Jessica Kubrusly

UMA INTRODUÇÃO À PROGRAMAÇÃO COM O R

Eduff

Jessica Kubrusly

UMA INTRODUÇÃO À PROGRAMAÇÃO COM O R

Eduff

Equipe de realização

Editor responsável: Renato Franco
Coordenador de produção: Ricardo Borges

Revisão: MC&G Editorial
Normalização: MC&G Editorial
Projeto gráfico, diagramação e capa: MC&G Editorial

Dados Internacionais de Catalogação-na-Publicação - CIP

K95 Kubrusly, Jessica.
Uma introdução à programação com o R [recurso eletrônico] / Jessica Kubrusly. – Niterói : Eduff, 2021. – 2,1 Mb. ; PDF. – (Biblioteca Básica).

Inclui bibliografia e índice.
ISBN 978-65-5831-118-8
BISAC MAT017000 MATHEMATICS / Linear & Nonlinear Programming

1. Estatística. 2. Programação. 3. Programa R. I. Título. II. Série.

CDD 005.131

Ficha catalográfica elaborada por Márcia Cristina dos Santos CRB7-4700

Direitos desta edição cedidos à
Eduff - Editora da Universidade Federal Fluminense
Rua Miguel de Frias, 9, anexo/sobreloja - Icaraí - Niterói - RJ
CEP 24220-008 - Brasil
Tel.: +55 21 2629-5287
www.eduff.uff.br - faleconosco.eduff@id.uff.br
Publicado no Brasil, 2022.
Foi feito o depósito legal.

SUMÁRIO

PREFÁCIO

Este texto foi desenvolvido a partir das notas de aula da disciplina de Programação Estatística, oferecida pelo Departamento de Estatística da Universidade Federal Fluminense (UFF). A ideia principal dessa disciplina, e também deste *ebook*, é reforçar conceitos de programação de computadores no Programa R. Dessa forma, o aluno passa a ver o Programa R não só como um *software* estatísticos com pacotes prontos, mas também como uma linguagem de programação em que ele pode criar seus próprios programas e, quem sabe, até pacotes.

O mais indicado é que o leitor já tenha tido contato com alguma linguagem de programação, isto é, que ele já conheça a lógica de programação. Mas acredito que não haja problema se esse for o seu primeiro curso de programação. Ao final do livro espera-se que o aluno tenha não só aprimorado sua habilidade computacional como também reforçado conceitos de estatística e de cálculo já vistos em outras disciplinas.

Gostaria de agradecer a todos os alunos que usaram este material antes da sua publicação e que por isso, de alguma forma, contribuíram para o seu aprimoramento. Gostaria de agradecer também à Universidade Federal Fluminense por oferecer um ambiente de trabalho onde foi possível desenvolver e publicar este livro. Por último, agradeço a todos os monitores que já orientei na disciplina de Programação Estatística e, em particular, agradeço ao monitor Lyncoln Sousa de Oliveira pela elaboração do gabarito dos exercícios.

Este livro é separado em três partes: a primeira parte, formada por cinco capítulos, expõe comandos básicos

de programação no R; a segunda parte, formada por três capítulos, apresenta a técnica de recursão, junto com muitos exemplos; por último, na terceira parte, que é formada por quatro capítulos, o leitor terá contato inicial com o cálculo numérico.

Para mais informações sobre a linguagem de programação R, entre no site oficial do projeto: *http://www.r-project.org/*. Lá é possível obter informações sobre como baixar e instalar o R em seu computador. Além disso, manuais introdutórios são disponibilizados gratuitamente.

PARTE I: CONCEITOS BÁSICOS DE PROGRAMAÇÃO NO R

Nessa primeira parte do livro serão vistos conceito básicos de programação no R. A ideia principal é aprender a usar o R para desenvolver suas próprias funções. É claro que as funções e pacotes já existentes no R são muito importantes e o leitor deve se sentir estimulado a usá-los a todo momento. Aprender a desenvolver nossas próprias funções nos torna capazes de resolver problemas mesmo quando não existem ainda funções prontas para isso.

CAPÍTULO 1: OBJETOS E CLASSES

Toda variável dentro da linguagem R é interpretada como um objeto que pertence a uma determinada classe. Para saber qual é a classe de um certo objeto podemos usar a função `class()`. Veja alguns exemplos.

```
a = 4
class(a)
## [1] "numeric"
class("abc")
## [1] "character"
```

Existem diversos tipos de classes. Algumas serão vistas ao longo deste curso e outras apenas em disciplinas futuras. Podemos citar alguns tipos básicos que serão vistos na aula de hoje: `"numeric"`, `"logical"`, `"character"`, `"matrix"` e `"list"`.

Números

Os objetos numéricos fazem parte da classe `"numeric"` e tratam-se dos números reais. Para criar um objeto desse tipo podemos usar, por exemplo, o comando `x<-3.4` ou `x = 3.4` (que são a mesma coisa). Para verificar a sua classe basta usar a função `class()`:

```
x = 3.4
class(x)
## [1] "numeric"
```

Para os objetos desse tipo já estão definidos alguns operadores (`+,-,*,/,^`) além de funções conhecidas, como por exemplo `sqrt()`, `log()`, `exp()`, `abs()`, entre outras.

```
a = 2 + 2; b = 3*a
a; b
## [1] 4
## [1] 12
sqrt(a)
## [1] 2
exp(a - 3)
## [1] 2.718282
log(b/12)
## [1] 0
abs(-b^2)
## [1] 144
```

Para saber mais detalhes sobre essas funções ou sobre a classe `"numeric"`, use o comando help do R.

```
help(log)
help("numeric")
```

Textos

Os objetos do tipo textos fazem parte da classe `"character"` e é dessa forma que o R guarda letras ou frases. Dentro do R, os textos sempre vão aparecer entre aspas. Para criar um objeto desse tipo podemos usar, por exemplo,

```
ch1<-"a";ch2<-"abc";ch3<-"1"
```

```
class(ch1);class(ch2);class(ch3)
## [1] "character"
## [1] "character"
## [1] "character"
```

Veja que `ch3` não é um número, e sim um texto, ou seja, é o número 1 armazenado como texto.

Uma função muito usada com objetos da classe `"character"` é a função `paste()`. Ela recebe como argumento de entrada objetos do tipo `"character"` e retorna um único objeto, também do tipo `"character"`, que é definido pela concatenação dos objetos passados como entrada. Quando chamamos o comando `paste()` podemos indicar como os objetos serão separados na concatenação usando o argumento de entrada `sep`. Se este argumento não for definido, ele será considerado como espaço.

```
paste("a","b","c")
## [1] "a b c"
paste("a","b","c",sep=",")
## [1] "a,b,c"
paste("a","b","c",sep="")
## [1] "abc"
ch4 <- paste(ch1,ch2,"a",sep=".")
ch4
## [1] "a.abc.a"
class(ch4)
## [1] "character"
```

Para saber mais sobre a classe `"character"`, consulte o *help* do R por meio do comando `help(character)`.

Lógicos

Os objetos do tipo lógicos fazem parte da classe `"logical"` e podem ser de dois tipos: `TRUE` ou `FALSE`. Eles também podem ser simplesmente representados pelas letras `T` ou `F`.

```
v<-TRUE
b<-T
class(v)
## [1] "logical"
class(b)
## [1] "logical"
v
## [1] TRUE
b
## [1] TRUE
```

Dentro do R já estão definidos os operadores lógicos E ($\cap$) e OU ($\cup$). Estes são aplicados a partir dos comandos `&&`, `&`, `||` e `|`. A tabela verdade a seguir indica a resposta para cada um desses dois operadores.

A	B	A && B	A || B
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

Vejamos alguns exemplos:

```
A = TRUE; B = FALSE
A&&B; A&B
## [1] FALSE
```

```
## [1] FALSE
A||B; A|B
## [1] TRUE
## [1] TRUE
```

Também podemos realizar a negação desses objetos usando o comando `!`.

```
NegA = !A; NegA
## [1] FALSE
NegB = !B; NegB
## [1] TRUE
```

Se quisermos testar se dois objetos são iguais ou diferentes, podemos usar os comandos `==` ou `!=` . A resposta será um objeto da classe `"logical"`: `TRUE` se os objetos forem iguais e `FALSE` se forem diferentes.

```
a<-1; b<-1; c<-2; d<-"2"
a==b
## [1] TRUE
a==c
## [1] FALSE
c==d
## [1] TRUE
```

Repare que no último comando ele respondeu `"TRUE"`, mesmo os objetos sendo de classes diferentes. O programa provavelmente converte todos para a mesma classe antes de testar se são iguais.

```
a!=b
```

```
## [1] FALSE
b!=c
## [1] TRUE
c!=d
## [1] FALSE
```

É possível também testar se um objeto é maior, menor, maior ou igual e menor ou igual com os operadores `>`, `<`, `>=` e `<=`. A resposta será um objeto do tipo `"logical"`. Veja nos exemplos a seguir que esses operadores podem ser usados entre objetos de vários tipos, não necessariamente entre números.

```
a>b
## [1] FALSE
b<=c
## [1] TRUE
"A"<"B"
## [1] TRUE
"A">="C"
## [1] FALSE
```

Podemos ainda usar os operadores `(+,-,*,/)` entre objetos do tipo `"logical"`. Nesse caso, os objetos iguais a `TRUE` serão interpretados como o número 1 e os iguais a `FALSE` como 0.

```
V_1 = TRUE; V_2 = TRUE; F_1 = FALSE; F_2 = FALSE
V_1 + V_2
## [1] 2
V_1 + F_1
```

```
## [1] 1
V_1 - V_2
## [1] 0
F_1 * V_2
## [1] 0
V_1 * V_2
## [1] 1
V_1/V_2
## [1] 1
F_1/V_2
## [1] 0
V_1/F_1
## [1] Inf
F_1/F_2
## [1] NaN
```

Para saber mais sobre a classe `"logical"`, consulte o help do R por meio do comando `help(logical)`.

Vetores

Antes de falarmos dos dois últimos tipos de objetos desta aula, vamos parar para entender um conceito importante em qualquer linguagem de programação, o conceito de vetor.

Em programação de computadores, um vetor é uma estrutura de dados que armazena uma coleção de elementos de tal forma que cada um dos elementos pode ser identificado por um índice. No R podemos entender como uma coleção de objetos, em que todos são de um mesmo tipo, obrigatoriamente.

Por exemplo, podemos ter um vetor de números reais (coleção de objetos do tipo `"numeric"`), um vetor de textos (coleção de objetos do tipo `"character"`), um vetor de "verdadeiro" ou "falso" (coleção de objetos do tipo `"logical"`), ou um vetor de objetos de qualquer outra classe do R.

No R existem várias maneiras de criarmos um vetor, a mais simples delas é usando a função `c()`, que combina objetos formando um vetor.

```
a<-c(1,2,3);a
## [1] 1 2 3
b<-c("a", "aa");b
## [1] "a"  "aa"
c<-c(T,T,F,F);c
## [1]  TRUE  TRUE FALSE FALSE
```

No R, vetor não é uma classe. Os objetos `a`, `b` e `c` são, respectivamente, das classes `"numeric"`, `"character"` e `"logical"`. Isso ocorre pois o R trata qualquer objeto como um *vetor*. No caso de ser um objeto simples, e não uma coleção de objetos, o R interpreta como se fosse um vetor de tamanho 1.

Uma função muito usada para vetores é a função `length()`, que retorna o tamanho do vetor.

```
length(a); length(b); length(c)
## [1] 3
## [1] 2
## [1] 4
```

Para acessarmos uma posição de um vetor usamos os colchetes (`[]`). Por exemplo,

```
a[1]
```

```
## [1] 1
b[2]
## [1] "aa"
c[3]
## [1] FALSE
```

Se acessarmos uma posição que não tenha sido definida, a resposta será `NA`, que representa o termo em inglês "*Not Available*".

```
a[5]
## [1] NA
```

Existem outras maneiras de criarmos um vetor dentro do R. Uma delas é usando o próprio colchetes (`[ ]`) para alocarmos uma posição específica.

```
d = 1
d
## [1] 1
d[2] = 3
d[4] = 5
d
## [1]  1  3 NA  5
d[4] = 7
d[3] = 5
d
## [1] 1 3 5 7
```

No exemplo anterior, o objeto `d` foi iniciado como um vetor de tamanho 1 da classe `"numeric"`. Temos também a possibilidade de iniciar um objeto como vazio, ou nulo.

```
e = NULL
e
## NULL
e[2] = 4
e
## [1] NA  4
e[1] = 2
e
## [1] 2 4
```

A função `c()` serve não só para concatenar objetos em um vetor, como também para concatenar um vetor com um novo objeto ou concatenar dois objetos, cada um já construído como um vetor. Lembrando que em um mesmo vetor todos os objetos são sempre da mesma classe.

```
a
## [1] 1 2 3
c(a,4)    #colocando um elemento no final do vetor
a
## [1] 1 2 3 4
c(0,a)    #colocando elemento no inicio do vetor
a
## [1] 0 1 2 3
a = c(0,a,4)
a
## [1] 0 1 2 3 4
```

```
e
## [1] 2 4
c(a,e)    #concatenando dois vetores
## [1] 0 1 2 3 4 2 4
b
## [1] "a"  "aa"
c(a,b)    #concatenando vetores de classes
diferentes, uma das classes eh transformada na
outra
## [1] "0"  "1"  "2"  "3"  "4"  "a"  "aa"
```

Temos ainda outras maneiras de criar um vetor no R, vejamos mais alguns exemplos:

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
seq(1,20,by=2)
##  [1]  1  3  5  7  9 11 13 15 17 19
rep(0,times=4)
## [1] 0 0 0 0
```

Uma última maneira de se iniciar um vetor quando já se sabe o tipo, mas ainda não sabemos os elementos:

```
v1 = numeric(2)
v1
## [1] 0 0
v2 = character(5)
v2
## [1] "" "" "" "" ""
```

```r
v3 = logical(4)
v3
## [1] FALSE FALSE FALSE FALSE
```

Antes de seguir para a próxima seção, veja que ainda é possível usar os operadores e funções das classes em uma vetor de tamanho maior que 1. Os operadores e funções acabam sendo aplicados a cada posição do vetor e retornam um vetor como resposta.

Veja primeiro alguns exemplos na classe `"numeric"`:

```r
a = seq(1:10)
b = rep(2,10)
a;b
##  [1]  1  2  3  4  5  6  7  8  9 10
##  [1] 2 2 2 2 2 2 2 2 2 2
a + b
##  [1]  3  4  5  6  7  8  9 10 11 12
a - b
##  [1] -1  0  1  2  3  4  5  6  7  8
a * b
##  [1]  2  4  6  8 10 12 14 16 18 20
a / b
##  [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
exp(a)
##  [1]     2.718282     7.389056    20.085537
54.598150   148.413159
##  [6]   403.428793  1096.633158  2980.957987
8103.083928 22026.465795
```

Veja mais alguns exemplos em outras classes:

```
a = c("a","aa","aaa")
b = c("b","bb","bbb")
paste(a,b)
## [1] "a b"     "aa bb"   "aaa bbb"
a1 = c("a","a","a")
a==a1
## [1]  TRUE FALSE FALSE
```

Matrizes

Dentro da linguagem R existe uma classe chamada `"matrix"`, que guarda objetos do mesmo tipo em forma de matriz, ou seja, guarda os objetos por linhas e colunas. Mesmo sendo uma estrutura de dados em que cada objeto pode ser identificado por um par de índices, como no caso do vetor, diferentemente do vetor as matrizes são uma classe no R.

Para criar um novo objeto do tipo `"matrix"` podemos usar a função `matrix(data=, nrow=, ncol=, byrow=, dimnames=)`. Ao lado do sinal de `=` devemos colocar as informações da matriz que está sendo criada:

NOME	DESCRIÇÃO
`data`	*vetor* com os objetos que serão alocados dentro da matriz
`ncol`	número de colunas da matriz
`nrow`	número de linhas da matriz
`byrow`	`TRUE` ou `FALSE`, que indica se os objetos serão alocados por linha (`TRUE`) ou por coluna (`FALSE`)
`dimnames`	uma lista com os nomes para as linhas e colunas da matriz

Se alguma das informações não for preenchida, será considerado o valor padrão para cada entrada, que nesse caso são:

```
data=NA, ncol=1, nrow=1, byrow = FALSE e dimnames = NULL.
```

Vejamos alguns exemplos de como criar matrizes no R:

```
m1 <- matrix(c(1,2,3,11,12,13), nrow = 2, ncol=3, byrow=TRUE)
m1
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
class(m1)
## [1] "matrix"
```

Veja que se mudarmos o número de linhas ou colunas, a matriz criada fica totalmente diferente:

```
m2 <- matrix(c(1,2,3,11,12,13), nrow = 3, ncol=2, byrow=TRUE)
m2
##      [,1] [,2]
## [1,]    1    2
## [2,]    3   11
## [3,]   12   13
```

Veja agora como ficaria a matriz se ela fosse alocada por colunas (`byrow=FALSE`):

```
m3 <- matrix(c(1,2,3,11,12,13), nrow = 3, ncol=2,
byrow=FALSE)
m3
##      [,1] [,2]
## [1,]    1   11
## [2,]    2   12
## [3,]    3   13
```

Quando criamos um objeto da classe `"matrix"`, o número de linhas e de colunas tem que ser determinado logo de início, e não pode ser alterado depois.

Os elementos de uma matriz também são acessados usando os colchetes (`[ , ]`), mas diferentemente do vetor, aqui temos que informar o índice da linha e da coluna que querermos acessar, separados por vírgula.

```
m3[2,1]
## [1] 2
```

Podemos usar os colchetes ainda para acessar uma linha inteira ou coluna inteira da matriz. Nesse caso, o objeto retornado será um vetor dos objetos da linha ou coluna da matriz.

```
m3[1,]
## [1]  1 11
m3[,2]
## [1] 11 12 13
```

Se quisermos o número de linhas de uma matriz, podemos usar a função `nrow()`, e se quisermos o número de colunas usamos a função `ncol()`.

```
nrow(m3)
```

```
## [1] 3
ncol(m3)
## [1] 2
```

Para saber mais sobre a classe `"matrix"`, consulte o *help* do R por meio do comando `help(matrix)`.

Listas

Na linguagem R a classe que guarda os objetos em forma de uma lista é denominada `"list"`. Um objeto da classe `"list"` se diferencia de um vetor principalmente pelo fato de poder guardar objetos de tipos diferentes. Além disso, as listas definem uma classe, e um vetor não.

Para criarmos uma lista vamos usar a função `list()`:

```
l1 <- list()
l1
## list()
```

O objeto `l1` criado representa uma lista vazia. Se dentro dos parênteses colocarmos uma sequência de objetos, a lista criada será composta por tais objetos.

```
l2 <- list(1,"a",TRUE)
l2
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
```

```
## [[3]]
## [1] TRUE
class(l2)
## [1] "list"
l3 <- list(c(1,2,3),c(1,1,1,1),TRUE,c("A","a"))
l3
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 1 1 1 1
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] "A" "a"
```

Podemos até criar uma lista de listas, por exemplo:

```
l4 <- list(l1,l2,l3)
```

Enquanto usamos colchetes simples (`[]`) para acessar uma posição de um vetor, para acessar as posições de uma lista usaremos colchetes duplo (`[[]]`).

```
l3[[2]]
## [1] 1 1 1 1
l3[[1]][2]
## [1] 2
```

Para saber o tamanho de uma lista, também podemos usar o comando `length()`.

```
length(l1)
## [1] 0
length(l2)
## [1] 3
length(l3)
## [1] 4
length(l4)
## [1] 3
```

Pergunta: Se digitarmos `length(l3[[2]])`, qual será a resposta retornada?

```
length(l3[[2]])
## [1] 4
```

Da mesma forma que em um vetor, novas posições de uma lista podem ser alocadas após a criação do objeto. No caso das listas, usaremos o colchete duplo para isso (`[[]]`). Além de alocar novas posições, ele também serve para modificar posições já existentes.

```
l2
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
```

```
## [[3]]
## [1] TRUE
l2[[4]]<-c(1,2,3)
l2
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1 2 3
```

Para saber mais sobre a classe `"list"`, consulte o *help* do R por meio do comando `help(list)`.

Data.frame

A última classe de objetos abordada neste texto será a classe `"data.frame"`. Trata-se de um objeto que guarda dados em forma bidimensional, como uma tabela. Cada valor guardado em objetos desse tipo pode ser acessado pelos índices de linha e coluna em que ele está alocado.

Para criar um novo objeto do tipo `"data.frame"` podemos usar a função `data.frame()`, e dentro dos parênteses indicamos as colunas do objeto em forma de um vetor. Importante: cada vetor que define uma coluna tem que ter a mesma dimensão, que será o número de linhas do objeto

"data.frame" criado. Vejamos alguns exemplos de como criar "data.frame" no R.

```
dados = data.frame(c("Maria","Daniel","Vicente",
"Laura"),c(4,5,1,3),c(F,F,F,T),stringsAsFactors =
FALSE)

class(dados)

## [1] "data.frame"

dados

##   c..Maria....Daniel....Vicente....Laura..
c.4..5..1..3. c.F..F..F..T.

## 1              Maria             4         FALSE

## 2             Daniel             5         FALSE

## 3            Vicente             1         FALSE

## 4              Laura             3          TRUE
```

No exemplo anterior foi criado o objeto dados do tipo "data.frame". O comando ringsAsFactors = FALSE colocado após os dados, ainda dentro da função data.frame(), faz com que os objetos do tipo "character" não sejam convertidos para o tipo "factor". Caso esse comando seja omitido os objetos do tipo "character" serão convertidos para "factor".

Para nomear as colunas, ou editar seus nomes, e assim facilitar a manipulação dos dados, você pode usar a função names(). Essa função serve tanto para retornar os nomes das colunas de um objeto "data.frame" quanto para editá-los.

```
names(dados)

## [1] "c..Maria....Daniel....Vicente....Laura.."

## [2] "c.4..5..1..3."

## [3] "c.F..F..F..T."
```

```r
names(dados) = c("nome", "periodo", "mora.
em.niteroi")
names(dados)
## [1] "nome"            "periodo"         "mora.
em.niteroi"
dados
##      nome periodo mora.em.niteroi
## 1   Maria       4           FALSE
## 2  Daniel       5           FALSE
## 3 Vicente       1           FALSE
## 4   Laura       3            TRUE
```

Uma alternativa seria criar o objeto `"data.frame"` já com os nomes das colunas desejados. Para isso, basta criar cada vetor coluna com o nome escolhido. Veja como chegar no mesmo resultado do exemplo anterior, agora com essa outra possibilidade.

```r
nome = c("Maria","Daniel","Vicente","Laura")
periodo = c(1,2,1,0)
mora.em.niteroi = c(F,T,F,T)
dados = data.frame(nome,periodo,mora.
em.niteroi,stringsAsFactors = FALSE)
dados
##      nome periodo mora.em.niteroi
## 1   Maria       1           FALSE
## 2  Daniel       2            TRUE
## 3 Vicente       1           FALSE
## 4   Laura       0            TRUE
names(dados)
```

```
## [1] "nome"      "periodo"   "mora.em.niteroi"
```

Outra forma ainda de fazer a mesma coisa:

```
dados = data.frame(nome =
c("Maria","Daniel","Vicente","Laura"),periodo
= c(1,2,1,0),mora.em.niteroi =
c(F,T,F,T),stringsAsFactors = FALSE)
dados
##      nome periodo mora.em.niteroi
## 1   Maria       1           FALSE
## 2  Daniel       2            TRUE
## 3 Vicente       1           FALSE
## 4   Laura       0            TRUE
names(dados)
## [1] “nome”      “periodo”   “mora.em.niteroi”
```

Cada elemento de um objeto `"data.frame"` pode ser acessado usando o comando colchetes (`[,]`) e indicando antes da vírgula o índice da linha e depois da vírgula o índice da coluna.

```
dados[1,1]
## [1] "Maria"
dados[3,2]
## [1] 1
```

Com o comando `[,]` também é possível acessar uma linha ou uma coluna inteira de um objeto do tipo `"data.frame"`. Nesse caso você deve indicar somente o índice da linha, para acessar uma linha, ou somente o índice da coluna, para acessar uma coluna.

```
dados[2,]
##     nome periodo mora.em.niteroi
## 2 Daniel       2            TRUE
dados[,3]
## [1] FALSE  TRUE FALSE  TRUE
```

Também é possível acessar cada coluna usando o comando `$` seguido pelo nome da coluna.

```
dados$nome
## [1] "Maria"   "Daniel"  "Vicente" "Laura"
dados$periodo
## [1] 1 2 1 0
dados$mora.em.niteroi
## [1] FALSE  TRUE FALSE  TRUE
```

As funções `nrow()` e `ncol()`, já vistas, também podem ser usadas para retornar o número de linhas e colunas de um objeto do tipo `"data.frame"`.

```
ncol(dados)
## [1] 3
nrow(dados)
## [1] 4
```

Vejamos agora como incluir novos dados em um objeto do tipo `"data.frame"` já criado. Com o comando `$` é possível adicionar uma nova coluna.

```
dados
##      nome periodo mora.em.niteroi
## 1   Maria       1           FALSE
```

```
## 2  Daniel       2            TRUE
## 3 Vicente       1           FALSE
## 4   Laura       0            TRUE
dados$curso = c("estatistica","engenharia",
"matematica","engenharia")
dados
##      nome periodo mora.em.niteroi       curso
## 1   Maria       1           FALSE estatistica
## 2  Daniel       2            TRUE  engenharia
## 3 Vicente       1           FALSE  matematica
## 4   Laura       0            TRUE  engenharia
```

Para adicionar uma nova linha, é preciso primeiro criar um objeto do tipo `"data.frame"` com os dados da nova linha e então juntar os dois objetos do tipo `"data.frame"` em um único objeto com a função `rbind()`. É importante que os dois objetos do tipo `"data.frame"` combinados pela função `rbind()` tenham as mesmas colunas, com os mesmos nomes inclusive. A função `rbind()` retorna um novo objeto do tipo `"data.frame"` formado pelos dois objetos passados como argumento, como se colocássemos "um embaixo do outro".

```
nova_linha = data.frame(nome =
"Thiago",periodo = 3,mora.em.niteroi =
F,curso="medicina",stringsAsFactors = FALSE)
nova_linha
##     nome periodo mora.em.niteroi    curso
## 1 Thiago       3           FALSE medicina
dados_completo = rbind(dados,nova_linha)
dados_completo
```

```
##      nome periodo mora.em.niteroi       curso
## 1   Maria       1           FALSE estatistica
## 2  Daniel       2            TRUE  engenharia
## 3 Vicente       1           FALSE  matematica
## 4   Laura       0            TRUE  engenharia
## 5  Thiago       3           FALSE    medicina
```

Se você quiser manter o mesmo nome que já tinha, basta alocar o retorno da função `rbind()` ao objeto já existente.

```
dados
##      nome periodo mora.em.niteroi       curso
## 1   Maria       1           FALSE estatistica
## 2  Daniel       2            TRUE  engenharia
## 3 Vicente       1           FALSE  matematica
## 4   Laura       0            TRUE  engenharia
dados = rbind(dados,nova_linha)
dados
##      nome periodo mora.em.niteroi       curso
## 1   Maria       1           FALSE estatistica
## 2  Daniel       2            TRUE  engenharia
## 3 Vicente       1           FALSE  matematica
## 4   Laura       0            TRUE  engenharia
## 5  Thiago       3           FALSE    medicina
```

Na prática, o que a função `rbind()` faz é juntar dois objetos `"data.frame"` em um só. Então ela serve não somente para colocar uma nova linha, mas também pode ser

usada para colocar várias novas linhas em um objeto do tipo "data.frame" já existente.

```
novas_linhas = data.frame(nome = c("Ana","Pedro"),
periodo = c(2,1), mora.em.niteroi = c(T,T) ,curso=c
("estatistica","engenharia"),stringsAsFactors =
FALSE)

dados = rbind(dados,novas_linhas)

dados

##      nome periodo mora.em.niteroi       curso
## 1   Maria       1           FALSE estatistica
## 2  Daniel       2            TRUE  engenharia
## 3 Vicente       1           FALSE  matematica
## 4   Laura       0            TRUE  engenharia
## 5  Thiago       3           FALSE    medicina
## 6     Ana       2            TRUE estatistica
## 7   Pedro       1            TRUE  engenharia
```

O que já deve ter sido notado, diferentemente dos objetos do tipo "matrix", é que nem todos os objetos guardados em um "data.frame" precisam ser do mesmo tipo. Em um "data.frame", as colunas representam valores de uma mesma variável e por isso estes precisam ser do mesmo tipo. Cada coluna de um "data.frame" é um vetor de um certo tipo.

```
dados$nome

## [1] "Maria"   "Daniel"  "Vicente" "Laura"
"Thiago"  "Ana"     "Pedro"

class(dados$nome)

## [1] "character"

dados$periodo
```

```
## [1] 1 2 1 0 3 2 1
class(dados$periodo)
## [1] "numeric"
dados$mora.em.niteroi
## [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
class(dados$mora.em.niteroi)
## [1] "logical"
```

Já as linhas são objetos do tipo `"data.frame"` e podem guardar objetos de diferentes tipos.

```
dados[3,]
##      nome periodo mora.em.niteroi      curso
## 3 Vicente       1           FALSE matematica
class(dados[1,])
## [1] "data.frame"
dados[6,]
##   nome periodo mora.em.niteroi       curso
## 6  Ana       2            TRUE estatistica
class(dados[6,])
## [1] "data.frame"
```

Para saber mais sobre a classe `"data.frame"`, consulte o *help* do R por meio do comando `help(data.frame)`.

Exercícios

1. Defina primeiro os objetos `x`, `y` e `z` como sendo do tipo `"numeric"` que guardam os valores `0`, `-1` e `32`, respectivamente. Para cada item a seguir defina os novos objetos

apresentados a partir de comandos e funções do R. Ao final verifique se os objetos criados guardam o valor que você esperava. Alguns dos resultados podem dar erro, tente justificar o motivo.

a. $a_1 = x + y + z, a_2 = yz, a_3 = \frac{z}{y}$

b. $b_1 = z^2, b_2 = z^3, b_3 = z^x$

c. $c_1 = \sqrt{a_1}, c_2 = \sqrt{x}, c_3 = \sqrt{\frac{b_2}{a_1}}$

d. $d_1 = \sqrt[3]{a_2}, d_2 = \sqrt[4]{-\frac{1}{a_3}}, d_3 = \sqrt[3]{z^2}$

e. $e_1 = |x|, e_2 = \sqrt{|a_2|}, e_3 = \left|\frac{1}{z}\right|$

f. $f_1 = e^x, f_2 = e^{x+y+z}, f_3 = e^{a_3}$

g. $g_1 = ln(x), g_2 = ln(x + y + z), g_3 = ln(yz)$

h. $h_1 = \sqrt{\pi}, h_2 = \sqrt{e^{-x^2}}, h_3 = \sqrt{3ln\left(\frac{4}{a_3}\right)}$

Obs.1: Veja que o objeto `pi`, pré-definido no R, guarda o valor (aproximado) para π.
Obs.2: Como podemos encontrar um valor aproximado pelo R para o número irracional *e*?

2. Defina `ch1="a"`, `ch2="b"` e `ch3="c"`, objetos do tipo `"character"`.

a. Usando a função `paste()` a partir de `ch1`, `ch2` e `ch3` crie um quarto objeto, também da classe `"character"`, `ch4`, definido como `"a.b.c"`.

b. Usando a função `paste()` a partir de `ch1`, `ch2` e `ch3` crie um quinto objeto, também da classe `"character"`, `ch5`, definido como `"abc"`.
c. Usando o comando `==` verifique se `ch4` e `ch5` são iguais ou diferentes.
d. Usando o comando `!=` verifique se `ch4` e `ch5` são iguais ou diferentes.

3. O operador `%%` fornece o resto da divisão entre dois números, por exemplo, `15%%4` retorna o resto da divisão de 15 por 4, que é 3. Esse comando será bastante usado durante o curso. Faça os itens a seguir primeiros no papel e depois verifique a resposta usando o R.

a. Qual é a resposta para `18%%5`, `-5%%2`, `15%%5` e `8.3%%3`?
b. Seja a um objeto do tipo `"numeric"`, que não sabemos o valor guardado nele. Sem ver o valor de a, como podemos usar o operador `%%` para testar se a guarda um número par? Faça o teste no *prompt* do R e use também os operadores `==` ou `!=` de forma que a resposta seja `TRUE` se o número for par e `FALSE` caso contrários.
c. Novamente sem ver o valor de `a`, como podemos usar o operador `%%` para testar se a guarda um número inteiro? Faça o teste no *prompt* do R e use também os operadores `==` ou `!=` de forma que a resposta seja `TRUE` se o número for inteiro e `FALSE` caso contrários.
d. Novamente sem ver o valor de `a`, como podemos testar se a guarda um número natural, isto é, inteiro e positivo? Faça o teste no *prompt* do R de forma que a resposta seja `TRUE` se o número for natural e `FALSE` caso contrário.

4. Digite no *prompt* do R: `a<-seq(1:10); b<-seq(1,20,by=2); d<-seq(20,1,by=-2)`. Usando os operadores `+,-,*,/` e também `==`, `!=`, `<`, `>` faça o que se pede nos itens a seguir.

a. Crie um vetor `x` em que cada posição de `x` é dada pela subtração entre as respectivas posições de `b` e `d`.

b. Crie um vetor `y` em que cada posição de `y` é o dobro de cada posição de `a`.
c. Crie um vetor `z` em que cada posição de `z` é um objeto da classe `"logical"`. A i-ésima posição de `z` vai guardar `TRUE` se `a[i]` for igual a `b[i]` e `FALSE` caso contrário.
d. Crie um vetor `w` em que cada posição de `w` é um objeto da classe `"logical"`. A i-ésima posição de `w` vai guardar `TRUE` se `d[i]` for maior que `b[i]` e `FALSE` caso contrário.

5. No R já existem alguns objetos predefinidos que são chamados de constantes. Como exemplo temos a constante `pi`, já usada no Exercício 1. Tem também `letters` e `LETTERS`, objetos do tipo `"character"` em forma de vetordefinidos pelas letras minúsculas e maiúsculas do alfabeto. Estes dois últimos serão usados neste exercício.

a. Primeiro digite `letters` e `LETTERS` para ver como são exatamente esses objetos.
b. Qual é a classe dos objetos `letters` e `LETTERS`? Primeiro tente responder sem usar a função `class()` e depois verifique a sua resposta usando-a.
c. Qual é a função que podemos usar para encontrar o tamanho de cada vetor `letters` e `LETTERS`? Use essa função e descubra o tamanho deles.
d. Se digitarmos `a<-c(LETTERS,letters)`, qual é a classe do objeto `a`, qual é o seu tamanho e como é este objeto? Tente responder antes sem o uso do computador e depois use o R para verificar a sua resposta.
e. Se digitarmos `b<-paste(LETTERS,letters)`, qual é a classe do objeto `b`, qual é o seu tamanho e como é este objeto. Tente responder antes sem o uso do computador e depois use o R para verificar a sua resposta.

6. Crie as seguintes matrizes no R:

```
##      [,1] [,2]
```

```
## [1,]    1  101
## [2,]    2  102
## [3,]    3  103
## [4,]    4  104
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]  101  102  103  104
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

7. Digite no *prompt* do R o seguinte comando: `A<-matrix-(c(1,2,3,4,5,6,7,8,9,10,11,12),4,3)`. Quais são as funções que podemos usar para encontrar o número de linhas e colunas da matriz `A`? Use tais funções e veja como elas funcionam.

8. Crie um objeto do tipo `"list"` com 4 elementos e dê o nome de `minha_lista`. O primeiro elemento é o seu nome (`"character"`). O segundo é sua idade (`"numeric"`). O terceiro é um vetor que guarda suas medidas de altura e peso, nessa ordem, em metros e Kg (`"numeric"`). E o quarto elemento é outro vetor que guarda `TRUE` para as respostas afirmativas e `FALSE` para as respostas negativas (`"logical"`) das seguintes perguntas: (i) Você já estagiou?; (ii) Você já participou de algum projeto como voluntário? (iii) Você tem interesse em assuntos relacionados ao meio ambiente?.

a. A partir do objeto `minha_lista` criado acesse o seu nome.
b. A partir do objeto `minha_lista` criado acesse a sua idade.

c. A partir do objeto `minha_lista` criado acesse a sua altura.
d. A partir do objeto `minha_lista` criado acesse o seu peso.
e. A partir do objeto `minha_lista` criado acesse a resposta para a pergunta: "Você tem interesse em assuntos relacionados ao meio ambiente?".

9. Como generalização do exercício anterior, vamos guardar os dados de mais de um aluno agora em um objeto do tipo `"data.frame"`. Os dados dos alunos desse `"data.frame"` serão: o nome (`"character"`); a idade em anos (`"numeric"`); a altura em metros (`"numeric"`); o peso em Kg (`"numeric"`); já estagiou? `TRUE` para sim e `FALSE` para não (`"logical"`); já participou de algum projeto como voluntário? `TRUE` para sim e `FALSE` para não (`"logical"`); tem interesse em assuntos relacionados ao meio ambiente? `TRUE` para sim e `FALSE` para não (`"logical"`). Repare que o objeto do tipo `"data.frame"` criado terá 7 colunas. Faça primeiro um objeto como sugerido com os dados de pelo menos três alunos, os seus e de mais dois amigos (ou dados fictícios). Chame esse objeto de `dados_alunos`.
a. A partir do objeto `dados_alunos` criado acesse o seu nome.
b. A partir do objeto `dados_alunos` criado acesse a altura de um dos seus amigos.
c. A partir do objeto `dados_alunos` criado crie um vetor com o nome de todos os alunos que responderam às perguntas. Esse objeto é de que classe?
d. A partir do objeto `dados_alunos` criado crie um vetor com todas as respostas de um de seus amigos.
e. Adicione ao objeto `dados_alunos` criado mais uma informação: o número de matrícula de cada aluno.
f. Repare que depois de feito o item (e) o objeto `dados_alunos` terá 8 colunas. Verifique isso com a função `ncol()`.

g. Adicione os dados de mais um amigo (ou dados fictícios).
h. Repare que depois de feito o item g o objeto `dados_alunos` terá mais uma linha. Verifique quantas linhas tem o objeto `dados_alunos` com a função `nrow()`.

10. Qual é a diferença entre os objetos `obj1`, `obj2` e `obj3` definidos a seguir?

```
obj1 <- list(1,2,3); obj2 <- list(c(1,2,3)); obj3
<- c(1,2,3)
```

11. Faça esse exercício sem o computador e depois use o computador para verificar a resposta. Imagine que sejam definidos no R os seguintes objetos: `X<-3; Y<-2; Z<-"2"; A<-"X"; B<-X; C<-"A"; D<-paste(A,C); E<-c(A,C); f<-list(A,C)`. Para cada item a seguir, diga se os objetos apresentados são iguais ou não. Caso eles sejam diferentes, explique a diferença entre eles.

a. `Y` e `Z`
b. `A` e `B`
c. `X` e `B`
d. `A` e `C`
e. `D`, `E` e `F`

CAPÍTULO 2: CONTROLE DE FLUXO

Os controles de fluxo são operações definidas em todas as linguagens de programação, como por exemplo Python, C, C++, Java, Fortran, Pascal, etc. Como não podia deixar de ser, tais operações também estão definidas dentro da linguagem R.

Cada linguagem de programação tem a sua própria sintaxe, isto é, sua própria regra de como essas operações devem ser usadas. Veremos nesta aula a sintaxe e mais detalhes sobre alguns controles de fluxo para a linguagem R.

If/else

Sintaxe:

```
if (condicao) {
  #comandos caso condicao seja verdadeira
} else {
  #comandos caso condicao seja falsa
}
```

ou

```
if (condicao) {
  #comandos caso condicao seja verdadeira
}
```

Descrição:

Dentro do par de parênteses seguidos do if tem que ter um objeto do tipo "logical". Os comandos dentro do primeiro par de chaves serão executados caso o objeto condicao seja TRUE. Caso contrário, os comandos de dentro do par de chaves depois do else serão executados. O comando else é opcional (segundo exemplo de sintaxe). No caso de o else não aparecer, nada será executado caso o objeto condicao seja FALSE.

Exemplos:

Vejamos um primeiro exemplo em que um texto é impresso na tela. O que será impresso depende do valor guardado na variável x. Se x receber o valor 2, veja qual texto será impresso:

```
x <- 2
if( x < 5 ){
   print(paste(x,"e menor que",5))
} else {
   print(paste(x,"e maior ou igual a",5))
}
## [1] "2 e menor que 5"
```

A função print() é responsável por imprimir na tela, e a função paste() por concatenar textos e criar um único objeto do tipo "character". Para mais detalhes digite no *prompt* do R o comando help(print) e help(paste).

Se a variável x receber um valor maior que 5 o texto impresso é outro.

```
x <- 7
if( x < 5 ){
```

```
   print(paste(x,"e menor que",5))
} else {
   print(paste(x,"e maior ou igual a",5))
}
## [1] "7 e maior ou igual a 5"
```

Considere agora um segundo exemplo. De acordo com a seguinte sequência de comandos, qual é o valor da variável y ao final desse código?

```
x <- 3
if (x > 2){
   y <- 2*x
} else {
   y <- 3*x
}
```

Respondeu certo quem disse 6. O controle de fluxo if/else será usado na maioria das vezes dentro de funções, como veremos no próximo capítulo.

for

Sintaxe:

```
for(i in valores){
  #comandos, que em geral dependem do valor de i
}
```

Descrição:

Dentro do par de parênteses valores, é um vetor de objetos, que pode ser de qualquer tipo. Os comandos de dentro do par de chaves serão executados, repetidamente, e em cada iteração o objeto i vai assumir o valor diferente, valores esses guardados no vetor valores.

O for é o primeiro exemplo de um controle de fluxo que executa uma estrutura de repetição, conhecida como laço ou pelo termo em inglês *loop*.

Exemplos:

Vejamos um primeiro exemplo, bem simples e bem comum.

```
y<-0
for(i in 1:10){
   y<-y+1
}
```

Ao final do comando qual é o valor guardado na variável y? Antes de responder vamos tentar entender cada passo.

Veja que y começa com o valor 0. Quando i = 1, y é incrementado de 1 unidade e passa a guardar o valor 0 + 1 = 1. Quando i = 2, y é incrementado de mais 1 unidade e passa a guardar o valor 1 + 1 = 2. Assim por diante até que temos i = 10, quando y recebe seu último incremente e passa a guardar o valor 10.

```
y
## [1] 10
```

Vejamos agora um segundo exemplo:

```
x<-3
```

```
for(var in 2:5){
   x<-x+var
}
```

E agora, ao final do comando, qual é o valor guardado na variável x? Vamos novamente entender cada passo que está sendo realizado.

Veja que x começa guardando o valor 3. Na primeira iteração do for, a variável var assume o valor 2 e dessa forma o valor de x é atualizado para 3 + 2 = 5. Na segunda iteração, var assume o valor 3 e assim o valor de x é atualizado para 5 + 3 = 8. Na iteração seguinte, var assume o valor 4 e x é atualizado para 8 + 4 = 12. Na última iteração, var assume o valor 5 e então o valor de x recebe a sua última atualização: x = 12 + 5 = 17. Sendo assim seu valor final igual a 17.

```
x
## [1] 17
```

O exemplo a seguir apresenta valores não numéricos para a variável de dentro do for. Não é tão comum, mas é possível de ser feito.

```
alfabeto = NULL
for(a in LETTERS){
  alfabeto = paste(alfabeto,a)
}
alfabeto
## [1] " A B C D E F G H I J K L M N O P Q R S T U
V W X Y Z"
```

Veja que uma pequena mudança no código pode mudar totalmente o resultado final:

```
alfabeto = NULL
for(a in LETTERS){
  alfabeto = paste(a,alfabeto)
}
```

Qual é a diferença entre esse último for e o anterior? Como você acha que ficou o objeto alfabeto após esse último for?

```
alfabeto
## [1] "Z Y X W V U T S R Q P O N M L K J I H G F E
D C B A "
```

Para terminar os exemplos sobre o controle de fluxo for, vamos apresentar alguns exemplos em que dois for são combinados, um dentro do outro. Suponha que temos uma matriz 5 x 5 cheia de zeros e queremos preencher cada posição dessa matriz com o número 1. O código a seguir executa essa tarefa.

```
M <- matrix(0,ncol=5,nrow=5)
for(i in 1:5){
 for(j in 1:5){
   M[i,j] <- 1
 }
}
```

Ao final das linhas de comando descritas, a matriz M, que foi iniciada como uma matriz nula, passa a ser uma matriz com todas as posições iguais a 1. Veja que para isso foi preciso usar dois comandos for, um para controlar o índice das linhas e o outro para controlar o índice das colunas.

Quando um for é usado dentro do outro, é preciso ter o cuidado de usar variáveis diferentes para cada um deles.

Para o exemplo anterior, usamos as variáveis i para o for de fora e a variável j para o for de dentro.

E como ficaria o código se quiséssemos preencher cada posição de uma matriz com o número que indica a sua linha? Por exemplo, a primeira linha da matriz com o número 1, a segunda com o número 2, e assim por diante?

```
for(i in 1:5){
 for(j in 1:5){
   M[i,j] <- i
 }
}
M
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    1    1
## [2,]    2    2    2    2    2
## [3,]    3    3    3    3    3
## [4,]    4    4    4    4    4
## [5,]    5    5    5    5    5
```

while

Sintaxe:

```
while (condicao) {
  #comandos
}
```

Descrição:

Dentro do par de parênteses seguidos do while, tem que ter um objeto do tipo "logical". Os comandos dentro do primeiro par de chaves serão executados repetidamente enquanto condicao = TRUE. É importante garantir que em algum momento teremos condicao = FALSE, se não o programa não termina de rodar, é o que chamamos de *loop* infinito.

Exemplos:

A sequência de comandos a seguir usa o controle de fluxo while para criar um vetor com os números de 1 até 100. Veja:

```
vet <- 1
while(length(vet) < 100){
   i <- length(vet)
   vet[i+1] <- i+1
}
vet
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28
29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46
47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64
65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82
83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

Veja que o tamanho do vetor cresce em cada iteração do while, dessa forma sabemos que em algum momento length(vet) < 100 será igual a FALSE; assim garantimos o fim do programa.

E como podemos usar o while para criar um vetor com os números pares entre 1 e 100? Veja uma alternativa, mas isso poderia ser feito de muitas maneiras diferentes.

```
vet = NULL
i   = 1
while(i < 50){
  vet = c(vet,2*i)
  i   = i + 1
}
vet
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28
30 32 34 36 38 40 42 44 46 48 50
## [26] 52 54 56 58 60 62 64 66 68 70 72 74 76 78
80 82 84 86 88 90 92 94 96 98
```

Você teria alguma sugestão diferente dessa para criar um vetor com os números pares menores que 100?

repeat/break

Sintaxe:

```
repeat{
 #comandos
 if(condicao)
   break
}
```

Descrição:

Dentro do par de parênteses seguidos do if, tem que ter um objeto do tipo "logical". Os comandos dentro do par de parênteses são executados repetidamente até que condicao = TRUE. É importante garantir que em algum momento teremos condicao = TRUE, se não o programa não termina de rodar, é o que chamamos de *loop* infinito.

Este controle de fluxo é muito semelhante ao while. A diferença é que o while executa os comandos enquanto uma certa condição for verdadeira, já o repeat executa os comandos até que uma certa condição se torne verdadeira.

Na prática, o comando break pode ser aplicado dentro de qualquer controle de fluxo, não apenas no repeat. Mas ele sempre tem que aparecer no repeat, pois ele é a forma de garantir o fim das iterações.

Exemplos:

Por exemplo, também podemos usar o repeat/break para criar um vetor com os 100 primeiros inteiros, veja o código a seguir.

```
y = 1
i = 1
repeat{
   i = i + 1
   y[i] = i
   if(i==100)
       break
}
y
##   [1]   1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18
```

```
##  [19]  19  20  21  22  23  24  25  26  27  28
29  30  31  32  33  34  35  36

##  [37]  37  38  39  40  41  42  43  44  45  46
47  48  49  50  51  52  53  54

##  [55]  55  56  57  58  59  60  61  62  63  64
65  66  67  68  69  70  71  72

##  [73]  73  74  75  76  77  78  79  80  81  82
83  84  85  86  87  88  89  90

##  [91]  91  92  93  94  95  96  97  98  99 100
```

Em geral podemos usar tanto o for quanto o while ou o repeat\break para implementar o mesmo programa. Às vezes um gera um código mais simples que o outro. Não tem um que seja mais fácil, depende muito da tarefa que queremos executar. Vejamos novamente o exemplo que vimos implementado com for agora com repeat\break, em que criamos um "character" com as letras do alfabeto na ordem inversa.

```
alfabeto = NULL
i = 1
repeat{
  a = LETTERS[i]
  alfabeto = paste(a,alfabeto)
  i = i + 1
  if(i > length(LETTERS))
    break
}
alfabeto
## [1] "Z Y X W V U T S R Q P O N M L K J I H G F E
D C B A "
```

Quais os valores das variáveis a e i ao final desse código? Vamos acompanhar o passo a passo feito pelo computador para responder a essa pergunta. A resposta é:

```
a
## [1] "Z"
i
## [1] 27
```

Para terminar, veja um exemplo em que é usado um controle de fluxo dentro do outro. Nele será criada uma lista com 5 posições. Em cada posição i teremos uma matriz i x i que guarda na diagonal principal o seu índice dentro da lista.

```
l = list()
i = 1
while(i <= 5){
  D = matrix(0,i,i)
  j = 1
  repeat{
    D[j,j] = i
    j = j + 1
    if(j > i)
      break
  }
  l[[i]] = D
  i = i + 1
}
l
```

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

```
## [4,]    0    0    0    5    0
## [5,]    0    0    0    0    5
```

Exercícios

1. Primeiro guarde nas variáveis a, b e c o tamanho dos lados de um triangulo qualquer. Em seguida implemente um código no R que imprime na tela uma mensagem informando se o triângulo em questão é equilátero, isósceles ou escaleno. Teste o código implementado para diferentes valores de a, b e c.

2. Para cada item a seguir, implemente um código no R para encontrar o que se pede. Não use a função seq() ou outra parecida. Dica: Comece com um vetor nulo e use o(s) controle(s) de fluxo que achar adequado para preenchê-lo.

   a. Crie um vetor com os 100 primeiros múltiplos de 3.
   b. Crie um vetor com todos os múltiplos de 3 menores que 100.
   c. Crie um vetor com os 100 primeiros números ímpares.

3. Repita cada item do Exercício 2 anterior usando um controle de fluxo diferente daquele que você usou quando resolveu esse exercício.

4. Usando os controles de fluxo vistos em sala de aula, faça o que se pede. Dica: A partir do segundo item vai ser preciso usar dois controles de fluxo, um dentro do outro.

   a. Comece com uma matriz 10 x 10 nula. Usando um dos controles de fluxos vistos em aula, implemente um código que preencha todas as posições da primeira linha dessa matriz com o número 1.
   b. Comece com uma matriz 10 x 10 nula. Usando um dos controles de fluxos vistos em aula, implemente um código que preencha cada uma de suas linhas com o

número que indica a linha em questão. Por exemplo, a primeira linha deve ser preenchida toda com o número 1, a segunda com o número 2 e assim por diante, até a décima linha que deve ser preenchida com 10.

c. Comece com uma matriz 100 x 100 nula. Usando um dos controles de fluxos vistos em aula, implemente um código que preenche cada coluna com o número que indica a coluna em questão (análogo ao item anterior).
d. Agora comece com uma matriz 100 x 100 nula. Usando um dos controles de fluxos vistos em aula, implemente um código que preenche as posições em linhas pares com o número 2 e as posições em linhas ímpares com o número 1.

5. Comece cada item a seguir com uma matriz 100 x 100 nula e não use a função seq() ou outra parecida.

a. Crie uma matriz diagonal 100 x 100 cujos elementos da diagonal principal são os números de 1 até 100 em ordem crescente.
b. Crie uma matriz diagonal 100 x 100 cujos elementos da diagonal principal são os números de 1 até 100 em ordem decrescente.

6. Usando os controles de fluxo vistos em aula, crie as listas definidas em cada item a seguir.

a. L1 é uma lista com 10 posições tal que cada posição i guarda o número i.
b. L2 é uma lista com 10 posições tal que cada posição i guarda um vetor de tamanho i com todas as posições iguais a 1.
c. L3 é uma lista com 10 posições tal que cada posição i guarda um vetor com os 10 primeiros múltiplos de i.
d. L4 é uma lista com 10 posições tal que cada posição i guarda um vetor com os i primeiros múltiplos de 2.

e. L5 é uma lista com 10 posições tal que cada posição i guarda a matriz identidade de tamanho i x i.

7. Usando as listas L1 e L3 do Exercício 6, faça o que se pede:
   a. Encontre o valor da soma de todos os números guardados em L1.
   b. Encontre o vetor tal que cada posição i guarda a soma de todos os elementos guardados na posição i da lista L3.

8. Usando a lista L4 do Exercício 6, faça o que se pede:
   a. Crie um vetor soma tal que a sua posição i guarda a soma dos elementos alocados na posição i da lista L4.
   b. Crie um vetor v de "character" tal que a sua posição i guarda o objeto soma[i] concatenado com o texto "é um múltiplo de 5" se a o valor de soma[i] for um múltiplo de 5. Caso contrário, guarde na posição i de v o objeto soma[i] concatenado com "não é um múltiplo de 5". Para concatenar textos use a função paste().
   c. A partir do vetor soma ou do vetor v criados nos itens anteriores, conte o número de vetores da lista L4 tais que a sua soma é um número múltiplos de 5. Não é para você visualizar soma ou v e contar, e sim para usar um controle de fluxo e uma variável que servirá de contador para realizar essa conta.

9. Uma progressão aritmética (p.a.) é uma sequência numérica em que cada termo, a partir do segundo, é igual à soma do termo anterior com uma constante r. O número r é chamado de razão. O primeiro termo da sequência será chamado de x_0.
   a. Faça um código em R que cria um vetor y com os 100 primeiros termos da progressão aritmética cuja termo inicial é $x_0 = 2$ e a razão é $r = 3$.
   b. Faça um código que determine a soma dos 35 primeiros termos dessa sequência. Compare o valor encon-

trado com o valor fornecido pela fórmula da soma de uma p.a. Você se lembra dessa fórmula?

c. Faça um código que conte um número de elementos em y múltiplos de 4 (lembre-se do comando %% visto no capítulo anterior, que fornece o resto da divisão).
d. Faça um código que conte um número de elementos em y múltiplos de 4 e múltiplos de 5 simultaneamente.
e. Faça um código que conte um número de elementos em y múltiplos de 4 ou múltiplos de 5.
f. Faça um código que cria um novo vetor x a partir de y, como descrito a seguir. O vetor x guarda na posição i o mesmo elemento que y guarda em i, caso esse elemento seja um número par. Se o elemento da posição i do vetor y for um número ímpar, x recebe na posição i o valor 0.

10. A famosa sequência de Fibonacci é definida da seguinte maneira: os dois primeiros elementos são iguais a [1, 1] e a partir do terceiro elemento cada termo da sequência é definido como a soma dos dois termos anteriores. Por exemplo, o terceiro termo é 2 (= 1 + 1), o quarto termo é 3 (= 1 + 2), o quinto termo é 5 (= 2 + 3) e assim por diante.

a. Faça um código em R que cria um vetor fib_12 com os 12 primeiros números da sequência de Fibonacci.
b. Faça um código em R que cria um vetor fib_m_300 com todos os números da sequência de Fibonacci menore que 300.
c. Faça um código em R que retorna o **número** de termos da sequência de Fibonacci menores que 1.000.000. Veja que nesse caso você não precisa (nem deve!) guardar os termos da sequência, apenas precisa contar o número de elementos.

CAPÍTULO 3: FUNÇÕES E O CONCEITO DE VARIÁVEL LOCAL

Nesse capítulo, veremos como criar funções dentro do R para tornar nosso código mais dinâmico e prático. Além disso será apresentado o conceito de variável local, que é de grande importância em qualquer linguagem de programação.

Funções

Para definir uma função em uma linguagem de programação, é preciso definir os seguintes itens:

- nome da função;
- argumentos (entrada);
- sequência de comandos (corpo);
- retorno (saída).

Tanto os argumentos de entrada quanto a saída de uma função podem ser nulos, ou vazios, mas em geral eles são definidos.

Depois que uma função é definida para executar a sua sequência de comandos, basta chamá-la pelo nome, passando os argumentos de entrada caso estes não sejam nulos. Veremos alguns exemplos ao longo do capítulo.

Para definir uma nova função no R, deve ser usada a seguinte sintaxe:

```
nome_da_funcao <- function(argumentos){
  # sequencia de comandos
  return(saida)
}
```

NOME	DESCRIÇÃO
Argumentos:	Define as variáveis cujos valores serão atribuídos pelo usuário quando a função for chamada.
Corpo da Função:	Contém os cálculos e tarefas executadas pela função.
Retorno:	Indica qual é o valor que a função retorna como saída.

Vejamos alguns exemplos. Primeiro vamos construir uma função que retorna o maior entre dois valores passados como argumentos. Essa função já existe pronta no R e se chama max(), o que vamos fazer é entender como ela funciona criando a nossa própria função.

```
maior <- function(a,b){
   if(a>b)
       return(a)
   else
       return(b)
}
```

Depois da função definida e compilada, podemos chamá-la sem ter que digitar todo o código novamente. Veja o que acontece quando a função é chamada no *prompt* do R:

```
maior(3,2)
## [1] 3
maior(-1,4)
## [1] 4
maior(10,10)
## [1] 10
maior("A","B")
## [1] "B"
```

Também podemos optar por guardar a saída da função em uma variável, o que pode ser bastante útil.

```
x = maior(-5,-3)
y = 2*x
x
## [1] -3
y
## [1] -6
```

E se quiséssemos encontrar o maior entre 3 números? Temos duas alternativas. A primeira é usar a função anterior composta com ela mesma. Veja como:

```
maior(1,maior(2,5))
## [1] 5
maior(10,maior(6,-1))
## [1] 10
maior(-3,maior(0,1))
## [1] 1
```

Outra alternativa é criar uma função com três argumentos própria para isso.

```
maior_de_3 <- function(a,b,c){
   if(a>b && a>c){
       return(a)
   } else {
       if(b>c)
           return(b)
       else
```

```
            return(c)
    }
}
y <- maior_de_3(2,15,6)
y
## [1] 15
```

Vamos agora fazer uma função que recebe como argumento um número natural n e retorna um vetor com os n os n primeiros múltiplos de 3. Nas atividades práticas do último capítulo, fizemos isso para n=100, agora a ideia é criar uma função que realizará essa tarefa para qualquer n escolhido pelo usuário.

```
multiplos_3 <- function(n){
  vet <- NULL
  for(i in 1:n){
    vet[i] <- 3*i
  }
  return(vet)
}
multiplos_3(10)
##  [1]  3  6  9 12 15 18 21 24 27 30
multiplos_3(15)
##  [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42
45
```

E se colocarmos como argumento um número que não seja natural, o que acontece? A função executa os comandos considerando o n que foi passado como argumento.

```
multiplos_3(1.5)
## [1] 3
multiplos_3(-1)
## [1] 3
```

Caso o programador não queira que um usuário use essa função para números não naturais, ele precisa dentro da função checar se o valor de n passado pelo usuário está correto. Caso o valor não seja correto, é possível usar a função stop para imprimir uma mensagem de erro e interromper a execução da função.

Veja um exemplo para a função multiplos_3, em que o argumento n será restrito para os naturais.

```
multiplos_3 <- function(n){
  if((n<=0)||(n%%1 != 0)){
    stop("n tem que ser um numero natural")
  }
  vet <- NULL
  for(i in 1:n){
    vet[i] <- 3*i
  }
  return(vet)
}
```

Variáveis Locais

As variáveis locais são aquelas usadas somente dentro do corpo da função. Para garantirmos que essa variável realmente não está assumindo um valor predefinido, é preciso que ela seja **iniciada** dentro do corpo da função.

Reveja o código da função multiplos_3. Veja que a variável vet é um variável que só faz sentido dentro da função, está é uma variável local. Ao digitar vet <- NULL, garantimos que essa variável dentro da função vai sempre começar como nula. Se essa linha de comando for omitida, a função pode retornar resultados não esperados pelo programador. Vejamos alguns exemplos. Primeiro a forma correta.

```
vet = c(1:10)
m   = multiplos_3(5)
m
## [1]  3  6  9 12 15
vet
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Veja no código anterior que tanto m quanto vet guardam os valores esperados tanto pelo programador quanto pelo usuário. O usuário da função multiplos_3 não tem como saber que dentro dessa função existe uma variável local chamada vet, então ele pode usar esse mesmo nome para outra variável, e isso não deve ser um problema.

Veja agora o que acontece se na função a variável local vet não for iniciada. Para isso a função multiplos_3 será implementada de maneira errada.

```
multiplos_3_errada <- function(n){
  if((n<=0)||(n%%1 != 0)){
    stop("n tem que ser um numero natural")
  }
  for(i in 1:n){
    vet[i] <- 3*i
  }
```

```
  return(vet)
}
```

Fazendo o mesmo teste que antes, agora com a função errada, veja o que acontece com variável m, que é a saída da função:

```
vet = c(1:10)
m   = multiplos_3_errada(5)
m
##  [1]  3  6  9 12 15  6  7  8  9 10
vet
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Já a variável vet criada pelo usuário permanece a mesma mesmo após a função multiplos_3_errada ser chamada.

Dentro de uma função, podemos ter dois tipos de variáveis:

- os argumentos de entrada;
- as variáveis locais.

Os argumentos de entrada **não podem** ser iniciados dentro do corpo da função, nesse caso o valor informado pelo usuário seria perdido. Já as variáveis locais, aquelas que não foram passadas como argumento, **tem que** ser iniciadas dentro da função.

Veja mais um exemplo de função antes da seção de exercícios. Suponha que queremos criar uma função que em vez de retornar os n primeiros múltiplos de 3 passe a retornar os n primeiros múltiplos de m.

```
multiplos <- function(n,m){
   if((n<=0)||(n%%1 != 0)){
       stop("n tem que ser um natural")
```

```
   }
   if((m<=0)||(m%%1 != 0)){
       stop("m tem que ser um natural")
   }
   vet <- NULL
   for(i in 1:n){
       vet[i] <- m*i
   }
   return(vet)
}
multiplos(10,7)
##  [1]  7 14 21 28 35 42 49 56 63 70
```

Mas essa não é a única maneira de implementar tal função. Podemos fazer isso de várias formas diferentes, por exemplo, usando os controles de fluxo while ou repeat.

```
multiplos_while <- function(n,m){
  if((n<=0)||(n%%1 != 0)){
    stop("n tem que ser um natural")
  }
  if((m<=0)||(m%%1 != 0)){
    stop("m tem que ser um natural")
  }
  vet = NULL
  i = 0   #i representa quantos elementos ja foram
alocados em vet
  while(i<n){
    i = i + 1
```

```
    vet = c(vet,m*i)
  }
  return(vet)
}
multiplos_while(10,7)
##  [1]  7 14 21 28 35 42 49 56 63 70
multiplos_repeat <- function(n,m){
  if((n<=0)||(n%%1 != 0)){
    stop("n tem que ser um natural")
  }
  if((m<=0)||(m%%1 != 0)){
    stop("m tem que ser um natural")
  }
  vet = NULL
  i = 0
  repeat{
    if(i >= n){
      break
    }
    i = i + 1
    vet = c(vet,m*i)
  }
  return(vet)
}
multiplos_repeat(10,7)
##  [1]  7 14 21 28 35 42 49 56 63 70
```

Exercícios

1. Para cada item a seguir implemente a função que se pede. Atenção: não use a função min já pronta no R.

   a. Implemente uma função que recebe como argumento dois números reais e retorna o menor entre eles.
   b. Implemente uma função que recebe como argumento três números reais e retorna o menor entre eles.
   c. Implemente uma função que recebe como argumento um vetor de números e retorna o menor número dentro do vetor.

2. Implemente uma função que recebe como argumento o tamanho de cada lado de um triângulo e retorna um objeto do tipo character com o texto informando se o triângulo é equilátero, isósceles ou escaleno. Antes de fazer o exercício pense:

   - Quantos argumentos a sua função vai receber?
   - Quais são os valores aceitáveis para esses argumentos?
   - Qual é o tipo de objeto que a sua função deve retornar?

3. Implemente uma função que recebe como argumento um vetor de números reais e retorna a quantidade de elementos positivos nesse vetor. Não se esqueça de iniciar todas as variáveis locais usadas em sua função. Depois que a sua função estiver pronta, invente vetores para o argumento de forma a verificar se a função está funcionando como o esperado. Por exemplo, use a função para contar o número de elementos positivos em v = c(1.0,3.2,-2.1,10.6,0.0,-1.7,-0.5).

4. Implemente uma função que recebe como argumento um vetor de "numerics" denominado v e um número real a. Essa função retorna o número de elementos em v menores que a.

5. Para cada item a seguir faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

a. Implemente uma função que recebe como argumento as variáveis n e m e retorna um vetor que guarda os n primeiros múltiplos de m.
b. Implemente uma função que recebe como argumento as variáveis m e k e retorna um vetor com os múltiplos de m menores que k.
c. Implemente uma função que recebe como argumento as variáveis m e k e retorna a quantidade de múltiplos de m menores que k.
d. Classifique cada variável que aparece dentro das funções implementadas nesse exercício como "variável local" ou "argumento de entrada" da função. Todas as variáveis locais foram iniciadas dentro do corpo da função?

6. Para cada item a seguir faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

a. Implemente uma função que recebe como entrada um número natural n e retorna uma matriz n x n tal que as posições em linhas pares recebem o número 2 e as posições em linhas ímpares o número 1.
b. Implemente uma função que recebe como entrada um número natural n e retorna uma matriz n x n tal que a coluna i dessa matriz guarda o valor i. Por exemplo, a primeira coluna deve ser preenchida com 1, a segunda com 2 e assim por diante, até a n-ésima coluna que deve ser preenchida com o número n.
c. Implemente uma função que recebe como entrada um número natural n e retorna uma matriz diagonal n x n tal que na diagonal principal aparecem os valores de 1 até n. Por exemplo, a posição (1,1) deve ser preenchido com 1, a posição (2,2) com 2 e assim por

diante. As demais posições devem ser nulas, uma vez que a matriz de saída é diagonal.

7. Para cada item a seguir faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

a. Implemente uma função que recebe como entrada um vetor de número reais v e retorna uma matriz diagonal com os elementos de v guardados na diagonal principal.
b. Implemente uma função que recebe como entrada um vetor de número reais v e retorna uma matriz quadrada cujas colunas são iguais ao vetor v.
c. Implemente uma função que recebe como entrada um vetor de número reais v e retorna uma matriz quadrada cujas linhas são iguais ao vetor v.

8. Para cada item a seguir, faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

a. Implemente uma função que recebe como argumento o valor inicial x0 e retorna os 10 primeiros termos de uma p.a. cuja razão é 3.
b. Implemente uma função que recebe como argumento o valor inicial x0, a razão r e retorna um vetor com os 10 primeiros termos dessa p.a.
c. Implemente uma função que recebe como argumentos o valor inicial x0, a razão r, um inteiro n e retorna um vetor com os n primeiros termos de uma p.a. Nomeie essa função de pa.
d. Implemente uma função que recebe como argumento o valor inicial x0, a razão r, um inteiro n e retorna a soma dos n primeiros termos de uma p.a. Nomeie essa função de soma_pa. Obs.: Você deve chamar no corpo da função soma_pa a função pa implementada no item anterior.

e. Classifique cada variável que aparece dentro das funções soma_pa e pa como "variável local" ou "argumento de entrada" da função. Todas as variáveis locais foram iniciadas dentro do corpo da função?

9. Implemente uma função que:
   a. recebe como argumento a variável n e retorna um vetor com os n primeiros termos da sequência de Fibonacci.
   b. recebe como argumento a variável k e retorna um vetor com os todos os termos da sequência de Fibonacci menores que k.
   c. recebe como argumento a variável k e retorna o número de termos da sequência de Fibonacci menores que k.

10. Suponha que a função f apresentada a seguir foi implementada no R com o intuito de gerar um vetor com os naturais de 1 até n, sendo n passado como argumento pelo usuário. Após f ter sido definida ela foi chamada a partir da sequência de comandos também apresentada a seguir.
   a. Sem usar o computador, qual é o valor de v e qual é o valor de vet ao final dessa sequência de comandos?
   b. A saída da função f, definida pelo vetor vet, é como esperado? Caso negativo, qual mudança você faria em f para que essa função passe a funcionar corretamente?

```
f <- function(n){
   for(i in 1:n){
       v[i] <- i
   }
   return(v)
}
v <- c(0,0,0,0,0)
vet <- f(3)
```

11. Uma progressão geométrica (p.g.) é uma sequência numérica em que cada termo, a partir do segundo, é igual ao produto do termo anterior por uma constante q. Esta constante q é chamada razão e o primeiro termo da sequência será chamado de x0.

a. Implemente uma função cujas entradas são x0, q e n e a saída é um vetor com os n primeiros termos de uma p.g. cujo termo inicial x0 e a razão é q.
b. Implemente uma função cujas entradas são x0, q e m e a saída é a soma dos m primeiros termos de uma p.g. com termo inicial x0 e razão q.
c. Usando as funções implementadas, encontre a soma dos 10 primeiros termos da p.g. de razão ½ e valor inicial ½.
d. Usando as funções implementadas, encontre a soma dos 30 primeiros termos da p.g. de razão ½ e valor inicial ½.
e. Você acredita que essa soma converge, ou seja, você acredita que

$$\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots + \frac{1}{2^n} = \sum_{i=1}^{n} \frac{1}{2^i} \; n \to \infty \to a \in R$$

?

f. É possível demonstrar se essa série convergência ou não pelo computador? Obs.: Use o comando options(digits=22) se quiser usar o número máximos de casas decimais que o R aceita, que é 22.

12. Para entender melhor sobre o crescimento exponencial, vamos pensar na seguinte situação: Você arranjou um emprego novo e a empresa perguntou qual das duas opções de salários você prefere. A primeira é um salário fixo de R$ 5.000,00 ao mês. A segunda opção é um pagamento diário, que começa no primeiro dia útil do mês com o pagamento de R$ 0,01 (1 centavo de real) e a cada novo dia útil você ganha o dobro do que ganhou no dia útil anterior.

a. Sem usar o computador ou uma calculadora, qual dos dois salários te parece mais interessante?
b. Agora no computador, implemente uma função que recebe como argumento de entrada o valor n e retorna o valor total pago pela segunda opção de salário em um mês com *n* dias úteis.
c. Use a função implementada no item anterior para saber qual seria o seu pagamento em um mês com 20 dias úteis. E se o mês tiver 22 dias úteis?

13. Questão Desafio.
    a. Implemente uma função que recebe como entrada um número natural positivo k e retorna um vetor com a fatoração de k em números primos. Isto é, o vetor de saída contém somente números primos tais que o produto deles é igual a k.
    b. Implemente uma função que recebe como entrada um número k e retorna TRUE se k for primo e FALSE caso contrário.
    c. Implemente uma função que recebe como entrada um número n e retorna um vetor com os n primeiros primos. Dica: Dentro desta função chame a função implementada no item (b).

CAPÍTULO 4: ALGORITMOS PARA CÁLCULOS ESTATÍSTICOS

Neste capítulo veremos algumas aplicações dos conceitos aprendidos para cálculos estatísticos. Não teremos conteúdo novo, vamos apenas aplicar os controles de fluxo e o uso de funções e aproveitar para rever parte do conteúdo de Estatística Básica.

A partir deste capítulo, vamos nos concentrar na análise e implementação de algoritmos, e por isso será evitada a apresentação de códigos prontos na linguagem R. Em vez disso serão apresentados e discutidos alguns pseudocódigos (passo a passo), como costuma aparecer na maioria dos livros. As aulas práticas serão dedicadas à implementação dos algoritmos a partir dos pseudocódigos apresentados em sala de aula.

A maioria das funções que serão implementadas nesta seção já estão prontas no R; mas não vamos usar as funções prontas, e sim criar as nossas próprias funções. O objetivo é parar para pensar como elas foram implementadas e escrever nosso próprio código.

Máximo

Definição: Seja A um conjunto de dados. Dizemos que $a \in A$ é o máximo desse conjunto se $a \geq r \; \forall \; r \in A$.

Queremos escrever um pseudocódigo que recebe como entrada um vetor de dados e retorna o seu máximo de acordo com a definição anterior. Na semana anterior foi implementada uma função que recebe dois ou três números e retorna o maior entre eles. O que está sendo pedido

agora é o caso geral: encontrar o máximo entre todos os elementos passados em um vetor.

A ideia principal do algoritmo apresentado a seguir é percorrer um vetor guardando o maior elemento encontrado até o momento. O algoritmo termina quando o vetor já foi percorrido por completo. Veja como isso pode ser feito no algoritmo a seguir.

Entrada: v = vetor com os dados.
Saída: valor máximo em v.

```
1. Defina n como o tamanho do vetor v;
2. Faça max = v[1];
3. Inicie i = 2;
4. Se v[i] > max , max = v[i];
5. Incremente i: i = i + 1;
6. Se i <= n, volta para a linha 4;
7. Retorne max.
```

Na linha 2, a variável max guarda o primeiro elemento do vetor, pois no início do algoritmo esse é o único valor observado, logo é o máximo até o momento. Na linha 4, acontece a troca do valor guardado em max, caso o novo valor observado seja maior que o máximo até o momento.

Veja que o algoritmo descreve um laço: as linhas 4-6 se repetem várias vezes. Como podemos reproduzir isso em uma linguagem de programação? A resposta é: usando um dos controles de fluxo que repete iterações vistos no Capítulo 2.

A escolha do controle de fluxo depende do algoritmo. No caso deste temos a variável i que começa assumindo o valor 2 e é incrementada até atingir o valor n, ou seja, conhecemos o valor inicial e final da variável i. Dessa forma

parece que o for é uma opção bastante razoável, em que i será a variável definida dentro dele.

Mínimo

Definição: Seja A um conjunto de dados. Dizemos que $a \in A$ é o mínimo desse conjunto se $a \leq r \ \forall \ r \in A$.}

Queremos agora escrever um pseudocódigo que recebe como entrada um vetor de dados e retorna o seu mínimo, de acordo com a definição anterior.

A ideia principal do algoritmo proposto é percorrer um vetor guardando o menor elemento encontrado até o momento. O algoritmo termina quando o vetor já foi percorrido por completo. Repare que esse algoritmo é análogo ao apresentado para o máximo, por isso a elaboração do pseudocódigo será deixada como exercício.

Média Amostral

Definição: Seja $A = \{a_1, a_2, ..., a_n\}$ um conjunto finito de dados e n o seu número de elementos. A média amostral desse conjunto é definido por:

$$média\ amostral = \frac{a_1+a_2+...+a_n}{n} = \frac{\sum_{i=1}^{n} a_i}{n}.$$

Queremos agora escrever um pseudocódigo que recebe como entrada um vetor de dados e retorna a sua média de acordo com a definição anterior. A ideia principal será percorrer o vetor v somando seus elementos uma a um. Quando v já tiver sido percorrido por completo, basta dividir a soma final pelo número total de elementos que encontraremos a média.

Entrada: v = vetor com os dados.
Saída: a média amostral dos valores de v.

```
1. Defina n como o tamanho do vetor v;
2. Inicie soma=0;
3. Inicie i=1;
4. Incremente a variável soma: soma = soma + v[i];
5. Incremente a variável i: i = i + 1;
6. Se i <= n, volta para a linha 4;
7. Faça media = soma/n;
8. Retorne media.
```

Repare que temos novamente a ocorrência de um laço dentro do algoritmo, basta notar a repetição das linhas 4-6. Então quando esse algoritmo for implementado em uma linguagem de programação será preciso escolher um controle de fluxo para realizar esse laço.

Mediana

Definição: Seja $A = \{a_1, a_2, ..., a_n\}$ um conjunto finito de dados e n o seu número de elementos. Considere a notação de estatística de ordem tal que $A = \{a_{(1)}, a_{(2)}, ..., a_{(n)}\}$ e $A = \{a_{(1)} \leq a_{(2)} \leq ... \leq a_{(n)}\}$. A mediana desse conjunto é definida informalmente como o ponto que separa o conjunto ordenado em duas partes de mesmo tamanho. Sua definição formal encontra-se a seguir:

$$mediana = \{ a_{\frac{n+1}{2}}, \ se\ n\ for\ ímpar;\ \frac{1}{2}\left(a_{\frac{n}{2}} + a_{\frac{n}{2}+1}\right), \ se\ n\ for\ par.$$

Queremos agora escrever um pseudocódigo que recebe como entrada um vetor de dados e retorna a sua mediana de acordo com a definição anterior.

Entrada: v = vetor com os dados.
Saída: a mediana dos valores de v.

```
1. Defina n como o tamanho do vetor v;
2. Defina v_o como o vetor v ordenado;
3. Se n é impar, faça mediana = v_o[(n+1)/2];
4. Se n é par, faça mediana = (v_o[n/2] +
v_o[(n/2)+1])/2;
5. Retorne mediana
```

Veja que nesse algoritmo não temos a ocorrência de laços.

Na linha 2 do pseudocódigo anterior, o vetor v é ordenado. Ainda não aprendemos como ordenar vetores, isso será visto somente no Capítulo 8. Por isso, por enquanto, usaremos o comando sort do R para ordenar vetores.

Quartis

Definição: Seja $A = \{a_1, a_2, ..., a_n\}$ um conjunto finito de dados e n o seu número de elementos. Considere a notação de estatística de ordem tal que $A = \{a_{(1)}, a_{(2)}, ..., a_{(n)}\}$ e $A = \{a_{(1)} \leq a_{(2)} \leq ... \leq a_{(n)}\}$. Os quartis desse conjunto são definidos informalmente como os três pontos que separam o conjunto ordenado em quatro partes de mesmo tamanho.

Existem alguns métodos para encontrar os quartis de um conjunto de dados. Vejamos a seguir dois desses métodos.

Método 1

1. Divida o conjunto de dados ordenados em dois subconjuntos. Um com os elementos de ordem menor que a ordem da mediana, esse chamado de A_1, e o outro com os elementos de ordem maior que a ordem da mediana, chamado de A_2.
2. O primeiro quartil é a mediana do subconjunto A_1.
3. O segundo quartil é a mediana do conjunto A.
4. O terceiro quartil é a mediana do subconjunto A_{12}.

Método 2

1. Divida o conjunto de dados ordenados em dois subconjuntos. Um com os elementos de ordem menor ou igual a ordem da mediana, A_1, e outro com os elementos de ordem maior ou igual a ordem da mediana, A_2.
2. O primeiro quartil é a mediana do subconjunto com os menores valores.
3. O segundo quartil é a mediana do conjunto original.
4. O terceiro quartil é a mediana do subconjunto com os maiores valores.

Vale destacar que os dois métodos fornecem o mesmo resultado se o número de elementos em A for par, mas podem fornecer resultados diferentes se o número de elementos no conjunto A for ímpar.

Queremos escrever um pseudocódigo que recebe como entrada um vetor de dados e retorna cada um dos seus quartis. A ideia é simplesmente seguir um dos métodos anteriores e recorrer ao cálculo da mediana, que já é conhecido. A seguir isso será feito considerando Método 1. Faça como exercício o pseudocódigo considerando o Método 2.

Entrada: v = vetor com os dados.
Saída: um vetor com os 3 quartis dos valores de v.

```
1. Defina n como o tamanho do vetor v;
2. Defina v_o como o vetor v ordenado;
3. Sendo n par, defina k = n/2 e j=k+1;
4. Sendo n impar, defina k = (n-1)/2 e j=k+2;
5. Defina v_1 como um vetor com os elementos de v_o
das posições de 1 até k;
6. Defina v_2 como um vetor com os elementos de v_o
das posições de j até n;
```

```
7. q_1 = mediana de v_1;
8. q_2 = mediana de v;
9. q_3 = mediana de v_2;
10. Retorna o vetor (q_1, q_2, q_3).
```

Esse é mais um algoritmo que não apresenta laço. Mas veja que nesse exemplo, para facilitar a implementação, usamos o resultado de outra função teoricamente já implementada, a função que retorna a mediana de um conjunto de valores guardados em um vetor.

Variância Amostral

Definição: Seja $A = \{a_1, a_2, ..., a_n\}$ um conjunto finito de dados e n o seu número de elementos. Seja m a média amostral de A. A variância amostral desse conjunto é definida por:

$$S^2 = \frac{\sum_{i=1}^{n} (a_i - m)^2}{n-1}.$$

Queremos escrever um pseudocódigo que recebe como entrada um vetor de dados e retorna a sua variância amostral. Para isso, primeiro será encontrada a média amostral. Em seguida o vetor de entrada será percorrido e assim será calculada a variância amostral.

Entrada: v = vetor com os dados.
Saída: variância amostral dos valores de v.

```
1. Defina n como o tamanho do vetor v;
2. Defina m como a média amostral do vetor v;
3. Inicie soma = 0;
4. Inicie i=1;
```

```
5. Incremente a variável soma: soma = soma + (v[i]
- m)^2;
6. Incremente i: i = i + 1;
7. Se i <= n, volta para a linha 5;
8. Faça s2 = soma/(n-1);
9. Retorne s2.
```

Covariância Amostral

Definição: Sejam $A = \{a_1, a_2, ..., a_n\}$ e $B = \{b_1, b_2, ..., b_n\}$ dois conjuntos de dados pareados e n o número de elementos de cada um deles. Seja m_A a média amostral de A e m_B a média amostral de B. A covariância amostral entre os conjuntos A e B é definida por:

$$cov_{A,B} = \frac{\sum_{i=1}^{n} (a_i - m_A)(b_i - m_B)}{n-1}.$$

Queremos agora escrever um pseudocódigo que recebe como entrada dois objetos, cada um deles com estrutura de um vetor de dados e retorna a covariância amostral entre eles.

Entrada: v = vetor com os dados, w = vetor com os dados.
Saída: covariância amostral entre os valores em v e w.

```
1. Defina n como o tamanho do vetor v;
2. Defina k como o tamanho do vetor w;
3. Se n e k forem diferentes, retorne uma mensagem
de erro e FIM.
4. m_v = média amostral de v;
5. m_w = média amostral de w;
6. Inicie soma = 0;
```

```
7. Inicie i = 1;
8. Faça soma = soma + (v[i] - m_v)*(w[i] - m_w);
9. Incremente i = i + 1;
10. Se i < = n, volte para a linha 8;
11. Faça cov = soma/(n-1);
12. Retorne cov.
```

Exercícios

Para todos os exercícios a seguir, não se esqueça de:

- Verificar sempre se as entradas passadas pelo usuário são viáveis para os cálculos das funções.
- Inventar várias entradas para as funções implementadas, a fim de verificar se elas estão funcionando corretamente.
- Sempre que possível, chamar as funções já implementadas dentro de uma nova função. Assim você simplifica bastante seu código.

1. Faça o que se pede sem usar as funções max() ou which.max() do R.

a. No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna o seu valor máximo.
b. Em seu caderno escreva um pseudocódigo para o algoritmo que recebe como entrada um vetor v e retorna a posição onde se encontra o máximo. Nesse item não é para usar o computador.
c. Agora, novamente no computador, implemente uma nova função que executa o pseudocódigo elaborado no item b. Não use o mesmo nome da função implementada no item a.

2. Faça o que se pede sem usar as funções min() e which.min() do R.

a. Primeiro escreva em seu caderno um pseudocódigo para o algoritmo que recebe como entrada um vetor v e retorna o valor mínimo guardado em v. Nesse item não é para usar o computador.
b. Agora no computador, implemente uma função que executa o pseudocódigo elaborado do item a.
c. De volta ao caderno, escreva um pseudocódigo para o algoritmo que recebe como entrada um vetor v e retorna a posição onde se encontra o mínimo. Nesse item não é para usar o computador.
d. Novamente no computador, implemente uma nova função que executa o pseudocódigo elaborado no item c. Não use o mesmo nome da função implementada no item a.

3. A amplitude de um conjunto A é definida por max (A) – min (A).

a. Primeiro escreva no caderno um pseudocódigo para o algoritmo que recebe como entrada um vetor v e retorna a sua amplitude. Considere que você já fez as funções que retornam o máximo e o mínimo de v(itens anteriores). Não é para usar o computador ainda.
b. Agora no computador, implemente o passo a passo feito do item a. Dentro da sua função chame as funções implementadas nos Exercício 1 e 2.

4. Faça o que se pede sem usar as funções mean() e sum() do R. No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna a média amostral dos valores em v. Obs.: Se quiser use a função mean() apenas para comparação.

5. Faça o que se pede sem usar a função median() do R. Para ordenar o vetor de entrada use a função sort() do R. No computador implemente o algoritmo visto em sala de

aula que recebe como entrada um vetor v e retorna a mediana de v. Obs.: Se quiser use a função median() apenas para comparação.

6. Faça o que se pede sem usar a função quantile() do R. Para ordenar o vetor de entrada use a função sort() do R.

a. No computador implemente o pseudocódigo visto em sala de aula, baseado no Método 1, que recebe como entrada um vetor v e retorna um vetor com os valores dos três quartis.
b. No caderno, escreva um pseudocódigo que calcula os três quartis a partir do Método 2.
c. No computador implemente o pseudocódigo escrito no item anterior.

7. A distância interquartílica de um conjunto A é definida por $q_3 - q_1$, em que q_1 e q_3 são, respectivamente, o primeiro e o terceiro quartil.

a. Primeiro escreva no caderno um pseudocódigo para o algoritmo que recebe como entrada um vetor v e retorna a sua Distância Interquartílica. Considere que você já sabe calcular os quartis de v. Não é para usar o computador ainda.
b. Agora no computador, implemente o passo a passo feito do item a. Dentro da sua função chame a função implementadas no Exercício 6.

8. Faça o que se pede sem usar as funções sd(), var(), mean() e sum() do R. No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna a sua variância amostral. Para simplificar use a função implementada no Exercício 4. Obs.: Se quiser use a função var() apenas para comparação.

9. O Desvio Médio de um conjunto de dados $A = \{a_1, ..., a_n\}$ é definido pela expressão a seguir , em que m é a média amostral de A.

$$dm = \frac{\sum_{i=1}^{n} |a_i - m|}{n}$$

a. Primeiro escreva no caderno um pseudocódigo para o algoritmo que recebe como entrada um vetor v e retorna o seu Desvio Médio. Não é para usar o computador ainda. Considere que você já sabe calcular a média amostral de v.
b. Agora no computador, implemente o pseudocódigo feito do item a.

10. Faça o que se pede sem usar as funções mean(), sum() ou qualquer outra pronta do R.

a. Implemente o algoritmo visto em sala de aula que recebe como entrada dois vetores v e w e retorna a covariância amostral entre eles. Para simplificar use a função implementada no Exercício 4.
b. Desafio: Vamos generalizar o item a. Implemente uma função que recebe como entrada uma matriz de dados. Considere que cada coluna dessa matriz representa um vetor de dados. A função implementada deve retornar a matriz de covariância amostral entre os vetores coluna dessa matriz. Lembre-se de que a matriz de covariância é definida pela matriz cuja posição (*i*, *j*) guarda a covariância da variável *i* com a *j*. Nesse item pode (e deve) chamar a função implementada no item a.

Para o próximo exercício considere A1 = c(6,9,7,3,9,2,2,6), A2 = c(4,4,6,1,1,2,7,4) e A3 = c(1,9,5,7,2,4,2,1,9,4).

11. Usando as funções que você implementou nos exercícios anteriores, faça as contas que se pedem. Tente fazer cada item usando apenas uma linha de comando no R.

a. Qual é a amplitude do conjunto A1?
b. Qual é a média amostral do conjunto A3?

c. Qual é a variância amostral do conjunto A2?
d. Se A4 for definido pela união dos conjuntos A1 e A2, qual é a média de A4?
e. Qual é a mediana de A4?
f. Qual é o desvio médio de A4?
g. Quais são os quartis de A3?
h. Qual é a distância interquartílica de A3?
i. Qual é a covariância entre A1 e A2? É possível fazer essa conta?
j. Qual é a covariância entre A1 e A3? É possível fazer essa conta?

CAPÍTULO 5: ALGORITMOS PARA CÁLCULOS MATRICIAIS

A motivação desse capítulo é implementar funções que realizam cálculos matriciais e com isso exercitar os laços duplos. Assim como no capítulo anterior, primeiro faremos os pseudocódigos e então estes serão implementados nos exercícios propostos ao final do capítulo.

Primeiro serão apresentadas algumas operações entre vetores. Em seguida serão vistas operações também que envolvem matrizes.

Multiplicação de vetor por escalar

Definição: Sejam $v = (v_1, v_2, ..., v_n) \in R^n$ e $\alpha \in R$. A multiplicação do vetor v pelo escalar α é o vetor $w = \alpha v$ definido por $w = (\alpha v_1, \alpha v_2, ..., \alpha v_n) \in R^n$.

Queremos escrever um pseudocódigo que recebe como entrada um vetor v e um escalar a e retorna o vetor definido por $w = \alpha v$. Para isso basta percorrer o vetor v multiplicando cada posição pelo escalar a. Veja como isso pode ser feito no pseudocódigo a seguir.

Entrada: v = vetor que guarda as coordenadas de um vetor, a = número real.
Saída: vetor definido pelo produto a com v.

```
1. Defina n como o tamanho do vetor v;
2. Inicie o vetor w como nulo;
3. Inicie i = 1;
4. Faça w[i] = a*v[i];
```

```
5. Incremente i: i = i + 1;
6. Se i < = n, volte para a linha 4;
7. Retorne w.
```

Soma de vetores

Definição: Sejam $v = (v_1, v_2, ..., v_n) \in R^n$ e $u = (u_1, u_2, ..., u_n) \in R^n$. A soma dos vetores v e u é o vetor $w = v + u$ definido por $w = (v_1 + u_1, v_2 + u_2, ..., v_n + u_n)$.

Queremos escrever um pseudocódigo que recebe como entrada dois vetores v e u e retorna o vetor definido pela soma deles. Veja que, para essa operação ser realizada, é preciso que ambos os vetores tenham mesma dimensão. Para encontrar a soma dos dois vetores de mesma dimensão, basta somar cada posição uma a uma. Veja o pseudocódigo a seguir:

Entrada: v = vetor, que guarda as coordenadas de um vetor; u = vetor, que guarda as coordenadas de outro vetor.
Saída: vetor com os elementos do vetor definido pela soma de v com u.

```
1. Defina n como o tamanho do vetor v;
2. Defina k como o tamanho do vetor u;
3. Se n e k forem diferentes, retorne uma mensagem
de erro e FIM.
4. Inicie o vetor w como nulo;
5. Inicie i = 1;
6. Faça w[i] = v[i] + u[i];
7. Incremente i: i = i + 1;
8. Se i < = n, volte para a linha 6;
9. Retorne w.
```

Subtração de vetores

Definição: Sejam $v = (v_1, v_2, ..., v_n) \in R^n$ e $u = (u_1, u_2, ..., u_n) \in R^n$. A substração do vetor v e u é o vetor $w = v - u$ definido por $w = (v_1 - u_1, v_2 - u_2, ..., v_n - u_n)$.

O pseudocódigo que recebe como entrada dois vetores e retorna o vetor definido pela subtração entre eles pode ser feito de duas maneiras. A primeira alternativa é fazer um pseudocódigo análogo ao definido para a soma entre vetores. A segunda alternativa é combinar a multiplicação de um vetor por um escalar com a soma de dois vetores: primeiro multiplica-se o vetor u pelo escalar -1 e em seguida soma o resultado com o vetor v.

Produto interno

Definição: Sejam $v = (v_1, v_2, ..., v_n) \in R^n$ e $u = (u_1, u_2, ..., u_n) \in R^n$. O produto interno entre v e u é o número real definido por:

$$< v, u >= v_1 u_1 + v_2 u_2 + \ ... \ + v_n u_n = \sum_{i=1}^{n} v_i u_i.$$

Queremos escrever o pseudocódigo que recebe como entrada dois vetores v e u e retorna o produto interno entre eles. Veja que essa operação só é possível se ambos os vetores tiverem mesma dimensão. A ideia é percorrer as posições dos vetores, multiplicando posição a posição e guardando a soma em uma variável local. Veja o pseudocódigo a seguir:

Entrada: v = vetor, que guarda as coordenadas de um vetor; u = vetor, que guarda as coordenadas de outro vetor.
Saída: o valor numérico do produto interno entre u e v.

```
1. Defina n como o tamanho do vetor v;
2. Defina k como o tamanho do vetor u;
```

```
3. Se n e k forem diferentes, retorne uma mensagem
de erro e FIM.
4. Inicie p = 0;
5. Inicie i = 1;
6. Incremente p: p = p + v[i]*u[i];
7. Incremente i: i = i + 1;
8. Se i < = n, volte para a linha 6;
9. Retorne p.
```

Multiplicação de matriz por escalar

Definição: Sejam A uma matriz de dimensão n x m e $\alpha \in R$ um escalar. A multiplicação de A pelo escalar α é a matriz $M = \alpha A$ de dimensão n x m em que cada posição (i, j) é definida por $M_{i,j} = \alpha A_{i,j}$.

Queremos escrever um pseudocódigo que recebe como entrada uma matriz A e um escalar a e retorna a matriz definida por $M = aA$. Para isso basta percorrer a matriz A multiplicando cada posição pelo escalar a. Mas lembre-se de que para percorrer uma matriz é preciso usar dois laços, um dentro do outro. Veja como isso pode ser feito no pseudocódigo a seguir.

Entrada: A = matriz de números reais; a = número real.
Saída: uma matriz definida pelo produto de a com A.

```
1. Defina n número de linhas da matriz A;
2. Defina m número de colunas da matriz A;
3. Inicie uma matriz M de dimensão n x m.
4. Inicie i = 1;
5. Inicie j = 1;
6. Faça M[i,j] = a*A[i,j];
```

```
7. Incremente j: j = j + 1;
8. Se j < = m volte para a linha 6;
9. Incremente i: i = i + 1;
10. Se i < = n volte para a linha 5;
11. Retorne M.
```

Reparou que temos dois laços? O primeiro está entre as linhas 6-8 e o outro entre as linhas 5-10. Como isso pode ser implementado em uma linguagem de programação, em particular na linguagem R?

Soma de matrizes

Definição: Sejam A e B duas matrizes de dimensão n x m. A soma das matrizes A e B é a matriz $M = A + B$ de dimensão n x m em que cada posição (i, j) é definida por $M_{i,j} = A_{i,j} + B_{i,j}$.

Queremos escrever o pseudocódigo que recebe como entrada duas matrizes A e B e retorna a matriz definida pela soma delas. Novamente como teremos que percorrer as linhas e colunas das matrizes, será preciso usar dois laços, um dentro de outro. Veja no pseudocódigo a seguir como isso pode ser feito.

Entrada: A = matriz de números reais; B = matriz de números reais.
Saída: uma matriz definida pelo soma de A com B.

```
1. Defina n número de linhas da matriz A;
2. Defina m número de colunas da matriz A;
3. Defina l número de linhas da matriz B;
4. Defina c número de colunas da matriz B;
5. Se n é diferente de l, retorne uma mensagem de
erro e FIM.
```

```
6. Se m é diferente de c, retorne uma mensagem de
erro e FIM.
7. Inicie uma matriz M de dimensão n x m.
8. Inicie i = 1;
9. Inicie j = 1;
10. Faça M[i,j] = A[i,j] + B[i,j];
11. Incremente j: j = j + 1;
12. Se j < = m volte para a linha 10;
13. Incremente i: i = i + 1;
14. Se i < = n volte para a linha 9;
15. Retorne M.
```

Subtração de Matrizes

Definição: Sejam A e B duas matrizes de dimensão n x m. A subtração da matriz A pela matriz B é a matriz $M = A - B$ de dimensão n x m em que cada posição (i, j) é definida por $M_{i,j} = A_{i,j} + B_{i,j}$.

Assim como a subtração de vetores, o pseudocódigo que recebe como entrada duas matrizes e retorna a matriz definida pela subtração entre elas pode ser feito de duas maneiras. A primeira alternativa é fazer um pseudocódigo análogo ao definido para a soma entre matrizes. A segunda alternativa é combinar a multiplicação de uma matriz por um escalar com a soma entre duas matrizes: primeiro multiplica-se a matriz B pelo escalar -1 e em seguida soma o resultado com a matriz A.

Transposição de Matrizes

Definição: Seja A uma matriz n x m. A transposta da matriz A é a matriz A^T de dimensão m x n em que cada posição (i, j) é definida por $A^T_{i,j} = A_{i,j}$.

Queremos escrever um pseudocódigo que recebe como entrada uma matriz A e retorna a sua transposta. Veja que só precisamos percorrer as linhas e colunas da matriz e guardar na posição (i, j) o elemento guardando na posição (j, i) da matriz de entrada. O pseudo-código a seguir realiza essa tarefa.

Entrada: A = matriz de números reais.
Saída: uma matriz definida pela transposta de A.

```
1. Defina n número de linhas da matriz A;
2. Defina m número de colunas da matriz A;
3. Inicie uma matriz M de dimensão m x n;
4. Inicie i = 1;
5. Inicie j = 1;
6. Faça M[i,j] = A[j,i];
7. Incremente j: j = j + 1;
8. Se j < = n volte para a linha 6;
9. Incremente i: i = i + 1;
10. Se i < = m volte para a linha 5;
11. Retorne M.
```

Multiplicação entre matriz e vetor

Definição: Seja A uma matriz n x m e $v \in R^m$ um vetor. Considere $a_i \in R^m$ o vetor formado pela i-ésima linha da

matriz A. O produto entre a matriz A e o vetor v é o vetor $w = Av \in R^n$ tal que cada posição é definida por $w_i = < a_i, v >$.

A partir da definição anterior, vamos escrever um pseudocódigo para uma função que recebe como entrada uma matriz A e um vetor v e retorna o vetor definido pelo produto entre eles. Vamos considerar que já sabemos calcular o produto interno entre dois vetores. Então basta percorrer as colunas da matriz e calcular o produto interno entre os vetores colunas e o vetor v. Veja que as dimensões de A e v devem ser testadas para verificar se a multiplicação realmente pode ser realizada: a conta só é possível quando o número de colunas da matriz é igual ao número de elementos no vetor. O pseudocódigo a seguir mostra como isso pode ser feito:

Entrada: A = matriz de números reais; v = vetor com os valores do vetor.
Saída: um vetor com os elementos do produto de A com v, isto é, Av.

```
1. Defina n número de linhas da matriz A;
2. Defina m número de colunas da matriz A;
3. Defina k número de elementos no vetor v;
4. Se m é diferente de k, retorne uma mensagem de
erro e FIM.
5. Inicie um vetor nulo w;
6. Inicie i = 1;
7. w[i] = produto interno entre a linha i da matriz
A e o vetor v;
8. Incremente i: i = i + 1;
9. Se i < = n volte para a linha 7;
10. Retorne w.
```

Dica: Na linguagem R se o objeto A é uma matriz, então A[i,] é um vetor do tipo "numeric" definido pela i-ésima linha da matriz A.

Multiplicação de matrizes

Definição: Seja A uma matriz n x m e B uma matriz m x k. Considere $a_i \in R^m$ o vetor formado pela i-ésima linha da matriz A e $b_j \in R^m$ o vetor formado pela j-ésima coluna da matriz B. O produto entre a matriz A e a matriz B é a matriz $M = AB$ de dimensão n x k em que cada posição é definida por $M_{i,j} = < a_i, b_j >$.

A partir da definição anterior, vamos escrever um pseudocódigo para uma função que recebe como entrada duas matrizes A e B e retorna outra matriz definida pelo produto entre elas. Vamos considerar que já sabemos calcular o produto interno entre dois vetores. Veja que as dimensões de A e B devem ser testadas para verificar se a multiplicação realmente pode ser realizada: só podemos multiplicar A por B se o número de colunas de A for igual ao número de linhas de B.

Entrada: A = matriz de números reais; B = matriz de números reais.
Saída: uma matriz definida pelo produto de A com B, isto é, AB.

```
1. Defina n número de linhas da matriz A;
2. Defina m número de colunas da matriz A;
3. Defina l número de linhas da matriz B;
4. Defina c número de colunas da matriz B;
5. Se m for diferente de l, retorne uma mensagem de erro e FIM.
6. Inicie uma matriz M de dimensão n x c.
```

```
7. Inicie i = 1;
8. Inicie j = 1;
9. Faça M[i,j] = produto interno entre a linha i da
matriz A e a coluna j de B;
10. Incremente j: j = j + 1;
11. Se j < = c volte para a linha 9;
12. Incremente i: i = i + 1;
13. Se i < = n volte para a linha 8;
14. Retorne M.
```

Exercícios

Para todos os exercícios a seguir, não se esqueça de:

- verificar sempre se as entradas passadas pelo usuário são viáveis para os cálculos das funções;
- inventar várias entradas para as funções implementadas, a fim de verificar se elas estão funcionando corretamente; e
- sempre que possível, chamar as funções já implementadas dentro de uma nova função. Assim você simplifica bastante seu código.

1. Implemente uma função que recebe como entrada um vetor (vetor de números) v e um escalar a e retorna o vetor (vetor de números) definido pelo produto av.

2. Implemente uma função que recebe como entrada dois vetores de números reais, representados por objetos do tipo "numeric" com a estrutura de um vetor, v e u, e retorna outro vetor definido pela soma entre eles.

3. No caderno, escreva um pseudocódigo para o algoritmo que recebe como entrada dois vetores de números reais, representados por objetos do tipo "numeric" com a estrutura de um vetor, v e u, e retorna outro vetor, isto é, um objeto do tipo "numeric", definido pela subtração do primeiro pelo segundo. Em seguida, no computador, implemente o pseudocódigo elaborado.

4. Implemente uma função que recebe como entrada dois vetores de números reais, representados por objetos do tipo "numeric" com a estrutura de um vetor, v e u, e retorna o produto interno entre eles.

5. Implemente uma função que recebe como entrada dois vetores de números reais, representados por objetos do tipo "numeric" com a estrutura de um vetor, e retorna TRUE caso eles forem ortogonais e FALSE caso contrário. Dica: Dá para saber se dois vetores são ortogonais a partir do produto interno entre eles.

6. Implemente uma função que recebe como entrada uma matriz A e um escalar r e retorna a matriz definida pelo produto de r com A.

7. Implemente uma função que recebe como entrada duas matrizes A e B e retorna outra matriz definida pela soma entre elas.

8. No caderno, escreva um pseudocódigo para o algoritmo que recebe como entrada duas matrizes A e B e retorna outra matriz definida pela subtração da primeira pela segunda. Agora no computador, implemente o pseudocódigo elaborado.

9. Implemente uma função que recebe como entrada uma matriz A e retorna a sua transposta.

10. No caderno, escreva um pseudocódigo para o algoritmo que recebe como entrada uma matriz A e retorna TRUE se essa matriz for simétrica e FALSE caso contrário. Lembre-se: uma matriz simétrica é uma matriz quadrada tal que $A_{i,j} = A_{j,i}$. Em seguida implemente o pseudocódigo elaborado.

11. Implemente uma função que recebe como entrada uma matriz A e um vetor (vetor de números) v e retorna o vetor (vetor de números) definido pelo produto Av.

12. Implemente uma função que recebe como entrada duas matrizes A e B e retorna a matriz definida pelo produto entre A e B. Atenção: O produto de matrizes não é uma operação comutativa, ou seja, $AB \neq BA$. Nesse exercício vamos definir que a matriz passada como primeiro argumento de entrada A será a matriz que fica do lado esquerdo do produto.

Para o próximo exercício considere α = 4, β = -3, v_1 = (2, -3, -1, 5, 0, -2), v_2 = (3, 4, -1, 0, 1, 1), v_3 = (1, 2, 3, 4, 5), v_4 =(0, 1, 1),

$$M_1 = \begin{pmatrix} 1 & 3 & 2 \\ -1 & 0 & 1 \end{pmatrix}, M_2 = \begin{pmatrix} 0 & -5 & 3 \\ -1 & 1 & -1 \\ 1 & 4 & 0 \end{pmatrix}, M_3 = \begin{pmatrix} 3 & 1 \\ -2 & 10 \\ 3 & -1 \end{pmatrix}$$

$$M_4 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, M_5 = \begin{pmatrix} 3 & 1 & 0 & 1 \\ 1 & 1 & 3 & 2 \\ 0 & 3 & -5 & 0 \\ 1 & 2 & 0 & 0 \end{pmatrix}$$

13. Usando as funções que você implementou nos exercícios anteriores faça as contas que se pede. Tente fazer cada item usando apenas uma linha de comando no R.

a. αv_3
b. $v_1 + v_2$
c. $v_3 - v_1$
d. $< v_1, v_2 >$

e. $< \alpha v_1, v_2 - v_1$
f. $< v_1 + v_2, v_1 - v_2 >$
g. Os vetores v_1 e v_2 anteriores são perpendiculares (ortogonais)?
h. βM_1
i. M_1^T
j. Verifique se as matrizes M_1, M_4 e M_5 são simétricas.
k. $M_1 v_4$
l. $M_2 v_4 + v_4$
m. $M_1 M_2$
n. $M_2 M_1$
o. $M_3^T M_2$
p. $(M_1 M_3) + M_4$
q. $M_1 M_2 M_3$
r. $M_1 M_2 M_3 - M_4$

PARTE II: RECURSÃO

Na ciência da computação, um algoritmo é dito recursivo quando ele chama a si próprio com entradas mais simples. De forma semelhante, dizemos que uma função é implementada de forma recursiva quando ela chama a si própria com argumentos mais simples.

Veremos que para todo algoritmo recursivo existe outro, que executa a mesma tarefa, de forma não recursiva. A vantagem em usar os algoritmos recursivos é a maior simplicidade e clareza no código (confira!). A desvantagem é que em geral tais algoritmos consomem muita memória, devido ao uso intensivo de pilha.

Nessa segunda parte do livro serão vistos alguns algoritmos recursivos. O objetivo é revisitar alguns algoritmos já estudados na Parte I, agora de forma recursiva, e aprender outros novos.

CAPÍTULO 6: ALGORITMOS RECURSIVOS SIMPLES

Neste capítulo, vamos explorar alguns exemplos mais simples de algoritmos recursivos. Todo os exemplos apresentados têm um único argumento de entrada e retornam um único objeto na saída.

Fatorial

O exemplo mais clássico em recursão é o cálculo do fatorial de *n*. Podemos fazer isso de forma recursiva ou não. Vejamos primeiro o algoritmo sem recursão.

Entrada: n
Saída: !n = n x (n-1) x ... x 2 x 1.

```
1. Caso n não seja inteiro não negativo, retorne erro.
2. Inicie fat = 1;
3. Se n=0, retorne fat.
4. Inicie i = 1;
5. Faça fat = fat*i;
6. Incremente i = i + 1;
7. Se i <= n, volte para a linha 5;
8. Retorne fat.
```

Veja como fica o código dessa função no R:

```
fatorial = function(n){
  if(!is.numeric(n))
    stop("o argumento de entrada tem que ser
numérico.")
  if(n%%1 != 0)
    stop("o argumento de entrada tem que ser um
número natural.")
  if(n<0)
    stop("o argumento de entrada tem que ser um
número não negativo")
  fat = 1
  if(n == 0)
    return(fat)
  for(i in 1:n){
    fat = fat*i
  }
  return(fat)
}
```

Agora o algoritmo recursivo. Para que o pseudocódigo fique claro, vamos também definir o nome da função nele.

Entrada: n
Saída: !n = n x (n-1) x ... x 2 x 1.
Nome: fatorial_rec

```
1. Caso n não seja inteiro não negativo, retorne
erro.
2. Se n=0, retorne 1.
3. Retorne n*fatorial_rec(n-1)
```

Veja que a chamada recursiva aparece na linha 3. A função fat-rec, que está sendo definida para uma entrada n, é chamada para a entrada n-1. Ou seja, o algoritmo chama a si próprio com o argumento de entrada simplificado. Dessa forma garantimos que em algum momento a função fat-rec será chamada para n=0, argumento para o qual temos resposta imediata, 1.

O código para a função recursiva fica assim:

```
fatorial_rec = function(n){
  if(!is.numeric(n))
    stop("o argumento de entrada tem que ser
numérico.")
  if(n<0)
    stop("o argumento de entrada tem que ser um
número não negativo")
  if(n == 0)
    return(1)
  return(n*fatorial_rec(n-1))
}
fatorial(5)
## [1] 120
fatorial_rec(5)
## [1] 120
```

Exemplos com Vetores

Também podemos usar a recursão para trabalhar com vetores. Veja primeiro o pseudocódigo sem usar recursão, como já fizemos anteriormente.

Entrada: v
Saída: O maior elemento de v
Nome: maior

```
1. Defina n como o tamanho do vetor v;
2. Defina max = v[1]
3. Inicie i = 2
4. Se v[i] > max, faça max = v[i];
5. Incremente i = i + 1;
6. Se i <=n, volte para a linha 4;
7. Retorne max.
```

Agora veja outro algoritmo, esse recursivo, que executa a mesma coisa.

Entrada: v
Saída: O maior elemento de v
Nome: maior_rec

```
1. Defina n como o tamanho do vetor v;
2. Se n = 1, retorne v[1];
3. Defina w = v[2:n];
4. max_w = maior_rec(w);
5. Se v[1] > max_w, retorne v[1].
6. Retorne max_w.
```

Veja que a chamada recursiva aparece na linha 4. Nesse exemplo, o argumento foi simplificado; uma vez que vetor passado como argumento de entrada tem sempre uma dimensão a menos. Assim garantimos que em algum momento a função será chamada para um vetor de tamanho 1, entrada para a qual temos uma resposta imediata.

Sequências Definidas a partir de Equações de Diferenças

Uma sequência de números reais $\{x_n\}_{n=1}^{\infty}$ é definida a partir de uma equação de diferenças quando o seu n-ésimo termo é escrito em função de termos anteriores. Ou seja, quando existe f tal que $x_n = f(n, x_{n-1}, x_{n-2}, \ldots)$.

Um exemplo de equação de diferenças é a Sequência de Fibonacci, que já foi vista em capítulos anteriores. Esse exemplo será revisto em breve. Antes disso vejamos alguns outros exemplos.

Encontrar a solução de uma equação de diferenças consiste em determinar o n-ésimo termo da sequência de forma direta, sem depender dos termos anteriores, dependendo apenas de n. Para isso existem muitas técnicas, mas aprender essas técnicas não é o foco do nosso curso.

Por exemplo, seja $\{x_n\}_{n=1}^{\infty}$ uma sequência definida por: $x_0 = 1$ e $x_n = 2x_{n-1} + 1$.

Com essas informações, podemos calcular todos os termos de forma iterativa: $x_0 = 1$, $x_1 = 2 \times 1 + 1 = 3$, $x_2 = 2 \times 3 + 1 = 7$, $x_3 = 2 \times 7 + 1 = 15$, ...

A solução dessa equação de diferenças é $x_n = 2^{n+1} - 1$ (verifique!). Mas neste livro não vamos aprender como chegar na solução teórica, nosso trabalho será montar a equação e resolver o problema de forma iterativa a partir de uma função implementada no R. Faremos isso usando e não usando recursão. Vejamos a seguir alguns problemas que utilizam equações de diferenças em sua modelagem.

Fibonacci

A sequência de Fibonacci é um caso particular de equações de diferenças de segunda ordem, isto é, uma equação em que x_n depende não só de x_{n-1} como também de x_{n-2}. Lembrando, uma sequência de Fibonacci é definida por: $F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Vamos escrever um pseudocódigo que recebe como entrada *n* e retorna o *n*-ésimo termo da sequência de Fibonacci (F_n). Primeiro sem recursão.

Entrada: n
Saída: F_n = valor do n-ésimo termo da sequência de Fibonacci.
Nome: Fibonacci

```
1. Caso n não seja inteiro positivo, retorne erro.
2. Se  n=1 ou n=2, retorne 1;
3. Defina F1 = 1 e F2 = 1;
4. Inicie i = 3;
5. Faça Fib = F1 + F2;
6. Faça F1 = Fib e F2 = F1;
7. Incremente i = i + 1;
8. Se i <= n, volte para a linha 5;
9. Retorne Fib.
```

Agora o pseudocódigo do algoritmo recursivo.

Entrada: n
Saída: F_n = valor do n-ésimo termo da Sequência de Fibonacci.
Nome: Fibonacci_rec

```
1. Caso n não seja inteiro positivo, retorne erro.
2. Se n=1 ou n=2, retorne 1;
3. Retorne Fibonacci_rec(n-1) + Fibonacci_rec(n-2);
```

Veja que, neste .exemplo, por se tratar de uma equação de diferenças de segunda ordem, a chamada recursiva é feita duas vezes: Fibonacci(n-1) e Fibonacci(n-2). Em ambas o argumento foi simplificado. Como temos dois casos

básicos, n=1 e n=2, garantimos que para qualquer natural n sempre teremos solução.

```
Fibonacci_rec = function(n){
  if(n<=0 || (n%%1 != 0))
    stop("O argumento de entrada tem que ser um
inteiro positivo.")
  if(n==1 || n==2){
    return(1)
  }
  return(Fibonacci_rec(n-1) + Fibonacci_rec(n-2))
}
Fibonacci_rec(10)
## [1] 55
```

Séries

Seja $\{x_i\}^{\infty}_{i=0}$ uma sequência de número reais. Defina outra sequência $\{S_n\}^{\infty}_{n=0}$ como a soma dos n primeiros termos dessa sequência

$$\{x_i\}_{i=0}^{\infty} : S_n = \sum_{i=0}^{n} x_i$$

A sequência $\{S_n\}^{\infty}_{n=0}$ é chamada de série.

Nesta seção, vamos usar a recursão para encontrar os termos de uma série

$$S_n = \sum_{i=0}^{n} x_i$$

dada a sequência $\{x_i\}$. Ou seja, para uma dada sequência $\{x_i\}$ e um natural n estamos interessados em encontrar a soma dos n primeiros termos dessa sequência.

Veja que o n-ésimo termo da série pode ser escrito como uma equação de diferenças: $S_n = S_{n-1} + x_n$ e esta será a ideia do algoritmo a seguir.

Como primeiro exemplo considere $x_i = i$ e

$$S_n = \sum_{i=0}^{n} x_i = \sum_{i=0}^{n} i$$

Veja que, para esse exemplo $S_n = 0 + 1 + 2 + ... + n$, vamos escrever um algoritmo que recebe como entrada n e retorna o n-ésimo termo dessa série, isto é, S_n.

Entrada: n
Saída: $S_n = 0 + 1 + 2 + ... + n$
Nome: Serie1

```
1. Caso n não seja inteiro não negativo, retorne
erro.
2. Se n=0, retorne 0.
3. Retorne Serie1(n-1) + n
```

Veja que a chamada recursiva usa como argumento $n - 1$. Dessa forma garantimos a simplificação do argumento e que em algum momento chegaremos ao caso mais simples e o algoritmo termina.

Vejamos agora outro exemplo. Considere

$$x_i = \frac{1}{i!} \text{ e defina } S_n = \sum_{i=0}^{n} x_i = \sum_{i=0}^{n} \frac{1}{i!}.$$

Podemos criar um algoritmo recursivo e usar o computador para encontrar qualquer termo n dessa série, como mostra o pseudocódigo a seguir.

Entrada: n
Saída:

$$S_n = \sum_{i=0}^{n} \frac{1}{i!}$$

Nome: Serie2

```
1. Caso n não seja inteiro não negativo, retorne
erro.
2. Se n=0, retorne 1.
3. Retorne Serie2(n-1) + 1/n!
```

Veja que no código anterior consideram que já sabemos calcular $n!$. Mas é verdade, já fizemos isso no capítulo anterior de forma recursiva. Então quando formos implementar esse último código no computador vamos chamar a função já feita que calcula $n!$.

Já foi (ou será) visto em algumas disciplinas que a sequência S_n definida converge para o número irracional e = 2.718282..., ou seja, $S_n n \to \infty \to e$. Então para n razoavelmente grande esperamos que S_n esteja próximo de e = 2.718282..., como indica o limite. Com essa ideia, podemos usar o algoritmo anterior para encontrar uma aproximação para o número irracional e. Basta chamar a função com valores grandes de n.

Exercícios

1. Implemente de forma recursiva uma função que recebe como entrada um número natural n e retorna $n!$. Não esqueça de verificar se o argumento passado como entrada é realmente um número natural.

2. Implemente de forma recursiva uma função que recebe como entrada um vetor v e retorna o valor máximo desse vetor.

3. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um vetor v e retorna a soma dos elementos desse vetor. Em seguida, no computador, implemente o pseudocódigo elaborado.

4. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um vetor v e retorna a posição onde se encontra o valor máximo desse vetor. Dica: Em vez de definir $w = (v_2, v_3, ..., v_n)$ defina $w = (v_1, v_2, ..., v_{n-1})$ para a chamada recursiva. Mas cuidado que isso muda um pouco a forma de pensar. Em seguida, no computador, implemente o pseudocódigo elaborado.

5. Vamos trabalhar com a sequência de Fibonacci.
   a. Implemente uma função recursiva que recebe como entrada um número natural n e retorna o n-ésimo termo da sequência de Fibonacci.
   b. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um número natural n e retorna a soma dos n primeiros termos da sequência de Fibonacci. Considere que você sabe encontrar a o termo n da sequência de Fibonacci, feito no item anterior.
   c. Em seguida, no computador, implemente o pseudocódigo elaborado. Use a função que retorna o n-ésimo termos da Sequência de Fibonacci implementada na semana passada.

6. Considere a sequência definida a seguir a partir de uma equação de diferenças de segunda ordem. $y_n = 2y_{n-1} + y_{n-2} + n$, sendo $y_1 = 0$ e $y_2 = 0$

a. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um número natural n e retorna o valor de y_n.
b. Agora no computador, implemente o pseudocódigo elaborado.

7. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um vetor de números e retorna o número de elementos nulos contidos nesse vetor. Em seguida, no computador, implemente o pseudocódigo elaborado.

8. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um vetor qualquer e retorna outro vetor definido pela ordem inversa do vetor de entrada. Em seguida, no computador, implemente o pseudocódigo elaborado.

9. Implemente de forma recursiva uma função que recebe como entrada um número natural n e retorna a soma de todos os naturais até n, isto é, retorna

$$S_n = \sum_{i=0}^{n} i = 0 + 1 + 2 + \ldots + n$$

10. Implemente uma função que recebe como entrada um número natural n e retorna o n-ésimo termo da série

$$S_n = \sum_{i=0}^{n} \frac{1}{i!}$$

Teste a função implementada para diferentes valores de n e veja se quando n cresce S_n se aproxima de e = 2.718282...

11. Considere a seguinte série:

$$S_n = \sum_{i=0}^{n} 4\frac{(-1)^i}{2i+1}$$

Essa séria foi desenvolvida por Leibniz em 1682 e é conhecida para calcular aproximações para o número irracional π, uma vez que S_n $n \to \infty \to \pi.\pi$

a. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um número natural n e retorna S_n de acordo com a fórmula anterior.
b. Agora no computador, implemente o pseudocódigo elaborado.
c. Teste a função implementada para diferentes valores de n e veja se quando n cresce S_n se aproxima de π = 3.141593...

12. Seja $\{x_i\}$ a sequência definida por

$$x_i = \frac{1}{3^i} \text{ . Defina } S_n = \sum_{i=0}^{n} x_i.$$

a. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um número natural n e retorna S_n.
b. Agora no computador, implemente o pseudocódigo elaborado.
c. Para $a \leq b$ e $a, b \in N$ defina $Soma(a, b) = x_a + x_{a+1} + ... + x_b$. Implemente no computador uma função que recebe como entrada a e b e retorna $Soma(a, b)$ de forma recursiva, sem chamar a função implementada no item b. Dica: O caso base acontece quando $a = b$, nesse caso qual deve ser a saída de $Soma(a, b)$? E se $a < b$, como será a chamada recursiva?

CAPÍTULO 7: ALGORITMOS RECURSIVOS (CONTINUAÇÃO)

Vamos continuar o estudo de algoritmos recursivos. Neste capítulo veremos mais exemplos. Diferentemente do capítulo anterior, neste serão apenas apresentados os algoritmos recursivos.

Exemplos de Matemática Financeira

Em Matemática Financeira, muitos problemas que envolvem juros, como investimentos ou dívidas, podem ser expressos em termos de sequências de diferenças. Veja dois exemplos a seguir.

Suponha que você investiu R$ 1.000,00 em um fundo de investimento que paga 0,7% de rendimento ao mês. Considere x_n como o total acumulado após n meses de investimento. Primeiro veja que o total acumulado após n meses depende diretamente do total acumulado após $n - 1$ meses:

$x_0 = 1.000,00$

$x_n = x_{n-1} + 0,007\, x_{n-1} = 1,007\, x_{n-1}$.

Essa é uma equação de diferenças homogênea de primeira ordem com coeficiente constante, ou seja, ela é do tipo: $x_n = cx_{n-1}$ com $c \in R$. Toda equação desse tipo tem uma solução teórica simples: $x_n = c^n x_0$ (verifique!). Mas não é isso que estamos buscando. Nesse capítulo queremos treinar a implementação de algoritmos usando recursão. Vamos então escrever um código que recebe como entrada n e retorna x_n, em que cada x_i é calculado de forma iterativa, como se não soubéssemos a solução teórica da equação de diferenças. Veja o pseudocódigo recursivo.

Entrada: n
Saída: Xn = total acumulado após n meses de investimento, considerando juros de 0,7% e investimento inicial de R$ 1.000,00.
Nome: Investimento_rec

```
1. Caso n não seja inteiro não negativo, retorne
erro.
2. Se n=0, retorne 1000;
3. Retorne 1.007*Investimento_rec(n-1)
```

Para os exemplos anteriores, o valor investido foi um dado fixo. E se quiséssemos criar uma função que recebesse como entrada, além do número de meses de investimentos *n*, também o valor inicial investido *I* e o juros *j*, como ficaria o pseudocódigo recursivo? O pseudocódigo não muda muito, mas ele passa a ter mais de um argumento de entrada o que requer um pouco mais de cuidado na chamada recursiva.

Entrada: n, I, j
Saída: Xn = total acumulado após n meses de investimento, considerando juros de j% ao mês e um investimento inicial de I reais.
Nome: Investimento_rec

```
1. Caso n não seja inteiro não negativo, retorne
erro.
2. Se n=0, retorne I;
3. Retorne (1 + j/100)*Investimento_rec(n-1,I,j)
```

Vejamos agora outro exemplo envolvendo juros: um financiamento.

Suponha que você vai pegar emprestado R$ 1.000,00 no banco e deverá pagar esse valor corrigido por juros

compostos de 1,5% ao mês. Ou seja, a cada mês sua dívida cresce 1,5%. Suponha que você está disposto a pagar parcelas mensais de R$ 200,00.

Considere y_n como sendo sua dívida após n meses do início do financiamento. Primeiro veja que a dívida após n meses depende diretamente da dívida após $n – 1$ meses:

$y_0 = 1.000,00$

$y_n = y_{n-1} + 0,015\ x_{n-1} – 200 = 1,015\ x_{n-1} – 200$

Com essas informações, podemos escrever um pseudocódigo que fornece a dívida após n meses. Só temos que tomar cuidado que essa sequência após um número finito de passos passa a ser negativa, caso contrário a dívida não seria paga nunca. Nesse caso, apesar de $y_n < 0$ a real dívida não existe mais, então a função deve retornar 0. Veja o pseudocódigo do algoritmo recursivo.

Entrada: n
Saída: yn = dívida após n meses, considerando juros de 1,5% ao mês e valor de empréstimo de R$ 1.000,00.
Nome: Divida_rec

```
1. Caso n não seja inteiro não negativo, retorne
erro.
2. Se n=0 retorne 1000;
3. Faça d = 1.015*Divida_rec(n-1) - 200;
4. Se d<0, retorne 0. Senão, retorne d.
```

Os exemplos de financiamento apresentados consideram o valor emprestado como um dado fixo, assim como os juros e as parcelas de pagamento. E se quiséssemos criar uma função que recebesse como entrada, além do número de meses de financiamento n, também o valor financiado V, os juros j e o valor das parcelas fixas K, como ficaria o pseudocódigo recursivo? Novamente é preciso ter um pou-

co mais de cuidado na chamada recursiva, pois a função agora terá mais de um argumento de entrada.

Entrada: n, V, j, K
Saída: yn = dívida após n meses, considerando juros de j% ao mês, valor emprestado de V reais e o pagamento de parcelas mensais de K reais.
Nome: Divida_rec

```
1. Caso n não seja inteiro não negativo, retorne
erro.
2. Se n=0 retorne V;
3. Faça d = (1 + j/100)*Divida_rec(n-1,V,j,K) - K;
4. Se d<0, retorne 0. Senão, retorne d.
```

Maior Divisor Comum

Um algoritmo recursivo bem famoso é o Algoritmo de Euclides, conhecido desde a obra *Elements* de Euclides, por volta de 300 a.C. Este algoritmo calcula o maior divisor comum entre dois números sem precisar fatorá-los. Lembrando, o maior divisor comum (MDC) entre dois números inteiros é o maior número inteiro que divide ambos sem deixar resto.

O algoritmo de Euclides é baseado no princípio de que o MDC não muda se o maior número for subtraído do menor. Por exemplo, o MDC entre 252 e 105 é 21. Veja que 252 = 21 × 12 e 105 = 21 × 5. Também é verdade que o MDC entre 252 – 105 = 147 e 105 é 21. Veja que 147 = 252 – 105 = 21 × 12 – 21 × 5 = 21 × 7.

A partir dessa ideia podemos afirmar que:

- $MDC\ (n, m) = MDC\ (m, n)$;
- Se $n = 0$, $MDC\ (n, m) = m$;

- Se $0 < n \leq m$, $MDC(n, m) = MDC(n, m - n) = MDC(m, m - n)$.

Podemos então criar uma função recursiva que encontra o *MDC* entre dois números inteiros quaisquer.

$$MDC(n,m) = \begin{cases} m & ,se\ n = 0 \\ n & ,se\ m = 0 \\ MDC(n, m-n) & ,se\ n \leq m \\ MDC(m, n-m) & ,se\ m < n \end{cases}$$

A partir da função anterior, podemos criar um pseudocódigo recursivo que recebe como entrada dois números inteiros não negativos *n* e *m* e retorna o maior divisor comum entre eles. Importante: Se os dois números forem 0, não faz sentido falar em MDC entre eles.

Entrada: *n* e *m*, inteiros não negativos
Saída: maior divisor comum entre *n* e *m*
Nome: MDC

```
1. Caso n ou m não seja inteiro não negativo,
retorne erro.
2. Seja maior= maior entre n e m;
3. Seja menor= menor entre n e m;
4. Se maior=0, retorne erro (isso significa que
m=n=0).
5. Se menor=0, retorne maior.
6. Retorne MDC(menor,maior-menor).
```

Veja que a chamada recursiva ocorre na linha 6 e que ambos os argumentos estão sendo simplificados. Isso nos faz garantir que em algum momento chega-se ao caso base, em que uma das entradas é zero.

Podemos melhorar ainda mais o desempenho do nosso algoritmo. Veja que se *m* é muito maior que *n* nossa

chamada recursiva sera MDC(n, $m - n$) várias vezes. Isso vai terminar quando já tivermos "tirado" todos os n que cabem dentro do m. Ou seja, a última chamada recursiva desse tipo será MDC(n, m%%n)! Então se substituirmos na linha 6 maior-menor por maior%%menor (resto da divisão de maior por menor), vamos economizar muitos passos da recursão. Nesse caso o pseudocódigo mais "enxuto" será:

Entrada: n e m, inteiros não negativos
Saída: maior divisor comum entre n e m
Nome: MDC_rapido

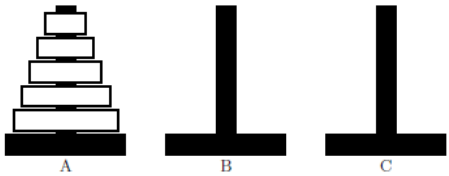
```
1. Caso n ou m não seja inteiro não negativo,
retorne erro.
2. Seja maior= maior entre n e m;
3. Seja menor= menor entre n e m;
4. Se maior=0, retorne erro.
5. Se menor=0, retorne maior.
6. Retorne MDC(menor,maior%%menor).
```

Torre de Hanoi

O problema ou quebra-cabeça conhecido como Torre de Hanói foi publicado em 1883 pelo matemático francês Édouard Anatole Lucas, também conhecido por seus estudos com a série Fibonacci.

O problema consiste em transferir a torre composta por N discos, como na figura a seguir, do pino A (origem) para o pino C (destino), utilizando o pino B como auxiliar. Somente um disco pode ser movimentado de cada vez e um disco não pode ser colocado sobre outro disco de menor diâmetro.

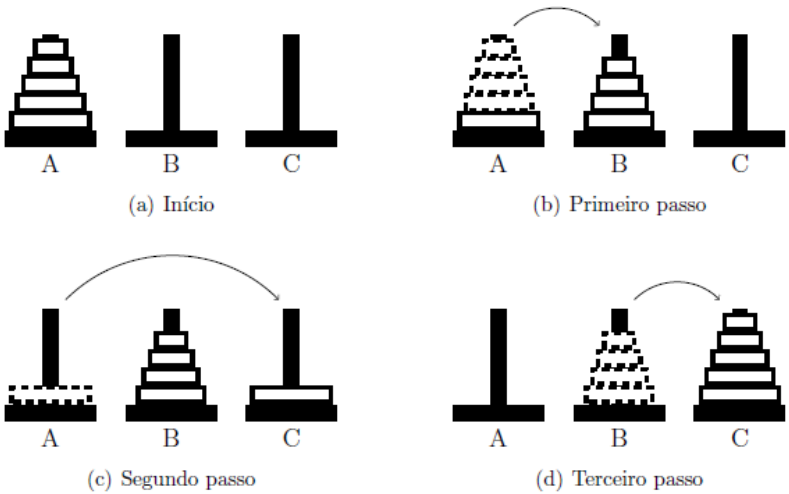
Torre de Hanoi



Veja que se existe um único disco, $N = 1$, a solução é imediata: mova este disco de A para C. E se existirem N discos, quais os movimentos que devemos fazer? É isso que vamos tentar resolver.

Suponha que sabemos a sequência de movimentos para mover $N - 1$ discos de um pino origem para um pino destino. Então para mover os N discos do pino A para o pino C devemos primeiro mover os primeiros $N - 1$ discos de A para B. Em seguida mova o disco que sobrou no pino A para o pino C. Para terminar basta mover novamente os $N - 1$ discos, agora do pino B para o pino C. Veja figura a seguir.

Esquema Recursivo da Torre de Hanoi



Dessa forma foi definida uma solução recursiva para o problema:

- Se $N = 1$: mova um disco do pino A para o pino C.
- Se $N > 1$:

» primeiro transfira $N-1$ discos de A para B;
» em seguida mova um disco do pino A para o pino C;
» por último transfira $N-1$ discos de B para C.

Baseado nessa ideia, vamos escrever o pseudocódigo que fornece a sequência de movimentos para transferir N discos do pino A (origem) para o pino C (destino).

Entrada: N, A, C, B (número de discos, nome do pino origem, nome do pino destino e nome do pino auxiliar).
Saída: Sequência de movimentos (que pode ser guardados em um vetor de "character")
Nome: Hanoi

```
1. Caso N não seja inteiro positivo, retorne erro.
2. Se N = 1, realize o movimento: "mova um disco do pino A para o pino C e fim".
3. Realize os movimentos de Hanoi(N-1,A,B,C)
4. Realize o movimento: "mova um disco do pino A para o pino C"
5. Realize os movimentos de Hanoi(N-1,B,C,A).
```

Vamos ver como fica esse algoritmo implementado no R.

```
Hanoi = function(N,A,C,B){
  if(N%%1 != 0 || N <=0)
    stop("N tem que ser um inteiro positivo")
  if(N==1)
    return(paste("mova um disco do pino ",A," para o pino", C,"."))

  movimentos_parte_1 = Hanoi(N-1,A,B,C)
```

```
  movimentos_parte_2 = paste("mova um disco do pino
",A," para o pino", C,".")
  movimentos_parte_3 = Hanoi(N-1,B,C,A)


  return(c(movimentos_parte_1,movimentos_
parte_2,movimentos_parte_3))
}
Hanoi(2,"A","C","B")
## [1] "mova um disco do pino  A  para o pino B ."
## [2] "mova um disco do pino  A  para o pino C ."
## [3] "mova um disco do pino  B  para o pino C ."
```

Veja que os argumentos de entrada são os nomes dos pinos, por isso eles entram como texto.

```
Hanoi(3,"Inicio","Fim","Aux")
## [1] "mova um disco do pino  Inicio  para o pino
Fim ."
## [2] "mova um disco do pino  Inicio  para o pino
Aux ."
## [3] "mova um disco do pino  Fim  para o pino Aux
."
## [4] "mova um disco do pino  Inicio  para o pino
Fim ."
## [5] "mova um disco do pino  Aux  para o pino
Inicio ."
## [6] "mova um disco do pino  Aux  para o pino Fim
."
## [7] "mova um disco do pino  Inicio  para o pino
Fim ."
```

Se quiser, testes os movimentos do seu programa no site *https://www.mathsisfun.com/games/towerofhanoi.html*.

Exercícios

1. Suponha que você vá investir R$ 500,00 na poupança e que esta rende 7,5% ao ano.
   a. Calcule na mão o quanto de dinheiro você teria no banco depois de 1, 2 e 3 anos de investimento.
   b. Tente achar uma equação que relacione o total de dinheiro acumulado em n anos de investimento com o total de dinheiro acumulado em $n - 1$ anos.
   c. Usando a equação encontrada, no caderno escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada um número natural n e retorna o dinheiro acumulado em n anos nesse investimento.
   d. Agora no computador, implemente o pseudocódigo elaborado.

2. A generalização do exercício sobre investimento.
   a. Seja I o valor investido em uma aplicação de rentabilidade j% ao ano. Implemente uma função que recebe como entrada I, j e n e retorna o total acumulado nessa aplicação após n anos.
   b. Use a função implementada para descobrir quanto de dinheiro teríamos a mais se investíssemos R$ 1.000,00 durante 2 anos em um fundo que rendesse 10% ao ano em vez de 7,5%.

3. Suponha que você vai fazer um financiamento de R$ 1.200,00 e vai pagar juros compostos de 2% ao mês. Considere que você pode pagar R$ 150,00 por mês.
   a. Calcule na mão o valor da sua dívida depois de 1, 2 e 3 meses.

b. Tente achar uma equação que relacione a sua dívida no mês n com a sua dívida no mês $n - 1$.
c. Usando a equação encontrada, escreva no caderno um pseudocódigo recursivo para o algoritmo que recebe como entrada um número natural n e retorna a sua dívida após n meses do início do financiamento. Não se esqueça de considerar o caso em que a dívida foi paga, nesse caso você deve retornar 0.
d. Agora no computador, implemente o pseudocódigo elaborado.

4. A generalização do exercício sobre financiamento.
e. Seja V o valor financiado a juros compostos de j% ao mês, considere K o valor das parcelas fixas que serão pagas todo mês. Implemente uma função recursiva que recebe como entrada V, j, K e n e retorna a dívida existente após n meses desde o início do financiamento.
f. Use a função implementada para comparar o valor da sua dívida de R$ 1.200,00 após 10 meses nos seguintes casos: parcela mensal de R$ 150,00 e parcela mensal de R$ 120,00. Considere os mesmos 2% de juros compostos ao mês.

5. Mais um exercício de matemática financeira. Agora estamos interessados na quantidade de meses para se pagar a dívida. Suponha que você vai fazer um financiamento de V reais e vai pagar juros compostos de j% ao mês. Considere que você pode pagar K reais por mês.
a. No caderno, escreva um pseudocódigo recursivo para o algoritmo que recebe como entrada V, j e K e retorna o número de meses que você vai demorar para pagar a sua dívida. Dica: A simplificação na chamada recursiva ocorre na entrada V.
b. Agora no computador, implemente o pseudocódigo elaborado.

c. Use a função implementada para comparar o número de meses até a quitação da dívida de R$ 1.200,00 nos seguintes casos: parcela mensal de R$ 120,00; parcela mensal de R$ 150,00 e parcela mensal de R$ 200,00. Considere os mesmos 2% de juros compostos ao mês.

6. Implemente de forma recursiva uma função que recebe como entrada dois números inteiros e retorna o maior divisor comum entre eles. Usando a função anterior, encontre o maior divisor comum entre os seguintes pares de números: 125 e 325, 2.829 e 861, 299 e 217.

7. Vamos estudar o crescimento exponencial de forma recursiva. Ou seja, repetir o Exercício 12 do Capítulo 3, porém de forma recursiva. Novamente pense na seguinte situação: você arranjou um emprego novo e a empresa perguntou qual das duas opções de salários você prefere. A primeira é um salário fixo de R$ 5.000,00 ao mês. A segunda opção é um pagamento diário, que começa no primeiro dia útil do mês com o pagamento de R$ 0,01 (1 centavo de real) e a cada novo dia útil você ganha o dobro do que ganhou no dia útil anterior.

a. Agora no computador, implemente uma função que recebe como argumento de entrada o valor n e retorna uma lista com dois objetos: o primeiro deles é o valor pago no *n*-ésimo dia útil do mês; e o segundo objeto da lista é o valor total pago no mês depois de passados *n* úteis, considerando a segunda opção de salário.
b. Use a função implementada no item anterior para saber qual seria o seu pagamento em um mês com 20 dias úteis. E se o mês tiver 22 dias úteis?

Obs.: O retorno de uma lista com 2 objetos é necessário para conseguir implementar a função de forma recursiva.

8. (Questão Desafio) Implemente uma função recursiva que recebe como entrada um vetor de dados e retorna todos as

permutações possíveis com ele. A saída deve ser uma lista em que cada posição guarda uma permutação em forma de um vetor do tipo "numeric". Quando for testar sua função, entre com entradas de dimensão pequena, não muito maior que 10, caso contrário vai demorar muito para rodar. Lembre-se de que para um conjunto com n elementos temos $n!$ permutações.

CAPÍTULO 8: ALGORITMOS DE ORDENAÇÃO

Nesse capítulo serão apresentados dois algoritmos de ordenação, isto é, algoritmos que recebem como entrada um vetor de números (ou de "character") e retorna outro vetor com os mesmos elementos do vetor de entrada, só que em ordem crescente. Veja que isso nada mais é do que a implementação da função sort() já pronta no R.

Existem vários algoritmos de ordenação, uns mais eficientes que os outros. Mas vamos nos concentrar em dois métodos: O método de Ordenação Bolha (*Bubble Sort*) e o método de Ordenação Rápida (*Quick Sort*).

Ordenação Bolha (*Bubble Sort*)

A ideia desse método é ordenar o vetor seguindo os seguintes passos:

- primeiro leve o maior elemento para a última posição, comparando os elementos dois a dois até a última posição;
- depois repita o processo e levar o segundo maior elemento para a segunda maior posição, comparando os elementos dois a dois até a penúltima posição;
- esse processo se repete várias vezes até que o vetor esteja todo ordenado.

Suponha v = (37, 33, 48, 12, 92, 25, 86, 57) o vetor de entrada, ou seja, o vetor que queremos ordenar. O primeiro passo é levar o maior elemento para a maior posição fazendo comparações de elementos dois a dois.

V1	V2	V3	V4	V5	V6	V7	V8	COMENTÁRIOS
37	33	48	12	92	25	86	57	como 37 > 33, troca
33	37	48	12	92	25	86	57	como 37 < 48, não troca
33	37	48	12	92	25	86	57	como 48 > 12, troca
33	37	12	48	92	25	86	57	como 48 < 92, não troca
33	37	12	48	92	25	86	57	como 92 > 25, troca
33	37	12	48	25	92	86	57	como 92 > 86, troca
33	37	12	48	25	86	92	57	como 92 > 57, troca
33	37	12	48	25	86	57	92	fim dessa etapa.

Veja que para realizar essa etapa foram necessárias 7 comparações. Agora que o maior elemento já está na última posição o próximo passo é levar o segundo maior elemento até a segunda maior posição. Repare que nesse caso as comparações serão até a penúltima posição, uma vez que a última posição já está com o seu devido elemento.

V1	V2	V3	V4	V5	V6	V7	V8	COMENTÁRIOS
33	37	12	48	25	86	57	92	como 33 < 37, não troca
33	37	12	48	25	86	57	92	como 37 > 12, troca
33	12	37	48	25	86	57	92	como 37 < 48, não troca
33	12	37	48	25	86	57	92	como 48 > 25, troca
33	12	37	25	48	86	57	92	como 48 < 86, não troca
33	12	37	25	48	86	57	92	como 86 > 57, troca
33	12	37	25	48	57	86	92	fim dessa etapa.

Veja que para realizar essa etapa foram necessárias 6 comparações. Agora o terceiro maior elemento será levado para a terceira maior posição.

V1	V2	V3	V4	V5	V6	V7	V8	COMENTÁRIOS
33	12	37	25	48	57	86	92	como 33 > 12, troca
12	33	37	25	48	57	86	92	como 33 < 37, não troca
12	33	37	25	48	57	86	92	como 37 > 25, troca

12	33	25	37	48	57	86	92	como 37 < 48, não troca
12	33	25	37	48	57	86	92	como 48 < 57, não troca
12	33	25	37	48	57	86	92	fim dessa etapa.

Veja que para realizar essa etapa foram necessárias 5 comparações. Agora o quarto maior elemento será levado para a quarta maior posição.

V1	V2	V3	V4	V5	V6	V7	V8	COMENTÁRIOS
12	33	25	37	48	57	86	92	como 12 < 33, não troca
12	33	25	37	48	57	86	92	como 33 > 25, troca
12	25	33	37	48	57	86	92	como 33 < 37, não troca
12	25	33	37	48	57	86	92	como 37 < 48, não troca
12	25	33	37	48	57	86	92	fim dessa etapa.

Veja que para realizar essa etapa foram necessárias 4 comparações.

Nesse momento, o vetor já está ordenado, porém o computador não tem como saber disso. Por isso as etapas seguintes serão realizadas, apesar de nenhuma troca ser feita.

Algumas observações sobre o método e sobre o exemplo anterior.

- Em um vetor de tamanho n é preciso de $n - 1$ comparações para realizar a primeira etapa, isto é, levar o maior elemento para a maior posição.
- Já para levar o segundo maior elemento para a segunda maior posição são realizadas $n - 2$ comparações.
- De forma geral, para levar o j-ésimo maior elemento para a j-ésima maior posição, isto é realizar a etapa j, são realizadas $n - j$ comparações.
- Em cada etapa j serão comparados os elementos das posições i e $i + 1$ para $i = 1, ..., n - j$.

- De forma geral, para ordenar um vetor de tamanho n são realizadas $n - 1$ etapas.

Agora pensando no pseudocódigo, veja que serão necessários dois laços, um dentro do outro. Um para controlar as etapas e outro para controlar as comparações dentro de cada etapa. Veja primeiro um pseudocódigo não recursivo para o algoritmo apresentado.

Entrada: v
Saída: vetor com os elementos de v ordenados em ordem crescente.
Nome: OrdenaBolha

```
1. Defina n = tamanho do vetor v;
2. Inicie j = 1;
3. Inicie i = 1;
4. Se v[i] > v[i+1], troque a posição i com posição
i+1 no vetor v; (ATENÇÃO!)
5. Incremente: i = i + 1;
6. Se i <= n-j, volte para a linha 4;
7. Incremente: j = j + 1;
8. Se j <= n-1, volte para a linha 3;
9. Retorne v.
```

Veja que a variável j controla as etapas, por isso ela varia de 1 até $n - 1$. Já a variável i controla as comparações dentro de cada etapa j, por isso ela varia de 1 até $n - j$.

Atenção: Na linha 4, o elemento da posição i será trocado com o elemento da posição $i + 1$. Para isso é necessário utilizar uma variável temporária, senão um dos elementos do vetor será perdido.

No R a troca dos elementos entre as posições i e $i + 1$ pode ser feita da seguinte maneira:

```
temp <- v[i]
v[i] <- v[i+1]
v[i+1] <- temp
```

Agora o pseudocódigo do algoritmo recursivo.

Entrada: *v*
Saída: vetor com os elementos de *v* ordenados em ordem crescente.
Nome: OrdenaBolhaRec

```
1. Defina n = tamanho do vetor v;
2. Se n=1, retorne v;
3. Inicie i = 1;
4. Se v[i] > v[i+1], troque a posição i com posição i+1 no vetor v; (ATENÇÃO!)
5. Incremente: i = i + 1;
6. Se i <= n-1, volte para a linha 4;
7. Defina w = v[1:(n-1)];
8. Defina wo = OrdenaBolhaRec(w);
9. Retorne vo = (wo,v[n]).
```

Veja que entre as linhas 2 e 6 o que está sendo feito é levar o maior elemento para a maior posição. Dessa forma o vetor *w* será formado pelos elementos do vetor *v* a menos do maior deles, que agora está na posição de índice *n*. Dessa forma *wo* guarda os elementos de *v*, a menos do maior deles, já em ordem crescente. Então *v* em ordem crescente será definido pela concatenação entre os elementos em *wo* e o maior elemento de *v*, que está guardado em v_n.

Ordenação Rápida (*Quick Sort*)

Nesta seção será visto outro algoritmo para ordenação de vetores. É uma opção que realiza menos comparações em média, mas é também de mais difícil compreensão. A ideia desse método é ordenar o vetor da seguinte maneira:

- Primeiro o valor guardado na primeira posição, chamado de pivô, será levado para a sua posição correta de forma que os elementos à esquerda são menores que o pivô e os elementos à direita são maiores que o pivô.
- O passo seguinte é repetir o mesmo processo no subvetor à esquerda e no subvetor à direita do pivô.

A questão principal é: como se leva o pivô para a sua posição correta? Já essa tarefa será feita da seguinte maneira:

- Primeiro percorra o vetor a partir da posição seguinte a do pivô até encontrar um elemento maior que o pivô. Seja i a posição desse elemento.
- Se não existir elemento maior que o pivô, a posição correta do pivô é a última e FIM.
- Caso exista, percorra o vetor do final para o início até encontrar um elemento menor ou igual ao pivô. Seja j a posição desse elemento.
- Se $i < j$, troque os elementos das posições i e j e recomece a busca por novos i e j a partir das posições trocadas.
- Se $i > j$, a posição correta para o pivô é a posição j.

Vejamos como o mesmo vetor v será ordenado pelo método de Ordenação Rápida.

V1	V2	V3	V4	V5	V6	V7	V8	COMENTÁRIOS
37	33	48	12	92	25	86	57	pivô = 37, $i \neq 2$
37	33	48	12	92	25	86	57	$i = 3$
37	33	48	12	92	25	86	57	$j \neq 8$
37	33	48	12	92	25	86	57	$j \neq 7$

37	33	48	12	92	25	86	57	$j = 6$
37	33	25	12	92	48	86	57	como $i < j$, troca v3 com v6
37	33	25	12	92	48	86	57	$i \neq 4$
37	33	25	12	92	48	86	57	$i = 5$
37	33	25	12	92	48	86	57	$j \neq 5$
37	33	25	12	92	48	86	57	$j = 4$
12	33	25	37	92	48	86	57	como $i > j$, troca pivô com v4

Veja que depois desses passos o pivô está em sua posição correta uma vez que antes dele estão os elementos menores que ele e depois os maiores. Agora o algoritmo é repetido para o subvetor à esquerda do pivô.

V1	V2	V3	COMENTÁRIOS
12	33	25	pivô = 12, $i = 2$
12	33	25	$j \neq 3$
12	33	25	$j \neq 2$
12	33	25	$j = 1$
12	33	25	como $i > j$, troca pivô com v1

Teoricamente o processo seria refeito no subvetor a esquerda do pivô 12, mas como não existe vamos para o subvetor à direita do pivô 12.

V1	V2	COMENTÁRIOS
33	25	pivô = 33, $i \neq 2$
25	33	não existe i, pivô vai para a última posição

Juntando o resultado do subvetor à esquerda do primeiro pivô, o 37, já temos um pedaço ordenado: (12, 25, 33, 37). Vamos agora trabalhar com o subvetor à direita do pivô 37.

V1	V2	V3	V4	COMENTÁRIOS
92	48	86	57	pivô = 92, $i \neq 2$
92	48	86	57	$i \neq 3$
92	48	86	57	$i \neq 4$
57	48	86	92	não existe *i*, pivô vai para a última posição

Repetir o processo no subvetor à esquerda do 92.

V1	V2	V3	COMENTÁRIOS
57	48	86	pivô = 57, $i \neq 2$
57	48	86	$i = 3$
57	48	86	$j \neq 3$
57	48	86	$j = 2$
48	57	86	como $i > j$, troca pivô com v2

Nesse próximo passo, o processo seria aplicado nos subvetores à esquerda e à direita do 57. Mas com tamanho 1 a ordenação é imediata, retorna o próprio vetor de entrada.

Percebeu que o vetor foi ordenado? Juntando tudo: (12, 25, 33, 37, 48, 57, 86, 92).

Reparou que a ideia do algoritmo é recursiva? Caso básico: se $n = 1$ retorna o próprio vetor de entrada. Se não, primeiro coloque o pivô na posição correta, como explicado. Em seguida defina *we* e *wd* como os subvetores à esquerda e à direita e aplique a ordenação rápida em cada um deles. O vetor ordenado será a concatenação entre *we* ordenado, o pivô e *wd* ordenado. Veja o pseudocódigo.

Entrada: *v*
Saída: vetor com os elementos de *v* ordenados em ordem crescente.
Nome: OrdenaRapidoRec

```
1. Defina n = tamanho do vetor v;

2. Se n=1, retorne v;

3. Defina pivo = v[1], i = 2 e j=n;

4. Se v[i] <= pivo e i < n, faça i = i + 1 e repete
a linha 4;

5. Se i = n e v[i] <= pivo: troca o pivo com v[n];
defina w=v[1:(n-1)]; wo = OrdenaRapidoRec(w) e
retorna c(wo,pivo).

6. Se v[j] > pivo, faça j = j - 1 e repete a linha
6.

7. Se j = 1: defina w=v[2:n]; wo =
OrdenaRapidoRec(w) e retorna c(pivo,wo).

8. Se j < i: troca o v[j] com o pivo;
defina we=v[1:(j-1)]; wd=v[j+1,n]; weo =
OrdenaRapidoRec(we); wdo = OrdenaRapidoRec(wd);
retorne c(weo,pivo,wdo).

9. Se i < j, troca o v[i] com o v[j] e volte para a
linha 4;
```

Como esse não é um algoritmo fácil de ser implementado, mesmo com o pseudocódigo, segue o código em R para a função OrdenaRapidoRec. Só uma última observação, essa função ficaria complicada de mais de forma não recursiva.

```
OrdenaRapidoRec = function(v){
  n = length(v)


  if(n==1)
    return(v)


  pivo = v[1]
```

```
  i = 2
  j = n

  repeat{
    #busca do i
    while(v[i] <= pivo && i < n){
      i = i + 1
    }

    if(i == n &&  v[i] <= pivo){ #nao encontramos i
tal que v[i]>pivô
      #troca o pivô com v[n]
      v[1] = v[n]
      v[n] = pivo

      w=v[1:(n-1)]
      wo = OrdenaRapidoRec(w)
      return(c(wo,pivo))
    }

    #busca do j
    while(v[j] > pivo){
      j = j - 1
    }

    if(j == 1){
```

```
      w=v[2:n]
      wo = OrdenaRapidoRec(w)
      return(c(pivo,wo))
    }

    if(j < i){
      #troca o v[j] com o pivô
      v[1] = v[j]
      v[j] = pivo

      we=v[1:(j-1)]
      wd=v[(j+1):n]
      weo = OrdenaRapidoRec(we)
      wdo = OrdenaRapidoRec(wd)

      return(c(weo,pivo,wdo))
    }

    #troca o v[i] com o v[j]
    aux = v[i]
    v[i] = v[j]
    v[j] = aux

  }
}
```

```
OrdenaRapidoRec(c(9,7,4,1,2,8,3,5,6))
## [1] 1 2 3 4 5 6 7 8 9
```

Exercícios

1. Implemente uma função que recebe como entrada um vetor *v* e retorna um outro vetor com os mesmos elementos de *v*, mas em ordem crescente a partir do algoritmo de Ordenação Bolha visto na última aula teórica.
   a. Faça primeiro a função não recursiva.
   b. Faça agora a função recursiva.

2. Modifique as duas funções implementadas no Exercício 1 de forma que elas retornem, além do vetor ordenado, o número de comparações realizadas para ordenar o vetor. Toda vez que dois valores são comparados, para saber quem é o maior (ou menor) uma comparação deve ser contabilizada.

3. Modifique as duas funções implementadas no Exercício 1 de forma que elas retornem, além do vetor ordenado, o número de trocas realizadas para ordenar o vetor. Toda vez que é feita uma troca de elementos dentro do vetor, uma troca deve ser contabilizada.

4. Para a função recursiva do Exercício 1 coloque antes de cada troca de posição o comando plot(v), em que no lugar de v deve entrar o nome que você deu ao vetor na sua função. Agora digite as linhas de comandos a seguir, em que novafuncao é o nome da função modificada nesse exercício. Veja os gráficos plotados e interprete.

```
v <- c(5,9,4,2,6,10,3,8,1,7)
w <- novafuncao(v)
plot(w)
```

5. Baseado na mesma ideia do algoritmo de Ordenação Bolha, escreva um pseudocódigo de uma função recursiva que recebe como entrada um vetor v e retorna um outro vetor com os mesmos elementos de v, mas em ordem decrescente. Em seguida, implemente no R o pseudocódigo elaborado.

6. Implemente uma função que recebe como entrada um vetor v e retorna um outro vetor com os mesmos elementos de v mas em ordem crescente a partir do algoritmo de Ordenação Rápida visto na última aula teórica. Obs.: Tente fazer primeiro sem olhar a implementação feita na apostila e em sala de aula.

7. Modifique a função implementada no Exercício 6 de forma que ela retorne, além do vetor ordenado, o número de comparações realizadas para ordenar o vetor. Toda vez que dois valores são comparados para saber quem é o maior (ou menor) uma comparação deve ser contabilizada.

8. Modifique a função implementada no Exercício 6 de forma que ela retorne, além do vetor ordenado, o número de trocas realizadas para ordenar o vetor. Toda vez que é feita uma troca de elementos dentro do vetor, uma troca deve ser contabilizada, inclusive quando o pivô é trocado com algum elemento para ser colocado na sua posição correta.

9. Baseado na mesma ideia do algoritmo de Ordenação Rápida, escreva um pseudocódigo de uma função recursiva que recebe como entrada um vetor v e retorna um outro vetor com os mesmos elementos de v mas em ordem decrescente. Em seguida, implemente no R o pseudocódigo elaborado.

10. Usando as funções implementadas nos Exercícios 2, 3, 7 e 8 verifique quantas comparações e quantas trocas cada um dos dois algoritmos de ordenação faz para ordenar os vetores a seguir. Obs.: No R é possível comparar palavras

com o comando <. Por exemplo, digite no cursor o comando "aaa" < "aba" e veja a resposta. Dessa forma você também pode passar como entrada para as funções implementadas um vetor de caracteres, como sugerido no item c.

a. v < -c (1, 3, 5, 5, 4, 0, -1, 2, 6, -2)
b. v < -c (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
c. v < -c ("fabio", "ana", "pedro", "bruno", "bruna", "maria", "clara", "julia", "vicente", "daniel")

PARTE III: UMA INTRODUÇÃO AO CÁLCULO NUMÉRICO

Para terminar este livro, serão apresentados mais quatro capítulos em que se pretende introduzir conceitos do cálculo numérico. A ideia é aplicar as técnicas apresentadas nos capítulos anteriores e mostrar como tudo o que o computador faz pode ser resumido às operações básicas (+, –, *, /) com tratamento de condições (if/else) e laços (for, while, repeat). A recursão pode ser um recurso importante para facilitar alguns algoritmos também.

CAPÍTULO 9: APROXIMAÇÃO DE FUNÇÕES

Nesta semana, vamos aprender como o computador avalia funções como e^x, $ln(x)$ e $sen(x)$. A ideia principal é que o computador faz somente as operações básicas, todo o resto é programado. Na prática, o computador encontra valores aproximados para $e^{-1,5}$, $ln(0,5)$ ou $sen(1)$, e se ele pode fazer isso usando apenas operações básicas nós também podemos.

Aproximação para $f(x) = e^x$

Resultados de cálculo, que não serão discutidos neste curso, nos mostram que

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{x^i}{i!} = e^x \qquad , \forall\, x \in R.$$

Essa é a Série de Taylor em torno de $x_0 = 0$ para a função $f(x) = e^x$. Ou seja, para qualquer $x \in R$, se $n \in N$ for um número bem grande,

$$\sum_{i=0}^{n} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} \approx e^x.$$

Vale destacar que a convergência é mais rápida para valores de x próximos de $x_0 = 0$. Isso significa que, considerando um número fixo de parcelas na equação acima, quanto mais próximo x está de zero, mais perto a soma está de e^x. Mas ela acontece qualquer que seja $x \in R$, ou seja, mesmo para x distante de zero a soma converge para e^x quando o número de parcelas cresce.

Exemplo: Vejamos como podemos usar a Série de Taylor apresentada para encontrar uma aproximação para $e^{-1,5}$, sem a ajuda do computador, usando apenas operações de soma e multiplicação.

Vamos começar encontrando uma aproximação usando $n = 2$:

$$e^{-1,5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} = 1 - \frac{3}{2} + \frac{9}{8} = \frac{8-12+9}{8} = \frac{5}{8} = 0,625$$

E se quisermos uma aproximação mais precisa podemos usar um valor maior de n. Vamos ter que fazer um pouco mais de contas, mas teremos uma aproximação melhor. Por exemplo, veja a aproximação para $n = 3$:

$$e^{-1,5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} + \frac{\left(-\frac{3}{2}\right)^3}{3!} = \frac{5}{8} - \frac{9}{16} = \frac{10-9}{16} = \frac{1}{16} = 0,062$$

Agora para $n = 4$:

$$e^{-1,5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} + \frac{\left(-\frac{3}{2}\right)^3}{3!} + \frac{\left(-\frac{3}{2}\right)^4}{4!} = \frac{1}{16} + \frac{27}{128} = \frac{35}{128} \approx 0,2734$$

E para $n = 5$:

$$e^{-1,5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} + \frac{\left(-\frac{3}{2}\right)^3}{3!} + \frac{\left(-\frac{3}{2}\right)^4}{4!} + \frac{\left(-\frac{3}{2}\right)^5}{5!} = \frac{35}{128} - \frac{81}{1280} = \frac{269}{1280} \approx 0,2101$$

Se usarmos uma calculadora (ou computador) vamos encontrar a seguinte aproximação para $e^{-1,5}$:

```
exp(-1.5)
## [1] 0.2231302
```

Veja que realmente as aproximações calculadas melhoraram quando n cresceu.

Critério de Parada

Se tivermos um computador, podemos criar um programa que faça as contas para um valor escolhido de n,

e assim conseguimos uma aproximação tão boa quanto a gente quiser. Mas como escolher o valor de n? Existem alternativas mais sofisticadas que conseguem controlar o erro da aproximação, mas por esse ser um curso introdutório, vamos optar por uma estratégia mais simples.

Vamos usar no mínimo $n = 10$. A partir desse valor de n calculamos atualizamos a aproximação até que o incremento entre duas aproximações consecutivas, que para esse caso é $x^n \div n!$, seja relativamente pequeno.

Pseudocódigo

Veja o seguinte pseudocódigo para encontrar uma aproximação para e^x.

Entrada: x e delta.
Saída: uma aproximação para e^x.
Nome: AproxExp

```
1. Calcule aprox = 1 + x + (x^2)/(2!) + (x^3)/(3!)
+ ... + (x^10)/(10!).
2. Faça n = 11.
3. Defina incr = (x^n)/(n!).
4. Faça aprox = aprox + incr.
5. Se |incr| < delta, retorne aprox.
6. Faça n = n + 1.
7. Volte para a linha 3.
```

O pseudocódigo anterior simplesmente calcula a série apresentada no início dessa seção até que o incremento da série, definido por incr, seja menor que o valor atribuído à variável delta, escolhido pelo usuário e passado como argumento de entrada. Garantimos que isso em algum momento acontece uma vez que $(x^n \div n!) \times n \to \infty \to 0$ para todo $x \in R$. A escolha do valor delta é feita pelo usuário. Ge-

ralmente se escolhem valores muito pequenos, como por exemplo delta = 0,001, ou até menores.

Veja que a linha 3 do pseudocódigo anterior poderia ser alterada para incr = incr*x/n. Isso tornaria o código mais eficiente, pois não seria necessário calcular o fatorial em cada iteração do programa. Mas para isso seria necessário garantir que o valor de incr foi iniciado em algum momento.

Aproximação para $f(x) = ln(x)$

Resultados de cálculo, que não serão discutidos neste livro, nos mostram que

$$\lim_{n\to\infty} \sum_{i=0}^{n} (-1)^i \frac{(x-1)^{i+1}}{i+1} = ln(x) \quad , \forall\ 0 < x < 2.$$

Essa é a Série de Taylor em torno de $x_0 = 1$ para a função $f(x) = ln(x)$ e só converge para $0 < x < 2$.

Ou seja, se $0 < x < 2$, se $n \in N$ for um número bem grande,

$$\sum_{i=0}^{n} (-1)^i \frac{(x-1)^{i+1}}{i+1} = (x-1) - \frac{(x-1)^2}{2} + \ldots + (-1)^n \frac{(x-1)^{n+1}}{n+1} \approx ln(x).$$

Vale destacar que a convergência é mais rápida para valores de x próximos de $x_0 = 1$. Isso significa que, considerando um número fixo de parcelas na equação anterior, quanto mais próximo x está de 1, mais perto a soma está de $ln(x)$. Mas ela acontece qualquer que seja $0 < x < 2$, ou seja, a soma converge para $ln(x)$ quando o número de parcelas cresce.

Exemplo: veja como encontrar uma aproximação para $ln(0,5)$ sem usar calculadora ou o computador, usando apenas as operações de soma e multiplicação. Para isso será aplicada a Série de Taylor apresentada acima, atribuindo algum valor para n. Quanto maior o valor de n, melhor será a aproximação. Veja como fica a aproximação para $n = 3$.

$$ln(0,5) \approx \left(\frac{1}{2} - 1\right) - \frac{\left(\frac{1}{2}-1\right)^2}{2} + \frac{\left(\frac{1}{2}-1\right)^3}{3} - \frac{\left(\frac{1}{2}-1\right)^4}{4} =$$

$$-\frac{1}{2} - \frac{1}{8} - \frac{1}{24} - \frac{1}{64} = \frac{-96-24-8-3}{192} \approx -0,677.$$

Já pelo computador, a aproximação fornecida para $ln(0,5)$ é:

```
log(0.5)
## [1] -0.6931472
```

Mas e se quisermos o valor de $ln(x)$ sendo $x \geq 2$? Nesse caso não podemos usar a Série de Taylor substituindo x, pois a convergência só acontece para $x \geq 20 < x < 2$. Mas podemos usar uma propriedade dos logaritmos para resolver esse problema. Veja que

$$ln\left(\frac{a}{b}\right) = ln(a) - ln(b)$$

sempre que $b \neq 0$. Então,

$$ln\left(\frac{1}{x}\right) = ln(1) - ln(x) = - ln(x) \Rightarrow ln(x) = - ln\left(\frac{1}{x}\right).$$

Além disso, se $x > 2 \Rightarrow 0 < 1/x < 2$, logo podemos usar a Série de Taylor para encontrar uma aproximação para $ln(1/x)$. Resumindo, se precisamos encontrar $ln(x)$ para $x > 2$, então vamos primeiro encontrar uma aproximação para $ln(1/x)$ e depois multiplicar por -1, que temos então uma aproximação para $ln(x)$. Veja um exemplo.

Exemplo: encontre uma aproximação para $ln(3)$ usando a Série de Taylor com $n = 3$.

$$ln(3) = -ln\left(\frac{1}{3}\right) \approx -\left(\left(\frac{1}{3} - 1\right) - \frac{\left(\frac{1}{3}-1\right)^2}{2} + \frac{\left(\frac{1}{3}-1\right)^3}{3} - \frac{\left(\frac{1}{3}-1\right)^4}{4}\right) =$$

$$-\left(-\frac{2}{3} - \frac{2}{9} - \frac{8}{81} - \frac{16}{324}\right) = \frac{336}{324} \approx 1,037$$

Usando agora a calculadora ou o computador é possível encontrar a seguinte aproximação para $ln(3)$:

```
log(3)
## [1] 1.098612
```

Critério de Parada

O critério de parada é o mesmo adotado na aproximação da exponencial: o algoritmo começa com $n = 10$ e depois incrementa n até encontrar duas aproximações consecutivas tais que a diferença é menor que o valor de delta definido pelo usuário.

Pseudocódigo

Veja no pseudocódigo a seguir como fica o algoritmo para encontrar uma aproximação para $ln(x)$.

Entrada: x e delta.
Saída: uma aproximação para $ln(x)$.
Nome: AproxLn

```
1. Se x <= 0, pare e retorne uma mensagem de erro.
2. Se x >= 2, retorne -AproxLn(1/x,delta).
3. Calcule aprox = (x-1) - ((x-1)^2)/2 + ((x-
1)^3)/3 - ...  + ((-1)^10)*((x-1)^11)/11.
4. Faça n = 11.
5. Defina incr = ((-1)^(n))*((x-1)^(n+1))/(n+1)
6. Faça aprox = aprox + incr.
7. Se |incr| < delta, retorne aprox.
8. Faça n = n + 1.
9. Volte para a linha 5.
```

Veja que a linha 5 poderia ser alterada para incr = -incr*(x-1)*(n-1)/n. Isso tornaria o código mais eficiente. Mas para isso seria necessário garantir que o valor de incr foi iniciado em algum momento.

Aproximação para $f(x) = sen(x)$

Resultados de cálculo, que não serão discutidos neste livro, nos mostram que

$$\lim_{n\to\infty} \sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = sen(x) \quad , \forall \ x \in R.$$

Essa é a Série de Taylor em torno de $x_0 = 0$ para a função $f(x) = sen(x)$.

Ou seja, para qualquer $x \in R$, se $n \in N$ for um número bem grande,

$$\sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} \approx sen(x)$$

Vale destacar que a convergência é mais rápida para valores de x próximos de $x_0 = 0$. Isso significa que, considerando um número fixo de parcelas na equação anterior, quanto mais próximo x está de 0, mais perto a soma está de $sen(x)$. Mas ela acontece qualquer que seja $x \in R$, ou seja, a soma sempre converge para $sen(x)$ quando o número de parcelas cresce.

Atenção: O argumento da função *sen* é a medida do ângulo em radianos. Dessa forma, *sen* é uma função periódica com período de tamanho 2π.

Exemplo: vejamos como encontrar uma aproximação para $sen(1)$ sem usar calculadora ou computador, usando apenas as operações de soma e multiplicação. Para isso vamos usar a Série de Taylor definida com $n = 4$.

$$sen(1) \approx 1 - \frac{1^3}{3!} + \frac{1^5}{5!} - \frac{1^7}{7!} + \frac{1^9}{9!} =$$

$$1 - \frac{1}{6} + \frac{1}{120} - \frac{1}{5.040} + \frac{1}{362.880} = 0,8414710$$

Se usarmos o computador, vamos encontrar a seguinte aproximação para *sen*(1):

```
sin(1)
```

```
## [1] 0.841471
```

Veja que com $n = 4$ já temos uma aproximação excelente. Isso se deve ao fato de que 1 está razoavelmente próximo de zero. Provavelmente a aproximação não seria tão boa com $n = 4$ para encontrar *sen*(10). Mas podemos aproveitar a periodicidade da função seno e aumentar a velocidade de convergência mesmo para valores de x longe de zero. Veja um outro exemplo.

Exemplo: usando o computador, encontramos a seguinte aproximação para *sen*(10):

```
sin(10)
```

```
## [1] -0.5440211
```

Vamos encontrar uma aproximação sem a ajuda do computador. Primeiro, a partir da Série de Taylor em torno de $x_0 = 0$ para a função $f(x) = sen(x)$ usando $n = 4$ e $x = 10$.

$$(10) \approx 10 - \frac{10^3}{3!} + \frac{10^5}{5!} - \frac{10^7}{7!} + \frac{x^9}{9!} = 10 - \frac{1.000}{6} +$$

$$\frac{100.000}{120} - \frac{10.000.000}{5.040} + \frac{1.000.000.000}{362.880} \approx -1.307,46.$$

Com $n = 4$, chegamos a uma aproximação muito ruim, precisaríamos de muito mais parcelas para chegar perto do valor correto de *sen*(10).

Veja que, pela periodicidade da função seno, *sen*(10) = *sen*(10 - 2π) ≈ *sen*(3,716815). E como 3,716815 está mais perto de zero do que 10, provavelmente a aproximação será melhor usando x = 3,716815 e o mesmo valor de n. Vamos às contas:

$$(10) = (10 - 2\pi) \approx sen(3,716815) = 3,716815 -$$

$$\frac{3,716815^3}{3!} + \frac{3,716815^5}{5!} - \frac{3,716815^7}{7!} + \frac{3,716815^9}{9!} \approx -0,6397173.$$

Veja como já melhorou! Mas ainda podemos melhorar mais, pois 10 - 2π - 2π = -2,566371 está ainda mais perto de zero.

$$(10) = (10 - 2\pi - 2\pi) \approx (-2,5664) = -2,5664 -$$

$$\frac{(-2,5664)^3}{3!} + \frac{(-2,5664)^5}{5!} - \frac{(-2,5664)^7}{7!} + \frac{(-2,5664)^9}{9!} \approx -0.5447608$$

O exemplo anterior nos mostra que aproveitar a periodicidade da função seno e escolher valores próximos de zero para serem aplicados na Série de Taylor garante uma melhor convergência com o mesmo número de parcelas. Esse mecanismo será aplicado no pseudocódigo a seguir.

Critério de Parada

O critério de parada é o mesmo adotado na aproximação da exponencial: o algoritmo começa com n = 10 e depois incrementa n até encontrar duas aproximações consecutivas tais que a diferença é menor que o valor de delta definido pelo usuário.

Pseudocódigo

Entrada: x e delta.
Saída: uma aproximação para *sen*(*x*).
Nome: AproxSeno

```
1. Se x > pi, retorne AproxSeno(x - 2*pi,delta).
2. Se x < -pi, retorne AproxSeno(x + 2*pi,delta).
3. Calcule aprox =  x - (x^3)/(3!) + (x^5)/(5!)
- (x^7)/(7!) + ... + ((-1)^(10))*(x^(2*10 + 1))/
((2*10 + 1)!).
4. Faça n = 11.
5. Defina incr = ((-1)^(n))*(x^(2*n + 1))/((2*n +
1)!)
6. Faça aprox = aprox + incr.
7. Se |incr| < delta, retorne aprox.
8. Faça n = n + 1.
9. Volte para a linha 5.
```

Veja que a linha 5 poderia ser alterada para incr = -incr*(x^2)/(n*(n+1)). Isso tornaria o código mais eficiente, pois não seria necessário calcular o fatorial em cada iteração do programa. Mas para isso seria necessário garantir que o valor de incr foi iniciado em algum momento.

Exercícios

1. A partir da Série de Taylor em torno de $x = 0$ para a função $f(x) = e^x$ apresentada na aula teórica, encontre aproximações para e^1 e e^3 usando $n = 4$. Faça as contas na mão.

2. Implemente uma função que recebe como entrada $x \in R$ e $n \in N$ e retorna o polinômio

$$\sum_{i=0}^{n} \frac{x^i}{i!}$$

avaliado em x. Chame essa função de pol_exp(x,n). Em seguida digite o código a seguir para ver como esse polinômio se aproxima da função exponencial quando n cresce. Veja

também que a aproximação melhora quanto mais perto de 0 for o ponto avaliado ou quando maior for o valor de n.

```
plot(exp,-4,4)
grid()
segments(x0=0,y0=0,x1=0,y1=150,lty=2)
curve(pol_exp(x,n=2),add=T,col="violet")
curve(pol_exp(x,n=3),add=T,col="red")
curve(pol_exp(x,n=4),add=T,col="blue")
curve(pol_exp(x,n=5),add=T,col="green")
```

3. Implemente o pseudocódigo visto na aula teórica que recebe como entrada x e o valor de delta e retorna uma aproximação para e^x. Sugestão: indique um valor padrão para o argumento de entrada delta, delta = 0.001, por exemplo, de forma que o usuário não precise informar esse valor caso ele não queria. Em seguida, use a função implementada para encontrar aproximações para e, e^{-1}, e^3, $\sqrt{e}$ e $e^{7.3}$. Compare os resultados obtidos com os valores fornecidos pela função exp já pronta no R.

4. Refaça o item 3 de forma que agora a função implementada retorne, além da aproximação para e^x, o valor de n usado para realizar tal aproximação.

5. A partir Série de Taylor em torno de $x_0 = 1$ para a função $f(x) = ln(x)$, use $n = 3$ e encontre aproximações para $ln(1/10)$, $ln(3/5)$ e $ln(4)$. Faça as contas na mão. Qual aproximação você acha que é mais precisa, a aproximação para $ln(1/10)$ ou para $ln(3/5)$? Por quê?

6. Implemente uma função que recebe como entrada $x \in R$ e $n \in N$ e retorna o polinômio

$$\sum_{i=0}^{n} (-1)^{i} \frac{(x-1)^{i+1}}{i+1}$$

avaliado em x. Chame essa função de pol_ln(x,n). Em seguida digite o código a seguir para ver como esse polinômio se aproxima da função ln quando n cresce. Veja também que a aproximação melhora quanto mais perto de 1 for o ponto avaliado ou quando maior for o valor de n.

```
plot(log,0,4)
grid()
segments(x0=1,y0=-4,x1=1,y1=10,lty=2)
curve(pol_ln(x,n=2),add=T,col="violet")
curve(pol_ln(x,n=3),add=T,col="red")
curve(pol_ln(x,n=4),add=T,col="blue")
curve(pol_ln(x,n=5),add=T,col="green")
```

7. Implemente o pseudocódigo visto em sala de aula que recebe como entrada um número real positivo x e o valor de delta e retorna uma aproximação para $ln(x)$.

Sugestão: indique um valor padrão para o argumento de entrada delta, delta = 0.001, por exemplo, de forma que o usuário não precise informar esse valor caso ele não queria. Em seguida, use a função implementada e encontre aproximações para $ln(0.1)$, $ln(2)$, $ln(10)$ e $ln(3.8)$. Compare os resultados com os valores fornecidos pela função log já pronta do R.

8. Implemente uma função que retorna o logaritmo de x em qualquer base, ou seja, essa função recebe como entrada x, b e delta e retorna uma aproximação para $log_b(x)$. Para isso, lembre-se da seguinte propriedade:

$$log_b(x) = \frac{log_a(x)}{log_a(b)}, \text{ quaisquer que sejam } a, b, x \in R^+$$

Dica: essa nova função deve chamar a função implementada no Exercício 7 e usar a propriedade anterior considerando $a = e$. Sugestão: indique um valor padrão para o argumento de entrada delta, delta = 0.001, por exemplo, de forma que o usuário não precise informar esse valor caso ele não queria.

9. Implemente o pseudocódigo visto em sala de aula que recebe como entrada um número real positivo x e o valor de delta e retorna uma aproximação para *sen*(*x*). Aproveite a periodicidade da função e, a partir de uma chamada recursiva, simplifique o argumento até que ele esteja entre -π e π. Sugestão: indique um valor padrão para o argumento de entrada delta, delta = 0.001, por exemplo, de forma que o usuário não precise informar esse valor caso ele não queria. Em seguida use a função implementada e encontre aproximações para *sen*(2), *sen*(25), *sen*(50º) e *sen*(π/3). Compare os resultados obtidos com os valores fornecidos pela função sin do R.

10. Implemente agora uma função que recebe como entrada x e o valor de delta e retorna *cos*(*x*). Para isso perceba que a função cosseno é a função seno deslocada de π/2, ou seja,

$$cos(x) = sen\left(x + \frac{\pi}{2}\right).$$

Sugestão: indique um valor padrão para o argumento de entrada delta, delta=0.001, por exemplo, de forma que o usuário não precise informar esse valor caso ele não queria. Em seguida use a função implementada e encontre aproximações para *cos*(1), *cos*(54), *sen*(45º) e *sen*(π/3). Compare os resultados obtidos com os valores fornecidos pela função cos do R.

CAPÍTULO 10: RAÍZES DE FUNÇÕES REAIS

Encontrar raízes de funções tem diversas aplicações práticas, como por exemplo, encontrar aproximações para números irracionais ou achar máximo de funções. Nesta semana vamos estudar um método numérico para encontrar aproximações de raízes: o Método da Bisseção. Existem ainda outros métodos que não serão apresentados neste livro, como por exemplo, Método de Newton-Raphson, Método do Ponto Fixo e Método da Secante.

Conceitos Básicos

Primeiro vale lembrar o que é a raiz de uma função real.

Definição: $f : R \to R$ uma função real. Dizemos que $x \in R$ é raiz de f quando $f(x) = 0$.

O problema de encontrar raízes de uma função real muitas vezes é bem fácil. Sabemos, por exemplo, achar facilmente as raízes das funções $f(x) = x + 3$, $f(x) = x^2 - 4$, $f(x) = 1 - e^x$, $f(x) = ln(x + 1)$. Mas dependendo da expressão da função f encontrar suas raízes pode ser uma tarefa muito difícil, como por exemplo para as funções $f(x) = x^3 + 2x^2 - x + 1$, $f(x) = x + ln(x)$, $f(x) = e^x + x^2 - 2$. Nesses casos, o problema será resolvido de forma aproximada a partir de métodos numéricos, como veremos nessa semana.

A base do Método da Bisseção está no Teorema de Bolzano, apresentado a seguir.

Teorema de Bolzano: Seja f uma função contínua em um intervalo $[a, b]$, tal que $f(a)f(b) < 0$ (isto é, o sinal de $f(a)$ e $f(b)$ são diferentes). Então a função f possui pelo menos uma raiz no intervalo (a, b).

Veja que, segundo o teorema, conhecendo um intervalo (a, b) tal que $f(a)$ e $f(b)$ tenham sinais diferentes nos garante a existência de pelo menos uma raiz dentro do intervalo. Por isso o primeiro passo do método será a busca por um intervalo qualquer com essa característica.

A busca por esse intervalo pode ser feita de várias maneiras. Pode ser simplesmente por um "chute" inicial ou então podemos fazer uma análise gráfica. A análise gráfica tem a vantagem de que as vezes podemos saber exatamente quantas raízes a função tem. Veja a seguir dois exemplos em que o gráfico de f é usado para encontrar um intervalo (a, b) em que f possui pelo menos uma raiz.

Método da Bisseção

Para que o método da Bisseção seja aplicado é preciso primeiro conhecer um intervalo (a, b) tal que $f(a)$ e $f(b)$ tenham sinais opostos, isto é, tal que $f(a)f(b) < 0$. Dessa forma, de acordo com o Teorema de Bolzano, podemos garantir que existe pelo menos uma raiz dentro desse intervalo. O que o método vai fazer é retornar uma aproximação para uma dessas raízes.

A ideia do método é dividir o intervalo (a, b) ao meio e, usando o resultado do Teorema de Bolzano, decidir em qual dos dois lados a raiz está. Ou seja, começamos com (a, b) tal que $f(a)f(b) < 0$ e depois de um passo chegamos em dois outros intervalos: (a, m) e (m, b), sendo $m = (a + b) \div 2$ o ponto médio do intervalo. Veja que nesse momento podemos garantir que m é uma aproximação para a raiz de f com erro menor que $b - m$, que é o raio do intervalo (a, b).

Calculando o valor de $f(m)$ e comparando o seu sinal com os sinais de $f(a)$ e $f(b)$ podemos determinar em qual dos dois subintervalos a raiz está: se $f(a)f(m) < 0$ sabemos que a raiz está no intervalo (a, m), caso contrário, temos $f(b)f(m) < 0$ e a raiz está no intervalo (m, b). Veja que dessa forma chegamos em um intervalo menor tal que a função f avaliada nos pontos extremos desse intervalo tem sinais opostos.

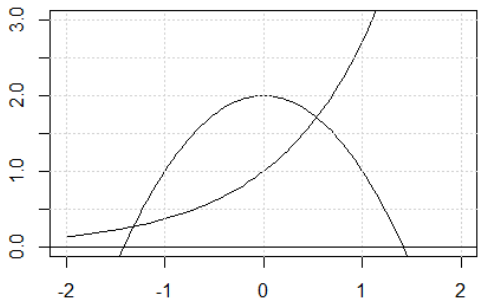
O processo é repetido e em cada iteração encontramos um intervalo e o ponto médio desse intervalo é uma aproximação para uma raiz de f com erro menor que o raio do intervalo.

Exemplo:

Seja $f(x) = e^x + x^2 - 2$. Queremos encontrar alguma raiz dessa função. Para iniciar o processo precisamos encontrar um intervalo (a, b) tal que $f(a)f(b) < 0$.

Veja que nesse caso é possível afirmar que $f(x) = g(x) - h(x)$, com $g(x) = e^x$ e $h(x) = 2 - x^2$. Logo, $f(x) = 0$ se e somente se $g(x) = h(x)$. Não sabemos ainda resolver a equação, mas sabemos esboçar os gráficos de g e h. As raízes de f serão as abcissas dos pontos de intercessão da curva de g com h.

Figura 1 — Gráficos das curvas $g(x) = e^x$ e $h(x) = 2 - x^2$



Logo, podemos afirmar que existe uma raiz de f no intervalo (-2, -1) e outra no intervalo (0, 1). Vamos procurar uma aproximação para a raiz dentro do intervalo (0, 1). Nesse exemplo, estou assumindo que sabemos calcular $f(x)$ para qualquer $x \in R$, o que pode ser feito assim:

```
f = function(x){
  exp(x) + x^2 - 2
}
```

ou, como aprendemos no capítulo anterior.

```
AproxExp = function(x,delta=0.00001){
  aprox = 1
  for(i in 1:10){
    aprox = aprox + (x^i)/factorial(i)
  }
  n = 11
  incr = (x^n)/factorial(n)
  repeat{
    aprox = aprox + incr
    if(abs(incr)<delta){
      return(aprox)
    }
    n = n + 1
    incr = incr*x/n
  }
}
f = function(x){
  AproxExp(x) + x^2 - 2
}
```

Veja que $f(0) < 0$ e $f(1) > 0$:

```
f(0)
## [1] -1
f(1)
## [1] 1.718282
```

Então podemos usar $a = 0$ e $b = 1$ para iniciar o Método da Bisseção. A partir de a e b calculamos o ponto mé-

dio: $m = (a + b) \div 2 = 1/2 = 0{,}5$. Essa pode ser considerada a nossa primeira estimativa para a raiz de f com erro menor que 0,5.

Vamos agora avaliar a função em m, ou seja, encontrar o valor de $f(m)$.

```
m = 0.5
f(m)
## [1] -0.1012787
```

Como $f(0{,}5) < 0$ e $f(1) > 0$, podemos afirmar que a raiz que buscamos está no intervalo (0,5, 1). Vamos agora encontrar o ponto médio desse intervalo, que será a nossa segunda aproximação: $m = (0{,}5 + 1) \div 2 = 0{,}75$. Veja que $m = 0{,}75$ é uma aproximação para a raiz de f com erro menor que o raio do intervalo (0,5, 1), isto é, com erro menor que $1 - 0{,}75 = 0{,}25$. O próximo passo é calcular $f(m)$ para saber em qual lado desse intervalo a raiz está, antes ou depois de m.

```
m = 0.75
f(m)
## [1] 0.6795
```

Como $f(0{,}75) > 0$ e $f(0{,}5) < 0$, sabemos que a raiz está no intervalo (0,5, 0,75). Vamos agora encontrar o ponto médio desse último intervalo, que será a nossa terceira aproximação, mais precisa que a segunda: $m = 0{,}5 + 0{,}752 = 0{,}625$. Veja que $m = 0{,}625$ é uma aproximação para a raiz com erro menor que o raio do intervalo (0,5, 0,75), isto é, com erro menor que $0{,}75 - 0{,}625 = 0{,}125$. O próximo passo é calcular $f(m)$ para saber em qual parte desse intervalo a raiz está, antes ou depois de m.

```
m = 0.625
f(m)
```

```
## [1] 0.258871
```

Como $f(0{,}625) > 0$ e $f(0{,}5) < 0$ sabemos que a raiz está no intervalo (0,5, 0,625). Vamos agora encontrar o ponto médio desse último intervalo, que será a nossa quarta aproximação: $m = (0{,}5 + 0{,}625) \div 2 = 0{,}5625$. Veja que m é uma aproximação para a raiz com erro menor que o raio do intervalo (0,5, 0,625), isto é, com erro menor que $0{,}625 - 0{,}5625 = 0{,}0625$. O próximo passo é calcular $f(m)$ para saber em qual parte desse intervalo a raiz está, antes ou depois de m.

```
m = 0.5625
f(m)
## [1] 0.07146091
```

Como $f(0{,}5625) > 0$ e $f(0{,}5) < 0$, sabemos que a raiz está no intervalo (0,5, 0,5625). Vamos agora encontrar o ponto médio desse último intervalo, que será a nossa próxima aproximação, mais precisa que a segunda: $m = (0{,}5 + 0{,}5625) \div 2 = 0{,}53125$. Veja que m é uma aproximação para a raiz com erro menor que o raio do intervalo (0,5, 0,5625), isto é, com erro menor que $0{,}5625 - 0{,}53125 = 0{,}03125$. O próximo passo é calcular $f(m)$ para saber em qual parte desse intervalo a raiz está, antes ou depois de m.

```
m =  0.53125
f(m)
## [1] -0.01671614
```

Como $f(0{,}53125) < 0$ e $f(0{,}5625) > 0$ sabemos que a raiz está no intervalo (0,53125, 0,5625). Vamos agora encontrar o ponto médio desse último intervalo, que será a nossa próxima aproximação, mais precisa que a segunda: $m = (0{,}53125 + 0{,}5625) \div 2 = 0{,}546875$. Veja que m é uma aproximação para a raiz com erro menor que o raio do in-

tervalo (0,53125, 0,5625), isto é, com erro menor que 0,5625 – 0,546875 = 0,015625.

E quando paramos? Já encontramos 5 aproximações, lembrando que cada valor de m encontrado foi uma aproximação. Como sabemos se já está bom?

Critério de Parada

O Método da Bisseção é um método que conseguimos controlar o erro da aproximação encontrada, como vimos no exemplo. Podemos repetir o processo até encontrar uma aproximação com um erro tão pequeno quanto a gente queira. Ou seja, o usuário indica um erro ε e quando o algoritmo chegar em um intervalo (a, b) de raio menor que ε, isto é $b - m < \varepsilon$, podemos afirmar que o ponto médio desse intervalo é uma aproximação para a raiz de f com erro menor que ε.

Pseudocódigo

Veja a seguir o pseudocódigo de uma função que encontra a raiz a partir do Método da Bisseção.

Entrada: a, b, f e e.
Saída: uma aproximação para uma raiz de f dentro do intervalo (a, b) com erro menor que e.
Nome: RaizBissecao

```
1. Se f(a)*f(b) >= 0, retorne erro.
2. Defina m = (a+b)/2.
3. Se f(m) = 0 ou |b - m| < e, retorne m.
4. Se f(a)*f(m) < 0, faça b = m e volte para a
linha 2.
5. Faça a = m e volte para a linha 2.
```

O pseudocódigo anterior será mais facilmente implementado se for usado o comando repeat. Mas acho que ainda mais simples é a sua versão recursiva.

Entrada: a, b, f e e.
Saída: uma aproximação para uma raiz de f dentro do intervalo (a, b) com erro menor que e.
Nome: RaizBissecaoRec

```
1. Se f(a)*f(b) >= 0, retorne erro.
2. Defina m = (a+b)/2.
3. Se f(m) = 0 ou |b - m| < e, retorne m.
4. Se f(a)*f(m) < 0, retorne
RaizBissecaoRec(a,m,f,e)
5. retorne RaizBissecaoRec(m,b,f,e)
```

Algumas vantagens desse método:

- método de fácil compreensão;
- se conseguimos um intervalo $[a, b]$ tal que $f(a)$ e $f(b)$ têm sinais opostos, temos a garantia de que o método converge para uma raiz de f;
- ao final do método sabemos que chegamos em uma aproximação com precisão ε, ou seja, o valor x fornecido pelo método é tal que $| x - \alpha | < \varepsilon$, em que α é a raiz procurada e desconhecida.

Algumas desvantagens desse método:

- é necessário conhecer um intervalo $[a, b]$ tal que $f(a)f(b) < 0$, às vezes esse intervalo nem existe;
- o processo de convergência não é dos mais rápidos.

Algumas Aplicações:

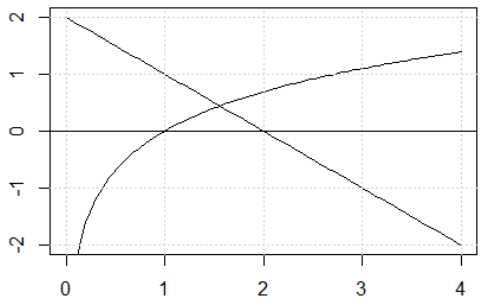
Os métodos para encontrar raiz de uma função também podem servir resolver outros problemas. Vejamos al-

guns exemplos. Para os exemplos a seguir, vamos supor já implementada no R a função RaizBissecaoRec.

Exemplo:

Encontre uma solução para a expressão $ln(x) = 2 - x$. Esse não é um problema direto de encontrar raiz, mas rapidamente pode ser transformado em um. Basta perceber que a solução dessa expressão é a raiz da função $f(x) = ln(x) - 2 + x$. E para solucionar esse problema o primeiro passo é encontrar valores de a e b, tais que $f(a)f(b) < 0$. Veja que como sabemos fazer um esboço de $g(x) = ln(x)$ e também de $h(x) = 2 - x$ fica mais fácil encontrar valores viáveis de a e b.

Figura 2 — Gráficos das curvas $g(x) = ln(x)$ e $h(x) = 2 - x$.

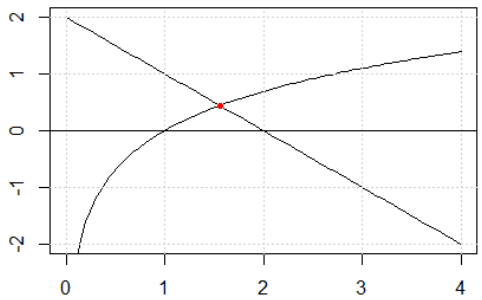


Assim podemos afirmar que a única raiz de f está dentro do intervalo (1, 2). Usando a função RaizBissecaoRec podemos encontrar o que procuramos:

```
(r = RaizBissecaoRec(1,2,f=function(x){log(x)-
2+x}))
## [1] 1.557145
log(r)-2+r
## [1] -7.887232e-07
log(r)
## [1] 0.4428541
```

```
2-r
## [1] 0.4428549
curve(log(x),ylim=c(-2,2),xlim=c(0,4),ylab="",xl
ab="")
curve(-x+2,add=T)
grid()
abline(h=0)
points(r,log(r),pch=20,col="red")
```

Figura 3 — Gráficos das curvas $g(x) = ln(x)$ e $h(x) = 2 - x$ e seu ponto de interseção



Exemplo:

No capítulo passado, vimos como avaliar algumas funções usando apenas operações básicas. Ou seja, sabemos encontrar uma aproximação para $ln(5)$, e^{-3} ou $sen(10)$ usando apenas lápis e papel. E se quisermos avaliar, usando apenas operações básicas, a função raiz quadrada, ou seja, encontrar o valor de

$$\sqrt{x_0}$$

para um dado $x_0 \geq 0$? Podemos usar o Método da Bisseção para isso.

Veja que para um dado x_0,

$$\sqrt{x_0}$$

é a raiz da função $f(x) = x^2 - x_0$. Ou seja, basta aplicar o Método da Bisseção nessa função. Precisamos então escolher quais valores de a e b começar. Veja que, supondo $x_0 \geq 0$, sempre podemos usar $a = 0$ e temos $f(a) = f(0) < 0$. Se $x_0 > 1$, podemos usar $b = x_0$ pois nesse caso $f(b) = f(x_0) = x_0^2 - x_0 = x_0(x_0 - 1) > 0$. Se $x_0 < 1$ usamos $b = 1$, pois $f(b) = f(1) = 1 - x_0 > 0$.

Vejamos então o pseudocódigo de uma função que recebe como entrada $x > 0$ e um erro e retorna uma aproximação para $\sqrt{x}$.

Entrada: x_0 e e.
Saída: uma aproximação para $\sqrt{x_0}$ com erro menor que e.
Nome: AproxRaiz

```
1. Se x0 < 0, retorne erro.
2. Se x0 = 1 ou x0 = 0, retorne x0.
3. Se x0 > 1, faça b = x0. Caso contrário faça b =
1.
4. retorna RaizBissecaoRec(0,b,f(x)=x^2-x0,e)
```

Veja como fica o código dessa função, supondo já feita a função RaizBissecaoRec.

```
AproxRaizQuad = function(x0,e=0.00000001){
  if(x0<0)
    stop("Erro")
  if( x0 == 1 || x0 == 0 )
    return(x0)
  if(x0>1){
    b = x0
  } else {
    b = 1
```

```
  }
  RaizBissecaoRec(0,b,f = function(x){x^2-x0},e)
}
AproxRaizQuad(4)
## [1] 2
AproxRaizQuad(2)
## [1] 1.414214
a = AproxRaizQuad(2)
a*a
## [1] 2
AproxRaizQuad(0.5)
## [1] 0.7071068
a = AproxRaizQuad(0.5)
a*a
## [1] 0.5
```

Exercícios

1. Implemente o pseudocódigo visto em sala de aula RaizBissecao, de forma não recursiva, que recebe como entrada os valores a, b, f, e e retorna uma aproximação para uma raiz de f no intervalo (a, b) com erro menor que e. Para testar se a sua função está funcionando corretamente verifique se a saída do comando RaizBissecao(0,3,f=function(x){x-1},0.000001) é um número muito perto de 1.

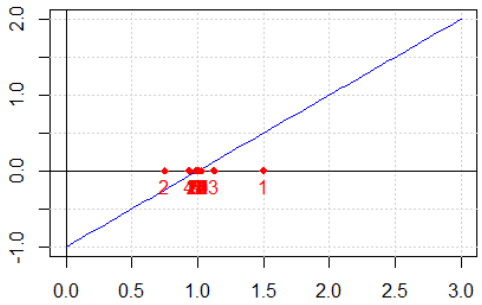
2. Implemente o pseudocódigo visto em sala de aula RaizBissecaoRec, de forma recursiva, que recebe como entrada os valores a, b, f, e e retorna uma aproximação para uma raiz de f no intervalo (a, b) com erro menor que e. Para

testar se a sua função está funcionando corretamente verifique se a saída do comando RaizBissecaoRec(0,3,f=function(x){x-1},0.000001) é um número muito perto de 1.

3. Vamos ver com mais detalhes como esse método funciona. Para isso modifique uma das funções implementadas nos Exercícios anteriores de forma que a função retorne um vetor com todos os pontos médios dos intervalos encontrados durante o método (todas as aproximações), e não somente a última com erro menor que *e*. Vamos chamar essa função de RaizBissecaoArray. Tente reproduzir o código e a saída a seguir. Amplie o seu gráfico e veja se você entende porque os valores em vermelho foram os retornados pelo método.

```
saida = RaizBissecaoArray(0,3,f=function(x){x-
1},0.000001)
plot(0,xlab="x",ylab="f(x)",xlim=c(0,3),ylim=c(-
1,2),type="n")
grid()
abline(h=0)
abline(v=0)
curve(x-1,col="blue",add=T)
for(i in 1:length(saida)){
  points(saida[i],0,col="red",pch=18,cex.lab=1)
  text(saida[i],-0.2,i,col="red")
}
```

Figura 4 — Exemplo de gráfico gerado pelo código do Exercício 3



4. Nesse exercício o Método da Bisseção será usado para encontrar a raiz de $f(x) = 2 - x^5 - x^3 - x$. Veja que esse é um polinômio de grau 5 com apenas 1 raiz real. Podemos afirmar isso pois essa é uma função decrescente (aqueles que estudaram derivada podem verificar que a sua derivada é negativa sempre) e por isso cruza uma única vez o eixo x.
   a. Determine valores de a e b para aplicar o Método da Bisseção nessa função.
   b. Use a função do Exercício 1 ou a do Exercício 2 (ou as duas) para encontrar uma aproximação para a raiz da função no intervalo (a, b).
   c. Verifique se o valor encontrado está correto. Como fazer isso?

5. Nesse exercício o Método da Bisseção será usado para encontrar a(s) raiz(es) de $f(x) = e^x - 1/x$.
   a. Primeiro descubra quantas raízes tem essa função. Para isso você pode analisar os gráficos de e^x e $1/x$ e encontrar quantas vezes eles se cruzam.
   b. Ainda analisando os gráficos das funções, determine valores de a e b para aplicar o Método da Bisseção.
   c. Use a função do Exercício 1 ou a do Exercício 2 (ou as duas) para encontrar uma aproximação para a raiz da função no intervalo (a, b).

d. Verifique se o valor encontrado está correto. Como fazer isso?

6. Nesse exercício o Método da Bisseção será usado para encontrar uma aproximação para o número irracional $\sqrt{3}$.

a. Escolha uma função f que tenha uma raiz em $\sqrt{3}$. Não vale $f(x) = x - \sqrt{3}$, pois estamos supondo que não sabemos calcular $\sqrt{3}$. A função deve conter apenas números racionais.
b. Escolha valores de a e b de forma que você garanta que f do item anterior tenha uma raiz no intervalo (a, b) e que essa raiz seja $\sqrt{3}$.
c. Use as respostas dos itens anteriores e a função do Exercício 1 ou a do Exercício 2 (ou as duas) para encontrar uma aproximação para $\sqrt{3}$.
d. Verificar se a sua aproximação está correta. Como fazer isso?

7. Nesse exercício o Método da Bisseção será usado para encontrar aproximações para todas as soluções da equação

$$\frac{x^3}{6} = 1 - x^2 - x$$

a. Primeiro descubra quantas soluções tem essa função. Como você pode fazer isso?
b. Defina uma função f para a qual as raízes de f são as soluções da equação.
c. Para cada uma das raízes de f: determine valores de a e b para os quais você garante que a raiz em questão está dentro do intervalo (a, b).
d. Para cada uma das raízes de f: use a função do Exercício 1 ou a do Exercício 2 (ou as duas) e os valores de a e b para encontrar uma aproximação para a raiz de f no intervalo (a, b).
e. Para cada uma das raízes de f: verifique se o valor encontrado está correto. Como fazer isso?

8. Encontre uma aproximação para π a partir do Método da Bisseção. Veja que π é uma raiz da função $sen(x)$. Quais os valores de a e b você pode escolher para aplicar o método? π

9. Implemente uma função que recebe como entrada um número $x \geq 0$ e retorna $\sqrt{x}$. Tente primeiro fazer sem olhar a solução já feita na parte teórica. Você vai precisar chamar a função já implementada no Exercício 1 ou a do Exercício 2. Depois, encontre aproximações para $\sqrt{2}$, $\sqrt{3}$ e $\sqrt{5}$. Compare o resultado de $\sqrt{3}$ com aquele encontrado no Exercício 6.

10. Vamos juntar as funções implementadas nessa lista com as da lista anterior. Combinando funções já feitas, escreva uma função que recebe como entrada um valor $x > 0$ e retorna $f(x) = e^{\sqrt{x}}$. Verifique se o valor encontrado está correto. Como fazer isso?

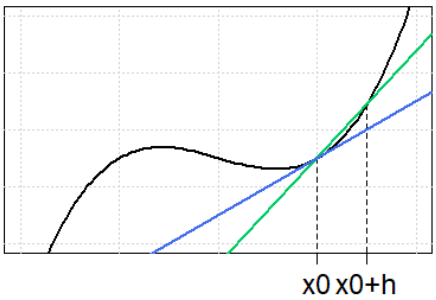
CAPÍTULO 11: DERIVAÇÃO NUMÉRICA

Neste capítulo, vamos estudar como usar o computador para encontrar uma aproximação para o coeficiente angular da reta tangente à f no ponto x_0, denominada $f'(x_0)$. Para isso vamos supor que sabemos avaliar f em uma vizinhança de x_0, ou pelo menos uma aproximação para isso.

Primeiro Método

Já foi estudado em cálculo, ou será em breve, que a derivada de uma função f em x_0, representada por $f'(x_0)$, é o coeficiente angular da reta tangente à f em x_0.

Figura 5 — Ilustração do cálculo da derivada de f em x_0 pelo primeiro método



Na figura anterior, a reta em azul indica a reta tangente à função f no ponto x_0, já a reta em verde é uma secante, que corta a função f nos pontos $(x_0, f(x_0))$ e $(x_0 + h, f(x_0 + h))$. Como conhecemos dois pontos de uma reta secante, é possível calcular o seu coeficiente angular, que é a tangente do ângulo entre a reta secante e o eixo horizontal. Lembrando, a tangente de um ângulo é a razão entre o cateto oposto e o cateto adjacente. Então,

$$a = \frac{f(x_0+h)-f(x_0)}{h}$$

Se escolhemos h bem pequeno, $x_0 \approx x_0 + h$ e a reta secante será bem próxima da reta tangente. Podemos então afirmar que:

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

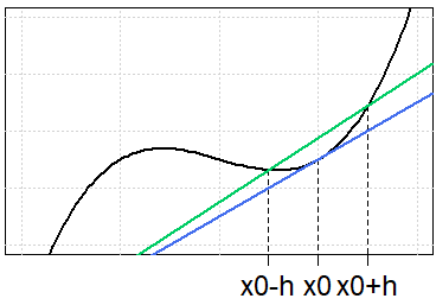
O que estamos fazendo nesse caso é aproximando o coeficiente angular da reta tangente à f no ponto x_0 (reta em azul) pelo coeficiente angular da reta secante (reta em verde). Veja que quanto menor o valor de h mais próxima está uma reta da outra, logo mais próximo está o coeficiente angular da reta secante do coeficiente angular da reta tangente. E por isso podemos escrever a equação abaixo, que vale qualquer x_0 no domínio de f:

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

Segundo Método

Vamos seguir somente com o exemplo da reta tangente à f no ponto x_0. Uma outra alternativa para calcular uma aproximação para $f'(x_0)$ é usar a reta secante que passa pelos pontos $(x_0 - h, f(x_0 - h))$ e $(x_0 + h, f(x_0 + h))$, reta em verde na figura a seguir.

Figura 6 — Ilustração do cálculo da derivada de f em x_0 pelo segundo método



Nesse caso o coeficiente angular da reta secante, que é a tangente do ângulo dessa reta com o eixo horizontal, pode ser escrito por:

$$a = \frac{f(x_0+h)-f(x_0-h)}{2h}$$

Da mesma forma que no método anterior, quanto menor o valor de h mais perto a reta secante (verde) está da reta tangente (azul), e por isso o coeficiente angular da reta tangente à f no ponto x_0 também ser escrita pelo limite:

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h)-f(x_0-h)}{2h}$$

Se escolhemos h bem pequeno, podemos afirmar que:

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$$

Pseudocódigo

Não serão apresentadas as contas, mas é possível mostrar que o Segundo Método converge mais rápido, isto é, para valores de h pequenos ele retornar aproximações mais precisas do que o Primeiro Método. Apesar disso, o primeiro método também converge. Mas, devido à maior eficiência do segundo, vamos apresentar o pseudocódigo somente para ele.

Para aproximar $f'(x_0)$, vamos começar com um $h > 0$. A partir desse valor de h calculamos uma primeira aproximação. Depois diminuímos o valor de h, por exemplo, dividindo-o por 2, e calculamos uma nova aproximação. Veja que as aproximações para $f'(x_0)$ são dadas pela expressão:

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$$

Realizamos esse procedimento diversas vezes até que a diferença entre duas aproximações consecutivas seja bem pequena, ou melhor, seja menor que o valor de δ determinado pelo usuário. Vale destacar que esse método

funciona se x_0 está dentro de um intervalo aberto contido no domínio de *f*.

Entrada: x_0, *f* e delta.
Saída: uma aproximação para $f'(x_0)$.
Nome: DerivadaNumérica

```
1. Se x0 não pertence à um intervalo aberto do
domínio de f, pare e retorna uma mensagem de erro.
2. Defina h = 1.
3. Se (x0 + h) ou (x0 - h) não pertencem ao domínio
de f, faz h = h/2 e repete a linha 3.
4. Calcule d1 = (f(x0 + h) - f(x0 - h))/(2h).
5. Atualize o valor de h: h = h/2.
6. Calcule d2 = (f(x0 + h) - f(x0 - h))/(2h).
7. Se |d1 - d2| < delta, retorna d2.
8. Caso contrário, faça d1 = d2 e volte para a
linha 5.
```

Dica: Talvez fique mais simples se o algoritmo for implementado usando o repeat.

Veja agora uma possibilidade de realizar o algoritmo de forma recursiva. Nesse caso vai ser bem mais simples se *h* também for passado como entrada.

Entrada: x_0, *f*, delta e $h = 1$.
Saída: uma aproximação para $f'(x_0)$.
Nome: DerivadaNuméricaRec

```
1. Se x0 não pertence à um intervalo aberto do
domínio de f, pare e retorna uma mensagem de erro.
2. Se (x0 + h) ou (x0 - h) não pertencem ao domínio
de f, retorna DerivadaNuméricaRec(x0,f,delta,h/2).
```

```
3. Calcule d1 = (f(x0 + h) - f(x0 - h))/(2h).
4. Atualize o valor de h: h = h/2.
5. Calcule d2 = (f(x0 + h) - f(x0 - h))/(2h).
6. Se |d1 - d2| < delta, retorna d2.
7. Caso contrário, retorna
DerivadaNumericaRec(x0,f,delta,h).
```

Exercícios

Para os exercícios a seguir, considere $f'(x_0)$ = coeficiente angular da reta tangente à f no ponto x_0.

1. Vamos agora implementar o método visto em sala de aula. Para isso considere a função

$$f(x) = \frac{1}{x^2+1}$$

   a. Qual é o domínio da função f?
   b. Implemente uma função que recebe como entrada $x_0 \in Dom(f)$ e o valor de δ e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.
   c. Vamos implementar agora o algoritmo de forma recursiva. Para facilitar considere h um argumento de entrada da sua função. Dessa forma, implemente uma função recursiva que recebe como entrada $x_0 \in Dom(f)$, o valor do δ e h, e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.
   d. Vamos verificar se as funções estão retornando o valor correto.
      I. A partir das duas funções implementadas, encontre aproximações para $f'(0)$, $f'(-1/5)$ e $f'(1/3)$ e verifique se as respostas estão parecidas. Se estiverem diferentes, parece que algo está estranho.

II. Escolha uma função implementada e um dos valores de x_0, 0, -1/5 ou 1/3, ou até todos, e plote na mesma janela o gráfico de f e a reta cujo coeficiente angular seja $f'(x_0)$ e passe pelo ponto $(x_0, f(x_0))$. Você consegue encontrar a equação dessa reta, só existe uma com essas características. Verifique se a reta de fato parece ser a reta tangente à f no ponto x_0.

2. Considere agora a função $f(x) = ln(x^2 + x - 2)$.
   a. Qual é o domínio da função f?
   b. Implemente uma função que recebe como entrada $x_0 \in Dom(f)$ e o valor de δ, e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.
   c. Vamos implementar agora o algoritmo de forma recursiva. Para facilitar considere novamente h um argumento de entrada da sua função. Dessa forma, implemente uma função recursiva que recebe como entrada $x_0 \in Dom(f)$, o valor de δ e h, e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.
   d. Vamos verificar se as funções estão retornando o valor correto.
      I. A partir das duas funções implementadas, encontre aproximações para $f'(3)$, $f'(-5/2)$ e $f'(4/3)$ e verifique se as respostas estão parecidas. Se estiverem diferentes, parece que algo está estranho.
      II. Escolha uma função implementada e um dos valores de x_0, 0, -1/5 ou 1/3, ou até todos, e plote na mesma janela o gráfico de f e a reta cujo coeficiente angular seja $f'(x_0)$ e passe pelo ponto $(x_0, f(x_0))$. Você consegue encontrar a equação dessa reta, só existe uma com essas características. Verifique se a reta de fato parece ser a reta tangente à f no ponto x_0.

3. Considere agora

$$f(x) = e^{-x/3}\left(1 + \frac{x}{x^2+1}\right) - 1$$

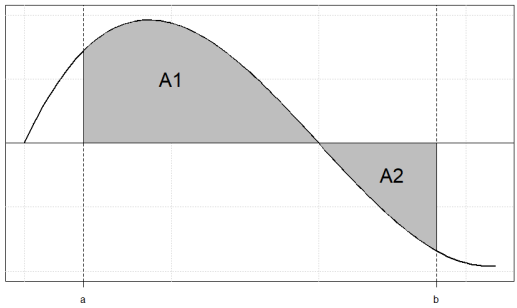
a. Qual é o domínio da função *f*?
b. Primeiro implemente uma função que recebe como entrada *x* e retorna uma aproximação para *f*(*x*). Vamos chamar essa função de f .
c. Implemente agora uma função que recebe como entrada *x* e retorna uma aproximação para $f'(x)$ considerando $\delta = 10^{-3}$. Vamos chamar essa função de df.
d. Nosso objetivo agora é usar o método da bisseção para encontrar os pontos de máximo e mínimo locais de *f*. Veja que esses pontos são os pontos x_0 tais que $f'(x_0) = 0$. Para isso siga os itens a seguir.
    I. Digite plot(f,xlim=c(-3,5)) e plot(df,xlim=c(-3,5)); abline(h=0) e, comparando os dois gráficos, veja onde estão os pontos x_0 tais que $f'(x_0) = 0$. Para encontrais aproximações para tais pontos vamos buscar as raízes da função df.
    II. Use o método da bisseção para encontrar uma aproximação para o mínimo local de *f*. A partir dos gráficos escolha valores para *a* e *b* de forma a garantir que o método converge para o mínimo local. Faça a sua função de forma que ela chame df já implementada.
    III. Use o método da bisseção para encontrar uma aproximação para o máximo local de *f*. A partir dos gráficos escolha valores para *a* e *b* de forma a garantir que o método converge para o máximo local. Faça a sua função de forma que ela chame df já implementada.
    IV. Vamos testar se deu certo. Guarde no objeto xmin a aproximação para o ponto de mínimo local e no objeto xmax a aproximação para o ponto de máximo local de *f*. Agora digite a seguinte sequência de comandos e discuta o gráfico gerado.

```
plot(df,xlim=c(-3,5))
abline(h=0)
segments(x0=xmin,y0=2,x1=xmin,y1=-2,lty=2)
points(xmin,0,pch=19,cex=1.2)
segments(x0=xmax,y0=1,x1=xmax,y1=-1,lty=2)
points(xmax,0,pch=19,cex=1.2)
```

CAPÍTULO 12: INTEGRAÇÃO NUMÉRICA

Neste capítulo, veremos um método iterativo que, a partir de uma função conhecida f e de valores $a, b \in D(f)$, encontra um valor aproximado para a área entre a curva de f e o eixo horizontal limitada no intervalo $[a, b]$, representada por cinza na figura a seguir.

Figura 7 — Ilustração para área entre a curva de f e o eixo horizontal, limitada no intervalo $[a, b]$



Dependendo da função f, essa conta pode ser feita de forma precisa a partir de integrais definidas, como vocês aprenderão em cálculo. Mas algumas vezes isso não é possível, ou seja, para algumas funções não é possível encontrar o valor dessa área de forma exata, mesmo usando os recursos de cálculo. Nesses casos a alternativa é encontrar uma aproximação a partir de métodos numéricos, como veremos nesse capítulo.

Antes de seguirmos com a ideia por trás dos métodos vale destacar que nesse contexto a área entre a curva de f e o eixo horizontal, a qual queremos encontrar, considera área anterior do eixo como positiva e área a seguir como negativa. Dessa forma, considerando a figura anterior, a

área total entre a curva de f e o eixo, denominada A, é $A = A1 - A2$, $A_1 > 0$ e $A_2 > 0$, em que $A1$ e $A2$ são os valores de área, ilustrados na mesma figura, ambos positivos.

Aproximação por retângulos e trapézios

Para fazer a aproximação por retângulos, primeiro vamos dividirmos o intervalo (a, b) em n subintervalos de mesmo tamanho. O tamanho de cada subintervalo será

$$\delta = \frac{b-a}{n}$$

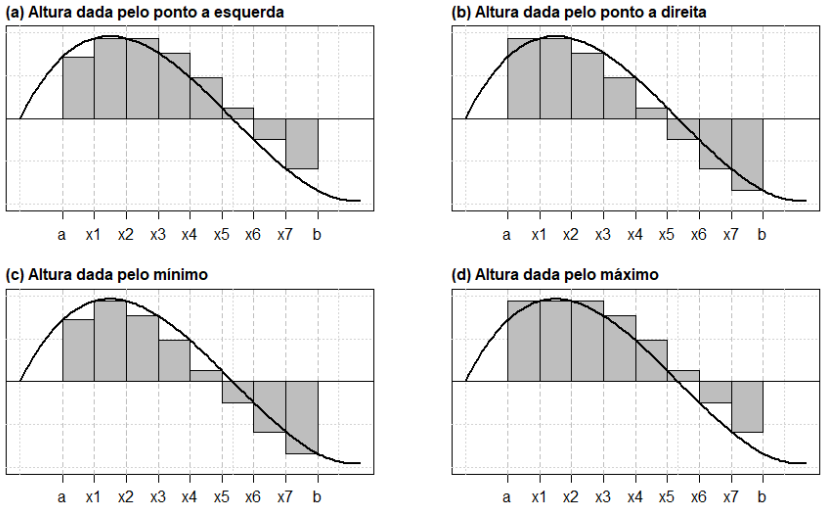
e os subintervalos são definidos pelos segmentos $[x_{i-1}, x_i]$, com $i = 1, ..., n$ e $x_0 = a$, $x_1 = a + \delta$, $x_2 = x_1 + \delta$, ..., $x_n = b$.

Para cada subintervalo $[x_{i-1}, x_i]$ vamos definir um retângulo cuja base é o próprio subintervalo e altura pode ser definida de diferentes formas, serão apresentadas 4 maneiras para fazer isso.

Primeiro considere que a altura é o valor de f avaliado no ponto mais à esquerda desse subintervalo, nesse caso a altura do retângulo de base $[x_{i-1}, x_i]$ é $f(x_{i-1})$ (figura (a) a seguir). Também podemos definir a altura do retângulo como o valor de f avaliado no ponto mais à direita desse subintervalo, nesse caso a altura do retângulo de base $[x_{i-1}, x_i]$ é $f(x_i)$ (figura (b) a seguir).

Podemos definir também a altura do retângulo de base $[x_{i-1}, x_i]$ pelo maior ou pelo menor valor entre $f(x_{i-1})$ e $f(x_i)$. Nesse caso a altura do retângulo de base $[x_{i-1}, x_i]$ pode ser o $min(f(x_{i-1}), f(x_i))$ (figura (c) a seguir) ou pode ser o $max(f(x_{i-1}), f(x_i))$ (figura (d) a seguir). A figura a seguir exemplifica essas quatro diferentes possibilidades.

Figura 8 — Quatro diferentes possibilidades para a aproximação por retângulos



Veja que podemos aproximar a área entre a curva de f e o eixo horizontal pela soma das áreas dos n retângulos. Veja também que quanto maior o número de retângulos, ou seja, quanto maior o valor de n, mais próximo está essa aproximação do real valor da área.

Uma observação interessante é que se definirmos a altura do retângulo de base $[x_{i-1}, x_i]$ como o menor valor entre $f(x_{i-1})$ e $f(x_i)$ (figura (c) acima), então a soma das áreas dos retângulos será sempre menor que o real valor da área. Dessa forma, dizemos que a soma das áreas desses retângulos, denominada $\hat{A}_I$, é um limite inferior para o real valor da área A, pois $\hat{A}_I \leq A$.

Por outro lado, se definirmos a altura do retângulo de base $[x_{i-1}, x_i]$ como o maior valor entre $f(x_{i-1})$ e $f(x_i)$ (figura (d)), então a soma das áreas dos retângulos será sempre maior que o real valor da área. Dessa forma, dizemos que a soma das áreas desses retângulos, denominada $\hat{A}_S$, é um limite superior para o real valor da área A, pois $\hat{A}_S \leq A$.

Juntando as duas ideias, podemos afirmar que $\hat{A}_L \leq A \leq \hat{A}_U$. Dessa forma, escolher como aproximação para o

valor de A a média entre $\hat{A}_I$ e $\hat{A}_S$ nos possibilita controlar o erro da aproximação. Como $\hat{A}_I \leq A \leq \hat{A}_S$, $A \in [\hat{A}_I, \hat{A}_S]$ e por isso o ponto médio desse intervalo, que é

$$\frac{\hat{A}_S + \hat{A}_I}{2}$$

dista de A no máximo o valor do raio desse intervalo, que é

$$\frac{\hat{A}_U - \hat{A}_L}{2}$$

Ou seja,

$$\left| A - \frac{\hat{A}_S + \hat{A}_I}{2} \right| < \frac{\hat{A}_S - \hat{A}_I}{2}$$

Também é possível mostrar que a média desses dois valores coincide com a aproximação caso escolhêssemos usar a área dada pelo trapézio de base $[x_{i-1}, x_i]$ e alturas $f(x_{i-1})$ e $f(x_i)$. Veja a demonstração de que para um subintervalo qualquer de base $[x_{i-1}, x_i]$, a média das áreas dos retângulos com alturas $max\{f(x_{i-1}), f(x_i)\}$ e $min\{f(x_{i-1}), f(x_i)\}$ é igual a área do trapézio de base $[x_{i-1}, x_i]$ e alturas $f(x_{i-1})$ e $f(x_i)$.

Sejam

$$\hat{A}_{I_i} = \delta min\left(f(x_{i-1}), f(x_i)\right) \text{ e } \hat{A}_{S_i} = \delta max\left(f(x_{i-1}), f(x_i)\right)$$

as áreas dos i-ésimos retângulos abaixo e acima da curva de f, respectivamente. Veja que o valor médio entre essas duas áreas pode ser calculado da seguinte maneira:

$$\frac{\hat{A}_{I_i} + \hat{A}_{S_i}}{2} = \frac{1}{2}\left(\delta\left(f\left(x_{i-1}\right), f\left(x_i\right)\right) + \delta\left(f\left(x_{i-1}\right), f\left(x_i\right)\right)\right) =$$

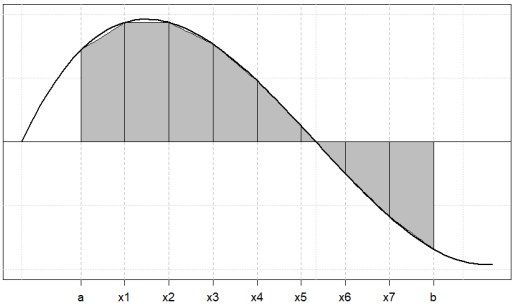
$$\frac{\delta}{2}\left(\left(f\left(x_{i-1}\right), f\left(x_i\right)\right) + \left(f\left(x_{i-1}\right), f\left(x_i\right)\right)\right) = \frac{\delta}{2}\left(f\left(x_{i-1}\right) + f\left(\ _i\right)\right)$$

O valor $\delta(f(x_{i-1}) + f(x_i))/2$ é exatamente a área do trapézio de base $[x_{i-1}, x_i]$ e alturas $f(x_{i-1})$ e $f(x_i)$.

Veja na figura a seguir que, para o mesmo valor de n, no caso do exemplo $n = 8$, a aproximação para A dada pela

área dos trapézios parece ser bem melhor que a aproximação pela soma das áreas dos retângulos, apresentados na figura anterior.

Figura 9 — Aproximação por trapézios



Critério de Parada

Como já foi discutido, vamos usar a área dos trapézios para aproximar o valor de A. Essa área será calculada a partir do valor médio das áreas dos retângulos com altura máxima e mínima. Precisamos então escolher o valor de n, números de retângulos, e calcular a soma das áreas dos retângulos. Mas como determinar o valor de n que irá fornecer uma aproximação relativamente boa?

Veja que quanto mais subdivisões tem o intervalo [a, b], mais precisa será a aproximação. E para esse caso, temos o controle do erro. Então, passado como entrada um valor de erro ε, vamos aumentando o número de retângulos, começando com $n = 100$, por exemplo, até que o erro da aproximação seja menor que ε, isto é, até que

$$\frac{\hat{A}_S - \hat{A}_I}{2} < \varepsilon \text{ ou, o que é equivalente, } (\hat{A}_S - \hat{A}_I) < 2\varepsilon$$

Quando isso acontecer podemos afirmar que o ponto médio entre $\hat{A}_I$ e $\hat{A}_S$, definido por

$$\frac{\hat{A}_S+\hat{A}_I}{2},$$

é uma aproximação para a área A com erro menor que ε.

Pseudocódigo

Entrada: *a*, *b*, *f* e *e*.
Saída: uma aproximação para a área entre a curva de *f* e o eixo horizontal, limitada pelo intervalor [*a*, *b*], com erro menor que *e*.
Nome: IntegralNumérica

```
1. Se o intervalo [a,b] não estiver contido no
domínio de f, pare e retorne erro.
2. Defina n = 100.
3. Defina d = (b-a)/n.
4. Defina A_S = 0 e A_I = 0.
5. Inicie x = a.
6. Faça A_S = A_S + d*max(f(x),f(x + d)).
7. Faça A_I = A_I + d*min(f(x),f(x + d)).
8. Incremente x = x + d.
9. Se x < b, volte para a linha 6.
10. Se (A_S - A_I) < 2*e, retorne (A_S + A_I)/2.
11. Caso contrário faça n = n+10 e volte para a
linha 3.
```

Podemos também fazer de forma recursiva, mas nesse caso o valor de *n* tem que ser um parâmetro de entrada.

Entrada: *a*, *b*, *f*, *e* e *n* = 100.

Saída: uma aproximação para a área entre a curva de f e o eixo horizontal, limitada pelo intervalor $[a, b]$, com erro menor que e.
Nome: IntegralNuméricaRec

```
1. Se o intervalo [a,b] não estiver contido no
domínio de f, pare e retorne erro.
2. Defina d = (b-a)/n.
3. Defina A_S = 0 e A_I = 0.
4. Inicie x = a.
5. Faça A_S = A_S + d*max(f(x),f(x + d)).
6. Faça A_I = A_I + d*min(f(x),f(x + d)).
7. Incremente x = x + d.
8. Se x < b, volte para a linha 5.
9. Se (A_S - A_I) < 2*e, retorne (A_S + A_I)/2.
10. Caso contrário retorne
IntegralNumericaRec(a,b,f,e,n+10).
```

Exercícios

1. Primeiro vamos aplicar o método em uma função que sabemos fazer as contas na mão, usando apenas áreas de figuras geométricas. Seja $f(x) = |x - 3| - 2$.
   a. Na mão, sem usar o computador, faça um esboço do gráfico de f e hachure a área entre a curva de f e o eixo horizontal, limitada no intervalo $[0, 5]$.
   b. Ainda na mão, calcule o valor da área hachurada. Lembre-se de considerar sinal positivo para área acima do eixo e sinal negativo para área abaixo dele. Esse é o valor que queremos aproximar.

c. Implemente uma função que recebe como entrada o valor n e retorna uma aproximação para a área hachurada dada pela soma das áreas dos n retângulos. Escolha como quiser a definição da altura dos retângulos: valor da função no ponto à esquerda; valor da função no ponto à direita, maior valor da função; menor valor da função. Obs.: Nessa questão ainda não é para fazer o algoritmo visto em sala de aula, que retorna a aproximação dada pela média ente a soma das áreas dos retângulos acima e abaixo da curva de f.
d. Usando a função implementada no item anterior, encontre uma aproximação para a área hachurada com: (i) 50 retângulos; (ii) 100 retângulos; (iii) 150 retângulos. Compare os resultados das 3 aproximações obtidas com o valor exato para a área encontrado no item b.

2. Implemente uma função denominada IntegralDominioReal que recebe como entrada uma função f com domínio real, valores a e b e o valor erro ε. Essa função retorna uma aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo $[a, b]$, com erro menor que ε. A sua implementação será testada nas questões a seguir. Veja que como a função tem domínio real, não é preciso se preocupar em verificar se os pontos estão no domínio de f.

3. Agora vamos aplicar a implementação feita no Exercício 2 para encontrar aproximações de áreas de duas funções de domínio real.

a. Seja $f(x) = x^2 - x - 1$. Encontre uma aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo [-1, 1], com erro menor que 0,005. Depois de encontrada a aproximação compare o resultado obtido com o valor exato, que é -4/3.
b. Seja $g(x) = xe^{x^2}$. Encontre uma aproximação para a área entre a curva de g e o eixo horizontal, limitada no intervalo [0,2], com erro menor que 0,01. Depois

de encontrada a aproximação compare o resultado obtido com o valor exato, que é $(e^4 - 1)/2$. Obs.: Esse item pode levar tempo para rodar, no meu computador ele demorou mais de 1 minuto para terminar o método. Tenha paciência.

4. Modifique a função implementada no Exercício 2 de forma que ela passe a retornar uma lista com dois objetos. O primeiro objeto da lista é a aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo $[a, b]$, com erro menor que ε. O segundo objeto da lista é o número de retângulos usados para conseguir essa aproximação, isto é, o valor final de n.

5. Usando a função implementada no Exercício 4, encontre o número de retângulos usados para encontrar as aproximações dos dois itens do Exercício 3.

6. Implemente o algoritmo visto em sala de aula que recebe como entrada uma função f com domínio R^+, valores a e b e o valor erro ε. Essa função retorna uma aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo $[a, b]$, com erro menor que ε. A sua implementação será testada nas questões a seguir. Veja que como a função não tem domínio real, é preciso se preocupar em verificar se os pontos estão no domínio de f.

7. Agora vamos aplicar a implementação feita no Exercício 6 para encontrar aproximações de áreas de duas funções de domínio R^+.

a. Seja $f(x) = \sqrt{x^2 + 3}$. Encontre uma aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo [1, 2], com erro menor que 0,0001. Depois de encontrada a aproximação compare o resultado obtido com o valor dado pela função integrate do R, que é 2,301763.

b. Seja $g(x) = \frac{\sqrt{|ln(x)|}}{x}$. Encontre uma aproximação para a área entre a curva de *g* e o eixo horizontal, limitada no intervalo [1/2, 3/2], com erro menor que 0,0001. Depois de encontrada a aproximação compare o resultado obtido com o valor dado pela integração exata, que é 0,5568449.

8. Seja $h(x) = ln(x - 1)$.
   a. Qual é o domínio da função *h*?
   b. Implemente uma função que recebe como entrada os valores de *a* e *b*, e o valor erro ε e retorna uma aproximação para a área entre a curva de *h* e o eixo horizontal, limitada no intervalo [*a*, *b*], com erro menor que ε.
   c. Encontre uma aproximação para a área entre a curva de *h* e o eixo horizontal, limitada no intervalo [3, 5], com erro menor que 0,0001. Depois de encontrada a aproximação compare o resultado obtido com o valor dado pela integração exata, que é 2,158883.
   d. E se quisermos uma aproximação para a área entre a curva de *h* e o eixo horizontal, limitada no intervalo [1, 2]? A função do item (b) serve para isso? Se for preciso, faça algum ajuste nela para encontrar o valor dessa área. Depois compare o resultado obtido com o valor dado pela integração exata, que é -1. Dica: Para eu conseguir uma aproximação com erro menor que 0,001, precisei de $n = 4.168$, então vai precisar de um *n* grande e se incrementar pouco em cada iteração o programa demora muito para rodar.

APÊNDICE

Gabarito dos Exercícios

Os gabaritos dos exercícios propostos em cada capítulo estão disponíveis nos *links* a seguir.

Capítulo 1: *https://rpubs.com/jessicakubrusly/gabarito-cap-1-objetos-e-classes*

Capítulo 2: *https://rpubs.com/jessicakubrusly/gabarito-cap-2-controle-de-fluxo*

Capítulo 3: *https://rpubs.com/jessicakubrusly/gabarito-cap-3-funcoes-e-var-locais*

Capítulo 4: *https://rpubs.com/jessicakubrusly/gabarito-cap-4-algoritmos-para-cal-est*

Capítulo 5: *https://rpubs.com/jessicakubrusly/gabarito-cap-5-algoritmos-matrizes*

Capítulo 6: *https://rpubs.com/jessicakubrusly/gabarito-cap-6-algoritmos-recursivos-simples*

Capítulo 7: *https://rpubs.com/jessicakubrusly/gabarito-cap-7-algoritmos-recursivos-cont*

Capítulo 8: *https://rpubs.com/jessicakubrusly/gabarito-cap-8-algoritmos-ordenacao*

Capítulo 9: *https://rpubs.com/jessicakubrusly/gabarito-cap-9-aproximacao-de-funcoes*

Capítulo 10: *https://rpubs.com/jessicakubrusly/gabarito-cap-10-raizes-de-funcoes*

Capítulo 11: *https://rpubs.com/jessicakubrusly/gabarito-cap-11-derivacao-numerica*

Capítulo 12: *https://rpubs.com/jessicakubrusly/gabarito-cap-12-integracao-numerica*

REFERÊNCIAS

Apresento a seguir algumas referências que podem aprofundar e complementar o conteúdo apresentado nesse livro.

DE SOUZA, M. A. F.; GOMES, M. M.; SOARES, M. V.; CONCILIO, R. *Algoritmos e lógica da programação.* [*S.l.*]: Cengage do Brasil, 2019.

MATLOFF, N. *The art of R programming*: a tour of statistical software design. [*S.l.*]: No Starch Press, 2011.

RUGGIERO, M. A. G.; LOPES, V. L. R. *Cálculo numérico*: aspectos teóricos e computacionais. 2. ed. [*S.l.*]: Pearson Universidades, 2000.