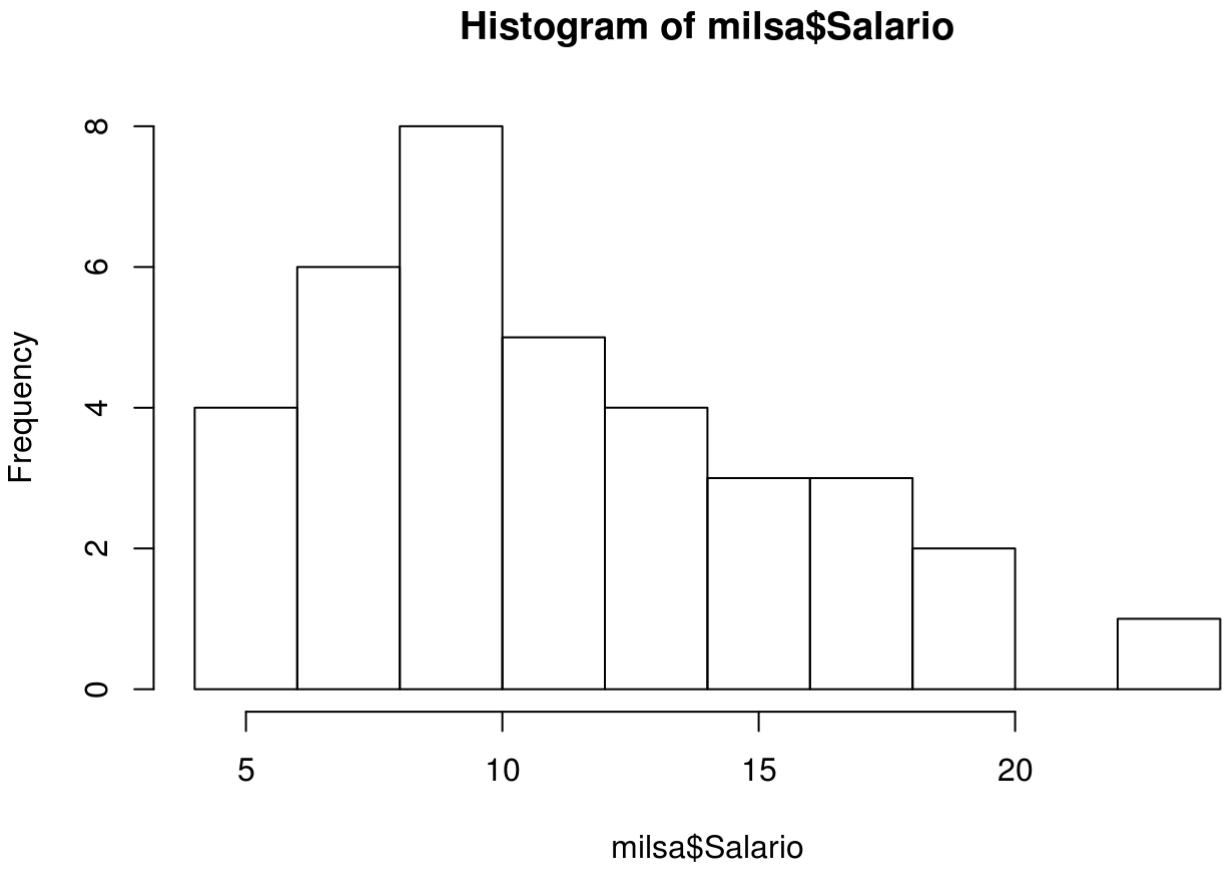
```
hist(milsa$Salario, nclass = 15)
```



Assim como na função `cut()`, os argumentos `include.lowest` e `right` são utilizados para controlar a borda das classes.

Uma característica importante da função `hist()` é que ela retorna não apenas o gráfico, mas também uma lista com as informações utilizadas para construir o gráfico. Associando um histograma a um objeto, podemos ver o seu conteúdo:

```
salario.hist <- hist(milsa$Salario)
```




```
salario.hist
$breaks
 [1]  4  6  8 10 12 14 16 18 20 22 24

$counts
 [1] 4 6 8 5 4 3 3 2 0 1

$density
 [1] 0.05555556 0.08333333 0.11111111 0.06944444 0.05555556 0.04166667
 [7] 0.04166667 0.02777778 0.00000000 0.01388889

$mids
 [1]  5  7  9 11 13 15 17 19 21 23

$xname
[1] "milsa$Salario"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
```

Estas informações podem então ser utilizadas para outros propósitos dentro do R.

Os **boxplots** são úteis para revelar o centro, a dispersão e a distribuição dos dados, além de **outliers**.

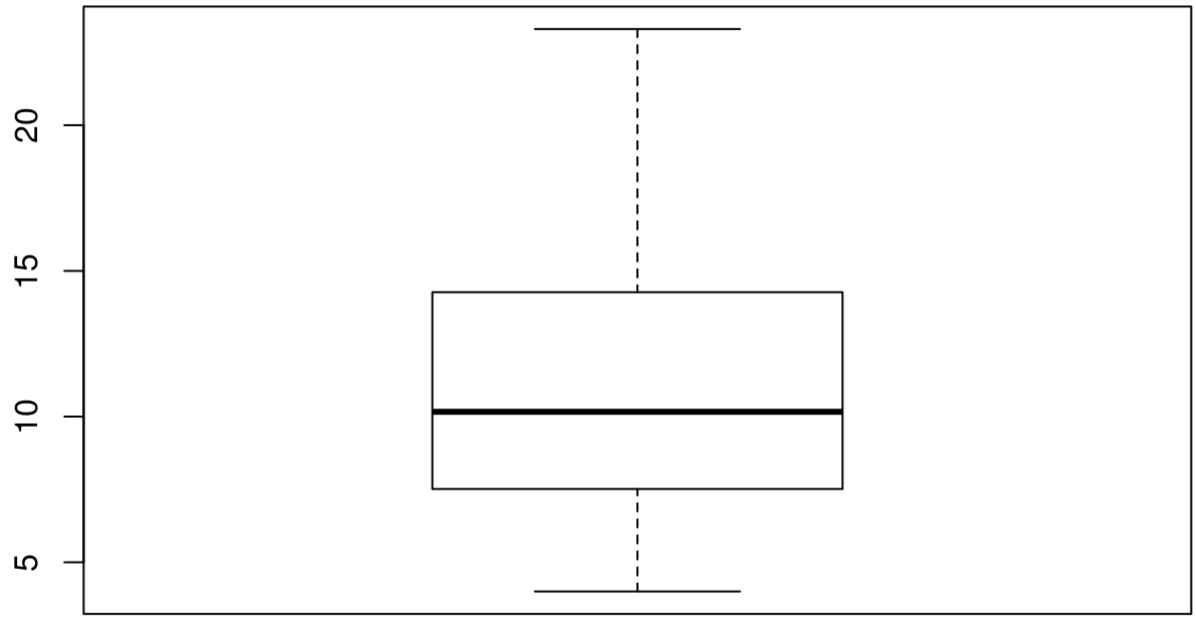
São construídos da seguinte forma:

- A linha central mais escura representa a mediana. Os extremos da caixa são o 1^o ($q1$) e o 3^o ($q3$) quartis.
- As linhas que se extendem das caixas são definidas como:

$$q1 - 1,5 \cdot IQR \qquad \text{e} \qquad q3 + 1,5 \cdot IQR$$

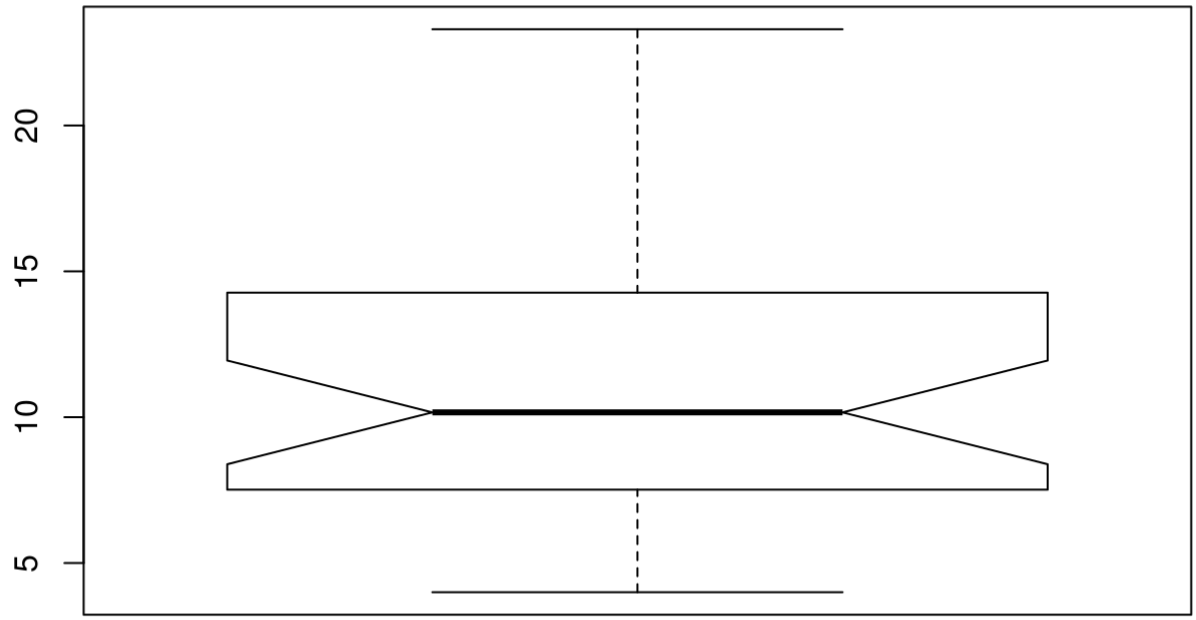
  onde IQR é o intervalo inter-quartil. As linhas vão até os valores máximo e mínimo que ainda se encontram dentro deste intervalo.

```
boxplot(milsa$Salario)
```



Existem também vários argumentos que permitem variações do *boxplot*, tais como caixas com tamanho proporcional aos tamanhos dos grupos (`varwidth = TRUE`), e caixas "acinturadas" (*notched boxplot*) (`notch = TRUE`).

```
boxplot(milsa$Salario, varwidth = TRUE, notch = TRUE)
```



Ambas opções são úteis quando há mais de um grupo e a comparação entre os boxplots é facilitada.

Finalmente, podemos obter as medidas de posição e dispersão da mesma forma que para variáveis discretas. Veja alguns exemplos a seguir. Note que aqui não é necessário o uso do argumento `na.rm = TRUE`, pois não existem `NA`s nesta variável.

```
## Mediana
median(milsa$Salario)
[1] 10.165
## Média
mean(milsa$Salario)
[1] 11.12222
## Média aparada
mean(milsa$Salario, trim = 0.1)
[1] 10.838
## Quartis
quantile(milsa$Salario)
     0%     25%     50%     75%    100%
 4.0000  7.5525 10.1650 14.0600 23.3000
## Máximo e mínimo
max(milsa$Salario)
[1] 23.3
min(milsa$Salario)
[1] 4
## As duas informações juntas
range(milsa$Salario)
```

```
[1]  4.0 23.3
## Amplitude é a diferença entre máximo e mínimo
diff(range(milsa$Salario))
[1] 19.3
## Variância
var(milsa$Salario)
[1] 21.04477
## Desvio-padrão
sd(milsa$Salario)
[1] 4.587458
## Coeficiente de variação
sd(milsa$Salario)/mean(milsa$Salario)
[1] 0.4124587
## Quartis
salario.qt <- quantile(milsa$Salario)
## Amplitude interquartílica
salario.qt[4] - salario.qt[2]
   75%
6.5075
```

6.3 Análise Bivariada

Na análise bivariada procuramos identificar relações entre duas variáveis. Assim como na análise univariada, estas relações podem ser resumidas por gráficos, tabelas e/ou medidas estatísticas. O tipo de resumo vai depender dos tipos das variáveis envolvidas. Vamos considerar três possibilidades:

- Qualitativa *vs* qualitativa
- Qualitativa *vs* quantitativa
- Quantitativa *vs* quantitativa

Salienta-se ainda que:

- As análise mostradas a seguir não esgotam as possibilidades de análises envolvendo duas variáveis e devem ser vistas apenas como uma sugestão inicial
- Relações entre duas variáveis devem ser examinadas com cautela pois podem ser mascaradas por uma ou mais variáveis adicionais não considerada na análise. Estas são chamadas **variáveis de confundimento**. Análises com variáveis de confundimento não serão discutidas neste ponto.

  Observação: de agora em diante, como serão consideradas mais de uma variável, usaremos a função `with()` para chamar a maioria das funções.

6.3.1 Qualitativa *vs* qualitativa

Vamos considerar as variáveis `Est.civil` (estado civil), e `Inst` (grau de instrução). A tabela envolvendo duas variáveis é chamada **tabela de cruzamento** ou **tabela de contingência**, e pode ser apresentada de várias formas, conforme discutido a seguir.

A forma mais adequada de apresentação vai depender dos objetivos da análise e da interpretação desejada para os dados. Inicialmente obtemos a tabela de frequências absolutas para o cruzamento das duas variávies, usando a função `table()`. A tabela extendida incluindo os totais marginais pode ser obtida com a função `addmargins()`.

```
## Tabela de frequências absolutas
civ.inst.tb <- with(milsa, table(Est.civil, Inst))
civ.inst.tb
          Inst
Est.civil  1o Grau 2o Grau Superior
  casado         5      12        3
  solteiro       7       6        3
addmargins(civ.inst.tb)
          Inst
Est.civil  1o Grau 2o Grau Superior Sum
  casado         5      12        3  20
  solteiro       7       6        3  16
  Sum           12      18        6  36
```

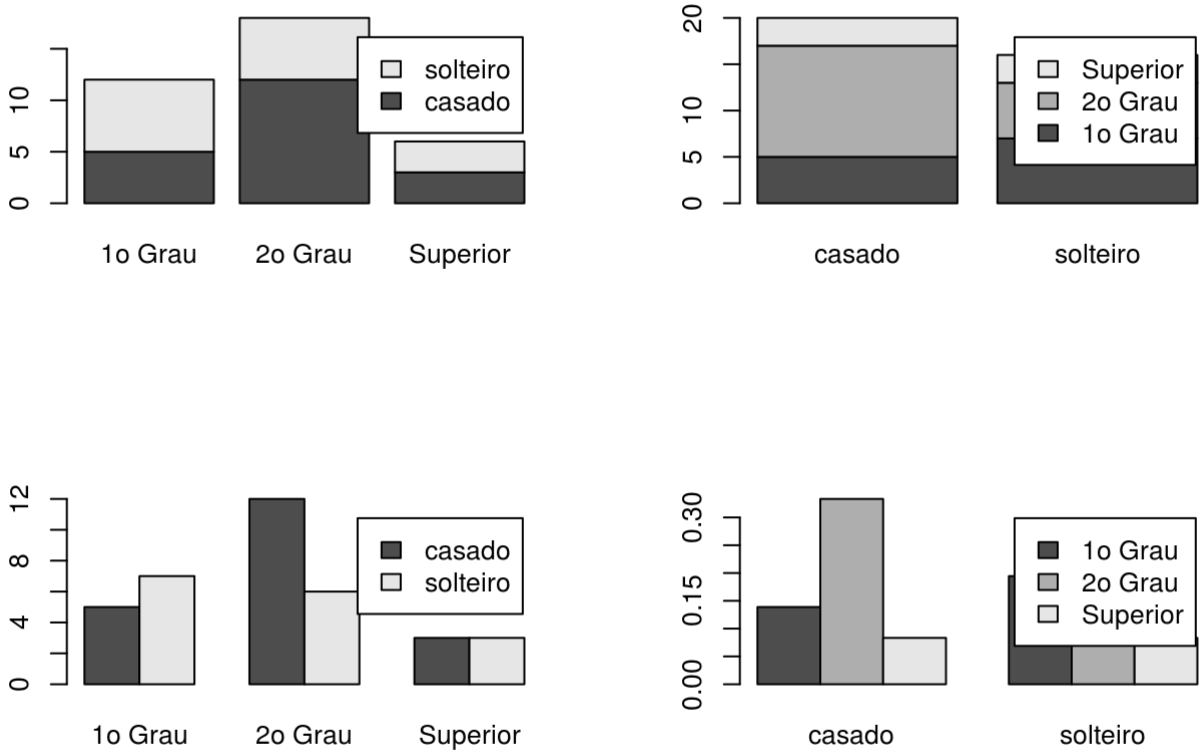
Tabelas de frequências relativas são obtidas com `prop.table()`, mas aqui existem três possibilidades para as proporções em cada casela:

- Em relação ao total geral
- Em relação aos totais por linha (`margin = 1`)
- Em relação aos totais por coluna (`margin = 2`)

```
## Frequência relativa global
prop.table(civ.inst.tb)
          Inst
Est.civil     1o Grau    2o Grau   Superior
  casado   0.13888889 0.33333333 0.08333333
  solteiro 0.19444444 0.16666667 0.08333333
## Frequência relativa por linha
prop.table(civ.inst.tb, margin = 1)
          Inst
Est.civil  1o Grau 2o Grau Superior
  casado    0.2500  0.6000   0.1500
  solteiro  0.4375  0.3750   0.1875
## Frequência relativa por coluna
prop.table(civ.inst.tb, margin = 2)
          Inst
Est.civil    1o Grau   2o Grau  Superior
  casado   0.4166667 0.6666667 0.5000000
  solteiro 0.5833333 0.3333333 0.5000000
```

Abaixo são representados quatro tipos de gráficos de barras que podem ser usados para representar o cruzamento das variáveis. A transposição da tabela com `t()` permite alterar a variável que define os grupos no eixo horizontal. O uso de `prop.table()` permite o obtenção de gráficos com frequências relativas.

```
par(mfrow = c(2,2))
barplot(civ.inst.tb, legend = TRUE)
barplot(t(civ.inst.tb), legend = TRUE)
barplot(civ.inst.tb, beside = TRUE, legend = TRUE)
barplot(t(prop.table(civ.inst.tb)), beside = TRUE, legend = TRUE)
par(mfrow = c(1,1))
```



6.3.2 Qualitativa *vs* quantitativa

Para exemplificar este caso vamos considerar as variáveis `Inst` e `Salario`.

Para se obter uma tabela de frequências é necessário agrupar a variável quantitativa em classes. No exemplo a seguir vamos agrupar a variável salário em 4 classes definidas pelos quartis usando a função `cut()`. Lembre-se que as classes são definidas por intervalos abertos à esquerda, então usamos o argumento `include.lowest = TRUE` para garantir que todos os dados, inclusive o menor (mínimo) seja incluído na primeira classe. Após agrupar esta variável, obtemos a(s) tabela(s) de cruzamento como mostrado no caso anterior.

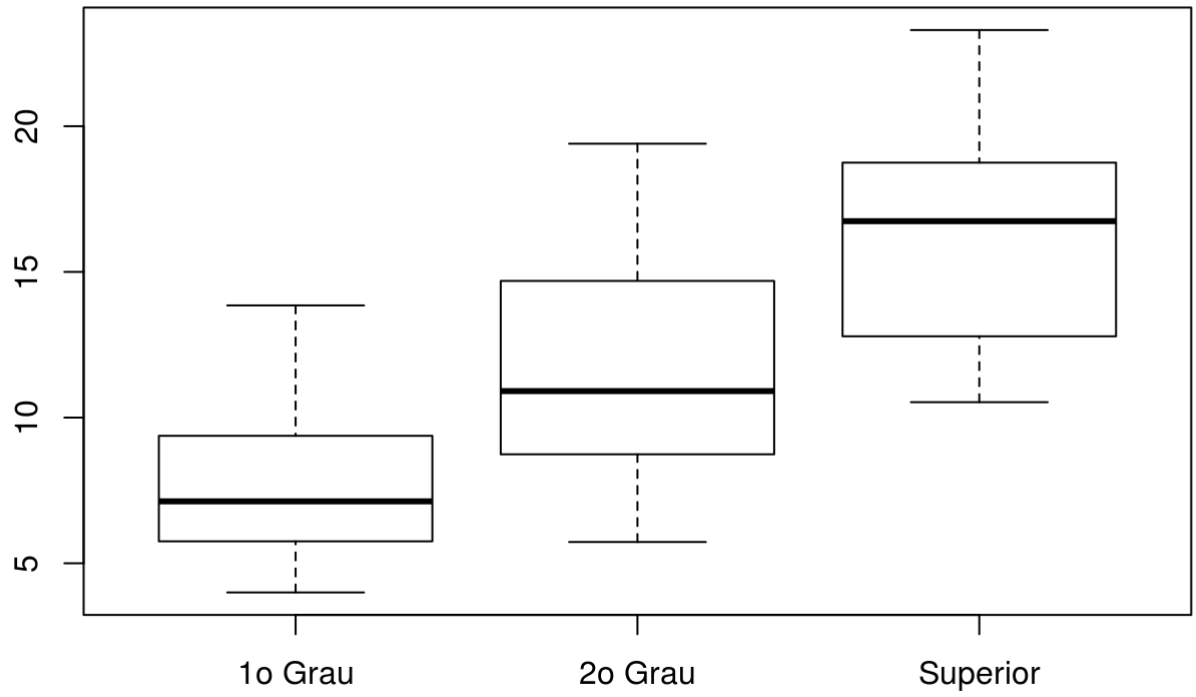
```
## Quartis de salario
quantile(milsa$Salario)
     0%     25%     50%     75%    100%
 4.0000  7.5525 10.1650 14.0600 23.3000
## Classificação de acordo com os quartis
salario.cut <- cut(milsa$Salario, breaks =  quantile(milsa$Salario),
                   include.lowest = TRUE)
## Tabela de frequências absolutas
inst.sal.tb <- table(milsa$Inst, salario.cut)
inst.sal.tb
          salario.cut
           [4,7.55] (7.55,10.2] (10.2,14.1] (14.1,23.3]
  1o Grau         7           3           2           0
  2o Grau         2           6           5           5
```

```
  Superior        0           0           2           4
prop.table(inst.sal.tb, margin = 1)
          salario.cut
            [4,7.55] (7.55,10.2] (10.2,14.1] (14.1,23.3]
  1o Grau  0.5833333   0.2500000   0.1666667   0.0000000
  2o Grau  0.1111111   0.3333333   0.2777778   0.2777778
  Superior 0.0000000   0.0000000   0.3333333   0.6666667
```

No gráfico vamos considerar que neste exemplo a instrução deve ser a variável explicativa e portanto colocada no eixo X, e o salário é a variável resposta, e portanto deve ser colocada no eixo Y. Isto é, consideramos que a instrução deve explicar, ainda que parcialmente, o salário (e não o contrário!).

Vamos então obter um *boxplot* dos salários para cada nível de instrução. Note que na função abaixo, usamos a notação de **fórmula** do R, com `Salario ~ Inst` indicando que a variável `Salario` é explicada, ou descrita, ($\sim$) pela variável `Inst`.

```
boxplot(Salario ~ Inst, data = milsa)
```



Poderíamos ainda fazer gráficos com a variável `Salario` agrupada em classes, e neste caso os gráficos seriam como no caso anterior com duas variáveis qualitativas.

Para as medidas descritivas, o usual é obter um resumo da variável quantitativa como mostrado na análise univariada, porém agora informando este resumo para cada nível do fator qualitativo de interesse.

A seguir mostramos alguns exemplos de como obter a média, desvio padrão e o resumo de cinco números do salário para cada nível de instrução.

```
with(milsa, tapply(Salario, Inst, mean))
  1o Grau   2o Grau  Superior
 7.836667 11.528333 16.475000
with(milsa, tapply(Salario, Inst, sd))
 1o Grau  2o Grau Superior
2.956464 3.715144 4.502438
with(milsa, tapply(Salario, Inst, quantile))
$`1o Grau`
     0%     25%     50%     75%    100%
 4.0000  6.0075  7.1250  9.1625 13.8500

$`2o Grau`
     0%     25%     50%     75%    100%
 5.7300  8.8375 10.9100 14.4175 19.4000

$Superior
     0%     25%     50%     75%    100%
10.5300 13.6475 16.7400 18.3775 23.3000
```

> NOTE que aqui usamos a função `tapply()`. Para saber mais sobre os recursos dessa função e de outras da família `*apply`, veja o script_gapminder.R.

6.3.3 Quantitativa *vs* Quantitativa

Para ilustrar este caso vamos considerar as variáveis `Salario` e `Idade`. Para se obter uma tabela é necessário agrupar as variáveis em classes conforme fizemos no caso anterior.

Nos comandos abaixo, agrupamos as duas variáveis em classes definidas pelos respectivos quartis, gerando portanto uma tabela de cruzamento 4×4.

```
## Classes de Idade
idade.cut <- with(milsa, cut(Idade, breaks = quantile(Idade),
                             include.lowest = TRUE))
table(idade.cut)
idade.cut
[20.8,30.7] (30.7,34.9] (34.9,40.5] (40.5,48.9]
          9           9           9           9
## Classes de salario
salario.cut <- with(milsa, cut(Salario, breaks = quantile(Salario),
                               include.lowest = TRUE))
table(salario.cut)
salario.cut
   [4,7.55] (7.55,10.2] (10.2,14.1] (14.1,23.3]
          9           9           9           9
## Tabela cruzada
table(idade.cut, salario.cut)
             salario.cut
idade.cut     [4,7.55] (7.55,10.2] (10.2,14.1] (14.1,23.3]
  [20.8,30.7]        4           2           2           1
  (30.7,34.9]        1           3           3           2
  (34.9,40.5]        1           3           2           3
  (40.5,48.9]        3           1           2           3
prop.table(table(idade.cut, salario.cut), margin = 1)
             salario.cut
```

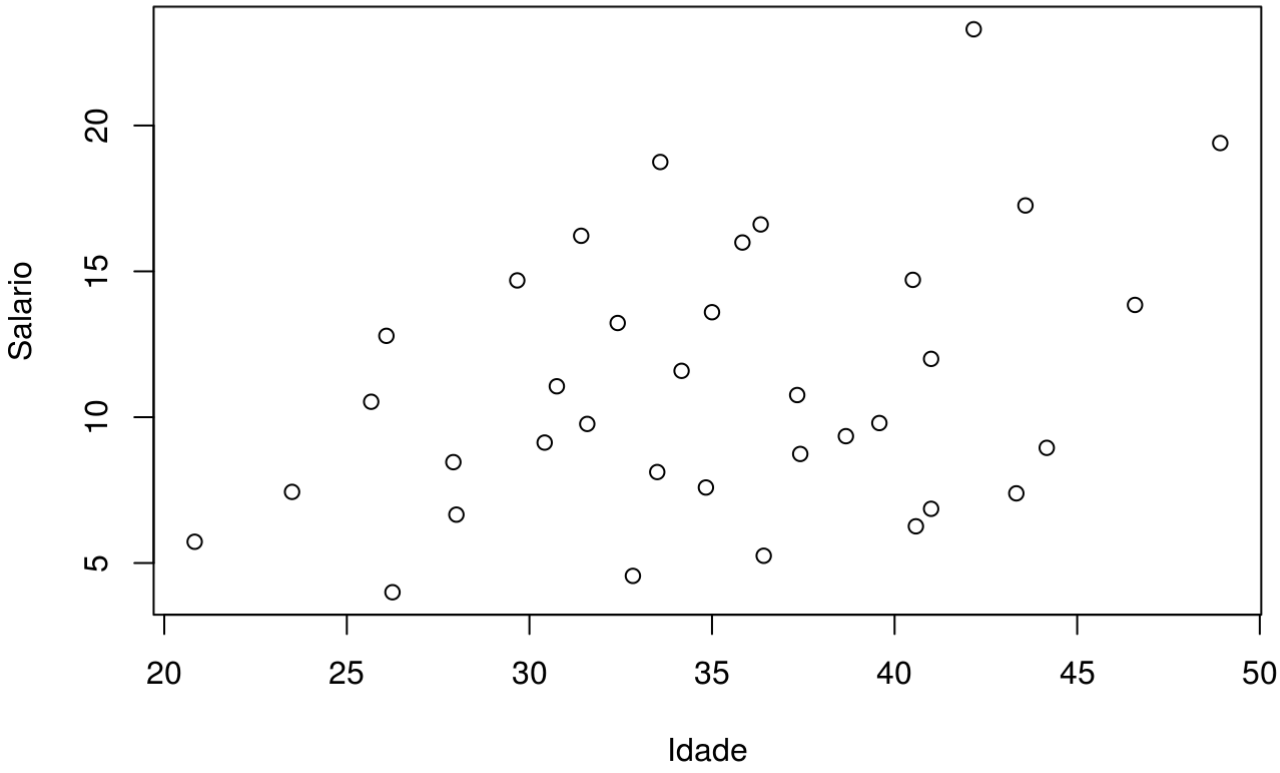
```
idade.cut      [4,7.55] (7.55,10.2] (10.2,14.1] (14.1,23.3]
  [20.8,30.7] 0.4444444   0.2222222   0.2222222   0.1111111
  (30.7,34.9] 0.1111111   0.3333333   0.3333333   0.2222222
  (34.9,40.5] 0.1111111   0.3333333   0.2222222   0.3333333
  (40.5,48.9] 0.3333333   0.1111111   0.2222222   0.3333333
```

Caso queiramos definir um número menor de classes podemos fazer como no exemplo a seguir onde cada variável é dividida em 3 classes e gerando um tabela de cruzamento 3×3.

```
idade.cut2 <- with(milsa, cut(Idade,
                              breaks = quantile(Idade, seq(0, 1, length = 4)),
                              include.lowest = TRUE))
salario.cut2 <- with(milsa, cut(Salario,
                                breaks = quantile(Salario, seq(0, 1, length = 4)),
                                include.lowest = TRUE))
table(idade.cut2, salario.cut2)
             salario.cut2
idade.cut2    [4,8.65] (8.65,12.9] (12.9,23.3]
  [20.8,32.1]        5           5           2
  (32.1,37.8]        4           3           5
  (37.8,48.9]        3           4           5
prop.table(table(idade.cut2, salario.cut2), margin = 1)
             salario.cut2
idade.cut2     [4,8.65] (8.65,12.9] (12.9,23.3]
  [20.8,32.1] 0.4166667   0.4166667   0.1666667
  (32.1,37.8] 0.3333333   0.2500000   0.4166667
  (37.8,48.9] 0.2500000   0.3333333   0.4166667
```

O gráfico adequado para representar duas variáveis quantitativas é um diagrama de dispersão. Note que se as variáveis envolvidas puderem ser classificadas como "explicativa" e "resposta" devemos colocar a primeira no eixo X e a segunda no eixo Y. Neste exemplo é razoável admitir que a idade deve explicar, ao menos parcialmente, o salário e portanto fazemos o gráfico com idade no eixo X. Note que na função `plot()`, podemos usar tanto os argumentos `x` e `y`, quanto o formato de fórmula (como visto anteriormente).

```
plot(x = milsa$Idade, y = milsa$Salario)
plot(Salario ~ Idade, data = milsa)
```



Para quantificar a associação entre variáveis deste tipo, usamos o coeficiente de correlação. A função `cor()` possui opção para três coeficientes de correlação, tendo como *default* o coeficiente de correlação linear de Pearson.

```
with(milsa, cor(Idade, Salario))
[1] 0.3651397
with(milsa, cor(Idade, Salario, method = "kendall"))
[1] 0.214456
with(milsa, cor(Idade, Salario, method = "spearman"))
[1] 0.2895939
```

Exercícios

1. Experimente as funções `mean()`, `var()`, `sd()`, `median()`, `quantile()` nos dados mostrados anteriormente (`milsa`). Veja a documentação das funções e as opções de uso.
2. Carregue o conjunto de dados `women` com `data(women)`. Veja o que são os dados com `help(women)`, e faça uma análise descritiva adequada.
3. Carregue o conjunto de dados `USArrests` com `data(USArrests)`. Examine a sua documentação com `help(USArrests)` e responda as perguntas a seguir:
    1. Qual o número médio e mediano de cada um dos crimes?
    2. Encontre a mediana e quartis para cada crime.
    3. Encontre o número máximo e mínimo para cada crime.

4. Faça um gráfico adequado para o número de assassinatos (`Murder`).
5. Faça um *boxplot* para o número de estupros (`Rape`).
6. Verifique se há correlação entre os diferentes tipos de crime.
7. Verifique se há correlação entre os crimes e a proporção de população urbana.
8. Encontre os estados com maior e menor ocorrência de cada tipo de crime.
9. Encontre os estados com maior e menor ocorrência per capta de cada tipo de crime.
10. Encontre os estados com maior e menor ocorrência do total de crimes.
11. Calcule a média de crimes (entre `Murder`, `Assault` e `Rape`) para cada estado.

A resolução de todos os exercícios desta página está disponível neste script.

Capítulo 7

Probabilidade e variáveis aleatórias

7.1 Conceitos básicos sobre distribuições de probabilidade

O objetivo desta sessão é mostrar o uso de funções do R em problemas de probabilidade. Exercícios que podem (e devem!) ser resolvidos analiticamente, serão utilizados para ilustrar o uso do programa e alguns de seus recursos para análises numéricas.

Os problemas desta sessão foram retirados do livro de Bussab e Morettin (2017). Veja também os códigos do R dos exemplos do livro para os capítulos 6 e 7, disponíveis aqui.

Exemplo 1: (Adaptado de Bussab e Morettin, cap. 7 ex. 1)

Dada a função

$$f(x) = \begin{cases} 2e^{-2x} & \text{, se } x \geq 0 \\ 0 & \text{, se } x < 0 \end{cases}$$

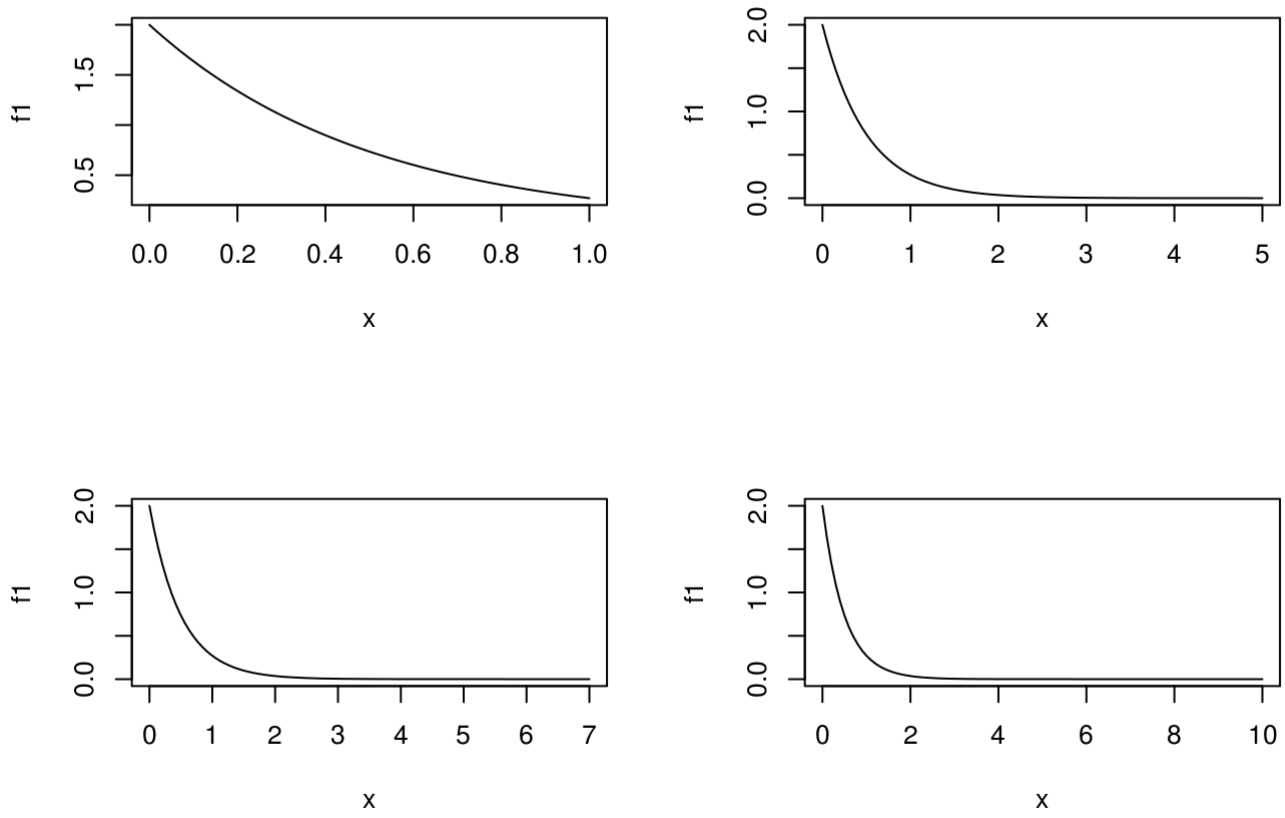
a. Mostre que está função é uma f.d.p.
b. Calcule a probabilidade de que $X > 1$
c. calcule a probabilidade de que $0.2 < X < 0.8$

Para ser f.d.p. a função não deve ter valores negativos e deve integrar 1 em seu domínio. Vamos começar definindo esta função como uma *função* no R para qual daremos o nome de `f1`.

```
f1 <- function(x){
    fx <- ifelse(x < 0, 0, 2 * exp(-2 * x))
    return(fx)
}
```

A seguir fazemos o gráfico da função. Como a função tem valores positivos para x no intervalo de zero a infinito, temos que definir um limite em x até onde vai o gráfico da função. Vamos achar este limite tentando vários valores, conforme mostram os comandos abaixo.

```
par(mfrow = c(2, 2))
plot(f1)
plot(f1, from = 0, to = 5)
plot(f1, from = 0, to = 7)
plot(f1, from = 0, to = 10)
par(mfrow = c(1, 1))
```



Para verificar que a a integral da função é igual a 1 podemos usar a função `integrate()`, que efetua integração numérica. A função recebe como argumentos o objeto com a função a ser integrada e os limites de integração. Neste exemplo o objeto é `f1` definido acima, e o domínio da função é $[0, \infty)$. A saída da função mostra o valor da integral acima, e o erro máximo da aproximação numérica.

```
integrate(f1, lower = 0, upper = Inf)
1 with absolute error < 5e-07
```

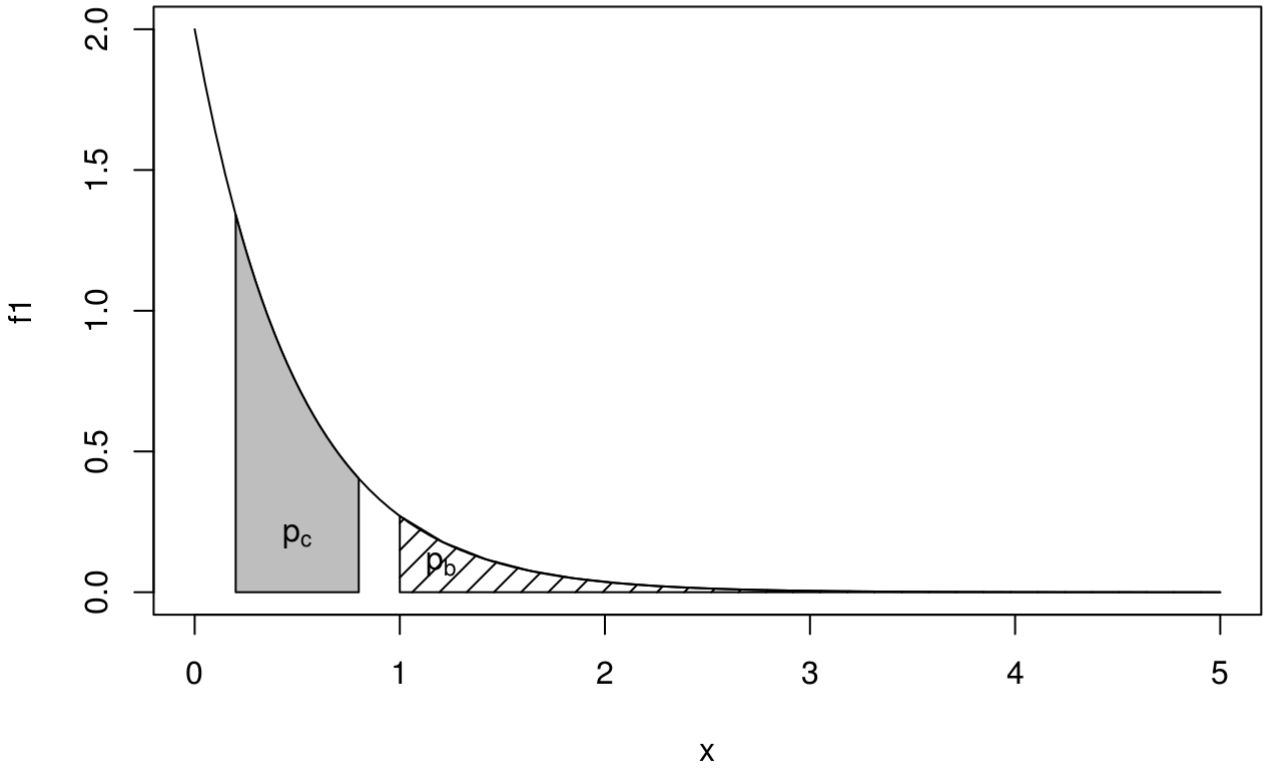
Para fazer cálculos pedidos nos itens (b) e (c) lembramos que a probabilidade é dada pela área sob a curva da função no intervalo pedido. Desta forma as soluções seriam dadas pelas expressões

$$\begin{aligned} p_b &= P(X > 1) = \int_1^{\infty} f(x)dx = \int_1^{\infty} 2\,e^{-2x}dx \\ p_c &= P(0,2 < X < 0,8) = \int_{0,2}^{0,8} f(x)dx = \int_{0.2}^{0.8} 2\,e^{-2x}dx \end{aligned}$$

cuja representação gráfica é mostrada na abaixo.

Os comandos a seguir mostram como fazer o gráfico dessas funções. O comando `plot()` desenha o gráfico principal, e para destacar as áreas que correspondem às probabilidades pedidas vamos usar a função `polygon()`. Esta função adiciona a um gráfico um polígono que é definido pelas coordenadas de seus vértices. Para sombrear a área usa-se o argumento `density`. Finalmente, para escrever um texto no gráfico usamos a função `text()` com as coordenadas de posição do texto.

```
plot(f1, from = 0, to = 5)
polygon(x = c(1, seq(1, 5, length.out = 20)),
        y = c(0, f1(seq(1, 5,length.out = 20))),
        density = 10)
polygon(x = c(0.2, seq(0.2, 0.8, length.out = 20), 0.8),
        y = c(0, f1(seq(0.2, 0.8, length.out = 20)), 0),
        col = "gray")
text(x = c(1.2, 0.5), y = c(0.1, 0.2),
     c(expression(p[b], p[c])))
```



E para obter as probabilidades pedidas usamos `integrate()`.

```
integrate(f1, lower = 1, upper = Inf)
0.1353353 with absolute error < 2.1e-05
integrate(f1, lower = 0.2, upper = 0.8)
0.4684235 with absolute error < 5.2e-15
```

Exemplo 2: (Adaptado de Bussab e Morettin, cap. 7 ex. 10)

A demanda diária de arroz em um supermercado, em centenas de quilos, é uma VA X com f.d.p. dada por

$$f(x) = \begin{cases} \frac{2}{3}x \text{ , se } 0 \leq x < 1 \\ -\frac{x}{3} + 1 \text{ , se } 1 \leq x < 3 \\ 0 \text{ , se } x < 0 \text{ ou } x \geq 3 \end{cases} \tag{7.1}$$

a. Calcular a probabilidade de que sejam vendidos mais que 150 kg.
b. Calcular a venda esperada em 30 dias.

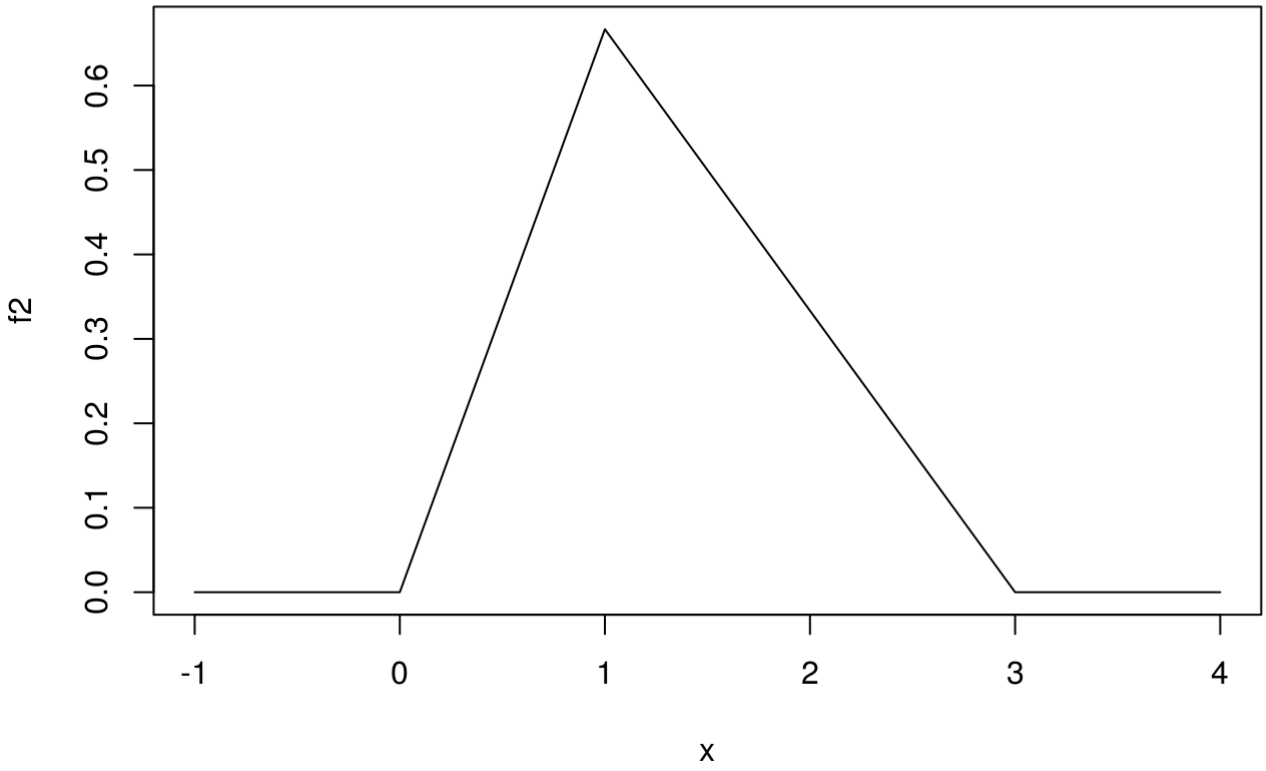
Novamente começamos definindo um objeto do R que contém a função dada acima.

Neste caso definimos um vetor do mesmo tamanho do argumento x para armazenar os valores de $f(x)$ e a seguir preenchemos os valores deste vetor para cada faixa de valor de x:

```
f2 <- function(x){
  fx <- numeric(length(x))
  fx[x < 0] <- 0
  fx[x >= 0 & x < 1] <- (2/3) * x[x >= 0 & x < 1]
  fx[x >= 1 & x < 3] <- (-1/3) * x[x >= 1 & x < 3] + 1
  fx[x >= 3] <- 0
  return(fx)
}
```

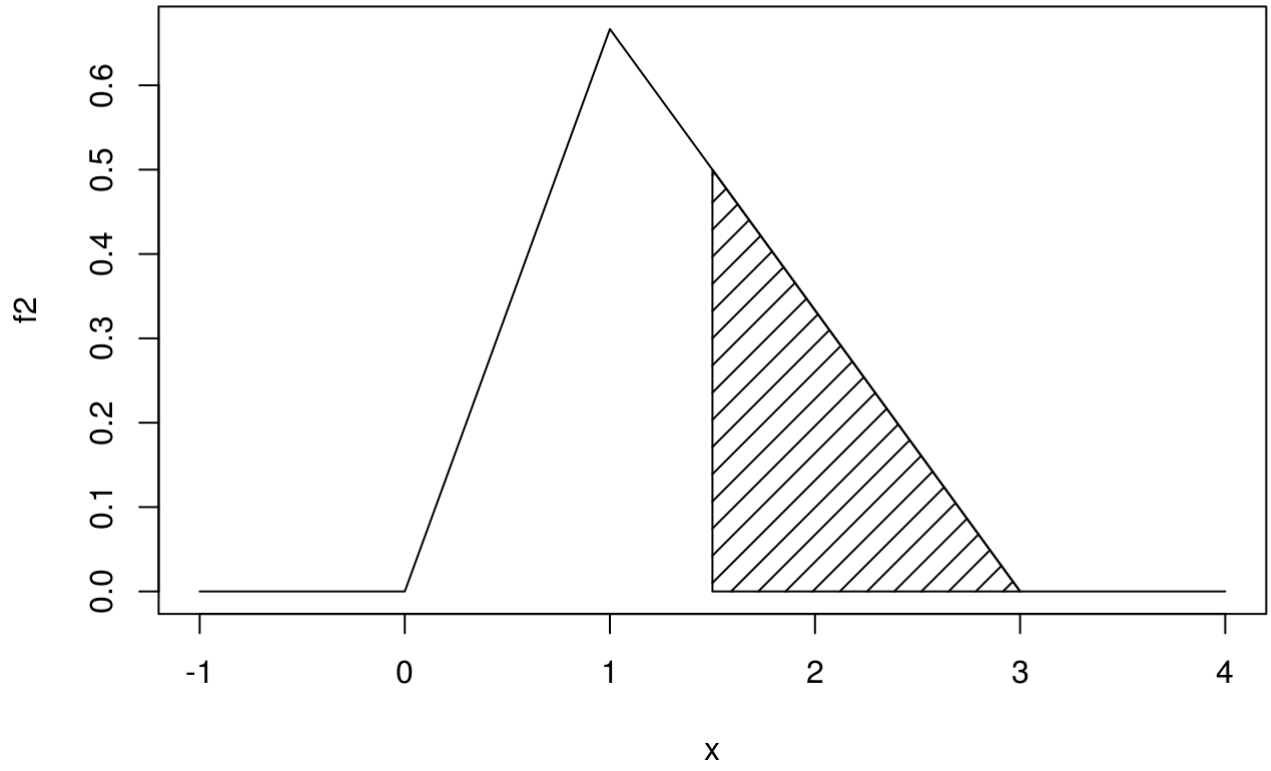
A seguir verificamos que a integral da função é 1 e fazemos o seu gráfico mostrado abaixo:

```
integrate(f2, 0, 3) ## verificando que a integral vale 1
1 with absolute error < 1.1e-15
plot(f2, -1, 4)     ## fazendo o gráfico da função
```



Agora vamos responder às questões levantadas. Na questão (a) pede-se a probabilidade de que sejam vendidos mais que 150 kg (1,5 centenas de quilos), portanto a probabilidade $P[X > 1,5]$. A probabilidade corresponde à área sob a função no intervalo pedido, ou seja, $P[X > 1,5] = \int_{1,5}^{\infty} f(x)dx$, que pode ser vista na figura abaixo.

```
plot(f2, -1, 4)
polygon(x = c(1.5, 1.5, 3), y = c(0, f2(1.5), 0), dens = 10)
```



A integral pode então ser resolvida numericamente com o comando:

```
integrate(f2, 1.5, 3)
0.375 with absolute error < 4.2e-15
```

A venda esperada em trinta dias é 30 vezes o valor esperado de venda em um dia. Para calcular a esperança $E[X] = \int xf(x)dx$ definimos uma nova função e resolvemos a integral. A função `integrate()` retorna uma lista onde um dos elementos (`value`) é o valor da integral.

```
ef2 <- function(x) { x * f2(x) }
integrate(ef2, 0, 3)
1.333333 with absolute error < 7.3e-05
30 * integrate(ef2, 0, 3)$value
[1] 40
```

Finalmente lembramos que os exemplos discutidos aqui são simples e não requerem soluções numéricas, devendo ser resolvidos analiticamente. Utilizamos estes exemplos somente para ilustrar a obtenção de soluções numéricas com o uso do R, que na prática deve ser utilizado em problemas mais complexos onde soluções analíticas não são triviais ou mesmo impossíveis.

Exercícios

1. (Adaptado de Bussab e Morettin, cap. 7, ex. 28). Em uma determinada localidade a distribuição de renda, em u.m. (unidade monetária) é uma variável aleatória X com função de distribuição de probabilidade:

$$f(x) = \begin{cases} \frac{1}{10}x + \frac{1}{10} & \text{se } 0 \leq x \leq 2 \\ -\frac{3}{40}x + \frac{9}{20} & \text{se } 2 < x \leq 6 \\ 0 & \text{se } x < 0 \text{ ou } x > 6 \end{cases}$$

   a. Mostre que $f(x)$ é uma f.d.p.
   b. Qual a renda média nesta localidade?
   c. Calcule a probabilidade de encontrar uma pessoa com renda acima de 4,5 u.m. e indique o resultado no gráfico da distribuição.

7.2 Distribuições de Probabilidade

O R inclui funcionalidade para operações com distribuições de probabilidades. Para cada distribuição há 4 operações básicas indicadas pelas letras:

- `d` calcula a densidade de probabilidade $f(x)$ no ponto x
- `p` calcula a função de probabilidade acumulada $F(x)$ no ponto x
- `q` calcula o quantil correspondente a uma dada probabilidade
- `r` retira uma amostra aleatória da distribuição

Para usar os funções deve-se combinar uma das letras acima com uma abreviatura do nome da distribuição. Por exemplo, para calcular probabilidades usamos: `pnorm()` para normal, `pexp()` para exponencial, `pbinom()` para binomial, `ppois()` para Poisson e assim por diante.

Vamos ver com mais detalhes algumas distribuições de probabilidades.

7.2.1 Distribuição Normal

A funcionalidade para distribuição normal é implementada por argumentos que combinam as letras acima com o termo `norm`. Vamos ver alguns exemplos com a distribuição normal padrão. Por `default` as funções assumem a distribuição normal padrão $N(\mu = 0, \sigma^2 = 1)$.

```
dnorm(-1)
[1] 0.2419707
pnorm(-1)
[1] 0.1586553
qnorm(0.975)
[1] 1.959964
rnorm(10)
 [1] -0.50219235  0.13153117 -0.07891709  0.88678481  0.11697127
 [6]  0.31863009 -0.58179068  0.71453271 -0.82525943 -0.35986213
```

O primeiro valor acima, de `dnorm(-1)`, corresponde ao valor da densidade da normal

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

com parâmetros $(\mu = 0, \sigma^2 = 1)$ no ponto $x = -1$. Portanto, o mesmo valor seria obtido substituindo x por -1 na expressão da normal:

```
mu <- 0
sigma <- 1
x <- -1
(1/(sigma * sqrt(2*pi))) * exp((-1/2) * ((x - mu)/sigma)^2)
[1] 0.2419707
```

A função `pnorm(-1)` calcula a probabilidade $P(X \leq -1)$. O comando `qnorm(0.975)` calcula o valor de a tal que $P(X \leq a) = 0.975$. Finalmente, o comando `rnorm(10)` gera uma amostra aleatória de 10 elementos da normal padrão. Note que os valores que você obtém rodando este comando podem ser diferentes dos mostrados acima.

As funções acima possuem argumentos adicionais, para os quais valores padrão (*default*) foram assumidos, e que podem ser modificados. Usamos `args()` para ver os argumentos de uma função e `help()` para visualizar a documentação detalhada:

```
args(rnorm)
function (n, mean = 0, sd = 1)
NULL
```

As funções relacionadas à distribuição normal possuem os argumentos `mean` e `sd` para definir média e desvio padrão da distribuição que podem ser modificados como nos exemplos a seguir. Note nestes exemplos que os argumentos podem ser passados de diferentes formas.

```
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
```

> **Observação**: na segunda linha de comando acima, foi utilizado um recurso do R chamado *partial matching*. Isso significa que nomes de argumentos de funções podem ser especificados pelo seu nome parcial, ou seja, com apenas o início do nome, desde que não haja ambiguação com outros argumentos.

Para informações mais detalhadas pode-se usar `help()`. O comando

```
help(rnorm)
```

irá exibir em uma janela a documentação da função que pode também ser chamada com `?rnorm`. Note que ao final da documentação são apresentados exemplos que podem ser rodados pelo usuário e que auxiliam na compreensão da funcionalidade. Note também que as 4 funções relacionadas à distribuição normal são documentadas conjuntamente, portanto `help(rnorm)`, `help(qnorm)`, `help(dnorm)` e `help(pnorm)` irão exibir a mesma documentação.

Cálculos de probabilidades usuais, para os quais utilizávamos tabelas estatísticas podem ser facilmente obtidos como no exemplo a seguir.

Seja X uma VA com distribuição $N(100, 100)$. Calcular as probabilidades:

- $P[X < 95]$
- $P[90 < X < 110]$
- $P[X > 95]$

Calcule estas probabilidades de forma usual, usando a tabela da normal. Depois compare com os resultados fornecidos pelo R. Os comandos do para obter as probabilidades pedidas são:

```
## P[X < 95]
pnorm(95, 100, 10)
[1] 0.3085375
## P[90 < X < 110]
pnorm(110, 100, 10) - pnorm(90, 100, 10)
```

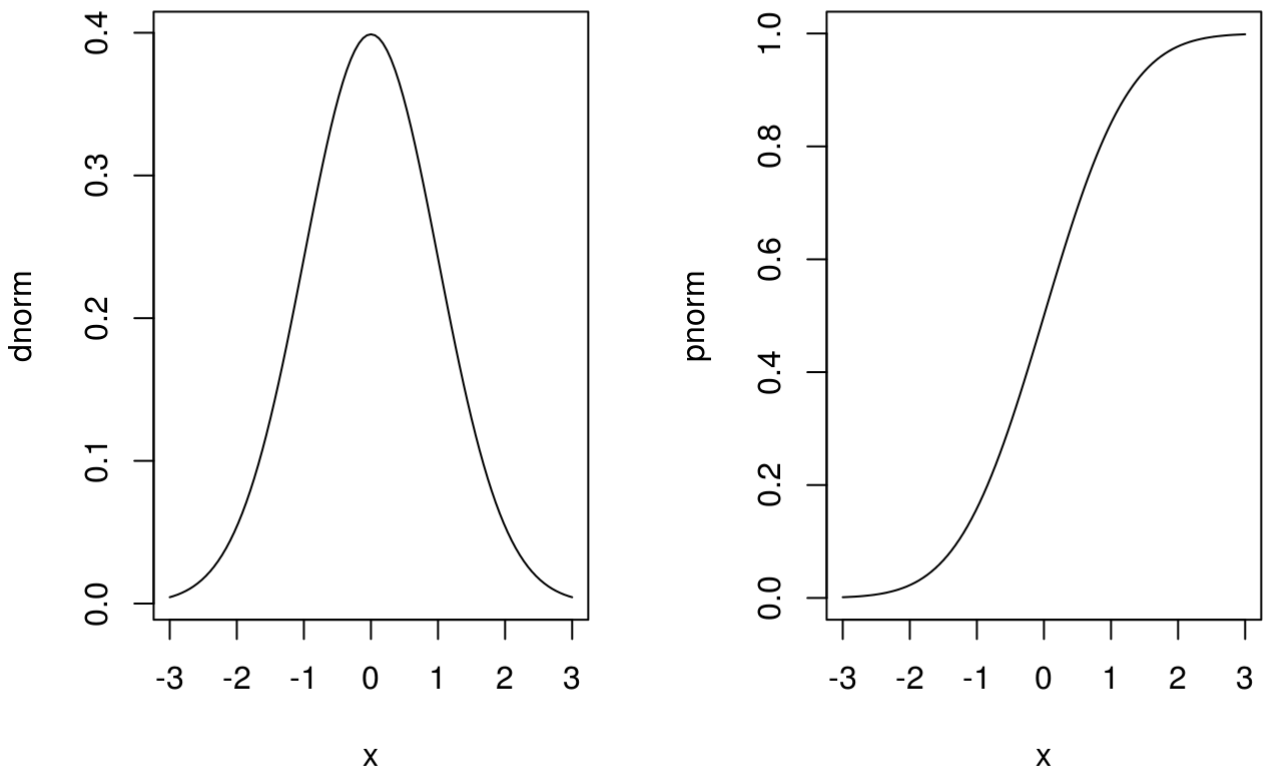
```
[1] 0.6826895
## P[X > 95] = 1 - P[X < 95]
1 - pnorm(95, 100, 10)
[1] 0.6914625
pnorm(95, 100, 10, lower.tail = FALSE) # melhor
[1] 0.6914625
```

Note que a última probabilidade foi calculada de duas formas diferentes, a segunda usando o argumento `lower.tail` que implementa um algorítmo de cálculo de probabilidades mais estável numericamente, e essa forma é preferida no lugar de usar o complementar.

A seguir vamos ver comandos para fazer gráficos de distribuições de probabilidade. Vamos fazer gráficos de funções de densidade e de probabilidade acumulada. Estude cuidadosamente os comandos abaixo e verifique os gráficos por eles produzidos.

A figura abaixo mostra gráficos da densidade (esquerda) e probabilidade acumulada (direita) da normal padrão, produzidos com os comandos a seguir. Para fazer o gráfico consideramos valores de X entre -3 e 3 que correspondem a +/- três desvios padrões da média, faixa que concentra 99,73% da massa de probabilidade da distribuição normal.

```
par(mfrow = c(1, 2))
plot(dnorm, from = -3, to = 3)
plot(pnorm, from = -3, to = 3)
par(mfrow = c(1, 1))
```

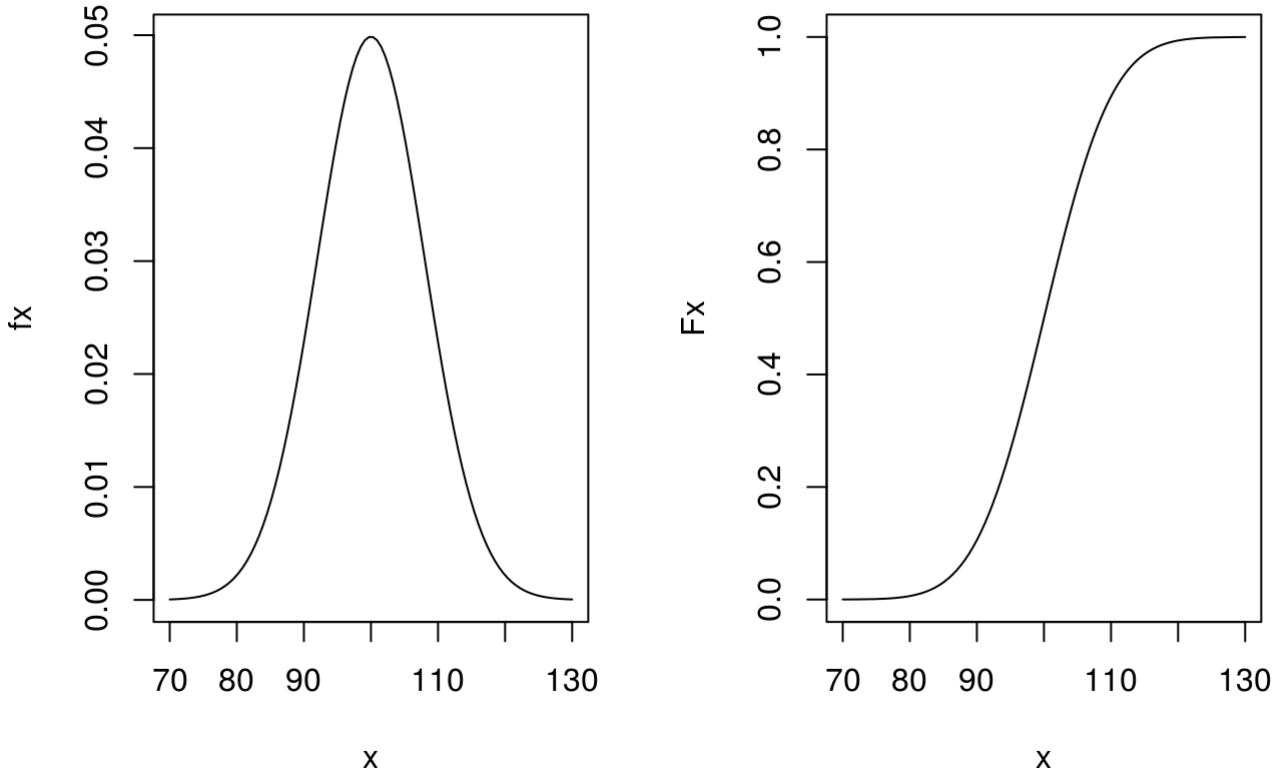


VEJA a página de ajuda da função `plot.function()` para entender os argumentos e

demais funcionalidades desta função gráfica para plotar gráficos de funções

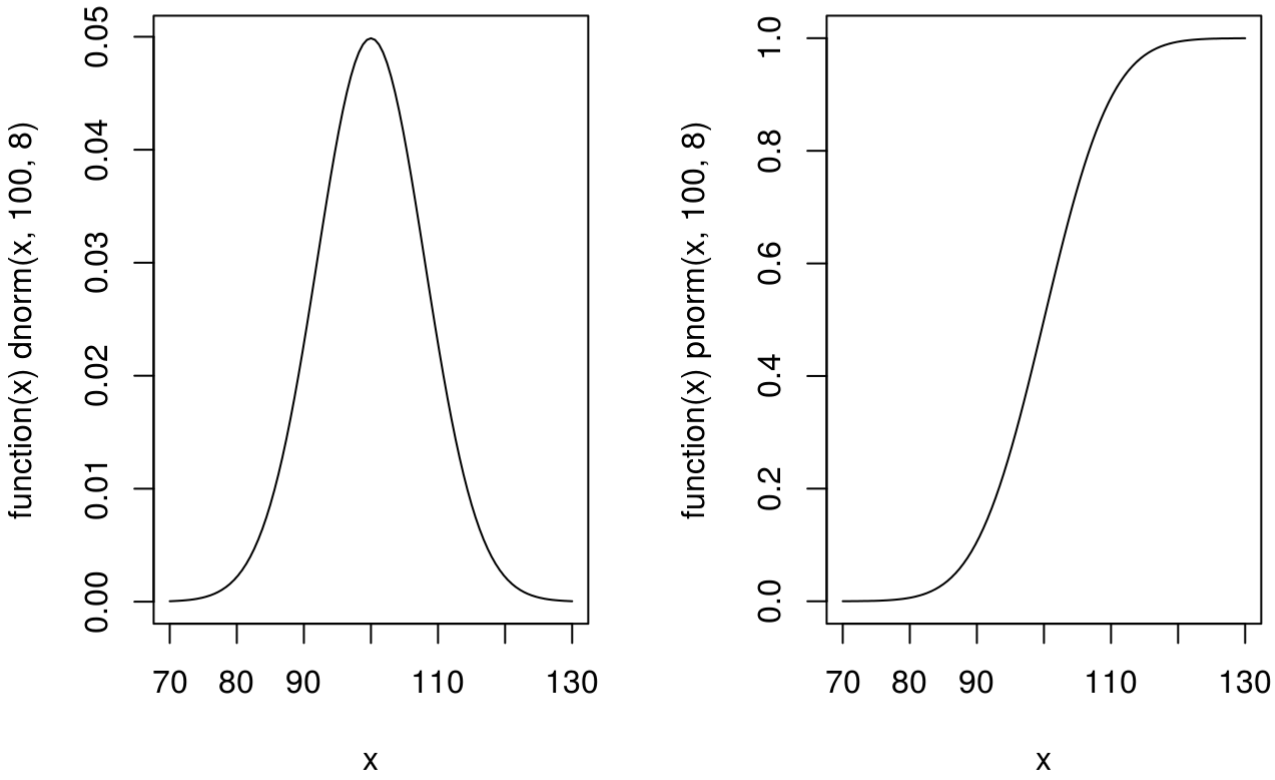
A seguinte figura mostra gráficos da densidade (esquerda) e probabilidade acumulada (direita) da $N(100, 64)$. Para fazer estes gráficos tomamos uma sequência de valores de X entre 70 e 130 e para cada um deles calculamos o valor das funções $f(x)$ e $F(x)$. Depois unimos os pontos $(x, f(x))$ em um gráfico e $(x, F(x))$ no outro.

```
par(mfrow = c(1, 2))
x <- seq(70, 130, length.out = 100)
fx <- dnorm(x, 100, 8)
plot(x, fx, type = "l")
Fx <- pnorm(x, 100, 8)
plot(x, Fx, type = "l")
par(mfrow = c(1, 1))
```



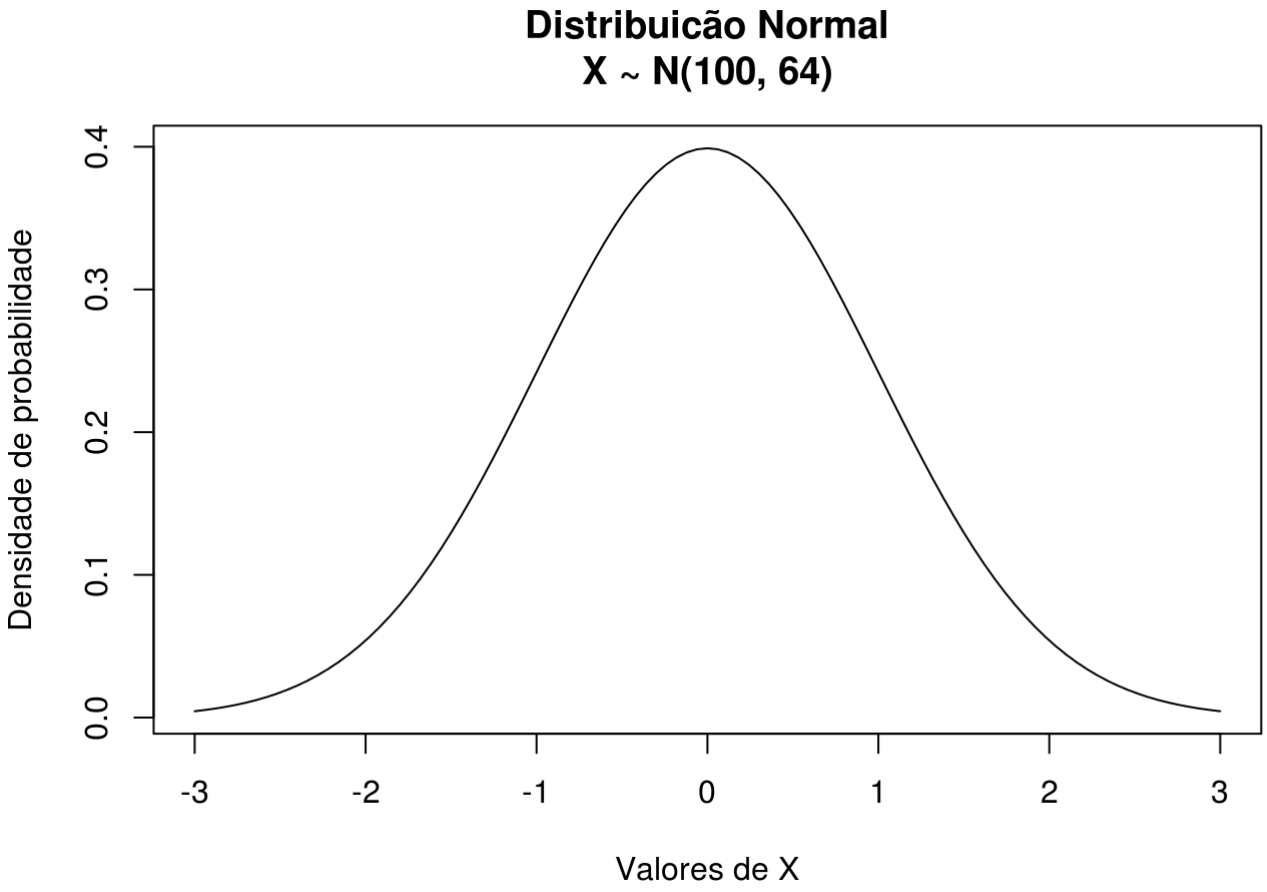
Note que, alternativamente, os mesmos gráficos poderiam ser produzidos com os comandos a seguir, onde fazemos usa da função `plot.function()`.

```
par(mfrow = c(1, 2))
plot(function(x) dnorm(x, 100, 8), from = 70, to = 130)
plot(function(x) pnorm(x, 100, 8), from = 70, to = 130)
par(mfrow = c(1, 1))
```



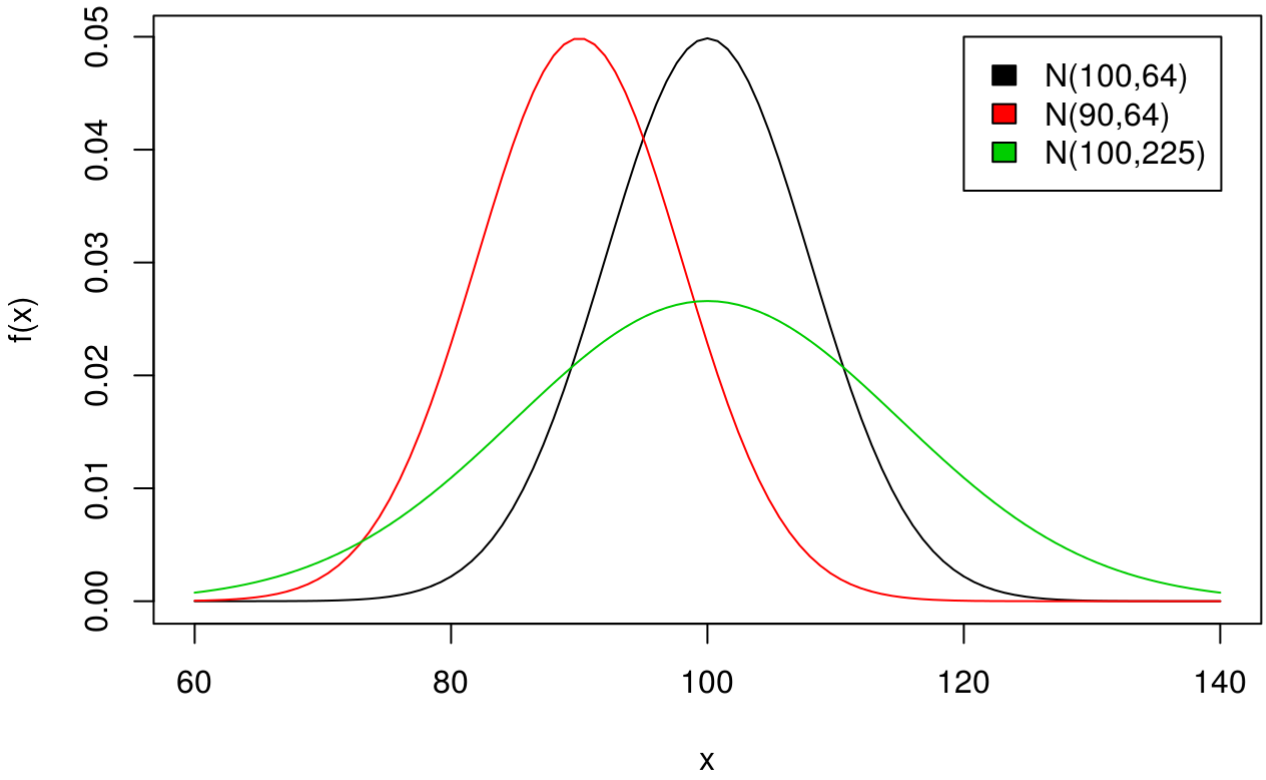
Comandos usuais do R podem ser usados para modificar a aparência dos gráficos. Por exemplo, podemos incluir títulos e mudar texto dos eixos conforme mostrado abaixo.

```
plot(dnorm, from = -3, to = 3,
     xlab = "Valores de X",
     ylab = "Densidade de probabilidade")
title("Distribuicão Normal\nX ~ N(100, 64)")
```



Os demais comandos abaixo mostram como colocar diferentes densidades em um mesmo gráfico, usando o argumento `add = TRUE`.

```
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(120, 0.05, fill = 1:3,
       legend = c("N(100,64)", "N(90,64)", "N(100,225)"))
```



7.2.2 Distribuição Binomial

Cálculos para a distribuição binomial são implementados combinando as letras básicas vistas acima com o termo `binom`. Vamos primeiro investigar argumentos e documentação com `args()` e `help()`.

```
args(dbinom)
function (x, size, prob, log = FALSE)
NULL
```

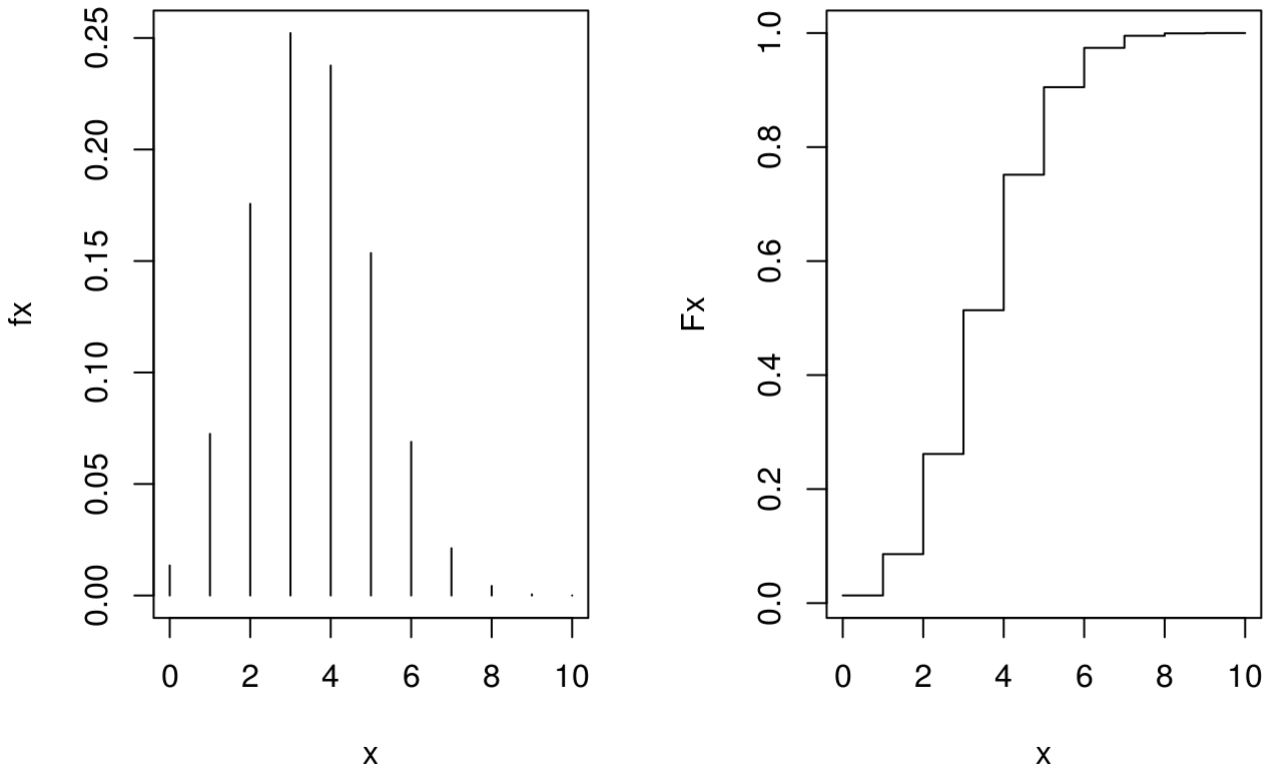
```
help(dbinom)
```

Seja X uma VA com distribuição Binomial, com $n = 10$ e $p = 0.35$. Vamos ver os comandos do R para:

- Fazer o gráfico da função de densidade
- Idem para a função de probabilidade
- Calcular $P[X = 7]$
- Calcular $P[X \leq 7]$
- Calcular $P[X > 7]$
- Calcular $P[3 < X \leq 6]$

Note que sendo uma distribuição discreta de probabilidades os gráficos são diferentes dos obtidos para distribuição normal e os cálculos de probabilidades devem considerar as probabilidades nos pontos. Os gráficos das funções de densidade e probabilidade são mostrados abaixo.

```
par(mfrow = c(1, 2))
x <- 0:10
fx <- dbinom(x, size = 10, prob = 0.35)
plot(x, fx, type = "h")
Fx <- pbinom(x, size = 10, prob = 0.35)
plot(x, Fx, type = "s")
par(mfrow = c(1, 1))
```



As probabilidades pedidas são obtidas com os comandos a seguir.

```
## P[X = 7]
dbinom(7, size = 10, prob = 0.35)
[1] 0.02120302
## P[X <= 7]
pbinom(7, size = 10, prob = 0.35)
[1] 0.9951787
# OU
sum(dbinom(0:7, size = 10, prob = 0.35))
[1] 0.9951787
## P[X > 7]
1 - pbinom(7, size = 10, prob = 0.35)
[1] 0.004821265
pbinom(7, size = 10, prob = 0.35, lower.tail = FALSE) # melhor
[1] 0.004821265
```

```
## P[3 < X <= 6]
pbinom(6, 10, 0.35) - pbinom(3, 10, 0.35)
[1] 0.4601487
# OU
sum(dbinom(4:6, 10, 0.35))
[1] 0.4601487
```

7.2.3 Distribuição de Poisson

Definição: Seja um experimento realizado nas seguintes condições: i. As ocorrências são independentes ii. As ocorrências são aleatórias iii. A variável aleatória X é o número de ocorrências de um evento **ao longo de algum intervalo**} (de tempo ou espaço)

Denominamos esse experimento de **processo de Poisson**. Vamos associar a V.A. X o número de ocorrências em um intervalo. Portanto X poderá assumir os valores $0, 1, \ldots,$ (sem limite superior).

A distribuição de Poisson é utilizada para descrever a probabilidade do **número de ocorrências** em um **intervalo contínuo** (de tempo ou espaço). No caso da distribuição binomial, a variável de interesse era o número de sucessos em um **intervalo discreto** (n ensaios de Bernoulli). A unidade de medida (tempo ou espaço) é uma variável contínua, mas a *variável aleatória*, o **número de ocorrências**}, é discreta.

Uma V.A. X segue o modelo de Poisson se surge a partir de um processo de Poisson, e sua **função de probabilidade**} for dada por

$$P[X = x] = \frac{e^{-\mu}\mu^x}{x!}, \qquad x = 0, 1, \ldots$$

onde

$$\mu = \lambda \cdot t$$

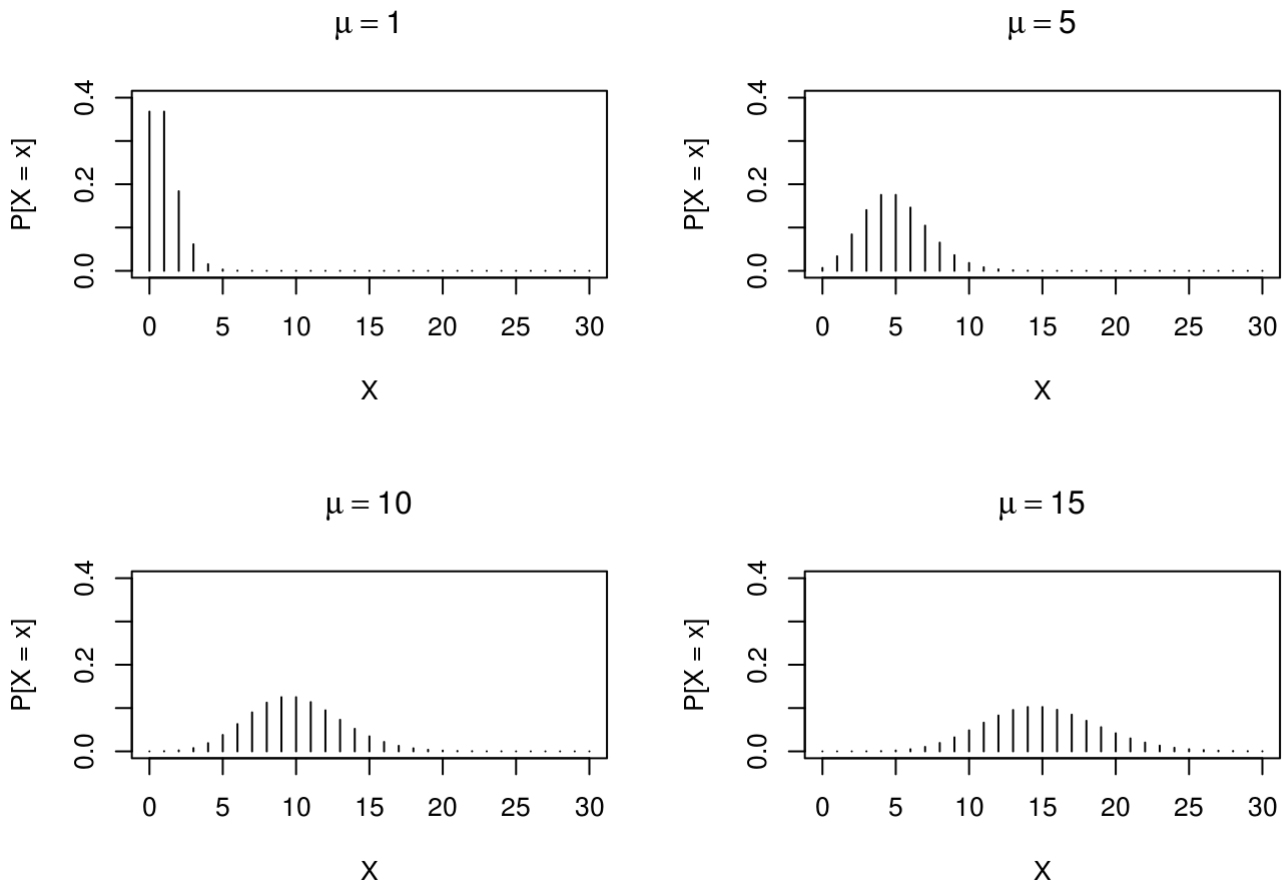
O parâmetro μ indica a taxa de ocorrência (λ) por unidade de medida (t), ou seja,

$$\lambda = \text{taxa de ocorrência} \quad \text{e} \quad t = \text{intervalo de tempo ou espaço}$$

- Notação: $X \sim \text{Pois}(\mu)$
- Esperança e variância: $\text{E}(X) = \mu = \text{Var}(X)$

Alguns exemplos de gráficos da distribuição de Poisson com diferentes valores do parâmetro μ.

```
par(mfrow=c(2,2))
plot(0:30, dpois(x = 0:30, lambda = 1), type = "h",
     xlab = "X", ylab = "P[X = x]", main = expression(mu == 1),
     ylim = c(0,.4))
plot(0:30, dpois(x = 0:30, lambda = 5), type = "h",
     xlab = "X", ylab = "P[X = x]", main = expression(mu == 5),
     ylim = c(0,.4))
plot(0:30, dpois(x = 0:30, lambda = 10), type = "h",
     xlab = "X", ylab = "P[X = x]", main = expression(mu == 10),
     ylim = c(0,.4))
plot(0:30, dpois(x = 0:30, lambda = 15), type = "h",
     xlab = "X", ylab = "P[X = x]", main = expression(mu == 15),
     ylim = c(0,.4))
```



- **Exemplo**: As chamadas telefônicas chegam a uma delegacia de polícia à uma taxa de 8 chamadas por hora, em dias úteis.
  a. Quantas chamadas de emergência são esperadas em um período de 15 minutos?
  b. Qual a probabilidade de nenhuma chamada em um período de 15 minutos?
  c. Qual a probabilidade de ocorrer pelo menos duas chamadas no período de 15 minutos?
  d. Qual a probabilidade de ocorrer exatamente duas chamadas em 20 minutos?

```
## a) E(X) = mu = lambda . t
lambda <- 8/60 # 8 chamadas/60 minutos
t <- 15 # 15 minutos
(mu <- lambda * t)
[1] 2
## b) P[x = 0]
ppois(0, mu)
[1] 0.1353353
dpois(0, mu)
[1] 0.1353353
## c) P[X >= 2] = 1 - P[X < 2]
1 - ppois(1, mu)
[1] 0.5939942
ppois(1, mu, lower.tail = FALSE)
[1] 0.5939942
## d) P[X = 2]
t <- 20
(mu <- lambda * t)
```

```
[1] 2.666667
dpois(2, mu)
[1] 0.2470523
```

- **Exemplo:** Suponha que 150 erros de impressão são distribuídos aleatoriamente em um livro de 200 páginas. Encontre a probabilidade de que em 2 páginas contenham:
  a. nenhum erro de impressão
  b. três erros de impressão
  c. um ou mais erros de impressão

```
## lambda = taxa de ocorrência por página
lambda <- 150/200
## intervalo de interesse
t <- 2
## Parâmetro mu = lambda . t
(mu <- lambda * t)
[1] 1.5
## a) P[X = 0]
dpois(0, mu)
[1] 0.2231302
## b) P[X = 3]
dpois(3, mu)
[1] 0.1255107
## c) P[X >= 1] = 1 - P[X < 1]
1 - ppois(0, mu)
[1] 0.7768698
ppois(0, mu, lower.tail = FALSE)
[1] 0.7768698
```

7.2.4 Distribuição Uniforme

7.2.4.1 Uniforme Contínua

Para a distribuição uniforme *contínua* usa-se as funções `*unif()` onde `*` deve ser p, q, d ou r como mencionado anteriormente. Nos comandos a seguir inspecionamos os argumentos, sorteamos 5 valores da $U(0,1)$ e calculamos a probabilidade acumulada até 0,75.

```
args(runif)
function (n, min = 0, max = 1)
NULL
runif(5)
[1] 0.5358112 0.7108038 0.5383487 0.7489722 0.4201015
punif(0.75)
[1] 0.75
```

Portanto, o *default* é uma distribuição uniforme no intervalo $[0,1]$ e os argumentos opcionais são `min` e `max`. Por exemplo, para simular 5 valores de $X \sim U(5,20)$ usamos:

```
runif(5, min = 5, max = 20)
[1]  7.571303 16.554524 18.229304 13.236451  9.165856
```

7.2.4.2 Uniforme Discreta

Não há entre as funções básicas do R uma função específica para a distribuição uniforme discreta com opções de prefixos r, d, p e d, provavelmente devido a sua simplicidade, embora algumas

outras funções possam ser usadas. Por exemplo para sortear números pode-se usar `sample()`, como no exemplo a seguir onde são sorteados 15 valores de uma uniforma discreta com valores (inteiros) entre 1 e 10 ($X \sim U_d(1,10)$).

```
sample(1:10, size = 15, replace = TRUE)
 [1]  5 10  4 10  7  9  2  7 10  2  4  9  8  9  7
```

7.2.5 A função `sample()`

A função `sample()` **não** é restrita à distribuição uniforme discreta, podendo ser usada para sorteios, com ou sem reposição (argumento `replace`, que por padrão é `FALSE`, ou seja, sem reposição), com a possibilidade de associar diferentes probabilidades a cada elemento (argumento `prob`, que por padrão associa probabilidades iguais para todos os elementos).

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

Vejamos alguns exemplos:

- Sorteio de 3 números entre os inteiros de 0 a 20

```
sample(0:20, size = 3)
[1] 10 15 16
```

- Sorteio de 5 números entre os elementos de um certo vetor `x`

```
x <- c(23, 34, 12, 22, 17, 28, 18, 19, 20, 13, 18)
sample(x, size = 5)
[1] 12 22 18 34 19
```

- Sorteio de 10 números entre os possíveis resultados do lançamento de um dado, com reposição

```
sample(1:6, size = 10, replace = TRUE)
 [1] 2 4 2 1 2 4 2 3 4 6
```

- Idem ao anterior, porém agora com a probabilidade de cada face proporcional ao valor da face.

```
sample(1:6, size = 10, replace = TRUE,  prob = 1:6)
 [1] 4 5 5 5 5 6 4 5 5 4
```

Este último exemplo ilustra ainda que os valores passados para o argumento `prob` não precisam ser probabilidades, são apenas entendidos como **pesos**. A própria função trata isto internamente fazendo a ponderação adequada.

7.3 Complementos sobre distribuições de probabilidade

Agora que já nos familiarizamos com o uso das distribuições de probabilidade vamos ver alguns detalhes adicionais sobre seu funcionamento.

7.3.1 Probabilidades e integrais

A probabilidade de um evento em uma distribuição contínua é uma área sob a curva da distribuição. Vamos reforçar esta idéia revisitando um exemplo visto na distribuição normal.

Seja X uma VA com distribuição $N(100, 100)$. Para calcular a probabilidade $P[X < 95]$ usamos o comando:

```
pnorm(95, mean = 100, sd = 10)
[1] 0.3085375
```

Vamos agora "esquecer" o comando `pnorm()` e ver uma outra forma de resolver usando integração numérica. Lembrando que a normal tem a função de densidade dada por

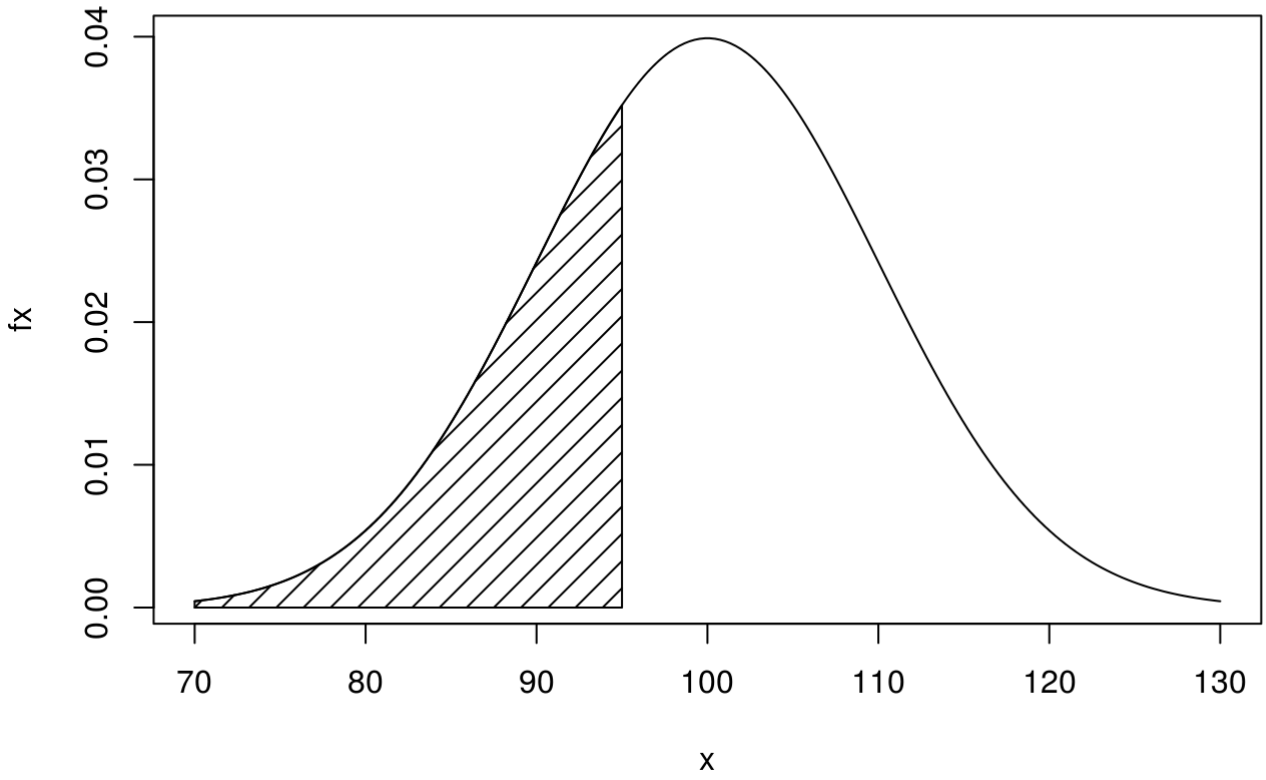
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

Podemos então definir uma função no R para calcular qualquer densidade em x

```
fn <- function(x, mu, sigma){
    (1/(sigma * sqrt(2*pi))) * exp((-1/2) * ((x - mu)/sigma)^2)
}
```

Para obter o gráfico desta distribuição, usamos o fato que a maior parte da função está no intervalo entre a média +/- três desvios padrões, portanto entre 70 e 130. Podemos então fazer como nos comandos que se seguem. Para marcar no gráfico a área que corresponde a probabilidade pedida criamos um polígono com coordenadas ax e ay definindo o perímetro desta área.

```
x <- seq(70, 130, length.out = 200)
fx <- fn(x, mu = 100, sigma = 10)
plot(x, fx, type = "l")
ax <- c(70, 70, x[x < 95], 95, 95)
ay <- c(0, fn(70, 100, 10), fx[x < 95], fn(95, 100, 10),0)
polygon(ax, ay, density = 10)
```



Para calcular a área pedida sem usar a função `pnorm()` podemos usar a função de integração numérica. Note que esta função, diferentemente da `pnorm()` reporta ainda o erro de aproximação numérica.

```
integrate(fn, mu = 100, sigma = 10, lower = -Inf, upper = 95)
0.3085375 with absolute error < 2.1e-06
```

Portanto para os demais ítens do problema, $P[90 < X < 110]$, e $P[X > 95]$ fazemos:

```
integrate(fn, mu = 100, sigma = 10, lower = 90, upper = 110)
0.6826895 with absolute error < 7.6e-15
integrate(fn, mu = 100, sigma = 10, lower = 95, upper = +Inf)
0.6914625 with absolute error < 8.1e-05
```

e os resultados acima evidentemente coincidem com os obtidos anterioriormente usando `pnorm()`,

```
pnorm(110, 100, 10) - pnorm(90, 100, 10)
[1] 0.6826895
pnorm(95, 100, 10, lower.tail = FALSE)
[1] 0.6914625
```

Note ainda que na prática não precisamos definir e usar a função `fn()`, pois ela fornece o mesmo resultado que a função `dnorm()`.

Exercícios

Nos exercícios abaixo iremos também usar o R como uma calculadora estatística para resolver alguns exemplos/exercícios de probabilidade tipicamente apresentados em um curso de estatística básica.

1. Para $X \sim N(90, 100)$, obtenha:
   a. $P(X \leq 115)$
   b. $P(X \geq 80)$
   c. $P(X \leq 75)$
   d. $P(85 \leq X \leq 110)$
2. Sendo X uma variável seguindo o modelo Binomial com parâmetros $n = 15$ e $p = 0.4$, pergunta-se:
   a. $P(X \geq 14)$
   b. $P(8 < X \leq 10)$
3. Uma empresa informa que 30% de suas contas a receber de outras empresas encontram-se vencidas. Se o contador da empresa seleciona aleatoriamente 5 contas, determine a probabilidade de:
   a. Nenhuma conta estar vencida
   b. Exatamente duas contas estarem vencidas
   c. Três ou mais contas estarem vencidas
4. Uma empresa recebe 720 emails em um intervalo de 8 horas. Qual a probabilidade de que:
   a. Em 6 minutos receba pelo menos 3 emails?
   b. Em 4 minutos não receba nenhum email?
5. O processo de empacotamento de uma fábrica de cereais foi ajustado de maneira que uma média de $\mu = 13,0$ kg de cereal seja colocado em cada caixa. Sabe-se que existe uma pequena variabilidade no enchimento dos pacotes devido à fatores aleatórios, e que o desvio-padrão do peso de enchimento é de $\sigma = 0,1$ kg. Assume-se que a distribuição dos pesos tem distribuição normal. Com isso, determine as probabilidades de que uma caixa escolhida ao acaso:
   a. Pese entre 13,0 e 13,2 kg.
   b. Tenha um peso maior do que 13,25 kg.
   c. Pese entre 12,8 e 13,1 kg.
   d. Pese entre 13,1 e 13,2 kg.
6. Faça os seguintes gráficos:
   a. da função de densidade de uma variável com distribuição de Poisson com parâmetro $\lambda = 5$
   b. da densidade de uma variável $X \sim N(90, 100)$
   c. sobreponha ao gráfico anterior a densidade de uma variável $Y \sim N(90, 80)$ e outra $Z \sim N(85, 100)$
   d. densidades de distribuições χ^2 com 1, 2 e 5 graus de liberdade.

Apêndice A

Programação Orientada a Objetos

Como vimos anteriormente, o R é uma linguagem de programação orientada à objetos. Dois conceitos fundamentais desse tipo de linguagem são os de **classe** e **método**. Já vimos também que todo objeto no R possui uma classe (que define sua estrutura) e analisamos algumas delas. O que seria então um método? Para responder essa pergunta precisamos entender inicialmente os tipos de orientação a objetos que o R possui.

O R possui 3 sitemas de orientação a objetos: **S3**, **S4**, e **RC**:

- **S3**: implementa um estilo de programação orientada a objeto chamada de *generic-function*. Esse é o estilo mais básico de programação em R (e também o mais utilizado). A ideia é que existam **funções genéricas** que decidem qual método aplicar de acordo com a classe do objeto. Os métodos são definidos da mesma forma que qualquer função, mas chamados de amenira diferente. É um estilo de programação mais "informal", mas possibilita uma grande liberdade para o programador.
- **S4**: é um estilo mais formal, no sentido que que as funções genéricas devem possuir uma classe formal definida. Além disso, é possível também fazer o **despacho múltiplo de métodos**, ao contrário da classe S3.
- **RC**: (*Reference Classes*, antes chamado de R5) é o sistema mais novo implementado no R. A principal diferença com os sistemas S3 e S4 é que métodos pertencem à objetos, não à funções. Isso faz com que objetos da classe RC se comportem mais como objetos da maioria das linguagens de programação, como Python, Java, e C#.

Nesta sessão vamos abordar como funcionam os métodos como definidos pelo sistema S3, por ser o mais utilizado na prática para se criar novas funções no R. Para saber mais sobre os outros métodos, consulte o livro Advanced R.

Vamos entender como uma função genérica pode ser criada através de um exemplo. Usando a função `methods()`, podemos verificar quais métodos estão disponíveis para uma determinada função, por exemplo, para a função `mean()`:

```
methods(mean)
[1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
see '?methods' for accessing help and source code
```

O resultado são expressões do tipo `mean.<classe>`, onde `<classe>` é uma classe de objeto como aquelas vistas anteriormente. Isso significa que a função `mean()`, quando aplicada à um objeto da classe `Date`, por exemplo, pode ter um comportamento diferente quando a mesma função for aplicada à um objeto de outra classe (numérica).

Suponha que temos o seguinte vetor numérico:

```
set.seed(1)
vec <- rnorm(100)
```

```
class(vec)
[1] "numeric"
```

e queremos calcular sua média. Basta aplicar a função `mean()` nesse objeto para obtermos o resultado esperado

```
mean(vec)
[1] 0.1088874
```

Mas isso só é possível porque existe um método definido espcificamente para um vetor da classe `numeric`, que nesse caso é a função `mean.default`. A função genérica nesse caso é a `mean()`, e a função método é a `mean.default`. Veja que não precisamos escrever onome inteiro da função genérica para que ela seja utilizada, como por exemplo,

```
mean.default(vec)
[1] 0.1088874
```

Uma vez passado um objeto para uma função, é a classe do objeto que irá definir qual método utilizar, de acordo com os métodos disponíveis. Veja o que acontece se forcarmos o uso da função `mean.Date()` nesse vetor

```
mean.Date(vec)
[1] "1970-01-01"
```

O resultado não faz sentido pois ele é específico para um objeto da classe `Date`.

Tudo isso acontece por causa de um mecanismo chamado de **despacho de métodos** (*method dispatch*), que é responsável por identificar a classe do objeto e utilizar ("despachar") a função método correta para aquela classe. Toda função genérica possui a mesma forma: uma chamada para a função `UseMethod()`, que especifica o nome genérico e o objeto a ser despachado. Por exemplo, veja o código fonte da função `mean()`

```
mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x569b8b8>
<environment: namespace:base>
```

Agora veja o código fonte da função `mean.default`, que é o método específico para vetores numéricos

```
mean.default
function (x, trim = 0, na.rm = FALSE, ...)
{
    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
        warning("argument is not numeric or logical: returning NA")
        return(NA_real_)
    }
    if (na.rm)
        x <- x[!is.na(x)]
    if (!is.numeric(trim) || length(trim) != 1L)
        stop("'trim' must be numeric of length one")
    n <- length(x)
    if (trim > 0 && n) {
        if (is.complex(x))
            stop("trimmed means are not defined for complex data")
        if (anyNA(x))
            return(NA_real_)
        if (trim >= 0.5)
            return(stats::median(x, na.rm = FALSE))
```

```
        lo <- floor(n * trim) + 1
        hi <- n + 1 - lo
        x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
    }
    .Internal(mean(x))
}
<bytecode: 0x569af18>
<environment: namespace:base>
```

Agora suponha que você ddeseja criar uma função que calcule a média para um objeto de uma classe diferente daquelas previamente definidas. Por exemplo, suponha que você quer que a função `mean()` retorne a média das linhas de uma matriz.

```
set.seed(1)
mat <- matrix(rnorm(50), nrow = 5)
mean(mat)
[1] 0.1004483
```

O resultado é a média de todos os elementos, e não de cada linha. Nesse caso, podemos definir nossa própria função método para fazer o cálculo que precisamos. Por exemplo:

```
mean.matrix <- function(x, ...) rowMeans(x)
```

Uma função método é sempre definida dessa forma: `<funçãogenérica>.<classe>`. Agora podemos ver novamente os métodos disponíveis para a função `mean()`

```
methods(mean)
[1] mean.Date     mean.default  mean.difftime mean.matrix   mean.POSIXct
[6] mean.POSIXlt
see '?methods' for accessing help and source code
```

e simplesmente aplicar a função genérica `mean()` à um objeto da classe `matrix` para obter o resultado que desejamos

```
class(mat)
[1] "matrix"
mean(mat)
[1]  0.09544402  0.12852087  0.06229588 -0.01993810  0.23591872
```

Esse exemplo ilustra como é simples criar funções método para diferentes classes de objetos. Poderíamos fazer o mesmo para objetos das classes `data.frame` e `list`

```
mean.data.frame <- function(x, ...) sapply(x, mean, ...)
mean.list <- function(x, ...) lapply(x, mean)
```

Aplicando em objetos dessas classes específicas, obtemos:

```
## Data frame
set.seed(1)
da <- data.frame(c1 = rnorm(10),
                 c2 = runif(10))
class(da)
[1] "data.frame"
mean(da)
       c1        c2
0.1322028 0.4183230
## Lista
set.seed(1)
dl <- list(rnorm(10), runif(50))
class(dl)
```

```
[1] "list"
mean(dl)
[[1]]
[1] 0.1322028

[[2]]
[1] 0.4946632
```

Obviamente esse processo todo é extremamente importante ao se criar novas funções no R. Podemos tanto criar uma função genérica (como a `mean()`) e diversos métodos para ela usando classes de objetos existentes, quanto (inclusive) criar novas classes e funções método para elas. Essa é uma das grandes lberdades que o método S3 de orientação à objetos permite, e possivelmente um dos motivos pelos quais é relativamente simples criar pacotes inteiros no R.