Programação em R
Parte II

Autor: Prof. Dr. Pedro Rafael Diniz Marinho

Universidade Federal da Paraíba
Departamento de Estatística da UFPB

Simplificando vs Preservando

Ao acessarmos subconjuntos de um objeto de tipo específico o comportamento a estrutura do objeto gerado poderá ser preservada ou simplificada.

Tabela: Simplificando ou preservando a estrutura de dados de um objeto.

Estrutura	Simplificando	Preservando
Vetor	`x[[1]]`	`x[1]`
Lista	`x[[1]]`	`x[1]`
Fator	`x[1:4, drop = T]`	`x[1:4]`
Array	`x[1, ]` ou `x[ ,1]`	`x[1, , drop = F]` ou `x[, 1, drop = F]`
Data Frame	`x[, 1]` ou `x[[1]]`	`x[, 1, drop = F]` ou `x[1]`

Simplificando vs Preservando

Exemplo (vetor atômico): A simplificação remove os nomes.

```
1: x <- c(a = 1, b = 2)
2: x[1]
#> a
#> 1
3: x[[1]]
#> [1] 1
```

Simplificando vs Preservando

Exemplo (lista): A simplificação retornará o objeto dentro da lista e não uma lista de um elemento.

```
1: y <- list(a = 1, b = 2)
2: str(y[1])
#> List of 1
#> $ a: num 1
3: str(y[[1]])
#> num 1
```

Simplificando vs Preservando

Exemplo (fator): A simplificação descarta os níveis não utilizados.

```
1: z <- factor(c("a", "b"))
2: z[1]
#> [1] a
#> Levels: a b
3: z[1, drop = TRUE]
#> [1] a
#> Levels: a
```

Simplificando vs Preservando

Exemplo (matriz ou array): Se os índices de ao menos uma das dimensões tem comprimento 1, cairá a dimensão.

```
1: a <- matrix(1:4, nrow = 2)
2: a[1, , drop = FALSE]
#>      [,1] [,2]
#> [1,]    1    3
3: a[1, ]
#> [1] 1 3
```

Simplificando vs Preservando

Exemplo (data frame): Se uma das dimensões é informada e a outra é deixada em branco, a estrutura será preservada.

```
1: df <- data.frame(a = 1:2, b = 1:2)
2: str(df[1])
#> 'data.frame':    2 obs. of 1 variable:
#> $ a: int 1 2
3: str(df[[1]])
#> int [1:2] 1 2
4: str(df[, "a", drop = FALSE])
#> 'data.frame': 2 obs. of 1 variable:
#> $ a: int 1 2
5: str(df[, "a"])
#> int [1:2] 1 2
```

Simplificando vs Preservando

Operadores `$` e `[[`

Fazer `x$y` é equivalente a fazer `x[["y", exact = FALSE]]`, porém há situações em que possamos preferir o uso do operador `[[` ao invés do operador `$`. O exemplo abaixo tenta mostrar isto.

Exemplo: Corra o código que segue:

```
1: var <- "cyl"
2: mtcars$var
#> NULL
# Instead use [[
3: mtcars[[var]]
#> [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4
#> 8 8 8 8 4 4 4 8 6 8 4
```

Simplificando vs Preservando

```
1: x <- list(abc = 1)
2: x$a
#> [1] 1
3: x[["a"]]
#> NULL
```

Nota: Perceba que o operador `$` permite que possamos completar parcialmente o nome de um elemento. Isto só poderá ser permitido com o operador `[[` caso viermos à passarmos como argumento `exact = FALSE` que por padrão é igual à `TRUE`.

Atribuição

Todos os operadores de subconjuntos estudados anteriormente podem ser combinados com atribuição para modificar valores selecionados do vetor de entrada. O exemplo que segue tenta mostrar um bom resumo sobre atribuição. Tente correr o exemplo e compreender os detalhes.

Exemplo: Corra o código que segue:

```
x <- 1:5
x[c(1, 2)] <- 2:3
x
#> [1] 2 3 3 4 5
```

Atribuição

```
# The length of the LHS needs to match the RHS
x[-1] <- 4:1
x
#> [1] 2 4 3 2 1

# Note that there's no checking for duplicate indices
x[c(1, 1)] <- 2:3
x
#> [1] 3 4 3 2 1
```

Atribuição

```
# You can't combine integer indices with NA
x[c(1, NA)] <- c(1, 2)

#> Error: NAs are not allowed in subscripted assignments
# But you can combine logical indices with NA
# (where they're treated as false).
x[c(T, F, NA)] <- 1
x
#> [1] 1 4 3 1 1
```

Atribuição

```
# This is mostly useful when conditionally modifying
# vectors
df <- data.frame(a = c(1, 10, NA))
df$a[df$a < 5] <- 0
df$a
#> [1] 0 10 NA
```

Nota: A função `lapply()` poderá ser bastante útil para retornar uma lista de mesmo tamanho do seu primeiro argumento (vetor, matriz, data frame, etc), em que cada elemento da lista retornada é o resultado da aplicação de uma função passada como segundo argumento à função `lapply()` aos elementos do objeto passado como primeiro argumento.

Atribuição

Exercício: Corra o código que segue. Discutam entre si o funcionamento da função `lapply()` e observe outros detalhes.

```
1: mtcars[] <- lapply(X = mtcars, FUN = as.integer)
2: mtcars
3: mtcars <- lapply(X = mtcars, FUN = as.integer)
4: mtcars
```

Correspondência

Suponha que tenhamos um vetor de notas inteiras e um uma descrição da propriedade da nota, como mostra o exemplo que segue:

Exemplo: Corra o código abaixo e preste atenção no que está sendo feito. Isto poderá ser útil.

Correspondência

Suponha que tenhamos um vetor de notas inteiras e um uma descrição da propriedade da nota, como mostra o exemplo que segue:

Exemplo: Corra o código abaixo e preste atenção no que está sendo feito. Isto poderá ser útil.

```
1: grades <- c(1, 2, 2, 3, 1)
2: info <- data.frame(grade = 3:1, desc = c("Excellent",
3:                     "Good", "Poor"), fail = c(F, F, T))
4: # Using match
5: id <- match(grades, info$grade)
6: info[id, ]
```

Correspondência

```
7: # Using rownames
8: rownames(info) <- info$grade
9: info[as.character(grades), ]
```

Exercício: Estude a documentação das funções `match()` e `%in%` (faça `?"%in%"`).

Interação com o usurário

A linguagem `R` possui diversas funções que têm que permitem o programa interagir com o utilizado como é o caso da função `print()` que poderá ser utilizada para escrever o conteúdo de qualquer objeto.

Exemplo: Corra o código que segue:

```
1: # O código na linha 3 não irá imprimir nada
2: # muito embora as sequências serão executadas.
3: for (i in 1:i) 1:i
4: # Imprimindo no prompt de comando.
5: for (i in 1:i) print(i)
```

Interação com o usurário

Uma outra função muito eficiente para escrever objetos e que possibilita receber um número qualquer de argumentos transformando seus argumentos em strings concatenadas é a função `cat()`.

Exemplo: Execute o código logo abaixo:

```
1: nota <- 7
2: cat("Joãozinho: Professor,\nqual foi minha nota?
3: Professor: Sua nota foi", nota, "Joãozinho\t...",
4:      sep = " ")
```

Exercício: Leia a documentação as funções `print()` e `cat()`.

Interação com o usurário

Também é possível que o usuário de um código `R` insira os dados durante a execução do código. Para isto é normalmente utilizado a função `scan()`.

```
1: x <- scan(n = 3)
#> 1: 1 2 3
#> Read 3 items
2: y <- scan(what = character())
3: "Como estatístico" "devo"
4: "gostar de " "programar ..."
5: cat(y)
#> Como estatístico devo gostar de  programar ...
```

Instruções Condicionais

É muito comum em qualquer código fazermos uso de instruções condicionais que permitem o programador explicitar diferentes alternativas que podem vir a serem executadas dependendo de alguma condição testada na altura da execução do programa pelo interpretador da linguagem.

A instrução `if()` permite que uma condição seja avaliada e caso sera verdadeira, o bloco correspondente à instrução é executado. Em caso que a instrução seja falsa, é possível fazer com que outro bloco de instruções seja executado.

Exemplo: Corra o código abaixo:

Instruções Condicionais

```
1: x <- 7
2: if(x > 0)
3: {
4:     cat('x é positivo.\n')
5:     y <- z / x
6: } else {
7:           cat('x não é positivo! \n')
8:           y <- z
9: }
```

Nota: É importante prestar atenção à indentação do código. Dessa forma, conseguiremos ressaltar e entender melhor a estrutura do código de forma a aumentar a legibilidade do código.

Instruções Condicionais

O código acima não seria tão legível se fosse apresentado na forma:

```
1: x <- 7
2: if(x > 0)
3: {
4: cat('x é positivo.\n')
5: y <- z / x
6: } else {
7: cat('x não é positivo! \n')
8: y <- z
9: }
```

Observação: Note que a cláusula `else` é opcional.

Instruções Condicionais

Também é possível aninhar diversas instruções `if()`. Observe o exemplo abaixo.

Exemplo:

```
1: idade <- 30
2: if (idade < 18) {
3:     grupo <- 1
4: } else if (idade < 35) {
5:     grupo <- 2
6: } else if (idade < 65) {
7:     grupo <- 3
8: } else {
9:     grupo <- 4
10:}
11:grupo
```

Instruções Condicionais

Notas:

1. Chamamos bloco de instruções o conjunto de código entre $\{\}$;
2. Muito embora os blocos de instruções no exemplo anterior apresente apenas uma instrução no interior do bloco, estes poderiam conter diversas outras instruções.

Instruções Condicionais

Observação: Caso a instrução `if` possua ser escrita em apenas uma única linha, as chaves de bloco poderão ser omitidas.

Exercício:

```
1: nota <- 7
2: if(nota < 7) cat ("Aluno: =(") else cat("Aluno: =)")
```

Nota: Como foi omitido o bloco na instrução `if`, note que a instrução `else` também teve que ser iniciada na linha 2 do código acima.

Instruções Condicionais

Uma função `ifelse()` também poderá ser utilizada para se tra balhar com o controle de fluxo da linguagem `R`. Esta função permite apenas três argumentos em que o primeiro é uma condição que será testada. Em caso verdadeiro, o resultado da expressão do segundo argumento será retornado. Caso contrário, será retornado a expressão passada como terceiro argumento à função.

Exemplo: Corra o código abaixo:

```
1: nota <- 7
2: ifelse(nota >= 7, "=)", "=(")
#> [1] "=)"
```

Instruções Condicionais

Exemplo: Observe um outro exemplo

```
1: set.seed(2)
2: x <- rnorm(n = 5,  mean = 0, sd = 1)
3: sig <- ifelse(x < 0, "-", "+")
4: sig
#> "-" "+" "+" "-" "-"
```

Nota: A linha 1 do código acima fixa um valor de semente para o gerador de números pseudo-aleatório que é utilizado na geração de números aleatórios de uma determinada distribuição de probabilidade. A linha 2 gera 5 números pseudo-aleatórios com distribuição normal centrada no zero e variância 1. Falaremos mais a frente com os devidos detalhes sobre geração de números pseudo-aleatórios.

Instruções Condicionais

Uma outra instrução condicional em que poderá ser utilizada para escolher uma de várias alternativas possíveis é a instrução `switch()`. Tal função consiste em uma série de argumentos que a depender do primeiro argumento algum dos outros argumentos será retornado.

Nota: O primeiro argumento poderá ser um número ou uma string ou qualquer expressão que resulte um número ou caractere.

```
1: set.seed(0) # Fixando semente do gerador.
2: expressao <- 1
3: vetor_normal <- rnorm(10) # Faça ?rnorm para detalhes.
4: switch(EXPR = expressao, round(mean(vetor_normal),1),
5: round(median(vetor_normal)))
#> 0.4
```

Exemplo: Replique o exemplo anterior fazendo `expressao = 3`. Nesse caso, nada será retornado, isto é, `NULL` será o tipo de retorno.

Exemplo: Considere o exemplo que segue em que o argumento `EXPR` da instrução `switch()` é uma string.

```
1: semaforo <- "verde"
2: switch(EXPR = semaforo, verde = "siga",
3:        amarelo = "atenção",
4:        vermelho = "pare")
```

Instruções Condicionais

Importante: As instruções poderão ser maiores, isto é, conter mais de uma linha. Dessa forma, o programador deverá utilizar um bloco de instruções ({}).

Exercício: Escreva um programa que calcule o imposto pago por mulheres e por homens, sabendo que as mulheres pagam 10% e que os homens pagam 5% a mais do que as mulheres.

Solução:

```
1: # Lendo entrada do teclado.
2: meu_salario <- function(){
3: salario <- as.numeric(readline(prompt =
4:                                "Entre com um salário: "))
5: sexo <- tolower(readline(prompt =
6:                          "Informe o sexo (m ou f): "))
7: switch(sexo,
8:        "m" = {
9:                  imposto = 0.15;
10:                 cat("O salário a ser pago é ",
11:                     salario - salario*imposto)
12:               },
```

Instruções Condicionais

```
13:       "f" = {
14:                   imposto = 0.1;
15:                   cat("O salário a ser pago é ",
16:                        salario - salario*imposto)
17:             }
18:       )
19:}
```

Exercício: Descreva para que servem as funções `tolower()`, `toupper()` e `readline()`. Apresente exemplos do uso de cada uma delas.

Loop

Loop

A linguagem `R` possui diversas instruções de laço que nos permite repetir blocos de instruções um número predefinido de vezes ou até que uma condição não seja mais verdadeira.

Uma das instruções bastante utilizada para repetição iterativa de um bloco de instruções é a instrução `while()`. A forma geral da sintaxe da instrução `while()` na linguagem `R` é:

Sintaxe:

```
while (condição boolena)
{
  bloco de insturções a se repetir.
}
```

Loop

Exemplo: Corra o código abaixo:

```
1: i <- 1
2: while(i<=7)
3: {
4:     cat("i",i, " = ", i, "\n", sep="")
5:     i <- i + 1
6: }
#> i1 =  1
#> i2 =  2
#> i3 =  3
#> i4 =  4
#> i5 =  5
#> i6 =  6
#> i7 =  7
```

Loop

Exercício: Escreva um programa que coloque na tela a tabuada de 7 utilizando a instrução `while()`.

Loop

Exercício: Escreva um programa que coloque na tela a tabuada de 7 utilizando a instrução `while()`.

```
1: i <- 1
2: while(i<=10)
3: {
4:     cat("7 x ", i, " = ", 7 * i, "\n")
5:     i <- i + 1 # incrementando i.
6: }
#> 7 x 1 = 7
#> 7 x 2 = 14
#> 7 x 3 = 21
#> 7 x 4 = 28
#> ...
```

Loop

Importante

Cuidado com loops infinitos. Uma instrução de loop deverá sempre alcançar um fim. Dessa forma, garanta que a condição do seu laço em algum momento se torne falsa. Esse erro é muito comum quando esquecemos de incrementar ou decrementar a variável de controle do laço.

Exemplo: Exemplo de loop infinito.

```
1: i <- 1
2: while(i<=10)
3: {
4:     cat("7 x ", i, " = ", 7 * i, "\n")
5: }
```

Loop

Exemplo: Imprimindo o conteúdo de `u` enquanto a expressão `u < 0.5` permanecer verdadeira.

```
1: u <- runif(n = 1, min = 0, max = 1)
2: while (u < 0.5)
3: {
4:   cat("u = ", u, "\n", sep="")
5:   u <- runif(n = 1, min = 0, max = 1)
6: }
```

Loop

Uma outra instrução de repetição bastante útil é a `repeat`. Tal instrução é executada indefinidamente até que alguma condição force sua interrupção.

Exemplo: Corra o código que segue:

```
1: texto <- c()
2: repeat {
3:   fr <- readline(prompt = "Introduza uma frase?
4:                     (frase vazia termina) ")
5:   if (fr == ‘’) break else texto <- c(texto,fr)
6:}
```

Importante

A instrução `break` faz com que uma instrução de repetição seja finalizada. Dessa forma, no exemplo anterior, temos que ao ser verdade a condição da instrução condicional `if()` a instrução `break` será avaliada, forçando assim o término da instrução `repeat`.

Nota: A instrução `break` poderá ser utilizada em qualquer instrução apresentadas (`while()` e `repeat`) bem como na instrução `for()` que será apresentada mais a frente.

Loop

Existe também uma outra instrução bastante útil quando utilizada em instruções de repetição que é a instrução `next`. Quando a instrução `next` é avaliada, todas as instruções que seguem são ignoradas e a instrução de repetição irá para a próxima iteração.

Exemplo: Corra o código:

```
1: vetor <- c()
2: repeat {
3:    nro <- as.numeric(readline(prompt =
4:           "Introduza um nro positivo ?
5:            (zero termina) "))
6:    if (nro < 0) next
7:    if (nro == 0) break
8:    vetor <- c(vetor,nro)
}
```

Loop

Assim como em quase todas as linguagens de programação, não poderia faltar a instrução `for()` em que uma variável de controle irá percorrer um conjunto.

Sintaxe:

```
for (var in conjunto)
{
   meu bloco de instruções
}
```

Exercício: Utilizando a instrução de repetição `for` construa um pequeno programa que com base em um vetor de valores no intervalo $[0, 1]$, some apenas os valores maiores que 0.7. **Dica**: (Para economizar tempo, faça `vetor <- runif(n = 10, min = 0, max = 1)` para gerar o vetor é observada de uma v.a. X_i, tal que $X_i \sim \mathcal{U}(0, 1)$, $i = 1, \ldots, 10$.)

Uma possível solução:

```
1: set.seed(0) # Fixando a semente do gerador.
2: vetor <- runif(n = 10, min = 0, max = 1)
3: soma <- 0
4:
5: for (indice in vetor)
6: {
7:    if (indice > 0.7)
8:    valor <- indice
9:    else valor <- 0
10:   soma <- soma + valor
11:}
12:
13: soma
#>  [1] 3.64797
```

Loop

Muito Importante

Há diversas situações em `R` em que podemos evitar o uso de instruções de repetições. Por exemplo, a situação do exercício acima é uma delas. Evitar estas situações ajudará muito a aumentar a eficiência do código.

Exercício: Refaça o exercício anterior sem utilizar nenhuma instrução de loop.

Muito Importante

Há diversas situações em R em que podemos evitar o uso de instruções de repetições. Por exemplo, a situação do exercício acima é uma delas. Evitar estas situações ajudará muito a aumentar a eficiência do código.

Exercício: Refaça o exercício anterior sem utilizar nenhuma instrução de loop.

Solução:

```
1: set.seed(0)
2: vetor <- runif(n = 10, min = 0, max = 1)
3: soma <- sum(vetor[vetor > 0.7])
4: soma
#> [1] 3.64797
```

Loop

Exercício: Dê uma consultada na documentação da função `Sys.time()`. Utilize esta função para obter o tempo de execução, em segundos, das funções implementadas nos dois exercícios anteriores.

Nota: Muitas vezes é suficiente utilizar a função `Sys.time()` para se ter uma ideia do tempo de execução de um programa ou de um trecho. Porém, existe funções específicas para mensurar o tempo de forma mais precisa. Estas funções propões realizar **benchmark** mais precisos.

O que é benchmark?

Resposta: Benchmark é o processo de executar um programa ou conjunto de operações a fim de avaliar o desempenho relativo de uma função ou objeto. O desempenho é testado por meio de um conjunto de testes padrões e ensaios sobre a função ou objeto.

Uma das formas eficientes de fazer benchmark na linguagem `R` é utilizar a função `microbenchmark()` do pacote **microbenchmark** que poderá ser instalado fazendo

```
install.packages("microbenchmark")
```

no prompt de comando da linguagem.

Loop

A função `microbenchmark()` fornece o desempenho com nanosegundos, em que $1\,\mathrm{s} = 1 \times 10^9\,\mathrm{ns}$ (nanosegundos).

Nota: É possível alterar a unidade de mensuração de tempo modificando o argumento `unit` da função `microbenchmark()`. Alguns argumentos possíveis são:

- ns: nanosegundos;
- ms: milissegundo
- s: segundos;
- Outros argumentos como unidade de frequência são também suportados, como por exemplo: hz, khz, mhz.

Loop

Exercício: Leia a documentação da função `microbenchmark()`. Reproduza os dois exemplos anteriores e realize o benchmarks nos dois casos.

Loop

Exercício: Leia a documentação da função `microbenchmark()`. Reproduza os dois exemplos anteriores e realize o benchmarks nos dois casos.

Solução:

```
1: library(microbenchmark)
2:
3: # Exemplo 1:
4:
5: set.seed(0)
6: vetor <- runif(n = 10, min = 0, max = 1)
7: soma <- 0
```

Loop

```
8:  microbenchmark(
9:  for (indice in vetor)
10: {
11:     if (indice > 0.7)
12:         valor <- indice
13:     else valor <- 0
14:     soma <- soma + valor
15: }, unit = "s")

16: # Exemplo 2:
17:
18: set.seed(0)
19: vetor <- runif(n = 10, min = 0, max = 1)
20: microbenchmark(soma <- sum(vetor[vetor > 0.7]),
21:                unit = "s")
```

Nota: O valor médio do benchmark é sempre uma boa estimativa para o desempenho de uma função ou trecho de código, uma vez que é a média de diversos benchmarks (por padrão 100). Para alterar o número de testes, é possível modificar o parâmetro `times` da função `microbenchmark()` para um outro inteiro.

Nota: O valor médio do benchmark é sempre uma boa estimativa para o desempenho de uma função ou trecho de código, uma vez que é a média de diversos benchmarks (por padrão 100). Para alterar o número de testes, é possível modificar o parâmetro `times` da função `microbenchmark()` para um outro inteiro.

Lição:

Loop

Nota: O valor médio do benchmark é sempre uma boa estimativa para o desempenho de uma função ou trecho de código, uma vez que é a média de diversos benchmarks (por padrão 100). Para alterar o número de testes, é possível modificar o parâmetro `times` da função `microbenchmark()` para um outro inteiro.

Lição: Loops são estruturas computacionalmente intensivas na linguagem `R` e que muitas vezes podemos evitar utilizando a vetorização das operações que a linguagem disponibiliza.

Loop

Tentei de tudo, e não achei formas de evitar o loop ...

Loop

Tentei de tudo, e não achei formas de evitar o loop ...

Sem problemas, use sem peso na consciência. De fator eles são verdadeiramente úteis e muitas vezes é realmente inevitável não utilizar loops ...

Loop

Sem problemas, use sem peso na consciência. De fator eles são verdadeiramente úteis e muitas vezes é realmente inevitável não utilizar loops ...

Loop

Exercício: Escreva o trecho de código abaixo utilizando a instrução `while()`.

```
1: for (i in 1:20)
2: {
3:   if (i == 10) next
4:   else cat("i = ", i, "\n", sep = "")
5: }
```

Loop

Exercício: Escreva um programa em `R` utilizando as instruções de loop vistas anteriormente de modo a fornecer a seguinte estrutura a depender do valor de n.

```
Para n = 1      Para n = 2      Para n = 3  ...
*               *               *
                **              **
                                ***
```

Uma solução possível:

```
1:  n <- 5
2:  for (i in 1:n)
3:  {
4:    j <- 1
5:    while (j <= i)
6:    {
7:       cat("*")
8:       j <- j + 1
9:    }
10:   cat("\n")
11: }
```

Loop

Exercício: Refaça o exemplo anterior de tal forma que a estrutura obtida seja:

```
Para n = 1      Para n = 2      Para n = 3  ...
A               A               A
                BB              BB
                                CCC
```

Um solução possível:

```
1:  n <- 5
2:  for (i in 1:n)
3:  {
4:    j <- 1
5:    while (j <= i)
6:    {
7:       cat(LETTERS[i])
8:       j <- j + 1
9:    }
10:   cat("\n")
11: }
```

Loop

Exercício: Suponha que um professor possa fazer um número qualquer de avaliações e deseja construir um programa em R que receba as notas do aluno. O programa deverá informar se o aluno foi aprovado, reprovado ou irá para prova final. Em caso do aluno ir para prova final, o programa deverá informar qual a nota a partir da qual o aluno irá assumir o status de aprovado na disciplina.

Regra para avaliação:

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;
2. O aluno que tiver nota interior à 4.0 assumirá o status de reprovado sem direito à avaliação final;

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;
2. O aluno que tiver nota interior à 4.0 assumirá o status de reprovado sem direito à avaliação final;
3. Alunos com notas no intervalo $[4.0, 7.0)$ terá direito à fazer a prova final.

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;
2. O aluno que tiver nota interior à 4.0 assumirá o status de reprovado sem direito à avaliação final;
3. Alunos com notas no intervalo $[4.0, 7.0)$ terá direito à fazer a prova final.
4. A média final será uma média ponderada que terá peso 4 para a nota final e peso 6 para a média das primeiras notas. O aluno com média final maior ou igual à 5.0 será considerado aprovado na disciplina.

Uma solução:

```
1: aluno <- function()
2: {
3:   n_avaliacoes <- as.numeric(readline(prompt =
4:                     "Quantas avaliações?: "))
5:   i <- 1
6:   v_provas <- NULL
7:   for (i in 1:n_avaliacoes)
8:      v_provas[i] <- as.numeric(readline(prompt =
9:                       paste("Nota ", i, " = ",
10:                      sep = "")))
11:   media <- mean(v_provas)
12:
13:   if (media >= 7) situacao <- "Aprovado(a)"
14:   else if (media < 4) situacao <- "Reprovado(a)."
```

Loop

```
15:    else{
16:
17:       cat("O aluno(a) irá para final e precisará ter
18:       tirado nota maior ou igual à ", (50-6*media)/4,
19:       " para passar.\n")
20:
21:       final <- as.numeric(readline(prompt =
22:                "\nQual a nota da final?: "))
23:
24:       if (final >= (50-6*media)/4)
25:          situacao <- "Aprovado(a)"
26:    }
27:
28:    list(media = round(media,2), situacao = situacao)
29: }
```

Nota: Soluções interativa, em geral, devem ser evitadas. Por exemplo, no problema acima poderíamos ter informado antecipadamente um vetor com as notas. Em geral, nas simulações que precisamos realizar, não desejaremos essa interação. No caso das simulações, desejamos colocar o código para ser interpretado mesmo quando não estamos com contato físico com o computador.

Loop

Exercício: Utilizando as instruções de loop, some todos elementos de uma matriz. **Dica**: Gere aleatoriamente uma matriz de dimensões 5 por 2.

Solução:

```
1: set.seed(0)
2: matriz <- matrix(runif(10), 5, 2)
3: soma <- 0
4: for (coluna in 1:ncol(matriz))
5:   for (linha in 1:nrow(matriz))
6:     soma <- soma + matriz[linha, coluna]
7: soma
#> [1] 6.35005
```

Loop

Tornando ainda a falar sobre evitar loops, lembre-se de muitas funções em R são vetorizáveis. No caso do exercício anterior temos, poderíamos ter solucionado o problema fazendo:

Exemplo: Corra o código abaixo:

```
1: set.seed(0) # Fixando a semente do gerador.
2: matriz <- matrix(runif(10), 5, 2)
3: sum(matriz) # Dois loops evitados.
#> [1] 6.35005
```

Muito mais simples e eficiente resolver este problema da forma acima.

Loop

Exercício: Utilizando as instruções de loop, retorne um vetor com as somas das colunas de uma matriz. **Dica**: Gere aleatoriamente uma matriz de dimensões 10 por 10.

Loop

Exercício: Utilizando as instruções de loop, retorne um vetor com as somas das colunas de uma matriz. **Dica**: Gere aleatoriamente uma matriz de dimensões 10 por 10.

```
1:  matriz <- matrix(runif(100, min = 0, max = 1),
2:                   ncol = 10, nrow = 10)
3:  soma <- 0
4:  vetor <- NULL
5:  for (coluna in 1:ncol(m))
6:  {
7:    for (linha in 1:nrow(m))
8:    {
9:       soma <- soma + matriz[linha, coluna]
10:   } vetor[coluna] <- soma
11: }
```

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

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

Informações sobre os parâmetros:

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

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

Informações sobre os parâmetros:

- `X`: Matriz ou array;

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

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

Informações sobre os parâmetros:

- `X`: Matriz ou array;
- `MARGIN`: 1 e 2 indicam linha e coluna, respectivamente;

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

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

Informações sobre os parâmetros:

- `X`: Matriz ou array;
- `MARGIN`: 1 e 2 indicam linha e coluna, respectivamente;
- `FUN`: Função que queremos aplicar.

Loop

Exemplo: O problema anterior poderia ser resolvido de forma mais eficiente como segue:

```
1: matriz <- matrix(runif(100, min = 0, max = 1),
2:                  ncol = 10, nrow = 10)
3: apply(X = matriz, NARGIN = 2, FUN = min)
```

Loop

Uma outra função bastante útil na linguagem `R` é a função `tapply()`. Tal função é bastante semelhante à função `apply()`, porém com a vantagem de ser possível aplicar uma função à um subconjunto dos dados.

Exemplo:

```
1: data(warpbreaks)
2: head(warpbreaks) # Faça ?head para mais detalhes.
3:
4: tapply(X = warpbreaks$breaks, INDEX =
5:         warpbreaks[, -1], FUN = sum)
#>    tension
#> wool   L   M   H
#> A    401 216 221
#> B    254 259 169
```

Loop

Uma outra função bastante útil para programadores de R é a função `sapply()` que permite aplicar uma função a todos os elementos de um vetor ou lista.

Exemplo:

```
1: lista <- list(t1 = sample(15), t2 =
2:                 c(7.7,3.4,4.7,8.02), f3 = runif(n = 100))
3: sapply(lista, quantile)
#>         t1    t2         f3
#> 0%     1.0 3.400 0.01339033
#> 25%    4.5 4.375 0.33342987
#> 50%    8.0 6.200 0.48781071
#> 75%   11.5 7.780 0.76719336
#> 100% 15.0 8.020 0.99268406
```

Loop

Exercício: Estude a documentação da função `quantile()`.

Exercício: A linguagem `R` também apresenta a função `lapply()`. Estude a documentação desta função. Cite a principal diferença entre as funções. Apresente exemplos.

Existe ainda uma versão multivariada das funções `lapply()` e `sapply()` que é a função `mapply()`. As funções `lapply()` e `sapply()` atuam somente sobre os elementos de uma única lista. Porém, a função `mapply()` a função agirá sobre o primeiro elemento de cada um dos argumentos, em seguida ao segundo elemento e assim sucessivamente.

Loop

Exercício: Corra o código abaixo. Depois, explique o funcionamento da função `mapply()`.

```
1: mapply(rep, 1:4, 4:1)
#> [[1]]
#> [1] 1 1 1 1
#>
#> [[2]]
#> [1] 2 2 2
#>
#> [[3]]
#> [1] 3 3
#>
#> [[4]]
#> [1] 4
```

Loop

Exemplo: Corra o código:

```
1: l1 <- list(a = LETTERS[c(4,6,12,6)],
2:            b = LETTERS[c(1,5,21,1)])
3: l2 <- list(c = LETTERS[c(4,14,22,20)],
4:            d = LETTERS[c(15,15,1,15)])
5: mapply(paste, l1$a, l1$b, l2$c, l2$d)
#> "D A D O" "F E N O" "L U V A" "F A T O"
```

Exercício: Walter está jogando um jogo com dois dados de 6 lados equiprováveis. O jogo consiste em lançar ambos os dados e caso o resultado seja divisível por 3, ele ganhará $\$6,00$ dólares, caso contrário, ele perderá $\$3,00$ dólares. Simule 100 mil repetições desse experimento e obtenha o valor esperado de um jogador.

Loop

Exercício: Walter está jogando um jogo com dois dados de 6 lados equiprováveis. O jogo consiste em lançar ambos os dados e caso o resultado seja divisível por 3, ele ganhará $\$6,00$ dólares, caso contrário, ele perderá $\$3,00$ dólares. Simule 100 mil repetições desse experimento e obtenha o valor esperado de um jogador.

Dica: Utilize a função `sample()` para realizar o sorteio das faces de um dado, isto é, sortear números de 1 à 6 para cada um dos dados. Consulte a documentação para maiores detalhes.

Loop

Exercício: Suponha que tenhamos uma urna com bolas de mesmo tamanho enumeradas de 1 à 100. Considere o experimento aleatório de retirar uma bola da urna e observar o seu número até obtermos a bola com número desejado. Nesse experimento, será considerado reposição, isto é, caso não tenha sido observado a numeração desejada, a bola será devolvida à urna. Simule no computador 10 mil repetições desse experimento e obtenha uma média das retiradas necessárias para se obter o número desejado.

Loop

Exercício: Um dono de cassino estuda disponibilizar um novo jogo e solicita uma consultoria estatística para saber se o jogo será viável para o cassino, isto é, se o valor esperado em dólares do lucro obtido será positivo. **Pergunta-se**: Qual o valor esperado do lucro do cassino sabendo que o jogador pagará $\$10,00$ dólares para ter direito a jogar e em cada lançamento o jogador irá lucrar $\$1,50$ dólares.

Loop

Exercício: Um dono de cassino estuda disponibilizar um novo jogo e solicita uma consultoria estatística para saber se o jogo será viável para o cassino, isto é, se o valor esperado em dólares do lucro obtido será positivo. **Pergunta-se**: Qual o valor esperado do lucro do cassino sabendo que o jogador pagará $\$10,00$ dólares para ter direito a jogar e em cada lançamento o jogador irá lucrar $\$1,50$ dólares.

1. Dois dados são lançados e caso a soma for 5, 6, 7, 8 ou 9 o jogo termina imediatamente.

Loop

Exercício: Um dono de cassino estuda disponibilizar um novo jogo e solicita uma consultoria estatística para saber se o jogo será viável para o cassino, isto é, se o valor esperado em dólares do lucro obtido será positivo. **Pergunta-se**: Qual o valor esperado do lucro do cassino sabendo que o jogador pagará $\$10,00$ dólares para ter direito a jogar e em cada lançamento o jogador irá lucrar $\$1,50$ dólares.

1. Dois dados são lançados e caso a soma for 5, 6, 7, 8 ou 9 o jogo termina imediatamente.
2. Se nenhum dos resultamos acima não forem obtido, o jogador continua lançando ambos os dados até obter uma soma igual à 11 ou 12.

Dica: Realize uma simulação considerando 100 mil jogos e obtenha o valor médio de lançamentos e a probabilidade (aproximada) de o jogador jogar apenas uma partida.

Exercício: Seja $X_1, \ldots, X_n$ uma amostra aleatória da variável aleatória $X \sim \mathcal{N}(\mu, \sigma^2)$. Além disso, considere os estimadores $\hat{\sigma}^2$ e S^2 apresentados abaixo:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2 \quad (1)$$

e

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2.$$

Nota: Uma forma de compararmos dois estimadores é através do **Erro Quadrático Médio** (EQM) dos estimadores. Dessa forma, sejam $\hat{\theta}_1$ e $\hat{\theta}_2$ dois estimadores para um parâmetro θ **desconhecido**. Diremos que $\hat{\theta}_1$ é **melhor** que $\hat{\theta}_2$ se

$$EQM(\hat{\theta}_1) \leq EQM(\hat{\theta}_2),$$

em que $EQM(\hat{\theta}) = E[(\hat{\theta} - \theta)^2]$.

Lembre-se também que o **viés** de um estimador $\hat{\theta}$ do parâmetro θ é dado por

$$B(\hat{\theta}) = E(\hat{\theta}) - \theta.$$

Se $B(\hat{\theta}) = 0$ diremos que $\hat{\theta}$ é um **estimador não viesado** para θ, isto é, $E(\hat{\theta}) = \theta$.

Loop

Vamos obter os resultados por meio de **simulação**. Entenda **simulação** como uma simplificação da realidade em um ambiente virtual.

Dessa forma, serão gerados diversos cenários, onde em cada um deles serão gerados uma sequência de observações de uma variável aleatória $X \sim \mathcal{N}(0, 1)$. A referida sequência de observações formam um conjunto de dados distinto em cada iteração. Esse conjunto de dados simula um conjunto de dados real que poderíamos obter em uma situação prática no mundo real. Muito provavelmente, tais conjuntos de dados são distintos no que diz respeito a sequência de valores observados como ocorre na realidade.

Observação

Como os estimadores $\hat{\sigma}^2$ e S^2 se propõe estimar σ^2 (parâmetro de uma distribuição gaussiana), devemos gerar observações de variáveis aleatórias com distribuição gaussiana.

Para termos resultados razoáveis nas nossas simulações, é importante considerarmos um número elevado de repetições do experimento.

Observação

Como os estimadores $\hat{\sigma}^2$ e S^2 se propõe estimar σ^2 (parâmetro de uma distribuição gaussiana), devemos gerar observações de variáveis aleatórias com distribuição gaussiana.

Para termos resultados razoáveis nas nossas simulações, é importante considerarmos um número elevado de repetições do experimento.

Dica

Não há um número exato de repetições que precisaremos considerar. De toda forma, é razoável considerar um número de iterações no intervalo $[10000, +\infty)$.

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

1. Gere uma amostra aleatória de tamanho n, tal que $X_i \sim \mathcal{N}(0, 1) \, \forall i$;

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

1. Gere uma amostra aleatória de tamanho n, tal que $X_i \sim \mathcal{N}(0, 1) \, \forall i$;
2. Obtenha as estimativas $\hat{\sigma}^2$ e S^2 e guarde as estimativas;

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

1. Gere uma amostra aleatória de tamanho n, tal que $X_i \sim \mathcal{N}(0,1)\, \forall i$;
2. Obtenha as estimativas $\hat{\sigma}^2$ e S^2 e guarde as estimativas;
3. Calcule o $EQM(\hat{\sigma}^2)$, $EQM(S^2)$, $B(\hat{\sigma}^2)$ e $B(S^2)$.

Ao final das 20 mil iterações, tire uma média de cada uma das estimativas no passo **2** e das estimativas do passo **3**.

Loop

Construa uma tabela com a média das estimativas obtidas por cada um dos estimadores ($\hat{\sigma}^2$ e S^2 obtidos no passo **2**) e com as estimativas obtidas no passo **3** para os tamanhos de amostra $n = 20, 60, 100$ e 1000. Discuta qual o melhor estimador com base nos resultados assintóticos obtidos pela simulação.

Loop

Construa uma tabela com a média das estimativas obtidas por cada um dos estimadores ($\hat{\sigma}^2$ e S^2 obtidos no passo **2**) e com as estimativas obtidas no passo **3** para os tamanhos de amostra $n = 20, 60, 100$ e 1000. Discuta qual o melhor estimador com base nos resultados assintóticos obtidos pela simulação.

Nota: Você talvez acaba de fazer sua primeira **simulação de Monte Carlo**. Mais adiante no curso apresentaremos mais detalhes a respeito de simulações de Monte Carlo.

Loop

Construa uma tabela com a média das estimativas obtidas por cada um dos estimadores ($\hat{\sigma}^2$ e S^2 obtidos no passo **2**) e com as estimativas obtidas no passo **3** para os tamanhos de amostra $n = 20, 60, 100$ e 1000. Discuta qual o melhor estimador com base nos resultados assintóticos obtidos pela simulação.

Nota: Você talvez acaba de fazer sua primeira **simulação de Monte Carlo**. Mais adiante no curso apresentaremos mais detalhes a respeito de simulações de Monte Carlo.

Observação: Avaliar assintoticamente um estimador quer dizer que iremos avaliar o estimador quando $n \to +\infty$.

Funções

Apesar da linguagem R possuir diversas funções disponíveis em uma infinidade de pacotes oficiais e não oficiais para os mais diversos fins, sempre precisaremos construir nossas funções. Sempre há algo que não tenha sido implementado ou as vezes precisaremos modificar um pouco a estrutura de uma função já existente implementada por você ou outro programador.

Funções

Apesar da linguagem R possuir diversas funções disponíveis em uma infinidade de pacotes oficiais e não oficiais para os mais diversos fins, sempre precisaremos construir nossas funções. Sempre há algo que não tenha sido implementado ou as vezes precisaremos modificar um pouco a estrutura de uma função já existente implementada por você ou outro programador.

Nota: Na maioria das vezes, sempre é útil dividir nosso programa em funções que realizam tarefas específicas. Estas funções devem ser implementadas da forma mais geral possível para que possam ser reaproveitadas em outros programas sem a necessidade de modificações.

Funções

Na linguagem `R` as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

Funções

Na linguagem `R` as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

1. Função possui uma lista de argumentos;

Funções

Na linguagem `R` as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

1. Função possui uma lista de argumentos;
2. Corpo;

Funções

Na linguagem `R` as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

1. Função possui uma lista de argumentos;
2. Corpo;
3. Ambiente em que a função está definida.

Funções

Funções

1. A lista de argumentos refere-se à uma lista de símbolos (os argumentos da função) que são separados por vírgula, em que tais argumentos podem possuir valores por padrão. Além disso, é possível que uma função possua um número variado de argumentos. Neste caso, é utilizado o argumento `...`. Daremos detalhes mais a frente.

Funções

1. A lista de argumentos refere-se à uma lista de símbolos (os argumentos da função) que são separados por vírgula, em que tais argumentos podem possuir valores por padrão. Além disso, é possível que uma função possua um número variado de argumentos. Neste caso, é utilizado o argumento `...`. Daremos detalhes mais a frente.

2. O corpo da função trata-se de um conjunto de instruções da linguagem `R` e normalmente forma um bloco de instruções;

Funções

1. A lista de argumentos refere-se à uma lista de símbolos (os argumentos da função) que são separados por vírgula, em que tais argumentos podem possuir valores por padrão. Além disso, é possível que uma função possua um número variado de argumentos. Neste caso, é utilizado o argumento `...`. Daremos detalhes mais a frente.

2. O corpo da função trata-se de um conjunto de instruções da linguagem `R` e normalmente forma um bloco de instruções;

3. O ambiente de definição da função refere-se ao ambiente ativo quando a função foi criada. Isso determina quais são os objetos visíveis pela função.

Sintaxe da declaração de uma função:

Funções

Sintaxe da declaração de uma função:

```
nome_da_funcao <- function(argumentos){
    bloco de instruções correspondente
    ao corpo da função.
}
```

Funções

Apesar de ser desejável que uma função contenha argumentos que possam receber dados e informações de como processar as informações passadas, uma função poderá possuir nenhum argumento. Normalmente essas funções são bastante específicas para se tornar útil mesmo sem possuir argumentos. Não é o caso da função que segue no exemplo que segue.

Exemplo: Corra o código seguinte:

```
1: funcao <- function(){
2:   cat(paste(c("<3", LETTERS[c(1,13,15)], " ",
3:   LETTERS[c(13,5,21)], " ", LETTERS[c(16,18,15,6,5,
4:   19,19,15,18)], "<3"), collapse = " "))
5: }
```

Funções

Note que a criação de uma função consiste na atribuição do conteúdo no bloco de código que compõe a função e seus argumentos a um nome, como qualquer outro objeto da linguagem `R`.

Exemplo: Corra o código abaixo que refere-se a implementação da função `celsiustofar()` que converter uma temperatura informada em graus Celsius (passada ao argumento `cel` da função) para Fahrenheit.

Funções

Note que a criação de uma função consiste na atribuição do conteúdo no bloco de código que compõe a função e seus argumentos a um nome, como qualquer outro objeto da linguagem `R`.

Exemplo: Corra o código abaixo que refere-se a implementação da função `celsiustofar()` que converter uma temperatura informada em graus Celsius (passada ao argumento `cel` da função) para Fahrenheit.

```
1: celsiustofar <- function(cel)
2: {
3:   temperatura <- 1.8 * cel + 32
4:   temperatura
5: }
6: celsiustofar(cel = 21.3)
#> [1] 70.34
```

Funções

Nota: Caso a função possa ser escrita em uma única linha, o uso das chaves que delimitam o bloco da função ($\{\}$) poderá ser omitido. Considere o exemplo abaixo:

```
1: celsiustofar <- function(cel) 1.8 * cel + 32
```

Exercício: Suponhamos que temos o vetor `temp` de temperaturas em graus Celsius que encontra-se apresentado mais a baixo. Construa um programa que utilize a função `celsiustofar()` e retorne um vetor de temperaturas convertidas para Fahrenheit.

```
1: # Considere no exemplo o vetor abaixo:
2: temp <- c(27.8,19.3,20.7,18.29,25.0,25.1,32.3,
3:           37.6,32.2,19.02,22.75)
```

Funções

Exercício: Altere a função `celsiustofar()` apresentada no exemplo anterior de modo que a função possa converter a temperatura de graus Celsius para Fahrenheit ou de Fahrenheit para graus Celsius.

Exercício: Altere a função `celsiustofar()` apresentada no exemplo anterior de modo que a função possa converter a temperatura de graus Celsius para Fahrenheit ou de Fahrenheit para graus Celsius.

Solução:

```
1: celsiustofar <- function(temp, ctof = TRUE)
2: {
3:     ctof <- as.logical(ctof)
4:     if(!is.logical(ctof) | is.na(ctof))
5:         stop("Um valor lógico não foi informado.")
6:
7:     temperatura <- ifelse(ctof == TRUE, 1.8 *
8:                           temp + 32, (temp - 32)/1.8)
9:     temperatura
10: }
```

Nota: O operador `!` é o operador **NOT** lógico na linguagem `R`.

Observe que a solução do exercício logo acima não ocasionaria problemas caso o usuário viesse a fornecer as strings (`"TRUE"`, `"FALSE"`, `"T"`, `"F"`, `"True"` e `"False"`) além dos valores lógicos `T`, `TRUE`, `F` e `FALSE`, propriamente ditos.

Exercício: Faça `help(is.na)` e leia a documentação. Faça o mesmo para a função `stop()`.

Funções

Importante

O resultado da última computação de uma função será o retorno da função. Nos exemplos acima, `temperatura` é o retorno da função `celsiustofar()`. Porém, em `R`, a função `return()` poderá ser utilizada para encerrar a execução da função no ponto da função em que `return()` se encontra. Sendo assim, o argumento de `return()` será retornado como resultado da função que estamos implementando.

Funções

Exercício: Construa uma função que calcula o IMC (Índice de Massa Corporal) de uma pessoa e retorne o IMC e a situação da pessoa com respeito ao seu peso.

Funções

Exercício: Construa uma função que calcula o IMC (Índice de Massa Corporal) de uma pessoa e retorne o IMC e a situação da pessoa com respeito ao seu peso.

Cálculo do índice: $\text{IMC} = \text{peso}/\text{altura}^2$.

Considere a tabela que segue a baixo as correspondências entre o IMC e a situação do paciente de acordo com o índice. Utilize essa tabela para construir o programa do exercício acima.

Tabela: Situação do peso segundo o IMC.

IMC	**Situação**
<17	Muito abaixo do peso
[17; 18.49]	Abaixo do peso
[18.5; 24.99]	Peso normal
[25; 29.99]	Acima do peso
[30; 34.99]	Obesidade I
[35; 39.99]	Obsesidade II (severa)
>40	Obesidade III (mórbida)

Uma possível solução:

```
1: imc <- function(peso,altura)
2: {
3:     # A altura deverá ser informada em metros.
4:     # O peso deverá ser informado em kg.
5:
6:     # Uma pessoa que colocar valores negativos
7:     # não merece nem comentário. Não retorne
8:     # nada.
9:     if (peso <= 0 | altura <= 0) return(NULL)
10:
11:    if (altura >= 3) warning("Você é desse planeta?")
12:
13:    imc <- peso/altura^2
```

Funções

```
14:   if (imc < 17) situacao <- "Muito abaixo do peso"
15:   else if (imc >= 17 & imc <= 18.49)
16:       situacao <- "Abaixo do peso"
17:   else if (imc >= 18.5 & imc <= 24.99)
18:       situacao <- "Peso normal"
19:   else if (imc >= 25 & imc <= 29.99)
20:       situacao <- "Acima do peso"
21:   else if (imc >= 30 & imc <= 34.99)
22:       situacao <- "Obesidade I"
23:   else if (imc >= 35 & imc <= 39.99)
24:       situacao <- "Obesidade II (severa)"
25:   else situacao <- "Obesidade III (mórbida)"
26:
27:   list(imc = imc, situacao = situacao)
28: }
```

Funções

Nota: Observe no programa acima as instruções na instrução condicional `if()` seja avaliada, a função retornará `NULL` e nada mais será avaliado.

Algumas funções matemáticas e estatísticas que podem ser úteis para a realização de computação científica serão listadas abaixo. Ficará como exercício o aluno ler a documentação de cada uma dessas funções e executarem os exemplos contidos na documentação.

Tais funções são úteis na implementações de novas funções que frequentemente utilizamos na computação científica.

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;
- `range(x)`: Vetor com o valor máximo e mínimo de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;
- `range(x)`: Vetor com o valor máximo e mínimo de `x`;
- `length(x)`: Quantidade de elementos de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;
- `range(x)`: Vetor com o valor máximo e mínimo de `x`;
- `length(x)`: Quantidade de elementos de `x`;
- `mean(x)`: Média dos valores de `x`;

Funções

- `median(x)`: Mediana dos valores de `x`;

Funções

- `median(x)`: Mediana dos valores de `x`;
- `sd(x)`: Desvio padrão dos elementos do vetor `x`;
- `var(x)`: Variância dos elementos do vetor `x`;
- `quantile(x)`: Os quartis de `x`;
- `scale(x)`: Normaliza os elementos do vetor `x`, isto é, subtrai os elementos da média e divide pelo desvio-padrão;
- `sort(x)`: Ordena os elementos do vetor `x`;
- `rank(x)`: Faz o *ranking* dos elementos de `x`;
- `log(x,base)`: Calcula o logaritmo dos elementos de `x` na `base` escolhida. Se nenhuma base for escolhida, será calculado o logaritmo natural;
- `exp(x)`: Calcula o exponencial dos elementos de `x`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;
- `match(x,y)`: Retorna as posições dos elementos de `x` em `y`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;
- `match(x,y)`: Retorna as posições dos elementos de `x` em `y`;
- `union(x,y)`: Retorna o vetor união;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;
- `match(x,y)`: Retorna as posições dos elementos de `x` em `y`;
- `union(x,y)`: Retorna o vetor união;
- `intersect(x,y)`: Obtém um vetor com a interseção dos vetores `x` e `y`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;
- `choose(n,k)`: Calcula o número de combinações n, k a k;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;
- `choose(n,k)`: Calcula o número de combinações n, k a k;
- `factorial(x)`: Calcula o fatorial dos elementos de `x`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;
- `choose(n,k)`: Calcula o número de combinações n, k a k;
- `factorial(x)`: Calcula o fatorial dos elementos de `x`;
- `pi`: Retorna o vetor com um único elemento com o valor de π;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;
- `det(A)`: Determinante da matriz `A`, se `A` é uma matriz quadrada;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;
- `det(A)`: Determinante da matriz `A`, se `A` é uma matriz quadrada;
- `solve(A)`: A inversa da matriz `A`, se `A` é uma matriz quadrada;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;
- `det(A)`: Determinante da matriz `A`, se `A` é uma matriz quadrada;
- `solve(A)`: A inversa da matriz `A`, se `A` é uma matriz quadrada;
- `solve(A,b)`: Resolve o sistema de equações lineares $Ax = b$;

Funções

- `eigen(A)`: Auto-valores e auto-vetores da matriz `A`, se `A` é uma matriz quadrada.

Exercício: Apresente exemplos do uso das funções estatísticas e algébricas apresentadas acima.

Exercício: Refaça o exercício referente as simulações de Monte Carlo para avaliação dos estimadores $\hat{\sigma}^2$ e S^2 em frames anteriores utilizando o conceito de funções.

Nota: Sabemos o limite da frequência relativa da ocorrência de um evento A qualquer obtido da realização de um experimento aleatório converge para a probabilidade da ocorrência de A, isto é

$$P(A) = \lim_{n \to +\infty} \frac{n_A}{n},$$

em que n_A é o número de ocorrência do evento de interesse A em n repetições do experimento aleatório (**teoria frequentista da probabilidade**).

Nota: Sabemos o limite da frequência relativa da ocorrência de um evento A qualquer obtido da realização de um experimento aleatório converge para a probabilidade da ocorrência de A, isto é

$$P(A) = \lim_{n \to +\infty} \frac{n_A}{n},$$

em que n_A é o número de ocorrência do evento de interesse A em n repetições do experimento aleatório (**teoria frequentista da probabilidade**).

Por exemplo, no experimento aleatório que consiste em lança uma moeda e verificar a ocorrência de "cara" poderíamos estar interessados no evento $A = \{\text{"ocorrência de cara e um lançamento"}\}$. Então n_A é o número de ocorrência de caras em n lançamentos da moeda.

Como a moeda é honesta, isto é, $P(\text{“}cara\text{”}) = P(\text{“}coroa\text{”})$, esperamos que

$$P(A) = P(\text{“}cara\text{”}) = \lim_{n \to +\infty} \frac{n_A}{n} = \frac{1}{2}.$$

Como a moeda é honesta, isto é, $P(\text{“}cara\text{”}) = P(\text{“}coroa\text{”})$, esperamos que

$$P(A) = P(\text{“}cara\text{”}) = \lim_{n \to +\infty} \frac{n_A}{n} = \frac{1}{2}.$$

Exercício: Construa uma função que simule o lançamento de uma moeda. Posteriormente simule para para $n = 10, 20, 50, 100, 1000$ e 100000 lançamentos da moeda. Obtenha o limite da frequência relativa para cada tamanho de amostra com uma aproximação à $P(A)$. Comente o código. **Dica**: utilize a função `sample()` para amostrar no conjunto $\{0, 1\}$, em que 0 refere-se a **coroa** e 1 refere-se à **cara**.

Funções

Nota: Observe que o exercício propõe simular um experimento real e bastante simples que consiste no lançamento de uma moeda honesta. Mais um exemplo de uma simulação de Monte Carlo.

Funções

Em R, caracteres especiais não são permitidos para nomes de funções. Por exemplo, não é permitido criar uma função de nome +, −, * ou /.

Muito embora tratamos os operadores acima como operadores binários, pois assim o são, poderemos fazer uso desses operadores literalmente especificando o nome da função (do operador) e os seus argumentos suportados.

Exemplo: Corra o código que segue:

```
1: "+"(2,5)
2: "*"(3,4)
3: "-"(5,1)
4: "/"(2,8)
5: "+"(1, "*"(2,3))
```

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

1. `body()`: mostra o código inserido no corpo da função;

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

1. `body()`: mostra o código inserido no corpo da função;
2. `formals()`: retorna a lista de argumentos que controla o funcionamento da função;

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

1. `body()`: mostra o código inserido no corpo da função;
2. `formals()`: retorna a lista de argumentos que controla o funcionamento da função;
3. `environment()`: retorna o "map" da localização das variáveis utilizadas na função.

Funções

Exemplo: Corra o código abaixo:

```
1: f <- function(x) x^2
2: f
#> function(x) x^2 # Não foi retornado o ambiente.
3:
4: formals(f)
#> $x
5: body(f)
#> x^2
6: environment(f)
#> <environment: R_GlobalEnv>
```

Funções

Nota: Tais funções também podem ser utilizadas para modificar uma função.

Exemplo: Corra o exemplo abaixo:

```
1: f <- function(x) x^2
2: body(f) <- expression(x^3 + c)
3: formals(f) <- list(x = 1, c = 2)
4: f
#> function (x = 1, c = 2)
#> x^3 + c
```

Funções

As funções `formals()`, `body()` e `environment()` poderão retornar `NULL` caso elas não estejam escritas em `R`.

Nota: Algumas funções em `R` podem ser escritas em `C`, `C++` ou mesmo `Fortran`.

Exemplo: A função `sum()` é escrita em `C`, em que o código é chamado pela função usando `.Primitive()`. Sendo assim, as funções `formals()`, `body()` e `environment()` retornarão `NULL`.

Funções

```
1: sum
#> function (..., na.rm = FALSE) .Primitive("sum")
2: formals(sum)
#> NULL
3: body(sum)
#> NULL
4: environment(sum)
#> NULL
```

Exercício: Construa a função `myderiv()` para o cálculo da derivada numérica de uma função em um ponto. Lembre-se, se f é uma função diferenciável em x_0, a derivada de $f(x)$ no ponto x_0 é dada por

$$f^{'}(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

Por meio dessa definição, tomando $h \neq 0$, temos que uma aproximação à $f^{'}(x_0)$ é dada por:

$$f^{'}(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

Dica: h não poderá ser "muito" pequeno uma vez que isso poderia implicar em cancelamentos catastróficos.

Solução:

```
1  f <- function(x){
2    x^4 * sin(x)
3  }
4
5  numeric_derivation <- function(func, x0 = 1, h = 0.0001){
6    if (h == 0 | h >= 1) stop("h devara assumir valores no
         intervalo (0,1).")
7
8    a <- func(x0 + h)
9    b <- f(x0)
10
11   # Buscando um novo valor de h.
12   if((a - b) == 0) { # Cancelamento catastrofico.
13     repeat{
14       h <- h * 10
15       a <- func(x0 + h)
16       b <- f(x0)
```

Funções

```
17        # Criterio de parada do repeat.
18        if((a-b) > 1e-5) break
19      }
20    }
21    return(list(derivada = round((a - b)/h, 4), h_usado = h))
22  }
23  numeric_derivation(func = f, x0 = 1)
```

Nota: Funções primitivas só são encontradas no pacote base e, como elas operam em um nível baixo, podem ser mais eficientes. Como são escritas em outras linguagens, estas normalmente possuem regras diferentes para correspondência dos argumentos. Normalmente a equipe de desenvolvimento da linguagem `R` tenta evitar a criação de funções primitivas, a menos que não haja outra opção.

Funções

Exercício: Construa a função `myloglikelihood()`, em `R`, que receba como argumentos uma função densidade de probabilidade implementada pelo usuário e uma amostra aleatória. A função deverá retornar o valor de log-verossimilhança com base nos argumentos.

Funções

Exercício: Construa a função `myloglikelihood()`, em `R`, que receba como argumentos uma função densidade de probabilidade implementada pelo usuário e uma amostra aleatória. A função deverá retornar o valor de log-verossimilhança com base nos argumentos.

Exercício: Melhore a função `myloglikelihood()` para que esta também identifique se a função digitada é ou não uma função densidade de probabilidade.

Depuração

É bastante comum implementarmos um programa em alguma linguagem de programação e observarmos ocorrências de erros. Também é comum não sabermos exatamente a fonte. Para facilitar a busca de tais problemas ou comportamentos imprevistos, algumas linguagens possuem comandos ou ferramentas que permitem a realização de **debugging**. A linguagem `R` possui diversas funções para este fim. No momento, abordaremos duas:

Depuração

Depuração

1. `traceback()`: Permite o programador obter a sequência de funções que foram chamadas até ocorrer o mau comportamento;

Depuração

1. `traceback()`: Permite o programador obter a sequência de funções que foram chamadas até ocorrer o mau comportamento;

2. `browser()`: Caso a função acima não ajude a resolver o problema, ou então se conseguimos encontrar sua origem mas não entendemos o seu porquê, poderemos utilizar a função `browser()` que fará que todo o código após sua posição seja executado passo a passo e sob a mesma ordem.

Nota: Ao encontrar a função `browser()` a linguagem `R` vai parar e só executará a instrução seguinte quando o programador solicita o avanço por meio de um **ENTER**, parando logo em seguida aguardando um novo comando.

Depuração

Nota: Em cada uma da paradas a linguagem `R` apresentará um prompt de comando especial (`Browse[1]>`) em que o programador poderá fazer quase tudo o que é permitido fazer no prompt de comando "normal" (>), como por exemplo checar o conteúdo de objetos.

Exemplo: Corra o código abaixo:

```
1: funcao <- function(n,x) {
2:    s <- 0
3:    for(i in 1:n) {
4:       s <- s + sqrt(x)
5:       x <- x-1
6:    }; s
7: }; funcao(2,4)
#> [1] 3.732051
```

Depuração

```
8: funcao(3,1)
#> [1] NaN
#> Warning message:
#> In sqrt(x) : NaNs produzidos
```

Depuração

```
8: funcao(3,1)
#> [1] NaN
#> Warning message:
#> In sqrt(x) : NaNs produzidos
```

Nota: Quando implementamos uma função, na grande maioria dos casos não queremos que o retorno seja `NaN`. O retorno `NaN` (**Not a Number**) normalmente refere-se ao retorno de uma expressão matemática que não pode ser avaliada. Por exemplo, `Inf - Inf`, `0 * Inf`, `Inf / Inf`, `0 / 0`, entre outros casos possíveis.

Depuração

Vamos incluir uma chamada à função `browser()` no corpo da função antes do local ao qual achamos que o erro encontra-se presente.

Exemplo:

```
1: funcao <- function(n,x) {
2:    s <- 0
3:    browser()
4:    for(i in 1:n) {
5:       s <- s + sqrt(x)
6:       x <- x-1
7:    }; s
8: }; funcao(3,1)
```

Escopo de Variáveis

Ao criarmo novos objetos (vetores, matrizes, funções, entre outros) na linguagem `R`, estamos criando-os em um ambiente denominado de **workspace** que também poderá ser chamado de **ambiente de topo**.

Por exemplo, ao invocarmos à função `ls()`, o que obtemos é um vetor de objetos nesse ambiente de topo. Porém, há diversas situações em que levam a criação de novos ambientes, levando assim a uma hierarquia de ambientes aninhando uns aos outros até o ambiente de topo.

Escopo de Variáveis

Qual a importância disso?

Escopo de Variáveis

Qual a importância disso?

Resposta: A importância de entender o escopo de variáveis reside nas regras de **scope** ou de **visibilidade** dos objetos em `R` que dependem do ambiente em que eles são definidos.

Escopo de Variáveis

Qual a importância disso?

Resposta: A importância de entender o escopo de variáveis reside nas regras de **scope** ou de **visibilidade** dos objetos em `R` que dependem do ambiente em que eles são definidos.

Importante: Quando criamos uma função, ela ficará associada ao ambiente ao qual foi criada, em que este ambiente normalmente é o **workspace**. Porém, ao chamarmos uma função, a linguagem `R` irá executá-la em um novo ambiente "**por baixo**" do ambiente em que ela foi chamada. Dessa forma, o ambiente onde serão executadas as instruções que formam o corpo da função não será o mesmo que executará a função.

Escopo de Variáveis

Essa diferença entre ambiente onde um objeto (nesse caso uma função) é chamado e ambiente onde as instruções são avaliadas tem impacto nos objetos que são visíveis em cada um destes dois ambientes.

Exemplo: Corra o código que segue:

```
1: x <- 2; z <- 3
2: f <- function(x) {
3:   a <- 34
4:   y <- x / 4 * a * z
5:   y
6: }
7: f(2)
#  [1] 51
8: a
# Error: Object "a" not found
```

Escopo de Variáveis

Observe duas coisas:

1. Note que o objeto a foi definido no bloco de instruções da função `f()` (linhas 3 à 5). Essas instruções são executadas em um ambiente mais a baixo do **workspace**. Sendo assim, ao chamar o objeto a na linha 8 (**workspace**), este não será reconhecido.

2. Por outro lado, note que na linha 4 fazemos referência ao objeto z. Muito embora z não esteja definido no ambiente que executará o bloco de instruções da função `f()`, o objeto será reconhecido no ambiente mais abaixo.

Escopo de Variáveis

Regra (Lexical Scoping):

Quando um objeto é utilizado em uma avaliação, este será procurado no ambiente ao qual foi invocado. Caso seja encontrado um objeto de mesmo nome no ambiente de invocação, este será utilizado. Caso o objeto não seja encontrado no ambiente de invocação, o R procura-o no ambiente acima e assim sucessivamente até alcançar o ambiente de topo. Se nesse processo um objeto de mesmo nome for encontrado, este será utilizado, caso contrário, será retornado um erro.

Escopo de Variáveis

Exemplo: Corra o código abaixo e justifique a saída obtida ao chamar `g(2)`.

```
1:  a <- 1
2:  b <- 2
3:  f <- function(x) {
4:    a * x + b
5:  }
6:  g <- function(x) {
7:    a <- 2
8:    b <- 1
9:    f(x)
10: }
11: g(2)
```

Escopo de Variáveis

Exemplo: Corra o código abaixo e justifique a saída obtida ao chamar `g(2)`.

```
1:  a <- 1
2:  b <- 2
3:  f <- function(x) {
4:    a * x + b
5:  }
6:  g <- function(x) {
7:    a <- 2
8:    b <- 1
9:    f(x)
10: }
11: g(2)
```

Observe que o resultado obtido será **4**.

Escopo de Variáveis

Exemplo: Corra o código abaixo e justifique a saída obtida ao chamar `g(2)`.

```
1:  f <- function(x) {
2:    a * x + b
3:  }
4:
5:  g <- function(x) {
6:    a <- 1
7:    b <- 2
8:    f <- function(x) {
9:      b * x + a
10:   }
11:   f(x)
12: }
13: g(2)
```

Escopo de Variáveis

Considere os dois exemplos anteriores:

No primeiro caso, a função `f()` invocada na linha 9 foi definida no **workspace**. Dessa forma, suas instruções será executada em um ambiente abaixo do ambiente de topo. Como neste ambiente não há referências aos objetos `a` e `b`, o `R` irá procurar objetos de nomes `a` e `b`, respectivamente, no ambiente mais acima, que nesse caso é o **workspace**. Em resumo, ao ser invocada uma função, os objetos utilizados pela função será obtido no ambiente de escopo da função ou no ambiente de definição da função, ou mesmo em ambientes superiores, respeitando essa hierarquia.

Escopo de Variáveis

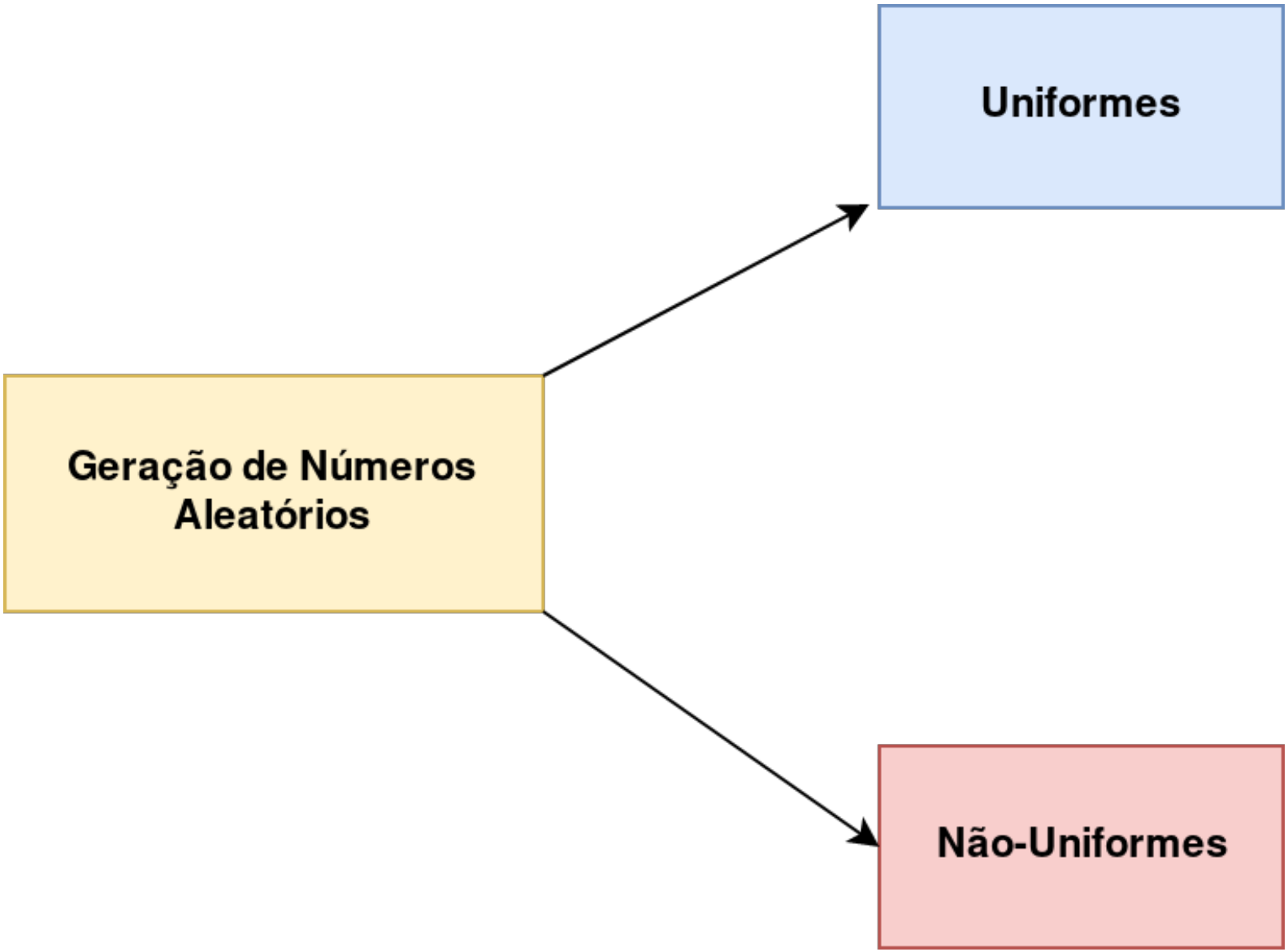
Já no segundo caso, perceba que a função `f()` foi definida dentro da função `g()`. Veja que a função `g()` foi definida no ambiente **workspace**. Em um ambiente mais abaixo será definido os objetos que compreendem o corpo da função `g()`, em que um desses objetos é a função `f()`. Abaixo desse ambiente mais interno teremos um outro ambiente que executará a instrução na linha 9 do segundo exemplo. Nesse ambiente, os objetos `a` e `b` não estão definidos. Dessa forma, seguindo a regra **lexical scoping**, o `R` irá procurar as definições dos objetos em em ambientes mais externos. No caso do exemplo, no ambiente interno ao ambiente da definição da função `g()` será encontrado que `a <- 1` e `b <- 2`.

Geração de Números Pseudo-Aleatórios

Uma das principais ferramentas necessária na estatística computacional é a habilidade de simular variáveis aleatórias de uma determinada distribuição de probabilidade, principalmente quando fazemos uso de estatísticas paramétricas, uma vez que nesses casos fazemos suposições sobre a distribuição dos dados.

Observação: Para que seja possível gerar adequadamente números **pseudo-aleatórios** de uma determinada distribuição de probabilidade, é preciso ter um bom gerador de números pseudo-aleatórios uniformemente distribuído.

Geração de Números Pseudo-Aleatórios



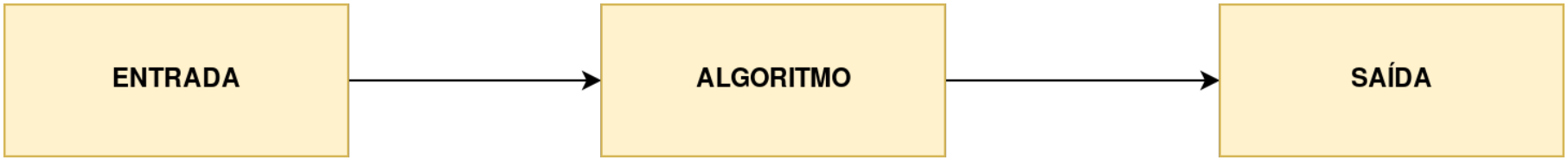
Geração de Números Pseudo-Aleatórios

Nota: Diversos materiais usam o título **Geração de Números Aleatórios**. Trata-se de um "péssimo" título uma vez que resultados puramente aleatórios **não são** reproduzíveis.

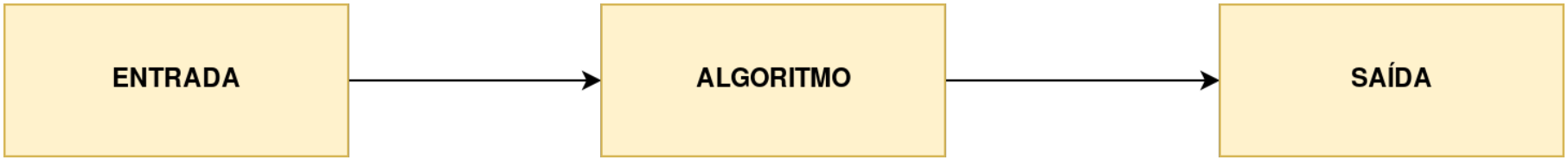
Um título mais adequado seria **Sequência Pseudo-Aleatória de Números**. De toda forma, sempre que for mencionado o título **Geração de Números Aleatórios** entenda que se estar a falar sobre **Sequência Pseudo-Aleatória de Números**.

Importante: A sequência de números pseudo-aleatórios são reproduzíveis uma vez que tal sequência é obtida por meio de um algoritmo.

Geração de Números Pseudo-Aleatórios

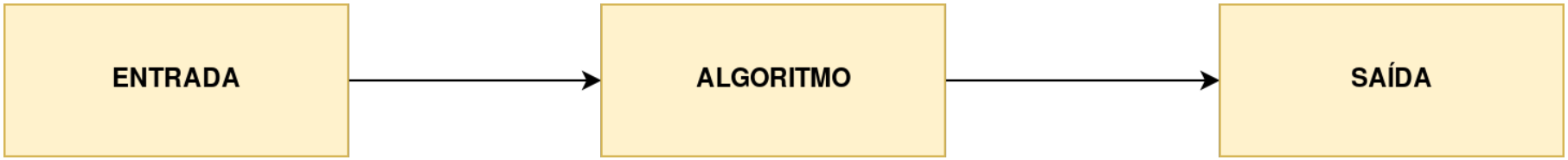


Geração de Números Pseudo-Aleatórios



Nota: A entrada normalmente é chamada de **semente** do gerador de números pseudo-aleatórios.

Geração de Números Pseudo-Aleatórios



Nota: A entrada normalmente é chamada de **semente** do gerador de números pseudo-aleatórios.

Fato: A partir de sequências uniformes, podemos gerar sequências não-uniformes.

Definição (Transformação Integral de Probabilidade): Seja X uma variável aleatória com função de distribuição F_X. A transformação de X tal que $U = F_X(X)$ é denominada **transformação integral de probabilidade**.

Definição (Transformação Integral de Probabilidade): Seja X uma variável aleatória com função de distribuição F_X. A transformação de X tal que $U = F_X(X)$ é denominada **transformação integral de probabilidade**.

Observe que o uso da transformação acima depende da possibilidade de invertermos a função F. A função inversa tem domínio em $[0, 1]$. Porém, se F tiver saltos ou for escada, F não admitirá inversa. Dessa forma, utilizaremos a **função inversa generalizada** e que por abuso de notação será representada por F^{-1}.

Definição (**Inversa Generalizada de** F): Seja F uma função de distribuição qualquer. A inversa generalizada denotada por F^{-1} é definida como:

Definição (**Inversa Generalizada de** F): Seja F uma função de distribuição qualquer. A inversa generalizada denotada por F^{-1} é definida como:

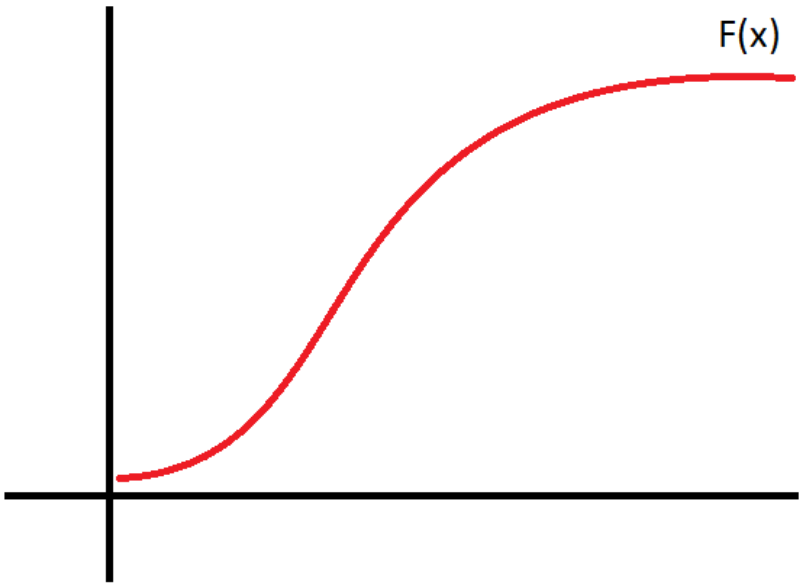
$$F^{-1}(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}.$$

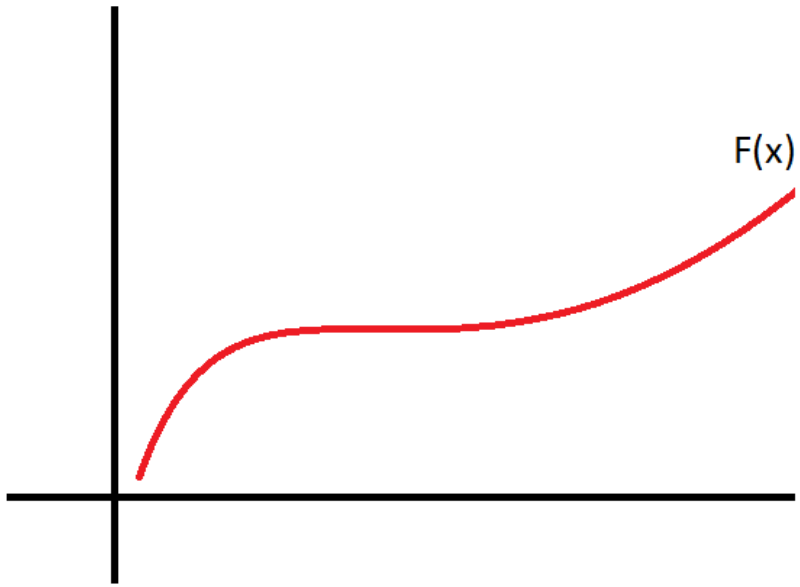
Definição (**Inversa Generalizada de** F): Seja F uma função de distribuição qualquer. A inversa generalizada denotada por F^{-1} é definida como:

$$F^{-1}(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}.$$

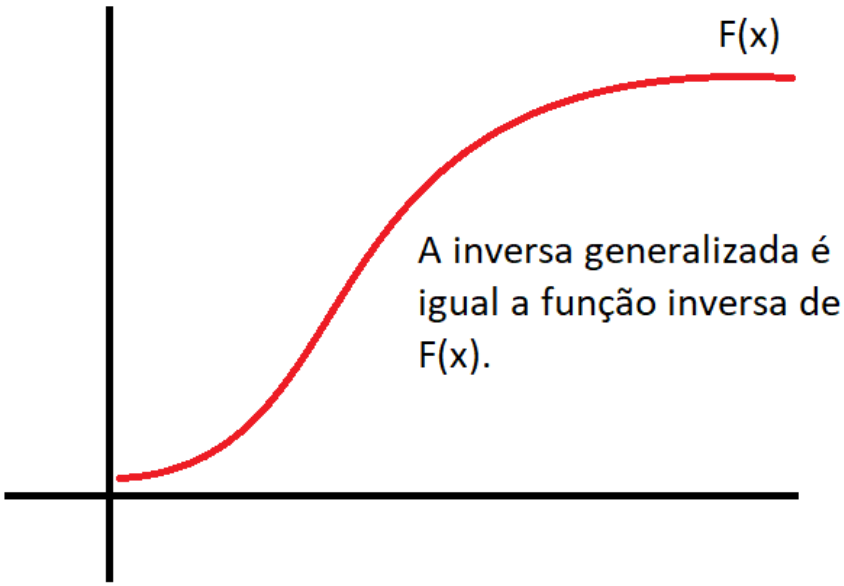
Nota: Caso a função inversa de F exista no sentido usual, esta coincidirá com a função inversa generalizada de F.

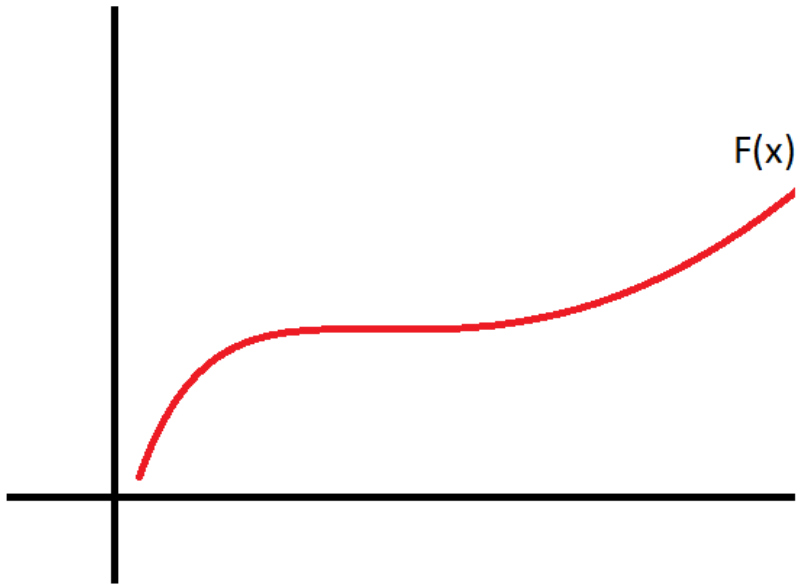
Geração de Números Pseudo-Aleatórios



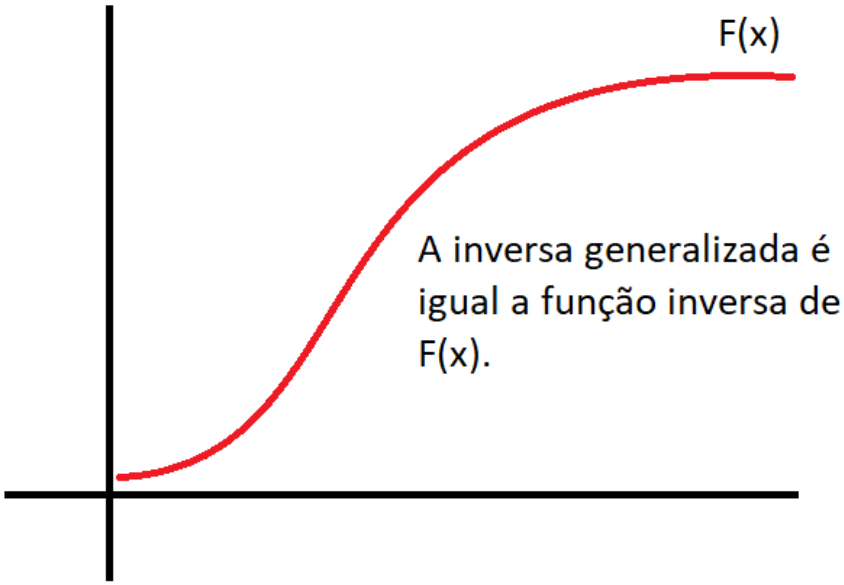


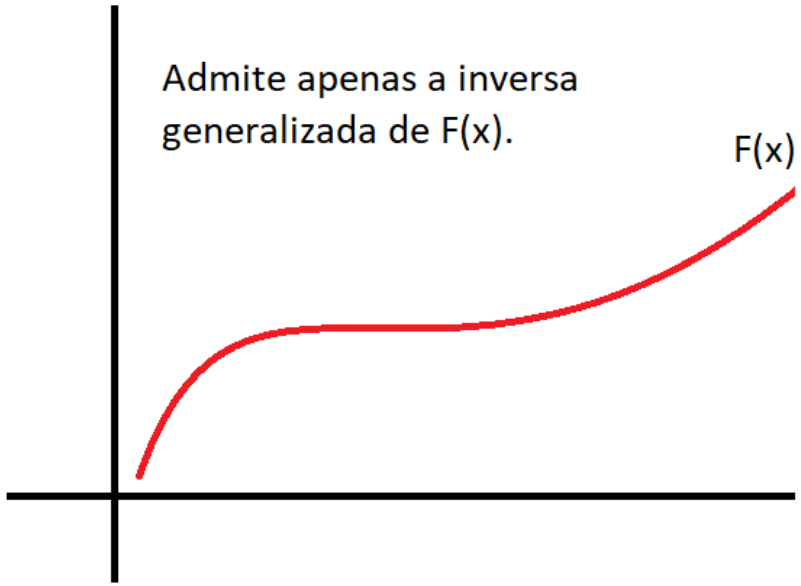
Geração de Números Pseudo-Aleatórios



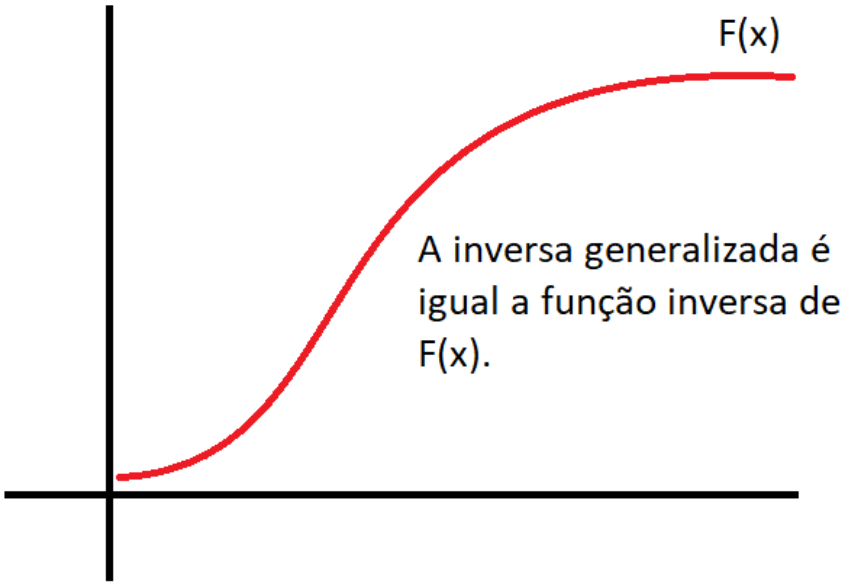


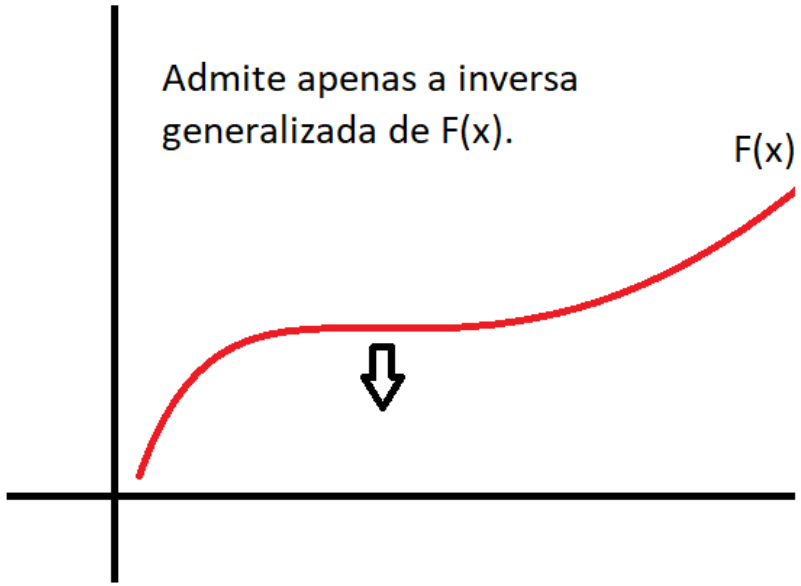
Geração de Números Pseudo-Aleatórios



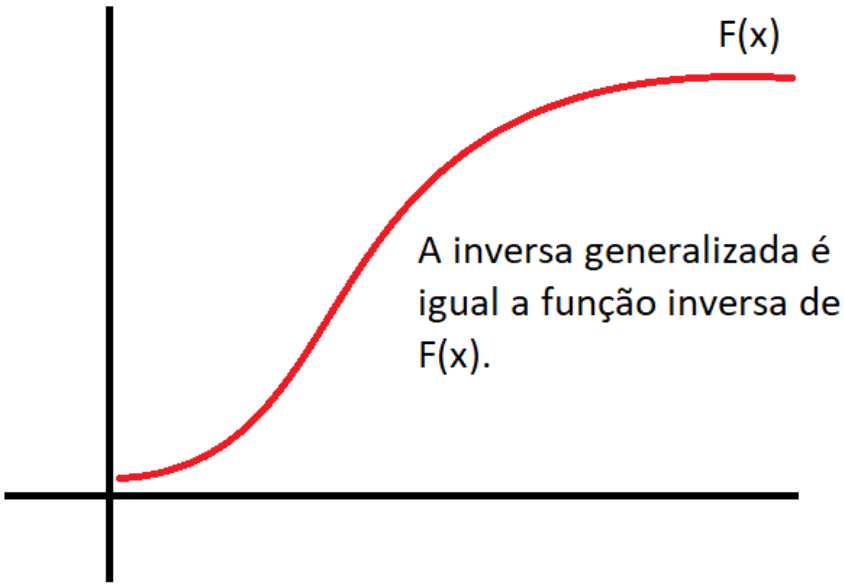


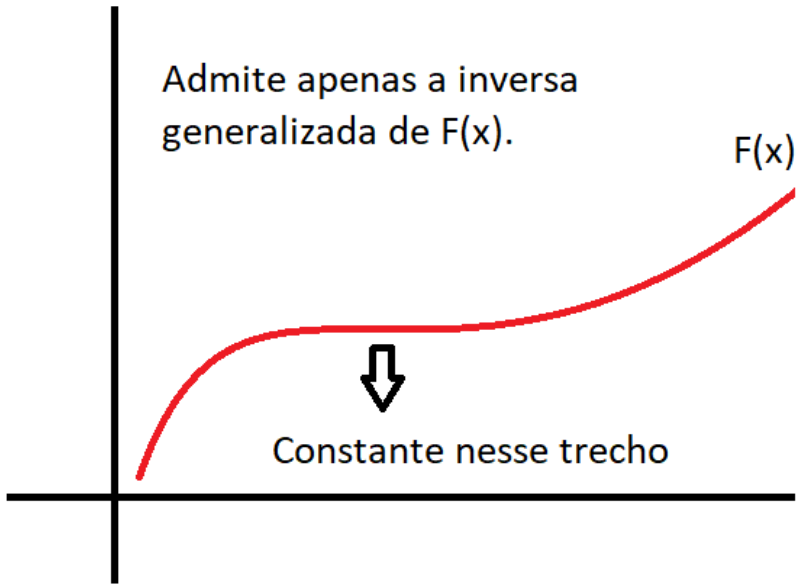
Geração de Números Pseudo-Aleatórios





Geração de Números Pseudo-Aleatórios





Observação Importante: A inversa generalizada cuida de situações em que a função F não é invertível.

Proposição: Seja X uma variável aleatória contínua com função de distribuição F. Sendo $U \sim U_c[0,1]$, então $X = F_X^{-1}(U)$ terá função de distribuição F.

Observação Importante: A inversa generalizada cuida de situações em que a função F não é invertível.

Proposição: Seja X uma variável aleatória contínua com função de distribuição F. Sendo $U \sim U_c[0,1]$, então $X = F_X^{-1}(U)$ terá função de distribuição F.

Prova:

$$P(X \leq x) = P(F_X^{-1}(U) \leq x) = P(U \leq F_X(x)) = F(x).$$

Geração de Números Pseudo-Aleatórios

O método poderá ser aplicado para geração de observações de variáveis aleatórias contínuas ou discreta de uma determinada distribuição de probabilidade.

Algoritmo (Método da Inversão)

O método poderá ser aplicado para geração de observações de variáveis aleatórias contínuas ou discreta de uma determinada distribuição de probabilidade.

Algoritmo (Método da Inversão)

1. Obtenha a inversa $F_X^{-1}(u)$.

Geração de Números Pseudo-Aleatórios

O método poderá ser aplicado para geração de observações de variáveis aleatórias contínuas ou discreta de uma determinada distribuição de probabilidade.

Algoritmo (Método da Inversão)

1. Obtenha a inversa $F_X^{-1}(u)$.
2. Escreva um comando ou função que calcule $F_X^{-1}(u)$.

Geração de Números Pseudo-Aleatórios

O método poderá ser aplicado para geração de observações de variáveis aleatórias contínuas ou discreta de uma determinada distribuição de probabilidade.

Algoritmo (Método da Inversão)

1. Obtenha a inversa $F_X^{-1}(u)$.
2. Escreva um comando ou função que calcule $F_X^{-1}(u)$.
3. Para cada observação de uma variável aleatória, faça:

O método poderá ser aplicado para geração de observações de variáveis aleatórias contínuas ou discreta de uma determinada distribuição de probabilidade.

Algoritmo (Método da Inversão)

1. Obtenha a inversa $F_X^{-1}(u)$.
2. Escreva um comando ou função que calcule $F_X^{-1}(u)$.
3. Para cada observação de uma variável aleatória, faça:
   a) Gere uma observação u de uma variável aleatória $U \sim U_c[0, 1]$.

O método poderá ser aplicado para geração de observações de variáveis aleatórias contínuas ou discreta de uma determinada distribuição de probabilidade.

Algoritmo (Método da Inversão)

1. Obtenha a inversa $F_X^{-1}(u)$.
2. Escreva um comando ou função que calcule $F_X^{-1}(u)$.
3. Para cada observação de uma variável aleatória, faça:
   a) Gere uma observação u de uma variável aleatória $U \sim U_c[0, 1]$.
   b) Calcule $x = F_X^{-1}(u)$.

Exercício: Considere a função densidade de probabilidade $f_X(x) = 3x^2, 0 \leq x \leq 1$. Obtenha utilizando o algoritmo acima uma sequência de 1000 números pseudo-aleatórios com distribuição $F_X(x)$.

Solução:

Temos que $F_X(x) = \int_0^x f_X(x)dx = x^3$, com $0 \leq x \leq 1$. Fazendo $u = x^3$, temos que

$$F_X^{-1}(u) = u^{1/3},\ 0 \leq u \leq 1.$$

Utilizaremos o algoritmo apresentado e $F_X^{-1}(u)$ obtido logo acima para obter uma sequência de observações $x_1, \ldots, x_n$.

Código R:

```
1:  # Função densidade de X.
2:  pdf_f <- function(x){
3:    3 * x^2
4:  }
5:
6:  # Gerando mil números pseudo-aleatórios
7:  # com distribuição U[0,1].
8:  set.seed(0); u <- runif(n = 1000, min = 0, max = 1)
9:
10: # Aplicando u à inversa de F_X(x).
11: x <- u^(1/3)
```

```
12: # Histograma da densidade da amostra.
13: hist(x, prob = TRUE, xlab = "Dominio",
14:       ylab = "Densidade",
15:       main = expression(f(x)==3*x^2))
16: dominio <- seq(from=0, to = 1, by = .01)
17: lines(dominio, pdf_f(dominio), lwd = 2, col = "red")
```

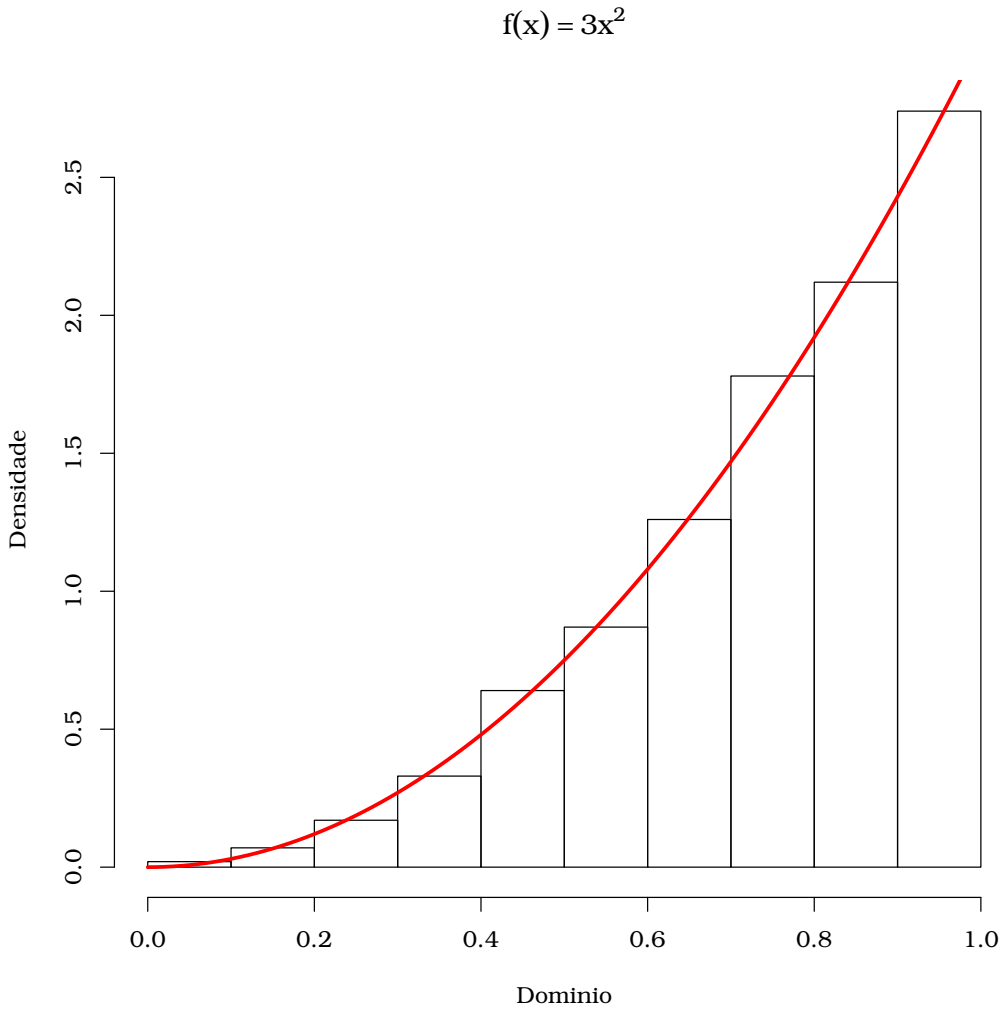
Será que deu certo?

```
12: # Histograma da densidade da amostra.
13: hist(x, prob = TRUE, xlab = "Dominio",
14:      ylab = "Densidade",
15:      main = expression(f(x)==3*x^2))
16: dominio <- seq(from=0, to = 1, by = .01)
17: lines(dominio, pdf_f(dominio), lwd = 2, col = "red")
```

Será que deu certo?

Observemos o ajustamento da distribuição $f_X(x) = 3x^2, 0 \leq x \leq 1$ **aos dados gerados**.

Geração de Números Pseudo-Aleatórios



Nota: Na Figura acima, o título possui uma expressão matemática. Este título é obtido especificando no argumento `main` a função `expression()` com a expressão matemática desejada. Consulte a documentação da linguagem para maiores detalhes.

Observação: O método da inversão é o algoritmo mais geral e utilizado para geração de números pseudo-aleatórios de uma distribuição qualquer.

Exercício: Seja $X \sim \text{Exp}(\lambda)$, tal que $f_X(x) = \lambda e^{-\lambda x}$, com $x > 0$ e $\lambda > 0$. Sem utilizar a função `rexp()`, construa uma função para geração de números pseudo-aleatórios de uma distribuição $\text{Exp}(\lambda)$.

Exercício: É possível fazer uso do método da inversão para implementarmos uma função que faz uso do método da inversão para geração de números pseudo-aleatórios de uma variável aleatória $X \sim \mathcal{N}(\mu, \sigma^2)$? Justifique sua resposta.

Geração de Números Pseudo-Aleatórios

Para gerarmos números pseudo-aleatórios de uma distribuição qualquer, foi preciso gerar números pseudo-aleatórios proveniente de uma distribuição uniforme.

Definição: Um gerador de números pseudo-aleatórios é um **algoritmo** que iniciando em um valor (**semente**) e usando uma transformação determinística D, produz uma sequência $u_i = D^i(u_0)$ de valores em $(0, 1)$.

Observação: O conhecimento de $u_1, \ldots, u_n$ não deverá conduzir ao conhecimento de $u_{n+1}, u_{n+2}, \ldots$. Dessa forma, desejamos que a sequência não poderá ser previsível.

Por exemplo, se gerarmos uma sequência $\{0, 1, \ldots, n\}$ (com n muito grande) e ao final dividirmos cada número por n, teremos uma sequência em $[0, 1]$, mas nesse caso há previsibilidade.

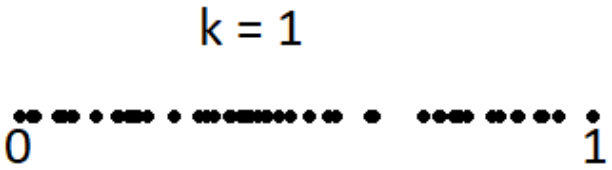
Muito Importante

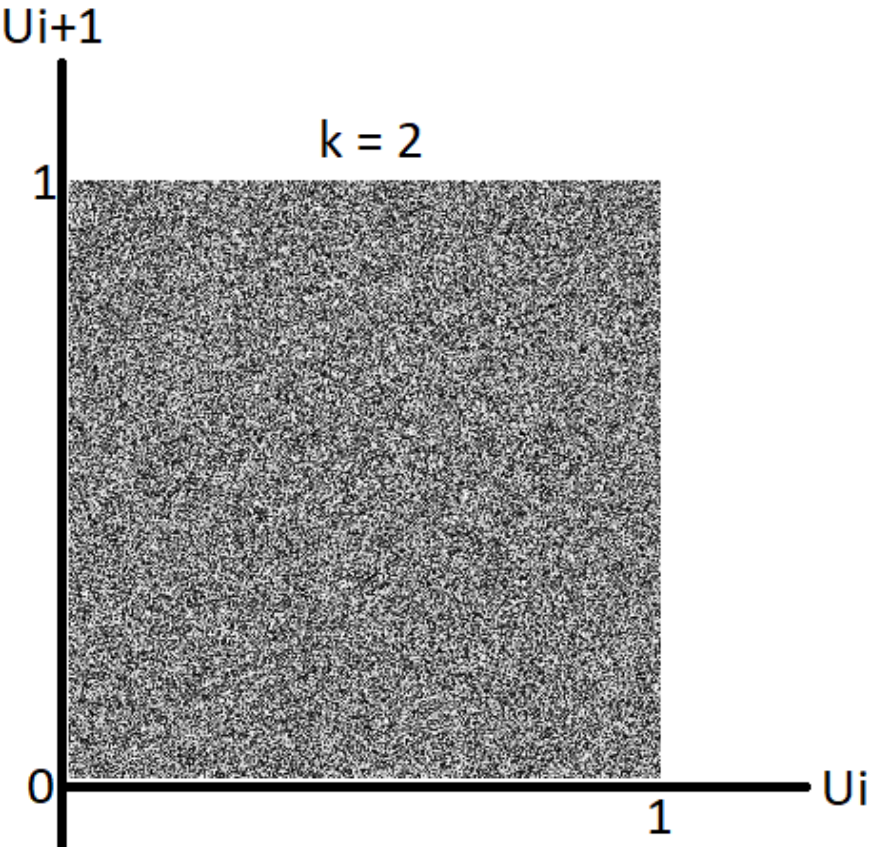
Um bom gerador de números pseudo-aleatórios com distribuição uniforme não permite a detecção de qualquer padrão em $[0, 1]^k$, com $k = 1, \ldots, 6$, isto é, não é possível detectar padrões até a sexta dimensão. Em outras palavras, é preciso que a uniformidade esteja presente até a sexta dimensão.

Geração de Números Pseudo-Aleatórios

Por exemplo, para $k = 1$ e $k = 2$, desejamos:

Por exemplo, para $k = 1$ e $k = 2$, desejamos:





Porém, lembre-se, precisamos de uniformidade até a sexta dimensão.

Geração de Números Pseudo-Aleatórios

Figura: John von Neumann.

Um dos primeiros algoritmos de número pseudo-aleatório (**Midsquare**) foi idealizado pelo matemático húngaro (naturalizado americano) **John von Neumann** (1903 - 1957).

von Neumann deixou contribuições em diversas áreas como ciência da computação, economia, teoria dos jogos, análise funcional, teoria dos conjuntos, entre outras. von Neumann foi um dos construtores do primeiro computador digital eletrônico (ENIAC).

Algoritmo Midsquare de von Neumann ($\sim$ 1949):

Algoritmo Midsquare de von Neumann ($\sim$ 1949):

1. Tome uma semente: um número inteiro de 4 dígitos.

Algoritmo Midsquare de von Neumann ($\sim$ 1949):

1. Tome uma semente: um número inteiro de 4 dígitos.
2. Calcule o quadrado desse número e preencha, se necessário, com zero à esquerda, para que o número gerado tenha 8 dígitos.

Algoritmo Midsquare de von Neumann ($\sim$ 1949):

1. Tome uma semente: um número inteiro de 4 dígitos.
2. Calcule o quadrado desse número e preencha, se necessário, com zero à esquerda, para que o número gerado tenha 8 dígitos.
3. Repita os passos 1 e 2 a quantidade de vezes desejadas e ao final divida cada número obtido, incluindo a semente por 10 mil.

Geração de Números Pseudo-Aleatórios

Exercício: Construa, em `R`, uma função que implemente o gerador Midsquare. **Dica**: Talvez seja necessário manipular strings para efetuar o passo 2 do algoritmo. As funções `strsplit()` e `paste()` podem ser úteis. Procurem maiores detalhes na documentação da linguagem.

Solução:

```
1: midsquare <- function(n = 1, semente = 1987){
2:   valores <- NULL; i = 1
3:   repeat{
4:      semente <- semente^2
5:      quadrado <- unlist(strsplit(as.character(semente),
6:                                  split=""))
7:      quadrado <- c(rep(0,8 - length(quadrado)),
8:                    quadrado)
9:      valores[i] <- as.numeric(paste(quadrado[3:6],
10:                                    collapse = ""))
```

Geração de Números Pseudo-Aleatórios

```
12:       semente <- valores[i]
13:       i <- i + 1
14:       if (i > n) break
15:   }
16:   valores
17:}
18: midsquare(n=4, semente = 1348)
#> [1] 8171 7652 5531 5919
```

Geração de Números Pseudo-Aleatórios

Exercício: O que acontece se um dado valor gerado $x_i = 0$? Discutam.

Geração de Números Pseudo-Aleatórios

Exercício: O que acontece se um dado valor gerado $x_i = 0$? Discutam.

Resposta: A sequência fica "presa" no zero, isto é:

$$x_{i+1} = x_{i+2} = \ldots = 0$$

.

Geração de Números Pseudo-Aleatórios

Exercício: O que acontece se um dado valor gerado $x_i = 0$? Discutam.

Resposta: A sequência fica "presa" no zero, isto é:

$$x_{i+1} = x_{i+2} = \ldots = 0$$

.

Exercício: Qual o problema do algoritmo ao considerar a semente ($x_0 = 3792$)? Discutam.

Geração de Números Pseudo-Aleatórios

Exercício: O que acontece se um dado valor gerado $x_i = 0$? Discutam.

Resposta: A sequência fica "presa" no zero, isto é:

$$x_{i+1} = x_{i+2} = \ldots = 0$$

.

Exercício: Qual o problema do algoritmo ao considerar a semente ($x_0 = 3792$)? Discutam.

Resposta: A sequência não é aparentemente aleatória.

Exercício: Qual o problema do algoritmo ao considerar $x_0 = 2100$?

Exercício: Qual o problema do algoritmo ao considerar $x_0 = 2100$?
Resposta: O ciclo do gerador é muito curto.

Exercício: Qual o problema do algoritmo ao considerar $x_0 = 2100$?
Resposta: O ciclo do gerador é muito curto.

Queremos geradores que possuam ciclos longos.

Exercício: Qual o problema do algoritmo ao considerar $x_0 = 2100$?
Resposta: O ciclo do gerador é muito curto.

Queremos geradores que possuam ciclos longos.

Sendo assim, precisaremos de um algoritmo melhor...

Gerador Congruencial:

$$x_i = (a * x_{i-1} + c) \bmod M, i = 1, 2, \ldots,$$

em que x_0 é a semente do gerador, a é o multiplicador, c é o deslocamento e M é o módulo.

Observação: Usualmente, tem-se que:

$$a, c, x_i \in \{0, 1, 2, \ldots, M-1\}.$$

Gerador Congruencial:

$$x_i = (a * x_{i-1} + c) \bmod M, i = 1, 2, \ldots,$$

em que x_0 é a semente do gerador, a é o multiplicador, c é o deslocamento e M é o módulo.

Observação: Usualmente, tem-se que:

$$a, c, x_i \in \{0, 1, 2, \ldots, M-1\}.$$

Definição: Chamaremos de **período** de um gerador ao número de termos gerador a partir de uma semente x_0 sem a sequência até então gerada se repetir.

Geração de Números Pseudo-Aleatórios

Nota: $a \bmod b$ representa o **resto da divisão** entre a e b.

Terminologias:

Nota: $a \bmod b$ representa o **resto da divisão** entre a e b.

Terminologias:

- Se $c \neq 0$, diremos que o gerador é **misto** (período máximo igual à M).

Nota: $a \bmod b$ representa o **resto da divisão** entre a e b.

Terminologias:

- Se $c \neq 0$, diremos que o gerador é **misto** (período máximo igual à M).
- Se $c = 0$, diremos que o gerador é **multiplicativo** (período máximo igual à $M - 1$). Nesse caso, excluímos o valor 0 da sequência.

Geração de Números Pseudo-Aleatórios

Exercício: Utilizando $M = 64$, $x_0 = 1$, $a = 19$ e $c = 15$, gere uma sequência de 10 números pseudo-aleatórios obtido pelo gerador congruencial.

Geração de Números Pseudo-Aleatórios

Exercício: Utilizando $M = 64$, $x_0 = 1$, $a = 19$ e $c = 15$, gere uma sequência de 10 números pseudo-aleatórios obtido pelo gerador congruencial.

Exercício: Utilizando a linguagem `R`, implemente uma função para o algoritmo do gerador congruencial.

Geração de Números Pseudo-Aleatórios

```
1: rcongruencial <- function(n = 10, semente = 1987,
2:                           parametros, unif = TRUE){
3:
4:    a <- parametros[1]; c <- parametros[2]
5:    M <- parametros[3]
6:
7:    i <- 2; vetor <- NULL;
8:    vetor[1] <- semente
9:
10:   repeat{
11:      vetor[i] <- (a * vetor[i-1] + c) %% M
12:      i <- i + 1
13:
14:      if(i > n + 1) break
15:   }
```

Geração de Números Pseudo-Aleatórios

```
16:  ifelse(unif == TRUE, vetor <- vetor[-1]/M,
17:  vetor <- vetor[-1])
18:  vetor
19:}
20:
21:rcongruencial(n = 10, semente = 1,
22:                 parametros = c(19,15,64), unif = F)
#> [1] 34 21 30 9 58 29 54 17 18 37
```

Geração de Números Pseudo-Aleatórios

Na linguagem `C`, a função `rand()` da biblioteca padrão **stdlib.h** utiliza-se do gerador. Mais especificamente, alguns compiladores de `C`, consideram:

Geração de Números Pseudo-Aleatórios

Na linguagem `C`, a função `rand()` da biblioteca padrão **stdlib.h** utiliza-se do gerador. Mais especificamente, alguns compiladores de `C`, consideram:

1. **gcc**: $M^{32}, a = 69069, c = 5$ (misto);

Geração de Números Pseudo-Aleatórios

Na linguagem `C`, a função `rand()` da biblioteca padrão **stdlib.h** utiliza-se do gerador. Mais especificamente, alguns compiladores de `C`, consideram:

1. **gcc**: $M^{32}, a = 69069, c = 5$ (misto);
2. **Microsoft**: $M^{32}, a = 214013, c = 2531011$ (misto);

Geração de Números Pseudo-Aleatórios

Na linguagem `C`, a função `rand()` da biblioteca padrão **stdlib.h** utiliza-se do gerador. Mais especificamente, alguns compiladores de `C`, consideram:

1. **gcc**: $M^{32}, a = 69069, c = 5$ (misto);
2. **Microsoft**: $M^{32}, a = 214013, c = 2531011$ (misto);
3. **Borland**: $M^{32}, a = 22695477, c = 1$ (misto).

Observação: Em `C`, a função `rand()` gera números entre 0 e `RAND_MAX` (constante simbólica que representa o maior inteiro representável pelo computador).

Geração de Números Pseudo-Aleatórios

Na linguagem de programação `Ox`, desenvolvida por Jurgen Doornik há o gerador de **Park-Miller** que é um gerador de números pseudo-aleatório congruencial multiplicativo. Nesse gerador, considera-se:

$$M = 2^{32} - 1, a = 16807, c = 0.$$

Geração de Números Pseudo-Aleatórios

Na linguagem de programação `Ox`, desenvolvida por Jurgen Doornik há o gerador de **Park-Miller** que é um gerador de números pseudo-aleatório congruencial multiplicativo. Nesse gerador, considera-se:

$$M = 2^{32} - 1, a = 16807, c = 0.$$

Comentário particular do professor: Não aconselharia o uso da linguagem `Ox` por se tratar de uma linguagem fechada mantida especialmente por uma única pessoa. Maiores detalhes em `http://www.doornik.com/`. Muito embora `Ox` é vendida como uma linguagem eficiente, temos a disposição diversas outras linguagem ainda mais eficientes e livres como é o caso de `C` e `C++`. Além disso, linguagem como, por exemplo, `C` e `C++` conversam facilmente com outras linguagem de programação.

Gerador Randu:

$$x_{i+1} = 65539 * x_i \mod 31.$$

Observação: Há diversos materiais falando sobre este gerador em que podemos citar Donald E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd edition (Addison-Wesley, Boston, 1998).

Nota: O gerador randu é um gerador congruencial que foi utilizado por muito tempo em manframes entre as décadas de 60 e 70. Porém, este gerador possui erros consideráveis e com o tempo ele foi deixado de lado. **Ele é considerado um dos piores algoritmos geradores de números pseudo-aleatórios já criado**.

Geração de Números Pseudo-Aleatórios

Exercício: Gere uma sequência de números pseudo-aleatórios utilizando o gerador randu.

Exercício: Implemente, utilizando a linguagem `R`, uma função para geração de números pseudo-aleatórios utilizando o gerador randu.

Solução:

```
1: rrandu <- function(n, semente){
2:     vetor <- NULL
3:    vetor[1] <- semente
4:    i <- 2
5:    repeat{
6:        vetor[i] <- (65539 * vetor[i-1]) %% 2^(31)
7:        i <- i + 1
8:        if (i > n+1) break
9:     }
10:    vetor[-1]
11:}
```

Geração de Números Pseudo-Aleatórios

Exemplo: Corra o código abaixo e descreva o motivo pelo qual o gerador não é razoável. Explique!

```
1: a <- NULL
2: for(i in 1:10)
3:    a[i] <- rrandu(n=1,semente=i)
4: plot(a, xlab = "x_i", ylab = "x_i+1",
5:      main = "Gerador Randu",
6:      pch = 16)
```

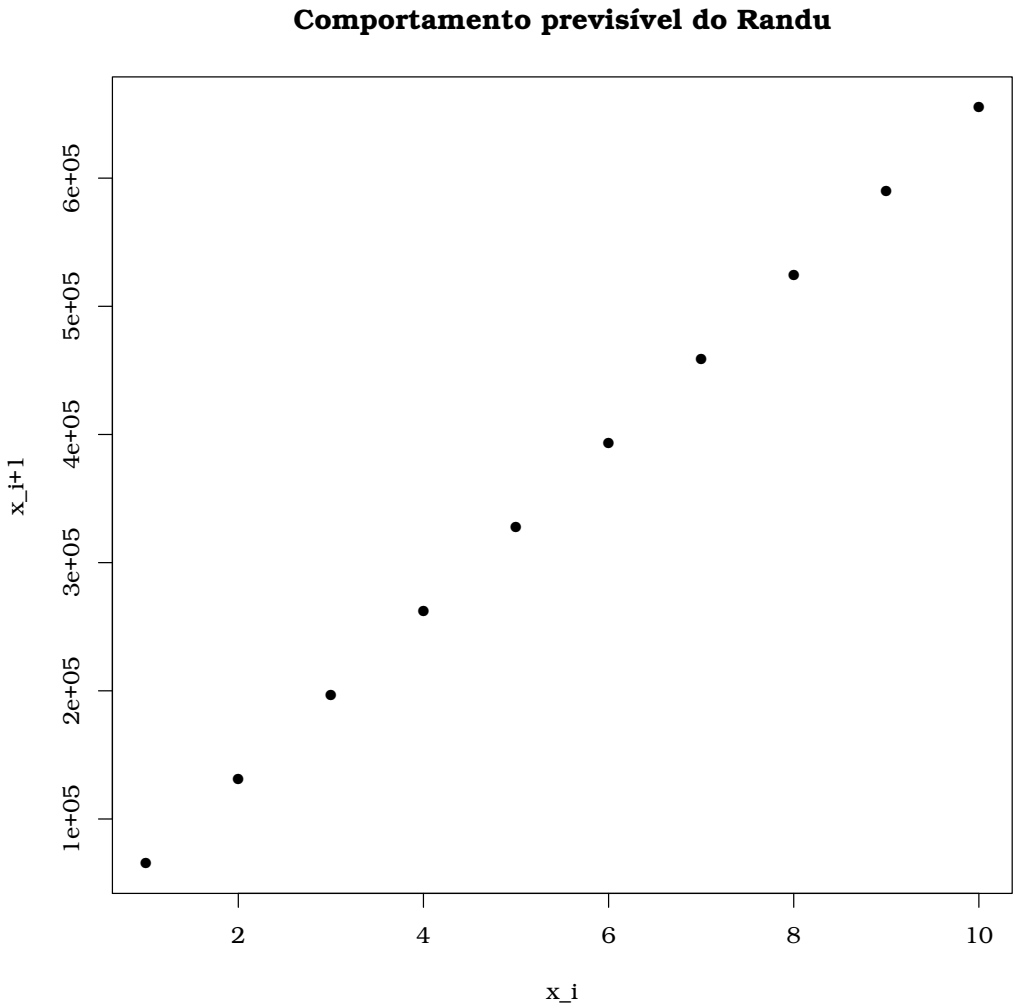
Geração de Números Pseudo-Aleatórios

Exemplo: Corra o código abaixo e descreva o motivo pelo qual o gerador não é razoável. Explique!

```
1: a <- NULL
2: for(i in 1:10)
3:    a[i] <- rrandu(n=1,semente=i)
4: plot(a, xlab = "x_i", ylab = "x_i+1",
5:      main = "Gerador Randu",
6:      pch = 16)
```

Observando o gráfico de x_i por x_{i+1}, é fácil perceber o comportamento previsível do gerador randu. Trata-se de uma característica muito indesejável em um gerador.

Geração de Números Pseudo-Aleatórios



Propriedades de Geradores Congruenciais

Notação: $a \equiv b \bmod M$.

Lê-se: "a é congruente para b módulo M".

Tal notação apresentada acima significa: $a - b$ é divisível por M.

Exemplo:

Propriedades de Geradores Congruenciais

Notação: $a \equiv b \mod M$.

Lê-se: "a é congruente para b módulo M".

Tal notação apresentada acima significa: $a - b$ é divisível por M.

Exemplo:

a) $10 \equiv 4 \mod 2$;

Propriedades de Geradores Congruenciais

Notação: $a \equiv b \mod M$.

Lê-se: "a é congruente para b módulo M" .

Tal notação apresentada acima significa: $a - b$ é divisível por M.

Exemplo:

a) $10 \equiv 4 \mod 2$;

b) $7 \equiv 1 \mod 3$;

Propriedades de Geradores Congruenciais

Notação: $a \equiv b \mod M$.

Lê-se: "a é congruente para b módulo M" .

Tal notação apresentada acima significa: $a - b$ é divisível por M.

Exemplo:

a) $10 \equiv 4 \mod 2$;

b) $7 \equiv 1 \mod 3$;

c) $-10 \equiv 2 \mod 2$;

Propriedades de Geradores Congruenciais

Notação: $a \equiv b \mod M$.

Lê-se: "a é congruente para b módulo M".

Tal notação apresentada acima significa: $a - b$ é divisível por M.

Exemplo:

a) $10 \equiv 4 \mod 2$;

b) $7 \equiv 1 \mod 3$;

c) $-10 \equiv 2 \mod 2$;

d) $10 \equiv 0 \mod 5$.

Propriedades de Geradores Congruenciais

Definição: a é raiz primitiva de M, se e só se:

Propriedades de Geradores Congruenciais

Definição: a é raiz primitiva de M, se e só se:

c1) $a^{M-1} - 1 \equiv 0 \mod M$, isto é, $a^{M-1} - 1$ é divisível por M;

Propriedades de Geradores Congruenciais

Definição: a é raiz primitiva de M, se e só se:

c1) $a^{M-1} - 1 \equiv 0 \mod M$, isto é, $a^{M-1} - 1$ é divisível por M;

c2) Para todo inteiro positivo I tal que $I < M - 1$, temos que $(a^I - 1)/M$ **não** é inteiro.

Propriedades de Geradores Congruenciais

Teorema (Gerador linear congruencial multiplicativo): Seja M um número primo. O gerador congruencial multiplicativo tem **período completo**, isto é, $M - 1$, se e somente se, a (multiplicador) é raiz primitiva de M.

Propriedades de Geradores Congruenciais

Teorema (Gerador linear congruencial multiplicativo): Seja M um número primo. O gerador congruencial multiplicativo tem **período completo**, isto é, $M - 1$, se e somente se, a (multiplicador) é raiz primitiva de M.

Exercício: Postule um gerador congruencial linear considerando $M = 13$ e aplique o teorema acima para determinar se seu gerador possui período completo igual à 12.

Propriedades de Geradores Congruenciais

Exercício: Considere o gerador:

$$x_i = a * x_{i-1} \mod 11.$$

Considere também $a = 2, 3, \ldots, 10$. Para quais valores de a somos conduzidos a período completo igual à 10?

Propriedades de Geradores Congruenciais

Exercício: Considere o gerador:

$$x_i = a * x_{i-1} \mod 11.$$

Considere também $a = 2, 3, \ldots, 10$. Para quais valores de a somos conduzidos a período completo igual à 10?

Resposta: $a \in \{2, 6, 7, 8\}$ conduzem o gerador $x_i = a * x_{i-1} \mod 11$ a ter período completo.

Propriedades de Geradores Congruenciais

Conhecendo uma raiz primitiva de M, é possível facilmente obter outros valores de a que conduzem a período completo considerando a regra abaixo:

Regra: Se a é raiz primitiva de M, então $b = a^k \mod M$ também é raiz primitiva de M se, e somente se, k e $M - 1$ forem primos relativos (**coprimos**), isto é, se o único divisor comum entre eles é o número 1.

Propriedades de Geradores Congruenciais

Conhecendo uma raiz primitiva de M, é possível facilmente obter outros valores de a que conduzem a período completo considerando a regra abaixo:

Regra: Se a é raiz primitiva de M, então $b = a^k \mod M$ também é raiz primitiva de M se, e somente se, k e $M - 1$ forem primos relativos (**coprimos**), isto é, se o único divisor comum entre eles é o número 1.

Exercício: Voltando ao exercício anterior, supomos que conhecemos que $a = 2$ é raiz primitiva de $M = 11$. Utilize a regra a cima para obter os valores de a que conduzem à período completo.

Propriedades de Geradores Congruenciais

Resposta:

Propriedades de Geradores Congruenciais

Resposta:

- $2^1 \bmod 11 = \mathbf{2}$;

Propriedades de Geradores Congruenciais

Resposta:

- $2^1 \bmod 11 = \mathbf{2}$;
- $2^3 \bmod 11 = \mathbf{8}$;

Resposta:

- $2^1 \bmod 11 = \mathbf{2}$;
- $2^3 \bmod 11 = \mathbf{8}$;
- $2^7 \bmod 11 = \mathbf{7}$;

Propriedades de Geradores Congruenciais

Resposta:

- $2^1 \bmod 11 = \mathbf{2}$;
- $2^3 \bmod 11 = \mathbf{8}$;
- $2^7 \bmod 11 = \mathbf{7}$;
- $2^9 \bmod 11 = \mathbf{6}$.

Resposta:

- $2^1 \bmod 11 = \mathbf{2}$;
- $2^3 \bmod 11 = \mathbf{8}$;
- $2^7 \bmod 11 = \mathbf{7}$;
- $2^9 \bmod 11 = \mathbf{6}$.

Assim, $a \in \{2, 6, 7, 8\}$ conduz a período completo.

Propriedades de Geradores Congruenciais

Teorema (Gerador congruencial linear misto, $c \neq 0$): Considere o gerador congruencial linear misto $x_i = (a * x_{i-1} + c) \bmod M$, com $c > 0$ (inteiro). Para $M = 2^\beta$ (β inteiro positivo), o gerador possui período completo (isto é, 2^β), se e somente se:

Propriedades de Geradores Congruenciais

Teorema (Gerador congruencial linear misto, $c \neq 0$): Considere o gerador congruencial linear misto $x_i = (a * x_{i-1} + c) \bmod M$, com $c > 0$ (inteiro). Para $M = 2^\beta$ (β inteiro positivo), o gerador possui período completo (isto é, 2^β), se e somente se:

c1) $a \equiv \bmod 4$ ($a - 1$ é divisível por 4);

Propriedades de Geradores Congruenciais

Teorema (Gerador congruencial linear misto, $c \neq 0$): Considere o gerador congruencial linear misto $x_i = (a * x_{i-1} + c) \bmod M$, com $c > 0$ (inteiro). Para $M = 2^\beta$ (β inteiro positivo), o gerador possui período completo (isto é, 2^β), se e somente se:

c1) $a \equiv \bmod 4$ ($a - 1$ é divisível por 4);

c2) $\bmod(c, M) = 1$ (c e M são primos relativos, isto é, coprimos).

Propriedades de Geradores Congruenciais

Teorema (Gerador congruencial linear misto, $c \neq 0$): Considere o gerador congruencial linear misto $x_i = (a * x_{i-1} + c) \bmod M$, com $c > 0$ (inteiro). Para $M = 2^\beta$ (β inteiro positivo), o gerador possui período completo (isto é, 2^β), se e somente se:

c1) $a \equiv \bmod 4$ ($a - 1$ é divisível por 4);

c2) $\bmod(c, M) = 1$ (c e M são primos relativos, isto é, coprimos).

Nota: Qualquer valor ímpar de c satisfaz a propriedade c2. Além disso, não é necessário que M seja um número primo.

Propriedades de Geradores Congruenciais

Exemplo: O gerador utilizado pelo compilador **gcc** da linguagem `C` é um gerador congruencial linear, tal que:

- $a = 69069$;
- $c = 5$ (é um gerador misto e satisfaz a propriedade c2);
- $M = 2^{32}$ (M não é primo).

Gerador Mersenne Twister

O gerador **Mersenne Twister** foi proposto por Makoto Matsumoto e Takuji Nishimura, 1998. O gerador é de longe o mais utilizado na computação e em estudos de simulações. Sendo assim, o Mersenne Twister é, normalmente, o gerador de números pseudo-aleatórios padrão de diversas linguagens, softwares e bibliotecas como é o caso das linguagens **R**,**Julia**, **Python**, **Ruby**, das bibliotecas **GNU Scientific Library (GSL)**, **GNU Multiple Precision Arithmetic Library**, **Cuda** e de softwares como **Mathematica** e **Maple**.

Gerador Mersenne Twister

Algumas características do gerador Mersenne Twister são:

Gerador Mersenne Twister

Algumas características do gerador Mersenne Twister são:

- Não é um gerador congruencial linear;

Gerador Mersenne Twister

Algumas características do gerador Mersenne Twister são:

- Não é um gerador congruencial linear;
- Produz observações geradas de forma pseudo-aleatória com alta qualidade;

Gerador Mersenne Twister

Algumas características do gerador Mersenne Twister são:

- Não é um gerador congruencial linear;
- Produz observações geradas de forma pseudo-aleatória com alta qualidade;
- É um gerador rápido;

Gerador Mersenne Twister

Algumas características do gerador Mersenne Twister são:

- Não é um gerador congruencial linear;
- Produz observações geradas de forma pseudo-aleatória com alta qualidade;
- É um gerador rápido;
- Passou por diversos testes estatísticos para mensurar sua qualidade;

Gerador Mersenne Twister

Algumas características do gerador Mersenne Twister são:

- Não é um gerador congruencial linear;
- Produz observações geradas de forma pseudo-aleatória com alta qualidade;
- É um gerador rápido;
- Passou por diversos testes estatísticos para mensurar sua qualidade;
- Tem período de ocorrência $2^{19937} - 1$;

Gerador Mersenne Twister

Algumas características do gerador Mersenne Twister são:

- Não é um gerador congruencial linear;
- Produz observações geradas de forma pseudo-aleatória com alta qualidade;
- É um gerador rápido;
- Passou por diversos testes estatísticos para mensurar sua qualidade;
- Tem período de ocorrência $2^{19937} - 1$;
- É baseado em números primos de Mersenne, isto é, se a é um número primo e $M_a = 2^a - 1$ também é, diremos que M_a é número primo de Mersenne.

Gerando Normal

Pelo método da inversão, necessitamos conhecer F_X^{-1} para geramos observações de X. Computacionalmente não há problemas em utilizarmos o método da inversão para gerarmos observações gaussianas, uma vez que que há implementado em `R` a a inversa da acumulada da normal que é chamada de **função erro**, em que

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \text{e}^{-t^2} dt.$$

Algoritmo do método de Box-Muller

Algoritmo do método de Box-Muller

1. Gere $U_1, U_2 \sim \mathcal{U}(0, 1)$;

Algoritmo do método de Box-Muller

1. Gere $U_1, U_2 \sim \mathcal{U}(0,1)$;
2. Faça $R^2 = -2 \log U_1$ (distribuição exponencial) e $S^2 = 2\pi U_2$ (distribuição uniforme);

Algoritmo do método de Box-Muller

1. Gere $U_1, U_2 \sim \mathcal{U}(0, 1)$;
2. Faça $R^2 = -2 \log U_1$ (distribuição exponencial) e $S^2 = 2\pi U_2$ (distribuição uniforme);
3. Retorne $X = R \cos(S^2)$ e $Y = R \sin(S^2)$.

As ocorrências X e Y são ocorrências da distribuição normal padrão.

Algoritmo do método de Box-Muller

1. Gere $U_1, U_2 \sim \mathcal{U}(0,1)$;
2. Faça $R^2 = -2\log U_1$ (distribuição exponencial) e $S^2 = 2\pi U_2$ (distribuição uniforme);
3. Retorne $X = R\cos(S^2)$ e $Y = R\sin(S^2)$.

As ocorrências X e Y são ocorrências da distribuição normal padrão.

Nota: Perceba que em cada execução completa do algoritmo geramos duas novas observações de $X \sim \mathcal{N}(0,1)$.

Gerando Normal

Algoritmo pelo método polar:

Gerando Normal

Algoritmo pelo método polar:

1. Gere $U_1, U_2 \sim \mathcal{U}(0, 1)$;

Gerando Normal

Algoritmo pelo método polar:

1. Gere $U_1, U_2 \sim \mathcal{U}(0,1)$;
2. Transforme $U_1 = 2U_1 - 1$, $U_2 = 2U_2 - 1$ e faça $W = U_1^2 + U_2^2$;

Gerando Normal

Algoritmo pelo método polar:

1. Gere $U_1, U_2 \sim \mathcal{U}(0,1)$;
2. Transforme $U_1 = 2U_1 - 1$, $U_2 = 2U_2 - 1$ e faça $W = U_1^2 + U_2^2$;
3. Se $W > 1$, volte ao primeiro passo;

Gerando Normal

Algoritmo pelo método polar:

1. Gere $U_1, U_2 \sim \mathcal{U}(0,1)$;
2. Transforme $U_1 = 2U_1 - 1$, $U_2 = 2U_2 - 1$ e faça $W = U_1^2 + U_2^2$;
3. Se $W > 1$, volte ao primeiro passo;
4. Retorne

$$X = \sqrt{\frac{-\log W}{W}} \times U_1 \text{ e } Y = \sqrt{\frac{-\log W}{W}} \times U_2.$$

Gerando Normal

Algoritmo pelo método polar:

1. Gere $U_1, U_2 \sim \mathcal{U}(0,1)$;
2. Transforme $U_1 = 2U_1 - 1$, $U_2 = 2U_2 - 1$ e faça $W = U_1^2 + U_2^2$;
3. Se $W > 1$, volte ao primeiro passo;
4. Retorne

$$X = \sqrt{\frac{-\log W}{W}} \times U_1 \text{ e } Y = \sqrt{\frac{-\log W}{W}} \times U_2.$$

A vantagem do algoritmo é que não requer avaliação de funções trigonométricas. Porém, a desvantagem é que normalmente precisamos gerar mais de duas uniformes para obter duas ocorrências gaussianas.

Gerando Normal

Exercício: Implemente uma função para geração de números pseudo-aleatórios com distribuição normal padrão. A função deverá implementar o método de Box-Muller e o método polar. Ao final obtenha um histograma com os números gerados (mil valores) e realize um teste de normalidade.

Importando dados

Importando dados

A linguagem R apresenta diversas estruturas de dados, sendo o data frame uma das estruturas mais utilizadas.

Como já mencionado em aulas anteriores, a forma mais conveniente de organizar uma base de dados de modo a facilitar as análises estatísticas é considerar um data frame. No artigo, **Tidy Data** publicado no **Journal Statistical Software** (2014), Hadley Wickham decorre sobre o assunto abordando a dificuldade de se organizar uma base de dados e como tornar o tratamento dos dados tão fácil e eficaz quanto possível.

Consulte aulas anteriores para maiores detalhes sobre o autor.

Importando dados

Diversos pacotes apresentam conjunto de dados como componente da estrutura do pacote. Normalmente estes dados empacotados servem para serem utilizados nos exemplos das funções implementadas pelos pacotes, podendo assim ser bastante útil no estudo de algumas funcionalidades da linguagem `R`. Normalmente esses conjuntos de dados apresentam a estrutura de data frame.

Nota: Para obtermos um vetor com os nomes dos dados disponíveis com os pacotes até então carregados, rode o comando `data()`.

Importando dados

Na estatística, é muito comum encontrarmos base de dados disponíveis na internet no formato de texto. O formato de texto mais comum para base de dados disponibilizadas à sociedade por diversos órgãos nacionais e internacionais é o **C**omma-**S**eparated **V**alues (**CSV**).

É bastante comum na vida de um estatístico ou analista de dados a situação de **não** termos acesso irrestrito à um **SGBD** (Sistema Gerenciador de Banco de Dados) ao qual gerencia dados de interesse ao qual iremos analisar.

Importando dados

Dessa forma, muitas organizações governamentais e privadas (no caso de organizações privadas é muito raro a disponibilização de dados) disponibilizam seus dados sem comprometer os dados originais, disponibilizando assim cópias de parte de dados em formatos alternativos.

Por exemplo, em âmbito nacional, o **DATASUS** (Departamento de Informática do Sistema Único de Saúde), **IBGE** (Instituto Brasileiro de Geografia e Estatística) e **IPEA** (Instituto de Pesquisa Econômica Aplicada) são exemplos de organizações governamentais que disponibilizam seus dados no formato **CSV**.

Importando dados

A não ser que o estatístico ou o profissional que irá fazer a análise de dados tenha a oportunidade de trabalhar diretamente com o **SGBD** que gerencia os dados originais, o formato **CSV** será disparadamente o formato de dados mais utilizado.

Observação: De toda forma, nada impede que o profissional monte uma estrutura paralela utilizando algum **SGBD** para organizar de forma eficiente uma cópia dos dados originais permitindo que possa trabalhar com essa estrutura paralela como se estivesse trabalhando com a estrutura original.

Nota: O arquivo **CSV** é um arquivo de texto em que as variáveis são separadas por algum caractere que normalmente poderá ser vírgula, ponto e vírgula ou espaço.

Importando dados

```
Name,City,Country
William,Yamrat,Nigeria
Diana,Jalasenga,Indonesia
Evelyn,Gourma Rharous,Mali
Christina,Sovetskiy,Russia
Christopher,ItororΓ³,Brazil
Amanda,ViΔΌΔni,Latvia
Shawn,Kokterek,Kazakhstan
Adam,Pontivy,France
Frank,Gaoleshan,China
Dennis,Anping,China
```

Figura: Arquivo **CSV** (deverá ser salvo com extensão .csv) com as variáveis separadas por vírgula.

Importando dados

Exemplo: Nesse exemplo criaremos um arquivo texto (com extensão **CSV**). Note que não precisamos abrir um editor de texto para salvar as informações, que nesse caso estão separadas por vírgula.

```
1  # Minhas variaveis
2  ID <- 1:10
3  nomes <- c("Pedro","Rafael","Maria","Walter","Marina",
4             "Jose", "Renata","Sabrina","Luiz","Paulo")
5  curso <- c("Estatistica","Matematica","Letras",
6             "Estatistica", "Matematica","Historia",
7             "Letras","Quimica","Quimica","Geografia")
8  cre <- c(8.3,7.7,9.1,7.8,7.1,9.4,8.9,8.8,9.7,7.3)
9
10 # Criando um data frame
11 informacoes <- data.frame(ID,nomes,curso,cre)
```

Importando dados

```
12 # Criando um nome para um arquivo temporario.
13 dados <- tempfile()
14
15 # Salvando meu arquivo CSV.
16 write.table(x = informacoes,file = paste(dados,".csv",
17              sep = ""), append = FALSE, sep = ",",
18              dec = ".", row.names = FALSE)
19
20 # Visualizando meu arquivo de texto.
21 file.show(paste(dados,".csv", sep = ""))
```

Importando dados

Exercício: Estude a documentação da função `write.table()` e discuta sobre os argumentos da função utilizados no exemplo anterior e como eles alteram o comportamento da função.

Exemplo: Caso venhamos precisar editar o conjunto de dados no formato de texto (**CSV**, por exemplo), poderemos fazer isso utilizando a função `file.edit()`. A partir dos objetos criados no exemplo anterior, temos:

```
1 file.edit(paste(dados,".csv", sep = ""))
```

Com o comando acima, teremos acesso ao arquivo e `paste(dados,".csv", sep = )` e poderemos fazer edição nesse arquivo diretamente no ambiente de programação, que no nosso caso é o **RStudio**.

Importando dados

```
"ID","nomes","curso","cre"
1,"Pedro","Estatística",8.3
2,"Rafael","Matemática",7.7
3,"Maria","Letras",9.1
4,"Walter","Estatística",7.8
5,"Marina","Matemática",7.1
6,"José","História",9.4
7,"Renata","Letras",8.9
8,"Sabrina","Química",8.8
9,"Luiz","Química",9.7
10,"Paulo","Geografia",7.3
```

Figura: Arquivo de texto no formato **CSV** para ser alterado manualmente.

Importando dados

```
"ID","nomes","curso","cre","Situacao"
1,"Pedro","Estatística",8.3,"bom"
2,"Rafael","Matemática",7.7,"razoavel"
3,"Maria","Letras",9.1,"otimo"
4,"Walter","Estatística",7.8,"razoavel"
5,"Marina","Matemática",7.1,"razoavel"
6,"José","História",9.4,"otimo"
7,"Renata","Letras",8.9,"bom"
8,"Sabrina","Química",8.8,"bom"
9,"Luiz","Química",9.7,"otimo"
10,"Paulo","Geografia",7.3,"razoavel"
```

Figura: Acrescentando a variável situação ao conjunto de dados no formato **CSV**.

Importando dados

A princípio estávamos trabalhando com o objeto `informacao` que refere-se a um data frame com 10 linhas e 4 colunas (variáveis). Esse data frame foi salvo no formato **CSV** para possivelmente ser utilizado futuramente.

Nota: Detalhes sobre o diretório e nome do arquivo encontra-se no objeto `dados`. Note também que o objeto `informacao` não está atualizado com as novas informações incorporadas no arquivo salvo.

Importando dados

Exemplo: Para atualizar o arquivo, precisamos ler novamente os dados modificados e salvar uma cópia dos dados no objeto `informacao`. Assim, teremos nosso objeto atualizado com as novas informações. Façamos da seguinte forma:

```
1 informacao <- read.table(paste(dados,".csv", sep = ""),
2                          sep = ",", header = TRUE)
3 informacao
4 #>      ID   nomes        curso cre Situacao
5 #> 1     1   Pedro  Estatistica 8.3      bom
6 #> 2     2  Rafael   Matematica 7.7 razoavel
7 #> 3     3   Maria       Letras 9.1    otimo
8 #> 4     4  Walter  Estatistica 7.8 razoavel
9 #> ...
```

Importando dados

Note que a função `tempfile()` retorna um vetor de caracteres referente ao diretório e nome do arquivo temporário. Porém, poderíamos alterar o diretório informando diretamente ao argumento `file` da função `write.table()` o diretório com o nome do arquivo a ser salvo.

Exemplo: Ainda utilizando os objetos criados nos exemplos anteriores, corra o código:

```
1 write.table(x = informacoes, file = "teste.csv",
2               append = FALSE, sep = ",", dec = ".",
3               row.names = FALSE)
```

Importando dados

Em qual diretório o arquivo `teste.csv` foi salvo?

Importando dados

Em qual diretório o arquivo `teste.csv` foi salvo?

Resposta: Uma vez que no código acima não especificamos o diretório ao qual o arquivo **teste.csv** será salvo, será considerado o diretório corrente de trabalho de processos `R`. Para saber qual o diretório corrente, faça `getwd()`.

Poderemos alterar esse diretório utilizando a função `setwd()`, ao qual passamos como argumento o diretório que queremos considerar como diretório padrão de trabalho.

Importando dados

Em qual diretório o arquivo `teste.csv` foi salvo?

Resposta: Uma vez que no código acima não especificamos o diretório ao qual o arquivo **teste.csv** será salvo, será considerado o diretório corrente de trabalho de processos `R`. Para saber qual o diretório corrente, faça `getwd()`.

Poderemos alterar esse diretório utilizando a função `setwd()`, ao qual passamos como argumento o diretório que queremos considerar como diretório padrão de trabalho.

Importante: Não é necessário alterar o diretório padrão de trabalho para salvar um objeto em um diretório distinto ao diretório padrão. Por exemplo, no código do exemplo anterior, poderíamos especificar o caminho completo do diretório para salvar o arquivo `teste.csv`.

Importando dados

Exercício: Salve o objeto `informacao` no arquivo de nome **teste1.csv** em um diretório qualquer que não seja o diretório padrão de trabalho.

É bastante útil instruir o download dos dados, uma vez que supostamente teremos acesso ao link que direciona ao arquivo que precisaremos tratar e analisar. Um pacote que fornece funções bastante útil para essa tarefa é o pacote **RCurl** (instale o pacote).

Exemplo: O código abaixo efetua o download de um conjunto de dados livre disponibilizado pelo governo americano no . Inicialmente um objeto

Importando dados

```
1  library(RCurl)
2
3  url1 = 'https://chronicdata.cdc.gov/api/views'
4  url2 = '/g4ie-h725/rows.csv?accessType=DOWNLOAD'
5  url <- paste(url1,url2,sep="")
6  dados <- getURL(url)
7  out <- read.csv(textConnection(dados), header = TRUE)
8  rm(dados) # Removendo os dados
9  gc() # Executa a coleta de lixo.
10 # Salvando o conjunto de dados no diretotrio de trabalho
       padrao
11 write.table(x = out, file = "dados.csv", sep = ";")
```

Importando dados

Nota: A função `gc()` é bastante útil para recuperar uma área da memória inutilizada por um objeto ou programa. O programador deve especificar explicitamente quando e quais objetos devem ser desalocados e retornados ao sistema. A especificação dos objetos que devem ser removidos é realizada com a função `rm()`. Como **regra de bolso**, é sempre útil correr a função `gc()` quando eliminamos um objeto muito grande.

Sumarizando dados

Ao se ter um conjunto de dados importados no `R` precisamos começar a analisá-los. Sendo assim, é imprescindível realizar um estudo descritivo dos dados.

A critério de exemplo, carregue o conjunto de dados **CO2** que vem com a linguagem `R` fazendo `data(CO2)`.

Sumarizando dados

Nota: O conjunto de dados **CO2** é formado pela concentração atmosférica de CO2 expressas em partes por milhão (ppm) relatadas na escala de fração molar manométrica de 1997. Os dados apresentam observações entre 1959 à 1997 do estado da Mississippi, Estados Unidos e a província de Quebec no Canadá. Maiores detalhes sobre os dados podem ser encontrado fazendo `help(CO2)`.

Exemplo: Uma das informações iniciais que desejamos saber em um conjunto de dados é sua dimensão.

```
1 nrow(CO2) # Foi visto em aulas anteriores
2 #> [1] 84
3
4 ncol(CO2) # Foi visto em aulas anteriores
5
6 #> [1] 5
```

Sumarizando dados

Poderia ter sido utilizado a função `dim()` para obter um vetor com as dimensões dos dados, número de linhas e colunas, respectivamente.

Uma outra função bastante utilizada para sumarizar um conjunto de dados (vetor, array, data frame, lista e fator) é a função `summary()`. A função `summary()` fornecerá algumas estatísticas descritivas básicas. Sempre será conveniente inspecionar o conjunto de dados utilizando esteja função.

Sumarizando dados

Por exemplo, com a estatística descritiva, podemos responder algumas perguntas:

Sumarizando dados

Por exemplo, com a estatística descritiva, podemos responder algumas perguntas:

1. Qual a dimensão dos dados?

Sumarizando dados

Por exemplo, com a estatística descritiva, podemos responder algumas perguntas:

1. Qual a dimensão dos dados?
2. Qual o nível de mensuração das variáveis de interesse?

Sumarizando dados

Por exemplo, com a estatística descritiva, podemos responder algumas perguntas:

1. Qual a dimensão dos dados?
2. Qual o nível de mensuração das variáveis de interesse?
3. Qual o domínio dos dados?

Sumarizando dados

Por exemplo, com a estatística descritiva, podemos responder algumas perguntas:

1. Qual a dimensão dos dados?
2. Qual o nível de mensuração das variáveis de interesse?
3. Qual o domínio dos dados?
4. Os dados são simétricos ou assimétricos?

Sumarizando dados

Por exemplo, com a estatística descritiva, podemos responder algumas perguntas:

1. Qual a dimensão dos dados?
2. Qual o nível de mensuração das variáveis de interesse?
3. Qual o domínio dos dados?
4. Os dados são simétricos ou assimétricos?
5. Os dados estão posicionados em torno de qual valor?

Sumarizando dados

Por exemplo, com a estatística descritiva, podemos responder algumas perguntas:

1. Qual a dimensão dos dados?
2. Qual o nível de mensuração das variáveis de interesse?
3. Qual o domínio dos dados?
4. Os dados são simétricos ou assimétricos?
5. Os dados estão posicionados em torno de qual valor?
6. Qual o nível de concentração dos dados em torno de um ponto de interesse?

É por meio da estatística descritiva que postulamos uma classe de modelos para a distribuição dos dados.

Sumarizando dados

Em resumo, a estatística descritiva é uma etapa fundamental que antecede o estudo inferencial. Basicamente a estatística descritiva refere-se a um conjunto de técnicas e gráficos que visam descrever e sumarizar os dados. Sendo assim, a estatística descritiva responde apenas sobre a amostra e não sobre a população. Porém, para se realizar uma boa inferência com respeito à população será necessário entender a amostra por meio de um estudo descritivo.

Sumarizando dados

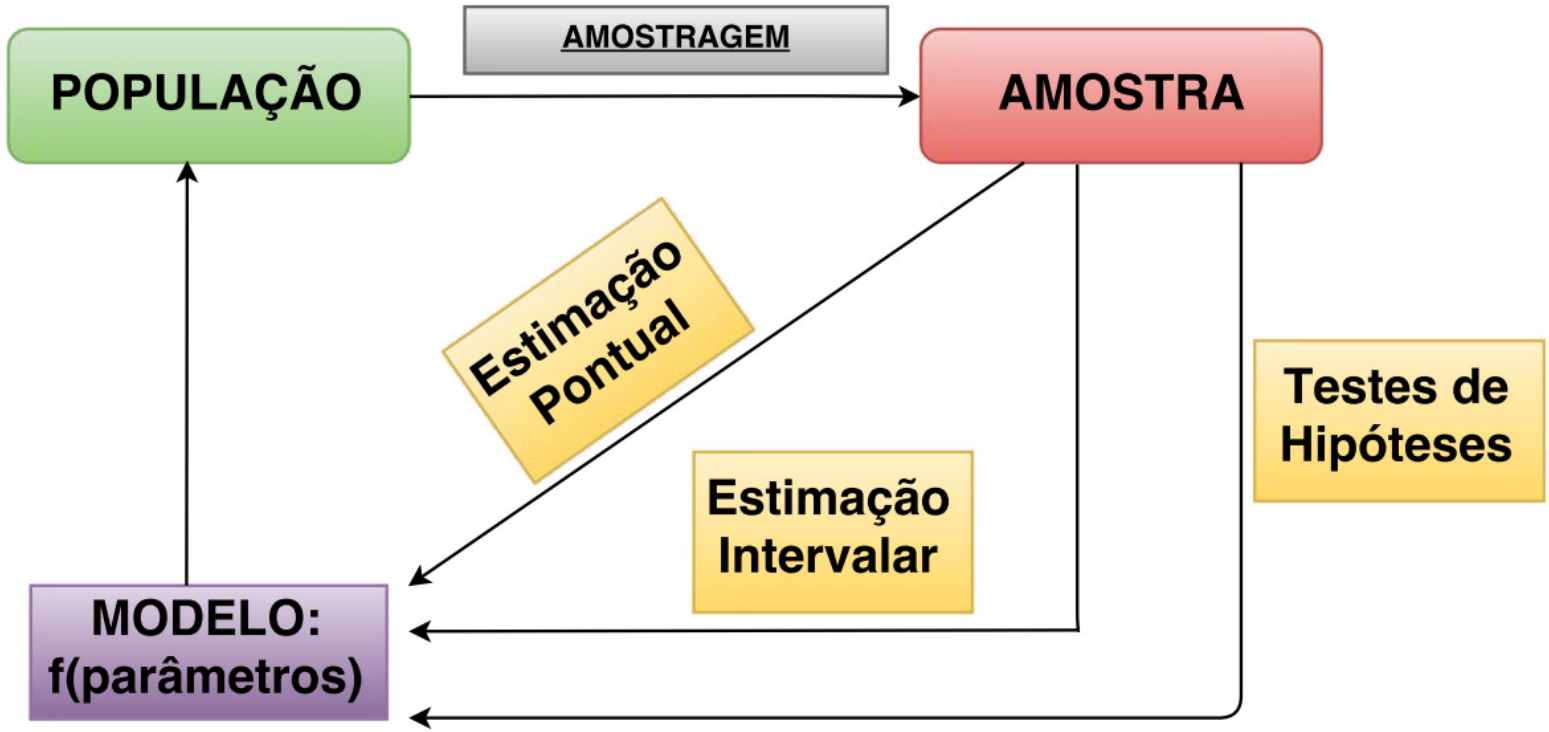


Figura: Visão macro de como utilizamos a estatística.

A estatística descritiva é um estudo que antecede o estudo inferencial (estimação pontual, estimação intervalar e testes de hipóteses).

Sumarizando dados

A função `summary()` é uma das funções mais versáteis da linguagem `R` e poderá ser aplicada em diversas situações, por exemplo, a resultados de funções `lm()` e `glm()` utilizadas para previsões de modelos lineares e modelos lineares generalizados, respectivamente.

```
1 summary(CO2)
```

Sumarizando dados

```
    Plant            Type         Treatment       conc          uptake
Qn1    : 7   Quebec      :42   nonchilled:42   Min.   :  95   Min.   : 7.70
Qn2    : 7   Mississippi:42   chilled    :42   1st Qu.: 175   1st Qu.:17.90
Qn3    : 7                                     Median : 350   Median :28.30
Qc1    : 7                                     Mean   : 435   Mean   :27.21
Qc3    : 7                                     3rd Qu.: 675   3rd Qu.:37.12
Qc2    : 7                                     Max.   :1000   Max.   :45.50
(Other):42
```

Figura: Saída da função `summary()` aplicada aos dados **CO2**.

Dica: Fazer um histograma das variáveis quantitativas é uma boa estratégia para se entender as variáveis de interesses no conjunto de dados.

Sumarizando dados

Muitas vezes queremos observar parte do conjunto de dados para se ter uma ideia visual de como os dados a serem analisados estão organizados. Porém, é comum que a base de dados tenha dimensões altas, dificultando assim a visualização de toda a base. Para se ter essa ideia visual do formato do conjunto de dados poderemos utilizar as funções:

Sumarizando dados

Muitas vezes queremos observar parte do conjunto de dados para se ter uma ideia visual de como os dados a serem analisados estão organizados. Porém, é comum que a base de dados tenha dimensões altas, dificultando assim a visualização de toda a base. Para se ter essa ideia visual do formato do conjunto de dados poderemos utilizar as funções:

1. `head()`: Visualizamos o topo do conjunto de dados. Dessa forma poderemos visualizar os nomes das variáveis que possivelmente iremos analisar;

Sumarizando dados

Muitas vezes queremos observar parte do conjunto de dados para se ter uma ideia visual de como os dados a serem analisados estão organizados. Porém, é comum que a base de dados tenha dimensões altas, dificultando assim a visualização de toda a base. Para se ter essa ideia visual do formato do conjunto de dados poderemos utilizar as funções:

1. `head()`: Visualizamos o topo do conjunto de dados. Dessa forma poderemos visualizar os nomes das variáveis que possivelmente iremos analisar;

2. `tail()`: Visualizamos o final do conjunto de dados.

Sumarizando dados

	Plant	Type	Treatment	conc	uptake
1	Qn1	Quebec	nonchilled	95	16.0
2	Qn1	Quebec	nonchilled	175	30.4
3	Qn1	Quebec	nonchilled	250	34.8
4	Qn1	Quebec	nonchilled	350	37.2
5	Qn1	Quebec	nonchilled	500	35.3
6	Qn1	Quebec	nonchilled	675	39.2

Figura: Saída da função `head()` aplicada aos dados **CO2** (topo do conjunto de dados).

Sumarizando dados

	Plant	Type	Treatment	conc	uptake
79	Mc3	Mississippi	chilled	175	18.0
80	Mc3	Mississippi	chilled	250	17.9
81	Mc3	Mississippi	chilled	350	17.9
82	Mc3	Mississippi	chilled	500	17.9
83	Mc3	Mississippi	chilled	675	18.9
84	Mc3	Mississippi	chilled	1000	19.9

Figura: Saída da função `tail()` aplicada aos dados **CO2** ("cauda" do conjunto de dados).

Sumarizando dados

Nota: Lembre-se que podemos utilizar a função `str()` para obter uma sumarização da estrutura dos dados que poderá nos fornecer algumas informações úteis. Para maiores detalhes, buscar a documentação da função.

Exercício: Utilizando o conjunto de dados **CO2**, faça:

Sumarizando dados

Nota: Lembre-se que podemos utilizar a função `str()` para obter uma sumarização da estrutura dos dados que poderá nos fornecer algumas informações úteis. Para maiores detalhes, buscar a documentação da função.

Exercício: Utilizando o conjunto de dados **CO2**, faça:

a) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é igual à Qn1;

Sumarizando dados

Nota: Lembre-se que podemos utilizar a função `str()` para obter uma sumarização da estrutura dos dados que poderá nos fornecer algumas informações úteis. Para maiores detalhes, buscar a documentação da função.

Exercício: Utilizando o conjunto de dados **CO2**, faça:

a) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é igual à Qn1;
b) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é Qn1 e Mn2;

Sumarizando dados

Nota: Lembre-se que podemos utilizar a função `str()` para obter uma sumarização da estrutura dos dados que poderá nos fornecer algumas informações úteis. Para maiores detalhes, buscar a documentação da função.

Exercício: Utilizando o conjunto de dados **CO2**, faça:

a) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é igual à Qn1;
b) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é Qn1 e Mn2;
c) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é Qc1 e $174 \leq$ **conc** ≤ 670;

Sumarizando dados

Nota: Lembre-se que podemos utilizar a função `str()` para obter uma sumarização da estrutura dos dados que poderá nos fornecer algumas informações úteis. Para maiores detalhes, buscar a documentação da função.

Exercício: Utilizando o conjunto de dados **CO2**, faça:

a) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é igual à Qn1;
b) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é Qn1 e Mn2;
c) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é Qc1 e $174 \leq$ **conc** ≤ 670;
d) Obtenha uma sumarização dos dados para o subconjunto dos dados em que a variável **Plant** é Qc1 e $174 \leq$ **conc** ≤ 670 e **uptake** é menor que 35.

Sumarizando dados

Resposta:

```
1  # Resposta do item a) do exercicio acima:
2  summary(CO2[CO2$Plant == "Qn1", ])
3
4  # Resposta do item b) do exercicio acima:
5  summary(CO2[CO2$Plant == "Qn1" | CO2$Plant == "Mn2", ])
6
7  # Resposta do item c) do exercicio acima:
8  summary(CO2[CO2$Plant == "Qn1" & (CO2$conc >= 174  & CO2$
       conc <= 670), ])
9
10 # Resposta do item d) do exercicio acima:
11 summary(CO2[CO2$Plant == "Qn1" & (CO2$conc >= 174  & CO2$
       conc <= 670) & CO2$uptake < 35, ])
```

O professor adverte

Ao persistirem dúvidas
as aulas anteriores deverão ser
consultadas

Sumarizando dados

Também é possível utilizar a função `subset()` para acessar um subconjunto de dados `R`.

Exercício: Leia a documentação da função `subset()`.

Sumarizando dados

Também é possível utilizar a função `subset()` para acessar um subconjunto de dados `R`.

Exercício: Leia a documentação da função `subset()`.

Exercício: Refaça o exercício anterior utilizando a função `subset()`.

Análise de Correspondência

As primeiras considerações matemáticas a respeito da **Análise de Correspondência** (AC) foram feitas por Hirschfeld em 1935:

H.O.P. Hirschfeld (1935). **A connection between correlation and contingency**, Cambridge Philosophical Sot. Proc. (Math. Proc.) 31, 520-524.

Alguns autores definem AC como um método de análise fatorial para variáveis categóricas. A AC foi primeiramente utilizada por Fisher em 1940 para análise de tabelas de contingência. Atualmente a técnica tem sido bastante empregada em diversas áreas do conhecimento, com ênfase maior na área de big data, em estudos de psicologia, neurociência, psiquiatria, entre outras áreas.

Análise de Correspondência

Uma aplicação comum da metodologia de AC tem sido na redução da dimensão dos dados, obtida através do mapeamento perceptual das relações de inter-dependência de informações amostrais que, na maioria das vezes, representam dados não métricos.

Importante

Quando é possível construir uma tabela de contingência em que cada linha e coluna da tabela de contingência possui categorias aos quais queremos buscar por relações, a AC poderá ser utilizada. Normalmente o retorno de uma AC é um gráfico em que a proximidade entre as categorias indica o nível de associação entre tais objetos **no âmbito amostral**.

Análise de Correspondência

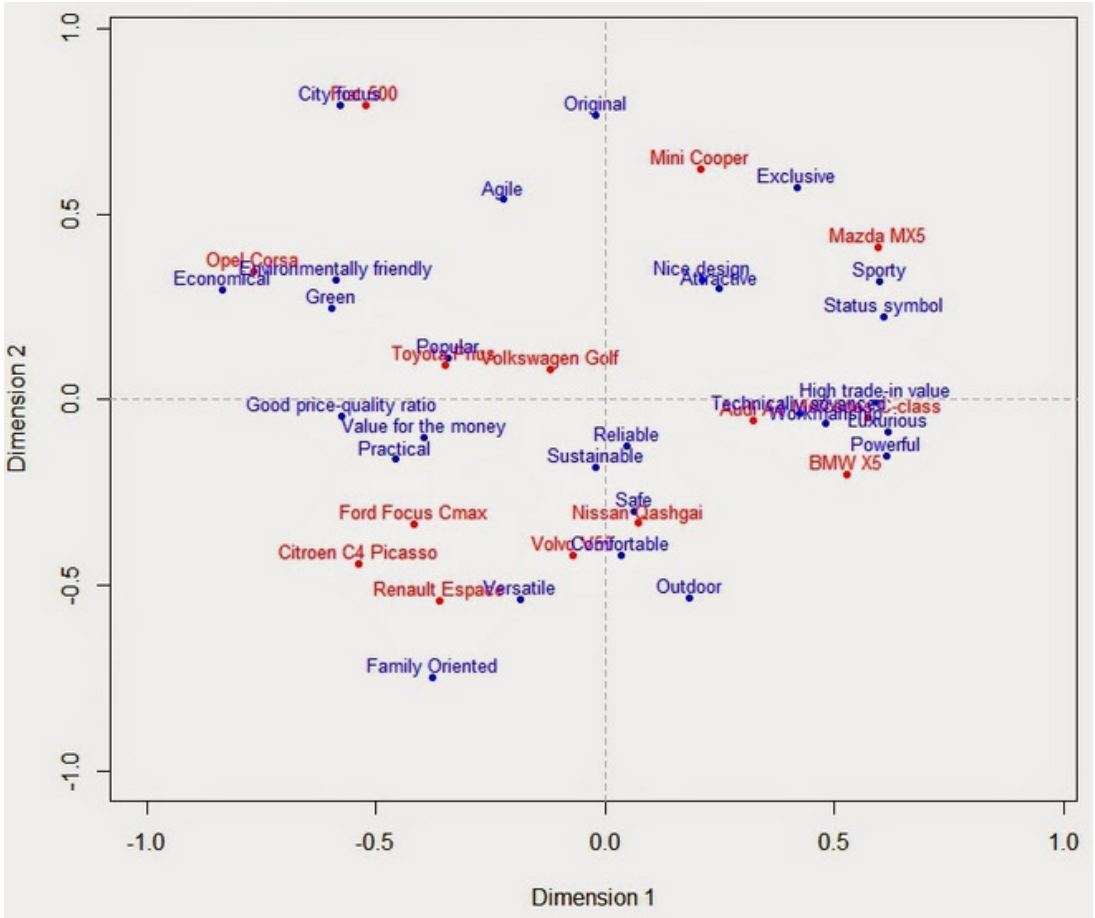


Figura: Gráfico da AC agrupando categorias de carros (modelos de carro) com características de interesse (categorias de interesse).

Análise de Correspondência

Nota: Em tempos em que os dados a serem analisados possuem grandes dimensões, reduzir a dimensão dos dados é algo muito necessário.

A AC é um meio de representar as linhas e colunas de uma tabela de contingência por pontos no espaço por meio de uma matriz de entrada retangular.

O tipo mais comum de entrada em uma AC é uma tabela de contingência com categorias específicas definindo as linhas e colunas. Para o caso da AC Simples, teremos uma tabela de dupla entrada.

Tabela: Estrutura dos dados para a Análise de Correspondência.

				B			
A	1	2	...	j	...	J	Total Linha
1	n_{11}	n_{12}	$\cdots$	n_{1j}	$\cdots$	n_{1J}	n_{1+}
2	n_{21}	n_{22}	$\cdots$	n_{2j}	$\cdots$	n_{2J}	n_{2+}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	n_{i1}	n_{i2}	$\cdots$	n_{ij}	$\cdots$	n_{iJ}	n_{i+}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
I	n_{I1}	n_{I2}	$\cdots$	n_{Ij}	$\cdots$	n_{IJ}	n_{I+}
Total Coluna	n_{+1}	n_{+2}	$\cdots$	n_{+j}	$\cdots$	n_{+J}	N

em que **A** é um conjunto de *I* categorias e **B** é um outro conjunto de *J* categorias. Além disso, temos que:

Análise de Correspondência

1. n_{ij}: refere-se a frequência absoluta observada de dados pertencentes à i-ésima categoria do conjunto **A** e à j-ésima categoria da do conjunto **B**;
2. n_{i+}: é a frequência total observada na i-ésima categoria de **A**;
3. n_{+j}: é a frequência total observada na i-ésima categoria de **B**;
4. N: é o total geral de frequências observadas.

Análise de Correspondência

Seja M a matriz de **frequências absolutas** observada na forma acima, isto é, $M = [n_{ij}]_{I \times J}$. Então, a matriz de **frequências relativas** P é dada por $P = \frac{1}{N}M$ e é chamada de **matriz de correspondência**, em que cada linha e coluna de P é um vetor de proporções.

Tabela: Matriz de correspondência.

	B						
A	1	2	...	j	...	J	Total Linha
1	p_{11}	p_{12}	$\cdots$	p_{1j}	$\cdots$	p_{1J}	p_{1+}
2	p_{21}	p_{22}	$\cdots$	p_{2j}	$\cdots$	p_{2J}	p_{2+}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	p_{i1}	p_{i2}	$\cdots$	p_{ij}	$\cdots$	p_{iJ}	p_{i+}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
I	p_{I1}	p_{I2}	$\cdots$	p_{Ij}	$\cdots$	p_{IJ}	p_{I+}
Total Coluna	p_{+1}	p_{+2}	$\cdots$	p_{+j}	$\cdots$	p_{+J}	1

em que $p_{ij} = \frac{n_{ij}}{N}$, $p_{i+} = \frac{n_{i+}}{N}$ e $p_{+j} = \frac{n_{+j}}{N}$.

Os vetores de frequências relativas marginais é denominado de **vetor de massas** e são calculados em relação ao total geral N. Sendo assim, temos que a massa da i-ésima linha é $\frac{n_{i+}}{N}$ e a massa da j-ésima coluna é definido por $\frac{n_{+j}}{N}$. Com as massas calculadas para linhas e colunas poderemos definir os vetores para linha e coluna (r e c), respectivamente. Assim,

$$\begin{aligned} r &= [p_{1+}, p_{2+}, \ldots, p_{I+}] \\ &\text{e} \\ c &= [p_{+1}, p_{+2}, \ldots, p_{+J}]. \end{aligned}$$

Algoritmo:

1. Obtenha a matriz de proporção esperada dada por $P_{esp} = rc^{'}$, $D_r = \text{diag}\{r\}$ e $D_c = \text{diag}\{c\}$;
2. Calcule $L = D_r^{-1/2} * (P - P_{esp}) * D_c^{-1/2}$;
3. Realize a decomposição Singular **Value Decomposition of a Matrix** da matriz L. **Dica**: Utilize a função `svd()` da linguagem `R`. Assim, $L = UDV^{'}$, em que U, D (matriz diagonal) e V são e obtidas especificadas na documentação da função `svd()`.
4. Faça $X = D_r^{-1} * \sqrt{D_r} * U * D$ e $Y = D_c^{-1} * \sqrt{D_c} * V * D$.

Análise de Correspondência

Exercício: Implemente por meio do algoritmo uma função que realize uma análise de correspondência para a matriz M de dupla entrada. Por exemplo, considere M como sendo:

```
1 M <- matrix(c(30, 53,73, 20, 46, 45, 16, 10, 4, 1, 6, 36, 6,
      28, 10, 16, 41, 1, 37, 59, 169, 39, 2, 1, 4, 13, 10, 5)
    , nrow=7) # Matriz contigencia.
2 dimnames(M)[[1]] <- c("P0","P1","P2","P3","P4","P5","P6")
3 dimnames(M)[[2]] <- c("A","B","C","D")
```

Análise de Correspondência

Apos obter X e Y como especificado no algoritmo acima, construa o gráfico da seguinte forma:

```
1 plot(c(-1,1), c(-1,1), type="n", xlab="1o. eixo", ylab="2o.
      eixo", main="Analise de Correspondencia", cex.main=0.8)
2 lines(c(-1,1),c(0,0))
3 lines(c(0,0),c(-1,1))
4 points(X, pch=19, col="red")
5 text(X, pos=3, labels = dimnames(M)[[1]],col="red", cex=0.8)
6 points(Y, pch=15, col="blue")
7 text(Y, pos=3, labels = dimnames(M)[[2]],col="blue", cex
      =0.8)
```

Análise de Correspondência

Solução:

Solução:

```
17 ac <- function(M,...){
18     N <- sum(M)
19     MP <- M/N # Matriz de correspondencia.
20     r <- rep(NA,nrow(M)) # vetor r
21     c <- rep(NA,ncol(M)) # vetor c
22
23     r <- as.vector(apply(MP,1,sum)) # Aqui esta sendo
           construido o vetor r.
24     c <- as.vector(apply(MP,2,sum)) # Aqui esta sendo
           construido o vetor c.
25
26     P_esp <- r%*%t(c)
27     D_r <- diag(r) # Matriz diagonal do vetor r.
28     D_c <- diag(c) # Matriz diagonal do vetor c.
29     L <- solve(sqrt(D_r))%*%(MP-P_esp)%*%solve(sqrt(D_c)) #
           Queremos minimizar isso aqui.
30     U <- svd(L)$u
31     V <- svd(L)$v
32     D <- diag(svd(L)$d)
```

Análise de Correspondência

```
33      X <- solve(D_r)%*%sqrt(D_r)%*%U%*%D
34      Y <- solve(D_c)%*%sqrt(D_c)%*%V%*%D
35
36      plot(c(-1,1), c(-1,1), type="n", xlab="1o. eixo", ylab="
            2o. eixo", main="AC", cex.main=0.8,...)
37      lines(c(-1,1),c(0,0), lty = 2)
38      lines(c(0,0),c(-1,1), lty = 2)
39      points(X, pch=19, col="red")
40      text(X, pos=3, labels = dimnames(M)[[1]],col="red", cex
            =0.8)
41      points(Y, pch=15, col="blue")
42      text(Y, pos=3, labels = dimnames(M)[[2]],col="blue", cex
            =0.8)
43  }
```

Análise de Correspondência

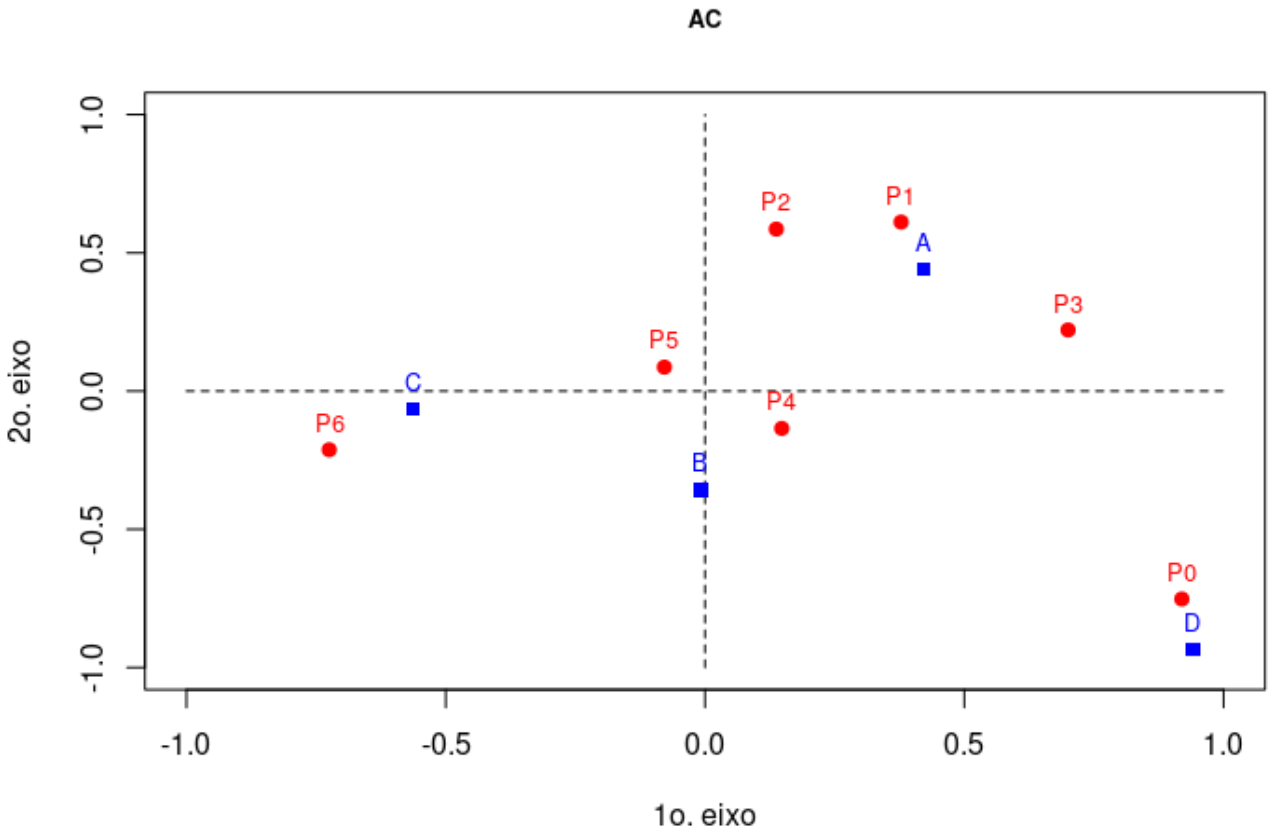


Figura: Gráfico perceptual para as categorias da matriz M do exercício anterior.

Identificando Grupos

Após sumarizar o conjunto de dados, poderemos realizar diversos estudos estatísticos. Por exemplo, poderíamos estar interessados em investigar similaridades entre elementos do conjunto de dados. Por exemplo, se cada linha de um data frame é um indivíduo e cada coluna refere-se a variáveis desse indivíduo, poderíamos estar interessados em reunir esses indivíduos em grupos. Sendo assim, queremos que indivíduos que pertencem ao mesmo grupo, sejam similares segundo suas características.

Identificando Grupos

Após sumarizar o conjunto de dados, poderemos realizar diversos estudos estatísticos. Por exemplo, poderíamos estar interessados em investigar similaridades entre elementos do conjunto de dados. Por exemplo, se cada linha de um data frame é um indivíduo e cada coluna refere-se a variáveis desse indivíduo, poderíamos estar interessados em reunir esses indivíduos em grupos. Sendo assim, queremos que indivíduos que pertencem ao mesmo grupo, sejam similares segundo suas características.

Em análise de agrupamento, estamos interessados em minimizar a variância dentro dos grupos o que equivale a maximizar a variância entre os grupos, isto é, queremos homogeneidade entre os grupos e heterogeneidade entre os grupos.

Identificando Grupos

Após sumarizar o conjunto de dados, poderemos realizar diversos estudos estatísticos. Por exemplo, poderíamos estar interessados em investigar similaridades entre elementos do conjunto de dados. Por exemplo, se cada linha de um data frame é um indivíduo e cada coluna refere-se a variáveis desse indivíduo, poderíamos estar interessados em reunir esses indivíduos em grupos. Sendo assim, queremos que indivíduos que pertencem ao mesmo grupo, sejam similares segundo suas características.

Em análise de agrupamento, estamos interessados em minimizar a variância dentro dos grupos o que equivale a maximizar a variância entre os grupos, isto é, queremos homogeneidade entre os grupos e heterogeneidade entre os grupos.

Um das classe de métodos de agrupamento mais utilizada é a classe de métodos não hierárquicos.

Diagrama de Voronoi - Métodos Não Hierárquicos

Diagrama de Voronoi

Diferentemente dos métodos **hierárquicos** em que as hierarquias são apresentadas em gráficos de árvore (dendograma), os clusters pelos métodos **não hierárquicos** são visualizados pelo **Diagrama Voronoi**.

Diagrama de Voronoi - Métodos Não Hierárquicos

Diagrama de Voronoi

Diferentemente dos métodos **hierárquicos** em que as hierarquias são apresentadas em gráficos de árvore (dendograma), os clusters pelos métodos **não hierárquicos** são visualizados pelo **Diagrama Voronoi**.

Definição formal do Diagrama de Voronoi

Seja $\mathbf{X} \neq \varnothing$ e $(\mathbf{X}, \mathbf{d})$ um espaço métrico munido de uma métrica d. Seja K o conjunto de índices e $(P_k)_{k \in K}$ uma tupla (coleção ordenada) de subconjuntos não vazios (sítios, sementes) em $\mathbf{X}$. Uma célula de Voronoi ou região de Voronoi R_k associa à semente P_k o conjunto de todos os pontos em $\mathbf{X}$ os quais a distância para P_k não é maior do que a distância para as outras sementes P_j. Ou seja, $R_k = \{x \in \mathbf{X} : \mathbf{d(x, P_k)} \leq \mathbf{d(x, P_j)},\ \forall \mathbf{j} \neq \mathbf{k}\}$.

Diagrama de Voronoi - Métodos Não Hierárquicos

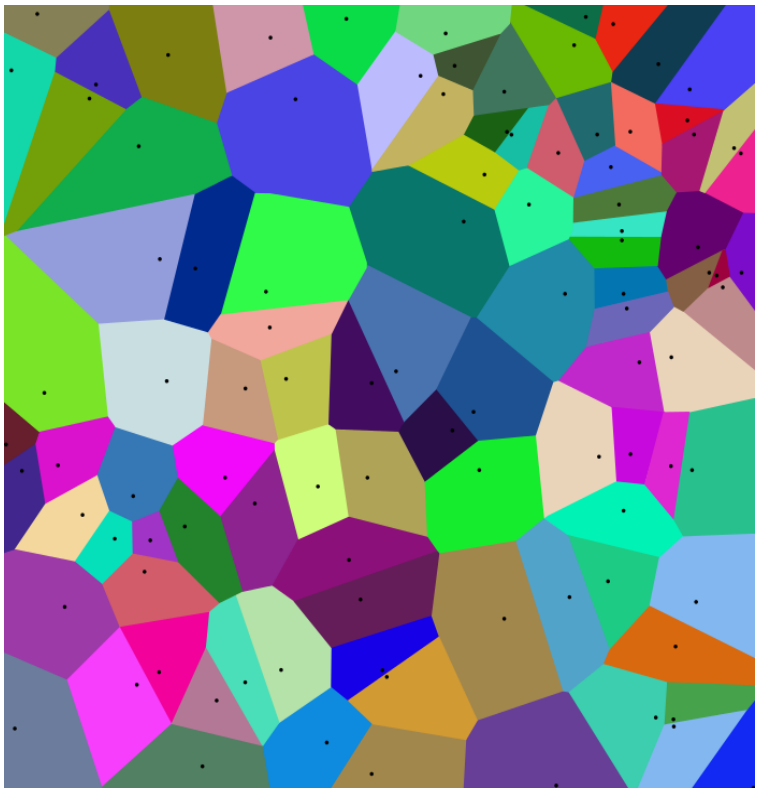


Figura: Diagrama de Voronoi.

Figura: Células de Voronoi em uma folha.

Curiosidade:

Figura: Lena Söderberg.

Curiosidade:

- Esta foto (conhecida como Lena) é frequentemente usado para testar algoritmos de processamento de imagem digital.

Figura: Lena Söderberg.

Curiosidade:

- Esta foto (conhecida como Lena) é frequentemente usado para testar algoritmos de processamento de imagem digital.
- Lena nasceu em 31 de março de 1951 na Suécia.

Figura: Lena Söderberg.

Curiosidade:

- Esta foto (conhecida como Lena) é frequentemente usado para testar algoritmos de processamento de imagem digital.
- Lena nasceu em 31 de março de 1951 na Suécia.
- Ela foi uma das convidadas a participar da **Conference of the Society for Imaging Science and Technology** em 1997.

Figura: Lena Söderberg.

Curiosidade:

- Esta foto (conhecida como Lena) é frequentemente usado para testar algoritmos de processamento de imagem digital.
- Lena nasceu em 31 de março de 1951 na Suécia.
- Ela foi uma das convidadas a participar da **Conference of the Society for Imaging Science and Technology** em 1997.
- É conhecida como a primeira dama da internet.

Figura: Lena Söderberg.

Simulação de células de Voronoi em uma imagem

Detalhes sobre o algoritmo em:
http://www.comp.nus.edu.sg/ tants/jfa.html.

Figura: Simulação com células de Voronoi geradas aleatoriamente em uma imagem.

Aplicações dos Diagramas de Voronoi

Aplicações dos Diagramas de Voronoi

- **Em análise de agrupamento de dados.**

Aplicações dos Diagramas de Voronoi

- **Em análise de agrupamento de dados.**

- Diagramas de Voronoi também são utilizados em computação gráfica para gerar alguns tipos de texturas orgânicas.

Aplicações dos Diagramas de Voronoi

- **Em análise de agrupamento de dados.**
- Diagramas de Voronoi também são utilizados em computação gráfica para gerar alguns tipos de texturas orgânicas.
- Também é utilizado em derivações da capacidade de uma rede sem fio.

Aplicações dos Diagramas de Voronoi

- **Em análise de agrupamento de dados.**
- Diagramas de Voronoi também são utilizados em computação gráfica para gerar alguns tipos de texturas orgânicas.
- Também é utilizado em derivações da capacidade de uma rede sem fio.
- Em robótica autônoma de navegação, Diagramas de Voronoi são utilizados para encontrar rotas livres. Se cada obstáculo do percurso for representado por um ponto, então as bordas do diagrama serão as rotas mais distantes dos obstáculos (afastando assim, em teoria, o risco de colisões).

Agrupamentos não hierárquico

Método **K-means**

Agrupamentos não hierárquico

Método **K-means**

- O termo **K-means** foi empregado primeiramente por James MacQueen em 1967 mas a ideia se deve a Hugo Steinhaus em 1957.

Agrupamentos não hierárquico

Método **K-means**

- O termo **K-means** foi empregado primeiramente por James MacQueen em 1967 mas a ideia se deve a Hugo Steinhaus em 1957.

- Em **big data** o método **K-means** trata-se de um método de **clustering** que objetiva particionar *n* objetos em *k clusters*.

Agrupamentos não hierárquico

Método **K-means**

- O termo **K-means** foi empregado primeiramente por James MacQueen em 1967 mas a ideia se deve a Hugo Steinhaus em 1957.
- Em **big data** o método **K-means** trata-se de um método de **clustering** que objetiva particionar *n* objetos em *k clusters*.
- As partições resultam em uma divisão do espaço em um **Diagrama de Voronoi**.

Agrupamentos não hierárquico

Método **K-means**

- O termo **K-means** foi empregado primeiramente por James MacQueen em 1967 mas a ideia se deve a Hugo Steinhaus em 1957.
- Em **big data** o método **K-means** trata-se de um método de **clustering** que objetiva particionar *n* objetos em *k clusters*.
- As partições resultam em uma divisão do espaço em um **Diagrama de Voronoi**.
- O problema é **NP-Hard**, no entanto, existem algoritmos heurísticos eficientes que são comumente empregados e convergem rapidamente.

Agrupamentos não hierárquico

Método **K-means**

Dado um conjunto de observações $\mathbf{X} = (\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n})$, em que cada observação é um vetor real d-dimensional, o método **K-means** visa particionar as n observações em k grupos de modo que $\mathbf{S} = (\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_k})$, com $k \leq n$, ou seja, $\mathbf{S}$ forma uma partição de $\mathbf{X}$. O objetivo do método **K-means** é:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{x_j \in \mathbf{S_i}} ||x_j - \mu_\mathbf{i}||^\mathbf{2}, \tag{2}$$

em que $\mu_\mathbf{i}$ é a media dos pontos em $\mathbf{S_i}$ e $||\cdot||$ é a distância euclidiana.

Agrupamentos não hierárquico

Em análise de agrupamento faz uso de medidas de similaridade (distância) entre pares de elementos. A depender do método de agrupamento, é possível utilizar diversas métricas de distâncias como a distância euclidiana, distância de Manhattan, distância de Camberra, distância de Mahalanobis, coeficiente de correlação, entre outras medidas.

Agrupamentos não hierárquico

Em análise de agrupamento faz uso de medidas de similaridade (distância) entre pares de elementos. A depender do método de agrupamento, é possível utilizar diversas métricas de distâncias como a distância euclidiana, distância de Manhattan, distância de Camberra, distância de Mahalanobis, coeficiente de correlação, entre outras medidas.

Nota: A medida de similaridade utiliza no método de **K-means** é a distância euclidiana.

Agrupamentos não hierárquico

Em análise de agrupamento faz uso de medidas de similaridade (distância) entre pares de elementos. A depender do método de agrupamento, é possível utilizar diversas métricas de distâncias como a distância euclidiana, distância de Manhattan, distância de Camberra, distância de Mahalanobis, coeficiente de correlação, entre outras medidas.

Nota: A medida de similaridade utiliza no método de **K-means** é a distância euclidiana.

Definição (Distância euclidiana entre dois pontos): Sejam $P = (p_1, p_2, \ldots, p_n)$ e $Q = (q_1, q_2, \ldots, q_n)$ pontos quaisquer em um espaço euclidiano. A distância euclidiana entre P e Q é dada por:

Agrupamentos não hierárquico

Em análise de agrupamento faz uso de medidas de similaridade (distância) entre pares de elementos. A depender do método de agrupamento, é possível utilizar diversas métricas de distâncias como a distância euclidiana, distância de Manhattan, distância de Camberra, distância de Mahalanobis, coeficiente de correlação, entre outras medidas.

Nota: A medida de similaridade utiliza no método de **K-means** é a distância euclidiana.

Definição (Distância euclidiana entre dois pontos): Sejam $P = (p_1, p_2, \ldots, p_n)$ e $Q = (q_1, q_2, \ldots, q_n)$ pontos quaisquer em um espaço euclidiano. A distância euclidiana entre P e Q é dada por:

$$||P - Q|| = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}.$$

Algoritmo **K-means**

O algoritmo mais comum faz uso de uma técnica de refinamento iterativo. Devido a sua boa performance ele é conhecido como algoritmo **k-means**. Também é conhecido como **algoritmo de Loyd** na comunidade de ciência da computação. Dado um conjunto de k médias iniciais $\mathbf{m_1^{(1)}}, \ldots, \mathbf{m_k^{(1)}}$ o algoritmo alterna (com $t \in \mathbb{N}$) entre os dois passos abaixo.

1. Faça para cada i:

$$\mathbf{S_i^{(t)}} = \{\mathbf{x_p} : ||\mathbf{x_p} - \mathbf{m_i^{(t)}}||^2 \leq ||\mathbf{x_p} - \mathbf{m_j^{(t)}}||^2 \, \forall \, \mathbf{j}, \mathbf{1} \leq \mathbf{j} \leq \mathbf{k}\}. \tag{3}$$

2. Atualize os sítios (sementes de células de Voronoi):

$$\mathbf{m_i^{(t+1)}} = \frac{\mathbf{1}}{|\mathbf{S_i^{(t)}}|} \sum_{\mathbf{x_j} \in \mathbf{S_i^{(t)}}} \mathbf{x_j}. \tag{4}$$

Visualizando - Algoritmo **K-means**

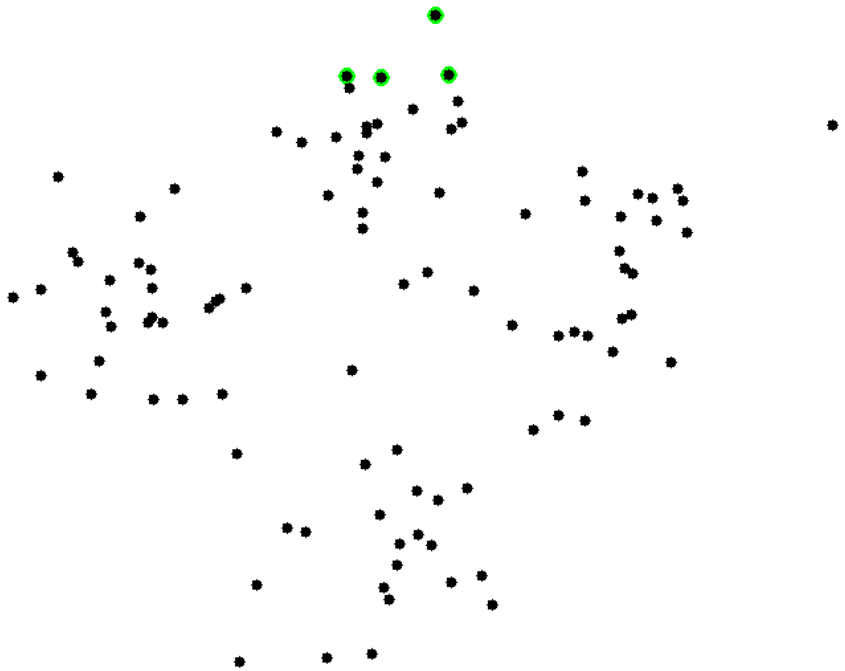
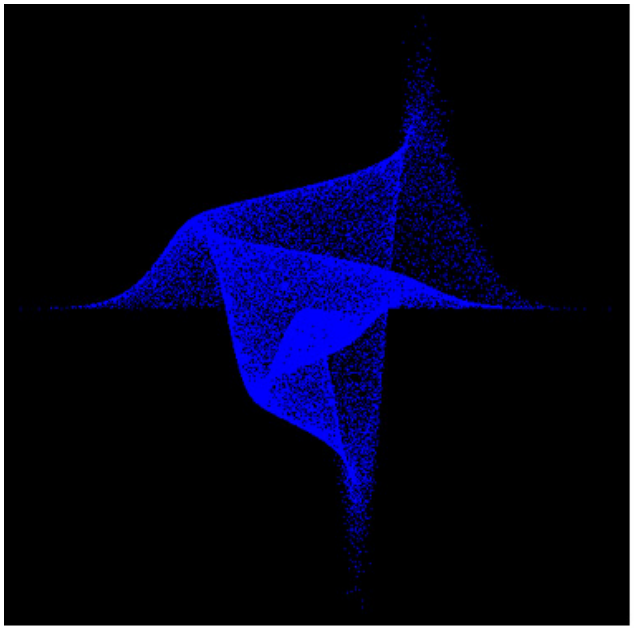
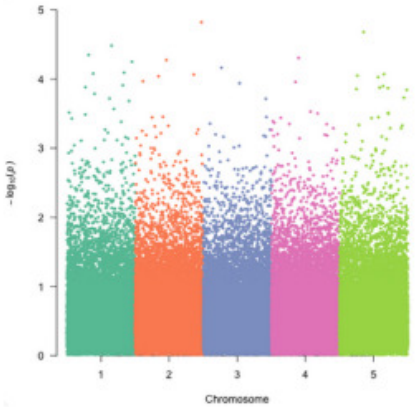
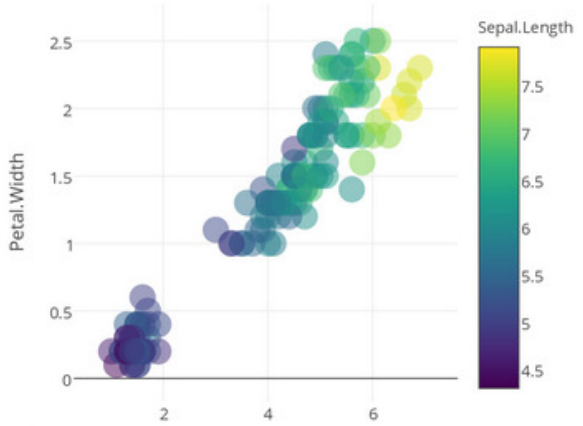


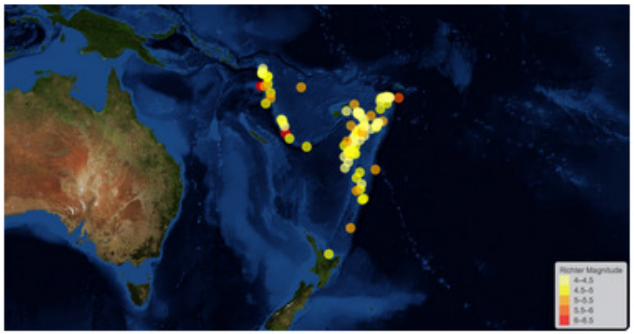
Figura: Simulação com células de Voronoi geradas aleatoriamente em uma imagem.

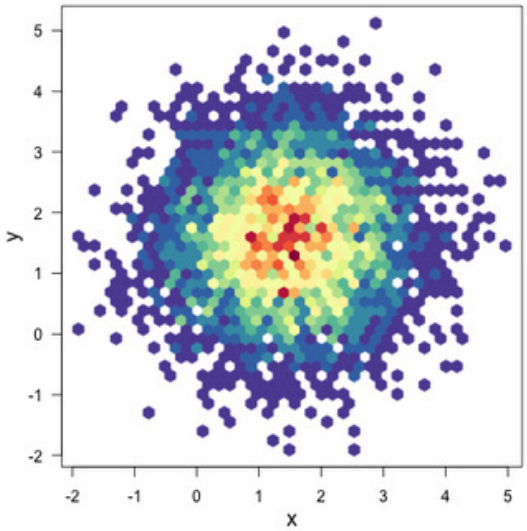
Gráficos

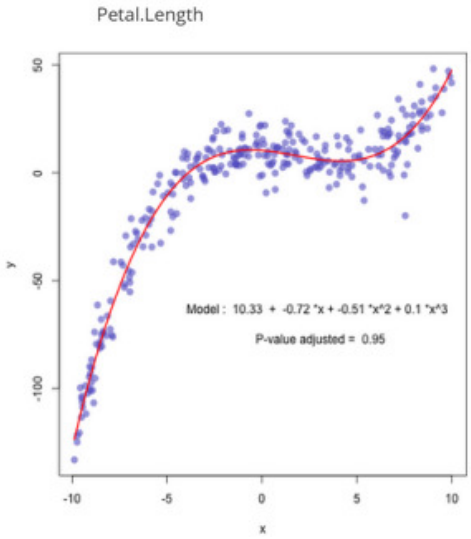



Chromosome


Sepal.Length
Petal.Width




x
y


Petal.Length
Model : 10.33 + -0.72 *x + -0.51 *x^2 + 0.1 *x^3
P-value adjusted = 0.95
x
y

Gráficos

"O grande valor de uma imagem é quando ela nos obriga a notar o que nunca esperávamos ver."

John W. Tukey

Na linguagem `R`, temos a nossa disposição diversos pacotes com propostas para soluções gráficas. Entre uma infinidade de pacotes, podemos citar:

1. **graphics**: Pacote instalado por padrão na linguagem `R`. É possível produzir muitos gráficos interessantes com esse pacote.
2. **plotly**: Biblioteca para produção de gráficos iterativos. Detalhes em `https://plot.ly/r/`.
3. **ggplot2**: Biblioteca baseada no **grammar of graphics**.

Gráficos

É muito comum trabalharmos com juntos de dados em que algumas variáveis podem ser categóricas. Nessas situações poderemos ter interesse em construir um gráfico que descreva algumas características descritivas dados por categoria como medidas de posição e dispersão. Nessas situações, o **boxplot** poderá nos fornecer algumas informações úteis.

Para construção de **boxplot** por categorias, é preciso entender uma **fórmula** em `R`.

Gráficos

É muito comum trabalharmos com juntos de dados em que algumas variáveis podem ser categóricas. Nessas situações poderemos ter interesse em construir um gráfico que descreva algumas características descritivas dados por categoria como medidas de posição e dispersão. Nessas situações, o **boxplot** poderá nos fornecer algumas informações úteis.

Para construção de **boxplot** por categorias, é preciso entender uma **fórmula** em R.

Estrutura de uma fórmula:

Gráficos

É muito comum trabalharmos com juntos de dados em que algumas variáveis podem ser categóricas. Nessas situações poderemos ter interesse em construir um gráfico que descreva algumas características descritivas dados por categoria como medidas de posição e dispersão. Nessas situações, o **boxplot** poderá nos fornecer algumas informações úteis.

Para construção de **boxplot** por categorias, é preciso entender uma **fórmula** em `R`.

Estrutura de uma fórmula:

```
variável ~ expressão
```

Gráficos

Normalmente a estrutura de fórmula é utilizada em modelos de regressão em que `variável` é a variável resposta do modelo e `expressão` é uma estrutura de regressão. Normalmente, `expressão` é uma fórmula matemática em que definimos as variáveis explicativas do modelo de regressão a serem consideradas.

Gráficos

Normalmente a estrutura de fórmula é utilizada em modelos de regressão em que `variável` é a variável resposta do modelo e `expressão` é uma estrutura de regressão. Normalmente, `expressão` é uma fórmula matemática em que definimos as variáveis explicativas do modelo de regressão a serem consideradas.

Porém, em algumas situações poderemos fazer o uso de expressão para construção de gráficos em `R`.

Gráficos

Normalmente a estrutura de fórmula é utilizada em modelos de regressão em que `variável` é a variável resposta do modelo e `expressão` é uma estrutura de regressão. Normalmente, `expressão` é uma fórmula matemática em que definimos as variáveis explicativas do modelo de regressão a serem consideradas.

Porém, em algumas situações poderemos fazer o uso de expressão para construção de gráficos em `R`.

Uma situação bastante útil no uso de uma **fórmula** matemática é na construção de **boxplot** por categoria.

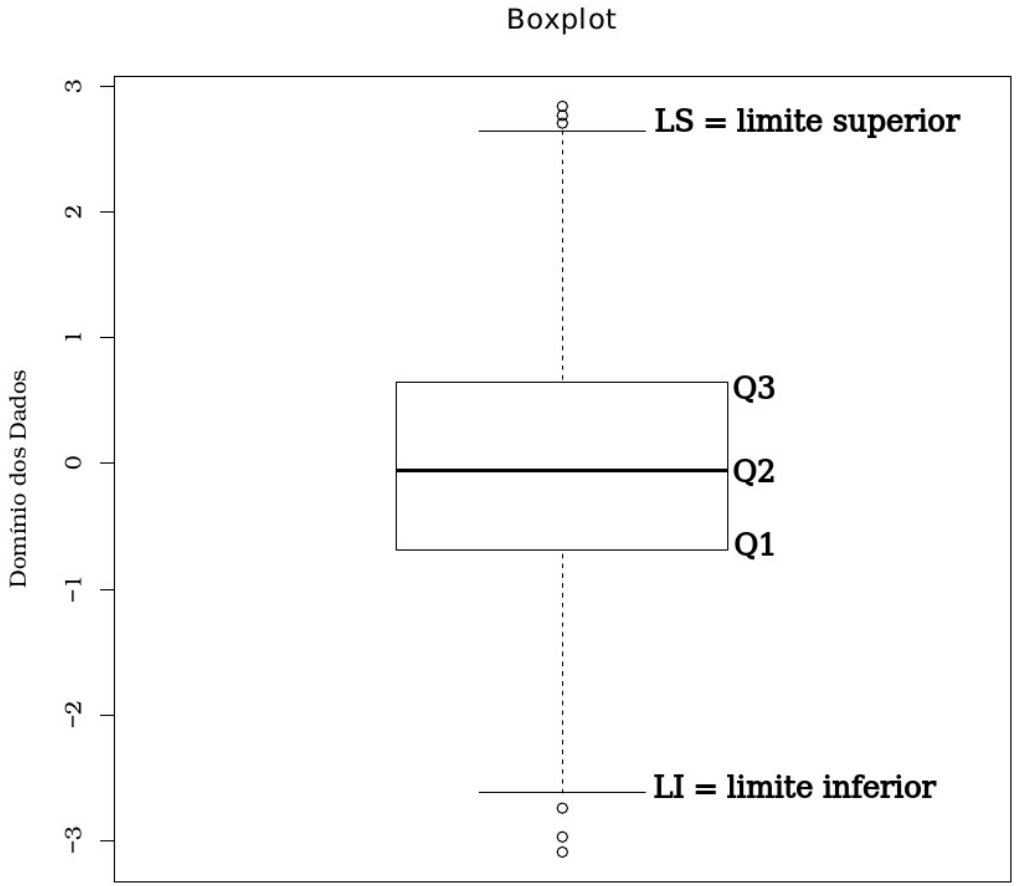


Figura: Boxplot de um conjuntos de dados qualquer.

Gráficos

Na Figura anterior, temos que:

Gráficos

Na Figura anterior, temos que:

1. **Q1**: é o primeiro quartil, isto é, distribuído abaixo dele encontra-se 25% dos dados;

Gráficos

Na Figura anterior, temos que:

1. **Q1**: é o primeiro quartil, isto é, distribuído abaixo dele encontra-se 25% dos dados;
2. **Q2**: é o segundo quartil, isto é, distribuído abaixo dele encontra-se 50% dos dados. Dessa forma, temos que

Gráficos

Na Figura anterior, temos que:

1. **Q1**: é o primeiro quartil, isto é, distribuído abaixo dele encontra-se 25% dos dados;
2. **Q2**: é o segundo quartil, isto é, distribuído abaixo dele encontra-se 50% dos dados. Dessa forma, temos que **Q2** é a **mediana** dos dados;

Gráficos

Na Figura anterior, temos que:

1. **Q1**: é o primeiro quartil, isto é, distribuído abaixo dele encontra-se 25% dos dados;
2. **Q2**: é o segundo quartil, isto é, distribuído abaixo dele encontra-se 50% dos dados. Dessa forma, temos que **Q2** é a **mediana** dos dados;
3. **Q3**: analogamente, **Q3** é o terceiro quartil, isto é, distribuído abaixo dele encontra-se 75% dos dados;

Gráficos

Na Figura anterior, temos que:

1. **Q1**: é o primeiro quartil, isto é, distribuído abaixo dele encontra-se 25% dos dados;
2. **Q2**: é o segundo quartil, isto é, distribuído abaixo dele encontra-se 50% dos dados. Dessa forma, temos que **Q2** é a **mediana** dos dados;
3. **Q3**: analogamente, **Q3** é o terceiro quartil, isto é, distribuído abaixo dele encontra-se 75% dos dados;
4. **LI**: é o limite inferior dos dados obtido por $LI \approx Q_1 - 1.5(Q_3 - Q_1)$;

Gráficos

Na Figura anterior, temos que:

1. **Q1**: é o primeiro quartil, isto é, distribuído abaixo dele encontra-se 25% dos dados;
2. **Q2**: é o segundo quartil, isto é, distribuído abaixo dele encontra-se 50% dos dados. Dessa forma, temos que **Q2** é a **mediana** dos dados;
3. **Q3**: analogamente, **Q3** é o terceiro quartil, isto é, distribuído abaixo dele encontra-se 75% dos dados;
4. **LI**: é o limite inferior dos dados obtido por $LI \approx Q_1 - 1.5(Q_3 - Q_1)$;
5. **LS**: é o limite superior dos dados obtido por $LS \approx Q_1 + 1.5(Q_3 - Q_1)$.

Gráficos

Todos os pontos no gráfico **boxplot** abaixo de **LI** ou acima **LS** são chamados de de **outliers**.

Todos os pontos no gráfico **boxplot** abaixo de **LI** ou acima **LS** são chamados de de **outliers**.

Um **outlier** na estatística refere-se à uma observação "distante" das demais observações. Tal distância pode ter sua origem em variabilidades na medição da observação ou poderá indicar um erro experimental. Em caso de erros experimentais, às vezes, tais dados poderão ser excluídos ou preferencialmente substituídos.

Todos os pontos no gráfico **boxplot** abaixo de **LI** ou acima **LS** são chamados de de **outliers**.

Um **outlier** na estatística refere-se à uma observação "distante" das demais observações. Tal distância pode ter sua origem em variabilidades na medição da observação ou poderá indicar um erro experimental. Em caso de erros experimentais, às vezes, tais dados poderão ser excluídos ou preferencialmente substituídos.

Importante: Os **outliers** poderá ocorrer ao acaso em qualquer distribuição, ou seja, nem sempre significará erro. Pelo contrário, é bastante comum à ocorrência de **outliers** e sua eliminação, na maioria das vezes, costuma ser um procedimento equivocado.

Exemplo: Construindo um **boxplot** para um vetor de dados gerado aleatoriamente.

Gráficos

Exemplo: Construindo um **boxplot** para um vetor de dados gerado aleatoriamente.

```
1 # Informar um diretorio de trabalho setwd("diretorio de
      trabalho")
2 # Salvando o grafico no formato PDF.
3 pdf(file="boxplot_1.pdf",width=9,height=9, paper="special",
4     family="Bookman",pointsize=14)
5     set.seed(0)
6     dados <- rnorm(1000)
7     boxplot(dados, main = c("Boxplot"), ylab = c("Dominio
          dos Dados"))
8 dev.off()
```

Gráficos

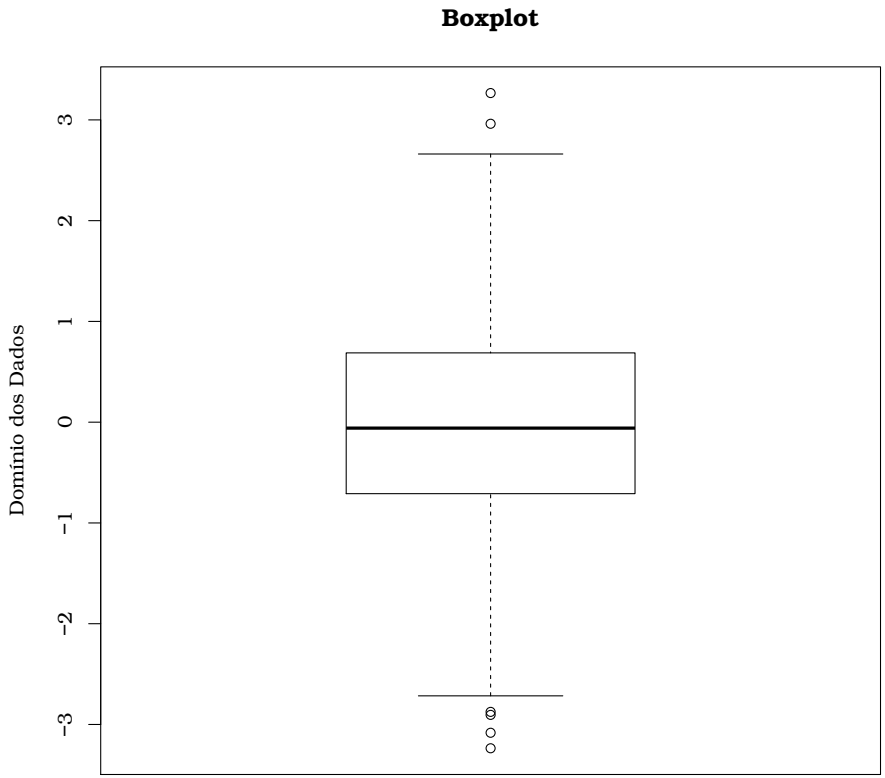


Figura: Boxplot para um vetor de dados gerado aleatoriamente (exemplo anterior).

Exercício: Construa um **boxplot** para cada uma das categorias da variável **Species** do conjunto de dados **iris** utilizando os dados da variável **Petal.Length**

Solução:

Exercício: Construa um **boxplot** para cada uma das categorias da variável **Species** do conjunto de dados **iris** utilizando os dados da variável **Petal.Length**

Solução:

```
1 pdf(file="boxplot_2.pdf",width=9,height=9, paper="special",
2 family="Bookman",pointsize=14)
3     boxplot(Petal.Width ~ Species, data = iris, ylab = c("
          Dominio dos Dados"), xlab = c("Categorias"))
4 dev.off()
```

Gráficos

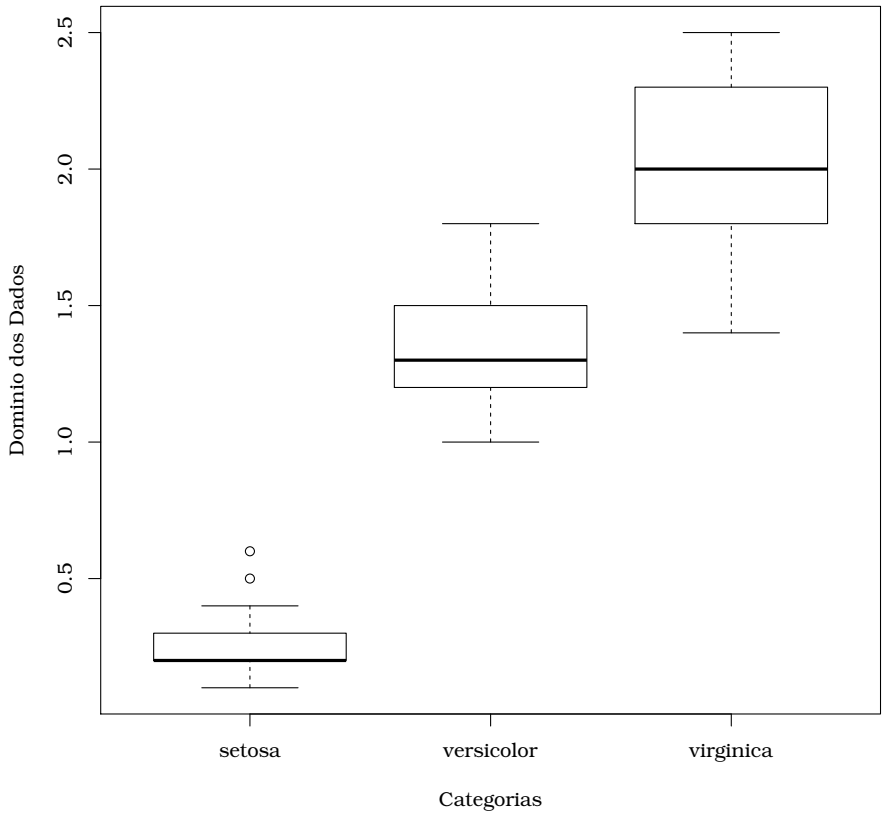


Figura: Boxplot para a variável **Petal.Width** por categorias do conjunto de dados **iris**.

Gráficos

Em situações em que temos dados discretos, é comum a necessidade de construção de **gráficos de barra**. Em `R`, utilizamos a função `barplot()` para a produção desse estilo de gráfico.

Exercício: Estude a documentação da função `barplot()`. Depois, construa um gráfico de barras para a variável **Species** tal que a variável **Petal.Width** é maior que 1 e $3.2 <$ **Sepal.Width** ≤ 4.5.

Solução:

```
1 # Especifique um diretorio ao argumento file ou fixe um
2 # diretorio de trabalho. Nao fazendo nada, o grafico sera
3 # salvo no diretorio de trabalho padrao. Utilize getwd()
4 # para obter informacao sobre o diretorio de trabalho.
5 pdf(file="barplot_1.pdf",width=9,height=9, paper="special",
6 family="Bookman",pointsize=14)
7 barplot(table(subset(iris, Petal.Width > 1 & Sepal.Width <=
      4.5 & Sepal.Width>3.2)$Species), ylab = c("Frequencia"),
       col = c("gray90", "gray60", "gray40"))
8 dev.off()
```

Gráficos

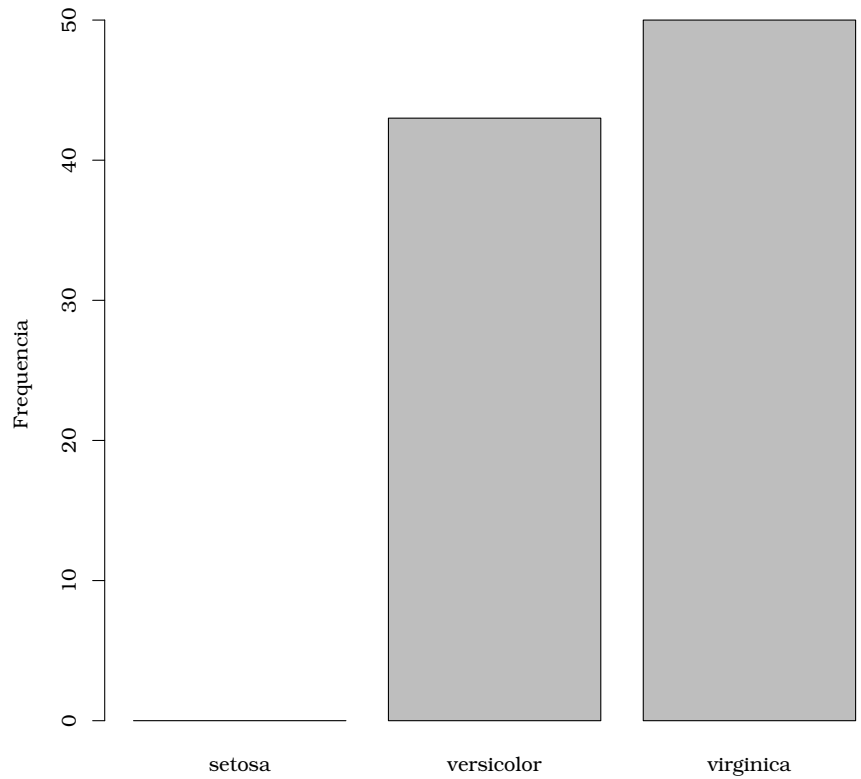


Figura: Gráfico de barra produzido pela função `barplot()` utilizada no código do exercício acima.

Gráficos

Também é possível construir gráficos em três dimensões (3D) em R. Isso se deve ao fato de que em diversas situações aplicamos funções que fazem uso dos dados e normalmente estas funções são definidas de $\mathbb{R}^p$ em $\mathbb{R}$. Um exemplo bastante comum disto são as funções log-verossimilhanças.

Gráficos

Também é possível construir gráficos em três dimensões (3D) em R. Isso se deve ao fato de que em diversas situações aplicamos funções que fazem uso dos dados e normalmente estas funções são definidas de $\mathbb{R}^p$ em $\mathbb{R}$. Um exemplo bastante comum disto são as funções log-verossimilhanças.

Poderemos produzir gráficos de superfícies em R de diversas formas. Apresentaremos a seguir duas formas de se obter tais gráficos.

Gráficos

Também é possível construir gráficos em três dimensões (3D) em R. Isso se deve ao fato de que em diversas situações aplicamos funções que fazem uso dos dados e normalmente estas funções são definidas de $\mathbb{R}^p$ em $\mathbb{R}$. Um exemplo bastante comum disto são as funções log-verossimilhanças.

Poderemos produzir gráficos de superfícies em R de diversas formas. Apresentaremos a seguir duas formas de se obter tais gráficos.

Nota: Instale o pacote **plot3D** para que seja possível produzir a segunda forma de gráfico de superfície apresentada adiante.

Gráficos

```
1  # Substitua o diretorio de trabalho abaixo para um diretorio
2  # de interesse em seu computador.
3  setwd("direitorio")
4
5  rosenbrock <- function(x,y) {
6      100 * (y - x * x)^2 + (1 - x)^2
7  }
8  # Forma 1
9  x <- seq(-100, 100, length = 30)
10 y <- x
11 z <- outer(X = x, Y = y, rosenbrock)
12
13 pdf(file="rosenbrock1.pdf", width=9, height=9, paper="
       special", family="Bookman",pointsize=14)
14     persp(x, y, z, theta = 3, phi = 40, expand = 0.5, col =
           "lightblue")
15 dev.off()
```

Gráficos

```
16 #Forma 2
17
18 library(plot3D)
19 M <- mesh(seq(-100, 100, length.out = 500),
20 seq(-100, 100, length.out = 500))
21 x  <- M$x ; y <- M$y
22 z <- rosenbrock(x=x,y=y)
23
24 pdf(file="rosenbrock2.pdf",width=9,height=9, paper="special"
       , family="Bookman",pointsize=14)
25     surf3D(x,y,z,inttype=1,bty="b2",phi=40, theta=3)
26 dev.off()
```

Gráficos

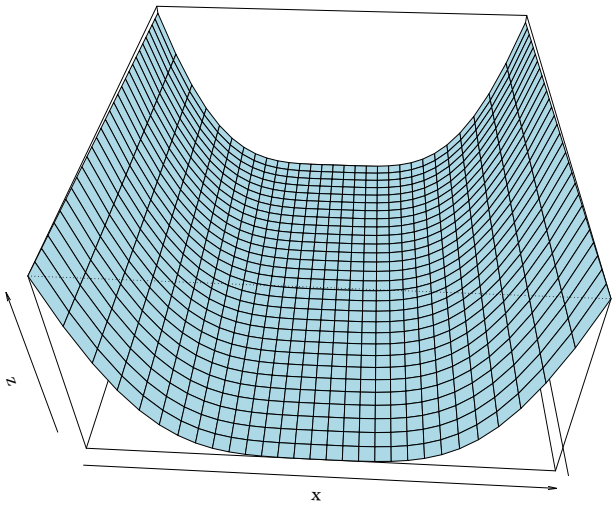


Figura: Forma 1 usando a função `persp()`.

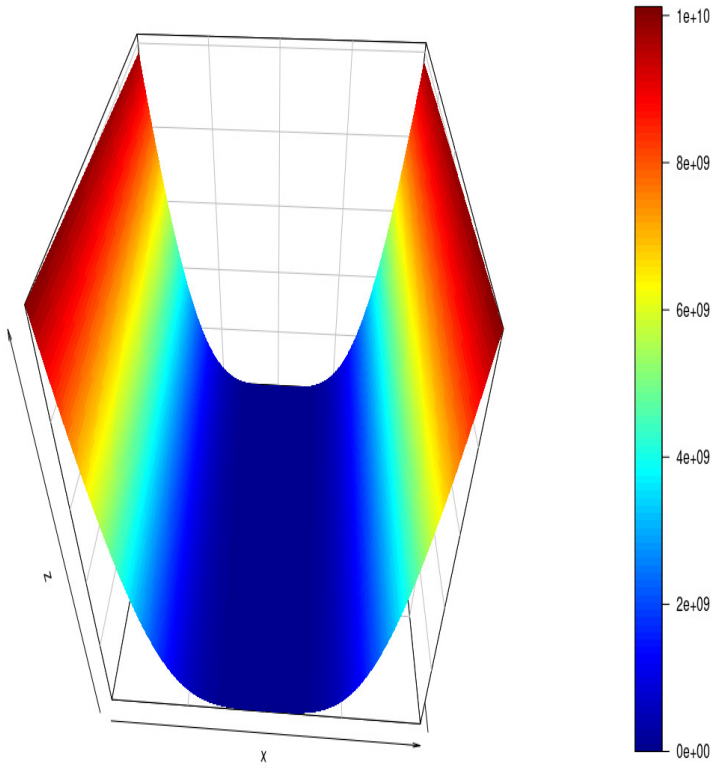


Figura: Forma 2 usando a função `surf3D()` do pacote **plot3D**.

Exercício: Consideremos o caso da função Hölder de características muito peculiar definida da forma abaixo.

$$f(x,y) = -\left|\sin(x)\cos(y)\exp\left(\left|1 - \frac{\sqrt{x^2+y^2}}{\pi}\right|\right)\right|,$$

em que $-10 \leq x,\, y \leq 10$. Construa o gráfico da função utilizando a função `persp()` e `surf3D()` (do pacote **plot3D**).

Gráficos

Gráficos em `R` são produzidos por um passo a passo de funções em que cada uma das funções acrescenta um detalhe ou característica ao que estamos querendo plotar. Tais funções geralmente possuem um conjunto de argumentos que podem ser utilizados para mudar características do gráfico.

Gráficos

Exemplo: Corra o código abaixo. Podemos ver como iniciar o layout de um gráfico que desejamos plotar.

```
1  plot.new()
2  plot.window(xlim = c(0, 1), ylim = c(5, 10))
3  abline(a = 6, b = 3)
4  axis(1)
5  axis(2)
6  title(main = "Titulo Principal")
7  title(xlab = "Eixo X")
8  title(ylab = "Eixo Y")
9  box()
10 grid()
```

Gráficos

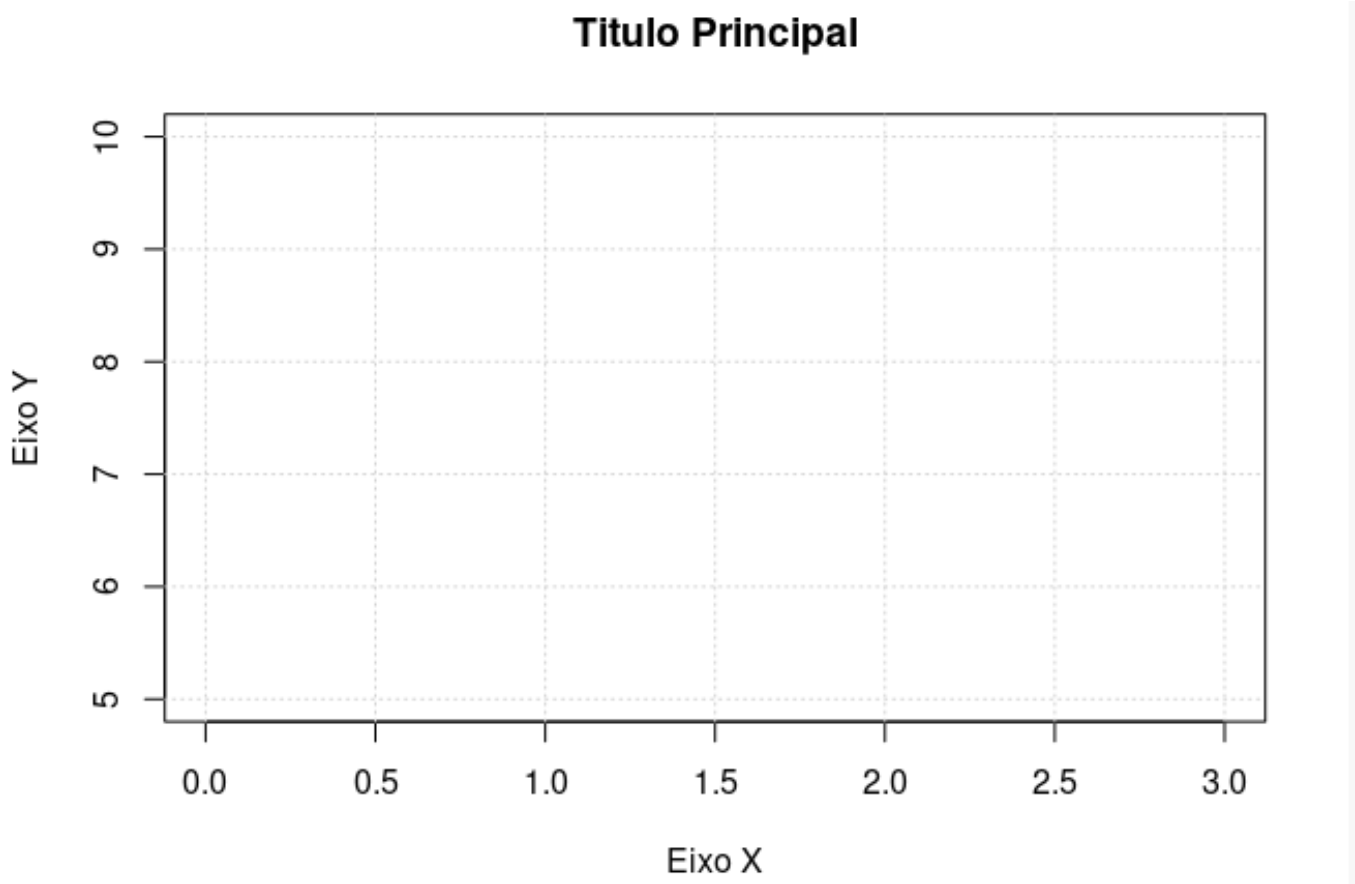


Figura: Saída gráfica do código acima.

Exercício: Estude a documentação das funções utilizadas no código acima procurando absorver a utilidade de alguns parâmetros.

Exercício: Utilizando a função `abline()`, adicione ao gráfico plotado no exemplo anterior uma linha vertical no ponto $x = 0$ e uma linha horizontal no ponto $y = 0$. Coloque também no gráfico as retas $y = 2x + 4$ e $y = 3x + 2$ com cores diferentes. Construa o gráfico omitindo a chamada da função `box()`. (**Dica**: Construa o gráfico com $-10 \leq x, y \leq 10$.)

Solução:

```
1  plot.new()
2  plot.window(xlim = c(-10, 10), ylim = c(-10, 10))
3  grid(lwd = 2)
4  abline(a = 4, b = 2, col = "red", lwd = 2)
5  abline(a = 2, b = 3, col = "blue", lwd = 2)
6  abline(h = 0, v = 0, col = "black", lwd = 2)
7  axis(side = 1, at = -10:10, cex.axis = .8, font = 1)
8  axis(side = 2, at = -10:10, cex.axis = .8, font = 1)
9  title(main = "Titulo Principal")
10 title(xlab = "Eixo X")
11 title(ylab = "Eixo Y")
12 box()
```

Gráficos

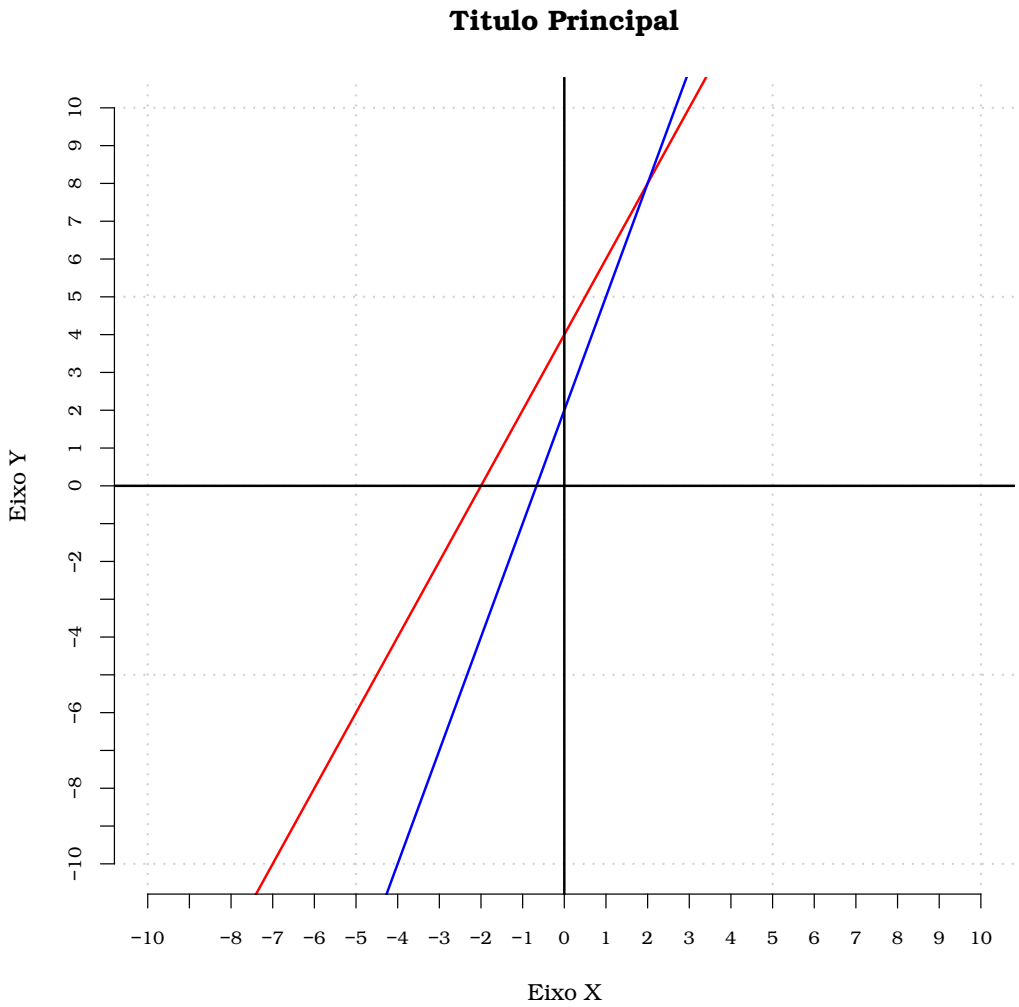


Figura: Gráfico produzido pelo código apresentado no exercício anterior.

Exercício: Fazendo uso da função `points()`, adicione ao gráfico do exercício anterior um ponto no cinza no ponto de encontro das duas retas.

Exercício: Fazendo uso da função `points()`, adicione ao gráfico do exercício anterior um ponto no cinza no ponto de encontro das duas retas.

Solução: Adicione ao código anterior a linha de código `points(x = 2, y = 8, pch = 19, col = "green")`.

Gráficos

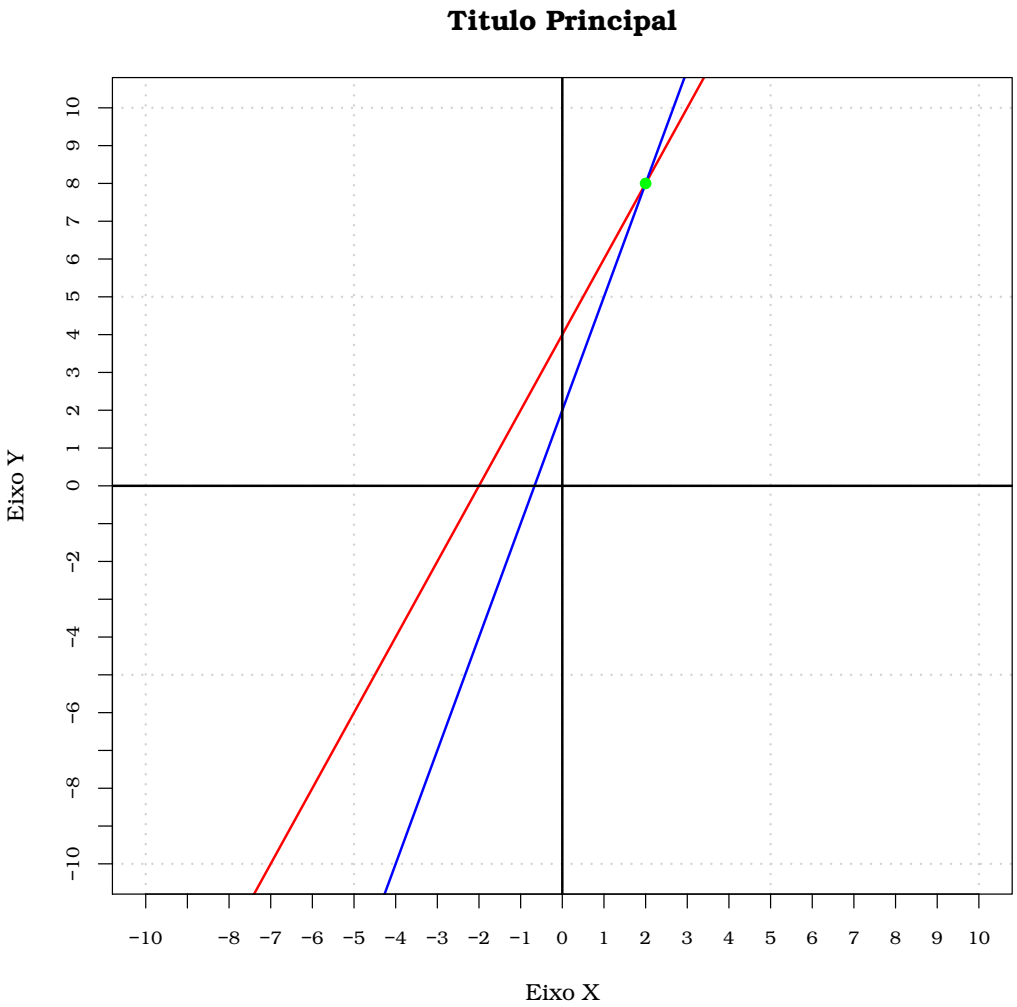


Figura: Gráfico produzido pelo código apresentado no exercício anterior.

Gráficos

Exercício: Trace um seguimentos de retas utilizando a função `segments()` de $x = 2$ até o ponto verde e um outro seguimento de reta de $x = 2$ até o ponto verde. Faça com que o seguimento de reta seja tracejado e de cor laranja.

Solução:

```
plot.new()
plot.window(xlim = c(-10, 10), ylim = c(-10, 10))
grid(lwd = 2)
abline(a = 4, b = 2, col = "red", lwd = 2)
abline(a = 2, b = 3, col = "blue", lwd = 2)
abline(h = 0, v = 0, col = "black", lwd = 2)
axis(side = 1, at = -10:10, cex.axis = .8, font = 1)
axis(side = 2, at = -10:10, cex.axis = .8, font = 1)
title(main = "Titulo Principal")
title(xlab = "Eixo X")
title(ylab = "Eixo Y")
segments(x0 = -11, y0 = 8, x1 = 2, y1 = 8, lty = 2, col = "
    orange", lwd = 3)
segments(x0 = 2, y0 = 8, x1 = 2, y1 = -11, lty = 2, col = "
    orange", lwd = 3)
points(x = 2, y = 8, pch = 19, col = "green")
```

Gráficos

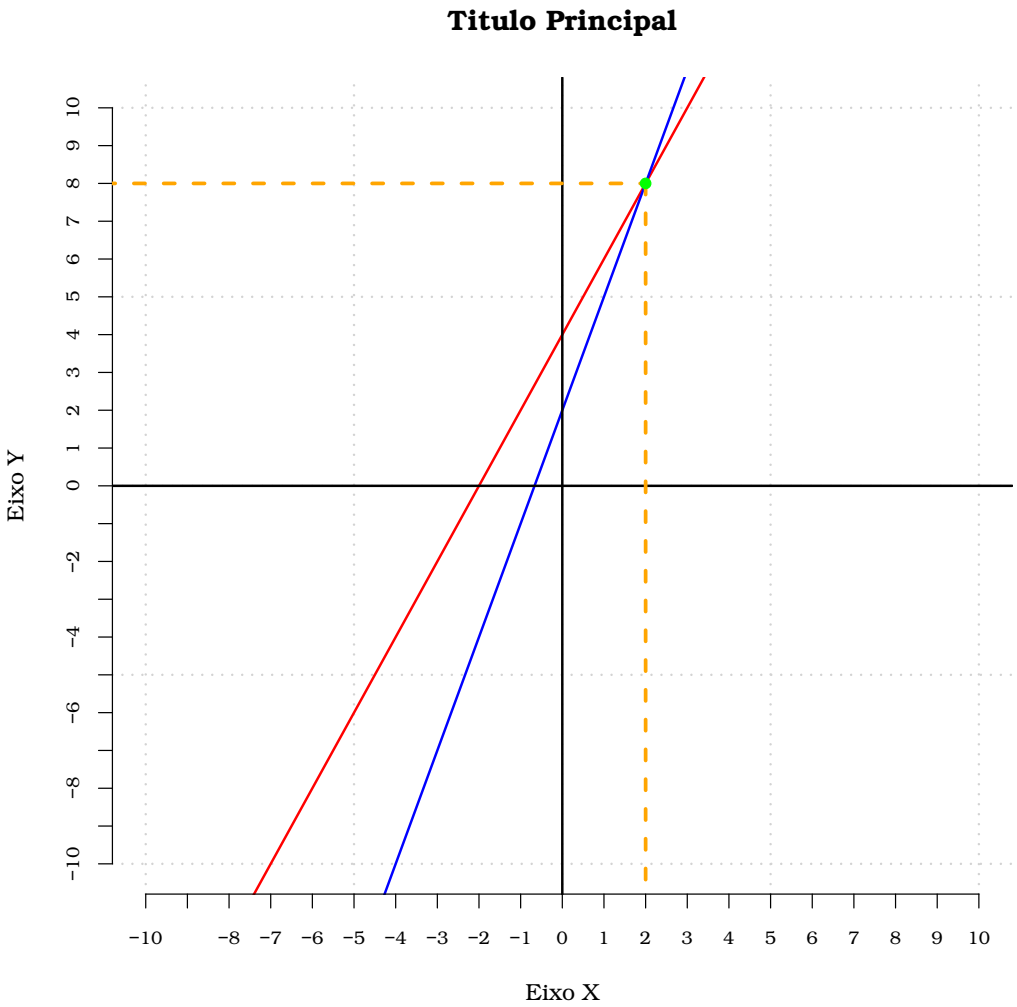


Figura: Gráfico produzido pelo código apresentado no exercício anterior.

Exercício: Acrescente texto no gráfico acima utilizando a função `text()`. Nesse exercício queremos acrescentar ao lado do ponto $(2, 8)$ o texto **P(2,8)** e ao lado esquerdo do ponto $(0, 0)$ desejamos acrescentar o texto **0**.

Gráficos

```
plot.new()
plot.window(xlim = c(-10, 10), ylim = c(-10, 10))
grid(lwd = 2)
abline(a = 4, b = 2, col = "red", lwd = 2)
abline(a = 2, b = 3, col = "blue", lwd = 2)
abline(h = 0, v = 0, col = "black", lwd = 2)
axis(side = 1, at = -10:10, cex.axis = .8, font = 1)
axis(side = 2, at = -10:10, cex.axis = .8, font = 1)
title(main = "Titulo Principal")
title(xlab = "Eixo X")
title(ylab = "Eixo Y")
segments(x0 = -11, y0 = 8, x1 = 2, y1 = 8, lty = 2, col = "
    orange", lwd = 3)
segments(x0 = 2, y0 = 8, x1 = 2, y1 = -11, lty = 2, col = "
    orange", lwd = 3)
points(x = 2, y = 8, pch = 19, col = "green")
text(x = 3, y = 8, labels = "P(2,8)", lwd = 2)
text(x = -0.2, y = -0.5, labels = "0", lwd = 2)
```

Gráficos

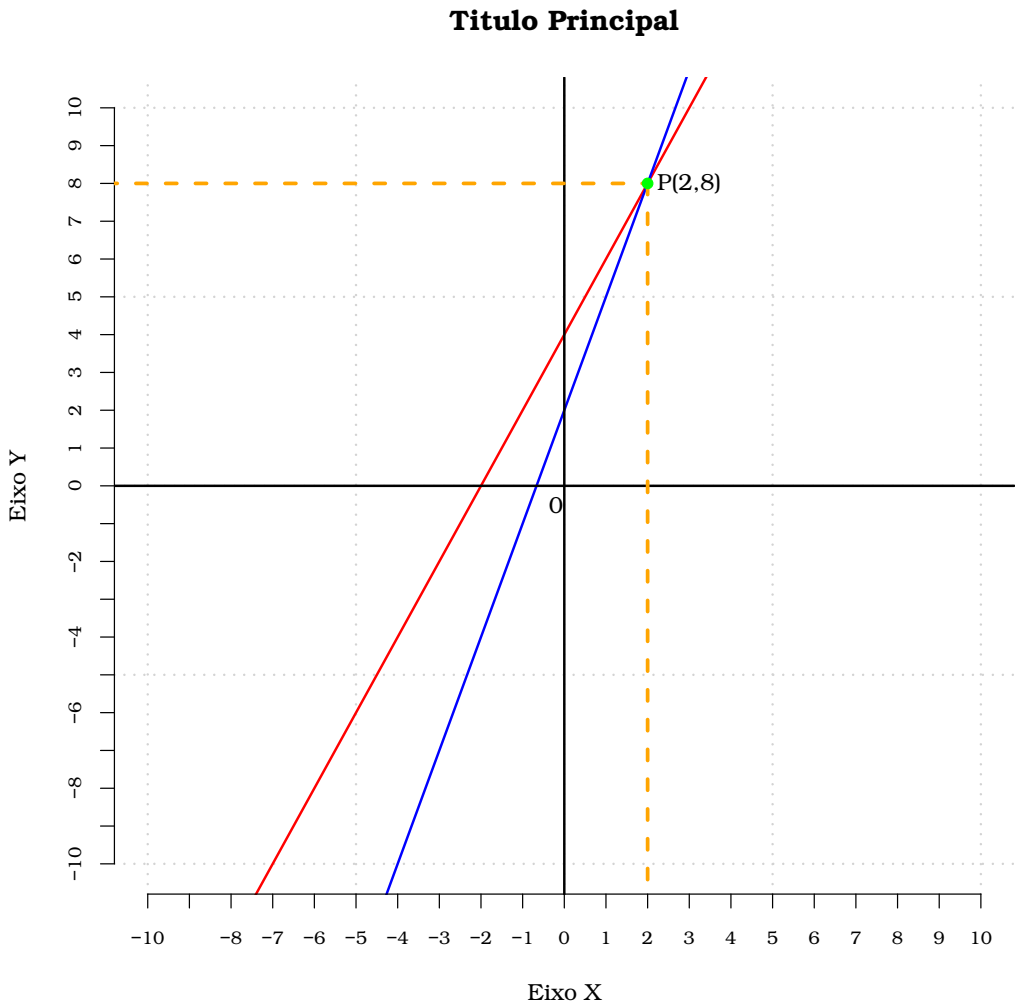


Figura: Gráfico produzido pelo código apresentado no exercício anterior.

Exercício: Acrescente com a função `mtext()` um subtítulo ao gráfico anterior. Por exemplo, acrescente o texto **Status - Empesa Júnior de Estatística - UFPB**.

Gráficos

```
plot.new()
plot.window(xlim = c(-10, 10), ylim = c(-10, 10))
grid(lwd = 2)
abline(a = 4, b = 2, col = "red", lwd = 2)
abline(a = 2, b = 3, col = "blue", lwd = 2)
abline(h = 0, v = 0, col = "black", lwd = 2)
axis(side = 1, at = -10:10, cex.axis = .8, font = 1)
axis(side = 2, at = -10:10, cex.axis = .8, font = 1)
title(main = "Titulo Principal")
title(xlab = "Eixo X")
title(ylab = "Eixo Y")
segments(x0 = -11, y0 = 8, x1 = 2, y1 = 8, lty = 2, col = "
    orange", lwd = 3)
segments(x0 = 2, y0 = 8, x1 = 2, y1 = -11, lty = 2, col = "
    orange", lwd = 3)
points(x = 2, y = 8, pch = 19, col = "green")
text(x = 3, y = 8, labels = "P(2,8)", lwd = 2)
text(x = -0.2, y = -0.5, labels = "0", lwd = 2)
mtext("Status - Empesa Junior de Estatistica - UFPB")
```

Gráficos

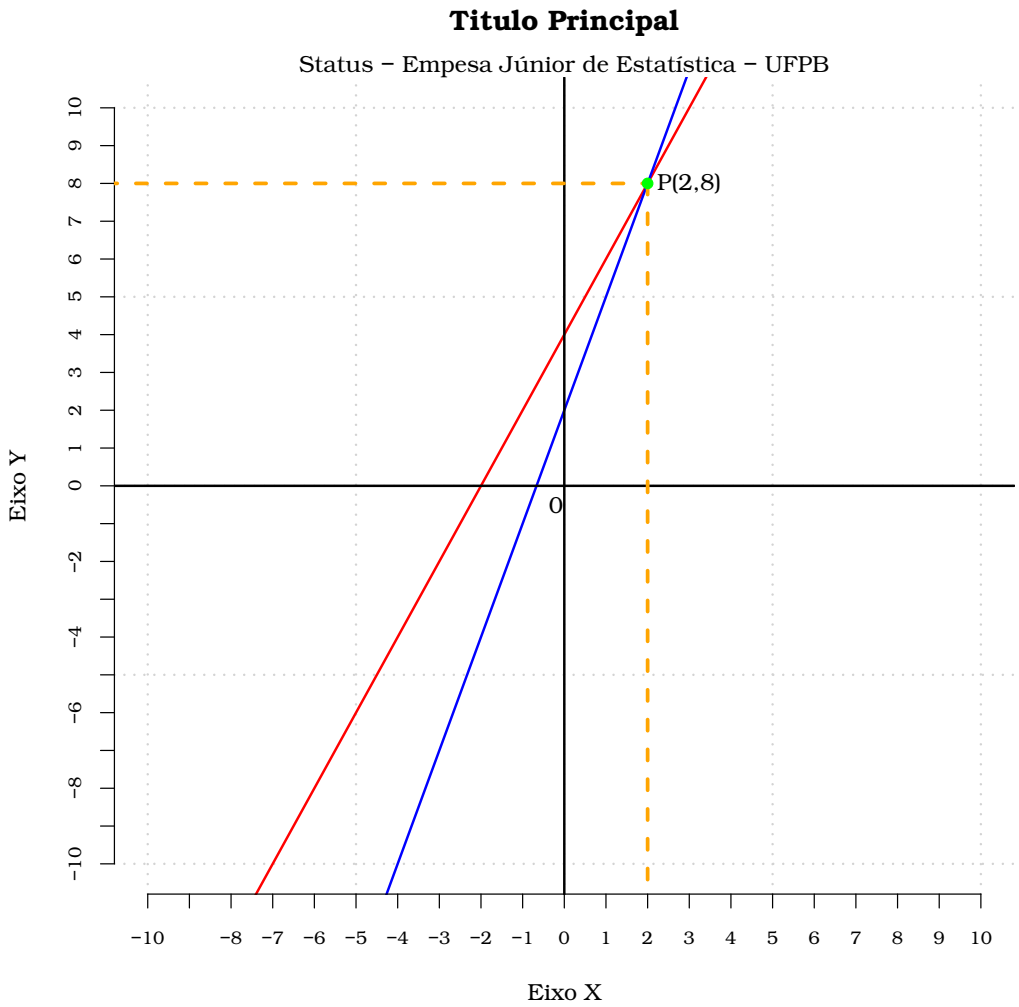


Figura: Gráfico produzido pelo código apresentado no exercício anterior.

Exemplo: Suponha que desejamos escrever o texto **Status** abaixo do eixo X do gráfico acima. Também poderemos fazer uso da função `mtext()`. Porém, precisamos assegurar que teremos espaço para acrescentar algum elemento ao gráfico. Dessa forma, precisaremos controlar as margens do gráfico. Para isso, iremos utilizar a função `par()`.

Nota: A função `par()` é uma das principais funções para controlar o aspecto de um gráfico. Por exemplo, é possível controlar as margens de um gráfico, acomodar mais de um gráfico em um único frame, combinar gráficos, entre diversas outras possibilidades.

Gráficos

```
plot.new()
par(oma = c(3,3,3,3)) # Ampliando as 4 margens.
plot.window(xlim = c(-10, 10), ylim = c(-10, 10))
grid(lwd = 2)
abline(a = 4, b = 2, col = "red", lwd = 2)
abline(a = 2, b = 3, col = "blue", lwd = 2)
abline(h = 0, v = 0, col = "black", lwd = 2)
axis(side = 1, at = -10:10, cex.axis = .8, font = 1)
axis(side = 2, at = -10:10, cex.axis = .8, font = 1)
title(main = "Titulo Principal")
title(xlab = "Eixo X"); title(ylab = "Eixo Y")
segments(x0 = -11, y0 = 8, x1 = 2, y1 = 8, lty = 2, col = "
    orange", lwd = 3)
segments(x0 = 2, y0 = 8, x1 = 2, y1 = -11, lty = 2, col = "
    orange", lwd = 3)
points(x = 2, y = 8, pch = 19, col = "green")
text(x = 3, y = 8, labels = "P(2,8)", lwd = 2)
text(x = -0.2, y = -0.5, labels = "0", lwd = 2)
mtext("Status - Empesa Junior de Estatistica - UFPB")
mtext("Status", line = 5, side = 1)
```

Gráficos

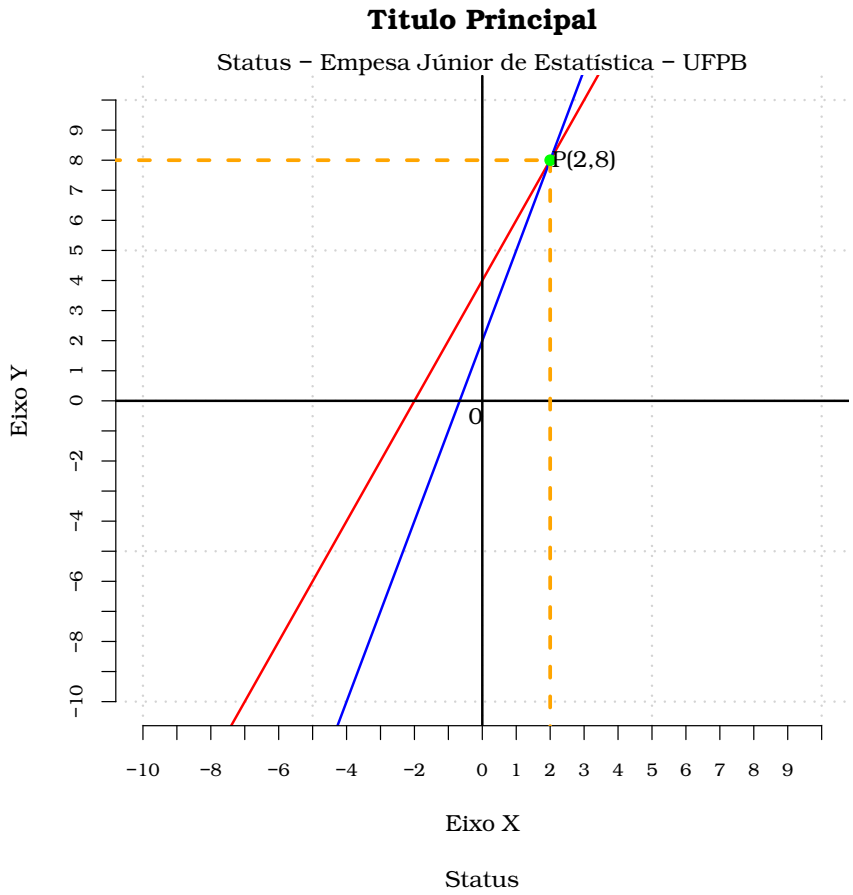


Figura: Gráfico produzido pelo código apresentado no exercício anterior.

Exercício: Podemos estar interessados em colocar uma cor ao fundo do gráfico anterior. Sendo assim, por meio da função `par()` altere o argumento **bg** para que seja colocado um fundo cinza no gráfico anterior. **Dica**: faça `colors()` para obter uma lista de cores disponíveis em `R`.

Gráficos

```
plot.new()
par(oma = c(3,3,3,3), bg = "gray") # Ampliando as 4 margens.
plot.window(xlim = c(-10, 10), ylim = c(-10, 10))
grid(lwd = 2)
abline(a = 4, b = 2, col = "red", lwd = 2)
abline(a = 2, b = 3, col = "blue", lwd = 2)
abline(h = 0, v = 0, col = "black", lwd = 2)
axis(side = 1, at = -10:10, cex.axis = .8, font = 1)
axis(side = 2, at = -10:10, cex.axis = .8, font = 1)
title(main = "Titulo Principal")
title(xlab = "Eixo X")
title(ylab = "Eixo Y")
segments(x0 = -11, y0 = 8, x1 = 2, y1 = 8, lty = 2, col = "
    orange", lwd = 3)
segments(x0 = 2, y0 = 8, x1 = 2, y1 = -11, lty = 2, col = "
    orange", lwd = 3)
points(x = 2, y = 8, pch = 19, col = "green")
text(x = 3, y = 8, labels = "P(2,8)", lwd = 2)
text(x = -0.2, y = -0.5, labels = "0", lwd = 2)
mtext("Status - Empesa Junior de Estatistica - UFPB")
mtext("Status", line = 5, side = 1)
```

Gráficos

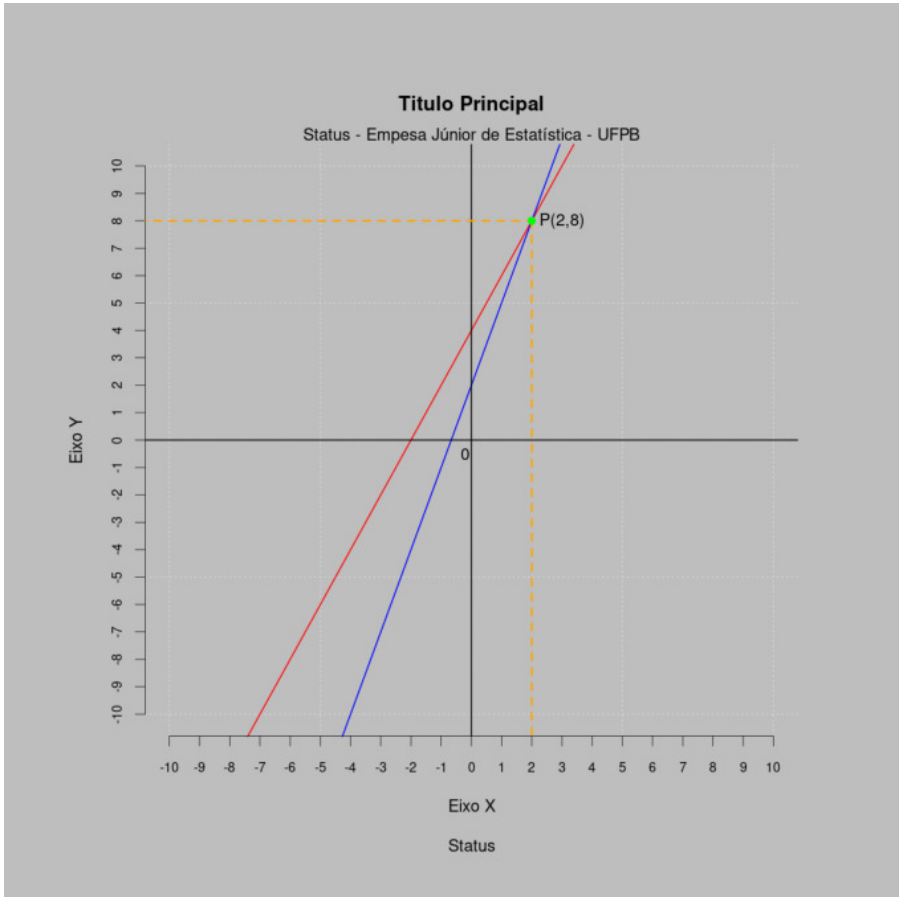


Figura: Gráfico produzido pelo código apresentado no exercício anterior.

Gráficos

A função `plot.window()` poderá ser utilizada em situações em que desejamos utilizar eixos invertidos em um gráfico, como o gráfico abaixo:

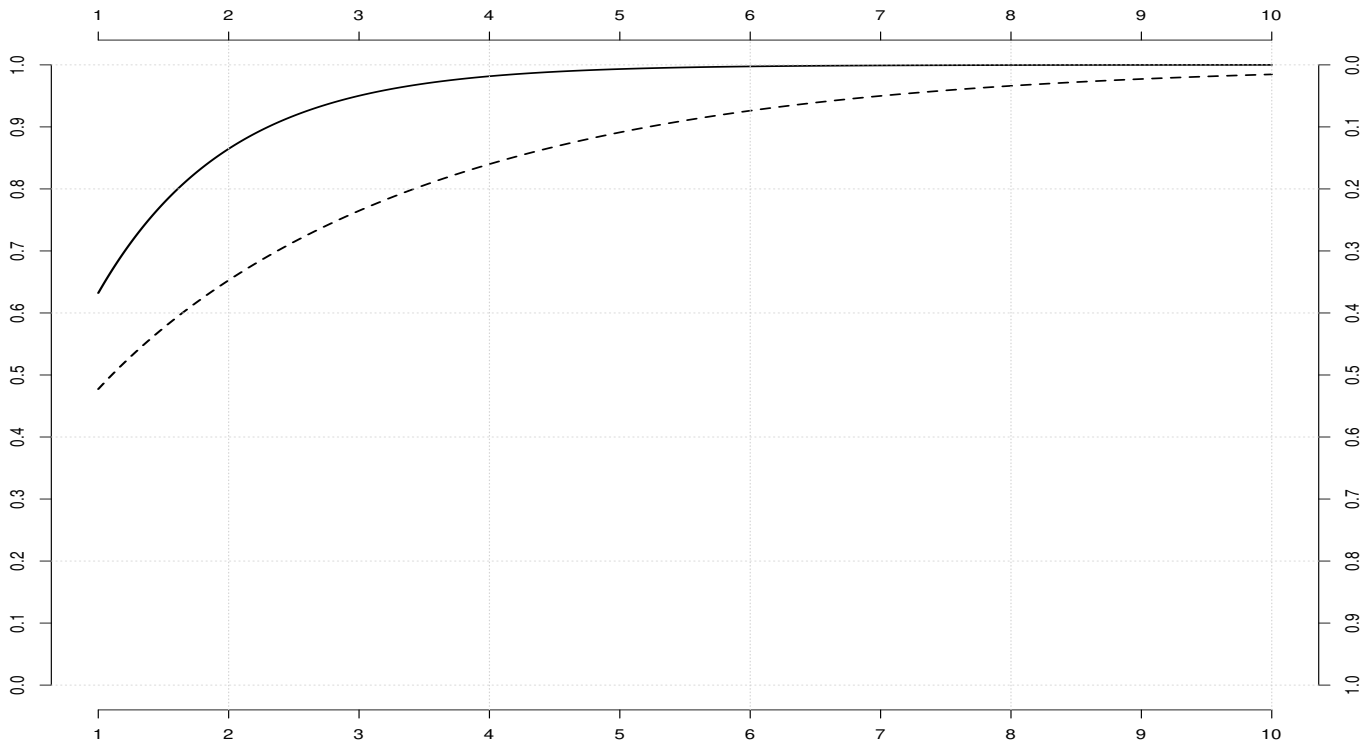


Figura: Gráficos com eixo *y* invertido.

Nota: O parâmetro `side` da função `axis()` especifica a margem e será colocado o eixo.

Gráficos

Nota: O parâmetro `side` da função `axis()` especifica a margem e será colocado o eixo.

```
1  # Abre uma janela em branco para o recebimento de comandos
       de plotagem.
2  plot.new()
3  plot.window(xlim = c(1, 10), ylim = c(1, 0))
4  axis(side = 4, at = rev(seq(0, 1, by = 0.1)))
5  # Correr novamente a funcao plot.window()
6  # permite que se possa definir novamente
7  # os limites dos eixos. O que ja foi desenhado
8  # permanecera sem alteracoes.
9  plot.window(xlim = c(1, 10), ylim = c(0, 1))
10 axis(side = 2, at = seq(0, 1, by = 0.1))
11 axis(side = 1, at = seq(1, 10, by = 1))
12 axis(side = 3, at = seq(1, 10, by = 1))
13 grid() # Colocando uma grade.
14 x <- seq(1,10, length.out = 500)
15 lines(x, pexp(x), lwd = 2, lty = 1)
16 lines(x, pchisq(x, 1, 1), lwd = 2, lty = 2)
```

Gráficos

No R, o padrão inicial de cores é definido pelas cores Black (preto), red (vermelho), green3 (verde), blue (azul), cyan, (ciano), magenta (magenta), yellow (amarelo) e gray (cinza). Corra o código abaixo para visualizar as cores.

```
1 barplot(rep(1,length(palette())), col = palette(),
2 names.arg= palette(), main = "Palheta de Cores", border =
      FALSE, axes = FALSE)
```

Nota: O função `palette()` retorna o nome das cores consideradas na palheta de cores. É possível fazer alterações nas cores consideradas na palheta.

Figura: Palheta de cores considera por padrão pela linguagem R.

Gráficos

A linguagem `R` também reconhece um conjunto mais amplo de cores que poderá ser listadas em um vetor atômico com os nomes das respectivas cores (são 657 cores a sua disposição). Corra o código `colors()`.

Gráficos

```
  [1] "white"           "aliceblue"         "antiquewhite"      "antiquewhite1"     "antiquewhite2"
  [6] "antiquewhite3"   "antiquewhite4"     "aquamarine"        "aquamarine1"       "aquamarine2"
 [11] "aquamarine3"     "aquamarine4"       "azure"             "azure1"            "azure2"
 [16] "azure3"          "azure4"            "beige"             "bisque"            "bisque1"
 [21] "bisque2"         "bisque3"           "bisque4"           "black"             "blanchedalmond"
 [26] "blue"            "blue1"             "blue2"             "blue3"             "blue4"
 [31] "blueviolet"      "brown"             "brown1"            "brown2"            "brown3"
 [36] "brown4"          "burlywood"         "burlywood1"        "burlywood2"        "burlywood3"
 [41] "burlywood4"      "cadetblue"         "cadetblue1"        "cadetblue2"        "cadetblue3"
 [46] "cadetblue4"      "chartreuse"        "chartreuse1"       "chartreuse2"       "chartreuse3"
 [51] "chartreuse4"     "chocolate"         "chocolate1"        "chocolate2"        "chocolate3"
 [56] "chocolate4"      "coral"             "coral1"            "coral2"            "coral3"
 [61] "coral4"          "cornflowerblue"    "cornsilk"          "cornsilk1"         "cornsilk2"
 [66] "cornsilk3"       "cornsilk4"         "cyan"              "cyan1"             "cyan2"
 [71] "cyan3"           "cyan4"             "darkblue"          "darkcyan"          "darkgoldenrod"
 [76] "darkgoldenrod1"  "darkgoldenrod2"    "darkgoldenrod3"    "darkgoldenrod4"    "darkgray"
 [81] "darkgreen"       "darkgrey"          "darkkhaki"         "darkmagenta"       "darkolivegreen"
 [86] "darkolivegreen1" "darkolivegreen2"   "darkolivegreen3"   "darkolivegreen4"   "darkorange"
 [91] "darkorange1"     "darkorange2"       "darkorange3"       "darkorange4"       "darkorchid"
 [96] "darkorchid1"     "darkorchid2"       "darkorchid3"       "darkorchid4"       "darkred"
[101] "darksalmon"      "darkseagreen"      "darkseagreen1"     "darkseagreen2"     "darkseagreen3"
[106] "darkseagreen4"   "darkslateblue"     "darkslategray"     "darkslategray1"    "darkslategray2"
```

Figura: Algumas das 657 cores reconhecidas pela linguagem.

Gráficos

Também é possível ampliar bastante o nosso universo de cores criando rampas entre duas ou mais cores. Dessa forma, poderemos obter cores intermediárias entre as cores que escolhemos para construir a rampa. Temos assim, uma espécie de degradê entre as cores, ampliando assim nosso leque de cores.

Exemplo: Utilizando a função `colorRampPalette()` poderemos construir uma rampa de cores em um intervalo de duas ou mais cores. Corra o código que segue:

Gráficos

Também é possível ampliar bastante o nosso universo de cores criando rampas entre duas ou mais cores. Dessa forma, poderemos obter cores intermediárias entre as cores que escolhemos para construir a rampa. Temos assim, uma espécie de degradê entre as cores, ampliando assim nosso leque de cores.

Exemplo: Utilizando a função `colorRampPalette()` poderemos construir uma rampa de cores em um intervalo de duas ou mais cores. Corra o código que segue:

```
1 rampa <- colorRampPalette(colors = c("cyan", "blue"))
2 cores <- rampa(20)
3 x <- rep(1, times = length(cores))
4 names(x) <- as.character(cores)
5 pie(x = x , col = cores, border = FALSE, cex = 0.7)
```

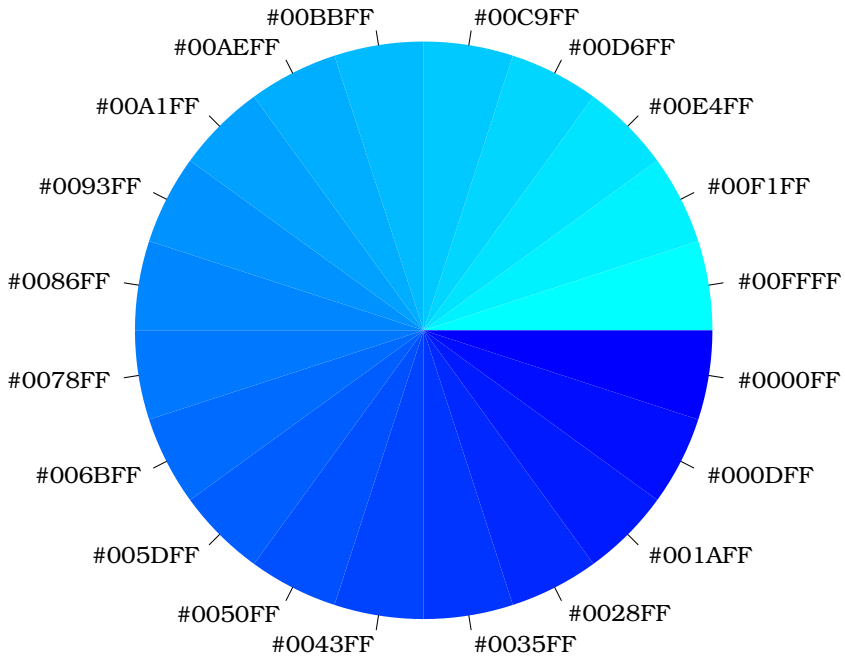


Figura: Rampa com 20 cores entre as cores **cyan** e **blue**. Cada cor representada por uma notação hexadecimal. No lugar de passar o nome da cor, poderemos passar ao parâmetro que controla a cor de um gráfico o número hexadecimal para o parâmetro da função que controla a cor do gráfico desejado.

Exemplo: É possível construir rampas muito longas de cores. No código abaixo construímos um gráfico de setores com rampa de 2 mil cores entre as cores **cyan** e **blue**.

Exemplo: É possível construir rampas muito longas de cores. No código abaixo construímos um gráfico de setores com rampa de 2 mil cores entre as cores **cyan** e **blue**.

```
1 rampa <- colorRampPalette(colors = c("cyan", "blue"))
2 cores <- rampa(2000)
3 x <- rep(1, times = length(cores))
4 pie(x = x , col = cores, border = FALSE, cex = 1.2, labels =
      NA)
```

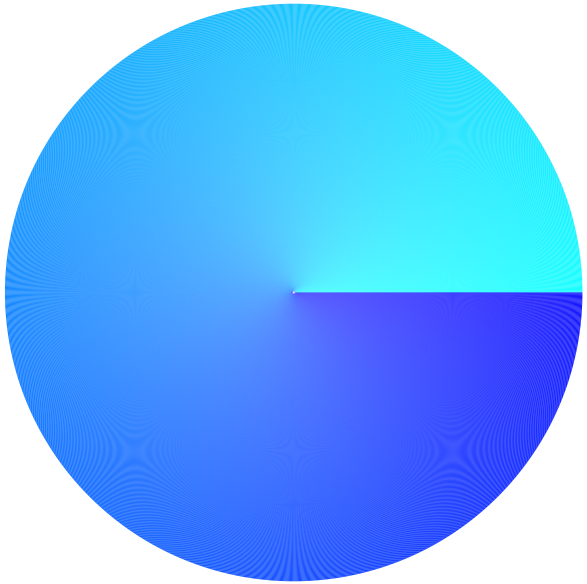


Figura: Rampa com 2000 cores entre as cores **cyan** e **blue**.

Exercício: Com a função `colorRampPalette()` é possível escolher mais de duas cores para criar uma rampa de cores. Construa um gráfico de setores considerando uma rampa de 2 mil cores entre as cores, sendo elas: **vermelho**, **ciano**, **azul**, **verde**, **amarelo**, **magenta** e novamente **vermelho**.

Nota: As cores são definidas por modelos matemáticos de cor onde são definidos os conceitos de espaço de cor de modo a obter uma discretização do espectro visível que torne viável a representação de cores em um computador. Sendo assim, existem diversos sistemas de cor utilizados para especificação e cálculos computacionais envolvendo cor. Com a função `colorRampPalette()` é possível utilizar os sistemas **RGB** (**R**ed, **G**reen e **B**lue) e **CIE Lab** que proporciona rampas de cores mais uniformes.

Solução pelo sistema RGB:

Solução pelo sistema RGB:

```
1 rampa <- colorRampPalette(colors = c("red","cyan", "blue", "
      green","yellow","magenta","red"),
2                           space = "rgb")
3 cores <- rampa(2000)
4 x <- rep(1, times = length(cores))
5 pie(x = x , col = cores, border = FALSE, cex = 1.2, labels =
       NA)
```

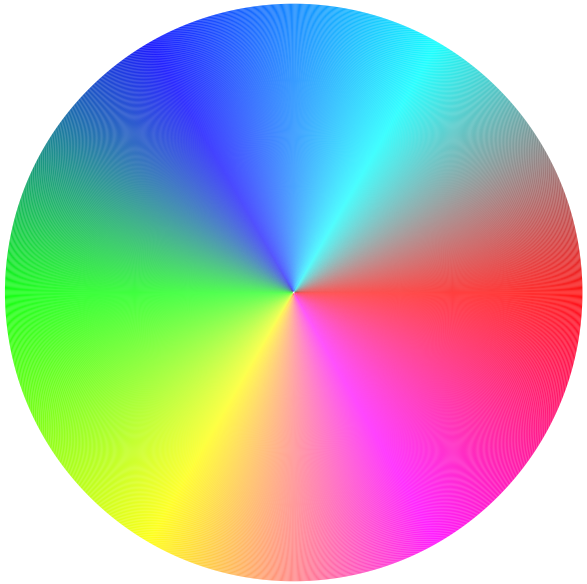


Figura: Rampa com 2000 cores pelo sistema **RGB**.

Solução pelo sistema CIE Lab:

Solução pelo sistema CIE Lab:

```
1 rampa <- colorRampPalette(colors = c("red","cyan", "blue", "
    green","yellow","magenta","red"),
2                           space = "Lab")
3 cores <- rampa(2000)
4 x <- rep(1, times = length(cores))
5 pie(x = x , col = cores, border = FALSE, cex = 1.2, labels =
     NA)
```

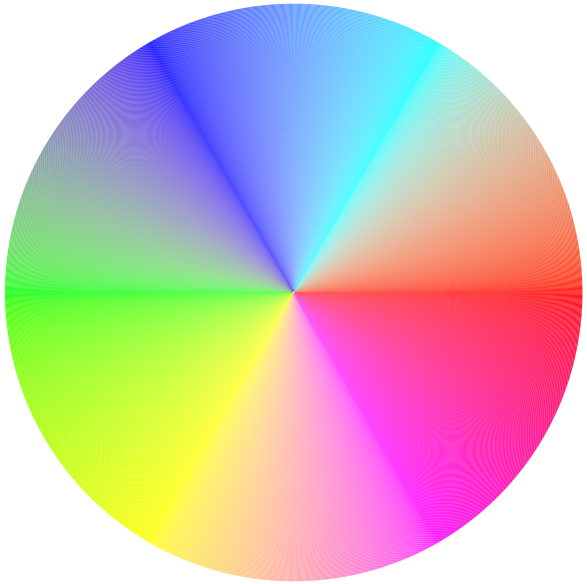


Figura: Rampa com 2000 cores pelo sistema **CIE Lab**.

Um pouco sobre o sistema RGB

O sistema de representação das cores **RGB** é um dos sistemas mais utilizados na computação mas não é o único. Como mencionado anteriormente, o sistema consiste em combinar luzes (**R**ed, **G**reen e **B**lue).

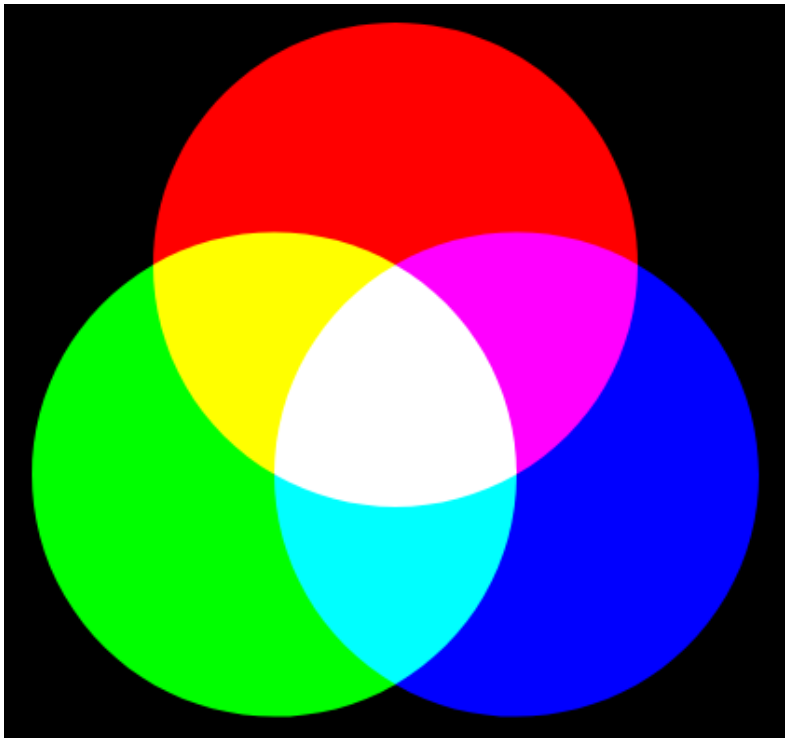


Figura: Sistema de representação de cores **RGB**.

Nota: Perceba que a combinação das cores-luzes primarias (vermelho, verde e azul) geram as cores-luzes ciano, magenta e amarela que são conhecida no mundo das artes de “cores primárias”.

Nota: Perceba que a combinação das cores-luzes primarias (vermelho, verde e azul) geram as cores-luzes ciano, magenta e amarela que são conhecida no mundo das artes de "cores primárias".

Importante

Aqui não estamos falando de combinação de pigmentos (tintas). O sistema **RGB** é válido para combinação de luzes. A depender da intensidade que combinamos as cores verdadeiramente primárias (vermelho, verde e azul), geraremos novas cores.

Gráficos

Cada pixel na tela de um computador ou de um smartphone é formado por um tubo de raios catódicos de cristal líquido, por exemplo, e pode ser representado com valores para vermelho, verde ou azul com as devidas intensidades.

A intensidade em que cada luz primária é empregada tem um alcance de 256 valores, sendo representadas de 0 (00 em hexadecimal - mais escuro) à 255 (FF em hexadecimal - mais claro).

Fazendo uma conta simples, podemos representar $256^3 = 16777216 \approx 16,7$ milhões de cores.

Gráficos

Sendo assim, temos que #000000 refere-se à cor-luz preta e `#FFFFFF` refere-se a cor-luz branca. Note que cada par de caracteres denota um número hexadecimal e ocupa no máximo 8 bits, uma vez que **255**, **FF** (hexadecimal) em binário é **11111111** (8 bits). Então,

Gráficos

Sendo assim, temos que #000000 refere-se à cor-luz preta e `#FFFFFF` refere-se a cor-luz branca. Note que cada par de caracteres denota um número hexadecimal e ocupa no máximo 8 bits, uma vez que **255**, **FF** (hexadecimal) em binário é **11111111** (8 bits). Então,

1. `#FF0000` - refere-se à cor-luz vermelha pura.

Gráficos

Sendo assim, temos que #000000 refere-se à cor-luz preta e `#FFFFFF` refere-se a cor-luz branca. Note que cada par de caracteres denota um número hexadecimal e ocupa no máximo 8 bits, uma vez que **255**, **FF** (hexadecimal) em binário é **11111111** (8 bits). Então,

1. `#FF0000` - refere-se à cor-luz vermelha pura.
2. `#00FF00` - refere-se à cor-luz verde pura.

Gráficos

Sendo assim, temos que #000000 refere-se à cor-luz preta e `#FFFFFF` refere-se a cor-luz branca. Note que cada par de caracteres denota um número hexadecimal e ocupa no máximo 8 bits, uma vez que **255**, **FF** (hexadecimal) em binário é **11111111** (8 bits). Então,

1. `#FF0000` - refere-se à cor-luz vermelha pura.
2. `#00FF00` - refere-se à cor-luz verde pura.
3. `#0000FF` - refere-se à cor-luz azul pura.

Exercício: Construa um gráfico de barras (use `barplot()`) com as cores primárias do sistema de representação de cores **RGB** utilizando a notação hexadecimal das cores.

Exercício: Construa um gráfico de barras utilizando a notação hexadecimal secundárias obtidas pelo sistema **RGB**.

Gráficos

Exercício: Utilizando o que foi apresentado cujo os dados são obtidos com o código `set.seed(0); rnorm(1000,0,1)`, construa um gráfico próximo ao apresentado a seguir. **Dica**: Valores menores ou iguais à -1.625 ou maiores ou iguais à 1.625 receberam maior nível de transparência.

Gráficos

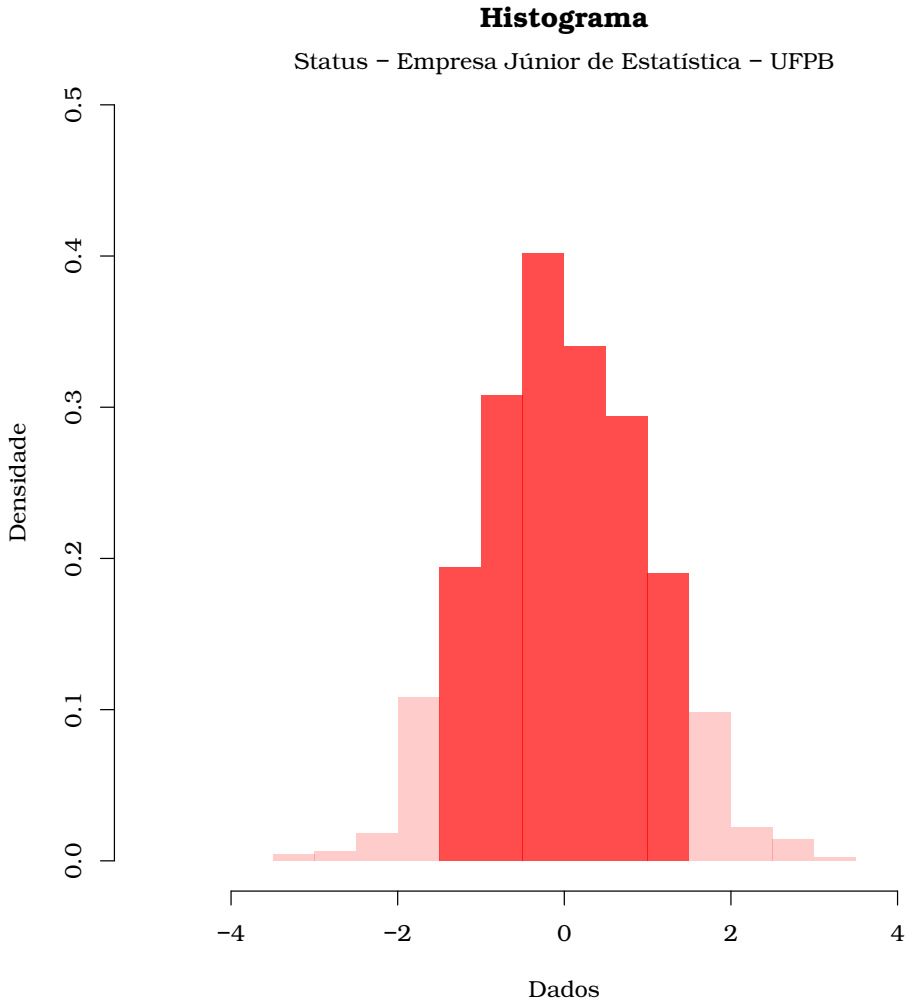


Figura: Histograma com estrutura e padrões de cores alterado.

Solução:

Solução:

```
1  set.seed(0); dados <- rnorm(1000,0,1)
2  histograma <- hist(dados)
3  # Salvando grafico no formato PDF.
4  pdf(file = "histograma_cor_1.pdf", width = 9, height = 9,
       paper = "special", pointsize = 14,
5      family = "Bookman")
6     plot.new()
7     plot.window(xlim = c(-5,5), ylim = c(0,0.5))
8     axis(1); axis(2);
9     title(xlab = "Dados", ylab = "Densidade", main = "
          Histograma")
10    mtext("Status - Empresa Junior de Estatistica - UFPB")
11    minhas_cores <- ifelse(histograma$mids <= quantile(
          histograma$mids)[2], rgb(1,0,0,0.2),
12    ifelse(histograma$mids >= quantile(histograma$mids)[4],
13       rgb(1,0,0,0.2), rgb(1,0,0,0.7)))
14    hist(dados, probability = TRUE, add = TRUE, col = minhas_
          cores, border = FALSE)
15 dev.off()
```

Exercício: Utilizando a função `lines()`, adicione ao gráfico acima a função densidade de $X \sim \mathcal{N}(0, 1)$.

Gráficos

```
set.seed(0); dados <- rnorm(1000,0,1)
histograma <- hist(dados)
# Salvando o grafico no formato PDF.
pdf(file = "histograma_cor_2.pdf", width = 9, height = 9,
    paper = "special", pointsize = 14,
family = "Bookman")
   set.seed(0); plot.new()
   plot.window(xlim = c(-5,5), ylim = c(0,0.5))
   axis(1); axis(2); dados <- rnorm(1000,0,1)
   title(xlab = "Dados", ylab = "Densidade", main = "
       Histograma")
   mtext("Status - Empresa Junior de Estatistica - UFPB")
   minhas_cores <- ifelse(histograma$mids <= quantile(
       histograma$mids)[2], rgb(1,0,0,0.2),
                          ifelse(histograma$mids >= quantile
                              (histograma$mids)[4],
   rgb(1,0,0,0.2), rgb(1,0,0,0.7)))
   hist(dados, probability = TRUE, add = TRUE, col = minhas_
       cores, border = FALSE)
   x <- seq(-4, 4, length.out = 2000)
   lines(x, dnorm(x, mean = 0, sd = 1), lwd = 3)
dev.off()
```

Gráficos

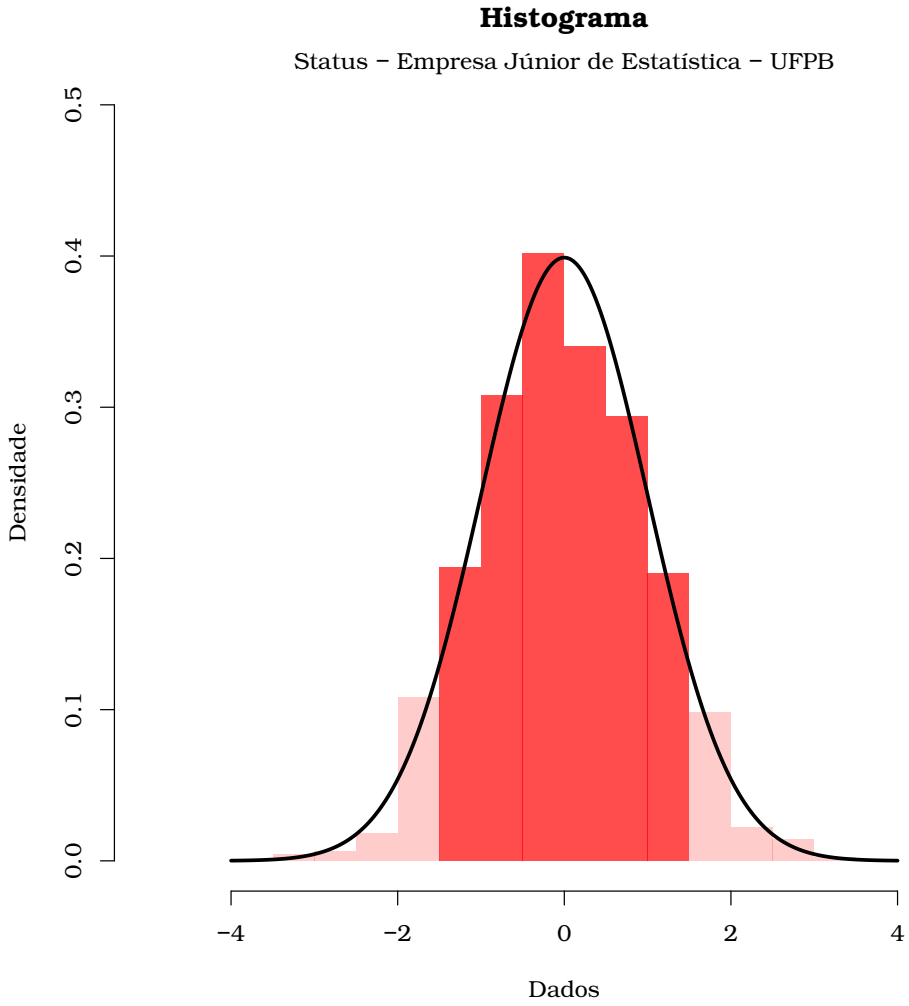


Figura: Gráfico produzido pelo código do exercício anterior.

Exercício: Adicione ao gráfico anterior um histograma de dados gerados pelo código `set.seed(0); rnorm(500, 4, 1.5)`. Adicione também a função densidade dos novos dados. **Dica**: Aumente a amplitude do domínio do gráfico anterior e ao utilizar o comando `hist()` na construção do novo histograma, certifique-se em adicionar o argumento **add = TRUE**.

Gráficos

```
1  set.seed(0); dados1 <- rnorm(1000,0,1)
2  set.seed(0); dados2 <- rnorm(500,4,1.5)
3  histograma1 <- hist(dados1); histograma2 <- hist(dados2)
4  pdf(file = "histograma_cor_3.pdf", width = 9, height = 9,
       paper = "special", pointsize = 14,
5      family = "Bookman")
6      plot.new(); plot.window(xlim = c(-5,10), ylim = c(0,0.5))
7      axis(1); axis(2)
8      title(xlab = "Dados", ylab = "Densidade", main = "
           Histograma")
9      mtext("Status - Empresa Junior de Estatistica - UFPB")
10     minhas_cores1 <- ifelse(histograma1$mids <= quantile(
           histograma1$mids)[2], rgb(1,0,0,0.2),
11     ifelse(histograma1$mids >= quantile(histograma1$mids)[4],
12     rgb(1,0,0,0.2), rgb(1,0,0,0.7)))
13     minhas_cores2 <- ifelse(histograma2$mids <= quantile(
           histograma2$mids)[2], rgb(1,0,0.7,0.4),
14     ifelse(histograma2$mids >= quantile(histograma2$mids)[4],
15     rgb(1,0,0.7,0.4), rgb(1,0,0.7,0.6)))
16     hist(dados1, probability = TRUE, add = TRUE, col = minhas
           _cores1, border = FALSE)
```

Gráficos

```
17    hist(dados2, probability = TRUE, add = TRUE, col = minhas
          _cores2, border = FALSE)
18    x <- seq(-4, 4, length.out = 2000)
19    lines(x, dnorm(x, mean = 0, sd = 1), lwd = 3)
20    x <- seq(-1, 9, length.out = 2000)
21    lines(x, dnorm(x, mean = 4, sd = 1.5), lwd = 3, lty = 2)
22 dev.off()
```

Gráficos

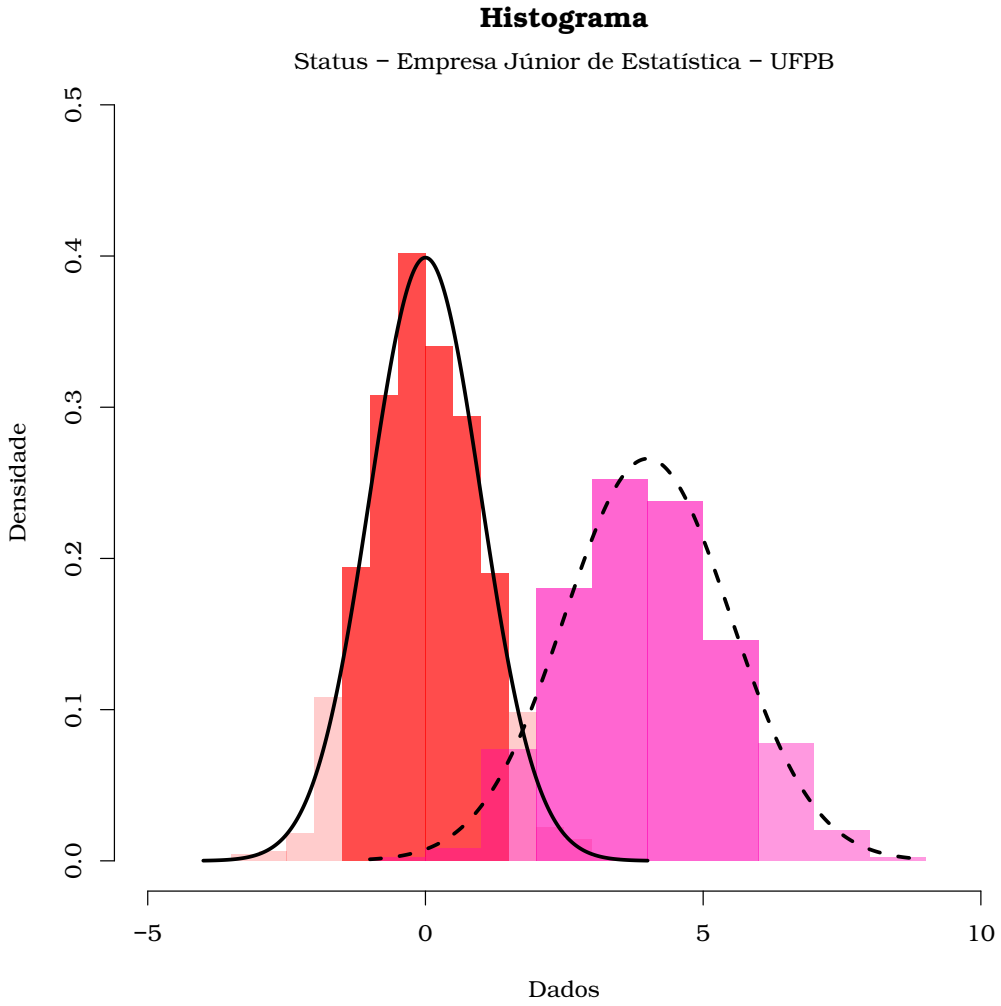


Figura: Gráfico produzido pelo código do exercício anterior.

Gráficos

Nota: Estude com detalhe o código do exercício anterior. Observe, por exemplo, que as bordas ao redor das barras do histograma são eliminadas fazendo **border = FALSE** na função `hist()`.

Ao construir um gráfico, é essencial sabermos acrescentar uma legenda para entendermos o que significa cada uma de suas componentes. Fazemos isso em `R` utilizando a função `legend()`.

Exercício: Leia a documentação da função `legend()`.

Exercício: Acrescente uma legenda ao gráfico do anterior especificando que a linha contínua refere-se à distribuição normal padrão e a curva pontilhada é a densidade da normal com $\mu = 4$ e $\sigma^2 = 1.5$. **Dica**: É possível utilizar a função `expression()` para escrever uma fórmula matemática em um título, label ou legenda de um gráfico. Por exemplo, corra o código `plot(1:10, main = expression(alpha + beta))`. Para detalhes de como escrever outras expressões matemáticas em `R`, faça `help(plotmath)` ou `demo(plotmath)`.

Solução:

Solução: Basta acrescentar ao final do código anterior o código que segue:

Solução: Basta acrescentar ao final do código anterior o código que segue:

```
# Adicionando uma legenda ao grafico produzido pelo codigo
# anterior.
# A funcao expression() serve para adicionarmos uma
# expressao matematica.
legend(x = 3.5, y = 0.4, legend = c(expression(N(mu == 0,
    sigma^2 == 1)), expression(N(mu == 4, sigma^2 == 1.5))),
     bg = "#EFEFEF", title = expression(bold("Distribuicoes
    de Probabilidade")), box.lwd = 0, lty = c(1,2), lwd = c
    (2,2), seg.len = c(3,3))
```

Solução: Basta acrescentar ao final do código anterior o código que segue:

```
# Adicionando uma legenda ao grafico produzido pelo codigo
# anterior.
# A funcao expression() serve para adicionarmos uma
# expressao matematica.
legend(x = 3.5, y = 0.4, legend = c(expression(N(mu == 0,
    sigma^2 == 1)), expression(N(mu == 4, sigma^2 == 1.5))),
     bg = "#EFEFEF", title = expression(bold("Distribuicoes
    de Probabilidade")), box.lwd = 0, lty = c(1,2), lwd = c
    (2,2), seg.len = c(3,3))
```

Nota: Note que o argumento **x** poderá receber **bottomright**, **bottom**, **bottomleft**, **left**, **topleft**, **top**, **topright**, **right** e **center**. Nesse caso, omitimos o argumento **y**.

Gráficos

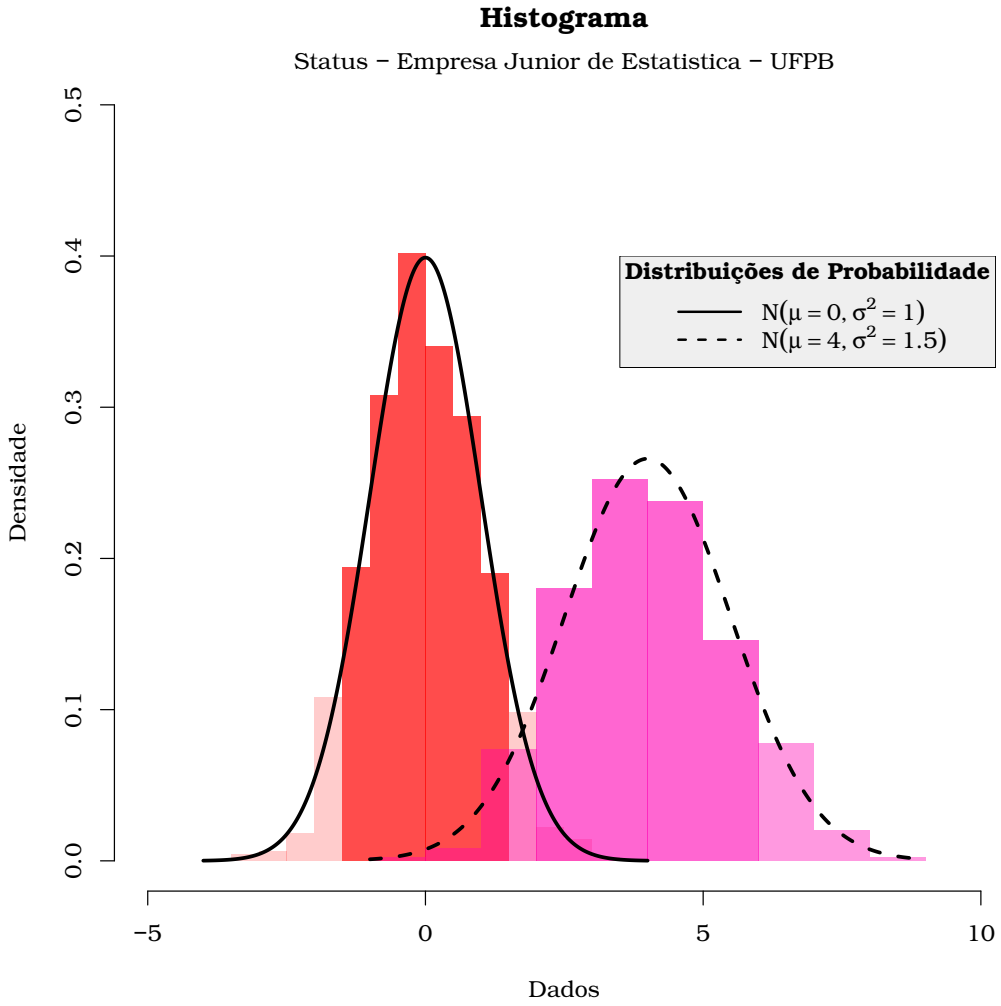


Figura: Gráfico produzido pelo código do exercício anterior.

Nota:

Como vimos anteriormente, é possível inserir diversas expressões matemáticas em gráficos produzidos em `R` (`help(plotmath)`). Porém, se quisermos ampliar a capacidade de tipografia de textos em gráficos, podemos fazer uso do pacote **tikzDevice**. Com o pacote **tikzDevice** é possível utilizar utilizar código LaTeX para produção de expressões matemáticas.

Nota:

Como vimos anteriormente, é possível inserir diversas expressões matemáticas em gráficos produzidos em R (`help(plotmath)`). Porém, se quisermos ampliar a capacidade de tipografia de textos em gráficos, podemos fazer uso do pacote **tikzDevice**. Com o pacote **tikzDevice** é possível utilizar utilizar código LaTeX para produção de expressões matemáticas.

Na verdade, o pacote converterá o código escrito em R para um código em LaTeX que produzirá o mesmo gráfico. Caso tenhamos interesse em obter um arquivo PDF, basta compilar o arquivo LaTeX gerado em um editor LaTeX ou diretamente no R fazendo:

Nota:

Como vimos anteriormente, é possível inserir diversas expressões matemáticas em gráficos produzidos em R (`help(plotmath)`). Porém, se quisermos ampliar a capacidade de tipografia de textos em gráficos, podemos fazer uso do pacote **tikzDevice**. Com o pacote **tikzDevice** é possível utilizar utilizar código LaTeX para produção de expressões matemáticas.

Na verdade, o pacote converterá o código escrito em R para um código em LaTeX que produzirá o mesmo gráfico. Caso tenhamos interesse em obter um arquivo PDF, basta compilar o arquivo LaTeX gerado em um editor LaTeX ou diretamente no R fazendo:

```
tools::texi2dvi(file = "arquivo.tex", pdf = TRUE)
```

Observação: O operador `::` é útil quando não carregamos um pacote com a função `library()`. Por exemplo, ao fazer

```
AdequacyModel::pso()
```

estamos fazendo uso da função `pso()` do pacote **AdequacyModel**, claro, se este tiver sido instalado. Utilizar o operador `::` é uma boa prática de programação uma vez que evita ambiguidades comum quando temos pacotes com funções com o mesmo nome. De forma geral temos:

```
nome_pacote::nome_funcao()
```

Gráficos

Exemplo: Construa o mesmo gráfico apresentado na solução do exercício anterior mudando a letra N na legenda por $\mathcal{N}$ obtido pelo comando LaTeX `\mathcal{N}`. **Dica**: Nesse exemplo, não necessitamos utilizar a função `expression()` uma vez que a expressão a ser inserida no gráfico não é uma expressão válida, por padrão, em `R`.

Gráficos

```
1  set.seed(0); dados1 <- rnorm(1000,0,1)
2  set.seed(0); dados2 <- rnorm(500,4,1.5)
3  histograma1 <- hist(dados1); histograma2 <- hist(dados2)
4
5  # A funcao tikz() utilizamos para converter e salvar a
       figura no formato TeX.
6  tikzDevice::tikz("grafico_em_latex.tex", width = 9, height =
       9, standAlone = TRUE)
7     plot.new(); plot.window(xlim = c(-5,10), ylim = c(0,0.5))
8     axis(1); axis(2)
9     title(xlab = "Dados", ylab = "Densidade", main = "
          Histograma")
10    mtext("Status - Empresa Junior de Estatistica - UFPB")
11    minhas_cores1 <- ifelse(histograma1$mids <= quantile(
          histograma1$mids)[2], rgb(1,0,0,0.2), ifelse(
          histograma1$mids >= quantile(histograma1$mids)[4],
          rgb(1,0,0,0.2), rgb(1,0,0,0.7)))
```

Gráficos

```
12    minhas_cores2 <- ifelse(histograma2$mids <= quantile(
          histograma2$mids)[2], rgb(1,0,0.7,0.4), ifelse(
          histograma2$mids >= quantile(histograma2$mids)[4],
          rgb(1,0,0.7,0.4), rgb(1,0,0.7,0.6)))
13    hist(dados1, probability = TRUE, add = TRUE, col = minhas
          _cores1, border = FALSE)
14    hist(dados2, probability = TRUE, add = TRUE, col = minhas
          _cores2, border = FALSE)
15    x <- seq(-4, 4, length.out = 2000)
16    lines(x, dnorm(x, mean = 0, sd = 1), lwd = 3)
17    x <- seq(-1, 9, length.out = 2000)
18    lines(x, dnorm(x, mean = 4, sd = 1.5), lwd = 3, lty = 2)
19    legend(x = 3.5, y = 0.4, legend = c("$\\mathcal{N}(\\mu =
           0,\\sigma^2 = 1)$", "$\\mathcal{N}(\\mu = 4, \\sigma
          ^2 = 1.5)$"), bg = "#EFEFEF", title = expression(bold
          ("Distribuicoes de Probabilidade")), box.lwd = 0, lty
           = c(1,2), lwd = c(2,2), seg.len = c(3,3))
20 dev.off()
```

Gráficos

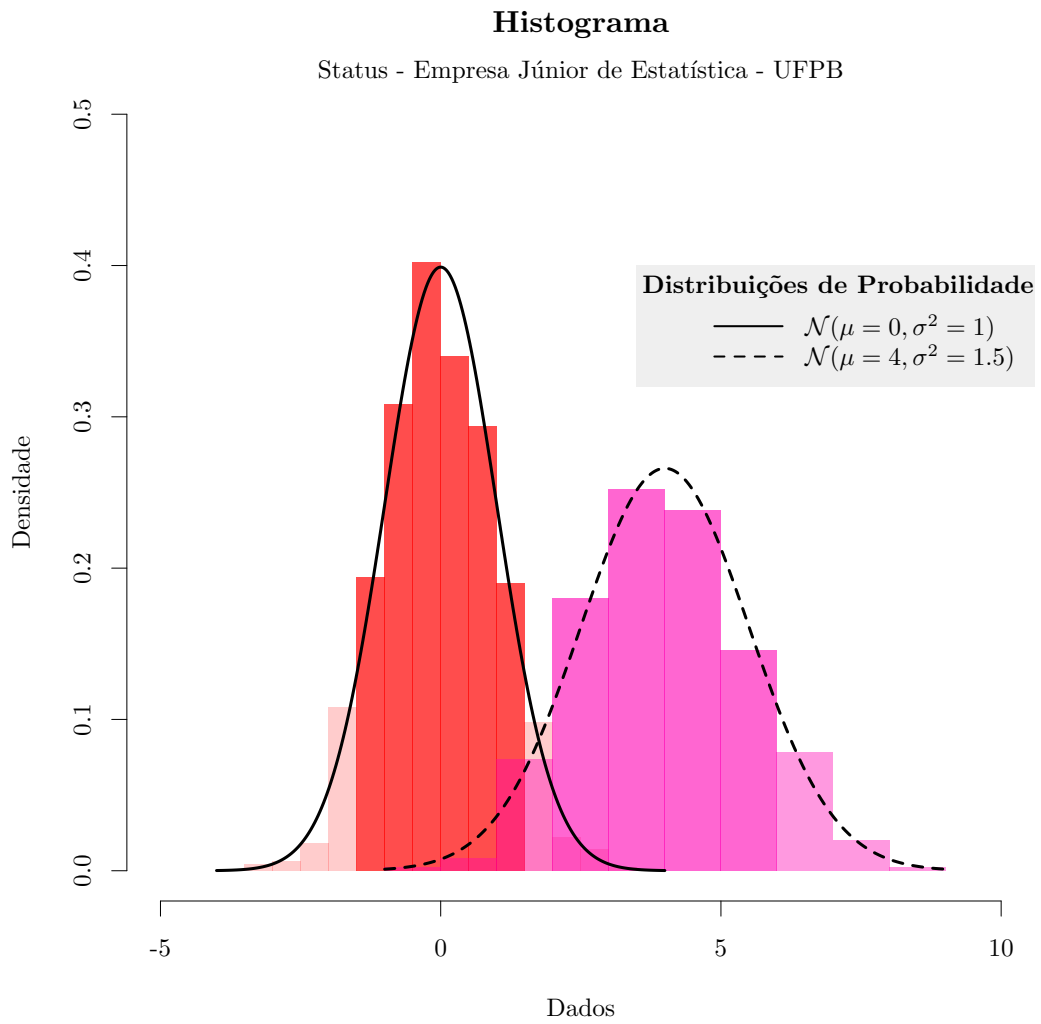


Figura: Gráfico produzido pelo código do exercício anterior.

Nota: Caso a figura possua palavras com acentuação, como é o caso da língua portuguesa, é necessário acrescentar no preâmbulo do arquivo TEX gerado o código

```
\usepackage[utf8]{inputenc}
```