

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL - UFRGS  
CAMPUS LITORAL NORTE  
CENTRO DE ESTUDOS COSTEIROS, LIMNOLÓGICOS E MARINHOS - CECLIMAR

## **Introdução ao software estatístico R**



Prof. Matias do Nascimento Ritter (CECLIMAR/UFRGS)  
Prof. Ng Haig They (CECLIMAR/UFRGS)  
Prof. Enéas Konzen (CECLIMAR/UFRGS)

Imbé, RS  
Julho de 2019

Centro de Estudos Costeiros, Limnológicos e Marinhos (Ceclimar)  
Av. Tramandaí, 976 Centro Imbé, RS, CEP 95625-000



O conteúdo deste documento pode ser redistribuído desde que com a devida citação, de acordo com os termos da GLP.

This document is copyright (C) 2019 Matias Ritter, Ng Haig They and Enéas Konzen  
You can redistribute it and/or modify it under the terms of version 4.0 the GNU General Public License as published by the Free Software Foundation.

Version: 2.0

Last updated: July 29th, 2019

To view a copy of the license, go to:

<http://www.fsf.org/copyleft/gpl.html>

To receive a copy of the GNU General Public License write the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Os autores declaram que não existe conflito de interesses no que tange à esta publicação.

Os conjuntos de dados utilizados neste documento estão disponíveis para download no repositório

**GITHUB:**

<https://github.com/haigthey/Curso-Introducao-ao-software-estatistico-R>

Dúvidas, sugestões e críticas:

Prof. Matias do Nascimento Ritter: e-mail: [matias.ritter@ufrgs.br](mailto:matias.ritter@ufrgs.br); (51) 3308-1276

Prof. Ng Haig They: e-mail: [haigthey@ufrgs.br](mailto:haigthey@ufrgs.br); (51) 3308-1273

Prof. Enéas Konzen: e-mail: [erkonzen@gmail.com](mailto:erkonzen@gmail.com); (51) 3308-1273

O R é um software livre, porém construído a partir do trabalho de muitas pessoas. Ao utilizá-lo, sempre o cite:

*R Core Team (2019) R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.*

E, lembre-se sempre de citar os pacotes utilizados também!!

## SUMÁRIO

<b>Apresentação do R e RStudio</b>	04
<b>Download e atualização do programa</b>	05
<b>Layout do R e RStudio: funções e operadores</b>	05
<b>Pacotes</b>	07
<b>Scripts</b>	07
<b>Exemplos: demos do R</b>	08
<b>Citação do R e seus pacotes</b>	08
<b>O R como calculadora</b>	10
<b>Menus de ajuda</b>	10
<b>Criação de objetos no R</b>	11
<b>Remoção e listagem de objetos no R</b>	15
<b>Importação de dados</b>	15
<b>Manipulação de vetores, data frames e matrizes</b>	18
<b>Gerando sequências, números aleatórios e repetições</b>	27
<b>Criação de gráficos no R</b>	31
<b>Gráficos em painéis</b>	36
<b>Alterando a aparência de gráficos</b>	37
<b>Exportação de gráficos</b>	50
<b>Estatística univariada: comparando duas ou mais médias, pressupostos, testes de associação entre variáveis</b>	52
<b>Comparação entre duas médias</b>	52
<b>Comparação entre &gt; 2 médias</b>	53
<b>Testes de associação entre variáveis</b>	59
<b>Estatística Multivariada: análise de cluster e métodos ordenação</b>	67
<b>Análise de agrupamento (cluster)</b>	67
<b>Análise de Componentes Principais</b>	70
<b>Escalonamento multidimensional não métrico</b>	74
<b>Análise de experimentos biológicos constituídos segundo delineamentos com níveis de complexidade crescentes</b>	80
<b>Análise de experimentos em DIC e DBC: a diferença do fator bloco</b>	80
<b>Análise de um experimento em DBC com mais de um fator</b>	83
<b>Testes de adequação ao modelo estatístico para ANOVA</b>	85
<b>Realização da ANOVA</b>	89
<b>Testes de comparação de médias</b>	89
<b>Análise de experimentos bifatoriais no pacote ExpDes 88</b>	91
<b>R aplicado à bioinformática: fundamentos sobre alinhamentos de sequências e reconstrução filogenética</b>	96
<b>Introdução a séries temporais</b>	103
<b>Criando sua própria função</b>	105
<b>Considerações finais</b>	108
<b>Referências</b>	109

## Apresentação do R e RStudio<sup>1</sup>

O R é ao mesmo tempo um tipo de linguagem e um software computacional e gráfico. Duas grandes vantagens tornam o programa bastante popular: a primeira é que ele é um software de código aberto, livre, grátis, do tipo GNU General Public License (GNU's Not Unix), ou seja, os usuários podem executar o programa como quiserem, podem estudar como funciona e adaptá-lo às necessidades (livre acesso aos códigos-fonte). Além disto, podem redistribuí-lo livremente e realizar melhorias. A segunda é que ele é altamente extensível, ou seja, pode ser utilizado para realizar qualquer atividade computacional, desde que compatível com suas capacidades. Isto é feito através da criação de funções próprias e dos pacotes, conjuntos de códigos/comandos relacionados a determinados temas. A expressão “ambiente R” é muitas vezes utilizada para se referir ao espaço computacional, ou seja, à sua característica de ser um sistema coerente e totalmente planejado. A página oficial do Projeto R pode ser acessada através do endereço: <https://www.r-project.org/>.

O R é bastante difundido como um software estatístico, mas realiza várias outras tarefas, como por exemplo análises de bioinformática, incluindo alinhamento de sequências e comparação com bases de dados, modelagem, produção de mapas e muitos outros. Caso não encontre algum pacote que atenda uma demanda específica (o que é bastante improvável!), qualquer pessoa pode desenvolver uma função ou pacote para isto. De um modo geral o R é uma ferramenta excelente para armazenar e manipular dados, realizar cálculos, realizar testes estatísticos, análises exploratórias e produzir gráficos.

Durante este curso utilizaremos o R Studio (<https://www.rstudio.com/products/rstudio/>), que é um ambiente integrado de desenvolvimento para o R, um programa que tem como base o R, mas que apresenta uma interface mais amigável e mais funcional, tais com janelas de plotagem de gráficos, de histórico, de gerenciamento do workspace e de editor de texto com realce de sintaxe que permite execução direta de código (script). E, assim como o R, também um software de código aberto. O RStudio apresenta uma versão aberta grátis e várias versões pagas com funcionalidades extras.

Este curso tem a modesta intenção de ser uma introdução e facilitar o primeiro contato com a ferramenta, servindo de pontapé sem esgotar o assunto de maneira alguma. Ao longo da apostila diversos apontamentos serão colocados sempre após o símbolo de cerquilha ( # ) ou hashtag; toda informação a partir deste símbolo não deverá ser digitada no R, mas caso seja não há problema, pois o R reconhece o símbolo e ignora o que está após este. O R é essencialmente um programa para autodidatas, isto é, somente se aprende utilizando o programa e descobrindo formas de resolver problemas à medida que eles forem surgindo. A comunidade que utiliza o R é muito grande e muito ativa e posta muitas soluções para problemas em fóruns, blogs, mídias sociais e sites. Portanto, as ferramentas de busca na internet serão suas

---

<sup>1</sup> <https://www.r-project.org/about.html>

melhores professoras nessa árdua, mas recompensadora viagem. Vamos começar?

## Download e atualização do programa

O download do R é feito através de um dos espelhos (“mirrors”) do CRAN (Comprehensive R Archive Network). Muitos países possuem estes espelhos, inclusive o Brasil, que inclusive possui vários deles.

Para realizar o download, acesse a página <https://cran.r-project.org/mirrors.html> e selecione um dos espelhos. Aparecerá uma página em que é possível selecionar um sistema compatível com os três sistemas operacionais: Windows, Linux e Mac (OS). Nesta página também é possível encontrar informações sobre as novas versões, sobre conserto de bugs, códigos-fonte etc. Aqui também é possível encontrar um link para o download de pacotes (“Contributed extension packages”) em formato comprimido, que pode posteriormente ser instalado diretamente sem a necessidade de internet.

Uma das poucas desvantagens do R é que ele não possui possibilidade de atualização da versão instalada, ou seja, a cada nova versão liberada pela equipe desenvolvedora do R, esta deve ser instalada do início. A versão antiga pode ser mantida ou removida, sem que haja nenhum conflito, podendo inclusive ser utilizada ao mesmo tempo. Importante: todos os pacotes da versão antiga devem ser instalados na versão nova, caso sejam ainda necessários.

O RStudio pode ser baixado a partir da página RStudio Desktop Open Source License - FREE: <https://www.rstudio.com/products/rstudio/download/>.

## Layout do R e RStudio: funções e operadores

Uma das características do R que muitas vezes intimida novos usuários é a interface gráfica, que é feita através de linhas de comando no prompt (janela em que é possível digitar comandos). No entanto, uma das grandes vantagens do uso deste recurso é que o usuário tem grande controle sobre o que está sendo feito, pois à exceção de argumentos padrão embutidos nas funções, todos os parâmetros do comando devem ser especificados pelo usuário.

Quando o prompt está pronto para receber uma novo comando, aparece um sinal de maior que ( > ) bem à esquerda na linha de comando. Ao longo da apostila todas as linhas de comando serão indicadas com este sinal no início. No entanto, ao copiar os comandos para o RStudio, não copie junto este sinal, pois o sinal ficará duplicado e o R acusará um erro:

>

Um comando é composto de uma ou mais funções com seus respectivos

argumentos. Uma função é um código que determina um algoritmo computacional, que pode ter um ou mais parâmetros a serem determinados ou modificados através dos argumentos, que são colocados entre parênteses logo após a função. Alguns argumentos são obrigatórios enquanto outros são facultativos; alguns obrigatórios possuem um modo padrão (default).

Exemplo: função log

```
> log(10)
```

# calcula o logaritmo natural do número 10. O número entre parênteses é o único argumento necessário para esta função.

```
[1] 2.302585
```

Esta função pode ser modificada com a inclusão de um segundo argumento para indicar a base do logaritmo a ser calculado. Digamos, por exemplo, que se queira a base 10. Isto é indicado colocando-se uma vírgula e o valor da base que se deseja logo após o número:

```
> log(10,10)
[1] 1
```

```
> log(100,10)
[1] 2
```

Se por acaso algum sinal de ( + ) aparecer logo após a inserção de algum comando, significa que algum argumento ou informação ficou faltando (frequentemente parênteses) e o R está esperando que este seja informado. Caso queira abortar um comando iniciado, simplesmente tecle **ESC** e depois **ENTER**:

```
> log(10
+
+ )
[1] 2.302585
```

```
> log(10
+
# tecle ESC
+ >
# tecle ENTER
>
```

O R possui ainda operadores matemáticos básicos que são bastante similares àqueles usados no Programa Excel® ou similares: ( + ) para somatório, ( - ) para subtração, ( / ) para divisão e ( \* ) para multiplicação. Eles são utilizados frequentemente dentro das funções.

## Pacotes

A versão básica do R já é automaticamente instalada com 8 ou mais pacotes: *base*, *compiler*, *datasets*, *graphics*, *grDevices*, *grid*, *methods*, *parallel*, *splines*, *stats*, *stats4*, *tcltk*, *tools* e *utils*. Isto pode ser acessado através do comando:

```
> subset(as.data.frame(installed.packages()), Priority
%in%("base"), select=c(Package, Priority))
```

	Package	Priority
base	base	base
compiler	compiler	base
datasets	datasets	base
graphics	graphics	base
grDevices	grDevices	base
grid	grid	base
methods	methods	base
parallel	parallel	base
splines	splines	base
stats	stats	base
stats4	stats4	base
tcltk	tcltk	base
tools	tools	base
utils	utils	base

No RStudio os pacotes são instalados através do botão drop down na parte superior da página: Tools → Install packages → [*Nome do pacote*]. Também pode ser feito diretamente a partir de uma linha de comando:

```
> install.packages("vegan")
# exemplo com o pacote Vegan
```

Uma vez instalado, o pacote precisa ser carregado, que é feito através do comando `library`:

```
> library(vegan)
# exemplo com o pacote Vegan
```

Toda vez que o R for aberto após ter tido uma sessão encerrada, os pacotes necessitam ser carregados novamente. A instalação de cada pacote é feita apenas uma vez para determinada versão do R, mas o carregamento deve ser feito após cada abertura de sessão. Não é necessário carregar todos os pacotes, apenas o que for utilizado naquela sessão.

## Scripts

Os scripts são arquivos do R com conjuntos de comandos que podem executados diretamente no R ou RStudio através de teclas de atalho. A sua vantagem

consiste no fato de que o programa executa cada linha de comando e já pula para a próxima, fazendo com que seja muito rápido executar várias linhas de comando em sequência apenas com as teclas de atalho do teclado. Sem o script é necessário copiar e colar no prompt de comando todas as linhas uma a uma e teclar ENTER entre todas elas. Muitas alterações em gráficos, por exemplo, exigem que todas as linhas de comando sejam executadas novamente para incluir a nova alteração, então os scripts agilizam bastante nessa hora.

Para executar scripts no R basta que o cursor do teclado esteja posicionado em qualquer ponto da linha de comando a ser executada e que seja pressionado CTRL + R. No RStudio a única diferença é que deve ser substituído por CTRL + ENTER.

Um script pode ser criado tanto no R quanto no RStudio. No R através do botão drop down Arquivo → Novo Script; no RStudio através do botão drop down “File → New File → R Script”. Será gerado um arquivo com extensão .r que será salvo no diretório de trabalho e que pode ser aberto para visualização em qualquer editor de texto (bloco de notas ou Notepad++, por exemplo). Caso você já tenha um arquivo comum em bloco de notas com uma rotina do R e queira transformá-lo em script, basta copiar o conteúdo deste em um Novo Script do R e nomeá-lo conforme queira.

### Exemplos: demos do R

O R possui uma função chamada `demo` que apresenta exemplos de alguns pacotes. Os exemplos são em geral bastante elaborados e incluem várias funções combinadas, como por exemplo para produção de gráficos complexos e funções avançadas.

```
> demo( )
# aparecerá uma lista de demos disponíveis. Escolha uma.

> demo(graphics)
# exemplo de demo da função graphics do pacote graphics.
São mostradas várias janelas com gráficos e a lista de
comandos no prompt. Para mudar as janelas de gráficos,
tecle ENTER.
```

### Citação do R e seus pacotes

Para citar o R, utilize o seguinte comando:

```
> citation( )

# aparecerão informações sobre como o R deve ser citado
em publicações. Para encontrar a versão do R vá no botão
drop down superior HELP e selecione About RStudio (no
RStudio) ou Ajuda → Sobre (no R).
```



To cite R in publications use:

R Core Team (2018). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.

A BibTeX entry for LaTeX users is

```
@Manual{,  
  title = {R: A Language and Environment for Statistical  
Computing},  
  author = {{R Core Team}},  
  organization = {R Foundation for Statistical Computing},  
  address = {Vienna, Austria},  
  year = {2018},  
  url = {https://www.R-project.org/},  
}
```

We have invested a lot of time and effort in creating R, please cite it when using it for data analysis. See also `'citation("pkgname")'` for citing R packages.

Ao realizar uma citação e referência, coloque R e a versão. Exemplo:

Citação:

As análises estatísticas foram realizadas no programa R v. 3.5.1 (R Core Team, 2018).

Referência:

R Core Team (2018). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.

Para citar os pacotes utilize também a função `citation`, colocando entre parênteses o nome do pacote entre aspas:

```
> citation("vegan") # exemplo com o pacote Vegan.
```

Jari Oksanen, F. Guillaume Blanchet, Michael Friendly,  
Roeland Kindt, Pierre Legendre, Dan McGlinn, Peter R.  
Minchin, R. B. O'Hara, Gavin L. Simpson, Peter Solymos, M.

Henry H. Stevens, Eduard Szoecs and Helene Wagner (2018).  
vegan: Community Ecology Package. R package version 2.5-1.  
<https://CRAN.R-project.org/package=vegan>

## O R como calculadora

O R realiza funções aritméticas básicas com os operadores já mencionados.  
Exemplos:

```
> 2 + 2
[1] 4

> 2 - 2
[1] 0

> 2 * 2
[1] 4

> 2 / 2
[1] 1

> log(64,8)
[1] 2

> log(9+1,10)
[1] 1
# calcula o log na base 10 do número 9 + 1

> sqrt(100)
[1] 10
#sqrt = square root (raiz quadrada)

> (9+3)/4
[1] 3

> 10^2
[1] 100
# potências são indicadas após o sinal de circunflexo

> 100^-2
[1] 1e-04
# e-04 é uma notação para 10-4
```

## Menus de ajuda

O RStudio apresenta uma facilidade que é a de abrir os menus de ajuda dentro de uma janela do próprio programa. Um menu de ajuda apresenta a função e o que ela faz (*description*), seus principais argumentos (*arguments*), observações sobre a função (*details*), referências e ainda exemplos de aplicação. É uma forma bastante útil

para descobrir como funciona uma função. Para se obter um menu de ajuda de uma função, digite ( ? ) seguido do nome da função:

```
> ?sum
#exemplo com a função soma
```

Ou, alternativamente,

```
> help(sum)
```

Observe o menu de ajuda da função soma e reproduza os exemplos.

## Criação de objetos no R

O R opera essencialmente com os chamados objetos. Para criar um objeto no R, deve-se determinar um nome para este, seguido de um símbolo de flecha ou igual ( <- ou = ) apontando para objeto e indicando que todo o conteúdo à direita da flecha deve ser inserido dentro dele. No exemplo abaixo iremos criar um objeto chamado "a" e inserir o número 5 dentro dele:

```
> a <- 5
> a
# digitando "a" e teclando ENTER, o R irá mostrar o
conteúdo do objeto "a".
[1] 5
```

Os tipos de dados que são inseridos nos objetos são divididos nas seguintes classes:

- i) character (categórico ou qualitativo)
- ii) numeric (numérico)
- iii) integer (inteiro)
- iv) logical (true/false) (lógico, verdadeiro/falso)
- v) complex (complexos, relacionados ao número imaginário "i") - não serão abordados aqui

Exemplos:

```
#character
> a <- "eu amo o R"
> a
[1] "eu amo o R"

> CECLIMAR <- "Imbé"
> CECLIMAR
[1] "Imbé"
```

#dados categóricos sempre devem estar entre aspas para não serem confundidos com objetos. As aspas significam "leia como está".

#### #numeric

```
> A <- 67.7
> A
[1] 67.7
```

```
> x <- 1
> y <- 2.5
```

#### #integer

```
> z <- as.integer(5.4)
> z
[1] 5
```

```
> z <- as.integer(9.8)
> z
[1] 9
```

# a função `as.integer` retorna o valor inteiro do número decimal. Note que ele não é uma função de arredondamento! Para isto use `round(9.87, 1)`, por exemplo. O primeiro argumento de `round` é o número que se quer arredondar e o segundo o número de casas decimais desejado.

#### # logical

```
> m <- x>y
> n <- x<y
> m
[1] FALSE
```

# o resultado indica que o conteúdo de `m` (`x>y`) é falso.

Para consultar a classe de um objeto, utilize a função `class()`:

```
> class(m)
[1] "logical"
```

```
> class(A)
[1] "numeric"
```

Algumas coisas importantes sobre nomes de objetos:

- O R é sensível a maiúscula e minúsculas (*case sensitive*). Então um objeto "a" é diferente de um objeto "A".
- Se você nomear um segundo objeto com um nome já utilizado por outro, o R irá sobrescrever o conteúdo do segundo objeto sobre o primeiro.
- Nunca deixe espaços entre nomes de objetos no R! Se você quiser nomear um

objeto de “objeto do R”, coloque algum símbolo ( `_` ) / ( `.` ) entre os nomes. Ex.: `objeto_do_R` ou `objeto.do.R`. Já para dados categóricos não há problema em deixar espaços.

```
> objeto_do_R <- "R"
> objeto_do_R
[1] R
```

- não é possível usar um número isolado para nomear objetos. Números acompanhados de letras, no entanto, podem ser utilizados:

```
> Omega3 <- 50
> Omega3
[1] 50
```

- Evite usar acentos, cedilhas, apóstrofes, aspas e outros caracteres de pontuação em nomes de objetos no R. Eles costumam causar muita confusão e são frequentemente causadores de mensagens de erros.

Os objetos, por sua vez, podem ser divididos nas seguintes classes:

- i) scalar (escalar)
- ii) vector (vetor)
- iii) array (arranjo)
- iv) list (listas)
- v) data frame (quadro de dados)
- vi) matrix (matriz)

i) **escalar**: é formado de um único valor numérico, inteiro ou decimal. Já foi utilizado em vários exemplos anteriores.

Exemplo no R:

```
> z <- 4
> z
[1] 4
```

ii) **vetor**: é um conjunto de valores numéricos ou de caracteres (letras, palavras). Possui uma única dimensão e só pode conter dados de uma única classe.

Exemplo no R:

```
> a <- c(1:100)
# vetor "a", numérico com 100 elementos (números de 1 a 100).

> b <- c("Semestre 1", "Semestre 2", "Semestre 3",
```

"Semestre 4") # vetor de nomes com 4 elementos

iii) **array**: é um vetor multidimensional. É construído a partir da função *array*.

```
> vetor1 <- c(5,9,3)
> vetor2 <- c(10,11,12,13,14,15)
> arranjo <- array(c(vetor1,vetor2),dim = c(3,3))
```

# a função *c* significa "concatenar" e irá produzir um vetor combinando os dois vetores entre parênteses. O argumento *dim* significa "dimensões" e especifica o número de linhas e colunas *dim=c(linhas, colunas)*.

```
> arranjo
      [,1] [,2] [,3]
[1,]     5    10    13
[2,]     9    11    14
[3,]     3    12    15
```

iv) **listas**: conjunto de vetores, dataframes ou de matrizes. Não precisam ter o mesmo comprimento e nem ser da mesma classe. É a forma que a maioria das funções retorna os resultados.

Exemplo no R:

```
> n <- c(5, 6, 8)
> s <- c("ab", "bc", "cd", "de", "ef")
> b <- c(FALSE, TRUE, TRUE, FALSE, FALSE)
> x <- list(n, s, b, 3) # x contém cópias de n, s, b
```

v) **dataframe**: similar à matriz, porém é mais genérico, aceitando vetores numéricos e de caracteres em um mesmo conjunto. É o mais comumente utilizado nas ciências naturais, pois normalmente temos observações numéricas associadas a variáveis categóricas.

Exemplo no R:

```
> n <- 1000
> a <- rnorm(n, 9, 1) # Cria um conjunto de dados com
1000 elementos, com média 9 e desvio-padrão igual a 1.

> b <- c(rep(1, 0.4*n), rep(2, 0.6*n))

> dados <- data.frame(group=b, grade1=sample(b), var1=a,
var2=a+rnorm(100, 5, 2), var3=sqrt(sample(1:n)))

> head(dados)
# a função head mostra por default as 6 primeiras linhas.
Para mostrar mais linhas, especifique o argumento n:
```

```
> head (dados, n=6L) # default
> head (dados, n=8L) # especifica as 8 primeiras linhas
```

Para ver as últimas linhas, use a função `tail()`, que funciona de maneira similar.

vi) **matriz**: é um conjunto de vetores em linhas e/ou colunas. Todos os vetores devem ser da mesma classe (numérico ou de caracteres, por exemplo).

Exemplo no R:

```
> matriz1 <- cbind(seq(1, 11, 2), rep(4, 6), 1:6)
> matriz1 # veja como ficou a matriz
# a função cbind (column bind) "cola" as colunas lado a
lado. Já a função rbind (row bind) cola linhas uma abaixo
da outra.
```

## Remoção e listagem de objetos no R

Para remover um objeto, usa a função `rm()` (remove), seguido do nome do objeto:

```
> rm(a)
# remove o objeto "a".
```

Observação: algumas vezes ao inserir data frames diferentes, mas que contenham nomes de variáveis iguais, o R pode não saber a que variável se está referindo, caso seja usada a função `attach()` (ver logo abaixo em Importação de dados). Neste caso pode ser necessário remover objetos ou até data frames inteiros para que não haja confusão no uso de variáveis.

Para listar todos os objetos que foram criados no R, use a função `ls()` (list), seguido de parênteses vazios:

```
> ls()
```

## Importação de dados

Existem inúmeros modos de se importar dados para o R. Aqui serão apresentados aqueles métodos mais comuns, provavelmente os quais vocês irão se deparar usando o R daqui para frente.

No R é possível definir um diretório de trabalho através da função `"setwd()"`. Este diretório, na prática, significa que você está dizendo ao R tanto o local onde estarão seus arquivos para importação, quanto a pasta em qual ele irá salvar seus arquivos exportados.

Com esta lógica, por exemplo, você pode salvar seus conjuntos de dados no seu diretório de trabalho, no formato de planilha de dados “.csv”, por exemplo; ou seja, separado por vírgulas. Neste caso, é muito simples importar um conjunto de dados para o R, utilizando os seguintes comandos:

```
> setwd("C:/Users/Fulano/R files")  
# Caminho da pasta com os arquivos  
  
> meusdados <- read.csv("meusdados.csv", sep=";")  
> head(meusdados)
```

\*Se o seu computador está configurado para o sistema decimal estadunidense, é provável que o comando possa ser simplificado apenas para:

```
> meusdados <- read.csv("meusdados.csv")
```

De modo similar, ainda poderíamos utilizar o seguinte comando como alternativa:

```
> read.table("meusdados.csv", header=TRUE, sep=",")  
# ou sep=";"
```

Entretanto, existem dezenas de outras formas de importação dos dados para o R. Um outro método mais simples, é utilizar o comando “read.table”. Neste caso, você seleciona, no arquivo do Excel®, os dados que você gostaria de inserir no R e os copia (CTRL + C), deixando estes dados na área de transferência. Neste caso, você pode adicioná-los ao R com os seguintes comandos:

```
> meusdados <- read.table("clipboard", header=FALSE)  
  
# O argumento header = FALSE indica que os dados não  
# contêm cabeçalho. Caso contrário, useu header=TRUE. Se a  
# primeira coluna contiver os nomes das linhas, inclua o  
# argumento row.names=1.  
  
# modo mais genérico; dados armazenados como “dataframe”,  
# o que permite mais operacionalidade. Pode ser usado para  
# importar dados diretamente de uma tabela do Excel®.
```

Arquivos de texto (.txt) também podem ser utilizados e facilmente inseridos no R, como demonstrado abaixo:

```
> meusdados <- read.table("meusdados.txt", header = T,  
sep = "\t")  
  
# Note que o argumento “sep=\t” pode variar conforme o  
# modo que os dados foram inseridos no arquivo de texto.
```



"t" indica que os dados estão separados por tabulação.

Para importar vetores, use a função `scan()`. Neste caso, o vetor deve ser copiado para a área de transferência. Primeiro se digita a função `scan` com parênteses vazios e se dá ENTER. Em seguida, os dados devem ser copiados para a área de transferência e o comando CTRL + V dado no R. Ele irá listar todos os elementos do vetor. Para indicar que a colagem terminou, deve-se teclar ENTER.

Digite o seguinte vetor no Excel (pode ser na linha ou coluna):

4  
5  
6  
7  
8

```
> r <- scan()
# tecla CTRL + C nos dados do vetor no Excel
1:
#aparecerá o número 1 seguido de dois pontos indicando
que o R está esperando a colagem. Tecle CTRL + V

1: 4
2: 5
3: 6
4: 7
5: 8
6:

# o R indicará que foram lidos 5 itens e ele está
esperando para adicionar mais. Caso não haja mais dados a
serem colados, tecla ENTER.
Read 5 items

> r
[1] 4 5 6 7 8
# confere o resultado do objeto r.
```

Para importar vetores de caracteres, use o argumento `what="character"`:

```
Eu
amo
o
R

> r <- scan(what="character")
1: Eu
2: amo
```

```

3: o
4: R
5:
Read 4 items

> r
[1] "Eu" "amo" "o"  "R"

```

Alguns usuários utilizam ainda a função `attach()`, a qual torna as variáveis (colunas) acessíveis apenas digitando o nome delas na linha de comando. Entretanto, se estamos trabalhando com dois `dataframes` diferentes, mas que possuem nomes das variáveis iguais, este artifício pode causar problemas. Neste caso, pode ser necessário realizar o `detach` de uma delas ou em alguns casos até remover uma delas.

```

> attach(meusdados)
> detach(meusdados)

```

\* caso haja dados faltando no vetor ou matriz, coloque "NA" no lugar

```

4
5
6
NA
8

> n <- scan()
> n
[1] 4 5 6 NA 8

> is.na(n)
[1] FALSE FALSE FALSE TRUE FALSE
# função que retorna um vetor lógico indicando se há NAs
nos dados.
sum(n)

> sum(n, na.rm=TRUE) # operações com vetores ou outros
objetos contendo NAs precisam que seja especificado que
estes sejam ignorados.

```

## Manipulação de vetores, data frames e matrizes

Uma vez que inserimos os dados no R, o próximo passo é a correta manipulação destes dados. Para exemplificarmos alguns comandos, vamos utilizar um conjunto de dados que já vem com o pacote básico do R. Utilize os seguintes comandos:

```

> data(trees)
# Esta função lista todos os conjuntos de dados

```

disponíveis. Este conjunto de dados, "trees", apresenta alguns dados de circunferência (*girth*), altura (*height*) e volume de algumas cerejeiras pretas nos EUA.

Se você utilizar o comando `head`, o R irá ilustrar as seis primeiras linhas dos dados:

```
> head(trees)

  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
```

Outros comandos que são valiosos para verificar se os dados foram corretamente importados, ou ainda para conhecermos as especificidades dos dados que estamos trabalhando, estão listados abaixo:

```
> str(trees) # Exibirá de forma resumida a estrutura
interna de um objeto R

'data.frame':  31 obs. of  3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1
11.2 ...
 $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
 $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2
22.6 19.9 ...

> dim(trees)# número de linhas e colunas
[1] 31  3

> summary(trees)# esta função mostra mínimo, máximo,
média, mediana, primeiro e terceiro quartis.
```

Girth	Height	Volume
Min. : 8.30	Min. : 63	Min. : 10.20
1st Qu.: 11.05	1st Qu.: 72	1st Qu.: 19.40
Median : 12.90	Median : 76	Median : 24.20
Mean : 13.25	Mean : 76	Mean : 30.17
3rd Qu.: 15.25	3rd Qu.: 80	3rd Qu.: 37.30
Max. : 20.60	Max. : 87	Max. : 77.00

O software R permite a manipulação de diversos tipos de objetos. Para muitos tipos, entretanto, o primeiro contato será devido a demandas específicas e pontuais.

## Vetores

Para substituir um ou mais valores em um vetor, use o seguinte comando:

```
> (r <- c(4, 5, 6, 7, 7, 8))  
[1] 4 5 6 7 7 8
```

```
> r[3]<- 10 # este comando indica que o 3º elemento do  
vetor deve ser substituído pelo dado à direita da seta.
```

```
> r  
[1] 4 5 10 7 8
```

```
> r[c(3,4)] <- c(10,20) # para substituir mais de um valor  
>r  
[1] 4 5 10 20 8
```

Para ordenar de forma crescente:

```
> sort(r)  
[1] 4 5 8 10 20
```

Para ordenar de forma decrescente, basta adicionar o argumento `decreasing=TRUE`:

```
> sort(r,decreasing=TRUE)  
[1] 20 10 8 5 4
```

Para verificar a posição (rank) de cada valor:

```
> order(r)  
[1] 1 2 5 3 4
```

## Data frames e Matrizes

Para exercitar as manipulações mais básicas em um conjunto de dados, vamos novamente utilizar o conjunto de dados chamado "trees":

```
> data(trees)
```

```
# Este conjunto de dados, "trees", apresenta alguns dados  
de circunferência (girth), altura (height) e volume de  
algumas cerejeiras pretas nos EUA.
```

Agora vamos selecionar (extrair) apenas algumas partes do nosso conjunto de dados `trees` usando `[ ]` colchetes. O uso de colchetes funciona com esta lógica: `[linhas, colunas]`, onde está escrito linhas você especifica as linhas desejadas, na maioria dos casos cada linha indica uma unidade amostral. Onde está escrito colunas,

você pode especificar as colunas (atributos) que deseja selecionar. Veja abaixo:

```
> trees[, 1] # extrai a primeira coluna e todas as linhas
[1] 8.3  8.6  8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3
11.4 11.4 11.7 12.0 12.9 12.9 13.3 13.7 13.8 14.0 14.2
14.5 16.0 16.3 17.3 17.5
[28] 17.9 18.0 18.0 20.6

> trees[, 2] # extrai a segunda coluna e todas as linhas
[1] 70 65 63 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86
71 64 78 80 74 72 77 81 82 80 80 80 87

> trees[1, ] # extrai a primeira linha e todas as colunas
Girth Height Volume
1    8.3      70   10.3

> trees[3, 3] # extrai a terceira linha e a terceira
coluna, 1 valor
[1] 10.2

> trees[1, 3] # extrai o valor da primeira linha e
terceira coluna
[1] 10.3

> trees[c(1:5),c(2, 3)] # extrai somente as linhas 1 a 5 e
as colunas 2 e 3
Height Volume
1      70    10.3
2      65    10.3
3      63    10.2
4      72    16.4
5      81    18.8
```

Uma forma eficiente de acessar as variáveis (colunas) pelo nome é usar o comando `cifrão $`. Neste caso, não recomendamos utilizar a função `attach()`, como citado anteriormente. O uso é basicamente o seguinte: `meusdados$variável` (`meusdados` especifica o conjunto de dados e `variável` a variável selecionada que deseja extrair). Por exemplo, para extrair os dados apenas da circunferência (*girth*) do conjunto de dados “trees”, utilize o seguinte comando:

```
> girth <- trees$Girth

> girth # para ver os valores apenas da circunferência
[1] 8.3  8.6  8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2
11.3 11.4 11.4 11.7 12.0 12.9 12.9 13.3 13.7 13.8 14.0
14.2 14.5 16.0 16.3 17.3 17.5
[28] 17.9 18.0 18.0 20.6
```

Este modo de selecionar variáveis é muito prático e será útil em diversos momentos nesta apostila. Como exemplo, observem o emprego de algumas funções úteis:

```
> mean(trees$Girth) # valor médio da circunferência das
árvores
[1] 13.24839

> mean(trees$Height) # valor médio da altura
[1] 76
```

Para avançarmos um pouco mais na manipulação de dados, vamos utilizar agora um outro conjunto de dados, disponível para o curso, chamado "insetos" (arquivos "insetos.csv").

Estes dados representam contagens numéricas (abundância) de insetos que foram coletadas de armadilhas de 101 locais em North Dakota, South Dakota, Wyoming e Montana (todos nos EUA). Cada site foi classificado como "*grassland*" ou "*forest*". Cada site foi também classificado como "*protected*" ou "*unprotected*" por agências estatais. Para cada espécie, também temos informações sobre o tamanho do corpo do adulto e sua origem (*native* ou *invasive*).

Relembrando, você poderá utilizar o seguinte comando para importar estes dados no R:

```
> meusdados <- read.csv("Insetos.csv", sep=",")
> head(meusdados) # como os dados devem ser
```

	site	habitat	region	status	sp.1	sp.2	sp.3	sp.4	sp.5	sp.6	sp.7	
sp.8	sp.9	sp.10	sp.11	sp.12	sp.13	sp.14	sp.15	sp.16	sp.17	sp.18		
1	1	grassland	WY	unprotected	0	1	6	4	0	0	10	0
0	2	0	1	6	4	2	0	10	0			
2	2	grassland	ND	unprotected	3	5	3	3	3	3	3	3
3	3	3	5	3	4	3	3	3	3			
3	3	grassland	WY	unprotected	3	3	4	9	3	3	3	3
6	3	3	3	4	9	5	4	3	3			
4	4	grassland	ND	unprotected	2	2	2	2	2	2	2	2
2	3	2	2	16	2	2	2	3	2			
5	5	grassland	ND	unprotected	3	4	3	3	3	3	3	3
3	3	3	6	3	4	3	3	3	3			
6	6	grassland	ND	unprotected	3	5	3	16	3	3	3	3
3	3	3	5	3	17	3	3	3	3			
	sp.19	sp.20	sp.21	sp.22	sp.23	sp.24	sp.25	sp.26	sp.27	sp.28	sp.29	sp.30
1	0	2	0	0	0	0	0	3	0	0	0	0
2	4	3	0	0	0	0	0	0	0	0	0	0
3	6	3	0	0	0	0	0	0	1	0	0	0
4	2	3	0	0	0	0	0	1	0	1	0	0
5	3	3	0	0	0	0	0	5	0	2	0	0
6	3	3	0	0	0	0	0	1	0	0	3	0

À medida que trabalhamos com grandes conjuntos de dados, é importante, em

um primeiro momento, conhecermos estes dados em termos de estrutura no R. Quais comandos poderíamos utilizar? Veja abaixo:

```
> dim(meusdados) # 101 linhas e 34 colunas

[1] 101  34

> str(meusdados)

'data.frame':  101 obs. of  34 variables:
  $ site      : int   1 2 3 4 5 6 7 8 9 10 ...
  $ habitat   : Factor w/ 2 levels "forest","grassland":
2 2 2 2 2 2 2 2 2 2 2 ...
  $ region    : Factor w/ 4 levels "MT","ND","SD",...: 4 2
4 2 2 2 4 4 2 2 ...
  $ status    : Factor w/ 2 levels
"protected","unprotected": 2 2 2 2 2 2 2 2 2 2 ...
  $ sp.1      : int   0 3 3 2 3 3 0 0 1 4 ...
  (...)
```

Notem que estamos trabalhando com um objeto do tipo “dataframe”, pois temos mais de um tipo de dados levantados. Estes dados são tanto categóricos quanto numéricos. Neste caso, um objeto do tipo matriz não seria adequado, uma vez que ele tem como premissa apenas um tipo de categoria de dados, como citado anteriormente.

Uma vez que inserimos estes dados no R, podemos começar a treinar algumas manipulações. Inicialmente, vamos somar os valores de colunas ou linhas usando as funções `colSums` para somar colunas e `rowSums` para somar linhas. Mas, iremos somar apenas aquelas colunas correspondentes às espécies de insetos. Vejam como é importante conhecermos os dados que estamos trabalhando.

As colunas que representam as espécies vão da coluna 5 à coluna 34. Portanto, vamos usar os colchetes para selecionar este intervalo:

```
> rowSums(meusdados[, 5:34])

[1]  51  66  84  59  72  95  29  15  41  94  27  74  55
45  55  55  62  37 117  63 102  80  51  36  45  66 179
18  62   7  85  52  93
[34]  52  85  77  21  92  59  39 375 172  98  44 294  54
27  57  55  29  26  22  62  43  19 131  34  36  28  24
58  40  27  15  23  38
[67]  73  49  22  17  26  25  38  44  51  37  19  46  62
45  49  18  23  27  41  64  24  30  51  43  66 107  14
24  59  26  27  40  53
[100]  62   0
```

Se cada linha representa uma armadilha, a soma da abundância parcial de cada espécie por observação resulta na abundância total de indivíduos por armadilha!

De modo similar, podemos somar cada coluna, e teremos a abundância total de cada espécie de inseto nas 101 armadilhas:

```
> colSums(meusdados[ , 5:34])
```

sp.1	sp.2	sp.3	sp.4	sp.5	sp.6	sp.7	sp.8	sp.9	sp.10	sp.11	sp.12	sp.13	sp.14	sp.15	sp.16	sp.17	sp.18	sp.19	sp.20	sp.21	sp.22	sp.23	sp.24	sp.25	sp.26	sp.27	sp.28	sp.29	sp.30
225	194	239	259	169	212	182	168	196	158	280	305	328	349	228	263	215	228	256	187	0	1	0	0	0	256	230	166	208	228

Acredito que agora ficou evidente o grande trunfo e a utilidade de se compreender bem a manipulação de dados no R. É útil observar que o argumento de qualquer função pode ser qualquer outro argumento, desde outra função até dados. E, nos casos dos dados, eles podem ser manipulados, selecionados, etc.

Neste contexto, outra medida interessante (pelo menos no contexto do propósito do curso) que requer manipulação de dados é tentar extrair dos dados a abundância média de insetos, por exemplo, entre as áreas protegidas e não protegidas. Como isto poderia ser executado no R?

Para tal, vamos utilizar a função `taapply()`. A função tem como lógica calcular alguma métrica a partir de dados numéricos entre grupos (categorias), como pode-se observar abaixo:

```
> tapply(rowSums(meusdados[ , 5:34]), meusdados$status,
mean)
```

protected	unprotected
39.55000	60.97531

De forma alternativa, anteriormente poderíamos ter criado um objeto para alocar a soma das linhas (abundância total):

```
> abund <- rowSums(meusdados[ , 5:34])
```

```
> abund
```

[1]	51	66	84	59	72	95	29	15	41	94	27	74	55	45	55	62	37	117	63	102	80	51	36	45	66	179	18	62	7	85	52	93
[34]	52	85	77	21	92	59	39	375	172	98	44	294	54	27	57	55	29	26	22	62	43	19	131	34	36	28	24	58				



```

40  27  15  23  38
[67]  73  49  22  17  26  25  38  44  51  37  19  46  62
45  49  18  23  27  41  64  24  30  51  43  66 107  14
24  59  26  27  40  53
[100]  62  0

```

Neste caso, o novo comando ficaria mais simplificado, muito embora em dois passos:

```

> tapply(abund, meusdados$status, mean)
protected unprotected
39.55000      60.97531

```

E, se ao invés de criarmos um novo objeto com os dados da abundância total de insetos, adicionássemos uma nova coluna permanente aos dados com estes valores? Como poderíamos realizar isto no R? É o que veremos a seguir:

Vamos utilizar a função `apply()` (notem que não é causal a semelhança com a função anteriormente usada, `tapply()`). A função tem os seguintes argumentos:

`apply(x, INDEX, fun)`; onde `x` representa um conjunto de dados; `INDEX`, que normalmente explicita se queremos aplicar uma função às linhas ou às colunas. O valor 1 assinala linhas e o valor 2, colunas. E, por fim, a função que queremos aplicar, tais como soma (`sum()`), média (`mean()`) etc. Além disto, como podemos criar funções no R – como veremos no final deste documento – poderíamos, portanto, utilizar uma função específica que atende às nossas demandas específicas. Mas, vejamos abaixo, como ficaria o comando para o nosso exemplo:

```

> meusdados$abund <-apply(meusdados[, 5:34], 1, sum)
> meusdados$abund

[1]  51  66  84  59  72  95  29  15  41  94  27  74  55
45  55  55  62  37 117  63 102  80  51  36  45  66 179
18  62   7  85  52  93
[34]  52  85  77  21  92  59  39 375 172  98  44 294  54
27  57  55  29  26  22  62  43  19 131  34  36  28  24
58  40  27  15  23  38
[67]  73  49  22  17  26  25  38  44  51  37  19  46  62
45  49  18  23  27  41  64  24  30  51  43  66 107  14
24  59  26  27  40  53
[100]  62  0

```

Percebam que utilizando o prático `$`, foi informado ao R para criar uma nova coluna chamada “`abund`”, que será o resultado da soma das linhas das colunas 5 a 34 (aquelas colunas com os valores de abundância de cada espécie de insetos por armadilha). Esta parte ainda será muito útil na parte dos gráficos, como veremos a frente.

E se quisermos adicionar o total da abundância das espécies (soma das linhas) linha de abundância ao dataframe? Uma maneira de construir isto, em dois passos, é a seguinte: primeiro criaremos um vetor contendo a abundância total das espécies e em seguida pediremos para que essa linha seja adicionada na posição 102 (uma vez que os dados têm 101 linhas) e entre as colunas 5 a 34:

```
> sp.abund <- apply(meusdados[, 5:34], 2, sum)
> sp.abund

> meusdados[102,5:34]<-sp.abund
# adiciona uma nova linha com o vetor sp.abund
> tail(meusdados)
#verifique os resultados
```

E se quisermos criar uma nova linha vazia (sem informação)? Lembre-se que o R não aceita valores em branco, ou seja, devemos preencher com NAs. Vamos efetuar os mesmos passos e criar uma 103ª linha com este vetor de NAs:

```
> vet.na <- rep("NA", 30)
> vet.na <- rep("NA", 30)
> vet.na
[1] "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA"
"NA" "NA" "NA"
[15] "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA" "NA"
"NA" "NA" "NA"
[29] "NA" "NA"
> meusdados[103,5:34]<-vet.na
> tail(meusdados) # confira como ficou
```

Para remover esta última linha, vamos sobrescrever o arquivo “meusdados” excluindo esta última linha:

```
> meusdados<-meusdados[-103,]
```

Outro ponto importante é a seleção de dados mais específicos. Por exemplo, no conjunto de dados em tela, a seleção de armadilhas na qual a abundância fosse apenas maior que 20 indivíduos. Como poderíamos fazer isto no R?

Para fazer isto, vamos utilizar a função `which()`. Esta função, de forma genérica, fornece resultados a partir de um objeto lógico. Esta função é muito importante para selecionar dados e será muito útil também na parte de construção de gráficos. No nosso caso, o comando abaixo irá selecionar apenas as linhas com valores de abundância maiores que 20 (para este novo dataframe, foi atribuído o nome de “dados\_abund\_20”, em referência a valores de abundância por armadilha maiores que 20):

```
> dados_abund_20 <- meusdados[which(meusdados$abund>20), ]
```

```
> head(dados_abund_20) # observe como ficaram os dados
```

Além de argumentos lógicos numéricos, é possível utilizar a função `which()` para selecionar objetos de caracteres (nominais). Por exemplo, como poderíamos selecionar apenas o conjunto de dados coletados em áreas protegidas? Vamos utilizar como base o nosso novo conjunto de dados, `dados_abund_20`.

```
> dados_abund_20_prot <-  
meusdados[which(meusdados$status=="protected"), ]
```

O comando acima utiliza uma lógica muito semelhante ao comando anterior, como é possível perceber. Esta é uma vantagem do R. Uma vez que se compreende a lógica por trás de cada função, a aplicabilidade da função é também naturalmente ampliada.

Nesta lógica, ao invés de selecionar uma parcela dos dados com base em algum objeto lógico, poderia se excluir as parcelas que não são do nosso interesse, como demonstrado abaixo.

```
> dados_abund_20_prot <- meusdados[  
  c(which(meusdados$abund<=20),  
    which(meusdados$status=="unprotected")), ]  
# note o sinal de (-) antes do c. Este sinal de negativo  
indica pra remover os dados especificados.
```

Com este único comando, realizamos os passos anteriores, excluindo valores de abundância menores que 20 e deletando as observações em áreas não protegidas. Sempre teste o resultado do comando:

```
> dados_abund_20_prot
```

Observação: caso você queira combinar vários critérios para filtrar os dados, use a função `c` e vários `which` combinados. Exemplo:

```
> dados_abund_20_prot_MD_ND <-  
meusdados[which(meusdados$abund>=20 &  
meusdados$status=="protected" & meusdados$region %in%  
c("MT", "ND")), ]  
#foram selecionados dados que tenham ao mesmo tempo  
abundância >20, sejam em áreas não protegidas  
(unprotected) e apenas nos estados MT e ND.
```

## Gerando sequências, números aleatórios e repetições

Este tópico apresenta alguns comandos que podem ser úteis em diversos momentos durante a formulação de um *script* mais complexo no R; ou artefatos que

podem ser úteis para gráficos e na criação de funções, como veremos ao longo deste texto. Mas, também poderão ser úteis na construção de gráficos, nosso próximo tópico.

De forma simples, para gerar uma sequência no R, utilizamos um comando muito simples:

```
< 1:10 # gera uma sequência de números de 1 a 10
[1] 1 2 3 4 5 6 7 8 9 10
```

Uma sequência pode ser adicionada a um novo objeto:

```
< a <- 1:100
< a # O resultado será uma sequência de números de 1 a
100
```

Um outro modo de criar uma sequência, mas com possibilidade de especificar os intervalos, é utilizar a função `seq()`. Os argumentos genéricos desta função são os seguintes:

`seq(from = 1, to = 100, by = 4 )`, sequência de um a 100, em intervalos de 4

```
> seq(1, 200, 10)
# sequência de 1 a 200 em intervalos de 10
[1] 1 11 21 31 41 51 61 71 81 91 101 111 121
131 141 151 161 171 181 191
```

Um outro comando interessante é aquele que gera repetições de números. Para tal, a função `rep()` é a adequada:

Para repetir o número quatro 20 vezes, usamos o seguinte comando:

```
< rep(5, 20)
[1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

Veja um pequeno artefato de variação neste comando:

```
< rep(1:10, each=2) # veja o resultado!
```

Para combinar repetições, use a função `concatenate (c)`:

```
< c(rep(1, 5), rep(2, 5))
```

Outros comandos que serão muito úteis são funções que criam conjunto de dados aleatórios. Vamos começar com a função `runif()`, a qual gera dados aleatórios com distribuição uniforme:

```
> runif(n, min=0, max=1) # gera uma distribuição uniforme  
com n valores, com valores máximos e mínimos.
```

Por exemplo, vamos gerar um conjunto de dados com 100 valores, que vão desde 20 (mínimo) até 60 (máximo):

```
> runif(100, 20, 60)  
[1] 59.17655 44.77081 28.51730 52.52110 37.88434 20.97518  
52.07002 53.89839 25.38896 23.45999 46.15411 24.18540  
48.74681 40.83567 50.60774 31.91596 44.13625 (...)
```

Lembre-se que você pode sempre alocar esta sequência de dados em um novo objeto, como demonstrado abaixo:

```
> temp <- runif(100, 20, 60)  
> temp # Verifique o resultado. Entretanto, como são  
números gerados aleatoriamente, eles podem diferir  
minimamente da primeira vez que executamos o comando.
```

Outro modo de geração de dados aleatórios, é variar o tipo de distribuição. Por exemplo, podemos gerar dados aleatórios com distribuição normal, utilizando a função `rnorm`, a qual tem os seguintes argumentos genéricos:

```
> rnorm(n, mean=0, sd=1) # Gera uma sequência aleatória  
com "n" elementos, com média igual a zero e desvio padrão  
(sd: standard deviation) igual a um.
```

Agora, vamos executar um exemplo:

```
> rnorm(200, 5, 3) # criar 200 valores com média 5 e  
desvio padrão 3
```

Na próxima etapa vamos trabalhar com gráficos e poderemos observar variações das distribuições de forma visual. Mas, de forma momentânea, utilize o help da função `?Distributions` para conhecer outras formas de gerar dados aleatórios com diferentes distribuições.

Ainda, outro modo de se extrair amostras aleatórias utilizando a função `sample()`. Ela é utilizada para extrair amostras aleatórias e funciona deste modo:

```
> sample(x, size=1, replace = FALSE) # onde x é o  
conjunto de dados do qual as amostras serão retiradas,  
size é o tamanho da amostra (n), e replace é o argumento  
onde é indicado se a amostra será executada com reposição  
(TRUE) ou sem reposição (FALSE).
```

Vejamos alguns exemplos:

```
> sample(1:20, 5) # extrai 5 números com valores entre 1 e 20.
```

Note que não foi especificado o argumento `replace` da função `sample()`. Neste caso, o R tem como padrão considerar que a amostra é sem reposição (`replace = F`). Vejamos como este argumento funciona:

```
> sample(1:20, 25) # erro, amostra maior que o conjunto de valores; solicitamos 25 números aleatórios, mas nosso conjunto de dados tem apenas 20 elementos (que vão de 1 a 20). Para que seja possível executar esta função corretamente, o R precisa replicar alguns números. Para tal, precisamos adicionar o argumento "replace=T", como segue:
```

```
> sample(1:20, 25, replace=TRUE) # agora sim! O R também compreende se você utilizar apenas replace=T OU replace=F.
```

Podemos adicionar outro argumento dentro desta função, como por exemplo a função `rep()`, vista anteriormente. Neste caso, podemos adicionar dentro da função `sample()` uma repetição desta extração "n" vezes, como é demonstrado abaixo:

```
> sample(1:20, 5, rep=100) # 100 repetições. Execute duas vezes e observe as diferenças!!
```

\* para amostragens ou geração de números aleatórios, pode ser interessante utilizar uma *seed*, ou seja, uma semente que determinará sempre a mesma amostragem ou número aleatório. Isso pode ser útil quando for pertinente reproduzir o resultado:

```
> set.seed(123) # estabelece seed = 123. o comando set.seed deve sempre ser repetido imediatamente antes de qualquer amostragem ou geração de número aleatório, ou seja, o comando não estabelece de maneira permanente que a seed seja sempre a mesma.
```

```
> sample(1:5,5)
[1] 2 4 5 3 1
> set.seed(123)
> sample(1:5,5)
[1] 2 4 5 3 1 # repita algumas vezes os dois comandos acima e compare os resultados.
# repita agora com seeds diferentes
```

## Criação de gráficos no R

Como cientistas, os gráficos são, de um modo geral, um dos principais resultados que almejamos além das análises estatísticas propriamente ditas. Os gráficos adicionam um componente visual aos números, facilitando a interpretação dos resultados. O R apresenta muitas possibilidades de editarmos os gráficos, tornando-os muito personificados. Dentro deste universo, aqui serão abordados os componentes básicos, que tangem à manipulação de comandos que são ferramentas úteis na construção destes gráficos.

A função básica que gera um gráfico no R é o `plot()`. O `plot` tem como argumentos básicos o conjunto dos nossos dados que queremos graficar; normalmente o comando genérico inicia com `plot(x, y)`. Vamos retornar aquele conjunto de dados chamados `trees` para fazermos nosso primeiro plot no R.

```
> data(trees)
```

```
?head(trees) #vamos relembrar as informações dos dados
```

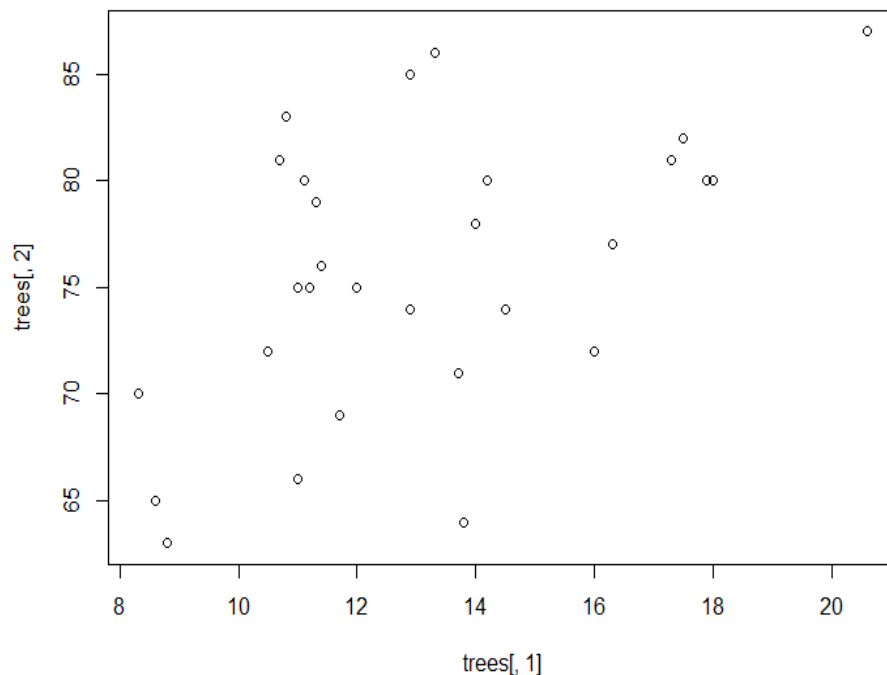
Bom, podemos de forma bem prática criar um gráfico denominado dispersão `xy`, onde vamos analisar visualmente como uma variável `y` (variável dependente) se comporta em função de uma variável `x` (variável independente). Sem pensarmos muito nas questões ecológicas, vamos ver como a altura (*height*) se comporta em relação à circunferência (*girth*):

```
> plot(trees$Girth, trees$Height) #
```

Podemos ainda utilizar o seguinte comando alternativamente:

```
> plot(trees[, 1], trees[, 2])
```

O output deste último comando é o seguinte:



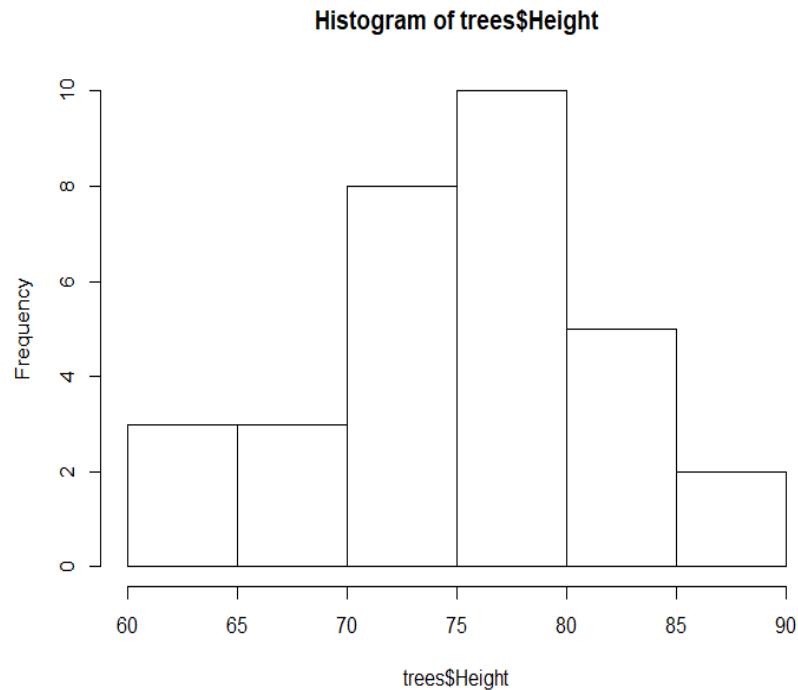
Este comando básico naturalmente não gera um gráfico visualmente bonito e publicável, mas é um primeiro passo importante. Alguns componentes visuais serão explorados aqui. Percebam como a correta manipulação dos dados é um argumento importante dentro da função `plot()`, como o uso do `$` e `[ ]`.

Outro tipo de gráfico comumente utilizado é o histograma. A função para criar um é `hist()`. Ela basicamente carece de um argumento, que são o conjunto de dados. Utilizando ainda o conjunto de dados `trees`, vamos ver como se distribui a altura das árvores em classes:

```
> hist(trees$Height)
```

O gráfico deve ter ficado como o abaixo:





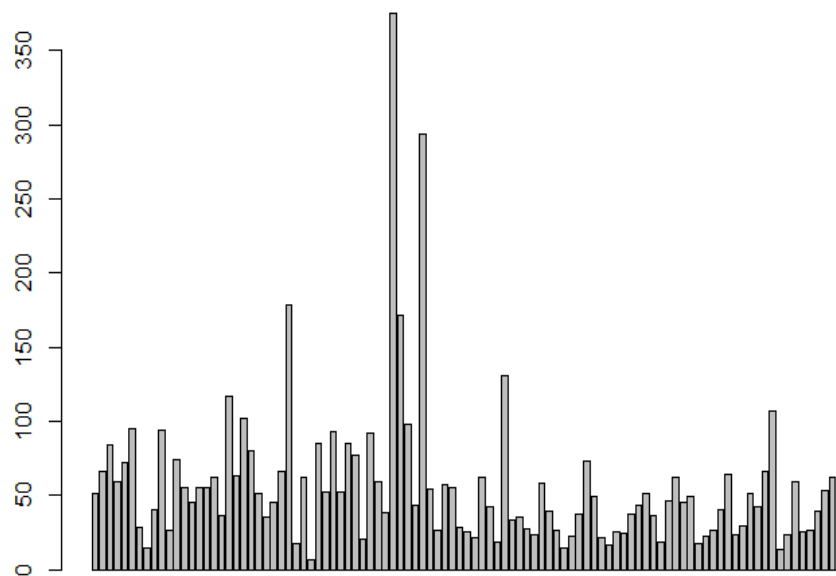
Outro tipo de gráfico bem usual é o de barras. A função que nos permite realizar este tipo de gráfico é o `barplot()`. Como exemplo, vamos voltar a utilizar aquele conjunto de dados de insetos coletados em 101 armadilhas, o qual foi disponibilizado durante o curso:

```
> head(meusdados) #vamos relembrar as variáveis deste  
conjunto de dados
```

Anteriormente, adicionamos uma coluna com a abundância de insetos em cada armadilha. Vamos utilizar estes dados para construir um gráfico de barras.

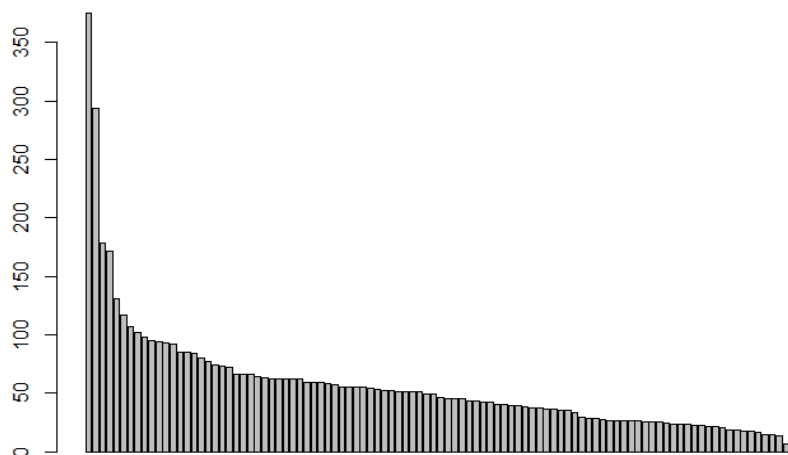
```
> barplot(meusdados$abund)
```

O output é o seguinte:



Um argumento interessante no `barplot`, é organizar as barras em alguma ordem lógica, como crescente ou de decrescente. O comando `sort ()` organiza os dados em ordem crescente de abundância. O comando `rev()` reverte a ordem. Veja o comando abaixo e o seu respectivo resultado:

```
> barplot (rev (sort (meusdados$abund)))
```



Você pode tentar algumas variações deste último comando, e ver os resultados gráficos. Tente, por exemplo:

```
> barplot (sort (meusdados$abund))
```

```
# Perceba o que o argumento rev() fez anteriormente!
```

Nos próximos passos da apostila vamos apresentar alguns argumentos para tornar os gráficos mais lógicos e publicáveis. No momento, vamos seguir para o próximo tipo de gráfico, também comumente utilizado, o boxplot. O comando para gerar este tipo de gráfico é, logicamente, `boxplot()`.

Esta função tem como argumento básico a seguinte construção:

```
> boxplot(x, ...)
```

Mas, ela também pode apresentar variações, e apresentar os seguintes argumentos:

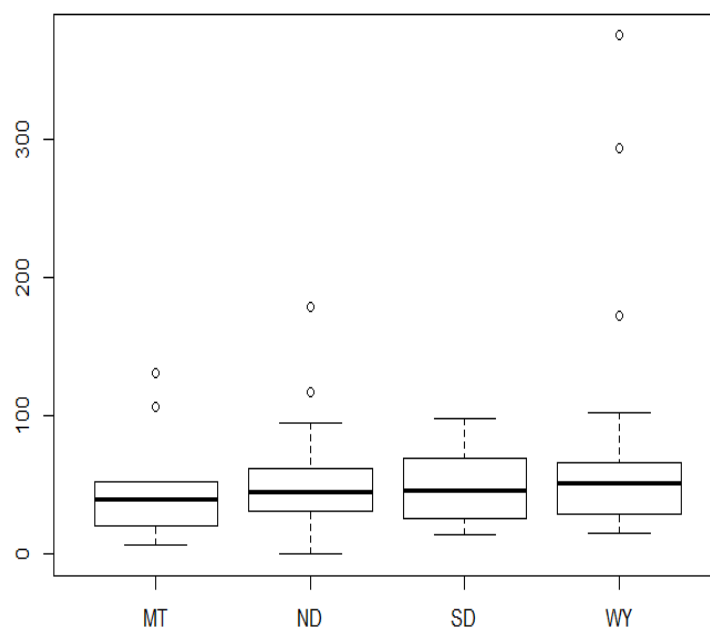
```
> boxplot(formula, ...)
```

Neste caso, o argumento `formula` pode ser algo como `y ~ grp`, onde `y` é um vetor numérico de valores de dados a serem divididos em grupos de acordo com a variável de agrupamento `grp` (geralmente um fator). o símbolo til (`~`) significa “modelado por” (veja mais detalhes no tópico *Estatística Univariada*)

Vamos continuar utilizando o conjunto de dados dos insetos. Vamos construir um boxplot que irá analisar como a abundância (vetor numérico de valores) variou ao longo dos locais de coleta (variável de agrupamento, um fator):

```
> boxplot(meusdados$abund ~ meusdados$region)
```

O gráfico resultante será este:



Uma vez que os tipos mais comuns de gráficos foram elucidados em termos de funções e argumentos básicos, os próximos tópicos irão focar em como mudar a aparência dos gráficos.

## Gráficos em painéis

Um recurso gráfico muito interessante diz respeito ao aproveitamento do espaço. De forma geral, a diagramação de gráficos é um aspecto importante, uma vez que os espaços para figuras em artigos, trabalhos de conclusão, dissertações e teses, dentre outros, são limitados. Uma das formas de se aproveitar este espaço é fazer uma única figura com vários gráficos. A tela que o R gera (e o painel de gráficos do R Studio) por padrão é de um gráfico por painel. Para colocar quatro figuras em um único painel, precisamos informar isto ao R, através de um comando. O comando mais utilizado é o seguinte: `par(mfrow = c(x, y))`, onde `x` é o argumento que especifica o número de linhas e `y` o número de colunas do painel:

Neste caso, para deixarmos em um único painel os quatro tipos de gráficos mais comuns, demonstrados aqui, precisamos dividir a área do painel em 2 linhas e 2 colunas, o que irá gerar quatro áreas em um único painel.

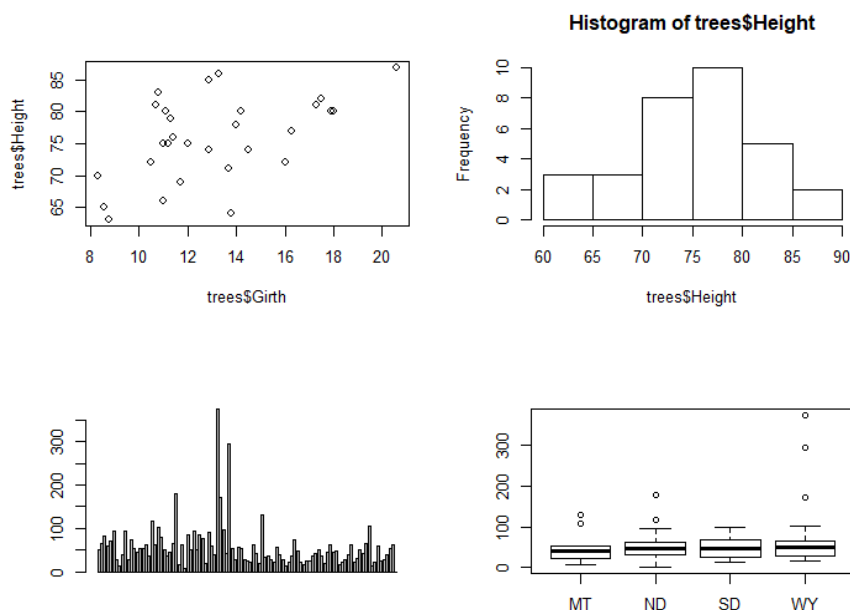
O argumento, portanto, dentro da função `par()`, pode ser o seguinte:

```
> par(mfrow = c(2, 2))
```

Agora, podemos inserir os comandos referentes aos quatro tipos de gráficos que ilustramos acima:

```
> plot(trees$Girth, trees$Height)
> hist(trees$Height)
> barplot(meusdados$abund)
> boxplot(meusdados$abund ~ meusdados$region)
```

O resultado será como este abaixo:



Um ponto importante é que todos os próximos gráficos que o R gerar a partir de comandos ficarão em 4 painéis. Tente executar novamente um dos quatro tipos de gráficos acima e você irá perceber isto. Para evitar este pequeno detalhe, é recomendado adicionar um comando para informar ao R que os próximos gráficos serão em painel único ou qualquer outro tipo de divisão que for do interesse.

Para voltar a termos apenas um gráfico por painel, utilize o seguinte comando:

```
> par(mfrow = c(1, 1))
```

Para testar a sua correta funcionalidade, teste executando qualquer um dos quatro tipos de gráficos que aprendemos acima, como por exemplo:

```
> hist(trees$Height)
```

Agora que aprendemos este pequeno truque, vamos explorar outros argumentos dentro das funções que produzem gráficos, com o propósito de melhorar a aparência daqueles quatro tipos básicos de gráficos.

## Alterando a aparência de gráficos

Vamos começar a trabalhar com aquele gráfico de dispersão xy, nosso primeiro tipo de análise gráfico explorado. O comando era o seguinte:

```
> plot(trees$Girth, trees$Height)
```

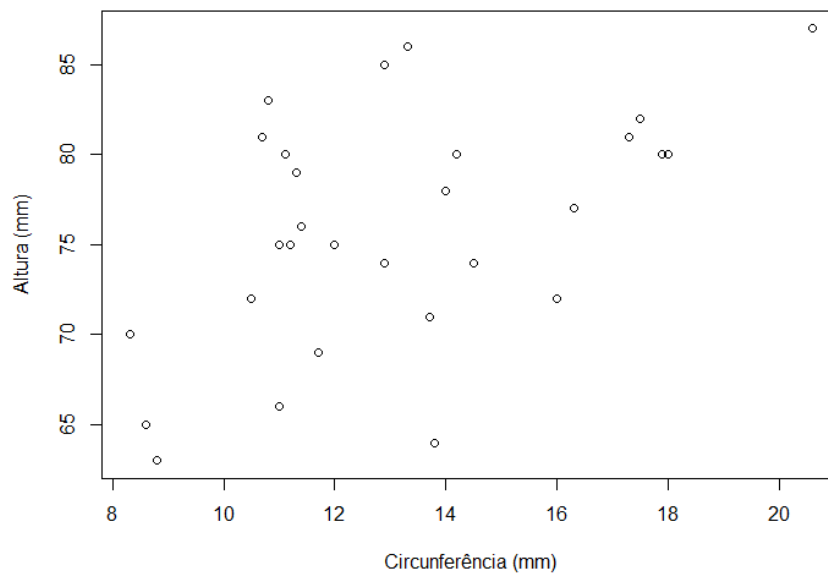
Um gráfico, assim como uma figura, tem como preceito a autoexplicação, ou seja, existe a necessidade de conter informações mínimas para que o leitor compreenda o propósito daquela imagem ou daquele gráfico. No caso do gráfico, especificamente, a correta assinalação dos eixos x e y é imprescindível, por exemplo.

Quase todos estes ajustes de aparência, são argumentos dentro das funções dos gráficos, neste caso, dentro da função `plot()`, como veremos abaixo.

O comando no qual especificamos o título do eixo x é `xlab`, bem como para o eixo y, é `ylab`. Nosso comando, então, neste caso, poderia ficar assim:

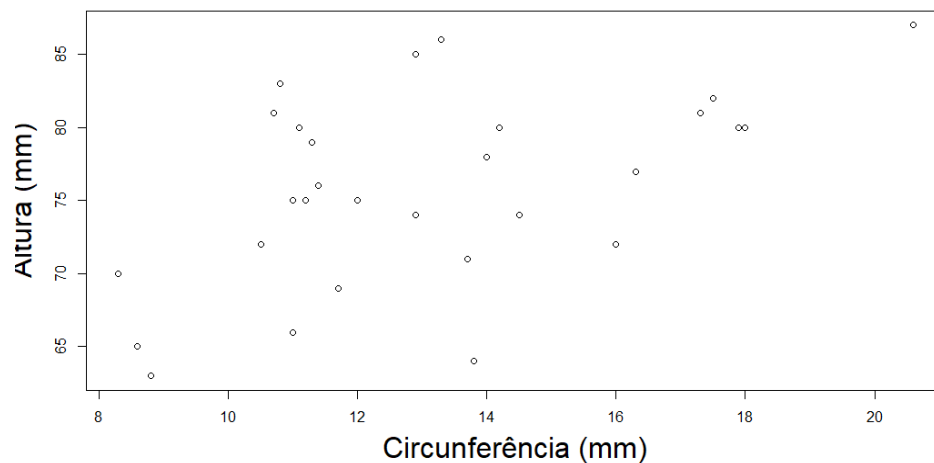
```
> plot(trees$Girth, trees$Height,xlab="Circunferência  
(mm)",ylab="Altura (mm)")
```

O resultado seria este:



Para alterar o tamanho das letras, utilize o argumento `cex.lab`:

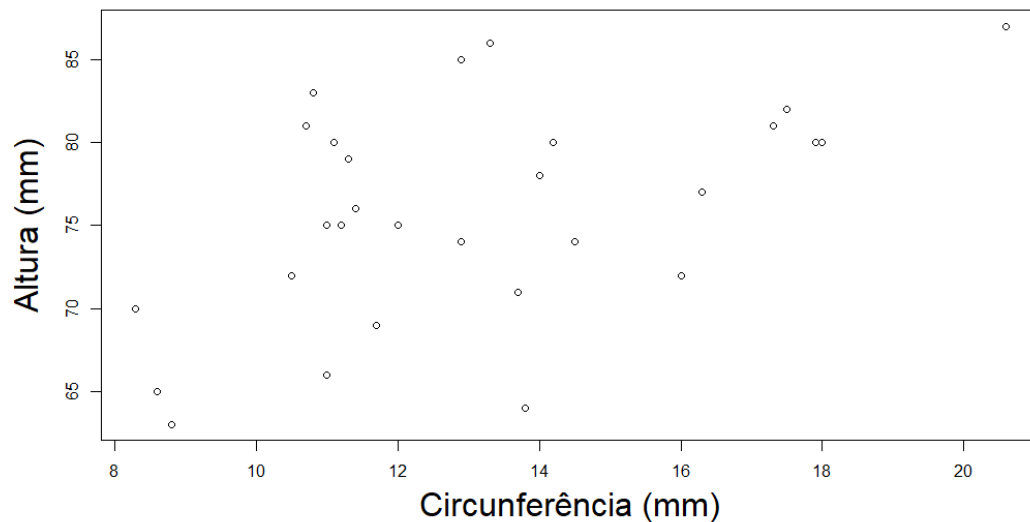
```
> plot(trees$Girth, trees$Height,xlab="Circunferência  
(mm)",ylab="Altura (mm)",cex.lab=2)
```



Observe que o aumento do tamanho da letra dos labels deixou pouca margem. Para ajustá-la, utilize o comando `par(mar=c(5,4,4,2))`. Este é o default para as margens c(inferior, esquerda, superior, direita). vamos alterar apenas a margem esquerda:

```
> par(mar=c(5,6,4,2))
> plot(trees$Girth, trees$Height, xlab="Circunferência
(mm)", ylab="Altura (mm)", cex.lab=2)
```

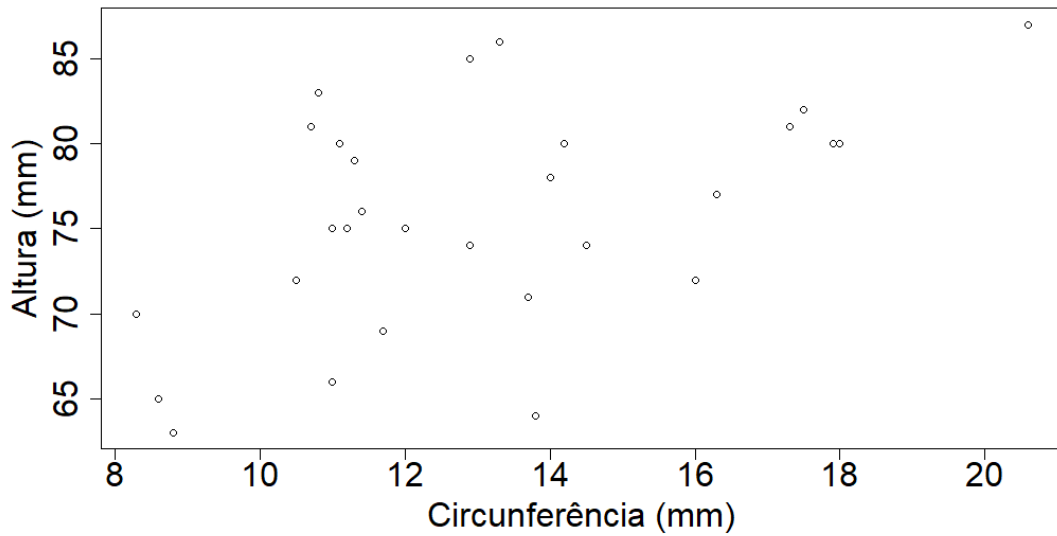
\*uma observação importante: o comando `par(mar())` altera a área de plotagem de maneira permanente. Para mudar, deve-se utilizar o comando novamente com outras especificações.



Para alterar o tamanho do texto dos eixos, utiliza o comando `cex.axis`.

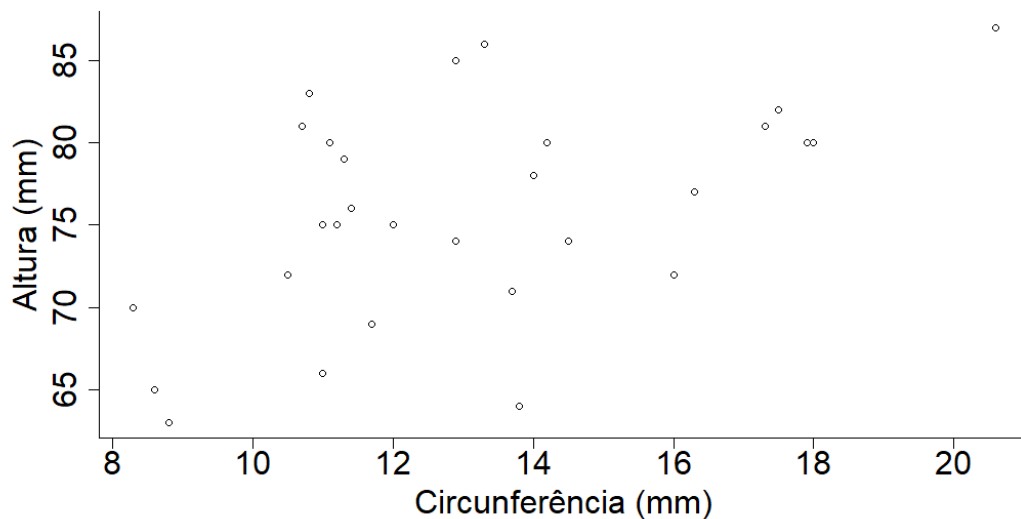
```
> plot(trees$Girth, trees$Height, xlab="Circunferência
```

```
(mm)", ylab="Altura (mm)", cex.lab=2, cex.axis=2)
```



Para remover a caixa em torno do gráfico e manter só as margens inferior e esquerda, use o argumento `bty="l"`.

```
> plot(trees$Girth, trees$Height, xlab="Circunferência  
(mm)", ylab="Altura (mm)", cex.lab=2, cex.axis=2, bty="l")
```



\*`bty="o"` produz o mesmo gráfico default e `bty="n"` deixa apenas os eixos, sem nenhuma das margens da caixa.

Existem uma série de outros argumentos, todavia, que poderíamos usar. Dentre todos, outro interessante é o `pch`. Este argumento diz respeito ao símbolo dos pontos. O padrão é estes círculos abertos, mas existem outros símbolos que poderíamos utilizar.



Para visualizar as possibilidades do argumento `pch`, vamos utilizar a ajuda do R (Veja mais detalhes em Quick R: Graphical Parameters <https://www.statmethods.net/advgraphs/parameters.html>)

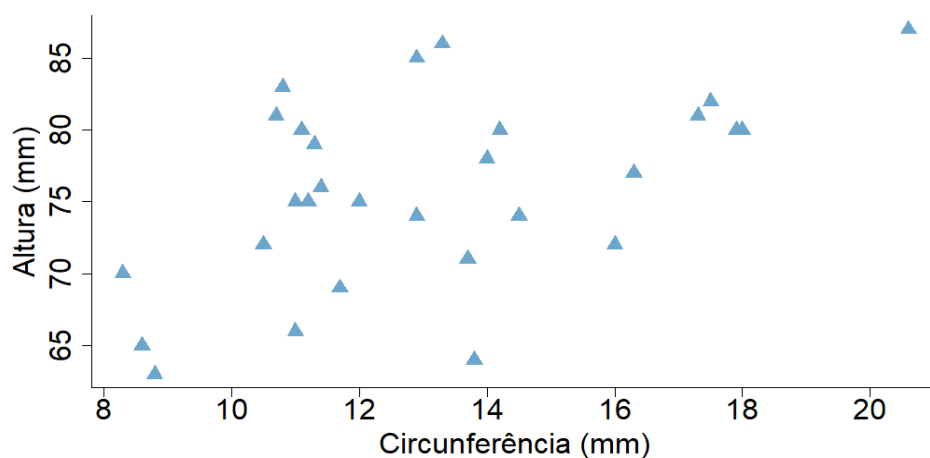
```
> ?pch
# abaixo a informação mais importante no momento
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
□	○	△	+	×	◇	▽	⊠	*	⊕	⊗	⊙	⊠	⊗	⊙	■	●	▲	◆	●	●	●	■	◆	▲	▽

Vamos, então, utilizar este argumento, conjuntamente com outro argumento importante, o que adiciona cores, `col()`. Consulte no google antes a cartilha em PDF com as cores padrão do R, procurando por “RColors”. Uma possibilidade de comando com estes dois argumentos poderia ser algo assim:

```
> plot(trees$Girth, trees$Height, xlab="Circunferência
(mm)", ylab="Altura (mm)", cex.lab=2, cex.axis=2, bty="l", cex=
2, pch=17, col="skyblue3")
#o argumento cex determina o tamanho dos pontos
```

O novo layout do gráfico ficará deste modo:



De modo similar, podemos adicionar título aos eixos e colorir nosso histograma, como por exemplo:

```
> hist(trees$Height, col="red3", xlab="Altura (mm)",
ylab="Frequência")
# Observe como ficou o novo layout do histograma
```

O argumento que adiciona um título principal padrão nos gráficos é `main`. Podemos adicionar este argumento para modificar o título principal que a função `hist` automaticamente insere no gráfico, como segue:

```
> hist(trees$Height, col="red3", xlab="Altura (mm)",  
ylab="Frequência", main="")
```

Outro aspecto importante no histograma é a divisão das classes e os limites do eixo que representam nossos dados. Para modificar isto, existe um argumento da função `hist` chamado `breaks`. Para gerar estes intervalos, vamos utilizar uma função já apresentada, as sequências numéricas, a função `seq()`.

Vamos tentar. Primeiramente, é importante conhecer os limites numéricos dos dados que estamos explorando:

```
> max(trees$Height)  
[1] 87
```

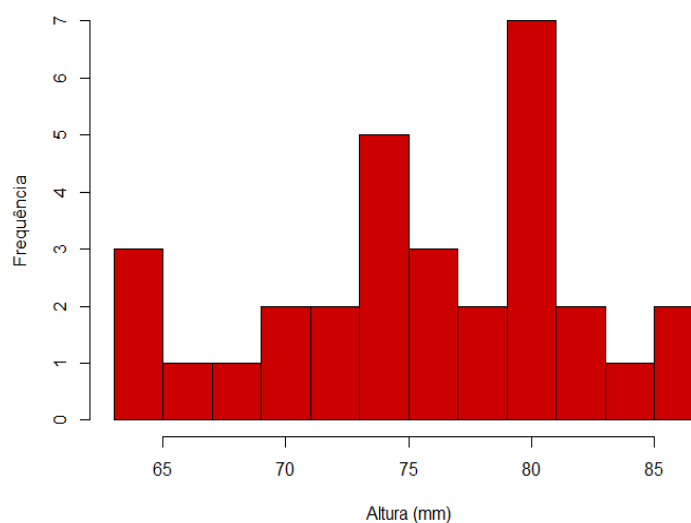
```
> min(trees$Height)  
[1] 63
```

Relembrando, a função `seq()` tem três argumentos, que são os limites e a divisão dos intervalos, como podemos ver abaixo:

```
> seq(63, 87, 1)
```

Incorporando isto como argumento de `breaks`, dentro da função `hist`, temos o seguinte comando e resultado:

```
> hist(trees$Height, breaks=seq(63, 87, 2), col="red3",  
xlab="Altura (mm)", ylab="Frequência", main=" ")
```



Está lembrando daquele tópico no qual abordamos como criar números aleatórios, dados com média e desvio padrão definidos, com a função `rnorm()`, por exemplo? Tentem explorar variações no layout do gráfico. Abaixo um exemplo:

```
> simul <- rnorm(500, 10, 3)
# conjunto de dados com n=500, média igual a 10 e desvio
padrão igual a 3
```

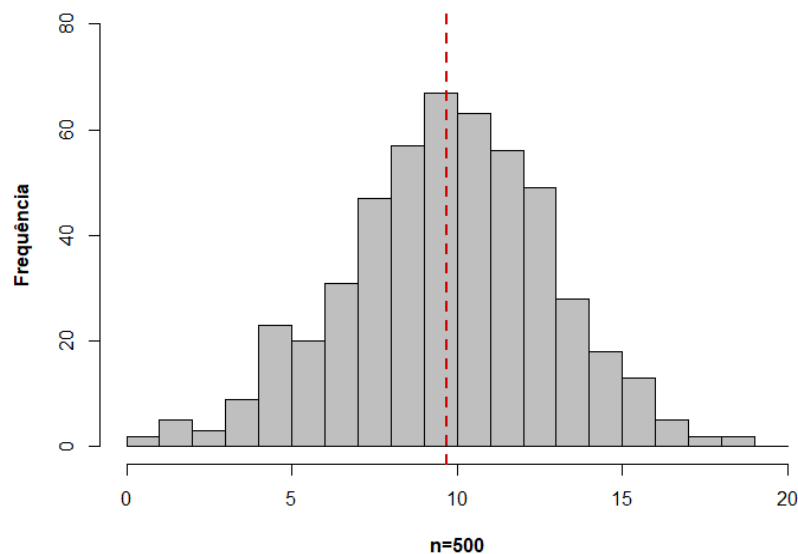
Um gráfico mais simples seria apenas o comando

```
> hist(simul) # veja o resultado
```

Agora, tente tornar mais rebuscado o gráfico, como por exemplo, adicionar uma linha pontilhada que representa a média (que será próximo a 10!). Vamos em partes:

```
> hist(simul, breaks=seq(0, 20, 1), col="gray75",
xlab="n=500", ylab="Frequência", main="", ylim=c(0, 80),
font.lab=2)
> abline(v=mean(simul), col="red3", lty=2, lwd=2)
```

O design do gráfico será o seguinte:



Notem que foram adicionados novos argumentos dentro da função `hist()`, como por exemplo o `font.lab=`, que está relacionado ao estilo de fonte dos títulos dos eixos do gráfico (itálico, negrito, sublinhado etc.).

Também foi adicionado uma linha vertical vermelha indicando o valor da média dos dados. Para isto, foi usado a função `abline()` e seus respectivos argumentos,

como é possível observar no comando anterior.

A função `abline()` também pode ser interessante para adicionarmos um modelo de regressão linear em gráficos do tipo XY (ou dispersão). Para ilustrar isto, vamos retomar o uso dados "trees".

Para criar um gráfico que analisou uma possível relação entre circunferência e altura de árvores do Hemisfério Norte. O último comando que usamos para observar esta possível relação, com pequenas alterações no layout, foi o seguinte:

```
> plot(trees$Girth, trees$Height, xlab="Circunferência  
(mm)", ylab="Altura (mm)", cex.lab=2, cex.axis=2, bty="l", cex=2, pch  
=17, col="skyblue3")
```

A função `lm()` é a responsável por criar um modelo linear de regressão entre variáveis dependentes (y) e independentes (x).

Ela tem como lógica os seguintes argumentos básicos:

```
> lm(y ~ x) # como y responde a x
```

No exemplo proposto, o comando seria algo do tipo:

```
> lm(trees$Height ~ trees$Girth)
```

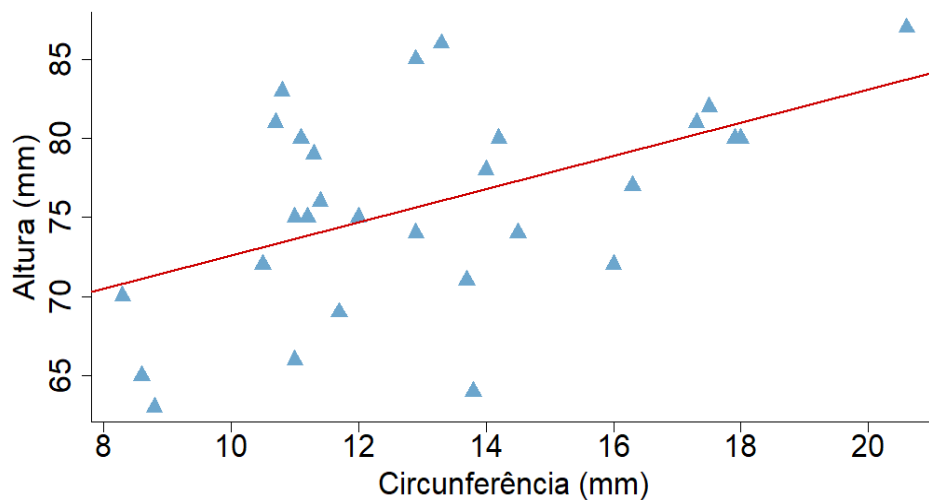
Mas, sempre é recomendado alocar estes resultados a um novo objeto, que aqui chamaremos de "a":

```
< a <- lm(trees$Height ~ trees$Girth)
```

Agora, podemos adicionar o resultado gráfico desta regressão linear:

```
< abline(a, col="red3", lwd=2)
```

O resultado será este, aproximadamente:



Alterações avançadas em gráficos:

Inserir texto: função `text(x, y, "meu texto")`. `x` e `y` são as posições no plot em que o texto deve ser adicionado:

```
<text(18, 65, "Eu amo o R", cex=2, col="red") # os
argumentos cex e col são usados para alterar o tamanho do
texto e a cor, respectivamente.
```

O texto pode ser colocado diretamente também através de clique do mouse com a função `locator()`.

```
<text(locator(), "Eu amo o R", cex=2, col="red")

# o R ficará aguardando a posição a ser especificada.
Clique com o botão esquerdo no local que o texto deve ser
colocado e em seguida tecle ESC. O texto será colocado de
maneira centralizada em relação ao local em que foi
clicado.
```

A função `locator()` pode ser usada também separadamente para se descobrir as coordenadas de pontos específicos dentro do plot.

```
<locator() # clique com o botão direito em todos os
pontos que se quer descobrir as coordenadas. Em seguida
tecle ESC e as coordenadas aparecerão no prompt de
comando. Isso pode ser útil para se descobrir a
coordenada para colocar na função text().
```

Alterar a aparência dos eixos de maneira independente: algumas modificações de eixos não são possíveis diretamente especificando argumentos através da função

`plot()`. Nestes casos, é necessário plotar o gráfico ocultando os eixos e adicioná-los posteriormente um a um de maneira customizada:

```
> plot(trees$Girth,
trees$Height, axes=F, bty="l", cex=2, pch=17,
col="skyblue3", ylab="", xlab="", bty="l", xlim=c(8, 20), ylim=c
(60, 90))
# axes=F ou FALSE indica que os eixos não devem ser
plotados. xlab="" e ylab="" indicam que não deve ser
escrito nada nos rótulos dos eixos.

> axis(1, cex.axis=2) # 1 indica o eixo x

> axis(2, cex.axis=2, las=2) # 2 indica o eixo y; o
argumento las rotaciona os labels do eixo

> mtext("Circunferência (mm)", side=1, cex=2)
# observe que o rótulo do eixo ficou em cima do eixo.
Ajuste a posição do texto com line (funciona também para a
posição do eixo, na função axis):

> mtext("Circunferência (mm)", side=1, cex=2, line=3)

> mtext("Altura (mm)", side=2, cex=2, line=3.5)
```

Note que o plot gera um “gap” entre os eixos, que deixa o gráfico não tão elegante. Para removê-lo, devemos adicionar um argumento junto à função `par`, o `xaxs` e o `yaxs`. Em seguida especificar os limites adequados de x e y, lembrando de alterá-los na colocação dos eixos também:

```
> par(mar=c(5, 8, 4, 4), xaxs="i", yaxs="i")
> plot(trees$Girth, trees$Height, axes=F,
bty="l", cex=2, pch=17, col="skyblue3",
ylab="", xlab="", bty="l", xlim=c(8, 20), ylim=c(60, 90))
> axis(1, cex.axis=2, xlim=c(8, 20))
> axis(2, cex.axis=2, las=2, ylim=c(60, 90))
> mtext("Circunferência (mm)", side=1, cex=2, line=3)
> mtext("Altura (mm)", side=2, cex=2, line=3.5)
```

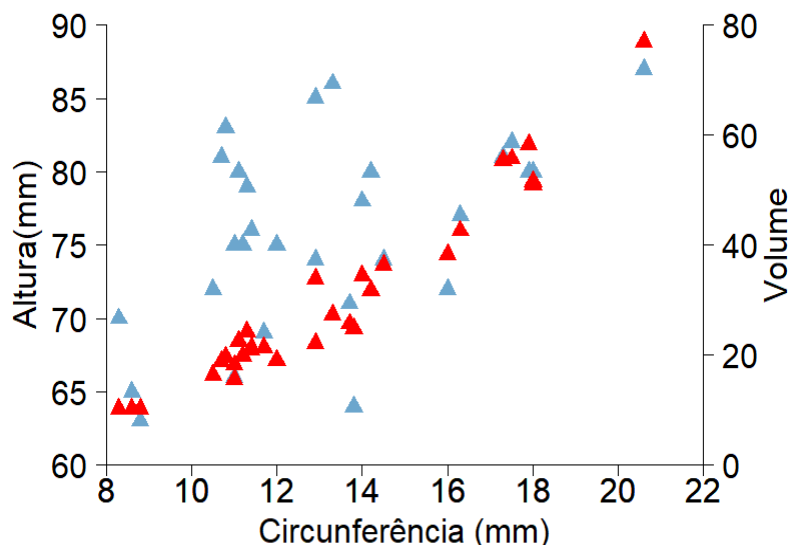
**IMPORTANTE:** sempre que os eixos forem adicionados desta maneira, deve-se ter o cuidado de conferir se as amplitudes de x e y são as mesmas. Caso contrário os plots serão feitos erroneamente em escalas diferentes:

```
> plot(trees$Girth,
trees$Height, axes=F, bty="l", cex=2, pch=17,
col="skyblue3", ylab="", xlab="", xlim=c(0, 20), ylim=c(0, 85))
> axis(1, cex.axis=2, xlim=c(0, 20))
> axis(2, cex.axis=2, ylim=c(0, 85))
```

```
> mtext("Circunferência (mm)", side=1, cex=2, line=3)
> mtext("Altura (mm)", side=2, cex=2, line=3.5)
```

Para plotar segundo eixo ou mais: Cada vez que se usa a função plot, a área de plotagem é limpa e substituída pelo novo comando. Para ativar a subscrição (plotagem em várias camadas), use a função par(new=T).

```
> par(mar=c(5,6,4,6), xaxs="i", yaxs="i")
> plot(trees$Girth, trees$Height, axes=F, bty="l", cex=2,
pch=17, col="skyblue3", ylab="", xlab="", xlim=c(8,22),
ylim=c(60,90))
]> axis(1, cex.axis=2, xlim=c(8,22))
> axis(2, cex.axis=2, las=2, ylim=c(60,90))
> mtext("Circunferência (mm)", side=1, cex=2, line=3)
> mtext("Altura (mm)", side=2, cex=2, line=3.5)
> par(new=T)
> plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2,
pch=17, col="red", ylab="", xlab="", bty="l", xlim=c(8,22),
ylim=c(0,80))
> axis(4, cex.axis=2, las=2, ylim=c(0,80))
> mtext("Volume", side=4, cex=2, line=3.5)
```



Caracteres especiais: abaixo alguns exemplos de como colocar caracteres especiais nos textos (pode ser aplicado aos títulos de eixos também):

```
> text(locator(), expression(paste("Biomassa Bacteriana", "
", "(x", " ", "10", " ", "mu", "g", " ", "C", " ", "mL^-1, ")"), cex=2)
> text(locator(), expression("PO"[4]^3), cex=2)
> text(locator(), expression("NO"[3]^-""), cex=2)
> text(locator(), expression("NO"[2]^-""), cex=2, )
> text(locator(), expression(paste(bold(Chl), "
", bold(italic(a))))), cex=2)
```

### Algumas funções adicionais úteis:

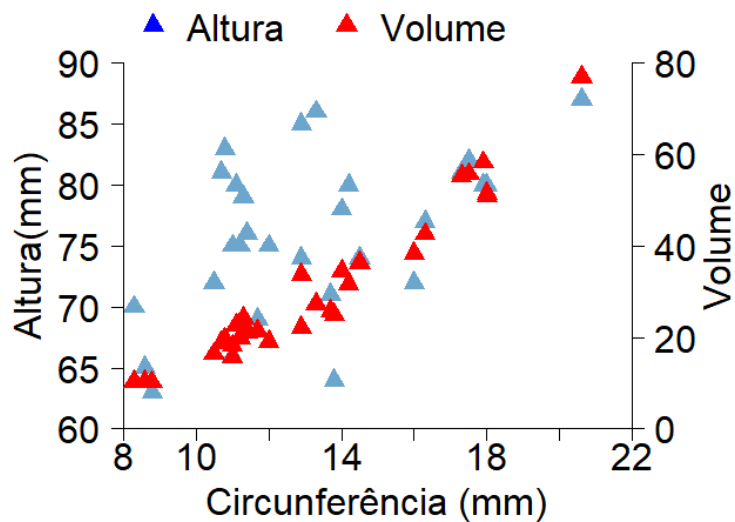
Existe um argumento dentro da função `par` que permite que sejam plotados dados, textos ou legendas, etc. fora da área de plotagem. Este é o `xpd=TRUE`. Ele pode ser adicionado junto com o par inicial que determina as características gerais da área de plotagem do gráfico ou pode ser utilizado em uma linha de comando independente:

```
> par(xpd=TRUE)
#Exemplo 1
> par(mar=c(5,6,4,6),xaxs="i",yaxs="i", xpd=TRUE)

# Exemplo 2: digamos que se queira colocar a legenda em
cima da figura. Neste caso devemos usar o par(xpd=TRUE)
antes do comando da legenda:

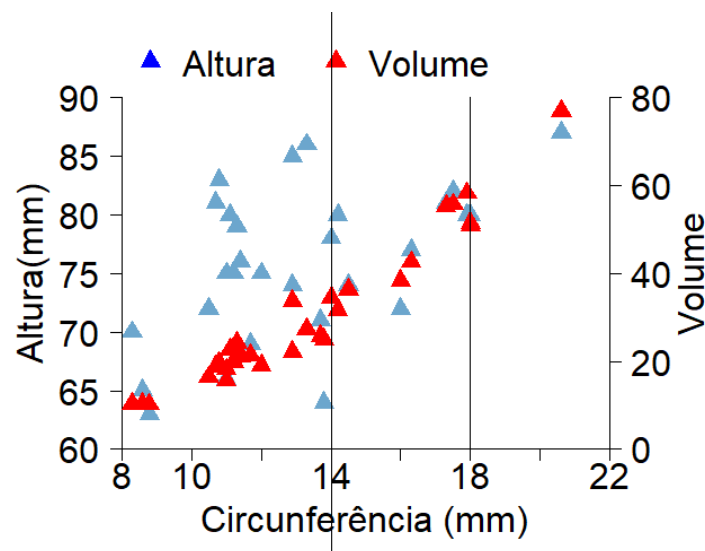
>plot(trees$Girth, trees$Height, axes=F, bty="l",
cex=2,pch=17, col="skyblue3", ylab="",xlab="",
xlim=c(8,22), ylim=c(60,90))
>axis(1,cex.axis=2,xlim=c(8,22))
>axis(2,cex.axis=2,las=2,ylim=c(60,90))
>mtext("Circunferência (mm)",side=1,cex=2,line=3)
>mtext("Altura(mm)",side=2,cex=2,line=3.5)
>par(new=T)
>plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2,
pch=17,col="red", ylab="", xlab="", bty="l",
xlim=c(8,22), ylim=c(0,80))
>axis(4, cex.axis=2,las=2,ylim=c(0,80))
>mtext("Volume",side=4,cex=2,line=3.5)
>par(xpd=TRUE)
>legend(locator(),c("Altura","Volume"),
pch=17,col=c("blue","red"),cex=2,bty="n", horiz=T)
# teste colocando na parte bem superior do gráfico. O
argumento horiz=T plota a legenda no modo horizontal:
```





Algumas funções, como o `abline` são circunscritas à área de plotagem. Se você usar `abline` após usar `par(xpd=TRUE)`, ele irá criar uma linha reta que irá trespassar esta área:

```
>abline(v=14)
> par(xpd=F)
> abline(v=18) # a reversão do xpd para "Falso" retorna
as plotagens à área de plotagem default:
```



Por último, ainda podemos alterar os intervalos das escalas dos eixos x e y. Lembre-se que alteramos os limites (a amplitude) dos eixos através da função `xlim` e `ylim`. Algumas vezes, o R retorna com alguma solução não muito elegante (intervalos muito longos ou muito curtos). Podemos customizar também estes intervalos de acordo com nossa preferência. Para isto basta trocarmos o argumento

`xlim` e `ylim` dentro de `axis` por `at=seq(mínimo, máximo, by=intervalo desejado)`. No exemplo abaixo mostramos apenas uma das linhas de comando do último gráfico na linha em que pedimos para que o eixo x seja plotado. Neste caso, queremos que o limite vá de 8 a 22, com intervalo de 4:

```
> axis(1,cex.axis=2,at=seq(8,22,by=4))
```

Por último, caso você se incomode com a notação científica, você pode mudar a configuração do R para que os números com menos de 999 dígitos sejam mostrados de maneira exata (999 foi um número escolhido ao acaso - poderia ser qualquer outro número).

```
> options(scipen = 999)
> options(scipen = 0) # para retornar ao default
```

## Exportação de gráficos

Uma das grandes vantagens do R é a possibilidade de fazer gráficos personalizados, como demonstramos com alguns poucos exemplos acima. Outra excelente vantagem é podermos exportar do R (ou RStudio) gráficos em formatos e layout pronto para publicação; seja no formato PDF, JPG ou TIFF. Podemos escolher a resolução, o tamanho da figura, dentre outras quase infinitas possibilidades.

Aqui, iremos focar em dois formatos mais usualmente utilizados para exportar figuras, o formato PDF e o formato TIFF. Ambos são vetoriais e, em nosso entendimento, armazenam mais informações gráficas.

Vamos utilizar como exemplo, a criação do último gráfico. Os comandos estão novamente inseridos abaixo:

```
> par(mar=c(5,6,4,6),xaxs="i",yaxs="i")
> plot(trees$Girth, trees$Height, axes=F, bty="l", cex=2,
pch=17, col="skyblue3", ylab="",xlab="", xlim=c(8,22),
ylim=c(60,90))
]>axis(1,cex.axis=2,xlim=c(8,22))
> axis(2,cex.axis=2,las=2,ylim=c(60,90))
> mtext("Circunferência (mm)",side=1,cex=2,line=3)
> mtext("Altura (mm)",side=2,cex=2,line=3.5)
> par(new=T)
> plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2,
pch=17,col="red", ylab="", xlab="", bty="l", xlim=c(8,22),
ylim=c(0,80))
> axis(4, cex.axis=2,las=2,ylim=c(0,80))
> mtext("Volume",side=4,cex=2,line=3.5)
```

Para exportar um gráfico em PDF, vamos utilizar a função `pdf()`. Veja mais detalhes desta função, utilizando a função de ajuda do R (`?pdf`).

O comando básico pode ser o que segue:

```
> pdf("Gráfico 1__Teste.pdf", width=7, height=5)
# O primeiro argumento é o nome que você quer dar para o
arquivo PDF que será exportado. E, aqui neste exemplo, o
tamanho do gráfico. Nesta função, as unidades estão em
inches (sistema americano). 1 inch representa 2.54 cm.
```

A lógica é que o primeiro comando seja o acima, depois todos aqueles comandos do gráfico, e por fim, um comando específico para que ele exporte este arquivo, sinalizando que você não irá mais adicionar argumentos ao gráfico:

```
> dev.off()
```

O comando todo está abaixo. Mas, para onde o R irá exportar este arquivo PDF? Para o diretório de trabalho que você definiu no início deste documento! Tente utilizar o comando `getwd()` que o R irá apontar o local, caso não recorde.

```
> pdf("Gráfico 1__Teste.pdf", width=7, height=5)
# Exportando um gráfico em PDF. Observe o nome genérico
dado ao arquivo
> op <- par(mar=c(5,6,4,6),xaxs="i",yaxs="i")
# Um outro modo de configurar as margens do gráfico

> plot(trees$Girth, trees$Height, axes=F, bty="l", cex=2,
pch=17, col="skyblue3", ylab="",xlab="", xlim=c(8,22),
ylim=c(60,90))
> axis(1, cex.axis=2, xlim=c(8,22))
> axis(2, cex.axis=2, las=2, ylim=c(60,90))
> mtext("Circunferência (mm)",side=1,cex=2,line=3,
font=2)
> mtext("Altura (mm)",side=2,cex=2,line=3.5, font=2)
> par(new=T)
> plot(trees$Girth, trees$Volume, axes=F, bty="l", cex=2,
pch=17, col="red", ylab="", xlab="", bty="l",
xlim=c(8,22), ylim=c(0,80))
> axis(4, cex.axis=2, las=2, ylim=c(0,80))
> mtext("Volume", side=4, cex=2, line=3.5, font=2)
> par(op)
# Reseta os parâmetros de margem previamente definidos
> dev.off()
```

Como ficou o gráfico?

De forma muito similar, podemos exportar este mesmo gráfico em formato TIFF. Basta apenas alterar o primeiro comando para:

```
> tiff("Gráfico 1__Teste.tiff", height=12, width=17,
units='cm', compression="lzw", res=300)
# Aqui alteramos as unidades para cm, definimos o tipo
mais comum de compreensão do arquivo e a resolução, 300
```

`dpi`. Veja mais argumentos desta função com a ajuda do R.

Inserir todos os demais comandos gráficos. A seguir, use o comando abaixo para sinalizar ao R que você quer exportar o gráfico:

```
> dev.off()
```

O R irá exportar uma figura em formato TIFF para o diretório de trabalho definido. Encontre a figura e veja como ficou. O mesmo vale para os demais formatos, como JPG, por exemplo.

## **Estatística univariada: comparando duas ou mais médias, pressupostos, testes de associação entre variáveis**

### **Comparação entre duas médias**

#### Teste-t

Usado para dados contínuos associados a duas categorias (grupos). Pode ser realizado através da função `t.test`. Para isto, vamos utilizar o conjunto de dados `Insetos.csv` (`meusdados`) e testar se a abundância total dos insetos difere entre áreas protegidas e não protegidas. Vamos verificar primeiro quantas observações (N) há em cada categoria:

```
> summary(meusdados$status)
protected unprotected
          20          81
```

Como o N difere entre os grupos, vamos fazer uma amostragem de 20 elementos ao acaso dentro da categoria `unprotected`. Para garantir a reprodutibilidade dos dados, vamos usar também a função `set.seed`:

```
> set.seed(200)
> t.test(meusdados[which(meusdados$status=="protected"), 35],
, sample(meusdados[which(meusdados$status=="unprotected"),
35], 20))
```

```
Welch Two Sample t-test
data:  meusdados[which(meusdados$status == "protected"),
35] and sample(meusdados[which(meusdados$status ==
"unprotected"), 35], meusdados[which(meusdados$status ==
"protected"), 35] and      20)
t = -2.116, df = 29.83, p-value = 0.0428
alternative hypothesis: true difference in means is not
equal to 0
95 percent confidence interval:
-33.9029775 -0.5970225
```

```
sample estimates:
mean of x mean of y
    39.55    56.80
```

\*o teste-t pode ser usado de maneira unicaudal ou bicaudal. O default é o bicaudal. Para alterar, acrescente o argumento

```
alternative=("greater","less","two.sided").
```

### Teste de Mann-Whitney-Wilcoxon (versão não paramétrica do teste-t):

```
> set.seed(200)
> wilcox.test(meusdados[which(meusdados$status=="protected"), 35],
sample(meusdados[which(meusdados$status=="unprotected"), 35], 20))
```

```
Wilcoxon rank sum test with continuity correction
data: meusdados[which(meusdados$status == "protected"),
34] and sample(meusdados[which(meusdados$status ==
"unprotected"), 35], meusdados[which(meusdados$status ==
"protected"), 35] and 20)
W = 141, p-value = 0.1133
alternative hypothesis: true location shift is not equal
to 0
# observe que o teste não paramétrico não apontou
diferença significativa. Isto ocorre porque este teste é
menos robusto que o teste paramétrico e necessita de
diferenças maiores entre os grupos para apontar um
resultado significativo.
```

## **Comparação entre > 2 médias**

### ANOVA (Análise de Variância)

A Análise de variância testa se há diferença entre pelo menos duas médias em um conjunto de 2 ou mais médias (categorias). Para realizar esta análise, vamos utilizar a função `lm` (*linear model*). Nesta função é necessário que seja especificado um fator. Vamos criar um fator fictício chamado de Fator 1:

```
> Fator1<-c(rep(1,5), rep(2,5), rep(3,5))
> Fator1
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
> Fator1<-factor(Fator1) # esta função indica que os dados
do objeto fator devem ser lidos como categóricos - um
fator com 3 níveis. Confira:
> Fator1
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

**Agora vamos gerar um vetor com 15 observações:**

```
> set.seed(176)
> a <- c(rnorm(5, mean=2, sd=1), rnorm(5, mean=10,
sd=1), rnorm(5, mean=15, sd=1))
# observe que foram gerados números aleatórios com mesmo
desvio-padrão, mas médias diferentes
```

**Agora sim a ANOVA:**

```
> modelo1 <- lm(a~Fator1)
> summary(modelo1)
```

Call:

```
lm(formula = a ~ Fator1)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.6842	-0.8113	-0.1642	0.8405	1.5556

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-4.9105	0.7655	-6.415	2.29e-05 ***
Fator1	7.0889	0.3543	20.006	3.78e-11 ***

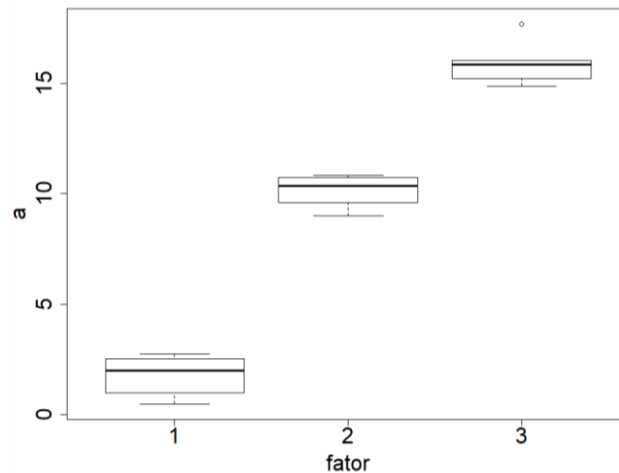
---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.121 on 13 degrees of freedom  
Multiple R-squared: 0.9685, Adjusted R-squared: 0.9661  
F-statistic: 400.2 on 1 and 13 DF, p-value: 3.784e-11

**Vamos inspecionar o gráfico:**

```
> boxplot(a~Fator1, cex.axis=2, ylab="a", xlab="Fator
1", cex.lab=2)
# teste refazendo o vetor a com desvios-padrão diferentes
e veja o que acontece.
```



**Teste de Kruskal-Wallis (versão não paramétrica da ANOVA):**

```
> modelo2 <- kruskal.test(a~Fator1)
> modelo2
```

Kruskal-Wallis rank sum test

```
data: a by Fator1
Kruskal-Wallis chi-squared = 12.5, df = 2, p-value =
0.00193
```

**Localizando as diferenças (teste post-hoc de Tukey (paramétrico) e de Mann-Whitney-Wilcoxon (não paramétrico))**

O teste de Tukey exige um objeto com formato aov:

```
> b <- aov(a~Fator1)
> TukeyHSD(b)
Tukey multiple comparisons of means
 95% family-wise confidence level
```

```
Fit: aov(formula = a ~ Fator1)
```

```
$`Fator1`
      diff      lwr      upr    p adj
2-1  8.347145  6.728374  9.965916 0.0e+00
3-1 14.177751 12.558980 15.796522 0.0e+00
3-2  5.830607  4.211836  7.449378 1.5e-06
```

**Já para o teste de Mann-Whitney, a função é `pairwise.wilcox.test`:**

```
> pairwise.wilcox.test(a,Fator1,p.adjust.method =
"bonferroni")
```

Pairwise comparisons using Wilcoxon rank sum test

```
data: a and Fator1
```

```
1      2
2 0.024 -
3 0.024 0.024
```

```
P value adjustment method: bonferroni
# procure em ?p.adjust para mais métodos de correção dos
valores de p
```

### ANOVA de 2 vias

Na ANOVA de 2 vias há 2 fatores e todas as combinações de níveis dos dois fatores devem estar presentes. Para isso vamos modificar o Fator 1 e criar um segundo fator (Fator 2):

```
> Fator1 <- c(rep("Alto",6),rep("Baixo",6))
> Fator1 <- factor(Fator1)
> Fator1
[1] Alto Alto Alto Alto Alto Alto Baixo Baixo Baixo
Baixo Baixo Baixo
Levels: Alto Baixo
```

```
> Fator2 <- rep(c("pequeno","grande"),6)
> Fator2 <- factor(Fator2)
> Fator2
[1] pequeno grande pequeno grande pequeno grande
pequeno grande pequeno grande
[11] pequeno grande
Levels: grande pequeno
5
# observe a combinação dos níveis:
> cbind(Fator1, Fator2)
> set.seed(1234)
> a <- c(rnorm(6,mean=2,sd=1),rnorm(6,mean=10,sd=1))
```

```
> modelo3 <- lm(a~Fator1 + Fator2 + Fator1:Fator2)
> library(car)# carrega o pacote car
> anova(modelo3)
```

Analysis of Variance Table

Response: a

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
Fator1	1	170.286	170.286	171.9035	1.089e-06	***
Fator2	1	0.602	0.602	0.6076	0.4581	
Fator1:Fator2	1	0.092	0.092	0.0928	0.7685	



```

Residuals      8      7.925      0.991
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

### Gráficos de interação

```

>Fator1 <- ordered(Fator1,levels=c("Baixo","Alto"))
>interaction.plot(Fator1,Fator2,a,cex.axis=2,cex.lab=2,lwd
=2,col=c("blue","red"),lty=c(2,1),legend="FALSE")
>legend("bottomright",c("pequeno","grande"),bty="n",lty=c(
2,1),lwd=2,col=c("blue","red"), title="Fator2",cex=2)

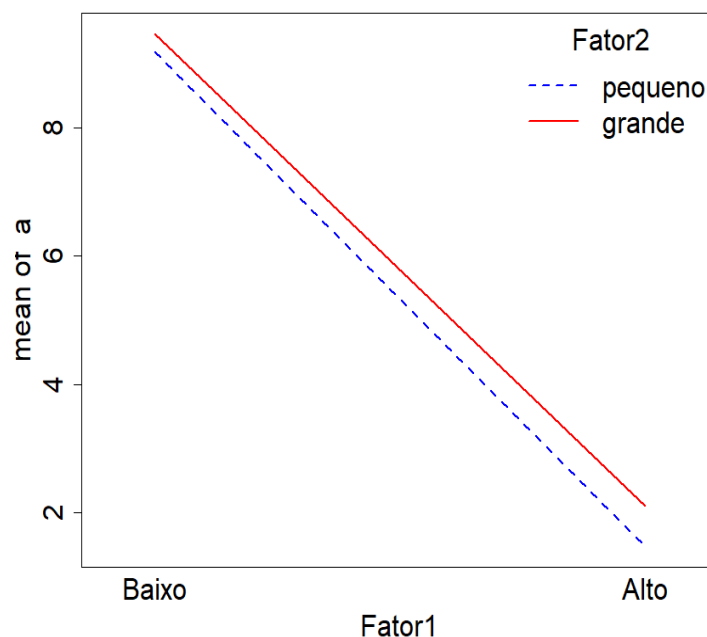
```

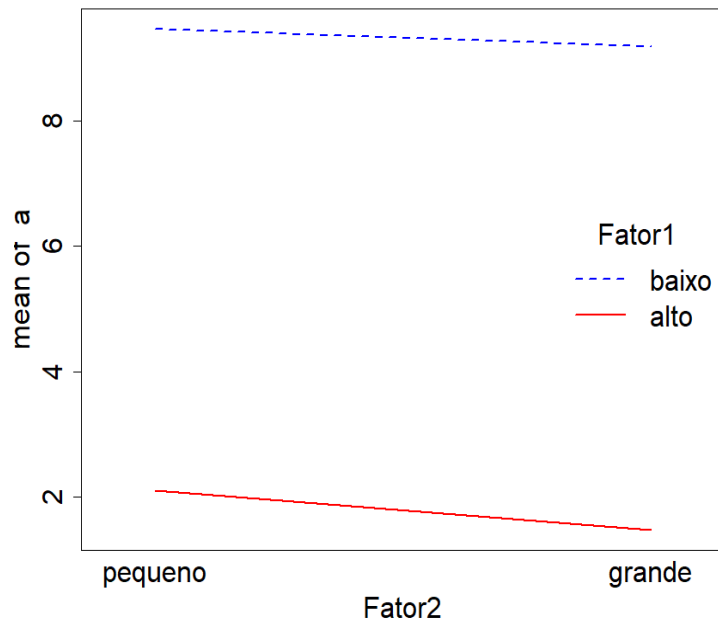
```

>Fator2<-ordered(Fator2,levels=c("pequeno","grande"))
>interaction.plot(Fator2,Fator1,a,cex.axis=2,cex.lab=2,lwd
=2,col=c("blue","red"),legend="FALSE")
>legend("right",c("baixo","alto"),bty="n",lty=c(2,1),lwd=2
,col=c("blue","red"), title="Fator1",cex=2)

```

# os níveis de variáveis categóricas são por default colocados em ordem alfabética. A função ordered coloca os níveis na ordem que for mais lógica ou conveniente:  
# plota os dois gráficos de interação:





Vamos agora testar os pressupostos da ANOVA, a homogeneidade de variâncias dos grupos (homocedasticidade, também com o pacote `car`) e a normalidade dos resíduos:

```
> leveneTest(a, Fator1)
Levene's Test for Homogeneity of Variance (center = median)
      Df F value Pr(>F)
group  1  0.0593 0.8125
      10

> leveneTest(a, Fator2)
Levene's Test for Homogeneity of Variance (center = median)
      Df F value Pr(>F)
group  1  2.5083 0.1443
      10

> modelo1 <- lm(a~Fator1)
> modelo1$residuals
> shapiro.test(modelo1$residuals) # teste de normalidade de Shapiro-Wilk.

Shapiro-Wilk normality test
data:  modelo1$residuals
W = 0.96142, p-value = 0.8039

> modelo2 <- lm(a~Fator2)
> modelo2$residuals
> shapiro.test(modelo2$residuals)
```

```
Shapiro-Wilk normality test
data:  modelo2$residuals
W = 0.79249, p-value = 0.007707
```

Os dois fatores apresentaram homogeneidade de variâncias, como esperado. Com relação aos resíduos, o Fator 1 apresentou homogeneidade de variâncias entre os grupos, o que não ocorreu com o Fator 2. Tente realizar alguma transformação da variável a para ajustar a normalidade dos resíduos.

## Testes de associação entre variáveis

Uma das formas de testar a associação entre duas variáveis, isto é, se quando ocorre aumento de uma ocorre aumento ou redução da outra, mas não se pressupõe nenhuma relação de causa e efeito entre elas, é a correlação. Calcular um índice de correlação no R é muito fácil com a função `cor`:

```
> a <- 1:10
> b <- 16:25
> cor(a,b)
[1] 1 # correlação perfeita. Visualize usando a função
plot.
> b[3] < -10 +incluindo um pouco de variabilidade nos
dados
> cor(a, b)
[1] 0.8408916
# observe que a correlação foi menor. Visualize novamente
usando a função plot
```

Vamos agora extrair o valor de probabilidade associado ao índice de correlação. Observe que o default é a correlação de Pearson. Para mudar o índice, use o argumento `method=("pearson", "spearman", "kendall")`.

```
> cor.test(a, b)

Pearson's product-moment correlation
data:  a and b
t = 4.3947, df = 8, p-value = 0.002303
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.4489729 0.9614700
sample estimates:
      cor
0.8408916

> cor.test(a,b,method="spearman")

Spearman's rank correlation rho
data:  a and b
```

```

S = 6, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.9636364

```

A correlação é sempre bivariada, mas podemos aplicar o teste a uma matriz ou data frame, sendo calculada então uma matriz de correlação com índices par a par. Para isso vamos carregar o conjunto de dados “Abioticos.txt”:

```

> Abioticos<-read.table("Abioticos.txt", header=TRUE,
row.names=1)
> cor(Abioticos)

```

	Temp	Sal	OD	N	P
Temp	1.00000000	0.71346946	-0.6434480	-0.3126558	0.07378633
Sal	0.71346946	1.00000000	-0.9422687	-0.4248663	-0.09584155
OD	-0.64344804	-0.94226867	1.00000000	0.4727692	0.11588103
N	-0.31265582	-0.42486629	0.4727692	1.00000000	0.44166642
P	0.07378633	-0.09584155	0.1158810	0.4416664	1.00000000

Calcule agora os valores de p. Para isto vamos precisar do pacote Hmisc.

```

> correlacao<-rcorr(as.matrix(Abioticos),type="pearson")
> correlacao
>write.table(correlacao$P,"clipboard",sep="\t")#
exportando dados para o excel. Este comando copia a tabela
de valores de p para a área de transferência. Abra uma
planilha do excel e pressione CTRL + V.
> write.table(correlacao$r,"clipboard",sep="\t")# realiza
a mesma função, mas com os valores de r

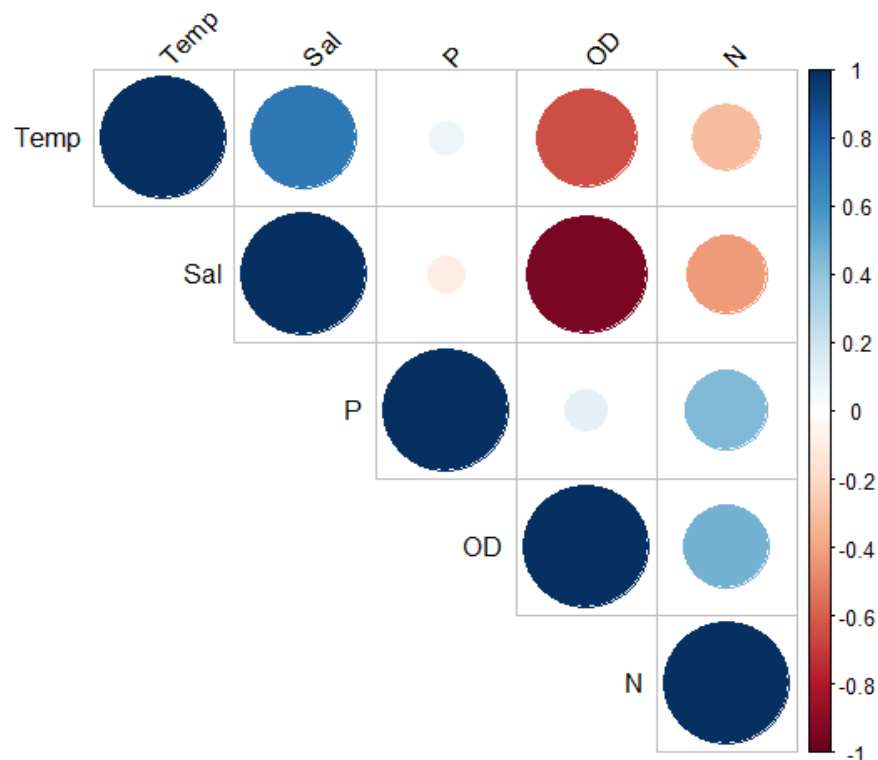
```

Uma outra ferramenta interessante para visualizar matrizes de correlação é oferecida pelo comando corrplot do pacote homônimo:

```

> library(corrplot)# caso não tenha ainda instalado,
instale-o antes de carregá-lo.
> Abioticos.cor<-cor(Abioticos)
> corrplot(Abioticos.cor, type = "upper", order =
"hclust", tl.col = "black", tl.srt = 45)
# os argumentos type="upper" mantém apenas a diagonal
superior, o tl.col="black" adiciona as letras em preto (o
default é vermelho) e o tl.srt=45 controla o ângulo dos
textos.

```



### Regressão linear

Uma das formas de se medir a associação entre duas variáveis quando se pressupõe interferência é a regressão linear simples. Para isto vamos continuar usando o conjunto de dados “Abioticos”, mas usando a função `attach` para podermos usar as variáveis diretamente:

```
> attach(Abioticos)
> modelo1<-lm(OD~Temp)# lembre-se que o til significa
“modelado por”. Neste caso, y~x (variável resposta y
modelada pela variável explanatória x).

> summary(modelo1)    #2

Call:
lm(formula = OD ~ Temp)
Residuals:
    Min       1Q   Median       3Q      Max
-1.9623 -0.4894 -0.1953  0.3451  2.2799
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  21.6042     5.6928   3.795  0.00351 **
Temp        -0.5646     0.2124  -2.658  0.02398 *
---

```

<sup>2</sup> Para entender melhor a diferença entre o  $r^2$  múltiplo e o ajustado, veja a página: <http://thestatsgeek.com/2013/10/28/r-squared-and-adjusted-r-squared/>

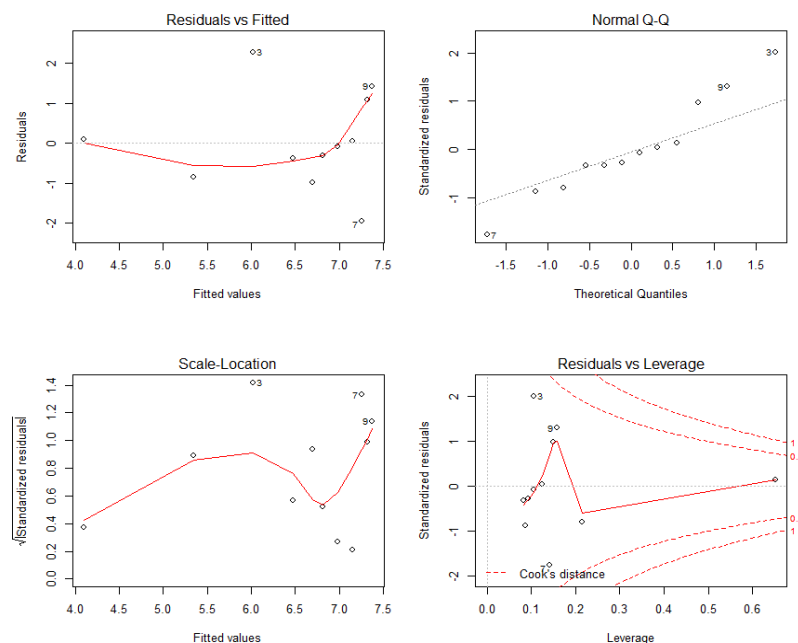
Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.198 on 10 degrees of freedom

Multiple R-squared: 0.414, Adjusted R-squared: 0.3554

F-statistic: 7.066 on 1 and 10 DF, p-value: 0.02398

```
> plot(Temp,OD) # sempre na ordem (x,y).Observe que as
variáveis estão em ordem diferente do lm().
> abline (modelo1)
> par(mfrow=c(2,2))
> plot (modelo1)# este comando irá plotar janelas em que
será possível realizar o diagnóstico da regressão3:
```



O primeiro plot (*Residuals vs fitted*) indica se há alguma tendência não linear nos dados; o ideal é que os pontos sejam distribuídos de maneira aleatória e simétrica em torno da linha tracejada horizontal 0, sem tendências a aumentar ou diminuir com os valores ajustados.

O segundo plot (*Normal Q-Q*) mostra se os resíduos são distribuídos de maneira normal; o ideal é que todos os pontos estejam sobre a linha.

O terceiro plot (*Scale-Location*) mostra a distribuição dos resíduos ao longo da amplitude da variável preditora, ou seja, a homocedasticidade. O ideal é que os resíduos estejam distribuídos de maneira uniforme em torno da linha vermelha, que deve ser horizontal. Deve-se ter cuidado com o efeito “cone” (amplitude dos resíduos aumentando ou diminuindo).

<sup>3</sup> <https://data.library.virginia.edu/diagnostic-plots/>

O último plot (*Residuals vs Leverage*) identifica os pontos que estão influenciando a regressão (outliers) para pontos extremos. Deve-se ter atenção com pontos que caíam acima dos cantos superior e inferior direito (fora da linha pontilhada mais externa), determinada pela distância de Cook.

Com base nos plots acima, o que você achou? Repita a análise e o diagnóstico da regressão realizando transformações dos dados (por exemplo, log).

É importante ressaltar que estes plots de diagnóstico são apenas visuais e servem para uma visão global dos pressupostos da regressão. Algumas análises com poucos pontos podem não ser efetivas para se determinar com segurança se a análise atende aos pressupostos ou não. Para ter um pouco mais de objetividade, vamos realizar um teste de normalidade dos resíduos, o teste de Shapiro-Wilk:

```
> modelo1$residuals # resíduos do modelo
> shapiro.test(modelo1$residuals)

Shapiro-Wilk normality test
data:  modelo1$residuals
W = 0.95442, p-value = 0.7023
# há alta probabilidade dos resíduos terem a mesma
distribuição de uma distribuição normal teórica (p =
0.7023), portanto conclui-se que a distribuição dos
resíduos é normal.
```

Este teste pode ser aplicado a uma variável também, para saber se os dados originais obedecem a uma distribuição normal:

```
> shapiro.test(Temp)
Shapiro-Wilk normality test
data:  Temp
W = 0.82945, p-value = 0.02066

> shapiro.test(OD)
Shapiro-Wilk normality test
data:  OD
W = 0.95821, p-value = 0.758
```

Vamos agora ver como é realizada uma regressão múltipla. Este tipo de análise é executado quando há uma variável resposta (Y) e várias variáveis explanatórias (X1, X2, X3...). Os comandos são muito parecidos com uma regressão linear simples, sendo a única diferença que se deve colocar todas as variáveis explanatórias com um sinal de (+) entre elas:

```
> mult1<-lm(OD~Temp+Sal+N+P)
> summary(mult1)

Call:
```

```
lm(formula = OD ~ Temp + Sal + N + P)
Residuals:
      Min       1Q   Median       3Q      Max
-1.17928 -0.13788  0.08296  0.24712  0.63636
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   9.140105    3.669674   2.491  0.04155 *
Temp          0.059603    0.156669   0.380  0.71490
Sal          -0.197072    0.037882  -5.202  0.00125 **
N             0.002432    0.003624   0.671  0.52364
P            -0.002879    0.016181  -0.178  0.86383
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6015 on 7 degrees of freedom
Multiple R-squared:  0.8965, Adjusted R-squared:  0.8374
F-statistic: 15.16 on 4 and 7 DF, p-value: 0.001474
```

Observe que a variável salinidade (Sal) é a única significativa. Vamos repetir a análise, suprimindo as variáveis não significativas, uma a uma, começando pela menos significativa (P). Este método de seleção de variáveis que irão compor o modelo linear é chamado de *stepwise backwards*:

```
> mult1<-lm(OD~Temp+Sal+N)
> summary(mult1)

Call:
lm(formula = OD ~ Temp + Sal + N)
Residuals:
      Min       1Q   Median       3Q      Max
-1.18172 -0.13435  0.06138  0.27564  0.60366
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   9.259208    3.382690   2.737  0.02556 *
Temp          0.053050    0.142764   0.372  0.71985
Sal          -0.196555    0.035410  -5.551  0.00054 ***
N             0.002137    0.003021   0.707  0.49936
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5639 on 8 degrees of freedom
Multiple R-squared:  0.8961, Adjusted R-squared:  0.8571
F-statistic: 22.99 on 3 and 8 DF, p-value: 0.000275
```

A salinidade continuou significativa, mas ainda restaram duas variáveis não significativas (*Temp* e *N*). Vamos remover a menos significativa primeiro (*Temp*):



```

> mult1 <- lm(OD~Sal+N)
> summary(mult1)

Call:
lm(formula = OD ~ Sal + N)

Residuals:
    Min       1Q   Median       3Q      Max
-1.16162 -0.17680  0.09292  0.28536  0.58464

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  10.480918   0.756635  13.852 2.25e-07 ***
Sal          -0.187670   0.024836  -7.556 3.48e-05 ***
N              0.002120   0.002872   0.738  0.479
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
' ' 1

Residual standard error: 0.5363 on 9 degrees of freedom
Multiple R-squared:  0.8943,    Adjusted R-squared:  0.8708
F-statistic: 38.06 on 2 and 9 DF, p-value: 4.063e-05

```

A salinidade ficou ainda mais significativa, mas ainda restou o nitrogênio, que não é significativo. Por último removeremos esta variável, resultando numa regressão linear simples, que já foi realizada nos exemplos anteriores.

Vamos realizar um segundo exemplo com o conjunto de dados *trees*:

```

> mult2 <- lm(Volume~Girth+Height,data=trees)
# observe que o comando data=trees é necessário para
indicar onde estão as variáveis porque não foi usado o
attach(trees).

Call:
lm(formula = Volume ~ Girth + Height, data = trees)

Residuals:
    Min       1Q   Median       3Q      Max
-6.4065 -2.6493 -0.2876  2.2003  8.4847

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -57.9877     8.6382  -6.713 2.75e-07 ***
Girth         4.7082     0.2643  17.816 < 2e-16 ***
Height        0.3393     0.1302   2.607  0.0145 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
' ' 1

Residual standard error: 3.882 on 28 degrees of freedom
Multiple R-squared:  0.948,    Adjusted R-squared:  0.9442
F-statistic: 255 on 2 and 28 DF, p-value: < 2.2e-16

```

Observe neste caso que as variáveis (*girth* e *height*) são significativas para determinar o volume das árvores. Vamos incluir um termo de interação entre elas para verificar se este também é significativo:

```
> mult3 <- lm(Volume~Girth+Height+Girth:Height,data=trees)
# outra forma de incluir o termo de interação é através de
mult3 <- lm(Volume~Girth*Height,data=trees)
```

Call:

```
lm(formula = Volume ~ Girth + Height + Girth:Height, data
= trees)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-6.5821	-1.0673	0.3026	1.5641	4.6649

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	69.39632	23.83575	2.911	0.00713	**
Girth	-5.85585	1.92134	-3.048	0.00511	**
Height	-1.29708	0.30984	-4.186	0.00027	***
Girth:Height	0.13465	0.02438	5.524	7.48e-06	***

---  
Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
Residual standard error: 2.709 on 27 degrees of freedom  
Multiple R-squared: 0.9756, Adjusted R-squared: 0.9728  
F-statistic: 359.3 on 3 and 27 DF, p-value: < 2.2e-16

Observe que o termo de interação é muito significativo, porém há um incremento grande em complexidade do modelo. Pelo princípio da parcimônia só se justifica manter um modelo mais complexo caso ele resulte em incrementos significativos em explicação. Vamos testar se há diferença significativa entre os modelos:

```
> anova(mult2, mult3)
```

Analysis of Variance Table

	Model	Res.Df	RSS	Df	Sum of Sq	F	Pr(>F)
1	Volume ~ Girth + Height	28	421.92				
2	Volume ~ Girth + Height + Girth:Height	27	198.08	1	223.84	30.512	7.484e-06 ***

---  
Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Os dois modelos são significativamente diferentes, portanto, justifica-se adotar o modelo mais complexo contendo a interação.

## Estatística Multivariada: análise de cluster e métodos ordenação

Enquanto a estatística uni e bivariada analisa como uma ou diversas variáveis independentes (x) influenciam uma única variável dependente (y), na multivariada podemos também ter diferentes variáveis resposta (vários Y). Nestes casos, a estatística normalmente trabalha no ordenamento em 2 ou 3 dimensões das variáveis com maior variância (como no caso da análise de componentes principais; PCA). Esta redução à bidimensionalidade normalmente acarreta perda de informação; mas, por outro, simplifica tanto à n-dimensionalidade quanto à interpretação correta dos gráficos.

Dentre uma gama de possibilidades, devido à proposta introdutória deste documento, iremos nos focar em apenas três tipos de análises multivariadas: cluster, análise de componentes principais e ordenamento multidimensional não métrico. O foco, entretanto, será nos comandos do R e no produto gráfico gerado; e não no mérito lógico da estatística, o qual deixaremos para outra oportunidade.

### Análise de agrupamento (cluster)

Uma das análises multivariadas exploratórias mais simples é a análise de agrupamento, que cria um dendrograma de relações entre as unidades amostrais, baseado na similaridade ou dissimilaridade (distância) entre elas. Existem inúmeras medidas de similaridade e dissimilaridade (índices ecológicos), como por exemplo Jaccard, Bray-Curtis, distância Euclidiana, distância de Hellinger etc. A discussão sobre critérios de seleção de qual o melhor índice está além dos objetivos desta apostila; aqui iremos apresentar uma demonstração de uma aplicação, lembrando sempre que o usuário tem grande liberdade para ajustar todos os parâmetros ideais para seu caso.

Vamos trabalhar com um novo conjunto de dados: análise de *fingerprinting* (*Temporal Temperature Gradient Gel Electrophoresis*) de bactérias planctônicas na barra do estuário da Lagoa dos Patos ao longo de um ano (junho de 2010 a maio de 2011). Durante o período houve uma transição de um evento de El Niño para um de La Niña, fenômeno este que afeta drasticamente a salinidade do estuário. Uma vez que a composição das bactérias é fortemente afetada pela salinidade, a hipótese é de que a composição da comunidade bacteriana irá apresentar variações no período. Para ter acesso ao banco de dados, abra a planilha "Cluster\_TTGE\_Barra" e copie para a área de transferência os dados, carregando no R:

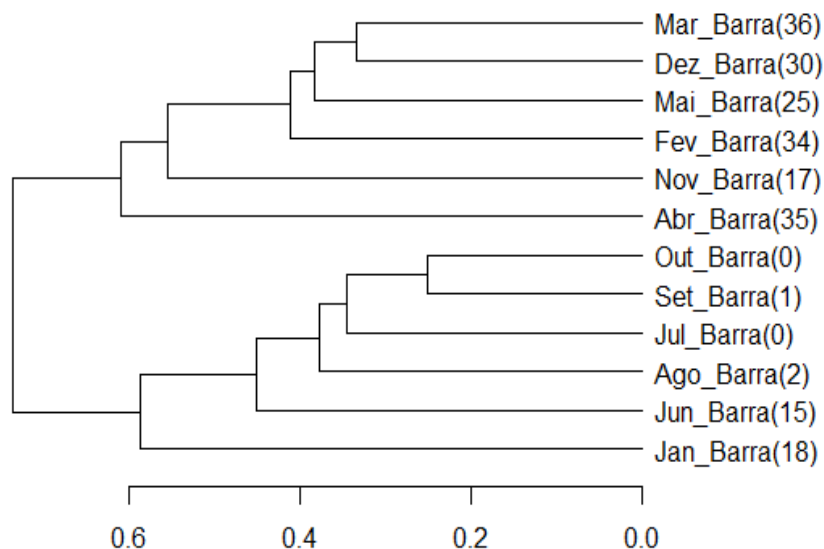
```
> TTGE <- read.table("clipboard", row.names=1, header=F)
> TTGE
# observe a estrutura dos dados. Note que os dados são de
# presença/ausência para cada Unidade Taxonômica
# Operacional (UTO) bacteriana

> TTGE.frame<-as.data.frame(TTGE)
# carregue o pacote "vegan"
```

```

> TTGE.frame.bray<-vegdist(TTGE.frame,"bray")
# vegdist calcula a matriz de distâncias entre unidades
amostrais usando a distância de Bray-Curtis. Observe a
estrutura da matriz de distâncias
> TTGE.frame.bray >TTGE.frame.bray.clust<-
hclust(TTGE.frame.bray,method="average")
# hclust seleciona o método de agrupamento, no caso
"average" é o UPGMA (Unweighted Pair Group Method with
Arithmetic Mean).
> par(mar=c(5,5,5,10))
> TTGE.dendrogram<-as.dendrogram(TTGE.frame.bray.clust)
> plot(TTGE.dendrogram, horiz=T)
# consulte os comandos vegdist para mais opções de
matrizes de distância e hclust para mais opções de
métodos de agrupamento.

```



A hipótese parece consistente? Observe que foram formados dois grandes grupos, um com amostras de junho até outubro de 2010 e janeiro de 2011 e outra com amostras de novembro a dezembro de 2010 e fevereiro a maio de 2011. Com exceção da amostra de janeiro, o padrão parece bem plausível, pois as amostras do primeiro período possuem salinidade menor (0-18), enquanto que a do segundo grupo salinidade maior (17-36), com pouca sobreposição.

No entanto, podemos melhorar um pouco o layout do gráfico:

```

> labels<-c(" "," "," "," "," "," "," "," "," "," "," "," "," "," "," "," ")
">
> TTGE.frame <- as.data.frame(TTGE,row.names=labels)
> TTGE.frame.bray <- vegdist(TTGE.frame,"bray")
> TTGE.frame.bray.clust <- hclust(TTGE.frame.bray,
method="ave>rage")
> TTGE.dendrogram<-as.dendrogram(TTGE.frame.bray.clust)

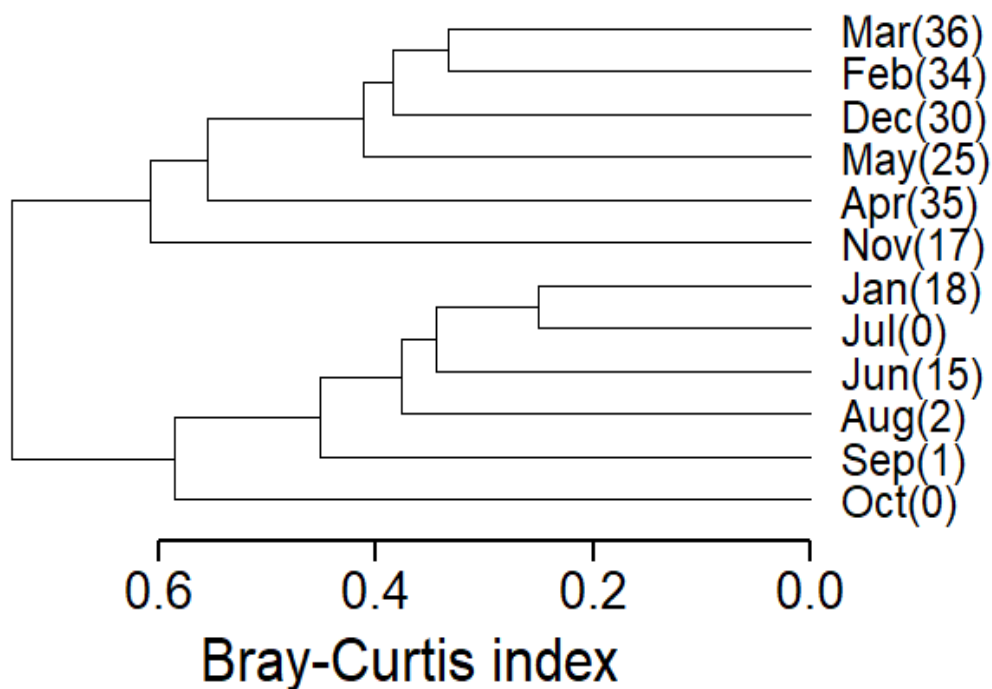
```

```

> TTGE.labels <-
c("Oct(0)", "Sep(1)", "Aug(2)", "Jun(15)", "Jul(0)", "Jan(18)",
"Nov(17)", "Apr(35)", "May(25)", "Dec(30)", "Feb(34)", "Mar(36)
")
> par(mar=c(5, 5, 5, 6))
> plot(TTGE.dendrogram, axes=F, lwd=3, horiz=T)
> axis(1, lwd=2, cex.axis=1.75)
> axis(4, at=1:12, lab=TTGE.labels,
cex.axis=1.5, tick=F, line=-1, las=2)
> mtext("Bray-Curtis index", side=1, line=3, cex=2)

```

#usamos um artifício para remover os rótulos originais dos dados para customizá-los com o comando TTGE.labels. Suprimimos os eixos no comando plot colocando axes=F e pedimos para o cluster ser plotado de maneira horizontal e não vertical através de horiz=T (horiz=V é o default). Os eixos foram posteriormente colocados um a um com os comandos axis e o rótulo do eixo x foi adicionado através do comando mtext indicando o índice de similaridade usado.

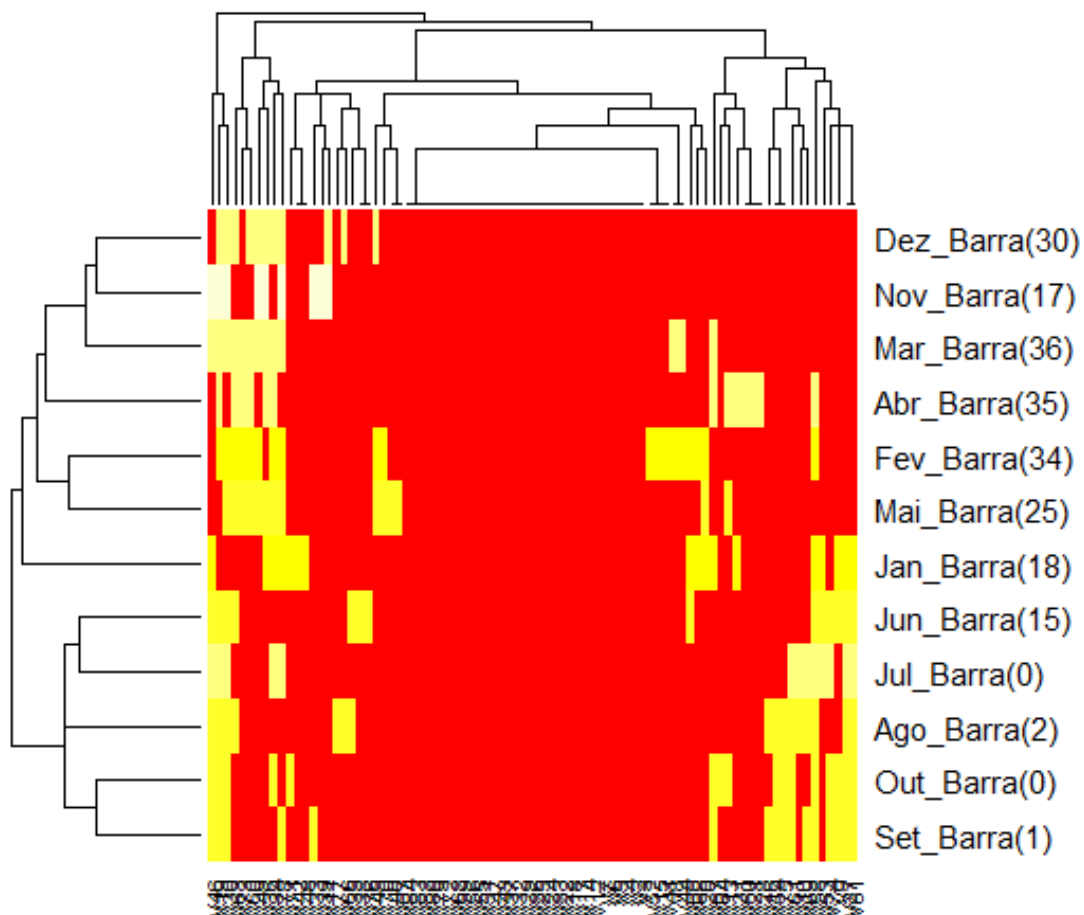


Um outro recurso interessante de análise de cluster é o `heatmap`, que plota dois dendrogramas combinados, um por unidades amostrais (à direita) e outro por variáveis (em cima), produzindo uma escala relativa com cores (*false color image*) indicando a média das linhas (unidades amostrais) e nas colunas (variáveis no centro).

Para isto vamos utilizar o data frame e forçar que seja lido como matriz através

do comando `as.matrix`. Uma série de parâmetros pode ser controlada através de argumentos dentro da função `heatmap` (por exemplo a ordem em que são apresentados), mas aqui vamos apenas indicar a distância de Bray-Curtis:

```
> heatmap(as.matrix(TTGE.frame, dist="bray") )
```



## Análise de Componentes Principais

A Análise de Componentes Principais (PCA) é uma das formas mais simples e diretas de ordenar variáveis em um espaço bidimensional. O uso primário da PCA é reduzir a dimensionalidade de dados multivariados. Em outras palavras, nós usamos a PCA para criar algumas variáveis-chave que caracterizam provavelmente toda a variação dos dados. O preceito básico é que ao transpor as variáveis a dois eixos arbitrários, eles sempre serão ortogonais entre si, ou seja, não são correlacionados entre si (independentes). Os eixos estão ordenados segundo a variância (inércia) dos dados: o eixo 1 possui a maior parte da variância dos dados, enquanto o eixo 2 a segunda e assim sucessivamente.

Vamos agora apresentar um dos modos de se fazer uma PCA no R, bem como plotar o resultado. Como exemplo, vamos utilizar um conjunto de dados oriundos do trabalho de MacLaren et al. (2017) sobre a disparidade das mandíbulas de

dinossauros herbívoros.

Primeiramente, como todos os casos, vamos fazer o upload dos dados para o R. Vamos seguir o comando padrão, muito embora existam diversos modos de se fazer este passo. Nesta altura vocês já podem tomar a decisão sobre quais os comandos preferem para fazer a inserção de dados no R.

```
> myd <- read.csv("MacLaren.csv", na.strings="?") # notem  
o "?" ao invés do ";"
```

Como os dados não são conhecidos da maioria dos leitores, sempre é importante fazermos esta primeira análise que chamamos: conhecendo a estrutura dos dados. No R, estes são alguns dos comandos, os quais já foram previamente apresentados:

```
> str(myd)  
> dim(myd)  
> head(myd)  
> summary(myd)
```

Neste conjunto de dados, temos medidas morfométricas da mandíbula de diversas espécies de dinossauros herbívoros. No total, temos 167 observações. Como citado, o objetivo primário da PCA é reduzir estas variáveis (medidas morfométricas) dentro de um espaço bidimensional, ordenando de acordo com a maior variância destas medidas. Na PCA, que usa distância euclidiana, é importante que todas as medidas estejam na mesma unidade, como é o caso deste exemplo.

O primeiro passo é colocarmos em um novo dataframe apenas as colunas com as observações dos caracteres morfológicos das mandíbulas dos dinossauros. Um exemplo:

```
> data <- myd[, 4:14] # 15 variáveis morfométricas  
> group <- factor(myd$Higher.Taxonomy)  
#grupos de dinossauros
```

Para reduzir a alta variação dos dados, vamos antes de realizar a PCA, utilizar um método de transformação dos dados:

```
> pca <- decostand(na.omit(data), method = "standardize")  
# note que excluímos os dados que estão assinalados como  
NAs  
# veja mais na ajuda da função decostand do pacote  
"vegan".
```

Existem diversos modos de se fazer uma PCA no R. Apresentamos aqui uma destas possibilidades. Vocês irão perceber que, embora a PCA seja uma análise simples e direta, é necessário certo conhecimento de estatística para extrair os

resultados corretos no R. O primeiro comando já requer isto:

```
> r2pca <- princomp(x=pca, cor=F, scores=T)
# Este é um comando para uma PCA baseado em uma matriz de
covariância (a mais usual). Por isto, o argumento "cor="
foi assinalado como "F" (False). "cor=T" retornaria uma
PCA baseada em uma matriz de correlação.
```

A PCA retorna autovetores (mais relacionados à direção) e autovalores (relacionados à variância). Deste modo que estamos ilustrando a PCA, precisamos extrair estes resultados (veja a importância da manipulação de dados no R):

```
> r2vectors <- r2pca$loadings # autovetores
> r2values <- r2pca$sdev^2     # autovalores
> r2scores <- r2pca$scores    # escores dos eixos
```

Podemos também ver os eixos da PCA:

```
> head(r2pca$scores)
```

Existe um comando para centralizar os eixos em torno do zero:

```
> PCscores <- scale(r2scores)
```

Agora, vamos transformar os escores dos autovalores dos três primeiros eixos em porcentagem e de forma concomitante, utilizar um modo diferente de dar nome aos eixos:

```
> xlab=paste("PC1", 100*round(r2values[1]/sum(r2values), 3), "%", sep="")
> ylab=paste("PC2", 100*round(r2values[2]/sum(r2values), 3), "%", sep="")
> zlab=paste("PC3", 100*round(r2values[3]/sum(r2values), 3), "%", sep="")
```

Vamos, antes de plotar a PCA, assinalar a um vetor com os nomes dos grupos de dinossauros:

```
> name_groups <- c("Ornithischia", "Marginocephalia",
"Ornithopoda", "Sauropodomorph", "Thyreophora")
```

Agora sim, vamos plotar a PCA:

```
> op <- par(mfcol=c(1,1), mar=c(4,4,2,1), mgp=c(2.5,
0.75, 0))
> plot(PCscores[,1],PCscores[,2], xlab=xlab, ylab=ylab,
cex=1.6, pch=16, col=c("red", "blue3", "orange3",
"grey75", "green")[group], ylim=c(min(PCscores[,1]),
```

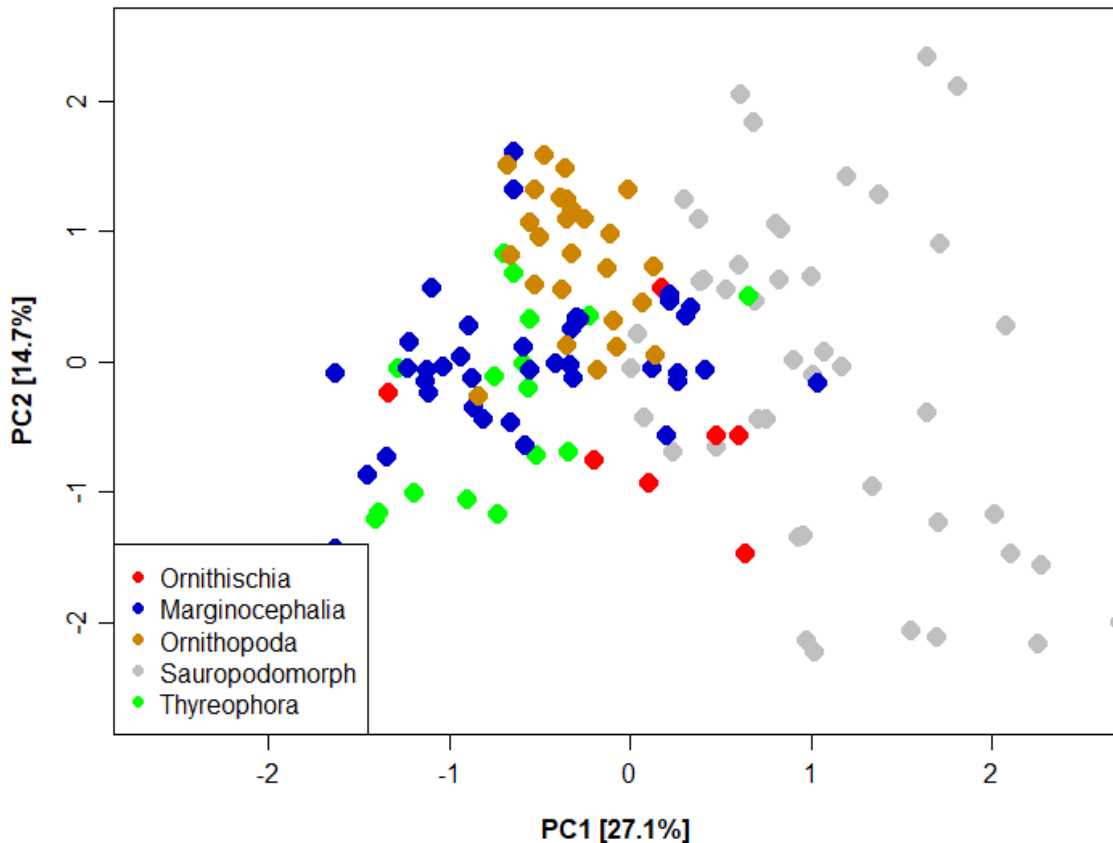


```

max(PCscores[,1])), font.lab=2)
> legend("bottomleft", name_groups, col=c("red", "blue3",
"orange3", "grey75", "green"), text.col = "black",
pch=16, cex=1.0, xpd=T)
> op(par)

```

O resultado gráfico será este:



De forma bem resumida, percebam que o eixo 1, aquele que possui a maior parte da variação dos dados, representa 27% da variância, seguido pelo eixo 2 (14.7%). Como exercício, tentam plotar o eixo 1 (x) versus o eixo 3 (y). Como ficou o resultado?

Aqueles que já viram uma PCA antes podem estranhar o formato do gráfico. Uma PCA tradicional poderia ser plotada com o comando abaixo, seguido do resultado gráfico:

```
> biplot(r2pca) # confira o resultado
```

Não obstante, existem ainda outros modos de plotar uma PCA e outras formas de realizar esta análise no R. Como estamos sistematicamente enfatizando, aqui são apresentadas algumas operações apenas como uma primeira introdução à linguagem e aos comandos básicos, sem detalhar as idiosincrasias de cada análise, bem como as possíveis derivações. Por favor, para tópicos mais avançados consulte outros

documentos e bibliografias listadas nas referências.

## **Escalonamento multidimensional não métrico**

O escalonamento multidimensional não métrico (MDS, também NMDS e NMS) é uma técnica de ordenação que difere de quase todos os outros métodos de ordenação (como o que vimos anteriormente). Na maioria dos métodos de ordenação, muitos eixos são calculados, mas apenas alguns são visualizados, devido a limitações gráficas. Em contrapartida, no NMDS, um pequeno número de eixos é explicitamente escolhido antes da análise (observe isto nos argumentos da função que executa o NMDS abaixo). Segundo a maioria dos outros métodos de ordenação são analíticos e, portanto, resultam em solução única para um determinado conjunto de dados. Em contraste, o NMDS é uma técnica numérica que busca uma solução de forma iterativa e a computação é finalizada quando uma solução aceitável for encontrada, ou depois de um número de tentativas, que também é explicitado a priori (também através de um argumento específico, no caso do R). Como resultado, o NMDS não é uma solução única. Uma análise subsequente do mesmo conjunto de dados seguindo a mesma metodologia provavelmente irá resultar em uma ordenação um pouco diferente. Isto pode ser mais evidente em conjunto de dados com um menor número de observações.

Ademais, o NMDS não produz nem um autovetor nem um autovalor como análise na análise de componentes principais (PCA); ou ainda como na análise de correspondência (CA) que ordena os dados de tal forma que o eixo 1 explica a maior parte variância, enquanto o eixo 2 explica a segunda maior variância e assim subsequentemente. Como resultado, a ordenação NMDS pode ser girada, invertida ou centrada em qualquer configuração desejada.

O NMDS tem como rotina encontrar o resultado mais simples possível em termos de ordenação. Além disto, é possível a utilização de qualquer índice de associação (Bray-Curtis, Euclidean, Manhattan etc.). O NMDS não gera um valor probabilístico (como um valor de  $p$ ), mas apenas um valor de STRESS, oriundo da soma residual dos quadrados da diferença entre os valores reais em relação àqueles gerados pelo modelo linear mais simples encontrado. Não existe um valor limítrofe de STRESS; mas, quanto menor o valor de STRESS, mais próxima está a relação entre as distâncias reais e as diferenças no NMDS. Vamos tentar visualizar um pouco disto no R, e aproveitar para aprender os comandos neste tipo de análise. Para tal, vamos voltar aos dados dos insetos (objeto chamado “meusdados”) para ilustrar como realizar uma análise de ordenamento nMDS, bem como o gráfico resultante. Vamos tentar explorar diversos tipos de manipulação de dados antes, como uma forma de revisarmos alguns comandos. Antes de tudo, vamos lembrar quais variáveis estes dados tinham:

```
> head(meusdados)
> dim(meusdados)
[1] 101  35
```

Basicamente, existem colunas de dados ambientais (categorias) e variáveis relacionadas às espécies de insetos (abundância). Podemos separar estas colunas em dois subconjuntos, um representado apenas as espécies e outros as demais informações:

```
> sp <- meusdados[, 5:34]
> var_amb <- meusdados[, 1:4]
```

Arbitrariamente, ainda, poderíamos excluir algumas linhas e colunas que apresentam poucas observações ou poucas espécies, respectivamente:

```
> n <- 10
> o <- 2
```

Podemos, então, excluir este conjunto de dados (isto deve ser feito para os dois subconjuntos):

```
> X <- which(colSums(sp>0)<=o)
> Y <- which(rowSums(sp)<=n)

> Data1 <- sp[-c(Y), -c(X)]
> grupos <- var_amb[-c(Y), ]
```

Vamos verificar com quantas linhas (observações) e com quantas variáveis nossos dados ficaram após estas exclusões. Quantas foram excluídas?

```
> dim(meusdados) - dim(Data1)
```

Muitas vezes, antes da análise propriamente dita, pode ser interessante transformarmos nossos dados. Aqui segue uma modificação chamada de dupla transformação. Para saber mais leia a ajuda do comando `deconstand()`.

```
> Data2 <- decostand(Data1, "max") #divide todos os
valores pelo valor máximo encontrado em uma linha
> NMDSdata <- decostand(Data2, "total") #divide todos os
valores pelo valor máximo encontrado nas colunas
```

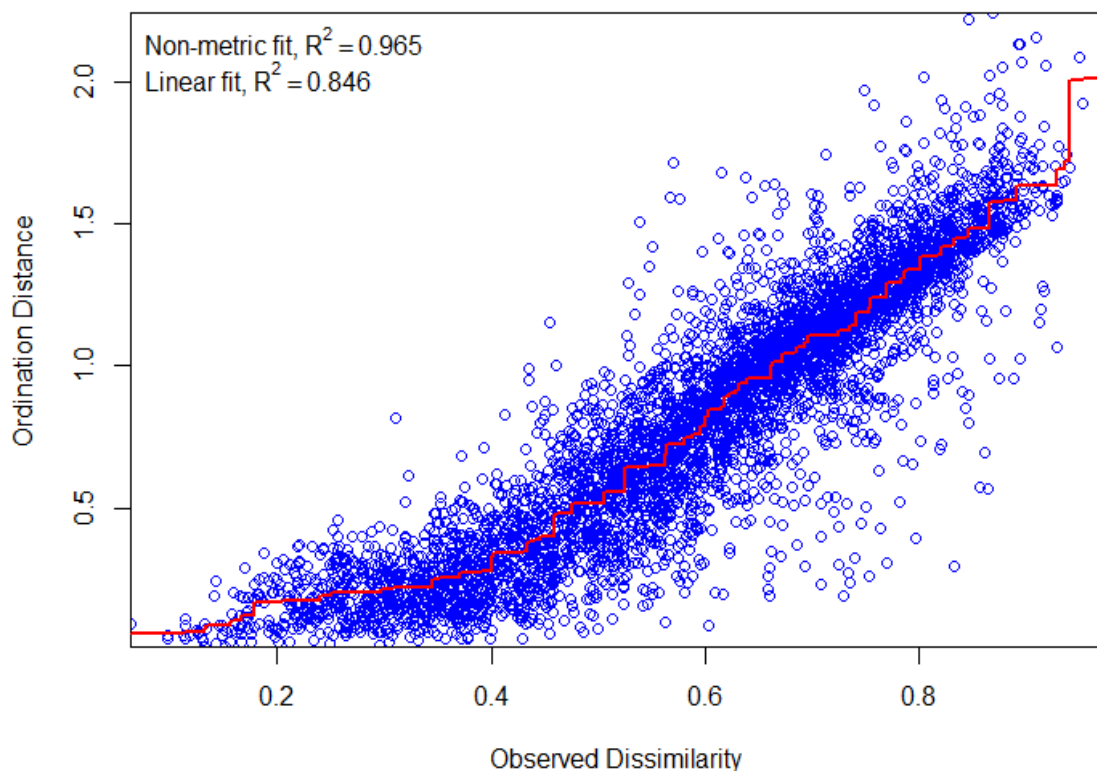
A execução de um nMDS é variável também. Particularmente, preferimos realizar em dois passos. Rodar uma primeira análise e depois novamente sobre os melhores resultados da anterior. Isto não é, todavia, muito comum, mas sim uma particularidade que varia de cientista para cientista.

```
> initial_nMDS1 <- metaMDS(NMDSdata, distance="bray",
k=2, trymax=100)
> nMDS1 <- metaMDS(NMDSdata, previous.best =
initial_nMDS1, k=2, trymax=100)
# Como informamos anteriormente, é possível informar ao R
tanto quantas dimensões queremos (k=), como também
```

quantas vezes ele irá procurar pelo resultado mais simples (`trymax=`). Teoricamente, quanto mais dimensões (eixos) pedirmos, menor o valor de STRESS. Lembrando que apenas dois eixos serão ilustrados. Portanto, deixe claro ao leitor quantos eixos foram arbitrariamente escolhidos.

Antes de plotar o resultado final do NMDS, vamos ter uma ideia básica de como é calculado o valor de STRESS, em termos gráficos. Utilize o comando abaixo e observe a relação entre os dados observados e aqueles oriundos da ordenação:

```
> scarplott(nMDS1)
```



Tente aumentar arbitrariamente tanto o número de dimensões ( $k$ ) quanto de número de interações (`trymax`) e observe como o `stressplot` pode variar, refletindo no valor de STRESS do NMDS.

Agora, finalmente, podemos plotar o resultado gráfico do nMDS. Segue um exemplo de como fazer isto:

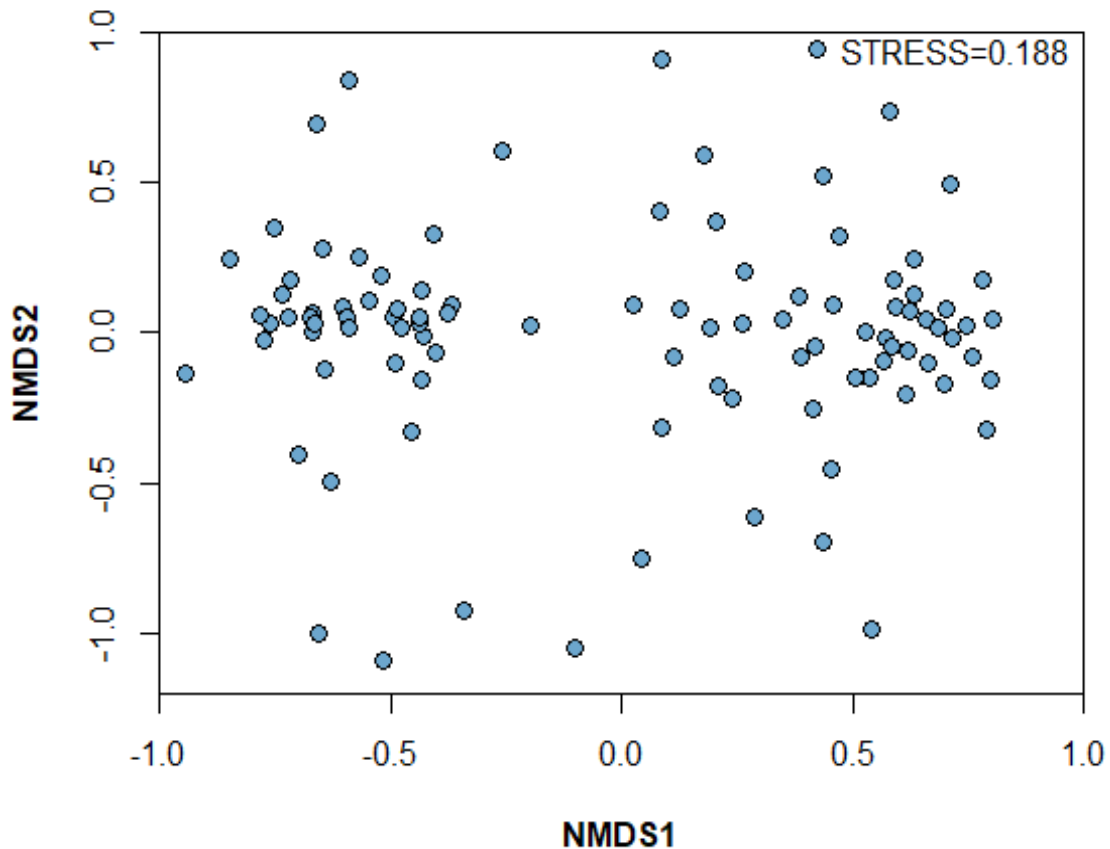
```
> op <- par(mar=c(4,4,1,1))
> plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2,
pch=21, bg="skyblue3", xlab='NMDS1', ylab='NMDS2',
cex=1.6, main="", xlim=c(-1, 1), ylim=c(-1.2, 1))
```

```

mtext(side=3, line=-1, adj=0.98,
paste('STRESS=',round(nMDS1$stress, 3),sep=''), cex=1.0)
> par(op)

```

O resultado gráfico será aproximadamente o seguinte:



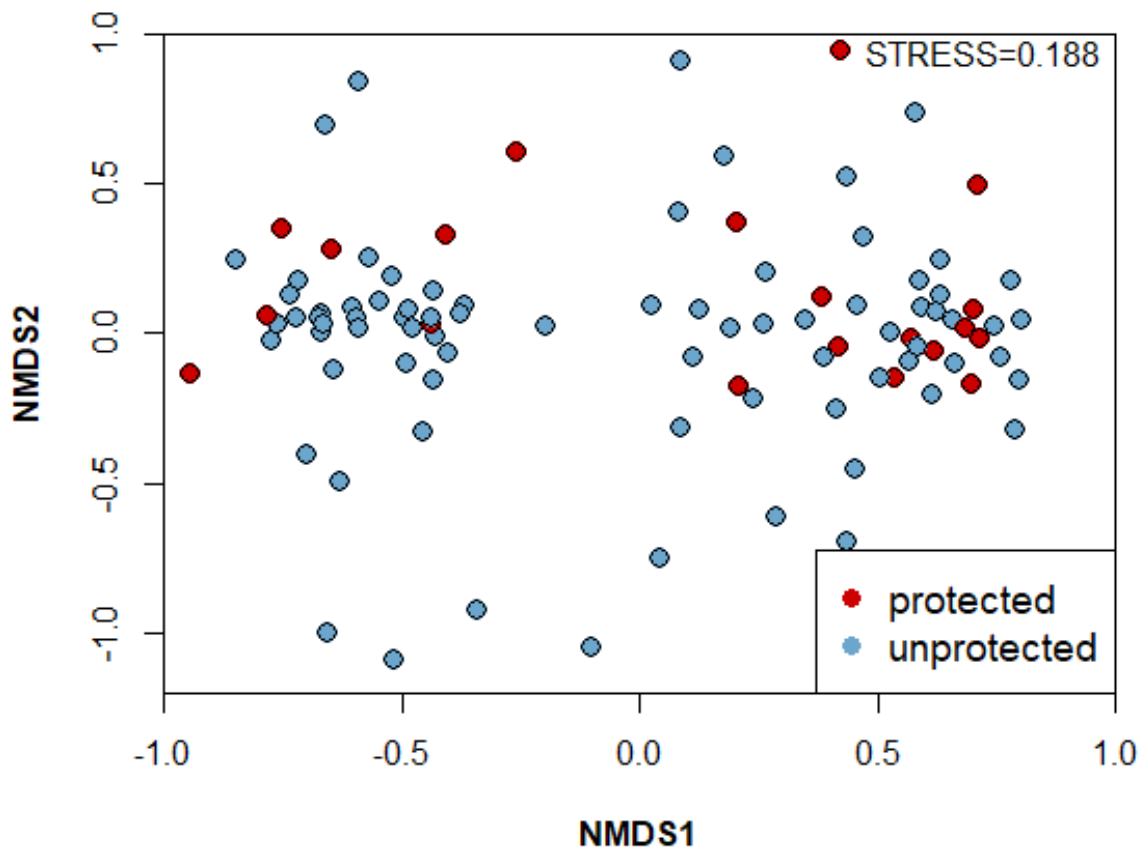
Um recurso interessante em gráficos de ordenamento é colorir cada observação conforme algum fator de interesse. Aqui, poderíamos colorir em relação ao status dos locais de coleta em termos de áreas protegidas e não protegidas. Isto é facilmente possível manipulando argumentos que já foram inseridos neste gráfico, como veremos a seguir. Dica: o R sempre coloca vetores de caracteres em ordem alfabética.

```

> op <- par(mar=c(4,4,1,1))
plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2,
pch=21, bg=c("red3", "skyblue3")[grupos$status],
xlab='NMDS1', ylab='NMDS2', cex=1.4, main="", xlim=c(-1,
1), ylim=c(-1.2, 1))
> mtext(side=3, line=-1, adj=0.98,
paste('STRESS=',round(nMDS1$stress, 3),sep=''), cex=1.0)
> legend("bottomright", c("protected", "unprotected"),
col=c('red3', "skyblue3"),
text.col = "black", pch=16, cex=1.2, xpd=T)

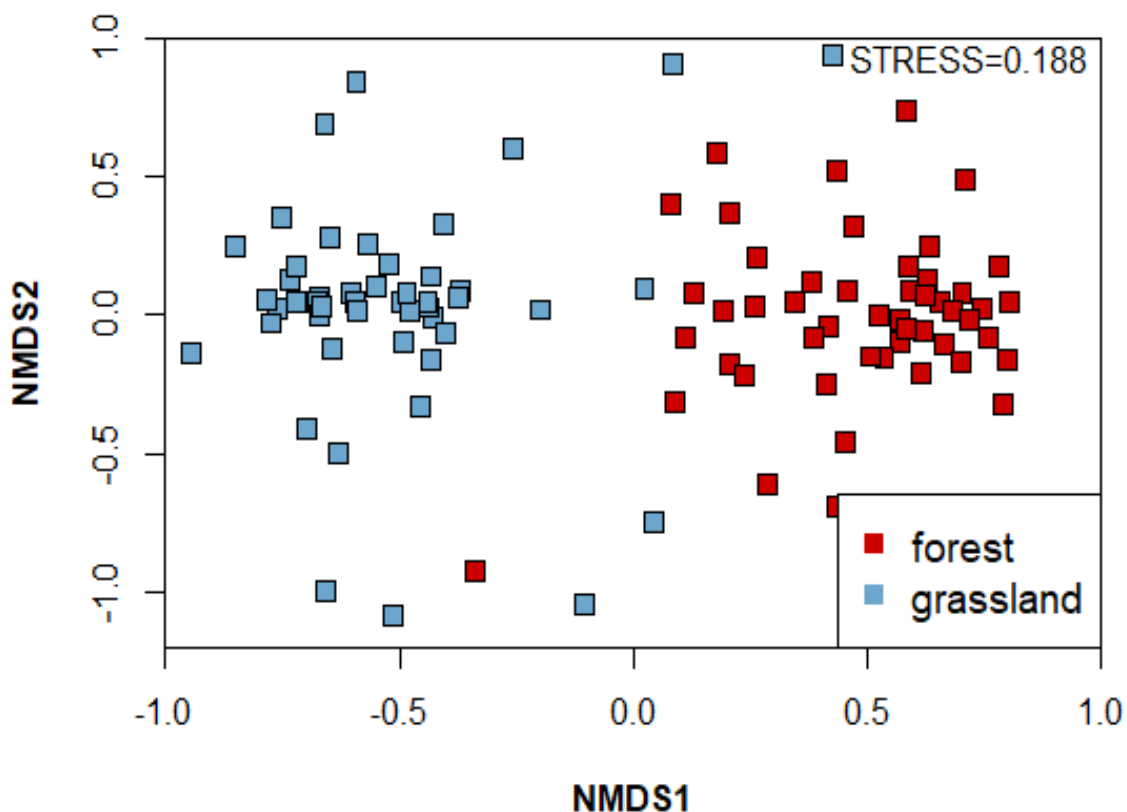
```

Observe que adicionamos um comando de legendas. O resultado final será provavelmente este:



Agora, podemos tentar com outros fatores, tal como o hábitat (*forest*, *grassland*). Vamos adicionar alguns outros argumentos possíveis para incrementar o gráfico. Veja os comandos abaixo.

```
> op <- par(mar=c(4,4,1,1))
> plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2,
pch=22, bg=c("red3", "skyblue3")[grupos$habitat],
xlab='NMDS1', ylab='NMDS2', cex=1.4, main="", xlim=c(-1,
1), ylim=c(-1, 1))
> mtext(side=3, line=-1, adj=0.98,
paste('STRESS=', round(nMDS1$stress, 3), sep=''), cex=1.0)
> legend("bottomright", c("forest", "grassland"),
col=c('red3', "skyblue3"), text.col = "black", pch=17,
cex=1.2, xpd=T)
```



É possível visualizar que as espécies parecem se distribuir de forma relativamente padronizada entre os dois habitats (ao contrário do fato de a área ser protegida ou não; gráfico anterior). Para saber se estes grupos são diferentes, precisamos realizar um teste estatístico multivariado para verificar se a hipótese nula é verdadeira ou falsa. Um teste muito utilizado e válido neste caso é a Análise de variância com permutação (PERMANOVA, na sigla em inglês). A função que realiza a PERMANOVA é a `adonis()`, e faz parte do pacote “vegan”. Veja os comandos abaixo:

```
> permanova <- adonis(NMDSdata ~ grupos$habitat,
permutations=999, distance='bray')
> permanova      # confira os resultados
> str(permanova) # observe como os resultados estão
otaginizados
```

Um ponto importante no R é extrair de uma análise apenas as informações que sejam importantes naquele momento, as quais podemos ainda adicionar no gráfico. Por exemplo, para extrair o valor de  $F$  e o valor de  $p$  da PERMANOVA, poderíamos utilizar os seguintes comandos:

```
> obs.F <- permanova$aov.tab$F.Model[1] # valor de F
> p.value <- permanova$aov.tab$Pr[1]    # valor de p
```

Com isto, podemos adicionar alguns novos argumentos ao gráfico. Veja o resultado destes comandos:

```
> op <- par(mar=c(4,4,2,1))
> plot(nMDS1$points[,1], nMDS1$points[,2], font.lab=2,
pch=22, bg=c("red3", "skyblue3")[grupos$habitat],
xlab='NMDS1', ylab='NMDS2', cex=1.4, main="", xlim=c(-1,
1), ylim=c(-1.2, 1))
> ordihull(nMDS1,
groups=grupos$habitat, draw="polygon", col=c("grey90",
"grey75")[grupos$habitat], label=F)
> mtext(side=3, line=-1, adj=0.98,
paste('STRESS=', round(nMDS1$stress, 3), sep=''), cex=1.0)
> mtext(side=3, line=0.2, adj=0.5, paste('PERMANOVA, p=',
round(p.value, 3), ', F=', round(obs.F, 3), sep=''),
cex=1.0)
> legend("bottomright", c("forest",
"grassland"), col=c("red3", "skyblue3"),
text.col = "black", pch=17, cex=1.2, xpd=T)
> par(op)
```

O que você encontrou?

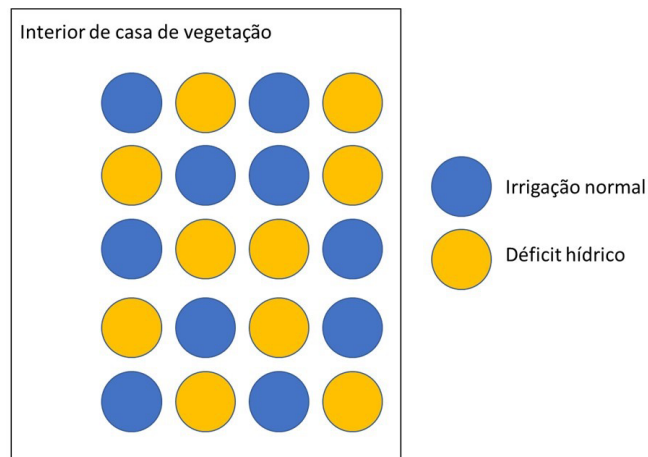
## **Análise de experimentos biológicos constituídos segundo delineamentos com níveis de complexidade crescentes**

### **Análise de experimentos em DIC e DBC: a diferença do fator bloco**

Antes de partirmos para as análises em R *per se* sobre este tópico, vamos fazer uma breve revisão sobre dois delineamentos experimentais muito comuns em experimentos biológicos. Seja um experimento em que as unidades experimentais (vasos, plantas individuais, indivíduos de aves ou peixes, placas de Petri contendo meio de cultura para crescimento de fungos ou bactérias, tubos de ensaio, frascos de vidro, etc.) são uniformes e homogêneas, não existindo um procedimento que permita uma distinção destas unidades em categorias por semelhanças. Neste caso, a variação a ser inserida é o tratamento experimental que será imposto.

Vamos considerar um breve exemplo para entender este delineamento e como sua análise pode ser realizada no R. Suponha que você submete 20 mudas da espécie arbórea conhecida popularmente como timbaúva (*Enterolobium contortisiliquum*), produzidas em vasos a partir de sementes de uma mesma árvore, a dois tratamentos: 10 recebem irrigação normal (à capacidade de campo do substrato) e 10 têm sua irrigação suspendida por um período significativo, visando induzir o estresse por déficit hídrico. Os vasos são distribuídos aleatoriamente em uma casa de vegetação, cada um recebendo um dos tratamentos, conforme a figura que segue.





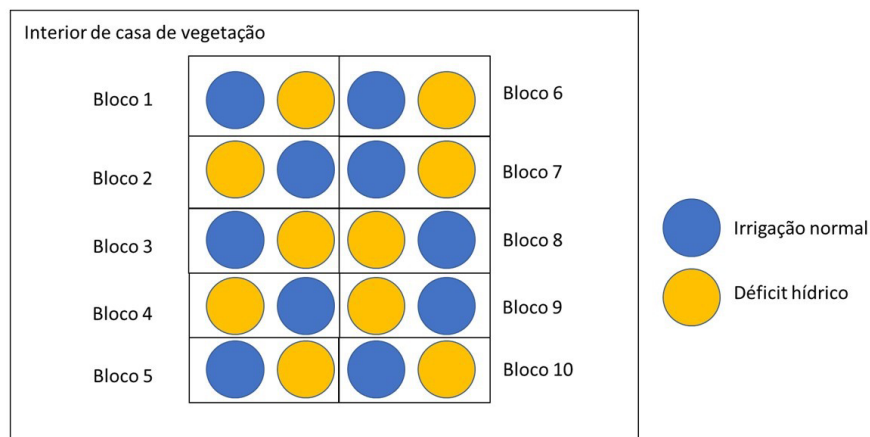
Este experimento tem a configuração de um delineamento inteiramente casualizado (*completely randomized design* – CRD): apresenta 10 repetições de cada tratamento; as repetições estão distribuídas aleatoriamente. O experimento é composto de dois tratamentos.

Se medirmos a variável altura das mudas ( $H$ ) após a aplicação dos tratamentos e submetermos a uma análise de variância, podemos efetuar da seguinte maneira em R (procedimento semelhante ao visto anteriormente nesta apostila):

```
> anova = aov(h ~ tratamento)
```

No entanto, é comum que experimentos biológicos apresentem delineamentos mais complexos, especialmente quando são maiores (maior número de unidades experimentais) e incluem muitos tratamentos. Esta complexidade é ainda maior quando são efetuados em condições de campo (*in situ* ou *ex situ*).

Para ilustrar, vamos considerar o mesmo experimento anterior e inserir uma restrição ao sorteio das unidades experimentais. Desta vez, vamos considerar que o interior de uma casa de vegetação apresenta variações na intensidade de radiação luminosa incidente. Um dos lados da casa frequentemente recebe mais luz do que outro, o que acontece de maneira mais pronunciada à medida que a latitude aumenta nos dois hemisférios do planeta. Nas condições anteriores, plantas de um dos lados poderiam receber mais luz do que as do outro, o que frequentemente influencia no seu crescimento e pode interferir na avaliação estatística do efeito de cada tratamento. Para contornar esta heterogeneidade do ambiente, podemos agrupar as unidades experimentais em blocos. Cada bloco será constituído por uma unidade experimental de cada tratamento, ou seja, um vaso com irrigação normal e um com suspensão da irrigação. Dessa maneira, 10 blocos poderão ser constituídos e distribuídos ao longo da casa de vegetação, conforme segue.



Na figura, as unidades experimentais permaneceram distribuídas da mesma forma, no entanto, introduzimos a restrição de criação de blocos. Cada bloco contém os dois tratamentos representados uma vez. Os blocos podem ser dispostos linearmente um ao lado do outro, mas também podem ser distribuídos de outras formas, de acordo com o espaço do ambiente experimental. Pode ser que o bloco 1 apresente um crescimento em altura diferente das mudas do bloco 10, simplesmente pela diferença de posição na casa de vegetação. Esta restrição nos permite identificar a magnitude do efeito da heterogeneidade ambiental na variável medida (controle local). Este é um exemplo simplificado de um experimento em delineamento em blocos casualizados (*random blocks design* – RBD): as unidades experimentais são agrupadas em grupos chamados blocos; cada bloco contém uma unidade de cada tratamento do experimento; os tratamentos são sorteados dentro de cada bloco e a distribuição dos blocos também é aleatorizada, embora sua numeração apresente-se sequencial (para efeitos de organização do experimento).

Em R, podemos analisar os dados da variável  $H$  implementando o fator de variação bloco no modelo de análise:

```
> anova = aov(h ~ bloco + tratamento)
```

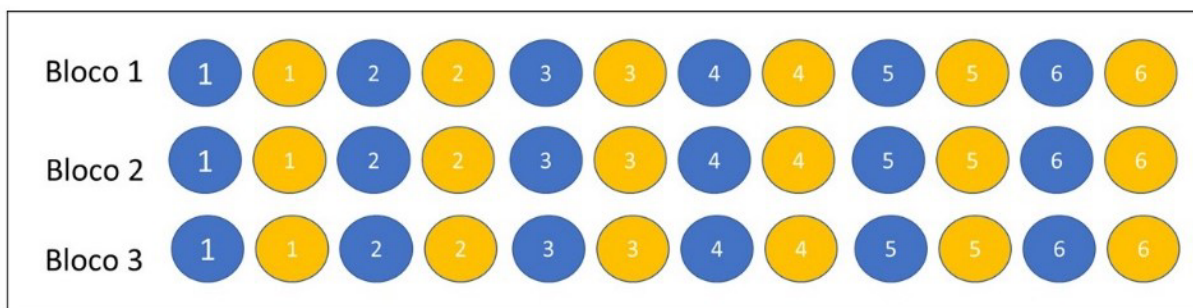
Caso o fator bloco apresente significância estatística (segundo o critério mais usual de  $P < 0,05$ ), a divisão do experimento em blocos foi adequada, devido à heterogeneidade ambiental. No entanto, as diferenças de blocos não devem causar interação e blocos e tratamentos, pois isso pode aumentar o erro experimental e reduzir a precisão do experimento (Storck et al., 2000). A verificação da interação entre blocos e tratamentos pode ser feita através do modelo:

```
> anova = aov(h ~ bloco + tratamento + blocos*tratamento)
```

Se não houver efeito de blocos (considerando o critério estatístico), o mesmo experimento poderia ser analisado como um delineamento inteiramente casualizado (conforme modelo anteriormente proposto para análise em R). Isso é justificado, pois a introdução de mais um fator de variação diminui o grau de liberdade do erro, aumentando o seu valor numérico.

### Análise de um experimento em DBC com mais de um fator

Vamos considerar os dados de um experimento hipotético realizado com feijoeiro (*Phaseolus vulgaris*), constituído em blocos casualizados, utilizando tratamentos semelhantes aos anteriores (regimes de irrigação). No entanto, este novo experimento apresenta mais um nível de complexidade. Além dos tratamentos relacionados ao regime de irrigação, seis diferentes genótipos de feijoeiro foram comparados. Chamamos cada tipo de tratamento de fator. O fator 1 é o genótipo, com seis níveis (seis genótipos) dentro deste fator. O fator 2 é o regime de irrigação, com dois níveis (irrigação normal e suspensão da irrigação). O experimento foi composto em três repetições (três blocos) e avaliado em casa de vegetação. Este é o caso de um experimento bifatorial. Experimentos bifatoriais envolvem duas categorias de tratamentos (dois fatores). Dentro de cada fator são constituídos os níveis (número de subtratamentos de cada fator). No nosso caso, o esquema seguinte ilustra o experimento com três blocos acomodando os dois fatores (genótipos representados pelos números e regimes de irrigação pelas cores). Na representação, no entanto, não foi feito o sorteio das unidades experimentais. Em situação real, é necessário aleatorizar todas as unidades de cada fator e cada nível dentro de cada bloco.



Vamos agora analisar dados hipotéticos para o experimento bifatorial descrito. Primeiramente, vamos fazer a leitura dos dados no R, inserindo-os em um novo objeto chamado dados. O operador decimal do conjunto já foi definido como o ponto (.) no próprio arquivo de dados, não sendo necessário indicá-lo na leitura da tabela.

```
> dados = read.table(file="exp_gen_seca.txt", sep="",  
header=T)  
> dados
```

bloco	genótipo	tratamento	DPV3	PR	H	MST	MSR
1	1	1	controle	18	70.1	145.2	2.1 0.819
2	1	1	seca	20	82.3	120.1	1.7 0.663
3	2	1	controle	19	72.5	150.5	2.2 0.858
4	2	1	seca	20	77.6	115.2	1.6 0.624
5	3	1	controle	18	74.9	140.1	2.3 0.897
6	3	1	seca	22	86.5	98.3	1.4 0.546
7	1	2	controle	17	56.8	110.5	2.4 0.936
8	1	2	seca	19	52.1	65.2	1.8 0.702
9	2	2	controle	18	62.4	114.7	2.5 0.975
10	2	2	seca	19	47.9	51.5	1.7 0.663
11	3	2	controle	17	66.3	138.3	2.6 1.014
12	3	2	seca	20	57.8	77.4	1.6 0.624
13	1	3	controle	21	94.5	75.5	1.8 0.702
14	1	3	seca	24	102.4	69.6	1.5 0.585
15	2	3	controle	22	97.5	80.1	1.5 0.585
16	2	3	seca	25	118.3	75.4	1.5 0.585
17	3	3	controle	22	92.1	92.3	1.6 0.624
18	3	3	seca	22	123.5	90.1	1.4 0.546
19	1	4	controle	16	56.8	115.3	2.1 0.819
20	1	4	seca	22	57.4	49.8	1.7 0.663
21	2	4	controle	15	65.3	128.1	3.1 1.209
22	2	4	seca	21	62.1	39.4	1.8 0.702
23	3	4	controle	17	72.9	136.6	3.2 1.248
24	3	4	seca	20	67.3	65.5	1.6 0.624
25	1	5	controle	20	39.8	87.5	2.5 0.975
26	1	5	seca	25	65.6	86.1	1.4 0.546
27	2	5	controle	21	37.4	89.4	2.6 1.014
28	2	5	seca	26	78.3	88.3	1.4 0.546
29	3	5	controle	22	32.5	102.1	2.5 0.975
30	3	5	seca	23	79.8	90.3	1.3 0.507
31	1	6	controle	23	82.1	117.5	2.4 0.936
32	1	6	seca	25	100.1	56.3	1.3 0.507
33	2	6	controle	25	80.3	125.5	2.5 0.975
34	2	6	seca	27	102.4	49.4	0.9 0.351
35	3	6	controle	24	66.5	135.3	2.5 0.975
36	3	6	seca	27	105.6	38.5	1.5 0.585

Prosseguindo, vamos ler as variáveis do conjunto:

```
> attach(dados)
> names(dados)
[1] "bloco"      "genótipo"   "tratamento" "DPV3"
"PR"         "H"          "MST"        "MSR"
```

As seguintes variáveis foram hipoteticamente medidas: *DPV3* – número de dias para atingir o estágio vegetativo V3, que consiste na completa expansão do primeiro trifólio da planta; *PR*: profundidade radicular; *H*: altura da planta; *MST*: massa seca total; *MSR*: massa seca radicular.

Neste conjunto de dados, os fatores são: bloco, genótipo e tratamento. O fator tratamento refere-se aos regimes de irrigação, embora os genótipos também sejam um fator de tratamentos. Foi apenas uma maneira de identificar os fatores na tabela.

No R, é essencial verificar se os fatores são, de fato, fatores, e se as variáveis numéricas estão efetivamente sendo interpretadas como números. Esta verificação pode ser feita como segue.

```
> is.factor(bloco)
[1] FALSE
```

A verificação do bloco indicou que este não é um fator. Então, é necessário convertê-lo a fator:

```
> as.factor(bloco)
[1] 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1
2 2 3 3 1 1 2 2 3 3
Levels: 1 2 3
```

Prosseguimos, verificando os outros fatores:

```
> is.factor(genótipo)
[1] FALSE
> as.factor(genótipo)
[1] 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 4 5 5
5 5 5 5 6 6 6 6 6 6
Levels: 1 2 3 4 5 6
> is.factor(tratamento)
[1] TRUE
```

Verificam-se, na sequência, as variáveis numéricas:

```
> is.numeric(DPV3)
[1] TRUE
```

## Testes de adequação ao modelo estatístico para ANOVA

Antes da análise de variância, efetuam-se os testes necessários para atender às pressuposições básicas de modelos estatísticos: normalidade dos resíduos e homogeneidade de variâncias. A normalidade pode ser feita pelo teste de Shapiro-Wilk para cada variável da tabela.

```
> shapiro.test(DPV3)

Shapiro-Wilk normality test

data:  DPV3
W = 0.97449, p-value = 0.5601

> shapiro.test(PR)
```

```
Shapiro-Wilk normality test
```

```
data: PR  
W = 0.98316, p-value = 0.8461
```

```
> shapiro.test(H)
```

```
Shapiro-Wilk normality test
```

```
data: H  
W = 0.96595, p-value = 0.3255
```

```
> shapiro.test(MST)
```

```
Shapiro-Wilk normality test
```

```
data: MST  
W = 0.93257, p-value = 0.02997
```

```
> shapiro.test(MSR)
```

```
Shapiro-Wilk normality test
```

```
data: MSR  
W = 0.93257, p-value = 0.02997
```

A análise sugere que as variáveis *MST* e *MSR* não apresentam normalidade, pois  $P < 0,05$ .

A homogeneidade das variâncias pode ser verificada, entre outros testes, pelo método de Bartlett. Por exemplo, no caso da variável *MST*, o teste mostrou não haver homogeneidade de variâncias nos fatores tratamentos e genótipos:

```
> bartlett.test(MST ~ tratamento)
```

```
Bartlett test of homogeneity of variances
```

```
data: MST by tratamento  
Bartlett's K-squared = 7.4516, df = 1, p-value = 0.006338
```

```
> bartlett.test(MST ~ genótipo)
```

```
Bartlett test of homogeneity of variances
```

```
data: MST by genótipo  
Bartlett's K-squared = 11.732, df = 5, p-value = 0.03865
```

```
> bartlett.test(MST ~ bloco)
```

Bartlett test of homogeneity of variances

```
data: MST by bloco
```

```
Bartlett's K-squared = 2.4318, df = 2, p-value = 0.2964
```

Os resultados dos testes sugerem não haver normalidade dos resíduos e também heterogeneidade de variâncias nas variáveis *MST* e *MSR*. Desse modo, pode-se proceder com a transformação dos dados para análise de variância. Um dos métodos para transformação de dados é o boxcox (Box & Cox, 1964). O teste visa estabilizar ou reduzir a variabilidade existente e normalizar os resíduos. A transformação é realizada segundo o seguinte critério matemático:

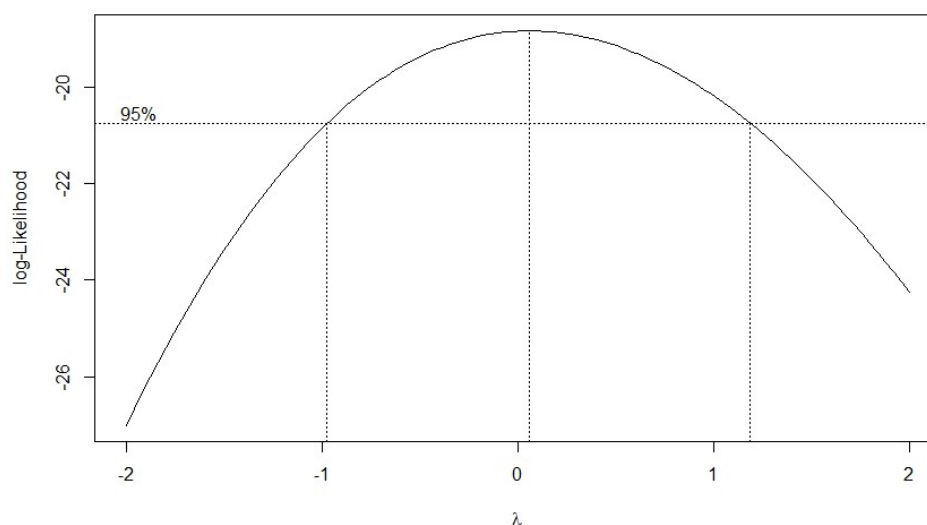
$$y^{(\lambda)} = \begin{cases} \frac{y^\lambda - 1}{\lambda}, & \text{se } \lambda \neq 0 \\ \log y, & \text{se } \lambda = 0 \end{cases}$$

Em R, a transformação boxcox pode ser efetuada através do pacote MASS (Venables & Ripley, 2002; Ripley et al., 2013). Com o comando `boxcox()` é gerado um gráfico envolvendo o conjunto de dados a ser transformado. Neste gráfico, o valor máximo do eixo *y* terá seu correspondente *x* como o valor  $\lambda$ .

```
> require(MASS)
```

```
> boxcox(MST ~ genótipo)
```

Ao digitar esta última linha de comando, é gerado um gráfico para determinar o valor de  $\lambda$ .



Agora, você deve digitar `locator()`, e clicar com o botão esquerdo do mouse no topo do gráfico, pressionando ESC em seguida. Você deverá obter algo como:

```
> locator()
$`x`
[1] 0.06246118

$y
[1] -18.85288
```

O valor de  $x$  é o valor de  $\lambda$  que você usará para aplicar na equação anterior. Você pode modificar a identificação das suas variáveis, por exemplo, adicionando um  $t$  logo após, como forma de diferenciação. No caso da variável  $MST$ , vamos criar a variável transformada  $MSTt$  e aplicar a equação de transformação. Como  $\lambda$  é diferente de zero, aplicamos a equação da seguinte maneira:

```
> MSTt = ((MST ^ 0.06246118)-1)/0.06246118
```

Todos os valores da variável  $MST$  serão transformados pela expressão acima, criando a variável  $MSTt$ . Agora, a variável  $MSTt$  pode ser submetida aos testes de normalidade e homogeneidade.

```
> bartlett.test(MSTt ~ genótipo)

Bartlett test of homogeneity of variances

data:  MSTt by genótipo
Bartlett's K-squared = 10.771, df = 5, p-value = 0.05611

> shapiro.test(MSTt)

Shapiro-Wilk normality test

data:  MSTt
W = 0.95137, p-value = 0.1155
```

Os testes sugerem que os dados transformados apresentam normalidade de resíduos e homogeneidade de variâncias (pelo critério de  $P < 0,05$ ).

Após a verificação dos critérios básicos de normalidade e homogeneidade de variâncias, com as transformações necessárias, pode-se efetuar a análise de variância de todas as variáveis. Se alguma variável, mesmo transformada, não atendeu aos pressupostos, testes não-paramétricos serão conduzidos, como visto anteriormente.



## Realização da ANOVA

No caso do exemplo proposto, a ANOVA pode ser efetuada de maneira simplificada pelo modelo a seguir, considerando a variável *MSTt*.

```
> an = aov(MSTt ~ bloco + tratamento + genótipo +  
tratamento*genótipo)
```

No modelo, cada fator é analisado separadamente (bloco, tratamento, genótipo), no entanto, também se considera a interação entre tratamentos e genótipos. O objetivo é entender se cada fator apresenta efeitos diferentes, mas também verificar se um fator influencia (interage) o outro. A análise dos dados retorna os seguintes resultados:

```
> summary(an)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
bloco	1	0.000004	0.000004	0.036	0.8511
tratamento	1	0.007408	0.007408	60.002	9.72e-09 ***
genótipo	1	0.000033	0.000033	0.269	0.6075
tratamento:genótipo	1	0.000767	0.000767	6.213	0.0182 *
Residuals	31	0.003828	0.000123		

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Pela análise, há diferenças entre os regimes de irrigação, mas não entre os genótipos. No entanto, há interferência de algum(s) genótipo(s) com os tratamentos de irrigação. O quadro sugere a necessidade de desdobrar a análise, para verificar quais tratamentos diferem.

## Testes de comparação de médias

Vamos efetuar comparações de médias entre os tratamentos, utilizando métodos como Tukey, Duncan e Scott-Knott. Para tanto, pode-se utilizar o pacote *laercio* (Silva, 2015).

```
> require(laercio)
```

```
> LTukey(an)
```

TUKEY TEST TO COMPARE MEANS

Confidence level: 0.95  
Dependent variable: MSTt  
Variation Coefficient: 1.068897 %

Independent variable: bloco

Factors	Means
3	1.04017391891725 a

1	1.03931499428973	a
2	1.03915627178815	a

Independent variable: tratamento

Factors	Means	
controle	1.05389374415566	a
seca	1.02520304584109	b

Independent variable: genótipo

Factors	Means	
4	1.04948363284198	a
2	1.04624904441631	ab
5	1.03971729985892	ab
1	1.03934087552518	ab
6	1.03493703689112	ab
3	1.02756248045675	b

> LDuncan(an)

DUNCAN TEST TO COMPARE MEANS

Confidence Level: 0.95

Dependent Variable: MSTt

Variation Coefficient: 1.068897 %

Independent Variable: bloco

Factors	Means	
3	1.04017391891725	a
1	1.03931499428973	ab
2	1.03915627178815	b

Independent Variable: tratamento

Factors	Means	
controle	1.05389374415566	a
seca	1.02520304584109	b

Independent Variable: genótipo

Factors	Means	
4	1.04948363284198	a
2	1.04624904441631	a
5	1.03971729985892	ab
1	1.03934087552518	ab
6	1.03493703689112	ab

```

3          1.02756248045675  b

> LScottKnott(an, "genótipo")

SCOTT-KNOTT ORIGINAL TEST

Confidence Level:  0.95
Independent variable:  genótipo

  FACTORS      MEANS
1          4 1.049484 A
2          2 1.046249 A
3          5 1.039717 A
4          1 1.039341 A
5          6 1.034937 A
6          3 1.027562 A

> LScottKnott(an, "tratamento")

```

```

SCOTT-KNOTT ORIGINAL TEST

Confidence Level:  0.95
Independent variable:  tratamento

```

```

  FACTORS      MEANS
1 controle 1.053894 A
2      seca 1.025203 B

```

Os três testes mostram algumas variações, especialmente quanto aos genótipos. Há algumas diferenças entre os genótipos que podem não ter sido verificadas pela análise inicial de variância (ANOVA). Um detalhe importante da análise realizada concerne ao uso de dados transformados. Efetivamente, utilizamos os valores transformados para conduzir as análises estatísticas. No entanto, são os valores originais que deverão ser colocados na tabela do seu relatório, artigo ou monografia. Ao lado dos valores originais, poderão ser utilizadas as informações da análise estatística (níveis de significância representados por asteriscos; ou letras diferentes obtidas pelos testes de comparação de médias) com os dados transformados. Você jamais usará valores transformados na sua tabela final.

### **Análise de experimentos bifatoriais no pacote ExpDes**

Uma maneira mais específica de análise de dados de experimentos bifatoriais pode ser realizada no pacote ExpDes (Ferreira et al., 2014). Através de linhas de comando simplificadas, o pacote permite a realização de ANOVA seguida por testes de comparações múltiplas, apresentando-se os desdobramentos de cada fator nas comparações. Ressalta-se que o pacote foi desenvolvido, no entanto, para realizar

análises de experimentos em delineamento inteiramente casualizado, blocos casualizados e quadrado latino. Além disso, permite analisar experimentos fatoriais duplos e triplos, parcelas subdivididas no tempo, dentre outras aplicações.

Vamos empregar o pacote para fazer a ANOVA e os desdobramentos dos fatores e suas interações, considerando o nosso exemplo dos genótipos de feijoeiro sob os dois regimes de irrigação.

```
> require(ExpDes)
```

No modelo, utilizamos fat2 para se referir a um fatorial duplo (bifatorial), em que os fatores são genótipo e tratamento. Além disso, o experimento é analisado em bloco (rbd) e a variável a ser analisada é DPV3, conforme segue.

```
> fat2.rbd(genótipo, tratamento, bloco, DPV3)
```

Esta linha de comando simples produz a série de análises apresentadas na sequência, desdobrando todas os fatores, seus níveis e suas possíveis interações.

```
> fat2.rbd(genótipo, tratamento, bloco, DPV3)
```

```
-----
Legend:
```

```
FACTOR 1:  F1
```

```
FACTOR 2:  F2
-----
```

```
Analysis of Variance Table
```

```
-----
              DF      SS      MS      Fc      Pr>Fc
Block          2    2.67    1.333    1.294 0.29420
F1              5 233.67  46.733  45.359 0.00000
F2              1  75.11  75.111  72.902 0.00000
F1*F2          5  10.89   2.178   2.114 0.10189
Residuals     22  22.67   1.030
Total          35 345.00
-----
```

```
CV = 4.8 %
```

```
-----
Shapiro-Wilk normality test
```

```
p-value: 0.2779497
```

```
According to Shapiro-Wilk normality test at 5% of significance, residuals can
be considered normal.
-----
```

```
No significant interaction: analyzing the simple effect
-----
```

```
F1
```

```
Tukey's test
-----
```

```
Groups Treatments Means
a          6    25.16667
b          5    22.83333
b          3    22.66667
c          1    19.5
c          4    18.5
c          2    18.33333
```

```

-----
F2
Tukey's test
-----
Groups Treatments Means
a      seca      22.61111
b      controle   19.72222
-----

```

No caso da variável *DTV3* não foram detectadas interações significativas. Desse modo, os testes complementares de médias (Tukey) foram realizados dentro de cada fator (comparando-se os níveis), i.e., comparam-se os seis genótipos dentro do fator genótipo, e também os dois regimes de irrigação do outro fator.

Observe que quando ocorrem interações, estas são desdobradas em detalhamento. Veja os resultados da ANOVA, por exemplo, para a variável *PR*.

```

> fat2.rbd(genótipo, tratamento, bloco, PR)
-----

```

```

Legend:
FACTOR 1:  F1
FACTOR 2:  F2
-----

```

```

Analysis of Variance Table
-----

```

	DF	SS	MS	Fc	Pr>Fc
Block	2	184.5	92.25	2.870	0.078057
F1	5	11526.6	2305.32	71.724	0.000000
F2	1	1685.1	1685.10	52.427	0.000000
F1*F2	5	2406.6	481.33	14.975	0.000002
Residuals	22	707.1	32.14		
Total	35	16509.9			

```

-----
CV = 7.59 %
-----

```

```

-----
Shapiro-Wilk normality test
p-value: 0.0999162

```

```

According to Shapiro-Wilk normality test at 5% of significance, residuals can
be considered normal.
-----

```

```

-----
Significant interaction: analyzing the interaction
-----

```

```

-----
Analyzing  F1  inside of each level of  F2
-----

```

```

-----
Analysis of Variance Table
-----

```

	DF	SS	MS	Fc	Pr.Fc
Block	2	184.505	92.2525	2.8702	0.0781
F2	1	1685.102	1685.1025	52.4275	0
F1:F2 controle	5	5510.732	1102.14633	34.2903	0
F1:F2 seca	5	8422.493	1684.49867	52.4087	0
Residuals	22	707.115	32.14159		

Total 35 16509.947

F1 inside of the level controle of F2

Tukey's test

Groups	Treatments	Means
a	3	94.7
b	6	76.3
bc	1	72.5
bc	4	65
c	2	61.83333
d	5	36.56667

F1 inside of the level seca of F2

Tukey's test

Groups	Treatments	Means
a	3	114.7333
a	6	102.7
b	1	82.13333
bc	5	74.56667
cd	4	62.26667
d	2	52.6

Analyzing F2 inside of each level of F1

Analysis of Variance Table

	DF	SS	MS	Fc	Pr.Fc
Block	2	184.50500	92.2525	2.8702	0.0781
F1	5	11526.59583	2305.31917	71.7239	0
F2:F1 1	1	139.20167	139.20167	4.3309	0.0493
F2:F1 2	1	127.88167	127.88167	3.9787	0.0586
F2:F1 3	1	602.00167	602.00167	18.7297	3e-04
F2:F1 4	1	11.20667	11.20667	0.3487	0.5609
F2:F1 5	1	2166.00000	2166	67.3893	0
F2:F1 6	1	1045.44000	1045.44	32.5261	0
Residuals	22	707.11500	32.14159		
Total	35	16509.94750			

F2 inside of the level 1 of F1

Tukey's test

Groups	Treatments	Means
a	2	82.13333
b	1	72.5

F2 inside of the level 2 of F1

According to the F test, the means of this factor are statistical equal.

Levels	Means
1	1 61.83333
2	2 52.60000

```

F2 inside of the level 3 of F1
-----
Tukey's test
-----
Groups Treatments Means
a      2      114.7333
b      1       94.7
-----

```

```

F2 inside of the level 4 of F1

According to the F test, the means of this factor are statistical equal.
-----
      Levels      Means
1         1  65.00000
2         2  62.26667
-----

```

```

F2 inside of the level 5 of F1
-----
Tukey's test
-----
Groups Treatments Means
a      2      74.56667
b      1      36.56667
-----

```

```

F2 inside of the level 6 of F1
-----
Tukey's test
-----
Groups Treatments Means
a      2      102.7
b      1       76.3
-----

```

Neste caso, houve interação significativa entre os fatores regime de irrigação e genótipo. Cada interação foi desdobrada em todos os seus níveis, para verificar possíveis diferenças pelo teste de Tukey ( $P < 0,05$ ).

No ExpDes, Ferreira et al. (2014) desenvolveram linhas de comando simplificadas para fatoriais duplos e triplos em delineamentos distintos:

- para fatorial duplo em delineamento inteiramente casualizado

```
> fat2.crd(factor1, factor2, resp)
```

- para fatorial duplo em delineamento em blocos casualizados

```
> fat2.rbd(factor1, factor2, block, resp)
```

- para fatorial triplo em delineamento inteiramente casualizado

```
> fat3.crd(factor1, factor2, factor3, resp)
```

- para fatorial triplo em delineamento em blocos casualizados

```
> fat3.rbd(factor1, factor2, factor3, block, resp)
```

## R aplicado à bioinformática: fundamentos sobre alinhamentos de sequências e reconstrução filogenética

O uso de pacotes em R também têm sido extensivamente aplicados a análises bioinformáticas. A bioinformática é um ramo da ciência que se dedica à aplicação da informática na análise de dados provenientes de estudos biológicos. Em outras palavras, aplicam-se técnicas computacionais e matemáticas à geração e gerenciamento de informações biológicas (Attwood et al., 2011).

Dentre as diversas aplicações da bioinformática, cabe destacar duas análises fundamentais que um profissional vinculado à área de ciências biológicas deve saber: (i) o alinhamento de sequências de nucleotídeos e aminoácidos; e (ii) a reconstrução filogenética. Vamos aplicar alguns pacotes de R para realizar o alinhamento de sequências de nucleotídeos de um pequeno conjunto de sequências de nucleotídeos e aminoácidos.

Primeiramente, você deve saber que sequências de nucleotídeos e aminoácidos precisam obedecer a um formato padrão para serem representadas. Um dos formatos mais comuns é o FASTA, em que a sequência é representada por um identificador, seguido da sequência propriamente dita. O início de cada sequência é delimitado pelo símbolo >. Veja um exemplo de formato FASTA para uma sequência de nucleotídeos da região que codifica uma proteína para o gene *PvDREB6B* (Konzen et al., 2019).

```
> MK874722.1 Phaseolus vulgaris genotype UCD CANARIO 707
DREB6B (DREB6B) gene, complete cds
ATGGGAACCGCTATAGATATGTACAACAACACCAACATCGTACCGGATTTATAGATCCGTATAGTGAGG
AGCTGATGAGAGCACTTGAGCCTTTTATGAAAAGTATATTTCTCTGCCTCTTCTAATTCTGCATCTCC
CTCACGTCAATCACACCCTTCTTCTCTCATTTTCATCTACTTCGTATTCTCTCCCCAACCAAATCAAAC
AACCACCTCACCTCAGATCAAATCTCCAGATTCAGGCCAGGTCTGGGATTCACACCCAGTGGGTCAAT
CCAACCAACGTCAAAATCAAGCCAGATGGGGCCAAACGTGTCCCATGAAACACGGTGGCGCGGCCGC
GAAAGCTGCGAAGCTGTACCGTGGGGTGCGGCAACGGCATTGGGGGAAGTGGGTCTGCCGAGATCAGACTC
CAGAAGAACCGCACGCGCTCTGGCTGGGAACATTCGACACCGGAGAGGAAGCGGCGCTGGCGTACGACA
ACGCAGCGTTTAAGCTCCGAGGCGAGTTCGCGCGCCTCAACTTCCCTCATCTGCGACACCAGGGAGCGTT
CGTATTTCGGCGACTTTGGAGATTACAAGCCGCTACCTTCTTCCGTGGACTCCAAGCTCCAAGCTATTTGT
GAAAGCATGGGAAAACAAGAACAAGGAGTGTGCTCCGTGCAAGACGTGAAGCCCGAGGTACACGCTG
CCGCTGAGCTGGCGCCGTTGGATTCTGACGTGGCGCAATCCAACGTTTGTCCCGAGTTGGACAATATTAA
GGTAGAAAACATGAACGAAACCCCAATGTTGTCATGGCCTGTGTCTGGCGAATCTTCTTGCCTGAATCG
GGTTTACTTTCTTGGATTTGTCTGATTTCTCATATTTCTAATTATGAGTGGGATGAAATAGAGAGTTTTG
GGTTGGAGAAGTACCTTCGGTGGAGATTGATTGGGCGGCTATATGA
```



Esta sequência está depositada no banco de dados Genbank. É uma sequência de feijoeiro-comum para um genótipo específico chamado de UCD Canario 707.

Agora, veja uma sequência FASTA de aminoácidos:

```
> QCI62111.1 DREB6B [Phaseolus vulgaris]
MGTAIDMYNNTNIVPDFLDPYSEELMRALEPFPMKTDYFSASSNSASPSRQSHPSLISSTSYSSPNQIKL
NQLTSDQILQIQAQVGIQHHVGHSHNRLNQAQLGPKRVPMKHGGAAAKAKLYRGVRQRHWGKWVAEIRL
QKNRTRLWLGTFTDTEEAALAYDNAAFKLRGEFARLNFPHLRHQGAFFVFGDFGDYKPLPSSVDSKLQAIC
ESMGKQEQQKCCSVEDVKPEVDAAELAPVESDVAQSNVCPELDNIKVENVNENPMLSWPVSGESSSPES
GFTFLDLSDFSYSNYEWDEIESFGLEKYPSVEIDWAAI
```

Para realizar alinhamentos e construir árvores filogenéticas você pode modificar a identificação da sequência para não aparecerem tantas codificações. Por exemplo, a sequência de nucleotídeos anterior poderia ser representada por:

```
> UCD Canario 707
ATGGGAACCGCTATAGATATGTACAACAACACCAACATCGTACCGGATTTCATAGATCCGTATAGTGAGG
AGCTGATGAGAGCACTTGAGCCTTTTATGAAAACCTGATTATTTCTCTGCCTCTTCTAATTCTGCATCTCC
CTCACGTCAATCACACCCTTCTTCTCTCATTTTCATCTACTTCGTATTCCTCCCCCAACCAAATCAAACCTA
AACCAACTCACCTCAGATCAAATCCTCCAGATTGAGGCCAGGTCGGGATTCACACACCACGTGGGTCATT
CCAACCAACGTCAAAATCAAGCCCAGATGGGGCCAAACGTGTCCCATGAAACACGGTGGCGCGGCCCGC
GAAAGCTGCGAAGCTGTACCGTGGGGTGCGGCAACGGCATTGGGGGAAGTGGGTGCGCGAGATCAGACTC
CAGAAGAACCGCACGCGCCTCTGGCTGGGAACATTGACACCGGAGAGGAAGCGGCGCTGGCGTACGACA
ACGCAGCGTTTAAAGCTCCGAGGCGAGTTCGCGCGCCTCAACTTCCCTCATCTGCGACACCAGGGAGCGTT
CGTATTCGGCGACTTTGGAGATTACAAGCCGCTACCTTCTTCCGTGGACTCCAAGCTCCAAGCTATTTGT
GAAAGCATGGGAAAACAAGAACAAAAAAGTGTTGCTCCGTGCAAGACGTGAAGCCCCGAGGTACACGCTG
CCGCTGAGCTGGCGCCGGTGGATTCTGACGTGGCGCAATCCAACGTTTGTCCCGAGTTGGACAATATTA
GGTAGAAAACATGAACGAAACCCCAATGTTGTCATGGCCTGTGTCTGGCGAATCTTCTTCGCCTGAATCG
GGTTTTACTTTCTTGATTGTGTCGGATTCTCATATTCTAATTATGAGTGGGATGAAATAGAGAGTTTTG
GGTTGGAGAAGTACCCTTCGGTGGAGATTGATTGGGCGCTATATGA
```

Podemos alinhar um conjunto de sequências de nucleotídeos para diversos genótipos diferentes que tiveram esse mesmo gene sequenciado, a fim de comparar as sequências e identificar variantes nucleotídicas (variantes de uma base ao longo da sequência são chamadas de polimorfismos de nucleotídeo simples ou SNP).

Para tanto, vamos usar um pacote em R chamado DECIPHER (Wright, 2016). Para instalar o pacote, você deve usar um caminho diferente do que normalmente utiliza para pacotes em geral:

```
> if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")
> BiocManager::install("DECIPHER")
```

Vamos abrir um arquivo de sequências de nucleotídeos salvo em formato FASTA para realizar o alinhamento.

Para isso, abra o pacote DECIPHER.

```
library(DECIPHER)
```

Agora, especifique a localização do arquivo que deseja inserir (você pode clicar sobre o arquivo no seu computador e especificar as propriedades de sua localização):

```
fas <- " C:\Users\Enéas\OneDrive\Documentos\1 - UFRGS  
DOCÊNCIA\4 - Projetos de extensão\Curso R"
```

No entanto, se você deixar somente uma \, o R não irá reconhecer. Você deve duplicar a barra ou invertê-la para / (neste caso você pode usar somente uma). Além disso, inclua o nome do arquivo no formato FA (fasta) no final:

```
> fas <- " C:\\Users\\Enéas\\OneDrive\\Documentos\\1 -  
UFRGS DOCÊNCIA\\4 - Projetos de extensão\\Curso  
R\\sequencias.fa"
```

Agora, vamos realizar a leitura das sequências do arquivo em um objeto:

```
> dna <- readDNAStringSet(fas)
```

Visualizando com:

```
> dna # the unaligned sequences
```

```
A DNAStringSet instance of length 18  
      width seq                                     names  
[1] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA  
Phytozome (PvDREB6B)  
[2] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA BAT 477  
[3] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA BAT 93  
[4] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA CAL 143  
[5] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA G12873  
...  
[14] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA SEA-5  
[15] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA SXB405  
[16] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA UCD-0801  
[17] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA UCD-Canario  
707  
[18] 957 ATGGGAACCGCTATAGATATGTACA...GAGATTGATTGGGCGGCTATATGA ICA-Bunsi
```

Existem algoritmos diferentes para alinhamento de sequências, mas não cabe a este documento relatar detalhes sobre tais procedimentos. Para alinhar as sequências de DNA usando DECIPHER, proceda da seguinte maneira:

```
> alinhamento <- AlignSeqs(dna)
```

O que deve aparecer na sequência é:

```

Determining distance matrix based on shared 9-mers:
|=====| 100%

Time difference of 0.07 secs

Clustering into groups by similarity:
|=====| 100%

Time difference of 0.01 secs

Aligning Sequences:
|=====| 100%

Time difference of 0.23 secs

Iteration 1 of 2:

Determining distance matrix based on alignment:
|=====| 100%

Time difference of 0 secs

Reclustering into groups by similarity:
|=====| 100%

Time difference of 0.01 secs

Realigning Sequences:
|=====| 100%

Time difference of 0.01 secs

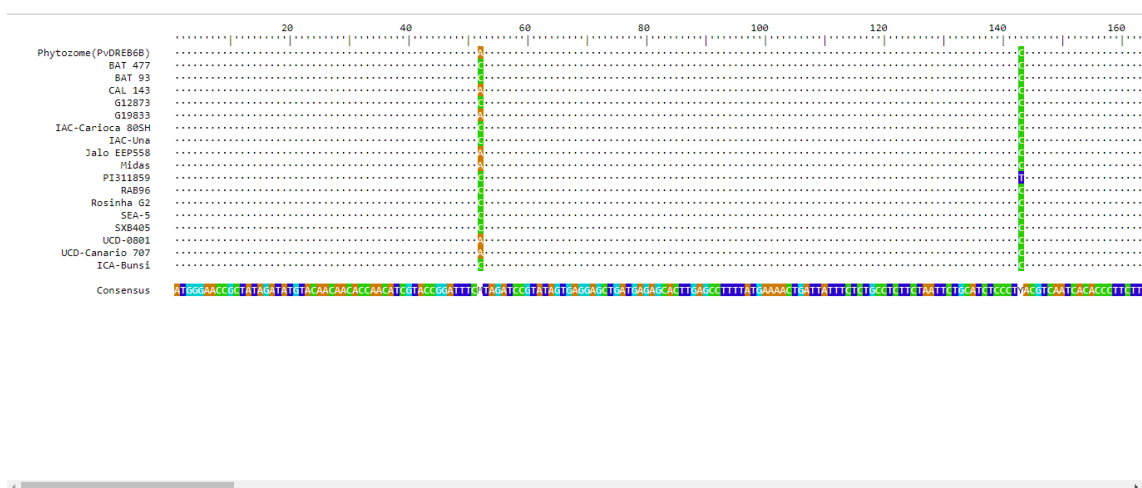
Alignment converged - skipping remaining iteration.

```

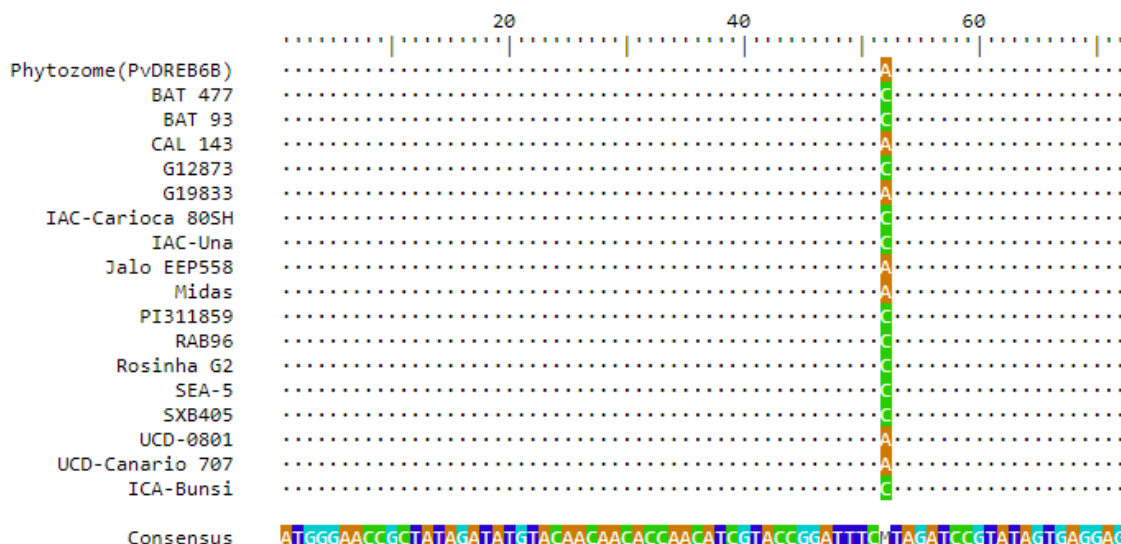
Para visualizar o alinhamento:

```
> BrowseSeqs(alinhamento, highligh=0)
```

O alinhamento será mostrado no seu browser da Internet:



Vamos entender o que essa figura representa a partir de um zoom:



Na figura você pode visualizar as 18 sequências de nucleotídeos identificadas pelos genótipos (desde Phytozome até ICA-Bunsi). A barra superior indica o número sequencial de bases que compõem a sequência. “Consensus” é a sequência consenso, ou seja, a sequência que representa a base que ocorre mais frequentemente entre todas as sequências ou uma base alternativa como M, que representa a presença de A ou C no sítio considerado. A maioria das bases são representadas por pontos (.), o que indica que todas as sequências, naquela posição, apresentam a mesma base. Esta pode ser facilmente identificada pela sequência consenso. Por exemplo, a primeira base de todas as sequências é A, a segunda é T e a terceira é G. No entanto, na base número 52 há variação ao longo das sequências, de modo que estas bases aparecem devidamente identificadas. Ou seja, há um polimorfismo de um nucleotídeo (SNP) nesta localização. Algumas sequências possuem a base A, enquanto as outras possuem a base C.

Podemos agora construir uma árvore filogenética a partir das sequências. Por serem sequências de DNA da mesma espécie, é comum construir árvores usando a própria sequência de nucleotídeos para identificar relações entre indivíduos, populações etc. desta espécie. Mas para análises envolvendo espécies diferentes (às vezes de filos ou outros grupos filogeneticamente distantes) é mais frequente utilizar sequências de aminoácidos.

Para construir uma árvore filogenética, algumas etapas cruciais precisam ser seguidas. O detalhamento destas etapas também está muito além dos objetivos deste documento. Para quem desejar, uma revisão simplificada sobre os principais métodos de inferência filogenética pode ser consultada no trabalho de Yang & Rannala (2012). Cabe destacar, no entanto, três etapas sequenciais para construção de uma árvore

filogenética quando se trata de um conjunto de sequências de nucleotídeos ou aminoácidos:

- 1) Alinhamento das sequências;
- 2) Escolha de um método de substituição de nucleotídeos ou aminoácidos adequado: existem modelos teóricos que examinam as taxas de mutação entre bases nitrogenadas ou entre aminoácidos. Com base em um conjunto de sequências que você analisa, um dos modelos poderá ser selecionado como mais adequado em um programa e ser, então, usado para construir a árvore filogenética;
- 3) Escolha do método de reconstrução filogenética: dependerá da abordagem do estudo, das sequências que estiver trabalhando (regiões gênicas ou não-gênicas, dentre outras) e da proximidade filogenética entre os espécimes sequenciados (são indivíduos da mesma espécie ou de espécies próximas ou ainda muito distintas). Os métodos mais frequentes de inferência podem ser categorizados em métodos por distância e métodos discretos (parcimônia, máxima verossimilhança e inferência bayesiana). Há combinações entre os métodos também. Frequentemente, utiliza-se a máxima verossimilhança ou *maximum likelihood* (ML).

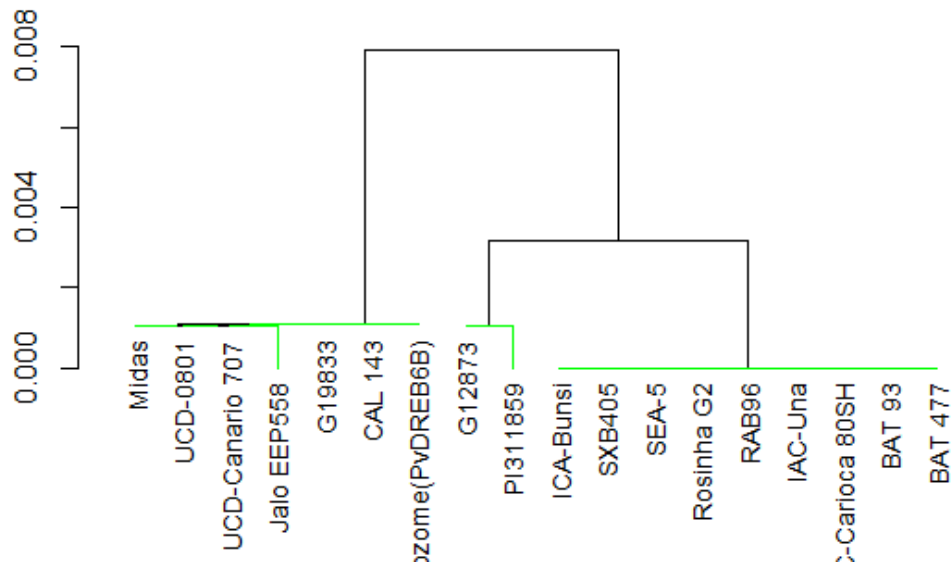
Vamos utilizar as sequências anteriores para construir uma árvore filogenética utilizando um modelo de substituição de nucleotídeos chamado de Jukes-Cantor (Jukes & Cantor, 1969). Vamos partir do alinhamento gerado anteriormente e construir uma matriz de distâncias a partir de diferenças de nucleotídeos ao longo da sequência do gene:

```
> d <- DistanceMatrix(alinhamento, correction="Jukes-Cantor", verbose=FALSE)
```

Agora, vamos construir a árvore filogenética baseada em método de máxima verossimilhança (ML):

```
> c <- IdClusters(d, method="ML", cutoff=.05, showPlot=TRUE, myXStringSet=dna, verbose=FALSE)
```

O resultado é uma árvore computada a partir da matriz de distâncias, utilizando algoritmos para o método de reconstrução por máxima verossimilhança.



Sabendo que as 18 sequências com as quais você trabalhou representam a região codificadora de um gene *DREB*, você pode também traduzir estas sequências, ou seja, converter os nucleotídeos a aminoácidos (utilizando o código genético universal, no caso do nosso exemplo com plantas). Usando DECIPHER você pode executar os seguintes passos:

```
> AA <- AlignTranslation(dna, type="AAStringSet") # align
the translationDNA
> BrowseSeqs(AA, highlight=1) # view the alignment
```

Neste alinhamento a primeira sequência aparece, pois foi destacada na linha de comando (`highlight = 1`).



Se desejar salvar o alinhamento, especifique o local conforme o exemplo:

```
> writeXStringSet(AA, file="C:\\Users\\Enéas\\OneDrive\\D
ocumentos\\1 - UFRGS DOCÊNCIA\\3 - Projetos de pesquisa\\
Bioinformática\\1001al.fas")
```

Diversas outras análises podem ser feitas com sequências como as apresentadas. O alinhamento e a construção de árvores filogenéticas, no entanto, são essenciais para entender outras etapas. O usuário iniciante em R se deparará com diversos novos pacotes para realizar estes e outros procedimentos. Cabe uma extensa revisão de literatura sobre os métodos e pacotes que forem mais apropriados para o tipo de análise que for necessária.

## Introdução a séries temporais

Séries temporais são usualmente decorrentes de coleta de informações a longo prazo, como os programas ecológicos de longa duração (PELDs). Aqui, iremos trabalhar com dados artificialmente criados, com o objetivo adicional de enriquecer o vocabulário R dos leitores.

Em nosso exemplo, vamos criar um conjunto de dados fictício que simula o tempo de leitura (em minutos) por dia de um aluno de graduação ao longo do primeiro ano de curso.

```
> dias <- 1:365 # números representando 365 dias do ano
> leitura <- dias/365+2*runif(365,0,(1:365)/10)# cria 365
valores que representam minutos de leitura
```

A série de comandos abaixo irá trabalhar com loops, isto é, funções que repetem uma análise ao longo de uma série especificada. Muito embora os comandos pareçam mais complicados, eles são intuitivos:

```
> resultado <-NULL # criar um objeto para alocar valores
apenas depois

> n <- length(dias) # um modo de não ficarmos limitados a
um valor, mas sim ao número total de valores de um vetor
```

Aqui, abaixo, vem a parte mais complicada do script. Aqui, vamos utilizar a função `for()`, o qual tem esta estrutura básica:

```
> for(i in conjunto de valores){
  # comandos que
  # serão repetidos
}
```

Por exemplo:

```
> for(i in 1:5){
  print(letters[i])
}
[1] "a"
[1] "b"
[1] "c"
```

```
[1] "d"
[1] "e"
```

No nosso exemplo, vamos tentar dividir este tempo de leitura a cada 7 dias (série semanal). Para isto, podemos adaptar a lógica do argumento para:

```
> for (i in seq(1, n ,7)) {
# definimos que o loop será a cada 7 dias. O { define o
início dos argumentos de uma função, como veremos a
seguir

mediasemanal <- cbind(i,mean(leitura[i:(i+6)]))
# aqui estamos dizendo para o R extrair um valor médio a
cada 7 dias; note o detalhe da soma (i + 6), o que soma 7
dias de observações

resultado <- rbind(resultado, mediasemanal)
}

> resultado # Observe o resultado
```

Uma vez que temos valores de tempo de leitura a cada 7 dias, podemos criar um gráfico que representa isto. O uso de gráficos para analisar séries temporais é sempre bem-vindo. Vamos criar dois gráficos em um único painel. O primeiro gráfico irá mostrar a variação diária do tempo de leitura, enquanto o segundo irá utilizar o resultado da média semanal. Os comandos poderiam ser os seguintes:

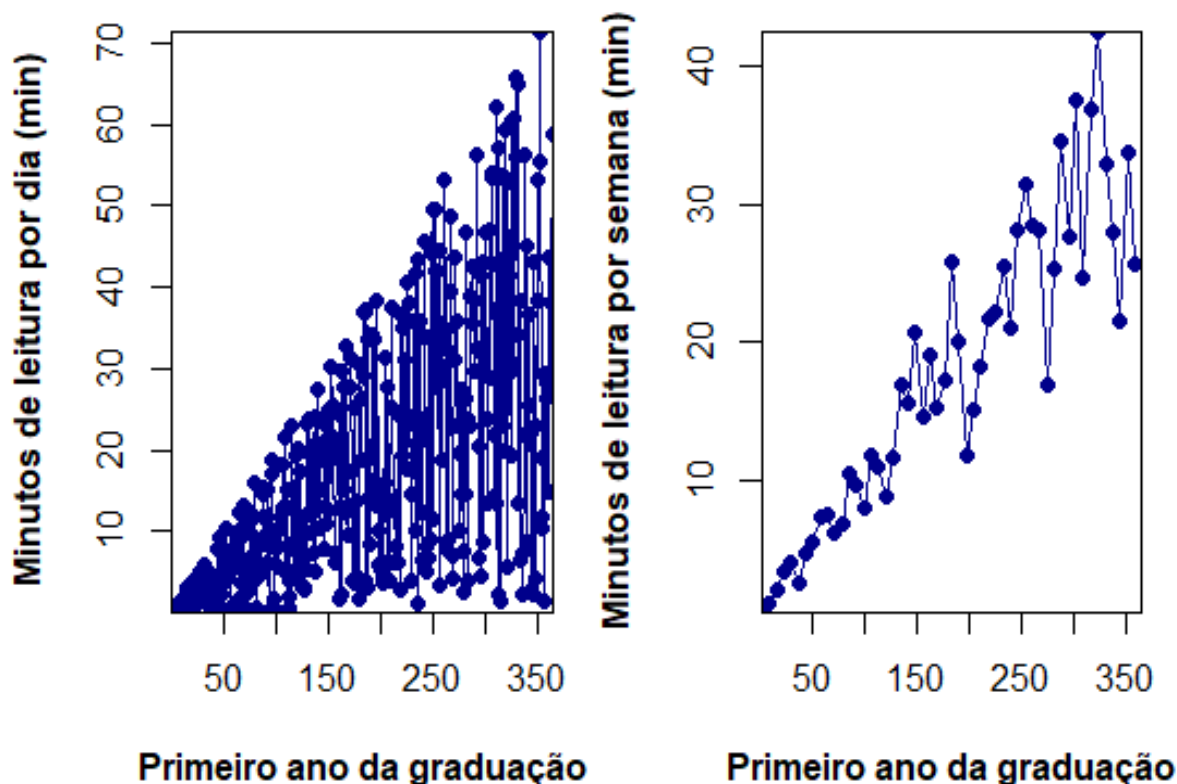
```
> op <- par(mfrow = c(1, 2),
mar=c(4,4,3,1),xaxs="i",yaxs="i") #painel

#Gráfico 1 - Por dia
> plot(dias, leitura,pch=16, cex=0.95, col="dark
blue",type="o",xlab="Primeiro ano da graduação",
ylab="Minutos de leitura por dia (min)", main=NULL,
font.lab=2)

#Gráfico 2 - Por média semanal
> plot(resultado[,1],resultado[,2],pch=16, cex=0.95,
col="dark blue", type="o", xlab="Primeiro ano da
graduação", ylab="Minutos de leitura por semana (min)",
main=NULL, font.lab=2)
> par(op)
```

O resultado para estes comandos será o seguinte gráfico:





### Criando sua própria função

Uma das grandes vantagens do R é a liberdade de criação. Como um escritor, ou um artista, um cientista de dados pode, dependendo do seu conhecimento prévio, compor novas funções que atendem às demandas específicas de sua pesquisa. Este tópico, um pouco avançado dentro do objeto primordial deste documento, é uma primeira exposição do poder do R aos usuários iniciantes.

O comando básico para criar uma função é o seguinte:

```
> minhafunção <- function(x) {      }
```

Note que nada foi especificado. Dentro das chaves é possível especificar qual tipo de cálculo (ou atributos lógicos) você quer que a função realize com “x”, que pode ser um vetor, dataframe, matriz etc.

Por exemplo, podemos fazer uma função que calcule o valor de “x” elevado ao quadrado:

```
> X2 <- function(a) { a^2 }
> X2(10)
[1] 100
```

Outro exemplo um pouco menos simples é uma função que calcula a média final de um conjunto de notas de uma disciplina e ainda assinala o conceito final para cada aluno. Os conceitos finais aqui utilizados, por praticidade, são aqueles normalmente aplicados pelos departamentos da UFRGS, sendo atribuído A (ótimo) para notas finais acima de 9.0, conceito B (bom), nota final entre 7.5 e 8.9, conceito C (regular), entre 6.0 e 7.4, e notas finais menores que 6, conceito D (reprovado).

```
> conceitofinal <- function(x) {y=mean(x)
{ifelse (y >= 9.0, "A", ifelse(y >= 7.5 & y <= 8.9, "B",
ifelse(y >= 6.0 & y <= 7.4, "C", "D"))))
}}
```

# o ifelse funciona com os seguintes argumentos (ifelse, condição, Sim, Não). Note que o "sim" e o "não" pode ser, novamente, o argumento ifelse.

Vamos utilizar um dataframe com a notas parciais de três avaliações de 10 alunos de uma disciplina de graduação (todos os dados são fictícios):

```
> notas <- read.csv("notas201X.csv") #lembre-se que
talvez seja necessário adicionar o argumento sep=";".
```

Vamos, por praticidade, adicionar uma nova coluna aos dados com as notas finais (médias das notas parciais), para fazer uma conferência dos conceitos aferidos no próximo comando:

```
> notas$NF <- apply(notas[,2:4], 1, mean)
```

Agora, vamos utilizar a função que criamos para adicionar os conceitos:

```
> notas$Conceito_Final <- apply(notas[,2:4], 1,
conceitofinal)
```

```
> head(notas) # conferir se a NF corresponde ao conceito
assinalado.
```

Ademais, seria útil exportar estes dados para uma planilha do excel para depois enviar aos alunos. Existem formas muito simples de se fazer isto, como o comando abaixo:

```
> write.csv(notas, "Notas__Finais__2019.csv")
# O arquivo, "Notas__Finais__2019.csv", será exportado,
assim como no caso das figuras, para a pasta que é o
diretório de trabalho do R. Você pode conferir através do
comando getwd().
```

Agora vamos demonstrar uma função mais completa que utiliza aqueles dados prévios da série temporal de leitura por dia durante o primeiro ano da graduação. O objetivo da função é simples; ao invés de utilizar três linhas de comando, vamos colocar tudo dentro de uma única função. Note que a lógica é a mesma dos comandos utilizados na análise de séries temporais:

```
> serietemporal <- function(x, y, serietemporal,
intervalo) {
  timedata <- NULL
  vardata <- NULL
  for (i in seq(1, serietemporal-intervalo,
intervalo)) {
    timedata <- c(timedata, x[i])
    vardata <- c(vardata, mean(y[i:(i + intervalo-
1)]))
    result <- cbind(timedata, vardata) }
  return(result) }
```

Vamos testar a função, pedindo para ela calcular uma média de leitura a cada 30 dias:

```
> teste30 <- serietemporal(dias, leitura, 365, 30)
> teste30
# Veja o resultado
```

Agora, podemos plotar o resultado, que é bem similar àqueles que fizemos previamente com as séries temporais.

```
> plot(teste30, pch=16, cex=0.95, col="dark blue",
type="o", xlab="Primeiro ano da graduação", ylab="Minutos
de leitura por mês (min)", main="", font.lab=2)
```

Confere com o que você esperava?

## Considerações finais

Como reportado no início deste documento, o objetivo primário desta introdução é apresentar tanto as funções quanto os argumentos mais comuns que podem ser muito úteis a um iniciante no R. Longe de esgotar qualquer conhecimento sobre o assunto, muitos dos comandos aqui empregados representam o estilo dos autores e, muitas vezes, existem outros modos de realizar a mesma operação. Nós mesmos compartilhamos muitas diferenças em realizar as mesmas operações e adotamos aqui consensos em alguns pontos. Este é um indicativo claro da gama de possibilidades que é trabalhar com R, além de divertido.

O R carece de uso constante e encarar novos desafios nesta plataforma aumentam nossa vivência, assim como no aprendizado de uma nova língua além da materna. É possível criar uma infinidade de resultados no R. Isto apenas demonstra o quanto podemos ainda aprender. São diversos os tipos de usuários; alguns aprendem apenas operações básicas que mais usam no dia a dia, assim como terá alguns que irão fazer de tudo para tornar cada linha de script mais elegante e prática possível.

Assim como este humilde documento, existem uma série de outras apostilas que têm um rico conhecimento sobre esta parte introdutória. Por favor, não fique restrito à apenas este documento. A ajuda (`help()`) e os fóruns oficiais do R também são uma inesgotável fonte de conhecimento. Use sem moderação.

Por fim, porém não menos importante, gostaríamos de reforçar que este documento estará sempre em constante modificação e atualização. Consulte sempre a versão do documento. Como qualquer trabalho, não estamos livres de erros e estes podem (e devem!) ser reportados diretamente aos autores. Sem mais delongas, esperamos que este seja um primeiro passo para você adentrar ao mundo desafiador e fascinante do R.

## Referências

- Adler, J. 2010. *R in a nutshell - a desktop quick reference*. O'Reilly. 609 p.
- Attwood, T.K.; Gisel, A.; Eriksson, N. E. & Bongcam-Rudloff, E. 2011. Concepts, historical milestones and the central place of bioinformatics in modern biology: a European perspective. *Bioinformatics-Trends and Methodologies*, **1**:1-31.
- Batista, J.L.F., Prado, P.I. & Oliveira, A. A. (eds.) 2009. *Introdução ao R - Uma Apostila on-line*. URL: <http://ecologia.ib.usp.br/bie5782>.
- Borcard, D.; Gillet, F. & Legendre, P. 2018. *Numerical Ecology with R*. 2nd ed. New York, Springer-Verlag, 306 p.
- Cox, T.F. & Cox, M.A.A. 2001. *Multidimensional scaling*. 2nd ed. Chapman & Hall, 308 p.
- Crawley, M. J. 2005. *Statistics - an introduction using R*. John Wiley & Sons, 327 p.
- Crawley, M.J. 2012. *The R Book*. 2nd ed. Wiley, 1076 p.
- Erthal, F. 2017. *Estatística multivariada em Tafonomia (Atualística)*. In: R.S. Horodyski & F. Erthal (orgs.) *Tafonomia: métodos, processos e aplicações*, CRV, p. 81-113.
- Ferreira, E.B.; Cavalcanti, P.P. & Nogueira, D.A. 2014. ExpDes: an R package for ANOVA and experimental designs. *Applied Mathematics*, **5**:2952-2958.
- Gotelli, N.J. & Ellison, A.M. 2013. *A primer of Ecological statistics*. 2nd ed. Sunderland, Sinauer, 614 p.
- Jari Oksanen, F. Guillaume Blanchet, Michael Friendly, Roeland Kindt, Pierre Legendre, Dan McGlinn, Peter R. Minchin, R. B. O'Hara, Gavin L. Simpson, Peter Solymos, M. Henry H. Stevens, Eduard Szoecs and Helene Wagner. 2018. *vegan: Community Ecology Package*. <https://CRAN.R-project.org/package=vegan>
- Konzen, E.R.; Recchia, G.H.; Cassieri, F.; Caldas, D.G.G.; Berny Mier y Teran, J.C.; Gepts, P. & Tsai, S.M. 2019. DREB Genes from common bean (*Phaseolus vulgaris* L.) show broad to specific abiotic stress responses and distinct levels of nucleotide diversity. *International Journal of Genomics*, **2019**:9520642.
- Landeiro, V. L. 2011. *Introdução ao uso do programa R*. Disponível em: <https://cran.r-project.org/doc/contrib/Landeiro-Introducao.pdf> [Apostila].
- Legendre, P. & Legendre, L. 2012. *Numerical Ecology*. 3rd ed. New York, Elsevier, 1006 p.
- MacLaren, J.A.; Anderson, P.S.L.; Barrett, P.M. & Rayfield, E.J. 2017. Herbivorous dinosaur jaw disparity and its relationship to extrinsic evolutionary drivers. *Paleobiology*, **43**:15-33.
- Murrell, P. 2006. *R graphics*. Chapman & Hall./CRC. 303 p.
- Owen, W. J. 2019. *The R Guide*. Disponível em: <https://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf>.
- R Development Core Team. 2019. *An Introduction to R*. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf>

- R Development Core Team. 2019. *R language definition*. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.pdf>.
- R Development Core Team. 2019: *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, <https://www.r-project.org/>.
- Ripley, B.; Venables, B.; Bates, D.M.; Hornik, K.; Gebhardt, A.; Firth, D. & Ripley, M.B. 2013. *Package 'mass'*. Cran R.
- Silva, L.J. 2015. *Package 'laercio'*. Cran R.
- Storck, L.; Garcia, D.C.; Lopes, S.J. & Estefanel, V. 2000. *Experimentação vegetal*. Santa Maria: UFSM, 198 p.
- Steven, H.M. 2009. *A Primer of Ecology with R (Use R!)* 1st ed. Springer, 388 p.
- Venables, W.N. & Ripley, B.D. 2002. Random and mixed effects. In: *Modern applied statistics with S*, Springer p. 271-300.
- Verzani, J. 2005. *Using R for introductory statistics*. Chapman & Hall/CRC. 402 p.
- Verzani, J. 2019. *SimpleR – Using R for Introductory Statistics*. Disponível em: <https://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf>.
- Wright, E.S. 2016. Using DECIPHER v2.0 to analyze big biological sequence data in R. *R Journal*, **8**:352-359.
- Yang, Z. & Rannala, B. 2012. Molecular phylogenetics: principles and practice. *Nature Reviews Genetics*, **13**:303-314.