

Introdução ao R

Renato de Paula

2024-09-17

Contents

1	Prefácio	7
2	Normas de Funcionamento	9
3	R e RStudio	11
3.1	Instalação e funcionalidades básicas	11
3.2	Navegar no RStudio	12
3.3	Atalhos	13
4	R como Calculadora e Operações Aritméticas	15
4.1	O Prompt do R	15
4.2	Objetos e Variáveis	16
4.3	Operadores Aritméticos em R	19
4.4	Funções <code>print()</code> , <code>readline()</code> , <code>paste()</code> e <code>cat()</code>	26
4.5	Operadores Lógicos e Relacionais	29
4.6	Quizz	31
4.7	Exercícios	32
5	Estrutura de Dados Básicas	35
5.1	Vetor	35
5.2	Fatores	43
5.3	Matriz e Array	45
5.4	Data-frame	52
5.5	Listas	63

6 Estruturas de Seleção	67
6.1 Condicional <code>if</code>	67
6.2 Estrutura <code>if...else</code>	68
6.3 Condicional <code>else if</code>	68
6.4 A função <code>ifelse()</code>	69
6.5 Exemplos	69
6.6 Exercícios	71
7 Funções	73
7.1 Exercícios	77
8 Scripts em R	79
8.1 Exercícios	80
9 Leitura de dados	81
9.1 Leitura de Dados da Entrada do Utilizador	81
9.2 Diretório de trabalho	81
9.3 A Função <code>read.table()</code>	83
9.4 A função <code>read.csv()</code>	84
9.5 A função <code>read.csv2()</code>	84
9.6 A Função <code>read_excel()</code> do Pacote <code>readxl</code>	84
9.7 Leitura de Dados Online	85
10 Pipe	87
10.1 O operador pipe	87
10.2 Exercícios	89
11 Loop while	91
11.1 Exercícios	93
12 Loop for	95
12.1 Exercícios	97

13 Família <code>Xapply()</code>	99
13.1 Função <code>apply()</code>	100
13.2 Função <code>lapply()</code>	100
13.3 Função <code>sapply()</code>	101
13.4 Função <code>tapply()</code>	102
13.5 Exercícios	106
14 Gráficos (R base)	109
14.1 Gráfico de Barras	109
14.2 Gráfico Circular (Pizza)	112
14.3 Histograma	113
14.4 Box-plot	115
14.5 Gráfico de Dispersão	117
14.6 Gráfico de Linhas	118
15 Manipulação de dados	121
15.1 Tibbles	121
15.2 Manipulação de Dados com <code>dplyr</code>	123
16 O pacote <code>ggplot2</code>	143
17 Simulação	145
17.1 Simulação e Geração de Números Pseudoaleatórios	149
17.2 A função <code>set.seed()</code>	150
17.3 A função <code>sample()</code>	150
17.4 Exercícios	152
18 Método da transformada inversa	153
18.1 Variável aleatória discreta	153
18.2 Variável aleatória contínua	160
19 Método da aceitação-rejeição	165

20 Distribuições univariadas no R	167
20.1 Função de distribuição empírica	171
20.2 Gerando uma variável aleatória com distribuição de Poisson . . .	179
20.3 Gerando uma variável aleatória com distribuição de Uniforme . .	185
20.4 Gerando uma variável aleatória com distribuição Exponencial . .	190
20.5 Gerando uma variável aleatória com distribuição Normal	195
20.6 Exercícios	200
21 Relatórios	207
21.1 Markdown	207
21.2 R Markdown	207
22 Referências	209
23 Respostas	211
23.1 O pacote dplyr	211

Chapter 1

Prefácio

Este material, “Introdução ao R”, foi desenvolvido com o objetivo de servir como um guia acessível e prático para os alunos do curso de Laboratório de Estatística I - Introdução à Simulação da Faculdade de Ciências da Universidade de Lisboa. Reconhecendo a importância crescente da análise de dados na ciência moderna, o material aqui apresentado procura introduzir os conceitos fundamentais e as funcionalidades do R, uma ferramenta poderosa e amplamente utilizada na análise estatística.

Ao longo deste material, os leitores serão guiados por uma série de tópicos essenciais, desde a instalação do software e a navegação no ambiente RStudio, até ao manuseamento de estruturas de dados complexas e à criação de gráficos sofisticados. Cada capítulo foi estruturado de modo a proporcionar uma compreensão sólida dos conceitos abordados, combinando explicações teóricas com exemplos práticos e exercícios que reforçam a aprendizagem.

Este material foi elaborado para ir ao encontro das necessidades dos alunos, independentemente do seu nível de experiência prévia em Estatística. Seja para aqueles que estão a começar os seus estudos ou para os que já têm alguma familiaridade com o tema, o material proporciona uma abordagem estruturada que facilita a compreensão e a aplicação dos conceitos estatísticos. Acreditamos que, ao final deste curso, os alunos terão adquirido uma base sólida que lhes permitirá aplicar técnicas estatísticas em diversas áreas do conhecimento, usando o R de forma eficiente como uma ferramenta essencial para as suas análises de dados.

Chapter 2

Normas de Funcionamento

Docente: Renato de Paula

Gabinete: 6.4.21

email: rrpaula@ciencias.ulisboa.pt

Monitora: Ana Catarina Fernandes

Gabinete: a definir

email: a definir

AVALIAÇÃO

- Dois exercícios práticos a realizar em aula e dois testes.

ou

- **Exame final**
 - Nota mínima de 8.5 valores em cada um dos testes.
 - Datas de realização dos testes:
 - * Primeiro teste: 5 de novembro de 2024
 - * Segundo teste: 21 de janeiro de 2025
- **Nota final:**
 - (1): $0.10 \times (Ex1) + 0.10 \times (Ex2) + 0.40 \times (Teste1) + 0.40 \times (Teste2)$

ou

- (2): Nota do Exame final

- Exercício prático não efetuado tem cotação igual a zero.

Consulta permitida

- Teste 1: formulário com dimensão máxima igual a uma folha A4, frente e verso.
- Teste 2: formulário com dimensão máxima igual a uma folha A4, frente e verso.
- O formulário deve ser preparado individualmente e escrito à mão (não podem ser fotocópias).

Chapter 3

R e RStudio

O R é um software de código aberto desenvolvido como uma implementação gratuita da linguagem S, que foi concebida especificamente para computação estatística, programação estatística e geração de gráficos. A principal intenção era proporcionar aos usuários a capacidade de explorar dados de maneira intuitiva e interativa, utilizando representações gráficas significativas para facilitar a compreensão dos dados. O software estatístico R foi originalmente criado por Ross Ihaka e Robert Gentleman, da Universidade de Auckland, Nova Zelândia.

O R oferece um conjunto integrado de ferramentas para manipulação de dados, cálculo e visualização gráfica. Ele oferece:

- Manipulação eficiente de dados e armazenamento flexível;
- Operadores poderosos para cálculos em arrays e matrizes;
- Uma coleção abrangente, coerente e integrada de ferramentas para análise de dados;
- Recursos gráficos avançados para análise e visualização de dados, seja na tela ou em cópia impressa;
- Uma linguagem de programação robusta, intuitiva e eficaz, que inclui estruturas condicionais, loops, funções recursivas definidas pelo usuário, além de recursos de entrada e saída de dados.

3.1 Instalação e funcionalidades básicas

- A versão base do R, que inclui o conjunto fundamental de comandos e funções, pode ser descarregada diretamente do site oficial: <https://www.r-project.org/>. Após instalar o R, é altamente recomendável instalar também um ambiente de desenvolvimento integrado (IDE) para facilitar o trabalho com o código R. Um IDE permite ao utilizador escrever, guardar

e organizar scripts de código R de forma mais eficiente, além de executar comandos diretamente no Console do R e gerir configurações e saídas de forma conveniente. Uma escolha popular de IDE é o RStudio, que é gratuito e pode ser descarregado em <https://www.rstudio.com/>.

- Além do conjunto base de funções, o R oferece uma vasta gama de pacotes adicionais desenvolvidos pela comunidade de utilizadores. Estes pacotes podem ser instalados diretamente pelo Console do R ou através do menu do RStudio. Para instalar um pacote no Console, pode usar a função `install.packages("nome_do_pacote")`. É importante lembrar que, para instalar pacotes, é necessário estar ligado à internet. Para visualizar todos os pacotes já instalados no seu ambiente R, pode utilizar a função `installed.packages()`.

3.2 Navegar no RStudio

Existem quatro painéis principais no ambiente de trabalho do RStudio:

- **Editor/Scripts.** Este painel é utilizado para criar, carregar e exibir scripts de código R. Ele oferece recursos como realce de sintaxe, preenchimento automático e a capacidade de executar o código linha por linha ou em blocos, o que facilita a edição e depuração do código.
- **Console.** No Console, os comandos são executados diretamente. Ele funciona de maneira semelhante ao console padrão do R, mas com melhorias significativas, como realce de sintaxe, preenchimento automático de código e integração com os demais painéis do RStudio. O Console é o local ideal para testes rápidos e execução de comandos imediatos.
- **Environment/Histórico.** O painel “Environment” (Ambiente) exibe informações sobre os objetos actualmente carregados na sessão do R, como conjuntos de dados, funções definidas pelo utilizador e outras variáveis. Isto ajuda a gerir e visualizar o conteúdo da memória de trabalho. A aba “History” (Histórico) armazena todos os comandos executados durante a sessão, permitindo fácil recuperação e reutilização de códigos anteriores.
- **Painel Inferior Direito.** Este painel é multifuncional e contém várias abas úteis:
 - **Files** (Ficheiros): lista todos os ficheiros no diretório de trabalho actual.
 - **Plots** (Gráficos): exibe quaisquer gráficos gerados durante a análise.
 - **Packages** (Pacotes): permite visualizar os pacotes instalados e carregados na sessão.
 - **Help** (Ajuda): fornece acesso ao sistema de ajuda embutido em HTML, que oferece documentação detalhada sobre funções e pacotes.

- **Viewer** (Visualizador): utilizado para visualizar documentos HTML, PDFs e outros conteúdos dentro do RStudio.

3.3 Atalhos

- **CTRL+ENTER**: executa a(s) linha(s) de código selecionada(s) no script.
- **ALT+-**: insere o operador de atribuição (`<-`) no script.
- **CTRL+SHIFT+M**: (`%>%`) operador pipe no script.
- **CTRL+1**: move o cursor para o painel de script.
- **CTRL+2**: move o cursor para o console.
- **CTRL+ALT+I**: insere um novo “chunk” de código no R Markdown.
- **CTRL+SHIFT+K**: compila um documento R Markdown.
- **ALT+SHIFT+K**: abre uma janela com todos os atalhos de teclado disponíveis.

No MacBook, os atalhos geralmente são os mesmos, substituindo o **CTRL** por **Command** e o **ALT** por **Option**.

Chapter 4

R como Calculadora e Operações Aritméticas

A Estatística está intimamente ligada à Álgebra, especialmente no que diz respeito ao uso de matrizes e vetores para representar conjuntos de dados e variáveis. No R, estas estruturas de dados são amplamente utilizadas para facilitar a manipulação e a análise estatística. Por isso, é importante primeiro entender como o R lida com estas estruturas antes de avançar para comandos estatísticos mais específicos.

4.1 O Prompt do R

O R utiliza uma interface de linha de comando para executar operações e aceitar comandos diretamente. Esta interface é marcada pelo símbolo `>`, conhecido como o **prompt**. Quando se digita um comando após o prompt e se pressiona Enter, o R interpreta o comando, executa a operação correspondente e exibe o resultado no ecrã.

```
print("Meu primeiro comando no R!")  
## [1] "Meu primeiro comando no R!"
```

Nas notas de aula, o código R digitado no console é exibido em caixas cinzentas. Quando se vê o símbolo `##` no início de uma linha, isso indica o resultado (output) gerado pelo console R após a execução do código.

No R, o caractere `#` é utilizado para inserir comentários no código. Qualquer texto que aparece após `#` na mesma linha é ignorado pelo R durante a execução. Comentários são úteis para adicionar explicações ou anotações no código, como no exemplo abaixo:

```
# Este é um exemplo de comentário
2 + 2 # O R vai calcular 2 mais 2
## [1] 4
```

Se conhecer o nome de uma função e desejar aprender mais sobre o seu funcionamento, pode utilizar o comando `?` seguido do nome da função para aceder à documentação de ajuda correspondente. Por exemplo:

```
?sum
```

Este comando abrirá a página de ajuda para a função `sum`, que é utilizada para calcular a soma de elementos.

Além disso, o R oferece uma forma prática de visualizar exemplos de como uma função pode ser utilizada. Para ver exemplos de uso de uma função, pode usar o comando `example()` com o nome da função como argumento:

```
example(sum)
```

Este comando mostrará exemplos de aplicação da função `sum`, ajudando a entender melhor como pode ser usada em diferentes contextos.

4.2 Objetos e Variáveis

Em R, um **objeto** é uma unidade de armazenamento que pode conter diferentes tipos de dados ou funções, sendo referenciado por um nome. Esses dados podem incluir números, caracteres, vetores, matrizes, data frames, listas ou até mesmo funções. Objetos são criados e manipulados através de comandos, permitindo que seus valores sejam reutilizados em qualquer parte do código. Em resumo, tudo o que é criado ou carregado na sessão do R, como dados ou funções, é considerado um objeto.

4.2.1 O que é uma Variável?

Uma **variável** é um nome ou identificador associado a um objeto. Quando se cria uma variável, está-se, na verdade, a criar um “rótulo” que aponta para o objeto armazenado na memória. Assim, uma variável é o nome que se utiliza para aceder aos dados ou funções armazenados no objeto. Ela permite manipular e referenciar o objeto de forma conveniente ao longo do seu script ou análise.

4.2.2 Atribuições

- A expressão `x <- 10` cria uma variável chamada `x` e atribui-lhe o valor 10. Note que o operador de atribuição `<-` atribui o valor do lado direito à variável do lado esquerdo. O lado esquerdo deve conter apenas um único nome de variável.
- Também é possível realizar atribuições usando o sinal de igualdade `=` ou o operador `->`. No entanto, para evitar confusões entre o operador de atribuição e o operador de igualdade, é recomendável usar `<-` para atribuições.

```
# Atribuição correta
```

```
a <- 10
```

```
b <- a + 1
```

```
# Atribuição incorreta
```

```
10 = a
```

```
a + 2 = 10 # Uma atribuição não é uma equação
```

- O comando `c(1,2,3,4,5)` combina os números 1, 2, 3, 4 e 5 em um vetor, criando uma sequência de valores.
- Vetores podem ser atribuídos a objetos. Por exemplo:

```
X <- c(2,12,22,32)
```

Esta linha de código atribui um vetor numérico de comprimento 4 ao objeto `X`. É importante lembrar que o R distingue entre maiúsculas e minúsculas, o que significa que `X` e `x` são considerados dois objetos distintos.

Ao definir objetos no console, está a modificar o espaço de trabalho actual. Para visualizar todas as variáveis e objetos actualmente guardados no seu espaço de trabalho, pode usar o comando:

```
ls()
```

No RStudio, a aba *Environment* exhibe todos os objetos e valores presentes no espaço de trabalho.

4.2.3 Regras para definição de variáveis

Os nomes de variáveis em R devem começar com uma letra ou um ponto final (desde que o ponto final seja seguido por uma letra) e podem incluir letras, números, pontos e sublinhados.

- O nome de uma variável **não pode** conter espaços ou caracteres especiais (como @, #, \$, %). Apenas letras, números, pontos e sublinhados (`_`) são permitidos. Exemplo de nome válido: `nome_cliente2`.
- Ao definir nomes de variáveis, **não é permitido** usar palavras reservadas do R. Palavras reservadas são termos que têm um significado especial no R e não podem ser redefinidos. Exemplos de palavras reservadas incluem: `if`, `else`, `for`, `while`, `class`, `FALSE`, `TRUE`, `exp`, `sum`.
- O R diferencia letras maiúsculas de minúsculas, o que significa que `fcu1` e `Fcu1` são tratadas como variáveis diferentes. Uma convenção comum é usar letras minúsculas para nomes de variáveis e separar palavras com sublinhados. Exemplo: `faculdade_de_ciencias`.
- Escolha nomes de variáveis que descrevam claramente a sua finalidade para que o código seja mais legível e compreensível. Por exemplo, use `nome` em vez de `x`.

```
idade <- 20
Idade <- 30
```

Neste exemplo, `idade` e `Idade` são duas variáveis diferentes devido à diferenciação entre letras maiúsculas e minúsculas.

4.2.4 Tipos de Dados

Em R, variáveis podem armazenar diferentes tipos de dados, incluindo:

- **Numeric**: números inteiros ou decimais. Exemplo: `42`, `3.14`.
- **Character**: sequências de caracteres (texto). Exemplo: `"Olá"`.
- **Logical**: valores booleanos que representam verdadeiro ou falso. Exemplo: `TRUE`, `FALSE`.
- **Vectors**: coleções de elementos do mesmo tipo. Exemplos: `c(1, 2, 3)` para números, `c("a", "b", "c")` para caracteres.
- **Data Frames**: estruturas de dados tabulares que contêm linhas e colunas, semelhantes a uma tabela de banco de dados ou a uma folha de cálculo.
- **Lists**: coleções que podem conter elementos de diferentes tipos, como números, caracteres, vetores, e até mesmo outros data frames.
- **Factors**: variáveis categóricas que representam dados categóricos e são armazenadas como inteiros. São especialmente úteis para análises estatísticas que envolvem dados categóricos.

```
# Numeric
a <- 3.14
```

```
# Character
b <- "Programação R"

# Logical
c <- 3 < 2

# Vectors
d <- c(1, 2, 3)
```

4.2.5 Comandos Importantes

Abaixo estão alguns comandos úteis para manipular variáveis e objetos no R:

```
ls() # Exibe a lista de variáveis atualmente armazenadas na memória
ls.str() # Mostra a estrutura das variáveis armazenadas na memória
rm(a) # Remove o objeto 'a' da memória
rm(list=ls()) # Remove todos os objetos da memória
save.image('nome-do-arquivo.RData') # Salva o espaço de trabalho atual em um arquivo .RData
```

4.3 Operadores Aritméticos em R

Operador	Descrição	Exemplo
+	Adiciona dois valores	5 + 2 resulta em 7
-	Subtrai dois valores	5 - 2 resulta em 3
*	Multiplica dois valores	5 * 2 resulta em 10
/	Divide dois valores (sem arredondamento)	5 / 2 resulta em 2.5
%%/%	Realiza divisão inteira	5 %/% 2 resulta em 2
%%	Retorna o resto da divisão	5 %% 2 resulta em 1
^	Realiza exponenciação	5 ^ 2 resulta em 25

Exemplos:

```
1+1
## [1] 2
```

```

5-2
## [1] 3

5*21
## [1] 105

sqrt(9)
## [1] 3

3^3
## [1] 27

3**3
## [1] 27

log(9)
## [1] 2.197225

log10(9)
## [1] 0.9542425

exp(1)
## [1] 2.718282

# prioridade de resolução
19 + 26 / 4 - 2 * 10
## [1] 5.5

((19 + 26) / (4 - 2)) * 10
## [1] 225

```

Ao contrário de funções simples como `ls()`, a maioria das funções no R requer um ou mais *argumentos*. Nos exemplos acima, utilizamos funções predefinidas do R como `sqrt()`, `log()`, `log10()` e `exp()`, que aceitam argumentos específicos.

4.3.1 Controle da Quantidade de Dígitos Mostrados

O R permite ajustar a precisão dos números exibidos alterando a configuração global de dígitos. Veja o exemplo a seguir:

```

exp(1)
## [1] 2.718282

```

```
options(digits = 20)
exp(1)
## [1] 2.7182818284590450908

options(digits = 3)
exp(1)
## [1] 2.72
```

4.3.2 Objetos Predefinidos, Infinito, Indefinido e Valores Ausentes

O R inclui diversos conjuntos de dados predefinidos que podem ser usados para prática e teste de funções. Para visualizar todos os conjuntos de dados disponíveis, pode-se utilizar o seguinte comando:

```
data()
```

Este comando exibe uma lista de nomes de objetos para cada conjunto de dados disponível. Esses conjuntos de dados podem ser utilizados diretamente ao simplesmente digitar o nome no console. Por exemplo, ao digitar:

```
co2
```

O R exibirá os dados de concentração atmosférica de CO2 coletados em Mauna Loa.

Além dos conjuntos de dados, o R também possui objetos predefinidos que representam constantes matemáticas, como `pi` para o número π e `Inf` para o ∞ .

```
pi
## [1] 3.14

1/0
## [1] Inf

2*Inf
## [1] Inf

-1/0
## [1] -Inf

0/0
## [1] NaN
```

```

0*Inf
## [1] NaN

Inf - Inf
## [1] NaN

sqrt(-1)
## Warning in sqrt(-1): NaNs produced
## [1] NaN

c(1,2,3,NA,5)
## [1] 1 2 3 NA 5

mean(c(1,2,3,NA,5))
## [1] NA

mean(c(1,2,3,NA,5), na.rm = TRUE)
## [1] 2.75

x <- c(1, 2, NaN, 4, 5)
y <- c(1, 2, NA, 4, 5)

# Note que isso não funciona
y == NA
## [1] NA NA NA NA NA

# E isso também não
y == "NA"
## [1] FALSE FALSE NA FALSE FALSE

is.na(x)
## [1] FALSE FALSE TRUE FALSE FALSE

is.nan(x)
## [1] FALSE FALSE TRUE FALSE FALSE

is.na(y)
## [1] FALSE FALSE TRUE FALSE FALSE

is.nan(y)
## [1] FALSE FALSE FALSE FALSE FALSE

# Operações com NaN e NA
sum(x) # Retorna: NaN, porque a soma envolve um NaN
## [1] NaN

```

```
sum(y) # Retorna: NA, porque a soma envolve um NA
## [1] NA

sum(x, na.rm = TRUE) # Retorna: 12, ignora NaN na soma
## [1] 12

sum(y, na.rm = TRUE) # Retorna: 12, ignora NA na soma
## [1] 12
```

- **NaN** (Not a Number): Representa resultados indefinidos de operações matemáticas. Por exemplo, operações como `0/0` ou `sqrt(-1)` geram um NaN porque o resultado não é um número real. No R, NaN é tecnicamente um tipo especial de NA que indica especificamente um resultado numérico indefinido.
- **NA** (Not Available): Indica dados ausentes ou valores que não estão disponíveis num conjunto de dados. Por exemplo, em um vetor de dados, se um valor está ausente ou não foi medido, ele é representado por NA. Ao contrário de NaN, NA é utilizado para representar qualquer tipo de dado ausente, não apenas valores numéricos.

4.3.3 Lidando com NaN e NA em Operações

Ao trabalhar com dados, é importante saber como lidar com NaN e NA para evitar erros inesperados. Funções como `mean()` e `sum()` podem retornar NA ou NaN se contiverem esses valores. Para ignorar NA ou NaN ao realizar cálculos, você pode usar o argumento `na.rm = TRUE`, que remove os valores ausentes ou indefinidos da operação.

```
mean(c(1, 2, 3, NA, 5), na.rm = TRUE) # Calcula a média ignorando NA
## [1] 2.75

sum(x, na.rm = TRUE) # Soma ignorando NaN
## [1] 12

sum(y, na.rm = TRUE) # Soma ignorando NA
## [1] 12
```

4.3.4 Funções Úteis para Detectar NaN e NA

Para verificar a presença de NA ou NaN em um vetor ou conjunto de dados, pode-se usar as funções `is.na()` e `is.nan()`. A função `is.na()` identifica todos os

valores que são NA ou NaN, enquanto `is.nan()` identifica especificamente valores que são NaN. Por exemplo:

```
is.na(x)
## [1] FALSE FALSE TRUE FALSE FALSE

is.nan(x)
## [1] FALSE FALSE TRUE FALSE FALSE

is.na(y)
## [1] FALSE FALSE TRUE FALSE FALSE

is.nan(y)
## [1] FALSE FALSE FALSE FALSE FALSE
```

Essas funções são úteis para limpar e preparar dados antes de realizar análises estatísticas, garantindo que os cálculos sejam precisos e significativos.

4.3.5 Tipagem Dinâmica em R

Em R, o tipo de uma variável é determinado dinamicamente com base no valor atribuído a ela. Isso significa que o R automaticamente define o tipo de dado de uma variável quando lhe é atribuído um valor.

```
x <- 5
class(x)
## [1] "numeric"

y <- "Cinco"
class(y)
## [1] "character"

z <- TRUE
class(z)
## [1] "logical"
```

- A função `class()` retorna a **classe** de um objeto em R. A classe de um objeto determina como ele será tratado pelas funções e operações que podem ser aplicadas a ele. Por exemplo, vetores, matrizes, data frames e listas são diferentes classes de objetos em R, cada uma com suas próprias características e métodos.
- A função `typeof()` em R é usada para retornar o **tipo de armazenamento interno** de um objeto. Ela fornece informações detalhadas sobre como os dados são representados na memória do computador.


```
x <- 1:10
class(x)
## [1] "integer"

typeof(x)
## [1] "integer"

y <- c(1.1, 2.2, 3.3)
class(y)
## [1] "numeric"

typeof(y)
## [1] "double"

z <- data.frame(a = 1:3, b = c("A", "B", "C"))
class(z)
## [1] "data.frame"

typeof(z)
## [1] "list"

w <- list(a = 1, b = "text")
class(w)
## [1] "list"

typeof(w)
## [1] "list"
```

Neste exemplo, `class(x)` e `typeof(x)` ambos retornam “integer” para um vetor de inteiros, enquanto `class(y)` retorna “numeric” para um vetor de números de ponto flutuante, e `typeof(y)` retorna “double”, mostrando o tipo específico de armazenamento na memória. Para um `data.frame`, `class(z)` retorna “data.frame”, enquanto `typeof(z)` retorna “list”, indicando que os `data.frames` são armazenados internamente como listas.

4.3.6 Conversão entre Tipos de Dados

Em R, é possível converter uma variável de um tipo de dado para outro usando funções de conversão. Isso é especialmente útil ao trabalhar com dados que podem ter sido importados de fontes externas e precisam ser manipulados ou analisados de diferentes maneiras.

```
# Convertendo um inteiro em uma string (caractere)
a <- 15
```

```

b <- as.character(15)
print(b)
## [1] "15"

# Convertendo um número de ponto flutuante (float) em um inteiro
x <- 1.5
y <- as.integer(x)
print(y)
## [1] 1

# Convertendo uma string em um número (float)
z <- "10"
w <- as.numeric(z)
print(w)
## [1] 10

```

Essas funções de conversão são essenciais quando há necessidade de manipular diferentes tipos de dados de forma eficiente em análises estatísticas e outras operações de programação.

4.4 Funções `print()`, `readline()`, `paste()` e `cat()`

No R, existem várias funções úteis para exibir, receber e concatenar informações. Aqui estão algumas das mais comuns:

- `print()`: Utilizada para exibir valores e resultados de expressões no console. É a função básica para mostrar a saída de dados no R.
- `readline()`: Usada para receber entradas do usuário via teclado. Esta função permite que o script pause e aguarde a entrada do usuário.
- `paste()`: Utilizada para concatenar (combinar) sequências de caracteres (strings) com um separador específico entre elas. Por padrão, o separador é um espaço.
- `paste0()`: Semelhante a `paste()`, mas concatena strings sem qualquer separador.
- `cat()`: Usada para concatenar e exibir uma ou mais strings ou valores de uma forma direta, sem estruturas de formatação adicionais como aspas. Ao contrário de `print()`, `cat()` não retorna o resultado em uma nova linha.

Exemplo 1:

```
nome1 <- "faculdade"
nome2 <- "ciências"
print(paste(nome1, nome2))
## [1] "faculdade ciências"
```

Neste exemplo, `paste()` concatena as duas strings com um espaço entre elas.

Exemplo 2:

```
# Solicitar entrada do usuário
n <- readline(prompt = "Digite um número: ")

# Converta a entrada em um valor numérico
n <- as.integer(n)

# Imprima o valor no console
print(n+1)
```

Aqui, `readline()` recebe a entrada do utilizador, e `as.integer()` converte essa entrada para um número inteiro. O resultado é incrementado em 1 e exibido.

Exemplo 3:

```
# Solicitar entrada do utilizador
nome <- readline(prompt = "Entre com o seu nome: ")

# Imprima uma mensagem de saudação
cat("Olá, ", nome, "!\n")
```

O uso de `cat()` aqui é para exibir uma mensagem de saudação que inclui o nome do utilizador.

Exemplo 4:

```
# Solicitar ao utilizador a entrada numérica
idade <- readline(prompt = "Digite a sua idade: ")

# Converta a entrada em um valor numérico
idade <- as.numeric(idade)

# Verifique se a entrada é numérica
if (is.na(idade)) {
  cat("Entrada inválida. Insira um valor numérico.\n")
} else {
  cat("Você tem ", idade, " anos.\n")
}
```

Este exemplo mostra como verificar se a entrada é numérica usando `is.na()` e fornecer feedback apropriado ao utilizador.

Concatenando Strings

```
result <- paste("Hello", "World")
print(result)
## [1] "Hello World"
```

Concatenando Múltiplas Strings

```
result <- paste("Data", "Science", "with", "R")
print(result)
## [1] "Data Science with R"
```

Concatenando Strings com um Separador Específico

```
result <- paste("2024", "04", "28", sep="-")
print(result)
## [1] "2024-04-28"
```

Concatenando Vetores de Strings

```
first_names <- c("Anna", "Bruno", "Carlos")
last_names <- c("Smith", "Oliveira", "Santos")
result <- paste(first_names, last_names)
print(result)
## [1] "Anna Smith"      "Bruno Oliveira" "Carlos Santos"
```

Concatene com cada elemento de um vetor

```
numbers <- 1:3
result <- paste("Number", numbers)
print(result)
## [1] "Number 1" "Number 2" "Number 3"
```

Usando `paste0()` para Concatenar sem Espaço

```
result <- paste0("Hello", "World")
print(result)
## [1] "HelloWorld"
```

Concatenando Strings com Números

```
age <- 25
result <- paste("I am", age, "years old")
print(result)
## [1] "I am 25 years old"
```

4.5 Operadores Lógicos e Relacionais

Em R, operadores lógicos e relacionais são utilizados para realizar comparações e tomar decisões com base nos resultados dessas comparações. Esses operadores são fundamentais para a construção de estruturas de controle de fluxo, como instruções condicionais (`if`, `else`) e loops (`for`, `while`).

4.5.1 Operadores Lógicos

Os operadores lógicos são usados para combinar ou modificar condições lógicas, retornando valores booleanos (`TRUE` ou `FALSE`).

- `&` (E lógico): Retorna `TRUE` se **todas** as expressões forem verdadeiras.
- `|` (OU lógico): Retorna `TRUE` se **pelo menos uma** das expressões for verdadeira.
- `!` (Não lógico): Inverte o valor de uma expressão booleana, transformando `TRUE` em `FALSE` e vice-versa.

Exemplos:

```
(5 > 3) & (4 > 2)    # Ambas as condições são verdadeiras
## [1] TRUE

(5 < 3) | (4 > 2)    # Apenas uma condição é verdadeira
## [1] TRUE

!(5 > 3)             # Inverte o valor lógico de TRUE para FALSE
## [1] FALSE
```

4.5.2 Operadores Relacionais

Os operadores relacionais são usados para comparar valores e retornam valores lógicos (`TRUE` ou `FALSE`) com base na comparação.

- `a == b`: Verifica se “a” é igual a “b”.

- `a != b`: Verifica se “a” é diferente de “b”.
- `a > b`: Verifica se “a” é maior que “b”.
- `a < b`: Verifica se “a” é menor que “b”.
- `a >= b`: Verifica se “a” é maior ou igual a “b”.
- `a <= b`: Verifica se “a” é menor ou igual a “b”.
- `is.na(a)`: Verifica se “a” é um valor ausente (NA).
- `is.null(a)`: Verifica se “a” é nulo (NULL).

Exemplos:

```

# Maior que
2 > 1
## [1] TRUE

1 > 2
## [1] FALSE

# Menor que
1 < 2
## [1] TRUE

# Maior ou igual a
0 >= (2+(-2))
## [1] TRUE

# Menor ou igual a
1 <= 3
## [1] TRUE

# Conjunção E (ambas as condições devem ser verdadeiras)
9 > 11 & 0 < 1
## [1] FALSE

# Disjunção OU (pelo menos uma condição deve ser verdadeira)
6 < 5 | 0 > -1
## [1] TRUE

# Igual a
1 == 2/2
## [1] TRUE

# Diferente de
1 != 2
## [1] TRUE

```

4.6 Quizz

1. Qual operador relacional verifica se dois valores são iguais?

- a) =
- b) ==
- c) !=
- d) <=

2. Qual operador verifica se um valor é maior que outro?

- a) >=
- b) <
- c) >
- d) <=

3. Qual o resultado de `5 != 3`?

- a) TRUE
- b) FALSE
- c) NA
- d) NULL

4. O que `x & y` faz em R, onde `x` e `y` são lógicos?

- a) Realiza uma operação “OR”.
- b) Verifica se ambos são TRUE.
- c) Verifica se ao menos um é TRUE.
- d) Inverte o valor lógico de `x`.

5. O que faz o operador `|` em expressões lógicas?

- a) Verifica se ambos os operandos são verdadeiros.
- b) Verifica se ao menos um dos operandos é verdadeiro.
- c) Verifica se ambos os operandos são falsos.
- d) Compara se os operandos são iguais.

6. O que faz o operador `!` em R?

- a) Inverte um valor lógico
- b) Multiplica dois valores
- c) Compara dois valores

- d) Calcula o resto da divisão

7. Qual o resultado de `!(TRUE)`?

- a) TRUE
- b) FALSE
- c) NA
- d) NULL

4.7 Exercícios

1. Escreva um programa em R que leia dois inteiros inseridos pelo utilizador e imprima:

- A soma dos dois números.
- O produto dos dois números.
- A diferença entre o primeiro e o segundo número.
- A divisão do primeiro pelo segundo número.
- O resto da divisão do primeiro pelo segundo.
- O resultado do primeiro número elevado à potência do segundo.

Dica: Use as funções `readline()` para entrada de dados e `as.integer()` para conversão de tipos.

2. Escreva um programa em R que leia dois números de ponto flutuante (números decimais) e imprima:

- A soma dos dois números.
- A diferença entre os dois números.
- O produto dos dois números.
- O resultado do primeiro número elevado à potência do segundo.

Dica: Use `as.numeric()` para converter a entrada para números de ponto flutuante.

3. Escreva um programa em R que leia uma distância em milhas inserida pelo utilizador e a converta para quilómetros usando a fórmula: $K = M * 1.609344$.

Dica: Lembre-se de usar `as.numeric()` para converter a entrada para um número.

4. Escreva um programa em R que leia três inteiros correspondentes ao comprimento, largura e altura de um paralelepípedo, e imprima seu volume.

5. Escreva um programa em R que leia três inteiros e imprima a média dos três números.

6. Escreva um programa em R que leia uma temperatura em graus Fahrenheit e a converta para graus Celsius usando a fórmula:: $C = \frac{F-32}{1.8}$.
7. Escreva um programa em R que leia uma hora no formato de 24 horas e imprima a hora correspondente no formato de 12 horas.
8. Está a olhar para um relógio e são exatamente 14h. Definiu um alarme para tocar dentro de 51 horas. A que horas o alarme tocará?
9. Escreva um programa em R que resolva a versão geral do problema acima. Peça ao utilizador para inserir a hora atual (em horas) e o número de horas de espera antes que o alarme toque. O programa deve imprimir a hora em que o alarme tocará.
10. Escreva um programa em R que leia um número inteiro fornecido pelo utilizador e verifique se esse número é maior que 10. O programa deve imprimir **TRUE** se o número for maior que 10 ou **FALSE** caso contrário.
11. Escreva um programa em R que leia dois números fornecidos pelo utilizador e verifique se eles são iguais. O programa deve imprimir **TRUE** se os números são iguais ou **FALSE** caso contrário.
12. Escreva um programa em R que peça ao utilizador para inserir dois números e verifique se o primeiro número é maior ou igual ao segundo. O programa deve imprimir **TRUE** ou **FALSE**.
13. Escreva um programa em R que peça ao utilizador para inserir um número e verifique se esse número está entre 0 e 100, inclusive. O programa deve imprimir **TRUE** se o número está no intervalo e **FALSE** caso contrário.
14. Escreva um programa em R que leia três números fornecidos pelo utilizador e verifique se o primeiro número é menor que o segundo e se o segundo é menor que o terceiro. O programa deve imprimir uma mensagem indicando se a condição é verdadeira ou falsa.

Chapter 5

Estrutura de Dados Básicas

Em R, temos dois tipos principais de objetos: funções e dados.

- **Funções:** São objetos que executam operações específicas.
 - Exemplos de funções:
 - * `cos()` - calcula o cosseno de um ângulo.
 - * `print()` - imprime valores no console.
 - * `plot()` - cria gráficos.
 - * `integrate()` - calcula a integral de uma função.
- **Dados:** São objetos que contêm informações, como números, textos ou outros tipos de dados.
 - Exemplos de dados:
 - * 23 (número)
 - * "Hello" (texto ou string)
 - * TRUE (valor lógico)
 - * `c(1, 2, 3)` (vetor numérico)
 - * `data.frame(nome = c("Alice", "Bob"), idade = c(25, 30))` (estrutura tabular)
 - * `list(numero = 42, nome = "Alice", flag = TRUE)` (coleção de elementos de diferentes tipos)
 - * `factor(c("homem", "mulher", "mulher", "homem"))` (dados categóricos)

5.1 Vetor

Um **vetor** é uma estrutura de dados básica que armazena uma sequência de elementos do **mesmo tipo**. Os vetores podem conter dados numéricos, caracteres,

valores lógicos (TRUE/FALSE), números complexos, entre outros.

- Todos os elementos de um vetor devem ser do mesmo tipo.
- Os elementos de um vetor são indexados a partir de 1 (ou seja, o primeiro elemento está na posição 1).
- Os vetores podem ser criados usando a função `c()` (concatenate) e são facilmente manipulados com uma variedade de funções.

5.1.1 Tipos Comuns de Vetores

```
# Vetor numérico
c(1.1, 2.2, 3.3)
## [1] 1.1 2.2 3.3

# Vetor de caracteres
c("a", "b", "c")
## [1] "a" "b" "c"
# ou
c('a', 'b', 'c')
## [1] "a" "b" "c"

# Vetor lógico
c(TRUE, 1==2)
## [1] TRUE FALSE

# Não podemos misturar tipos de dados em um vetor...
c(3, 1==2, "a") # Observe que o R converteu tudo para "character"!
## [1] "3"      "FALSE" "a"
```

5.1.2 Construindo Vetores

```
# Inteiros de 1 a 10
x <- 1:10
x
## [1] 1 2 3 4 5 6 7 8 9 10

b <- 10:1
b
## [1] 10 9 8 7 6 5 4 3 2 1

# Sequência de 0 a 50 com incrementos de 10
```

```

a <- seq(from = 0, to = 50, by=10)
a
## [1] 0 10 20 30 40 50

# Sequência de 15 números de 0 a 1
y <- seq(0,1, length=15)
y
## [1] 0.0000 0.0714 0.1429 0.2143 0.2857 0.3571 0.4286 0.5000 0.5714 0.6429
## [11] 0.7143 0.7857 0.8571 0.9286 1.0000

# Repetição de um vetor várias vezes
z <- rep(1:3, times=4)
z
## [1] 1 2 3 1 2 3 1 2 3 1 2 3

# Repetição de cada elemento do vetor várias vezes
t <- rep(1:3, each=4)
t
## [1] 1 1 1 1 2 2 2 2 3 3 3 3

# Combine números, vetores ou ambos em um novo vetor
w <- c(x,z,5)
w
## [1] 1 2 3 4 5 6 7 8 9 10 1 2 3 1 2 3 1 2 3 1 2 3 5

```

5.1.3 Acesso a Elementos de um Vetor

Pode aceder a elementos específicos de um vetor utilizando colchetes `[]` e índices. O R utiliza indexação baseada em 1, o que significa que o primeiro elemento de um vetor tem índice 1.

```

# Defina um vetor com inteiros de (-5) a 5 e extraia os números com valor absoluto menor que 3:
x <- (-5):5
x
## [1] -5 -4 -3 -2 -1 0 1 2 3 4 5

# Acesso por índice:
x[4:8]
## [1] -2 -1 0 1 2

# Seleção negativa (excluindo elementos):
x[-c(1:3,9:11)]
## [1] -2 -1 0 1 2

```

```
# Todos menos o último
x[-length(x)]
## [1] -5 -4 -3 -2 -1  0  1  2  3  4

# Vetor lógico para seleção
index <- abs(x)<3
index
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE

# Utilizando o vetor lógico para extrair elementos desejados:
x[index]
## [1] -2 -1  0  1  2

# Ou de forma compacta:
x[abs(x) < 3]
## [1] -2 -1  0  1  2

# Acesso a elementos com vetores predefinidos
letters[1:3]
## [1] "a" "b" "c"

letters[c(2,4,6)]
## [1] "b" "d" "f"

LETTERS[1:3]
## [1] "A" "B" "C"

y <- 1:10
y[ (y>5) ] # Seleciona qualquer número > 5
## [1]  6  7  8  9 10

y[ (y%%2==0) ] # Números divisíveis por 2
## [1]  2  4  6  8 10

y[ (y%%2==1) ] # Números não divisíveis por 2
## [1]  1  3  5  7  9

y[5] <- NA
y[!is.na(y)] # Todos os valores de y que não são NA
## [1]  1  2  3  4  6  7  8  9 10
```

5.1.4 Funções Comuns para Vetores

Os vetores são uma das estruturas de dados mais utilizadas no R, e existem diversas funções para manipular e obter informações sobre eles. Abaixo estão algumas das funções mais comuns usadas com vetores numéricos:

```
num_vector <- c(2.2, 1.1, 3.3)

# Obtém o comprimento (número de elementos) de um vetor
length(num_vector)
## [1] 3

# Calcula o valor máximo de um vetor
max(num_vector)
## [1] 3.3

# Calcula o valor mínimo de um vetor
min(num_vector)
## [1] 1.1

# Calcula a soma dos elementos de um vetor
sum(num_vector)
## [1] 6.6

# Calcula a média (valor médio) dos elementos de um vetor
mean(num_vector)
## [1] 2.2

# Calcula a mediana dos elementos de um vetor
median(num_vector)
## [1] 2.2

# Retorna um vetor contendo o mínimo e o máximo
range(num_vector)
## [1] 1.1 3.3

# Calcula a variância amostral dos elementos de um vetor
var(num_vector)
## [1] 1.21

# Calcula os quantis dos elementos de um vetor
quantile(num_vector, type = 2)
## 0% 25% 50% 75% 100%
## 1.1 1.1 2.2 3.3 3.3

# Calcula a soma cumulativa dos elementos de um vetor
```

```
cumsum(num_vector)
## [1] 2.2 3.3 6.6

# Calcula o produto cumulativo dos elementos de um vetor
cumprod(num_vector)
## [1] 2.20 2.42 7.99

# Ordena os elementos de um vetor em ordem crescente
sort(num_vector)
## [1] 1.1 2.2 3.3

# Ordena os elementos de um vetor em ordem decrescente
sort(num_vector, decreasing = TRUE)
## [1] 3.3 2.2 1.1

# Remove elementos duplicados de um vetor
duplicate_vector <- c(1, 2, 2, 3, 3, 3)
unique(duplicate_vector)
## [1] 1 2 3
```

A função `which` é utilizada para encontrar os índices dos elementos de um vetor que satisfazem uma condição específica. Isto é útil quando se quer localizar a posição de certos valores dentro de um vetor.

```
y <- c(8, 3, 5, 7, 6, 6, 8, 9, 2, 3, 9, 4, 10, 4, 11)

# Encontrar os índices dos elementos que são maiores que 5
which(y > 5)
## [1] 1 4 5 6 7 8 11 13 15
```

Aqui, a função `which(y > 5)` retorna os índices dos elementos em `y` que são maiores que 5. Se quiser ver os valores em `y` que são maiores que 5, basta fazer:

```
y[y>5]
## [1] 8 7 6 6 8 9 9 10 11
```

5.1.5 Operações com Vetores

Os vetores no R suportam operações aritméticas e lógicas de forma elementar. Isto significa que as operações são aplicadas a cada elemento do vetor.

```
# Adição de 1 a cada elemento do vetor
num_vector + 1
```



```
## [1] 3.2 2.1 4.3

# Multiplicação de cada elemento por 2
num_vector * 2
## [1] 4.4 2.2 6.6

# Comparações: verifica se cada elemento é maior que 2
num_vector > 2
## [1] TRUE FALSE TRUE

# Exponenciação de elementos
c(2, 3, 5, 7)^2
## [1] 4 9 25 49

c(2, 3, 5, 7)^c(2, 3)
## [1] 4 27 25 343

c(1, 2, 3, 4, 5, 6)^c(2, 3, 4)
## [1] 1 8 81 16 125 1296

c(2, 3, 5, 7)^c(2, 3, 4)
## [1] 4 27 625 49
```

Os exemplos acima ilustram a **propriedade de reciclagem** do R. Quando operações são realizadas entre vetores de diferentes comprimentos, o R “recicla” (ou repete) o vetor menor até que corresponda ao comprimento do vetor maior. Se o comprimento do vetor maior não for um múltiplo inteiro do comprimento do vetor menor, o R emitirá um aviso.

Por exemplo:

```
c(2,3,5,7)^c(2,3)
## [1] 4 27 25 343
```

Este comando é expandido internamente para:

```
c(2,3,5,7)^c(2,3,2,3)
## [1] 4 27 25 343
```

No entanto, se os vetores não puderem ser “reciclados” perfeitamente, o R irá gerar um aviso:

```
c(2,3,5,7)^c(2,3,4)
## Warning in c(2, 3, 5, 7)^c(2, 3, 4): longer object length is not a multiple of
## shorter object length
## [1] 4 27 625 49
```

Neste caso, $c(2,3,5,7) \sim c(2,3,4)$ é expandido para:

```
c(2,3,5,7) ~ c(2,3,4,2)
## [1] 4 27 625 49
```

Observe que o último elemento do vetor menor foi reciclado para corresponder ao comprimento do vetor maior, mas não completamente, resultando no aviso.

5.1.6 Exercícios

1. Crie os vetores:

- (a) $(1, 2, 3, \dots, 19, 20)$
- (b) $(20, 19, \dots, 2, 1)$
- (c) $(1, 2, 3, \dots, 19, 20, 19, 18, \dots, 2, 1)$
- (d) $(10, 20, 30, \dots, 90, 10)$
- (e) $(1, 1, \dots, 1, 2, 2, \dots, 2, 3, 3, \dots, 3)$ onde existem 10 ocorrências do 1, 20 ocorrências do 2 e 30 ocorrências do 3.

2. Use a função `paste()` para criar o seguinte vetor de caracteres de tamanho 20:

("nome 1", "nome 2", ..., "nome 20")

3. Crie um vetor x_1 igual a "A" "A" "B" "B" "C" "C" "D" "D" "E" "E"

4. Crie um vetor x_2 igual a "a" "b" "c" "d" "e" "a" "b" "c" "d" "e"

5. Crie um vetor x_3 igual as palavras "uva" 10 vezes, "maçã" 9 vezes, "laranja" 6 vezes e "banana" 1 vez.

6. Crie um vetor de 15 números aleatórios entre 1 e 100 (use a função `sample()`). Ordene esse vetor em ordem crescente e depois em ordem decrescente. Encontre o menor e o maior valor no vetor.

7. Crie um vetor de 20 números aleatórios entre 1 e 50 (use a função `sample()`). Calcule a soma, a média, o desvio padrão e o produto de todos os elementos do vetor.

8. Crie um vetor de 10 números aleatórios entre 1 e 100 (use a função `sample()`). Extraia os elementos do vetor que são maiores que 50. De seguida, substitua os valores menores que 30 por 0.

9. Crie um vetor de 10 números. Verifique quais elementos são maiores que 5 e quais são pares. Crie um novo vetor que contenha apenas os números que satisfazem ambas as condições.

10. Crie um vetor com 10 números inteiros. Multiplique os elementos nas posições 2, 4 e 6 por 2. Substitua o último elemento por 100.

11. Calcule a média dos vetores:

(a) $x = (1, 0, NA, 5, 7)$

(b) $y = (-Inf, 0, 1, NA, 2, 7, Inf)$

12. Crie:

(a) um vetor com valores $e^x \sin(x)$ nos pontos $x = 2, 2.1, 2.2, \dots, 6$

(b) um vetor com valores $\left(3, \frac{3^2}{2}, \frac{3^3}{3}, \dots, \frac{3^{30}}{30}\right)$

13. Calcule:

(a)

$$\sum_{i=10}^{100} i^3 + 4i^2$$

(b)

$$\sum_{i=1}^{25} \frac{2^i}{i} + \frac{3^i}{i^2}$$

5.2 Fatores

Em R, um **fator** é uma estrutura de dados usada para representar dados **categóricos**, ou seja, dados que podem ser classificados em categorias distintas. Os fatores são amplamente utilizados em análises estatísticas e visualizações de dados, pois permitem o tratamento eficiente e consistente de variáveis categóricas.

- **Níveis:** Os fatores possuem **níveis** (ou **levels**), que representam os diferentes valores possíveis que a variável categórica pode assumir. Por exemplo, para uma variável categórica que representa tamanho de roupa, os níveis poderiam ser “Pequeno”, “Médio” e “Grande”.
- **Armazenamento Interno:** Internamente, os fatores são armazenados como inteiros, onde cada inteiro corresponde a um nível específico. No entanto, quando exibidos, os fatores mostram os seus rótulos (labels) para facilitar a compreensão.

- **Fatores Ordenados e Não Ordenados:** Os fatores podem ser **ordenados** (quando há uma ordem lógica entre os níveis, como “Baixo”, “Médio”, “Alto”) ou **não ordenados** (quando os níveis não têm uma ordem intrínseca).

Exemplos:

```
# Vetor de dados categóricos
data <- c("low", "medium", "high", "medium", "low", "high")

# Criar um fator não ordenado a partir dos dados categóricos
factor_data <- factor(data)

print(factor_data)
## [1] low    medium high    medium low    high
## Levels: high low medium
```

Por padrão, os níveis são ordenados alfabeticamente. Podemos especificar a ordem dos níveis de acordo com a lógica desejada:

```
# Especificar a ordem dos níveis do fator
factor_data <- factor(data, levels = c("low", "medium", "high"))
print(factor_data)
## [1] low    medium high    medium low    high
## Levels: low medium high
```

Para criar um fator ordenado, onde os níveis têm uma ordem específica, usamos o argumento `ordered = TRUE`:

```
# Criar um fator ordenado
ordered_factor <- factor(data, levels = c("low", "medium", "high"), ordered = TRUE)
print(ordered_factor)
## [1] low    medium high    medium low    high
## Levels: low < medium < high
```

5.2.1 Manipulação de Fatores

Podemos utilizar várias funções para verificar e modificar os níveis de um fator:

```
# Verificar Níveis
levels(factor_data)
## [1] "low"    "medium" "high"
```

```
# Modificar Níveis
levels(factor_data) <- c("Low", "Medium", "High")
print(factor_data)
## [1] Low      Medium High      Medium Low      High
## Levels: Low Medium High
```

5.2.2 Gerando Fatores com gl()

A função `gl()` (generate levels) é usada para criar fatores de maneira eficiente, especialmente quando se deseja gerar fatores com padrões repetitivos.

```
# Gerar níveis de fator com gl()
gl(4,3)
## [1] 1 1 1 2 2 2 3 3 3 4 4 4
## Levels: 1 2 3 4

# Gerar fatores com labels personalizados
gl(2, 10, labels = c("mulher", "homem"))
## [1] mulher mulher mulher mulher mulher mulher mulher mulher mulher mulher
## [11] homem homem homem homem homem homem homem homem homem homem
## Levels: mulher homem

# Também podemos fazer
as.factor(c(rep("mulher", 10), rep("homem", 10)))
## [1] mulher mulher mulher mulher mulher mulher mulher mulher mulher mulher
## [11] homem homem homem homem homem homem homem homem homem homem
## Levels: homem mulher
```

5.3 Matriz e Array

Em R, uma **matriz** é uma estrutura de dados bidimensional que contém elementos do mesmo tipo (como numérico, lógico, etc.), organizados em linhas e colunas. Já um **array** é uma generalização da matriz que pode ter mais de duas dimensões.

- **nrow**: número de linhas;
- **ncol**: número de colunas.

5.3.1 Criando matrizes

Podemos criar uma matriz em R usando a função `matrix()`. Veja o exemplo abaixo:

```
matrix(c(1,2,3,4,5,6)+exp(1),nrow=2)
##      [,1] [,2] [,3]
## [1,] 3.72 5.72 7.72
## [2,] 4.72 6.72 8.72
```

Podemos também realizar operações lógicas em matrizes:

```
# Verifica se os elementos da matriz são maiores que 6
matrix(c(1,2,3,4,5,6)+exp(1),nrow=2) > 6
##      [,1] [,2] [,3]
## [1,] FALSE FALSE TRUE
## [2,] FALSE  TRUE TRUE
```

5.3.2 Criando um array

Um array pode ter mais de duas dimensões e é criado usando a função `array()`:

```
# Criar um array com 4 linhas, 3 colunas, e 2 camadas
array(c(1:24), dim=c(4,3,2))
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
## [4,]   16   20   24
```

5.3.3 Construindo Matrizes

- `rbind()` (row bind): Combina objetos por linhas, empilhando-os verticalmente.
- `cbind()` (column bind): Combina objetos por colunas, empilhando-os horizontalmente.

Exemplo com Vetores

```
# Criar dois vetores
vector1 <- c(1, 2, 3)
vector2 <- c(4, 5, 6)

# Combinar os vetores por linhas
result <- rbind(vector1, vector2)
print(result)
##           [,1] [,2] [,3]
## vector1     1     2     3
## vector2     4     5     6

# Combinar os vetores por colunas
result <- cbind(vector1, vector2)
print(result)
##      vector1 vector2
## [1,]       1       4
## [2,]       2       5
## [3,]       3       6
```

Exemplo com Matrizes

```
# Criar duas matrizes
matrix1 <- matrix(1:6, nrow = 2, ncol = 3)
matrix2 <- matrix(7:12, nrow = 2, ncol = 3)

# Combinar as matrizes por linhas
result <- rbind(matrix1, matrix2)
print(result)
##           [,1] [,2] [,3]
## [1,]       1     3     5
## [2,]       2     4     6
## [3,]       7     9    11
## [4,]       8    10    12

# Combinar as matrizes por colunas
result <- cbind(matrix1, matrix2)
print(result)
##           [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]       1     3     5     7     9    11
## [2,]       2     4     6     8    10    12
```

5.3.4 Acessar Elementos de uma Matriz

Podemos aceder a elementos específicos de uma matriz utilizando índices ou expressões lógicas.

```

# Criar uma matriz
A <- matrix((-4):5, nrow=2, ncol=5)
A
##           [,1] [,2] [,3] [,4] [,5]
## [1,]    -4    -2     0     2     4
## [2,]    -3    -1     1     3     5

# Aceder a um elemento específico
A[1,2]
## [1] -2

# Selecionar elementos negativos
A[A<0]
## [1] -4 -3 -2 -1

# Atribuição condicional
A[A<0]<-0
A
##           [,1] [,2] [,3] [,4] [,5]
## [1,]      0     0     0     2     4
## [2,]      0     0     1     3     5

# Selecionar uma linha específica
A[2,]
## [1] 0 0 1 3 5

# Selecionar colunas específicas
A[,c(2,4)]
##           [,1] [,2]
## [1,]      0     2
## [2,]      0     3

```

5.3.5 Nomear Linhas e Colunas de uma Matriz

```

# Criar uma matriz
x <- matrix(rnorm(12),nrow=4)
x
##           [,1] [,2] [,3]
## [1,] -0.508 -0.523 -0.0258
## [2,]  1.864  2.422  0.3408
## [3,] -0.230  0.314 -1.9076
## [4,] -0.571  0.478 -0.5948

```



```

# Nomear colunas
colnames(x) <- paste("dados",1:3,sep="")
x
##      dados1 dados2 dados3
## [1,] -0.508 -0.523 -0.0258
## [2,]  1.864  2.422  0.3408
## [3,] -0.230  0.314 -1.9076
## [4,] -0.571  0.478 -0.5948

# Nomear linhas e colunas de outra matriz
y <- matrix(rnorm(15),nrow=5)
y
##      [,1] [,2] [,3]
## [1,] -0.559 -1.705  0.672
## [2,] -0.796  1.176 -0.566
## [3,]  0.855 -1.474  0.804
## [4,] -0.630 -1.786 -0.973
## [5,]  1.261  0.447  0.285

colnames(y) <- LETTERS[1:ncol(y)]
rownames(y) <- letters[1:nrow(y)]

y
##      A      B      C
## a -0.559 -1.705  0.672
## b -0.796  1.176 -0.566
## c  0.855 -1.474  0.804
## d -0.630 -1.786 -0.973
## e  1.261  0.447  0.285

```

5.3.6 Multiplicação de matrizes

```

M<-matrix(rnorm(20),nrow=4,ncol=5)
N<-matrix(rnorm(15),nrow=5,ncol=3)

# Multiplicação de matrizes
M%*%N
##      [,1] [,2] [,3]
## [1,] -1.67927  0.8103 -3.405
## [2,] -0.33112 -0.9712 -2.352
## [3,] -0.83679 -0.2961  0.140
## [4,] -0.00563 -0.0709 -0.113

```

5.3.7 Adicionar Linhas e Colunas a uma Matriz

```
X <- matrix(c(1,2,3,4,5,6),nrow=2)
X
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

# Adicionar uma coluna
cbind(X, c(7,8))
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8

# Adicionar uma linha
rbind(X, c(7,8,9))
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    7    8    9
```

5.3.8 Algumas outras funções

Seja M uma matriz quadrada.

- dimensão de uma matriz $\rightarrow \text{dim}(M)$
- transposta de uma matriz $\rightarrow \text{t}(M)$
- determinante de uma matriz $\rightarrow \text{det}(M)$
- inversa de uma matriz $\rightarrow \text{solve}(M)$
- autovalores e autovetores $\rightarrow \text{eigen}(M)$
- soma dos elementos de uma matriz $\rightarrow \text{sum}(M)$
- média dos elementos de uma matriz $\rightarrow \text{mean}(M)$
- aplicar uma função a cada linha ou coluna $\rightarrow \text{apply}(M, 1, \text{sum})$ # soma de cada linha
- aplicar uma função a cada linha ou coluna $\rightarrow \text{apply}(M, 2, \text{mean})$ # média de cada coluna

5.3.9 Exercícios

1. Crie uma matriz 3×4 com os números de 1 a 12, preenchendo a matriz por colunas. Exiba a matriz e determine a soma dos elementos da segunda coluna.
2. Crie duas matrizes 2×3 chamadas A e B , cada uma preenchida com números aleatórios inteiros de 1 a 10. Em seguida, some as duas matrizes e multiplique-as elemento por elemento.
3. Crie uma matriz 3×3 chamada M com números aleatórios entre 1 e 9. Crie também um vetor de comprimento 3 chamado v . Realize a multiplicação entre a matriz M e o vetor v (ou seja, $M \times v$).
4. Crie uma matriz 4×2 chamada N com números sequenciais de 1 a 8. Transponha a matriz e calcule a soma dos elementos de cada linha da matriz transposta.
5. Crie uma matriz 3×3 chamada P com valores de sua escolha. Calcule o determinante da matriz P . Caso o determinante seja diferente de zero, calcule também a matriz inversa de P .
6. Crie uma matriz 5×5 chamada Q com números aleatórios de 1 a 25. Extraia a submatriz composta pelas 2ª, 3ª e 4ª linhas e colunas.
7. Crie uma matriz 3×3 com valores sequenciais de 1 a 9. Calcule a soma de todos os elementos da primeira linha e a soma de todos os elementos da terceira coluna.
8. Considere a matriz

$$A = \begin{bmatrix} 1 & 1 & 3 \\ 5 & 2 & 6 \\ -2 & -1 & -3 \end{bmatrix}$$

- (a) Verifique que $A^3 = 0$.
- (b) Troque a terceira coluna pela soma da coluna 1 e coluna 3.
- (c) Considere a matriz

$$M = \begin{bmatrix} 20 & 22 & 23 \\ 34 & 55 & 57 \\ 99 & 97 & 71 \\ 12 & 16 & 19 \\ 10 & 53 & 24 \\ 14 & 21 & 28 \end{bmatrix},$$

e troque os números pares por 0.

5.4 Data-frame

Um **data frame** em R é uma estrutura de dados bidimensional usada para armazenar dados tabulares. Cada coluna num data frame pode conter valores de diferentes tipos (como numéricos, caracteres, fatores, etc.), mas todos os elementos dentro de uma coluna devem ser do mesmo tipo. Um data frame é semelhante a uma tabela em um banco de dados ou uma folha de cálculo em programas como o Excel. Os data frames podem ser criados a partir de ficheiros de dados ou convertendo vetores usando a função `as.data.frame()`.

5.4.1 Criar um Data Frame

Para criar um data frame, pode usar a função `data.frame()`, especificando os nomes das colunas e os dados correspondentes:

```
df <- data.frame(
  id = 1:4,
  nome = c("Ana", "Bruno", "Carlos", "Diana"),
  idade = c(23, 35, 31, 28),
  salario = c(5000, 6000, 7000, 8000))
df
##   id  nome idade salario
## 1  1   Ana    23   5000
## 2  2 Bruno    35   6000
## 3  3 Carlos   31   7000
## 4  4 Diana    28   8000
```

Um data frame pode parecer semelhante a uma matriz, mas há diferenças importantes. Veja o exemplo de uma matriz criada com os mesmos dados:

```
# Comparando com uma matriz
cbind(id = 1:4,
  nome = c("Ana", "Bruno", "Carlos", "Diana"),
  idade = c(23, 35, 31, 28),
  salario = c(5000, 6000, 7000, 8000))
##      id  nome      idade salario
## [1,] "1" "Ana"     "23"  "5000"
## [2,] "2" "Bruno"   "35"  "6000"
## [3,] "3" "Carlos"  "31"  "7000"
## [4,] "4" "Diana"   "28"  "8000"
```

Observe que na matriz, todos os dados são convertidos para o tipo `character`, enquanto em um data frame, cada coluna mantém seu próprio tipo de dados.

5.4.2 Aceder a Linhas e Colunas

Existem várias maneiras de aceder a linhas e colunas num data frame:

```
# Aceder à coluna 'id' usando índice
df[,1]
## [1] 1 2 3 4

# Aceder à coluna 'id' usando o nome da coluna
df$id
## [1] 1 2 3 4

# Aceder à coluna 'id' usando double brackets
df[["id"]]
## [1] 1 2 3 4

# Aceder à primeira linha
df[1, ]
##   id nome idade salario
## 1  1  Ana   23    5000

# Aceder à segunda coluna
df[, 2]
## [1] "Ana"    "Bruno"  "Carlos" "Diana"

# Aceder ao elemento na primeira linha, segunda coluna
df[1, 2]
## [1] "Ana"

# Subconjunto das primeiras duas linhas e colunas
df[1:2, 1:2]
##   id nome
## 1  1  Ana
## 2  2 Bruno

# Aceder a uma entrada por nome de coluna
df[1, "nome"]
## [1] "Ana"

# Aceder a múltiplas colunas por nome
df[c("nome", "idade")]
##      nome idade
## 1    Ana   23
## 2  Bruno   35
## 3 Carlos   31
## 4  Diana   28
```

5.4.3 Adicionar e Remover Colunas

Adicionar e remover colunas de um data frame é simples e direto:

```
# Adicionar uma nova coluna calculada
df$novos_salario <- df$salario * 1.1
df
##   id  nome idade salario novos_salario
## 1  1   Ana   23   5000         5500
## 2  2 Bruno   35   6000         6600
## 3  3 Carlos  31   7000         7700
## 4  4 Diana   28   8000         8800

# Remover uma coluna existente
df$id <- NULL
df
##      nome idade salario novos_salario
## 1   Ana   23   5000         5500
## 2 Bruno   35   6000         6600
## 3 Carlos  31   7000         7700
## 4 Diana   28   8000         8800
```

5.4.4 Fundir Data Frames

Data frames podem ser combinados ou fundidos usando a função `merge()`.

```
# Criar dois data frames
df1 <- data.frame(curso = c("PE", "LE", "CAL"), horas = c(60, 75, 90))
df2 <- data.frame(curso = c("CAL", "PE", "LE"), creditos = c(8, 6, 7))

# Fundir data frames pela variável 'curso'
df12 <- merge(df1, df2, by = "curso")
df12
##   curso horas creditos
## 1  CAL    90         8
## 2   LE    75         7
## 3   PE    60         6
```

5.4.5 Explorar o Data Frame

Funções úteis para explorar e entender a estrutura de um data frame:

```

df <- iris

# Nomes das colunas
names(df)
## [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"

# Classe dos dados de uma coluna
class(df$Sepal.Length)
## [1] "numeric"

class(df$Species)
## [1] "factor"

# Dimensão do data frame
dim(df)
## [1] 150 5

# Número de linhas
nrow(df)
## [1] 150

# Número de colunas
ncol(df)
## [1] 5

# Estrutura do data frame
str(df)
## '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 ...

# Visualização das primeiras linhas
head(df, 3)
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1 5.1 3.5 1.4 0.2 setosa
## 2 4.9 3.0 1.4 0.2 setosa
## 3 4.7 3.2 1.3 0.2 setosa

# Visualização das últimas linhas
tail(df, 5)
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 146 6.7 3.0 5.2 2.3 virginica

```

```
## 147      6.3      2.5      5.0      1.9 virginica
## 148      6.5      3.0      5.2      2.0 virginica
## 149      6.2      3.4      5.4      2.3 virginica
## 150      5.9      3.0      5.1      1.8 virginica
```

5.4.6 A função `subset()`

A função `subset()` em R é usada para criar subconjuntos de um data frame com base em condições lógicas. É particularmente útil quando se deseja filtrar linhas que satisfaçam certos critérios e selecionar colunas específicas ao mesmo tempo.

```
# Carregar o conjunto de dados 'iris'
df <- iris

# Criar um subconjunto com Sepal.Width maior que 3, selecionando apenas as colunas Petal.Width e Species
df1 <- df[df$Sepal.Width > 3, c("Petal.Width", "Species")]
head(df1)
##   Petal.Width Species
## 1         0.2  setosa
## 3         0.2  setosa
## 4         0.2  setosa
## 5         0.2  setosa
## 6         0.4  setosa
## 7         0.3  setosa

# Utilizando a função subset() para o mesmo subconjunto
(df2 <- subset(df, Sepal.Width > 3, select = c(Petal.Width, Species)))
##   Petal.Width Species
## 1         0.2  setosa
## 3         0.2  setosa
## 4         0.2  setosa
## 5         0.2  setosa
## 6         0.4  setosa
## 7         0.3  setosa
## 8         0.2  setosa
## 10        0.1  setosa
## 11        0.2  setosa
## 12        0.2  setosa
## 15        0.2  setosa
## 16        0.4  setosa
## 17        0.4  setosa
## 18        0.3  setosa
## 19        0.3  setosa
```



```
## 20      0.3    setosa
## 21      0.2    setosa
## 22      0.4    setosa
## 23      0.2    setosa
## 24      0.5    setosa
## 25      0.2    setosa
## 27      0.4    setosa
## 28      0.2    setosa
## 29      0.2    setosa
## 30      0.2    setosa
## 31      0.2    setosa
## 32      0.4    setosa
## 33      0.1    setosa
## 34      0.2    setosa
## 35      0.2    setosa
## 36      0.2    setosa
## 37      0.2    setosa
## 38      0.1    setosa
## 40      0.2    setosa
## 41      0.3    setosa
## 43      0.2    setosa
## 44      0.6    setosa
## 45      0.4    setosa
## 47      0.2    setosa
## 48      0.2    setosa
## 49      0.2    setosa
## 50      0.2    setosa
## 51      1.4 versicolor
## 52      1.5 versicolor
## 53      1.5 versicolor
## 57      1.6 versicolor
## 66      1.4 versicolor
## 71      1.8 versicolor
## 86      1.6 versicolor
## 87      1.5 versicolor
## 101     2.5 virginica
## 110     2.5 virginica
## 111     2.0 virginica
## 116     2.3 virginica
## 118     2.2 virginica
## 121     2.3 virginica
## 125     2.1 virginica
## 126     1.8 virginica
## 132     2.0 virginica
## 137     2.4 virginica
```

```
## 138      1.8 virginica
## 140      2.1 virginica
## 141      2.4 virginica
## 142      2.3 virginica
## 144      2.3 virginica
## 145      2.5 virginica
## 149      2.3 virginica
```

No exemplo acima, ambas as abordagens (indexação direta e `subset()`) produzem o mesmo resultado, mas `subset()` é geralmente mais legível e conveniente quando se trata de filtragem condicional.

```
# Criar um subconjunto com Petal.Width igual a 0.3, excluindo a coluna Sepal.Width
(df3 <- subset(df, Petal.Width == 0.3, select = -Sepal.Width))
##      Sepal.Length Petal.Length Petal.Width Species
## 7              4.6           1.4          0.3   setosa
## 18             5.1           1.4          0.3   setosa
## 19             5.7           1.7          0.3   setosa
## 20             5.1           1.5          0.3   setosa
## 41             5.0           1.3          0.3   setosa
## 42             4.5           1.3          0.3   setosa
## 46             4.8           1.4          0.3   setosa
```

Neste exemplo, a função `subset()` é utilizada para filtrar linhas em que `Petal.Width` é exatamente 0.3, ao mesmo tempo que exclui a coluna `Sepal.Width`.

```
# Criar um subconjunto selecionando as colunas de Sepal.Width a Petal.Width
(df4 <- subset(df, select = Sepal.Width:Petal.Width))
##      Sepal.Width Petal.Length Petal.Width
## 1              3.5           1.4          0.2
## 2              3.0           1.4          0.2
## 3              3.2           1.3          0.2
## 4              3.1           1.5          0.2
## 5              3.6           1.4          0.2
## 6              3.9           1.7          0.4
## 7              3.4           1.4          0.3
## 8              3.4           1.5          0.2
## 9              2.9           1.4          0.2
## 10             3.1           1.5          0.1
## 11             3.7           1.5          0.2
## 12             3.4           1.6          0.2
## 13             3.0           1.4          0.1
## 14             3.0           1.1          0.1
## 15             4.0           1.2          0.2
```

## 16	4.4	1.5	0.4
## 17	3.9	1.3	0.4
## 18	3.5	1.4	0.3
## 19	3.8	1.7	0.3
## 20	3.8	1.5	0.3
## 21	3.4	1.7	0.2
## 22	3.7	1.5	0.4
## 23	3.6	1.0	0.2
## 24	3.3	1.7	0.5
## 25	3.4	1.9	0.2
## 26	3.0	1.6	0.2
## 27	3.4	1.6	0.4
## 28	3.5	1.5	0.2
## 29	3.4	1.4	0.2
## 30	3.2	1.6	0.2
## 31	3.1	1.6	0.2
## 32	3.4	1.5	0.4
## 33	4.1	1.5	0.1
## 34	4.2	1.4	0.2
## 35	3.1	1.5	0.2
## 36	3.2	1.2	0.2
## 37	3.5	1.3	0.2
## 38	3.6	1.4	0.1
## 39	3.0	1.3	0.2
## 40	3.4	1.5	0.2
## 41	3.5	1.3	0.3
## 42	2.3	1.3	0.3
## 43	3.2	1.3	0.2
## 44	3.5	1.6	0.6
## 45	3.8	1.9	0.4
## 46	3.0	1.4	0.3
## 47	3.8	1.6	0.2
## 48	3.2	1.4	0.2
## 49	3.7	1.5	0.2
## 50	3.3	1.4	0.2
## 51	3.2	4.7	1.4
## 52	3.2	4.5	1.5
## 53	3.1	4.9	1.5
## 54	2.3	4.0	1.3
## 55	2.8	4.6	1.5
## 56	2.8	4.5	1.3
## 57	3.3	4.7	1.6
## 58	2.4	3.3	1.0
## 59	2.9	4.6	1.3
## 60	2.7	3.9	1.4

## 61	2.0	3.5	1.0
## 62	3.0	4.2	1.5
## 63	2.2	4.0	1.0
## 64	2.9	4.7	1.4
## 65	2.9	3.6	1.3
## 66	3.1	4.4	1.4
## 67	3.0	4.5	1.5
## 68	2.7	4.1	1.0
## 69	2.2	4.5	1.5
## 70	2.5	3.9	1.1
## 71	3.2	4.8	1.8
## 72	2.8	4.0	1.3
## 73	2.5	4.9	1.5
## 74	2.8	4.7	1.2
## 75	2.9	4.3	1.3
## 76	3.0	4.4	1.4
## 77	2.8	4.8	1.4
## 78	3.0	5.0	1.7
## 79	2.9	4.5	1.5
## 80	2.6	3.5	1.0
## 81	2.4	3.8	1.1
## 82	2.4	3.7	1.0
## 83	2.7	3.9	1.2
## 84	2.7	5.1	1.6
## 85	3.0	4.5	1.5
## 86	3.4	4.5	1.6
## 87	3.1	4.7	1.5
## 88	2.3	4.4	1.3
## 89	3.0	4.1	1.3
## 90	2.5	4.0	1.3
## 91	2.6	4.4	1.2
## 92	3.0	4.6	1.4
## 93	2.6	4.0	1.2
## 94	2.3	3.3	1.0
## 95	2.7	4.2	1.3
## 96	3.0	4.2	1.2
## 97	2.9	4.2	1.3
## 98	2.9	4.3	1.3
## 99	2.5	3.0	1.1
## 100	2.8	4.1	1.3
## 101	3.3	6.0	2.5
## 102	2.7	5.1	1.9
## 103	3.0	5.9	2.1
## 104	2.9	5.6	1.8
## 105	3.0	5.8	2.2

## 106	3.0	6.6	2.1
## 107	2.5	4.5	1.7
## 108	2.9	6.3	1.8
## 109	2.5	5.8	1.8
## 110	3.6	6.1	2.5
## 111	3.2	5.1	2.0
## 112	2.7	5.3	1.9
## 113	3.0	5.5	2.1
## 114	2.5	5.0	2.0
## 115	2.8	5.1	2.4
## 116	3.2	5.3	2.3
## 117	3.0	5.5	1.8
## 118	3.8	6.7	2.2
## 119	2.6	6.9	2.3
## 120	2.2	5.0	1.5
## 121	3.2	5.7	2.3
## 122	2.8	4.9	2.0
## 123	2.8	6.7	2.0
## 124	2.7	4.9	1.8
## 125	3.3	5.7	2.1
## 126	3.2	6.0	1.8
## 127	2.8	4.8	1.8
## 128	3.0	4.9	1.8
## 129	2.8	5.6	2.1
## 130	3.0	5.8	1.6
## 131	2.8	6.1	1.9
## 132	3.8	6.4	2.0
## 133	2.8	5.6	2.2
## 134	2.8	5.1	1.5
## 135	2.6	5.6	1.4
## 136	3.0	6.1	2.3
## 137	3.4	5.6	2.4
## 138	3.1	5.5	1.8
## 139	3.0	4.8	1.8
## 140	3.1	5.4	2.1
## 141	3.1	5.6	2.4
## 142	3.1	5.1	2.3
## 143	2.7	5.1	1.9
## 144	3.2	5.9	2.3
## 145	3.3	5.7	2.5
## 146	3.0	5.2	2.3
## 147	2.5	5.0	1.9
## 148	3.0	5.2	2.0
## 149	3.4	5.4	2.3
## 150	3.0	5.1	1.8

Aqui, `subset()` é usado para selecionar todas as colunas desde `Sepal.Width` até `Petal.Width`.

5.4.7 A Função `summary()`

A função `summary()` é amplamente usada em R para fornecer resumos estatísticos dos dados. O comportamento de `summary()` varia conforme o tipo de objeto ao qual é aplicado.

```
# Aplicando summary() a um vetor numérico
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
summary(x)
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      1.00   3.25   5.50   5.50   7.75   10.00

# Aplicando summary() a uma coluna numérica de um data frame
summary(iris$Sepal.Length)
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      4.30   5.10   5.80   5.84   6.40   7.90

# Aplicando summary() a um data frame completo
summary(iris)
##   Sepal.Length  Sepal.Width  Petal.Length  Petal.Width  Species
##   Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1   setosa   :50
##   1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3   versicolor:50
##   Median :5.80   Median :3.00   Median :4.35   Median :1.3   virginica :50
##   Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
##   3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
##   Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
```

Aplicar `summary()` ao data frame `iris` inteiro fornece resumos estatísticos para todas as colunas, incluindo estatísticas descritivas para variáveis numéricas e uma contagem de frequências para variáveis categóricas (fatores).

5.4.8 Valores faltantes

5.4.9 Exercícios

1. Crie um data frame chamado `estudantes` com as colunas: Nome (caracteres), Idade (números inteiros), Curso (caracteres) e Nota (números decimais). Adicione os dados de cinco estudantes fictícios e exiba o data frame.
2. Utilize o data frame `estudantes` criado no exercício anterior. Aceda à coluna `Nota` e aumente todas as notas em 0.5. Substitua o valor da coluna `Curso` para “Estatística” para todos os estudantes com nota superior a 8.0.

3. Dado o data frame `mtcars` embutido no R, filtre as linhas onde o número de cilindros (`cyl`) é igual a 6 e a potência (`hp`) é maior que 100. Crie um novo data frame com essas informações.
4. Adicione uma nova coluna ao data frame `mtcars` chamada `Peso_KG` que converta o peso dos carros (`wt`) de mil libras para quilogramas (multiplicando por 453.592).
5. Utilizando o data frame `mtcars`, calcule a média, o mínimo e o máximo da potência (`hp`) para cada número diferente de cilindros (`cyl`). Utilize as funções `aggregate()` ou `dplyr` para realizar esta operação.
6. Transforme a coluna `am` do data frame `mtcars` num fator com rótulos significativos: “Automático” para 0 e “Manual” para 1. Depois, conte quantos carros existem para cada tipo de transmissão.
7. Utilizando o data frame `mtcars`, aplique a função `sapply()` para calcular o desvio padrão de todas as colunas numéricas.
8. Ordene o data frame `mtcars` pela coluna `mpg` em ordem decrescente. Mostre os 5 carros mais económicos e os 5 menos económicos.
9. Crie dois data frames, `carros1` e `carros2`, que contenham diferentes subconjuntos de informações do data frame `mtcars`. Depois, use a função `merge()` para combinar esses data frames com base na coluna `car`.

5.5 Listas

Em R, uma **lista** é uma estrutura de dados versátil que pode armazenar elementos de diferentes tipos, como vetores, matrizes, data frames, funções e até outras listas. Esta flexibilidade torna as listas uma poderosa ferramenta para manipulação de dados complexos, permitindo armazenar uma coleção heterogênea de elementos em um único objeto.

- **Flexibilidade:** Ao contrário dos vetores, que são homogêneos e podem conter apenas elementos de um único tipo, as listas podem conter elementos de diferentes tipos e tamanhos.
- **Indexação:** Os elementos de uma lista podem ser acessados de várias maneiras:
- **Colchetes duplos** `[[]]`: Usados para acessar elementos específicos de uma lista.
- **Operador \$:** Usado para acessar elementos nomeados.
- **Colchetes simples** `[]`: Retornam uma sublista contendo os elementos especificados.

5.5.1 Criar e Manipular Listas

Vamos criar uma lista que contém diferentes tipos de elementos, incluindo um vetor, um vetor de caracteres, uma matriz e uma função:

```
# Criar uma lista com diferentes tipos de elementos
minha_lista <- list(
  nome = "Estudante",
  idade = 21,
  notas = c(85, 90, 92),
  disciplinas = c("Matemática", "Estatística", "Computação"),
  matriz_exemplo = matrix(1:9, nrow = 3, byrow = TRUE),
  media= function(x) mean(x)
)

# Visualizar a lista
print(minha_lista)
## $nome
## [1] "Estudante"
##
## $idade
## [1] 21
##
## $notas
## [1] 85 90 92
##
## $disciplinas
## [1] "Matemática" "Estatística" "Computação"
##
## $matriz_exemplo
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
##
## $media
## function(x) mean(x)

# Aceder a um elemento pelo nome usando $
print(minha_lista$nome)
## [1] "Estudante"

# Aceder a um elemento pelo índice
print(minha_lista[[1]])
## [1] "Estudante"
```



```
# Aceder a uma sublista contendo os dois primeiros elementos
print(minha_lista[1:2])
## $nome
## [1] "Estudante"
##
## $idade
## [1] 21

# Aceder à segunda nota do vetor "notas" dentro da lista
print(minha_lista$notas[2])
## [1] 90
```

5.5.2 Por que usar listas?

Listas são extremamente úteis em R para:

- **Armazenar resultados de diferentes tipos:** Em análises estatísticas complexas, os resultados muitas vezes consistem em diferentes tipos de dados (como números, tabelas, gráficos e modelos estatísticos).
- **Agrupar dados heterogêneos:** Quando se precisa manipular um conjunto de dados que inclui diversos tipos de informação (por exemplo, dados brutos, estatísticas resumidas, e resultados de modelos).
- **Passar múltiplos argumentos para funções:** As listas são usadas para agrupar múltiplos parâmetros para funções, tornando-as uma ferramenta poderosa para programação funcional.

5.5.3 Exercícios

1. Crie uma lista em R chamada `dados_estudante` que contenha as seguintes informações sobre um estudante:

- Nome: “João”
- Idade: 21
- Notas: Um vetor com as notas em Estatística (85), Matemática (90), e Computação (95).

Depois de criar a lista, acesse e imprima:

1. O nome do estudante.
2. A idade do estudante.
3. A nota em Computação.

2. Considere a lista `dados_estudante` criada no exercício anterior. Adicione um novo elemento à lista que contenha o status de aprovação do estudante, com valor “Aprovado”. Em seguida, substitua a nota de Estatística para 88. Por fim, imprima a lista completa.

3. Crie duas listas chamadas `estudante1` e `estudante2` com as mesmas estruturas da lista `dados_estudante`. Em `estudante1`, use os valores:

- Nome: “Maria”
- Idade: 22
- Notas: 78, 85, 90
- Status: “Aprovado”

Em `estudante2`, use os valores:

- Nome: “Carlos”
- Idade: 23
- Notas: 70, 75, 80
- Status: “Aprovado”

Agora, combine `estudante1` e `estudante2` numa nova lista chamada `turma`, e imprima a lista `turma`.

4. Utilizando a lista `turma` criada no exercício anterior, faça as seguintes operações:

- (a) Extraia e imprima o nome do segundo estudante.
- (b) Calcule a média das notas do primeiro estudante.
- (c) Altere o status do segundo estudante para “Reprovado” e imprima a lista atualizada.

5. Crie uma lista chamada `estatistica_aplicada` que contenha duas listas internas: `turma1` e `turma2`. Cada uma dessas listas internas deve conter as informações de dois estudantes (com as mesmas estruturas utilizadas anteriormente). Por exemplo:

- `turma1`: Contendo `estudante1` e `estudante2`.
- `turma2`: Contendo dois novos estudantes de sua escolha.

Aceda e imprima:

- (a) O nome do primeiro estudante da `turma2`.
- (b) A média das notas do segundo estudante da `turma1`.
- (c) A lista completa `estatistica_aplicada`.

Chapter 6

Estruturas de Seleção

Em R, as estruturas de seleção ou decisão são usadas para controlar o fluxo de execução do código com base em condições específicas. Estas estruturas permitem executar diferentes blocos de código dependendo de valores ou condições lógicas. As estruturas de seleção mais comuns em R são `if`, `else`, e `else if`.

6.1 Condicional `if`

A instrução `if` executa um bloco de código apenas se uma condição for verdadeira. Caso contrário, o bloco de código dentro do `if` é ignorado.

```
# Sintaxe
if (condição) {
    # Código a ser executado se a condição for TRUE
}
```

Exemplo 1:

```
idade <- 18

if (idade >= 18) {
    print("Maior de idade")
}
## [1] "Maior de idade"
```

Neste exemplo, como a variável `idade` é 18, a condição `idade >= 18` é verdadeira, então o R executa o código dentro do `if` e imprime "Maior de idade".

6.2 Estrutura if...else

A estrutura `if...else` permite executar um bloco de código se a condição for verdadeira, e um outro bloco de código se a condição for falsa.

```
# Sintaxe
if (condição) {
    # Código a ser executado se a condição for TRUE
} else {
    # Código a ser executado se a condição for FALSE
}
```

Exemplo 2:

```
idade <- 16

if (idade >= 18) {
    print("Maior de idade")
} else {
    print("Menor de idade")
}
## [1] "Menor de idade"
```

Como a variável `idade` é 16, a condição `idade >= 18` é falsa, então o R executa o bloco de código dentro do `else` e imprime "Menor de idade".

6.3 Condicional else if

A estrutura `else if` permite testar múltiplas condições. Quando uma das condições é verdadeira, o R executa o código associado e ignora os restantes `else if` ou `else`.

```
# Sintaxe
if (condição1) {
    # Código se condição1 for TRUE
} else if (condição2) {
    # Código se condição2 for TRUE
} else {
    # Código se nenhuma condição anterior for TRUE
}
```

Exemplo 3:

```
nota <- 85

if (nota >= 90) {
  print("Excelente")
} else if (nota >= 70) {
  print("Bom")
} else {
  print("Insuficiente")
}
## [1] "Bom"
```

Neste caso, a variável `nota` é 85, então o código imprime "Bom" porque a condição `nota >= 70` é verdadeira.

6.4 A função ifelse()

A função `ifelse` é uma versão vetorizada de `if...else` que retorna valores dependendo de uma condição. É especialmente útil para aplicar condições a vetores ou data frames.

```
# Sintaxe
resultado <- ifelse(condição, valor_se_true, valor_se_false)
```

Exemplo 4:

```
valores <- c(4, 6, 9, 3)
resultado <- ifelse(valores > 5, "maior que 5", "não é maior que 5")
print(resultado)
## [1] "não é maior que 5" "maior que 5"      "maior que 5"
## [4] "não é maior que 5"
```

Aqui, `ifelse()` aplica a condição `valores > 5` a cada elemento do vetor `valores` e retorna “maior que 5” se a condição for verdadeira ou “não é maior que 5” caso contrário.

6.5 Exemplos

Exemplo 5: Indique o(os) erro(s) no código abaixo

```
if (x%%2 = 0){  
    print("Par")  
} else {  
    print("Ímpar")  
}
```

Código correto

```
if (x%%2 == 0){  
    print("Par")  
} else {  
    print("Ímpar")  
}
```

Exemplo 6: Indique o(os) erro(os) no código abaixo

```
if (a>0) {  
    print("Positivo")  
    if (a%%5 = 0)  
        print("Divisível por 5")  
} else if (a==0)  
    print("Zero")  
else if {  
    print("Negativo")  
}
```

Código correto

```
if (a>0) {  
    print("Positivo")  
    if (a%%5 == 0) {  
        print("Divisível por 5")  
    }  
} else if (a==0) {  
    print("Zero")  
} else {  
    print("Negativo")  
}
```

Exemplo 7: Quais são os valores de x e y no final da execução?

```
x = 1  
y = 0  
if (x == 1){
```

```
y = y - 1
}
if (y == 1){
  x = x + 1
}
```

Exemplo 8:

- Se $x=1$ qual será o valor de x no final da execução?
- Qual teria de ser o valor de x para que no final da execução fosse -1?
- Há uma parte do programa que nunca é executada: qual é e porquê?

```
if (x == 1){
  x = x + 1
  if (x == 1){
    x = x + 1
  } else {
    x = x - 1
  }
} else {
  x = x - 1
}
```

6.6 Exercícios

1. Escreva um programa em R que leia um número do utilizador e exiba “O número é positivo” se for maior que zero.
2. Crie um programa em R que leia um número inteiro do utilizador e imprima “Par” se o número for par, e “Ímpar” caso contrário.
3. Escreva um programa em R que leia um número e exiba se ele está no intervalo $[0, 10]$, $[11, 20]$, ou maior que 20. Considere que os intervalos $[0, 10]$ e $[11, 20]$ são inclusivos. Exemplo: para o número 15, o programa deve exibir “O número está no intervalo $[11, 20]$ ”.
4. **Exercício 4:** Escreva um programa em R que leia a idade de uma pessoa e a classifique como “Criança” (0-12 anos), “Adolescente” (13-17 anos), “Adulto” (18-64 anos), ou “Idoso” (65 anos ou mais). Se a idade inserida for negativa, o programa deve exibir “Idade inválida”.
5. Escreva um programa em R que leia o preço original de um produto e aplique um desconto com base no seguinte critério: * Desconto de 5% se o

preço for inferior a \$100 * Desconto de 10% se o preço estiver entre \$100 e \$500 (inclusive) * Desconto de 15% se o preço for superior a \$500

Se o preço inserido for negativo, o programa deve exibir “Preço inválido”. Exemplo de entrada: 350; Saída esperada: “O preço com desconto é \$315”.

6. Exercício 6: Escreva um programa em R que leia as coordenadas (x, y) de um ponto e determine em qual quadrante o ponto está localizado. Se o ponto estiver em um dos eixos, o programa deve especificar “Eixo X” ou “Eixo Y”. Exemplo de entrada: (x = -3, y = 2); Saída esperada: “Segundo quadrante”.

7. Escreva um programa em R que leia um ano e determine se é um ano bissexto. Um ano é bissexto se: * É divisível por 4, mas não divisível por 100, exceto quando divisível por 400.

Se o ano inserido for negativo, o programa deve exibir “Ano inválido”. Exemplo de entrada: 2000; Saída esperada: “Ano bissexto”.

8. Escreva um programa que leia um número inteiro entre 0 e 999, e o descreva em termos de centenas, dezenas e unidades. Por exemplo, para 304, o programa deve exibir “3 centenas 0 dezenas e 4 unidades”. Caso o número inteiro não pertença ao intervalo [0;999], deverá imprimir um aviso “Número fora do intervalo”.

9. Escreva um programa em R que leia um número inteiro positivo (menor ou igual a 5) e escreva no ecrã a sua representação em numeração romana. Se o número inserido for maior que 5 ou menor que 1, o programa deve exibir “Número fora do intervalo”. Exemplo de entrada: 3; Saída esperada: “III”.

10. Escreva um programa em R que leia quatro números inteiros (um de cada vez) e exiba após cada iteração qual o menor número lido até ao momento. Exemplo:

```
Introduza um numero inteiro: 4

`O menor numero introduzido até agora é 4.`

Introduza um numero inteiro: 6`

`O menor numero introduzido até agora é 4.`

Introduza um numero inteiro: 2`

`O menor numero introduzido até agora é 2.`
```


Chapter 7

Funções

Em R, uma **função** é um bloco de código que realiza uma tarefa específica e é executado somente quando chamada. Funções são fundamentais na programação, pois permitem a modularização do código, facilitando a manutenção, reutilização e compreensão.

Características de uma Função

- **Reutilizável:** Uma vez definida, uma função pode ser chamada repetidamente em diferentes partes de um script.
- **Modularização:** Funções ajudam a dividir o código em partes menores e mais gerenciáveis, o que torna o código mais organizado e legível.
- **Parâmetros (Argumentos):** Dados podem ser passados para uma função na forma de parâmetros ou argumentos, permitindo que a função opere sobre entradas específicas.

A estrutura básica de uma função em R é a seguinte:

```
# Sintaxe
nome_da_funcao <- function(argumentos) {
  # Corpo da função: código que realiza operações
  resultado <- ... # Cálculos ou operações
  return(resultado) # Retorna o valor final
}
```

- **Nome da Função:** Um identificador único que você escolhe para a função.
- **Argumentos:** Valores de entrada que a função recebe quando chamada. Podem ser opcionais ou obrigatórios.
- **Corpo da Função:** O bloco de código que executa as operações definidas.

- **Return (Retorno):** O valor que a função devolve após a execução.

O principal objetivo das funções é promover a reutilização de código e reduzir a redundância, facilitando a manutenção e o desenvolvimento eficiente de scripts em R.

Exemplo: Cálculo de Área Retangular

Vamos criar uma função para calcular a área de um retângulo, dada a sua largura e altura:

```
calcula_area <- function(largura, altura) {  
  area <- largura * altura  
  return(area)  
}  
  
largura_obj <- as.numeric(readline("Insira a largura (em cm): "))  
altura_obj <- as.numeric(readline("Insira a altura (em cm): "))  
  
# Chamada da função para calcular a área  
area_obj <- calcula_area(largura_obj, altura_obj)  
  
print(paste("A área do retângulo é:", area_obj, "cm²"))
```

- **Variáveis Locais:** Dentro da função `calcula_area`, as variáveis `largura` e `altura` são **locais**. Elas existem apenas durante a execução da função e são destruídas após a conclusão.
- **Variáveis Globais:** As variáveis `largura_obj` e `altura_obj` são **globais** dentro do script, ou seja, são acessíveis em todo o script, fora da função.
- **Importância da Nomenclatura:** É uma boa prática usar nomes diferentes para variáveis locais e globais para evitar confusão e melhorar a legibilidade do código.

Passagem de Argumentos com Valores Padrão

Funções em R podem ser definidas com valores padrão para seus argumentos, o que permite chamadas de função sem a necessidade de especificar todos os argumentos.

```
calcula_area <- function(largura, altura=2) {  
  area <- largura * altura  
  return(area)  
}  
  
# Chamada da função com um argumento omitido  
area_obj <- calcula_area(4)
```

```
print(area_obj)
## [1] 8
```

- **Argumentos com Valores Padrão:** Se a `altura` não for especificada na chamada da função, seu valor padrão será 2. Assim, a função calcula a área usando largura 4 e altura 2.

Passagem de Argumentos por Palavra-chave

Em R, ao chamar uma função, os argumentos podem ser passados por nome (palavra-chave), permitindo maior flexibilidade na ordem dos argumentos.

```
calcula_area <- function(largura, altura=2) {
  area <- largura * altura
  return(area)
}

# Chamada da função com argumentos nomeados
area_obj <- calcula_area(altura=4, largura=3)

print(area_obj)
## [1] 12
```

- **Flexibilidade na Ordem dos Argumentos:** Ao usar argumentos nomeados, a ordem em que os argumentos são passados não importa. O R associa os valores aos nomes corretos.

Uso da Função `stop()` para Controle de Erro

A função `stop()` em R é usada para interromper a execução de uma função ou script, exibindo uma mensagem de erro personalizada. É útil para tratar casos em que certas condições pré-definidas não são atendidas.

```
f <- function(x) {
  if (x < 0) {
    stop("Erro: x não pode ser negativo") # Interrompe a função com uma mensagem de erro
  }
  return(sqrt(x))
}

f(-2)
## Error in f(-2): Erro: x não pode ser negativo
```

- **Uso de `stop()`:** Quando a condição `x < 0` é verdadeira, a função `stop()` é chamada, o que interrompe a execução e gera uma mensagem de erro. Isso é útil para prevenir operações inválidas ou resultados inesperados.

Exercício 1: Qual será o output do script abaixo?

```
x <- 10

minha_funcao <- function() {
  x <- 5
  return(x)
}

print(minha_funcao())
print(x)
```

Exercício 2: Qual é o resultado da chamada da função `dados_estudante`?

```
dados_estudante <- function(nome, altura=167){
  print(paste("O(A) estudante",nome,"tem",altura,"centímetros de altura."))
}

dados_estudante("Joana",160)
```

Exercício 3: Qual é a sintaxe correta para definir uma função em R que soma dois números?

- (a) `sum <- function(x, y) {return(x + y)}`
- (b) `function sum(x, y) {return(x + y)}`
- (c) `def sum(x, y) {return(x + y)}`
- (d) `sum(x, y) = function {return(x + y)}`

Exercício 4: Qual das seguintes chamadas à função estão corretas?

```
dados_estudante <- function(nome, altura=167) {
  print(paste("O(A) estudante",nome,"tem",altura,"centímetros de altura."))
}

dados_estudante("Joana",160)
dados_estudante(altura=160, nome="Joana")
dados_estudante(nome = "Joana", 160)
dados_estudante(altura=160, "Joana")
dados_estudante(160)
```

`dados_estudante(160)` - Esta chamada está errada porque 160 será interpretado como nome, e altura usará seu valor padrão, 167. Isso resultará na

impressão: "0(A) estudante 160 tem 167 centímetros de altura." A chamada está tecnicamente correta no sentido de sintaxe, mas o resultado não faz sentido lógico, já que 160 não é um nome válido para um estudante.

Exercício 5: Qual o resultado do seguinte programa?

```
adi <- function(a,b) {  
  return(c(a+5, b+5))  
}  
  
resultado <- adi(3,2)
```

7.1 Exercícios

1. Crie uma função chamada `quadrado()` que receba um único argumento numérico `x` e devolva o quadrado de `x`. Teste a função com os valores 3, 5 e 7.
2. Implemente uma função chamada `hipotenusa(a,b)` que calcule a hipotenusa de um triângulo retângulo dados os comprimentos dos dois catetos, `a` e `b`. A função deve devolver o comprimento da hipotenusa utilizando o Teorema de Pitágoras.
3. Escreva uma função chamada `divisao_segura(numerador, denominador)` que devolva o resultado da divisão ou uma mensagem de erro apropriada se o denominador for zero.
4. Escreva uma função que receba dois inteiros e devolva o maior deles. Teste a função escrevendo um programa que receba dois números inteiros do utilizador e imprime o resultado da chamada à função. Depois, escreva uma segunda função que devolva o menor deles, aproveitando a primeira função.
5. Escreva uma função que devolva o maior de três números inteiros, utilizando a função `maior()` do exercício anterior.
6. Crie uma função chamada `analise_vetor()` que receba um vetor numérico e devolva outro vetor contendo a soma, a média, o desvio padrão e o valor máximo do vetor. Use a função para analisar o vetor `c(4, 8, 15, 16, 23, 42)`. Funções auxiliares `sum()`, `mean()`, `sd()`, `max()`.
7. Implemente uma função chamada `classificar_numero()` que classifique um número inteiro como “positivo”, “negativo”, ou “zero”.
8. Escreva uma função que calcule o IMC (Índice de Massa Corporal) com base no peso (kg) e altura (m) fornecidos como argumentos e classifique o resultado com base na tabela:

IMC (kg/m ²)	Classificação
Menor que 18.5	Baixo peso
De 18.5 a 24.9	Peso normal
De 25 a 29.9	Sobrepeso
De 30 a 34.9	Obesidade grau I
De 35 a 39.9	Obesidade grau II
Igual ou maior que 40	Obesidade grau III

Lembre-se de que $IMC = \frac{Peso}{Altura^2}$.

9. Crie funções em R para converter valores entre diferentes moedas com base nas taxas de câmbio. Implemente e teste as funções, e escreva um programa que solicite ao utilizador um valor em euros e imprima a conversão desse valor para algumas das principais moedas, como dólar, libra, iene e real.

Chapter 8

Scripts em R

Um **script** em R é um ficheiro que contém um conjunto de definições, como variáveis, funções e blocos de código, que podem ser reutilizados em outros programas R. Scripts são ferramentas essenciais para automatizar tarefas e para garantir a reprodutibilidade da análise de dados.

- **Criar um Script:** Para criar um script em R, basta gravar o código num ficheiro com a extensão “.R”. Este ficheiro pode então ser executado ou reutilizado noutras sessões de R.

Exemplo: Guarde o seguinte código num ficheiro chamado `meu_script.R`:

```
produto function(x,y){  
  return(x*y)  
}
```

Executar o script

Para executar o script que criou, utilize a função `source()` no console do R ou dentro de outro script. Isto permitirá que o R leia e execute todas as linhas de código do script.

1. **Executar no Diretório de Trabalho Atual:** Se o ficheiro `meu_script.R` estiver no diretório de trabalho atual, pode executá-lo simplesmente escrevendo:

```
source("meu_script.R")
```

2. **Executar a Partir de Outro Diretório:** Se o ficheiro estiver localizado num diretório diferente do diretório de trabalho atual, terá de fornecer o caminho completo para o ficheiro:

```
source("/caminho/para/o/seu/script/meu_script.R")
```

O que Acontece Quando Executamos um Script?

Quando utiliza o comando `source()`, o R lê e executa todas as linhas do script. Qualquer função, variável ou resultado de operações realizadas dentro do script será carregado e ficará disponível no ambiente de trabalho após a execução. Por exemplo, após executar o script acima, a função `produto()` estará disponível para ser utilizada:

```
# Agora pode utilizar a função 'produto'  
produto(2,3)  
## [1] 6
```

Notas Importantes

- **Diretório de trabalho:** O diretório de trabalho é o local onde o R procura por ficheiros para leitura ou onde guarda ficheiros. Pode verificar o diretório de trabalho atual utilizando `getwd()` e mudar para um novo diretório com `setwd("caminho/do/diretorio")`.

8.1 Exercícios

1. Crie os seguintes scripts:

- (a) Crie um script `quadrado.R` que contenha funções para calcular o perímetro e a área de um quadrado, dado o comprimento do lado. Utilize o script `quadrado.R` noutro script para calcular o perímetro e a área de um quadrado com lado de comprimento 5.
- (b) Crie um script `estatistica.R` que contenha funções para ordenar uma amostra, calcular a média, a variância e o desvio padrão de um vetor numérico.
- (c) Crie um programa que utilize o script `estatistica.R` para calcular a mediana, a variância e o desvio padrão da amostra `amostra <- c(1, 2, 3, 4, 5, 6, 7)`.

Chapter 9

Leitura de dados

O R é uma linguagem poderosa para análise de dados e oferece diversas funções para importar dados de diferentes formatos. Independentemente do formato, o processo básico de leitura de dados em R segue alguns passos comuns:

1. **Especificar o caminho do ficheiro:** Onde o ficheiro está localizado.
2. **Indicar as características do ficheiro:** Como delimitador, presença de cabeçalhos, tipos de dados, etc.
3. **Ler os dados e armazená-los num objeto:** Geralmente um data frame para fácil manipulação e análise.

9.1 Leitura de Dados da Entrada do Utilizador

A função `scan()` é usada para ler dados da entrada padrão (teclado) ou de um ficheiro, e devolve um vetor. Um uso comum de `scan()` é para entrada de dados manual:

```
# Solicita ao utilizador para introduzir números  
numeros <- scan()
```

Após escrever `scan()`, o utilizador pode inserir uma série de números separados por espaços e pressionar Enter para concluir a entrada.

9.2 Diretório de trabalho

O **diretório de trabalho** (working directory) em R é o local padrão onde o R lê e grava ficheiros. Quando importa dados ou guarda resultados, o R utiliza o

diretório de trabalho como ponto de referência. Ter um diretório de trabalho configurado corretamente é essencial para facilitar o acesso a ficheiros de dados e gravação de saídas.

Funções Principais para Manipulação de Diretório

- `getwd()`: Retorna o caminho do diretório atual de trabalho.
- `setwd("caminho/do/diretorio")`: Define o diretório de trabalho para o caminho especificado.

Para verificar o diretório de trabalho atual, utilize `getwd()`:

```
getwd()
```

A saída pode ser semelhante a esta, dependendo do seu sistema operativo:

```
# Exemplo de saída no Windows
## [1] "C:/Users/Utilizador/Documents"

# Exemplo de saída no Mac
## [1] "/Users/Utilizador/Documents"
```

Para alterar o diretório de trabalho para uma pasta específica, use `setwd()`:

```
# Define o diretório de trabalho para "C:/Users/Utilizador/Documents/ProjetosR"
setwd("C:/Users/Utilizador/Documents/ProjetosR")

# Verifica se o diretório de trabalho foi alterado
print(getwd())
## [1] "C:/Users/Utilizador/Documents/ProjetosR"
```

Ao definir corretamente o diretório de trabalho, pode importar ficheiros diretamente sem precisar especificar o caminho completo. Por exemplo, se o ficheiro `dados.csv` estiver no diretório de trabalho, pode lê-lo facilmente:

```
# Lê o ficheiro "dados.csv" no diretório de trabalho atual
dados <- read.csv("dados.csv")

# Exibe as primeiras linhas do conjunto de dados
head(dados)
```

9.3 A Função `read.table()`

A função `read.table()` é uma das mais versáteis em R para a leitura de ficheiros de texto em formato tabular. Ela permite importar dados de ficheiros de texto e configurá-los de acordo com suas características. Faça o download do ficheiro `dados.txt` em link para praticar.

```
# Leitura de ficheiro txt com read.table()
dados <- read.table(file = "dados.txt", header = TRUE, sep = ",")
print(dados)
```

Argumentos Comuns de `read.table()`

- **file**: Especifica o nome do ficheiro ou o caminho do ficheiro para leitura.
- **header**: Indica se a primeira linha do ficheiro contém os nomes das colunas. Se `header = TRUE`, a primeira linha é considerada como cabeçalho.
- **sep**: Define o caractere separador entre os campos. O padrão é uma vírgula (,), mas pode ser especificado para outros caracteres, como espaço (" "), ponto e vírgula (;), etc.
- **dec**: Define o caractere utilizado para separar a parte inteira da parte decimal dos números. Por exemplo, pode ser ponto (.) ou vírgula (,).
- **nrows**: Especifica o número máximo de linhas a serem lidas do ficheiro, útil para ler apenas uma parte do ficheiro.
- **na.strings**: Define quais valores no ficheiro devem ser interpretados como NA (valores ausentes), como "NA", "N/A", etc.
- **skip**: Indica o número de linhas a serem ignoradas no início do ficheiro antes de começar a leitura.
- **comment.char**: Define o caractere usado para denotar comentários no ficheiro. Linhas que começam com este caractere serão ignoradas.

Para ver todas as opções disponíveis na função `read.table()`, pode usar:

```
args(read.table)
```

```
## function (file, header = FALSE, sep = ",", quote = "\"", dec = ".",
##     numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
##     col.names, as.is = !stringsAsFactors, tryLogical = TRUE,
##     na.strings = "NA", colClasses = NA, nrows = -1, skip = 0,
##     check.names = TRUE, fill = !blank.lines.skip, strip.white = FALSE,
##     blank.lines.skip = TRUE, comment.char = "#", allowEscapes = FALSE,
##     flush = FALSE, stringsAsFactors = FALSE, fileEncoding = "",
##     encoding = "unknown", text, skipNul = FALSE)
## NULL
```

Se o ficheiro for um CSV (valores separados por vírgula), pode utilizar `read.table()` com o argumento `sep` apropriado:

```
dados_csv <- read.table("dados.csv", header = TRUE, sep = ",")
```

9.4 A função `read.csv()`

A função `read.csv()` está otimizada para a leitura de ficheiros CSV (Comma-Separated Values), que são ficheiros de texto onde cada linha representa um registo e os valores são separados por vírgulas. A principal diferença entre `read.csv()` e `read.table()` é que o separador padrão em `read.csv()` é uma vírgula.

```
# Leitura de ficheiro CSV com read.csv  
dados_csv <- read.csv("dados.csv", header = TRUE)
```

Aqui, o argumento `header = TRUE` indica que a primeira linha do ficheiro contém os nomes das colunas. Outros argumentos são semelhantes aos de `read.table()`.

9.5 A função `read.csv2()`

A função `read.csv2()` é semelhante à `read.csv()`, mas é projetada para lidar com ficheiros CSV onde o separador padrão é um ponto e vírgula (;) e o separador decimal é uma vírgula (,). Este formato é comum em alguns países europeus.

```
# Leitura de CSV com separador ponto e vírgula  
dados_csv2 <- read.csv2("dados.csv2", header = TRUE)
```

9.6 A Função `read_excel()` do Pacote `readxl`

Para ler ficheiros Excel (.xls e .xlsx), usamos a função `read_excel()` do pacote `readxl`. Este pacote precisa ser instalado e carregado antes da sua utilização.

```
library(readxl)  
  
# Sintaxe básica  
read_excel(path, sheet = NULL, range = NULL, col_names = TRUE,  
            col_types = NULL, na = "", trim_ws = TRUE, skip = 0, n_max = Inf,  
            guess_max = min(1000, n_max), progress = readxl_progress())
```

Principais Parâmetros de `read_excel()`

- **path**: O caminho do ficheiro Excel a ser lido. Este é um argumento obrigatório e pode ser um caminho local ou uma URL.
- **sheet**: O nome ou o índice da aba (folha) do Excel a ser lida. Se não for especificado, a função irá ler a primeira folha.
- **range**: Um intervalo específico a ser lido, como "A1:D10". Se `NULL` (padrão), a função lê toda a folha.
- **col_names**: Um argumento lógico (`TRUE` ou `FALSE`) que indica se a primeira linha da folha deve ser usada como nomes das colunas no data frame. O padrão é `TRUE`.
- **col_types**: Um vetor de tipos de colunas que podem ser "blank", "numeric", "date", "text", ou "guess". O padrão é `NULL`, o que permite que a função adivinhe os tipos de coluna automaticamente.
- **na**: Um vetor de strings que devem ser interpretadas como valores ausentes (NA). O padrão é string vazia ("").
- **trim_ws**: Um argumento lógico (`TRUE` ou `FALSE`) que indica se os espaços em branco devem ser removidos do início e do fim dos valores de texto. O padrão é `TRUE`.
- **skip**: Número de linhas a serem ignoradas antes de começar a leitura. O padrão é 0.
- **n_max**: Número máximo de linhas a serem lidas. O padrão é `Inf`, o que significa que todas as linhas são lidas.

Exemplo de Utilização: Vamos considerar um exemplo em que temos um ficheiro Excel chamado "instagram.xlsx" com três folhas: "Sheet1", "Sheet2" e "Sheet3". Este ficheiro pode ser descarregado aqui. Vamos ler a primeira folha (Sheet1):

```
dados <- read_excel(path = "instagram.xlsx",
                    sheet = "Sheet1",
                    col_names = TRUE,
                    trim_ws = TRUE)
```

9.7 Leitura de Dados Online

O R também permite ler dados diretamente de URLs. Pode usar funções como `read.table()`, `read.csv()`, ou `read_excel()` (dependendo do formato do ficheiro) para importar dados diretamente de um link online.

```
# Leitura de dados online com read.table
url <- "https://example.com/data.csv"
dados_online <- read.table(url, header = TRUE, sep = ",")
```

Neste exemplo, `url` contém o endereço do ficheiro CSV na web, e `read.table()` é usado para importar o conteúdo como um data frame.

Chapter 10

Pipe

10.1 O operador pipe

O operador pipe (`%>%`) em R é uma ferramenta poderosa que facilita a escrita de código mais limpo e legível. Ele permite que o resultado de uma função seja passado diretamente como entrada para a próxima função, sem a necessidade de criar variáveis intermediárias. Este operador ajuda a tornar o fluxo de dados mais intuitivo e o código mais fácil de ler.

O operador pipe foi popularizado pelo pacote **magrittr**, desenvolvido por Stefan Milton Bache e lançado em 2014. O nome “magrittr” é uma referência ao artista surrealista René Magritte, famoso pela pintura “Ceci n’est pas une pipe” (Isto não é um cachimbo), que inspirou a ideia de que o operador pipe permite que dados fluam através de uma sequência de operações de maneira intuitiva e sem a necessidade de variáveis temporárias.

Para começar a utilizar o operador pipe, é necessário instalar e carregar o pacote **magrittr**.

```
install.packages("magrittr")  
library(magrittr)
```

O operador pipe `%>%` permite que o valor de uma expressão à esquerda seja passado como o primeiro argumento para a função à direita. Isso permite que o código seja lido de maneira sequencial, de cima para baixo, em vez de dentro para fora, o que facilita a compreensão do fluxo de dados. Por exemplo, considere o seguinte código em R sem o uso do pipe:

```
x <- c(1, 2, 3, 4)  
sqrt(sum(x))  
## [1] 3.16
```

Com o operador pipe, o código pode ser reescrito de forma mais clara e legível:

```
library(magrittr)
x %>% sum() %>% sqrt()
## [1] 3.16
```

Considere um exemplo onde queremos calcular a raiz quadrada da soma dos logaritmos dos valores absolutos de um vetor `x`:

```
resultado <- sqrt(sum(log(abs(x))))
```

Usando o operador pipe, o mesmo código pode ser reescrito de maneira mais clara:

```
resultado <- x %>%
  abs() %>%
  log() %>%
  sum() %>%
  sqrt()
```

Uso do Ponto (.) no Pipe

Em alguns casos, o resultado do lado esquerdo do operador pipe deve ser passado para um argumento que não é o primeiro na função do lado direito. Para esses casos, usamos um ponto (.) como um marcador para indicar onde o valor deve ser inserido.

Por exemplo, ao usar o pipe para construir um modelo linear com o conjunto de dados `airquality`, queremos que o dataset seja passado como o segundo argumento (`data =`) da função `lm()`:

```
airquality %>%
  na.omit() %>% # Remove valores ausentes
  lm(Ozone ~ Wind + Temp + Solar.R, data = .) %>% # Cria o modelo linear
  summary() # Gera o resumo do modelo
##
## Call:
## lm(formula = Ozone ~ Wind + Temp + Solar.R, data = .)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -40.48  -14.22   -3.55   10.10   95.62
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
```



```
## (Intercept) -64.3421    23.0547    -2.79    0.0062 **
## Wind        -3.3336     0.6544    -5.09    1.5e-06 ***
## Temp         1.6521     0.2535     6.52    2.4e-09 ***
## Solar.R      0.0598     0.0232     2.58    0.0112 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 21.2 on 107 degrees of freedom
## Multiple R-squared:  0.606, Adjusted R-squared:  0.595
## F-statistic: 54.8 on 3 and 107 DF,  p-value: <2e-16
```

Neste exemplo, o dataset `airquality` é pré-processado com `na.omit()` para remover os valores ausentes, e o resultado é usado como o argumento `data` na função `lm()` para criar um modelo linear. O pipe permite que este fluxo de operações seja descrito de forma clara e linear.

10.2 Exercícios

1. Reescreva a expressão abaixo utilizando o `%>%`,

```
round(mean(sum(1:10)/3), digits = 1)
```

Dica: utilize a função `magrittr::divide_by()`, Veja o help da função para mais informações.

```
1:10 %>%
  sum() %>%
  magrittr::divide_by(3) %>%
  mean() %>%
  round(digits=1)
## [1] 18.3
```

2. Reescreva o código abaixo utilizando o `%>%`.

```
x <- rnorm(100)

x.pos <- x[x>0]

media <- mean(x.pos)

saida <- round(media, 2)
```

Dica: utilize a função `magrittr::extract()`. Veja o help da função para mais informações.

```
set.seed(123)
rnorm(100) %>%
  magrittr::extract(>0) %>%
  mean() %>%
  round(digits = 2)
## [1] 0.79
```

3. Sem executar, diga qual a saída do código abaixo. Consulte o help das funções caso precise.

```
2 %>%
  add(2) %>%
  c(6, NA) %>%
  mean(na.rm = T) %>%
  equals(5)
```

- Primeiro, somamos 2 com 2, gerando o valor 4.
- Em seguida, colocamos esse valor num vetor com os valores 6 e NA.
- Depois, calculamos a média desse vetor, desconsiderando o NA, e obtemos o valor 5.
- Finalmente, verificamos se o valor é igual a 5, obtendo TRUE.

Chapter 11

Loop while

A instrução `while` em R é uma estrutura de controle de fluxo que permite executar repetidamente um bloco de código enquanto uma condição específica for verdadeira. Esse tipo de loop é especialmente útil quando o número de iterações não é conhecido antecipadamente e depende de uma condição lógica ser atendida.

```
# Sintaxe  
while (condição) {  
  # Bloco de código a ser executado repetidamente  
}
```

- **condição:** Uma expressão lógica que é avaliada antes de cada iteração do loop. Enquanto essa condição for `TRUE`, o bloco de código dentro do `while` será executado.
- **Bloco de código:** As instruções que são repetidas enquanto a condição for verdadeira.

Funcionamento do Loop While

- **Avaliação da Condição:** Antes de cada execução do bloco de código, a condição é avaliada.
- **Execução do Bloco de Código:** Se a condição for `TRUE`, o bloco de código dentro do `while` é executado.
- **Reavaliação:** Após a execução do bloco de código, a condição é reavaliada. Se a condição continuar a ser `TRUE`, o ciclo repete-se. Se a condição se tornar `FALSE`, o loop termina e o controle do programa continua com a próxima instrução após o `while`.

Exemplo 1: Somando números até um limite. O exemplo seguinte calcula a soma de todos os números de 1 a 10.

```
limite <- 10
soma <- 0
contador <- 1

while (contador <= limite) {
  soma <- soma + contador
  contador <- contador + 1
}

print(paste("A soma dos números de 1 a", limite, "é:", soma))
## [1] "A soma dos números de 1 a 10 é: 55"
```

Neste exemplo:

- O loop while continua enquanto `contador <= limite`.
- Em cada iteração, o valor de `contador` é adicionado a soma, e `contador` é incrementado em 1.
- Quando `contador` excede `limite`, o loop termina.

Exemplo 2: Escrever a tabuada de um número inteiro. O programa abaixo imprime a tabuada de um número fornecido pelo utilizador.

```
n <- as.numeric(readline("Digite um número inteiro: "))

print(paste("Tabuada do", n, ":"))
i <- 1
while (i <= 10){
  print(paste(n, "x", i, "=", n*i))
  i + 1
}
```

Explique por que o programa acima não termina. Qual é o erro no nosso código? O loop no código original não terminava porque a linha `i + 1` não atualizava a variável `i`. A expressão `i + 1` calcula o novo valor, mas não o atribui de volta a `i`. O código correto usa `i <- i + 1` para atualizar a variável `i` a cada iteração.

Exemplo 3: Uso do `break` para interromper um loop. O próximo exemplo demonstra como usar a instrução `break` para sair de um loop antecipadamente.

```
limite <- 10
soma <- 0
contador <- 1
```

```
while (contador <= limite) {  
  soma <- soma + contador  
  print(contador)  
  if (contador == 3){  
    break # Interrompe o loop quando contador é igual a 3  
  }  
  contador <- contador + 1  
}  
## [1] 1  
## [1] 2  
## [1] 3
```

- **break:** A instrução **break** é usada para sair imediatamente do loop, independentemente de a condição de **while** ainda ser **TRUE**.

Considerações Importantes sobre o Uso do **while()**

- **Condição de Paragem:** É fundamental garantir que a condição do **while** eventualmente se torne **FALSE** para evitar loops infinitos que podem bloquear o programa.
- **Incremento/Decremento:** A variável que controla a condição deve ser atualizada corretamente dentro do loop. Falhas em fazer isso corretamente também podem causar loops infinitos.
- **Desempenho:** Loops **while** podem ser menos eficientes do que operações vetorizadas em R. Para grandes conjuntos de dados, considere usar funções vetorizadas como **apply**, **lapply**, **sapply**, etc., para melhorar o desempenho.

11.1 Exercícios

1. Escreva um programa em R que solicite ao utilizador um número inteiro n . O programa deve imprimir todos os números inteiros de 0 até n , um número por linha.
2. Crie um programa em R que solicite ao utilizador um número inteiro, calcule o seu quadrado e exiba o resultado. Em seguida, pergunte ao utilizador se ele deseja continuar ou terminar o programa. O loop deve continuar até que o utilizador escolha terminar.
3. Desenvolva um programa em R que peça ao utilizador um número inteiro n e calcule a soma dos primeiros n inteiros (de 1 até n). O resultado deve ser exibido no console.

4. Escreva um programa em R que solicite ao utilizador um número inteiro n e conte quantos números pares existem no intervalo de 0 até n . O resultado deve ser impresso no console.

5. Use um loop **while** para somar números inteiros positivos até que um número negativo seja inserido.

Crie um programa em R que leia números inteiros inseridos pelo usuário e some-os. O loop deve continuar até que o utilizador insira um número negativo, momento em que o programa deve parar e exibir a soma dos números positivos inseridos.

6. Use um loop **while** para implementar uma contagem regressiva.

Implemente um programa em R que solicite ao utilizador um número inteiro positivo e , em seguida, execute uma contagem regressiva até zero, imprimindo cada número no console.

7. Escreva um programa em R que solicite ao utilizador um número inteiro negativo e , em seguida, conte progressivamente até zero, imprimindo cada número no console.

8. Desenvolva um programa em R que peça ao utilizador para inserir um número. Se o número for positivo, o programa deve contar regressivamente até zero. Se o número for negativo, o programa deve contar progressivamente até zero. Cada número deve ser impresso no console.

9. Crie um programa em R que solicite ao utilizador um número inteiro não negativo e calcule seu fatorial usando um loop **while**. Lembre-se que $0! = 1$ e que $4! = 4 \times 3 \times 2 \times 1 = 24$.

10. Escreva um programa em R que leia um número inteiro positivo e calcule a soma de seus dígitos usando um loop **while**. Por exemplo, para o número 23, o programa deve calcular $2 + 3 = 5$.

11. Desenvolva um programa em R que gere e imprima os números da sequência de Fibonacci até atingir um valor máximo especificado pelo utilizador. A sequência de Fibonacci começa com 1, 1 e cada número subsequente é a soma dos dois anteriores (1, 1, 2, 3, 5, 8, 13...).

Chapter 12

Loop for

O loop `for` em R é uma estrutura de controlo de fluxo que permite executar repetidamente um bloco de código para cada elemento de uma sequência. Ele é particularmente útil quando o número de iterações é conhecido de antecipadamente. O loop `for` é amplamente utilizado em R para iterar sobre vetores, listas, data frames, e outras estruturas de dados.

```
# Sintaxe
for (variável in sequência) {
  # Bloco de código a ser executado
}
```

- **variável:** Uma variável de controlo que, em cada iteração, assume o valor do próximo elemento da sequência.
- **sequência:** Uma estrutura de dados como um vetor, lista, ou outro objeto sobre o qual se deseja iterar.
- **bloco de código:** As instruções a serem executadas a cada iteração do loop.

Funcionamento do Loop for

- **Inicialização:** Antes de iniciar o loop, a variável de controlo é definida com o primeiro elemento da sequência.
- **Iteração:** Para cada iteração, a variável de controlo é atualizada com o próximo valor da sequência.
- **Execução do Bloco de Código:** O bloco de código dentro do loop é executado para cada valor da variável de controlo.

- **Finalização:** O loop termina quando todos os elementos da sequência tiverem sido processados.

Exemplo 1: Imprima os números de 0 a 10 no ecrã.

```
for (i in 0:10) {  
  print(i)  
}  
## [1] 0  
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5  
## [1] 6  
## [1] 7  
## [1] 8  
## [1] 9  
## [1] 10
```

Exemplo 2: Soma dos elementos de um vetor.

```
numeros <- c(1, 2, 3, 4, 5)  
soma <- 0  
  
for (num in numeros) {  
  soma <- soma + num  
}  
  
print(paste("A soma dos números é:", soma))  
## [1] "A soma dos números é: 15"
```

Exemplo 3: Uso do `for` com índices. Também pode usar o loop `for` para iterar sobre índices de vetores ou listas, o que pode ser útil quando se deseja aceder ou modificar elementos em posições específicas.

```
numeros <- c(10, 20, 30, 40, 50)  
  
for (i in 1:length(numeros)) {  
  numeros[i] <- numeros[i] * 2  
}  
  
print("Elementos do vetor multiplicados por 2:")  
print(numeros)  
## [1] 20 40 60 80 100
```


Exemplo 4: Iterar sobre matrizes. O loop `for` também pode ser utilizado para percorrer elementos de uma matriz, seja por linha ou por coluna. No exemplo abaixo, calculamos a soma dos elementos de cada linha de uma matriz.

```
matriz <- matrix(1:9, nrow=3, ncol=3)
soma_linhas <- numeric(nrow(matriz))

for (i in 1:nrow(matriz)) {
  soma_linhas[i] <- sum(matriz[i, ])
}

print("Soma dos elementos de cada linha:")
print(soma_linhas)
## [1] 12 15 18
```

Exemplo 5: Cálculo de Médias de Colunas num Data Frame. Neste exemplo, calculamos a média de cada coluna de um data frame usando um loop `for`.

```
dados <- data.frame(
  A = c(1, 2, 3),
  B = c(4, 5, 6),
  C = c(7, 8, 9)
)

medias <- numeric(ncol(dados))

for (col in 1:ncol(dados)) {
  medias[col] <- mean(dados[, col])
}

print("Médias das colunas do data frame:")
print(medias)
## [1] 2 5 8
```

12.1 Exercícios

1. Escreva um programa em R que use um loop `for` para imprimir todos os números de 1 a 10.
2. Escreva um programa em R que use um loop `for` para imprimir o quadrado de cada número de 1 a 5.
3. Escreva um programa em R que some todos os elementos do vetor `numeros <- c(2, 4, 6, 8, 10)` usando um loop `for`.

4. Escreva um programa em R que calcule a média das notas no vetor `notas <- c(85, 90, 78, 92, 88)` usando um loop `for`.
5. Escreva um programa em R que imprima todos os números ímpares do vetor `valores <- c(1, 3, 4, 6, 9, 11, 14)`. Utilize um loop `for`.
6. Escreva um programa em R que leia um número inteiro e use um loop `for` para imprimir sua a tabuada de 1 a 10.
7. Escreva um programa em R que leia um número inteiro e use um loop `for` para determinar se o número é perfeito ou não. Um número perfeito é aquele cuja soma de seus divisores próprios é igual ao próprio número (por exemplo, 6 tem divisores próprios 1, 2 e 3, e $1+2+3 = 6$).
8. Escreva um programa em R que use um loop `for` para encontrar e imprimir todos os números perfeitos entre 1 e 1000.
9. Escreva um programa em R que leia dois inteiros n e m e imprima os números entre 0 e n , inclusive, com um intervalo de m entre eles. Utilize um loop `for` para o efeito.
10. Escreva um programa que calcule a média de um conjunto de notas fornecidas pelo utilizador. O número de notas a serem inseridas é indicado pelo utilizador antes da entrada das notas. Utilize um loop `for` para recolher as notas e calcular a média. Segue-se um exemplo da interação com o computador:

Digite o número de notas que serão inseridas: 4

Digite a nota: 12

Digite a nota: 13

Digite a nota: 15.5

Digite a nota: 16

A média das notas é 14.125

11. Crie uma matriz 4×4 com números de 1 a 16. Utilizando o comando de repetição `for` calcule:

- (a) a média da terceira coluna da matriz.
- (b) a média da terceira linha da matriz.
- (c) a média de cada coluna da matriz.
- (d) a média de cada linha da matriz.
- (e) a média dos números pares de cada coluna da matriz.
- (f) a média dos números ímpares de cada linha da matriz.

Obs: Use estruturas de decisão (`if...else...`) sempre que necessário para condicionar a lógica dentro dos loops.

Chapter 13

Família `Xapply()`

A família `Xapply()` no R refere-se a um conjunto de funções desenhadas para iterar de forma eficiente sobre objetos, evitando a necessidade de loops (ciclos) explícitos como `for`. Essas funções são especialmente úteis para realizar operações repetitivas em listas, vetores, matrizes, data frames e outros objetos, de forma concisa e frequentemente mais rápida.

Função	Argumentos	Objetivo	Input	Output
<code>apply</code>	<code>apply(x, MARGIN, FUN)</code>	Aplica uma função às linhas ou colunas ou a ambas	Data frame ou matriz	vetor, lista, array
<code>lapply</code>	<code>lapply(x, FUN)</code>	Aplica uma função a todos os elementos da entrada	Lista, vetor ou data frame	lista
<code>sapply</code>	<code>sapply(x, FUN)</code>	Aplica uma função a todos os elementos da entrada	Lista, vetor ou data frame	vetor ou matriz
<code>tapply</code>	<code>tapply(x, INDEX, FUN)</code>	Aplica uma função a cada fator	Vetor ou data frame	array

13.1 Função apply()

A função `apply()` aplica uma função a margens específicas (linhas ou colunas) de uma matriz ou data frame e devolve um vetor, lista ou array. É frequentemente usada para evitar loops explícitos.

```
# Sintaxe  
apply(X, MARGIN, FUN)
```

- X: A matriz ou data frame a ser manipulado.
- MARGIN: Um valor que indica se a função deve ser aplicada às linhas (1) ou colunas (2).
- FUN: A função que será aplicada.

Exemplo: Calcular a soma, a média e a raiz quadrada de cada coluna de uma matriz.

```
matriz <- matrix(1:9, nrow = 3)  
  
apply(matriz, 2, sum)  
## [1] 6 15 24  
  
apply(matriz, 2, mean)  
## [1] 2 5 8  
  
f <- function(x) sqrt(x)  
apply(matriz, 2, f)  
##      [,1] [,2] [,3]  
## [1,] 1.00 2.00 2.65  
## [2,] 1.41 2.24 2.83  
## [3,] 1.73 2.45 3.00
```

13.2 Função lapply()

A função `lapply()` aplica uma função a cada elemento de uma lista ou vetor e devolve uma lista. É útil quando se deseja manter a estrutura de saída como uma lista.

```
# Sintaxe  
lapply(X, FUN, ...)
```

- X: A lista ou vetor de entrada.
- FUN: A função que será aplicada.

Exemplo 1:

```
nomes <- c("ANA", "JOAO", "PAULO", "FILIPA")
nomes_minusc <- lapply(nomes, tolower)
print(nomes_minusc)
## [[1]]
## [1] "ana"
##
## [[2]]
## [1] "joao"
##
## [[3]]
## [1] "paulo"
##
## [[4]]
## [1] "filipa"

str(nomes_minusc)
## List of 4
## $ : chr "ana"
## $ : chr "joao"
## $ : chr "paulo"
## $ : chr "filipa"
```

Exemplo 2:

```
# Aplicar a função sqrt a cada elemento de uma lista
vetor_dados <- list(a = 1:4, b = 5:8)
lapply(vetor_dados, sqrt)
## $a
## [1] 1.00 1.41 1.73 2.00
##
## $b
## [1] 2.24 2.45 2.65 2.83
```

13.3 Função `sapply()`

A função `sapply()` é semelhante à `lapply()`, mas tenta simplificar o resultado para um vetor ou matriz, se possível.

```
# Sintaxe  
sapply(X, FUN, ...)
```

Exemplo

```
dados <- 1:5  
f <- function(x) x^2  
  
lapply(dados, f)  
## [[1]]  
## [1] 1  
##  
## [[2]]  
## [1] 4  
##  
## [[3]]  
## [1] 9  
##  
## [[4]]  
## [1] 16  
##  
## [[5]]  
## [1] 25  
  
sapply(dados, f)  
## [1] 1 4 9 16 25
```

13.4 Função tapply()

A função `tapply()` aplica uma função a subconjuntos de um vetor, divididos de acordo com fatores. É ideal para operações em subconjuntos de dados categorizados.

```
# Sintaxe  
tapply(X, INDEX, FUN, ...)
```

- X: O vetor de dados.
- INDEX: Um fator ou lista de fatores que definem os grupos.
- FUN: A função que será aplicada a cada grupo.

Exemplo: O dataset `iris` em R é um conjunto de dados amplamente utilizado para exemplificar análises estatísticas. Foi introduzido por Ronald A. Fisher

em 1936 no seu artigo sobre a utilização de modelos estatísticos para discriminação de espécies de plantas. Contém informações sobre três espécies de flores (*Setosa*, *Versicolor*, *Virginica*) e suas características morfológicas (comprimento e largura das sépalas e pétalas). Vamos calcular a média do comprimento das pétalas por espécie usando `tapply()`:

```
iris
##      Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 1           5.1         3.5         1.4         0.2    setosa
## 2           4.9         3.0         1.4         0.2    setosa
## 3           4.7         3.2         1.3         0.2    setosa
## 4           4.6         3.1         1.5         0.2    setosa
## 5           5.0         3.6         1.4         0.2    setosa
## 6           5.4         3.9         1.7         0.4    setosa
## 7           4.6         3.4         1.4         0.3    setosa
## 8           5.0         3.4         1.5         0.2    setosa
## 9           4.4         2.9         1.4         0.2    setosa
## 10          4.9         3.1         1.5         0.1    setosa
## 11          5.4         3.7         1.5         0.2    setosa
## 12          4.8         3.4         1.6         0.2    setosa
## 13          4.8         3.0         1.4         0.1    setosa
## 14          4.3         3.0         1.1         0.1    setosa
## 15          5.8         4.0         1.2         0.2    setosa
## 16          5.7         4.4         1.5         0.4    setosa
## 17          5.4         3.9         1.3         0.4    setosa
## 18          5.1         3.5         1.4         0.3    setosa
## 19          5.7         3.8         1.7         0.3    setosa
## 20          5.1         3.8         1.5         0.3    setosa
## 21          5.4         3.4         1.7         0.2    setosa
## 22          5.1         3.7         1.5         0.4    setosa
## 23          4.6         3.6         1.0         0.2    setosa
## 24          5.1         3.3         1.7         0.5    setosa
## 25          4.8         3.4         1.9         0.2    setosa
## 26          5.0         3.0         1.6         0.2    setosa
## 27          5.0         3.4         1.6         0.4    setosa
## 28          5.2         3.5         1.5         0.2    setosa
## 29          5.2         3.4         1.4         0.2    setosa
## 30          4.7         3.2         1.6         0.2    setosa
## 31          4.8         3.1         1.6         0.2    setosa
## 32          5.4         3.4         1.5         0.4    setosa
## 33          5.2         4.1         1.5         0.1    setosa
## 34          5.5         4.2         1.4         0.2    setosa
## 35          4.9         3.1         1.5         0.2    setosa
## 36          5.0         3.2         1.2         0.2    setosa
## 37          5.5         3.5         1.3         0.2    setosa
## 38          4.9         3.6         1.4         0.1    setosa
```

```
## 39      4.4      3.0      1.3      0.2      setosa
## 40      5.1      3.4      1.5      0.2      setosa
## 41      5.0      3.5      1.3      0.3      setosa
## 42      4.5      2.3      1.3      0.3      setosa
## 43      4.4      3.2      1.3      0.2      setosa
## 44      5.0      3.5      1.6      0.6      setosa
## 45      5.1      3.8      1.9      0.4      setosa
## 46      4.8      3.0      1.4      0.3      setosa
## 47      5.1      3.8      1.6      0.2      setosa
## 48      4.6      3.2      1.4      0.2      setosa
## 49      5.3      3.7      1.5      0.2      setosa
## 50      5.0      3.3      1.4      0.2      setosa
## 51      7.0      3.2      4.7      1.4 versicolor
## 52      6.4      3.2      4.5      1.5 versicolor
## 53      6.9      3.1      4.9      1.5 versicolor
## 54      5.5      2.3      4.0      1.3 versicolor
## 55      6.5      2.8      4.6      1.5 versicolor
## 56      5.7      2.8      4.5      1.3 versicolor
## 57      6.3      3.3      4.7      1.6 versicolor
## 58      4.9      2.4      3.3      1.0 versicolor
## 59      6.6      2.9      4.6      1.3 versicolor
## 60      5.2      2.7      3.9      1.4 versicolor
## 61      5.0      2.0      3.5      1.0 versicolor
## 62      5.9      3.0      4.2      1.5 versicolor
## 63      6.0      2.2      4.0      1.0 versicolor
## 64      6.1      2.9      4.7      1.4 versicolor
## 65      5.6      2.9      3.6      1.3 versicolor
## 66      6.7      3.1      4.4      1.4 versicolor
## 67      5.6      3.0      4.5      1.5 versicolor
## 68      5.8      2.7      4.1      1.0 versicolor
## 69      6.2      2.2      4.5      1.5 versicolor
## 70      5.6      2.5      3.9      1.1 versicolor
## 71      5.9      3.2      4.8      1.8 versicolor
## 72      6.1      2.8      4.0      1.3 versicolor
## 73      6.3      2.5      4.9      1.5 versicolor
## 74      6.1      2.8      4.7      1.2 versicolor
## 75      6.4      2.9      4.3      1.3 versicolor
## 76      6.6      3.0      4.4      1.4 versicolor
## 77      6.8      2.8      4.8      1.4 versicolor
## 78      6.7      3.0      5.0      1.7 versicolor
## 79      6.0      2.9      4.5      1.5 versicolor
## 80      5.7      2.6      3.5      1.0 versicolor
## 81      5.5      2.4      3.8      1.1 versicolor
## 82      5.5      2.4      3.7      1.0 versicolor
## 83      5.8      2.7      3.9      1.2 versicolor
```



```

## 84      6.0      2.7      5.1      1.6 versicolor
## 85      5.4      3.0      4.5      1.5 versicolor
## 86      6.0      3.4      4.5      1.6 versicolor
## 87      6.7      3.1      4.7      1.5 versicolor
## 88      6.3      2.3      4.4      1.3 versicolor
## 89      5.6      3.0      4.1      1.3 versicolor
## 90      5.5      2.5      4.0      1.3 versicolor
## 91      5.5      2.6      4.4      1.2 versicolor
## 92      6.1      3.0      4.6      1.4 versicolor
## 93      5.8      2.6      4.0      1.2 versicolor
## 94      5.0      2.3      3.3      1.0 versicolor
## 95      5.6      2.7      4.2      1.3 versicolor
## 96      5.7      3.0      4.2      1.2 versicolor
## 97      5.7      2.9      4.2      1.3 versicolor
## 98      6.2      2.9      4.3      1.3 versicolor
## 99      5.1      2.5      3.0      1.1 versicolor
## 100     5.7      2.8      4.1      1.3 versicolor
## 101     6.3      3.3      6.0      2.5 virginica
## 102     5.8      2.7      5.1      1.9 virginica
## 103     7.1      3.0      5.9      2.1 virginica
## 104     6.3      2.9      5.6      1.8 virginica
## 105     6.5      3.0      5.8      2.2 virginica
## 106     7.6      3.0      6.6      2.1 virginica
## 107     4.9      2.5      4.5      1.7 virginica
## 108     7.3      2.9      6.3      1.8 virginica
## 109     6.7      2.5      5.8      1.8 virginica
## 110     7.2      3.6      6.1      2.5 virginica
## 111     6.5      3.2      5.1      2.0 virginica
## 112     6.4      2.7      5.3      1.9 virginica
## 113     6.8      3.0      5.5      2.1 virginica
## 114     5.7      2.5      5.0      2.0 virginica
## 115     5.8      2.8      5.1      2.4 virginica
## 116     6.4      3.2      5.3      2.3 virginica
## 117     6.5      3.0      5.5      1.8 virginica
## 118     7.7      3.8      6.7      2.2 virginica
## 119     7.7      2.6      6.9      2.3 virginica
## 120     6.0      2.2      5.0      1.5 virginica
## 121     6.9      3.2      5.7      2.3 virginica
## 122     5.6      2.8      4.9      2.0 virginica
## 123     7.7      2.8      6.7      2.0 virginica
## 124     6.3      2.7      4.9      1.8 virginica
## 125     6.7      3.3      5.7      2.1 virginica
## 126     7.2      3.2      6.0      1.8 virginica
## 127     6.2      2.8      4.8      1.8 virginica
## 128     6.1      3.0      4.9      1.8 virginica

```

```
## 129      6.4      2.8      5.6      2.1 virginica
## 130      7.2      3.0      5.8      1.6 virginica
## 131      7.4      2.8      6.1      1.9 virginica
## 132      7.9      3.8      6.4      2.0 virginica
## 133      6.4      2.8      5.6      2.2 virginica
## 134      6.3      2.8      5.1      1.5 virginica
## 135      6.1      2.6      5.6      1.4 virginica
## 136      7.7      3.0      6.1      2.3 virginica
## 137      6.3      3.4      5.6      2.4 virginica
## 138      6.4      3.1      5.5      1.8 virginica
## 139      6.0      3.0      4.8      1.8 virginica
## 140      6.9      3.1      5.4      2.1 virginica
## 141      6.7      3.1      5.6      2.4 virginica
## 142      6.9      3.1      5.1      2.3 virginica
## 143      5.8      2.7      5.1      1.9 virginica
## 144      6.8      3.2      5.9      2.3 virginica
## 145      6.7      3.3      5.7      2.5 virginica
## 146      6.7      3.0      5.2      2.3 virginica
## 147      6.3      2.5      5.0      1.9 virginica
## 148      6.5      3.0      5.2      2.0 virginica
## 149      6.2      3.4      5.4      2.3 virginica
## 150      5.9      3.0      5.1      1.8 virginica

tapply(iris$Petal.Length, iris$Species, mean)
##      setosa versicolor virginica
##      1.46      4.26      5.55
```

13.5 Exercícios

1. Crie um vetor chamado `nomes` contendo os nomes “Alice”, “Bruno”, “Carla”, e “Daniel”. Use a função `lapply()` para converter todos os nomes para maiúsculas. Verifique o resultado.

```
nomes <- c("Alice", "Bruno", "Carla", "Daniel")
# Sua solução aqui
```

2. Crie um vetor chamado `palavras` contendo as palavras “cachorro”, “gato”, “elefante”, e “leão”. Use a função `sapply()` para calcular o comprimento (número de caracteres) de cada palavra. Verifique o resultado. Dica, veja `?nchar`.

```
palavras <- c("cachorro", "gato", "elefante", "leão")
# Sua solução aqui
```

3. Crie uma matriz 3x3 chamada `matriz_exemplo` com os números de 1 a 9. Use a função `apply()` para calcular a soma de cada linha da matriz. Em seguida, use a função `apply()` novamente para calcular a média de cada coluna.

```
matriz_exemplo <- matrix(1:9, nrow = 3)
# Sua solução aqui
```

4. Use o conjunto de dados embutido `mtcars` em R. Use a função `tapply()` para calcular a média do consumo de combustível (`mpg`) para cada número de cilindros (`cyl`).

```
data(mtcars)
# Sua solução aqui
```

5. Crie uma lista chamada `minha_lista` contendo três elementos: um vetor numérico de 1 a 5, um vetor de caracteres com os valores “A”, “B”, “C”, e uma matriz 2x2 com valores de 1 a 4. Use `lapply()` para calcular o comprimento de cada elemento da lista e, em seguida, use `sapply()` para simplificar o resultado num vetor.

```
minha_lista <- list(1:5, c("A", "B", "C"), matrix(1:4, nrow = 2))
# Sua solução aqui
```

6. Crie dois vetores numéricos, `a <- c(2, 4, 6)` e `b <- c(3, 5, 7)`. Use `mapply()` para multiplicar cada elemento de `a` com o elemento correspondente de `b`. Verifique o resultado.

```
a <- c(2, 4, 6)
b <- c(3, 5, 7)
# Sua solução aqui
```

7. Crie um vetor `numeros` contendo os números de 1 a 10. Use a função `sapply()` para aplicar uma função personalizada que retorne “par” se o número for par e “ímpar” se o número for ímpar.

```
numeros <- 1:10
# Sua solução aqui
```

8. Use o conjunto de dados `iris`. Utilize `tapply()` para calcular a média do comprimento das pétalas (`Petal.Length`) por espécie (`Species`) e a variância da largura das sépalas (`Sepal.Width`) por espécie.

```
data(iris)
# Sua solução aqui
```

9. Use o conjunto de dados `mtcars`. Use `apply()` para normalizar (escala de 0 a 1) todas as colunas numéricas do data frame. Em seguida, use `lapply()` para aplicar a função `summary()` a cada coluna normalizada para verificar o resultado.

```
data(mtcars)
# Sua solução aqui
```

Chapter 14

Gráficos (R base)

O R base oferece uma variedade de funções para criar gráficos estatísticos básicos e personalizá-los de acordo com as necessidades da análise. A criação de gráficos é uma parte essencial da análise de dados, pois permite visualizar padrões, tendências e distribuições de forma intuitiva.

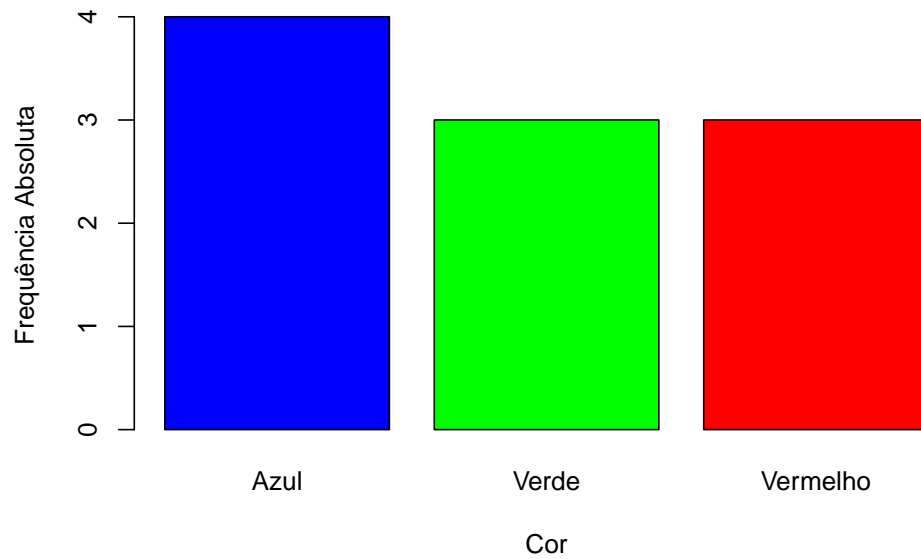
14.1 Gráfico de Barras

Gráfico de barras: Um gráfico de barras consiste em barras verticais ou horizontais, onde cada barra representa uma categoria e sua altura (ou comprimento) indica a frequência (absoluta ou relativa) dessa categoria. A largura das barras é constante e não possui significado estatístico.

```
# Dados de exemplo: cores favoritas
cores <- c("Azul", "Vermelho", "Verde", "Azul", "Verde",
"Vermelho", "Azul", "Verde", "Azul", "Vermelho")

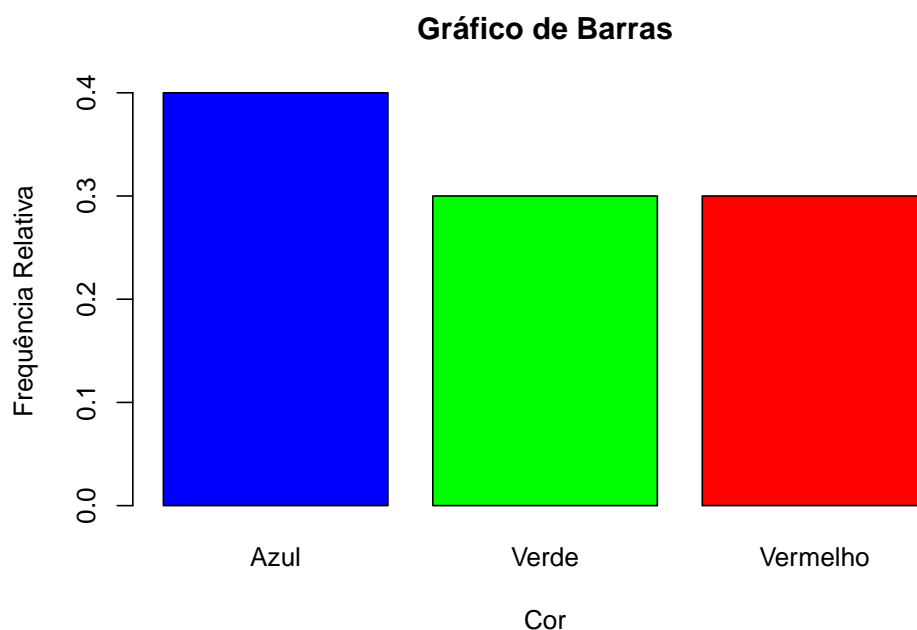
# Calcular as frequências absolutas
frequencia_absoluta <- table(cores)

# Criar o gráfico de barras com frequências absolutas
barplot(frequencia_absoluta,
  main = "Gráfico de Barras",
  xlab = "Cor",
  ylab = "Frequência Absoluta",
  col = c("blue", "green", "red"),
  border = "black")
```

Gráfico de Barras

```
# Calcular as frequências relativas
frequencia_relativa <- frequencia_absoluta / length(cores)

# Criar o gráfico de barras com frequências relativas
barplot(frequencia_relativa,
  main = "Gráfico de Barras",
  xlab = "Cor",
  ylab = "Frequência Relativa",
  col = c("blue", "green", "red"),
  border = "black")
```

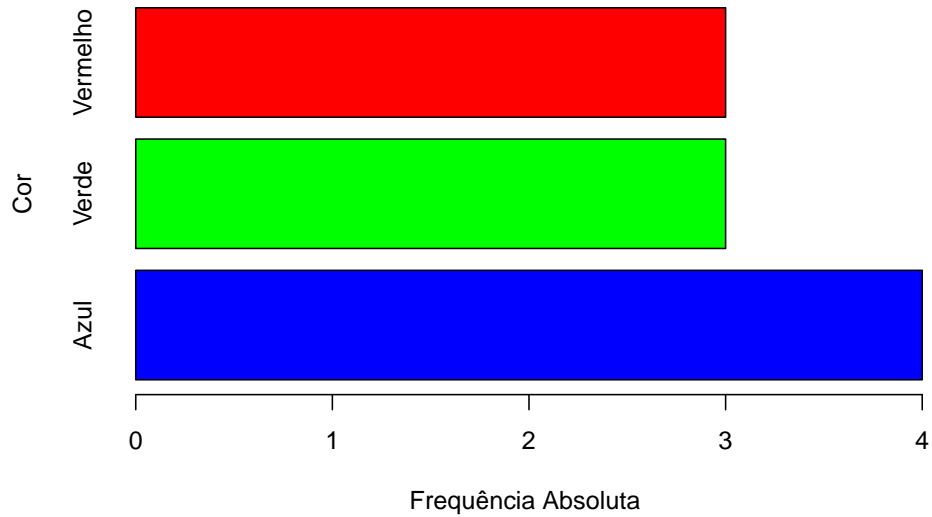


Personalização de Gráficos de Barras

Pode personalizar ainda mais os gráficos de barras usando argumentos como `horiz` (para gráficos de barras horizontais), e `beside` (para barras lado a lado).

```
# Gráfico de barras horizontal
barplot(frequencia_absoluta,
        main = "Gráfico de Barras Horizontal",
        xlab = "Frequência Absoluta",
        ylab = "Cor",
        col = c("blue", "green", "red"),
        horiz = TRUE)
```

Gráfico de Barras Horizontal

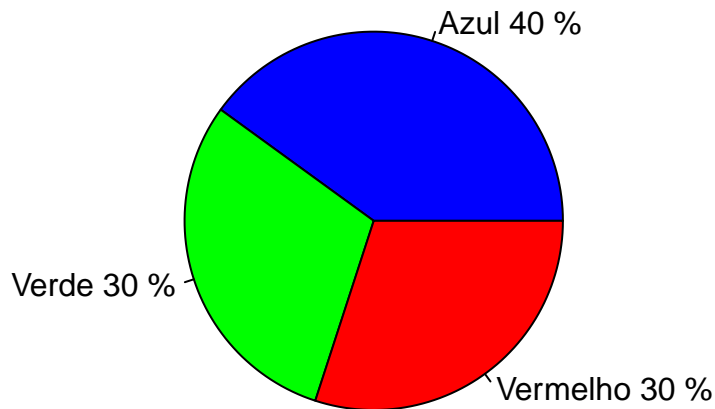


14.2 Gráfico Circular (Pizza)

Gráfico circular (pizza): Mostra a proporção de diferentes categorias em relação ao total. Cada fatia representa uma categoria e o tamanho de cada fatia é proporcional à sua porcentagem.

```
# Criar gráfico circular
pie(frequencia_relativa,
    main = "Gráfico Circular (Pizza)",
    col = c("blue", "green", "red"),
    labels = paste(names(frequencia_relativa),
        round(frequencia_relativa * 100, 1), "%"))
```


Gráfico Circular (Pizza)



14.3 Histograma

Histograma: é uma representação gráfica dos dados em que se marcam as classes (intervalos) no eixo horizontal e as frequências (absoluta ou relativa) no eixo vertical.

- Cada retângulo corresponde a uma classe.
- A largura de cada retângulo é igual à amplitude da classe.
- Se as classes tiverem todas a mesma amplitude, a altura do retângulo é proporcional à frequência.

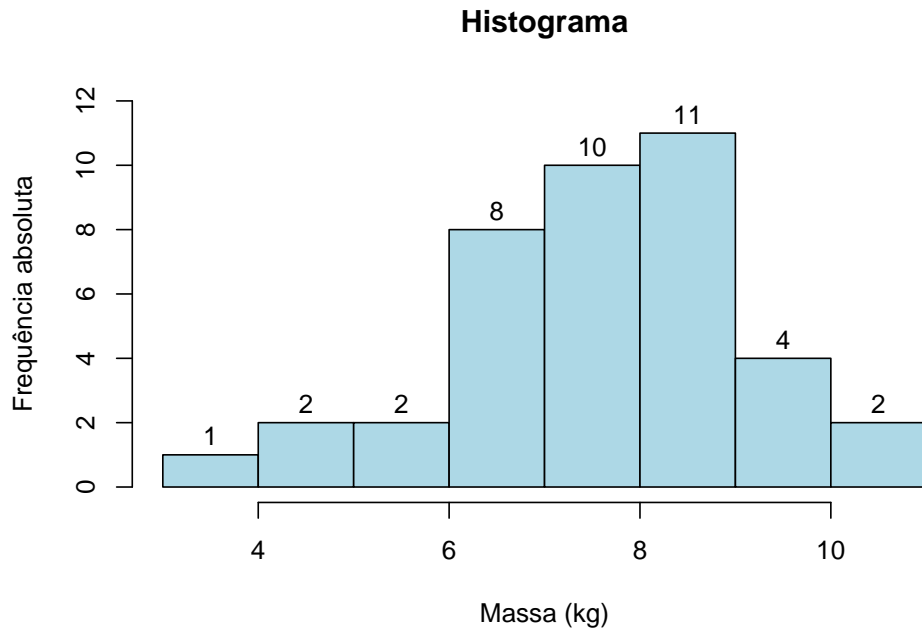
Por default, o R utiliza a frequência absoluta para construir o histograma. Se tiver interesse em representar as frequências relativas, utilize a opção `freq=FALSE` nos argumentos da função `hist()`. O padrão de intervalo de classe no R é $(a, b]$.

```
# Dados referentes à massa (em kg) de 40 bicicletas
```

```
bicicletas <- c(4.3,6.8,9.2,7.2,8.7,8.6,6.6,5.2,8.1,10.9,7.4,4.5,3.8,7.6,6.8,7.8,8.4,7.5,10.5,6.0,
```

```
h <- hist(bicicletas,  
  main = "Histograma",  
  xlab = "Massa (kg)",  
  ylab = "Frequência absoluta",
```

```
col = "lightblue",
border = "black",
ylim = c(0,12),
labels = TRUE)
```

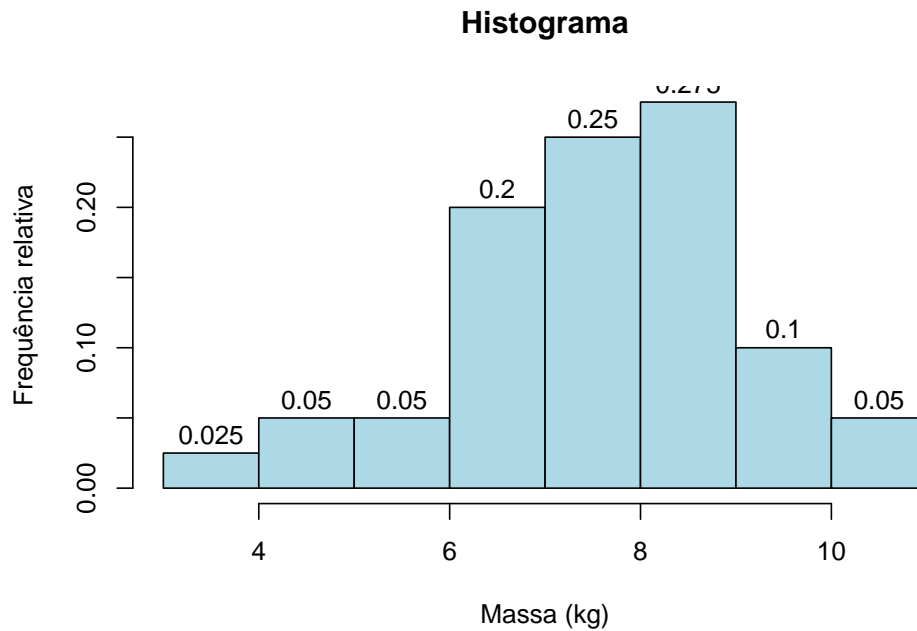


```
# Pontos limites das classes
h$breaks
## [1] 3 4 5 6 7 8 9 10 11
```

```
# O comando h$counts retorna um vetor com as frequências absolutas dentro de cada classe
h$counts
## [1] 1 2 2 8 10 11 4 2
```

```
# Histograma com frequência relativa
```

```
hist(bicicletas,
     main = "Histograma",
     xlab = "Massa (kg)",
     ylab = "Frequência relativa",
     freq = FALSE,
     col = "lightblue",
     border = "black",
     labels = TRUE)
```



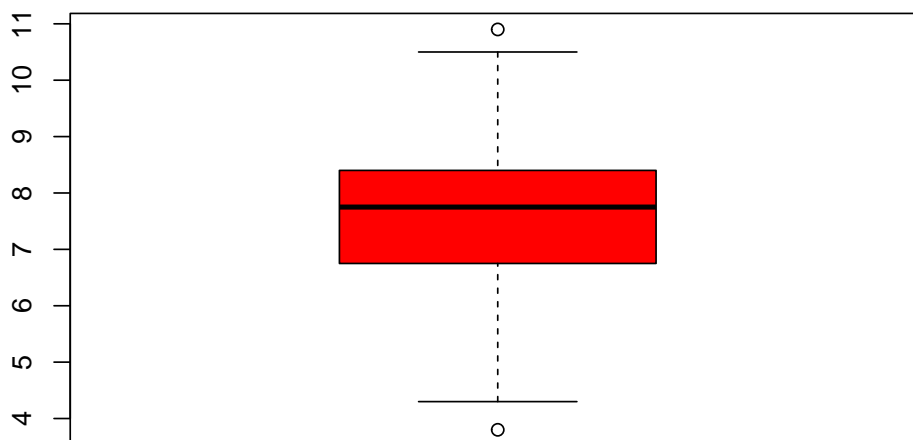
Pode controlar o número de intervalos (bins) usando o argumento `breaks`. Por exemplo, `breaks = 5` divide os dados em 5 intervalos.

14.4 Box-plot

Box-plot (Caixa de Bigodes): O box-plot é uma representação gráfica que resume a distribuição de uma variável através de cinco estatísticas principais: mínimo, primeiro quartil (Q1), mediana (Q2), terceiro quartil (Q3) e máximo. Também pode mostrar outliers (valores atípicos).

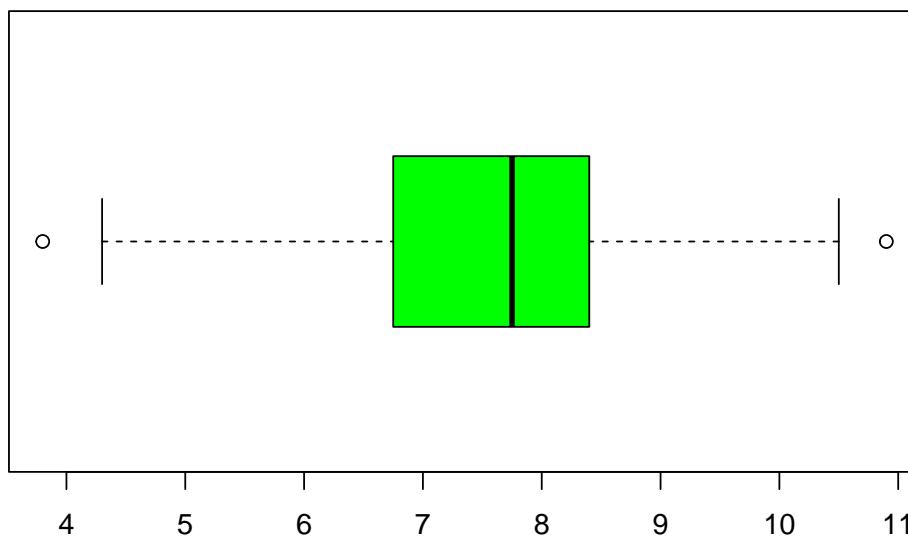
```
# Box-plot vertical
boxplot(bicicletas,
        main = "Box-plot Vertical",
        col="red")
```

Box-plot Vertical



```
# Box-plot horizontal  
boxplot(bicicletas,  
        main = "Box-plot Horizontal",  
        col="green",  
        horizontal = TRUE)
```

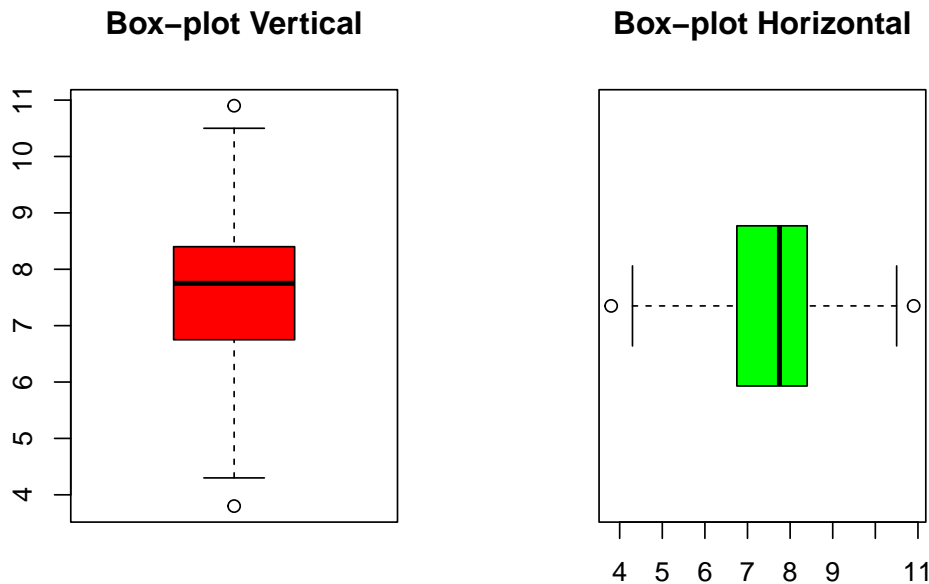
Box-plot Horizontal



Pode comparar a distribuição de múltiplas variáveis lado a lado com box-plots. Use a função `par()` para ajustar o layout gráfico.

```
# Ajuste do layout gráfico
par(mfrow = c(1, 2))

# Box-plot vertical e horizontal lado a lado
boxplot(bicicletas, main = "Box-plot Vertical", col = "red")
boxplot(bicicletas, main = "Box-plot Horizontal", col = "green", horizontal = TRUE)
```



```
# Resetar layout gráfico para o padrão
par(mfrow = c(1, 1))
```

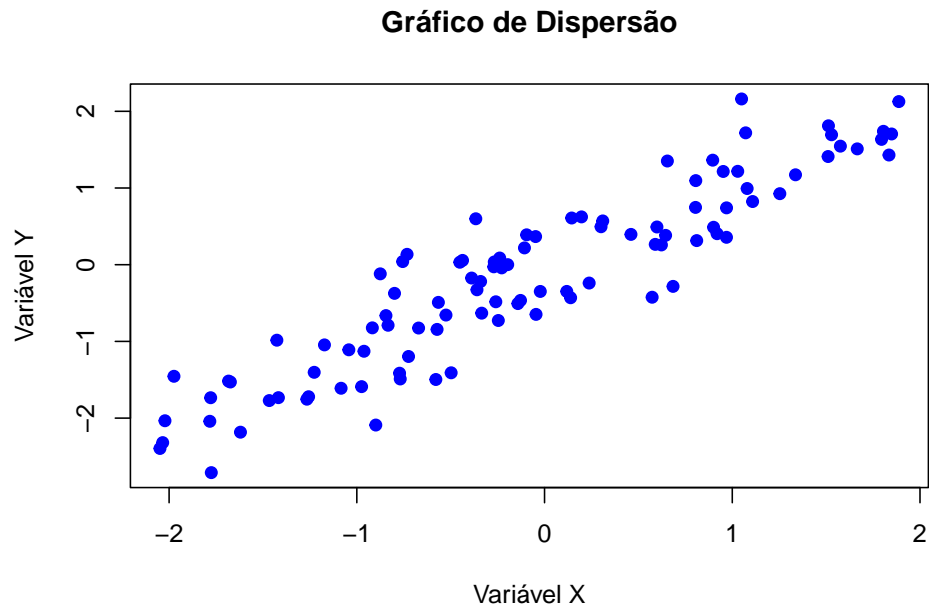
14.5 Gráfico de Dispersão

Gráfico de Dispersão (Scatter Plot): Útil para visualizar a relação entre duas variáveis quantitativas. É uma boa ferramenta para identificar correlações, tendências e outliers.

```
# Dados de exemplo
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)

# Criar gráfico de dispersão
plot(x, y,
     main = "Gráfico de Dispersão",
     xlab = "Variável X",
```

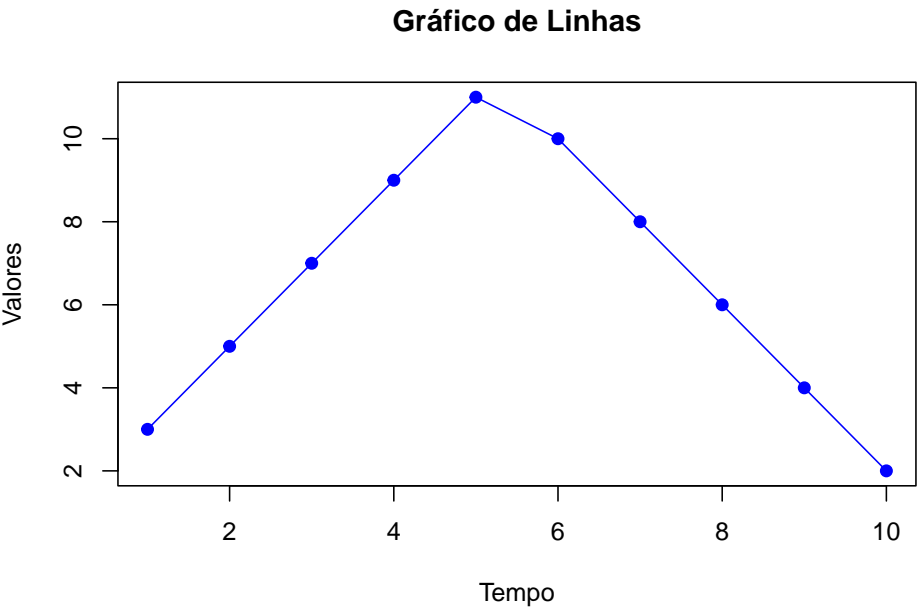
```
ylab = "Variável Y",  
col = "blue",  
pch = 19)
```



14.6 Gráfico de Linhas

Gráfico de Linhas: É usado para mostrar informações de séries temporais ou tendências ao longo de intervalos contínuos, como o tempo.

```
# Dados de exemplo  
tempo <- 1:10  
valores <- c(3, 5, 7, 9, 11, 10, 8, 6, 4, 2)  
  
# Criar gráfico de linhas  
plot(tempo, valores,  
      type = "o",  
      main = "Gráfico de Linhas",  
      xlab = "Tempo",  
      ylab = "Valores",  
      col = "blue",  
      pch = 19)
```



Chapter 15

Manipulação de dados

15.1 Tibbles

Tibbles são uma versão moderna e melhorada dos data frames em R, introduzida pelo pacote `tibble`, que faz parte do **tidyverse** — um conjunto de pacotes desenvolvido para facilitar a ciência de dados em R. Os tibbles foram projetados para ultrapassar algumas das limitações dos data frames tradicionais, oferecendo uma estrutura de dados mais robusta e intuitiva para manipulação e análise de dados.

Diferenças Principais entre Tibble e Data Frame

Os tibbles têm várias vantagens em relação aos data frames, especialmente em termos de usabilidade e integração com o **tidyverse**. A tabela abaixo resume as principais diferenças:

Característica	Data Frame	Tibble
Impressão no Console	Exibe todos os dados	Exibe um resumo com 10 linhas e colunas visíveis
Conversão de Strings	Converte strings para fatores por padrão	Mantém strings como caracteres
Manuseio de Colunas	Apenas vetores, listas podem ser problemáticas	Permite listas, funções, outros tibbles
Nomes de Colunas	Nomes devem ser únicos e sem espaços	Nomes podem ter espaços, ser duplicados, etc.
Retorno de Subsetting (<code>[]</code>)	Pode retornar um vetor ou data frame	Sempre retorna um tibble
Compatibilidade e Integração	Totalmente compatível com base R	Compatível, mas otimizado para uso com tidyverse

Característica	Data Frame	Tibble
Manipulação de Dados	Menos intuitivo para algumas operações.	Mais intuitivo e fácil de usar, especialmente com operações encadeadas.

15.1.1 Criar e trabalhar com Tibbles

Criar um `tibble` é semelhante a criar um data frame, mas com uma sintaxe ligeiramente diferente que facilita a criação de colunas complexas.

```
# Carregar o pacote tibble
library(tibble)

# Criando um tibble
tb <- tibble(
  x = 1:5,
  y = c("A", "B", "C", "D", "E"),
  z = x * 2
)

print(tb)
## # A tibble: 5 × 3
##       x y       z
##   <int> <chr> <dbl>
## 1     1 A         2
## 2     2 B         4
## 3     3 C         6
## 4     4 D         8
## 5     5 E        10
```

Também pode converter um data frame existente para um tibble usando a função `as_tibble()`:

```
# Criar um data frame
df <- data.frame(
  x = 1:5,
  y = c("A", "B", "C", "D", "E")
)

# Converter para tibble
tb <- as_tibble(df)

print(tb)
```

```
## # A tibble: 5 × 2
##       x y
##   <int> <chr>
## 1     1 A
## 2     2 B
## 3     3 C
## 4     4 D
## 5     5 E
```

15.2 Manipulação de Dados com dplyr

O `dplyr` é um pacote fundamental no `tidyverse` para manipulação de dados em R. Ele proporciona uma sintaxe mais limpa e legível para operações comuns em data frames e tibbles, como seleção, filtragem, mutação (criação/modificação de colunas), ordenação e agrupamento de dados.

As principais funções do `dplyr` são:

- `select()`: Seleciona colunas de um data frame. `*arrange()`: Ordena as linhas de um data frame com base nos valores de uma ou mais colunas.
- `filter()`: Filtra linhas com base em condições lógicas.
- `mutate()`: Adiciona novas colunas ou modifica colunas existentes.
- `group_by()`: Agrupa os dados com base nos valores de uma ou mais colunas.
- `summarise()`: Resume os dados, tipicamente usado após `group_by()` para calcular estatísticas agregadas.

Vamos explorar cada uma dessas funções usando o dataset `starwars` disponível no pacote `dplyr`.

Assim, utilizaremos o objeto `sw` para acessar os dados.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##   filter, lag

## The following objects are masked from 'package:base':
##
##   intersect, setdiff, setequal, union
```

```
# Carregar o dataset Star Wars
sw <- starwars
glimpse(sw) # Exibe uma visão geral dos dados
```

```
## Rows: 87
## Columns: 14
## $ name      <chr> "Luke Skywalker", "C-3P0", "R2-D2", "Darth Vader", "Leia Or~
## $ height    <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2~
## $ mass      <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.~
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N~
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "~
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",~
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, ~
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female",~
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini~
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T~
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma~
## $ films      <list> <"A New Hope", "The Empire Strikes Back", "Return of the J~
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp~
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",~
```

15.2.1 Seleção de Colunas com select()

A função `select()` é usada para selecionar uma ou mais colunas de um tibble ou data frame. Pode usar nomes de colunas diretamente sem aspas.

```
# Selecionar uma coluna
select(sw, name)
```

```
## # A tibble: 87 x 1
##   name
##   <chr>
## 1 Luke Skywalker
## 2 C-3P0
## 3 R2-D2
## 4 Darth Vader
## 5 Leia Organa
## 6 Owen Lars
## 7 Beru Whitesun Lars
## 8 R5-D4
## 9 Biggs Darklighter
## 10 Obi-Wan Kenobi
## # i 77 more rows
```

```
# Selecionar várias colunas
select(sw, name, mass, hair_color)
```

```
## # A tibble: 87 x 3
##   name                mass hair_color
##   <chr>              <dbl> <chr>
## 1 Luke Skywalker      77 blond
## 2 C-3PO                75 <NA>
## 3 R2-D2                32 <NA>
## 4 Darth Vader        136 none
## 5 Leia Organa         49 brown
## 6 Owen Lars          120 brown, grey
## 7 Beru Whitesun Lars   75 brown
## 8 R5-D4                32 <NA>
## 9 Biggs Darklighter   84 black
## 10 Obi-Wan Kenobi      77 auburn, white
## # i 77 more rows
```

```
# Selecionar um intervalo de colunas
select(sw, name:hair_color)
```

```
## # A tibble: 87 x 4
##   name                height mass hair_color
##   <chr>              <int> <dbl> <chr>
## 1 Luke Skywalker      172    77 blond
## 2 C-3PO                167    75 <NA>
## 3 R2-D2                96    32 <NA>
## 4 Darth Vader        202   136 none
## 5 Leia Organa         150    49 brown
## 6 Owen Lars          178   120 brown, grey
## 7 Beru Whitesun Lars   165    75 brown
## 8 R5-D4                97    32 <NA>
## 9 Biggs Darklighter   183    84 black
## 10 Obi-Wan Kenobi      182    77 auburn, white
## # i 77 more rows
```

O dplyr possui um conjunto de funções auxiliares muito úteis para seleção de colunas. As principais são:

- `starts_with()`: para colunas que começam com um texto específico.
- `ends_with()`: para colunas que terminam com um texto específico.
- `contains()`: para colunas que contêm um texto específico.

```
# Selecionar colunas que terminam com "color"
select(sw, ends_with("color"))
```

```
## # A tibble: 87 x 3
##   hair_color skin_color eye_color
##   <chr>      <chr>      <chr>
## 1 blond      fair          blue
## 2 <NA>       gold          yellow
## 3 <NA>       white, blue   red
## 4 none       white         yellow
## 5 brown      light         brown
## 6 brown, grey light         blue
## 7 brown      light         blue
## 8 <NA>       white, red    red
## 9 black      light         brown
## 10 auburn, white fair         blue-gray
## # i 77 more rows
```

Para remover colunas, basta acrescentar um - antes da seleção.

```
select(sw, -name, -hair_color)
```

```
## # A tibble: 87 x 12
##   height mass skin_color eye_color birth_year sex gender homeworld species
##   <int> <dbl> <chr>      <chr>      <dbl> <chr> <chr> <chr> <chr>
## 1 172 77 fair          blue          19 male mascu~ Tatooine Human
## 2 167 75 gold          yellow        112 none mascu~ Tatooine Droid
## 3 96 32 white, blue   red           33 none mascu~ Naboo Droid
## 4 202 136 white         yellow        41.9 male mascu~ Tatooine Human
## 5 150 49 light         brown          19 female femin~ Alderaan Human
## 6 178 120 light         blue           52 male mascu~ Tatooine Human
## 7 165 75 light         blue           47 female femin~ Tatooine Human
## 8 97 32 white, red    red            NA none mascu~ Tatooine Droid
## 9 183 84 light         brown           24 male mascu~ Tatooine Human
## 10 182 77 fair         blue-gray       57 male mascu~ Stewjon Human
## # i 77 more rows
## # i 3 more variables: films <list>, vehicles <list>, starships <list>
```

```
# Remover múltiplas colunas
select(sw, -ends_with("color"))
```

```
## # A tibble: 87 x 11
##   name height mass birth_year sex gender homeworld species films vehicles
```

```
##      <chr>      <int> <dbl>      <dbl> <chr> <chr> <chr>      <chr> <lis> <list>
## 1 Luke S~      172    77          19  male  mascu~ Tatooine Human <chr> <chr>
## 2 C-3P0        167    75         112  none  mascu~ Tatooine Droid <chr> <chr>
## 3 R2-D2         96    32          33  none  mascu~ Naboo   Droid <chr> <chr>
## 4 Darth ~      202   136         41.9 male  mascu~ Tatooine Human <chr> <chr>
## 5 Leia O~      150    49          19  fema~ femin~ Alderaan Human <chr> <chr>
## 6 Owen L~      178   120          52  male  mascu~ Tatooine Human <chr> <chr>
## 7 Beru W~      165    75          47  fema~ femin~ Tatooine Human <chr> <chr>
## 8 R5-D4         97    32          NA  none  mascu~ Tatooine Droid <chr> <chr>
## 9 Biggs ~      183    84          24  male  mascu~ Tatooine Human <chr> <chr>
## 10 Obi-Wa~     182    77          57  male  mascu~ Stewjon Human <chr> <chr>
## # i 77 more rows
## # i 1 more variable: starships <list>
```

15.2.2 Exercícios

Utilize a base `sw` nos exercícios a seguir.

1. Utilize a função `glimpse()` do pacote `dplyr` na base `sw`. O que ela faz?
2. Crie uma tabela chamada `sw_simple` contendo apenas as colunas `name`, `gender` e `films`.
3. Selecione apenas as colunas `hair_color`, `skin_color` e `eye_color` usando a função auxiliar `contains()`.
4. Usando a função `select()` e as suas funções auxiliares, escreva códigos que retornem a base `sw` sem as colunas `hair_color`, `skin_color` e `eye_color`. Escreva todas as soluções diferentes que conseguir pensar.

15.2.3 Ordenando a Base de Dados

A função `arrange()` permite ordenar as linhas de uma tabela com base nos valores de uma ou mais colunas. Pode ordenar em ordem crescente ou decrescente, conforme a necessidade.

```
# Exemplo de ordenação crescente
arrange(sw, mass)
```

```
## # A tibble: 87 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Ratts T~      79    15 none      grey, blue unknown      NA male  mascu~
## 2 Yoda         66    17 white      green      brown      896 male  mascu~
## 3 Wicket ~      88    20 brown      brown      brown        8 male  mascu~
```

```
## 4 R2-D2      96    32 <NA>      white, bl~ red      33 none mascu~
## 5 R5-D4      97    32 <NA>      white, red red      NA none mascu~
## 6 Sebulba   112    40 none      grey, red  orange      NA male mascu~
## 7 Padmé A~  185    45 brown     light      brown      46 fema~ femin~
## 8 Dud Bolt   94    45 none      blue, grey yellow     NA male mascu~
## 9 Wat Tam~  193    48 none      green, gr~ unknown     NA male mascu~
## 10 Sly Moo~ 178    48 none      pale       white      NA <NA> <NA>
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
# Exemplo de ordenação decrescente
arrange(sw, desc(mass))
```

```
## # A tibble: 87 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Jabba D~    175  1358 <NA>      green-tan~ orange      600  herm~ mascu~
## 2 Grievous    216  159 none      brown, wh~ green, y~    NA   male mascu~
## 3 IG-88       200  140 none      metal      red         15   none mascu~
## 4 Darth V~    202  136 none      white      yellow      41.9 male mascu~
## 5 Tarfful     234  136 brown     brown      blue        NA   male mascu~
## 6 Owen La~    178  120 brown, gr~ light      blue        52   male mascu~
## 7 Bossk       190  113 none      green      red         53   male mascu~
## 8 Chewbac~    228  112 brown     unknown    blue        200  male mascu~
## 9 Jek Ton~    180  110 brown     fair       blue        NA   <NA> <NA>
## 10 Dexter ~   198  102 none      brown      yellow      NA   male mascu~
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
# Ordenar por múltiplas colunas
arrange(sw, desc(height), desc(mass))
```

```
## # A tibble: 87 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Yarael ~    264    NA none      white      yellow      NA   male mascu~
## 2 Tarfful     234  136 brown     brown      blue        NA   male mascu~
## 3 Lama Su     229   88 none      grey       black       NA   male mascu~
## 4 Chewbac~    228  112 brown     unknown    blue        200  male mascu~
## 5 Roos Ta~    224   82 none      grey       orange      NA   male mascu~
## 6 Grievous    216  159 none      brown, wh~ green, y~    NA   male mascu~
## 7 Taun We     213    NA none      grey       black       NA   fema~ femin~
```



```
## 8 Tion Me~      206      80 none      grey      black      NA      male mascul~
## 9 Rugor N~      206      NA none      green      orange      NA      male mascul~
## 10 Darth V~     202     136 none      white      yellow     41.9 male mascul~
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

15.2.4 Exercícios

1. Ordene a base `sw` pela coluna `mass` em ordem crescente e pela coluna `birth_year` em ordem decrescente. Guarde o resultado em um objeto chamado `sw_ordenados`.

```
sw_ordenados <- arrange(sw, mass, desc(birth_year))
sw_ordenados
```

2. Selecione apenas as colunas `name` e `birth_year`, então ordene de forma decrescente pelo `birth_year`.

```
# Usando aninhamento de funções
arrange(select(sw, name, birth_year), desc(birth_year))

# Usando um objeto intermediário
sw_aux <- select(sw, name, birth_year)
arrange(sw_aux, desc(birth_year))

# Usando pipes para simplificar
sw %>%
  select(name, birth_year) %>%
  arrange(desc(birth_year))
```

15.2.5 Filtrando Linhas da Base de Dados

Para selecionar linhas específicas com base em uma ou mais condições, utilizamos a função `filter()`.

```
# filter(sw, height > 170)
# Ou
sw %>% filter(height > 170)
```

```
## # A tibble: 55 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
```

```
## 1 Luke Sk~ 172 77 blond fair blue 19 male mascu~
## 2 Darth V~ 202 136 none white yellow 41.9 male mascu~
## 3 Owen La~ 178 120 brown, gr~ light blue 52 male mascu~
## 4 Biggs D~ 183 84 black light brown 24 male mascu~
## 5 Obi-Wan~ 182 77 auburn, w~ fair blue-gray 57 male mascu~
## 6 Anakin ~ 188 84 blond fair blue 41.9 male mascu~
## 7 Wilhuff~ 180 NA auburn, g~ fair blue 64 male mascu~
## 8 Chewbac~ 228 112 brown unknown blue 200 male mascu~
## 9 Han Solo 180 80 brown fair brown 29 male mascu~
## 10 Greedo 173 74 <NA> green black 44 male mascu~
## # i 45 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## # vehicles <list>, starships <list>
```

Podemos selecionar apenas as colunas `name` e `height` para visualizarmos as alturas:

```
sw %>%
  filter(height > 170) %>%
  select(name, height)
```

```
## # A tibble: 55 x 2
##   name          height
##   <chr>         <int>
## 1 Luke Skywalker 172
## 2 Darth Vader    202
## 3 Owen Lars     178
## 4 Biggs Darklighter 183
## 5 Obi-Wan Kenobi 182
## 6 Anakin Skywalker 188
## 7 Wilhuff Tarkin 180
## 8 Chewbacca     228
## 9 Han Solo      180
## 10 Greedo        173
## # i 45 more rows
```

Podemos estender o filtro para duas ou mais colunas.

```
filter(sw, height>170, mass>80)
```

```
## # A tibble: 21 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>    <int> <dbl> <chr>    <chr>    <chr>    <dbl> <chr> <chr>
## 1 Darth V~    202 136 none    white    yellow    41.9 male  mascu~
```

```
## 2 Owen La~ 178 120 brown, gr~ light blue 52 male mascu~
## 3 Biggs D~ 183 84 black light brown 24 male mascu~
## 4 Anakin ~ 188 84 blond fair blue 41.9 male mascu~
## 5 Chewbac~ 228 112 brown unknown blue 200 male mascu~
## 6 Jabba D~ 175 1358 <NA> green-tan~ orange 600 herm~ mascu~
## 7 Jek Ton~ 180 110 brown fair blue NA <NA> <NA>
## 8 IG-88 200 140 none metal red 15 none mascu~
## 9 Bossk 190 113 none green red 53 male mascu~
## 10 Ackbar 180 83 none brown mot~ orange 41 male mascu~
## # i 11 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## # vehicles <list>, starships <list>
```

```
sw %>% filter(height > 170, mass > 80)
```

```
## # A tibble: 21 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Darth V~ 202 136 none white yellow 41.9 male mascu~
## 2 Owen La~ 178 120 brown, gr~ light blue 52 male mascu~
## 3 Biggs D~ 183 84 black light brown 24 male mascu~
## 4 Anakin ~ 188 84 blond fair blue 41.9 male mascu~
## 5 Chewbac~ 228 112 brown unknown blue 200 male mascu~
## 6 Jabba D~ 175 1358 <NA> green-tan~ orange 600 herm~ mascu~
## 7 Jek Ton~ 180 110 brown fair blue NA <NA> <NA>
## 8 IG-88 200 140 none metal red 15 none mascu~
## 9 Bossk 190 113 none green red 53 male mascu~
## 10 Ackbar 180 83 none brown mot~ orange 41 male mascu~
## # i 11 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## # vehicles <list>, starships <list>
```

Podemos filtrar colunas categóricas. O exemplo abaixo retorna uma tabela apenas com os personagens com cabelo preto ou castanho.

```
# Filtrar texto com correspondência exata
filter(sw, hair_color == "black" | hair_color == "brown")
```

```
## # A tibble: 31 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Leia Or~ 150 49 brown light brown 19 fema~ femin~
## 2 Beru Wh~ 165 75 brown light blue 47 fema~ femin~
## 3 Biggs D~ 183 84 black light brown 24 male mascu~
```

```
## 4 Chewbac~ 228 112 brown unknown blue 200 male mascu~
## 5 Han Solo 180 80 brown fair brown 29 male mascu~
## 6 Wedge A~ 170 77 brown fair hazel 21 male mascu~
## 7 Jek Ton~ 180 110 brown fair blue NA <NA> <NA>
## 8 Boba Fe~ 183 78.2 black fair brown 31.5 male mascu~
## 9 Lando C~ 177 79 black dark brown 31 male mascu~
## 10 Arvel C~ NA NA brown fair brown NA male mascu~
## # i 21 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## # vehicles <list>, starships <list>
```

```
# Filtrar texto com correspondência exata
sw %>% filter(hair_color %in% c("black", "brown"))
```

```
## # A tibble: 31 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Leia Or~ 150 49 brown light brown 19 fema~ femin~
## 2 Beru Wh~ 165 75 brown light blue 47 fema~ femin~
## 3 Biggs D~ 183 84 black light brown 24 male mascu~
## 4 Chewbac~ 228 112 brown unknown blue 200 male mascu~
## 5 Han Solo 180 80 brown fair brown 29 male mascu~
## 6 Wedge A~ 170 77 brown fair hazel 21 male mascu~
## 7 Jek Ton~ 180 110 brown fair blue NA <NA> <NA>
## 8 Boba Fe~ 183 78.2 black fair brown 31.5 male mascu~
## 9 Lando C~ 177 79 black dark brown 31 male mascu~
## 10 Arvel C~ NA NA brown fair brown NA male mascu~
## # i 21 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## # vehicles <list>, starships <list>
```

Para filtrar textos sem correspondência exata, podemos utilizar a função auxiliar `str_detect()` do pacote `{stringr}`. Ela serve para verificar se cada string de um vetor contém um determinado padrão de texto.

```
library(stringr)

str_detect(
  string = c("a", "aa", "abc", "bc", "A", NA),
  pattern = "a"
)
```

```
## [1] TRUE TRUE TRUE FALSE FALSE NA
```

Podemos utilizá-la para filtrar apenas os personagens com cabelo grey.

```
# Podemos detectar se o cabelo grey aparece na string
str_detect(
  string = sw$hair_color,
  pattern = "grey"
)
```

```
## [1] FALSE    NA    NA FALSE FALSE TRUE FALSE    NA FALSE FALSE FALSE TRUE
## [13] FALSE FALSE    NA    NA FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [85] FALSE FALSE FALSE
```

```
library(stringr)
sw %>% filter(str_detect(hair_color, "grey"))
```

```
## # A tibble: 3 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Owen Lars    178    120 brown, gr~ light      blue          52 male  mascu~
## 2 Wilhuff ~    180     NA auburn, g~ fair      blue          64 male  mascu~
## 3 Palpatine    170     75 grey      pale      yellow        82 male  mascu~
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

15.2.6 Exercícios

Utilize a base `sw` nos exercícios a seguir.

1. Crie um objeto chamado `humanos` apenas com personagens que sejam humanos.
2. Crie um objeto chamado `altos_fortes` com personagens que tenham mais de 200 cm de altura e peso maior que 100 kg.
3. Retorne tabelas (`tibbles`) apenas com:
 - a. Personagens humanos que nasceram antes de 100 anos antes da batalha de Yavin (`birth_year < 100`).
 - b. Personagens com cor `light` ou `red`.
 - c. Personagens com massa maior que 100 kg, ordenados de forma decrescente por altura, mostrando apenas as colunas `name`, `mass` e `height`.

- d. Personagens que sejam “Humano” ou “Droid”, e tenham uma altura maior que 170 cm.
- e. Personagens que não possuem informação tanto de altura (`height`) quanto de massa (`mass`), ou seja, possuem NA em ambas as colunas.

15.2.7 Modificar e criar novas colunas

A função `mutate()` permite criar novas colunas ou modificar colunas existentes, facilitando transformações de dados. O código abaixo divide os valores da coluna `height` por 100, mudando a unidade de medida dessa variável de centímetros para metros.

```
# Modificar a unidade de medida da coluna height
sw %>% mutate(height = height/100)
```

```
## # A tibble: 87 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <dbl> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Luke Sk~    1.72    77 blond      fair        blue         19  male  mascu~
## 2 C-3PO      1.67    75 <NA>      gold        yellow       112  none  mascu~
## 3 R2-D2       0.96    32 <NA>      white, bl~  red         33  none  mascu~
## 4 Darth V~    2.02   136 none      white       yellow      41.9  male  mascu~
## 5 Leia Or~    1.5     49 brown     light       brown        19  fema~  femin~
## 6 Owen La~    1.78   120 brown, gr~ light       blue         52  male  mascu~
## 7 Beru Wh~    1.65    75 brown     light       blue         47  fema~  femin~
## 8 R5-D4       0.97    32 <NA>      white, red  red          NA  none  mascu~
## 9 Biggs D~    1.83    84 black     light       brown        24  male  mascu~
## 10 Obi-Wan~   1.82    77 auburn, w~ fair        blue-gray    57  male  mascu~
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Também poderíamos ter criado essa variável em uma nova coluna. Repare que a nova coluna `height_meters` é colocada no final da tabela.

```
sw %>% mutate(height_meters = height/100)
```

```
## # A tibble: 87 x 15
##   name      height mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Luke Sk~    172    77 blond      fair        blue         19  male  mascu~
## 2 C-3PO      167    75 <NA>      gold        yellow       112  none  mascu~
## 3 R2-D2       96    32 <NA>      white, bl~  red         33  none  mascu~
```

```
## 4 Darth V~ 202 136 none white yellow 41.9 male mascu~
## 5 Leia Or~ 150 49 brown light brown 19 fema~ femin~
## 6 Owen La~ 178 120 brown, gr~ light blue 52 male mascu~
## 7 Beru Wh~ 165 75 brown light blue 47 fema~ femin~
## 8 R5-D4 97 32 <NA> white, red red NA none mascu~
## 9 Biggs D~ 183 84 black light brown 24 male mascu~
## 10 Obi-Wan~ 182 77 auburn, w~ fair blue-gray 57 male mascu~
## # i 77 more rows
## # i 6 more variables: homeworld <chr>, species <chr>, films <list>,
## # vehicles <list>, starships <list>, height_meters <dbl>
```

Podemos fazer qualquer operação com uma ou mais colunas. Abaixo vamos criar um tibble que contenha as colunas `name`, `height`, `mass`, e uma nova coluna `BMI`, que calcule o Índice de Massa Corporal (IMC) de cada personagem, usando a fórmula $\text{mass} / (\text{height}/100)^2$. Caso `height` ou `mass` seja `NA`, a coluna `BMI` deve ser `NA`.

```
sw %>%
  mutate(BMI = ifelse(!is.na(height) & !is.na(mass), mass / (height/100)^2, NA)) %>%
  select(name, height, mass, BMI)
```

```
## # A tibble: 87 x 4
##   name          height mass BMI
##   <chr>         <int> <dbl> <dbl>
## 1 Luke Skywalker 172 77 26.0
## 2 C-3PO         167 75 26.9
## 3 R2-D2         96 32 34.7
## 4 Darth Vader   202 136 33.3
## 5 Leia Organa   150 49 21.8
## 6 Owen Lars     178 120 37.9
## 7 Beru Whitesun Lars 165 75 27.5
## 8 R5-D4         97 32 34.0
## 9 Biggs Darklighter 183 84 25.1
## 10 Obi-Wan Kenobi 182 77 23.2
## # i 77 more rows
```

15.2.8 Exercícios

1. Crie uma coluna chamada `dif_peso_altura` (diferença entre altura e peso) e salve a nova tabela em um objeto chamado `starwars_dif`. Em seguida, filtre apenas os personagens que têm altura maior que o peso e ordene a tabela por ordem crescente de `dif_peso_altura`.

2. Crie as seguintes colunas:

- a. `indice_massa_altura = mass / height`
 - b. `indice_massa_medio = mean(mass, na.rm = TRUE)`
 - c. `indice_relativo = (indice_massa_altura - indice_massa_medio) / indice_massa_medio`
 - d. `acima_media = ifelse(indice_massa_altura > indice_massa_medio, "sim", "não")`
3. Crie uma nova coluna que classifique o personagem em “recente” (nascido após 100 anos antes da batalha de Yavin) e “antigo” (nascido há 100 anos ou mais).

15.2.9 Sumarizar a Base de Dados

A função `summarize()` permite calcular estatísticas agregadas, como média, mediana, variância, etc. Para agregar dados por categorias, combinamos `summarize()` com `group_by()`.

O código abaixo resume a coluna `mass` pela sua média.

```
# Exemplo de sumarização
sw %>% summarize(media_massa = mean(mass, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   media_massa
##         <dbl>
## 1         97.3
```

```
# Sumarização agrupada por categorias
sw %>%
  group_by(species) %>%
  summarize(media_massa = mean(mass, na.rm = TRUE))
```

```
## # A tibble: 38 x 2
##   species  media_massa
##   <chr>      <dbl>
## 1 Aleena      15
## 2 Besalisk  102
## 3 Cerean     82
## 4 Chagrian   NaN
## 5 Clawdite   55
## 6 Droid     69.8
## 7 Dug        40
## 8 Ewok       20
```



```
## 9 Geonosian      80
## 10 Gungan        74
## # i 28 more rows
```

Podemos calcular ao mesmo tempo sumarizações diferentes.

```
sw %>% summarize(
  media_massa = mean(mass, na.rm = TRUE),
  mediana_massa = median(mass, na.rm = TRUE),
  variancia_massa = var(mass, na.rm = TRUE)
)
```

```
## # A tibble: 1 x 3
##   media_massa mediana_massa variancia_massa
##   <dbl>         <dbl>         <dbl>
## 1      97.3           79           28716.
```

Podemos também sumarizar diversas colunas.

```
sw %>% summarize(
  media_massa = mean(mass, na.rm = TRUE),
  media_altura = mean(height, na.rm = TRUE),
  media_ano = mean(birth_year, na.rm = TRUE)
)
```

```
## # A tibble: 1 x 3
##   media_massa media_altura media_ano
##   <dbl>         <dbl>         <dbl>
## 1      97.3          175.           87.6
```

Para sumarizar uma coluna agrupada pelas categorias de uma segunda coluna usamos além do `summarize()` a função `group_by()`.

O código a abaixo calcula a altura média dos personagens para cada categoria da coluna `hair_color`.

```
sw %>%
  filter(!is.na(hair_color), !is.na(height)) %>%
  group_by(hair_color) %>%
  summarize(media_altura = mean(height, na.rm = TRUE))
```

```
## # A tibble: 11 x 2
##   hair_color      media_altura
##   <chr>          <dbl>
```

##	1 auburn	150
##	2 auburn, grey	180
##	3 auburn, white	182
##	4 black	174.
##	5 blond	177.
##	6 blonde	168
##	7 brown	177.
##	8 brown, grey	178
##	9 grey	170
##	10 none	181.
##	11 white	156

15.2.10 Exercícios

Utilize a base `sw` nos exercícios a seguir.

1. Calcule a altura média e mediana dos personagens.
2. Calcule a massa média dos personagens cuja altura é maior que 175 cm.
3. Apresente na mesma tabela a massa média dos personagens com altura menor que 175 cm e a massa média dos personagens com altura maior ou igual a 175 cm.
4. Retorne tabelas (`tibbles`) apenas com:
 - a. A altura média dos personagens por espécie.
 - b. A massa média e mediana dos personagens por espécie.
 - c. Apenas o nome dos personagens que participaram de mais de 2 filmes.

15.2.11 Juntando Tabelas em R com `dplyr`

Em ciência de dados, é comum precisar combinar informações de diferentes fontes em uma única tabela. No R, o pacote `dplyr` oferece várias funções para realizar operações de junção, que permitem combinar duas ou mais tabelas (data frames ou tibbles) com base em uma ou mais colunas compartilhadas.

Tipos de Junção

As junções mais comuns em manipulação de dados são:

- `left_join()`: Mantém todas as linhas da tabela à esquerda e adiciona as colunas da tabela à direita onde há correspondência de valores na coluna chave.
- `right_join()`: Mantém todas as linhas da tabela à direita e adiciona as colunas da tabela à esquerda onde há correspondência.
- `full_join()`: Mantém todas as linhas de ambas as tabelas e adiciona NA onde não há correspondência.

Essas funções são extremamente úteis quando é necessário combinar dados de diferentes fontes que compartilham uma chave comum, como um identificador único.

Exemplo: Personagens de Star Wars

Para ilustrar como estas funções funcionam, vamos usar a base de dados `sw` e criar dois subconjuntos de dados:

1. `personagens_altos`: Tabela contendo apenas personagens com altura superior a 180 cm.
2. `personagens_humanos`: Tabela contendo apenas personagens que são humanos.

Vamos criar os nossos subconjuntos de dados usando a função `filter()` do `dplyr`.

```
# Criar subconjunto de personagens altos (altura > 180 cm)
personagens_altos <- sw %>%
  filter(height > 180) %>%
  select(name, height)

# Criar subconjunto de personagens humanos
personagens_humanos <- sw %>%
  filter(species == "Human") %>%
  select(name, species)
```

Agora que temos duas tabelas, `personagens_altos` e `personagens_humanos`, podemos usar as funções de junção para combiná-las: `left_join()`, `right_join()`, e `full_join()`.

A função `left_join()` junta duas tabelas, mantendo todas as linhas da tabela à esquerda (primeira tabela) e adicionando colunas da tabela à direita (segunda tabela) para as quais existe uma correspondência. Valores sem correspondência entre as bases receberão NA na nova base.

```
# Combinar tabelas mantendo todas as linhas de personagens_altos
humanos_altos_left_join <- left_join(personagens_altos, personagens_humanos, by = "name")

# Visualizar o resultado
print(humanos_altos_left_join)
```

```
## # A tibble: 39 x 3
##   name          height species
##   <chr>         <int> <chr>
## 1 Darth Vader     202 Human
```

```
## 2 Biggs Darklighter      183 Human
## 3 Obi-Wan Kenobi         182 Human
## 4 Anakin Skywalker       188 Human
## 5 Chewbacca              228 <NA>
## 6 Boba Fett              183 Human
## 7 IG-88                  200 <NA>
## 8 Bossk                  190 <NA>
## 9 Qui-Gon Jinn           193 Human
## 10 Nute Gunray           191 <NA>
## # i 29 more rows
```

Neste exemplo, a tabela resultante `humanos_altos_left_join` inclui todos os personagens altos e adiciona informações sobre a espécie (se disponível) da tabela `personagens_humanos`. Se um personagem não for humano, as colunas de espécie serão preenchidas com NA.

A função `right_join()` faz o oposto de `left_join()`: mantém todas as linhas da tabela à direita e adiciona colunas da tabela à esquerda para as quais existe uma correspondência.

```
# Combinar tabelas mantendo todas as linhas de personagens_humanos
humanos_altos_right_join <- right_join(personagens_altos, personagens_humanos, by = "name")

# Visualizar o resultado
print(humanos_altos_right_join)
```

```
## # A tibble: 35 x 3
##   name          height species
##   <chr>         <int> <chr>
## 1 Darth Vader      202 Human
## 2 Biggs Darklighter 183 Human
## 3 Obi-Wan Kenobi   182 Human
## 4 Anakin Skywalker 188 Human
## 5 Boba Fett        183 Human
## 6 Qui-Gon Jinn     193 Human
## 7 Padmé Amidala    185 Human
## 8 Ric Olié         183 Human
## 9 Quarsh Panaka    183 Human
## 10 Mace Windu       188 Human
## # i 25 more rows
```

A tabela `humanos_altos_right_join` inclui todos os personagens humanos e adiciona informações sobre a altura (se disponível) da tabela `personagens_altos`. Se um personagem humano não for alto, a coluna de altura será preenchida com NA.

A função `full_join()` mantém todas as linhas de ambas as tabelas e adiciona NA onde não há correspondência.

```
# Combinar todas as informações de personagens altos e humanos
humanos_altos_full_join <- full_join(personagens_altos, personagens_humanos, by = "name")

# Visualizar o resultado
print(humanos_altos_full_join)
```

```
## # A tibble: 59 x 3
##   name          height species
##   <chr>         <int> <chr>
## 1 Darth Vader     202 Human
## 2 Biggs Darklighter 183 Human
## 3 Obi-Wan Kenobi   182 Human
## 4 Anakin Skywalker 188 Human
## 5 Chewbacca       228 <NA>
## 6 Boba Fett       183 Human
## 7 IG-88           200 <NA>
## 8 Bossk           190 <NA>
## 9 Qui-Gon Jinn     193 Human
## 10 Nute Gunray     191 <NA>
## # i 49 more rows
```

A tabela `humanos_altos_full_join` contém todos os personagens altos e humanos. Se um personagem está em apenas uma das tabelas, as colunas da outra tabela são preenchidas com NA.

15.2.12 Exercícios

1. Crie uma tabela `personagens_altos` contendo apenas personagens com altura superior a 180 cm e uma outra tabela `personagens_leves` contendo apenas personagens com massa inferior a 75 kg. Use `left_join()` para combinar as duas tabelas com base no nome do personagem.

```
# Criação dos subconjuntos
personagens_altos <- sw %>%
  filter(height > 180) %>%
  select(name, height)

personagens_leves <- sw %>%
  filter(mass < 75) %>%
  select(name, mass)
```

```
# Combinação com left_join
left_join(personagens_altos, personagens_leves, by = "name")
```

Dica: Observe como `left_join()` mantém todos os personagens altos e adiciona informações sobre massa apenas para aqueles que também são leves.

2. Crie uma tabela `personagens_humanos` contendo apenas personagens humanos e uma outra tabela `cor_cabelo` contendo apenas personagens com cor de cabelo diferente de NA. Use `right_join()` para combinar `personagens_humanos` e `cor_cabelo` com base no nome do personagem.

```
# Criação dos subconjuntos
personagens_humanos <- sw %>%
  filter(species == "Human") %>%
  select(name, species)

cor_cabelo <- sw %>%
  filter(!is.na(hair_color)) %>%
  select(name, hair_color,)

# Combinação com right_join
right_join(personagens_humanos, cor_cabelo, by = "name")
```

Dica: Observe como `right_join()` mantém todos os personagens com cor de cabelo conhecida e adiciona informações sobre a espécie apenas para aqueles que também são humanos.

3. Crie uma tabela `especies_personagens` contendo apenas personagens de espécies conhecidas (não NA) e uma outra tabela `cor_olhos` contendo apenas personagens com cor de olho conhecida (não NA). Use `full_join()` para combinar as duas tabelas com base no nome do personagem.

```
# Criação dos subconjuntos
especies_personagens <- sw %>%
  filter(!is.na(species)) %>%
  select(name, species)

cor_olhos <- sw %>%
  filter(!is.na(eye_color)) %>%
  select(name, eye_color)

# Combinação com full_join
full_join(especies_personagens, cor_olhos, by = "name")
```

Dica: `full_join()` combina todas as informações disponíveis, preenchendo com NA onde não há correspondências.

Chapter 16

O pacote ggplot2

O `ggplot2` é um dos pacotes mais populares do R para criar gráficos. Ele implementa o conceito de *Grammar of Graphics*, que oferece uma maneira sistemática de descrever e construir gráficos. Este conceito está apresentado no livro *The Grammar of graphics*.

Chapter 17

Simulação

A simulação é uma ferramenta poderosa que utiliza a capacidade dos computadores modernos para realizar cálculos que, de outra forma, seriam difíceis ou até impossíveis de serem resolvidos analiticamente. A Lei dos Grandes Números assegura-nos que, ao observarmos uma grande amostra de variáveis aleatórias independentes e identicamente distribuídas (i.i.d.) com média finita, a média dessas observações tende a convergir para a média verdadeira da distribuição à medida que o tamanho da amostra aumenta.

Em vez de nos esforçarmos para encontrar essa média através de métodos analíticos complexos, podemos utilizar o poder computacional para gerar uma amostra suficientemente grande dessas variáveis aleatórias. A partir dessa amostra, calculamos a média observada, que serve como uma estimativa consistente da média verdadeira da distribuição. No entanto, a eficácia desse método depende de três fatores cruciais:

1. **Identificação correta dos tipos de variáveis aleatórias necessárias para o problema em questão:** É essencial determinar quais distribuições de probabilidade descrevem adequadamente os processos aleatórios que estamos a modelar.
2. **Capacidade do computador de gerar essas variáveis de forma precisa:** Os algoritmos utilizados para gerar números pseudoaleatórios devem ser robustos e capazes de produzir amostras que representem fielmente as distribuições desejadas.
3. **Determinação do tamanho adequado da amostra:** O tamanho da amostra deve ser suficientemente grande para que a média da amostra seja uma boa aproximação da média real. Além disso, um tamanho de amostra maior geralmente reduz a variabilidade das estimativas, aumentando a fiabilidade dos resultados.

A simulação, portanto, simplifica o processo de resolução de problemas

complexos, proporcionando uma abordagem prática e eficiente para explorar cenários onde o cálculo analítico é impraticável ou impossível. Ela permite aos estatísticos e cientistas de dados testar hipóteses, realizar análises de sensibilidade e prever resultados em condições controladas de incerteza.

Para ilustrar o potencial da simulação, começamos com alguns exemplos básicos onde a solução analítica já é conhecida. Esses exemplos demonstram que a simulação pode reproduzir resultados teóricos com precisão. Além disso, estes casos introdutórios servem para destacar aspectos importantes que precisam ser considerados ao resolver problemas mais complexos por meio de simulação, como o tratamento de variáveis dependentes, correlações entre variáveis, e questões de eficiência computacional.

Exemplo 1 (Estimando a Média de uma Distribuição Uniforme): A média da distribuição uniforme no intervalo $[0, 1]$ é conhecida por ser $\frac{1}{2}$. Se tivermos um grande número de variáveis aleatórias independentes e identicamente distribuídas (i.i.d.) a partir dessa distribuição uniforme, denotadas por X_1, X_2, \dots, X_n , a Lei dos Grandes Números diz-nos que a média amostral, dada por $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$, deve aproximar-se da média verdadeira de 0.5 à medida que o tamanho da amostra n aumenta.

A tabela abaixo apresenta as médias de diferentes amostras simuladas de tamanho n geradas a partir da distribuição uniforme $[0, 1]$ para vários valores de n . Observamos que as médias amostrais estão geralmente próximas de 0.5, mas existe uma considerável variação, especialmente quando $n = 100$. Nota-se que há menos variação para $n = 1.000$, e a variação diminui ainda mais para os dois maiores valores de n (10.000 e 100.000).

n	Replicações da Simulação				
100	0.485	0.481	0.484	0.569	0.441
1.000	0.497	0.506	0.480	0.498	0.499
10.000	0.502	0.501	0.499	0.498	0.498
100.000	0.502	0.499	0.500	0.498	0.499

Para gerar uma amostra aleatória uniforme no intervalo $[0, 1]$, utilizamos a função `runif` no R:

```
runif(100, 0, 1)
```

Como discutido anteriormente, neste exemplo não há necessidade real de simulação, pois conhecemos a média da distribuição uniforme analiticamente. Este exemplo serve para ilustrar que a simulação pode replicar resultados teóricos. No entanto, é crucial entender que, independentemente do tamanho da amostra gerada, a média amostral de variáveis aleatórias i.i.d. não será necessariamente exatamente igual à média verdadeira da distribuição. A simulação introduz uma

variabilidade que precisa ser considerada; à medida que o tamanho da amostra aumenta, a variabilidade da média amostral diminui, aproximando-a da média real. Esta propriedade sublinha a importância de escolher um tamanho de amostra apropriado para obter estimativas confiáveis.

Exemplo de Aplicação da Simulação

A seguir, exploramos um exemplo onde as questões fundamentais são relativamente fáceis de descrever, mas a solução analítica seria, na melhor das hipóteses, complexa e trabalhosa.

Problema: Esperando por uma Pausa. Dois atendentes, A e B, num restaurante fast-food começam a servir clientes ao mesmo tempo. Eles concordam em encontrar-se para uma pausa assim que ambos tiverem atendido 10 clientes. No entanto, um deles provavelmente terminará antes do outro e terá de esperar. O nosso objetivo é determinar, em média, quanto tempo um dos atendentes terá de esperar pelo outro.

Para modelar este problema, assumimos que os tempos de serviço de cada cliente, independentemente do atendente, são variáveis aleatórias independentes e identicamente distribuídas (i.i.d.) com uma distribuição exponencial de taxa $\lambda = 0.3$ clientes por minuto. Portanto, o tempo total que um atendente leva para atender 10 clientes segue uma distribuição gama com parâmetros de forma $k = 10$ e taxa $\lambda = 0.3$.

Seja X o tempo que o atendente A leva para atender 10 clientes e Y o tempo que o atendente B leva para atender 10 clientes. Precisamos de calcular a média do valor absoluto da diferença de tempo entre eles, $E(|X - Y|)$.

A solução analítica para este problema exigiria a resolução de uma integral dupla sobre a união de duas regiões não retangulares. Em vez disso, a simulação oferece uma abordagem mais prática e eficiente.

Solução Usando Simulação

Suponhamos que podemos gerar um grande número de variáveis aleatórias gama independentes. Podemos, então, simular um par (X, Y) e calcular $Z = |X - Y|$. Repetindo este processo independentemente várias vezes, podemos calcular a média de todos os valores observados de Z , que servirá como uma estimativa da média de $|X - Y|$.

Vamos utilizar o R para realizar a simulação:

```
# Definir a semente para reprodutibilidade
set.seed(123)

# Gerar amostras da distribuição gama para X e Y
x <- rgamma(10000, shape = 10, rate = 0.3)
y <- rgamma(10000, shape = 10, rate = 0.3)
```

```
# Calcular a diferença absoluta entre X e Y
z <- abs(x-y)

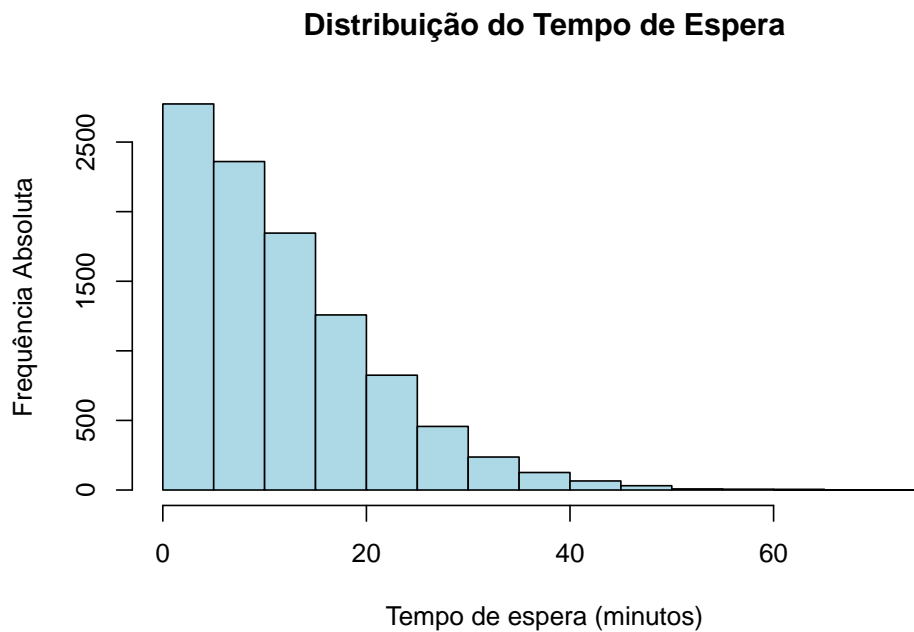
# Calcular a média de Z
mean(z)
```

```
## [1] 11.75882
```

A saída do código acima dá-nos um tempo médio de espera de aproximadamente 11.75 minutos.

Para compreender melhor a distribuição do tempo de espera, podemos visualizar a distribuição dos valores de Z utilizando um histograma:

```
hist(z,
     main = "Distribuição do Tempo de Espera",
     xlab = "Tempo de espera (minutos)",
     ylab = "Frequência Absoluta",
     col = "lightblue",
     border = "black")
```



17.1 Simulação e Geração de Números Pseudoaleatórios

Números Aleatórios

Números aleatórios são valores gerados de forma imprevisível, sem seguir nenhum padrão determinístico. Numa sequência de números aleatórios, cada número é escolhido independentemente dos outros, e não há qualquer correlação entre eles. Na prática, números aleatórios são fundamentais em diversas áreas, como criptografia, simulações estatísticas, jogos de azar e processos de modelagem onde a imprevisibilidade é crucial.

A verdadeira aleatoriedade geralmente deriva de processos físicos inerentemente imprevisíveis, como a radiação cósmica, o ruído térmico em circuitos eletrônicos ou o decaimento radioativo. No entanto, em computação, obter números verdadeiramente aleatórios pode ser difícil e, frequentemente, desnecessário. Para muitas aplicações, uma forma de aleatoriedade que seja suficientemente imprevisível é suficiente.

Números Pseudoaleatórios

Números pseudoaleatórios, por outro lado, são gerados por algoritmos determinísticos que produzem sequências de números que parecem aleatórios, mas são, na verdade, baseados num valor inicial conhecido como semente (*seed*). Se um algoritmo de geração de números pseudoaleatórios for iniciado com a mesma semente, ele produzirá exatamente a mesma sequência de números em cada execução.

Embora os números pseudoaleatórios não sejam verdadeiramente aleatórios, são amplamente utilizados porque podem ser gerados de forma rápida e eficiente e, para muitas aplicações, são suficientemente aleatórios. A principal vantagem de usar números pseudoaleatórios é que, ao utilizar a mesma semente, podemos replicar experiências ou simulações, o que é extremamente útil para fins de investigação e depuração de códigos.

Método Congruencial Multiplicativo

Uma das abordagens mais comuns para gerar números pseudoaleatórios é o **Método Congruencial Multiplicativo**:

- **Semente Inicial:** Considere um valor inicial x_0 , chamado de semente (*seed*).
- **Cálculo Recursivo:** Para $n \geq 1$, os valores sucessivos x_n são calculados recursivamente usando a fórmula:

$$x_n = (a \cdot x_{n-1}) \bmod m,$$

onde a e m são inteiros positivos dados. Aqui, x_n é o resto da divisão inteira de $a \cdot x_{n-1}$ por m .

- **Números Pseudoaleatórios:** A quantidade $\frac{x_n}{m}$ é considerada um número pseudoaleatório, uma aproximação de uma variável aleatória uniforme no intervalo $[0, 1]$.

Critérios para Escolha de a e m

Para que os números gerados sejam úteis em simulações, as constantes a e m devem satisfazer os seguintes critérios:

1. **Aparência de Aleatoriedade:** A sequência gerada deve ter a “aparência” de uma sequência de variáveis aleatórias uniformemente distribuídas no intervalo $(0,1)$.
2. **Período Longo:** A sequência deve ter um período longo antes de começar a se repetir, o que significa que deve gerar um grande número de variáveis antes que ocorra uma repetição.
3. **Eficiência Computacional:** Os valores devem ser computados de forma eficiente para que o algoritmo seja utilizável em aplicações práticas.

17.2 A função `set.seed()`

A função `set.seed()` no R é utilizada para garantir a reprodutibilidade dos resultados ao gerar números aleatórios. Sempre que utilizamos funções como `sample()` ou `runif()`, o R gera uma sequência de números pseudoaleatórios. Ao especificar um valor na função `set.seed()`, conseguimos que a mesma sequência seja gerada sempre que o código for executado.

```
set.seed(123)
sample(1:100, 5)
```

```
## [1] 31 79 51 14 67
```

Com isso, você sempre obterá os mesmos 5 números aleatórios ao usar esse código.

17.3 A função `sample()`

A função `sample()` em R é amplamente utilizada para gerar amostras aleatórias a partir de um conjunto de dados ou de uma sequência de números.

É uma função muito flexível que permite definir o tamanho da amostra, se a amostragem é feita com ou sem reposição, e se os elementos têm diferentes probabilidades de serem selecionados.

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

- `x`: O vetor de elementos a serem amostrados.
- `size`: O tamanho da amostra desejada.
- `replace`: Um valor lógico que indica se a amostragem é feita com reposição (`TRUE`) ou sem reposição (`FALSE`).
- `prob`: Um vetor de probabilidades associadas a cada elemento em `x`. Se não especificado, assume-se que todos os elementos têm a mesma probabilidade de serem selecionados.

Exemplo 1 (Amostragem Simples sem Reposição): Neste exemplo, extraímos uma amostra de 5 elementos a partir de uma população de números inteiros de 1 a 10, sem reposição:

```
# População de 1 a 10  
pop <- 1:10  
  
# Amostra de 5 elementos sem reposição  
amostra <- sample(pop, size = 5, replace = FALSE)  
print(amostra)  
## [1] 5 2 4 10 8
```

Exemplo 2 (Amostragem com Reposição): Neste exemplo, permitimos que os mesmos elementos sejam selecionados mais de uma vez:

```
# Amostra de 5 elementos com reposição  
amostra_repos <- sample(pop, size = 5, replace = TRUE)  
print(amostra_repos)  
## [1] 6 8 9 9 6
```

Exemplo 3 (Amostragem com Probabilidades Diferentes): Neste exemplo, associamos diferentes probabilidades a cada elemento da população:

```
# Probabilidades associadas a cada elemento  
prob <- c(0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.05, 0.05)  
  
# Amostra com probabilidades diferentes  
amostra_prob <- sample(pop, size = 5, prob = prob)  
print(amostra_prob)  
## [1] 2 4 6 5 3
```

17.4 Exercícios

1. Crie um vetor com os números de 1 a 20. Utilize a função `sample()` para selecionar uma amostra aleatória de 5 elementos desse vetor. A amostragem deve ser feita sem reposição.
2. Suponha que tem uma população representada pelos números de 1 a 10. Utilize a função `sample()` para selecionar uma amostra de 10 elementos com reposição.
3. Crie um vetor com as letras A, B, C, D, E. Aplique a função `sample()` para selecionar uma amostra de 3 letras, onde a probabilidade de cada letra ser selecionada é dada pelo vetor `c(0.1, 0.2, 0.3, 0.25, 0.15)`.
4. Crie um vetor com os números de 1 a 10. Utilize a função `sample()` para reordenar aleatoriamente os elementos desse vetor.
5. Crie um vetor com os nomes de cinco frutas: “Maçã”, “Banana”, “Laranja”, “Uva”, “Pera”. Utilizando a função `sample()`, selecione aleatoriamente uma fruta desse vetor. Em seguida, selecione uma amostra de 3 frutas.
6. Você é responsável por realizar um teste de qualidade em uma fábrica. Há 1000 produtos fabricados, numerados de 1 a 1000. Selecione uma amostra aleatória de 50 produtos para inspeção, garantindo que não haja reposição na seleção.
7. Simule o lançamento de dois dados justos 10000 vezes e registre as somas das faces resultantes. Utilize a função `sample()` para realizar a simulação. Em seguida, crie um histograma das somas obtidas.
8. Tem um vetor de 200 estudantes classificados em três turmas: A, B, e C. As turmas têm tamanhos diferentes (50, 100, e 50 alunos, respectivamente). Usando `sample()`, selecione uma amostra de 20 alunos, mantendo a proporção original das turmas.
9. Um cartão de Bingo contém 24 números aleatórios entre 1 e 75 (excluindo o número central “free”). Crie 5 cartões de Bingo únicos usando a função `sample()`.
10. Em um estudo clínico, 30 pacientes devem ser randomizados em dois grupos: tratamento e controle. O grupo de tratamento deve conter 20 pacientes e o grupo de controle 10. Usando `sample()`, faça a randomização dos pacientes. Dica: use a função `setdiff()`.

Chapter 18

Método da transformada inversa

18.1 Variável aleatória discreta

Suponha que queremos gerar o valor de uma variável aleatória discreta X com função massa de probabilidade $P(X = x_i) = p_i$, $i = 0, 1, \dots$, $\sum_i p_i = 1$. Para isso, basta gerar um número aleatório $U \sim U(0, 1)$ e considerar:

$$X = \begin{cases} x_0, & \text{se } U < p_0 \\ x_1, & \text{se } p_0 \leq U < p_0 + p_1 \\ \vdots & \\ x_i, & \text{se } \sum_{j=0}^{i-1} p_j \leq U < \sum_{j=0}^i p_j \\ \vdots & \end{cases}$$

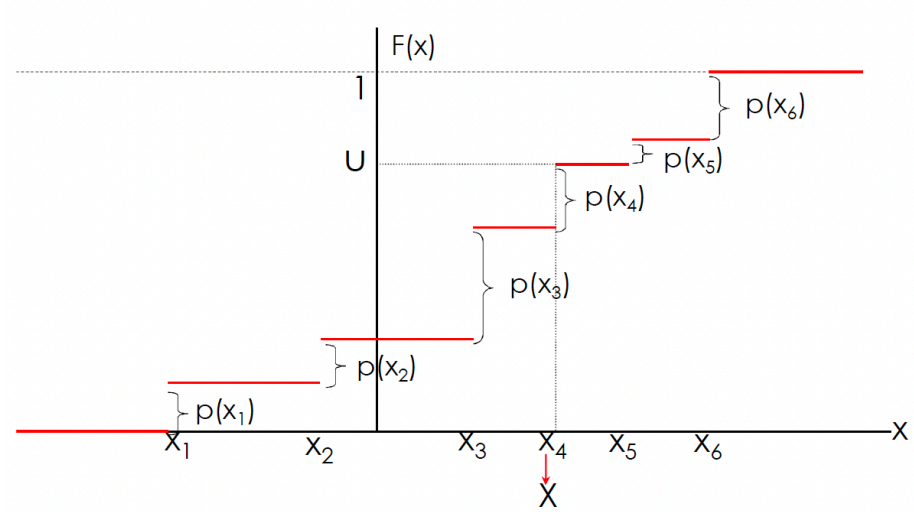
Como, para $0 < a < b < 1$, $P(a \leq U < b) = b - a$, temos que

$$P(X = x_i) = P\left(\sum_{j=0}^{i-1} p_j \leq U < \sum_{j=0}^i p_j\right) = p_i.$$

Se os x_i , $i \geq 0$, estão ordenados $x_0 < x_1 < \dots$ e se denotarmos por F a função de distribuição de X , então $F(x_k) = \sum_{i=0}^k p_i$ e assim

$$X = x_i \quad \text{se} \quad F(x_{i-1}) \leq U < F(x_i)$$

Em outras palavras, depois de gerar um número aleatório U nós determinamos o valor de X encontrando o intervalo $[F(x_{i-1}), F(x_i)]$ no qual U pertence (ou, equivalentemente, encontrando a inversa de $F(U)$).



Exemplo 1: Seja X uma variável aleatória discreta tal que $p_1 = 0.20$, $p_2 = 0.15$, $p_3 = 0.25$, $p_4 = 0.40$ onde $p_j = P(X = j)$. Gere 1000 valores dessa variável aleatória.

Para a variável aleatória X , a função de distribuição acumulada é dada pela soma cumulativa das probabilidades:

$$F(x) = \begin{cases} 0, & \text{se } x < 1 \\ p_1, & \text{se } 1 \leq x < 2 \\ p_1 + p_2, & \text{se } 2 \leq x < 3 \\ p_1 + p_2 + p_3, & \text{se } 3 \leq x < 4 \\ 1, & \text{se } x \geq 4 \end{cases}$$

Com os valores fornecidos:

$$F(x) = \begin{cases} 0, & \text{se } x < 1 \\ 0.20, & \text{se } 1 \leq x < 2 \\ 0.35, & \text{se } 2 \leq x < 3 \\ 0.60, & \text{se } 3 \leq x < 4 \\ 1, & \text{se } x \geq 4 \end{cases}$$

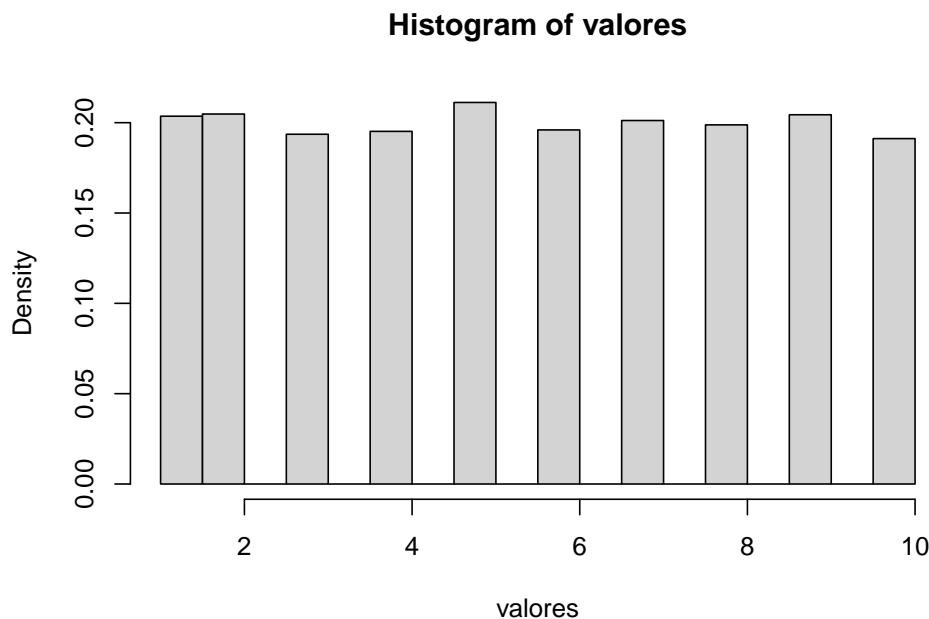
Gerar um número aleatório uniforme U no intervalo $[0,1]$. Para determinar o valor de X correspondente a U :

- Se $U < 0.20$, então $X = 1$
- Se $0.20 \leq U < 0.35$, então $X = 2$
- Se $0.35 \leq U < 0.60$, então $X = 3$

- Se $0.60 \leq U \leq 1$, então $X = 4$

Exemplo 2: Seja X uma variável aleatória discreta assumindo os valores: $1, 2, \dots, 10$ com probabilidade $1/10$ para $x = 1, 2, \dots, 10$. Gerar 5000 valores dessa variável aleatória. Representar graficamente e determinar: média, desvio padrão e mediana.

```
gerar_va_inversa <- function(){  
  # Gerar número aleatório entre 0 e 1  
  u <- runif(1,0,1)  
  p <- 1/10 # primeira probabilidade P(X=1)  
  F <- p # inicializar a função de distribuição acumulada  
  X <- 1 # inicializar o valor da va X  
  
  while(u > F){  
    X <- X+1  
    F <- F+p  
  }  
  return(X)  
}  
valores <- replicate(5000,gerar_va_inversa())  
hist(valores, freq = FALSE)
```



```
mean(valores)
```

```
## [1] 5.4786
```

```
sd(valores)
```

```
## [1] 2.868447
```

```
median(valores)
```

```
## [1] 5
```

Exemplo 3: Geração de uma variável aleatória com distribuição de Bernoulli. A variável aleatória X é de Bernoulli com parâmetro p se

$$P(X = x) = \begin{cases} 1 - p, & \text{se } x = 0 \\ p, & \text{se } x = 1 \end{cases}$$

Para gerar uma Bernoulli(p) podemos usar o seguinte algoritmo que é equivalente ao método da transformada inversa

1. Gerar um número aleatório U ;
2. Se $U \leq p$ então $X = 1$ senão $X = 0$.

```
# Gerando uma variável aleatória com distribuição de Bernoulli(p)
gerar_bernoulli_inversa <- function(p){
  U <- runif(1)
  if (U <= p){
    X <- 1
  } else {
    X <- 0
  }
  return(X)
}

valores <- replicate(100,gerar_bernoulli_inversa(0.8))
sum(valores)/100
```

```
## [1] 0.73
```

Exemplo 4: Gerar uma variável aleatória com distribuição Binomial(n, p). Aqui podemos usar o facto de que se X_1, X_2, \dots, X_n são Bernoullis i.i.d., então

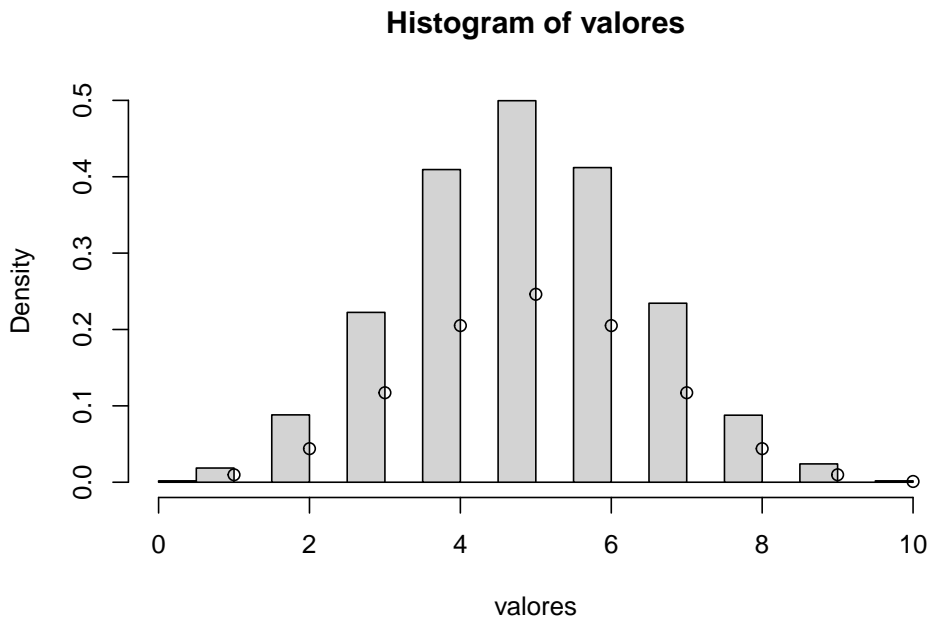
$$X = X_1 + X_2 + \dots + X_n$$

é uma Binomial(n, p).

```
# Gerando uma variável aleatória com distribuição Binomial(n,p)

gerar_binomial_inversa <- function(n,p){
  X <- sum(replicate(n, gerar_bernoulli_inversa(p)))
  return(X)
}

valores <- replicate(10000, gerar_binomial_inversa(10, 0.5))
hist(valores, freq = FALSE)
points(1:10, dbinom(1:10, 10, 0.5))
```



Exemplo 5: Geração de uma variável aleatória com distribuição Geométrica(p). Seja $X \sim Geometrica(p)$. Lembre que

$$P(X = x) = p(1 - p)^{x-1}$$

e que

$$F(x) = P(X \leq x) = \begin{cases} 0, & \text{se } x < 1 \\ 1 - (1 - p)^x, & \text{se } x \geq 1 \end{cases}$$

O seguinte algoritmo é equivalente ao método da transformada inversa:

1. Gerar um número aleatório U ;
2. Fazer $X = \lfloor \ln(U)/\ln(1-p) \rfloor$

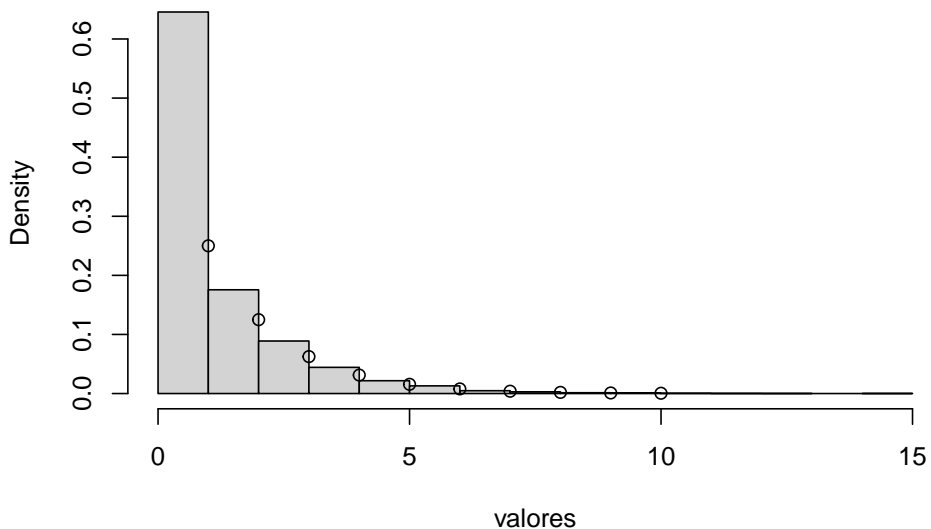
onde $\lfloor \cdot \rfloor$ = maior inteiro.

```
# Gerar uma variável aleatória com distribuição Geométrica(p)

gerar_geometrica_inversa <- function(p){
  U <- runif(1)
  X <- round(log(U)/log(1-p))
  return(X)
}

valores <- replicate(10000, gerar_geometrica_inversa(0.5))
hist(valores, freq = FALSE)
points(1:10, dgeom(1:10,0.5))
```

Histogram of valores



Exemplo 6: Geração de uma variável aleatória com distribuição de Poisson. A variável aleatória X é de Poisson com média λ se

$$p_i = P(X = i) = \frac{e^{-\lambda} \lambda^i}{i!}, \quad i = 0, 1, \dots$$

A chave para usar o método da transformada inversa para gerar uma tal variável aleatória é dada pela seguinte identidade:

$$p_{i+1} = \frac{\lambda}{i+1} p_i, \quad i \geq 0.$$

Ao utilizar a recursão acima para calcular as probabilidades de Poisson quando elas são necessárias, o algoritmo da transformada inversa para gerar uma variável aleatória de Poisson com média λ pode ser expresso da seguinte forma.

```
# Gerando uma va com distribuição de Poisson

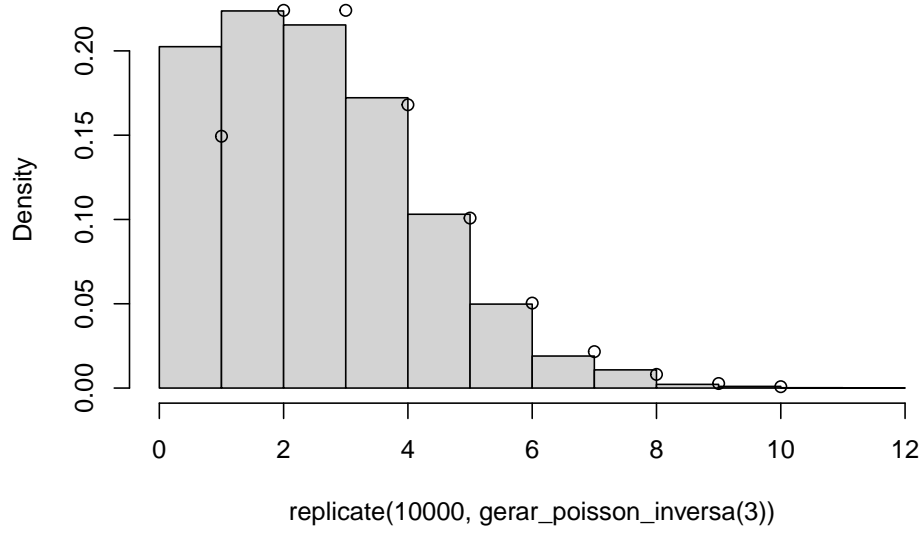
lambda <- 3 # exemplo com lambda = 3

# Função para gerar uma variável aleatória de Poisson usando o método da transformada inversa
gerar_poisson_inversa <- function(lambda) {
  U <- runif(1) # Gerar um número aleatório uniforme entre 0 e 1
  p <- exp(-lambda) # Inicializar a primeira probabilidade P(X=0)
  F <- p # Inicializar a função de distribuição acumulada (CDF)
  X <- 0 # Inicializar o valor da variável aleatória

  # Acumular probabilidades até que a CDF exceda U
  while (U > F) {
    X <- X + 1
    p <- p * lambda / X # Atualizar a probabilidade P(X=k)
    F <- F + p # Atualizar a CDF
  }

  return(X)
}

hist(replicate(10000,gerar_poisson_inversa(3)),freq = FALSE)
points(1:10,dpois(1:10,3))
```

Histogram of replicate(10000, gerar_poisson_inversa(3))

18.2 Variável aleatória contínua

Uma variável aleatória X tem densidade $f(x) = 2x$, para $0 < x < 1$, e 0, caso contrário. Suponha que queremos simular observações de X . Nesta secção, apresentaremos um método simples e flexível para simulação de uma distribuição contínua.

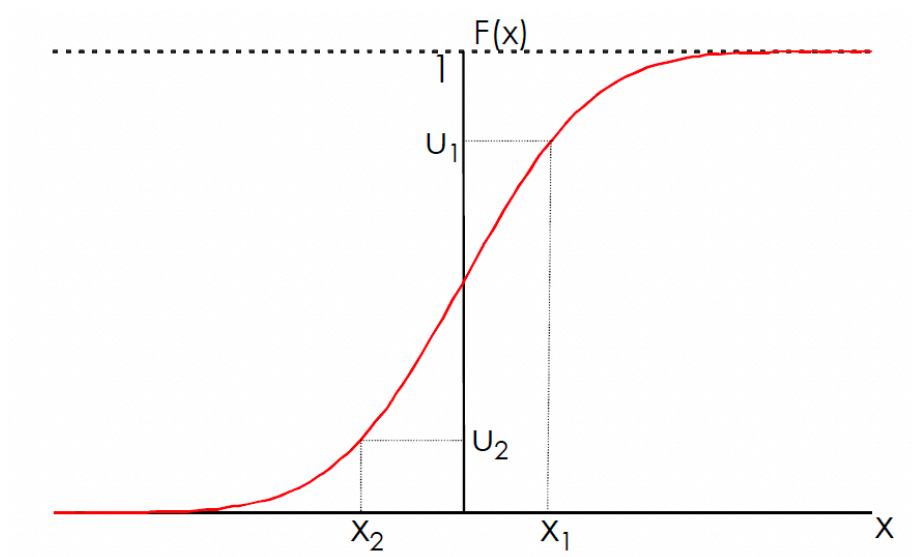
Proposição: Suponha que X é uma variável aleatória com função de distribuição F , onde F é invertível com função inversa F^{-1} . Seja U uma variável aleatória uniforme $(0, 1)$. Então a distribuição de $F^{-1}(U)$ é igual a distribuição de X , ou seja, a variável aleatória X definida por $X = F^{-1}(U)$ tem distribuição F .

A prova desta proposição é fácil e rápida. Precisamos mostrar que $F^{-1}(U)$ tem a mesma distribuição que X . Assim,

$$\begin{aligned}
 P(X \leq x) &= P(F^{-1}(U) \leq x) = P(FF^{-1}(U) \leq F(x)) \\
 &= P(U \leq F(x)) = P(0 \leq U \leq F(x)) \\
 &= F(x) - 0 \\
 &= F(x).
 \end{aligned}$$

A última igualdade decorre do facto de que $U \sim U(0, 1)$ e $0 \leq F(x) \leq 1$.

Esta proposição mostra que se pode gerar uma variável aleatória X de uma função de distribuição contínua F , gerando um número aleatório U e tomando $X = F^{-1}(U)$.

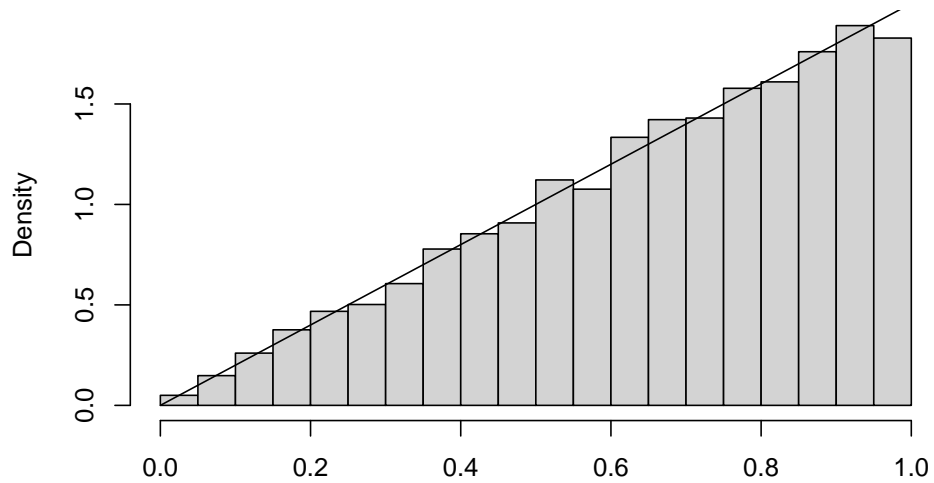


Exemplo 1: Considere nossa variável aleatória X com densidade $f(x) = 2x$. A função de distribuição de X é

$$F(x) = P(X \leq x) = \int_0^x 2t \, dt = x^2, \quad \text{para } 0 < x < 1.$$

A função $F(x) = x^2$ é invertível no intervalo $(0, 1)$ e $F^{-1}(x) = \sqrt{x}$. O método da transformada inversa diz que se $U \sim U(0, 1)$, então $F^{-1}(U) = \sqrt{U}$ tem a mesma distribuição que X . Portanto, para simular X , basta gerar \sqrt{U} .

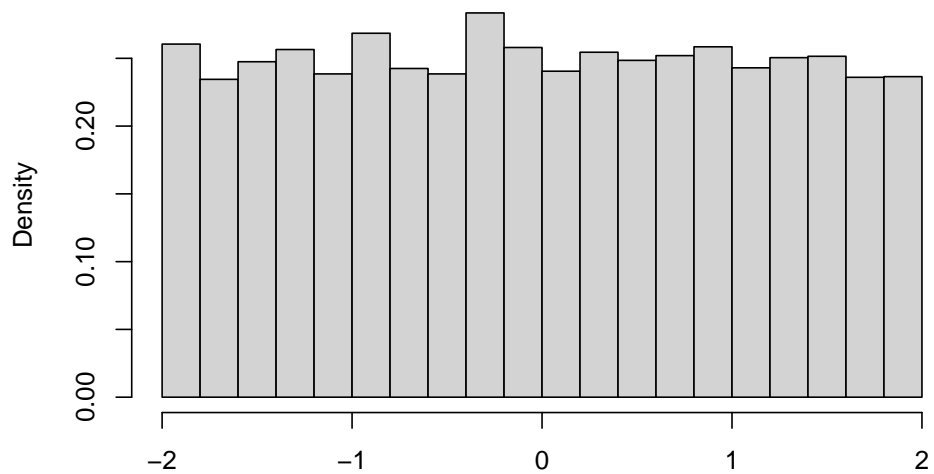
```
n <- 10000
set.seed(123)
simlist <- sqrt(runif(n))
hist(simlist, prob=T, main="", xlab="")
curve(2*x, 0,1, add=T)
```



Exemplo 2: Geração de uma variável aleatória uniforme(a,b). A geração é feita através de

$$X = a + (b - a)U.$$

```
# Geração de uma va uniforme(-2,2)
a <- -2
b <- 2
n <- 10000
set.seed(123)
simlist <- a+(b-a)*runif(n)
hist(simlist, prob=T, main="", xlab="")
```



Exemplo 3: Geração de uma variável aleatória exponencial. Seja X uma variável aleatória exponencial com taxa 1, então sua função de distribuição é

dada por

$$F(x) = 1 - e^{-x}.$$

Como $0 \leq F(x) \leq 1$, tomando $F(x) = u$, onde $u \sim U(0, 1)$ tem-se:

$$u = F(x) = 1 - e^{-x}$$

ou

$$1 - u = e^{-x}$$

ou, aplicando o logaritmo

$$x = -\ln(1 - u).$$

Daí, pode-se gerar uma exponencial com parâmetro 1 gerando um número aleatório U e em seguida fazendo

$$X = F^{-1}(U) = -\ln(1 - U).$$

Uma pequena economia de tempo pode ser obtida ao notar que $1 - U$ também é uniforme em $(0, 1)$, e assim, $-\ln(1 - U)$ tem a mesma distribuição que $-\ln(U)$. Isto é, o logaritmo negativo de um número aleatório é exponencialmente distribuído com taxa 1.

Além disso, note que se X é uma exponencial com média 1, então para qualquer constante c , cX é uma exponencial com média c . Assim, uma variável aleatória exponencial X com taxa λ (média $\frac{1}{\lambda}$) pode ser gerada através da geração de um número aleatório U e fazendo

$$X = -\frac{1}{\lambda} \ln(U).$$

```
# Definir a sequência de valores x
x <- seq(0,3, by = 0.02)

# Definir o parâmetro lambda da distribuição exponencial
lambda <- 3

# Número de simulação
n <- 10000

# Simular valores de uma distribuição exponencial
set.seed(123)
simlist <- -log(runif(n))/lambda

# Plotar o histograma da simulação com a densidade de probabilidade
hist(simlist, probability = TRUE, main = "Comparação da Distribuição Exponencial Simulada e Teórica",
      xlab = "Valores", ylab = "Densidade", col = "lightblue", border = "black")

# Adicionar a curva de densidade teórica
```

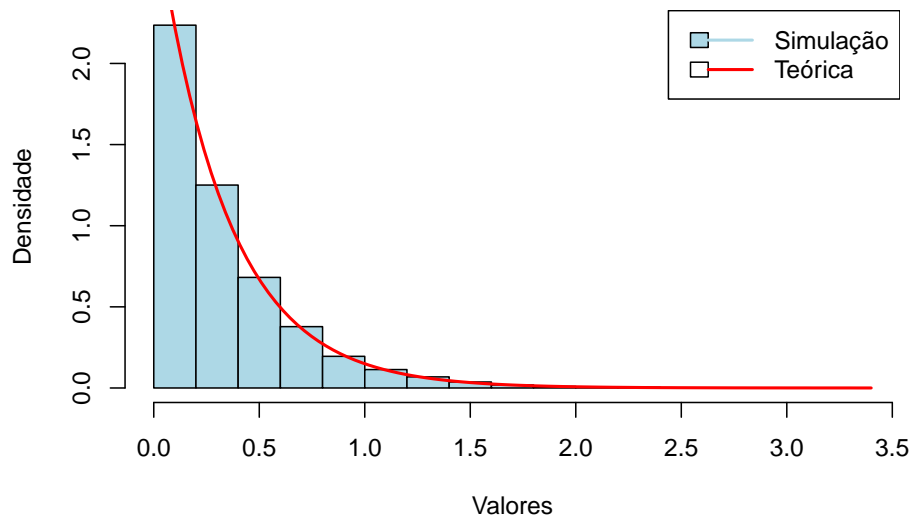
```

curve(dexp(x, rate = lambda), add = TRUE, col = "red", lwd = 2)

# Adicionar uma legenda
legend("topright", legend = c("Simulação", "Teórica"), col = c("lightblue", "red"), lwd = 2)

```

Comparação da Distribuição Exponencial Simulada e Teórica



Exercício: Seja X uma variável aleatória com distribuição $W(\alpha, \beta)$. Assim a fdp de X é

$$f(x) = \begin{cases} \alpha \beta^{-\alpha} x^{\alpha-1} e^{-(x/\beta)^\alpha}, & \text{se } x > 0 \\ 0, & \text{se } x \leq 0 \end{cases}$$

A função de distribuição de X é:

$$F(x) = \int_0^x f(u) du = \begin{cases} 1 - e^{-(x/\beta)^\alpha}, & \text{se } x > 0 \\ 0, & \text{se } x \leq 0 \end{cases}$$

Mostre que $X = \beta[-\ln(U)]^{1/\alpha}$. Gere 10000 valores de uma $W(2, 3)$. Represente graficamente a distribuição.

Chapter 19

Método da aceitação-rejeição

Chapter 20

Distribuições univariadas no R

No R temos acesso as mais comuns distribuições univariadas. Todas as funções tem as seguintes formas:

Função	Descrição
p nome(...)	função de distribuição
d nome(...)	função de probabilidade ou densidade de probabilidade
q nome(...)	inversa da função de distribuição
r nome(...)	geração de números aleatórios com a distribuição especificada

o **nome** é uma abreviatura do nome usual da distribuição (**binom**, **geom**, **pois**, **unif**, **exp**, **norm**, ...).

Exemplo 1: Simule o lançamento de três moedas honestas e a contagem do número de caras X .

(a) Use a sua simulação para estimar $P(X = 1)$ e $E(X)$.

(b) Modifique a alínea anterior para permitir uma moeda viciada onde $P(\text{cara}) = 3/4$.

```
set.seed(123)
n <- 10000
sim1 <- numeric(n)
sim2 <- numeric(n)
for (i in 1:n) {
```

```

moedas <- sample(0:1,3,replace=T)
sim1[i] <- if (sum(moedas)==1) 1 else 0
sim2[i] <- sum(moedas)
}
#  $P(X=1)$ 
mean(sim1)

```

```
## [1] 0.3821
```

```

#  $E(X)$ 
mean(sim2)

```

```
## [1] 1.4928
```

```

set.seed(123)
n <- 10000
sim1 <- numeric(n)
sim2 <- numeric(n)
for (i in 1:n) {
  moedas <- sample(c(0,1),3,prob=c(1/4,3/4),replace=T)
  sim1[i] <- if (sum(moedas)==1) 1 else 0
  sim2[i] <- sum(moedas)
}
#  $P(X=1)$ 
mean(sim1)

```

```
## [1] 0.1384
```

```

#  $E(X)$ 
mean(sim2)

```

```
## [1] 2.2503
```

Sabemos também que X – número de caras no lançamento de três moedas honestas tem distribuição $Binomial(n = 3, p = 0.5)$. Assim, podemos resolver a questão da seguinte maneira

```

set.seed(123)
valores <- rbinom(10000,3,0.5)
#  $P(X=1)$ 
sum(valores == 1)/length(valores)

```

```
## [1] 0.383
```



```
# E(X)
sum(valores)/length(valores)
```

```
## [1] 1.4897
```

```
mean(valores)
```

```
## [1] 1.4897
```

No segundo caso teremos $X \sim \text{Binomial}(n = 3, p = 3/4)$.

```
set.seed(123)
valores <- rbinom(10000, 3, 3/4)
# P(X=1)
sum(valores == 1)/length(valores)
```

```
## [1] 0.1365
```

```
# E(X)
sum(valores)/length(valores)
```

```
## [1] 2.2558
```

```
mean(valores)
```

```
## [1] 2.2558
```

Exemplo 2: O tempo até a chegada de um autocarro tem uma distribuição exponencial com média de 30 minutos.

(a) Use o comando `rexp()` para simular a probabilidade do autocarro chegar nos primeiros 20 minutos.

(b) Use o comando `pexp()` para comparar com a probabilidade exata.

```
set.seed(123)
valores <- rexp(10000, 1/30)
# Probabilidade P(X <= 20)
sum(valores < 20)/length(valores)
```

```
## [1] 0.4832
```

```
# Probabilidade exata
pexp(20, 1/30)
```

```
## [1] 0.4865829
```

Exemplo 3: As cartas são retiradas de um baralho padrão, com reposição, até que um ás apareça. Simule a média e a variância do número de cartas necessárias.

```
set.seed(123)
n <- 10000
# Denote os ases por 1,2,3,4
simlist <- numeric(n)

for (i in 1:n) {
  ct <- 0
  as <- 0
  while (as == 0) {
    carta <- sample(1:52,1,replace=T)
    ct <- ct + 1
    if (carta <= 4){
      as <- 1
    }
  }
  simlist[i] <- ct
}
mean(simlist)
```

```
## [1] 12.8081
```

```
var(simlist)
```

```
## [1] 147.5318
```

Podemos notar aqui também que X — número de provas de Bernoulli até o primeiro sucesso (aparecer um ás), que tem distribuição *Geomtrica*($p = 4/52$). Lembre que o R trabalha com a geométrica como sendo X — número de insucessos até o primeiro sucesso.

```
set.seed(123)

valores <- rgeom(10000, 4/52) + 1

# Média e variância
mean(valores)
```

```
## [1] 13.0108
```

```
var(valores)
```

```
## [1] 152.0335
```

20.1 Função de distribuição empírica

A função de distribuição empírica é uma função de distribuição acumulada que descreve a proporção ou contagem de observações em um conjunto de dados que são menores ou iguais a um determinado valor. É uma ferramenta útil para visualizar a distribuição de dados observados e comparar distribuições amostrais.

- É uma função definida para todo número real x e que para cada x dá a proporção de elementos da amostra menores ou iguais a x :

$$F_n(x) = \frac{\# \text{ observações} \leq x}{n}$$

- Para construir a função de distribuição empírica precisamos primeiramente ordenar os dados em ordem crescente: $(x_{(1)}, \dots, x_{(n)})$
- A definição da função de distribuição empírica é

$$F_n(x) = \begin{cases} 0, & x < x_{(1)} \\ \frac{i}{n}, & x_{(i)} \leq x < x_{(i+1)}, \quad i = 1, \dots, n-1 \\ 1, & x \geq x_{(n)} \end{cases}$$

- Passo a passo para a construção da função
 - Inicie desenhando a função do valor mais à esquerda para o mais à direita.
 - Atribua o valor 0 para todos os valores menores que o menor valor da amostra, $x_{(1)}$.
 - Atribua o valor $\frac{1}{n}$ para o intervalo entre $x_{(1)}$ e $x_{(2)}$, o valor $\frac{2}{n}$ para o intervalo entre $x_{(2)}$ e $x_{(3)}$, e assim por diante, até atingir todos os valores da amostra.
 - Para valores iguais ou superiores ao maior valor da amostra, $x_{(n)}$, a função tomará o valor 1.
 - Se um valor na amostra se repetir k vezes, o salto da função para esse ponto será $\frac{k}{n}$, em vez de $\frac{1}{n}$.

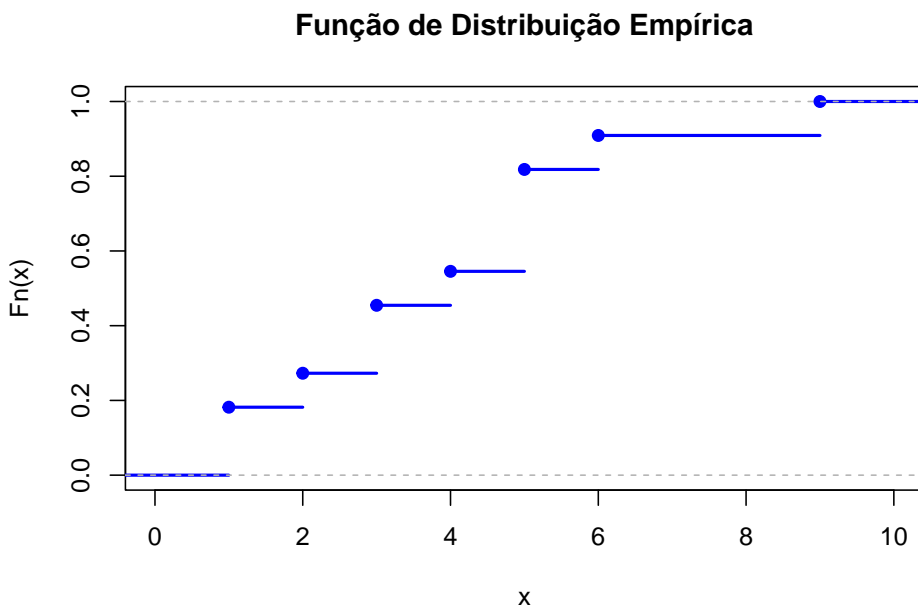
20.1.1 Função de distribuição empírica no R, função `ecdf()`

A função `ecdf()` no R é usada para calcular a função de distribuição empírica (Empirical Cumulative Distribution Function - ECDF) de um conjunto de dados.

```
# Conjunto de dados
dados <- c(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)

# Calcular a ECDF usando a função ecdf()
Fn <- ecdf(dados)

# Plotar a ECDF usando a função ecdf()
plot(Fn, main = "Função de Distribuição Empírica", xlab = "x", ylab = "Fn(x)", col = "blue")
```



Exemplo 1: Resolva o exemplo 1 usando a função de distribuição empírica.

```
valores <- rexp(10000, 1/30)
# Função de distribuição empírica
Fn <- ecdf(valores)
# Probabilidade P(X<=20)
Fn(20)
```

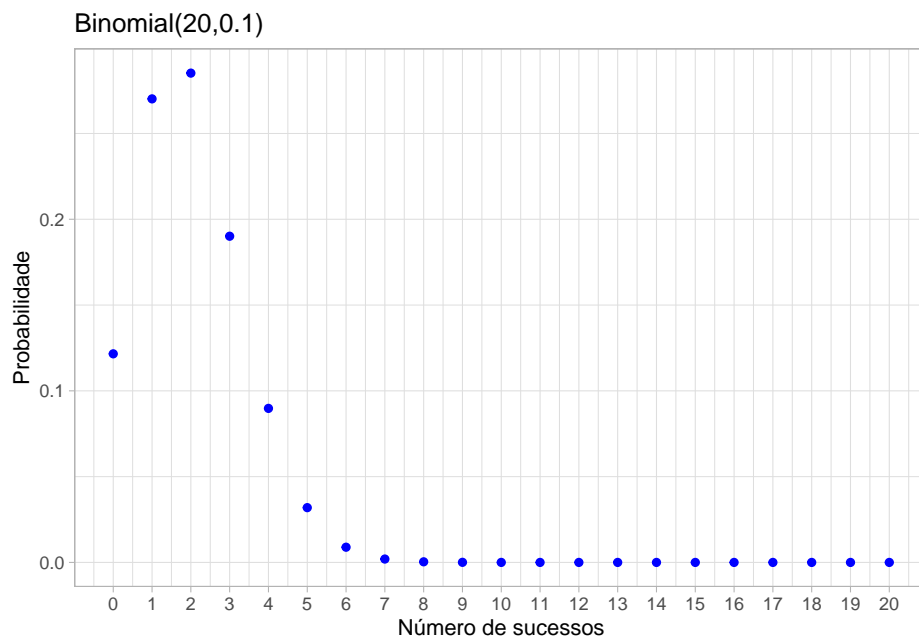
```
## [1] 0.4874
```

```
# Probabilidade exata  
pexp(20, 1/30)
```

```
## [1] 0.4865829
```

20.1.2 Função massa de probabilidade (teórica)

```
# Simulação de Variáveis aleatórias  
  
# Função massa de probabilidade Binomial(n,p)  
n <- 20  
p <- 0.1  
x <- 0:20  
  
teorico <- data.frame(x = x, y=dbinom(x, size = n, prob = p))  
  
# Carregue o pacote ggplot2  
library(ggplot2)  
  
ggplot(teorico) +  
  geom_point(aes(x = x, y=y), color = "blue") +  
  scale_x_continuous(breaks = 0:n) +  
  labs(title = "Binomial(20,0.1)", x = "Número de sucessos", y = "Probabilidade") +  
  theme_light()
```



20.1.3 Função massa de probabilidade (simulação)

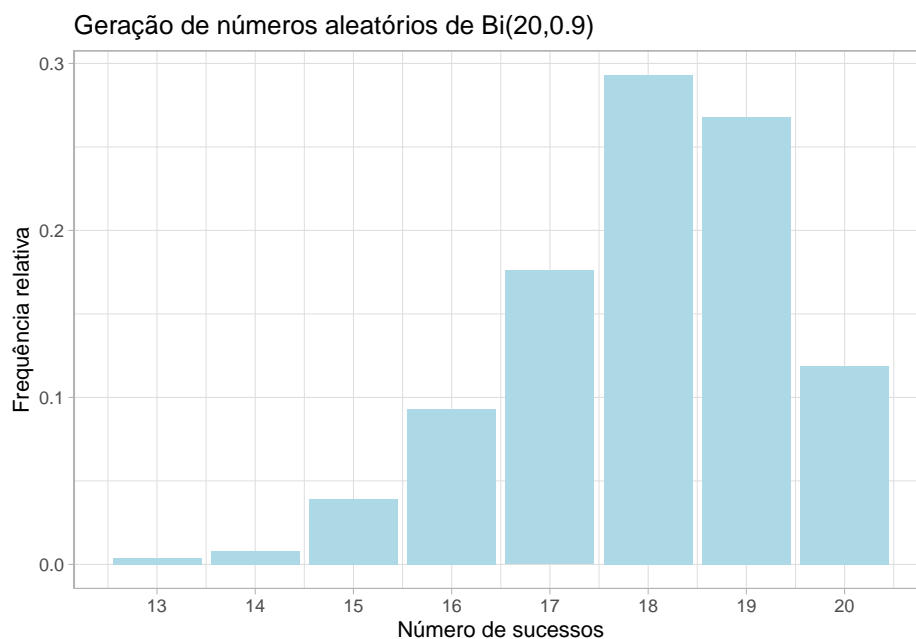
```
set.seed(1234)

n <- 20
p <- 0.9
k <- 1000 # número de simulações

dados <- data.frame(X = rbinom(k, size = n, prob = p))

# Carregue o pacote ggplot2library(ggplot2)

ggplot(dados) +
  geom_bar(aes(x=X, y=after_stat(prop)), fill = "lightblue") +
  scale_x_continuous(breaks = 0:n) +
  labs(title = "Geração de números aleatórios de Bi(20,0.9)", x="Número de sucessos",
y="Frequência relativa") +
  theme_light()
```



20.1.4 Comparação

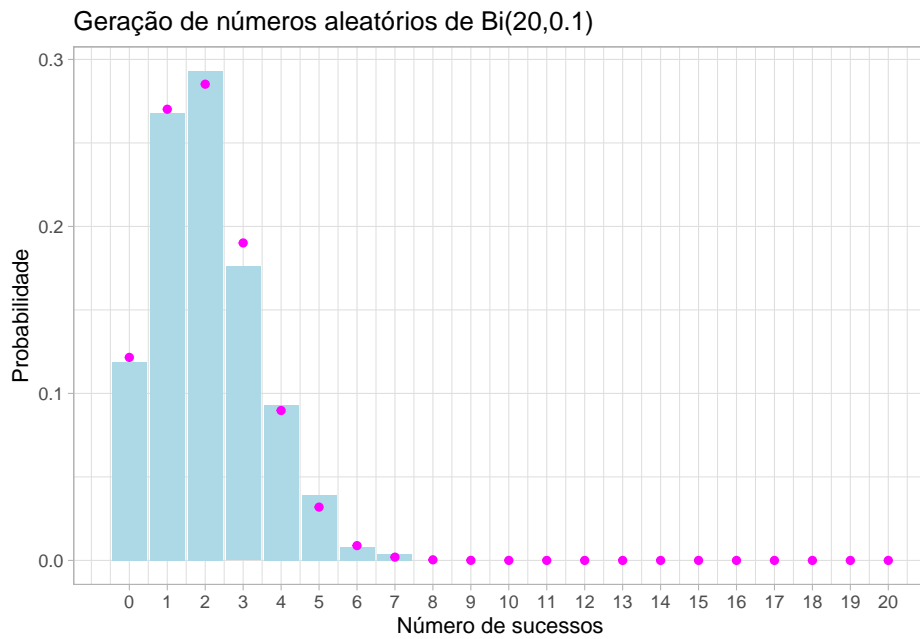
```
set.seed(1234)

n <- 20
p <- 0.1
k <- 1000 # número de simulações

dados <- data.frame(X = rbinom(k, size = n, prob = p))
teorico <- data.frame(x = 0:n, y=dbinom(0:n, size = n, prob = p))

# Carregue o pacote ggplot2
library(ggplot2)

ggplot(dados) +
  geom_bar(aes(x = X, y = after_stat(prop)), fill = "lightblue") +
  geom_point(data = teorico, aes(x, y), color = "magenta") +
  scale_x_continuous(breaks = 0:n) +
  labs(title = "Geração de números aleatórios de Bi(20,0.1)", x = "Número de sucessos",
    y = "Probabilidade") +
  theme_light()
```



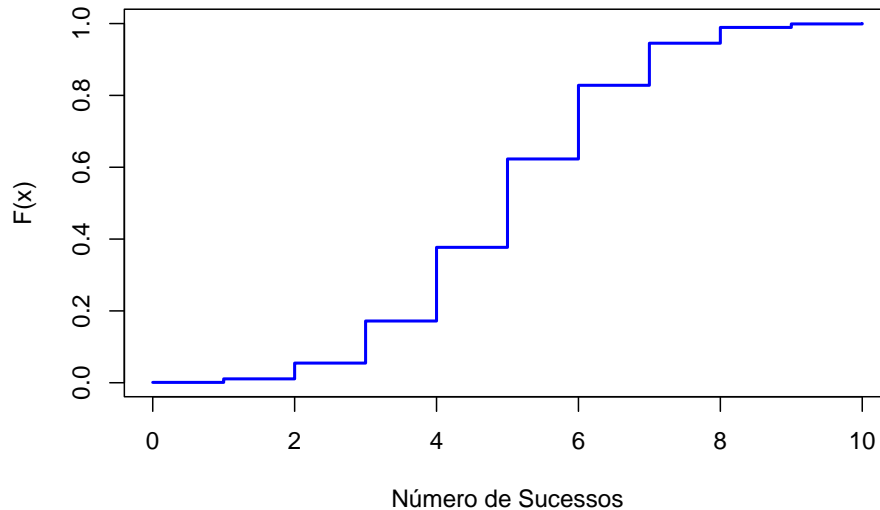
20.1.5 Função de distribuição

```
# Definir os parâmetros da distribuição binomial
n <- 10 # Número de tentativas
p <- 0.5 # Probabilidade de sucesso

# Valores possíveis de sucessos (0 a n)
x <- 0:n

# Calcular a FD
cdf_values <- pbinom(x, size = n, prob = p)

# Plotar a FD
plot(x, cdf_values, type = "s", lwd = 2, col = "blue",
     xlab = "Número de Sucessos", ylab = "F(x)",
     main = "Função de Distribuição Acumulada da Binomial(n = 10, p = 0.5)")
```

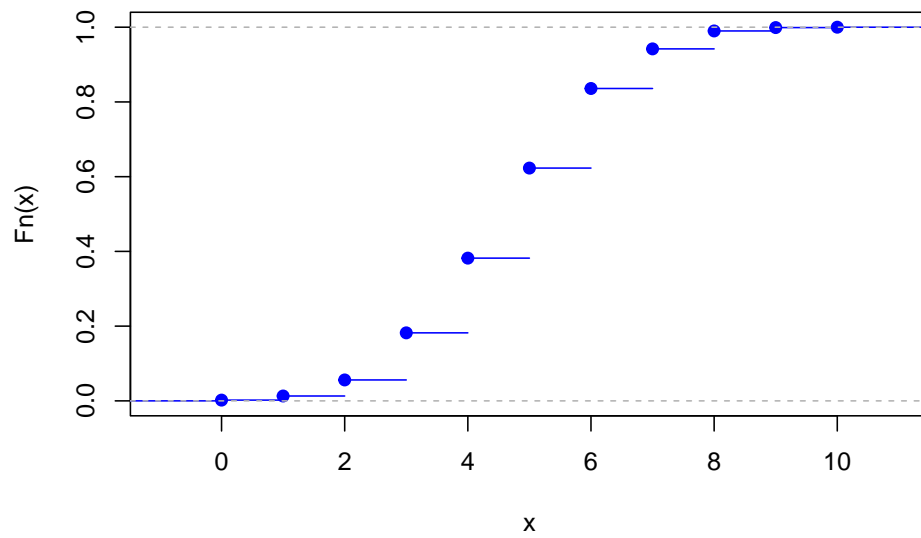

Função de Distribuição Acumulada da Binomial($n = 10$, $p = 0.5$)**20.1.6 Função de distribuição empírica**

```
# Definir os parâmetros da distribuição binomial
n <- 10 # Número de tentativas
p <- 0.5 # Probabilidade de sucesso

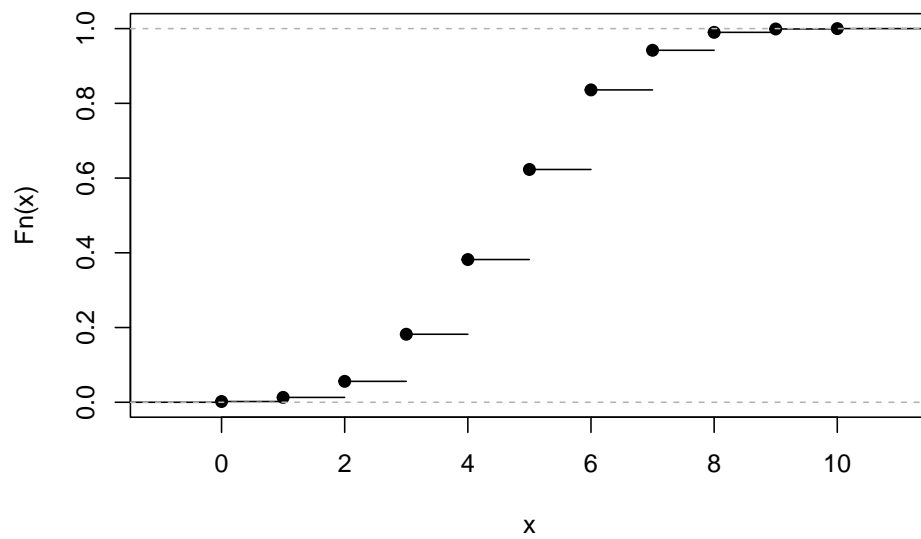
set.seed(123)
# Amostra aleatória de dimensão 1000
amostra <- rbinom(1000, size = n, prob = p)

# Distribuição empírica
Fn <- ecdf(amostra)

# Plotar CDF
plot(Fn, main = "Função de Distribuição Empírica", xlab = "x",
     ylab = "Fn(x)", col = "blue")
```

Função de Distribuição Empírica

```
# OU
plot.ecdf(amostra)
```

ecdf(x)

Cálculo de probabilidade: Seja $X \sim \text{Binomial}(n = 10, p = 0.5)$.

$$P(X \leq 4) = \text{pbinom}(4, 10, 0.5) = 0.377$$

$$P(X \leq 4) \approx \text{Fn}(4) = 0.382$$

20.2 Gerando uma variável aleatória com distribuição de Poisson

20.2.1 Cálculo de probabilidades

Seja $X \sim \text{Poisson}(\lambda = 5)$.

$P(X = 4) \rightarrow \text{dpois}(4, 5) = 0.1755$

$P(X \leq 4) \rightarrow \text{ppois}(4, 5) = 0.4405$

$P(X > 4) \rightarrow \text{ppois}(4, 5, \text{lower.tail}=\text{FALSE}) = 0.5595$

20.2.2 Função massa de probabilidade (teórica)

```
# Definir os valores de lambda e x
p <- c(0.1, 1, 2.5, 5, 15, 30)
x <- 0:50
```

```
# Carregar os pacotes necessários
library(ggplot2)
library(latex2exp)
library(gridExtra)
```

```
##
## Attaching package: 'gridExtra'

## The following object is masked from 'package:dplyr':
##
## combine
```

```
# Inicializar uma lista para armazenar os gráficos
plots <- list()

# Loop para criar os data frames e gráficos
for (i in 1:length(p)) {
  teorico <- data.frame(x = x, y = dpois(x, lambda = p[i]))

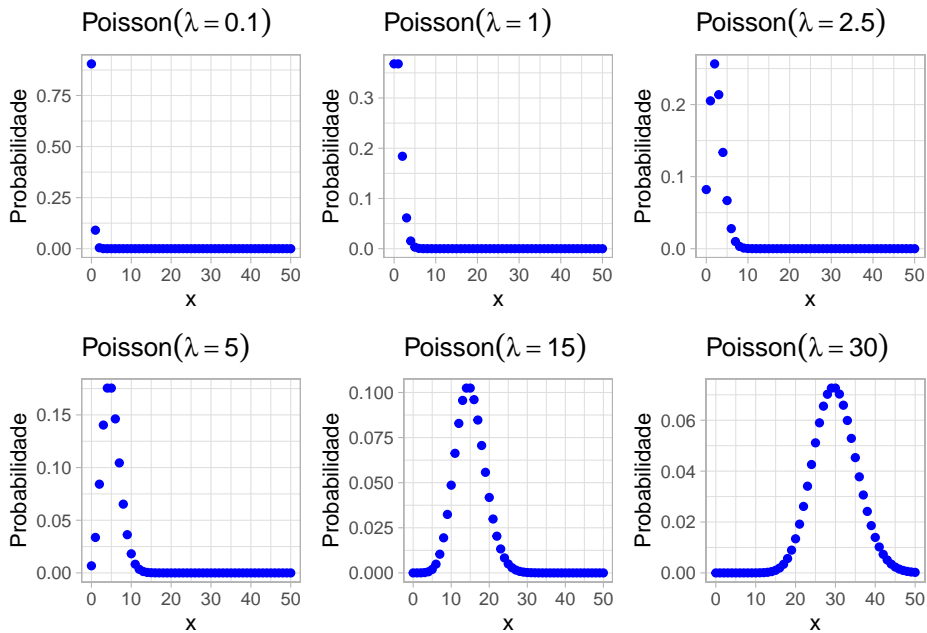
  plots[[i]] <- ggplot(teorico) +
    geom_point(aes(x = x, y = y), color = "blue") +
    scale_x_continuous(breaks = seq(0, 50, by = 10)) +
    labs(title = TeX(paste0("$Poisson(lambda=", p[i], ")$")), x="x", y="Probabilidade") +
    theme_light()
```

```

}

# Dispor os gráficos em uma grade 2x3
grid.arrange(grobs = plots, nrow = 2, ncol = 3)

```



20.2.3 Função massa de probabilidade (simulação)

```

p <- c(0.1, 1, 2.5, 5, 15, 30)
n <- 1000

# Carregar os pacotes necessários
library(ggplot2)
library(latex2exp)
library(gridExtra)

# Inicializar uma lista para armazenar os gráficos
plots <- list()

# Loop para criar os data frames e gráficos
for (i in 1:length(p)) {
  dados <- data.frame(X = rpois(n, lambda = p[i]))

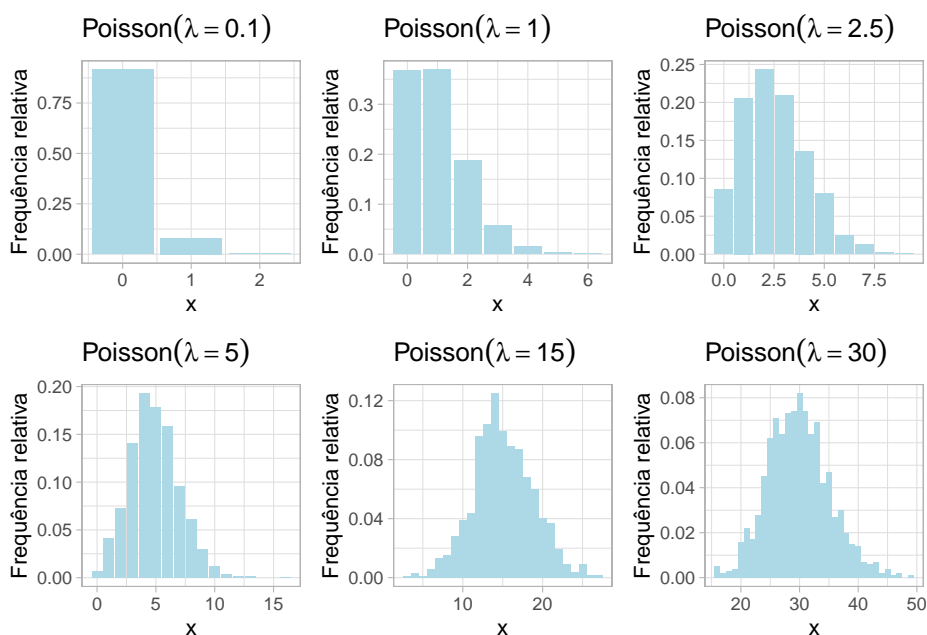
```

```

plots[[i]] <- ggplot(dados) +
  geom_bar(aes(x = X, y = after_stat(prop)), fill = "lightblue") +
  labs(title = TeX(paste("$Poisson(lambda=", p[i], ")$")),
        x = "x", y = "Frequência relativa") +
  theme_light()
}

# Dispor os gráficos em uma grade 2x3
grid.arrange(grobs = plots, nrow = 2, ncol = 3)

```



20.2.4 Comparação

```

p <- c(0.1, 1, 2.5, 5, 15, 30)
n <- 1000

# Carregar os pacotes necessários
library(ggplot2)
library(latex2exp)
library(gridExtra)

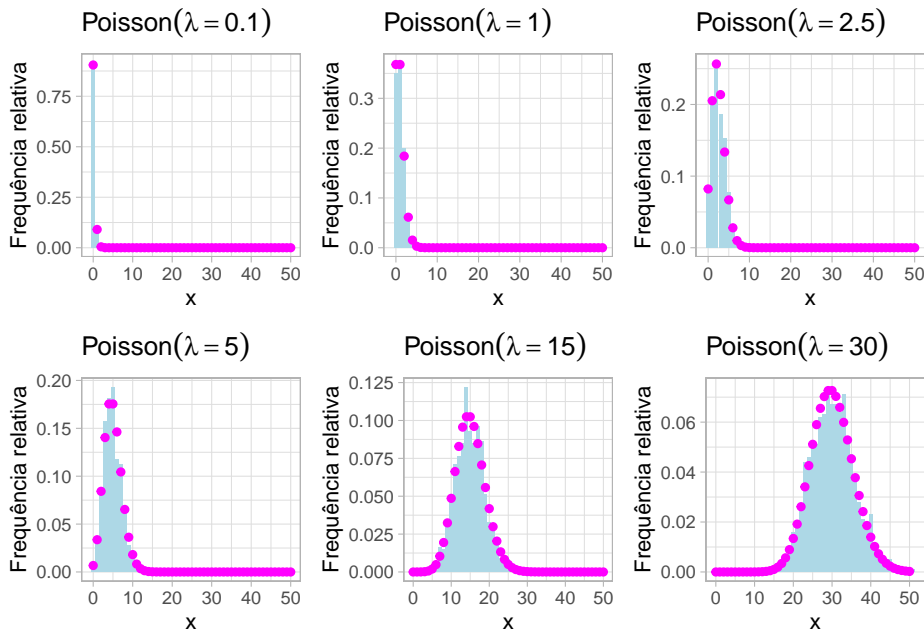
# Inicializar uma lista para armazenar os gráficos
plots <- list()

```

```
# Loop para criar os data frames e gráficos
for (i in 1:length(p)) {
  dados <- data.frame(X = rpois(n, lambda = p[i]))
  teorico <- data.frame(x=0:50, y=dpois(0:50,p[i]))

  plots[[i]] <- ggplot(dados) +
    geom_bar(aes(x = X, y =after_stat(prop)), fill="lightblue") +
    geom_point(data = teorico, aes(x, y), color = "magenta") +
    scale_x_continuous(breaks = seq(0, 50, by = 10)) +
    labs(title=TeX(paste("$Poisson(lambda=", p[i], ")$")),
         x = "x", y = "Frequência relativa") +
    theme_light()
}

# Dispor os gráficos em uma grade 2x3
grid.arrange(grobs = plots, nrow = 2, ncol = 3)
```

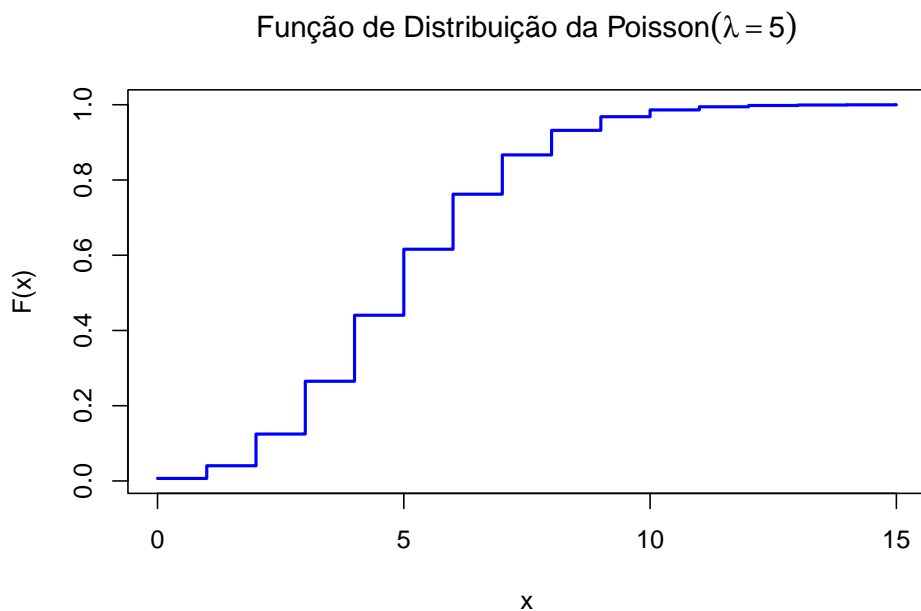


20.2.5 Função de distribuição

```
lambda <- 5 # Parâmetro da Poisson
x <- 0:15   # Valores de x para plotar a distribuição
```

```
# Calcular a FD
y <- ppois(x, lambda = lambda)

# Plotar a FD
plot(x,y, type="s", lwd=2, col="blue",
     main=TeX(paste("Função de Distribuição da $Poisson (lambda =", lambda, ")$")),
     xlab = "x",
     ylab = "F(x)")
```

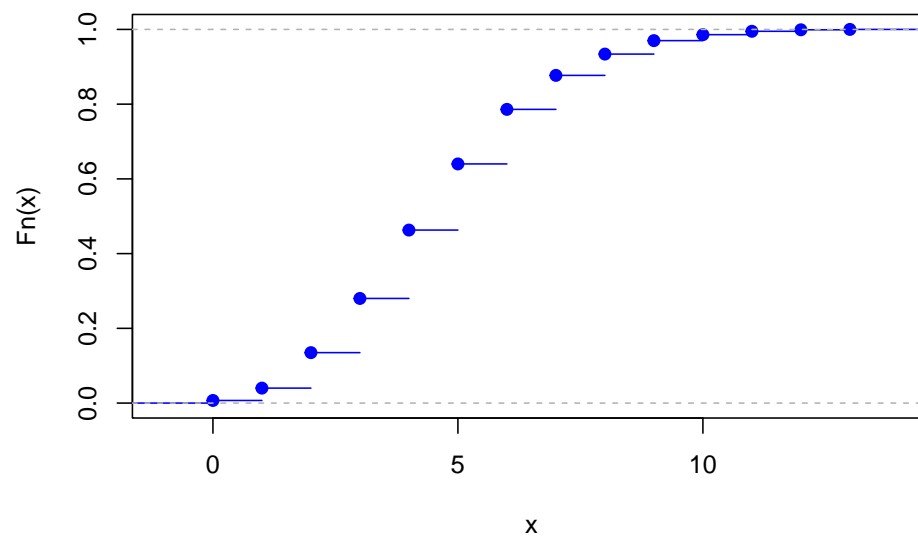


20.2.6 Função de distribuição empírica

```
library(latex2exp)
# Definir os parâmetros da distribuição de Poisson
lambda <- 5

dados <- rpois(1000, lambda = lambda)
Fn <- ecdf(dados)

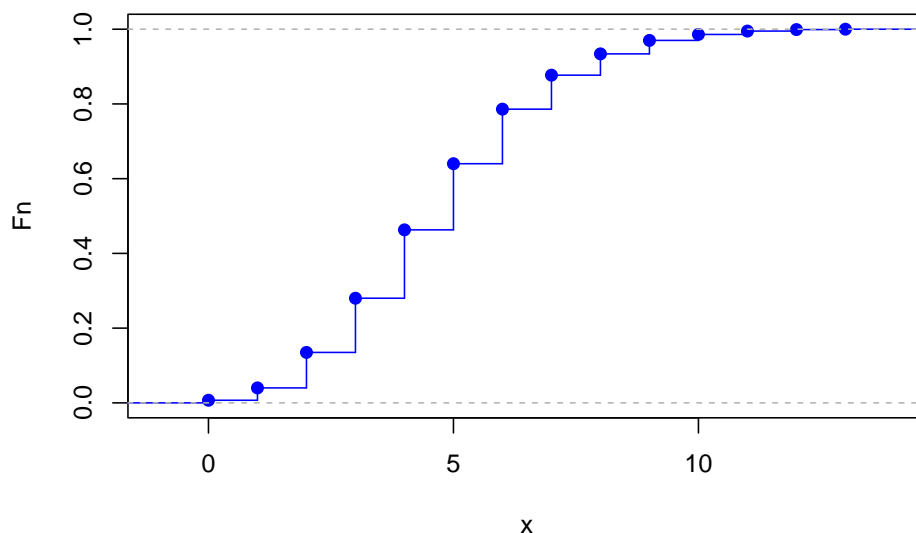
# Plotar CDF
plot(Fn, main=TeX("Função de Distribuição Empírica da $Poisson(lambda = 5)$"),
     xlab = "x",
     ylab = "Fn(x)",
     col = "blue")
```

Função de Distribuição Empírica da Poisson($\lambda = 5$)

```
# OU
#plot.ecdf(dados)

plot(Fn, main="Função de Distribuição Empírica",
     xlab="x",
     ylab="Fn",
     col="blue",
     verticals = TRUE)
```


Função de Distribuição Empírica



Cálculo de probabilidades: Seja $X \sim \text{Poisson}(\lambda = 5)$.

$$P(X \leq 4) \rightarrow \text{ppois}(4, 5) = 0.4405$$

$$P(X \leq 4) \rightarrow F_n(4) = 0.433$$

20.3 Gerando uma variável aleatória com distribuição de Uniforme

20.3.1 Cálculo de probabilidades

Seja $X \sim \text{Uniforme}(0, 1)$

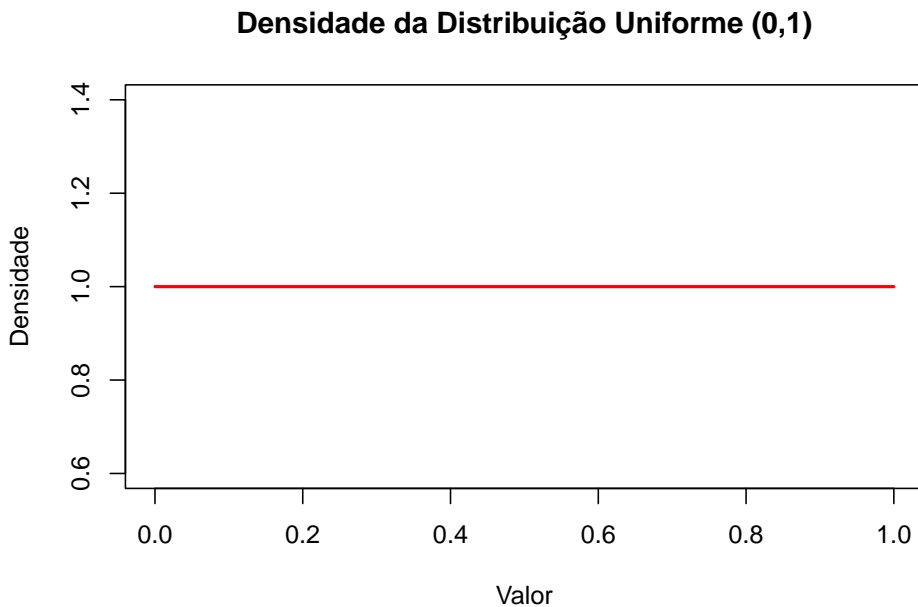
- $P(X \leq 0.5) \rightarrow \text{punif}(0.5, \text{min} = 0, \text{max} = 1) = 0.5$
- $P(X > 0.5) \rightarrow \text{punif}(0.5, \text{min} = 0, \text{max} = 1, \text{lower.tail} = \text{FALSE}) = 0.5$

20.3.2 Função densidade de probabilidade

```
# Gerar os valores x para a densidade teórica
x_vals <- seq(0, 1, length.out = 100)
```

```
# Calcular a densidade teórica para os valores x
y_vals <- dunif(x_vals, min = 0, max = 1)

# Desenhar o gráfico da função densidade de probabilidade
plot(x_vals, y_vals, type = "l",
     col = "red", lwd = 2,
     main = "Densidade da Distribuição Uniforme (0,1)",
     xlab = "Valor", ylab = "Densidade")
```



20.3.3 Função densidade de probabilidade (simulação)

```
# Definir o tamanho da amostra
n <- 10000

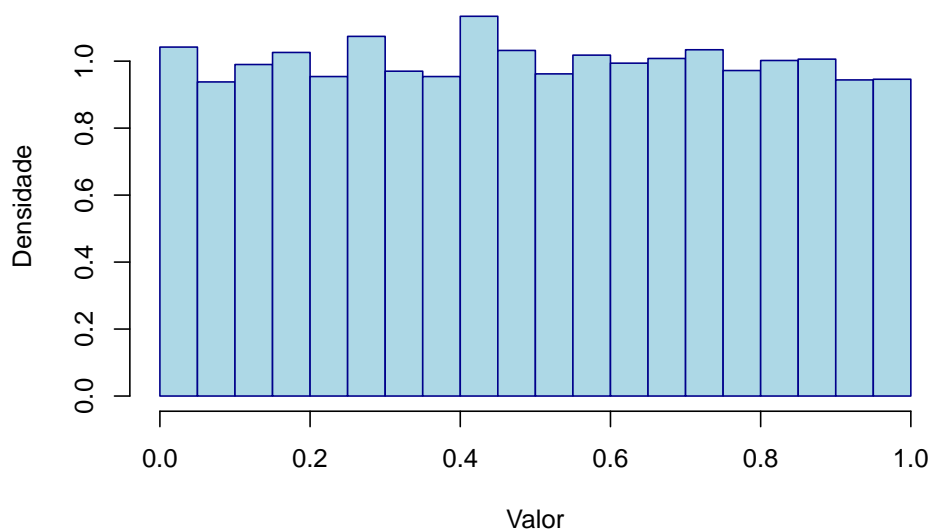
# Fixar a semente para reprodutibilidade
set.seed(123)

# Gerar a variável aleatória com distribuição uniforme (0,1)
uniform_data <- runif(n, min = 0, max = 1)

# Criar um histograma da amostra
hist(uniform_data, probability = TRUE,
     main = "Histograma da Densidade - Uniforme(0,1)",
```

```
xlab = "Valor",  
ylab = "Densidade",  
col = "lightblue",  
border = "darkblue")
```

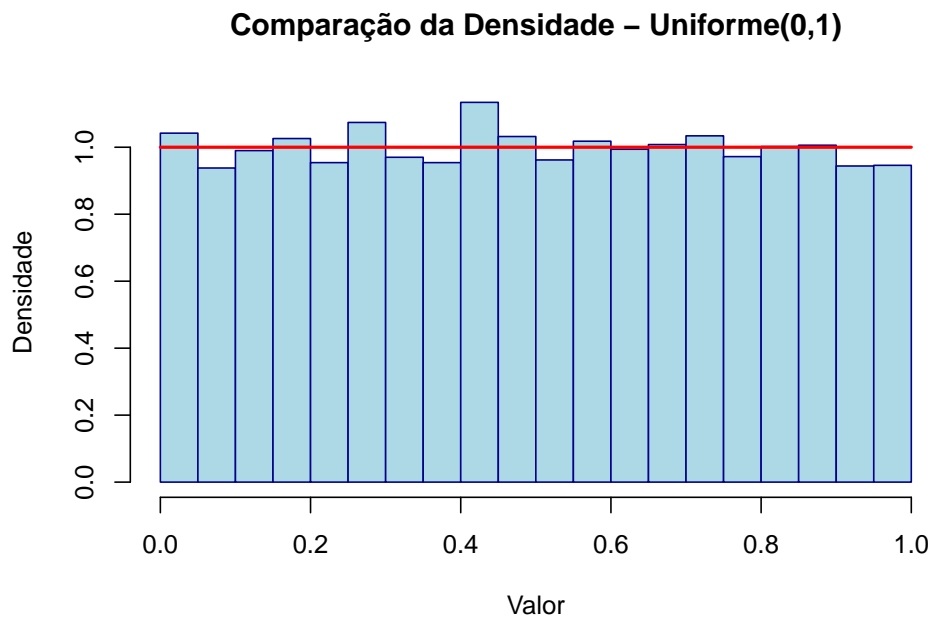
Histograma da Densidade – Uniforme(0,1)



20.3.4 Comparação

```
# Definir o tamanho da amostra  
n <- 10000  
  
# Fixar a semente para reprodutibilidade  
set.seed(123)  
  
# Gerar a variável aleatória com distribuição uniforme (0,1)  
uniform_data <- runif(n, min = 0, max = 1)  
  
# Criar um histograma da amostra com densidade  
hist(uniform_data, probability = TRUE,  
     main = "Comparação da Densidade – Uniforme(0,1)",  
     xlab = "Valor",  
     ylab = "Densidade",  
     col = "lightblue",  
     border = "darkblue")
```

```
# Adicionar a curva da densidade teórica
curve(dunif(x, min = 0, max = 1),
      add = TRUE,
      col = "red",
      lwd = 2)
```



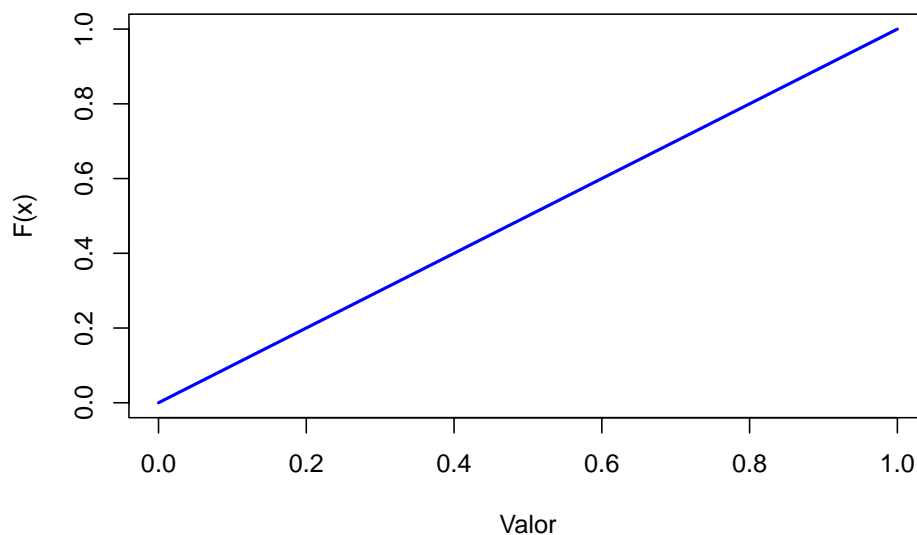
20.3.5 Função de distribuição

```
# Gerar os valores x para a FD teórica
x_vals <- seq(0, 1, length.out = 100)

# Calcular a FD teórica para os valores x
y_vals <- punif(x_vals, min = 0, max = 1)

# Desenhar o gráfico da função de distribuição acumulada
plot(x_vals, y_vals, type = "l",
     col = "blue", lwd = 2,
     main = "Função de Distribuição Uniforme (0,1)",
     xlab = "Valor", ylab = "F(x)")
```

Função de Distribuição Uniforme (0,1)



20.3.6 Função de distribuição empírica

```
# Definir o tamanho da amostra
n <- 10000

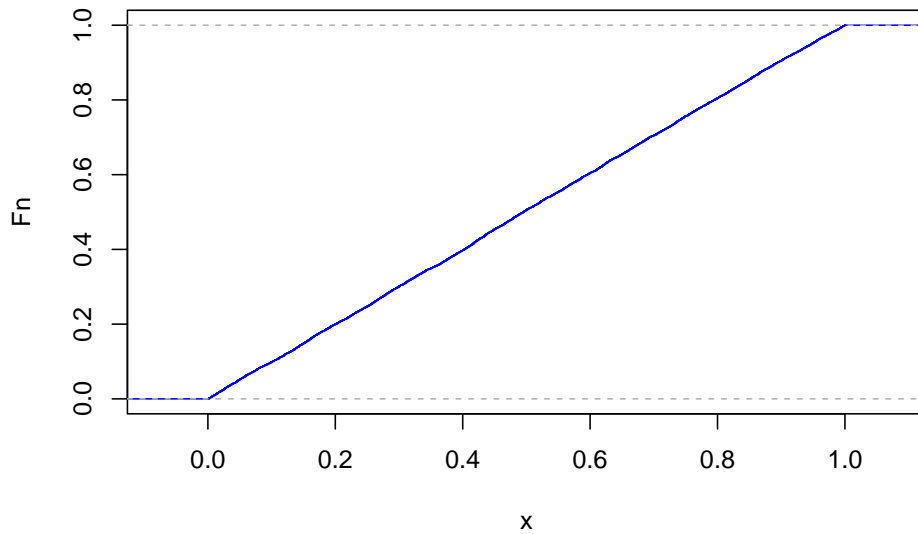
# Fixar a semente para reprodutibilidade
set.seed(123)

# Gerar a variável aleatória com distribuição uniforme (0,1)
uniform_data <- runif(n, min = 0, max = 1)

# Função de distribuição empírica
Fn <- ecdf(uniform_data)

plot(Fn, main="Função de Distribuição Empírica",
     xlab="x",
     ylab="Fn",
     col="blue")
```

Função de Distribuição Empírica



```
# OU  
#plot.ecdf(uniform_data)
```

20.4 Gerando uma variável aleatória com distribuição Exponencial

20.4.1 Cálculo de probabilidades

Seja $X \sim \text{Exponencial}(\lambda = 1)$.

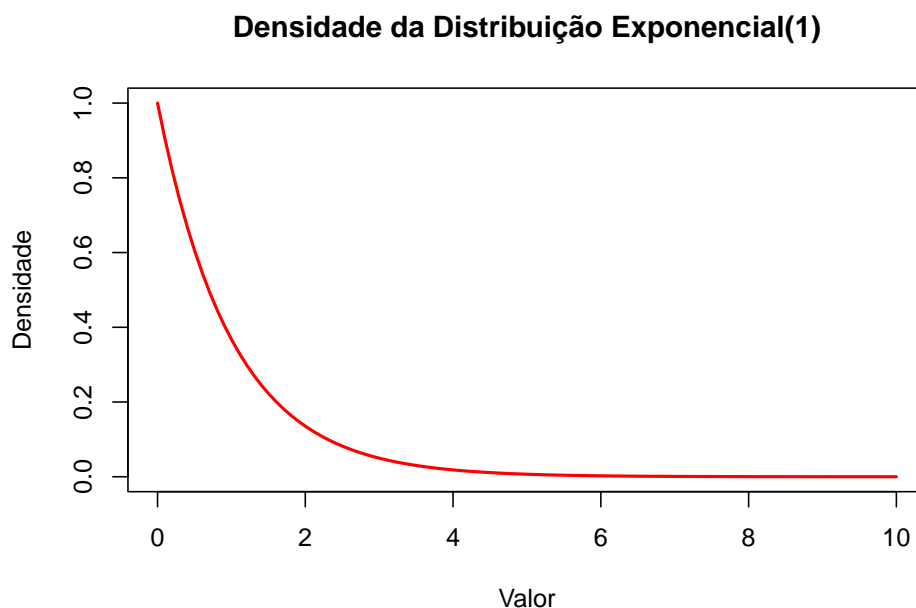
$P(X \leq 0.5) \rightarrow \text{pexp}(0.5, \text{rate}=1) = 0.3935$

$P(X > 0.5) \rightarrow \text{pexp}(0.5, \text{rate}=1, \text{lower.tail}=FALSE) = 0.6065$

20.4.2 Função densidade de probabilidade (teórica)

```
# Gerar os valores x para a densidade teórica  
x_vals <- seq(0, 10, length.out = 100)  
  
# Calcular a densidade teórica para os valores x  
y_vals <- dexp(x_vals, rate=1)
```

```
# Desenhar o gráfico da função densidade de probabilidade
plot(x_vals, y_vals, type = "l",
     col = "red", lwd = 2,
     main = "Densidade da Distribuição Exponencial(1)",
     xlab = "Valor", ylab = "Densidade")
```



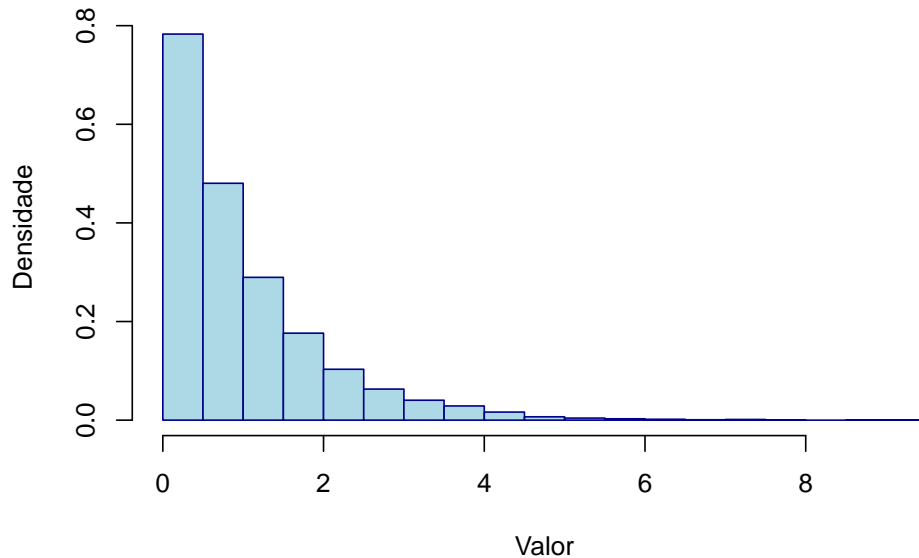
20.4.3 Função densidade de probabilidade (simulação)

```
# Definir o tamanho da amostra
n <- 10000

# Fixar a semente para reprodutibilidade
set.seed(123)

# Gerar a variável aleatória com distribuição exponencial(1)
expo_data <- rexp(n, rate=1)

# Criar um histograma da amostra
hist(expo_data, probability = TRUE,
     main = "Histograma da Densidade - Exponencial(1)",
     xlab = "Valor",
     ylab = "Densidade",
     col = "lightblue",
     border = "darkblue")
```

Histograma da Densidade – Exponencial(1)**20.4.4 Comparação**

```
# Definir o tamanho da amostra
n <- 10000

# Fixar a semente para reprodutibilidade
set.seed(123)

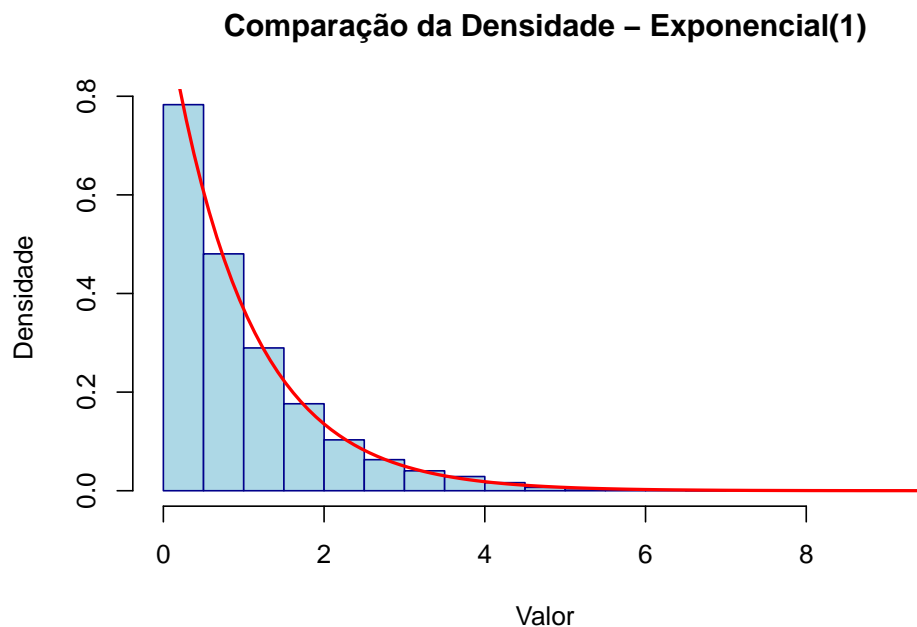
# Gerar a variável aleatória com distribuição exponencial(1)
expo_data <- rexp(n, rate=1)

# Criar um histograma da amostra
hist(expo_data, probability = TRUE,
     main = "Comparação da Densidade - Exponencial(1)",
     xlab = "Valor",
     ylab = "Densidade",
     col = "lightblue",
     border = "darkblue")

# Adicionar curva da densidade teórica
curve(dexp(x, rate=1),
      add=TRUE,
      col="red",
```



```
lwd=2)
```



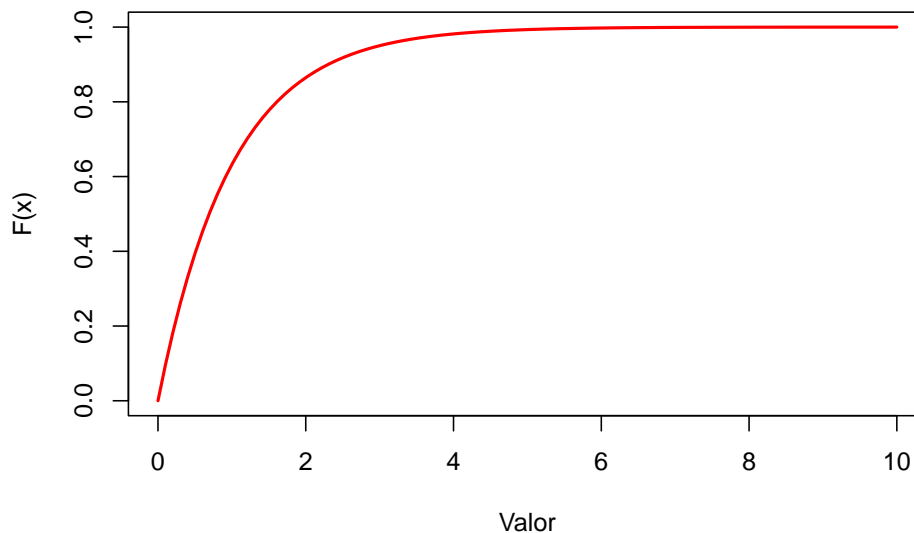
20.4.5 Função de distribuição

```
# Gerar os valores x para a FD teórica
x_vals <- seq(0, 10, length.out = 100)

# Calcular a FD teórica para os valores x
y_vals <- pexp(x_vals, rate=1)

# Desenhar o gráfico da FD
plot(x_vals, y_vals, type = "l",
     col = "red", lwd = 2,
     main = "Função de Distribuição Exponencial(1)",
     xlab = "Valor", ylab = "F(x)")
```

Função de Distribuição Exponencial(1)



20.4.6 Função de distribuição empírica

```
# Definir o tamanho da amostra
n <- 10000

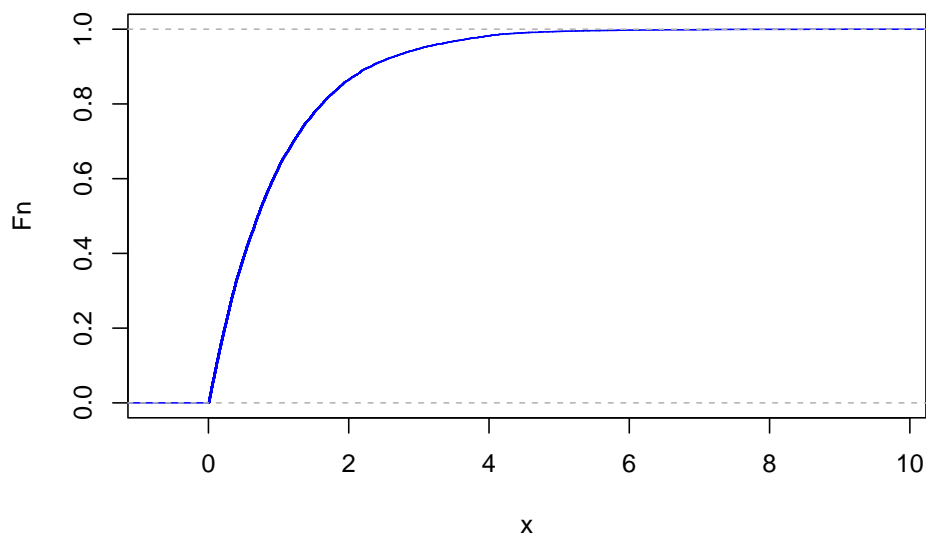
# Fixar a semente para reprodutibilidade
set.seed(123)

# Gerar a variável aleatória com distribuição exponencial(1)
expo_data <- rexp(n, rate=1)

# Função de distribuição empírica
Fn <- ecdf(expo_data)

plot(Fn, main="Função de Distribuição Empírica",
     xlab="x",
     ylab="Fn",
     col="blue")
```

Função de Distribuição Empírica



20.5 Gerando uma variável aleatória com distribuição Normal

20.5.1 Cálculo de probabilidades

Seja $X \sim \text{Normal}(0, 1)$.

$P(X \leq 0.5) \rightarrow \text{pnorm}(0.5, \text{mean}=0, \text{sd}=1) = 0.6915$

$P(X > 0.5) \rightarrow \text{pnorm}(0.5, \text{mean}=0, \text{sd}=1, \text{lower.tail}=\text{FALSE}) = 0.3085$

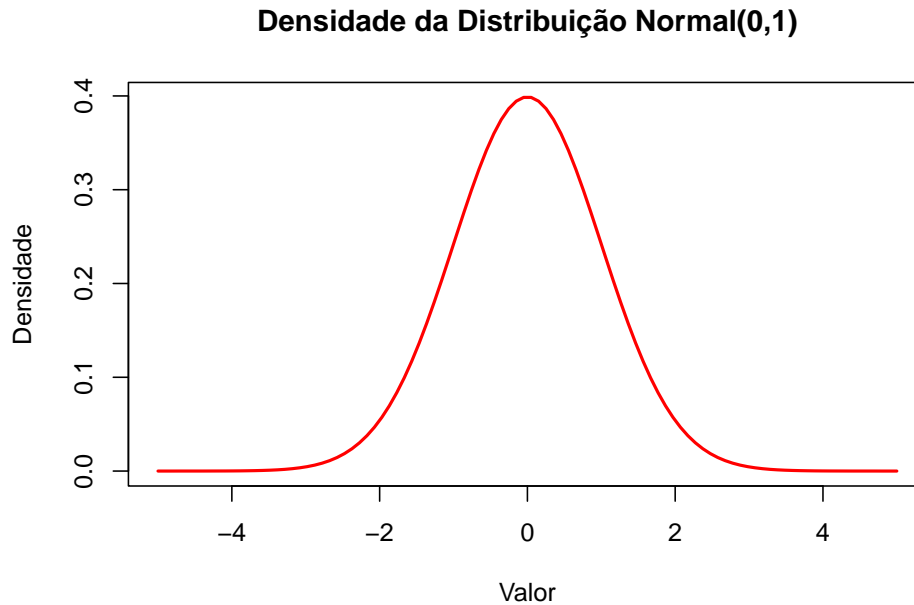
20.5.2 Função densidade de probabilidade (teórica)

```
# Gerar os valores x para a densidade teórica
x_vals <- seq(-5, 5, length.out = 100)

# Calcular a densidade teórica para os valores x
y_vals <- dnorm(x_vals, mean = 0, sd = 1)

# Desenhar o gráfico da função densidade de probabilidade
plot(x_vals, y_vals, type = "l",
     col = "red", lwd = 2,
```

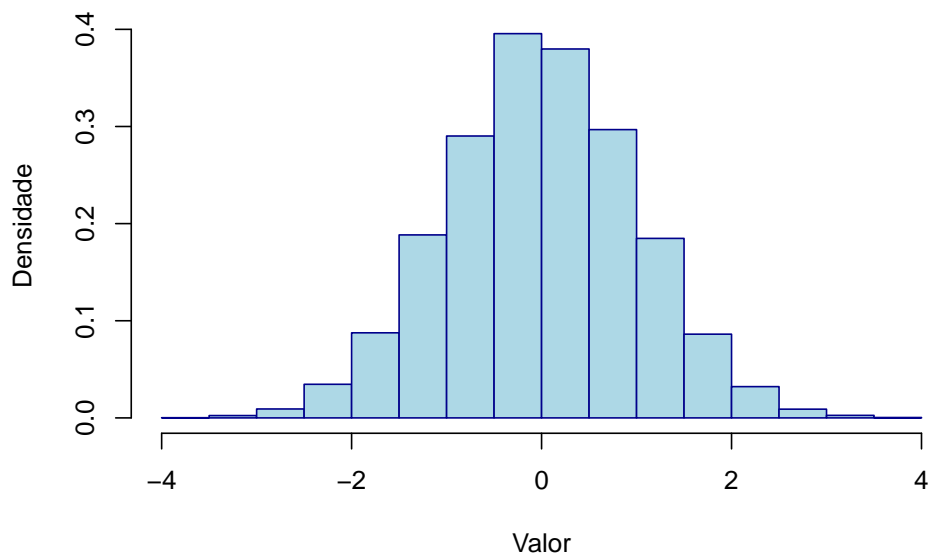
```
main = "Densidade da Distribuição Normal(0,1)",  
xlab = "Valor", ylab = "Densidade")
```



20.5.3 Função densidade de probabilidade (simulação)

```
# Definir o tamanho da amostra  
n <- 10000  
  
# Fixar a semente para reprodutibilidade  
set.seed(123)  
  
# Gerar a variável aleatória com distribuição Normal(0,1)  
normal_data <- rnorm(n, mean = 0, sd = 1)  
  
# Criar um histograma da amostra com densidade  
hist(normal_data, probability = TRUE,  
      main = "Comparação da Densidade - Normal(0,1)",  
      xlab = "Valor",  
      ylab = "Densidade",  
      col = "lightblue",  
      border = "darkblue")
```

Comparação da Densidade – Normal(0,1)



20.5.4 Comparação

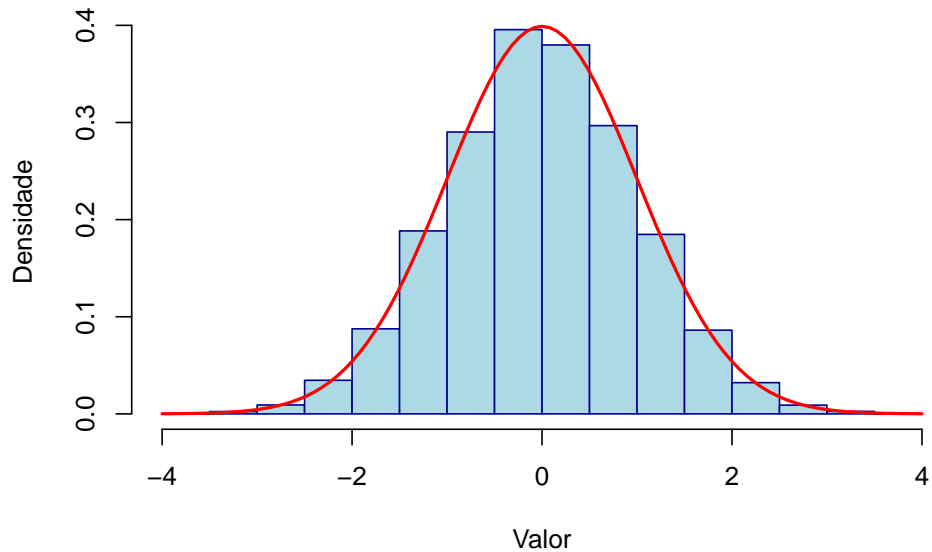
```
# Definir o tamanho da amostra
n <- 10000

# Fixar a semente para reprodutibilidade
set.seed(123)

# Gerar a variável aleatória com distribuição Normal(0,1)
normal_data <- rnorm(n, mean = 0, sd = 1)

# Criar um histograma da amostra com densidade
hist(normal_data, probability = TRUE,
      main = "Comparação da Densidade - Normal(0,1)",
      xlab = "Valor",
      ylab = "Densidade",
      col = "lightblue",
      border = "darkblue")

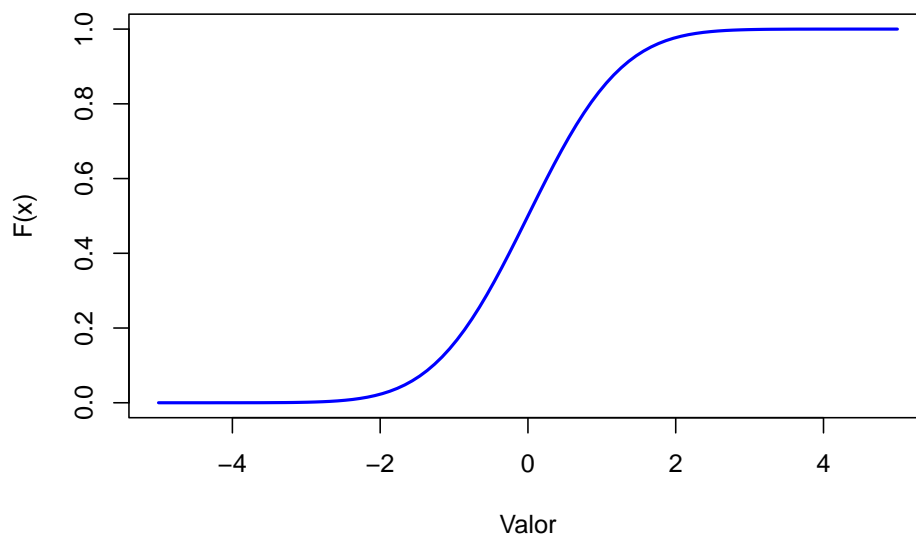
# Adicionar a curva da densidade teórica
curve(dnorm(x, mean = 0, sd = 1),
      add = TRUE,
      col = "red",
      lwd = 2)
```

Comparação da Densidade – Normal(0,1)**20.5.5 Função de distribuição**

```
# Gerar os valores x para a FD teórica
x_vals <- seq(-5, 5, length.out = 100)

# Calcular a FD teórica para os valores x
y_vals <- pnorm(x_vals, mean = 0, sd = 1)

# Desenhar o gráfico da função de distribuição
plot(x_vals, y_vals, type = "l",
     col = "blue", lwd = 2,
     main = "Função de Distribuição Normal(0,1)",
     xlab = "Valor", ylab = "F(x)")
```

Função de Distribuição Normal(0,1)**20.5.6 Função de distribuição empírica**

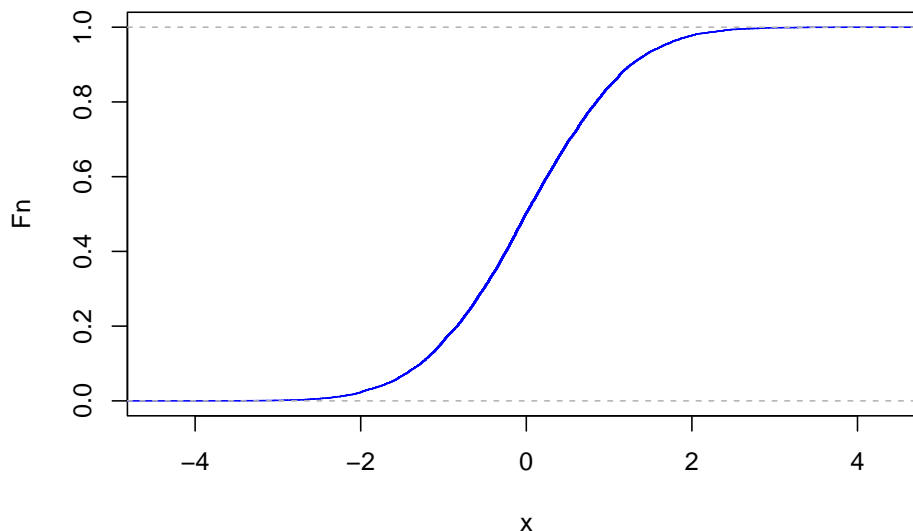
```
# Definir o tamanho da amostra
n <- 10000

# Fixar a semente para reprodutibilidade
set.seed(123)

# Gerar a variável aleatória com distribuição Normal(0,1)
normal_data <- rnorm(n, mean = 0, sd = 1)

# Função de distribuição empírica
Fn <- ecdf(normal_data)

plot(Fn, main="Função de Distribuição Empírica",
     xlab="x",
     ylab="Fn",
     col="blue")
```

Função de Distribuição Empírica**20.6 Exercícios**

1. Usando o R e fixando a semente em 123, simule 1000 lançamentos de uma moeda com probabilidade de 0.5 de sair cara. Conte o número de caras em cada lançamento e plote um histograma dos resultados.
2. Usando o R e fixando a semente em 123, gere uma amostra aleatória de 5000 observações de uma variável aleatória binomial com parâmetros $n = 10$ e $p = 0.3$. Calcule a média e a variância das observações geradas.
3. Usando o R e fixando a semente em 123, gere uma amostra aleatória de 2300 observações de uma variável aleatória de Poisson com parâmetro $\lambda = 4$. Calcule a média e o desvio padrão das observações geradas.
4. Em um processo de qualidade, considere uma variável aleatória X que representa o número de produtos defeituosos em um lote de 50 produtos, onde a probabilidade de um produto ser defeituoso é 0.1. Usando o R e fixando a semente em 123 gere uma amostra aleatória de 10000 observações de X . Conte a frequência de lotes com exatamente 5 produtos defeituosos. Calcule a proporção de lotes com exatamente 5 produtos defeituosos e compare o valor obtido com a probabilidade $P(X = 5)$, onde $X \sim \text{Binomial}(50, 0.1)$.
5. Usando o R e fixando a semente em 123, gere uma amostra aleatória de 5000 observações de uma variável aleatória X binomial com parâmetros $n = 20$ e $p = 0.7$.

(a) Faça um histograma de frequência relativa associado aos valores amostrais. Sobreponha no gráfico a distribuição de probabilidade de X .

(b) Use a função de distribuição empírica para estimar $P(X \leq 10)$ e compare com o valor teórico.

6. Usando o R e fixando a semente em 543, gere uma amostra aleatória de 2400 observações de uma variável aleatória Y de Poisson com parâmetro $\lambda = 6$.

(a) Faça um histograma de frequência relativa associado aos valores amostrais. Sobreponha no gráfico a distribuição de probabilidade de X .

(b) Use a função de distribuição empírica para estimar $P(Y > 5)$ e compare com o valor teórico.

7. Usando o R e fixando a semente em 345, gere uma amostra aleatória de 3450 observações de uma variável aleatória Z uniforme no intervalo $[0, 1]$. Use a função de distribuição empírica para estimar $P(Z \leq 0.5)$ e compare com o valor teórico.

8. Usando o R e fixando a semente em 123, gere uma amostra aleatória de 3467 observações de uma variável aleatória W normal com média $\mu = 0$ e desvio padrão $\sigma = 1$.

(a) Faça um histograma de frequência relativa associado aos valores amostrais. Sobreponha no gráfico a distribuição de X .

(b) Use a função de distribuição empírica para estimar $P(W > 1)$ e compare com o valor teórico.

9. Usando o R e fixando a semente em 123, gere uma amostra aleatória de 1234 observações de uma variável aleatória V exponencial com parâmetro $\lambda = 0.5$.

(a) Faça um histograma de frequência relativa associado aos valores amostrais. Sobreponha no gráfico a distribuição de probabilidade de X .

(b) Use a função de distribuição empírica para estimar $P(V > 2)$ e compare com o valor teórico.

10. O número de acertos num alvo em 30 tentativas onde a probabilidade de acerto é 0.4, é modelado por uma variável aleatória X com distribuição Binomial de parâmetros $n = 30$ e $p = 0.4$. Usando o R e fixando a semente em 123, gere uma amostra de dimensão $n = 700$ dessa variável. Para essa amostra:

(a) Faça um histograma de frequência relativa associado aos valores amostrais. Sobreponha no gráfico a distribuição de probabilidade de X .

(b) Calcule a função de distribuição empírica e com base nessa função estime a probabilidade do número de acertos no alvo, em 30 tentativas, ser maior que 15. Calcule ainda o valor teórico dessa probabilidade.

11. Usando o R e fixando a semente em 123, gere amostras de tamanho crescente $n = 100, 1000, 10000, 100000$ de uma variável aleatória X com distribuição

de Poisson com parâmetro $\lambda = 3$. Para cada tamanho de amostra, calcule a média amostral e compare-a com o valor esperado teórico. Observe e comente a convergência das médias amostrais.

12. Usando o R e fixando a semente em 123, gere amostras de tamanho crescente $n = 100, 1000, 10000, 100000$ de uma variável aleatória W com distribuição uniforme no intervalo $[0, 1]$. Para cada tamanho de amostra, calcule a média amostral e compare-a com o valor esperado teórico. Observe e comente a convergência das médias amostrais.

13. Um grupo de estudantes de Estatística está realizando uma pesquisa para avaliar o grau de satisfação dos alunos com um novo curso oferecido pela universidade. Cada estudante responde a uma pergunta onde pode indicar se está satisfeito ou insatisfeito com o curso. A probabilidade de um estudante estar satisfeito é de 0.75.

- Usando o R e fixando a semente em 42, simule amostras de tamanho crescente $n = 100, 500, 1000, 5000, 10000$ de uma variável aleatória X com distribuição binomial, onde X representa o número de estudantes satisfeitos. Para cada tamanho de amostra, calcule a proporção de estudantes satisfeitos e compare-a com a probabilidade teórica de satisfação (0.75).

14. Usando o R e fixando a semente em 1058, gere 9060 amostras de dimensão 9 de uma população, $X \sim \text{Binomial}(41, 0.81)$. Calcule a média de cada uma dessas amostras, obtendo uma amostra de médias. Calcule ainda o valor esperado da distribuição teórica de X e compare com a média da amostra de médias.

15. Em um hospital, o tempo de atendimento de pacientes segue uma distribuição exponencial com média de 30 minutos. Um pesquisador deseja estimar o tempo médio de atendimento coletando amostras de diferentes tamanhos.

- Usando o R e fixando a semente em 456, simule 1000 amostras de tamanho 50, 100 e 1000 do tempo de atendimento. Para cada tamanho de amostra, calcule a média de cada amostra e plote o histograma das médias amostrais para cada tamanho. Compare essas distribuições com a distribuição normal com média $E(X)$ e desvio padrão $\sqrt{V(X)n}$ e comente sobre a aplicação do Teorema do Limite Central.

16. O tempo de espera (em minutos) para o atendimento no setor de informações de um banco é modelado por uma variável aleatória X com distribuição Uniforme($a = 5, b = 20$). Usando o R e fixando a semente em 1430, gere 8000 amostras de dimensão $n = 100$ dessa variável. Para essas amostras:

(a) Calcule a soma de cada uma das amostras obtendo assim valores da distribuição da soma $S_n = \sum_{i=1}^n X_n$.

(b) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da soma e sobreponha no gráfico uma curva com distribuição normal de valor esperado $nE(X)$ e desvio padrão $\sqrt{V(X)n}$.

(c) Calcule a média de cada uma das amostras obtendo assim valores da distribuição da média \bar{X}_n .

(d) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da média \bar{X}_n . Sobreponha no gráfico uma curva com distribuição normal com valor esperado $E(X)$ e desvio padrão $\sqrt{V(x)/n}$.

17. O tempo de atendimento (em minutos), de doentes graves num determinado hospital, é modelado por uma variável aleatória X com distribuição Exponencial($\lambda = 0.21$). Usando o R e fixando a semente em 1580, gere 1234 amostras de dimensão $n = 50$ dessa variável. Para essas amostras:

(a) Calcule a soma de cada uma das amostras obtendo assim valores da distribuição da soma $S_n = \sum_{i=1}^n X_n$.

(b) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da soma e sobreponha no gráfico uma curva com distribuição normal de valor esperado $nE(X)$ e desvio padrão $\sqrt{V(X)n}$.

(c) Calcule agora a soma padronizada

$$\frac{S_n - E(S_n)}{\sqrt{V(S_n)}}$$

e faça um histograma de frequência relativa associado aos valores obtidos da distribuição da soma padronizada. Sobreponha no gráfico uma curva com distribuição normal de valor esperado 0 e desvio padrão 1.

(d) Calcule a média de cada uma das amostras obtendo assim valores da distribuição da média \bar{X}_n .

(e) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da média \bar{X}_n . Sobreponha no gráfico uma curva com distribuição normal com valor esperado $E(X)$ e desvio padrão $\sqrt{V(x)/n}$.

18. A altura (em centímetros) dos alunos de uma escola é modelada por uma variável aleatória X com distribuição Normal($\mu = 170, \sigma = 10$). Usando o R e fixando a semente em 678, gere 9876 amostras de dimensão $n = 80$ dessa variável. Para essas amostras:

(a) Calcule a soma de cada uma das amostras obtendo assim valores da distribuição da soma $S_n = \sum_{i=1}^n X_n$.

(b) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da soma e sobreponha no gráfico uma curva com distribuição normal de valor esperado $nE(X)$ e desvio padrão $\sqrt{V(X)n}$.

(c) Calcule agora a soma padronizada

$$\frac{S_n - E(S_n)}{\sqrt{V(S_n)}}$$

e faça um histograma de frequência relativa associado aos valores obtidos da distribuição da soma padronizada. Sobreponha no gráfico uma curva com distribuição normal de valor esperado 0 e desvio padrão 1.

(d) Calcule a média de cada uma das amostras obtendo assim valores da distribuição da média \bar{X}_n .

(e) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da média \bar{X}_n . Sobreponha no gráfico uma curva com distribuição normal com valor esperado $E(X)$ e desvio padrão $\sqrt{V(x)/n}$.

(f) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da média padronizada

$$\frac{\bar{X}_n - E(\bar{X}_n)}{\sqrt{V(\bar{X}_n)}}$$

e sobreponha no gráfico com uma curva com distribuição Normal com valor esperado 0 e desvio padrão 1.

19. A chegada de clientes em uma loja durante 1 hora, assumindo uma taxa média de 20 clientes por hora pode ser modelada por uma variável aleatória X com distribuição de Poisson($\lambda = 20$). Usando o R e fixando a semente em 1222, gere 8050 amostras de dimensão 30 de X .

(a) Calcule a soma de cada uma das amostras obtendo assim valores da distribuição da soma $S_n = \sum_{i=1}^n X_n$.

(b) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da soma e sobreponha no gráfico uma curva com distribuição normal de valor esperado $nE(X)$ e desvio padrão $\sqrt{V(X)n}$.

(c) Calcule agora a soma padronizada

$$\frac{S_n - E(S_n)}{\sqrt{V(S_n)}}$$

e faça um histograma de frequência relativa associado aos valores obtidos da distribuição da soma padronizada. Sobreponha no gráfico uma curva com distribuição normal de valor esperado 0 e desvio padrão 1.

(d) Calcule a média de cada uma das amostras obtendo assim valores da distribuição da média \bar{X}_n .

(e) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da média \bar{X}_n . Sobreponha no gráfico uma curva com distribuição normal com valor esperado $E(X)$ e desvio padrão $\sqrt{V(x)/n}$.

(f) Faça um histograma de frequência relativa associado aos valores obtidos da distribuição da média padronizada

$$\frac{\bar{X}_n - E(\bar{X}_n)}{\sqrt{V(\bar{X}_n)}}$$

e sobreponha no gráfico com uma curva com distribuição Normal com valor esperado 0 e desvio padrão 1.

Chapter 21

Relatórios

21.1 Markdown

21.2 R Markdown

Chapter 22

Referências

- <https://cemapre.iseg.ulisboa.pt/~nbrites/CTA/index.html>
- <https://livro.curso-r.com/>
- <https://www.cs.upc.edu/~robert/teaching/estadistica/TheRBook.pdf>
- Ross, S. M.: *Simulation*. Sixth edition. Academic Press, 2022.

Chapter 23

Respostas

23.1 O pacote dplyr

23.1.1 Selecionando colunas

1. Teste aplicar a função `glimpse()` do pacote `dplyr` à base `sw`. O que ela faz?

```
glimpse(sw)
```

```
## Rows: 87
## Columns: 14
## $ name      <chr> "Luke Skywalker", "C-3P0", "R2-D2", "Darth Vader", "Leia Or~
## $ height    <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2~
## $ mass      <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.~
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N~
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "~
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",~
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, ~
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female",~
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini~
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T~
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma~
## $ films      <list> <"A New Hope", "The Empire Strikes Back", "Return of the J~
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp~
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",~
```

Mostra os nomes das variáveis, os tipos de dados e os primeiros valores de cada coluna em uma única visualização, tudo de forma horizontal.

2. Crie uma tabela com apenas as colunas `name`, `gender`, e `films`. Salve em um objeto chamado `sw_simples`.

```
sw_simples <- select(sw, name, gender, films)
sw_simples
```

```
## # A tibble: 87 x 3
##   name                gender  films
##   <chr>              <chr>   <list>
## 1 Luke Skywalker    masculine <chr [5]>
## 2 C-3P0             masculine <chr [6]>
## 3 R2-D2             masculine <chr [7]>
## 4 Darth Vader       masculine <chr [4]>
## 5 Leia Organa       feminine <chr [5]>
## 6 Owen Lars         masculine <chr [3]>
## 7 Beru Whitesun Lars feminine <chr [3]>
## 8 R5-D4             masculine <chr [1]>
## 9 Biggs Darklighter masculine <chr [1]>
## 10 Obi-Wan Kenobi    masculine <chr [6]>
## # i 77 more rows
```

3. Selecione apenas as colunas `hair_color`, `skin_color` e `eye_color` usando a função auxiliar `contains()`.

```
select(sw, contains("color"))
```

```
## # A tibble: 87 x 3
##   hair_color  skin_color eye_color
##   <chr>      <chr>      <chr>
## 1 blond     fair       blue
## 2 <NA>      gold       yellow
## 3 <NA>      white, blue red
## 4 none      white      yellow
## 5 brown     light      brown
## 6 brown, grey light      blue
## 7 brown     light      blue
## 8 <NA>      white, red red
## 9 black     light      brown
## 10 auburn, white fair      blue-gray
## # i 77 more rows
```

4. Usando a função `select()` (e suas funções auxiliares), escreva códigos que retornem a base `sw` sem as colunas `hair_color`, `skin_color` e `eye_color`. Escreva todas as soluções diferentes que você conseguir pensar.

```
select(sw, -hair_color, -skin_color, -eye_color)

select(sw, -contains("color"))

select(sw, -ends_with("color"))

select(sw, name:mass, birth_year:starships)
```

23.1.2 Ordenando a base

1. Ordene `mass` em ordem crescente e `birth_year` em ordem decrescente e salve em um objeto chamado `sw_ordenados`.

```
sw_ordenados <- arrange(sw, mass, desc(birth_year))
sw_ordenados
```

```
## # A tibble: 87 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex  gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Ratts T~    79    15 none      grey, blue unknown      NA male  mascu~
## 2 Yoda        66    17 white     green      brown      896 male  mascu~
## 3 Wicket ~    88    20 brown     brown      brown      8 male  mascu~
## 4 R2-D2       96    32 <NA>      white, bl~ red       33 none  mascu~
## 5 R5-D4       97    32 <NA>      white, red red       NA none  mascu~
## 6 Sebulba    112    40 none      grey, red  orange     NA male  mascu~
## 7 Padmé A~   185    45 brown     light      brown     46 fema~ femin~
## 8 Dud Bolt    94    45 none      blue, grey yellow     NA male  mascu~
## 9 Wat Tam~   193    48 none      green, gr~ unknown    NA male  mascu~
## 10 Sly Moo~  178    48 none      pale       white      NA <NA>  <NA>
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

2. Selecione apenas as colunas `name` e `birth_year` e então ordene de forma decrescente pelo `birth_year`.

```
# Aninhando funções
arrange(select(sw, name, birth_year), desc(birth_year))

# Criando um objeto intermediário
sw_aux <- select(sw, name, birth_year)
arrange(sw_aux, desc(birth_year))
```

```
# Usando pipe
sw %>%
  select(name, birth_year) %>%
  arrange(desc(birth_year))
```

23.1.3 Filtrando linhas

Utilize a base `sw` nos exercícios a seguir.

1. Crie um objeto chamado `humanos` apenas com personagens que sejam humanos.

```
humanos <- filter(sw, species == "Human")

# O pipe
humanos <- sw %>%
  filter(species == "Human")
```

2. Crie um objeto chamado `altos_fortes` com personagens que tenham mais de 200 cm de altura e peso maior que 100 kg.

```
altos_fortes <- filter(sw, height > 200, mass > 100)
```

3. Retorne tabelas (`tibbles`) apenas com:

- a. Personagens humanos que nasceram antes de 100 anos antes da batalha de Yavin (`birth_year < 100`).

```
filter(sw, species == "Human", birth_year < 100)
```

- b. Personagens com cor `light` ou `red`.

```
filter(sw, skin_color == "light" | skin_color == "red")
```

- c. Personagens com massa maior que 100 kg, ordenados de forma decrescente por altura, mostrando apenas as colunas `name`, `mass` e `height`.

```
select(arrange(filter(sw, mass > 100), desc(height)), name, mass, height)

# usando o pipe
sw %>%
  filter(mass > 100) %>%
  arrange(desc(height)) %>%
  select(name, mass, height)
```

d. Personagens que sejam “Humano” ou “Droid”, e tenham uma altura maior que 170 cm.

```
filter(sw, species == "Human" | species == "Droid", height > 170)

# usando o pipe
sw %>%
  filter(species %in% c("Human", "Droid"), height > 170)
```

e. Personagens que não possuem informação tanto de altura quanto de massa, ou seja, possuem NA em ambas as colunas.

```
filter(sw, is.na(height), is.na(mass))
```

23.1.4 Modificando e criando novas colunas

1. Crie uma coluna chamada `dif_peso_altura` (diferença entre altura e peso) e salve a nova tabela em um objeto chamado `sw_dif`. Em seguida, filtre apenas os personagens que têm altura maior que o peso e ordene a tabela por ordem crescente de `dif_peso_altura`.

```
sw_dif <- mutate(sw, dif_peso_altura = height - mass)

arrange(filter(sw_dif, height > mass), dif_peso_altura)

# usando o pipe
sw_dif <- sw %>%
  mutate(dif_peso_altura = height - mass)

sw_dif %>%
  filter(height > mass) %>%
  arrange(dif_peso_altura)
```

2. Fazendo apenas uma chamada da função `mutate()`, crie as seguintes colunas novas na base `sw`:

- a. `indice_massa_altura = mass / height`
- b. `indice_massa_medio = mean(mass, na.rm = TRUE)`
- c. `indice_relativo = (indice_massa_altura - indice_massa_medio) / indice_massa_medio`
- d. `acima_media = ifelse(indice_massa_altura > indice_massa_medio, "sim", "não")`

```
mutate(sw,
  indice_massa_altura = mass/height,
  indice_massa_medio = mean(mass, na.rm = TRUE),
  indice_relativo = (indice_massa_altura - indice_massa_medio) / indice_massa_medio,
  acima_media = ifelse(indice_massa_altura > indice_massa_medio, "sim", "não"))
```

23.1.5 Sumarizando a base

Utilize a base `sw` nos exercícios a seguir.

1. Calcule a altura média e mediana dos personagens.

```
summarize(sw,
  media_altura = mean(height, na.rm=TRUE),
  mediana_altura = median(height, na.rm = TRUE))
```

```
## # A tibble: 1 x 2
##   media_altura mediana_altura
##       <dbl>         <int>
## 1         175.           180
```

2. Calcule a massa média dos personagens cuja altura é maior que 175 cm.

```
sw %>%
  filter(height > 175) %>%
  summarize(media_massa = mean(mass, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   media_massa
##       <dbl>
## 1         87.2
```

3. Apresente na mesma tabela a massa média dos personagens com altura menor que 175 cm e a massa média dos personagens com altura maior ou igual a 175 cm.

```
sw %>%
  mutate(alturas = ifelse(height < 175, "menor 175", "maior 175")) %>%
  filter(!is.na(height)) %>%
  group_by(alturas) %>%
  summarize(altura_media = mean(height, na.rm=TRUE))
)
```



```
## # A tibble: 2 x 2
##   alturas  altura_media
##   <chr>      <dbl>
## 1 maior 175      193.
## 2 menor 175      142.
```

4. Retorne tabelas (tibbles) apenas com:

a. A altura média dos personagens por espécie.

```
sw %>%
  group_by(species) %>%
  summarize(altura_media = mean(height, na.rm = TRUE))
```

```
## # A tibble: 38 x 2
##   species  altura_media
##   <chr>      <dbl>
## 1 Aleena      79
## 2 Besalisk    198
## 3 Cerean     198
## 4 Chagrian    196
## 5 Clawdite    168
## 6 Droid      131.
## 7 Dug        112
## 8 Ewok         88
## 9 Geonosian   183
## 10 Gungan     209.
## # i 28 more rows
```

b. A massa média e mediana dos personagens por espécie.

```
sw %>%
  filter(!is.na(mass)) %>%
  group_by(species) %>%
  summarize(massa_media = mean(mass, na.rm = TRUE),
            massa_mediana = median(mass, na.rm = TRUE))
```

```
## # A tibble: 32 x 3
##   species  massa_media massa_mediana
##   <chr>      <dbl>      <dbl>
## 1 Aleena      15         15
## 2 Besalisk   102        102
## 3 Cerean      82         82
## 4 Clawdite    55         55
```

```
## 5 Droid          69.8      53.5
## 6 Dug            40       40
## 7 Ewok           20       20
## 8 Geonosian      80       80
## 9 Gungan        74       74
## 10 Human         81.3     79
## # i 22 more rows
```

c. Apenas o nome dos personagens que participaram de mais de 2 filmes.

```
sw %>%
  filter(length(films) > 2) %>%
  select(name)
```

```
## # A tibble: 87 x 1
##   name
##   <chr>
## 1 Luke Skywalker
## 2 C-3PO
## 3 R2-D2
## 4 Darth Vader
## 5 Leia Organa
## 6 Owen Lars
## 7 Beru Whitesun Lars
## 8 R5-D4
## 9 Biggs Darklighter
## 10 Obi-Wan Kenobi
## # i 77 more rows
```