Minicurso de introdução ao R

Semana acadêmica da economia

Marcelo Gelati

12, 14 e 16 de setembro de 2016

Contents

1 Introdução ao R **3**

1.1 Como o R funciona . . . 3
1.2 Criando objetos . . . 3
1.3 Funções e operadores . . . 4
1.4 Scripts . . . 5
1.5 Pacotes . . . 6

2 Aprofundando o R **7**

2.1 Estruturas de dados . . . 7
2.1.1 Vetores atômicos . . . 7
2.1.2 Listas . . . 9
2.1.3 Atributos . . . 10
2.1.4 Fatores . . . 11
2.1.5 Matrizes . . . 12
2.1.6 Arrays . . . 12
2.1.7 Data frames . . . 13
2.2 Subsetting . . . 14
2.2.1 Vetores atômicos . . . 14
2.2.2 Listas . . . 15
2.2.3 Matrizes . . . 16
2.2.4 Data frames . . . 16
2.2.5 Preservação e simplificação . . . 18
2.2.6 Subset e assignment . . . 20
2.3 Gráficos . . . 20
2.3.1 Layout . . . 20
2.3.2 Funções . . . 23
2.3.3 Parâmetros . . . 23

3 Usando o R na prática **26**

3.1 Importar dados reais 26

3.1.1 Diretório 26

3.1.2 Lendo em .csv 26

3.1.3 Lendo em .xlsx 27

3.2 Análise descritiva 27

3.3 Regressão 28

3.4 Gerando PDFs 31

4 Referências **33**

1 Introdução ao R

1.1 Como o R funciona

O R é uma linguagem de programação que funciona, basicamente, através de objetos, operadores e funções (que elas mesmo são objetos). Os *objetos* possuem um nome e podem ser dados, variáveis, funções, resultados, entre outros. Você pode realizar operações com esses objetos através de *operadores* ou de *funções*, sendo que estas necessitam de argumentos para retornar um resultado. A figura 1 resume isto:

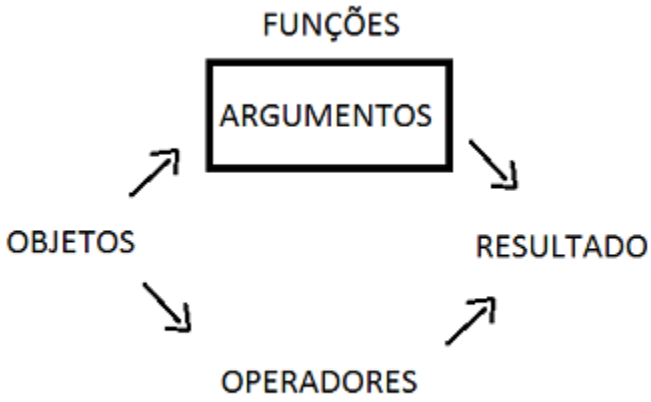


Figure 1: Objetos e funções

1.2 Criando objetos

Para criar um objeto, é preciso usar o operador assign `<-`. O operador também pode ser usado de maneira invertida `->`. Por exemplo, se queremos atribuir o valor **10** a um objeto **n**, devemos fazer o seguinte:

```
> n <- 10
```

A operação realizada acima deve ser lida da seguinte maneira: "n recebe 10". Para verificarmos o que está armazenado no objeto, basta digitar seu nome e apertar enter:

```
> n
[1] 10
```

É importante ressaltar que o R é case-sensitive, isto é, sensível a maiúsculas e minúsculas. Isto significa que um objeto **N** é diferente do objeto **n** por nós criado. Veja:

```
> N <- 20
> N
[1] 20
```

Todo objeto criado no R deve começar com uma letra (a-z ou A-Z). Os caracteres seguintes podem ser letras, números (0-9), underlines (_) e/ou pontos (.).

Podemos atribuir operações matemáticas a um objeto:

```
> soma <- 22 + 5
> soma
[1] 27
```

Mas note que podemos realizar a mesma operação sem ter que criar o objeto! Neste caso, o resultado será mostrado no console.

```
> 22 + 5
[1] 27
```

1.3 Funções e operadores

Como mencionado acima, as funções precisam de argumentos para que possam ser rodadas. Por exemplo, a função `is.atomic()` requer apenas um argumento: o objeto a ser testado. Caso você queira ler mais detalhes sobre alguma função, basta acessar a documentação do R. Para fazer isso, deve-se utilizar uma interrogação `?` antes da função:

```
> ?is.atomic
```

A documentação de operadores (como o `+`, por exemplo) não pode ser acessada através de `?`. Neste caso, deve-se utilizar a função `help`:

```
> help("+")
```

Uma breve lista dos operadores mais básicos está no cartão de referência. Por este motivo não falaremos sobre os operadores.

Quanto às funções, trabalharemos apenas com três: `rep()`, `seq()` e `sample()`.

A função `rep()` serve para criar um vetor com elementos repetidos. Três de seus argumentos são:

- `x`: um vetor
- `times`: um inteiro **m** que fará com que o vetor seja repetido **m** vezes
- `each`: um inteiro **m** que fará com que cada elemento do vetor x seja repetido **m** vezes

Mostrando alguns exemplos:

```
> rep(1:3, times = 2)
[1] 1 2 3 1 2 3
> rep(1:3, each = 2)
[1] 1 1 2 2 3 3
> rep(1:3, times = c(1, 2, 3))
[1] 1 2 2 3 3 3
```

Note que se botarmos um vetor para o argumento `times` e não um inteiro, os valores do vetor x foram repetidos tantas vezes quanto os valores do vetor de `times` dizem. Em nosso exemplo, o valor 1 foi repetido uma vez, o valor 2 foi repetido duas vezes e o valor 3 foi repetido três vezes.

A função `seq()` serve para criarmos uma sequência de números. Alguns de seus argumentos são:

- `from`: um inteiro **m** que denota o valor inicial

- `to`: um inteiro **m** que denota o valor máximo
- `by`: um inteiro **m** que dá o aumento da sequência

Mais alguns exemplos:

```
> seq(from = 2, to = 3, by = 0.1)
 [1] 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0
> seq(from = 4, to = 2, by = -1)
[1] 4 3 2
> seq(from = 1, to = 2, by = 0.3)
[1] 1.0 1.3 1.6 1.9
```

A função `sample()` serve para gerarmos amostras aleatórias. Seus argumentos são:

- `x`: um vetor
- `size`: tamanho da amostra
- `replace`: um valor lógico que indica se a amostra deve ter reposição
- `prob`: um vetor que contém a probabilidade de cada cair cada valor incluso no vetor x

Exemplos:

```
> sample(c("mexerica", "amora", "cereja"), size = 10, replace = TRUE)
 [1] "mexerica" "cereja"   "mexerica" "cereja"   "amora"    "mexerica"
 [7] "cereja"   "mexerica" "cereja"   "cereja"
> sample(1:6, size = 6, replace = FALSE)
[1] 5 1 3 4 6 2
> sample(0:1, size = 15, replace = TRUE, prob = c(0.3, 0.7))
 [1] 1 0 0 1 0 1 1 1 0 0 1 0 1 0 0
```

1.4 Scripts

Um script é basicamente um arquivo de texto que contém praticamente os mesmos comandos que você usaria no console do R. Se você estiver usando o RStudio, é possível criar um novo script utilizando o atalho *Ctrl + Shift + N* ou clicando embaixo de File, no canto superior esquerdo, no ícone da folha de papel com um símbolo de mais `+`.

Não há nenhum mistério no script. Como comentado logo acima, ele é basicamente um arquivo de texto. Ele servirá, basicamente, para você anotar comentários, lembrar de pacotes que utilizou em um trabalho, guardar funções importantes ou qualquer outra coisa que você queira salvar.

Na linguagem do R, se o texto for precedido de uma cerquilha `#` ele será interpretado como um comentário e não será executado. Por exemplo, suponha que queiramos criar um script para nos lembrar como criar objetos e realizar operações entre eles:

```
####### CRIANDO OBJETOS

# Neste script, mostraremos como criar um objeto
# Para criar qualquer objeto, precisamos utilizar o operador assign

x <- 2
```

```r
# Acima estamos criando o objeto x e estamos atribuindo a ele o valor 2
# Vamos criar agora o objeto y com o valor 3

y <- 3

# Agora, vamos realizar uma série de operações com os objetos

x + y
x*y
x - y
y - x
```

Se você estiver utilizando o RStudio, há duas maneiras de rodar o script: clicando em Run no canto superior direito da janela do script ou utilizando o atalho *Ctrl + Enter*. Ao clicar em Run, apenas a linha selecionada será rodada. Para rodar o script por inteiro selecione todo o arquivo e rode.

1.5 Pacotes

Todas funções que aqui utilizamos são do pacote `base` do R. É possível que aquilo que você deseja fazer não tenha sido implementado no R. No entanto, por ele ser open source e contar com uma comunidade gigantesca, é muito possível que alguém já tenha criado um pacote que resolva seu problema.

Para instalar um pacote é preciso utilizar a função `install.packages()`. É preciso que o nome do pacote esteja entre aspas. Após o pacote ter sido instalado em seu computador é preciso rodá-lo no R. Para isso, pode-se usar tanto a função `require()` como a função `library()`. Diferentemente da função de instalação, estas últimas não requerem que o nome do pacote esteja entre aspas. Para você usar um pacote é preciso rodá-lo toda vez que você inicia uma nova sessão do R.

Abaixo listamos alguns pacotes:

- `installr`: verifica se existe alguma atualização para o R e a instala
- `ggplot2`: gráficos mais elegantes
- `googlesheets`: importar dados das planilhas Google
- `gdata`: importar arquivos .xls e .xlsx; requer perl para rodar
- `XLConnect`: importar arquivos .xls e .xlsx; requer Java para rodar

2 Aprofundando o R

2.1 Estruturas de dados

Podemos separar as estruturas de objetos no R pela sua composição ou pela dimensão. A composição pode ser homogênea (apenas um tipo no objeto) ou heterogênea (vários tipos diferentes no objeto). A dimensão pode variar de 1 a n dimensões.

Dim/Comp	Homogêneo	Heterogêneo
1 dimensão	Vetor atômico	Lista
2 dimensões	Matriz	Data frame
n dimensões	Array	

Uma função extremamente útil para verificar a estrutura de um objeto é `str()`, que fornece uma breve descrição sobre o objeto. Outra função de resumo de objetos é a `summary()`.

Todas cinco estruturas listadas acima possuem *atributos*. De todos atributos possíveis, há dois que são intrínsecos a qualquer estrutra: o tipo (type) e o comprimento (length).

2.1.1 Vetores atômicos

Para criarmos um vetor atômico utilizamos a função `c()`.

```
> n <- c(1, 2, 3)
```

Note que os vetores atômicos são *planos*, isto é, criar um vetor atômico dentro de outro não traz nenhuma diferença ao resultado final.

```
> x <- c(1, 2, 3, 4)
> y <- c(1, 2, c(3, 4))
> identical(x, y)
[1] TRUE
```

Como comentado acima, as estruturas de dados possuem um atributo intrínseco chamado tipo. Trabalharemos aqui com quatro tipos diferentes: **logical**, **integer**, **double** e **character**.

Para criarmos um vetor do tipo lógico (logical), devemos utilizar os valores `TRUE` e `FALSE`. `T` e `F` também são reconhecidos pelo R. Para criar um vetor de inteiros (integer) devemos utilizar o sufixo L. Para criar um vetor de reais (double) basta criar um vetor com números. Para criar um vetor caractérico (char) devemos botar as palavras entre aspas:

```
> logic <- c(T, F, TRUE, FALSE)
> integer <- c(1L, 2L, 3L)
> double <- c(2, 8, 9, 3)
> charac <- c("maria", "joao")
>
> logic
> integer
> double
> charac
[1]  TRUE FALSE  TRUE FALSE
```

```
[1] 1 2 3
[1] 2 8 9 3
[1] "maria" "joao"
```

Para verificarmos o tipo de um objeto, utilizamos a função `typeof()`:

```
> typeof(logic)
[1] "logical"
> typeof(integer)
[1] "integer"
> typeof(double)
[1] "double"
> typeof(charac)
[1] "character"
```

Se quiseremos testar o tipo de um objeto, utilizamos as funções `is.character()`, `is.double()`, `is.logical()`, entre outras.

```
> is.logical(logic)
[1] TRUE
> is.character(charac)
[1] TRUE
```

Note que a função `is.numeric()` teste se o vetor é numérico ou não. Logo, retornará verdadeiro tanto para um vetor do tipo integer como um do tipo double.

```
> is.numeric(charac)
[1] FALSE
> is.numeric(double)
[1] TRUE
```

O que acontece se criarmos um vetor com dois tipos diferentes em sua composição?

```
> b <- c("piscina", 1, 0, FALSE)
> typeof(b)
[1] "character"
```

O vetor atômico b assumiu o tipo caractérico porque existe algo no R chamado de *regra de flexibilização*. Esta regra atesta que em uma estrutura homogênea apenas um tipo será suportado.

A ordem de flexibilidade dos tipos, do menos para o mais flexível, é a seguinte:

- logical
- integer
- double
- character

Assim, sempre que houver mais de um tipo diferente no mesmo vetor atômico, todos valores assumirão o tipo mais flexível.

É possível forçar um vetor a assumir um tipo específico através das funções `as.character()`, `as.double()`, `as.integer()` e `as.logical()`. Veja a tabela abaixo para entender o que acontece na coerção forçada:

	as.logical	as.numeric
Lógico	permanece igual	FALSE = 0, TRUE = 1
Numérico	0 = FALSE, outros números = 1	permanece igual
Caractérico	NA, se o texto não for lógico	NA, se o texto não for um número

	as.character
Lógico	fica em formato textual
Numérico	fica em formato textual
Caractérico	permanece igual

2.1.2 Listas

As listas, diferentemente dos vetores, são heterogêneas. Isto significa que suportam mais de um tipo em sua composição. Veja:

```
> ls1 <- list(1:4, c(1, 5), c("massa", "pizza", "macarrao"), c(T, F, F))
> str(ls)
function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
    pattern, sorted = TRUE)
```

Outro aspecto que diferencia as listas de vetores atômicos é que as listas são recursivas. Isso significa que é possível criar uma lista dentro de outra lista.

```
> ls2 <- list(1, 2, 3, 4)
> ls2
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 4
>
> ls3 <- list(list(1, 2), 3, 4)
> ls3
[[1]]
[[1]][[1]]
[1] 1

[[1]][[2]]
[1] 2


[[2]]
[1] 3
```

```
[[3]]
[1] 4
```

É possível retirar todos os objetos de dentro de uma lista usando a função `unlist()`. Ao fazer isso, o vetor atômico resultante obedecerá as regras de flexibilização.

```
> ls1
[[1]]
[1] 1 2 3 4

[[2]]
[1] 1 5

[[3]]
[1] "massa"    "pizza"    "macarrao"

[[4]]
[1]  TRUE FALSE FALSE
> unl <- unlist(ls1)
> str(unl)
 chr [1:12] "1" "2" "3" "4" "1" "5" "massa" "pizza" ...
```

2.1.3 Atributos

Como falamos no início do capítulo, há dois atributos intrínsecos em qualquer objeto: o tipo e o comprimento. Para verificarmos o tipo de um objeto utilizamos a função `typeof()`. Para verificarmos o comprimento de um objeto utilizamos a função `length()`.

Vamos falar do de um novo atributo agora: nomes (names). Podemos nomear os elementos de um vetor de duas maneiras - na hora de criar o vetor ou por atribuição.

```
> d <- c(a = 1, b = 2, c = 3)
> d
a b c
1 2 3
>
> e <- c(1, 2, 3)
> names(e) <- c("a", "b", "c")
> e
a b c
1 2 3
```

Podemos verificar os atributos que um objeto contém com a função `attributes()`. Esta função retornará todos atributos adicionais que um objeto tem.

```
> attributes(d)
$names
[1] "a" "b" "c"
>
> b
[1] "piscina" "1"       "0"       "FALSE"
> attributes(b)
NULL
```

O atributo dimensão (dim) refere-se às dimensões do objeto. Como estamos tratando apenas de estruturas unidimensionais, este atributo não está presente.

```
> dim(d)
NULL
```

Para eliminar os nomes de um objeto, basta utilizar a função `unname()` ou atribuir o valor NULL aos nomes do objeto.

```
> d <- unname(d)
> d
[1] 1 2 3
>
> names(e) <- NULL
> e
[1] 1 2 3
```

2.1.4 Fatores

Os fatores são casos especiais de vetores atômicos por conterem um atributo restrito a eles, os léveis.

```
> fac <- factor(1:6)
> fac
[1] 1 2 3 4 5 6
Levels: 1 2 3 4 5 6
> attributes(fac)
$levels
[1] "1" "2" "3" "4" "5" "6"

$class
[1] "factor"
```

Os fatores são úteis para tratarmos de dados categóricos por causa da função `table()`.

```
> frutas <- sample(c("kiwi", "amora", "ameixa", "morango"), size = 1000,
+                  replace = TRUE, prob = c(0.2, 0.3, 0.15, 0.35))
> fac_frutas <- factor(frutas)
> table(fac_frutas)
fac_frutas
 ameixa   amora    kiwi morango
    175     253     202     370
```

Não é possível inserir nenhum valor que não esteja nos léveis do fator. Ao tentar fazer isso, um valor NA será introduzido no lugar.

```
> fac
[1] 1 2 3 4 5 6
Levels: 1 2 3 4 5 6
> fac[1] <- "mexerica"
Warning in `[<-.factor`(`*tmp*`, 1, value = "mexerica"): invalid factor
level, NA generated
> fac
[1] <NA> 2    3    4    5    6
Levels: 1 2 3 4 5 6
```

2.1.5 Matrizes

Para criar uma matriz utilizamos a função `matrix()`. Seus argumentos são:

- `x`: os valores dentro da matriz
- `ncol`: o número de colunas
- `nrow`: o número de linhas
- `byrow`: um valor lógico; define se os valores vão ser organizados por linhas

```
> mat1 <- matrix(1:6, ncol = 2, nrow = 3, byrow = FALSE)
> mat1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> mat2 <- matrix(1:6, ncol = 2, nrow = 3, byrow = TRUE)
> mat2
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

A função `length()` retornará todos os valores dentro da matriz. Se você quiser ver as dimensões, use a função `dim()`.

```
> length(mat1)
[1] 6
> dim(mat1)
[1] 3 2
```

É possível criar uma matriz através das funções `cbind()` e `rbind()`. A primeira unirá vetores através de colunas, já a segunda unirá através de linhas.

```
> a <- c(1, 2, 3)
> b <- c(4, 5, 6)
>
> cbind(a, b)
     a b
[1,] 1 4
[2,] 2 5
[3,] 3 6
> rbind(a, b)
  [,1] [,2] [,3]
a    1    2    3
b    4    5    6
```

2.1.6 Arrays

Arrays são versões de matrizes generalizadas para n dimensões (sendo n maior que 2). Não nos estenderemos nesta parte, resumindo-a a um exemplo.

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

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

, , 2

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

2.1.7 Data frames

Data frames são muito parecidos com matrizes. Como são heterogêneos, cada coluna pode suportar um tipo diferente.

```
> dat <- data.frame(numeros = c(1, 5, 6),
+                   frutas = c("mirtilo", "banana", "laranja"),
+                   logicos = c(T, F, F))
> dat
  numeros  frutas logicos
1       1 mirtilo    TRUE
2       5  banana   FALSE
3       6 laranja   FALSE
> str(dat)
'data.frame':   3 obs. of  3 variables:
 $ numeros: num  1 5 6
 $ frutas : Factor w/ 3 levels "banana","laranja",..: 3 1 2
 $ logicos: logi  TRUE FALSE FALSE
```

Note que a coluna *frutas* foi criada como sendo um fator. Para suprimir este comportamento, basta usar o argumento `stringAsFactors = FALSE`

```
> dat2 <- data.frame(numeros = c(1, 5, 6),
+                   frutas = c("mirtilo", "banana", "laranja"),
+                   logicos = c(T, F, F),
+                   stringsAsFactors = FALSE)
> dat2
  numeros  frutas logicos
1       1 mirtilo    TRUE
2       5  banana   FALSE
3       6 laranja   FALSE
> str(dat2)
'data.frame':   3 obs. of  3 variables:
 $ numeros: num  1 5 6
```

```
 $ frutas : chr  "mirtilo" "banana" "laranja"
 $ logicos: logi  TRUE FALSE FALSE
```

Se você usar `typeof()` em um data frame o R lhe retornará list. Isso acontece porque data frames são listas em duas dimensões. Para verificar o tipo de uma coluna especifica, use o cifrão `$` e o nome da coluna depois do objeto.

```
> typeof(dat)
[1] "list"
> typeof(dat$numeros)
[1] "double"
```

2.2 Subsetting

Subsetting é o ato de retirar de um objeto apenas as partes que lhe interessam. Imagine que você tem um data frame com 23 colunas, mas que você precisa apenas da quinta para realizar sua análise. Com o subsetting é possível retirar apenas esta quinta coluna, sem precisar ter que trazer todo o conjunto de dados novamente.

Começaremos a falar de subsetting em vetores atômicos, que são a estrutura mais simples, para depois generalizarmos para as outras dimensões.

2.2.1 Vetores atômicos

Vamos criar um vetor atômico x:

```
> x <- c(2.1, 4.2, 5.3, 8.4)
> x
[1] 2.1 4.2 5.3 8.4
```

A casa decimal que tem em cada valor do vetor corresponde a sua posição dentro do vetor. Caso você queira retirar o segundo valor do vetor, basta usar o operador `[]` depois do vetor.

```
> x[2]
[1] 4.2
```

É possível fazer subsetting com vetores atômicos para retirar mais de um valor.

```
> x[c(1, 3)]
[1] 2.1 5.3
>
> # Note que a ordem que você especifica o subsetting importa.
>
> x[c(3, 1)]
[1] 5.3 2.1
```

É possível também pedir valores repetidos e não-inteiros.

```
> x[c(2, 2)]
[1] 4.2 4.2
> x[c(2.1, 2.9)]
[1] 4.2 4.2
```

Se colocarmos um sinal negativo - antes do vetor dentro do subsetting, aqueles valores serão omitidos.

```
> x[-c(3, 4)]
[1] 2.1 4.2
```

O subsetting pode ser feito também com valores lógicos ou condições.

```
> x[c(T, F, T, F)]
[1] 2.1 5.3
>
> # A reciclagem ocorre para o subsetting. O subset acima é equivalente ao de baixo.
>
> x[c(T, F)]
[1] 2.1 5.3
>
> x[x >= 5]
[1] 5.3 8.4
```

É possível fazer o subset por nome caso o vetor tenha nomes.

```
> y <- c(a = 1, b = 2, c = 3)
> y["a"]
a
1
```

2.2.2 Listas

O subset de listas é extremamente parecido com o de vetores atômicos. Além do subset através dos colchetes simples `[]`, é possível realizar o subset através dos colchetes duplos `[[]]`. O resultado no final será diferente devido à *simplificação* ou *preservação* do objeto. Discutiremos este assunto logo em seguida. No entanto, verifique a diferença dos resultados.

```
> ls <- list(1:5, c(T, F, T), c("pera", "melão", "melancia"))
> ls
[[1]]
[1] 1 2 3 4 5

[[2]]
[1]  TRUE FALSE  TRUE

[[3]]
[1] "pera"     "melão"    "melancia"
```

```
> ls[2]
```

```
[[1]]
[1]  TRUE FALSE  TRUE
```

```
> ls[[2]]
```

```
[1]  TRUE FALSE  TRUE
```

```
> str(ls[2])
```

```
List of 1
 $ : logi [1:3] TRUE FALSE TRUE
```

```
> str(ls[[2]])
```

```
 logi [1:3] TRUE FALSE TRUE
```

2.2.3 Matrizes

As matrizes possuem duas dimensões. Isto significa que o operador de subset `[]` necessitará de dois argumentos: um para o número de linhas a serem retiradas e outro para o número de colunas. Veja:

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

Vamos retirar a primeira e a terceira linha junto com a segunda e a terceira coluna.

```
> mat[c(1, 3), c(2, 3)]
     [,1] [,2]
[1,]    4    7
[2,]    6    9
>
> # Não especificar um dos argumentos é o mesmo que pedir para que todas as colunas sejam retiradas
>
> mat[c(1, 3), ]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    3    6    9
```

2.2.4 Data frames

O subset de data frames é igual ao subsetting de matrizes (2 dimensões) ou ao subsetting de listas (heterogeneidade). É possível realizar tanto de um modo como de outro.

```
> dat <- data.frame(a = 1:10, b = letters[1:10], c = LETTERS[1:10])
> dat
    a b c
1   1 a A
2   2 b B
3   3 c C
4   4 d D
5   5 e E
6   6 f F
7   7 g G
```

```
8   8 h H
9   9 i I
10 10 j J
```

Vamos fazer o subset primeiro como se fosse uma lista.

```
> dat[1]
    a
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10 10
```

Agora como se fosse uma matriz.

```
> dat[ , 1]
 [1]  1  2  3  4  5  6  7  8  9 10
```

Note que os resultados foram diferentes. Isto é a *preservação* e a *simplificação*, respectivamente.

Uma outra maneira interessante de realizar subsets em data frames é através de condições lógicas. Veja a construção abaixo passo a passo.

```
> # Queremos que o nosso data frame seja retornado apenas com as linhas que tenham valor
> # maior ou igual a 5 na coluna a.
> # Primeiro fazemos a condição.
>
> dat[ , 1] >= 5
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
>
> # Note que isto retornou um vetor lógico. Coloquemos a condição acima dentro
> # de um subsetting.
>
> dat[dat[ , 1] >= 5, ]
    a b c
5   5 e E
6   6 f F
7   7 g G
8   8 h H
9   9 i I
10 10 j J
>
> # Assim, retiramos as linhas que não queríamos. Isto acontece, pois o
> # subset acima é equivalente ao subset abaixo.
>
> dat[c(F, F, F, F, T, T, T, T, T, T), ]
    a b c
```

```
5   5 e E
6   6 f F
7   7 g G
8   8 h H
9   9 i I
10 10 j J
>
> # Que significa que não queremos as cinco primeiras linhas.
```

2.2.5 Preservação e simplificação

Imagine as estruturas de dados como vagões de trens com cargas em seus interiores. Fazendo uma analogia, a preservação irá retirar o vagão com a carga dentro na hora de fazer o subsetting. A simplificação retirará apenas a carga.

Abaixo temos uma tabela com um resumo dos operadores de subsetting para realizar a simplificação ou a preservação.

Estrutura de dados	Simplificação	Preservação
Vetor atômico	x[[ ]]	x[ ]
Lista	x[[ ]] ou x$	x[ ]
Fator	x[, drop = T]	x[ ]
Matriz	x[, 1]	x[, 1, drop = F]
Data frame	x[, 1], x[[1]] ou x$	x[, 1, drop = F] ou x[1]

Vamos realizar exemplos de simplificação agora.

2.2.5.1 Vetores atômicos

Se um vetor atômico possuir nome, a simplificação retornará os valores sem seu nome.

```
> y
a b c
1 2 3
> y[[1]]
[1] 1
```

2.2.5.2 Listas

As listas perderão sua estrutura de lista.

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

[[2]]
[1]  TRUE FALSE  TRUE

[[3]]
[1] "pera"     "melão"    "melancia"
```

```
> ls[[1]]
[1] 1 2 3 4 5
> is.list(ls[[1]])
[1] FALSE
```

2.2.5.3 Fatores

O fator perderá léveis que não aparecem no subset.

```
> fac
[1] <NA> 2    3    4    5    6
Levels: 1 2 3 4 5 6
> fac[2, drop = T]
[1] 2
Levels: 2
```

2.2.5.4 Matrizes

As matrizes perderão sua estrutura bidimensional.

```
> mat
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> mat[ , 1]
[1] 1 2 3
> is.matrix(mat[ , 1])
[1] FALSE
```

2.2.5.5 Data frames

Os data frames perderão sua estrutura bidimensional.

```
> dat
    a b c
1   1 a A
2   2 b B
3   3 c C
4   4 d D
5   5 e E
6   6 f F
7   7 g G
8   8 h H
9   9 i I
10 10 j J
> dat$a
 [1]  1  2  3  4  5  6  7  8  9 10
> is.atomic(dat$a)
[1] TRUE
```

2.2.6 Subset e assignment

Como você já deve ter realizado, é possível realizar o subsetting em conjunto com o assignment caso você queira mudar um valor de um vetor, por exemplo,

```
> x
[1] 2.1 4.2 5.3 8.4
> x[2] <- 23
> x
[1]  2.1 23.0  5.3  8.4
```

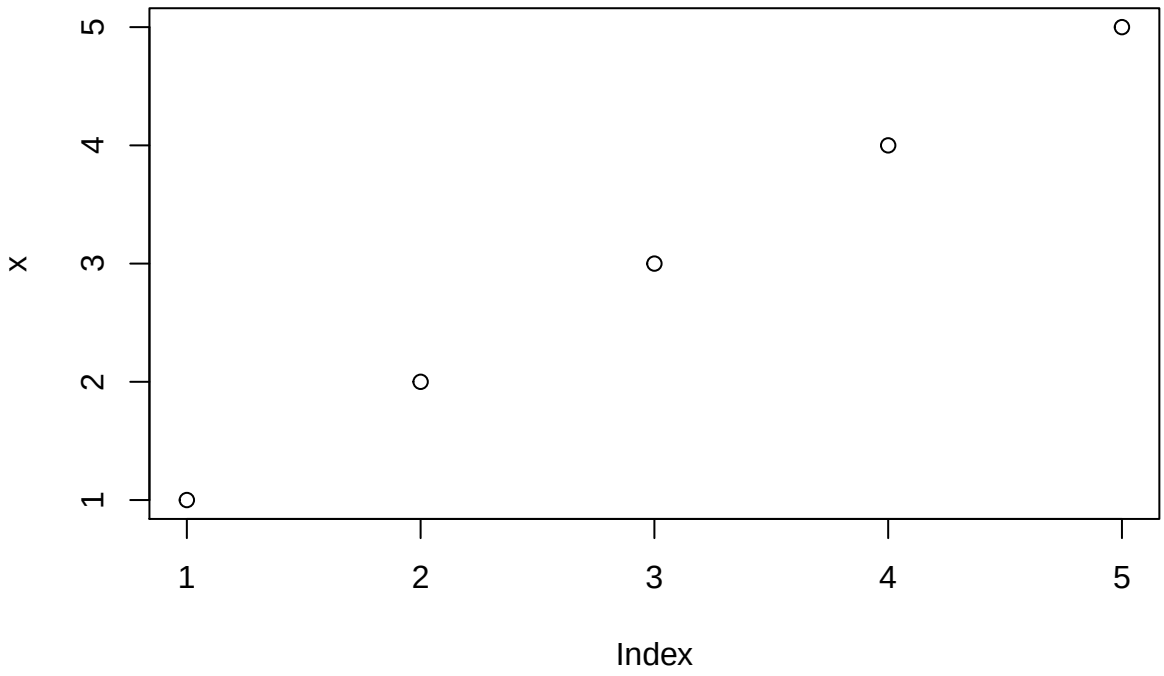
2.3 Gráficos

É possível criar muitos gráficose customizá-los de muitas maneiras através do R. Você pode verificar as demonstrações que o R tem de gráficos através de `demo(graphics)` ou `demo(persp)`.

2.3.1 Layout

No RStudio os gráficos são apresentados na tela no canto inferior direito. Toda vez que você cria um gráfico, uma nova janela é criada. Veja:

```
> x <- 1:5
> plot(x)
```

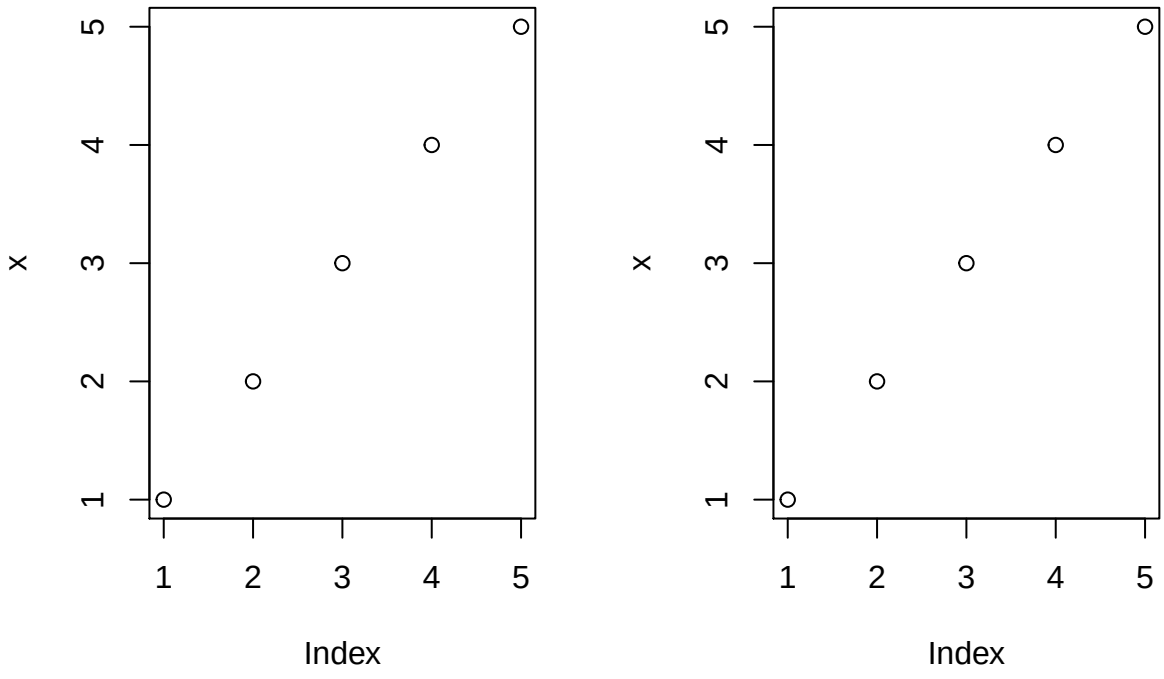


No entanto, isso pode ser modificado através de `layout()`. Esta função, em conjunto com matrizes, permite que você altere a disposição dos gráficos na tela. Se você tiver a seguinte matriz:

```
> matriz <- matrix(1:2, nrow = 1)
> matriz
     [,1] [,2]
[1,]    1    2
```

E colocá-la dentro da função `layout()`, a disposição de gráficos será repartida na metade:

```
> layout(matriz)
> plot(x)
> plot(x)
```

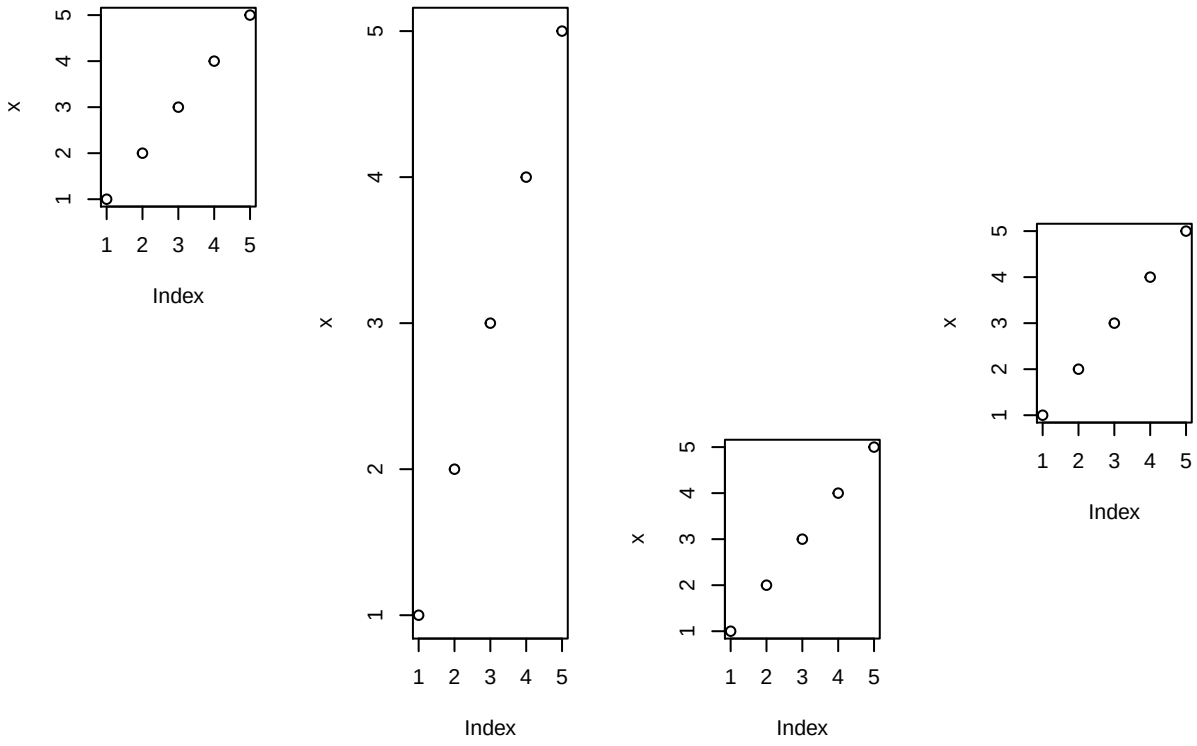


Você pode criar disposições à maneira que quiser. Vamos criar uma disposição extremamente bizarra para exemplificar.

```
> a <- c(1, 2, 0, 0)
> b <- c(1, 2, 0, 4)
> c <- c(0, 2, 3, 4)
> d <- c(0, 2, 3, 0)
>
> matriz2 <- rbind(a, b, c, d)
> matriz2
  [,1] [,2] [,3] [,4]
```

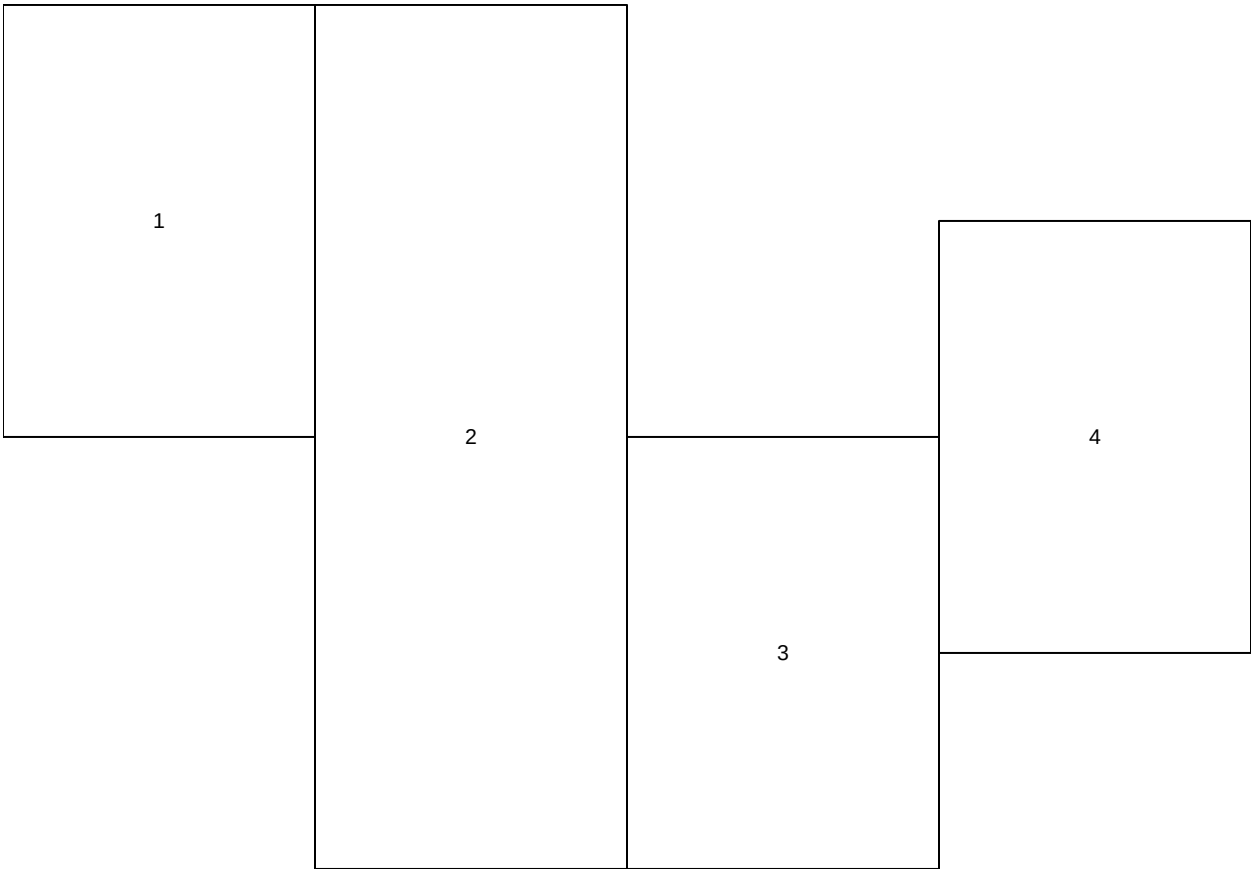
```
a    1    2    0    0
b    1    2    0    4
c    0    2    3    4
d    0    2    3    0
```

```
> layout(matriz2)
> plot(x)
> plot(x)
> plot(x)
> plot(x)
```



É possível verificar a disposição que você fez através da função `layout.show()`.

```
> layout(matriz2)
> layout.show(4)
```



Para resetar à disposição original, basta usar `layout(1)`.

2.3.2 Funções

As funções relacionadas à gráficos podem ser divididas em duas categorias: funções de alto e de baixo nível.

2.3.2.1 Funções de alto nível

Estas funções referem-se à criação de gráficos do zero. Funções como `plot()`, `pie()`, `barplot()`, entre outras, são funções de alto nível.

2.3.2.2 Funções de baixo nível

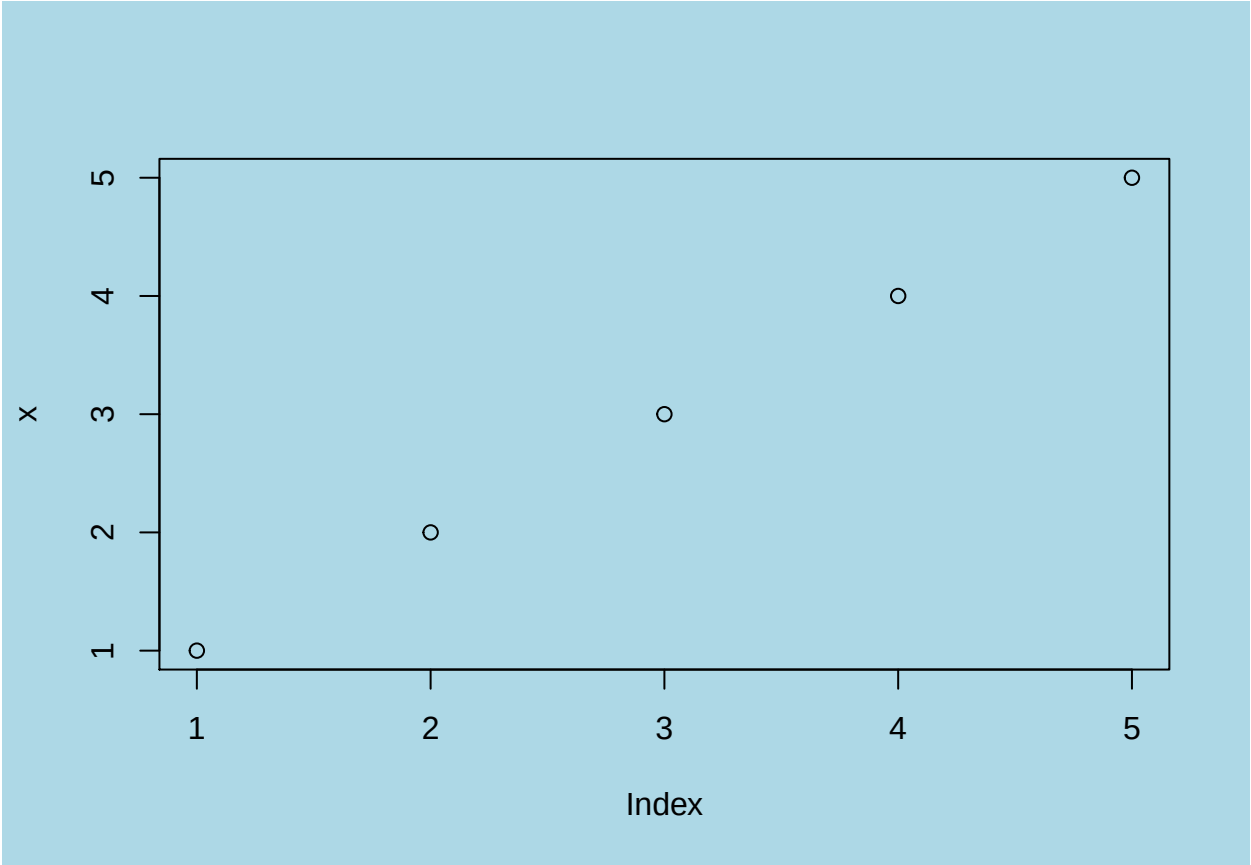
São as funções que adicionam elementos em um gráfico já criado. Alguns exemplos são `abline()`, `rect()` e `points()`. Alguns outros exemplos podem ser encontrados no cartão de referência.

2.3.3 Parâmetros

Os parâmetros são os argumentos que você usa para colocar as especificações nos gráficos. Uma lista completa pode ser visualizada através de `help(par)`.

Você pode definir os parâmetros de duas maneiras: definir os parâmetros globais previamente, ou definir nos argumentos de um gráfico. Para definir previamente, deve-se usar a função `par()` com os argumentos que você quer. Por exemplo, suponha que você queira transformar o fundo em azul claro:

```
> par(bg = "lightblue")
> plot(x)
```

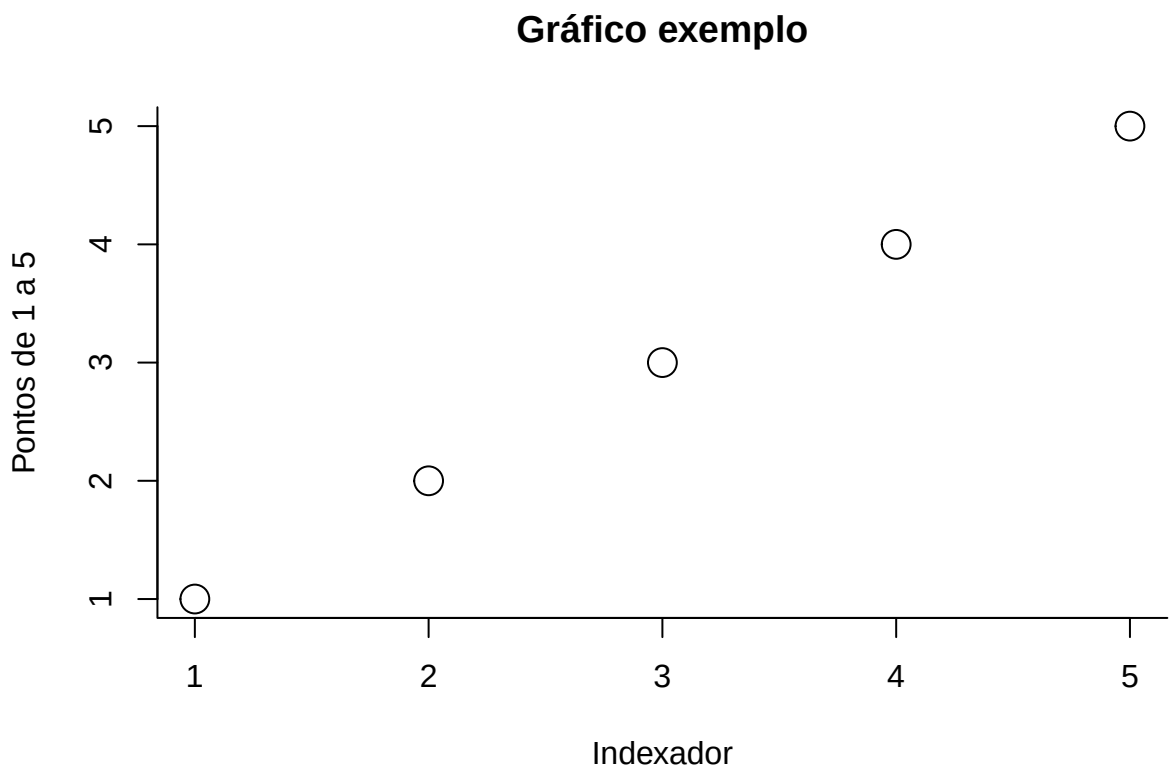


A partir de agora, todas gráficos que você criar terão fundo azul claro. Para resetar para os parâmetros originais, basta usar `dev.off()`

```
> dev.off()
null device
          1
```

A outra maneira comentada é utilizar os parâmetros na hora de criar a função.

```
> plot(x, main = "Gráfico exemplo", xlab = "Indexador", ylab = "Pontos de 1 a 5",
+      bty = "l", cex = 2)
```




```
> # Estes parâmetros servem, respectivamente, para criar o título do gráfico,
> # para criar o texto do eixo x, para criar o texto do eixo y, para definir o
> # formato da caixa e para definir o tamanho dos pontos
```

Uma lista com os parâmetros mais comuns está no cartão de referência.

3 Usando o R na prática

3.1 Importar dados reais

3.1.1 Diretório

É possível importar dados para o R. Antes de vermos como se faz isso, é preciso falar sobre o local do diretório. O local do diretório é o destino que o R está trabalhando neste exato momento. Para verificar qual é este destino, basta usar a função `getwd()`

```
> getwd()
[1] "C:/Users/Marcelo/Desktop/Aprender_R/Apostilas"
```

Como você pode ver, este é o diretório que o meu R está trabalhando neste momento. No entanto, os dados estão em outra pasta, não na pasta *Apostilas*. Para mudar o diretório, basta utilizar a função `setwd()`

```
> setwd("../Dados")
>
> # O comando "../Dados" significa que quero que o que o destino volte uma
> # pasta para trás e, após isso, entre na pasta Dados
>
> getwd()
[1] "C:/Users/Marcelo/Desktop/Aprender_R/Dados"
```

3.1.2 Lendo em .csv

Agora que o destino está corretamente localizado, posso importar os dados. Os dados chamam-se *bwght* e estão disponíveis no site da Cengage em formato *.xls*. Para esta análise, converti os dados para *.csv*. Os dados são sobre o peso de recém-nascidos.

Para importar os dados para o R, utilizamos a função `read.csv()`

```
> peso <- read.csv("bwght.csv", header = TRUE, sep = ";")
> dim(peso)
[1] 1388   14
```

Como você pode ver, os dados possuem muitas linhas. Para verificar apenas as linhas iniciais, utilize a função `head()`.

```
> head(peso)
  faminc cigtax cigprice bwght fatheduc motheduc parity male white cigs
1   13.5   16.5    122.3   109       12       12      1    1     1    0
2    7.5   16.5    122.3   133        6       12      2    1     0    0
3    0.5   16.5    122.3   129        .       12      2    0     0    0
4   15.5   16.5    122.3   126       12       12      2    1     0    0
5   27.5   16.5    122.3   134       14       12      2    1     1    0
6    7.5   16.5    122.3   118       12       14      6    1     0    0
    lbwght bwghtlbs packs    lfaminc
1 4.691348   6.8125     0  2.6026900
2 4.890349   8.3125     0  2.0149030
3 4.859812   8.0625     0 -0.6931472
4 4.836282   7.8750     0  2.7408400
5 4.897840   8.3750     0  3.3141860
6 4.770685   7.3750     0  2.0149030
```

3.1.3 Lendo em .xlsx

O R não possui funções que leiam a extensão *.xlsx* ou *.xls*. Para isso, é precisa instalar alguns pacotes. Um pacote que eu recomendo a instalação é o `XLConnect`. Outra opção possível é o `xlsx`. No entanto, estes pacotes precisam ter o Java instalado para que possam rodar. Na apostila de instalação existe um tutorial de como instalar o Java para poder utilizar estes pacotes.

3.2 Análise descritiva

Agora vamos fazer algumas micro-análises utilizando estes dados que importamos.

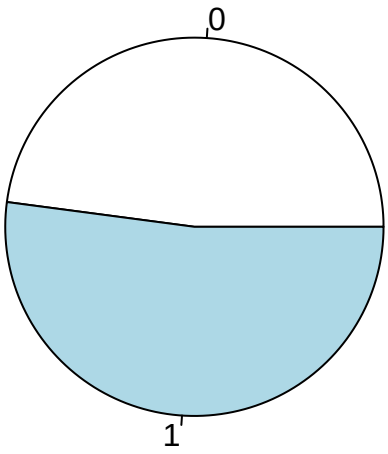
```
> # Verificar um resumo de quantos homens existem na amostra
> # Observação: o valor 0 corresponde a mulheres e o 1 a homens
>
> table(peso$male)
```

```
  0   1
665 723
```

```
> # Verificar a proporção de homens na amostra
>
> table(peso$male)/length(peso$male)
```

```
        0         1
0.4791066 0.5208934
```

```
> # Fazer um gráfico de setores para o resultado acima
>
> pie(table(peso$male)/length(peso$male))
```




```
> # Ver a média da renda familiar para todas pessoas na amostra
>
> mean(peso$faminc)
```

```
[1] 29.02666
```

```
> # Ver um resumo da renda familiar
>
> summary(peso$faminc)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.50   14.50   27.50   29.03   37.50   65.00
```

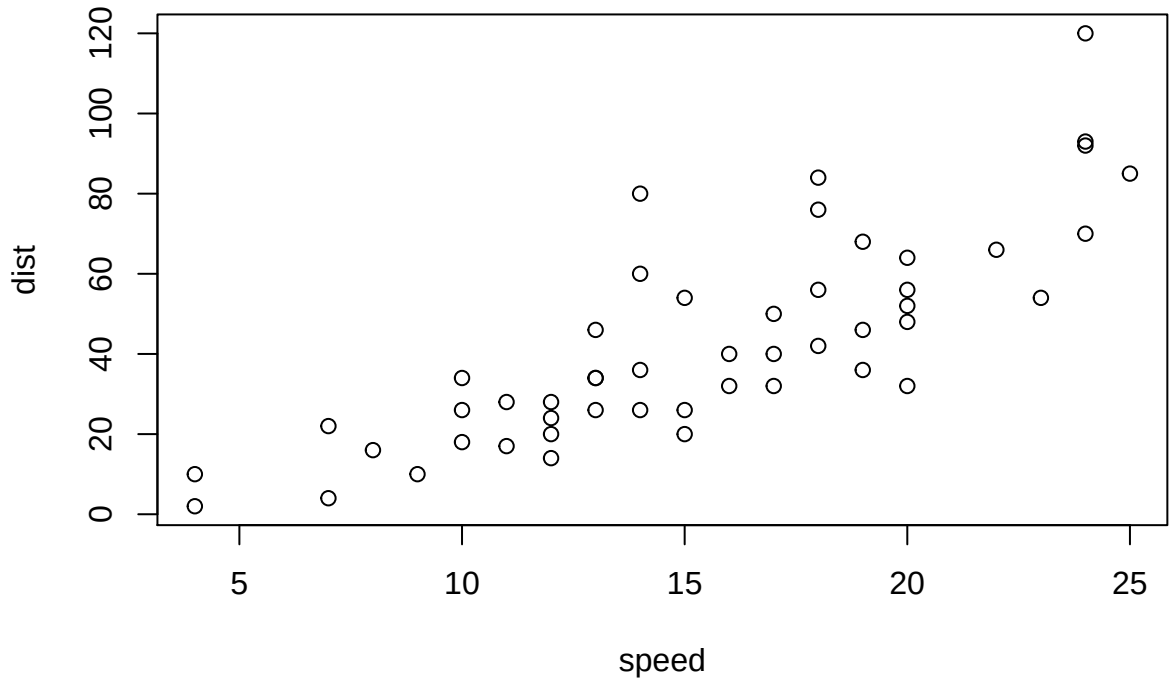
```
> # Ver a correlação entre renda familiar e peso no nascimento
>
> cor(peso$faminc, peso$bwght)
```

```
[1] 0.1089368
```

3.3 Regressão

Para esta parte de regressão, trabalharemos com uma base de dados embutida no R chamada *cars*.

```
> cars
   speed dist
1      4    2
2      4   10
3      7    4
4      7   22
5      8   16
6      9   10
7     10   18
8     10   26
9     10   34
10    11   17
11    11   28
12    12   14
13    12   20
14    12   24
15    12   28
16    13   26
17    13   34
18    13   34
19    13   46
20    14   26
21    14   36
22    14   60
23    14   80
24    15   20
25    15   26
26    15   54
27    16   32
28    16   40
29    17   32
30    17   40
31    17   50
32    18   42
33    18   56
34    18   76
35    18   84
36    19   36
37    19   46
38    19   68
39    20   32
40    20   48
41    20   52
42    20   56
43    20   64
44    22   66
45    23   54
46    24   70
47    24   92
48    24   93
49    24  120
50    25   85
> plot(cars)
```



A função da regressão é `lm()`. O primeiro argumento é a variável resposta (Y) e o segundo é a variável regressora (X). Vamos fazer a regressão de uma coluna pela outra.

```
> lm(cars$dist ~ cars$speed)
```

```
Call:
lm(formula = cars$dist ~ cars$speed)

Coefficients:
(Intercept)   cars$speed
    -17.579        3.932
```

Para ver mais detalhes da regressão, utilizamos `summary()`.

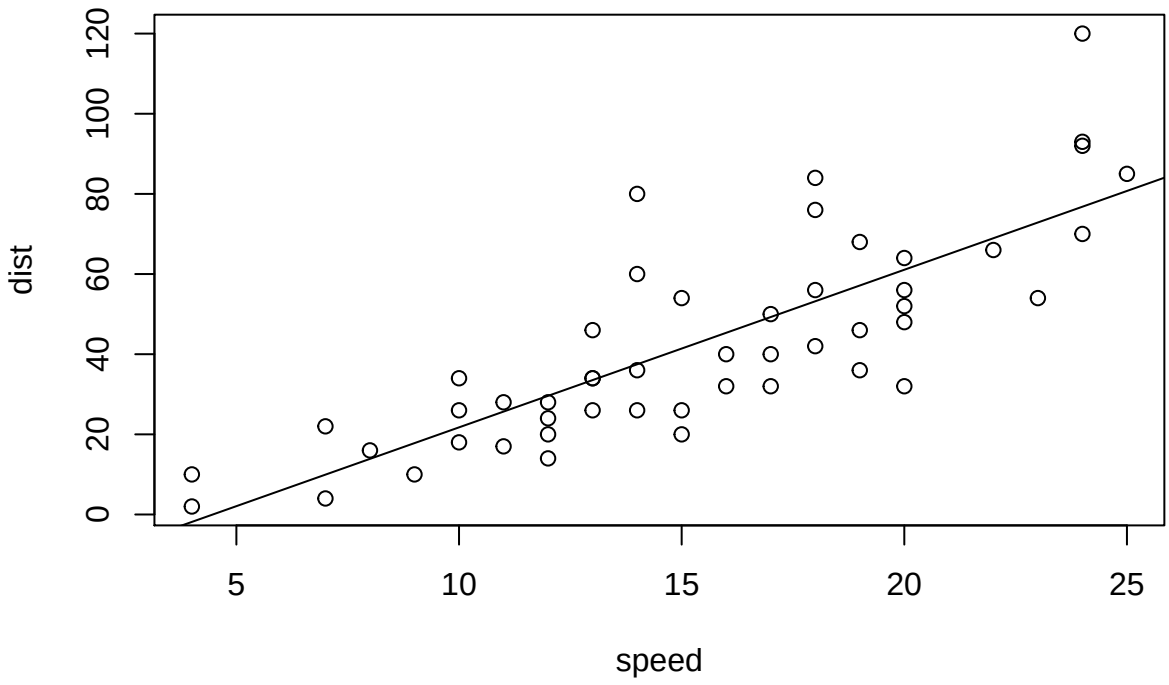
```
> summary(lm(cars$dist ~ cars$speed))
```

```
Call:
lm(formula = cars$dist ~ cars$speed)

Residuals:
    Min      1Q  Median      3Q     Max
-29.069  -9.525  -2.272   9.215  43.201
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -17.5791     6.7584  -2.601   0.0123 *
cars$speed    3.9324     0.4155   9.464 1.49e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 15.38 on 48 degrees of freedom
Multiple R-squared:  0.6511,    Adjusted R-squared:  0.6438
F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Para desenhar a reta de regressão no gráfico, podemos utilizar a função de baixo nível `abline()`.

```
> plot(cars)
> abline(lm(cars$dist ~ cars$speed))
```



3.4 Gerando PDFs

Para gerar um PDF através do R, é necessário ter o MiKTeX instalado. O passo-a-passo para a instalação está na apostila de instalações.

Para criar um PDF, basta clicar no canto superior esquerdo, no ícone embaixo de File. Clique ali e em seguida em R Markdown. Selecione Document, bote o título e o nome do autor e selecione a opção PDF.

Ao fazer isso, será aberto um arquivo *.rmd*. Agora, você pode começar a escrever em RMarkdown para gerar seu PDF. Toda a sintaxe do Rmarkdown está resumida no Reference Guide do RMarkdown. O link está no

capítulo Referências. Além da linguagem plana de markdown, existem os chunks de códigos que servem para escrever códigos na própria linguagem do R. No Reference Guide também tem a explicação de como criar chunks.

Após você ter escrito seu PDF, basta clicar em Knit PDF e o R compilará tudo que foi escrito para um PDF.

Todos os códigos das apostilas criadas estão sendo disponibilizados para que seja possível verificar exemplos e se familizar com a estrutura do RMarkdown.

4 Referências

Using R for Introductory Econometrics - Florian Heiss (http://www.urfie.net/read/read.html)

Introdução ao Ambiente Estatístico R - Paulo Justiniano Ribeiro Junior (http://www.leg.ufpr.br/~paulojus/embrapa/Rembrapa/)

R For Begginers - Emamnuel Paradis (https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf)

Advanced R - Hadley Wickham (http://adv-r.had.co.nz)

R Markdown Reference Guide - Rstudio Team (https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf)