

PROGRAMAÇÃO EM R

EPRO7021

Programa de Pós Graduação em Engenharia de Produção - PPGE

Profa Dra Mariana Kleina

Sumário

1	INTRODUÇÃO.....	5
1.1	Site Oficial.....	5
1.2	Instalação.....	5
1.3	Manual.....	5
1.4	RStudio.....	5
1.5	Iniciando o R.....	5
1.6	Encerrando o R.....	5
1.7	Criando e Salvando uma Área de Trabalho (<i>Workspace</i>).....	6
1.8	Aproveitando Comandos já Digitados.....	6
2	R e OPERAÇÕES MATEMÁTICA.....	6
2.1	Operações Matemáticas Básicas.....	6
2.2	Atribuição de Variáveis.....	8
2.3	Impressão de Variáveis.....	8
2.4	Operadores de Comparação.....	8
2.5	Operadores Lógicos.....	9
2.6	Ajuda com Funções.....	9
3	VETORES.....	10
3.1	Operações com Vetores.....	10
3.2	Acessando Posições de um Vetor.....	11
3.3	Excluindo Elementos de um Vetor.....	11
3.4	Comandos Especiais para Vetores.....	11
	• <i>which()</i>	11
	• <i>unique()</i>	12
	• <i>table()</i>	12
	• <i>%in%</i>	12
3.5	Funções para Vetores.....	13
4	MATRIZES.....	14
4.1	Acessando Posições de uma Matriz.....	15
4.2	Excluindo Linhas e Colunas de uma Matriz.....	16
4.3	Os Comandos <i>rbind</i> e <i>cbind</i>	16
4.4	Operações com Matrizes.....	17
4.5	Funções para Matrizes.....	18
4.6	Transformando uma Matriz em um Vetor.....	18
4.7	Resolvendo Sistemas Lineares.....	18
4.8	Autovalores e Autovetores de uma Matriz.....	19

5	DATA.FRAME	20
6	LISTAS	22
7	MANIPULANDO STRINGS	24
7.1	O comando <i>paste()</i>	24
7.2	O comando <i>substr()</i>	25
7.3	O comando <i>nchar()</i>	25
7.4	Os comandos <i>toupper()</i> e <i>tolower()</i>	25
7.5	Os comandos <i>grep()</i> e <i>grepl()</i>	25
7.6	Os comandos <i>as.character()</i> e <i>is.character()</i>	26
8	COMANDOS MUITO USADOS	27
8.1	Informações sobre variáveis	27
8.2	Datas	28
8.3	NA e NULL	29
	• NA	29
	• NULL	29
	• NA vs. NULL	29
9	FUNÇÕES DE PACOTES NÃO-BÁSICOS	30
9.1	Observações Sobre Instalação de Pacotes	31
10	CONJUNTOS DE DADOS PRONTOS	31
11	LEITURA E GRAVAÇÃO DE DADOS	32
11.1	Leitura e Gravação de Arquivos de Texto	32
11.2	Leitura e Gravação de Planilhas do Excel	34
11.3	Leitura e Gravação de Arquivos Binários	36
12	EDITOR DE TEXTO	37
13	ESTRUTURAS CONDICIONAIS E LAÇOS	38
13.1	Estruturas Condicionais	38
13.2	Laços	40
13.3	Evitando Laços: a “família” * <i>apply</i>	43
14	ESCREVENDO SUAS PRÓPRIAS ROTINAS E FUNÇÕES	43
14.1	Escrevendo Funções	43
14.2	Escrevendo Rotinas	44
15	GRÁFICOS	46
15.1	Gráfico de Dispersão	46
15.2	Gráfico de Linha	48
15.3	Gráfico de Linha e Pontos	50
15.4	Gráfico de Barra	52

15.5	Gráfico de Pizza	53
15.6	Inserir Texto em Gráficos	54
15.7	Legenda em Gráficos	54
15.8	Salvando Gráficos	55
15.9	Animação	55
16	ESTATÍSTICA	56
16.1	Estatística Descritiva	56
16.2	Distribuições de Probabilidade	58
16.3	Examinando a Distribuição de um Conjunto de Dados.....	60
16.4	Comparando Duas Amostras	65
16.5	Regressão Linear.....	66
16.6	Previsão Usando Resultados de Regressão Linear	72
17	REFERÊNCIAS	73

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)

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 oficial www.rstudio.com 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 é só clicar nos ícones  ou  que foram criados no momento da instalação do R ou RStudio, respectivamente.

No Linux basta digitar R na linha de comando no terminal.

O 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 um parênteses, chaves ou colchetes. Para voltar ao ambiente de digitação (>), basta apertar a tecla 'Esc'.

1.6 Encerrando o R

Para fechar o R, basta digitar

> q()

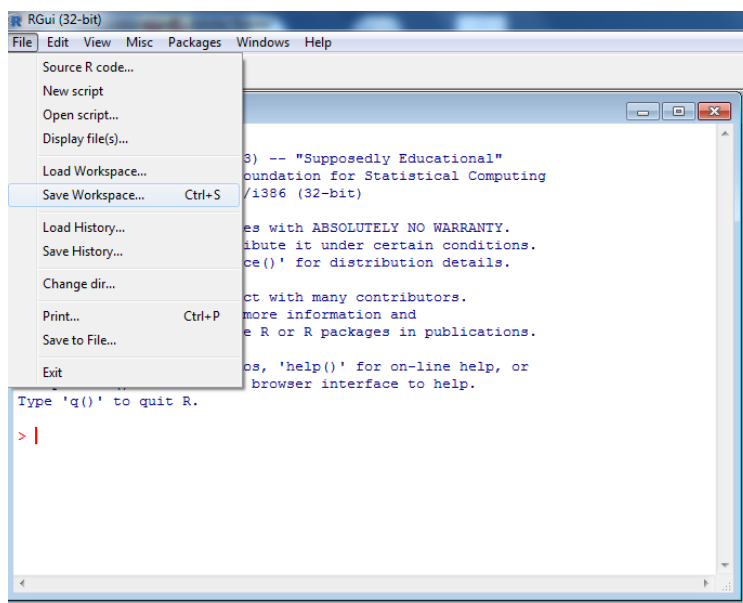
então será perguntado se deseja salvar a área de trabalho.

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

No Windows, 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 / no lugar de \).

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, vá em File >> Save Workspace e salve com a extensão .RData em uma pasta destinada somente para trabalhar com o R.



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.

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).

1.8 Aproveitando Comandos já Digitados

Você pode recuperar comandos já digitados no R por meio das teclas \uparrow e \downarrow .

2 R e OPERAÇÕES MATEMÁTICA

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
> 4-2
> 2*2
> 5*(1+6)
> 7/3 # isto é um comentário
> 6^2 # exponenciação, que também pode ser feita da forma 6**2
> 5%/%2 # divisão inteira
```

```
> 5%%2 # resto da divisão
```

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(0) # arco tangente
[1] 0
> atan(0)/pi*180
[1] 0
```

Funções para arredondamento:

```
> ceiling(4.3478) # teto
[1] 5
> floor(4.3478) # chão
[1] 4
> round(4.3478) # arredonda com 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
> 2*NA # NA significa Not Available
[1] NA
```

2.2 Atribuição de Variáveis

Pode-se atribuir valores à variáveis com o operador `<-`.

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

Observe que não precisa definir a variável antes, nem informar o seu tipo.

O símbolo `=` também pode ser usado para atribuição, porém `<-` é mais utilizado.

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.

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(b)
20.8>
> cat(b,'\n') # '\n' pula de linha
20.8
```

Com o comando `cat()` também é possível mesclar texto e variáveis.

```
> cat('O conteúdo de b é',b,'\n')
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 booleano (*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
```

2.5 Operadores Lógicos

Os operadores lógicos são:

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

O resultado é um valor booleano.

Obs: && avalia a 1ª condição e somente se ela é verdadeira então avaliará a segunda condição. Por outro lado, & avalia ambas 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
```

2.6 Ajuda com Funções

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 argumentos de uma função.

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



- 1) Tem-se duas variáveis a e b. Troque o conteúdo das variáveis, isto é, a receba o conteúdo de b e vice-versa.
 - 2) O que acontece com o valor de pi (3.141593) se é feito `pi<-15` ?
 - 3) No R, calcule $\frac{\sqrt[5]{(\ln(4)+\pi)}}{e^{2\sin(2,56)}}$ e atribua o resultado a uma variável.
 - 4) Veja qual é o resultado das operações `TRUE + TRUE`, `FALSE + TRUE` e `FALSE + FALSE`. O que se pode concluir a respeito do valor de `TRUE` e `FALSE`?
-

3 VETORES

Um vetor pode ser criado de diferentes formas. A mais comum é concatenando números um do lado do outro, por meio da função `c()`, sempre separando os números 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) # pode-se escrever somente as iniciais do parâmetro se nenhum outro
parâmetro da função tiver as mesmas iniciais
> 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
```

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,] -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.

3.2 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
```

3.3 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 -9.0 1.1
> x<-x[-3] # excluindo o terceiro elemento de x
> x
[1] 5 -9
```

3.4 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, seja o mesmo vetor `y` anterior de 14 posições 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:

```
> (1:14)[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 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 booleano sempre do tamanho do primeiro vetor.

3.5 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
```



Lição de casa !

1) Crie os seguintes vetores:

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

2) Se você tem dois vetores $v = [v_1, v_2, \dots, v_{1000}]$ e $w = [w_1, w_2, \dots, w_{500000}]$, como você faria para concatená-los, isto é, $z = [v_1, v_2, \dots, v_{1000}, w_1, w_2, \dots, w_{500000}]$?

- 3) 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 do comando `matrix()`, que recebe como argumentos os dados e o número de linhas ou o número de colunas (ou ambos).

```
> 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 `ncol()` e `nrow()`, respectivamente:

```
> ncol(A)
```

```
[1] 4
```

```
> nrow(A)
```

```
[1] 2
```

```
> 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 nem o número de linhas, nem o número de colunas são informados, 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 $\text{ncol} \times \text{nrow}$, 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 $\text{ncol} \times \text{nrow}$ 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 acrescentar o comando `byrow=TRUE` no momento da definição da 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
```

Uma matriz deve conter variáveis do mesmo tipo, isto é, todos os exemplos apresentados anteriormente são matrizes numéricas. Mas matrizes de strings, 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   4   7
[2,] -99   5   8
[3,]  3   6   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 Os Comandos rbind e cbind

Existem dois comandos que podem ser bastante úteis quando se trabalha com matrizes. Eles 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
```

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
```

O comando `rbind()` também pode servir 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
```

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
```

```

      [,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

> 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

```

4.6 Transformando uma Matriz em um Vetor

É possível extrair todos os elementos de uma matriz na forma de um vetor com o comando `as.vector()`:

```

> A
      [,1] [,2]
[1,]  -6   1
[2,]   2   4
> v<-as.vector(A)
> v
[1] -6  2  1  4

```

4.7 Resolvendo Sistemas Lineares

Já foi visto que a função `solve()` inverte uma matriz. Porém esta mesma função também resolve um sistema de equações lineares, em que são fornecidos a matriz A e o vetor b do sistema $Ax=b$. O resultado é o vetor solução x .

```

> A
      [,1] [,2]

```

```
[1,] -6 1
[2,] 2 4
> b<-c(4,-1)
> solve(A,b)
[1] -0.65384615 0.07692308
```

4.8 Autovalores e Autovetores de uma Matriz

Os autovalores e autovetores de uma matriz podem ser obtidos por meio da função *eigen()*.

```
> A<- matrix(c(4,2,5,1),ncol=2)
> eigen(A)
eigen() decomposition
$values
[1] 6 -1

$vectors
      [,1] [,2]
[1,] 0.9284767 -0.7071068
[2,] 0.3713907 0.7071068
```



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

2) 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.

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

4) 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 arbitrário de dimensões. Digite no R o comando `?array`, rode e inspecione os exemplos contidos na documentação. Veja o exemplo a seguir:

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

      [,1] [,2] [,3] [,4]
[1,] 1 4 7 10
```

```

[2,] 2 5 8 11
[3,] 3 6 9 12

, , 2

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

```

5 DATA.FRAME

A estrutura data.frame é como a de uma matriz, porém permite mesclar variáveis de tipos diferentes. Geralmente se dá um nome para cada coluna da data.frame e associa-se um vetor de dados a essa coluna.

```

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

```

Esta estrutura é útil quando se quer trabalhar com informações de clientes, de produtos,...

```

> empresa<-data.frame(nome=c('João','José','Cláudia','Ivo','Márcia','Fernanda'),
nascimento=c(as.POSIXlt('1990-05-12'),as.POSIXlt('1985-12-02'),
as.POSIXlt('1994-07-26'),as.POSIXlt('1988-03-15'),
as.POSIXlt('1974-08-21'),as.POSIXlt('1991-01-09')),cargo=c('assistente
comercial','gerente','estagiário','técnico de produto','auxiliar
administrativo','estagiário'),salário=c(4000,6500,1000,3200,3400,1000),
horas_trabalhadas=c(8,8,6,8,8,6))

> empresa
  nome nascimento      cargo salário 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 `ncol()` e `nrow()` também podem ser usadas para data.frame:

```

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

```

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

```

> empresa[3,1]
[1] Cláudia
Levels: Cláudia Fernanda Ivo João José Márcia

```

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

```

> empresa$nome[3]
[1] Cláudia
Levels: Cláudia Fernanda Ivo João José Márcia

```

Uma coluna inteira pode ser obtida:

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

Assim como uma linha inteira pode ser obtida:

```
> empresa[4,]
  nome nascimento      cargo salário 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 das pessoas que ganham salário maior que 3300:

```
> empresa[empresa$salário>3300,]
  nome nascimento      cargo salário 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\$salário>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, salário>3300)
  nome nascimento      cargo salário 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

> subset(empresa, salário>3300 & salário<5000)
  nome nascimento      cargo salário 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
Levels: assistente comercial auxiliar administrativo estagiário gerente técnico de produto
```

Ou usando a função *subset()*:

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

É possível excluir uma coluna inteira de uma *data.frame*:

```
> empresa<-empresa[,-2] # ou empresa$nascimento<-NULL
> empresa
  nome      cargo salário horas_trabalhadas
1  João  assistente comercial      4000              8
2  José      gerente      6500              8
```

3	Cláudia	estagiário	1000	6
4	Ivo	técnico de produto	3200	8
5	Márcia	auxiliar administrativo	3400	8
6	Fernanda	estagiário	1000	6

ou excluir uma linha inteira de uma data.frame:

```
> empresa<-empresa[-5,]
> empresa
```

	nome	cargo	salário	horas_trabalhadas
1	João	assistente comercial	4000	8
2	José	gerente	6500	8
3	Cláudia	estagiário	1000	6
4	Ivo	técnico de produto	3200	8
6	Fernanda	estagiário	1000	6

ou excluir uma linha e uma coluna ao mesmo tempo:

```
> empresa<-empresa[-2,-4]
> empresa
```

	nome	cargo	salário
1	João	assistente comercial	4000
3	Cláudia	estagiário	1000
4	Ivo	técnico de produto	3200
6	Fernanda	estagiário	1000

ou excluir várias linhas e colunas ao mesmo tempo:

```
> empresa<-empresa[c(-1,-2),c(-2,-1)]
> empresa
[1] 3200 1000
```

Lição de casa !

- 1) Crie uma data.frame 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 estas mesmas informações de dois colegas seus. Então, acrescente uma nova informação a essa data.frame: o email.

6 LISTAS

Uma lista tem a mesma finalidade de uma data.frame, porém aceita vetores de tamanhos diferentes.

```
> 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
```

Na verdade, é possível colocar qualquer estrutura dentro de uma lista e não necessariamente apenas vetores. 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
```

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

```
> l[[2]]
[1] "aa" "bb" "cc" "dd"
```

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

```
> l[[1]][2]
[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
```

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 data.frame?
-

7 MANIPULANDO STRINGS

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

7.1 O comando *paste()*

O comando *paste()* literalmente “cola” objetos, transformando-os em strings. 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"
```

7.2 O comando *substr()*

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

```
> substr("Programação", 3, 7)
[1] "ogram"
```

Neste exemplo, foi extraída uma nova string a partir da posição 3 até a posição 7 da palavra “Programação”. O resultado é sempre uma string.

7.3 O comando *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
```

7.4 Os comandos *toupper()* e *tolower()*

Estas funções mudam as letras de uma string para caixa alta e baixa, respectivamente.

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

7.5 Os comandos *grep()* e *grepl()*

Funções usadas para verificar se strings estão contidas em outras strings.

```
> grep("oi","jkoikj")
[1] 1
> grep("oi","jkkkj")
integer(0)
```

Para vetores de strings.

```
> string1<-"at"
> vetor_string<-c("ass","ear","eye","heat")
> grepl(string1,vetor_string)
[1] FALSE FALSE FALSE TRUE

> string1<-"at|as" # operador lógico (ou) usado
> grepl(string1,vetor_string)
```

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

7.6 Os comandos *as.character()* e *is.character()*

A função *as.character()* transforma um valor numérico em string.

```
> x<-87
> y<-as.character(x)
> y
[1] "87"
```

Já a função *as.character()* retorna um valor booleano se o parâmetro da função é uma string ou não.

```
> is.character(x)
[1] FALSE
> is.character(y)
[1] TRUE
```

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



Lição de casa !

- 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”.
- 2) Deseja-se saber se há linhas repetidas no seguinte registro (em forma de *data.frame*) e retirar as repetições:

Nome	idade	casado
Mario	50	1
Joana	47	0
Mario	50	0
Sara	18	1
Sofia	45	0
Mario	50	0

```
> dados<-data.frame(nome=c("Mario","Joana","Mario","Sara","Sofia","Mario"),idade
=c(50,47,50,18,45,50),casado=c(1,0,0,1,0,0))
```

Claro que este é somente um exemplo pequeno, imagine um grande conjunto de dados. Como você faria para tirar registros repetidos?

Por exemplo, o resultado final deveria ser:

Nome	idade	casado
Mario	50	1
Joana	47	0
Mario	50	0
Sara	18	1
Sofia	45	0

8 COMANDOS MUITO USADOS

8.1 Informações sobre variáveis

Algumas informações importantes sobre variáveis podem ser obtidas por meio de funções prontas. São elas:

Mostrar todas as variáveis criadas:

```
> ls()
[1] "a" "A" "b" "c" "M1" "M2" "p" "S" "v" "w" "x" "y" "z"
```

A estrutura de uma variável:

```
> str(b)
num [1:2] 4 -1
> str(M2)
num [1:2, 1:2] 2 1 -6 3
```

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

```
> ls.str()
a : num 5
A : num [1:2, 1:2] -6 2 1 4
b : num [1:2] 4 -1
c : num 5
M1 : num [1:2, 1:2] -5 0 2 4
M2 : num [1:2, 1:2] 2 1 -6 3
p : int [1:15] -1 0 1 2 3 -1 0 1 2 3 ...
S : chr [1:2, 1:3] "a" "d" "b" "e" "c" "f"
v : num [1:8] 2 6 8 3 2 5 8 0
w : num [1:8] 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
x : num [1:2] 5 -9
y : num [1:14] 4 6 8 10 12 14 16 18 20 22 ...
z : num [1:3] 0 4.5 9
```

A classe de uma variável:

```
> class(v)
[1] "numeric"
> class(M1)
[1] "matrix"
```

Um resumo sobre uma variável:

```
> summary(y) # mínimo, 1º quartil, mediana, média, 3º quartil e máximo, respec.
  Min. 1st Qu. Median Mean 3rd Qu. Max.
  4.0   10.5   17.0   17.0   23.5   30.0
```

```
> summary(A) # v1 é a 1ª coluna e v2 é a 2ª coluna
      v1      v2
Min.   :-6   Min.   :1.00
1st Qu.: -4   1st Qu.:1.75
Median : -2   Median :2.50
Mean    :-2   Mean    :2.50
3rd Qu.:  0   3rd Qu.:3.25
Max.    :  2   Max.    :4.00
```

Remoção de uma ou mais variáveis:

```
> rm(a)
> rm(M1,M2,y,p,c)
> ls()
[1] "A" "b" "S" "v" "w" "x" "z"
```

Remoção de todas as variáveis:

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

8.2 Datas

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

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

Para digitar uma data específica 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 tem-se horário de verão).

Existe também a função *as.POSIXct()*.

- "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 (como "hora" e "mon", por exemplo).

Veja um exemplo para entender a diferença:

```
> date1 <- as.POSIXct(1268736919, origin="1970-01-01", tz="GMT")
> date2 <- as.POSIXlt(1268736919, origin="1970-01-01", tz="GMT")
> date1
[1] "2010-03-16 10:55:19 GMT"
> date2
[1] "2010-03-16 10:55:19 GMT"
> unclass(date1)
[1] 1268736919
attr(,"tzone")
[1] "GMT"
> unclass(date2)
$sec
[1] 19

$min
[1] 55

$hour
[1] 10

$mday
[1] 16

$mon
[1] 2

$year
[1] 110

$yday
[1] 2

$yday
[1] 74

$isdst
[1] 0

attr(,"tzone")
[1] "GMT"
```

Para saber a diferença entre tempos, a função *difftime()* pode ser usada. Por exemplo, tem-se duas variáveis *t1* e *t2*:

```
> 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
```

8.3 NA e NULL

- **NA**

NA, como já mencionado, significa “Not Available”, e em um conjunto de dados é certamente um problema na coleta desse dado.

É possível verificar se um conjunto de dados tem problemas 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
```

- **NULL**

O comando *NULL* tem muitas utilidades. Uma delas, que já foi vista, é a exclusão de uma coluna de uma data.frame (para linha não funciona, nem para matrizes) e exclusão de uma posição de uma lista. Entretanto, outra utilidade muito usada é a inicialização de variáveis. Por exemplo:

```
> d<-data.frame(nome=NULL,idade=NULL)
> d
  nome idade
1 Maria   45
> d<-rbind(d,data.frame(nome="Maria",idade=45))
...
> d<-rbind(d,data.frame(nome=c("Paulo","Renata"),idade=c(25,18)))
> d
  nome idade
1 Maria   45
2 Paulo   25
3 Renata   18
```

- **NA vs. NULL**

Uma boa maneira de entender a diferença entre *NA* e *NULL* é por meio de exemplos.

NA	NULL
<pre>> NA [1] NA > class(NA) [1] "logical" > NA>1 [1] NA</pre>	<pre>> NULL NULL > class(NULL) [1] "NULL" > NULL>1 logical(0)</pre>
<pre>> v<- c(1, NA, NULL) > v [1] 1 NA</pre>	<pre>> list(1, NA, NULL) [[1]] [1] 1</pre>

	<pre>[[2]] [1] NA [[3]] NULL</pre>
<pre>> v[1]<-NA > v [1] NA NA</pre>	<pre>> v[1] <- NULL Error in v[1] <- NULL : re placement has length zero</pre>
<pre>> li <- list(1, 2, 3) > li[[1]]<-NA > li [[1]] [1] NA [[2]] [1] 2 [[3]] [1] 3</pre>	<pre>> li <- list(1, 2, 3) > li[[1]] <- NULL > li [[1]] [1] 2 [[2]] [1] 3</pre>

9 FUNÇÕES DE PACOTES NÃO-BÁSICOS

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:

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

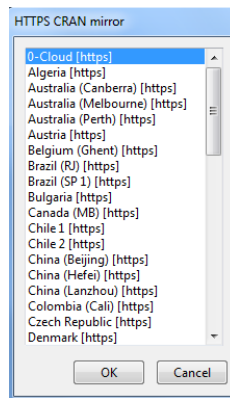
Porém, nem sempre um pacote está instalado no R (aos pacotes não instalados dá-se o nome de **pacotes contribuídos**). Assim, antes de ativá-lo, é preciso instalá-lo por meio do comando *install.packages()*. Por exemplo, o pacote *CircStats* não vem instalado no R:

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

Para instalá-lo:

```
> install.packages('CircStats')
Installing package into 'C:/Users/Pessoa1/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 Brasil (PR), Brasil (SP 1) ou 0-Cloud caso não apareça Brasil:



```
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')
```

Para mostrar todos os pacotes do R instalados no seu computador, basta 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')
```

9.1 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

10 CONJUNTOS DE DADOS PRONTOS

No R, existem muitos dados que estão disponíveis para serem utilizados para testes, análises, enfim, para qualquer finalidade. Estes dados são acessíveis para que não se precise ficar “inventando” ou 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))
```

E para usar qualquer conjunto de dados, primeiro deve-se carregar o seu respectivo pacote e depois carregá-lo por meio do comando `data(nome_conjunto_dados)`.



- 1) Descubra o que fazem as funções `attach()` e `detach()`.
 - 2) Carregue o conjunto de dados prontos `longley`. Transforme essa `data.frame` em uma lista.
 - 3) Carregue o conjunto de dados prontos `longley`. Transforme essa `data.frame` em uma matriz.
-

11 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.

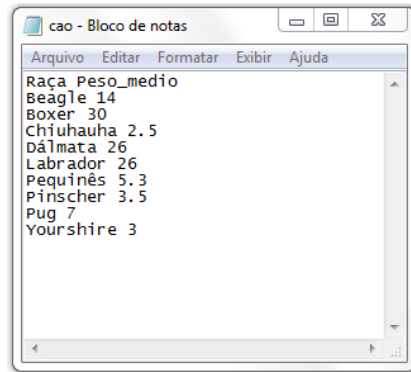
O R consegue fazer a leitura de dados de três extensões muito utilizadas: texto (.txt), planilha Excel (.xlsx) e binário (.bin).

11.1 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 possui 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:



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

```
> r
```

	Raça	Peso_medio
1	Beagle	14.0
2	Boxer	30.0
3	Chihuahua	2.5
4	Dálmata	26.0
5	Labrador	26.0
6	Pequês	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 para verdadeiro este parâmetro, pois neste exemplo havia um cabeçalho.

Se é esquecido de colocar `header=FALSE`, veja o que acontece:

```
> r<-read.table('cao.txt')
```

```
> r
```

	V1	V2
1	Raça	Peso_medio
2	Beagle	14
3	Boxer	30
4	Chihuahua	2.5
5	Dálmata	26
6	Labrador	26
7	Pequês	5.3
8	Pinscher	3.5
9	Pug	7
10	Yorkshire	3

Observe que a segunda coluna mistura dados numéricos com strings. Isso faz com que toda a segunda coluna se torne um fator.

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

```
> r<-read.table('cao.txt',skip=3) # pula o cabeçalho + 2 linhas de dados
> r<-read.table('cao.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 "."), entre outros. Veja todos os parâmetros por meio do `?read.table`

- **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('cao.txt',header=TRUE)
```

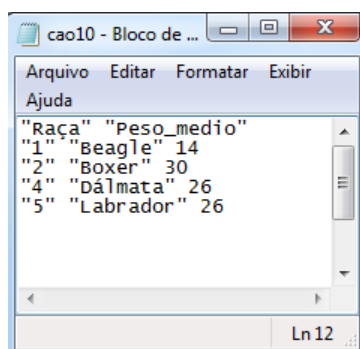
Suponha que se necessita filtrar os dados para raças de cães com peso médio superior a 10 kg:

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

	Raça	Peso_medio
1	Beagle	14
2	Boxer	30
4	Dálmata	26
5	Labrador	26

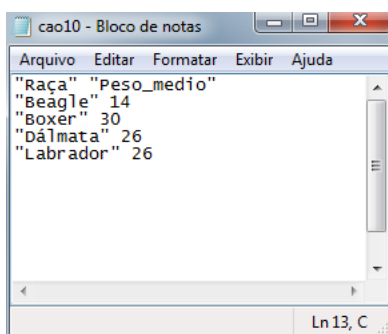
E agora se deseja salvar estes dados filtrados. Para isso usa-se:

```
> write.table(r10,'cao10.txt')
```



Os índices das linhas foram gravados também. O ideal é que o arquivo “cao10.txt” fosse similar ao arquivo “cão.txt”.

```
> write.table(r10,'cao10.txt',col.names=FALSE,row.names=FALSE)
```



Apesar do cabeçalho e das raças terem sido gravados com aspas, isso não é um problema, pois o R lê normalmente:

```
> testeaspas<-read.table('cao10.txt',header=TRUE)
> testeaspas
```

	Raça	Peso_medio
1	Beagle	14
2	Boxer	30
3	Dálmata	26
4	Labrador	26

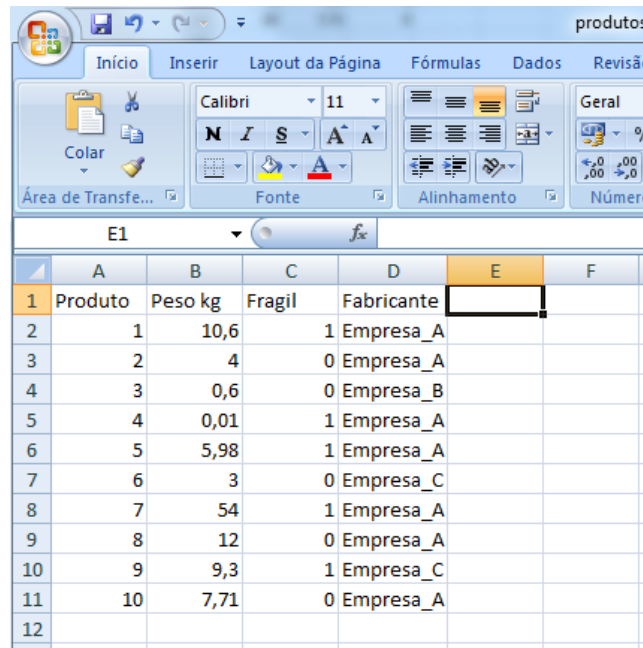
11.2 Leitura e Gravação de Planilhas do Excel

• 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 salva no diretório de trabalho:



	A	B	C	D	E	F
1	Produto	Peso kg	Fragil	Fabricante		
2	1	10,6	1	Empresa_A		
3	2	4	0	Empresa_A		
4	3	0,6	0	Empresa_B		
5	4	0,01	1	Empresa_A		
6	5	5,98	1	Empresa_A		
7	6	3	0	Empresa_C		
8	7	54	1	Empresa_A		
9	8	12	0	Empresa_A		
10	9	9,3	1	Empresa_C		
11	10	7,71	0	Empresa_A		
12						

Para lê-la no R:

```
> prods<-read_excel("produtos.xlsx")
> prods
  Produto Peso.kg Fragil Fabricante
1      1    10.60     1  Empresa_A
2      2     4.00     0  Empresa_A
3      3     0.60     0  Empresa_B
4      4     0.01     1  Empresa_A
5      5     5.98     1  Empresa_A
6      6     3.00     0  Empresa_C
7      7    54.00     1  Empresa_A
8      8    12.00     0  Empresa_A
9      9     9.30     1  Empresa_C
10     10     7.71     0  Empresa_A
> str(prods)
'data.frame':   10 obs. of  4 variables:
 $ Produto    : num  1 2 3 4 5 6 7 8 9 10
 $ Peso.kg    : num  10.6 4 0.6 0.01 5.98 3 54 12 9.3 7.71
 $ Fragil     : num  1 0 0 1 1 0 1 0 1 0
 $ Fabricante : Factor w/ 3 levels "Empresa_A","Empresa_B",...: 1 1 2 1 1 3 1 1 3 1
```

• 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 1 ...
> write_xlsx(iris,'Iris.xlsx')

```

A planilha gravada será da forma:

	A	B	C	D	E	F
1		Sepal.Len	Sepal.Wid	Petal.Len	Petal.Wid	Species
2	1	5,1	3,5	1,4	0,2	setosa
3	2	4,9	3	1,4	0,2	setosa
4	3	4,7	3,2	1,3	0,2	setosa
5	4	4,6	3,1	1,5	0,2	setosa
6	5	5	3,6	1,4	0,2	setosa
7	6	5,4	3,9	1,7	0,4	setosa
8	7	4,6	3,4	1,4	0,3	setosa
9	8	5	3,4	1,5	0,2	setosa
10	9	4,4	2,9	1,4	0,2	setosa
11	10	4,9	3,1	1,5	0,1	setosa
12	11	5,4	3,7	1,5	0,2	setosa
13	12	4,8	3,4	1,6	0,2	setosa
14	13	4,8	3	1,4	0,1	setosa
15	14	4,3	3	1,1	0,1	setosa
16	15	5,8	4	1,2	0,2	setosa
17	16	5,7	4,4	1,5	0,4	setosa
18	17	5,4	3,9	1,3	0,4	setosa
19	18	5,1	3,5	1,4	0,3	setosa
20	19	5,7	3,8	1,7	0,3	setosa
21	20	5,1	3,8	1,5	0,3	setosa
22	21	5,4	3,4	1,7	0,2	setosa
23	22	5,1	3,7	1,5	0,4	setosa
24	23	4,6	3,6	1	0,2	setosa
25	24	5,1	3,3	1,7	0,5	setosa
26	25	4,8	3,4	1,9	0,2	setosa
27	26	5	3	1,6	0,2	setosa
28	27	5	3,4	1,6	0,4	setosa
29	28	5,2	3,5	1,5	0,2	setosa
30	29	5,2	3,4	1,4	0,2	setosa
31	30	4,7	3,2	1,6	0,2	setosa
32	31	4,8	3,1	1,6	0,2	setosa
33	32	5,4	3,4	1,5	0,4	setosa
34	33	5,2	4,1	1,5	0,1	setosa
35	34	5,5	4,2	1,4	0,2	setosa
36	35	4,9	3,1	1,5	0,2	setosa
37	36	5	3,2	1,2	0,2	setosa
38	37	5,5	3,5	1,3	0,2	setosa
39	38	4,9	3,6	1,4	0,1	setosa

11.3 Leitura e Gravação de Arquivos Binários

Arquivos binários são muito utilizados na prática, pois ocupam menos 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('xlsx')
> prods<-read.xlsx("produtos.xlsx", 1)
> i<-read.xlsx("Iris.xlsx", 1)

```

```
> ls()
[1] "a"      "i"      "nomes"  "prods"
> save(nomes,prods,a,file='variaveis.bin') #não quero salvar i, por exemplo
```

- **Leitura**

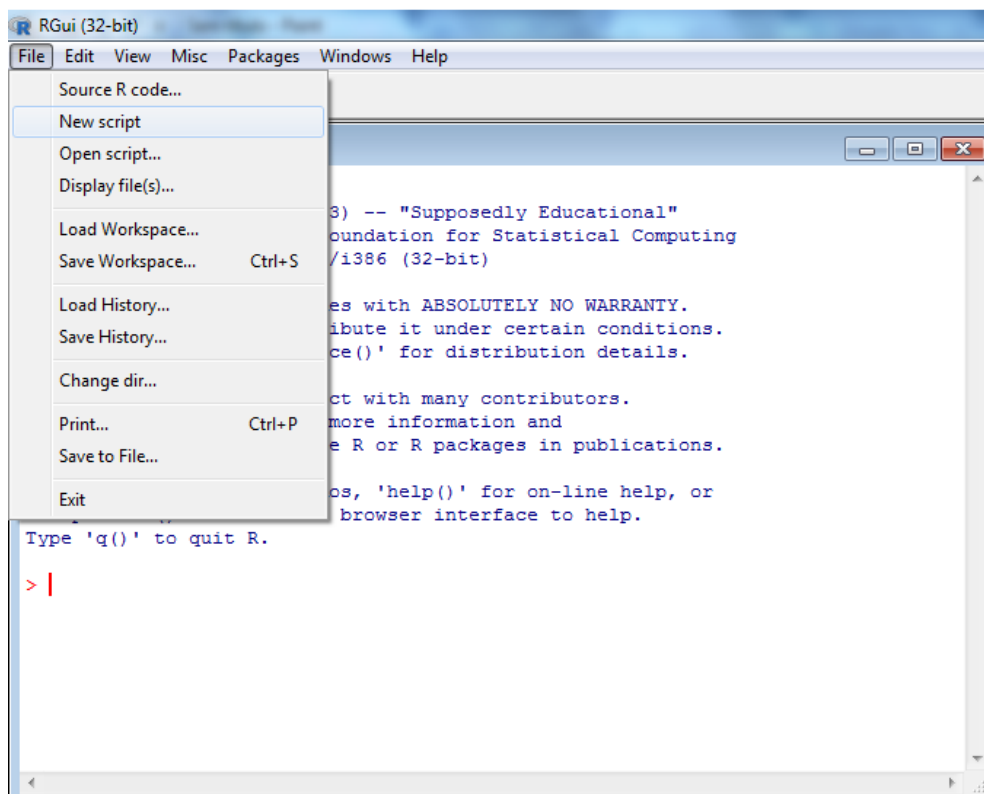
Para fazer a leitura de um arquivo binário, usa-se a função *load()*, que tem como parâmetro obrigatório o nome do arquivo binário a ser lido. Essa função fará a leitura de um arquivo salvo com a função *save()*.

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

12 EDITOR DE TEXTO

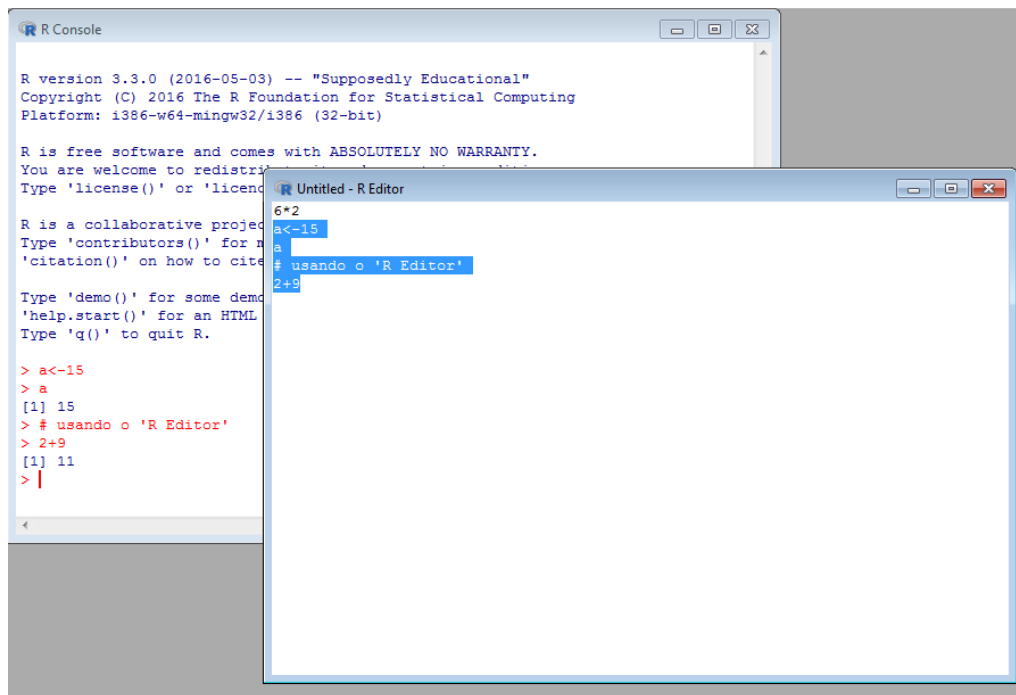
Foi visto que comandos podem ser digitados diretamente na janela do R, porém é aconselhável que se utilize um editor de texto para escrever scripts e programas. Editores de texto facilitam a visualização e permitem a indentação do código, permitem uma flexibilidade para percorrer entre as linhas do código, facilitam a correção de eventuais erros, podem ser salvos e recuperados posteriormente, entre outras vantagens.

O R do Windows tem um editor de texto próprio, que pode ser acessado em File >> New script.



O script pode ser escrito normalmente neste editor e para executá-lo basta copiá-lo e colá-lo no 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.



Um script automaticamente 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 o script anterior 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, porém observe a criação da variável `a`).

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

13 ESTRUTURAS CONDICIONAIS E LAÇOS

13.1 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:
[1] -45

Observação: se dentro do `{...}` só existir apenas uma linha comando, então as chaves são opcionais.

Exemplo: Verificar se um número informado 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:
[1] "é par"

ou

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

Resultado:
[1] "é par"

- `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 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:

```
[1] "é positivo"
```

ou

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

Resultado:

```
[1] "é zero"
```

- **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
```

13.2 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+1
  print(i)
}
```

Resultado:

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



```
[1] 7
[1] 8
```

Não necessariamente o contador deve variar em uma sequência regular de passo 1. Veja o exemplo a seguir:

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

Resultado:

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

Para forçar a saída de um laço, usa-se o comando *break*.

```
soma<-0
for(i in 1:8){
  soma<-soma+1
  if (soma>5) break
  print(i)
}
```

Resultado:

```
[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 a *condição* para falsa em algum momento, pois caso contrário, entrará em 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
}
print(soma)
```

Resultado:

```
[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
[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
[1] NA NA NA NA NA NA NA NA NA NA 10 NA NA NA NA 25 NA
[30] NA NA NA 45 NA NA NA NA 70 NA NA NA 175 NA NA NA 220
[59] NA NA NA NA 270 NA NA NA 325 NA NA NA NA NA NA NA
[88] NA NA NA NA NA 675 NA NA NA NA 760 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
[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 $\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \dots$

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
```

```
> summary(v1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.5000  1.0000  1.0000  0.9999  1.0000  1.0000
> summary(v2)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.5000	1.0000	1.0000	0.9999	1.0000	1.0000

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).



- 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: $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$
- 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.

13.3 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.

14 ESCRREVENDO SUAS PRÓPRIAS ROTINAS E FUNÇÕES

O R é uma ferramenta poderosíssima 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 scripts e funções. Tudo o que será visto adiante, será escrito em um editor de texto, pois é incomum se escrever tais scripts e funções diretamente no R.

14.1 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)*. Para retornar mais do que uma variável, usa-se a estrutura de lista dentro do *return*.

Exemplos:

- 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')
```

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"
```

- Receber um número e dobrar seu valor.

```
dobra<-function(x) x<-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
```

- 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
```

14.2 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

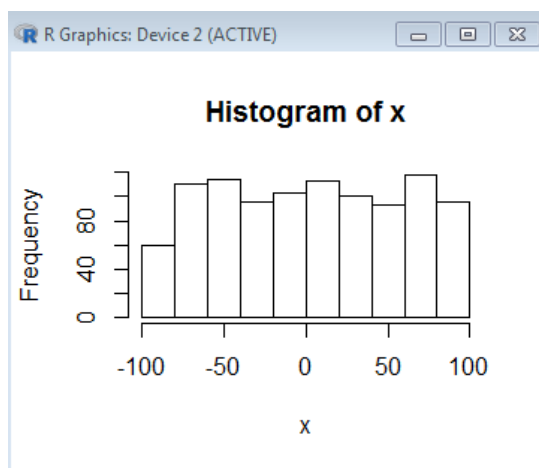
Exemplo:

- 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:

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



Lição de casa !

- 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.
-

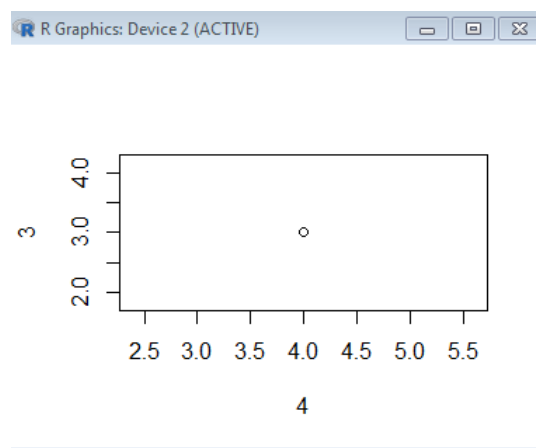
15 GRÁFICOS

Existem muitos tipos de gráficos no R, porém os mais comuns são 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/>

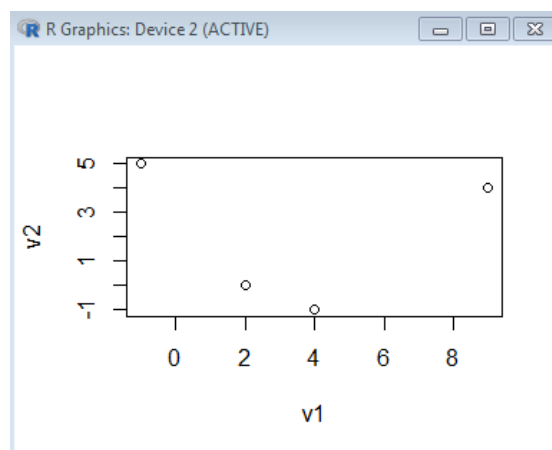
15.1 Gráfico de Dispersão

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.

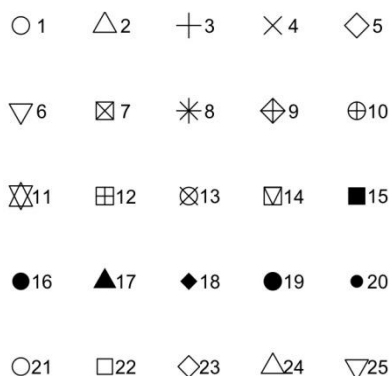
```
> plot(4,3)
```



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

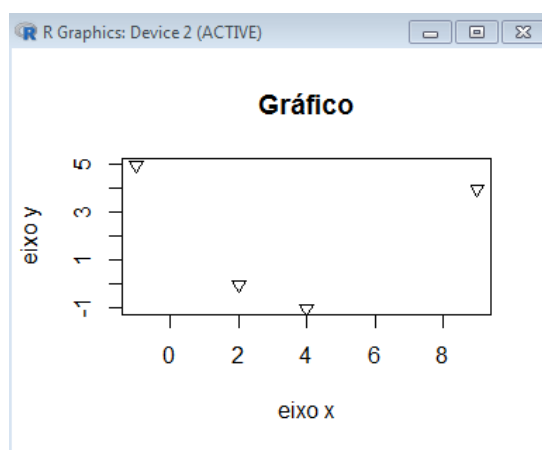


Observe que por padrão os pontos são plotados com o símbolo 'o' e os eixos recebem os nomes das variáveis plotadas. O símbolo pode ser alterado com o comando *pch=n*, onde n é um número inteiro ou um símbolo, conforme figura:



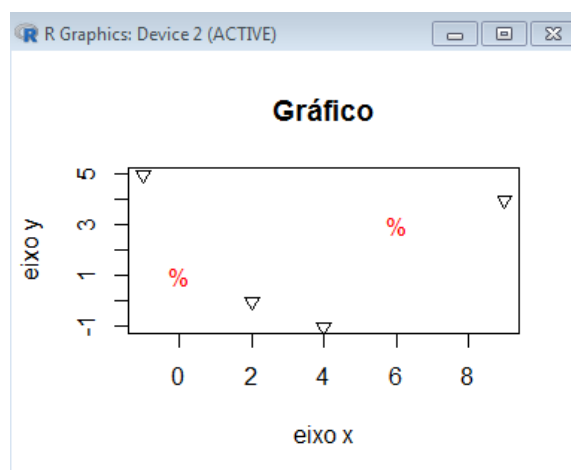
Os nomes dos eixos podem ser alterados com os comandos *xlab* e *ylab*. Um título pode ser acrescentado com o parâmetro *main*. Todos estes parâmetros são alterados dentro da função *plot()*.

```
> 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:

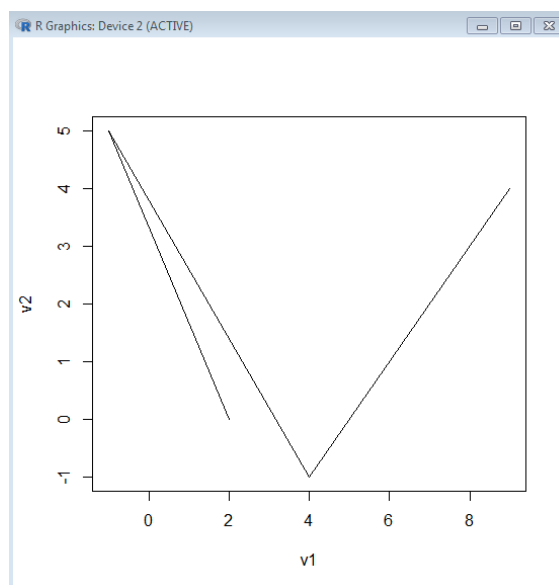


Portanto, para pintar algo de vermelho, por exemplo, pode-se fazer `col='red'` ou `col=2`. Existem muito mais cores do que as apresentadas na figura anterior.

15.2 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 o comando `plot()`, porém se acrescenta como parâmetro para a função `plot` o comando `type='l'`, que significa que o tipo a ser plotado é uma linha (l = linha). Por padrão do `plot`, `type='p'` (p=points).

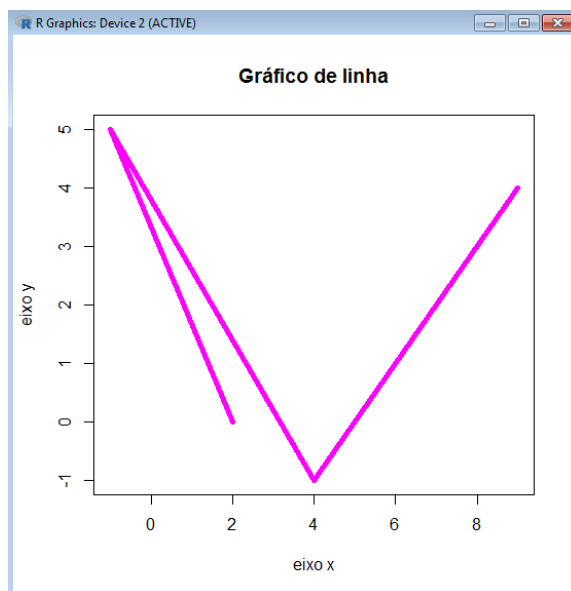
```
> v1<-c(2,-1,4,9)
> v2<-c(0,5,-1,4)
> plot(v1,v2,type='l')
```



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 `n`, mais “grossa” fica a linha.

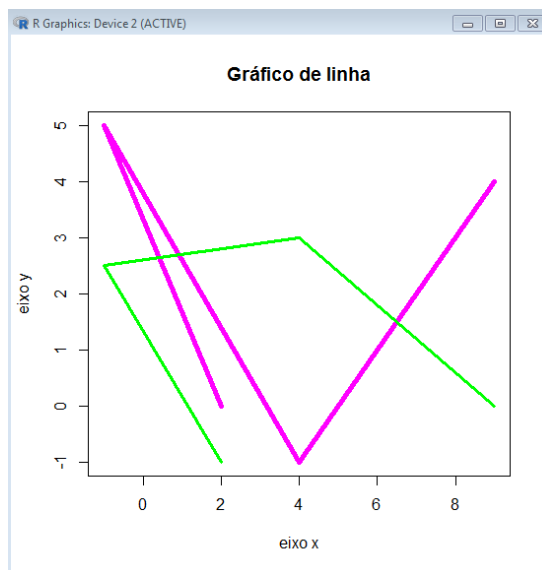
Alterando a aparência do gráfico anterior:

```
> 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'`.

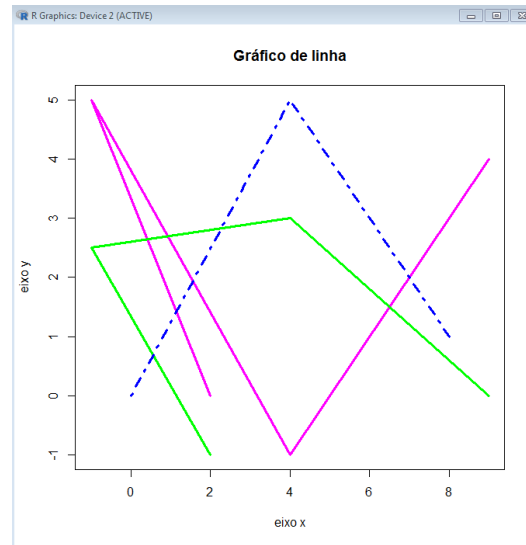
```
> v3<-c(-1,2.5,3,0)
> lines(v1,v3, col='green', lwd=3)
```



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:

- | | |
|---------------|---|
| 0. 'blank' | |
| 1. 'solid' | <hr/> |
| 2. 'dashed' | <hr style="border-top: 1px dashed black;"/> |
| 3. 'dotted' | <hr style="border-top: 1px dotted black;"/> |
| 4. 'dotdash' | <hr style="border-top: 1px dotdash black;"/> |
| 5. 'longdash' | <hr style="border-top: 1px longdash black;"/> |
| 6. 'twodash' | <hr style="border-top: 1px twodash black;"/> |

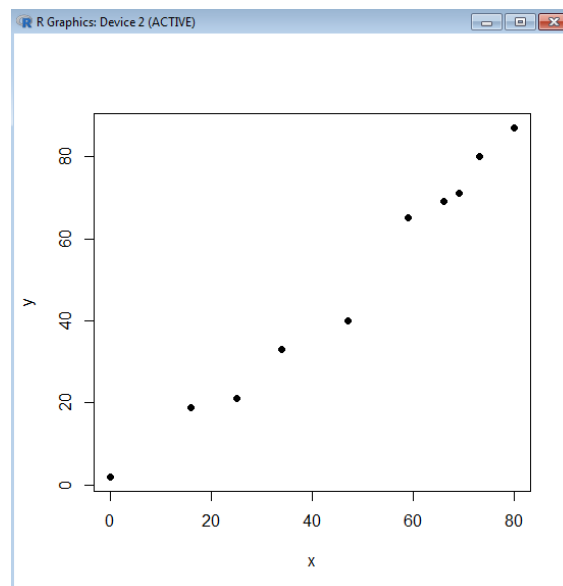
```
> lines(c(0,4,8),c(0,5,1), col='blue',lwd=3,lty=4)
```



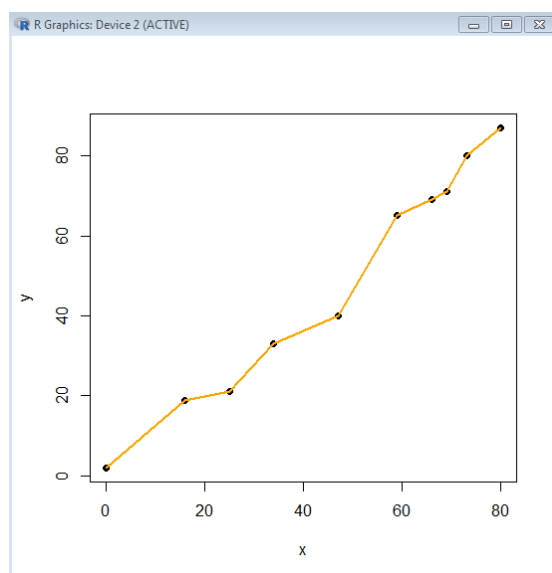
15.3 Gráfico de Linha e Pontos

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()* para linhas e *points()* para pontos. Segue um outro exemplo.

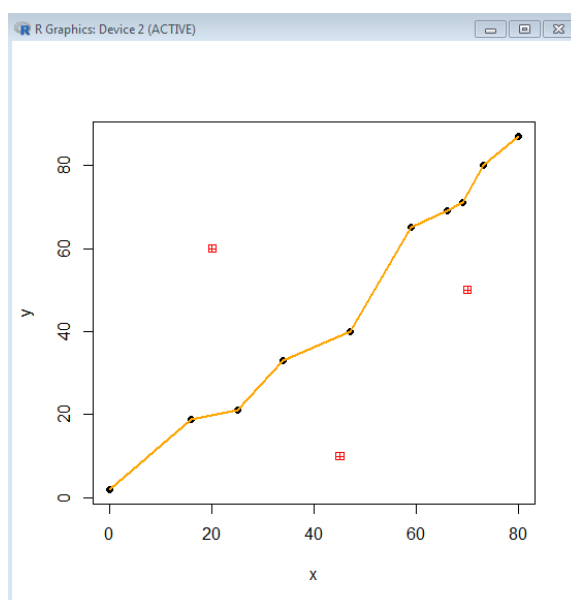
```
> x<-c(0,16,25,34,47,59,66,69,73,80)
> y<-c(2,19,21,33,40,65,69,71,80,87)
> plot(x,y,pch=16)
```



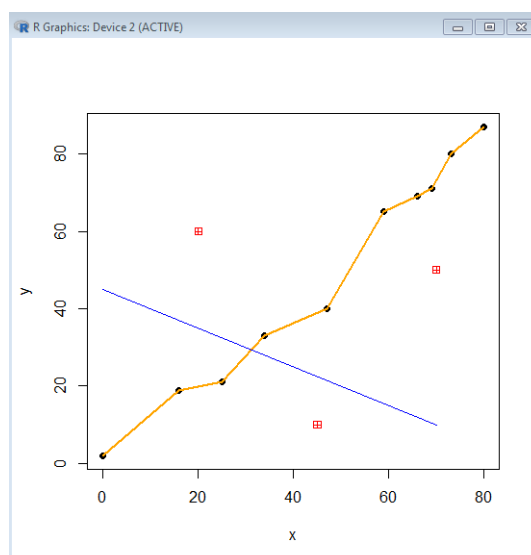
```
> lines(x,y,col='orange',lwd=2)
```



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

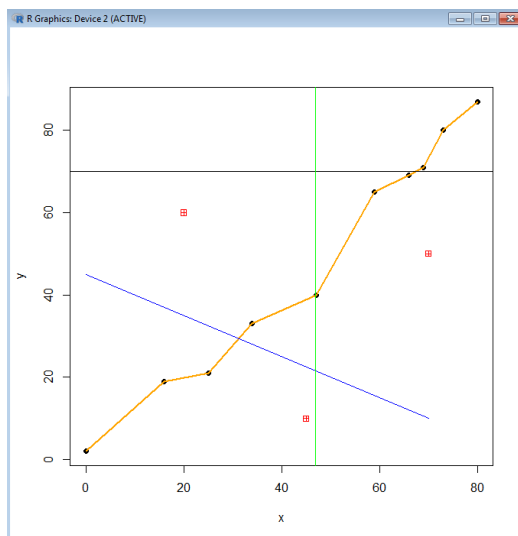


```
> lines(c(0,70),c(45,10),col='blue')
```



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

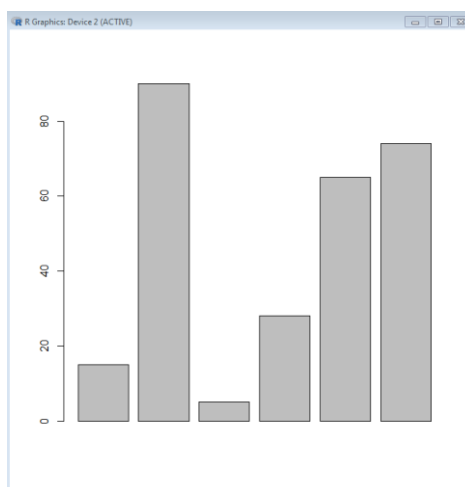
```
> abline(h=70) # o h é de horizontal
> abline(v=mean(x),col='green') # o v é de vertical
```



15.4 Gráfico de Barra

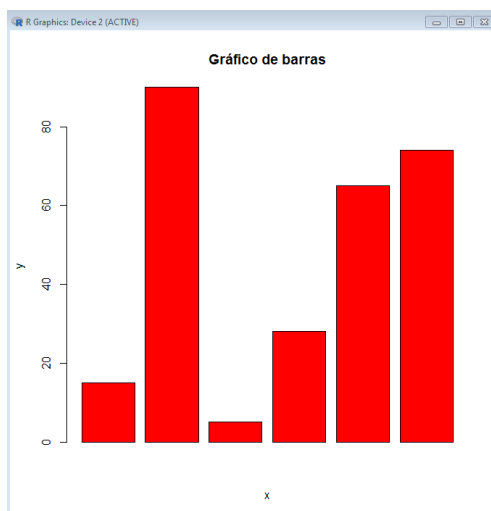
Para fazer gráfico de barras, usa-se a função *barplot()* em que o argumento é um vetor.

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



Pode-se alterar a aparência do gráfico:

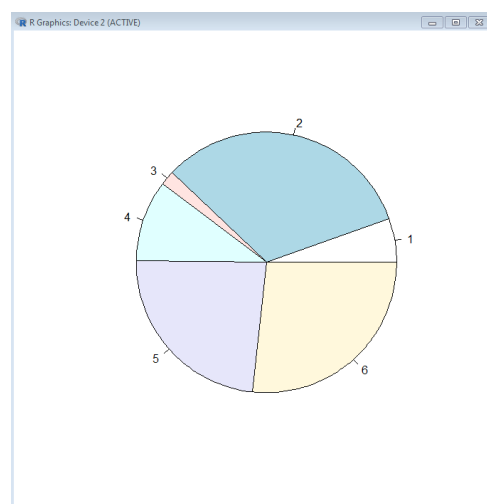
```
> barplot(c(15,90,5,28,65,74),col='red',xlab='x',ylab='y',main='Gráfico de barras')
```



15.5 Gráfico de Pizza

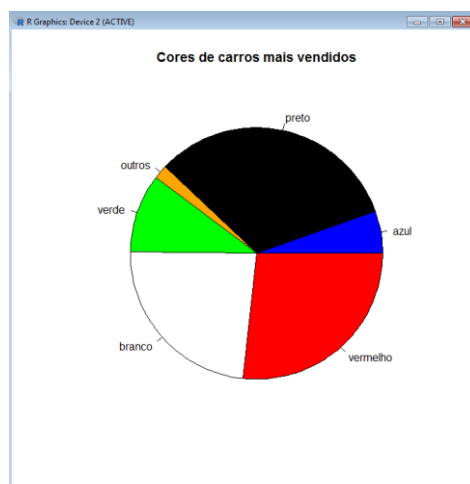
Para fazer gráfico de pizza, usa-se a função `pie()` em que o argumento é um vetor.

```
> 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')
```

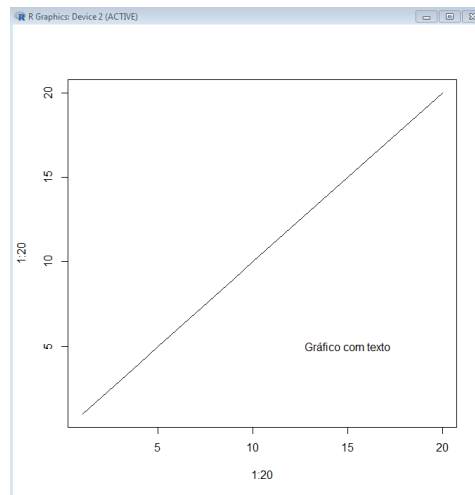


15.6 Inserir Texto em Gráficos

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')
```

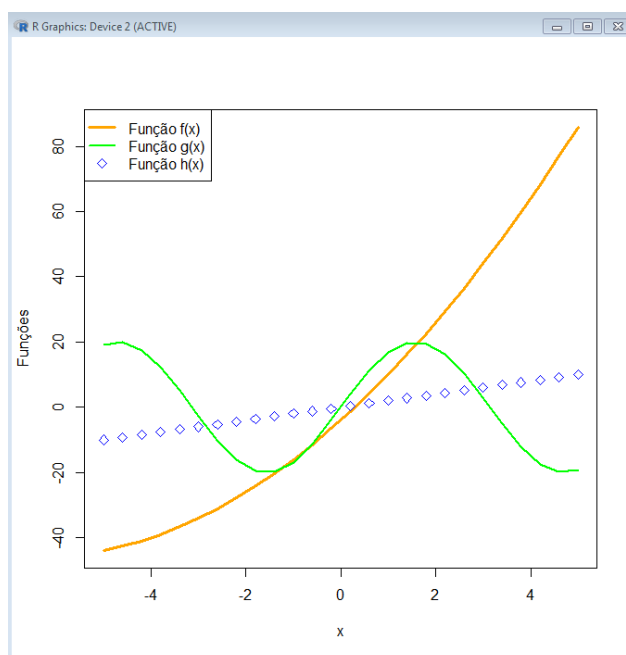


15.7 Legenda em Gráficos

Inserimos legenda 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.

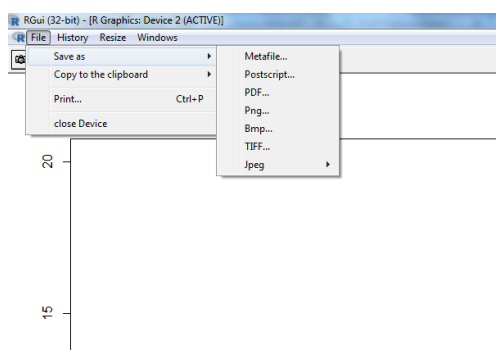
Exemplo:

```
> 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'))
```



15.8 Salvando Gráficos

- No Windows:



- No Linux/Windows:

```
> 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')`

15.9 Animação

Para quem tiver curiosidade sobre animação no R, execute o código a seguir e observe o arquivo Mandelbrot.gif que será gerado:

```

library(fields) # para tim.colors
library(caTools) # para write.gif
m<-400 # grid size
C<-complex( real=rep(seq(-1.8,0.6, length.out=m), each=m ),
  Imag<-rep(seq(-1.2,1.2, length.out=m), m ) )
C<-matrix(C,m,m)

Z<- 0
X<- array(0, c(m,m,20))
for (k in 1:20) {
  Z<-Z^2+C
  X[, ,k] = exp(-abs(Z))
}
write.gif(X, "Mandelbrot.gif", col=tim.colors(256), delay=100)

```



- 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 $y = \sin(x)e^{-x}$. Trace uma linha de cor azul e nomeie adequadamente os eixos.
-

16 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/testes estatísticos já estão programadas e prontas para o uso. Nesta apostila, serão vistos comandos básicos para realizar algumas das análises mais comuns usadas em manipulação de dados.

16.1 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(as.vector(y))
  m <- names(z)[z == 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.

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
```

Observe que com a função `summary()`, os valores foram arredondados.

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

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

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

Lição de casa !

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

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

16.2 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
Logística	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 Normal, *ppois()* para Poisson e assim por diante.

Exemplo:

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

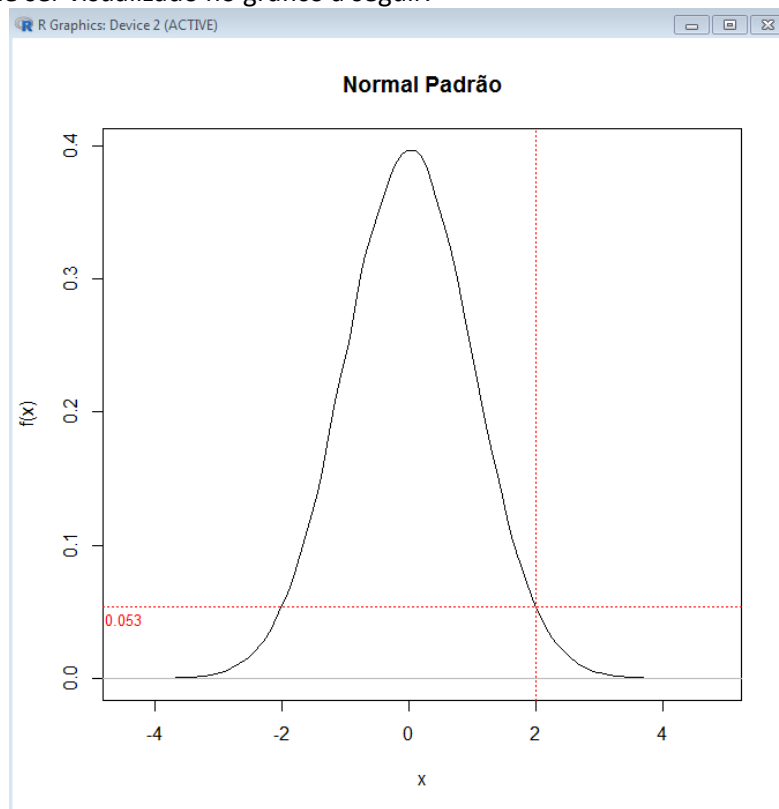
Este valor corresponde ao valor da densidade da Normal:

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

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) # ou rnorm(10,5,2)
[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 do comando `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:
 - $P(X \leq 95)$
 - $P(X > 95)$
-

16.3 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.

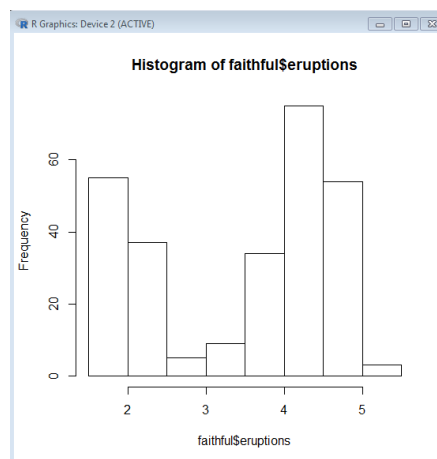
```
> data(faithful)
> ls()
[1] "faithful"
> str(faithful)
'data.frame': 272 obs. of 2 variables:
 $ eruptions: num 3.6 1.8 3.33 2.28 4.53 ...
 $ waiting : num 79 54 74 62 85 55 88 85 51 85 ...
```

Numericamente:

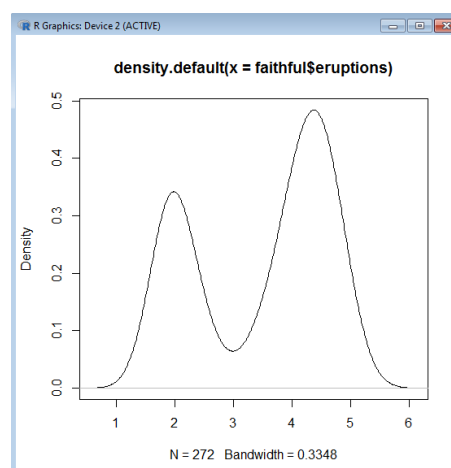
```
> summary(faithful$eruptions)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.600  2.163   4.000   3.488   4.454   5.100
> summary(faithful$waiting)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 43.0   58.0   76.0   70.9   82.0   96.0
```

Graficamente:

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

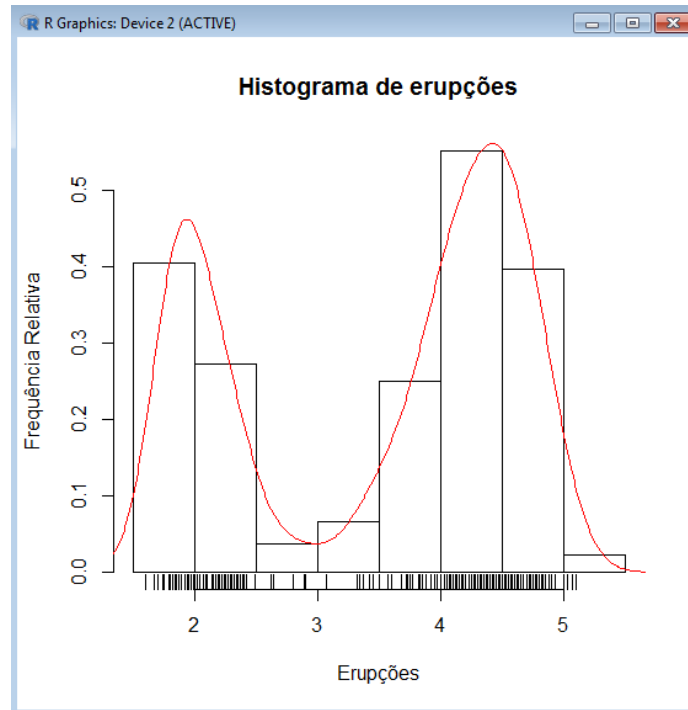


```
> plot(density(faithful$eruptions)) # função densidade de probabilidade
```



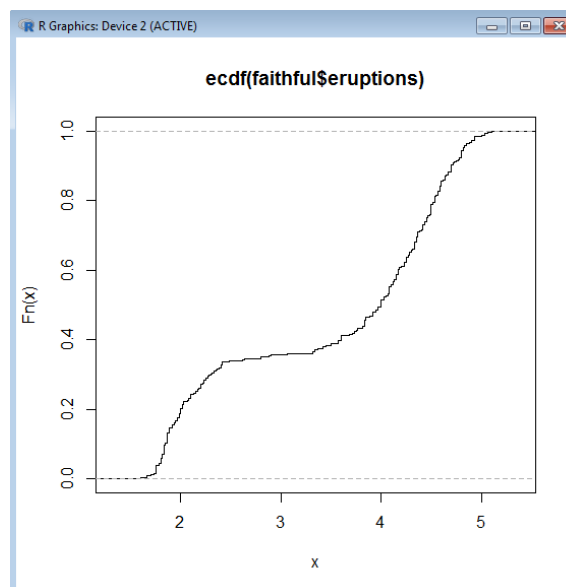
Pode-se plotar os dois gráficos juntos e melhorar a aparência dele.

```
> hist(faithful$eruptions, seq(1.5, 5.5, 0.5), prob=TRUE, main='Histograma de
erupções', xlab='Erupções', ylab='Frequência Relativa')
> lines(density(faithful$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)
```

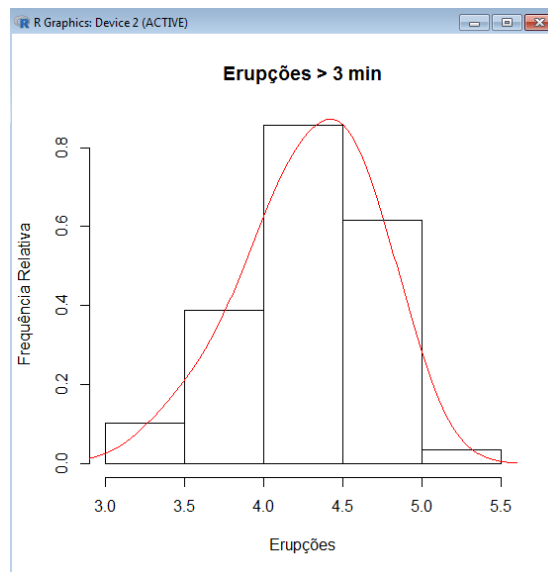


Como se pode observar, esta distribuição não se parece com a distribuição normal. Porém, o que dizer sobre as erupções que duram mais que 3 minutos? A seguir, serão feitas análises para verificar se esse conjunto de dados chamado de “longo” se ajusta a uma distribuição normal.

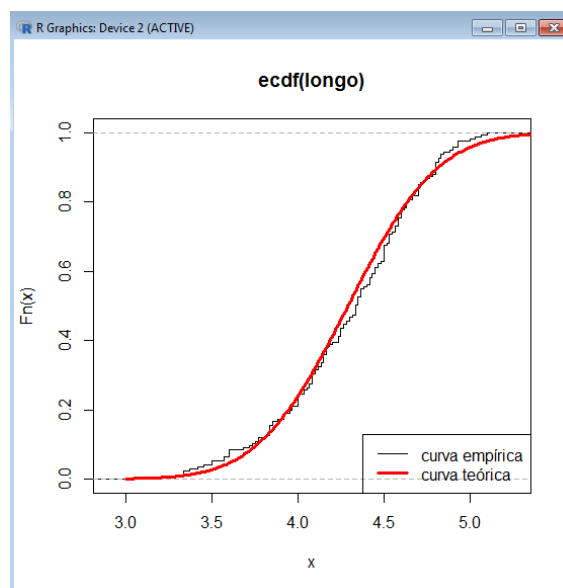
```
> longo<-faithful$eruptions[faithful$eruptions > 3]
> # ou similarmente longo<-faithful[faithful$eruptions>3,1]
> str(longo)
```

```
num [1:175] 3.6 3.33 4.53 4.7 3.6 ...
```

```
> hist(longo, seq(3, 5.5, 0.5), prob=TRUE, main='Erupções > 3 min',
xlab='Erupções', ylab='Frequência Relativa')
> lines(density(longo, bw=0.2), col='red')
```

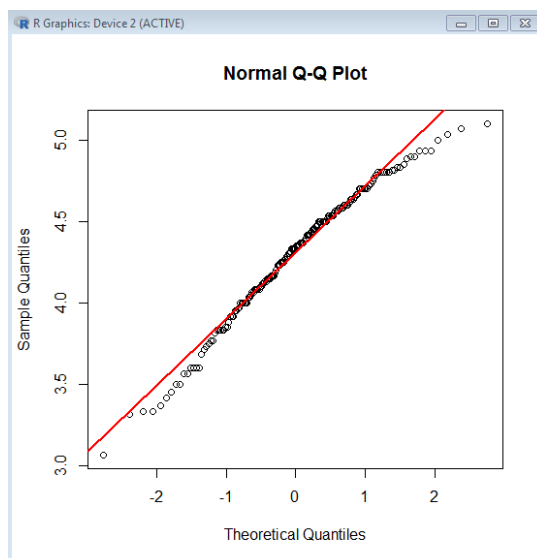


```
> x<-seq(3,5.5,by=0.1)
> plot(ecdf(longo), do.points=FALSE, verticals=TRUE)
> lines(x, pnorm(x, mean=mean(longo), sd=sd(longo)), col='red', lwd=3)
> legend('bottomright', c('curva empírica', 'curva teórica'),
col=c('black', 'red'), lwd=c(1,3))
```



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 $p\text{-valor} < 0.05$; entretanto o teste Kolmogorov-Smirnov retornou um p-valor de 0.4284, significando a aceitação da normalidade dos dados, pois $p\text{-valor} > 0.4284$. 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.

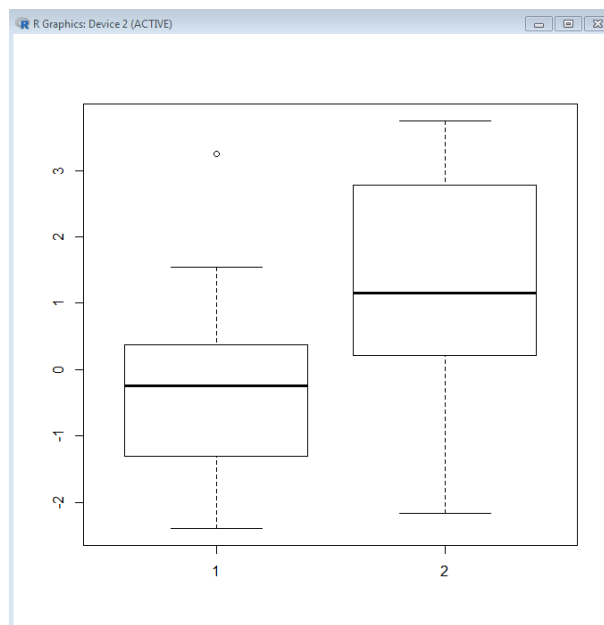
16.4 Comparando Duas Amostras

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 *a*:

```
> IQR<-quantile(a,0.75)-quantile(a,0.25)
> upper_whisker<-min(max(a),quantile(a,0.75)+1.5*IQR)
> lower_whisker<-max(min(a),quantile(a,0.25)-1.5*IQR)
> upper_whisker
[1] 2.812492
> lower_whisker
[1] -2.394206
```

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.
-

16.5 Regressão Linear

Regressão linear significa estudar 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 uma linha reta. 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; 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
$Y \sim X + I(X^2)$	$Y = \beta_0 + \beta_1 X + \beta_2 X^2$	Modelo polinomial; note que a função identidade $I(\cdot)$ 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 em A e B sem

		termos de interação
$Y \sim X_1 : X_2$	$Y = \beta_0 + \beta_1 X_1 X_2$	Um modelo contendo somente a interação de primeira ordem entre X_1 e X_2
$Y \sim X_1 * X_2$	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$	Um modelo de primeira ordem completo com um termo; um código equivalente é $Y \sim X_1 + X_2 + X_1 : X_2$
$Y \sim (X_1 + X_2 + X_3)^2$	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_1 X_2 + \beta_5 X_1 X_3 + \beta_6 X_2 X_3$	Um modelo incluindo todas as interações de primeira ordem até a n -ésima ordem, onde n é dado por $()^n$. 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 (\sim) 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:

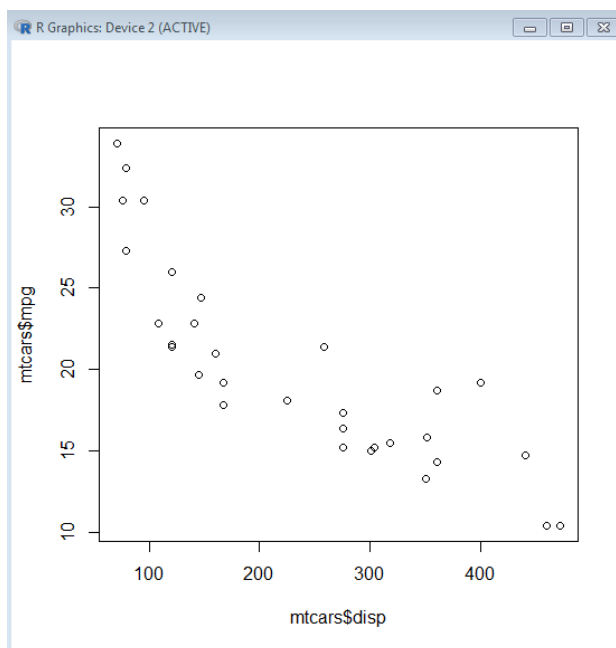
```
[, 1] mpg Miles/(US) gallon
[, 2] cyl Number of cylinders
[, 3] disp Displacement (cu.in.)
[, 4] hp Gross horsepower
[, 5] drat Rear axle ratio
[, 6] wt Weight (1000 lbs)
[, 7] qsec 1/4 mile time
[, 8] vs V/S
[, 9] am Transmission (0 = automatic, 1 = manual)
[,10] gear Number of forward gears
[,11] 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 $Y = \beta_0 + \beta_1 X$.

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

```
call:
lm(formula = mtcars$mpg ~ mtcars$disp)
```

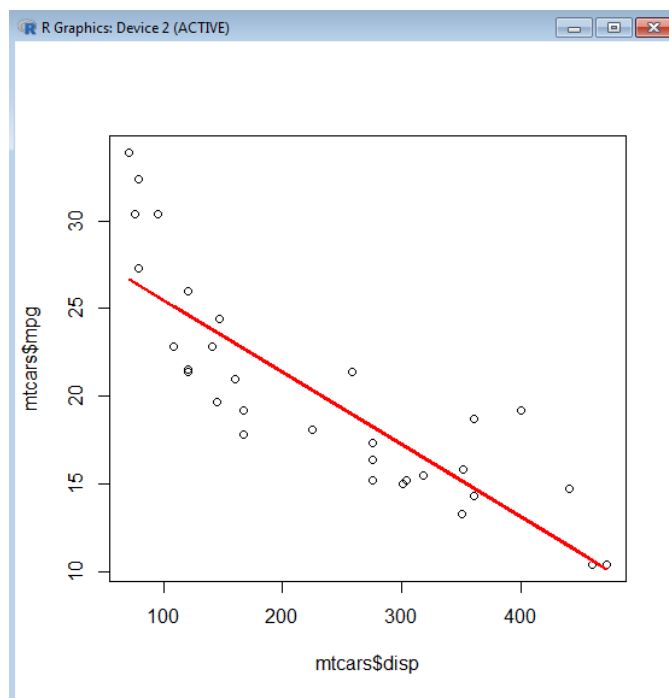
```
coefficients:
(Intercept)  mtcars$disp
 29.59985    -0.04122
```

Extraindo os coeficientes do modelo:

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

Equação da reta ajustada:

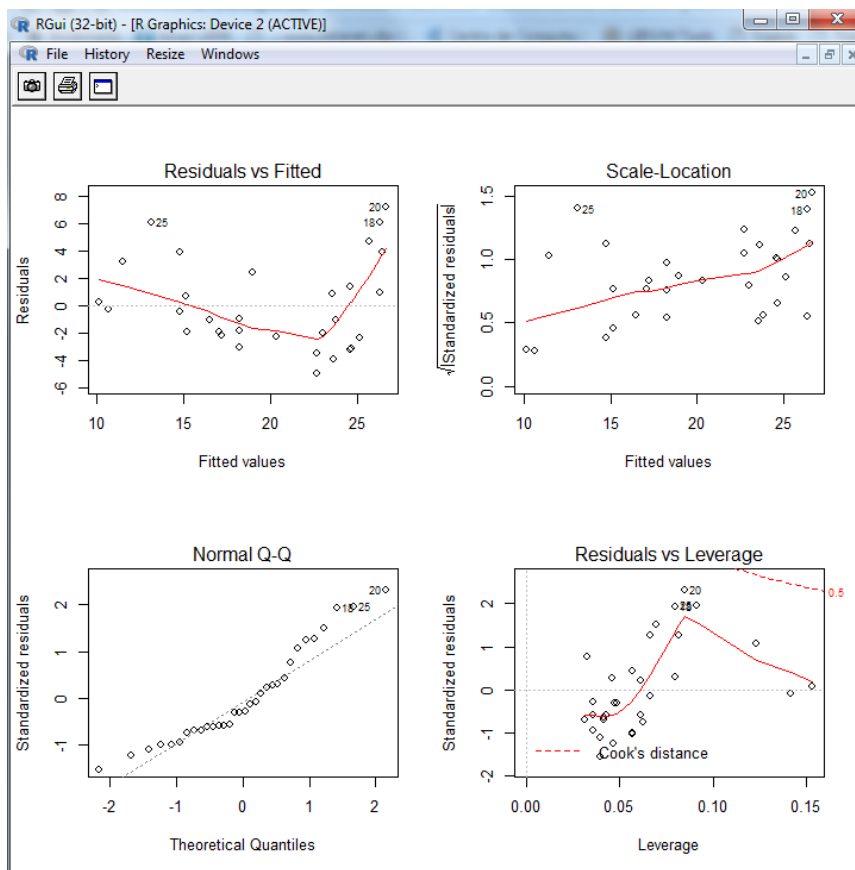
```
> ajuste<-function(x, modelo) coef(modelo)[1]+coef(modelo)[2]*x
> lines(mtcars$disp,ajuste(mtcars$disp,modelo),col='red',lwd=2)
```



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

Mais informações por meio gráfico:

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



Resumo do modelo:

```
> summary(modelo)
```

```
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
```

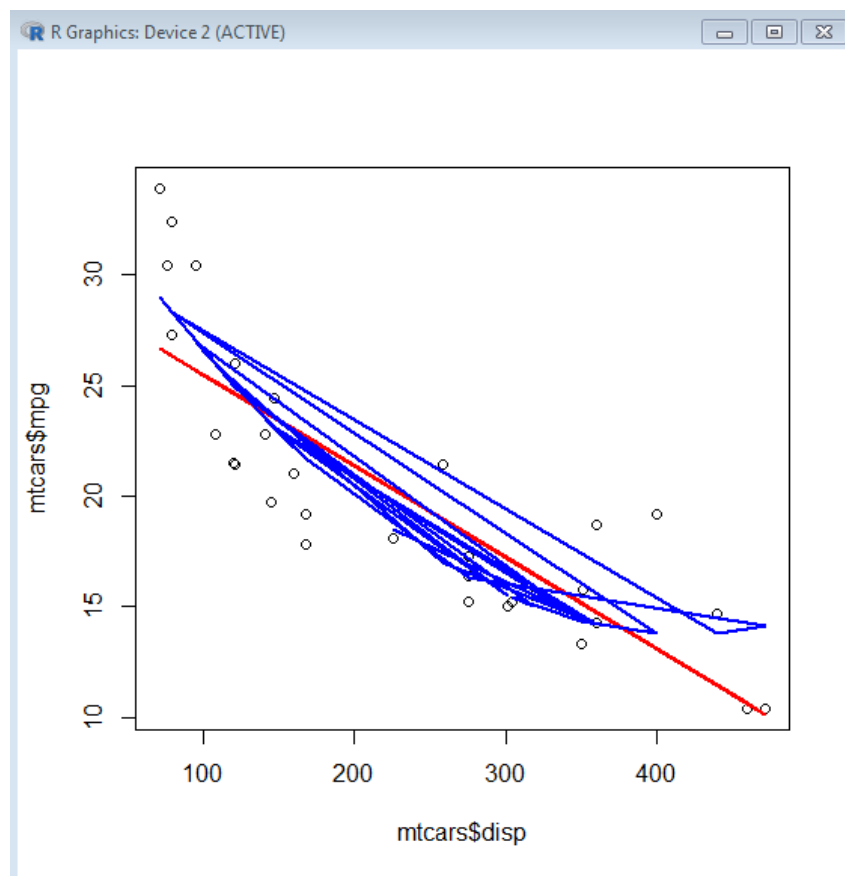
Uma maneira de avaliar se o ajuste foi satisfatório é verificar o indicador *R* quadrado ajustado, que quando maior melhor. Para este exemplo deu 0.709.

Agora ajustando um polinômio de 2º grau:

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

Plotando no gráfico:

```
> ajuste2<-function(x,modelo) coef(modelo)[1]+coef(modelo)[2]*x+coef(modelo)[3]*x^2
> lines(mtcars$disp,ajuste2(mtcars$disp,modelo2),col='blue',lwd=2)
```

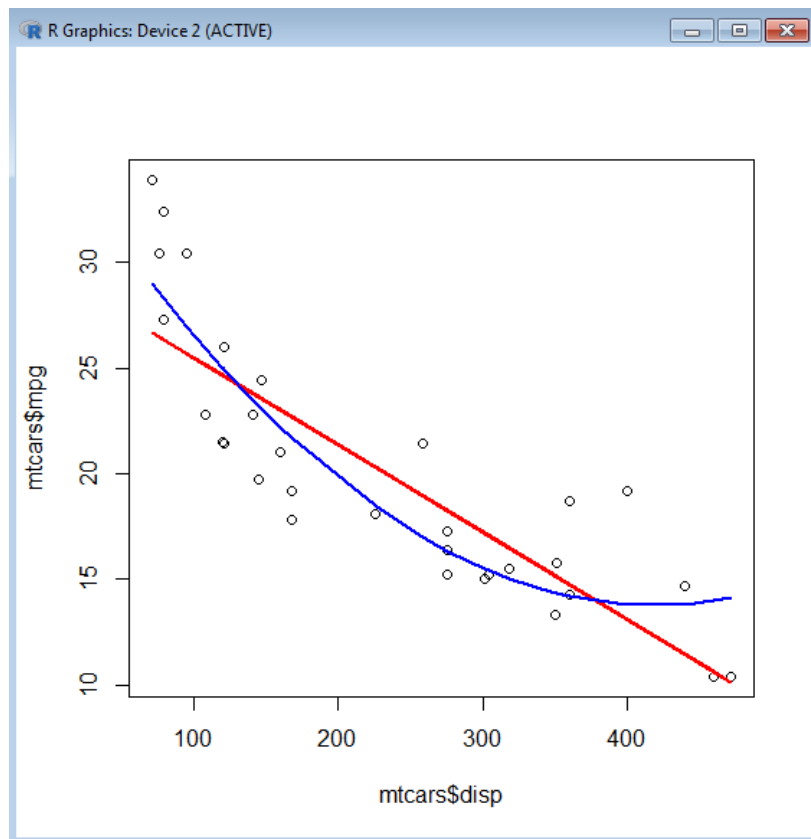


A plotagem não ficou como esperado pois `mtcars$disp` (que são os dados do eixo x) é um vetor não ordenado. Para que se obtenha a curva esperada ordena-se o vetor `mtcars$disp` e indexa-se o vetor pelos índices do vetor ordenado:

```
> ind<-sort.int(mtcars$disp,index.return=TRUE)
> str(ind)
List of 2
 $ x : num [1:32] 71.1 75.7 78.7 79 95.1 ...
 $ ix: int [1:32] 20 19 18 26 28 3 21 27 32 9 ...
```

E recomeçando a plotagem (pois a antiga ficou comprometida):

```
> plot(mtcars$disp,mtcars$mpg)
> lines(mtcars$disp,ajuste(mtcars$disp),col='red',lwd=2)
> lines(mtcars$disp[ind$ix],ajuste2(mtcars$disp[ind$ix]),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.

16.6 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(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”.

```
> 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”.

```
> 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 `predict.glm()`.
-

17 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, URL <http://www.R-project.org>.

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

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

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>