UNIVERSIDADE FEDERAL DO PARANÁ

APOSTILA DE PROGRAMAÇÃO EM R

Profª Drª Mariana Kleina

2025

CURITIBA - PR

Sumário

1 INTRODUÇÃO

R é um ambiente de programação livre para manipulação de dados, cálculos matemáticos e estatísticos, com visualização gráfica bem desenvolvida.
Possui diversos pacotes (também chamados de bibliotecas) padrões já instalados, porém pacotes não padrões podem ser facilmente instalados, bastando uma conexão com a internet.
Cada pacote possui uma coleção de funções prontas para serem usadas. Entretanto, você mesmo pode escrever suas próprias funções.

1.1 Site oficial

No site oficial são encontrados manuais, livros sobre o R, links para *download*, entre outras funcionalidades.
www.r-project.org

1.2 Instalação

cran.r-project.org (escolher sistema operacional, disponível para Windows, Linux e MacOS).

1.3 Manual

Manual introdutório em inglês: cran.r-project.org/doc/manuals/r-release/R-intro.pdf

1.4 RStudio

Ambiente de desenvolvimento integrado para R. Possui uma interface mais amigável e intuitiva, que ajuda o programador a ter um melhor controle e visualização do código que se está escrevendo. No site https://posit.co/download/rstudio-desktop/ é possível fazer o download do ambiente.
O RStudio é apenas uma interface para o R e, portanto, necessita primeiro da instalação do R.
Nesta apostila, todos os comandos podem ser feitos em qualquer um dos ambientes, R ou RStudio.

1.5 Iniciando o R

No Windows ou MacOS é só clicar nos ícones que foram criados no momento da instalação do R ( ) ou RStudio ( ). No Linux basta digitar R na linha de comando no terminal.

No console do R, símbolo > indica que o programa está pronto para o uso. Então, pode-se começar a digitar os comandos desejados.
Se o símbolo + aparecer no lugar do >, significa que a linha de comando foi quebrada ou se está dentro de um laço ou foi esquecido de fechar parênteses, chaves ou colchetes. Para voltar ao ambiente de digitação (>), basta apertar a tecla *Esc*.
Ao iniciar o R, é apresentada uma mensagem inicial, onde é informada a versão do software que está instalada no computador.

1.6 Encerrando o R

Para fechar o R, basta digitar

```
> q()
```

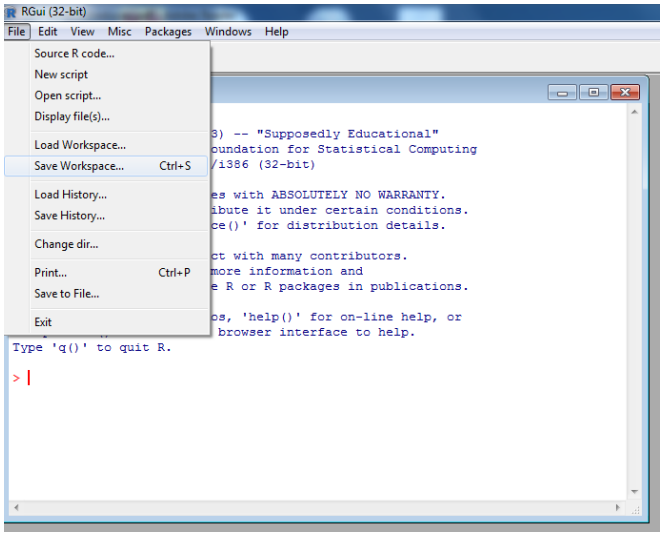
então será perguntado se deseja salvar a área de trabalho.

1.7 Criando e salvando uma Área de Trabalho (*Workspace*)

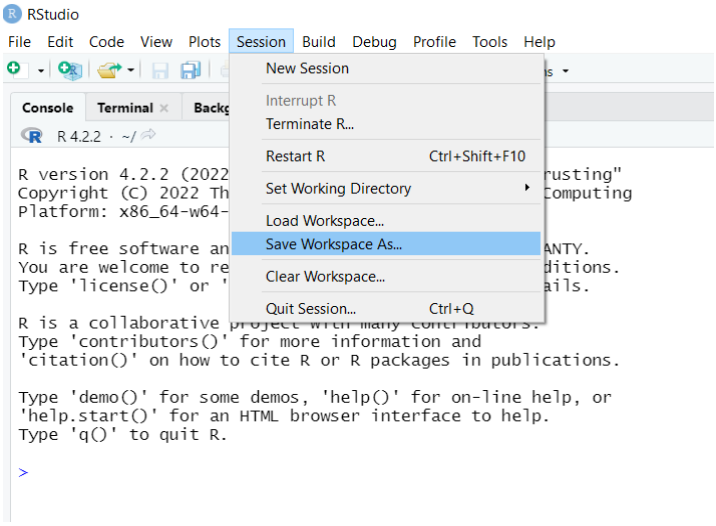
No Windows ou MacOS, quando se inicia o R, é possível verificar qual é o local que se está trabalhando, por meio do comando `getwd()`. Pode-se alterar o diretório de trabalho com o comando `setwd()`, passando como argumento o caminho do novo diretório, com / (barra) no lugar de \ (contra-barra), entre aspas.
O comando `dir()` lista os arquivos do diretório que se está trabalhando.

Para facilitar, pode-se criar um workspace para se aprender os comandos desta apostila. Para isso, abra o R normalmente, e salve (conforme imagem a seguir no R e RStudio, respectivamente) com a extensão *.RData* em uma pasta destinada somente para trabalhar com o R.

R



RStudio

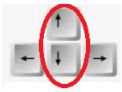


Feche o R e vá até esta pasta. Note que foi criado um atalho para o R com o nome que você deu ao workspace. Abra o R clicando nesse atalho.
A ideia de salvar um workspace é salvar variáveis para posterior utilização. Assim, ao abrir o R em uma próxima seção, as variáveis salvas são carregadas automaticamente. Ao abrir um novo workspace (sem ter salvado), a lista de variáveis estará vazia.

Por facilidade, salve nesta pasta todos os arquivos para posterior leitura no R, assim não será necessário passar o caminho onde se encontram estes arquivos.

No Linux, crie um diretório para trabalhar, e execute o R na linha de comando (*shell*) a partir desse diretório.

1.8 Aproveitando comandos já digitados

Você pode recuperar comandos já digitados no R por meio das teclas 

2 R e OPERAÇÕES MATEMÁTICAS

2.1 Operações matemáticas básicas

Antes de tudo, é preciso dizer que a representação decimal no R é o ponto (`.`) e não a vírgula (`,`).

```
> 1+8.97 # soma - isto é um comentário
[1] 9.97
> 4-2 # subtração
```

```
[1] 2
> 2*2 # multiplicação
[1] 4
> 5*(1+6) # parênteses priorizam operações
[1] 35
> 7/3 # divisão
[1] 2.333333
> 6^2 # exponenciação, que também pode ser feita da forma 6**2
[1] 36
> 5%/%2 # divisão inteira
[1] 2
> 5%%2 # resto da divisão
[1] 1
```

Obs 1.: O símbolo # indica que tudo que está à direita dele é um comentário, e o R não interpreta como comando, ou seja, não é executado. Comentários são úteis para explicar/lembrar o que o código está fazendo. Comentários podem vir após um comando ou em linhas sem comandos.
Obs 2.: O termo [1] antes do resultado da operação não faz parte do resultado, e significa apenas que a impressão do resultado ocupa uma linha do console.

Duas ou mais operações podem ser feitas em uma mesma linha de código, basta colocar um ponto e vírgula entre elas.

```
> 3+9; 17-8*2
[1] 12
[1] 1
```

Algumas funções matemáticas comuns já estão prontas para o uso:

```
> sqrt(9) # raiz quadrada
[1] 3
> abs(-89.65) # valor absoluto
[1] 89.65
> exp(1) # exponencial
[1] 2.718282
> exp(5.6)
[1] 270.4264
> log(10) # logaritmo natural ou neperiano
[1] 2.302585
> log10(10) # logaritmo base 10
[1] 1
> log2(10) # logaritmo base 2
[1] 3.321928
> factorial(4) # fatorial
[1] 24
```

Funções trigonométricas (sempre em radianos):

```
> pi
[1] 3.141593
> sin(0.5*pi) # seno
[1] 1
> cos(2*pi) #cosseno
[1] 1
> tan(pi) # tangente
[1] -1.224606e-16
> asin(1) # arco seno
[1] 1.570796
> asin(1)/pi*180
[1] 90
> acos(0) # arco cosseno
```

```
[1] 1.570796
> acos(0)/pi*180
[1] 90
> atan(1) # arco tangente
[1] 0.7853982
> atan(1)/pi*180
[1] 45
```

Funções para arredondamento:

```
> ceiling(4.3478) # teto
[1] 5
> floor(4.3478) # chão
[1] 4
> trunc(4.3478) # retorna a parte inteira de um número
[1] 4
> round(4.3478) # arredonda sem nenhuma casa decimal
[1] 4
> round(4.3478,3) # arredonda com 3 casas decimais
[1] 4.348
> round(4.3478,2) # arredonda com 2 casas decimais
[1] 4.35
```

O R também trabalha com operações matemáticas que envolvem elementos infinitos e elementos indeterminados:

```
> 1/0
[1] Inf
> -5/0
[1] -Inf
> 0/0
[1] NaN # NaN significa Not a Number
> Inf/Inf
[1] NaN
> log(0)
[1] -Inf
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

O R também tem um outro tipo de indefinição, que será visto adiante em exemplos, que é o NA (*Not Available*), onde ele representa alguma inconsistência nos dados. Importante falar que operações matemáticas com NA (escrito em maiúsculo mesmo) resultam em NA.

```
> 2*NA # NA significa Not Available
[1] NA
```

Em resumo:

- `Inf`: representa um número muito grande que o computador não consegue representar ou um limite matemático;
- NaN: representa indefinições matemáticas;
- NA: representa ausência de informação.

2.2 Atribuição de variáveis

O operador `<-` é usado para atribuir valores à variáveis.

```
> a<-5
> b<-2.6*8
```

Observe que não precisa definir o tipo da variável previamente.

O símbolo = também pode ser usado para atribuição.

É possível atribuir o mesmo valor a diversas variáveis de uma só vez utilizando atribuições múltiplas de valores.

```
> a<-c<-5
```

O R é *case sensitive*, isto é, faz distinção entre letras maiúsculas e minúsculas.

Outra observação importante é a não permissão de caracteres especiais, espaços e início numérico para nomes de variáveis. Também se recomenda a não utilização de acentuação. Essas mesmas regras valem para nome de funções, que serão vistas adiante.

A seguir é apresentado um resumo das principais classes de variáveis no R.

Tipo	Descrição	Propriedades	Exemplos
Texto (`character`)	Caracteres alfanuméricos	Não permite operações matemáticas. Devem ser definidos usando aspas.	"Texto", "teste", "xxx", "dia 10"
Numérica (`numeric`)	Números	Permite operações matemáticas	36.21, 0.2, -9.147, 10
Inteira (`integer`)	Números do conjunto dos inteiros	Proibição de valores decimais. O R coloca o sufixo `L` para diferenciar de uma variável numérica.	1L, 5L, -8L
Lógica (`logical`)	Categorias TRUE ou FALSE	Não são reconhecidos como textos. Não são definidos usando aspas.	TRUE, FALSE
Fator (`factor`)	Variáveis categóricas	Assume um número finito de categorias. As categorias podem ser caracteres, números, inteiros...	"Q1", "Q2", "Q3" ou "Q4" "bom", "mediano" ou "ruim" 0, 50 ou 100
Número complexo (`complex`)	Número complexo, composto por uma parte real e uma parte imaginária	Suporta aritmética complexa em R, além dos operadores aritméticos +, -, *, / e ^.	1+2i

Obs.: Aspas no R podem ser simples ou dupla. A única observação é que se foram abertas aspas simples, então devem-se fechar aspas simples (mesma coisa para aspas dupla).

É possível descobrir a classe de uma variável por meio da função `class()`.

```
> teste<-1L

> teste
[1] 1
> class(teste)
[1] "integer"
```

Outras duas funções úteis são para descobrir se uma variável é de uma classe (`is.xxx()`) e para transformar uma variável em uma classe (`as.xxx()`), quando possível. Obs.: `xxx` pode ser `character`, `numeric`, `integer` ou `logical`.

```
> is.character("texto")
[1] TRUE
> is.character(a)
[1] FALSE
> is.numeric(a)
[1] TRUE

> as.character(a)
[1] "5"
> as.numeric("5.25")
[1] 5.25
```

```
> as.numeric(TRUE) # veja que TRUE é equivalente ao valor numérico 1
[1] 1
> as.numeric(FALSE) # veja que FALSE é equivalente ao valor numérico 0
[1] 0

> as.numeric("testando")
[1] NA
Warning message:
NAs introduced by coercion
```

É possível ler uma expressão digitada diretamente do teclado e alocar em uma variável do tipo caractere, por meio da função `readline()`.

```
> teste<-readline("Digite algo: ")
Digite algo: 66
> teste
[1] "66"
```

Para listar todas as variáveis criadas no seu *workspace*, basta usar a função `ls()`.

```
> ls()
 [1] "a"  "b"  "c"  "teste"
```

A estrutura de uma variável:

```
> str(a)
 num 5
```

A função `ls.str()` lista todas as variáveis criadas e mostra a estrutura de cada uma delas:

```
> ls.str()
a :  num 5
b :  num 20.8
c :  num 5
teste :  chr "66"
```

Remoção de uma ou mais variáveis:

```
> rm(b,c)
> ls()
 [1] "a"  "teste"
```

Remoção de todas as variáveis:

```
> rm(list=ls())
> ls()
character(0)
```

2.3 Impressão de variáveis

Pode-se mostrar o valor de uma variável digitando o nome dela no *console* ou por meio dos comandos `print()` ou `cat()`.

```
> a
[1] 5
> print(a)
[1] 5
> cat(a)
5
> cat(a,b) #imprimindo duas variáveis
5 20.8
```

O resultado do `cat()` acima é no RStudio. No R, por padrão, o resultado é:

```
> cat(a)
```

```
5>
```
Ou seja, o console fica pronto para digitação na mesma linha do resultado do `cat()`. Para pular de linha e obter o mesmo resultado que o RStudio, faça:
```
> cat(a,'\n') # '\n' pula de linha
5
```

Com o comando `cat()` também é possível mesclar texto e variáveis.
```
> cat('O conteúdo de b é',b)
O conteúdo de b é 20.8
```

2.4 Operadores de comparação

Os operadores de comparação são:

- `==` (igual a);
- `!=` (diferente de);
- `>` (maior que);
- `>=` (maior ou igual a);
- `<` (menor que);
- `<=` (menor ou igual a).

O resultado é um valor lógico (TRUE ou FALSE).

```
> 1==1
[1] TRUE
> 1==3
[1] FALSE
> 1!=3
[1] TRUE
> 5>10
[1] FALSE
> 10>3
[1] TRUE
> 4>=4
[1] TRUE
> 9<5
[1] FALSE
> 8<=10
[1] TRUE
```

Operadores de comparação não tem prioridade uns sobre os outros, ou seja, se dois ou mais operadores de comparação aparecerem em uma mesma expressão, então eles serão avaliados da esquerda para a direita. Lembrando que parênteses priorizam operações.

2.5 Operadores lógicos

Os operadores lógicos são:

- `&` e `&&` (**e** - conjunção);
- `|` e `||` (**ou** - disjunção);
- `!` (**não** - negação).

O resultado é um valor lógico.

Lembrando o resultado do uso de operadores lógicos:

&	|	!
TRUE & TRUE resulta TRUE	TRUE | TRUE resulta TRUE	! TRUE resulta FALSE
TRUE & FALSE resulta FALSE	TRUE | FALSE resulta TRUE	! FALSE resulta TRUE
FALSE & TRUE resulta FALSE	FALSE | TRUE resulta TRUE	
FALSE & FALSE resulta FALSE	FALSE | FALSE resulta FALSE	

Obs.: && avalia a 1ª condição e, somente se ela é verdadeira, então avaliará a segunda condição. Por outro lado, & avalia todas as condições. A ideia é a mesma para | e ||.

```
> 5>3 & 10>8
[1] TRUE
> 5>3 & 10>11
[1] FALSE
> 5>3 | 10>11
[1] TRUE
> 9!=9 | FALSE
[1] FALSE
> !5>3
[1] FALSE
> 8>4 & 3<1 & 7/'b'
Error in 7/"b" : non-numeric argument to binary operator
> 8>4 && 3<1 && 7/'b'
[1] FALSE
> 3<4 || 7/"b"
[1] TRUE
> 3<4 | 7/"b"
Error in 7/"b" : non-numeric argument to binary operator
```

A ordem de prioridade de operadores lógicos é:
1º: !
2º: &
3º: |
Veja o seguinte exemplo:

```
> TRUE & TRUE | FALSE & FALSE
[1] TRUE
```

Pela ordem de prioridade, as comparações dos dois operadores & são feitos primeiramente, para depois ser feita a comparação do operador |. Se a ordem de prioridade fosse a mesma para os três operados, então o resultado seria FALSE.

2.6 Ajuda com funções e parâmetros

Para obter ajuda sobre uma função do R, tal como entrada de parâmetros, saída, exemplos, etc., basta digitar um ponto de interrogação na frente do nome da função ou digitar `help(nome_da_função)`:

```
> ?round # help(round)
```

Usa-se `args()` para ver os parâmetros (também chamados de argumentos) de uma função.

```
> args(round)
function (x, digits = 0)
```

Parâmetros podem ser obrigatórios ou opcionais. Por exemplo, o parâmetro x do exemplo anterior é obrigatório, pois nenhum valor padrão é atribuído a ele. Já o parâmetro `digits` é opcional, visto que, caso o usuário não informe nenhum valor, por padrão o valor dele é 0.

Parâmetros seguem uma ordem para serem informados no momento de chamar uma função. Se a ordem usada for a mesma que a ordem com que aparecem na descrição da função, então o nome do parâmetro pode ser omitido.

```
 > round(5.65, 1)
[1] 5.7
```

Ou seja, o número 5.65 foi arredondado com uma casa após a vírgula. Os nomes dos parâmetros poderiam ser explicitados também:

```
> round(x=5.65, digits=1)
[1] 5.7
```

Ou podem ser parcialmente explicitados:

```
> round(5.65, digits=1)
```

```
[1] 5.7
```

Entretanto, veja o que acontece quando os nomes dos parâmetros não são explicitados e não são informados na ordem correta:

```
> round(1, 5.65)
[1] 1
```

Neste caso, a função arredondou o número 1 com 5.65 casas após a vírgula, que tem como resultado o valor 1.
Para usar a ordem não habitual, o nome dos parâmetros se faz necessária:

```
> round(digits=1, x=5.65)
[1] 5.7
```

Outra informação importante é que se pode escrever somente as iniciais do parâmetro se nenhum outro parâmetro da função tiver as mesmas iniciais. Por exemplo:

```
> round(di=1, x=5.65)
[1] 5.7
```

2.7 Datas e tempos

No R, datas (ano, mês, dia) são representadas pela classe `Date`. Tempos (ano, mês, dia, hora, minuto, segundo etc.) são representados pelas classes `POSIXct` ou `POSIXlt`.

Internamente, as datas são armazenadas como o número de dias desde 01/01/1970. Horas são armazenadas como o número de segundos desde 01/01/1970.

Para digitar uma data específica no R, a função `as.Date()` pode ser usada:

```
> x<-as.Date("2025-02-28")
> x
[1] "2025-02-28"
```

A data será exibida no formato ano-mês-dia.

O parâmetro `format` permite ao usuário especificar como a data será construída. Por exemplo, no comando a seguir, `format="%m-%d-%Y"` faz com que a data seja criada a partir do formato mês-dia-ano.

```
>as.Date("02-28-2025", format = "%m-%d-%Y")
[1] "2025-02-28"
```

A representação interna de um objeto da classe `Date` pode ser vista por meio da função `unclass()`:

```
> x<-as.Date("2025-02-28")
> unclass(x)
 [1] 20147
```

Para digitar um tempo específico no R, a função `as.POSIXlt()` pode ser usada:

```
> as.POSIXlt("2018-01-31 16:23")
[1] "2018-01-31 16:23:00 -02"
> as.POSIXlt("2000/06/12")
[1] "2000-06-12 -03"
```

Obs.: -02 e -03 se referem a diferença de fuso horário para UTC (-02 quando se tem horário de verão).

Além disso, também existe a função `as.POSIXct()`, que é uma representação mais compacta de data e hora, geralmente preferida para cálculos, pois utiliza menos memória do que `POSIXlt()`.

- `ct` significa tempo de calendário, ele armazena o número de segundos desde a origem.
- `lt`, ou hora local, mantém a data como uma lista de atributos de hora.

Para saber a diferença entre tempos, a função `difftime()` pode ser usada. Por exemplo, tem-se duas variáveis `t1` e `t2`, depois basta fazer a diferença entre elas.

```
> t1<-as.POSIXlt("2019-03-12 13:30:54")
> t2<-as.POSIXlt("2020-02-17 05:47:11")
> difftime(t2,t1,units="days")
Time difference of 341.678 days
> difftime(t2,t1,units="min")
Time difference of 492016.3 mins
```

Para descobrir a data e hora atual, basta digitar na linha de comando:

```
> Sys.time()
```

1) Qual a diferença entre os códigos a seguir?

```
> 44/11
> x <- 44/11
```

2) Por que o nome `minha-idade` não pode ser usado para criar uma variável? O que significa a mensagem de erro resultante?

```
> minha-idade<-20
Error in minha - idade <- 20 : object 'minha' not found
```

3) Multiplique sua idade por 12 e salve o resultado em uma variável chamada `idade_em_meses`. Em seguida, multiplique essa variável por 30 e salve o resultado em uma variável chamada `idade_em_dias`.

4) Tem-se duas variáveis `a` e `b`. Troque o conteúdo das variáveis, isto é, faça com que `a` receba o conteúdo de `b` e vice-versa.

5) No R, calcule $\frac{\sqrt[5]{\ln(4)+\pi}}{e^2 sen(2{,}56)}$ e atribua o resultado a uma variável.

6) Veja qual é o resultado das operações `TRUE + TRUE`, `FALSE + TRUE` e `FALSE + FALSE`. O que se pode concluir a respeito do valor numérico de `TRUE` e `FALSE`?

7) Faça o cálculo a seguir mentalmente e depois confira o resultado no R:

```
>  7 %% 2 + (4*2 >= 2.6*2)*2
```

8) O código:

```
> segredo<-round(runif(1, min = 0, max = 10))
```

guarda na variável `segredo` um número inteiro entre 0 e 10. Sem olhar qual número foi guardado, resolva os seguintes itens:

- Teste se o `segredo` é maior que 5;
- Teste se o `segredo` é par;
- Teste se `segredo*5` é maior que a sua idade.

---

3 VETORES

Vetores são conjuntos indexados de valores de mesma classe, ou seja, cada valor dentro do vetor tem uma posição. Essa posição é dada pela ordem em que os elementos foram colocados no momento da criação do vetor.
Um vetor pode ser criado de diferentes formas. A mais comum é concatenando elementos um do lado do outro, por meio da função `c()`, sempre separando os parâmetros por vírgula.

```
> x<-c(5,-9,1.1)
```

Uma sequência de números também é um vetor e é facilmente criado da forma:

```
> y<-seq(4,30,by=2) # começa em 4 e acaba em 30, com passo de 2
> y
[1] 4  6  8 10 12 14 16 18 20 22 24 26 28 30

> z<-seq(0,9,length.out=3) # começa em 0 e acaba em 9, com 3 elementos
> z
[1] 0.0 4.5 9.0

> z<-seq(0,9,len=3) # usando a ideia de abreviar nome de parâmetros
> z
[1] 0.0 4.5 9.0
```

Para criar vetores de números com intervalo fixo unitário (intervalo de 1), se utiliza o operador sequencial (`:`):

```
> w<-1:8
> w
[1] 1 2 3 4 5 6 7 8

> w<-2.5:10
> w
[1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
```

Para criar um vetor com um mesmo elemento repetidas vezes, usa-se a função `rep()`, em que é preciso informar quantas vezes o elemento deve ser repetido:

```
> p<-rep(7,times=12)
> p
 [1] 7 7 7 7 7 7 7 7 7 7 7 7

> p<-rep(seq(-1,3),times=3)
> p
 [1] -1  0  1  2  3 -1  0  1  2  3 -1  0  1  2  3
```

Para ler dados numéricos diretamente do teclado e alocá-los em um vetor, usa-se a função `scan()`. Para parar a leitura de dados, basta apertar *Enter* no teclado (gravando os dados lidos) ou *Esc* (não gravando os dados lidos).

```
> s<-scan()
1: 5
2: 6
3: 8
4:
Read 3 items
> s
[1] 5 6 8
```

Obs.: Por padrão, vetores construídos com os dois pontos (`:`) ou com `seq()` são `integer`, enquanto que vetores construídos com `c()` ou com `rep()` ou com `scan()` são `numeric`. Veja o exemplo:

```
> g<-1:5
> class(g)
[1] "integer"

> g<-c(1,2,3,4,5)
> class(g)
[1] "numeric"
```

A classe `integer` ocupa menos espaço na memória do que a classe `numeric`:

```
> g <- 1:100
> object.size(g)
448 bytes
> object.size(as.numeric(g))
848 bytes
```

Embora `integer` ocupe menos memória, esta classe só consegue representar números entre [-2147483646, 2147483647].

```
> .Machine$integer.max
[1] 2147483647

> as.integer(2147483647)
[1] 2147483647
as.integer(2147483647+1)
> [1] NA
```

Se na criação de um vetor, duas ou mais classes são misturadas, ocorrerá a chamada **coerção** no R.

```
> v<-c(1,2,"a")
> v
[1] "1" "2" "a"
> class(v)
[1] "character"
```

Como um vetor só pode ter uma classe de objetos dentro dele, classes mais fracas são reprimidas pelas classes dominantes (neste caso, a classe character domina a classe inteira). Assim, neste exemplo, todo o vetor ficou com elementos textuais.
A ordem de dominância é: `character` > `numeric` > `integer` > `logical`.
Valores lógicos são convertidos em números da forma: TRUE é convertido para 1 e FALSE para 0.

3.1 Operações com vetores

Operações matemáticas podem ser realizadas com vetores:

```
> x+z
[1]  5.0 -4.5 10.1
> x/2
[1]  2.50 -4.50  0.55
> x^2
[1] 25.00 81.00  1.21
> x*z # multiplicação valor a valor
[1]   0.0 -40.5   9.9
> x%*%z # multiplicação vetorial (produto escalar)
      [,1]
[1,] -30.6
> sqrt(z)
[1] 0.00000 2.12132 3.00000
> log(x)
[1] 1.60943791        NaN 0.09531018
Warning message:
In log(x) : NaNs produced
> log(abs(x))
[1] 1.60943791 2.19722458 0.09531018
```

Vetores de tamanhos diferentes são operados no R, porém um alerta é mostrado:

```
> c(1,2,3)+c(1,2,3,4,5)
[1] 2 4 6 5 7
Warning message:
In c(1, 2, 3) + c(1, 2, 3, 4, 5) :
longer object length is not a multiple of shorter object length
```

Neste caso, o R completa o vetor menor (começa a repetir os elementos) para que fique com o mesmo tamanho do vetor maior: `c(1,2,3,1,2)+c(1,2,3,4,5)`. Isso se chama *Regra da Ciclagem* e pode ser observada em muitas outras situações.
Porém, quando um vetor tem o seu tamanho como sendo múltiplo do outro vetor que se está operando, também ocorre a regra de ciclagem, entretanto nenhuma mensagem é apresentada.

```
> c(1,2)+c(1,2,3,4)
[1] 2 4 4 6
```

A regra de ciclagem pode ter te parecido um pouco estranha, mas ela é muito útil e constantemente usada em operações no R. Veja o exemplo:

```
> c(10,20,30,40,50)+1
[1] 11 21 31 41 51
```

Na operação anterior, o que de fato está sendo feito é a ciclagem do número 1, ou seja, `c(10,20,30,40,50)+c(1,1,1,1,1)`.

3.2 Funções para vetores

Informações de um vetor podem ser obtidas por meio de funções prontas:

```
> sum(y) # soma dos elementos do vetor
[1] 238
> length(y) # tamanho do vetor: quantidade de elementos
[1] 14
> min(y) # elemento mínimo do vetor
[1] 4
> max(y) # elemento máximo do vetor
[1] 30
> range(y) # valores mínimo e máximo do vetor, respectivamente
[1]  4 30
> mean(y) # media do vetor
[1] 17
> median(y) # mediana do vetor
[1] 17
> var(y) # variância do vetor
[1] 70
> sd(y) # desvio padrão do vetor: raiz quadrada da variância
[1] 8.3666
> sort(y) # ordena o vetor
 [1]  4  6  8 10 12 14 16 18 20 22 24 26 28 30
> rev(y) # inverte o vetor y
 [1] 30 28 26 24 22 20 18 16 14 12 10  8  6  4
> append(y,9999,5) # anexa ao vetor y o número 9999 após a 5ª posição
 [1]    4    6    8   10   12 9999   14   16   18   20   22   24   26   28   30
> head(y) # mostra o começo do vetor
[1]  4  6  8 10 12 14
> tail(y) # mostra o final do vetor
[1] 20 22 24 26 28 30
> summary(y) # faz um resumo estatístico do vetor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    4.0    10.5    17.0    17.0    23.5    30.0
> prod(y) # faz o produto entre os elementos do vetor
[1] 2.142494e+16
```

3.3 Acessando posições de um vetor

Posições (índices) no R começam sempre no 1. Posições de um vetor podem ser acessadas com o [ ]:

```
> y
 [1]  4  6  8 10 12 14 16 18 20 22 24 26 28 30
> y[3]
[1] 8
> y[c(2,7)]
[1]  6 16
> y[50] # não existe a posição 50 no vetor y
[1] NA
```

3.4 Trocando elementos de um vetor

É possível substituir um ou mais elementos de um vetor por meio de seu índice e o novo valor que a posição irá receber.

```
> x
[1]  5.0 -9.0  1.1
> x[2] <- 14
> x
[1]  5.0 14.0  1.1
```

3.5 Excluindo elementos de um vetor

Elementos de um vetor podem ser excluídos por meio do seu índice com o sinal negativo.

```
> x
[1]  5.0 14.0  1.1
> x<-x[-1] # excluindo o primeiro elemento de x
> x
[1]  14.0  1.1
```

3.6 Acrescentando elementos em um vetor

Pode-se acrescentar elementos em qualquer posição de um vetor por meio do comando `append()`, já visto.

```
> x<-append(x,-5,0) # colocando o valor -5 na 1ª posição do vetor
[1]  -5.0 14.0  1.1
```

É possível criar novas posições, que ainda não existem no vetor, e colocar elementos:

```
> x[4]<-2
[1]  -5.0 14.0  1.1  2.0
```

Porém, o criar uma posição "espaçada" do último elemento, as posições intermediárias recebem NA.

```
> x[7]<-45
[1]  -5.0 14.0  1.1  2.0  NA  NA  45.0
```

3.7 Comandos especiais para vetores

- **`which()`**

O comando `which()` recebe como argumento um (ou mais) valor(es) lógico(s) e retorna o(s) índice(s) onde o resultado é TRUE.

```
> which(y==24)
[1] 11
```

Observe que:

```
> y==24
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
```

Ou seja, somente a posição 11 do vetor `y` continha o valor 24.

```
> which(y<10)
[1] 1 2 3
> which(y<10 | y>25)
[1]  1  2  3 12 13 14
> y[which(y<10 | y>25)] # valores de y em que a condição é TRUE
[1]  4  6  8 26 28 30
> y[y<10 | y>25] # para este resultado, não precisava do which
[1]  4  6  8 26 28 30
```

Os índices onde se encontram os elementos mínimos e máximos de um vetor podem ser obtidos, respectivamente, por:

```
> which.max(y)
[1] 14
> which.min(y)
[1] 1
```

Obs.: O comando `which()` é um facilitador de operação, porém é possível reproduzir o seu resultado. Por exemplo, considere o mesmo vetor `y` anterior e lembrando uma operação já realizada por meio do comando `which()`:

```
> which(y<10 | y>25)
[1]  1  2  3 12 13 14
```

O mesmo resultado pode ser obtido indexando os índices do vetor na mesma condição:

```
> (1:length(y))[y<10 | y>25]
[1]  1  2  3 12 13 14
```

- **`unique()`**

O comando `unique()` recebe como argumento um vetor e retorna o vetor sem repetições.

```
> v<-c(2,6,8,3,2,5,8,0)
> unique(v)
[1] 2 6 8 3 5 0
```

- **`table()`**

O comando `table()` retorna uma tabela com o número de ocorrências (frequência) de cada elemento do vetor ordenado.

```
> table(v)
v
0 2 3 5 6 8
1 2 1 1 1 2
```

- **`%in%`**

Para saber se um elemento se encontra em um vetor, isto pode ser feito da seguinte maneira:

```
> v<-c(2,6,8,3,2,5,8,0)
> v==3
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
> which(v==3)
[1] 4
```

O comando `%in%` busca se um vetor está contido em outro vetor (basta trocar `==` por `%in%`):

```
> v %in% c(0,6)
[1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

Outro exemplo:

```
> v<-c(2,6,8,3,2,5,8,0)
> w<-c(15,0,-4,3)
> v %in% w
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE

> w %in% v
[1] FALSE  TRUE FALSE  TRUE
```

Observe que o retorno é um vetor lógico sempre do tamanho do primeiro vetor.

3.8 Tratamento de NAs

Sabe-se que dados ausentes (NA) são comuns em conjuntos de dados.
É possível verificar se um conjunto de dados tem ausência de informações por meio da função `is.na()`. Por exemplo:

```
> v<-c(15,32,89,0,NA,8,-87,NA,12)
> is.na(v)
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

Para se retirar esses dados faltantes do conjunto de dados, existem algumas formas:

```
> v[!is.na(v)]
[1]  15  32  89   0   8 -87  12
> v[-which(is.na(v))]
[1]  15  32  89   0   8 -87  12
> na.omit(v)
[1]  15  32  89   0   8 -87  12
attr(,"na.action")
[1] 5 8
attr(,"class")
[1] "omit"
```

Dados faltantes são tão comuns em conjunto de dados que muitas funções já tratam desse problema por meio de parâmetros.

```
> mean(v,na.rm=TRUE)
[1] 9.857143
```

3.9 Criando um vetor vazio

É possível iniciar um vetor vazio da seguinte forma:

```
> v<-NULL
> v
NULL
```

E depois ir acrescentando valores a esse vetor:

```
> v[5]<-1
> v
[1] NA NA NA NA  1
> v[8]<-7
> v
[1] NA NA NA NA  1 NA NA  7
> v[c(1,10)]<-c(0,50)
> v
[1]  0 NA NA NA  1 NA NA  7 NA 50
```

Note que nas posições onde não foram definidos valores, o padrão NA é inserido.

1) Crie os seguintes vetores:

- $v = [20,19,\ldots,2,1]$
- $v = [1,2,3,\ldots,20,19,18,\ldots,2,1]$
- $v = [4,6,3,4,6,3,\ldots,4,6,3]$ em que existem 10 ocorrências da sequência 4, 6, 3.
- $v = [4,4,\ldots,4,6,6,\ldots,6,3,3,\ldots,3]$ em que existem 10 ocorrências do 4, 20 ocorrências do 6 e 30 ocorrências do 3.
- $v = [0.1^3, 0.2^4, 0.1^5, 0.2^6, \ldots, 0.1^{15}, 0.2^{16}]$

2) Se você tem dois vetores $v = [v_1, v_2, \ldots, v_{1000}]$ e $w = [w_1, w_2, \ldots, w_{500000}]$, como você faria para obter o vetor $z = [v_1, v_2, \ldots, v_{1000}, w_1, w_2, \ldots, w_{500000}]$?

3) Qual é o resultado da soma: 1 + 2 + 3 + ... + 100?

4) Faça um código que conte o número de NAs do vetor x:

```
x<-c(1, NA, NA, 5, NA, -2, NA, NA, 0, 5, NA, 12, 71, NA)
```

5) Escreva um código que devolva apenas os valores maiores ou iguais a 5 do vetor y:

```
y<-c(1, 10, 5, -1, 9, -2, 0, 5, 10, 12, -1.5)
```

6) Crie um vetor vazio. Coloque na 3ª posição qualquer valor. Substitua todos os valores do vetor por 0. Acrescente ao final do vetor o valor 200.

7) Dado dois vetores (podem ser de tamanhos diferentes), encontrar os elementos do primeiro vetor que também estão no segundo vetor (sem repetições).
Exemplo: $v = [2\ \ 6\ \ 8\ \ 3\ \ 2\ \ 5\ \ 8\ \ 0]$ e $w = [4\ \ 9\ \ 2\ \ 0\ \ 12]$ Resposta: $[2\ \ 0]$

---

4 MATRIZES

Matrizes são declaradas por meio da função `matrix()`, que recebe como argumentos os dados e o número de linhas e/ou o número de colunas.

```
> A<-matrix(c(1,2,3,4,5,6,7,8,9),ncol=3)
> A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9


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

É possível extrair o número de linhas e o número de colunas de uma matriz por meio dos comandos `nrow()` e `ncol()`, respectivamente:

```
> nrow(A)
[1] 2
> ncol(A)
[1] 4
> dim(A) # número de linhas e colunas simultaneamente
[1] 2 4
> length(A) # número de elementos da matriz
[1] 8

> A<-matrix(seq(0,50,len=6),ncol=2,nrow=3)
> A
     [,1] [,2]
[1,]    0   30
[2,]   10   40
[3,]   20   50
```

Observe que bastava informar apenas o número de linhas ou o número de colunas, pois com 6 elementos e 2 colunas por exemplo, o número de linhas é calculado automaticamente (neste caso 3).

```
> A<-matrix(seq(0,50,len=6),ncol=2)
> A
     [,1] [,2]
[1,]    0   30
[2,]   10   40
[3,]   20   50
```

Se o número de linhas e o número de colunas não são informados, por padrão é gerada uma matriz de uma coluna:

```
> A<-matrix(seq(0,50,len=6))
> A
     [,1]
[1,]    0
[2,]   10
```

```
[3,]   20
[4,]   30
[5,]   40
[6,]   50
```

Se forem informados mais elementos do que `ncol` × `nrow`, então os últimos elementos são descartados e um aviso é mostrado:

```
> A<-matrix(1:15,ncol=3,nrow=4)
Warning message:
In matrix(1:15, ncol = 3, nrow = 4) :
  data length [15] is not a sub-multiple or multiple of the number of rows [4]
> A
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Entretanto, se `ncol` × `nrow` for maior do que a quantidade de elementos informados, as últimas posições da matriz serão preenchidas com a repetição dos elementos (Regra da Ciclagem), e uma mensagem é mostrada:

```
> A<-matrix(1:15,ncol=3,nrow=6)
Warning message:
In matrix(1:15, ncol = 3, nrow = 6) :
  data length [15] is not a sub-multiple or multiple of the number of rows [6]
> A
     [,1] [,2] [,3]
[1,]    1    7   13
[2,]    2    8   14
[3,]    3    9   15
[4,]    4   10    1
[5,]    5   11    2
[6,]    6   12    3
```

Conforme se pode notar, os dados informados são alocados por coluna na matriz. Para que sejam dispostos por linha, basta mudar o parâmetro `byrow=TRUE` no momento da definição da matriz, que por padrão tem o valor `FALSE`:

```
> A<-matrix(1:9,ncol=3,byrow=TRUE)
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Uma matriz deve conter variáveis do mesmo tipo, isto é, todos os exemplos apresentados anteriormente são matrizes numéricas. Mas, matrizes de texto, por exemplo, também podem ser criadas:

```
> S<-matrix(c('a','b','c','d','e','f'),nrow=2,byrow=TRUE)
> S
     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "d"  "e"  "f"
```

4.1 Acessando posições de uma matriz

Para acessar posições de uma matriz, usa-se [ ], indicando o número da linha e o número da coluna que se quer acessar, respectivamente:

```
> A[2,1]
[1] 4
```

Se o número da coluna não é informado, acessam-se todas as colunas:

```
> A[3,]
[1] 7 8 9
```

Se o número da linha não é informado, acessam-se todas as linhas:

```
> A[,2]
[1] 2 5 8
```

Se nem o número da linha e nem o número da coluna são informados, a matriz é acessada inteiramente:

```
> A[,]
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Para substituir um elemento específico da matriz, basta acessar a posição e substituir o seu valor:

```
> A[2,1]<--99
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]  -99    5    6
[3,]    7    8    9
```

4.2 Excluindo linhas e colunas de uma matriz

Para excluir uma linha inteira de uma matriz, usa-se novamente a indexação negativa do(s) número(s) da(s) linha(s) que se deseja excluir:

```
> A<-matrix(1:25,ncol=5,byrow=TRUE)
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
> A<-A[c(-2,-4),] # exclui as linhas 2 e 4
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]   11   12   13   14   15
[3,]   21   22   23   24   25
```

Para excluir coluna(s):

```
> A<-A[,-1] # exclui a coluna 1
> A
     [,1] [,2] [,3] [,4]
[1,]    2    3    4    5
[2,]   12   13   14   15
[3,]   22   23   24   25
```

Pode-se excluir linha(s) e coluna(s) simultaneamente:

```
> A<-A[-2,-1] # exclui a linha 2 e a coluna 1
> A
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]   23   24   25
```

4.3 Acrescentando linhas e colunas em uma matriz

Existem duas funções que podem ser bastante úteis quando se trabalha com matrizes. Elas servem para acrescentar linhas (`rbind()`) ou colunas (`cbind()`) em uma matriz:

```
> A<-matrix(1:9,ncol=3,byrow=TRUE)
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> A<-rbind(A,c(10,11,12))
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> A<-cbind(A,c(13,14,15,16))
> A
     [,1] [,2] [,3] [,4]
[1,]    1    2    3   13
[2,]    4    5    6   14
[3,]    7    8    9   15
[4,]   10   11   12   16
```

Obs.: Para colar uma coluna na frente da matriz, por exemplo, bastaria fazer `cbind(c(13,14,15,16),A)`. Idem para linhas.

Observe novamente a Regra da Ciclagem:

```
> A<-cbind(A,c(17,18,19))
Warning message:
In cbind(A, c(17, 18, 19)) :
  number of rows of result is not a multiple of vector length (arg 2)
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3   13   17
[2,]    4    5    6   14   18
[3,]    7    8    9   15   19
[4,]   10   11   12   16   17
```

Porém, se são fornecidos mais elementos do que precisa, os últimos elementos são descartados.

O comando `rbind()` também serve para criar matrizes por meio da colagem de vetores, um embaixo do outro.

```
> rbind(c(6,8,9),c(-3,0,12.6))
     [,1] [,2] [,3]
[1,]    6    8  9.0
[2,]   -3    0 12.6
```

Da mesma forma, o comando `cbind()` pode ser usado para colar colunas e criar uma matriz.

4.4 Operações com matrizes

Operações aritméticas são efetuadas com matrizes:

```
> M1<-matrix(c(-5,0,2,4),ncol=2)
> M2<-matrix(c(2,1,-6,3),ncol=2)
> M1+5
     [,1] [,2]
[1,]    0    7
[2,]    5    9
> M1^2
     [,1] [,2]
```

```
[1,]   25    4
[2,]    0   16
> M1/7
           [,1]      [,2]
[1,] -0.7142857 0.2857143
[2,]  0.0000000 0.5714286
> M1+M2
     [,1] [,2]
[1,]   -3   -4
[2,]    1    7
> M1-5*M2
     [,1] [,2]
[1,]  -15   32
[2,]   -5  -11
> M1*M2 # multiplicação de elemento a elemento (operação não muito usual)
     [,1] [,2]
[1,]  -10  -12
[2,]    0   12
> M1%*%M2 # multiplicação matricial
     [,1] [,2]
[1,]   -8   36
[2,]    4   12
```

4.5 Funções para matrizes

```
> A<-matrix(c(-6,1,2,4),ncol=2,byrow=TRUE)
> A
     [,1] [,2]
[1,]   -6    1
[2,]    2    4

> det(A) # determinante da matriz
[1] -26

> diag(A) # extrai a diagonal da matriz
[1] -6  4

> colSums(A) # soma os elementos das colunas da matriz
[1] -4  5

> rowSums(A) # soma os elementos das linhas da matriz
[1] -5  6

> colMeans(A) # média dos elementos das colunas da matriz
[1] -2.0  2.5

> rowMeans(A) # média dos elementos das linhas da matriz
[1] -2.5  3.0

> as.vector(A) # dispõe como um vetor os elementos da matriz (por coluna)
[1] -6  2  1  4

> t(A) # transposta da matriz
     [,1] [,2]
[1,]   -6    2
[2,]    1    4

> solve(A) # inversa da matriz
           [,1]       [,2]
[1,] -0.15384615 0.03846154
[2,]  0.07692308 0.23076923


> b<-c(4,-1)
> solve(A, b) # resolve o sistema linear Ax=b
[1] -0.65384615  0.07692308
```

```
> A<-matrix(c(4,2,5,1),ncol=2)
> eigen(A) # autovalores (values) e autovetores (vectors) da matriz
eigen() decomposition
$values
[1] -6.196152  4.196152

$vectors
           [,1]        [,2]
[1,] -0.9813000 -0.09760789
[2,]  0.1924844 -0.99522495
```

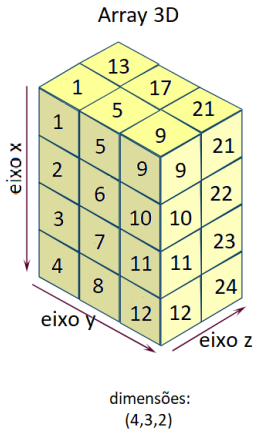
4.6 *Arrays*: matrizes de mais de duas dimensões

O conceito de *array* generaliza a ideia de matriz. Em uma matriz os elementos são organizados em duas dimensões (linhas e colunas), em um *array* os elementos podem ser organizados em um número maior de dimensões.
Suas dimensões são compostas por matrizes, onde cada camada comporta uma matriz.

Veja o exemplo a seguir, para o caso de três dimensões:

```
> ar<-array(1:24,dim=c(4,3,2))
> ar
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

Esse *array* é composto de 4 linhas (1ª dimensão – eixo x), 3 colunas (2ª dimensão – eixo y) e 2 profundidades (3ª dimensão – eixo z), conforme ilustração a seguir



Para acessar uma posição do *array*, usa-se a indexação tripla neste exemplo:

```
> ar[2,3,1] #acessando a linha 2, coluna 3 e profundidade 1
[1] 10
```

1) O que acontece ao criar uma matriz com números e caracteres?

2) Traço de uma matriz é a soma dos elementos da diagonal de uma matriz quadrada. Como determinar o traço de uma matriz no R?

3) Faça o que se pede:

- Crie a seguinte matriz de dimensão 3x500:

$$A = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 3 & 3 & 3 & \dots & 3 \\ 5 & 5 & 5 & \dots & 5 \end{bmatrix}$$

- Acrescente em *A,* a 501ª coluna.
- Acrescente em *A,* a 4ª linha.
- Remova simultaneamente de *A* as colunas de números 38 à 57, e a linha número 2.

4) Suponha que você tenha duas matrizes A e B tais que o nº de colunas da A é igual ao nº de linhas da B. Faça o produto escalar de uma linha da A (linha 2, por exemplo) por uma coluna da B (coluna 3, por exemplo).

Exemplo:

$$A = \begin{bmatrix} -11 & 41 & 1 & 12 & 1 \\ 3 & 6.9 & 3 & -1 & 3 \\ 6 & 0 & 5 & 9.1 & 4 \end{bmatrix} \text{ e } B = \begin{bmatrix} 2 & 12 & 7 \\ 4 & 12 & -8 \\ -2 & 3.9 & 16 \\ 0.8 & -9 & 19 \\ 13 & 6 & 0.8 \end{bmatrix}$$

Resposta: $3 \times 7 + 6.9 \times (-8) + 3 \times 16 + (-1) \times 19 + 3 \times 0.8 = -2.8$

5) Use exemplos no R para verificar as afirmações para matrizes quadradas:

a) Se $B = A\,A^t A^{-1}$, então $\det(A) = \det(B)$

b) $\det(A + B) \neq \det(A) + \det(B)$

c) $(A + B)^2 = A^2 + AB + BA + B^2$ (obs.: $A^2 = A \times A$)

6) Dadas as matrizes:

$$A = \begin{bmatrix} 1 & -3 & 2 \\ 2 & 1 & -3 \\ 4 & -3 & -1 \end{bmatrix}, B = \begin{bmatrix} 1 & 4 & 1 & 0 \\ 2 & 1 & 1 & 1 \\ 1 & -2 & 1 & 2 \end{bmatrix} \text{ e } C = \begin{bmatrix} 2 & 1 & -1 & -2 \\ 3 & -2 & -1 & -1 \\ 2 & -5 & -1 & 0 \end{bmatrix}$$

Verifique que $AB = AC$.

---

5 DATAFRAME

A estrutura dataframe é similar a de uma matriz (duas dimensões), porém permite que as colunas sejam de tipos diferentes, entretanto cada coluna deve conter uma variável com tipo único. Geralmente se dá um nome para cada coluna e associa-se um vetor de dados a essa coluna.

```
> d<-data.frame(números=c(5,-1,2),letras=c('a','b','c'))
> d
  números  letras
1       5       a
2      -1       b
3       2       c
```

Veja o exemplo a seguir onde é criada uma dataframe com nomes de funcionários de uma empresa por exemplo, data de nascimento, cargo, salário e a quantidade de horas trabalhadas por dia. Os dados do exemplo são fictícios e podem não representar a realidade.

```
> empresa<-data.frame(
nome=c('João','José','Cláudia','Ivo','Márcia','Fernanda'),
```

```
nascimento=c(as.Date('1990-05-12'),as.Date('1985-12-02'),
as.Date('1994-07-26'),as.Date('1988-03-15'),
as.Date('1974-08-21'),as.Date('1991-01-09')),
cargo=c('assistente comercial','gerente','estagiário','técnico de produto','auxiliar
administrativo','estagiário'),
salario=c(4000,6500,1000,3200,3400,1000),
horas_trabalhadas=c(8,8,6,8,8,6))
```

```
> empresa
      nome nascimento                   cargo salario horas_trabalhadas
1     João 1990-05-12    assistente comercial    4000                8
2     José 1985-12-02                 gerente    6500                8
3  Cláudia 1994-07-26               estagiário    1000                6
4      Ivo 1988-03-15      técnico de produto    3200                8
5   Márcia 1974-08-21 auxiliar administrativo    3400                8
6 Fernanda 1991-01-09               estagiário    1000                6
```

As funções `nrow()`, `ncol()` e `dim()` também podem ser usadas para dataframe:

```
> nrow(empresa)
[1] 6
> ncol(empresa)
[1] 5
> dim(empresa)
[1] 6 5
```

A função `length()` fornece o número de colunas da dataframe.

```
> length(empresa)
[1] 5
```

5.1 Acessando dados de uma dataframe

É possível ter acesso às informações da dataframe pelo `[ ]` informando o número de linha e o número da coluna:

```
> empresa[3,1]
[1] "Cláudia"
```

ou com o `[ ]` informando o número de linha e o nome da coluna entre aspas:

```
> empresa[3,"nome"]
[1] "Cláudia"
```

ou com o símbolo `$` seguido pelo nome da coluna e o número da linha dentro do `[ ]`.

```
> empresa$nome[3]
[1] "Cláudia"
```

Uma coluna inteira pode ser obtida:

```
> empresa$nascimento # ou empresa[,2]
[1] "1990-05-12" "1985-12-02" "1994-07-26" "1988-03-15" "1974-08-21"  "1991-01-09"
```

Assim como uma linha inteira pode ser obtida:

```
> empresa[4,]
  nome nascimento              cargo salario horas_trabalhadas
4  Ivo 1988-03-15 técnico de produto    3200                8
```

Com as informações dispostas nesta estrutura de dados, é possível filtrar informações que se deseja. Por exemplo, obter as informações dos funcionários que ganham salário maior que 3300:

```
> empresa[empresa$salario>3300,]
```

```
    nome nascimento                   cargo salario horas_trabalhadas
1   João 1990-05-12    assistente comercial    4000                 8
2   José 1985-12-02                 gerente    6500                 8
5 Márcia 1974-08-21 auxiliar administrativo    3400                 8
```

Neste exemplo, se está fazendo a indexação da variável *empresa* nas linhas em que a condição `empresa$salario>3300` é `TRUE`, e em todas as colunas (indicado por nenhum argumento após a vírgula).

O mesmo resultado é obtido usando a função `subset()`:

```
> subset(empresa, salario>3300)
    nome nascimento                   cargo salario horas_trabalhadas
1   João 1990-05-12    assistente comercial    4000                 8
2   José 1985-12-02                 gerente    6500                 8
5 Márcia 1974-08-21 auxiliar administrativo    3400                 8
```

Porém, qual será a média desses salários?

```
> mean(empresa[empresa$salario>3300,"salario"])
[1] 4633.333
```

Salários maiores que 3300 e menores que 5000:

```
> subset(empresa, salario>3300 & salario<5000)
    nome nascimento                   cargo salario horas_trabalhadas
1   João 1990-05-12    assistente comercial    4000                 8
5 Márcia 1974-08-21 auxiliar administrativo    3400                 8
```

Agora suponha que se queira saber o cargo das pessoas que trabalham menos que 8 horas diárias. Para isto, basta indexar *empresa* nas linhas em que a condição `empresa$horas_trabalhadas<8`, e selecionar a coluna 3 (que representa a coluna do cargo):

```
> empresa[empresa$horas_trabalhadas<8,3]
[1] "estagiário"  "estagiário"
```

Ou usando a função `subset()`:

```
> subset(empresa, horas_trabalhadas<8,3) #subset(empresa, horas_trabalhadas<8,'cargo')
       cargo
3 estagiário
6 estagiário
```

Observe que na função `subset()` é possível selecionar apenas colunas específicas, como anteriormente foi selecionada apenas a 3ª coluna.

5.2 Acrescentando linhas e colunas em uma dataframe

Para acrescentar colunas e linhas em uma dataframe, usam-se as funções `cbind()` e `rbind()`, respectivamente.

Será acrescentada uma nova coluna chamada *tempo_empresa*, que representará o número em dias que o funcionário está trabalhando na empresa. Como a dataframe tem 6 linhas, essa nova variável terá também 6 novas informações. Essa nova coluna será colocada, por exemplo, à frente do nome do funcionário, ou seja, será a primeira coluna (isso é feito colocando os novos dados como primeiro parâmetro e a dataframe como segundo parâmetro da função `cbind`).

```
> empresa<-cbind(tempo_empresa=c(255,620,90,365,421,15),empresa)
> empresa
  tempo_empresa     nome nascimento                   cargo salario horas_trabalhadas
1           255     João 1990-05-12    assistente comercial    4000                 8
2           620     José 1985-12-02                 gerente    6500                 8
3            90  Cláudia 1994-07-26              estagiário    1000                 6
4           365      Ivo 1988-03-15       técnico de produto    3200                 8
5           421   Márcia 1974-08-21 auxiliar administrativo    3400                 8
```

```
6            15 Fernanda 1991-01-09              estagiário    1000                 6
```

Para acrescentar uma nova linha, é preciso criar uma outra dataframe e basicamente apendar ela na dataframe original. Uma observação importante é que essa nova dataframe precisa ter a mesma estrutura que a outra (mesmo número e nome de colunas).

```
>   novos<-data.frame(tempo_empresa=200,   nome="Carlos",   nascimento=as.Date("2001-02-
24"),cargo="analista", salario = 4400, horas_trabalhadas=7.5)
> novos
  tempo_empresa   nome nascimento    cargo salario horas_trabalhadas
1           200 Carlos 2001-02-24 analista    4400               7.5
```

Assim, com a função `rbind()` é possível juntar as duas dataframes (neste exemplo, a nova linha foi colocada abaixo dos dados originais).

```
> empresa<-rbind(empresa,novos)
> empresa
  tempo_empresa     nome nascimento                  cargo salario horas_trabalhadas
1           255     João 1990-05-12    assistente comercial    4000               8.0
2           620     José 1985-12-02                gerente    6500               8.0
3            90  Cláudia 1994-07-26              estagiário    1000               6.0
4           365      Ivo 1988-03-15      técnico de produto    3200               8.0
5           421   Márcia 1974-08-21 auxiliar administrativo    3400               8.0
6            15 Fernanda 1991-01-09              estagiário    1000               6.0
7           200   Carlos 2001-02-24                analista    4400               7.5
```

Uma dica importante é: para descobrir os nomes das colunas de uma dataframe usa-se a função `names()`.

```
> names(empresa)
[1] "tempo_empresa" "nome" "nascimento" "cargo" "salario" "horas_trabalhadas"
```

Assim, para não correr o risco de errar o nome de alguma coluna no momento da criação da nova dataframe que se quer agregar à dataframe original, o que se pode fazer é criar essa dataframe sem os nomes das colunas por exemplo (ou pode-se colocar quaisquer nomes) e depois renomear essas colunas da seguinte forma:

```
> novos<-data.frame(200, Carlos", as.Date("2001-02-24"),"analista", 4400, 7.5)
> names(novos)<-names(empresa)
```

5.3 Excluindo linhas e colunas de uma dataframe

É possível excluir uma coluna inteira de uma dataframe:

```
> empresa<-empresa[,-2] # ou empresa$nome<-NULL
> empresa
  tempo_empresa nascimento                  cargo salário horas_trabalhadas
1           255 1990-05-12    assistente comercial    4000               8.0
2           620 1985-12-02                gerente    6500               8.0
3            90 1994-07-26              estagiário    1000               6.0
4           365 1988-03-15      técnico de produto    3200               8.0
5           421 1974-08-21 auxiliar administrativo    3400               8.0
6            15 1991-01-09              estagiário    1000               6.0
7           200 2001-02-24                analista    4400               7.5
```

ou excluir uma linha inteira de uma dataframe:

```
> empresa<-empresa[-5,] # exclusão da linha 5
> empresa
  tempo_empresa nascimento                cargo salário horas_trabalhadas
1           255 1990-05-12 assistente comercial    4000               8.0
2           620 1985-12-02              gerente    6500               8.0
3            90 1994-07-26            estagiário    1000               6.0
```

```
4           365 1988-03-15   técnico de produto    3200              8.0
6            15 1991-01-09           estagiário    1000              6.0
7           200 2001-02-24             analista    4400              7.5
```

ou excluir uma linha e uma coluna ao mesmo tempo:

```
> empresa<-empresa[-2,-1] # exclusão da linha 2 e da coluna 1
> empresa
  nascimento               cargo salário horas_trabalhadas
1 1990-05-12 assistente comercial    4000              8.0
3 1994-07-26           estagiário    1000              6.0
4 1988-03-15   técnico de produto    3200              8.0
6 1991-01-09           estagiário    1000              6.0
7 2001-02-24             analista    4400              7.5
```

ou excluir várias linhas e colunas ao mesmo tempo:

```
> empresa<-empresa[c(-1,-7),c(-1,-4)] #exclusão das linhas 1 e 7, e das colunas 1 e 4
> empresa
               cargo salário
3          estagiário    1000
4 técnico de produto    3200
6          estagiário    1000
7            analista    4400
```

1) Crie uma dataframe com 4 colunas: x, x^3, e^x e $log(x)$, com x variado de 1 a 100.

2) Crie uma data.frame com as seguintes informações suas: nome, idade, profissão e telefone. Agora acrescente a esta dataframe estas mesmas informações de dois colegas seus (mas não digite seus dados novamente). Então, acrescente uma nova informação a essa dataframe: o email de todos.

3) Digite no R o seguinte comando: `data("women")`. Agora você tem uma dataframe chamada *women*, que tem duas colunas: *height* (altura) e *weight* (peso). Apresente comandos para responder às perguntas:

- Qual é a média de peso do conjunto de dados?
- Qual é a média de peso das mulheres que tem altura entre [60, 70)?

4) Digite no R o seguinte comando: `data("iris")`. Agora você tem uma dataframe chamada *iris* (representando espécie de flores) que tem 5 colunas: *Sepal.Length* (comprimento da sépala), *Sepal.Width* (largura da sépala), *Petal.Length* (comprimento da pétala), *Petal.Width* (largura da sépala) e *Species* (espécies). Apresente comandos para responder às perguntas:

- Quantas observações são da espécie *versicolor*?
- Qual é a média da largura da sépala da espécie *setosa*?
- Apresente todas as observações que tem comprimento da pétala entre [3, 4].
- Qual é a espécie que tem em média menor comprimento da sépala?
- Digite no R o seguinte comando: `iris2<-iris[sample(1:150,150),]`. Por meio de `?sample`, explique o que representa a variável *iris2*.
- Acrescente uma coluna na dataframe *iris2*, com a seguinte regra: coloque o valor 0 nas linhas onde a espécie é *setosa*, o valor 1 onde a espécie é *versicolor* e o valor 2 onde a espécie é *virginica*.

5) Descubra o que fazem as funções `attach()` e `detach()`. Use elas, por exemplo, no conjunto *iris*.

---

6 LISTAS

Uma lista é um objeto versátil que pode conter uma coleção ordenada de elementos, que podem ser de diferentes tipos. Ao contrário de vetores, que exigem que todos os elementos sejam do mesmo tipo, uma lista pode conter números, *strings*, vetores, matrizes, dataframes, outras listas e até mesmo funções. A função `list()` é usada para criar uma lista.

```
> n<-c(2,3,5)
> s<-c('aa','bb','cc','dd')
> b<-c(TRUE,FALSE,TRUE,FALSE,FALSE)
> l<-list(n,s,b,-8)
> l
[[1]]
[1] 2 3 5

[[2]]
[1] "aa" "bb" "cc" "dd"

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

[[4]]
[1] -8
```

Será acrescentado na lista anterior uma matriz na 5ª posição.

```
> l[[5]]<-matrix(c(0,1,2,-6,3,9),ncol=3)
> l
[[1]]
[1] 2 3 5

[[2]]
[1] "aa" "bb" "cc" "dd"

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

[[4]]
[1] -8

[[5]]
     [,1] [,2] [,3]
[1,]    0    2    3
[2,]    1   -6    9
```

O comando `length()` pode ser usado para extrair o número de elementos de uma lista.

```
> length(l)
[1] 5
```

Para acessar um índice da lista, usa-se `[[ ]]`:

```
> l[[2]] # posição 2 da lista
[1] "aa" "bb" "cc" "dd"
```

E para acessar um subíndice da lista, usa-se `[[ ]][ ]`:

```
> l[[1]][2] # acessando o 2º elemento da posição 1 da lista
[1] 3

> l[[5]][1,3] # acessando a linha 1 e coluna 3 da posição 5 (matriz) da lista
[1] 3
```

Pode-se excluir um subíndice da lista:

```
> l[[1]]<-l[[1]][-2]
> l[[1]]
[1] 2 5
```

Também pode-se excluir uma posição inteira da lista:

```
> l[[3]]<-NULL
> l
[[1]]
[1] 2 3 5

[[2]]
[1] "aa" "bb" "cc" "dd"

[[3]]
[1] -8

[[4]]
     [,1] [,2] [,3]
[1,]    0    2    3
[2,]    1   -6    9
```

Obs.: Para exclusão de uma posição inteira da lista, o comando `l<-l[[-3]]` não funciona.

Você pode dar nomes aos elementos de uma lista:

```
> lista<-list(a=c(1,2),b="hi",c=3i)
> lista
$a
[1] 1 2

$b
[1] "hi"

$c
[1] 0+3i
```

e acessar índices pelo nome com o símbolo `$`, da forma:

```
> lista$b
[1] "hi"
```

ou

```
> lista[['b']]
[1] "hi"
```

ou ainda

```
> lista[[2]]
[1] "hi"
```

1) Descubra o que faz a função `unlist()`.

2) Seja um vetor numérico de qualquer tamanho. Crie uma lista de 3 posições contendo: na primeira posição os elementos do vetor que são menores que -50; na segunda posição os elementos do vetor que são maiores ou iguais a -50 e menores que 50; e na terceira posição os elementos do vetor que são maiores que 50.

3) Pode-se, ao invés de criar uma lista no exercício 2, criar uma matriz de 3 colunas ou uma dataframe?

4) Digite no R:

```
> data(Harman23.cor)
```

Note que agora você tem uma lista chamada Harman23.cor.

- Qual o menor valor contido na matriz que está na primeira posição dessa lista?
- Substitua o vetor que está na segunda posição dessa lista pela média das colunas da matriz que está na primeira posição.

---

7 MANIPULANDO *STRINGS*

Strings (sequência de caracteres) podem ser muito úteis quando, por exemplo, têm-se informações categóricas em um conjunto de dados (entre muitas outras situações). Serão vistos alguns dos comandos mais básicos para manipulação de *strings*.

A função `paste()`

A função `paste()` literalmente "cola" objetos, transformando-os em caracteres. Recebe como parâmetro os elementos que deseja colar.

```
> paste(21,8)
[1] "21 8"
> paste(21,'olá')
[1] "21 olá"
> paste(21,'olá',c(3,6))
[1] "21 olá 3" "21 olá 6"
> nome<-'João'
> idade<-54
> paste(nome, 'tem', idade, 'anos')
[1] "João tem 54 anos"
```

Como padrão, a separação para a colagem é um espaço em branco. Pode-se alterar a separação com o argumento `sep`.

```
> paste(1,2,345,sep='_')
[1] "1_2_345"
> paste(1,2,345,sep='$')
[1] "1$2$345"
> paste(1,2,345,sep='#')
[1] "1#2#345"
> paste(1,2,345,sep='ss')
[1] "1ss2ss345"
> paste(1,2,345,sep='')
[1] "12345"
```

Se não se quer separação, isto é, `sep=""`, então existe outra função para colagem, que é a função `paste0()`.

```
> paste0(1,2,345)
[1] "12345"
```

A função `substr()`

O comando `substr()` retira uma parte da *string* informada a partir de uma certa posição até uma outra posição.

```
> substr("Programação", 4, 7)
[1] "gram"
```

Neste exemplo, foi extraída uma nova *string* a partir da posição 4 até a posição 7 da palavra "Programação".
O resultado é sempre uma *string.*

A função nchar()

Esta função retorna o número de caracteres incluindo espaços em uma *string*. O resultado é sempre um valor inteiro.

```
> nchar("Aula de R")
[1] 9
```

As funções toupper() e tolower()

Estas funções mudam as letras de uma *string* para maiúscula e minúscula, respectivamente.

```
> toupper("Boa tarDE")
[1] "BOA TARDE"
> tolower("OLÁ, tudO BEM?")
[1] "olá, tudo bem?"
```

As funções grep() e grepl()

A funções `grep()` é usada para verificar se *strings* estão contidas em outras *strings* (retornando o valor 1) ou não (retornando `integer(0)`).

```
> grep("pato","sapato")
[1] 1
> grep("s","palavra")
integer(0)
```

Para buscar uma *string* dentro de um vetor de *strings*, usa-se a função `grepl()`, que retorna um vetor lógico indicando se na posição do vetor a *string* foi encontrada ou não.

```
> string1<-"at"
> vetor_string<-c("mar","atlântico","baleias","atlas")
> grepl(string1,vetor_string)
[1] FALSE TRUE FALSE  TRUE

> string1<-"at|as" # operador lógico OU usado
> grepl(string1,vetor_string)
[1]  FALSE TRUE TRUE  TRUE
```

Para muitas outras formas de manipular *strings*, veja o pacote `stringr`.

1) Suponha que existam seis vetores de mesmo tamanho, cada um contendo: anos, meses, dias, horas, minutos e segundos. Crie um único vetor representando datas no seguinte formato: "ano-mês-dia hora:minuto:segundo". Por exemplo, ano=2017, mês=08, dia=22, hora=12, minuto=45 e segundo=31; o resultado será "2017-08-22 12:45:31".

---

8 INSTALAÇÃO E ATIVAÇÃO DE PACOTES

Conforme já mencionado, algumas funções do R estão prontas para o uso, pois são funções de pacotes básicos, como é o caso das funções `mean()`, `sum()` e `solve()`, por exemplo. Porém, há funções que antes de serem usadas, é preciso ativar o pacote as quais elas pertencem (estes pacotes são chamados de **pacotes recomendados**).
Por exemplo, a função `Diagonal()` pertence ao pacote não-básico *Matrix*. Ao tentar usar a função `Diagonal()` sem ativar o pacote *Matrix*, observe o que acontece:

```
> Diagonal(3)
Error in Diagonal(3) : could not find function "Diagonal"
```

Entretanto, ativando antes o pacote com a função `library()`:

```
> library('Matrix')
> Diagonal(3)
3 x 3 diagonal matrix of class "ddiMatrix"
     [,1] [,2] [,3]
[1,]    1    .    .
[2,]    .    1    .
[3,]    .    .    1
```

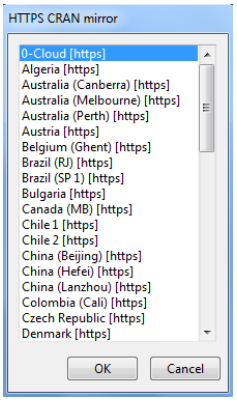
Porém, nem sempre um pacote está instalado no R (aos pacotes não instalados dá-se o nome de **pacotes contribuídos**). Assim, antes de ativá-lo, é preciso instalá-lo por meio do comando `install.packages()`. Por exemplo, por padrão o pacote *CircStats* não vem instalado no R:

```
> library('CircStats')
Error in library("CircStats") : there is no package called 'CircStats'
```

Para instalá-lo:

```
> install.packages('CircStats')
Installing package into 'C:/Users/Pessoal/Documents/R/win-library/3.3'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
```

Então uma janela é aberta solicitando um local de origem. Escolhe-se geralmente Brazil (PR), Brazil (SP 1) ou 0-Cloud caso não apareça Brazil:




```
Installing package into 'C:/Users/Pessoal/Documents/R/win-library/3.3'
(as 'lib' is unspecified)
trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.3/CircStats_0.2-4.zip'
Content type 'application/zip' length 119148 bytes (116 KB)
downloaded 116 KB

package 'CircStats' successfully unpacked and MD5 sums checked
```

```
The downloaded binary packages are in
        C:\Users\Pessoal\AppData\Local\Temp\RtmpaizQpT\downloaded_packages
```

Depois de instalado, o pacote deve ser ativado no R:

```
> library('CircStats')
```

Outra forma de ativar um pacote é por meio da função `require()`. Ela tem a mesma finalidade da função `library()`, entretanto a função `require()` retorna TRUE ou FALSE se o pacote está ou não instalado, respectivamente. Já a função `library()` retorna apenas uma mensagem de erro se um pacote não está instalado.

Assim, com a função `require()` é possível automatizar a verificação da presença do pacote e somente depois disso, acontece instalação deste, pois ele retorna TRUE ou FALSE. Segue exemplo para instalar o pacote *CircStats,* caso ele ainda não esteja instalado:

```
> instalado <- require(CircStats)
> if (instalado == FALSE) {
>        install.packages("CircStats")
> }
```

Obs.: O comando `if` será visto adiante nessa apostila.

Para mostrar todos os pacotes do R instalados no seu computador, bastar digitar:

```
> library()
```

Para mostrar todos os pacotes carregados no workspace de trabalho, basta digitar:

```
> search()
```

Para remover um pacote (por exemplo o pacote *CircStats*), usa-se:

```
> remove.packages('CircStats')
```

8.1 Observações sobre instalação de pacotes

- É preciso estar conectado à internet para instalar um pacote;
- Nem sempre a instalação de um pacote funciona devido à versão do R. Isso acontece porque novos pacotes são criados a todo o momento e somente em versões mais recentes do R que estes podem ser instalados.
- A lista completa de pacotes disponíveis do R pode ser obtida em: https://cran.r-project.org/web/packages/available_packages_by_name.html

9 CONJUNTOS DE DADOS PRONTOS

No R, existem muitos dados que estão disponíveis para serem utilizados para testes, análises, ou qualquer outra finalidade. Estes dados são acessíveis para que não se precise ficar gerando dados aleatórios quando informações são necessárias.

No pacote básico *datasets* (já vem instalado e carregado), há uma diversidade de dados, dos mais diferentes tipos, prontos para serem usados.

Veja a lista com o nome dos dados no link:
https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html
ou digite no R: `data()`

Para carregar um dado qualquer, usa-se a função `data()`, que tem como argumento obrigatório o nome do conjunto de dados que se deseja carregar. Quando se carrega um conjunto de dados no R, automaticamente é criada uma variável com o mesmo nome do conjunto de dados.
Exemplo:

```
> data(presidents)
> ls()
[1] "presidents"
> str(presidents)
 Time-Series [1:120] from 1945 to 1975: NA 87 82 75 63 50 43 32 35 60 ...
> data(volcano)
> str(volcano)
 num [1:87, 1:61] 100 101 102 103 104 105 105 106 107 108 ...
> str(quakes) #o conjunto de dados ainda não foi carregado, porém ele já existe
'data.frame':   1000 obs. of  5 variables:
 $ lat     : num  -20.4 -20.6 -26 -18 -20.4 ...
 $ long    : num  182 181 184 182 182 ...
 $ depth   : int  562 650 42 626 649 195 82 194 211 622 ...
 $ mag     : num  4.8 4.2 5.4 4.1 4 4 4.8 4.4 4.7 4.3 ...
 $ stations: int  41 15 43 19 11 12 43 15 35 19 ...
> ls()
[1] "presidents" "volcano"
> data(quakes)
> ls()
[1] "presidents" "quakes"     "volcano"
```

O comando a seguir apresenta os conjuntos de dados de todos os pacotes instalados em seu computador.

```
> data(package=.packages(all.available = TRUE))
```

Muitos pacotes do R contém conjuntos de dados próprios. Assim, para utilizá-los, basta instalar/carregar o pacote e depois carregá-lo por meio do comando `data(nome_conjunto_dados)`.

Por exemplo:

```
> data(votes.repub)
Warning message:
In data(votes.repub) : data set ‘votes.repub’ not found
> library('cluster')
> data(votes.repub)
> votes.repub
               X1856 X1860 X1864 X1868 X1872 X1876 X1880 X1884 X1888
Alabama           NA    NA    NA 51.44 53.19 40.02 36.98 38.44 32.28
Alaska            NA    NA    NA    NA    NA    NA    NA    NA    NA
Arizona           NA    NA    NA    NA    NA    NA    NA    NA    NA
Arkansas          NA    NA    NA 53.73 52.17 39.88 39.55 40.50 38.07
California     18.77 32.96 58.63 50.24 56.38 50.88 48.92 52.08 49.95
...
```

1) Carregue o conjunto de dados prontos `longley`. Transforme essa dataframe em uma lista.

2) Carregue o conjunto de dados prontos `longley`. Transforme essa dataframe em uma matriz.

3) Escolha um conjunto de dados do pacote `base` que seja composto por variáveis quantitativas. Use funções tais como `summary()`, `mean()` e `max()` para explorar esse conjunto de dados.

---

10 LEITURA E GRAVAÇÃO DE DADOS

Muitas vezes é necessário analisar um grande conjunto de dados que estão dispostos em um arquivo. Também é preciso gravar em arquivos um conjunto ou subconjunto de dados. O R é capaz de fazer essa leitura/gravação de uma forma simples e rápida.
O R consegue fazer a leitura de dados de três extensões muito utilizadas: texto (.txt), planilha Excel (.xlsx) e binário (.bin).

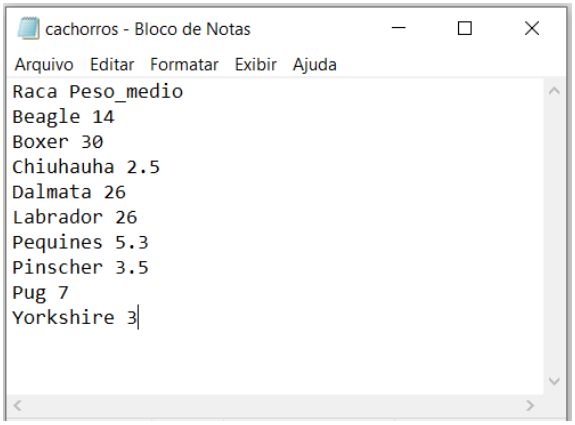
Obs.: Todos os exemplos a seguir farão leitura ou salvamento de arquivos no diretório atual. Porém, pode-se alterar o endereço do arquivo apenas colocando à frente do nome do arquivo o diretório onde ele se encontra (para leitura) ou onde se quer salvá-lo (no caso de gravação). Lembrando apenas que o diretório precisa estar com a barra (/) no lugar do contra-barra (\).

10.1 Leitura e gravação de arquivos de texto

- **Leitura**

Para ler um arquivo de texto no R, usa-se a função `read.table()`, que obrigatoriamente recebe como parâmetro o nome do arquivo para leitura, porém tem outros parâmetros opcionais.

Exemplo: fazer a leitura no R do seguinte arquivo de texto (está salvo no mesmo diretório do R) que contém um cabeçalho e duas colunas. Os dados do exemplo são fictícios e podem não representar a realidade.




```
> r<-read.table('cachorros.txt',header=TRUE)
> r
       Raca Peso_medio
1    Beagle       14.0
2     Boxer       30.0
3 Chiuhauha        2.5
4   Dalmata       26.0
5  Labrador       26.0
6  Pequines        5.3
7  Pinscher        3.5
8       Pug        7.0
9 Yorkshire        3.0
```

Todo o conteúdo do arquivo é colocado na variável `r`. Por padrão, `header=FALSE`, então é necessário alterar este parâmetro para verdadeiro, pois neste exemplo há um cabeçalho.
Se é esquecido de colocar `header=FALSE`, veja o que acontece:

```
> r<-read.table('cachorros.txt')
> r
          V1         V2
1       Raca Peso_medio
2     Beagle         14
3      Boxer         30
4  Chiuhauha        2.5
```

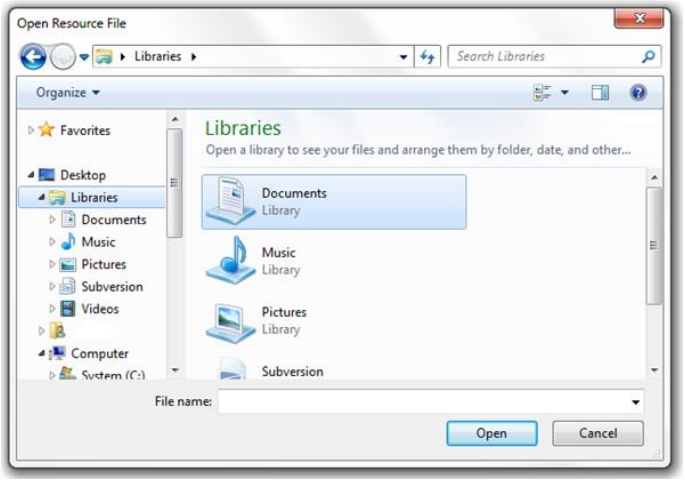
```
5    Dalmata         26
6   Labrador         26
7   Pequines        5.3
8   Pinscher        3.5
9        Pug          7
10 Yorkshire          3
```

Observe que a segunda coluna mistura dados numéricos com caracteres. Isso faz com que toda a segunda coluna se torne `character`, por coerção.

Outra forma de informar o nome do arquivo a ser lido é deixar que o usuário busque em seu computador o nome e o local do arquivo, de modo iterativo:

```
> r<-read.table(file.choose())
```

Assim, uma Janela como está será aberta e o usuário precisará escolher o arquivo a ser lido.



Se deseja-se pular algumas linhas no começo do arquivo, usa-se o argumento `skip`.

```
> r<-read.table('cachorros.txt',skip=3) # pula o cabeçalho + 2 linhas de dados
> r<-read.table('cachorros.txt',header=TRUE,skip=3) # pula 3 linhas de dados
```

Existem diversos outros parâmetros que podem ser passados para a função `read.table()`, como por exemplo a separação dos dados (por padrão a separação é um espaço), a pontuação para representar decimais (por padrão é o ponto), entre outros. Veja todos os parâmetros por meio do comando `?read.table`

Obs. 1: Arquivos separados por vírgula (.csv) podem ser lidos por meio das funções `read.csv()` e `read.csv2()`. A diferença entre elas está em dois parâmetros: `sep` (separador de colunas) e `dec` (separador de casas decimais). Em `read.csv()`, o padrão é `sep=","` e `dec="."`. Já em `read.csv2()`, tem-se `sep=";"` e `dec=","`.

Obs. 2: As funções `read.table()`, `read.csv()` e `read.csv2()` também podem receber um endereço web como caminho para leitura de arquivos.

- **Gravação**

Para gravar dados em um arquivo de texto, usa-se a função `write.table()`, que recebe obrigatoriamente como parâmetro a variável a ser gravada e o nome do arquivo.

Usando o arquivo anterior lido da forma:

```
> r<-read.table('cachorros.txt',header=TRUE)
```

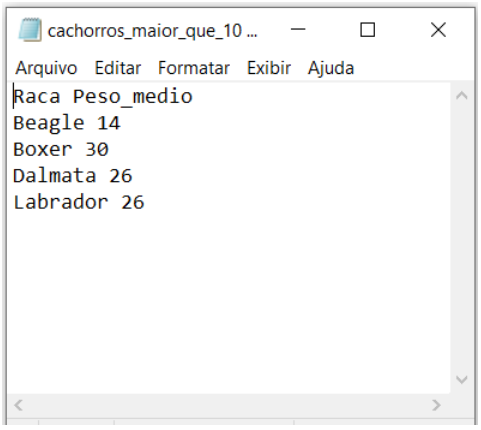
Suponha que se necessita filtrar os dados para raças de cães com peso médio superior a 10 kg:

```
> r10<-r[r$Peso_medio>10,] # estou criando a variável r10
```

```
> r10
      Raca Peso_medio
1   Beagle          14
2    Boxer          30
4  Dalmata          26
5 Labrador          26
```

E agora se deseja salvar estes dados filtrados. Para isso usa-se:

```
> write.table(r10,'cachorros_maior_que_10.txt', row.names=FALSE, quote = FALSE)
```



O parâmetro `row.names=FALSE` faz com que índices para as linhas não sejam gravados no arquivo. Já o parâmetro `quote=FALSE` evita que os dados sejam gravados com aspas.

10.2 Leitura e gravação de planilhas do Excel

- **Leitura**

Existem muitas formas de ler dados de uma planilha do Excel no R. Muitos pacotes fazem essa leitura com funções diferentes.
Será utilizada a função `read_excel()` do pacote *readxl*. Essa função é capaz de ler dados com a extensão .xlsx e .xls. O argumento obrigatório da função é o nome do arquivo e como argumento opcional tem-se o nome (ou número) da planilha que deseja ser lida (caso este argumento não seja informado, então a primeira aba é lida).

Seja a seguinte planilha, chamada *produtos.xlsx*, salva no diretório de trabalho:

	A	B	C	D
1	Produto	Peso	Fragil	Fabricante
2	1	0,5	1	A
3	2	5	0	B
4	3	20	1	A
5	4	52,4	0	C
6	5	12	0	C
7	6	10	1	B

Para lê-la no R:

```
> prods<-read_excel("produtos.xlsx")
> prods
# A tibble: 6 × 4
  Produto  Peso Fragil Fabricante
    <dbl> <dbl>  <dbl> <chr>
1       1   0.5      1 A
2       2   5        0 B
3       3  20        1 A
```

```
4       4  52.4       0 C
5       5  12         0 C
6       6  10         1 B

> str(prods)
tibble [6 × 4] (S3: tbl_df/tbl/data.frame)
 $ Produto   : num [1:6] 1 2 3 4 5 6
 $ Peso      : num [1:6] 0.5 5 20 52.4 12 10
 $ Fragil    : num [1:6] 1 0 1 0 0 1
 $ Fabricante: chr [1:6] "A" "B" "A" "C" ...
```

Obs.: *Tibbles* são uma espécie de dataframe e são mais usadas por analistas de dados. *Tibbles* podem ser facilmente convertidas em dataframes por meio do comando `as.data.frame()`.

- **Gravação**

No pacote *writexl*, existe a função `write_xlsx()`, que tem como argumento obrigatório a variável a ser gravada e o nome do arquivo a ser gravado.
Carregando o conjunto de dados *iris* do R e gravando em uma planilha do Excel:

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

A planilha gravada será da forma:

	A	B	C	D	E
1	**Sepal.Length**	**Sepal.Width**	**Petal.Length**	**Petal.Width**	**Species**
2	5,1	3,5	1,4	0,2	setosa
3	4,9	3	1,4	0,2	setosa
4	4,7	3,2	1,3	0,2	setosa
5	4,6	3,1	1,5	0,2	setosa
6	5	3,6	1,4	0,2	setosa
7	5,4	3,9	1,7	0,4	setosa
8	4,6	3,4	1,4	0,3	setosa
9	5	3,4	1,5	0,2	setosa
10	4,4	2,9	1,4	0,2	setosa
11	4,9	3,1	1,5	0,1	setosa
12	5,4	3,7	1,5	0,2	setosa
13	4,8	3,4	1,6	0,2	setosa
14	4,8	3	1,4	0,1	setosa
15	4,3	3	1,1	0,1	setosa
16	5,8	4	1,2	0,2	setosa
17	5,7	4,4	1,5	0,4	setosa
18	5,4	3,9	1,3	0,4	setosa
19	5,1	3,5	1,4	0,3	setosa
20	5,7	3,8	1,7	0,3	setosa
21	5,1	3,8	1,5	0,3	setosa
22	5,4	3,4	1,7	0,2	setosa
23	5,1	3,7	1,5	0,4	setosa
24	4,6	3,6	1	0,2	setosa
25	5,1	3,3	1,7	0,5	setosa
26	4,8	3,4	1,9	0,2	setosa
27	5	3	1,6	0,2	setosa

Obs.: Fatores (*factors*), que apareceram na estrutura da variável *Species* do conjunto *iris*, são variáveis categóricas, ou seja, pertencem a um número limitado de categorias (no exemplo, são 3 categorias: *setosa*, *versicolor* e *virginica*). Outros exemplos de variáveis categorias são: estações do ano, meses, cores, ...

10.3 Leitura e gravação de arquivos binários

Arquivos binários são muito utilizados na prática, pois ocupam menos espaço na memória do computador do que arquivos de texto, por exemplo.
Um arquivo binário não é formado por caracteres *ascii* e sim por *bytes* em um dado formato que um programa específico lê. Entretanto, cada tipo de arquivo binário é um tipo diferente! Assim, os arquivos binários que serão vistos nesta apostila são específicos para o R.

- **Gravação**

Para gravar um arquivo binário no R, usa-se a função `save()`, que tem como parâmetros obrigatórios as variáveis que se deseja salvar e o nome do arquivo que será gravado. Portanto, diversas variáveis podem ser salvas em um mesmo arquivo binário.

```
> a<-15
> nomes<-c("João","Maria","José","Ana")
> library('readxl')
> prods<-read_excel("produtos.xlsx")
> i<-read_excel("Iris.xlsx")
> ls()
[1] "a"      "i"      "nomes" "prods"
> save(nomes,prods,a,file='variaveis.bin') # salvando todas as variáveis, exceto i
```

- **Leitura**

Para fazer a leitura de um arquivo binário, usa-se a função `load()`, que tem como parâmetro obrigatório o nome do arquivo binário a ser lido. Essa função fará a leitura de um arquivo salvo com a função `save()`.

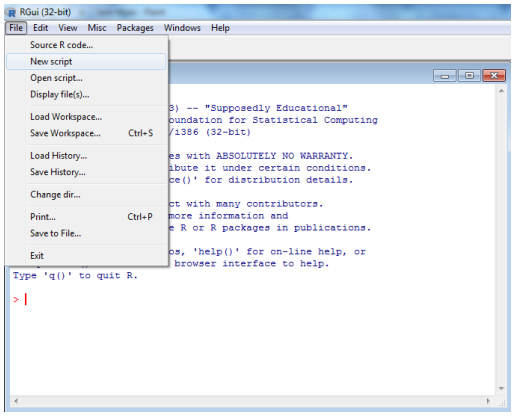
```
> rm(list=ls())
> ls()
character(0)
> load('variaveis.bin')
> ls()
[1] "a"      "nomes" "prods"
```

11 ESTRUTURAS CONDICIONAIS E LAÇOS

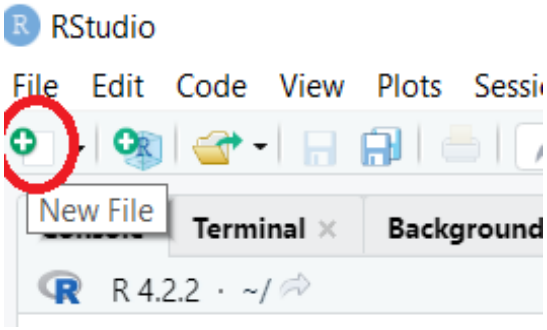
Tudo o que será visto adiante, será escrito em um editor de texto, pois é incomum (mas não é regra) se usar os comandos a seguir diretamente no *console* do R.
Editores de texto facilitam a visualização e permitem a indentação do código, permitem uma flexibilidade para percorrer entre as linhas do código, facilitam a correção de eventuais erros, podem ser salvos e recuperados posteriormente, entre outras vantagens.

O R tem um editor de texto próprio, que pode ser acessado em *File >> New script* ou *New File* (no RStudio).

R



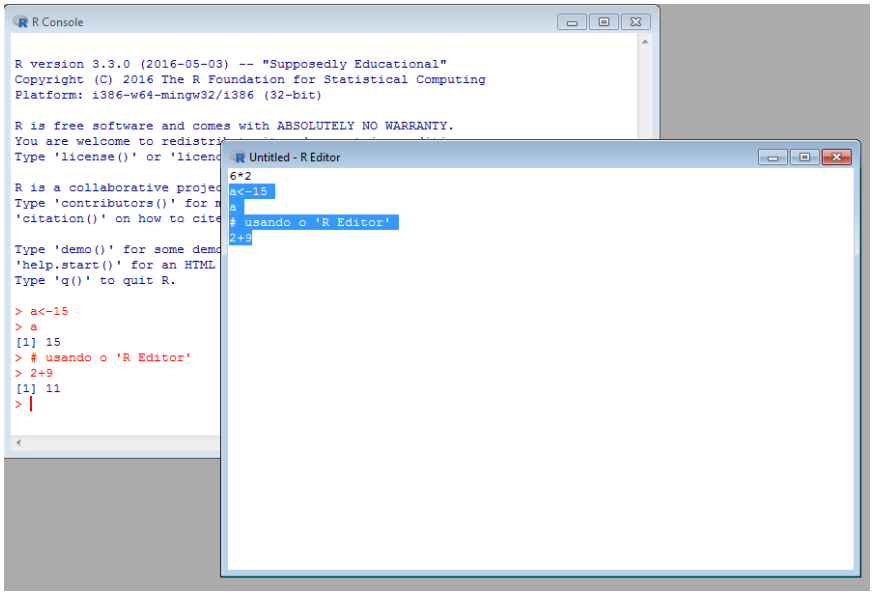
RStudio



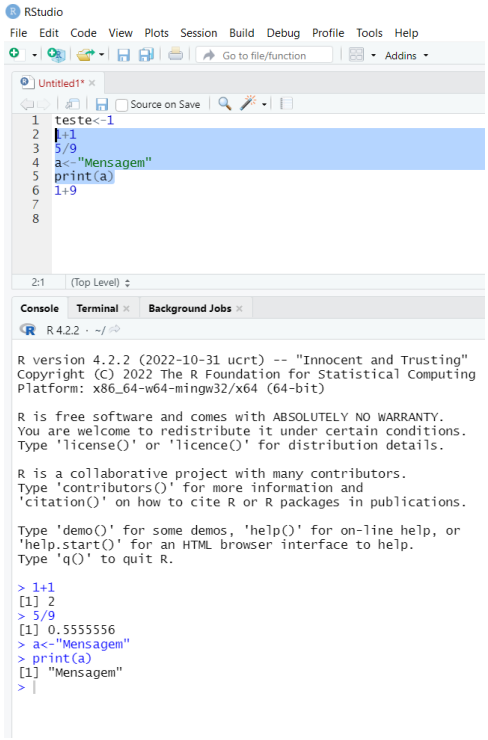
O *script* pode ser escrito normalmente neste editor e para executá-lo basta copiá-lo e colá-lo no *console* do R. A execução do código se dará na ordem em que as linhas de comando aparecem no arquivo, ou seja, de cima para baixo.

O *script* também pode ser transferido para o R por meio da seleção das linhas que se deseja transferir e a combinação de teclas *Ctrl+R* (no R) ou *Ctrl+Enter* (no RStudio), assim as linhas selecionadas são copiadas, coladas e executadas no R.

R



RStudio



Um *script* é salvo com a extensão .R, assim também existe a opção de fazer a execução desse *script* com a função `source()`. Essa função recebe como parâmetro o nome do arquivo a ser executado no R (juntamente com o caminho em que esse arquivo se encontra – no Windows, ao obter o caminho do arquivo, tem que trocar \ por /). Por padrão, o `source()` não mostra na tela nem os comandos executados nem seus resultados, porém todo o código é executado normalmente como se você copiasse e colasse o *script*.
Por exemplo, suponha que se tenha salvo algum *script* em um arquivo chamado *PrimeiroPrograma.R* no diretório C:\Users\UFPR e deseja executá-lo.

```
> source('C:/Users/UFPR/PrimeiroPrograma.R')
```

Assim, o *script PrimeiroPrograma.R* é transferido para o R e executado simultaneamente (apesar de não aparecer o código, nem o resultado da execução).

O bloco de notas também pode ser usado como editor de texto, porém os comandos de transferência *Ctrl+R* (no R) ou *Ctrl+Enter* (no RStudio) não funcionam, e para salvar um arquivo tem que obrigatoriamente colocar após o nome do arquivo a extensão .R

Nesta apostila é fácil identificar quando se está digitando diretamente no *console* do R: quando o símbolo > está à frente dos comandos.

Agora sim vamos para as estruturas condicionais.

11.1 Estruturas condicionais

São comandos que executam determinadas ações desde que alguma(s) condição(ões) seja(m) verdadeira(s).

```
if(condição){
    ...
}
```

O bloco de comandos dentro do `{...}` só é executado se `condição` for `TRUE`, caso contrário, nada é executado.

Exemplo: Verificar se um número é positivo. Se sim, trocar seu sinal e imprimir o novo número.

```
x<-45
if(x>0){
    x<--x
    print(x)
}
```

Resultado (quando colado no *console* do R):
```
[1] -45
```

Obs.: se dentro do `{...}` só existir apenas uma linha de comando ou se você colocar todas as linhas de comando em apenas uma linha (nesse caso, separadas por ponto e vírgula), então as chaves são opcionais; caso contrário elas são obrigatórias.

Observe a indentação no código anterior. Ela é opcional no R (em algumas linguagens de programação ela é obrigatória), mas deixa o código mais legível e amigável. Consiste em dar um recuo na margem à esquerda das linhas de código que fazem parte das ações do `if`.

Exemplo: Verificar se um número é positivo. Se sim, trocar seu sinal.

```
x<-45
if(x>0){
    x<--x
}
```

ou

```
x<-45
if(x>0) x<--x
```

```
if(condição){
    ...
} else{
    ...
}
```

Se `condição` for verdadeira, então o primeiro bloco de comandos `{...}` é executado e o segundo não. Caso `condição` for falsa, então o primeiro bloco de comandos não é executado e o segundo sim.

Exemplo: Verificar se um número é par ou ímpar.

```
x<-12
if(x%%2==0){
    print('é par')
}else {
    print('é ímpar')
}
```

Resultado (quando colado no *console* do R):
```
[1] "é par"
```

ou

```
x<-12
if(x%%2==0) print('é par') else print('é ímpar')
```

Resultado (quando colado no *console* do R):
```
[1] "é par"
```

```
ifelse(condição,ação se condição for verdadeira, ação se condição for falsa)
```

```
> x<--3
> ifelse(x<0,'x é negativo','x é zero ou positivo')
[1] "x é negativo"
> x<-0
> ifelse(x<0,'x é negativo','x é zero ou positivo')
[1] "x é zero ou positivo"
> x<-65
> ifelse(x<0,'x é negativo','x é zero ou positivo')
[1] "x é zero ou positivo"
```

Esta estrutura permite a atribuição do resultado em uma variável:

```
> y<-ifelse(15<10,1,-1)
> y
[1] -1
```

```
if(condição1){
    ...
} else if(condição2){
    ...
} else{
    ...
}
```

Se `condição1` for verdadeira, então o primeiro bloco de comandos `{...}` é executado e nem o segundo nem o terceiro bloco são executados. Se `condição1` for falsa e `condição2` for verdadeira, então somente o segundo bloco de comandos é executado e os demais não. Entretanto, se `condição1` e `condição2` forem ambas falsas, então somente o terceiro bloco de comandos é executado.

Exemplo: Verificar se um número é positivo, negativo ou zero.

```
x<-7
if(x<0){
    print('é negativo')
} else if(x==0){
    print('é zero')
} else{
    print('é positivo')
}
```

Resultado (quando colado no *console* do R):
```
[1] "é positivo"
```

ou

```
x<-0
if(x<0) print('é negativo') else if(x==0) print('é zero') else print('é positivo')
```

Resultado (quando colado no *console* do R):
```
[1] "é zero"
```

11.2 Laços

Também chamados de ciclos ou *loops*, esta estrutura permite repetir um bloco de comandos um número específico de vezes ou até que uma certa condição se torne falsa.

for( ){...}

É usado para repetir um bloco de comandos um determinado número de vezes. Usa-se uma variável como contador, que varia de um valor inicial até um valor final.

Exemplo: Imprimir os 8 primeiros números inteiros maiores do que zero e somá-los.

```
soma<-0
for(i in 1:8){ # i é o contador e varia de 1 a 8
    soma<-soma+i
    print(i)
}
```

Resultado (quando colado no *console* do R):

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8

> soma
[1] 36
```

Neste exemplo, o contador `i` assume inicialmente o valor 1, e os dois comandos (`soma<-soma+i` e `print(i)`) são executados com `i` valendo 1. Depois `i` assume o valor 2, e assim por diante, até que `i` assume o último valor, isto é, 8. Após assumir o valor 8, o laço é encerrado.
A variável `soma` inicialmente vale 0, e a cada iteração (nome que se dá a cada passagem pelo laço), a variável é incrementada pelo valor de `i`. Assim, ela acumula parcialmente o valor da soma (ela inicia com 0, depois vale 0+1, depois vale 0+1+2, e assim por diante, até valer 0+1+2+3+4+5+6+7+8=36).

Não necessariamente o contador deve variar em uma sequência regular de passo 1. Ele pode variar em números contidos em um vetor numérico. Veja o exemplo a seguir:

```
for(i in c(-50,4,2.2,pi)) print(i)
```

Resultado (quando colado no *console* do R):

```
[1] -50
[1] 4
[1] 2.2
[1] 3.141593
```

```
soma<-0
for(i in c(-50,4,2.2,pi)) soma<-soma+i

> soma
[1] -40.65841
```

Para forçar a saída de um laço antes do término natural dele, usa-se o comando `break` dentro de um `if`.

```
for(i in 1:8){
   if (i>5) break
   print(i)
}
```

Resultado (quando colado no *console* do R):

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

```
while(condição){
    ...
}
```

Enquanto `condição` é verdadeira, o bloco de comandos `{...}` é executado. Quando a `condição` se tornar falsa, o bloco de comandos não será mais executado. Assim, o código dentro do laço deve ser capaz de alterar `condição` para falsa em algum momento, pois caso contrário, se gerará um laço infinito.

Exemplo: somar todos os números múltiplos de 5 entre 10 e 100.

```
soma<-0
i<-10
while(i<=100){
    soma<-soma+i
    i<-i+5
}
soma
```

Resultado (quando colado no *console* do R):
```
[1] 1045
```

Nos exemplos anteriores, a variável `soma` acumulava o resultado de um somatório. Caso cada soma parcial fosse requerida, ela poderia ser alocada em um vetor. Esse vetor deve ser inicializado e cada soma parcial deve ser concatenada a ele. A inicialização desse vetor deve ser feita por meio do comando `NULL`.

```
somas_parciais<-NULL
soma<-0
i<-10
while(i<=100){
    soma<-soma+i
    somas_parciais<-c(somas_parciais,soma)
    i<-i+5
}
somas_parciais
```

Resultado (quando colado no *console* do R):
```
 [1]  10  25  45  70 100 135 175 220 270 325 385 450 520 595 675 760 850 945 1045
```

Agora, veja uma outra forma de obter o resultado anterior:

```
somas_parciais<-NULL
soma<-0
i<-10
while(i<=100){
    soma<-soma+i
    somas_parciais[i]<-soma
    i<-i+5
}
somas_parciais
```

Resultado (quando colado no *console* do R):
```
 [1]   NA   NA   NA   NA   NA   NA   NA   NA   NA   10   NA   NA   NA   NA   25   NA
  NA   NA   NA   45   NA   NA   NA   NA   70   NA   NA   NA   NA
[30] 100   NA   NA   NA   NA  135   NA   NA   NA   NA  175   NA   NA   NA   NA  220
   NA   NA   NA   NA  270   NA   NA   NA   NA  325   NA   NA   NA
[59]   NA  385   NA   NA   NA   NA  450   NA   NA   NA   NA  520   NA   NA   NA   NA
  595   NA   NA   NA   NA  675   NA   NA   NA   NA  760   NA   NA
[88]   NA   NA  850   NA   NA   NA   NA  945   NA   NA   NA   NA 1045
```

Observe que o resultado não é o mesmo, pois a variável `i`, que vai de 5 em 5, está sendo usada para indexar o vetor. As posições que não são múltiplas de 5 recebem `NA`.
Para obter o resultado esperado, usa-se uma outra variável `j` como indexadora de posições:

```
somas_parciais<-NULL
soma<-0
i<-10
j<-1
while(i<=100){
    soma<-soma+i
    somas_parciais[j]<-soma
    i<-i+5
    j<-j+1
}
somas_parciais
```

Resultado (quando colado no *console* do R):

```
[1] 10 25 45 70 100 135 175 220 270 325 385 450 520 595 675 760 850 945 1045
```

O comando `c()` para concatenar variáveis é muito útil, porém deve ser usado com cautela. Veja o exemplo a seguir: gerar os 100000 primeiros termos da sequência $\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \ldots$.

Este exemplo será feito de duas formas e calcular-se-á o tempo de execução.

```
v1<-NULL
t0<-Sys.time()
for (i in 1: 100000){
   v1<-c(v1,i/(i+1))
}
t1<-Sys.time()
difftime(t1,t0,units="secs")
Time difference of 43.60956 secs

v2<-NULL
t0<-Sys.time()
for (i in 1: 100000){
   v2[i]<-i/(i+1)
}
t1<-Sys.time()
difftime(t1,t0,units="secs")
Time difference of 0.1102011 secs

> identical(v1,v2)
[1] TRUE
```

Ou seja, o resultado é o mesmo, entretanto o tempo computacional é muito diferente. Assim, sempre que possível, é aconselhável evitar o concatenador (nesse exemplo foi fácil a substituição, entretanto nem sempre é possível).

Obs.: A função `identical()` verifica se uma variável (vetor, matriz, dataframe,...) é idêntica a outra.

1) Crie as seguintes variáveis com seus respectivos valores: $a\_1 = 1$, $a\_2 = 2$, ..., $a\_500 = 500$. Dica: use a função `assign()`.

2) Dado um número x, calcular a soma dos 10 primeiros termos de: $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$

3) Dado um número inteiro *n*, crie uma variável que receba 1 se *n* é par, -1 se *n* é ímpar ou 0 se *n* é zero.

4) Como você calcularia o valor absoluto de um número, caso não conhecesse a função `abs()`?

5) Use o comando `while()` para criar uma rotina que encontra a raiz real de uma função pelo método da Bissecção.

11.3 Evitando laços: a "família" *apply

O R é uma linguagem vetorial e laços podem e **devem** ser substituídos por outras formas de cálculo sempre que possível. Usualmente usa-se as funções `apply()`, `sapply()`, `tapply()` e `lapply()` para implementar cálculos de forma mais eficiente. Seguem alguns exemplos.

- `apply()` para uso em matrizes ou data-frames;
- `tapply()` para uso em vetores, sempre retornando uma lista;
- `sapply()` para uso em vetores, simplificando a estrutura de dados do resultado se possível (para vetor ou matriz);
- `mapply()` para uso em vetores, versão multivariada de `sapply()`;
- `lapply()` para ser aplicado em listas.

12 ESCREVENDO SUAS PRÓPRIAS ROTINAS E FUNÇÕES

O R é uma excelente ferramenta para cálculos matemáticos e estatísticos, sendo que existem milhares de funções prontas para serem utilizadas. Porém, sempre vai existir a necessidade, por parte do usuário, de utilizar uma função de uma maneira diferente, específica para um determinado propósito. Além disso, é comum no ambiente computacional, se desenvolver sistemas operacionais com *scripts* complexos e que demandam uma eficiente implementação computacional. Assim, será visto como escrever seus próprios *scrips* e funções

12.1 Escrevendo funções

Funções são escritas da forma:

```
nome_funcao<-function(parâmetros){...}
```

`nome_funcao` é o nome que sua função vai receber, `parâmetros` são os argumentos de entrada da sua função e são separados por vírgula (parâmetros são opcionais) e `{...}` é o corpo da função, onde vão os comandos.
Por padrão, uma função retorna a última variável atribuída. Para retornar qualquer variável desejada (não necessariamente a última atribuição), usa-se `return(nome_variavel)`. A variável retornada pode ser qualquer tipo de variável (números, vetores, matrizes, dataframes,...).

Exemplos:

- Ex. 1: Imprimir uma mensagem de boas vindas ao R.

```
boas_vindas<-function(){
      print('Boas vindas ao R')
}
```

ou (lembrando que apenas uma linha de comando as chaves são opcionais):

```
boas_vindas<-function() print('Boas vindas ao R')
```

Note que esta função não tem parâmetros – parênteses vazio, o que significa que o resultado dela é independente de dados externos e o sempre dará o mesmo resultado.
Neste exemplo, nenhuma variável foi atribuída e quando a função for chamada, apenas será mostrada a mensagem.

```
> boas_vindas() #chamando a função no R
[1] "Boas vindas ao R"
```

- Ex. 2: Receber um número e dobrar seu valor.

```
dobra<-function(x) y<-x*2
```

Chamando a função no R:

```
> dobra(5)
```

Note que nada aconteceu, pois a função foi chamada, mas o seu resultado não foi atribuído a nenhuma variável. Assim, para que se tenha de fato o resultado da chamada da função, é necessário atribuir a uma variável.

```
> res<-dobra(5)
> res
[1] 10
```

Observe que esta função tem um parâmetro (x), e dependendo do valor informado para ele o resultado será outro:

```
> res<-dobra(3.3)
> res
[1] 6.6
```

O uso do `return()` neste caso é opcional, pois apenas a variável y foi criada dentro da função e é ela quem está sendo retornada. No caso de haver mais do que uma variável criada e não necessariamente a última variável criada é a que deva ser retornada, então o `return()` é obrigatório:

```
dobra<-function(x){
      y<-x*2
      x<-x # absolutamente nada está sendo feito aqui
      return(y)
}
```

Se o `return()` não tivesse sido usado, a função retornaria o valor da variável x, e não o seu dobro.

- Ex. 3: Receber um vetor e retornar a soma e a média de seus elementos.

```
soma_media<-function(v){
      s<-sum(v)
      m<-mean(v)
      return(list(soma=s,media=m))
}
```

No R:

```
> v<-c(-15,0,2,0.6,14,-2.5,9)
> res<-soma_media(v)
> res
$soma
[1] 8.1

$media
[1] 1.157143
```

No exemplo anterior, para retornar as duas variáveis (soma e média) foi criada uma lista. Ao invés da lista, um vetor de duas posições poderia ter sido criado, por exemplo, com a soma na primeira posição do vetor e a média na segunda posição.

12.2 Escrevendo rotinas

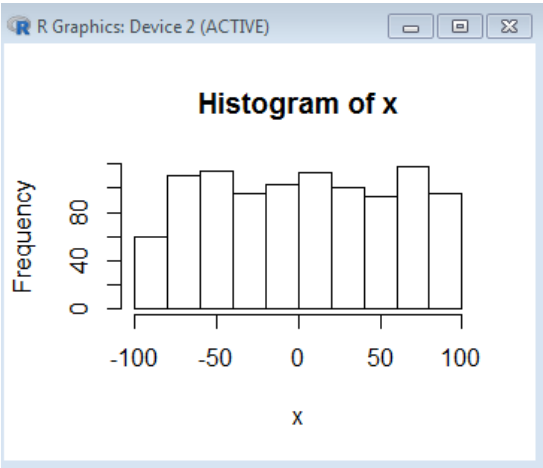
Rotinas ou *scripts* são linhas de códigos compostas por atribuições de variáveis, criação e chamada de funções, leitura e gravação de dados, remoção de variáveis,...
Rotinas são escritas em editores de texto e podem ser salvas com a extensão .R

Exemplo: Gerar 1000 números inteiros aleatoriamente entre -90 a 100, calcular a média e o desvio padrão da amostra gerada e fazer o histograma dos dados.

```
x<-round(runif(1000,-90,100),0)

media_desvio<-function(v){
      m<-mean(v)
      dp<-sd(v)
      return(list(media=m,desviopadrao=dp))
}

r<-media_desvio(x)
r$media
r$desviopadrao

hist(x)
```

Resultado (quando colado no *console* do R):

```
[1] 5.363
[1] 55.4345
```



1) O que acontece se você digitar no R o nome de uma função sem passar nenhum parâmetro a ela e sem os parênteses?

2) Escreva uma função que retorne as raízes reais (se existirem) da equação de segundo grau $ax^2 + bx + c = 0$, em que são fornecidos os coeficientes a, b e c.

3) Escreva uma função que retorne um vetor contendo os *n* primeiros números primos, sendo *n* fornecido como parâmetro.

4) Dado um número inteiro positivo *n,* escreva uma função que calcule o fatorial de *n.*

5) Sabe-se que a multiplicação das matrizes A e B é dada pelo comando `A%*%B`. Suponha que esse comando não é conhecido. Escreva uma função que faça essa multiplicação.

---

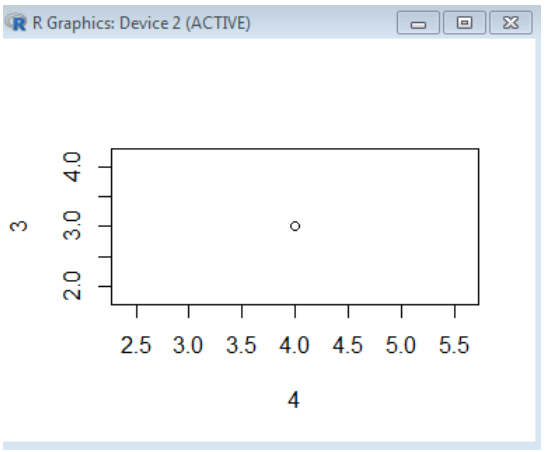
13 GRÁFICOS

Existem muitos tipos de gráficos no R, porém são comuns gráficos de dispersão, boxplot, de barras, entre outros. Além de gráficos simples, o R permite a criação de figuras, gráficos em 3D, animações. Para mais informações sobre tipos de gráficos no R, visite o site http://rgraphgallery.blogspot.com.br/2013

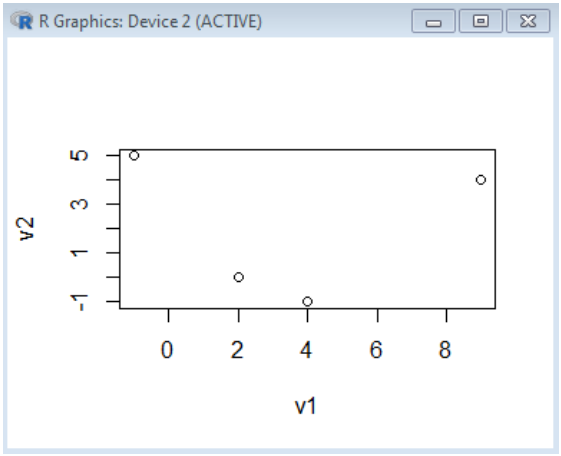
13.1 Gráfico de dispersão

Gráficos de dispersão (pontos) são os mais comuns e fáceis de serem plotados. São feitos por meio da função `plot()` que recebe, no mínimo, dois argumentos: x e y, onde estes são dois números ou dois vetores de mesmo tamanho.

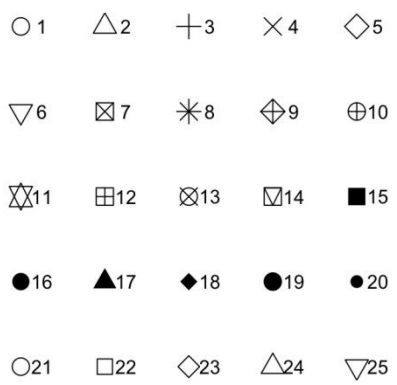
```
> plot(4,3)
```




```
> v1<-c(2,-1,4,9)
> v2<-c(0,5,-1,4)
> plot(v1,v2)
```

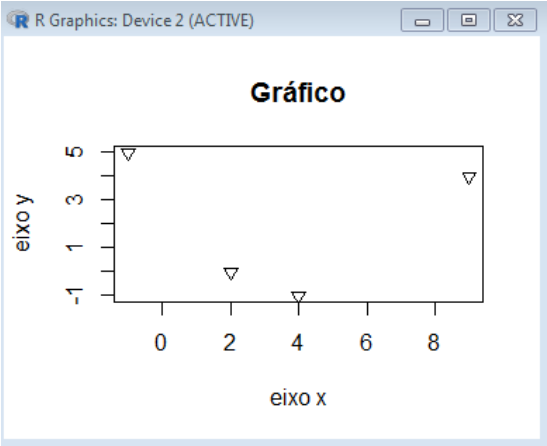


Observe que por padrão os pontos são plotados com o símbolo `'o'` e os eixos recebem os nomes das variáveis plotadas. O símbolo pode ser alterado com o parâmetro `pch=n`, onde `n` é um número inteiro ou um símbolo, conforme figura:



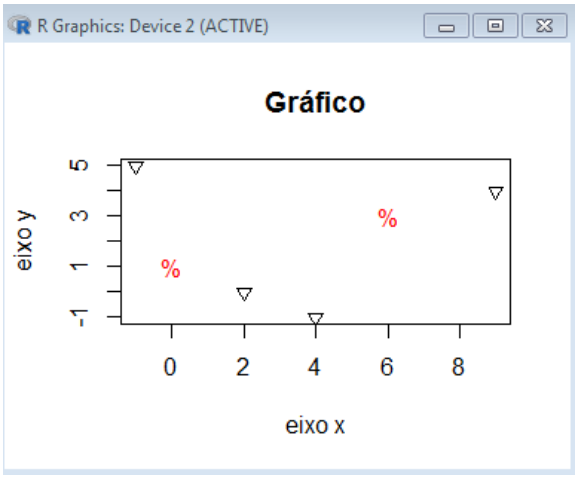
Os nomes dos eixos podem ser alterados com os parâmetros `xlab` e `ylab`. Um título pode ser acrescentado com o parâmetro `main`. Todos estes parâmetros são alterados dentro da função `plot()`.

```
> plot(v1,v2,pch=25,xlab='eixo x', ylab='eixo y', main='Gráfico')
```



Se você fizer um novo `plot`, o gráfico atual será apagado e um novo gráfico será gerado. Para acrescentar pontos ao gráfico existente, utiliza-se a função `points()` que recebe os mesmos argumentos da função `plot()`.

```
> points(c(6,0),c(3,1),pch='%',col='red')
```

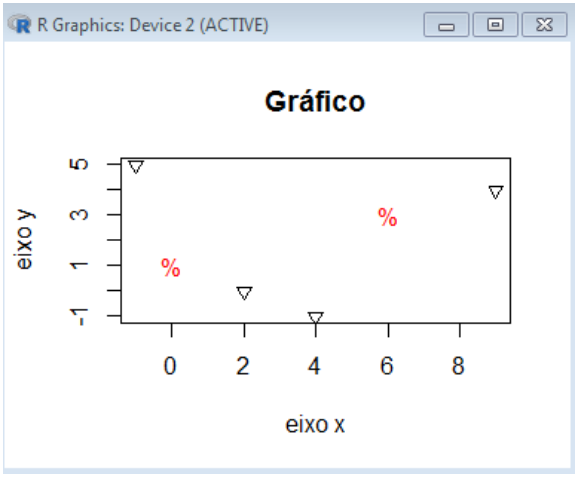


O argumento `col` permite escolher a cor que se deseja plotar uma linha, símbolo etc. Este argumento recebe o nome de uma cor ou um número, conforme figura:

Portanto, para pintar algo de vermelho, por exemplo, pode-se fazer `col='red'` ou `col=2`. Existem muito mais cores do que as apresentadas na figura anterior (teste outros valores maiores que 8).

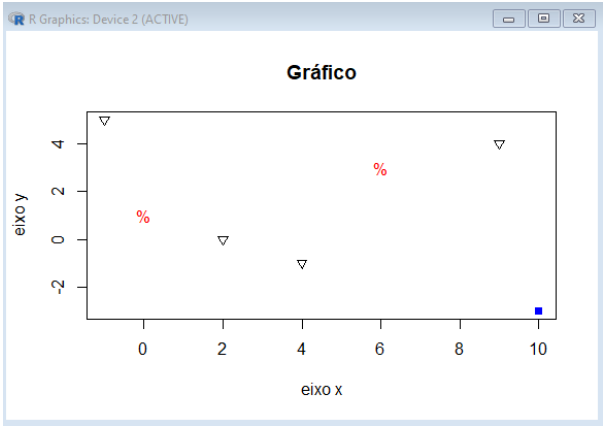
Os limites dos eixos são definidos automaticamente quando o primeiro gráfico é gerado, de modo que a área de plotagem englobe tudo o que está sendo plotado. Porém, ao tentar plotar algo fora da área delimitada pelo primeiro gráfico, nada acontece. Veja o exemplo:

```
> points(10,-3,pch=15,col='blue')
```



Para definir de maneira manual os limites dos eixos é necessário usar os parâmetros `xlim` e `ylim`, informando para cada um deles um vetor de duas posições, com os limites mínimo e máximo, respectivamente, para cada eixo. Veja o exemplo agora:

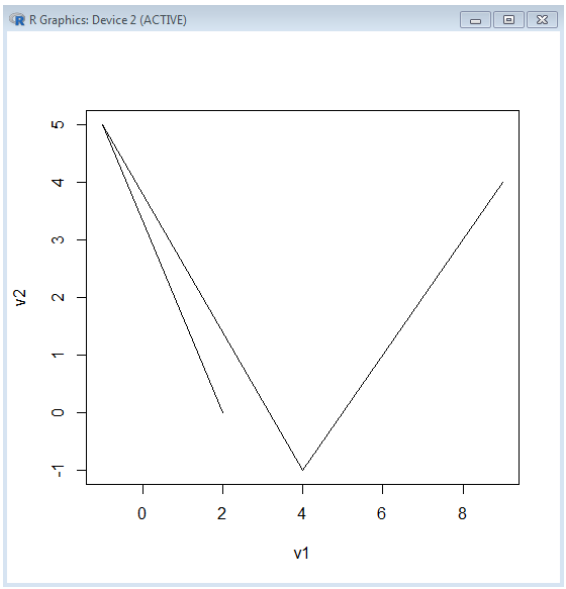
```
> plot(v1,v2,pch=25,xlab='eixo x', ylab='eixo y', main='Gráfico',xlim=c(-1,10),ylim=c(-3,5))
> points(c(6,0),c(3,1),pch='%',col='red')
> points(10,-3,pch=15,col='blue')
```



13.2 Gráfico de linha

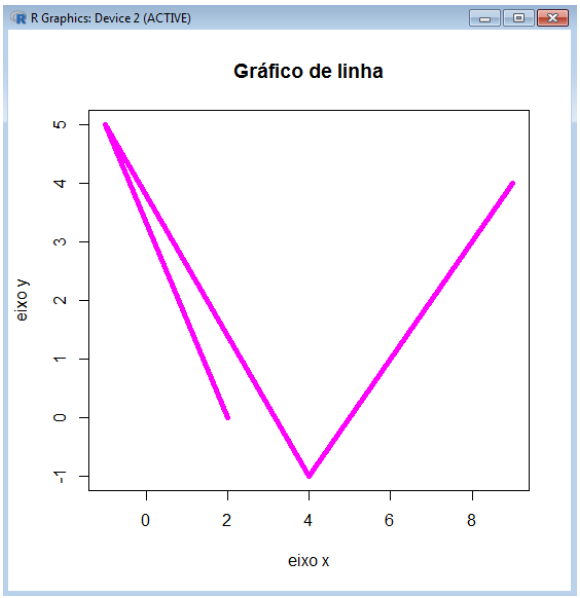
Ao invés de pontos, um gráfico pode conter linhas para representar um conjunto de dados. Uma linha pode ser feita também com a função `plot()`, porém se acrescenta como parâmetro `type='l'`, que significa que o tipo a ser plotado é uma linha (`l` = linha). O padrão é `type='p'` (`p` = points).

```
> v1<-c(2,-1,4,9)
> v2<-c(0,5,-1,4)
> plot(v1,v2,type='l')
```



É possível alterar a espessura da linha por meio do parâmetro `lwd=n`, onde `n` é um número real positivo. Por padrão, `lwd=1`. Quanto maior o valor de `n`, maior é a espessura da linha.
Alterando a aparência do gráfico anterior:

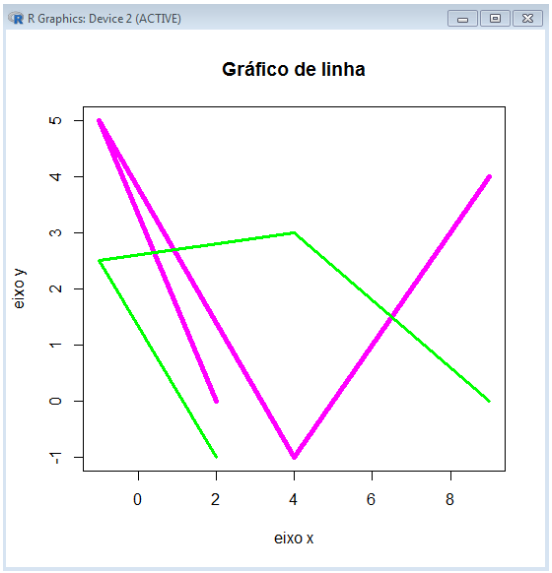
```
> plot(v1,v2,type='l',lwd=3,col='magenta',main='Gráfico de linha', xlab='eixo x', ylab='eixo y')
```



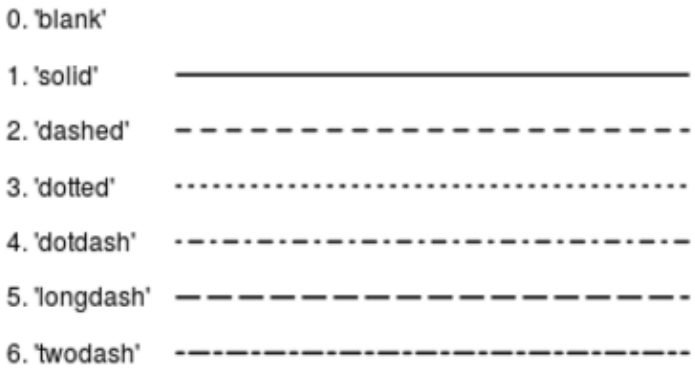
É possível acrescentar uma linha em um gráfico de linha já existente. Se você fizer um novo `plot(...,type='l')`, o gráfico atual será apagado e um novo gráfico será gerado. Para não apagar o gráfico existente, utiliza-se a função `lines()` que recebe os mesmos argumentos da função `plot()`, porém não há a necessidade de usar o argumento `type='l'`.

```
> v3<-c(-1,2.5,3,0)
```

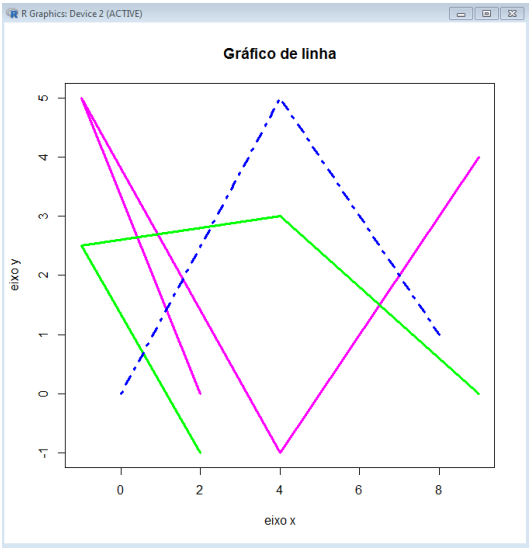
```
> lines(v1,v3, col='green',lwd=3)
```



Também é possível alterar o tipo da linha por meio do parâmetro `lty`, que recebe um número inteiro entre 0 e 6, de acordo com a figura:



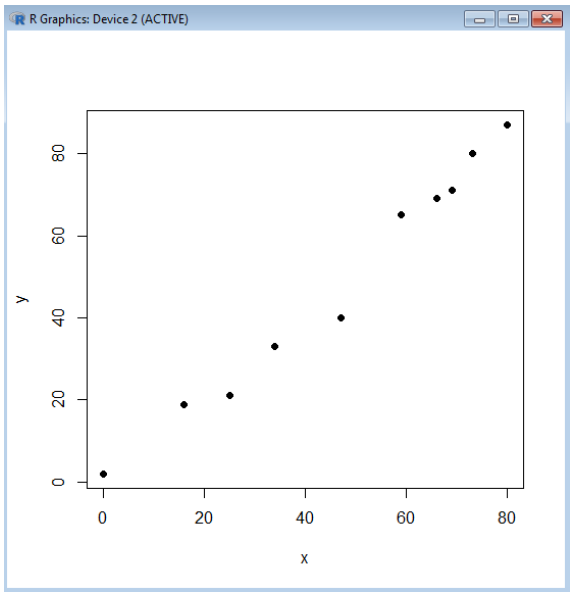

```
> lines(c(0,4,8),c(0,5,1), col='blue',lwd=3,lty=4)
```



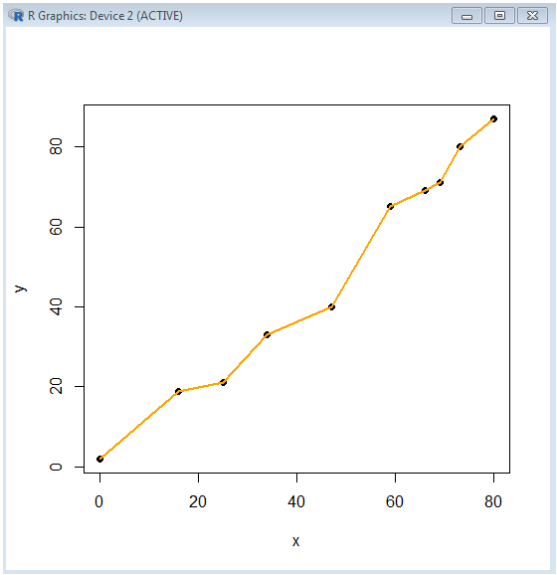
13.3 Gráfico de linha e pontos

É possível plotar linhas e pontos juntos no mesmo gráfico. Para isso, basta lembrar que o primeiro `plot()` gerará o primeiro gráfico (se `type='l'` será plotada uma linha; se o tipo não estiver definido ou `type='p'` então serão plotados pontos). Para acrescentar linhas ou pontos usa-se `lines()` ou `points()`, respectivamente. Segue um outro exemplo.

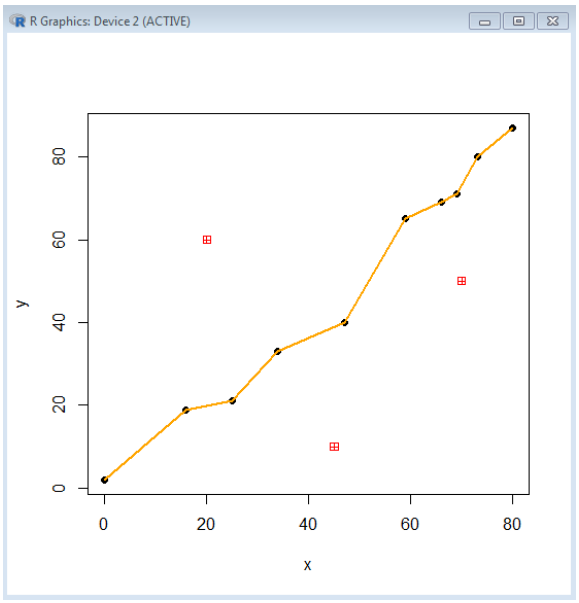
```
> x<-c(0,16,25,34,47,59,66,69,73,80)
> y<-c(2,19,21,33,40,65,69,71,80,87)
> plot(x,y,pch=16)
```



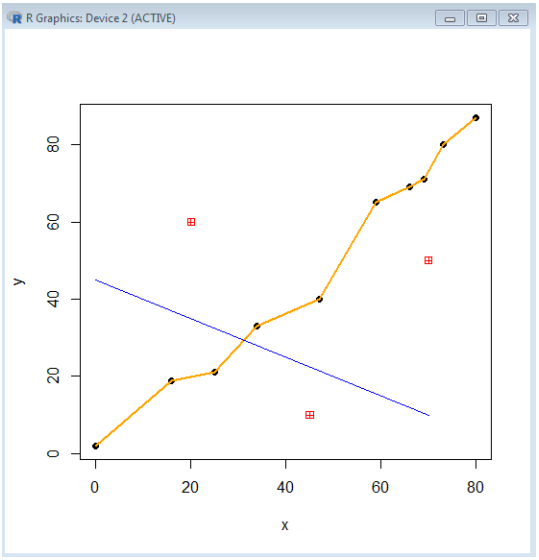

```
> lines(x,y,col='orange',lwd=2)
```




```
> points(c(20,45,70),c(60,10,50),pch=12,col='red')
```

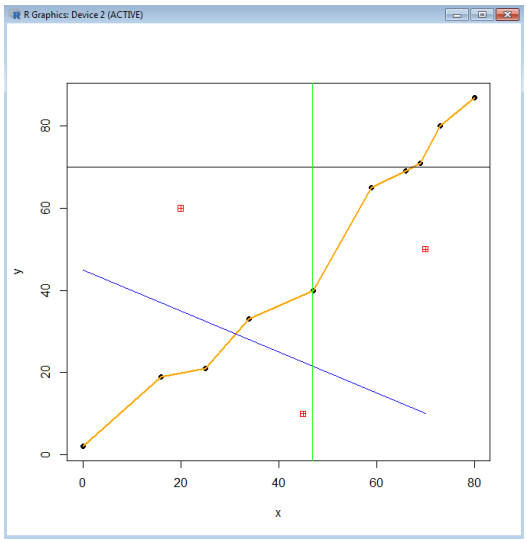



```
> lines(c(0,70),c(45,10),col='blue')
```



Para acrescentar uma linha horizontal ou vertical, basta usar a função `abline()`.

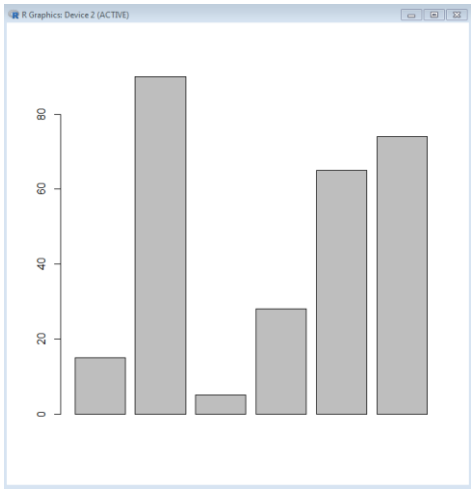
```
> abline(h=70) # o h é de horizontal
> abline(v=mean(x),col='green') # o v é de vertical
```



13.4 Gráfico de barra

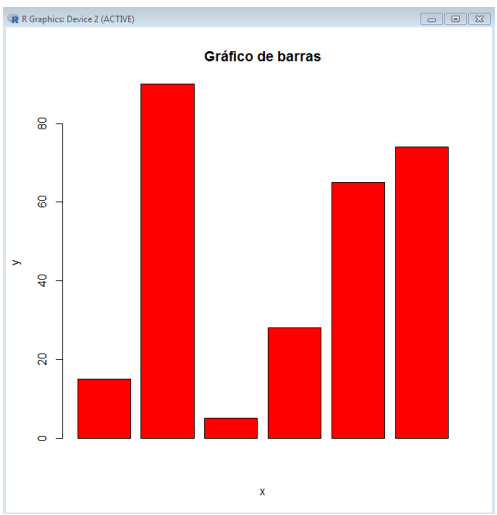
Para fazer gráfico de barras, usa-se a função `barplot()` em que o argumento é um vetor.

```
> barplot(c(15,90,5,28,65,74))
```



Pode-se alterar a aparência do gráfico:

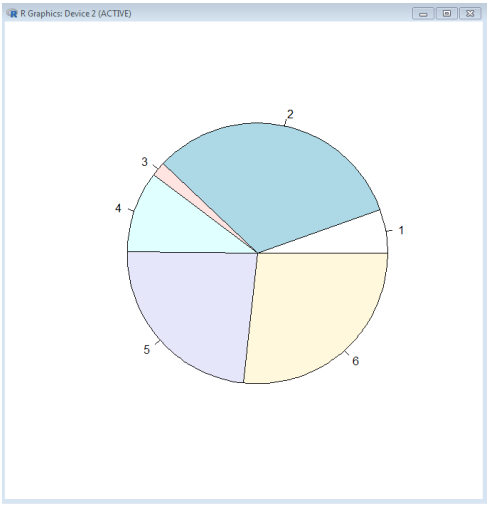
```
> barplot(c(15,90,5,28,65,74),col='red',xlab='x',ylab='y',main='Gráfico de barras')
```



13.5 Gráfico de pizza

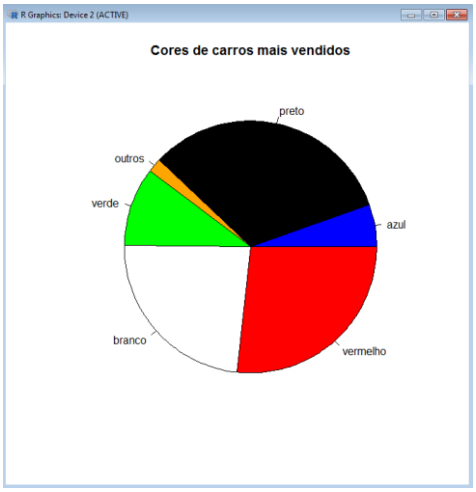
Para fazer gráfico de pizza, usa-se a função `pie()` em que o argumento é um vetor.

```
> pie(c(15,90,5,28,65,74))
```



Alterando a aparência do gráfico:

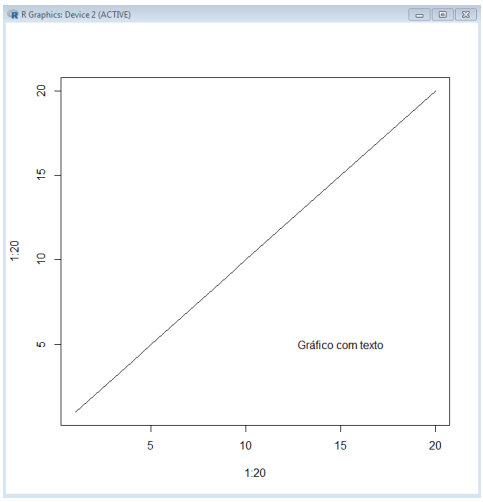
```
>pie(c(15,90,5,28,65,74),labels=c('azul','preto','outros','verde','branco','vermelho')
,col=c('blue','black','orange','green','white','red'),main='Cores de carros mais
vendidos')
```



13.6 Inserir texto em gráficos

Para inserir texto em gráficos, usa-se a função `text()`. A sintaxe é: `text(coord_x,coord_y,'texto')`, em que `coord_x` e `coord_y` são as coordenadas cartesianas onde `'texto'` deverá ser inserido.
Criando um gráfico simples para exemplificar.

```
> plot(1:20,1:20,type='l')
> text(15,5,'Gráfico com texto')
```

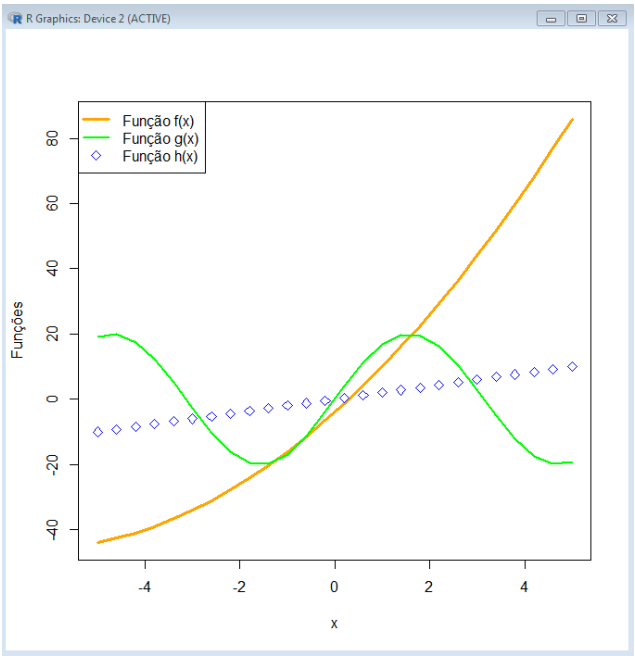


13.7 Legenda em gráficos

Legendas são inseridas em gráficos por meio da função `legend`. Os parâmetros obrigatórios são a localização da legenda (`'bottomright'`, `'bottom'`, `'bottomleft'`, `'left'`, `'topleft'`, `'top'`, `'topright'`, `'right'` ou `'center'`) e a escrita que vai na legenda. Há outros parâmetros adicionais que são essenciais para o entendimento da legenda, tais como cores, linhas etc.

Exemplo:

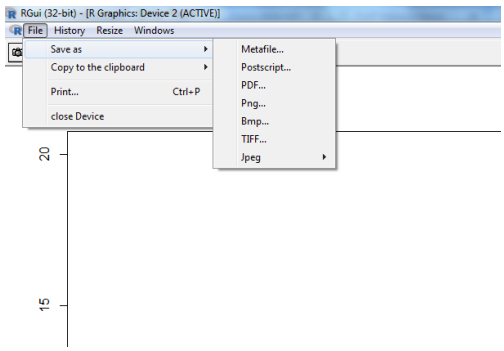
```
> x<-seq(-5,5,by=0.4)
> f<-function(x) x^2+13*x-4
> g<-function(x) 20*sin(x)
> h<-function(x) 2*x
> plot(x,f(x),type='l',lwd=3,col='orange',ylab='Funções')
> lines(x,g(x),lwd=2,col='green')
> points(x,h(x),col='blue',pch=5)
> legend('topleft',c('Função f(x)','Função g(x)','Função h(x)'),lwd=c(3,2,NA),
lty=c(1,1,NA),pch=c(NA,NA,5),col=c('orange','green','blue'))
```



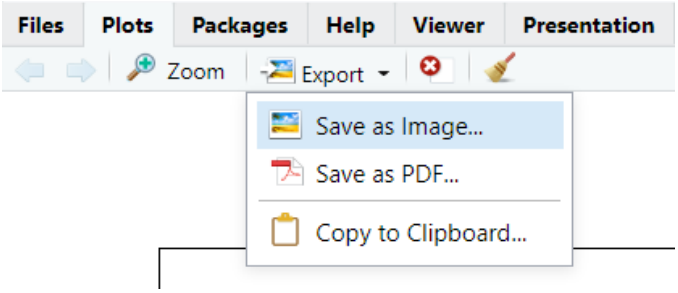
13.8 Salvando gráficos

- No Windows ou MacOS:

R



RStudio



- No Linux/Windows/MacOS:

```
> pdf('grafico.pdf')
> plot(1:20,1:20,type='l')
> dev.off()
```

Todos os comandos gráficos digitados após o `pdf('grafico.pdf')` e antes do `dev.off()` serão executados e consequentemente gravados no arquivo gráfico.pdf.
As outras extensões são: `png()`, `jpeg()` e `postscript()`.

Salvando um gráfico desta forma, automaticamente ele é salvo na pasta que se está trabalhando. Pode-se alterar o local passando todo o caminho do diretório que se deseja salvar a imagem.
Exemplo (Windows): `pdf('C:/Users/UFPR/Desktop/grafico.pdf')`

1) Sejam os seguintes vetores:
```
riqueza<-c(15,18,22,24,25,30,31,34,37,39,41,45)
area<-c(2,4.5,6,10,30,34,50,56,60,77.5,80,85)
```
Faça o gráfico de dispersão dos dados, plotando o símbolo `'*'` na cor laranja. Nomeie os eixos e dê um título ao gráfico. Depois acrescente uma linha pontilhada que passa pelos pontos.

2) Agora suponha que os mesmos dados do exercício 1 estão dispostos da seguinte forma:
```
riqueza<-c(24,37,22,18,30,34,41,39,45,15,25,31)
area<-c(10,60,6,4.5,34,56,80,77.5,85,2,30,50)
```
que representam os mesmos dados, porém não estão ordenados da menor para a maior riqueza. A partir destes dois vetores, como você faria para obter o conjunto de dados do exercício 1?
Plote seu novo conjunto de dados e verifique se o gráfico ficou igual ao do exercício 1.

3) Plote o gráfico x versus y, em que x é um vetor com 100 valores igualmente espaçados no intervalo $[-1,1]$ e $y = sen(x)e^{-x}$. Trace uma linha de cor azul e nomeie adequadamente os eixos.

---

14 ESTATÍSTICA

Com certeza você já ouviu falar que o R é um dos *softwares* preferidos dos estatísticos. Isso porque o R é muito bom para manipulação de dados e todas as funções e testes estatísticos já estão programados e prontos para o uso. Nesta

apostila, serão vistos comandos básicos para realizar algumas das análises mais comuns usadas em manipulação de dados.

14.1 Estatística descritiva

A fim de facilitar o entendimento, considere o seguinte conjunto de dados:

```
> data(USAccDeaths)
> str(USAccDeaths)
 Time-Series [1:72] from 1973 to 1979: 9007 8106 8928 9137 10017 ...
> USAccDeaths
      Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
1973  9007  8106  8928  9137 10017 10826 11317 10744  9713  9938  9161  8927
1974  7750  6981  8038  8422  8714  9512 10120  9823  8743  9129  8710  8680
1975  8162  7306  8124  7870  9387  9556 10093  9620  8285  8466  8160  8034
1976  7717  7461  7767  7925  8623  8945 10078  9179  8037  8488  7874  8647
1977  7792  6957  7726  8106  8890  9299 10625  9302  8314  8850  8265  8796
1978  7836  6892  7791  8192  9115  9434 10484  9827  9110  9070  8633  9240
```

que representa o número de mortes por acidentes nos Estados Unidos de 1973 a 1978. Para simplificar, esta variável será chamada x.

```
x<-USAccDeaths
```

- **Média aritmética:** `mean()`

```
> mean(x)
[1] 8788.792
```

- **Mediana:** `median()`

```
> median(x)
[1] 8728.5
```

- **Moda:** no R, esta função não está programada, porém uma implementação postada no R-Help por Dr. Brian D. Ripley (e modificada um pouquinho por mim) para distribuições discretas é:

```
moda <- function(y) {
      z <- table(y)
      m <- names(z)[which.max(z)]
      return(as.numeric(m))
}

> moda(x)
[1] 8106
> x
      Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
1973  9007 [8106] 8928  9137 10017 10826 11317 10744  9713  9938  9161  8927
1974  7750  6981  8038  8422  8714  9512 10120  9823  8743  9129  8710  8680
1975  8162  7306  8124  7870  9387  9556 10093  9620  8285  8466  8160  8034
1976  7717  7461  7767  7925  8623  8945 10078  9179  8037  8488  7874  8647
1977  7792  6957  7726 [8106] 8890  9299 10625  9302  8314  8850  8265  8796
1978  7836  6892  7791  8192  9115  9434 10484  9827  9110  9070  8633  9240
```

- **Variância:** `var()`

```
> var(x)
[1] 917290.1
```

- **Desvio-padrão:** `sd()`

```
> sd(x)
[1] 957.7526
```

- **Resumo das variáveis:** `summary()`

Esta função faz um resumo da variável em questão. Ela apresenta seis medidas de posição que descrevem os dados (os valores mínimo e máximo, a média e a mediana, o primeiro e o terceiro quartis).

```
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6892    8089    8728    8789    9323   11320
```

Observe que função `summary()` não retorna os valores exatos das medidas, mas sim um arredondamento.

Valores mínimo e máximo podem ser obtidos com as funções `min()` e `max()`, respectivamente.

```
> min(x)
[1] 6892
> max(x)
[1] 11317
```

E os quartis podem ser obtidos com a função `quantile()`.

```
> quantile(x)
      0%      25%      50%      75%     100%
 6892.00  8089.00  8728.50  9323.25 11317.00

> quantile(x,0.75)
      75%
 9323.25
```

Com a função `quantile()`, qualquer percentil pode ser obtido:

```
> quantile(x,0.4)
     40%
 8474.8

> quantile(x,seq(0,1,0.1)) # todos os decis
   0%   10%   20%   30%   40%   50%   60%   70%   80%   90%  100%
 6892  7752  7947  8161  8475  8728  8982  9174  9547 10072 11317
```

1) Escreva suas próprias funções que calculem a variância (s^2) e o desvio-padrão (s) amostral.

$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \quad s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$

14.2 Distribuições de probabilidade

As distribuições estatísticas mais comuns são:

Distribuição	Nome no R	Parâmetros adicionais
Beta	`beta`	`shape1, shape2, ncp`
Binomial	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
Chi-quadrado	`chisq`	`df, ncp`
Exponencial	`exp`	`rate`

F	f	df1, df2, ncp
Gamma	gamma	shape, scale
Geométrica	geom	prob
Hipergeométrica	hyper	m, n, k
Lognormal	lnorm	meanlog, sdlog
Logistica	logis	location, scale
Normal	norm	mean, sd
Poisson	pois	lambda
T de Student	t	df, ncp
Uniforme	unif	min, max
Weibull	weibull	shape, scale

Para cada uma das distribuições, um prefixo deve ser adicionado:

Prefixo	Significado
d	Densidade de probabilidade $f(x)$
p	Função distribuição acumulada (CDF) $F(x)$
q	quartil
r	Retira uma amostra da distribuição

Os argumentos obrigatórios são:

função	Parâmetro
dxxx	x
pxxx	q
qxxx	p
rxxx	n

Para usar as funções, deve-se combinar uma das letras acima com a abreviatura do nome da distribuição. Por exemplo, para calcular probabilidades usa-se `pnorm()` para a distribuição Normal, `ppois()` para Poisson e assim por diante.

Exemplo:

```
> dnorm(2)
[1] 0.05399097
```

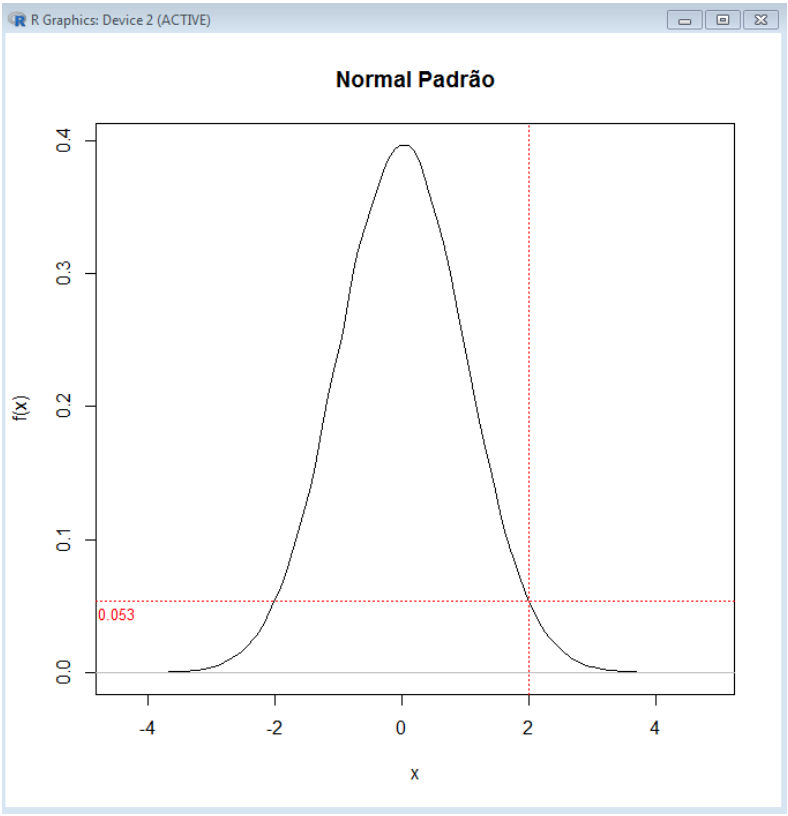
Este valor corresponde ao valor da densidade da distribuição Normal:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

com parâmetros $\mu = 0$ e $\sigma^2 = 1$ no ponto 2. Assim, o mesmo valor seria obtido substituindo x por 2 na expressão da Normal padrão:

```
> (1/sqrt(2*pi)) * exp((-1/2)*2^2)
[1] 0.05399097
```

Esse valor de `dnorm(2)` pode ser visualizado no gráfico a seguir.



O gráfico foi feito por meio dos comandos:

```
> y<-rnorm(100000)
> plot(density(y),xlab='x',ylab='f(x)',main='Normal Padrão')
> abline(h=dnorm(2),col='red',lty=3)
> abline(v=2,col='red',lty=3)
> text(x=-4.5,y=0.045,'0.053',col=2,cex=0.8)
```

A função `pnorm(2)` calcula a probabilidade $P(X \leq 2)$, ou seja, calcula o valor de $\int_{-\infty}^{2} f(X)dX$.

```
> pnorm(2)
[1] 0.9772499
```

A função `qnorm(0.92)` calcula o valor de a tal que $P(X \leq a) = 0{,}92$, ou seja, calcula o valor de a tal que $\int_{-\infty}^{a} f(X)dX = 0{,}92$.

```
> qnorm(0.92)
[1] 1.405072
```

A função `rnorm(5)` gera uma amostra de 5 elementos da normal padrão. Note que os valores que você irá obter executando este comando serão diferentes dos mostrados a seguir:

```
> rnorm(5)
[1]  0.8989409  1.0224997  1.4404664 -1.4831249  0.4698488
```

As funções anteriores possuem argumentos adicionais, para os quais valores padrão (*default*) foram assumidos, e que podem ser modificados.

```
> args(rnorm)
function (n, mean = 0, sd = 1)
```

As funções relacionadas à distribuição normal possuem os argumentos `mean` e `sd` para definir média e desvio padrão da distribuição, que podem ser modificados como no exemplo a seguir.

```
> rnorm(10,mean=-5,sd=2)
 [1] -4.538312 -6.484938 -5.149787 -4.092391 -3.553389 -5.324865 -3.587911
 [8] -4.708409 -1.940701 -3.875505
```

Obs.: para travar a aleatoriedade, pode-se fixar uma semente por meio da função `set.seed()`, que recebe um número inteiro como parâmetro:

```
> set.seed(6) # escolhi arbitrariamente uma semente de número 6
> rnorm(10,mean=-5,sd=2)
  [1] -4.460788 -6.259971 -3.262680 -1.545609 -4.951625 -4.263950 -7.618409
  [8] -3.522756 -4.910254 -7.096794
```

Observe que se usar a mesma semente (ou seja, 6), você obterá o mesmo conjunto de dados.

1) Gere 100 números aleatórios entre -15 e 50 e atribua a um vetor. Calcule os quartis desse vetor.

2) Seja X uma variável com distribuição N(100,100). Calcular as probabilidades:

- $P(X \leq 95)$
- $P(X > 95)$

14.3 Examinando a distribuição de um conjunto de dados

Seja um conjunto de dados univariados, pode-se examinar sua distribuição de diferentes formas. É possível analisar numericamente e graficamente os dados, assim como aplicar testes estatísticos para verificar alguma hipótese sobre o conjunto de dados.

Para exemplificar, será carregado um conjunto de dados chamado *faithful* que contém duas variáveis: *eruptions* e *waiting*, as quais representam, respectivamente, o tempo em minutos da duração da erupção do Gêiser Old Faithful nos Estados Unidos, e o tempo de espera em minutos entre uma erupção e outra.

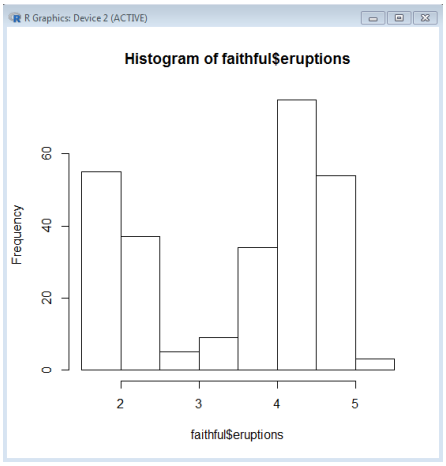
```
> data(faithful)
> ls()
[1] "faithful"
> str(faithful)
'data.frame':   272 obs. of  2 variables:
 $ eruptions: num  3.6 1.8 3.33 2.28 4.53 ...
 $ waiting  : num  79 54 74 62 85 55 88 85 51 85 ...
```

Numericamente:

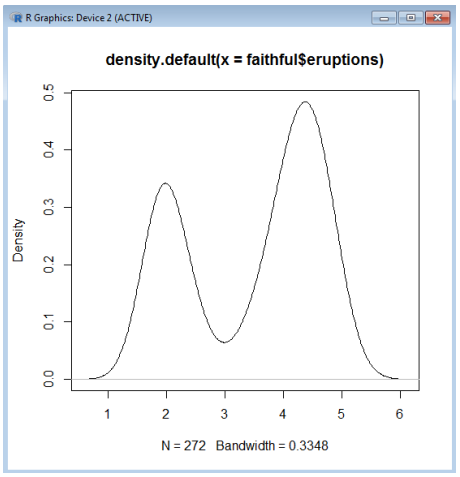
```
> summary(faithful)
   eruptions       waiting
 Min.   :1.60   Min.   :43.0
 1st Qu.:2.16   1st Qu.:58.0
 Median :4.00   Median :76.0
 Mean   :3.49   Mean   :70.9
 3rd Qu.:4.45   3rd Qu.:82.0
 Max.   :5.10   Max.   :96.0
```

Graficamente:

```
> hist(faithful$eruptions) # histograma das erupções
```

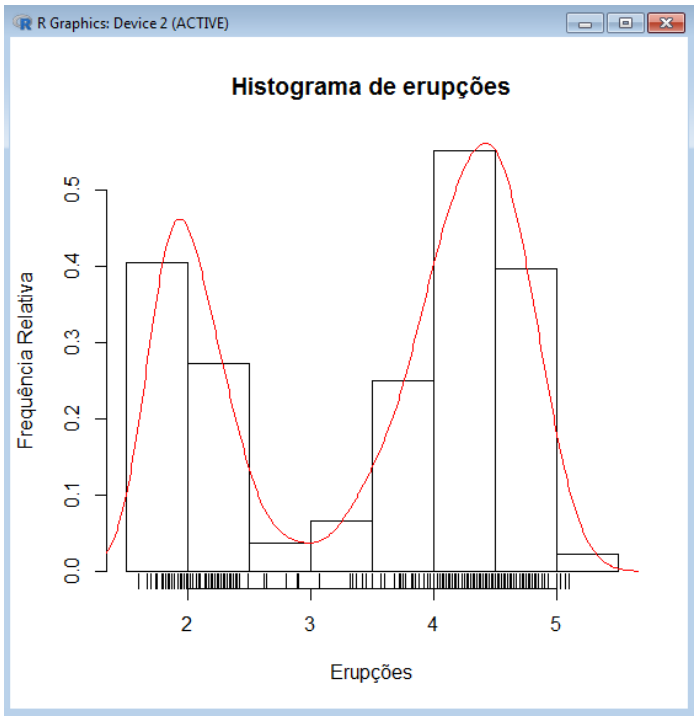



```
> plot(density(faithful$eruptions)) # função densidade de probabilidade
```



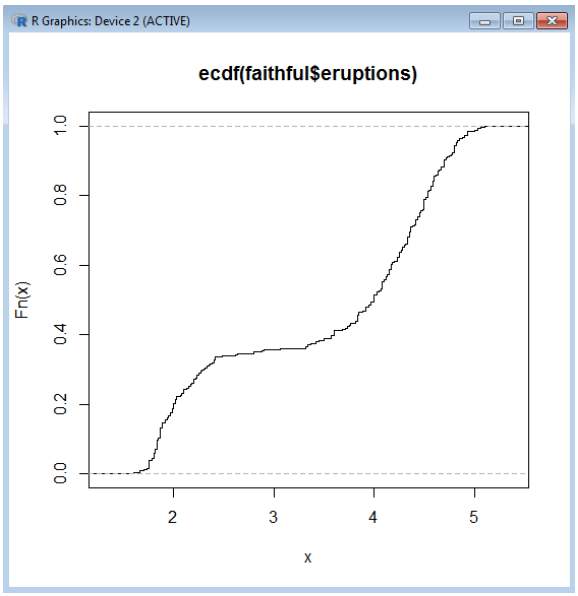
Pode-se plotar os dois gráficos juntos e melhorar a aparência dele.

```
> hist(faithful$eruptions, prob=TRUE,main='Histograma de
erupções',xlab='Erupções',ylab='Frequência Relativa')
> lines(density(faithful$eruptions, bw=0.2),col='red')
> rug(faithful$eruptions) # mostra os dados no eixo x
```



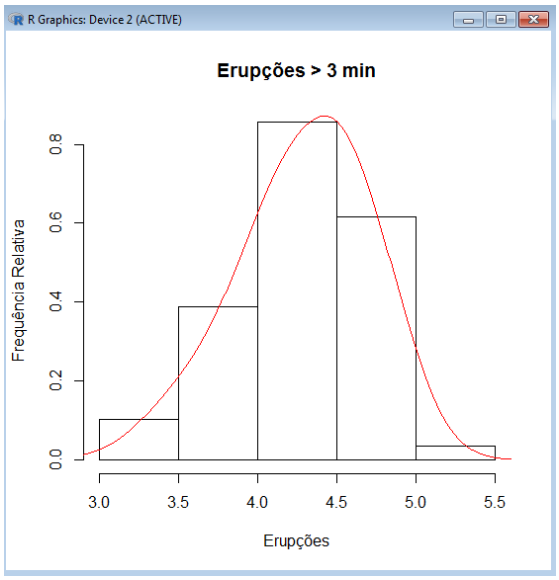
Também pode-se plotar a distribuição acumulada empírica, usando a função `ecdf()`.

```
> plot(ecdf(faithful$eruptions),do.points=FALSE,verticals=TRUE)
```

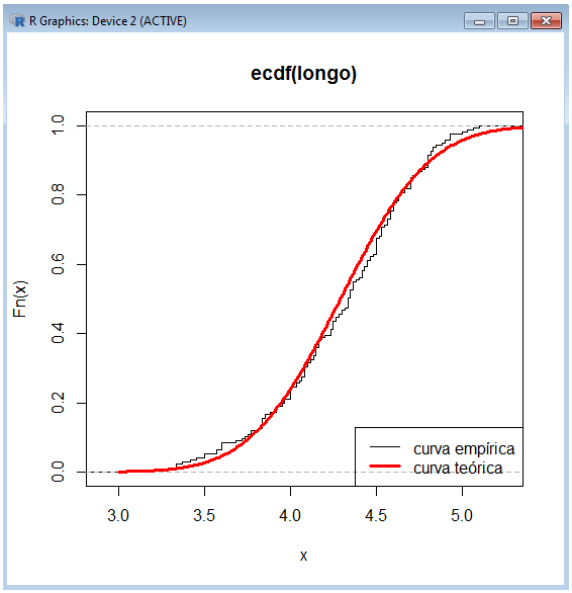


Como se pode observar, esta distribuição não se parece com a distribuição normal. Porém, o que dizer sobre as erupções que duram mais que 3 minutos? A seguir, serão feitas análises para verificar se esse conjunto de dados, que será chamado de *longo,* se ajusta a uma distribuição normal.

```
> longo<-faithful$eruptions[faithful$eruptions > 3]
> # ou similarmente longo<-faithful[faithful$eruptions>3,1]
> str(longo)
 num [1:175] 3.6 3.33 4.53 4.7 3.6 ...

> hist(longo, prob=TRUE,main='Erupções > 3 min', xlab='Erupções',ylab='Frequência
Relativa')
> lines(density(longo,bw=0.2),col='red')
```

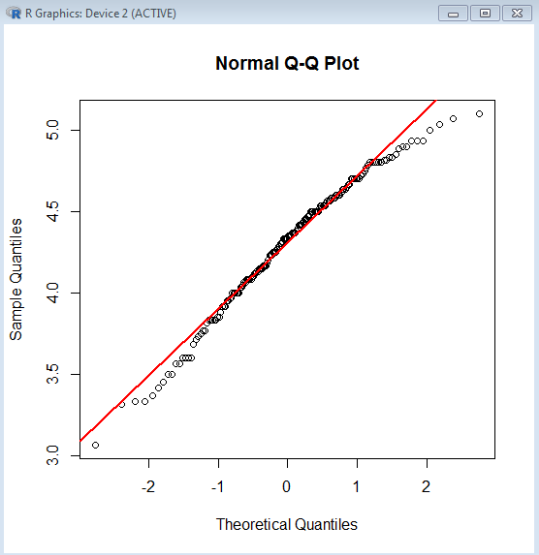



```
> plot(ecdf(longo), do.points=FALSE, verticals=TRUE)
> lines(x, pnorm(x, mean=mean(longo), sd=sd(longo)), col='red',lwd=3)
> x<-seq(3,5.5,by=0.1)
> legend('bottomright',c('curva empírica','curva teórica'),
col=c('black','red'),lwd=c(1,3))
```



Observa-se que a curva de densidade se ajusta ao histograma e a curva de probabilidade acumulada se ajusta à acumulada empírica. O gráfico quantil-quantil (ou gráfico Q-Q) é outra técnica para verificar a adequação de uma distribuição dos dados à uma distribuição de probabilidades.

```
> qqnorm(longo)
> qqline(longo,lwd=2, col='red')
```



Mais formalmente, o R fornece testes de aderência a distribuições de probabilidade:

- *Shapiro-Wilk*: teste usado para verificar apenas a normalidade de um conjunto de dados.

```
> shapiro.test(longo)

        Shapiro-Wilk normality test

data:  longo
W = 0.97934, p-value = 0.01052
```

- *Kolmogorov-Smirnov*: teste usado para verificar se um conjunto de dados adere a uma certa distribuição, necessitando ser informada a qual distribuição se quer testar.

```
> ks.test(longo,'pnorm',mean=mean(longo),sd=sd(longo))

        One-sample Kolmogorov-Smirnov test

data:  longo
D = 0.066133, p-value = 0.4284
alternative hypothesis: two-sided

Warning message:
In ks.test(longo, "pnorm", mean = mean(longo), sd = sd(longo)) :
  ties should not be present for the Kolmogorov-Smirnov test
```

Conclusões acerca da normalidade do conjunto de dados *longo*: o teste de *Shapiro-Wilk* retornou um p-valor de 0,01052 e tomando um nível de confiança de 95%, rejeita-se a hipótese de normalidade dos dados pois p-valor<0,05; entretanto o teste *Kolmogorov-Smirnov* retornou um p-valor de 0,4284, significando a aceitação da normalidade dos dados, pois p-valor>0,05. Portanto são necessários mais testes para verificar a normalidade dos dados.

Observações sobre o teste de *Kolmogorov-Smirnov*:
O teste de *Kolmogorov-Smirnov* não serve apenas para testar normalidade, mas sim aderência a qualquer distribuição. Apesar de na ajuda da função `ks.test()` não estar explícito, os parâmetros da distribuição a ser testada devem ser fornecidos:

```
…parameters of the distribution specified (as a character string) by y.
```

No caso da distribuição Normal, se os parâmetros média e desvio-padrão não são informados, é assumida a hipótese de normalidade padrão, ou seja, média 0 e desvio-padrão de 1.

Por exemplo, será gerada uma amostra Normal de 1000 valores com média 50 e desvio-padrão 5:

```
> x<-rnorm(1000,mean=50,sd=5)
> ks.test(x,'pnorm')

      One-sample Kolmogorov-Smirnov test

data:  x
D = 1, p-value < 2.2e-16
alternative hypothesis: two-sided
```

O que resultou em um p-valor errado, rejeitando a hipótese de normalidade.
Agora fornecendo os parâmetros da distribuição:

```
> ks.test(x,'pnorm',mean=mean(x),sd=sd(x))

      One-sample Kolmogorov-Smirnov test

data:  x
D = 0.019397, p-value = 0.8461
alternative hypothesis: two-sided
```

Resultou um p-valor grande, significando o aceite da normalidade dos dados.

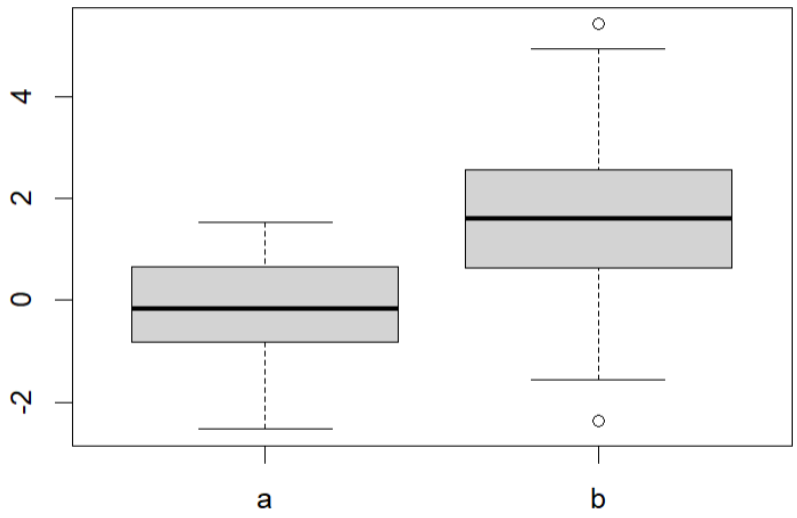
14.4 Comparando duas amostras

Até agora foi analisado um único conjunto de dados, porém é mais comum comparar aspectos de duas amostras. Considere os conjuntos a e b de 50 elementos cada, normalmente distribuídos:

```
> a<-rnorm(50,mean=0,sd=1)
> b<-rnorm(50,mean=1.5,sd=1.5)
```

Boxplot force uma comparação simples entre duas (ou mais) amostras.

```
> boxplot(a,b)
```



Este gráfico indica que o conjunto amostral a possui elementos menores e mais concentrados do que o conjunto b, o que é óbvio devido aos parâmetros usados na função `rnorm()` no momento da geração dos conjuntos.

O boxplot, também chamado de diagrama de caixas, representa a variação dos dados observados de uma variável numérica por meio dos quartis. As bordas das caixas representam o 1º quartil - Q1 (borda inferior) e o 3º quartil - Q3 (borda superior) e a reta dentro da caixa é a mediana. As retas (*whisker* ou fio de bigode) fora da caixa indicam a variabilidade fora dos quartis superior e inferior e são calculados da seguinte forma:

$$IQR = Q3 - Q1$$
$$upper\_whisker = min\,(max(x)\,, Q3 + 1.5 * IQR)$$
$$lower\_whisker = max\,(min(x)\,, Q1 - 1.5 * IQR)$$

No R, por exemplo, para o conjunto de dados b*:*

```
> IQR<-quantile(b,0.75)-quantile(b,0.25)
> upper_whisker<-min(max(b),quantile(b,0.75)+1.5*IQR)
> lower_whisker<-max(min(b),quantile(b,0.25)-1.5*IQR)
> upper_whisker
[1] 5.364307
> lower_whisker
[1] -2.143993
```

Os pontos acima ou abaixo dos bigodes são dados discrepantes (*outliers*).

Para testar se há ou não diferença nas médias dos conjuntos, pode-se usar a função `t.test()`.

1) Faça os seguintes gráficos:

- Da função densidade de uma variável com distribuição de Poisson com parâmetro $\lambda = 5$
- Da densidade de uma variável $X \sim N(90{,}100)$
- Sobreponha ao gráfico anterior a densidade de uma variável $Y \sim N(85{,}95)$

2) A distribuição da soma de duas variáveis aleatórias uniformes não é uniforme. Verifique isto gerando dois vetores x e y com distribuição uniforme $[0, 1]$ com 3000 valores cada e fazendo $z = x + y$. Obtenha o histograma para x, y e z. Descreva os comandos que utilizou.

14.5 Regressão linear

Regressão linear significa analisar uma variável de interesse (variável dependente ou variável resposta, aqui denotada por y) em função de outras variáveis que ajudarão a entendê-la (variáveis independentes, aqui denotadas por x).

Quando discutimos modelos, o termo "linear" não significa o ajuste de uma reta ou um plano. Ao invés disso, um modelo linear contém termos aditivos, cada qual contendo um único parâmetro multiplicativo. Assim, as equações

$$y = \beta_0 + \beta_1 x \qquad y = \beta_0 + \beta_1 x^2 \qquad y = \beta_0 + \beta_1 x_1 + \beta_2 \log(x_2)$$

são modelos lineares no parâmetro β; entretanto a equação $y = \beta_0 x^{\beta_1}$ não é um modelo linear.
Portanto, o que se está buscando são os coeficientes β.

Sintaxe (fórmula no R)	Modelo	Comentário
$Y \sim X$	$Y = \beta_0 + \beta_1 X$	Linha reta com intercepto implícito
$Y \sim -1 + X$	$Y = \beta_1 X$	Linha reta sem intercepto, isto é, um modelo forçado a passar pela origem
$Y \sim X + I(X\text{^}2)$	$Y = \beta_0 + \beta_1 X + \beta_2 X^2$	Modelo polinomial; note que a função identidade `I()` permite termos no modelo para incluir símbolos matemáticos
$Y \sim X1 + X2$	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$	Um modelo de primeira ordem sem termos de interação
$Y \sim X1 : X2$	$Y = \beta_0 + \beta_1 X_1 X_2$	Um modelo contendo somente a interação de primeira ordem entre X_1 e X_2

$Y \sim X1 * X2$	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$	Um modelo de primeira ordem completo com um termo; um código equivalente é $Y \sim X_1 + X_2 + X_1 : X_2$
$Y \sim (X1 + X2 + X3)\text{^}2$	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_1 X_2 + \beta_5 X_1 X_3 + \beta_6 X_2 X_3$	Um modelo incluindo todas as interações de primeira ordem até a *n*-ésima ordem, onde *n* é dado por $()\text{^}n$. Um código equivalente nesse caso é $Y \sim X_1 * X_2 * X_3 - X_1 : X_2 : X_3$

No R, o método de regressão linear é dado pela função `lm()`, que recebe como parâmetro obrigatório uma fórmula (dada pelas expressões da primeira coluna do quadro anterior).

O til (~) significa "em relação a" ou "modelado por". Por exemplo, o modelo $Y \sim X$ é interpretado como Y em relação a X ou Y modelado por X.

Para exemplificar, será carregado o conjunto de dados *mtcars*, que contém 11 atributos de 32 observações. São eles:

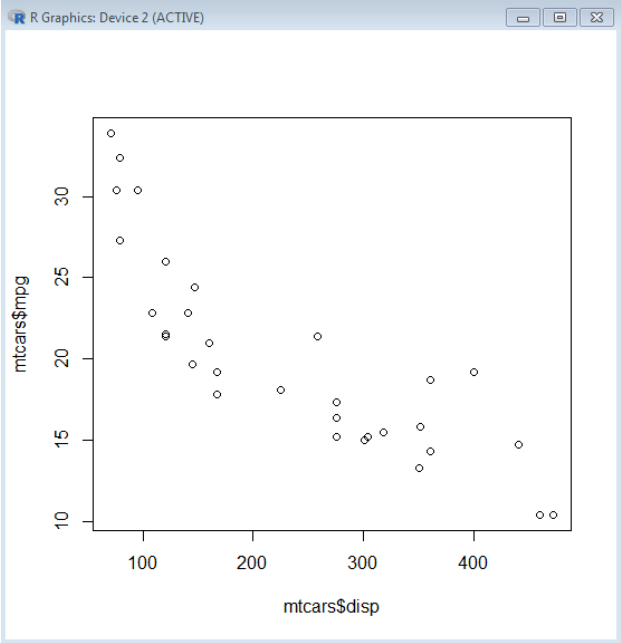
[, 1] mpg Miles/(US) gallon
[, 2] cyl Number of cylinders
[, 3] disp Displacement (cu.in.)
[, 4] hp Gross horsepower
[, 5] drat Rear axle ratio
[, 6] wt Weight (1000 lbs)
[, 7] qsec 1/4 mile time
[, 8] vs V/S
[, 9] am Transmission (0 = automatic, 1 = manual)
[,10] gear Number of forward gears
[,11] carb Number of carburetors

```
> data(mtcars)
> str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Os dados foram coletados da revista Motor Trend de 1974, e compreendem 11 aspectos de 32 automóveis (modelos 1973-74).

Analisando, por exemplo, a relação entre as variáveis `mpg` (milhas por galão) e `disp` (deslocamento).

```
> plot(mtcars$disp,mtcars$mpg) # gráfico de dispersão
```



Este mesmo gráfico pode ser obtido por meio de `plot(mtcars$mpg ~ mtcars$disp)`.

Nota-se uma relação de linearidade entre as variáveis, portanto será realizada uma regressão linear, ajustando uma reta da forma $Y = \beta_0 + \beta_1 X$.

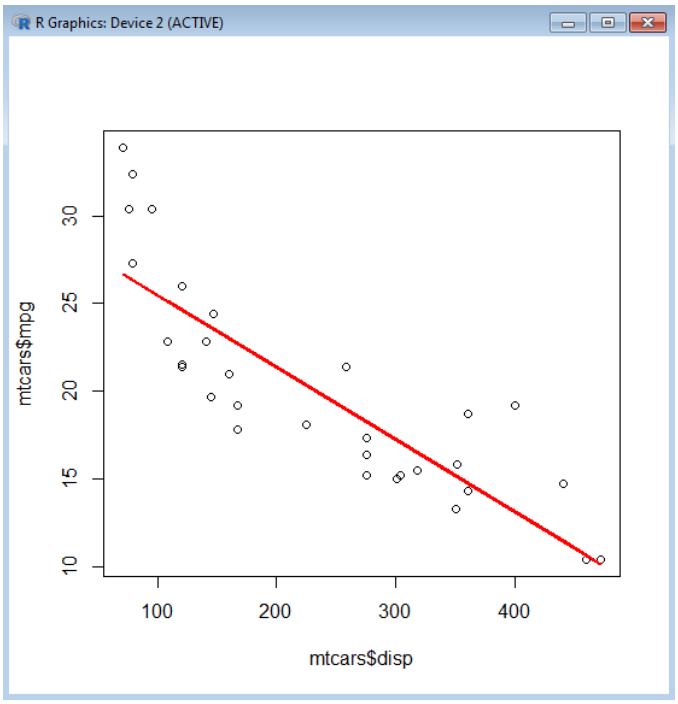
```
> modelo1<-lm(mtcars$mpg ~ mtcars$disp) # significado: mpg modelado por disp
> modelo1

Call:
lm(formula = mtcars$mpg ~ mtcars$disp)

Coefficients:
(Intercept)  mtcars$disp
   29.59985     -0.04122
```

Extraindo os coeficientes do modelo:

```
> coef(modelo1)
(Intercept) mtcars$disp
29.59985476 -0.04121512
```

Equação da reta ajustada:

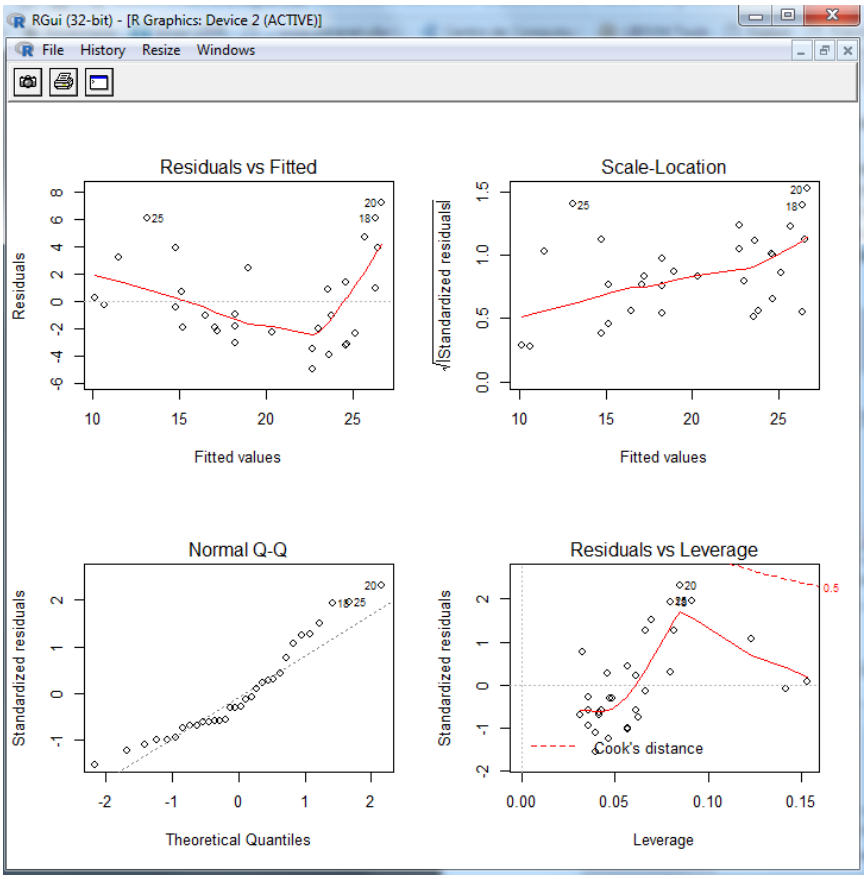
```
> ajuste_linear<-function(x, model) coef(model)[1]+coef(model)[2]*x
> lines(mtcars$disp,ajuste_linear(mtcars$disp,modelo1),col='red',lwd=2)
```



Esta mesma reta poderia ser plotada por meio do comando `abline(modelo1, col='red',lwd=2)`

Mais informações por meio gráfico:

```
> layout(matrix(1:4,2,2))
> plot(modelo1)
```



Resumo do modelo:

```
> summary(modelo1)

Call:
lm(formula = mtcars$mpg ~ mtcars$disp)

Residuals:
    Min      1Q  Median      3Q     Max
-4.8922 -2.2022 -0.9631  1.6272  7.2305

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 29.599855   1.229720  24.070  < 2e-16 ***
mtcars$disp -0.041215   0.004712  -8.747 9.38e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.251 on 30 degrees of freedom
Multiple R-squared:  0.7183,    Adjusted R-squared:  0.709
F-statistic: 76.51 on 1 and 30 DF,  p-value: 9.38e-10
```

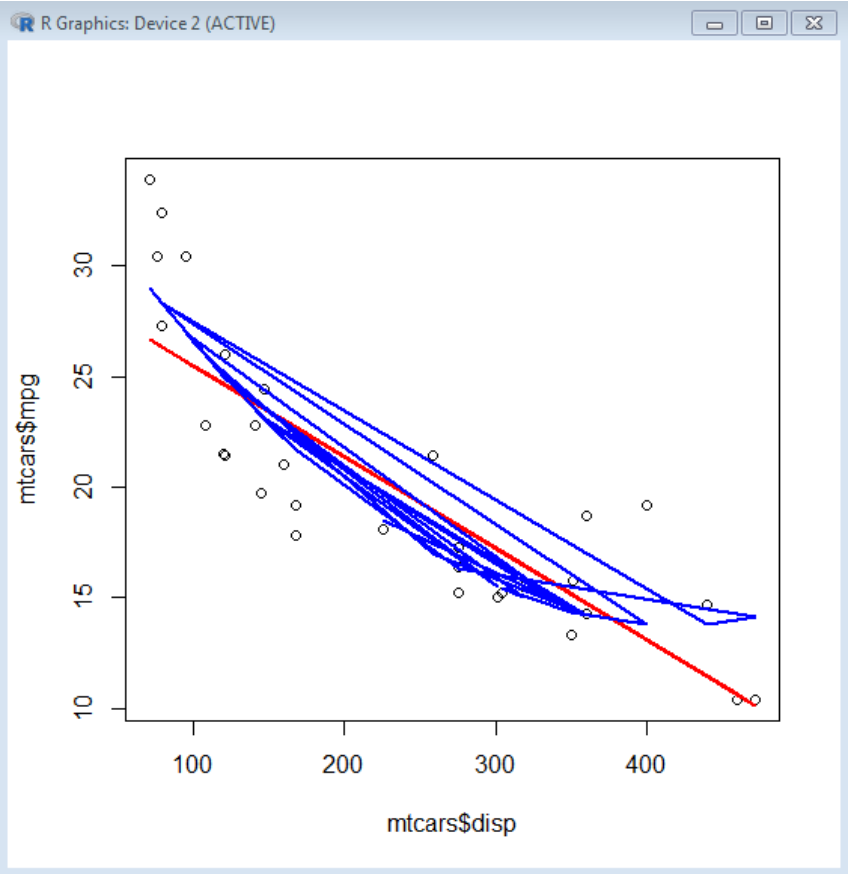
Uma maneira de avaliar se o ajuste foi satisfatório é verificar o indicador *R* quadrado ajustado, que quando maior melhor, e que para este exemplo deu 0,709.

Agora ajustando um polinômio de 2º grau:

```
> modelo2<-lm(mtcars$mpg~mtcars$disp+I(mtcars$disp^2))
```

Plotando no gráfico:

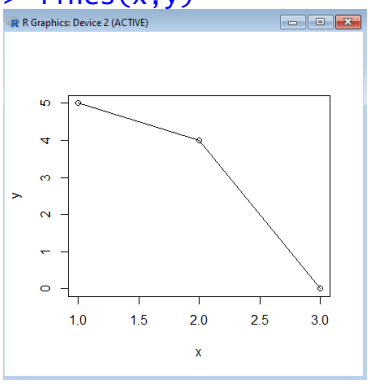
```
>ajuste_quad<-function(x,model) coef(model)[1] + coef(model)[2]*x + coef(model)[3]*x^2
>lines(mtcars$disp,ajuste_quad(mtcars$disp,modelo2),col='blue',lwd=2)
```



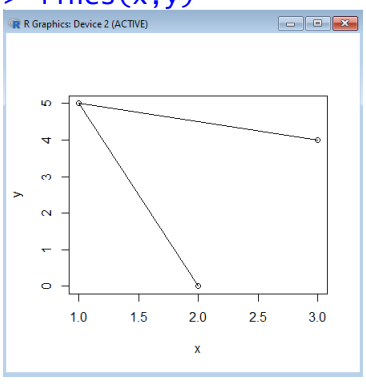
A plotagem não ficou como esperado, pois, `mtcars$disp` (que são os dados do eixo x) é um vetor não ordenado.

Um adendo sobre plotagem de linhas, veja o seguinte exemplo:

```
> x<-c(1,2,3)
> y<-c(5,4,0)
> plot(x,y)
> lines(x,y)
```



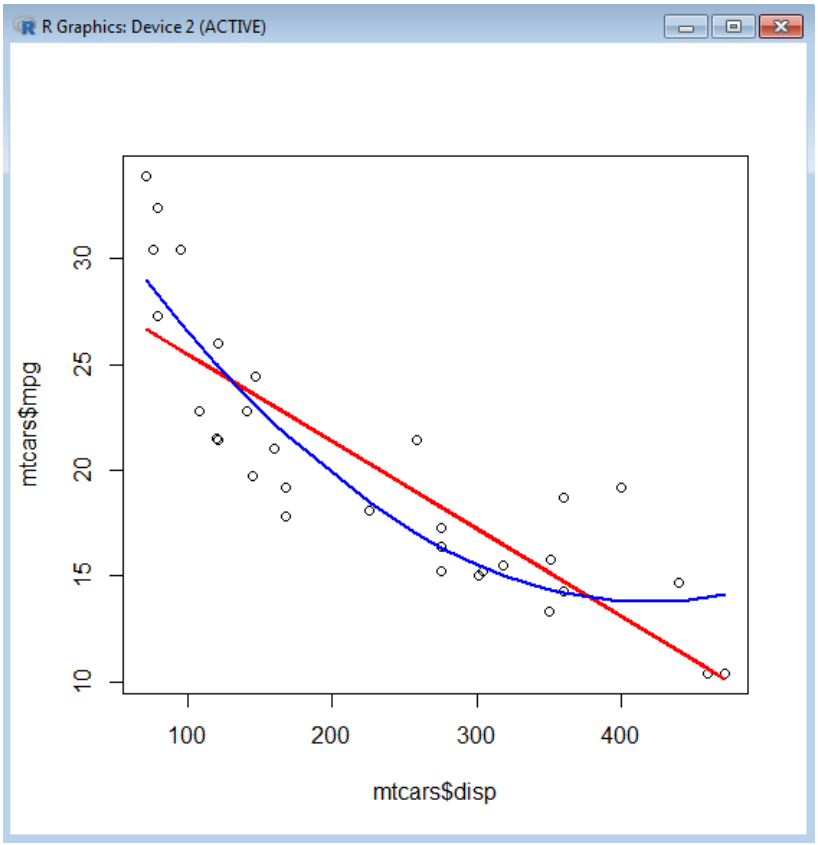

```
> x<-c(2,1,3)
> y<-c(4,5,0)
> plot(x,y)
> lines(x,y)
```



Os dados são exatamente os mesmos, mas a plotagem da linha ficou diferente pelo fato de que a plotagem inicia ligando o primeiro ponto com o segundo, o segundo com o terceiro, e assim por diante.

Voltando ao exemplo: para que se obtenha a curva esperada, ordena-se crescentemente o vetor `mtcars$disp`, e recomeça a plotagem (pois a antiga ficou comprometida):

```
> plot(mtcars$disp,mtcars$mpg)
> lines(mtcars$disp,ajuste_linear(mtcars$disp,modelo1),col='red',lwd=2)
> lines(sort(mtcars$disp),ajuste_quad(sort(mtcars$disp),modelo2),col='blue',lwd=2)
```



Analisando o resumo do modelo:

```
> summary(modelo2)

Call:
lm(formula = mtcars$mpg ~ mtcars$disp + I(mtcars$disp^2))

Residuals:
    Min      1Q  Median      3Q     Max
```

```
-3.9112 -1.5269 -0.3124  1.3489  5.3946

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)       3.583e+01  2.209e+00  16.221 4.39e-16 ***
mtcars$disp      -1.053e-01  2.028e-02  -5.192 1.49e-05 ***
I(mtcars$disp^2)  1.255e-04  3.891e-05   3.226   0.0031 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.837 on 29 degrees of freedom
Multiple R-squared:  0.7927,    Adjusted R-squared:  0.7784
F-statistic: 55.46 on 2 and 29 DF,  p-value: 1.229e-10
```

observa-se que o *R* quadrado ajustado deu 0,7784, maior que 0,709 que foi o *R* quadrado ajustado da reta, significando que a equação de 2º grau se ajustou melhor aos dados do que a reta.

14.6 Previsão usando resultados de regressão linear

Além de descrever relações, os modelos de regressão também podem ser usados para prever valores para novos dados. No R, a função que faz previsão é `predict()` que recebe como parâmetro obrigatório um modelo oriundo da função `lm()`.

Tomando ainda o exemplo anterior, um fabricante de automóveis tem três projetos para um novo carro e deseja saber qual é a quilometragem prevista com base no peso (1,7; 2,4 e 3,6) de cada novo *design*.
Primeiramente, aplicando a regressão linear para modelar a relação entre as variáveis peso (`mtcars$wt`) e milhas (`mtcars$mpg`).

```
> data(mtcars)
> peso<-mtcars$wt
> milhas<-mtcars$mpg
> mod<-lm(milhas ~ peso)
> novopeso<- data.frame(peso=c(1.7, 2.4, 3.6)) # aqui é importante manter o mesmo nome que foi utilizado no modelo
> predict(mod,newdata=novopeso)
 1        2        3
28.19952 24.45839 18.04503
```

Assim, o carro mais leve tem uma milhagem prevista de 28,2 milhas por galão e o carro mais pesado 18 milhas por galão, de acordo com este modelo.

O mesmo resultado pode ser obtido aplicando-se os novos dados ao modelo ajustado:

```
> ajuste_linear(novopeso,mod)
      peso
1 28.19952
2 24.45839
3 18.04503
```

Para ter uma ideia sobre a precisão das previsões, pode-se obter intervalos em torno de sua previsão. Para obter uma matriz com a previsão e um intervalo de confiança de 95% em torno da previsão média, definimos o argumento intervalo como confiança `confidence` (confiança).

```
> predict(mod,newdata=novopeso,interval='confidence')
      fit      lwr      upr
1 28.19952 26.14755 30.25150
2 24.45839 23.01617 25.90062
3 18.04503 16.86172 19.22834
```

Assim, de acordo com o modelo, um carro com um peso de 2,4 toneladas tem 95% de chance de fazer entre 23 e 25,9 milhas por galão. Da mesma forma, pode-se solicitar um intervalo de previsão de 95%, definindo o argumento de intervalo como previsão `prediction` (previsão).

```
> predict(mod,newdata=novopeso,interval='prediction')
       fit      lwr      upr
1 28.19952 21.64930 34.74975
2 24.45839 18.07287 30.84392
3 18.04503 11.71296 24.37710
```

Esta informação diz que 95% dos carros com um peso de 2,4 toneladas fazem uma milhagem entre 18,1 e 30,8 milhas por galão, com base no modelo.

1) Para uma amostra de oito operadores de máquina, foram coletados o número de horas de treinamento (x) e o tempo necessário para completar o trabalho (y). Os dados coletados encontram-se na tabela a seguir.

x	5,2	5,1	4,9	4,6	4,7	4,8	4,6	4,9
y	13	16	18	20	19	18	21	17

Pede-se:

- Faça o gráfico de dispersão para esses dados.
- Determine o modelo de regressão linear entre as variáveis x e y.
- Trace no gráfico, o modelo encontrado.

2) Descubra o que faz a função `glm()` para modelos lineares generalizados.

3) Estude as funções de previsão `predict.lm()` e `precitc.glm()`.

---

15 REFERÊNCIAS

R DEVELOPMENT CORE TEAM (2009). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, https://www.r-project.org/

ZEVIANI, W. *Blog Ridículas – Dicas curtas sobre o R*. ridiculas.wordpress.com

WICKHAM, H. *Advanced R*. 2ª ed, CRC Press, 2019. https://adv-r.hadley.nz/index.html

JUSTINIANO, P. *Site pessoal*. http://www.leg.ufpr.br/doku.php/sofware:material-r

FREIRE, S.M. *Introdução ao R*. 2021. https://www.lampada.uerj.br/arquivosdb/_book2/

OLIVEIRA, C.; ZEVIANI, W. *Guia Rápido do Usuário R*. http://www.leg.ufpr.br/~walmes/cursoR/guia_rapido_R.pdf

LANDEIRO, V., L. *Introdução ao uso do programa R.* https://cran.r-project.org/doc/contrib/Landeiro-Introducao.pdf

LINEAR – Laboratório de Análise de Dados. https://www.linearlab.com.br/manuseando-estrutura-de-dados-em-r

CURSO-R. Ciência de Dados em R. https://livro.curso-r.com/index.html