Universidade de São Paulo
Programa de Pós-Graduação em Ecologia

CURSO

Uso da Linguagem R:

Apostila

Paulo Inácio Prado
João Batista
Alexandre Adalardo de Oliveira

diagramação: Sara Mortara

São Paulo, 29 de março de 2016

Apresentação

Esta apostila foi desenvolvida para apoio do curso Uso da Linguagem R em Ecologia. Depois de muitos ciclos de aperfeiçoamento decidimos publicá-la na *internet* como um material para estudo individual.

Ela está em constante aperfeiçoamento, e ficaremos felizes com suas críticas e sugestões. Entre em contato com os editores. Veja também a nossa seção de tutoriais.

Autores

Editores

João Luís Ferreira Batista

- Centro de Métodos Quantitativos, Dep. de Ciências Florestais, Escola de Agricultura Luis de Queiroz da Universidade de São Paulo.
- `email = paste("parsival", "usp.br", sep="@")`

Paulo Inácio K. L. Prado

- Laboratório de Ecologia Teórica, Dep. de Ecologia, Instituto de Biociências da Universidade de São Paulo.
- `email = paste("prado", "ib.usp.br", sep="@")`

Alexandre Adalardo de Oliveira

- Laboratório de Ecologia de Florestas Tropicais, Dep. de Ecologia, Instituto de Biociências da Universidade de São Paulo.
- `email=paste("adalardo","usp.br",sep="@")`

Colaboradores

Cristina Banks-Leite

- Imperial College, London
- `email=paste("crisbanksleite","gmail.com",sep="@")`

Rodrigo Augusto Santinelo Pereira

- Laboratório de Biologia Reprodutiva de Ficus, Dep. de Biologia, Faculdade de Filosofia Ciências e Letras de Ribeirão Preto da Universidade de São Paulo.
- `email=paste("raspereira","yahoo.com.br",sep="@")`

Citação

Batista, J.L.F., Prado, P.I. e Oliveira, A. A. (Eds.) 2016. Introdução ao R - Uma Apostila *on-line*. **URL**: http://ecologia.ib.usp.br/bie5782.

bibtex:

```
@BOOK{Batista2016,
  title = {Introdução ao R - Uma Apostila online},
  publisher = {Universidade de São Paulo},
  year = {2009},
  editor = {Batista, J.L.F., Prado P.I., Oliveira, A. A.},
  url = {http://ecologia.ib.usp.br/bie5782}
}
```

Sumário

1 Introdução

Links externos para o material do curso *Uso da Linguagem R* associado ao material aqui apresentado:

- Apostila
- Tutorial
- Exercícios

1.1 O R e sua Filosofia de Trabalho

O **Manual do R** (*Venables et al.*, 2007) define o R como um ambiente de programação com um conjunto integrado de ferramentas de software para manipulação de dados, cálculos e apresentação gráfica.

Como ambiente, entende-se um sistema coerente e totalmente planejado.

O R não é um software do tipo aplicativo, a preocupação não é com amigabilidade, mas com flexibilidade e capacidade de manipulação de dados e realização de análises.

Notar que na definição não se usou o termo *Estatística*. Embora a maioria das pessoas usem o R como um software estatístico, seus definidores (*Venables et al.*, 2007) preferem definí-lo como um ambiente onde muitas técnicas estatísticas, clássicas e modernas, podem ser implementadas, entre outras coisas. Algumas dessas técnicas estão implementadas no ambiente básico do R (**R base**), mas muitas estão implementadas em pacotes adicionais (**packages**).

1.2 Breve Histórico da Linguagem S e do R

- Tudo começou com a *Linguagem e o Ambiente S* desenvolvido por pesquisadores do *AT&T Bell Labs* na década de 80. O S começou no sistema operacional UNIX e já era uma linguagem e ambiente para análise de dados e criação de gráficos. A base da linguagem S é apresentada no livro de *Becker et al.* (1988), sendo que este é ainda uma referência básica na linguagem S.

- No início da década de 90, a linguagem S foi incrementada com uma notação para modelos estatísticos que facilitou a construção de modelos. Essa nova abordagem é apresentada em detalhes no livro de Chambers and Hastie (1992), tendo resultado numa significativa economia de esforço de programação para modelagem estatística de dados.

- No final da década de 90, foi implementada uma revisão na linguagem S que a tornou uma linguagem de alto padrão totalmente baseada em programação por

objetos. Essa é versão atual da linguagem S implementada no R, sendo apresentada em detalhes por *Chambers* (1998).

- O ambiente R foi desenvolvido baseado na linguagem S, no final da década de 90 início dos anos 2000. A sua estrutura de código aberto (que vem da linguagem S) e de software público e gratuito atraiu um grande número de desenvolvedores, sendo que há hoje inúmeros pacotes para o R.

PARA SABER UM POUCO MAIS

Execute estes comandos no R:

```
contributors()
citation()
```

1.3 Página Oficial do R

Esta é a referência básica para usuários de R, que inclui programas para download, listas de discussão, e muita documentação e ajuda: http://www.r-project.org/.

Explore as seções, começando pelas FAQ. Boa parte do que tratamos aqui está na seção 2 (**R Basics**) das FAQ, além de várias outras informações úteis.

A página tem uma grande lista de documentação, na seção "Documentation". Há um wiki em construção, e ainda um pouco irregular, mas com boas seções, como a de dicas. Além disso, há excelentes manuais introdutórios feitos por vários voluntários na seção de "Contributed documentation".

Exercício: *Uma Sessão Introdutória ao R*

Para se acostumar com a linguagem, siga as instruções deste tutorial, que está na (página oficial do R)

1.4 Referências

Becker, R.A.; Chambers, J.M.; Wilks, A.R.; *The New S Language: a programming environment for data analysis and graphics*. Pacific Grove: Wadsworth & Brooks, 1988.

Chambers, J.M. *Programming with Data*. New York: Springer-Verlag, 1998.

Chambers, J.M.; Hastie, T.J.; *Statistical Models in S*. Pacific Grove: Wadsworth & Brooks, 1992.

Venables, W.N.; Smith, D.M.; R Development Core Team *An Introduction to R - Notes on R: a programming environment for data analysis and graphics*. Version 2.5.1 (2007-06-27).

Matéria sobre o R no New York Times, Janeiro de 2009: matéria on-line, versão pdf

Publish your code - it is good enough, por Nick Barnes: um ensaio que estimula pesquisadores a partilharem seus códigos de análise de dados, mesmo que pareçam imperfeitos. artigo na Nature News, 2010.

2 Primeiros Passos

2.1 Instalação do R

Os arquivos necessários para instalar o ambiente R estão disponíveis gratuitamente no sítio oficial http://www.r-project.org/.

A página oficial do R é a referência básica para seus usuários e desenvolvedores, onde você também encontra instruções de instalação, listas de discussão, tutoriais, documentação e muitas informações úteis.

2.2 Iniciando o R

Em ambiente UNIX (como o Linux, por exemplo), podemos iniciar o R a partir do interpretador de comandos (*shell*) digitando o comando `R`:

```
parsival@jatai $ R
```

Já no sistema MS-Windows, é necessário clicar no ícone apropriado (no desktop) ou buscar o programa a partir do menu **Iniciar**.

Independentemente de como inicia, o R apresenta uma tela com (aproximadamente) a seguinte forma:

```
R version 2.7.0 (2008-04-22)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R é um software livre e vem sem GARANTIA ALGUMA.
Você pode redistribuí-lo sob certas circunstâncias.
Digite 'license()' ou 'licence()' para detalhes de distribuição.

R é um projeto colaborativo com muitos contribuidores.
Digite 'contributors()' para obter mais informações e
'citation()' para saber como citar o R ou pacotes do R em publicações.

Digite 'demo()' para demonstrações, 'help()' para o sistema on-line de
    ajuda,
ou 'help.start()' para abrir o sistema de ajuda em HTML no seu navegador.
Digite 'q()' para sair do R.

[Área de trabalho anterior carregada]

>
```

2.2.1 A Linha de Comando

O R é uma linguagem interativa, ou seja, que permite ao usuário enviar um comando por vez e receber o resultado[1]. Para isso, usamos a linha de comando, que tem o sinal ">"quando o R está pronto para receber um comando.

Os outros dois estados da linha de comando são o de execução e o de espera para a conlusão do comando. No modo de execução não é exibido nenhum sinal e não é possível digitar outro comando. Você só perceberá isso se der um comando que tenha um tempo de execução muito longo.

O estado de espera ocorre quando o usuário envia um comando incompleto, o que é indicado por um sinal de "+":

```
>
> log(2
+ )
[1] 0.6931472
>
```

Na primeira linha, não fechamos os parênteses da função `log` e demos *enter*. O R responde com o sinal de espera (+), indicando que o comando está incompleto. Digitando o parêntese que falta e apertando a tecla *enter* novamente o R retorna o resultado do comando, precedido de um índice numérico.[2]

2.2.2 Sintaxe Básica dos Comandos

Um comando no R em geral inclui uma ou mais funções, que seguem a seguinte sintaxe:

função(argumento1 = valor, argumento2 = valor , ...)

Como nos exemplos abaixo:

```
> plot(x=area,y=riqueza)
> plot(area, riqueza)
> plot(area,riqueza,xlab="Área (ha)", ylab="Riqueza")
```

- No primeiro caso, o valor de cada argumento usado está explicitado. O argumento `x` da função `plot` é a variável independente, e o argumento `y` é a variável dependente.
- Se o nome dos argumentos é omitido, como no segundo caso, o R usa o critério de posição: o primeiro valor é atribuído ao primeiro argumento, o segundo valor ao segundo argumento, e assim por diante. Como os dois primeiros argumentos da função plot são `x` e `y`, o segundo comando acima equivale ao primeiro.

[1] é possível também de executar um lote de comandos, mas neste wiki trabalharemos apenas com o modo interativo.

[2] o significado desse índice ficará claro na seção sobre indexação. Por enquanto basta saber que os resultados do R são precedidos por um indicador numérico entre colchetes

- Os dois critérios podem ser combinados, como no terceiro comando: como x e y são os dois primeiros argumentos, não é preciso declará-los. Como os outros dois argumentos que se deseja usar (`xlab` e `ylab`) não vêm em seguida, é preciso declará-los.

Mais detalhes na seção sobre funções.

2.2.3 Criação de Objetos: Atribuição

Você pode "guardar"o resultado de um comando com a operação de *atribuição*, que tem a sintaxe:

objeto *recebe* **valor**

Há dois operadores que atribuem valores a um objeto dessa maneira:

- Sinal de menor seguido de hífen, formando uma seta para a esquerda: <-
- Sinal de igual: =

Uma forma de atribuição menos usada é:

valor *atribuído a* **objeto**

Nesse caso, o sinal de atribuição é o hífen seguido de sinal de maior, formando uma seta para direita: ->

Há, portanto, três maneiras de guardar os resultados de um comando em um objeto:

```
> a <- sqrt(4)
> b = sqrt(4)
> sqrt(4) -> c
```

Para exibir o conteúdo de um objeto, basta digitar seu nome

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

Se a atribuição é para um objeto que não existe, esse objeto é criado. **Mas cuidado**: se já há um objeto com o mesmo nome na sua área de trabalho, seus valores serão substituídos:

```
> a <- sqrt(4)
> a
[1] 2
> a <- 10^2
> a
[1] 100
```

2.2.4 Mensagens de Erro e de Avisos

Como em qualquer linguagem, o R tem regras de sintaxe e grafia. Mas contrário das linguagens humanas, mesmo um pequeno erro torna a mensagem incompreensível para o R, que então retorna uma mensagem de erro:

```
> logaritmo(2)
Erro: não foi posssível encontrar a função "logaritmo"
> log(2))
Erro: unexpected ')' in "log(2))"
> log(2, basse=10)
Erro: unused argument(s) (basse = 10)
> log(2, base=10)
[1] 0.30103
```

Em outros casos, o comando pode ser executado, mas com um resultado que possivelmente você não desejava. O R cria mensagens de alerta para os casos mais comuns desses resultados que merecem atenção :

```
> log(-2)
[1] NaN
Warning message:
In log(-2) : NaNs produzidos
```

2.3 Para Sair

Para sair do R, a forma mais fácil é usar o comando `q` (do inglês *quit*). Nesse caso o R, lhe pergunda se você deseja *salvar* (gravar) sua sessão de trabalho.

```
> q()
Save workspace image? [y/n/c]:
```

As opções são: `y = yes, n = no, c = cancel`.

2.4 Como o R Guarda os Dados?

O que significa a pergunta feita quando damos o comando `q()`?

A resposta passa por três conceitos importantíssimos, que são o **diretório de trabalho**, a **sessão** e a **área virtual de trabalho (*workspace*)**.

Cada vez que você inicia o R, dizemos que se inicia uma **sessão**.

O diretório a partir do qual você iniciou o R é o **diretório de trabalho** dessa sessão. Para verificar seu diretório de trabalho, use o comando `getwd`[3]:

```
> getwd()
[1] "/home/paulo/work/Pos_grad/Eco_USP/cursoR"
```

Para alterar o diretório de trabalho há a função `setwd`:

[3] acrônimo de "get working directory"

```
> setwd("/home/paulo/work/treinos_R/")
> getwd()
[1] "/home/paulo/work/treinos_R"
```

A sessão iniciada está ligada a uma área de trabalho particular chamada de **workspace**.

Tudo o que você faz durante uma sessão (leitura de dados, cálculos, análises estatísticas) é realizado no **workspace**.

Mas o **workspace permanece na memória do computador**. Somente quando você dá o comando de sair (`q()`) é que o R lhe pergunta se você deseja **gravar** o seu **workspace**. Se você responde 'y', o R grava um arquivo chamado `.RData`[4] em seu **diretório de trabalho**. Na próxima vez que o R for chamado desse diretório, o conteúdo do arquivo `.RData` será carregado para o "workspace".

2.4.1 Aprenda este Comando para Não Perder Seu Trabalho

Se acontecer do computador ser desligado durante uma sessão do R, tudo que foi feito será perdido!!! Para evitar isso, é interessante gravar com frequência o **workspace** utilizando o comando `save.image()`:

```
> save.image()
>
```

Por *default*, o R gravará o workspace no arquivo `.RData`, e quando você reiniciar uma sessão, o R automaticamente **carrega** esse arquivo. Mas você pode salvar em outro arquivo utilizando o **argumento** `file` da função:

```
> save.image(file="minha-sessao-introdutoria.RData")
>
```

Como o R carrega automaticamente apenas o arquivo `.RData` que está no diretório de trabalho, caso deseje carregar outros arquivos você deverá usar a função `load`:

```
># Carrega um arquivo de workspace no mesmo diretório
> load(file="minha-sessao-introdutoria.RData")
># Carrega o arquivo default de workspace de outro diretório:
> load(file="/home/paulo/work/treinos_R/.RData")
```

No código acima podemos ver o símbolo "#"seguido de comentários que explicam o que a função está fazendo. Quando você coloca "#", o R irá ignorar o que vem depois do símbolo na linha. Ou seja, se você copiar uma linha começando com "#"no console e pedir para o R executar a mesma, não vai acontecer nada. Este símbolo é muito útil para comentar o seu código.

Se você quiser salvar apenas alguns objetos (digamos, os resultados das suas análises), você pode usar o comando `save`:

[4] este é um arquivo oculto, pois seu nome inicia-se com um ponto

```
> save(modelo1, file="meu_modelo.RData")
> save(dados, modelo1, modelo2, file="meus_modelos.RData")
```

Tome cuidado com a sintaxe do comando `save`: ele aceita o nome de vários objetos que existem no seu workspace e um nome de arquivo, que deve ser passado com `file=`.

O comando `save` aceita o resultado de outros comandos. Por exemplo, o código abaixo equivale ao comando save.image():

```
> save (list=ls(), file="tudo.RData")
```

2.4.2 Aprenda este Procedimento para Organizar Seu Trabalho

Crie um diretório de trabalho para cada análise, ou mesmo para versões diferentes da mesma análise, e chame o R desse diretório. Fazendo isso você recupera seu trabalho de maneira simples, bastando salvar as alterações regularmente com o comando `save.image()` e confirmar a gravação das alterações ao encerrar a sessão.

Ao contrário do que você pode estar acostumado(a), não é uma boa idéia manter vários arquivos com diferentes versões dos dados ou análises em um mesmo diretório. Os usuários de R em geral mantém o padrão da linguagem, de um único arquivo *default* por análise, o `.RData`, criando quantos diretórios forem necessários para organizar o trabalho.

2.5 Gerenciando a Área de Trabalho

2.5.1 Listando Objetos

O comando `ls` lista todo o conteúdo da área de trabalho, se não é fornecido argumento:

```
> ls()
[1] "consoantes" "CONSOANTES" "vogais" "VOGAIS"
```

A função `ls` possui argumentos que podem refinar seus resultados, consulte a ajuda para os detalhes.

2.5.2 Apagando Objetos

O comando `rm` apaga objetos da área de trabalho:

```
> ls()
[1] "consoantes" "CONSOANTES" "vogais"     "VOGAIS"
> rm(consoantes)
> ls()
[1] "CONSOANTES" "vogais"     "VOGAIS"
```

Consulte a ajuda da função `rm` para seus argumentos.

2.5.3 Exercícios

2.1. Exercício para Usuários Windows: Diretório de Trabalho

1. Crie um diretório para seus exercícios da disciplina.
2. Chame o R, clicando no ícone da área de trabalho ou na barra de tarefas.
3. Verifique o seu diretório de trabalho.
4. Mude o diretório de trabalho para o diretório que você criou.
5. Verifique o conteúdo da área de trabalho.
6. Carregue o arquivo letras.RData (apagar extensão .pdf).
7. Verifique novamente sua área de trabalho.
8. Saia do R, tomando o cuidado de salvar sua área de trabalho.
9. Repita os passos 2 a 5.

Pergunta: Que problemas você percebeu? Há uma maneira de iniciar o R no windows que evite esses problemas?

2.6 Como Conseguir Ajuda no R

No R é essencial aprender a procurar auxílio e manuais sozinho. Após um aprendizado inicial, não há meio de evoluir no conhecimento do ambiente R se não se buscar os ***helps*** que ele possui.

2.6.1 O Comando "help"

Para conseguir ajuda sobre um comando pode ser usada a função `help`:

```
> help(mean)
```

ou então o operador de ajuda que é o sinal de interrogação '?':

```
?mean
```

No caso se buscar ajuda sobre um **operador** (símbolo para operações aritméticas e algébricas) devemos utilizar as aspas duplas:

```
> help("+")
> ?"*"
```

Ao utilizar esses comandos (help e ?) o R entra num modo interativo diferente: **help interativo**. Ele apresenta uma ***help page*** que você pode examinar usando os seguintes comandos de teclado:

- tecla de espaço = um página para frente;
- tecla "f"= uma página para frente;
- tecla "b"= uma página para trás;
- tecla enter = uma linha para frente;
- tecla j = uma linha para frente;
- tecla k = uma linha para trás;
- tecla q = sai do modo **help interativo** e retorna à linha de comando.

No modo **help interativo** também é possível se fazer uma busca por uma palavra (*string*) usando a tecla '/', na forma:

```
/string
```

Executado um comando de busca temos dois novos comandos válidos:

- tecla n = próxima string, e
- tecla N = string anterior.

A Tela de Ajuda O conteúdo do **help interativo** pode parecer árido à primeira vista, mas é extremamente informativo, assim que nos acostumamos com ele. As seções são padronizadas, e seguem sempre a mesma ordem:

- Um cabeçalho com o nome da função, o pacote do R à qual pertence, e a classe do documento de ajuda
- O nome completo da função
- A sintaxe da função, que pode estar especificada para diferentes tipos de dados ou métodos
- A explicação de cada um dos argumentos da função
- O valor retornado pela função
- Referências bibliográficas
- Indicação de outras funções relacionadas

- Exemplos (colar esses comandos no R é uma das melhores maneiras de entender uma função).

Abaixo o conteúdo da ajuda para a função `mean`:

```
mean                    package:base                    R Documentation

Arithmetic Mean

Description:

     Generic function for the (trimmed) arithmetic mean.

Usage:

     mean(x, ...)

     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:

       x: An R object.  Currently there are methods for numeric data
          frames, numeric vectors and dates.  A complex vector is
          allowed for 'trim = 0', only.

    trim: the fraction (0 to 0.5) of observations to be trimmed from
          each end of 'x' before the mean is computed.

   na.rm: a logical value indicating whether 'NA' values should be
          stripped before the computation proceeds.

     ...: further arguments passed to or from other methods.

Value:

     For a data frame, a named vector with the appropriate method being
     applied column by column.

     If 'trim' is zero (the default), the arithmetic mean of the values
     in 'x' is computed, as a numeric or complex vector of length one.
     If any argument is not logical (coerced to numeric), integer,
     numeric or complex, 'NA' is returned, with a warning.

     If 'trim' is non-zero, a symmetrically trimmed mean is computed
     with a fraction of 'trim' observations deleted from each end
     before the mean is computed.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole.
```

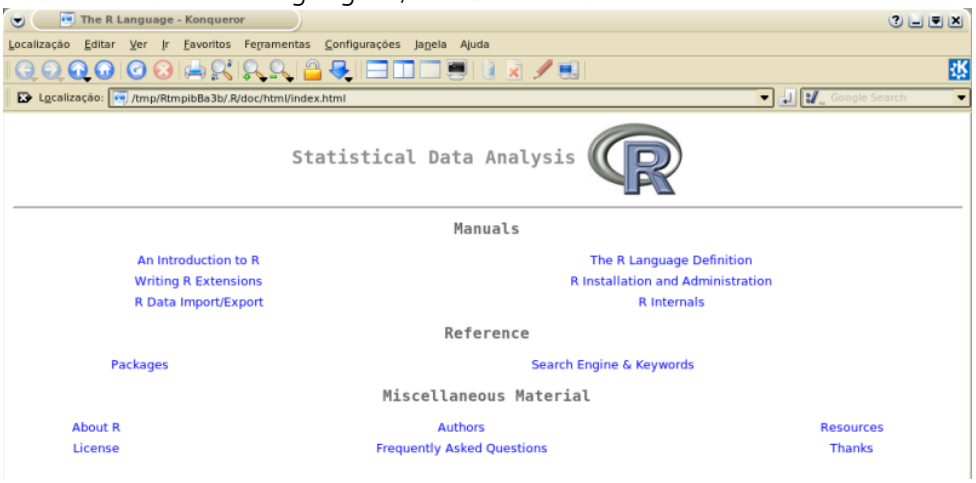
```
See Also:

     'weighted.mean', 'mean.POSIXct'

Examples:

     x <- c(0:10, 50)
     xm <- mean(x)
     c(xm, mean(x, trim = 0.10))

     mean(USArrests, trim = 0.2)
```

2.6.2 Ajuda em Hipertexto: "help.start"

Para os que preferem trabalhar com ajuda em hipertexto (html), há o comando `help.start()`:

```
> help.start()
Making links in per-session dir ...
If 'sensible-browser' is already running, it is *not* restarted, and
    you must switch to its window.
Otherwise, be patient ...
>
```

Após esta mensagem, um portal de ajuda em hipertexto será aberto no navegador de internet padrão do sistema. A partir dessa página inicial é possível fazer pesquisas por palavras-chave, consultar funções por pacote ou por ordem alfabética, obter textos introdutórios sobre a linguagem, e muito mais.



2.6.3 Pesquisa por Palavras-Chave na Ajuda

Outro comando muito útil é o `apropos`. Ele possibilita sabermos quais funções do R tem no nome uma certa palavra (*string*):

```
> apropos(plot)
 [1] "biplot"                "interaction.plot"      "lag.plot"
 [4] "monthplot"             "plot.density"          "plot.ecdf"
 [7] "plot.lm"               "plot.mlm"              "plot.spec"
[10] "plot.spec.coherency"   "plot.spec.phase"       "plot.stepfun"
[13] "plot.ts"               "plot.TukeyHSD"         "preplot"
[16] "qqplot"                "screeplot"             "termplot"
[19] "ts.plot"               "assocplot"             "barplot"
[22] "barplot.default"       "boxplot"               "boxplot.default"
[25] "cdplot"                "coplot"                "fourfoldplot"
[28] "matplot"               "mosaicplot"            "plot"
[31] "plot.default"          "plot.design"           "plot.new"
[34] "plot.window"           "plot.xy"               "spineplot"
[37] "sunflowerplot"         "boxplot.stats"         ".__C__recordedplot"
```

Para pesquisas mais complexas e refinadas há ainda a função **help.search()**. Por exemplo, para pesquisar funções que tenham a palavra "skew"no título:

```
> help.search(field="title","skew")

Help files with title matching 'skew' using regular expression
matching:

skewnormal1(VGAM)        Univariate Skew-Normal Distribution Family
                         Function
snorm(VGAM)              Skew-Normal Distribution
k3.linear(boot)          Linear Skewness Estimate

Type 'help(FOO, package = PKG)' to inspect entry 'FOO(PKG) TITLE'.
```

2.6.4 Busca de Exemplos e de Argumentos das Funções

Para obter apenas os exemplos que estão na ajuda de uma função, use a função **example**:

```
> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.1))
[1] 8.75 5.50

mean> mean(USArrests, trim = 0.2)
  Murder   Assault  UrbanPop     notGood
    7.42    167.60     66.20     20.16
```

Muitas vezes precisamos apenas nos lembrar dos argumentos de uma função. Para isso, use a função **args**, que retorna os argumentos de uma função:

```
> args(chisq.test)
function (x, y = NULL, correct = TRUE, p = rep(1/length(x), length(x)),
    rescale.p = FALSE, simulate.p.value = FALSE, B = 2000)
NULL
```

Argumentos com valores atribuídos são os valores *default* da função. Por exemplo, por *default* a função de teste de Qui-quadrado estima a significância pela distribuição de Qui-quadrado e não por randomização (argumento `simulate.p.value=FALSE`).

2.6.5 Exercícios

Exercício 2.2. Use a Ajuda para Conhecer Argumentos das Funções

1. Execute o R, usando o diretório de trabalho criado no exercício anterior.
2. Use a função `load` para carregar o arquivo bichos.RData (apagar extensão .pdf) no *workspace*.
3. Consulte a ajuda da funções `rm` e `ls` para descobrir como apagar apenas os objetos cujos nomes começam com "temp".

2.7 Pacotes

Pacotes são conjuntos de funcionalidades (funções e dados) distribuídos em conjunto para realizar tarefas específicas. Por exemplo, o pacotes **vegan** carrega na sua área de trabalho (deixa disponível para uso) um conjunto de ferramentas para análises de dados de ecologia de comunidade. Para usar os pacotes disponíveis no R [5] é necessário entender as diferenças entre **baixar** (download) o pacote do repositório e **carregar** em sua área de trabalho. Para baixar algum pacote disponível no repositório CRAN do R é necessário utilizar o comando *install.packages()* com o nome do pacote entre dentro do parenteses[6].

```
install.packages("vegan")
```

Outra forma é usar o menu da interface gráfica e selecionar. Siga as instruções: (1) selecione o repositório mais próximo (p.ex: *Brazil(SP1)*) e em seguida navegue na barra de pacotes e selecione o que deseja. Se não houver nenhuma mensagem de erro, significa que o download do pacote foi realizado com sucesso.

Caso o pacote esteja instalado ele aparecerá entre os hiperlinks da página de ajuda hipertexto (**??**) da função *help.start()*. Entre na página do pacote e navegue pelas

[5] "Currently, the CRAN package repository features 5217 available packages.-- *Alexandre Adalardo de Oliveira 2014/02/17 14:31*

[6] a princípio todas as palavras que escrevemos sem aspas no R ele busca como sendo objetos presentes em nossa área de trabalho ou pacotes carregados ou instalados

opções e funções que forem de seu interesse. Escolha uma função (decorana) e em seguida tente apresentar o ajuda dela pelo R:

```
help.start()
help(decorana)
```

A mensagem (ou algo similar): *"No documentation for 'decorana' in specified packages and libraries..."*, significa que a sua sessão do R não encontrou a documentação referente a função, apesar do pacote estar instalado em nosso computador. Isso aconteceu porque não carregamos a função em nossa área de trabalho, para cada projeto, precisamos carregar aqueles pacotes que vamos necessitar (normalmente nas primeiras linhas de comando do nosso código):

```
library(vegan)

example(vegan)
```

Podemos imaginar a nossa sessão do R como uma bancada de trabalho em uma oficina, cercada por vários armários que contém as ferramentas que precisamos para realizar uma tarefa. Dependendo da tarefa que vamos realizar (arrumar uma moto, construir uma cadeira...) abrimos os armários que contem as ferramentas necessárias à tarefa desejada e apenas esses (função *library()*). Caso não tenhamos as ferramentas necessárias para uma tarefa específica (consertar um relógio), precisamos ir na loja de ferramentas (repositório) e comprar conjunto de ferramentas de relojoeiro (função *install.packates("watch")* que vem em um armário que colocamos ao lado dos outros em nossa oficina.

2.8 Particularidades dos Comandos no R

Como em toda linguagem, o R tem algumas regras básicas de sintaxe e grafia:

1. O nome de comandos e objetos no R pode ser compostos de:
    - letras (minúsculas ou maiúsculas),
    - números, e
    - o ponto (.).
2. Evite qualquer outro caracter especial, incluindo *o espaço em branco*.
3. O nome *não pode ser iniciado* por um número.
4. O R é sensível a caixa: o comando `q` é diferente do comando `Q` (que não existe!!!).

Para que um comando seja **executado** pelo R é necessário ser acompanhado de parênteses '()'. Compare esse comando

```
> q()
Save workspace image? [y/n/c]: c
>
```

com o comando

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
>
```

O comando sem os parênteses é na verdade o **nome do comando**. Sendo o R um software de código aberto, toda vez que se digita o nome de um comando, ele **não executa** o comando mas **mostra o conteúdo** do comando (o código).

3 Funções Matemáticas e Estatísticas

Links externos para o material referente a esse tópico no wiki da disciplina:

- Tutorial
- Exercícios
- Apostila

3.1 O R como uma Calculadora Fora do Comum

3.1.1 Operações Aritméticas Básicas

A linha de comando do R funciona como uma calculadora. Todas operações aritméticas e funções matemáticas principais estão disponíveis. Exemplo:

```
> 4 + 9
[1] 13
> 4 - 5
[1] -1
> 4 * 5
[1] 20
> 4 / 5
[1] 0.8
> 4^5
[1] 1024
>
```

A notação básica de operações algébricas, como a aplicação hierárquica de parênteses, também pode ser utilizada:

```
> (4 + 5 ) * 7 - (36/18)^3
[1] 55
> (2 * ( 2 * ( 2 * (3-4))))
[1] -8
>
```

Note que somente os parênteses podem ser utilizados nas expressões matemáticas. As chaves ("{}") e os colchetes ("[]") têm outras funções no R:

```
> (2 * { 2 * [ 2 * (3-4)]})
Error: syntax error in "(2 * { 2 * ["
>
```

Por que o R é uma calculadora **fora do comum** ? Experimente fazer a seguinte operação matemática na sua calculadora:

```
> 1 - (1 + 10^(-15))
```

3.1.2 Funções Matemáticas Comuns

As funções matemáticas comuns também estão disponíveis e podem ser aplicadas diretamente na linha de comando:

```
> sqrt(9)   # Raiz Quadrada
[1] 3
> abs( - 1 )  # Módulo ou valor absoluto
[1] 1
> abs( 1 )
[1] 1
> log( 10 )   # Logaritmo natural ou neperiano
[1] 2.302585
> log( 10, base = 10) # Logaritmo base 10
[1] 1
> log10(10)   # Também logaritmo de base 10
[1] 1
> log( 10, base = 3.4076) # Logaritmo base 3.4076
[1] 1.878116
> exp( 1 )       # Exponencial
[1] 2.718282
>
```

As funções trigonométricas:

```
> sin(0.5*pi)     # Seno
[1] 1
> cos(2*pi)       # Coseno
[1] 1
> tan(pi)         # Tangente
[1] -1.224647e-16
```

```
>
> asin(1)       # Arco seno (em radianos)
[1] 1.570796
> asin(1) / pi * 180
[1] 90
>
> acos(0)      # Arco coseno (em radianos)
[1] 1.570796
> acos(0) / pi * 180
[1] 90
> atan(0)    # Arco tangente (em radianos)
[1] 0
> atan(0) / pi * 180
[1] 0
>
```

Funções para arredondamento:

```
> ceiling( 4.3478 )
[1] 5
> floor( 4.3478 )
[1] 4
> round( 4.3478 )
[1] 4
> round( 4.3478 , digits=3)
[1] 4.348
> round( 4.3478 , digits=2)
[1] 4.35
>
```

Funções matemáticas de especial interesse estatístico:

```
> factorial( 4 )          # Fatorial de 4
[1] 24
> choose(10, 3)           # Coeficientes binomiais: combinação de 10 3-a-3
[1] 120
>
```

3.1.3 Criando Variáveis com Atribuição

Mais do que simples operações aritméticas, o R permite que executemos operações **algébricas** operando sobre variáveis pré-definidas.

Para definir uma variável, basta escolher um nome (*lembre-se das regras de nomes no R*) e atribuir a ela um valor:

```
>
> a = 3.6
> b = sqrt( 35 )
> c = -2.1
> a
[1] 3.6
```

```
> b
[1] 5.91608
> c
[1] -2.1
>
> a * b / c
[1] -10.14185
> b^c
[1] 0.02391820
> a + exp(c) - log(b)
[1] 1.944782
>
> a - b * c / d
Error: object "d" not found
>
```

Não esqueça de definir as variáveis previamente!!

Exercício 3.1. ***Estimador de Pollard***

Pollard (1971) propôs o seguinte estimador para estimar a densidade no método de quadrantes:

$$\widehat{N}_p = \frac{4(4n-1)}{\pi \sum_{i=1}^{n} \sum_{j=1}^{4} r_{ij}^2}$$

onde, `rij` é a distância de árvore do quadrante *j* no ponto *i* ao centro do ponto quadrante e *n* é o número de pontos quadrantes.
A variância desse estimador é:

$$\mathbf{Var}\{\widehat{N}_p\} = \frac{\widehat{N}_p}{4n-2}$$

Imagine que foram amostrados 30 quadrantes, e que o valor da soma do quadrado das distâncias de cada árvore ao centro de seu quadrante foi de:

$$\sum_{i=1}^{30} \sum_{j=1}^{4} r_{ij}^2 = 2531,794$$

1. Qual a densidade estimada?
2. Qual a variância?

Exercício 3.2. *Área transversal de uma Árvore*

A área transversal de uma árvore é calculada assumindo que a secção transversal do tronco à altura do peito (1,3m) é perfeitamente circular. Se o diâmetro à altura do peito (DAP) de uma árvore for 13.5cm, qual a área transversal?
Se uma árvore possui três fustes com DAPs de: 7cm, 9cm e 12cm, qual a sua área transversal?

Exercício 3.3. *Área transversal de uma Árvore (Revisitado)*

Se uma árvore possui três fustes com DAPs de: 7cm, 9cm e 12cm, qual o diâmetro (único) que é equivalente à sua área transversal?

Exercício 3.4. *Cálculo da Biomassa de Árvores do Cerrado*

O modelo alométrico de biomassa ajustado para árvores do Cerradão estabele que a biomassa é dada pela expressão:

$$\widehat{b} = \exp(-1,7953)\, d^{2,2974}$$

onde `b` é a biomassa em *kg* e `d` é o DAP em *cm*.
Já um outro modelo para biomassa das árvores na mesma situação tem a forma:

$$\widehat{\ln(b)} = -2,6464 + 1,9960 \ln(d) + 0,7558 \ln(h)$$

onde `h` é a altura das árvores em *m*.
Para uma árvore com DAP de 15*cm* e altura de 12*m*, os modelos resultarão em estimativas muito distintas?

3.5. Exercício Conceitual: Criando Variáveis com Nomes Reservados

O que acontece se você criar uma variável com o nome `pi`? Por exemplo,

```
> pi = 10
```

O que acontece com a constante *pi*?
E se for criada uma constante de nome `sqrt`? O que acontece com a função raíz quadrada (`sqrt()`)?
DICA: O que faz a função `search`, no comando:

```
> search()
```

3.1.4 Mantendo a Coerência Lógica-Matemática

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

```
> 1/0
[1] Inf
> -5/0
[1] -Inf
> 500000000000000000/Inf
[1] 0
> 0/0
[1] NaN
> Inf/Inf
[1] NaN
> log(0)
[1] -Inf
> exp(-Inf)
[1] 0
> sqrt(Inf)
[1] Inf
> sqrt( - 1 )
[1] NaN
Warning message:
NaNs produced in: sqrt(-1)
> 2 * NA
[1] NA
> 2 * NaN
[1] NaN
> NA / 10
[1] NA
> NaN / -1
[1] NaN
>
```

Note que determinadas **palavras** (além do nome das funções) estão reservadas no R, pois são utilizadas com significado especial:

- `pi` - constante <pi = 3.141593 ;
- `Inf` - infinito;
- `NaN` - indeterminado (Not a Number), normalmente resultado de uma operação matemática indeterminada;
- `NA` - indeterminado (Not Available), normalmente caracterizando uma observação perdida (*missing value*).

Na operações matemáticas, `NaN` e `NA` atuam sempre como **indeterminado**.

3.6. Exercício Conceitual: O que é uma Observação Perdida

Como se caracteriza uma **observação perdida**?
Quando o diâmetro de uma árvore deve ter o valor **zero** ou o valor **NA**?
E o peso de um animal? E a biomassa de uma floresta? E a espécie de uma ave?

3.2 O R como uma Calculadora Vetorial

3.2.1 Criação de Vetores

O R, e a linguagem S, foram criados para operar não apenas *número-a-número* como uma calculadora convencional.

O R é um ambiente **vetorial**, isto é, quase todas suas operações atuam sobre um *conjunto de valores*, que genericamente chamaremos de vetores[7].

Uma definição mais detalhada dos vetores está na seção sobre manipulação de dados. Aqui fornecemos apenas algumas definições e funções importantes para compreender as operações numéricas com vetores.

Concatenação de Elementos em um Vetor: a Função c Para criar um vetor, podemos usar a função c (c = colar, concatenar). Essa função simplesmente junta todos os argumentos dados a ela, formando um vetor:

```
> a = c(1, 10, 3.4, pi, pi/4, exp(-1), log( 2.23 ), sin(pi/7) )
> a
[1]  1.0000000 10.0000000  3.4000000  3.1415927  0.7853982  0.3678794
   0.8020016  0.4338837
>
```

Criação de Sequências: Operador : e Função seq Para criar vetores de números com intervalo fixo unitário (intervalo de 1) se utiliza o *operador seqüencial* (:):

```
> b = 1:8
> b
[1] 1 2 3 4 5 6 7 8
> c = 20:32
> c
 [1] 20 21 22 23 24 25 26 27 28 29 30 31 32
> d = 2.5:10
> d
[1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
```

[7] No R, "vetores"são uma classe de objetos definida simplesmente como conjuntos de elementos de um mesmo tipo. Os vetores do R não correspondem a vetores de valores da álgebra matricial, para os quais há outra classe de objetos, que é "matrix"

Uma forma mais flexível de criar seqüências de números (inteiros ou reais) é usando a função `seq`:

```
> seq(10, 30)
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
> seq(10, 30, by=2)
 [1] 10 12 14 16 18 20 22 24 26 28 30
> seq(1.5, 7.9, length=20)
 [1] 1.500000 1.836842 2.173684 2.510526 2.847368 3.184211 3.521053
     3.857895
 [9] 4.194737 4.531579 4.868421 5.205263 5.542105 5.878947 6.215789
     6.552632
[17] 6.889474 7.226316 7.563158 7.900000
```

Vetores de Valores Repetidos: Função rep Também é fácil criar uma seqüência de números repetidos utilizando a função `rep`

```
> rep(5, 3)
[1] 5 5 5
> rep(1:5, 3)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> rep(1:5,each=3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
>
```

Exercício 3.7. Palmeira com Muitos Fustes I

Uma palmeira perfilhada possui 10 fustes com os seguintes diâmetros: 5, 6, 7, 5, 10, 11, 6, 8, 9 e 7.
Crie um vetor `dap` com os diâmetros acimas e uma seqüência que enumera os fustes.

3.2.2 Vetores: Operações Matemáticas

Todas operações matemáticas aplicadas sobre um vetor, serão aplicadas sobre cada elemento desse vetor:

```
> 2 * a
[1]  2.0000000 20.0000000  6.8000000  6.2831853  1.5707963  0.7357589
     1.6040032
[8]  0.8677675
> sqrt( a )
[1] 1.0000000 3.1622777 1.8439089 1.7724539 0.8862269 0.6065307 0.8955454
[8] 0.6586985
>
> log( a )
[1]  0.0000000  2.3025851  1.2237754  1.1447299 -0.2415645 -1.0000000
    -0.2206447
```

```
[8] -0.8349787
>
```

Se as variáveis que trabalhamos são vetores, operações matemáticas entre variáveis serão realizadas pareando os elementos dos vetores:

```
> a* b
[1]  1.000000 20.000000 10.200000 12.566371  3.926991  2.207277  5.614011
[8]  3.471070
> a - b
[1]  0.0000000  8.0000000  0.4000000 -0.8584073 -4.2146018 -5.6321206
    -6.1979984
[8] -7.5661163
> a^(1/b)
[1] 1.0000000 3.1622777 1.5036946 1.3313354 0.9528356 0.8464817 0.9689709
[8] 0.9008898
>
> sqrt( a )
[1] 1.0000000 3.1622777 1.8439089 1.7724539 0.8862269 0.6065307 0.8955454
[8] 0.6586985
> log( b )
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
[8] 2.0794415
>
```

Comprimento de Vetores e a Função length A função `length` retorna o número de elementos de um objeto:

```
> a <- seq(from=0, to=10, by=2)
> a
[1]  0  2  4  6  8 10
> length(a)
[1] 6
> length(1:20)
[1] 20
> length(rep(1:10,each=10))
[1] 100
>
```

A Regra da Ciclagem O comprimento é muito importante para as operações vetoriais, pois o R permite operações entre dois vetores de comprimentos diferentes, com a seguinte regra:

Ciclagem de Valores

Operações entre vetores de comprimentos diferentes são realizadas pareando-se seus elementos. Os elementos do vetor mais curto são repetidos sequencialmente até que a operação seja aplicada a todos os elementos do vetor mais longo

Quando o comprimento do vetor maior não é múltiplo do comprimento do maior, o R retorna o resultado e um aviso:

```
> b
 [1] 0 0 0 0 0 1 1 1 1 1
> c
[1] 1 2 3
> c*b
 [1] 0 0 0 0 0 3 1 2 3 1
Warning message:
In c * b : longer object length is not a multiple of shorter object length
> length(b)
[1] 10
> length(c)
[1] 3
>
```

Mas se o comprimento do vetor maior é um múltiplo do maior, o R retorna apenas o resultado, sem nenhum alerta:

```
> a
[1] 1 2
> b
 [1] 0 0 0 0 0 1 1 1 1 1
> a*b
 [1] 0 0 0 0 0 2 1 2 1 2
> length(b)/length(a)
[1] 5
>
```

Portanto **muito cuidado com as operações entre vetores de diferentes comprimentos**. A regra da ciclagem é um recurso poderoso da linguagem R [8], mas se você não tiver clareza do que deseja fazer, pode obter resultados indesejados.

Exercício 3.8. Palmeira com Muitos Fustes II

Uma palmeira perfilhada possui 10 fustes com os seguintes diâmetros: 5, 6, 7, 5, 10, 11, 6, 8, 9 e 7.

1. Calcule a área transversal de cada fuste dessa palmeira. Guarde este resultado em novo objeto.
2. Calcule a média das áreas transversais, sem usar a função `mean`.
3. Calcule a variância das áreas transversais, sem usar a função `var`

[8] A vantagem mais óbvia da regra da ciclagem é a possibilidade de multiplicação de um vetor por um valor único. Você compreende por que?

Exercício 3.9. Bits e Bytes

Como construir uma seqüência que representa o aumento do número de bits por byte de computador, quando se dobra o tamanho dos bytes?
Essa seqüência numérica parte do 2 e dobra os valores a cada passo.

3.2.3 Vetores: Operações Estatísticas

As funções matemáticas sobre vetores operam *elemento-a-elemento*. Já as funções estatísticas operam no vetor **como um todo**:

```
> mean( a )
[1] 2.491344
> var( b )
[1] 6
> max( c )
[1] 32
> sd( a )
[1] 3.259248
> sum( c )
[1] 338
> min( b )
[1] 1
> range( c )
[1] 20 32
>
```

Algumas funções úteis que não são estatísticas, mas operam no vetor são:

```
> a
[1]  1.0000000 10.0000000  3.4000000  3.1415927  0.7853982  0.3678794
   0.8020016
[8]  0.4338837
> sort(a)
[1]  0.3678794  0.4338837  0.7853982  0.8020016  1.0000000  3.1415927
   3.4000000
[8] 10.0000000
> rev(sort(a))
[1] 10.0000000  3.4000000  3.1415927  1.0000000  0.8020016  0.7853982
   0.4338837
[8]  0.3678794
> cumsum(sort(a))
[1]  0.3678794  0.8017632  1.5871613  2.3891629  3.3891629  6.5307556
   9.9307556
[8] 19.9307556
> cumsum(a)
[1]  1.00000 11.00000 14.40000 17.54159 18.32699 18.69487 19.49687 19.93076
> diff(a)
[1]  9.0000000 -6.6000000 -0.2584073 -2.3561945 -0.4175187  0.4341221
   -0.3681178
```

```
> diff( seq(10, 34, length=15) )
 [1] 1.714286 1.714286 1.714286 1.714286 1.714286 1.714286 1.714286
     1.714286
 [9] 1.714286 1.714286 1.714286 1.714286 1.714286 1.714286
>
```

Exercício 3.10. Conta de Luz

As leituras mensais do medidor de consumo de eletricidade de uma casa foram:

Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
9839	10149	10486	10746	11264	11684	12082	12599	13004	13350	13717	14052

1. Calcule o consumo de cada mês neste período.
2. Qual foi o máximo e mínimo de consumo mensal?
3. Qual a média, mediana e variância dos consumos mensais?

3.3 As Funções no R

Já foi visto que ao se digitar o nome de uma função na linha de comando, o R retorna o **código** da função. Veja a diferença de:

```
> ls()
```

para:

```
> ls
```

A maioria das funções precisa de certas **informações** para orientar o seu procedimento, tais informações são chamados de **argumentos**.

Os argumentos de qualquer função são detalhadamente explicados nas páginas de ajuda sobre a função. Mas para uma rápida consulta dos argumentos de uma função podemos usar a função `'args`:

```
> args(ls)
function (name, pos = -1, envir = as.environment(pos), all.names = FALSE,
    pattern)
NULL
> args(q)
function (save = "default", status = 0, runLast = TRUE)
NULL
> args(save.image)
function (file = ".RData", version = NULL, ascii = FALSE, compress = !ascii
    ,
    safe = TRUE)
NULL
>
```

Algumas funções, entretanto, são primitivas ou internas e seus argumentos não são apresentados. Geralmente, nesses casos os argumentos são bastante óbvios:

```
> args(sin)
NULL
> sin
.Primitive("sin")
>
```

Outras funções simplesmente não possuem argumentos:

```
> args(getwd)
function ()
NULL
> getwd
function ()
.Internal(getwd())
<environment: namespace:base>
>
```

Ao observar o resultado da função `args`, você notará que alguns argumentos são seguidos de uma expressão que se inicia com o sinal de igualdade (=). A expressão após o sinal de igualdade é chamada de **valor default** do argumento. Se o usuário não informar o valor para um dado argumento, a função usa o valor default. Como exemplo veja a função `save.image`:

```
> args(save.image)
function (file = ".RData", version = NULL, ascii = FALSE, compress = !ascii
    ,
    safe = TRUE)
NULL
>
```

Se o usuário simplesmente evocar a função `save.image()`, sem informar o nome do arquivo onde a área de trabalho deve ser gravada, o R gravará as informações num arquivo com nome `.RData`.

Exercício 3.11. Argumentos de Funções Estatísticas

Quais são os argumentos (e seus valores default) das seguintes funções:

- **mean**
- **sd**
- **range**
- **cumsum**

Exercício 3.12. Argumentos de Funções de Uso Comum

Quais são os argumentos (e seus valores default) das funções:

- **sort**
- **log**
- **seq**

O que é o argumento **". . ."**?

3.4 Distribuições Estatísticas: Funções no R

Sendo um ambiente para análise de dados, o R dispõe de um grande conjunto de funções para trabalhar com *Distribuições Estatísticas*. Essas funções ajudam não só na análise de dados, como também permitem a *simulação* de dados.

3.4.1 Distribuição Normal

A distribuição Normal é a distribuição central da teoria estatística. Para gerar uma amostra de observações de uma distribuição normal utilizamos a função `rnorm`:

```
> args( rnorm )
function (n, mean = 0, sd = 1)
NULL
> vn1 = rnorm( 1000, mean = 40,  sd = 9 )
> mean( vn1 )
[1] 39.47248
> sd( vn1 )
[1] 8.523735
> range( vn1 )
[1] 14.93126 62.11959
>
> vn2 = rnorm( 100000, mean = 40,  sd = 9 )
> mean( vn2 )
[1] 40.02547
> sd( vn2 )
[1] 9.025218
> range( vn2 )
[1]  3.40680 78.25496
>
```

Se quisermos saber a *probabilidade acumulada* até um certo valor de uma variável com distribuição normal utilizamos a função `pnorm`:

```
> args(pnorm )
function (q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

```
NULL
>
> pnorm( 1.96, mean = 0 , sd = 1 )
[1] 0.9750021
> pnorm( 1.96 )
[1] 0.9750021
>
> pnorm( 27, mean = 20, sd = 7 )
[1] 0.8413447
> pnorm( 13, mean = 20, sd = 7 )
[1] 0.1586553
>
```

Se quisermos obter o valor de um *quantil* da distribuição normal utilizamos a função `qnorm`:

```
> args( qnorm )
function (p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
NULL
> qnorm( 0.90 )
[1] 1.281552
> qnorm( 0.30 )
[1] -0.5244005
>
> qnorm( 0.90, 20, 7)
[1] 28.97086
> qnorm( 0.30, 20, 7)
[1] 16.32920
>
```

A função `dnorm` fornece a *densidade probabilística* para cada valor de uma variável Normal:

```
> args( dnorm )
function (x, mean = 0, sd = 1, log = FALSE)
NULL
> x = seq(-4, 4, length=10000)            # Sequencia de -4 a 4 com 10.000
    valores
>
> plot(x, dnorm(x))                       # Curva da Dist. Normal com mé
   dia 0 e desvio padrão 1
> points(x, dnorm(x, sd=2))               # Curva da Dist. Normal com mé
   dia 0 e desvio padrão 2 (adicionada ao gráfico)
>
```

Exercício 3.13 Amplitude Normal

Tomando uma variável que segue a Distribuição Normal, o que acontence com a *amplitude de variação* dos dados à medida que o tamanho da amostra cresce (por exemplo n= 100, 1000, 10000)?
Dica: use as funções `range` e `diff`

Exercício 3.14. Intervalo Normal I

Qual o intervalo da Distribuição Normal Padronizada que têm a média no centro e contem 50% das observações?

Exercício 3.15. Intervalo Normal II

Qual a probabilidade de uma observação da variável Normal Padronizada estar no intervalo [-1.96 , 1.96]?

3.4.2 As Funções que Operam em Distribuições Estatísticas

O que foi apresentado para Distribuição Normal pode ser generalizado para todas as distribuições que o R trabalha.

Há quatro funções para se trabalhar com distribuições estatísticas:

- **d**`distrib` - retorna a *densidade probabilística* para um dado valor da variável;
- **p**`distrib` - retorna a *probabilidade acumulada* para um dado valor da variável;
- **q**`distrib` - retorna o *quantil* para um dado valor de probabilidade acumulada;
- **r**`distrib` - retorna *valores* (números aleatórios) gerados a partir da distribuição;

No caso da Distribuição Normal: *distrib* = `norm`. Para outras distribuições temos:

DISTRIBUIÇÕES ESTATÍSTICAS NO R

Distribuição	Nome no R	Parâmetros[a]
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
qui-quadrado	chisq	df, ncp
exponential	exp	rate
F	f	df1, df2, ncp
gamma	gamma	shape, scale
geométrica	geom	prob
hypergeométrica	hyper	m, n, k
log-normal	lnorm	meanlog, sdlog
logística	logis	location, scale
binomial negativa	nbinom	size, prob
normal	norm	mean, sd
Poisson	pois	lambda
t de Student	t	df, ncp
uniforme	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

[a] os argumentos de cada função incluem estes parâmetros, entre outras coisas

Exercício 3.16. Teste t

Você realizou um teste *t* de Student bilateral e obteve o valor $t = 2.2$ com 19 graus de liberdade.
O teste é significativo ao nível de probabilidade de 5%? E se o valor observado fosse $t = 1.9$?

Exercício 3.17. Teste F

Você realizou um teste *F* e obteve o valor $F = 2.2$ com 19 graus de liberdade no numerador e 24 graus de liberdade no denominador.
O teste é significativo ao nível de probabilidade de 5%? E se o valor observado fosse $F = 2.5$?

Exercício 3.18. Padrão Espacial I

Gere duas amostras (p.ex.: *x* e *y*) de tamanho 1000 (n=1000) de números da distribuição Uniforme.
Faça um gráfico plotando uma amostra contra a outra (`plot(x,y)`). Qual o padrão espacial observado?
Você consegue explicá-lo?

Exercício 3.19. Padrão Espacial II

Gere duas amostras (p.ex.: *xp* e *yp*) de tamanho 10 (n=10) de números da distribuição Uniforme, com valor mínimo de zero e máximo de 100.
Gere duas amostras (p.ex.: *xf* e *yf*) de tamanho 1000 (n=1000) de números da distribuição Normal com média zero e desvio padrão 2)
Faça um gráfico plotando a soma das amostras X (*xp+xf*) contra a soma das amostras Y (*yp+yf*) (`plot(xp+xf,yp+yf)`).
Qual o padrão espacial observado? Você consegue explicá-lo?

Exercício 3.20. Gráfico Quantil-Quantil

Construa uma seqüência **ordenada** de 1000 números entre 0 e 1:

```
> p = seq(0, 1, length=1000)
```

O vetor `'p` representa um vetor de probabilidades acumuladas.
Gere 1000 números aleatórios da distribuição Normal com média e desvio-padrão 1 (um) e coloque os números em ordem:

```
> x = sort( rnorm(1000, mean=1) )
```

Faça um gráfico dos quantis da distribuição Normal, tomando o vetor `'p` de probabilidades, contra os valores de `'x`:

```
> plot( qnorm(p, mean=1), x )
```

Como é o gráfico resultante?
Repita o mesmo processo para a distribuição Exponencial (`rexp`), cujo valor *default* resulta em média = 1 . Como é o gráfico resultante? Por que?

3.5 Soluções dos Exercícios

Aqui você encontra os códigos que solucionam alguns dos exercícios propostos.

Se o seu código for diferente, não quer dizer necessariamente que errou. Compare os dois resultados! Como qualquer linguagem, o R é criativo: em muitos casos há mais de uma maneira de solucionar um problema.

4 Leitura e Manipulação de Dados

Links externos para o material referente a esse tópico no wiki da disciplina:

- Tutorial
- Exercícios
- Apostila

4.1 Leitura de Dados

4.1.1 Entrada de Dados Diretamente no R

Função "c()"(concatenate ou combine) As funções de criação de vetores já foram detalhadas na seção anterior. Basta lembrar aqui que todas elas são usadas para entrar diretamente dados em vetores no R:

```
> meu.vetor <- c(10.5,11.3,12.4,5.7)
> meu.vetor
[1] 10.5 11.3 12.4  5.7
>
> vetor.vazio <- c()
> vetor.vazio
NULL
```

Função "matrix()" A função `matrix` cria uma matriz com os valores do argumento `data`. O números de linhas e colunas são definidos pelos argumentos `nrow` e `ncol`:

```
> minha.matriz <- matrix(data=1:12,nrow=3,ncol=4)
> minha.matriz
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Como o *default* do argumento `data` é `NA`, se ele é omitido o resultado é uma matriz vazia:

```
> matriz.vazia <- matrix(nrow=3,ncol=4)
> matriz.vazia
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

Também por *default*, os valores são preenchidos por coluna. Para preencher por linha basta o alterar o argumento `byrow` para `TRUE`:

```
> minha.matriz <- matrix(data=1:12,nrow=3,ncol=4,byrow=T)
> minha.matriz
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Se o argumento `data` tem menos elementos do que a matriz, eles são repetidos até preenchê-la:

```
> elementos <- matrix(c("ar","água","terra","fogo","Leeloo"),ncol=4,nrow=4)
Warning message:
comprimento dos dados [5] não e um submultiplo ou multiplo do numero de
    linhas [4] na matrix
> elementos
     [,1]    [,2]     [,3]     [,4]
[1,] "ar"    "Leeloo" "fogo"   "terra"
[2,] "água"  "ar"     "Leeloo" "fogo"
[3,] "terra" "água"   "ar"     "Leeloo"
[4,] "fogo"  "terra"  "água"   "ar"
```

Função "data.frame()" Com a função `data.frame` reunimos vetores de mesmo comprimento em um só objeto:

```
> nome <- c("Didi","Dedé","Mussum","Zacarias")
> ano.nasc <- c(1936,1936,1941,1934)
> vive <- c("V","V","F","F")
> trapalhoes <- data.frame(nomes,ano.nasc,vive)
> trapalhoes
     nomes ano.nasc vive
1     Didi     1936    V
2     Dedé     1936    V
3   Mussum     1941    F
4 Zacarias     1934    F
>
# O mesmo, em um só comando:
> trapalhoes <- data.frame(nomes=c("Didi","Dedé","Mussum","Zacarias"), ano.
    nasc=c(1936,1936,1941,1934), vive=c("V","V","F","F"))
```

Funçao "edit()" Esta função abre uma interface simples de edição de dados em formato planilha, e é útil para pequenas modificações. Mas para salvar as modificações atribua o resultado da função `edit` a um objeto:

```
trapalhoes.2<-edit(trapalhoes)
```

	nomes	ano.nasc	vive
1	Didi	1936	V
2	Dedé	1936	V
3	Mussum	1941	F
4	Zacarias	1934	F

4.1.2 Dados que já Estão em Arquivos

Leitura e Exportação de Arquivos-Texto: "read.table()" e "write.table()" Para conjuntos de dados grandes, é mais prático gerar um arquivo de texto (ASCII) a partir e uma planilha ou banco de dados, e usar a função `read.table` para ler os dados para um objeto no R.

Para criar um objeto com os dados do arquivo gbmam93.csv (apagar extensão .pdf), por exemplo, digitamos:

```
> gbmam93 <- read.table(file="gbmam93.txt",header=T,row.names=1,sep=",")
> gbmam93
   a b c d e f g h i j k l m n o p q r s
1  1 1 1 0 1 1 1 0 0 1 0 1 1 1 1 1 1 1 1
2  1 1 0 1 1 0 1 0 1 0 0 1 1 0 1 1 1 1 1
3  1 0 0 1 1 0 1 0 1 0 0 1 1 1 1 1 1 1 1
4  1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1
5  1 0 0 1 1 0 1 0 1 0 0 1 1 0 0 0 1 0 0
6  1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 1 1 1
7  1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
8  1 0 1 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1
9  1 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0
10 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 0 0
11 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1
12 1 0 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 1
13 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1
14 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
```

O argumento `header=T` indica que a primeira linha são os nomes das variáveis, assim como `row.names=1` indica que a primeira coluna deve ser usada para os nomes das linhas. O argumento `sep` indica qual é o sinal de separação de cada registro, no caso vírgulas.

Esses e os outros argumentos da função a tornam extremamente flexível para ler dados em arquivos texto. Consulte a ajuda para mais informações, e também para

conhecer as variantes `read.csv` e `read.delim`.

Para exportar um objeto para um arquivo texto, use a função `write.table`, que tem a mesma lógica.

Conjuntos de Dados Distribuídos com os Pacotes do R Muitos pacotes do R incluem conjuntos de dados para exemplos, treinamento e verificação de análises. Se o pacote já está carregado (funções `library` ou `require`) todos os seus objetos estão disponíveis, inclusive os objetos de dados. Incluindo as séries temporais de número de peles de linces caçados no Canadá, analisadas pelo ecólogo Charles Elton obtém-se:

```
> lynx
Time Series:
Start = 1821
End = 1934
Frequency = 1
  [1]  269  321  585  871 1475 2821 3928 5943 4950 2577  523   98  184  279
       409
 [16] 2285 2685 3409 1824  409  151   45   68  213  546 1033 2129 2536  957
       361
 [31]  377  225  360  731 1638 2725 2871 2119  684  299  236  245  552 1623
      3311
 [46] 6721 4254  687  255  473  358  784 1594 1676 2251 1426  756  299  201
       229
 [61]  469  736 2042 2811 4431 2511  389   73   39   49   59  188  377 1292
      4031
 [76] 3495  587  105  153  387  758 1307 3465 6991 6313 3794 1836  345  382
       808
 [91] 1388 2713 3800 3091 2985 3790  674   81   80  108  229  399 1132 2432
      3574
[106] 2935 1537  529  485  662 1000 1590 2657 3396
```

Como qualquer objeto de um pacote, `lynx` tem um arquivo de ajuda, que é exibido com o comando `help(lynx)` ou `?lynx`:

```
lynx                package:datasets                R Documentation

Annual Canadian Lynx trappings 1821–1934

Description:

     Annual numbers of lynx trappings for 1821–1934 in Canada. Taken
     from Brockwell & Davis (1991), this appears to be the series
     considered by Campbell & Walker (1977).

Usage:

     lynx

Source:

     Brockwell, P. J. and Davis, R. A. (1991) _Time Series and
```

```
     Forecasting Methods._ Second edition. Springer. Series G (page
     557).

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole.

     Campbell, M. J.and A. M. Walker (1977). A Survey of statistical
     work on the Mackenzie River series of annual Canadian lynx
     trappings for the years 1821-1934 and a new analysis. _Journal of
     the Royal Statistical Society series A_, *140*, 411-431.
```

A página de ajuda mostra que o objeto de dados **lynx** está no pacote **datasets** que contém uma grande quantidade de conjuntos de dados. Para ter mais informações, execute o comando:

```
help(datasets)
```

Esse pacote faz parte da distribuição básica do R, e é carregado automaticamente quando se executa o R:

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Para fazer uma cópia de um objeto de dados de um pacote em sua área de trabalho, use a função **data**:

```
> ls()
[1] "gbmam93"      "trapalhoes"   "vetor.vazio"
> data(lynx)
> ls()
[1] "gbmam93"      "lynx"         "trapalhoes"   "vetor.vazio"
> data(BCI,package="vegan")
> ls()
[1] "gbmam93"      "lynx"    "BCI"      "trapalhoes"   "vetor.vazio"
```

No segundo caso, o pacote *vegan*, que tem o conjunto de dados, não está carregado e por isso deve ser indicado no argumento **package**.

Importacao de Pacotes Estatísticos O pacote *foreign* contém funções para importar e exportar diretamente arquivos de pacotes estatísticos.

Como é um pacote recomendado pelo *R Core Team*, faz parte da distribuição básica do R, e provavelmente está disponível **mas precisa ser carregado** com o comando `library(foreign)`.

Para mais informações, digite `help(package=foreign)`.

4.1.3 Exercícios

Exercício 4.1. Construir uma matriz de distâncias

Abaixo as distâncias por estradas entre quatro cidades da Europa, em quilômetros:

- Atenas a Madri: 3949
- Atenas a Paris: 3000
- Atenas a Estocolmo: 3927
- Madri a Paris: 1273
- Madri a Estocolomo: 3188
- Paris a Estocolmo: 1827

1. Construa uma matriz de distâncias com esses valores.
2. Compare sua matriz com objeto `eurodist`, disponível no pacote `datasets`.

Exercício 4.2. Criação de um data frame

Imagine um experimento em que hamsters de dois fenótipos (claros e escuros) recebem três tipos diferentes de dieta, e no qual as diferenças dos pesos (g) entre o fim e o início do experimento sejam:

	DIETA A	DIETA B	DIETA C
CLAROS	0.1 1.1 3.7	5.7 -1.2 -1.5	3.0 -0.4 0.6
ESCUROS	1.5 -0.1 2.0	0.6 -3.0 -0.3	-0.2 0.3 1.5

1. Crie um *data frame* com esses dados, na qual cada hamster seja uma linha, e as colunas sejam as variáveis cor, dieta e variação do peso.

DICA: Use as funções de gerar repetições para criar os vetores dos tratamentos.

Exercício 4.3. Leitura de Arquivo Texto

Crie um objeto com os dados do arquivo animais.txt

Exercício 4.4. Buscando um Arquivo de um Pacote

1. Descubra um objeto de dados de um pacote já carregado no R, e carregue esse pacote em sua área de trabalho.
2. Faça o mesmo para um pacote que está disponível em sua instalação de R, mas **não** está carregado.
3. Por que dados de pacotes carregados não são exibidos pelo comando `ls()`?
4. Consulte a página de ajuda da função `ls` para descobrir como listar objetos de um pacote carregado.

DICAS:

- O comando `search()` retorna uma lista dos pacotes carregados e o comando `library()` lista os pacotes disponíveis em seu computador para serem carregados. O comando `library(nome do pacote)` carrega um pacote.
- Para listar os objetos de dados nos pacotes carregados, use `data()`. Na saída desse comando há instruções de como listar todos os objetos de dados, incluindo os de pacotes não carregados.

4.2 Tipos de Objetos de Dados

Vetores, matrizes e listas são objetos com características diferentes, formalmente definidas no R como *atributos*. Em uma linguagem orientada a objetos, são esses atributos que definem o contexto para a execução de um comando e, portanto, seu resultado. Fazendo uma analogia com o mundo físico, uma mesma ação tem resultados diferentes de acordo com as características do objeto em que é aplicada.

O que é importante reter aqui é que os resultados que obtemos de um comando no R (incluindo as mensagens de erro!) são em boa parte definidas pelo objeto de dados, e não pela função. Por isso, quando enfrentar algum problema, verifique com cuidado na documentação se e como o seu comando se aplica à classe de objeto de dados que você está usando.[9]

4.2.1 Atributos de um Objeto de Dados

Todo objeto no R tem dois atributos básicos, que são o tipo de dado que contém((em termos técnicos, trata-se do modo de armazenamento, e.g., apenas números, apenas

[9] em termos técnicos: verifique se e como a função usada define um método para o objeto de dados usado

caracteres, ou uma mistura, que é uma lista)) e o número de elementos que contêm. As funções **mode** e **length** retornam esses atributos:

```
> pares
[1]  2  4  6  8 10
> mode(pares)
[1] "numeric"
> length(pares)
[1] 5
```

Objetos também podem ter uma ou mais classes. Um vetor numérico pode ser da classe dos inteiros ou da classe dos fatores. Um objeto da classe matriz pode ter dados do tipo numérico, lógicos[10] ou caracteres. O comando **class** retorna a classe de um objeto:

```
> matriz.letras
     [,1] [,2] [,3] [,4]
[1,] "a"  "b"  "c"  "d"
[2,] "a"  "b"  "c"  "d"
[3,] "a"  "b"  "c"  "d"
> mode(matriz.letras)
[1] "character"
> class(matriz.letras)
[1] "matrix"
> matriz.numeros
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    2    3    4
[3,]    1    2    3    4
> mode(matriz.numeros)
[1] "numeric"
> class(matriz.numeros)
[1] "matrix"
```

4.2.2 Vetores

São um conjunto de elementos do mesmo tipo, como números ou caracteres. Há várias classes de vetores. Os vetores que temos trabalhado até agora são numéricos:

```
> class( a )
[1] "numeric"
> class( b )
[1] "integer"
> class( c )
[1] "integer"
```

A classe **numeric** designa números reais enquanto que a classe **integer** designa números inteiros.

É possível ter no R um vetor tipo **character** formado por palavras ou frases:

[10] V= verdadeiro ou F = Falso

```
> sp = c( "Myrcia sulfiflora", "Syagrus romanzoffianus" , "Tabebuia
    cassinoides", "Myrcia sulfiflora" )
> mode( sp )
[1] "character"
>
```

4.2.3 Vetores da Classe Fator e as Funções "table"e "tapply"

Os fatores são uma classe especial de vetores, que definem variáveis categóricas de classificação, como os tratamentos em um experimento fatorial, ou categorias em uma tabela de contingência.

A função `factor` cria um fator, a partir de um vetor :

```
> sexo <- factor(rep(c("F","M"),each=9))
> sexo
 [1] M M M M M M M M M F F F F F F F F
Levels: F M
>
> numeros <- rep(1:3,each=3)
> numeros
[1] 1 1 1 2 2 2 3 3 3
> numeros.f <- factor(numeros)
> numeros.f
[1] 1 1 1 2 2 2 3 3 3
Levels: 1 2 3
```

Em muitos casos, indicar que um vetor é um fator é importante para a análise, e várias funções no R exigem variáveis dessa classe, ou têm respostas específicas para ela [11].

Note que fatores têm um atributo que especifica seu níveis ou categorias (`levels`) , que seguem ordem alfanumérica crescente, por *default*. Como essa ordem é importante para muitas análises, pode-se alterá-la com o argumento `levels`, por exemplo para colocar o controle antes dos tratamentos:

```
> tratamentos <- factor(rep(c("Controle","Adubo A","Adubo B"),each=4))
> tratamentos
 [1] Controle Controle Controle Controle Adubo A  Adubo A  Adubo A  Adubo A
 [9] Adubo B  Adubo B  Adubo B  Adubo B
Levels: Adubo A Adubo B Controle
>
> tratamentos <- factor(rep(c("Controle","Adubo A","Adubo B"),each=4),
    levels=c("Controle","Adubo A","Adubo B"))
> tratamentos
 [1] Controle Controle Controle Controle Adubo A  Adubo A  Adubo A  Adubo A
 [9] Adubo B  Adubo B  Adubo B  Adubo B
```

[11] em termos técnicos, dizemos que há métodos para cada classe de objeto, e que algumas funções têm métodos específicos para fatores, ou só têm para essa classe. Veja a seção sobre programação para detalhes

```
Levels: Controle Adubo A Adubo B
```

Há ainda a função **levels**, que retorna os níveis de um fator:

```
> tratamentos <- factor(rep(1:3,each=4))
> tratamentos
 [1] 1 1 1 1 2 2 2 2 3 3 3 3
Levels: 1 2 3
>
> levels(tratamentos)
[1] "1" "2" "3"
```

Fatores podem conter níveis não usados (vazios):

```
> politicos <- factor(rep("corrupto",10),levels=c("corrupto","honesto"))
> politicos
 [1] corrupto corrupto corrupto corrupto corrupto corrupto corrupto
    corrupto
 [9] corrupto corrupto
Levels: corrupto honesto
```

Duas Coisa que Você Precisa Saber sobre Fatores

- Um fator pode ter níveis para os quais não há valores. Isso pode acontecer quando alguns valores são selecionados ou excluídos (ver item sobre indexação, nessa seção). Isso é muito importante, pois afeta o resultado de muitas funções (Veja os exemplos da função **tapply**, a seguir e também no item sobre indexação de fatores).
- Ao importar um arquivo texto, a função **read.table** transforma por *default* em fatores todas as variáveis que tenham caracteres. Você pode indicar quais dessas variáveis não são fatores com o argumento **as.is**.

A função "tapply" Para aplicar uma função aos subconjuntos de um vetor definidos por um fator use a função **tapply**:

```
> pop.2007
Feira de Santana         Salvador        São Paulo          Niterói
          544113          2714119         11104712           476669
     Nova Iguaçu           Recife      Santo André   Rio de Janeiro
          858150          1528970           676846          6178762
        Sorocaba         Campinas           Osasco        Guarulhos
          590846          1073020           724368          1289047
        Jaboatão
          661901
> regiao
 [1] NE NE SE SE SE NE SE SE SE SE SE SE NE
Levels: NE SE
```

```
##Soma do número de habitantes das cidades por região
> tapply(X=pop.2007,INDEX=regiao,FUN=sum)
      NE       SE
 5449103 22972420

#Número de habitantes da cidade mais populosa de cada região
> tapply(X=pop.2007,INDEX=regiao,FUN=max)
      NE       SE
 2714119 11104712
```

Os argumentos básicos são o vetor de valores (`X`), o fator que será usado para definir os subconjuntos (`INDEX`), e a função que será aplicada (`FUN`). É possível usar mais de um fator para definir os subconjuntos:

```
> sexo <- factor(rep(c("F","M"),each=9))
> dieta <- factor(rep(rep(c("normal","light","diet"),each=3),2), levels=c("
    normal","light","diet"))
> peso <- c(65,69,83,90,58,84,85,74,92,71,72,78,67,65,62,74,73,68)
> sexo
 [1] F F F F F F F F F M M M M M M M M M
Levels: F M
> dieta
 [1] normal normal normal light  light  light  diet   diet   diet   normal
[11] normal normal light  light  light  diet   diet   diet
Levels: normal light diet
> peso
 [1] 65 69 83 90 58 84 85 74 92 71 72 78 67 65 62 74 73 68
> ##Media de peso por sexo e dieta
> tapply(peso,list(sexo,dieta),mean)
      diet    light   normal
F 83.66667 77.33333 72.33333
M 71.66667 64.66667 73.66667
```

A função "table" Para contar elementos em cada nível de um fator, use a função `table`:

```
> table(politicos)
politicos
corrupto  honesto
      10        0
```

A função pode fazer tabulações cruzadas, gerando uma tabela de contingência[12]:

```
> table(sexo,dieta)
    dieta
sexo normal light diet
   F      3     3    3
   M      3     3    3
```

[12] em termos técnicos, a função table retorna um "array"(veja abaixo) com o mesmo número de dimensões que os fatores fornecidos como argumentos

A função `table` trata cada valor de um vetor como um nível de um fator. Portanto, é útil também para contar a frequência de valores em vetores de números inteiros e de caracteres:

```
> x <- rep(1:5,each=10)
> class(x)
[1] "integer"
> table(x)
x
 1  2  3  4  5
10 10 10 10 10

> y <- rep(c("a","b","c"),20)
> class(y)
[1] "character"
> table(y)
y
 a  b  c
20 20 20
```

4.2.4 Listas

Uma lista é um objeto composto de vetores que podem ser diferentes classes e tamanhos, e podem ser criadas com o comando `list`

```
> minha.lista <- list(um.vetor=1:5, uma.matriz=matrix(1:6,2,3), um.dframe=
    data.frame(seculo=c("XIX","XX","XXI"),inicio=c(1801,1901,2001)))
> minha.lista
$um.vetor
[1] 1 2 3 4 5

$uma.matriz
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

$um.dframe
  seculo inicio
1    XIX   1801
2     XX   1901
3    XXI   2001
```

Nas palavras de Bill Venables, listas são como "varais"onde se pode pendurar qualquer outro objeto, inclusive outras listas, o que as torna **objetos recursivos**:

```
super.lista <- list(lista.velha=minha.lista, um.numero=1)
```

Alguns objetos têm como atributo o nome de seus elementos, como é o caso das listas. A função `names` retorna esses nomes:

```
> names(minha.lista)
```

```
[1] "um.vetor"    "uma.matriz" "um.dframe"
>
> names(super.lista)
[1] "lista.velha" "um.numero"
```

Seleção Rápida de um Objeto em uma Lista O operador $ permite selecionar rapidamente um objeto de uma lista:

```
> minha.lista$um.vetor
[1] 1 2 3 4 5
>
> names(super.lista)
[1] "lista.velha" "um.numero"
>
> names(super.lista$lista.velha)
[1] "um.vetor"    "uma.matriz" "um.dframe"
>
> names(super.lista$lista.velha$um.dframe)
[1] "seculo" "inicio"
```

4.2.5 Data Frames

A tabela de dados (*data frame*) é um tipo especial de lista, composta por vetores de mesmo tamanho, mas que podem ser de classes diferentes:

```
> names(trapalhoes)
[1] "nomes"     "ano.nasc" "vive"
>
> trapalhoes
     nomes ano.nasc  vive
1     Didi     1936  TRUE
2     Dedé     1936  TRUE
3   Mussum     1941 FALSE
4 Zacarias     1934 FALSE
>
> class(trapalhoes$nomes)
[1] "character"
> class(trapalhoes$ano.nasc)
[1] "numeric"
> class(trapalhoes$vive)
[1] "logical"
```

Seleção Rápida de Variáveis em Data Frames Todas as operações já descritas para listas são válidas para os *data frames*.

Assim como para listas, o operador $ pode ser usado selecionar um dos vetores que compõem um *data frame*, como no exemplo acima.

Esse operador também pode ser usado para criar novas variáveis (vetores) e acrescentá-las ao objeto. Para isso, basta acrescentar após o operador o nome da nova variável, e atribuir a ela um valor:

```
> trapalhoes$idade.2008 <- 2008 - trapalhoes$ano.nasc
> trapalhoes
     nomes ano.nasc  vive idade.2007
1     Didi     1936  TRUE         72
2     Dedé     1936  TRUE         72
3   Mussum     1941 FALSE         67
4 Zacarias     1934 FALSE         74
```

A Função "aggregate" A função `aggregate` gera subconjuntos de cada um dos vetores de um *data frame*, executa uma função para cada um desses subconjuntos, e retorna um novo *data frame* com os resultados.

Como seu resultado é sempre um *data frame*, a função `aggregate` é mais adequada que `tapply` para fazer estatíticas de muitos casos por uma ou muitas combinações de critérios:

```
> carros.marcas
 [1] Chevrolet Chevrolet Chevrolet Chevrolet Chevrolet Chevrolet Chevrolet
 [8] Chevrolet Ford      Ford      Ford      Ford      Ford      Ford
[15] Ford      Ford      Toyota    Toyota    Toyota    Toyota
Levels: Chevrolet Ford Toyota
> carros.numeros
   Price Horsepower Weight
12  13.4        110   2490
13  11.4        110   2785
14  15.1        160   3240
15  15.9        110   3195
16  16.3        170   3715
17  16.6        165   4025
18  18.8        170   3910
19  38.0        300   3380
31   7.4         63   1845
32  10.1        127   2530
33  11.3         96   2690
34  15.9        105   2850
35  14.0        115   2710
36  19.9        145   3735
37  20.2        140   3325
38  20.9        190   3950
84   9.8         82   2055
85  18.4        135   2950
86  18.2        130   3030
87  22.7        138   3785

> aggregate(x=carros.numeros,by=list(carros.marcas),FUN=mean)
    Group.1    Price Horsepower    Weight
1 Chevrolet 18.1875     161.875 3342.500
```

```
2      Ford 14.9625    122.625 2954.375
3    Toyota 17.2750    121.250 2955.000
```

Os argumentos básicos da função são o *data.frame* com os valores (`x`), o(s) fator(es) que definem os subconjuntos (`by`, que deve ser uma lista), e a função a calcular de cada vetor do *data frame* (`FUN`).

4.2.6 Matrizes e Arrays

Matrizes são vetores cujos valores são referenciados por dois índices, o número da linha e o número da coluna.

```
> my.matrix
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    2    2    2    2
[3,]    3    3    3    3
```

Os índices entre colchetes são a referência do par [linha,coluna][13]. Esses índices são exibidos quando as dimensões não têm nomes, que são controlados pelas funções `rownames` e `colnames`:

```
> rownames(my.matrix) <- c("R1","R2","R3")
> colnames(my.matrix) <- c("C1","C2","C3","C4")
>
> my.matrix
   C1 C2 C3 C4
R1  1  1  1  1
R2  2  2  2  2
R3  3  3  3  3
```

A função `dimnames` retorna uma lista com os nomes de cada dimensão de uma matriz

```
> dimnames(my.matrix)
[[1]]
[1] "R1" "R2" "R3"

[[2]]
[1] "C1" "C2" "C3" "C4"
```

A função `dim` retorna o comprimento de cada dimensão de uma matriz, no caso três linhas e quatro colunas:

```
> dim(my.matrix)
[1] 3 4
```

[13] a notação [,1] significa "todas as linhas da coluna 1", mais detalhes no ítem sobre indexação, mais abaixo

Totais Marginais: a função "apply" Para aplicar qualquer função a uma das dimensões de uma matriz, use a função `apply`:

```
##Soma dos valores de cada linha:
> apply(X=my.matrix,MARGIN=1,FUN=sum)
R1 R2 R3
 4  8 12
>
##Máximo de cada coluna
> apply(X=my.matrix,MARGIN=2,FUN=max)
C1 C2 C3 C4
 3  3  3  3
```

Os argumentos básicos da função são a matriz (`X`), a dimensão (`MARGIN`, valor 1 para linhas, valor 2 para colunas) e a função a aplicar (`FUN`).

Álgebra Matricial Todas as operações matriciais podem ser realizadas com as matrizes numéricas. O R possui funções para estas operações, como `%*%`, para multiplicação, entre outras:

```
> m
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
[4,]    1    1    1
> n
[1] 1 2 3
##Multiplicação usual: NÃO É MULTIPLICAÇÃO MATRICIAL
> m*n
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
##Diagonal da matriz resultante
> diag(m*n)
[1] 1 2 3
>
##Multiplicação matricial
> m \[,1][1,] 6[2,] 6[3,] 6
```

Consulte a ajuda para detalhes.

Arrays Os *arrays* são a generalização das matrizes para mais de duas dimensões. Um exemplo é o objeto `Titanic`, com as seguintes dimensões:

```
> dim(Titanic)
[1] 4 2 2 2
```

Com os seguintes nomes:

```
> dimnames(Titanic)
$Class
[1] "1st"  "2nd"  "3rd"  "Crew"

$Sex
[1] "Male"   "Female"

$Age
[1] "Child" "Adult"

$Survived
[1] "No"  "Yes"
```

Todas as operações aplicáveis a matrizes também o são para *arrays*:

```
> adultos.por.sexo <- apply(Titanic, c(2, 4), sum)
##Mulheres primeiro?
> adultos.por.sexo
        Survived
Sex        No Yes
  Male   1364 367
  Female  126 344

##Como o objeto resultante é uma matriz, pode-se aplicar apply novamente:
> adultos.por.sexo/apply(adultos.por.sexo,1,sum)
        Survived
Sex             No       Yes
  Male    0.7879838 0.2120162
  Female  0.2680851 0.7319149
```

4.2.7 Exercícios

Exercício 4.5. Classes de Objetos

O pacote "datasets"contém vários conjuntos de dados para uso em treinamento com a linguagem R.
O conjunto "iris"é distribuído de duas formas diferentes, nos objetos `iris` e `iris3`. São quatro medidas de flores de três espécies de *Iris* (Iridaceae).

1. Quais são as classes desses dois objetos?
2. Calcule a média de cada uma das quatro medidas por espécie, dos dois objetos.
3. Os nomes das variaveis estão em inglês. Mude-os para português no objeto `iris`. (*DICA: Como tudo mais no R, os resultados da função* `names` *podem ser armazenados em um objeto*)

Exercício 4.6. Importação e modificação de um arquivo texto

1. Crie um objeto com os dados do arquivo-texto esaligna.csv.
2. Verifique o conteúdo do objeto resultante, com a função `summary`.
3. Acrescente uma nova coluna ao data frame resultante, com a soma das biomassas de folhas e do tronco de cada árvore.
4. Acrescente outra coluna, com o valor da área basal de cada árvore.

Exercício 4.7. Alteração de Atributos de um Objeto

1. Crie o seguinte objeto da classe matriz:

```
> minha.matriz <- matrix(seq(from=2,to=10,by=2),ncol=5,nrow=3,byrow=T)
```

1. Mude os nomes das linhas para "L1"a "L3"e das colunas para "C1"a "C5". **Dica**: A função `paste` pode poupar trabalho.
2. O que acontece com a matriz após o comando:

```
> dim(minha.matriz) <- NULL
```

3. Como reverter este resultado?
4. Se você respondeu ao item anterior, percebeu que os nomes da matriz foram perdidos. Como restituí-los sem ter que refazer o passo 2? **Dica**: o resultado da função `dimnames` é uma lista com os nomes de cada dimensão.

Exercício 4.8. Agregação

1. Crie um *data frame* com os dados do arquivo itirapina.csv
2. A partir desse objeto, crie um novo *data frame* com o número de ordens, famílias, gêneros e espécies de insetos por tribo de planta.

DICA: para calcular a riqueza, crie esta função, digitando o comando abaixo:

```
riqueza <- function(x) { length(na.omit(unique(x))) }
```

Detalhes sobre construção de funções estão no tópico "Noções de Programação em Linguagem S". Por ora basta saber que essa função conta quantos elementos diferentes há num vetor, excluindo os valores faltantes, e.g.:

```
> letras <- rep( c(letters[1:3],NA), each=2)
> letras
[1] "a" "a" "b" "b" "c" "c" NA  NA
> riqueza(letras)
[1] 3
```

Exercício 4.9. Operações com Matrizes

As matrizes de transição são uma maneira conveniente de modelar o crescimento de uma população dividida em faixas etárias, ou estágios de desenvolvimento. Para uma população de *Coryphanta robinsorum* (Cactaceae) no deserto do Arizona, divida em três estágios, a matriz de transição foi:

```
0,43    0    0,56
0,33  0,61    0
   0  0,30  0,96
```

Os elementos da matriz são a probabilidades de transição, num intervalo de tempo, do estágio correspondente ao número da coluna para o estágio correspondente ao número da linha. Por exemplo, a chance de um indivíduo passar do estágio 1 para o 2 é 0,33, e de permanecer em 1 é de 0,43.

1. Crie um objeto da classe matriz com esses valores. Isso permite realizar as operações matriciais a seguir.

2. Para calcular o número de indivíduos em cada estágio após um intervalo de tempo, basta multiplicar a matriz de transição pelas abundâncias dos indivíduos em cada estágio. Começando com 50 indivíduos do estágio 1, 25 do estágio 2 e 10 no estágio 3, qual será o número de plantas em cada estágio após três intervalos?

3. **Opcional**: A taxa de crescimento geométrico da população é o primeiro autovalor da matriz de transição, que pode ser calculado com a função `eigen`[a]. Se a taxa é maior que um a população está crescendo. É o caso dessa população?

[a] consulte a ajuda para interpretar o resultados dessa função

4.3 O R como Ambiente de Operações Vetoriais

Na verdade, o R é muito mais que uma simples calculadora. O R é um **ambiente** onde podemos realizar operações vetoriais e matriciais.

Além das regras básicas para operações com vetores numéricos (ver), há operações aplicáveis a outros tipos de dados, e as importantíssimas **operações lógicas**, aplicáveis a qualquer classe.

4.3.1 Operações com Caracteres

Para vetores do tipo `character` operações matemáticas não fazem sentido e retornam uma mensagem de erro e o valor `NA`:

```
> mean( sp )
[1] NA
Warning message:
argument is not numeric or logical: returning NA in: mean.default(sp)
>
```

Mas existem algumas operações que são próprias desse tipo de vetores:

```
> sort( sp )
[1] "Myrcia sulfiflora"      "Myrcia sulfiflora"      "Syagrus
   romanzoffianus"
[4] "Tabebuia cassinoides"
>
> grep("Myrcia", sp)
[1] 1 4
>
> table( sp )
sp
     Myrcia sulfiflora Syagrus romanzoffianus   Tabebuia cassinoides
                     2                       1                      1
```

```
> bicho <- c("pato","gato","boi")
> cor <- c("branco","preto","vermeio")
> paste(bicho,cor)
[1] "pato branco" "gato preto"  "boi vermeio"
```

4.3.2 Operações Lógicas

Algumas operações são válidas para qualquer tipo de vetor. Essas operações envolvem comparações e são chamadas de operações lógicas:

```
> "Tabebuia cassinoides" == sp
[1] FALSE FALSE  TRUE FALSE
>
> a <= 7
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
>
> b
[1] 1 2 3 4 5 6 7 8
> b >= 4
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
>
> c
 [1] 20 21 22 23 24 25 26 27 28 29 30 31 32
> (c %% 2) != 0
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
    TRUE
[13] FALSE
>
```

Como o R é um **ambiente vetorial** o resultado de operações lógicas também podem ser guardadas em vetores. Assim, surgem os vetores de classe `logical`:

```
> b
[1] 1 2 3 4 5 6 7 8
> f <- b <= 5
> f
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> class( f )
[1] "logical"
>
```

Teste lógico para valores faltantes: função "is.na()" O indicador de valor faltante (*missing values*) no R é `NA`, e o de valores não-numéricos (em geral resultantes de operações que não têm um valor definido) é `NaN`. A operação lógica para testar esses valores é feita com a função `is.na`, e não com os operadores == ou `!=`

```
> a <- seq(-100,100,50)
> a
[1] -100  -50    0   50  100
> b <- sqrt(a)
Warning message:
NaNs produzidos in: sqrt(a)
> b
[1]       NaN       NaN  0.000000  7.071068 10.000000
> b==NA
[1] NA NA NA NA NA
> b=="NA"
[1] FALSE FALSE FALSE FALSE FALSE
> is.na(b)
[1]  TRUE  TRUE FALSE FALSE FALSE
> !is.na(b)
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

Operadores Lógicos no R [a]

[a] Incluímos aqui a função `is.na` para lembrar que para testar a ocorrência de valores faltantes ou não numéricos (Na e NaN) ela deve ser usada, e não os operadores == ou !=

```
=======================================
 OPERADOR          DESCRIÇÃO
=======================================
   ==              igual
   !=              diferente
   >               maior
   <               menor
   >=              maior ou igual
   <=              menor ou igual
   &               e (and)
   |               ou (or)
   !               não
   is.na()         valor faltante ou
                   não numérico
=======================================
```

4.3.3 Sinais de Atribuição e de Igualdade

Já tratamos dos sinais de atribuição no item sobre criação de objetos da seção de Introdução ao R, onde vimos que um dos sinais de atribuição é um sinal de igual (**=**):

```
> a = log(2)
> a
[1] 0.6931472
>
```

Um **detalhe importantíssimo** é diferenciar um **sinal de igualdade** de um **sinal de atribuição**.

O sinal de igualdade faz uma comparação entre dois elementos. No R o sinal de igualdade são dois sinais de igual seguidos (**==**). Este operador retorna o resultado do teste lógico *"a igual b?"*, que só pode ter dois valores, `T` (verdadeiro), ou `F` (falso):

```
> a = 2 + 2
> a == 4
[1] TRUE
> a == 2+2
[1] TRUE
> a == 2
[1] FALSE
>
```

Uma maneira simples de quantificar frequências Os vetores lógicos (`logical`) podem participar de operações matemáticas. Nesse caso o valor `TRUE` assume o valor 1, e valor `FALSE` assume o valor 0:

```
> f
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> f * 7
[1] 7 7 7 7 7 0 0 0
>
> f * (1:length(f))
[1] 1 2 3 4 5 0 0 0
>
```

Para ter frequência de dados que satisfaçam uma certa condição basta somar o vetor lógico resultante:

```
> notas.dos.alunos
 [1]  6.0  5.1  6.8  2.8  6.1  9.0  4.3 10.4  6.0  7.9  8.9  6.8  9.8  4.6
    11.3
[16]  8.0  6.7  4.5
##Quantos valores iguais ou maiores que cinco?
> sum(notas.dos.alunos>=5)
[1] 14
##Qual a proporção deste valores em relação ao total?
> sum(notas.dos.alunos>=5)/length(notas.dos.alunos)
[1] 0.7777778
```

4.3.4 Exercícios

Exercício 4.10. Tabela de Cores

Considere o seguinte vetor:

```
>  cores = c("amarelo","vermelho","azul","laranja")
>
```

Para gerar uma amostra, com reposição, dessas cores execute o comando:

```
> muitas.cores = sample(cores, 20, TRUE)
> muitas.cores
 [1] "amarelo"  "azul"     "amarelo"  "amarelo"  "vermelho" "laranja"
 [7] "laranja"  "azul"     "amarelo"  "vermelho" "amarelo"  "laranja"
[13] "azul"     "amarelo"  "amarelo"  "amarelo"  "amarelo"  "vermelho"
[19] "azul"     "laranja"
>
```

Como podemos obter uma tabela de freqüência das cores?

Exercício 4.11. Vetor Normal

Para gerar uma amostra de 1000 números de uma distribuição Normal com média 23 e desvio padrão 5, utilize o comando:

```
> vnormal = rnorm(1000, 23, 5)
```

Quantas observações no vetor `'vnormal` são maiores que 28? E maiores que 33?

Exercício 4.12. R Colors I

A função `colors()` lista todas as cores que o R é capaz de gerar.
Quantas dessas cores são variantes da cor salmão (*salmon*)?
Quantas são variantes de verde?

4.4 Subconjuntos e Indexação

Freqüentemente teremos que trabalhar não com um vetor inteiro, mas com um *subconjunto* dele. Para obter subconjuntos de um vetor temos que realizar operações de **indexação**, isto é, associar ao vetor um outro vetor de mesmo tamanho com os **índices** do elementos selecionados.

O **operador** de indexação é o colchetes `[]`, e um vetor pode ser indexado de três formas principais:

A) **Vetor de números inteiros positivos**: os números se referem às posições desejadas do vetor indexado.

```
> a
[1]  1.0000000 10.0000000  3.4000000  3.1415927  0.7853982  0.3678794
   0.8020016
[8]  0.4338837
> a[ 2:4 ]                              # subconjuntos dos elementos nas
   posições 2 a 4
[1] 10.000000  3.400000  3.141593
> b
[1] 1 2 3 4 5 6 7 8
> b[ c(2,5,8) ]                         # subconjuntos dos elementos nas
   posições 2, 5 e 8
[1] 2 5 8
>
```

B) **Vetor de números inteiros negativos**: os números se referem as posições **não** desejadas do vetor indexado.

```
> a
[1]  1.0000000 10.0000000  3.4000000  3.1415927  0.7853982  0.3678794
   0.8020016
[8]  0.4338837
> a[ -(2:4) ]                                          # Exclui as posiç
   ões de 2 a 4
```

```
[1] 1.0000000 0.7853982 0.3678794 0.8020016 0.4338837
> b
[1] 1 2 3 4 5 6 7 8
> b[ -c(2,5,8) ]                                          # Exclui as posiç
   ões de 2, 5 e 8
[1] 1 3 4 6 7
>
```

C) **Vetor lógico**: os elementos do vetor lógico correspondentes a `TRUE` são selecionados, os elementos correspondentes a `FALSE` são excluídos.

```
> b
[1] 1 2 3 4 5 6 7 8
> b[ c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, TRUE) ]
[1] 1 2 6 7 8
> b[ b< 5 ]
[1] 1 2 3 4
> b[ b >= 7 ]
[1] 7 8
> b[ b == max(b) ]
[1] 8
> b[ b == min(b) ]
[1] 1
>
```

Na indexação por vetores lógicos, esses vetores devem ter **o mesmo comprimento** do vetor indexado. Caso contrário a operação será defeituosa:

```
> b
[1] 1 2 3 4 5 6 7 8
> b[ c(TRUE, TRUE) ]
[1] 1 2 3 4 5 6 7 8
> b[ c(FALSE, FALSE) ]
integer(0)
>
```

As operações por vetores lógicos podem combinar vários critérios, por meio dos operadores "E", "OU"e "NÃO":

> Por questão de segurança do wiki contra spam algumas palavras são proibidas. Nos exemplos a seguir a palavra "estupro"em inglês foi substituida por "Abuso", por esse motivo para rodar as linhas de código deve retornar a palavra para o idioma inglesa.

```
## Primeiras 5 linhas do data frame USArrests (crimes/1000 habitantes em
   cada estado dos EUA, em 1973):
> USArrests[1:5,]
           Murder Assault UrbanPop Abusos
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
```

```
Arkansas       8.8      190        50 19.5
California     9.0      276        91 40.6

##População Urbana dos estados em que a razão entre assassinatos e assaltos
     foi maior que 20
> USArrests$UrbanPop[USArrests$Murder>USArrests$Assault/20]
 [1] 58 60 83 65 66 52 66 44 70 53 75 72 48 59 80 63 39

##Mesma condição acima, apenas estados com população menor do 55 milhões
> USArrests$UrbanPop[USArrests$Murder>USArrests$Assault/20 & USArrests$
    UrbanPop<55]
[1] 52 44 53 48 39
```

D) **Vetor caracter**: nesse caso o vetor deve ser *nomeado* (função `names`) por um vetor `character`:

```
> zoo = c(4, 10, 2, 45)
> names(zoo) = c("onça", "anta", "tatu", "guará")
> zoo[ c("anta","guará") ]
 anta guará
   10    45
> zoo[ grep("ç", names(zoo)) ]
onça
   4
>
```

4.4.1 Indexação de Fatores

A indexação de um fator pode resultar em níveis não usados. Caso você queira exluir esses níveis, use o argumento `drop`, do operador `[]`:

```
> tratamentos
 [1] Controle Controle Controle Controle Adubo A  Adubo A  Adubo A  Adubo A
 [9] Adubo B  Adubo B  Adubo B  Adubo B
Levels: Controle Adubo A Adubo B
> resposta
 [1] 1.8 3.0 0.9 1.7 2.4 2.7 2.6 1.5 3.0 2.7 0.8 3.0
> resp.sem.controle <- resposta[tratamentos!="Controle"]
> trat.sem.controle <- tratamentos[tratamentos!="Controle"]
> tapply(resp.sem.controle,trat.sem.controle,mean)
Controle  Adubo A  Adubo B
      NA    2.300    2.375
>
> ## Para eliminar níveis vazios do fator:
> trat.sem.controle <- tratamentos[tratamentos!="Controle",drop=T]
> tapply(resp.sem.controle,trat.sem.controle,mean)
Adubo A Adubo B
  2.300   2.375
```

4.4.2 Indexação de Matrizes e Data Frames

O modo de indexação de matrizes é `[linhas,colunas]`:

```
> matriz
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> matriz[1,1]
[1] 1
> matriz[1:2,1]
[1] 1 2
> matriz[1:2,1:2]
     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

A mesma notação é válida para *data frames*:

```
> USArrests[1:5,c(2,4)]
           Assault abuso
Alabama        236  21.2
Alaska         263  44.5
Arizona        294  31.0
Arkansas       190  19.5
California     276  40.6
>
> USArrests[1:5,c("Assault","Abuso")]
           Assault Abuso
Alabama        236  21.2
Alaska         263  44.5
Arizona        294  31.0
Arkansas       190  19.5
California     276  40.6
>
> USArrests[USArrests$UrbanPop>80,c("Assault","Abuso")]
              Assault Abuso
California        276  40.6
Hawaii             46  20.2
Illinois          249  24.0
Massachusetts     149  16.3
Nevada            252  46.0
New Jersey        159  18.8
```

Para incluir todas as linhas ou colunas, omita o valor (mas mantenha a vírgula!):

```
> matriz[,1]
[1] 1 2 3
> matriz[,c(1,4)]
     [,1] [,2]
[1,]    1   10
[2,]    2   11
```

```
[3,]    3   12
>
> USArrests[1:5,]
           Murder Assault UrbanPop Abuso
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
>
> USArrests[grep("C",row.names(USArrests)),]
               Murder Assault UrbanPop Abuso
California        9.0     276       91 40.6
Colorado          7.9     204       78 38.7
Connecticut       3.3     110       77 11.1
North Carolina   13.0     337       45 16.1
South Carolina   14.4     279       48 22.5
>
```

A notação é estendida para um *array* de qualquer dimensão, como o objeto `Titanic`, que tem quatro dimensões:

```
> dimnames(Titanic)
$Class
[1] "1st"   "2nd"   "3rd"   "Crew"

$Sex
[1] "Male"   "Female"

$Age
[1] "Child" "Adult"

$Survived
[1] "No"  "Yes"

## Adultos sobreviventes
> Titanic["2,2]
      Sex
Class  Male Female
  1st    57    140
  2nd    14     80
  3rd    75     76
  Crew  192     20

## O mesmo, usando os nomes
> Titanic[""Adult","Yes"]
      Sex
Class  Male Female
  1st    57    140
  2nd    14     80
  3rd    75     76
  Crew  192     20
```

4.4.3 Usando Indexação para Alterar Valores

Combinando as operações de indexação e de atribuição é possível alterar os valores de qualquer parte de um objeto:

```
> zoo
 onça  anta  tatu guará
    4    10     2    45
> names(zoo)[4] <- "lobo-guará"
> zoo
      onça       anta       tatu lobo-guará
         4         10          2         45
>
> matriz
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> matriz[,c(1,4)] <- NA
> matriz
     [,1] [,2] [,3] [,4]
[1,]   NA    4    7   NA
[2,]   NA    5    8   NA
[3,]   NA    6    9   NA
> matriz[is.na(matriz)==T] <- 0
> matriz
     [,1] [,2] [,3] [,4]
[1,]    0    4    7    0
[2,]    0    5    8    0
[3,]    0    6    9    0
```

4.4.4 Ordenação por Indexação: Função "order()"

Com a indexação é possível mudar a ordem de um vetor:

```
> zoo
      onça       anta       tatu lobo-guará
         4         10          2         45
>##Invertendo a ordem
> zoo[4:1]
lobo-guará       tatu       anta       onça
        45          2         10          4
##Uma ordem arbitrária
> zoo[c(3,4,2,1)]
      tatu lobo-guará       anta       onça
         2         45         10          4
```

A função `order` retorna os **índices** dos elementos de um vetor:

```
##Vetor de nomes do vetor zoo:
> names(zoo)
[1] "onça"       "anta"       "tatu"       "lobo-guará"
```

```
## O nome "anta", que tem o índice 2, é o primeiro na ordem alfabética:
> order(names(zoo))
[1] 2 4 1 3
```

E com isso podemos usar seu resultado para ordenar um vetor em função de quaisquer outros:

```
> zoo[order(names(zoo))]
      anta lobo-guará       onça       tatu
        10         45          4          2
```

O argumento da função comporta múltiplos vetores de critério. Em caso de empate pelo(s) primeiro(s) critério(s), os seguintes são usados:

```
> cidades
                regiao estado pop.2007
Feira de Santana     NE     BA   544113
Salvador             NE     BA  2714119
São Paulo            SE     SP 11104712
Niterói              SE     RJ   476669
Nova Iguaçu          SE     RJ   858150
Recife               NE     PE  1528970
Santo André          SE     SP   676846
Rio de Janeiro       SE     RJ  6178762
Sorocaba             SE     SP   590846
Campinas             SE     SP  1073020
Osasco               SE     SP   724368
Guarulhos            SE     SP  1289047
Jaboatão             NE     PE   661901
>
> cidades[order(cidades$regiao,cidades$estado,cidades$pop.2007,decreasing=T
  ),]
                regiao estado pop.2007
São Paulo            SE     SP 11104712
Guarulhos            SE     SP  1289047
Campinas             SE     SP  1073020
Osasco               SE     SP   724368
Santo André          SE     SP   676846
Sorocaba             SE     SP   590846
Rio de Janeiro       SE     RJ  6178762
Nova Iguaçu          SE     RJ   858150
Niterói              SE     RJ   476669
Recife               NE     PE  1528970
Jaboatão             NE     PE   661901
Salvador             NE     BA  2714119
Feira de Santana     NE     BA   544113
```

Prefira Ordenar por Indexação

A função `sort` é limitada porque ordena apenas o vetor ao qual é aplicada.
A função `order` permite uma ordenação mais flexível, pois pode usar critérios múltiplos, e os índices resultantes podem ser aplicados a qualquer objeto de igual comprimento.

4.4.5 Exercícios

Exercício 4.13. Comando Curto, Resultado nem Tanto

Verifique o resultado do comando:

```
> ?"["
```

Exercício 4.14 Indexação de Listas

Crie uma lista com o comando:

```
minha.lista <- list(um.vetor=1:5, uma.matriz=matrix(1:6,2,3),
                    um.dframe=data.frame(seculo=c("XIX","XX","XXI"),
                    inicio=c(1801,1901,2001)))
```

Qual a diferença entre os subconjuntos obtidos com os três comandos a seguir:

```
minha.lista[1]
minha.lista[[1]]
minha.lista$um.vetor
```

Exercício 4.15. Vetor Normal II

Para gerar uma amostra de 10.000 números de uma distribuição Normal com média 30 e desvio padrão 7, utilize o comando:

```
> vnormal = rnorm(10000, 30, 7)
```

Qual o somatório das observações no vetor `'vnormal` que são maiores que 44?
E maiores que 51?
Como você excluiria a *maior* observação do vetor `'vnormal`?

Exercício 4.16. R Colors II

Das cores que o R pode gerar, quais são as cores variantes de salmão?
Quais são as variantes de rosa?

Exercício 4.17. Modificação de Data Frame

1. Ops! Há um erro no arquivo criado no exercício 4.3, no nome do *Diplodocus*. Como corrigir?

2. Os valores de massa cerebral das três espécies de dinossauros agora estão disponíveis, mas no objeto estão como valores faltantes (`NA`). Substitua-os, usando o operador de indexação.

Diplodocus	50
Triceratops	70
Brachiosaurus	154,5

Exercício 4.18. Aninhamento de comunidades

O termo "aninhamento"(*nesting*) é usado para a situação em que comunidades mais pobres em espécies são um subconjunto das comunidades mais ricas. Uma análise exploratória rápida de aninhamento é ordenar as linhas e as colunas de uma matriz binária de ocorrência das espécies por comunidades.

1. Crie um objeto da classe matriz com a matriz de ocorrência de mamíferos em topos de montanhas. (*DICA: a função* `read.table` *retorna um* data frame. *Use a função* `as.matrix` *para mudar a classe para matriz.*)

2. Use o ordenamento por indexação para criar uma matriz com as comunidades por ordem decrescente de espécies, e as espécies por ordem decrescente de frequência de ocorrência. (OUTRA DICA: lembre-se da função `apply`!).

3. A matriz resultante tem sinais de aninhamento? Por que?

5 Análise Exploratória de Dados

Links externos para o material referente a esse tópico no wiki da disciplina:

- Tutorial
- Exercícios
- Apostila

Podemos pensar a análise exploratório de dados de duas formas:

- análise numérica: computa estatísticas descritivas;
- análise gráfica: explora o comportamento e a relação entre as variáveis através de gráficos.

Nesse tópico utlizaremos os arquivos de dados:

- Levantamento em caixetais: caixeta.csv (apagar extensão .pdf)
- Dados de biomassa de árvores: esaligna.csv (apagar extensão .pdf)
- Inventário em florestas plantadas:egrandis.csv (apagar extensão .pdf)

5.1 Estatísticas Descritivas

A forma mais direta de se obter um resumo estatístico das variáveis num `data.frame` é através da função `summary`. Ela apresenta estatísticas descritivas para as variáveis numéricas.

```
> cax = read.csv("caixeta.csv", header=TRUE, as.is=TRUE)
>
> summary(cax)
    local               parcela          arvore          fuste
       cap                h
 Length:1027         Min.   :1.000   Min.   :  1.0   Min.   : 1.000   Min.
      :  20.0   Min.   :  5.00
 Class :character   1st Qu.:2.000   1st Qu.: 51.0   1st Qu.: 1.000   1st Qu
    .: 190.0   1st Qu.: 60.00
 Mode  :character   Median :3.000   Median : 99.0   Median : 1.000   Median
     : 270.0   Median : 90.00
                    Mean   :2.821   Mean   :108.5   Mean   : 1.711   Mean
                         : 299.7   Mean   : 90.28
                    3rd Qu.:4.000   3rd Qu.:159.0   3rd Qu.: 2.000   3rd Qu
                        .: 360.0   3rd Qu.:110.00
```

```
                    Max.   :5.000    Max.   :291.0    Max.   :11.000    Max.
                           :2100.0   Max.   :480.00
   especie
 Length:1027
 Class :character
 Mode  :character

>
```

Outras estatísticas devem ser calculadas individualmente pelo analista:

```
> resumo1 = aggregate( cax[ , c("cap","h")], list(local=cax$local), mean )
> resumo2 = aggregate( cax[ , c("cap","h")], list(local=cax$local), sd )
>
> resumo = merge( resumo1, resumo2, by="local", suffixes=c(".mean",".sd") )
> resumo
   local cap.mean    h.mean   cap.sd     h.sd
1 chauas 293.6385  89.60094 139.8761 37.00023
2 jureia 404.4813 109.70954 213.8512 31.68068
3 retiro 236.5972  78.08333 137.2203 30.46426
```

5.1.1 Exercícios

Exercício: Estatísticas do Caixetal

Construa um `data.frame` com os dados de **área basal** por `local` e `parcela`.
Encontre a média e desvio padrão da área basal por `local`.
Calcule o intervalo de confiança de 95% da área basal por `local`.

5.2 Analisando a Distribuição das Variáveis: Gráficos Univariados

5.2.1 Histogramas

A primeira abordagem ao se estudar a distribuição de uma variável é o uso de **histogramas**:

```
> cax$dap = (pi/4)* (cax$cap/10)
> hist( cax$dap )
> hist( cax$dap[ cax$local == "chauas" ] )
> hist( cax$dap[ cax$local == "jureia" ] )
> hist( cax$dap[ cax$local == "retiro" ] )
```

A função `hist` também pode fornecer os dados do histograma, sem gerar o histograma propriamente dito:

```
> hist( cax$dap, plot=FALSE )
$breaks
 [1]   0  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150 160
    170
```

```
$counts
 [1]  72 408 329 116  62  20  10   3   4   1   1   0   0   0   0   0   1

$intensities
 [1] 7.010709e-03 3.972736e-02 3.203505e-02 1.129503e-02 6.037001e-03
 [6] 1.947420e-03 9.737098e-04 2.921130e-04 3.894839e-04 9.737098e-05
[11] 9.737098e-05 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
[16] 0.000000e+00 9.737098e-05

$density
 [1] 7.010709e-03 3.972736e-02 3.203505e-02 1.129503e-02 6.037001e-03
 [6] 1.947420e-03 9.737098e-04 2.921130e-04 3.894839e-04 9.737098e-05
[11] 9.737098e-05 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
[16] 0.000000e+00 9.737098e-05

$mids
 [1]   5  15  25  35  45  55  65  75  85  95 105 115 125 135 145 155 165

$xname
[1] "cax$dap"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
>
```

Note que o objeto gerado pela função **hist** tem classes **histogram**, logo pode ser guardado e grafado posteriormente:

```
> dap.hist = hist( cax$dap, plot=FALSE )
> class(dap.hist)
[1] "histogram"
>
> plot(dap.hist)
```

Alguns parâmetros gráficos podem tornar o gráfico mais apresentável. Esses parâmetros gráficos pode ser utilizados como *argumentos* em diversas funções gráficas onde são pertinentes:

- **xlab** e **ylab** = nomes dos eixos X e Y, respectivamente;
- **main** = nome do título do histograma;
- **col** = cor da barra (histograma), ou de linhas e símbolos plotados;

```
> hist( cax$dap[ cax$local=="chauas" ],
+ xlab="Diâmetro     Altura do Peito – DAP (cm)",
+ ylab="Freq ência",
```

```
+ main="Histograma DAP - Chauás",
+ col = "blue" )
>
```

Muitas vêzes desejamos comparar gráficos, sendo útil termos mais de uma janela gráfica. A função `X11()` (no UNIX) abre uma janela gráfica, sendo que podemos abrir várias:

```
> hist( cax$dap[ cax$local=="chauas" ] , main="Chauás" )
> X11()
> hist( cax$dap[ cax$local=="jureia" ] , main="Juréia" )
> X11()
> hist( cax$dap[ cax$local=="retiro" ] , main="Retiro" )
>
> dev.off()
X11
  2
> dev.off()
X11
  3
>
```

A função `dev.off()` fecha uma janela gráfica e faz parte de um conjunto de funções que manipula as janelas gráficas. Nessa manipulação, somente uma janela gráfica pode estar `ACTIVE` de cada vez, e as janelas são consideradas como estando num círculo, onde podemos passar de uma para outra:

- `dev.off()` - fecha a janela gráfica;
- `dev.cur()` - diz qual janela gráfica está `ACTIVE`;
- `dev.set(which=dev.cur())` - define qual janela deverá ficar ativa, o argumento `which` deve ser o número da janela;
- `dev.next(which=dev.cur())` - informa o número da próxima janela gráfica;
- `dev.prev(which=dev.cur())` - informa o número da janela gráfica anterior;
- `graphics.off()` - fecha todas as janelas gráficas.

```
> X11()
> hist( cax$dap[ cax$local=="retiro" ] , main="Retiro" )
> X11()
> hist( cax$dap[ cax$local=="jureia" ] , main="Juréia" )
> dev.cur()
X11
  3
> dev.set(2)
X11
  2
> hist( cax$dap[ cax$local=="retiro" ] , main="Retiro" , col="blue")
```

```
> dev.set(3)
X11
  3
> hist( cax$dap[ cax$local=="jureia" ] , main="Juréia" , col="green")
>
> graphics.off()
>
```

5.2.2 Exercício

Exercício: Altura de Árvores em Caixetais I

Construa um histograma da altura das árvores do caixetal.
Construa histogramas da altura das árvores para os diferentes caixetais (`local`).
Há diferenças entre as estruturas (distribuição de tamanhos) dos caixetais?

Exercício: Diâmetros de Árvores de Eucalipto

Construa um histogram do DAP das árvores de *E. saligna*. Procure interpretar o histograma.

5.2.3 Gráficos de Densidade

Uma outra forma de explorar a distribuição de uma variável é trabalharmos com um gráfico de *densidade*. O gráfico de densidade é gerado como se fosse um histograma com uma classe móvel, isto é, a classe que tem uma certa amplitude, se move da esquerda para direita e em cada ponto estima a *densidade probabilística da variável*. Tecnicamente, a função `density` é um **estimador de densidade de kernel gaussiano**.

A função `density` faz a análise da densidade, mas não faz o gráfico, devendo ser utilizada juntamente com a função `plot`, para criar o gráfico, ou a função `lines`, para adicionar uma linha a um gráfico já criado:

```
> plot( density(cax$dap) )
>
```

O parâmetro que controla o comportamento do estimador de densidade é a amplitude da janela de observação *bandwidth* (`bw`). Janela pequenas, geram estimativas de densidade com viés pequeno, mas com variância grande. Janelas grandes geram estimativas de densidade com viés grande, mas pequena variância. O ideal é o equilíbrio entre os extremos e o R possui algumas funções que buscam automaticamente da *bandwidth* apropriada, mas o analista tem controle sobre esse parâmetro:

```
> plot( density(cax$dap, bw=0.5), col="red" )
> lines( density(cax$dap, bw=5), col="blue" )
> lines( density(cax$dap, bw=1.5), col="green" )
>
```

5.2.4 Exercícios

Exercício: Distribuição de DAP nos Caixetais

Realize uma análise de densidade do DAP para cada um dos caixetais. Os resultados confirmam o que foi visto nos histogramas?

5.2.5 Boxplot

Boxplot é um gráfico estatístico também utilizado para estudar o comportamento de variáveis. Ele é composto dos elementos:

- Uma caixa (*box*) que representa a região entre o primeiro e o terceiro quartis (quantis 25% e 75%), ou seja, 50% dos dados estão dentro da caixa.
- Uma linha dentro da caixa que representa a posição da mediana (segundo quartil ou quantil 50%).
- Linhas que se prolongam a partir da caixa até no máximo 1,5 vêzes a distância interquartil (diferença entre o 1o. e 3o. quartis).
- As observações que passarem essa distância são representadas individualmente por pontos.

```
>de uma distribuição boxplot( cax$dap )
>
> esa = read.csv("dados/esaligna.csv",header=TRUE)
> boxplot( esa$dap )
>
```

O **boxplot** é útil para analisar a *simetria* de uma distribuição, o *espalhamento* das observações e a presença de observações discrepantes. Ele é problemático quando a variável analisada não é **unimodal**. Ele também é uma ferramenta útil para comparar distribuições, isso é realizado quando desejamos um **boxplot** para cada situação:

```
> boxplot( dap ~ local , data=cax )
>
```

Note que o primeiro argumento da função `boxplot` não é um vetor nesse caso:

```
> class( dap ~ local )
[1] "formula"
>
```

O primeiro argumento é uma `formula` estatística onde o símbolo **~** tem um significado especial.

A fórmula `dap ~ local` deve ser lida como: *modele a variável* `dap` *como função da variável* `local`.

O argumento `data` informa em qual `data.frame` estão as variáveis citadas na fórmula e é um argumento essencial toda vez que se utiliza uma fórmula.

A utilização da fórmula permite a construção de gráficos mais complexos, pensando na **interação** entre dois fatores influenciando a variável DAP:

```
> boxplot( dap ~ local * parcela, data=cax)
>
```

Mas os gráficos no R podem ficar realmente *sofisticados*, mas é necessário um pouco mais de programação:

```
> boxname = paste( sort(rep(unique(cax$local),5)), rep(1:5,3) )
> boxcor = sort(rep(c("navy","darkgreen","salmon1"),5))
> boxplot( dap ~ local * parcela, data=cax , names=boxname, col=boxcor,
    horizontal=T, las=1, xlab="DAP (cm)")
>
```

5.2.6 Exercícios

Exercício: Altura de Árvores em Caixetais II

Utilize o gráfico boxplot para analisar a altura das árvores em caixetais.

Exercício: Estrutura de Eucaliptais

Utilize o gráfico boxplot para analisar o DAP de árvores de *E. grandis* em função das variáveis região (`regiao`) e rotação (`rot`).

5.2.7 Gráficos Quantil-Quantil

Gráficos Quantil-Quantil também são uma forma de estudar o comportamento de variáveis, mas utilizando as propriedades que emergem de uma variável quando trabalhamos com os seus quantis.

O gráfico quantil-quantil mais tradicional é aquele usado para verificar se uma variável possui distribuição Normal. No R isso é realizado com a função `qqnorm`, associada à função `qqline` que adiciona uma linha ao gráfico:

```
> qqnorm( cax$dap )
> qqline( cax$dap )
>
```

A idéia central do gráfico quantil-quantil é a seguinte: quando um variável segue uma dada distribuição (como a distribuição Normal) os **quantis empíricos**, isto é, calculados a partir de uma amostra, formam uma linha reta contra os **quantis teóricos**, calculados a partir das estimativas dos parâmetros da distribuição (no caso da Normal: média e desvio padrão).

É isso que a função `qqnorm` faz para distribuição Normal:

```
> vn1 = rnorm( 10000 )
> qqnorm( vn1 )
> qqline( vn1 )
>
> ve1 = rexp( 100000 )
> qqnorm( ve1 )
> qqline( ve1 )
>
> ve2 = apply( matrix(ve1, ncol=100), 1, mean)
> qqnorm( ve2 )
> qqline( ve2 )
>
```

Também é possível comparar duas distribuições a partir dos **quantis empíricos**:

```
> qqplot( cax$dap[ cax$local=="retiro" ], cax$dap[ cax$local=="jureia" ] )
> abline( 0, 1, col="red" )
>
> a = min( cax$dap[ cax$local=="jureia" ] )
> abline( a, 1, col="navy" )
>
```

Nota: a função `abline( a, b)` adiciona a um gráfico um reta com intercepto *a* e inclinação *b*.

5.2.8 Exercícios

Exercício: Inventário em Floresta Plantada

Verifique se as variáveis quantitativas obtidas no inventário de florestas plantadas tem distribuição Normal: `dap`, `ht` e `hdom`.

Exercício: Altura de Árvores em Caixetais III

Verifique se a distribuição da altura das árvores tem o mesmo comportamento nos diferentes caixetais.

5.2.9 Gráfico de Variável Quantitativa por Classes

A maneira clássica de se apresentar uma variável quantitativa associada a uma classe é o famoso gráfico de barras.

Vejamos um exemplo comum em *fitossociologia* que é apresentar a densidade relativa das espécies:

```
> da = table( cax$especie[ cax$local=="jureia" ] )
> da = sort(da, decreasing=TRUE )
> dr = da/sum(da) * 100
```

Para obter o gráfico de barras basta usar a função **barplot**:

```
> barplot( dr )
```

O resultado não é muito apropriado para interpretações, mas podemos fazer algumas melhoras:

```
> barplot( dr , xlab="Densidade Relativa (%)", horiz=T, las=1)
```

O nome das espécies precisam de mais espaço. É possível alterar o espaço trabalhando os parâmetros da função **par** que controla todos os parâmetros gráficos de uma janela gráfica. Nesse caso, o parâmetro **'omd=c(x1,x2,y1,y2)** define o início e final da região de plotagem em termos relativos. O valor *default* é **omd=c(0, 1, 0, 1)**.

```
> par( omd=c(0.2,1,0,1) )
> barplot( dr , xlab="Densidade Relativa (%)", horiz=T, las=1)
>
```

Na verdade, o gráfico de barras não é um gráfico muito apropriado para o que se propõe, apesar do uso generalizado que se faz dele na comunidade científica.

No gráfico de barras, somos levados a comparar o comprimento das barras para estabelecer um julgamento entre as categorias. No gráfico de densidade relativa, compararmos os comprimentos de barra para obter uma visão das densidades relativas das espécies.

Existe no R, um gráfico que faz a mesma coisa de modo muito mais simples e direto:

```
> par( omd=c(0,1,0,1) )     # Primeiro é necessário re-estabelecer o parâ
    metro omd
>
> dotchart( dr )
>
```

No **dotchart**, somos levados a comparar a posição relativa dos pontos, e a relação entre as categorias fica muito mais rápida e direta.

Como nessa floresta a *Tabebuia cassinoides* (caixeta) é a espécie dominante, é interessantes fazer o gráfico na escala logarítmica para enfatizar a diferença entre as outras espécies:

```
> dotchart( log(dr), xlab="Logaritmo Natural da Densidade Relativa (%)")
```

5.2.10 Exercícios

Exercício: Dominância em Caixetais

Construa um gráfico da dominância das espécies nos caixetais.

Exercício: Inventário em Floresta Plantada

Utilizando a função **dotchart** investigue o número de árvores no inventário em função da região (**regiao**) e rotação (**rot**).

5.3 Análise Gráfica: Relação entre Variáveis

5.3.1 Gráfico de Dispersão

Os gráficos de dispersão (ou gráficos x-y) são os gráficos mais utilizados para estudar a relação entre duas variáveis.

A função genérica no R para gráficos de dispersão é a função `plot`:

```
> plot( x = cax$dap, y = cax$h )
```

Na função `plot`, o primeiro argumento é plotado nas *abscissas* (eixo-x) e o segundo argumento nas *ordenadas* (eixo-y).

Ao investigar a relação entre duas variáveis, freqüentemente a densidade de pontos no gráfico torna o julgamento da relação problemática, pois é muito difícil considerar a variação da densidade ao se julgar a relação no gráfico de dispersão.

Há no R uma função adicional que auxilia o julgamento adicionando ao gráfico de dispersão uma linha não-paramétrica de tendência (**smooth** ou suavização):

```
> scatter.smooth( cax$dap, cax$h , col="red")
```

Uma série de parâmetros gráficos podem ser utilizados diretamente nas funções `plot` e `scatter.smooth`:

```
> scatter.smooth( cax$dap, cax$h , col="red", xlab="DAP (cm)", ylab="Altura
    (dm)", main="Caixetais")
> scatter.smooth( cax$dap, cax$h , col="red", xlab="DAP (cm)", ylab="Altura
    (dm)", log="x")
> scatter.smooth( cax$dap, cax$h , col="red", xlab="DAP (cm)", ylab="Altura
    (dm)", log="y")
> scatter.smooth( cax$dap, cax$h , col="red", xlab="DAP (cm)", ylab="Altura
    (dm)", log="xy")
```

O R também permite um certo grau de **interação** com gráficos de dispersão. Uma delas é a identificação de observações no gráfico:

```
> scatter.smooth( cax$dap, cax$h )
> dim( cax )
[1] 1027    8
> identify( cax$dap, cax$h, 1:1027 )
[1] 362 556 557
>
>
> cax[ c(362, 556, 557), ]
     local parcela arvore fuste  cap   h                  especie       dap
362 chauas       5    232     1  130 480     Tabebuia cassinoides  10.21018
556 jureia       4    105     1 1400 100     Tabebuia cassinoides 109.95574
557 jureia       4    106     1 2100 160 Calophyllum brasiliensis 164.93361
>
```

A função `identify` atua sobre um gráfico produzido (plot) e possui três argumentos. Os dois primeiros são os mesmos argumentos que geraram o gráfico. O terceiro

argumento é uma variável de identificação. No exemplo acima a variável de identificação é o índice que identifica a observação (linha do `data.frame`).

Ao executar a função `identify`, o R entra num modo interativo com o gráfico. Ao posicionar o *mouse* sobre uma observação no gráfico e pressionar o **botão esquerdo**, o R identifica a observação. É possível identificar tantas observações quanto se desejar. Para sair do modo interativo, pressiona-se o **botão direito** do *mouse*.

No exemplo acima, as três observações discrepantes do gráfico parecem de fato muito erradas.Assim, podemos eliminá-las e continuar o estudo da relação:

```
> cax2 = cax[ -c(362, 556, 557), ]
> scatter.smooth( cax2$dap, cax2$h , col="red" )
```

Também na função `plot` é possível se utilizar como argumento inicial uma `formula`, seguida do `data.frame` que contem as variáveis:

```
> plot( h ~ dap, data=cax2 )
```

Nesse caso, para adicionar a linha não-paramétrica de tendência é necessário um segundo comando:

```
> plot( h ~ dap, data=cax2 )
> lines( lowess( cax2$dap, cax2$h ) , col="red")
```

O uso da `formula` permite a utilização da função `coplot` para formação de gráficos de dispersão em função de variáveis categóricas:

```
> coplot( h ~ dap | local , data=cax2 )
> coplot( h ~ dap | local*parcela , data=cax2 )
```

Também é possível adicionar uma linha de tendência em cada gráfico gerado pela função `coplot`:

```
> coplot( h ~ dap | local , data=cax2 , panel= panel.smooth)
> coplot( h ~ dap | local*parcela , data=cax2 , panel=panel.smooth)
```

Na `formula` acima, surgiram elementos novos:

- A barra vertical indica uma situação condicional, no caso fazer um gráfico de dispersão para cada `local`.
- O asterísco (`*`) indica **interação**, no caso o gráfico de dispersão é realizado para cada interação entre as variáveis `local` e `parcela`.

A função `coplot` atua de forma diferente, se as variáveis que classificam o gráfico de dispersão são variáveis categóricas (`factor`) ou numéricas (`numeric`):

```
> egr = read.table("dados/egrandis.csv",header=TRUE,sep=";")
> coplot( ht ~ dap | idade, data=egr, panel = panel.smooth )
> coplot( ht ~ dap | idade * rot , data=egr, panel = panel.smooth )
> coplot( ht ~ dap | idade * as.factor(rot) , data=egr, panel = panel.
   smooth )
```

5.3.2 Exercícios

Exercício: Relação Hipsométrica da Caixeta

Analise a relação dap-altura (`dap` e `h`) em função do caixetal, mas **somente** para as árvores de caixeta (*Tabebuia cassinoides*).

Exercício: Inventário em Floresta Plantada II Analise a relação entre as variáveis `hdom` (altura das árvores dominantes) e `dap` para diferentes regiões (`regiao`) e rotações (`rot`).

5.3.3 Painel de Gráficos de Dispersão

Quando o objetivo é explorar a relação entre variáveis quantitativas com o objetivo de construir modelos ou analisar a estrutura de correlação é útil poder fazer gráficos de dispersão das variáveis duas-a-duas. A função **pairs** realiza essa operação automaticamente:

```
> pairs( egr[ , c("dap","ht","hdom","idade")] )
```

Sempre é possível sofisticar os gráficos. No exemplo abaixo o painel apresenta a relação entre as variáveis quantitativas utilizando cores para mostrar as variáveis região e rotação:

```
> pairs( egr[ , c("dap","ht","hdom","idade")] , pch=21, bg=c("red","blue","
    green")[unclass(egr$regiao)] )
> pairs( egr[ , c("dap","ht","hdom","idade")] , pch=21, bg=c("red","green")
    [unclass(egr$rot)] )
```

5.3.4 Exercícios

Exercício: Biomassa de Árvores de Eucalipto

Analise a relação entre as variáveis quantitativas do conjunto de dados sobre biomassa das árvores de *E. saligna*.
Qual a influência da variável classe (`classe`) sobre a relação entre as variáveis?

5.4 Gráficos em Painel: O Pacote Lattice

Para ampliar a capacidade de análise gráfica exploratória e mesmo *modelagem gráfica* dos dados, existe no R o pacote **lattice**. Para carregar o pacote usa-se o comando:

```
> library(lattice)
```

O pacote **lattice** oferece uma série de funções análogas às funções gráficas do R, mas permite a construção de **paineis**. Um painel é um série de gráficos de mesmo tipo (dispersão, histograma, etc.) colocados lado-a-lado acompanhando uma variável categórica ou quantitativa.

5.4.1 Gráficos de Dispersão

Para construir gráficos de dispersão no lattice usa-se a função **xyplot**:

```
> egr = read.csv("egrandis.csv",header=T)
> xyplot( ht ~ dap, data=egr )
```

Note que no lattice, os gráficos são construídos com base em fórmulas. Essas fórmulas permitem estrutura mais complexas de análise:

```
> xyplot( ht ~ dap | regiao , data=egr )
> xyplot( ht ~ dap | regiao * rot , data=egr )
```

Também é possível construir gráficos com suavização:

```
> xyplot( ht ~ dap | regiao * rot , data=egr,
+ panel = function(x,y)
+ {
+         panel.xyplot(x,y)
+         panel.loess(x,y, span=1, col="red")
+ } )
>
```

5.4.2 Exercícios

Exercício: Relação Hipsométrica da Caixeta II

Utilizando o pacote lattice, analise a relação dap-altura (`dap` e `h`) em função do caixetal, mas **somente** para as árvores de caixeta (*Tabebuia cassinoides*).

Exercício: Relação Altura das Dominantes - Idade em Florestas Plantadas

Utilizando os dados de floresta plantada (*E. grandis*), analise a relação entre altura das árvores dominantes (`hdom`) e idade (`idade`) por rotação (`rot`) e região (`regiao`).

5.4.3 Painel de Gráficos de Dispersão

O pacote lattice também possui uma função específica para fazer um painel de gráficos de dispersão: **splom** (*scatter plot*):

```
> splom( egr[ , c("dap","ht","hdom","idade")]  )
```

Identificar grupos em cada gráfico de dispersão é mais fácil com a função **splom**, basta utilizar o argumento `'group`:

```
> splom( egr[ , c("dap","ht","hdom","idade")] , group=egr$regiao )
> splom( egr[ , c("dap","ht","hdom","idade")] , group=egr$rot )
```

Também é possível adicionar uma *linha de suavização*, mas é necessário definir a função de painel (argumento `panel`):

```
> splom( egr[ , c("dap","ht","hdom","idade")] , group=egr$regiao,
+ panel = function(x,y,...)
+ {
+     panel.splom(x,y,...)
+     panel.loess(x,y,...)
+ }
+ )
>
```

A função **panel.loess** é a função que efetivamente faz a suavização em cada gráfico de dispersão.

5.4.4 Exercícios

Exercício: Biomassa de Árvores de Eucalipto

Analise a relação entre as variáveis quantitativas dos dados de biomassa de *E. saligna* utilizando a função **splom**. Inclua na sua análise a variável `classe`.

5.4.5 Histogramas e Gráficos de Densidade

No lattice, todos os tipos de gráficos podem ser construídos na forma de painel. Para estudar a distribuição de variáveis temos a função **histogram** e **densityplot**:

```
> cax = read.csv("caixeta.csv",header=T)
> cax$dap = cax$cap / pi
>
> histogram( ~ dap, data=cax )
> histogram( ~ dap | local , data=cax )
>
> densityplot( ~ dap, data=cax )
> densityplot( ~ dap | local , data=cax )
```

Também é possível construir um histograma com linhas de densidade, para isso o tipo do histograma deve ser definido como `density`:

```
> histogram( ~ ht | regiao * rot , dat=egr, type="density",
+     panel = function(x, ...){
+             panel.histogram(x, ...)
+             panel.densityplot(x, col="red", ...)
+     }
```

```
+ )
```

As funções de histograma e densidade podem se tornar mais complexas. No exemplo abaixo, uma curva de densidade assumindo a distribuição Normal é adicionada aos histogramas, os quais são construídos com a densidade nas ordenadas:

```
> histogram( ~ ht | regiao * rot , dat=egr, type="density",
+     panel = function(x, ...){
+           panel.histogram(x, ...)
+           panel.mathdensity(dmath=dnorm, col="black", args=list(mean=mean
    (x),sd=sd(x)))
+     }
+ )
```

5.4.6 Exercícios

Exercício: Altura das Árvores Dominantes em Florestas Plantadas

Explore o comportamento da variável altura das árvores dominantes (`hdom`) por região (`regiao`) e rotação (`rot`) na floresta plantada de *E. grandis*.

Exercício: Altura de Árvores de Caixeta

Analise o comportamento da variável altura (`h`) das árvores de caixeta.

5.4.7 Gráficos Quantil-Quantil

O pacote **lattice** implementa a construção de gráficos sempre através de fórmulas, isso pode ser conveniente no caso de se verificar a distribuição de uma variável em várias situações:

```
> qqmath( ~ dap | local, data=cax )
```

Para adicionar a linha do gráfico qq é necessário editar a função de painel:

```
> qqmath( ~ dap | local, data=cax,
+     panel = function(x, ...)
+    {
+       panel.qqmath(x, ...)
+       panel.qqmathline(x, ...)
+    }
+ )
```

Uma vantagem do pacote lattice é a possibilidade de gráficos quantil-quantil com outras distribuições além da distribuição normal. Nos gráficos abaixo, o DAP das árvores dos caixetais é comparada com a distribuição exponencial (`qexp`).

```
> qqmath( ~ dap | local , data=cax, distribution = function(p) qexp(p, 1/
    mean(x)) )
>
> qqmath( ~ dap | local , data=cax, distribution = function(p) qexp(p, 1/
    mean(x)),
+     panel = function(x,...)
+     {
+         panel.qqmath(x,...)
+         panel.qqmathline(x,...)
+     }
+ )
>
```

Também é possível fazer gráficos quantil-quantil de um conjuntos de dados empíricos usando a função **qq**:

```
> qq( local ~ dap, data=cax, subset = ( local=="chauas" | local=="jureia" )
    )
> qq( local ~ dap, data=cax, subset = ( local=="chauas" | local=="retiro" )
    )
> qq( local ~ dap, data=cax, subset = ( local=="jureia" | local=="retiro" )
    )
```

Dois aspectos devem ser notados no código acima:

1. A variável `local` (categórica) aparece **à esquerda** do sinal de modelagem.
2. O argumento `subset` faz com que a variável `local` fique com apenas duas categorias.

5.4.8 Exercícios

Exercício: Altura das Árvores em Florestas Plantadas

Verifique se a altura das árvores (`ht`) nas florestas plantadas de *E. grandis* segue distribuição Normal.
Faça uma análise geral e depois por região (`regiao`) e rotação (`rot`).

Exercício: Biomassa de Árvores de Eucalipto

Verifique se biomassa total (`total`) e a biomassa do tronco (`tronco`) das árvores de *E. saligna* possuem distribuição semelhante. E a biomassa das folhas (`folha`), tem distribuição semelhante à biomassa do tronco?

6 Criação e Edição de Gráficos no R

Links externos para o material referente a esse tópico no wiki da disciplina:

- Tutorial
- Exercícios
- Apostila

6.1 Fazendo Gráficos no R

Aqui você irá aprender como fazer gráficos para publicação. Nesta aula, iremos passar apenas pelos gráficos mais simples como **gráficos de dispersão, de barras e box-plot**, pois estes serão os gráficos usados pela grande maioria dos alunos durante o curso de pós-graduação. Porém, lembre-se que no R é possível construir uma variedade incrível de gráficos e figuras. Para mais exemplos basta entrar no R Graph Gallery e ver as possibilidades.

Custo Benefício de Fazer Gráficos no R Nesta apostila você aprenderá a editar os gráficos e adequá-los para dissertações, teses ou revistas científicas. Editar gráficos no R não é fácil, demora tempo (pode demorar horas para fazer apenas uma figura) e é muitas vezes frustrante, pois cada passo requer uma série de ajustes. Porém, o R permite mudar quase todos os parâmetros dentro de um gráfico, uma liberdade que (quase) nenhum outro pacote estatístico possui. E lembre-se, bons gráficos dizem mais que apenas o conjunto de dados a ser apresentado. Bons gráficos mostram vários resultados em um pequeno espaço de papel, são facilmente interpretáveis e podem aumentar suas chances de ter trabalhos aceitos em boas revistas científicas. Por isso, é muito importante investir bastante tempo em fazer figuras bonitas e bem explicativas.

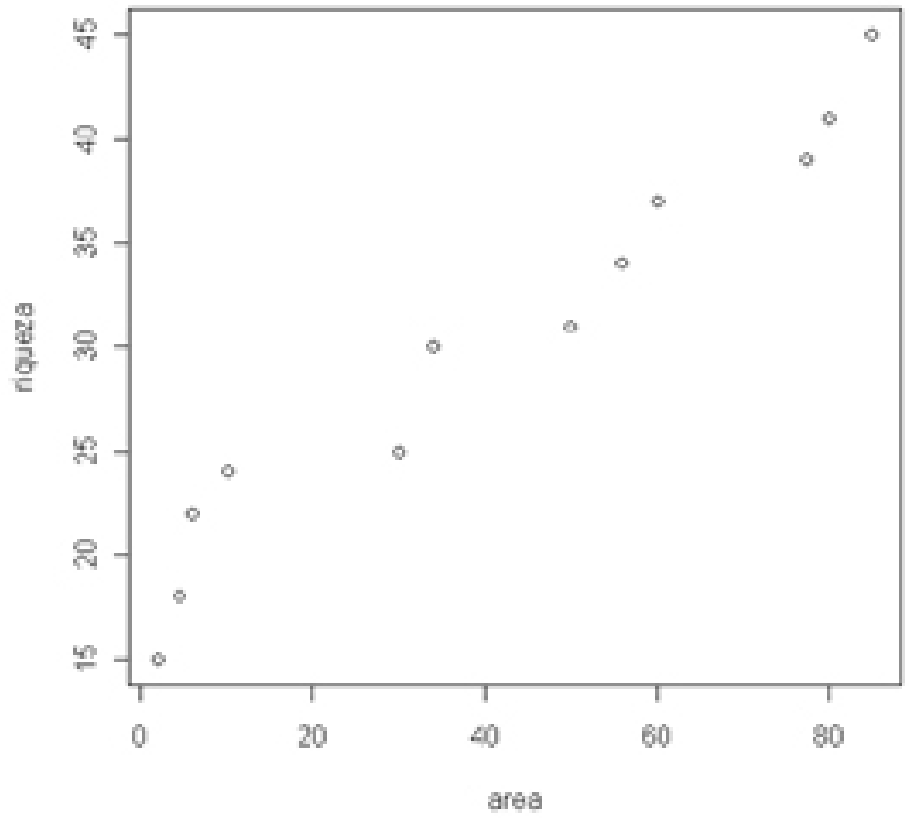
6.2 Criando Gráficos

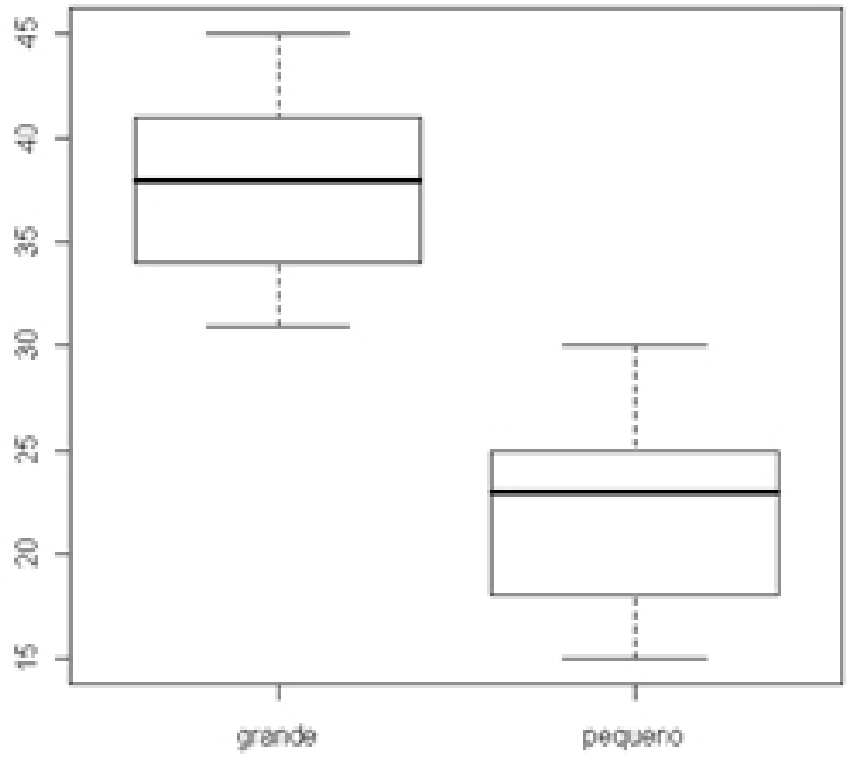
Fazer gráficos rapidamente no R é fácil. Basta dizer qual tipo de gráfico se deseja e quais são as variáveis.

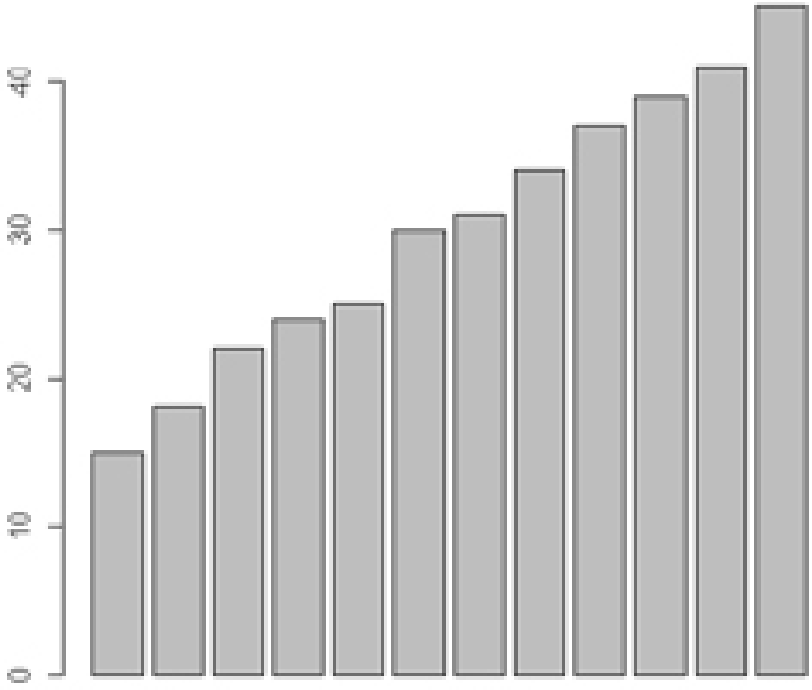
Há duas maneiras de se especificar as variáveis

Cartesiana – plot(x,y)
Fórmula – plot(y∼x)
Ambas as formas são corretas, mas como a grande maioria das análises feitas são no formato y∼x, em vez de x,y, acaba ficando mais fácil usar y∼x.

```
riqueza <- c(15,18,22,24,25,30,31,34,37,39,41,45)
area <- c(2,4.5,6,10,30,34,50,56,60,77.5,80,85)
area.cate <- rep(c("pequeno", "grande"), each=6)

plot(riqueza~area)
plot(area,riqueza) # o mesmo que o anterior
boxplot(riqueza~area.cate)
barplot(riqueza)
```




45
40
35
30
25
20
15
grande
pequeno



As figuras padrão (`default`) que o R produz não são publicáveis, mas trazem toda a informação que foi usada para gerar o gráfico e podem perfeitamente ser usadas para uma interpretação inicial dos resultados. O `plot` ou scatterplot é um gráfico de dispersão, sendo que cada ponto no `plot` representa uma das réplicas (e.g. 12 réplicas, 12 pontos). Na sua forma mais simples, as legendas dos eixos vêm com o nome das variáveis usadas para criar o plot.

Quando as variáveis são categóricas, o gráfico padrão que o R produz é o boxplot ou "box and whiskers plot" (chamado em português de desenho esquemático, desenho da caixa, ou desenho de caixa e bigode). No boxplot, a linha grossa do meio representa a mediana, a caixa representa o 1º e 3º quartil, e os "bigodes"podem representar ou os valores máximos e mínimos, ou 1.5 vezes o valor dos quartis (aproximadamente 2 desvios padrões); é desenhado o que for menor. Às vezes, alguns pontos são desenhados individualmente além dos bigodes, estes são os "outliers", que podem ser suprimidos com o argumento `outline=F`.

O barplot, ou gráfico de barras, mostra cada ponto da variável especificada como uma barra. Na sua forma mais simples, são apresentados apenas os valores brutos e não há informação alguma quanto à dispersão dos dados. No barplot nenhum dos eixos vem com legendas (aliás, o eixo x também não é desenhado).

Exercício 1 – Fazendo os Primeiros Gráficos

Construa "plot", boxplot e barplot usando as variáveis do conjunto de dados Conjunto de Dados: Dados de Biomassa de Árvores de Eucalyptus Saligna, para explorar relações entre:

`dap e ht`
`ht e tronco`
`dap e classe`
`dap e talhao`
`dap`
`ht`

Note: `barplot` só aceita uma variável

6.3 Editando Gráficos

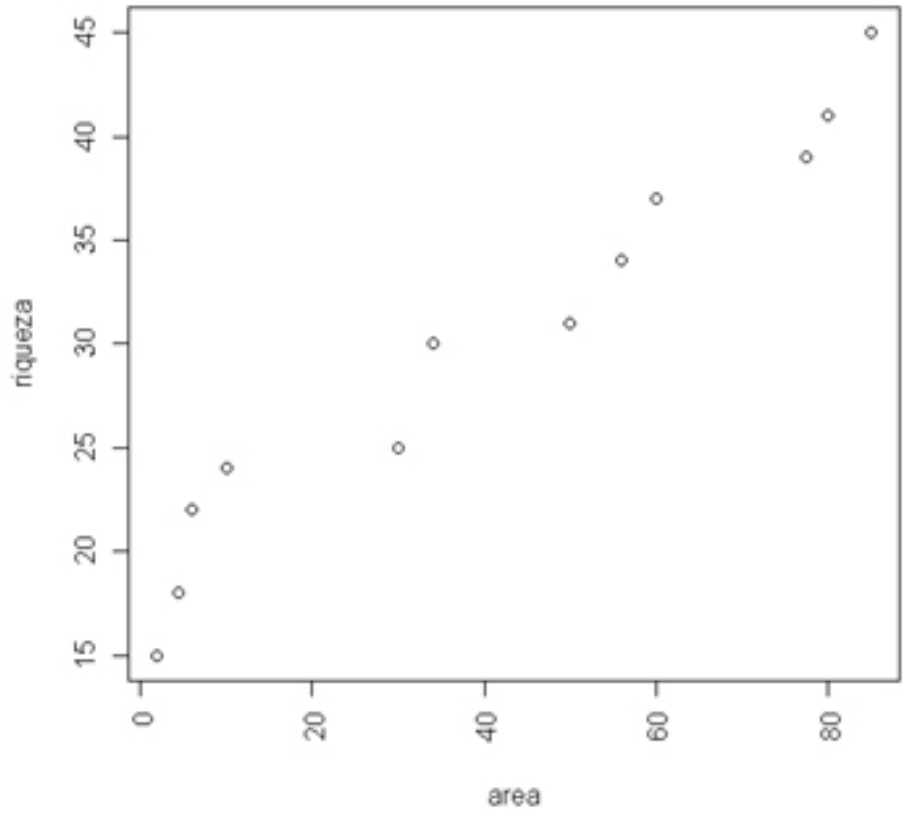
Aqui começa a parte mais complicada. Não porque é difícil mudar a forma como os gráficos são feitos, mas porque para chegar num resultado final adequado requer um processo iterativo. Em outras palavras, se o objetivo é mudar o tamanho da fonte, será necessário testar vários tamanhos até se atingir o "tamanho ideal"para incluir no manuscrito e/ou tese.

Existem duas maneiras de se mudar parâmetros no gráfico; uma é por dentro do gráfico, ou seja, dentro da função `plot, boxplot`, ou `barplot`, e a outra é pela função `par()`. Alguns argumentos só podem ser chamados **exclusivamente** por uma destas maneiras. Por exemplo `ylab` e `xlab` modificam o nome (label) dos eixos e só podem ser chamadas por dentro do gráfico, já outras funções só podem ser chamadas pelo `par()`, como por exemplo, `mar` que controla o tamanho das margens do gráfico e `mfrow` que controla quantos gráficos serão mostrados no mesmo dispositivo.

Para que as alterações controladas pelo `par()` possam surtir efeito, elas sempre devem vir antes do gráfico. Se um novo dispositivo gráfico não for aberto, todas as funções já controladas pelo `par()` continuarão valendo, mesmo que o gráfico mude.

Em geral, a informação que vem por último é a informação que o R vai tomar como verdadeira. Por exemplo, `las` controla a direção das legendas dos eixos (`las= 1`, legendas escritas sempre na horizontal, `las=3`, legendas sempre na vertical), sejam os números da escala ou o nome do eixo. Se o seguinte comando é dado:

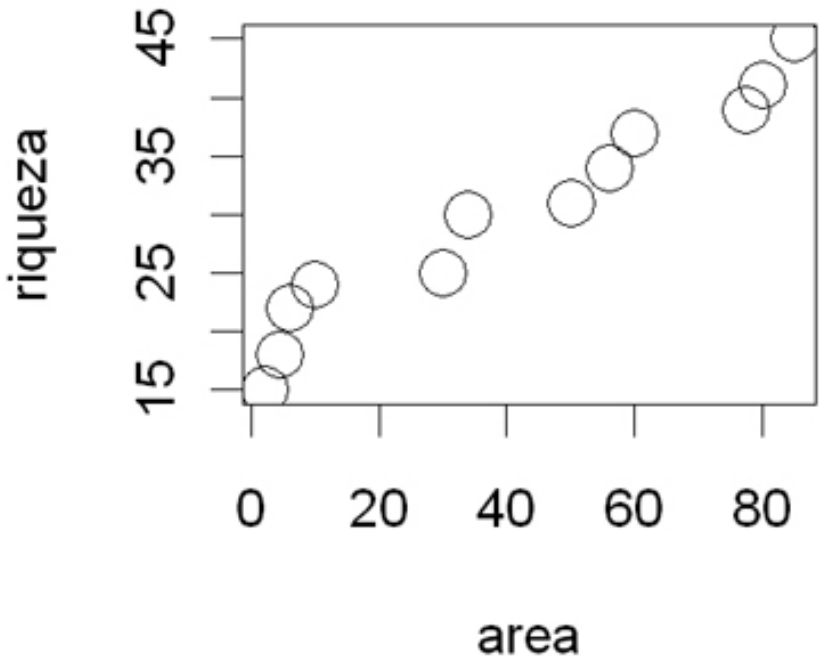
```
par(las=1)
plot(riqueza~area, las=3)
```



O resultado final será um gráfico com legendas na vertical. Isso a princípio pode parecer sem sentido, porém imagine um caso em que há vários gráficos no mesmo dispositivo gráfico e em todos os casos se deseja ter legendas horizontais, com exceção de um gráfico apenas em que um dos eixos será desenhado verticalmente. São em casos que nem este que se torna necessário poder dar informações "conflitantes" para o R.

Outro caso que é importante saber é a função **cex**. Em sua forma geral, ela se aplica ao tamanho de fonte das legendas, título, pontos, entre outros. Se o seguinte comando é dado:

```
par(cex=2)
plot(riqueza~area, cex=2)
```



O resultado final terá legendas com tamanho 2 (default=1) e pontos com tamanho 4. Isto ocorre pois `par(cex=2)` tem a função geral de aumentar todas as fontes e pontos, enquanto que no `plot(cex=2)` tem a função de aumentar só o pontos. E quando neste caso específico, em vez das informações entrarem em conflito, como no caso anterior, elas se multiplicam.

Exercício 2 – Aprendendo a Editar Gráficos

Entre no R e digite:
?plot
Agora, usando as variáveis:

```
riqueza <- c(15,18,22,24,25,30,31,34,37,39,41,45)
area <- c(2,4.5,6,10,30,34,50,56,60,77.5,80,85)
```

Mude:
O nome do eixo x para "Tamanho da Ilha (ha)"
O nome do eixo y para "Riqueza de Espécies"
O título do gráfico para "Aves das Ilhas Samoa"
Agora entre no:
?par
Usando o mesmo gráfico anterior, mude:
O tipo de ponto (numero de 0 a 25)
O tamanho dos pontos e legendas
A direção da escala do gráfico (para ficar tudo na horizontal)
O tipo de fonte das legendas (para ficar tudo como em Times New Roman – dica= "serif")

Apesar das páginas de ajuda do R não serem muito amigáveis no começo, é preciso ter calma e aprender a procurar a informação desejada. A página do `par()` é uma das mais procuradas por todos que estão fazendo gráficos no R, e por isso é importante que se gaste um tempo para aprender qual tipo de informação ela fornece, onde está a informação, e como mudar os parâmetros do R.

DICA

No começo, quando ainda não se conhece direito todas as funções do `par()` é aconselhável que se imprima a página de ajuda para que se possa visualizar todas os argumentos.

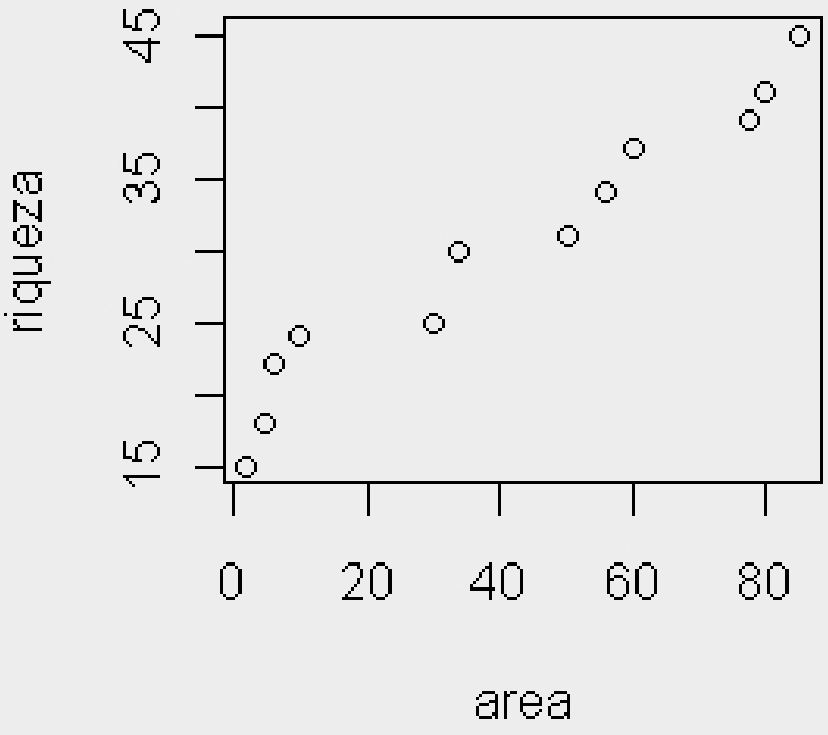
O outro, o `par(mar=c())` controla o "tamanho das margens"do gráfico e como a figura ficará disposta dentro do dispositivo. O vetor contido dentro da função `mar=()`, controla as posições das margens, sendo que o 1 numero controla a margem da parte de baixo do gráfico, o 2 controla a margem do lado esquerdo, o 3 numero controla a parte de cima e o 4 numero controla o tamanho da margem do lado direito do gráfico.

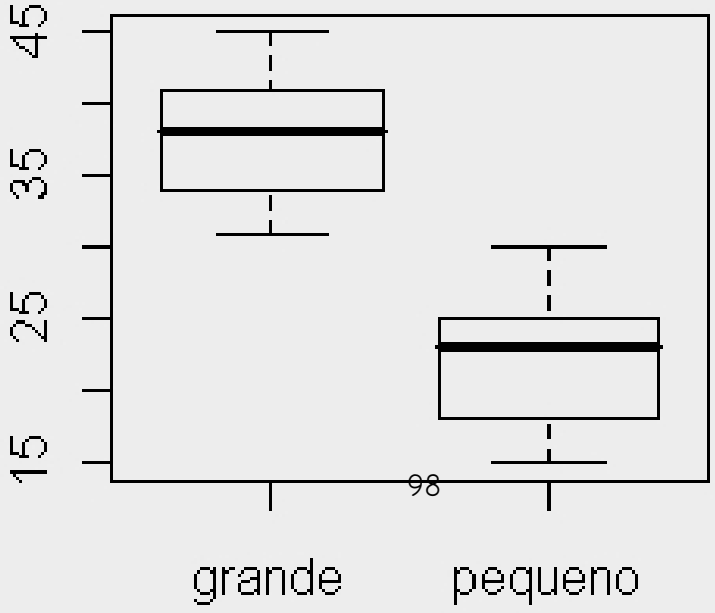
DICA

As figuras abaixo foram preenchidas de cinza para facilitar a visualização com o parâmetro `par(bg="gray93")`

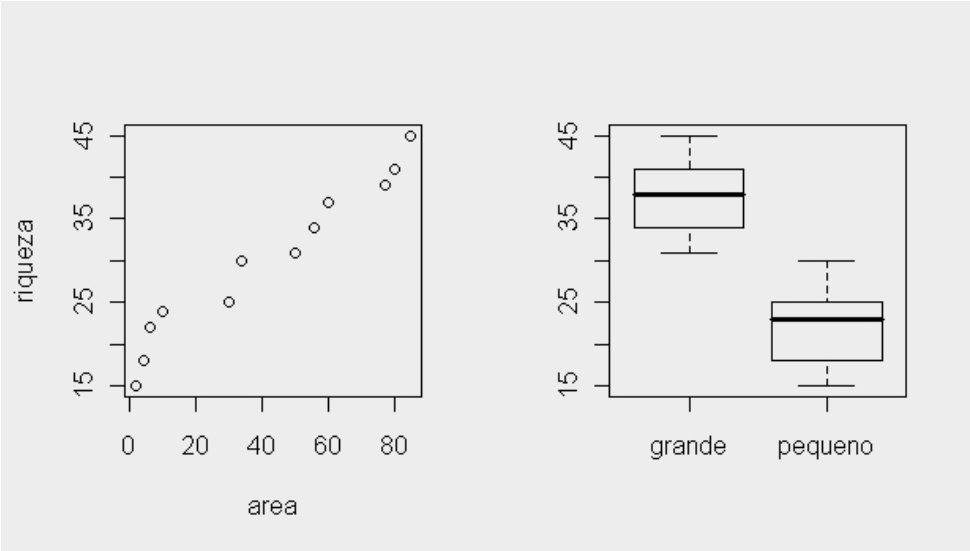
Exemplos de `par(mfrow=c())`

```
par(mfrow=c(2,1))
plot(riqueza~area)
boxplot(riqueza~area.cate)
```


riqueza
15
25
35
45
0
20
40
60
80
area

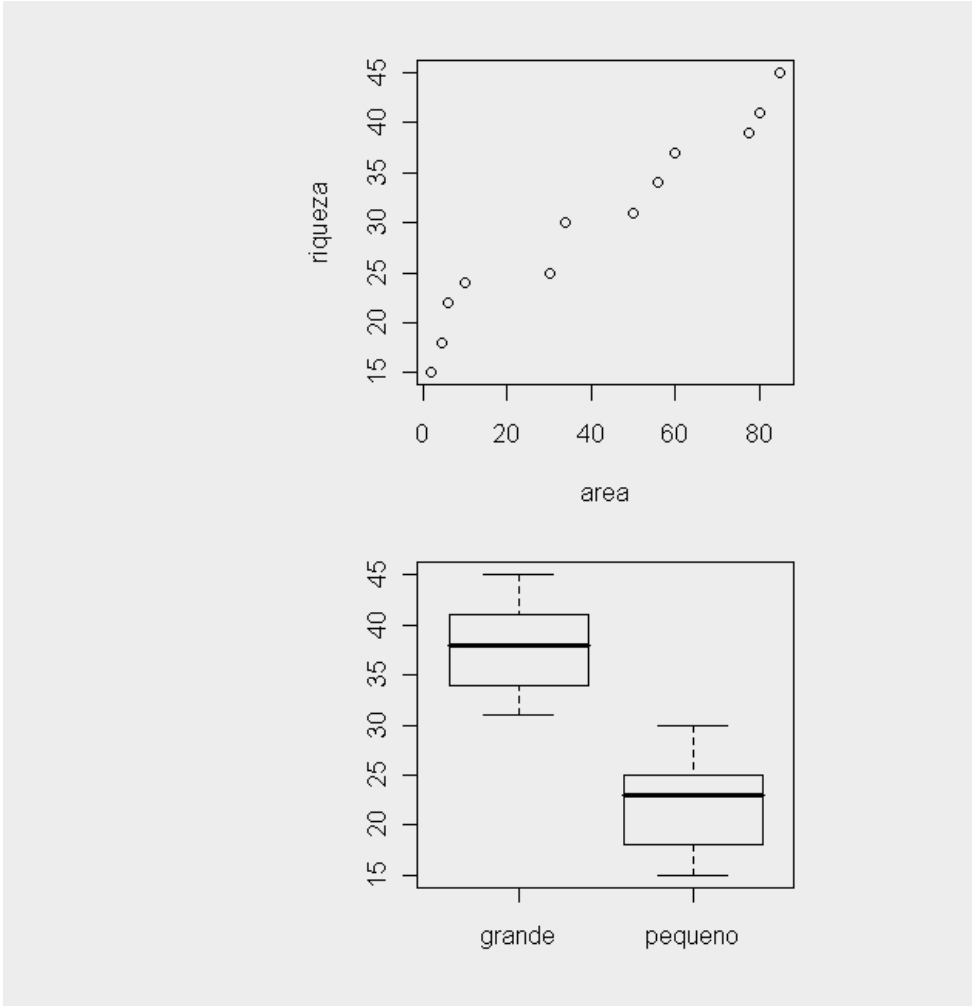

15
25
35
45
grande
pequeno

```
par(mfrow=c(1,2))
plot(riqueza~area)
boxplot(riqueza~area.cate)
```

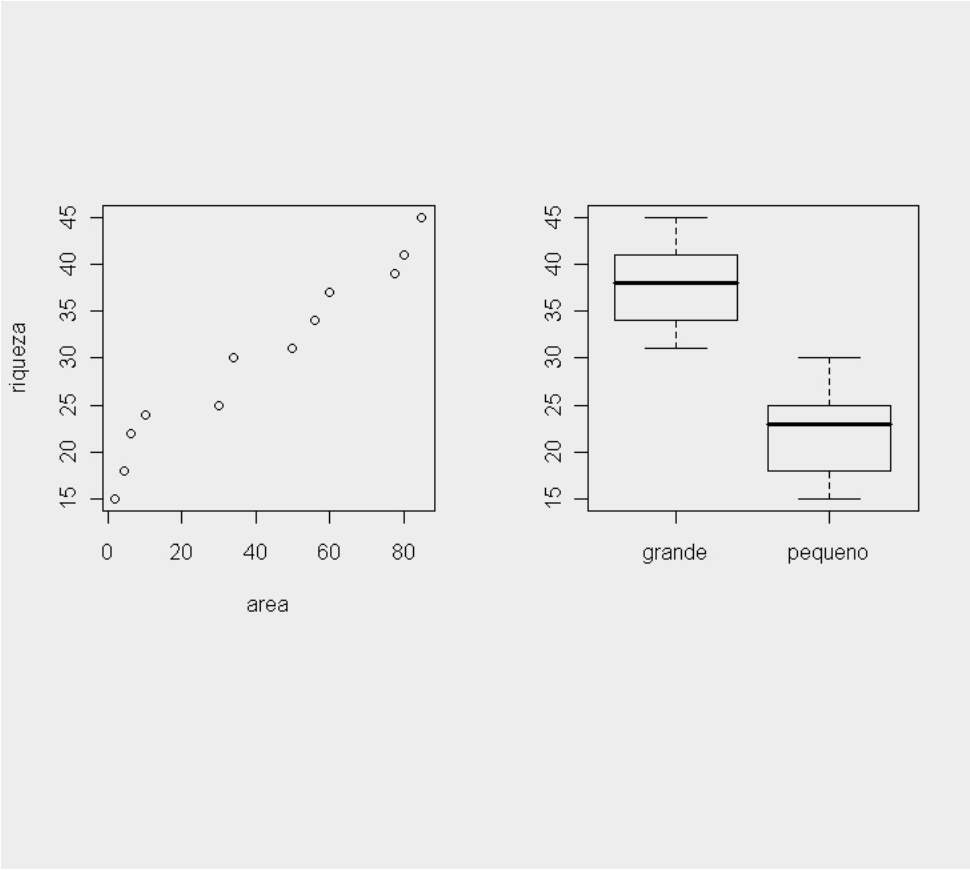


Exemplos de `par(mar=c())`

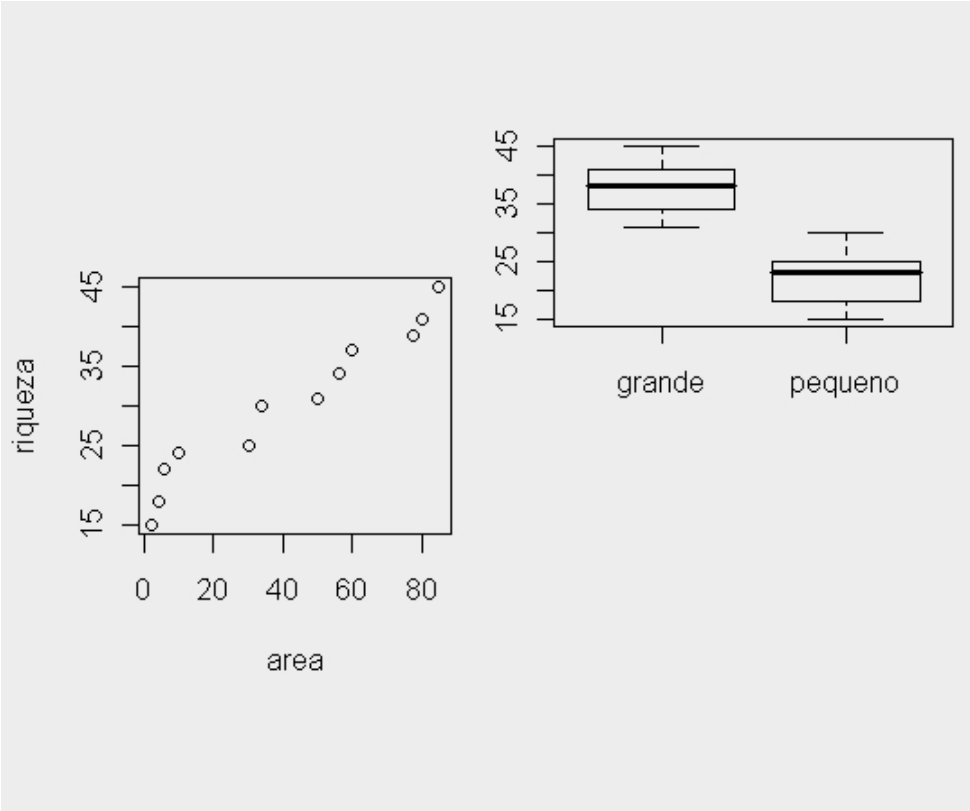
```
par(mfrow=c(2,1))
par(mar=c(4,14,2,6))
plot(riqueza~area)
boxplot(riqueza~area.cate)
```




```
par(mfrow=c(1,2))
par(mar=c(14,4,8,2))
plot(riqueza~area)
boxplot(riqueza~area.cate)
```




```
par(mfrow=c(1,2))
par(mar=c(8,4,8,1))
plot(riqueza~area)
par(mar=c(14,2,4,0.5))
boxplot(riqueza~area.cate)
```



6.4 Diferenças Entre Tipos De Gráfico

Infelizmente, a forma como se muda argumentos do `plot()`, `boxplot()` e `barplot()` não é sempre a mesma, ou seja, comandos que funcionam perfeitamente para o `plot()` podem não produzir efeito algum no `boxplot()`, e vice-versa. Esta característica, de fato, atrapalha um pouco, mas assim que se acostuma fica mais fácil. Há duas dicas para resolver este problema: (i) tente sempre jogar os argumentos para o `par()` pois às vezes eles podem não funcionar se chamadas por dentro do `plot()`, `boxplot()`, etc, mas irão funcionar pelo `par()`; (ii) descubra o nome em inglês do parâmetro que se quer mudar (`label`, `tick`, `legend`) e jogue no Google "legend boxplot". Com certeza, alguém já teve este mesmo problema, e entrando dentro da lista do R (as diversas que existem) ou em aulas disponibilizadas na internet, com certeza se acha uma solução.

Exercício 3 – Mudando diferentes Gráficos

Com as variáveis:

```
riqueza <- c(15,18,22,24,25,30,31,34,37,39,41,45)
area <- c(2,4.5,6,10,30,34,50,56,60,77.5,80,85)
area.cate <- rep(c(‘‘pequeno’’, ‘‘grande’’), each=6)
```

Crie:

```
plot(riqueza~area)
```

E agora:

```
 plot(riqueza~area, bty="l", tcl=0.3)
```

Perecebeu o que mudou?
Agora tente:

```
boxplot(riqueza~area.cate, bty=’’l’’, tcl=0.3)
```

O que aconteceu?
E agora?

```
par(bty=’’l’’)
par(tcl=0.3)
boxplot(riqueza~area.cate)
```

Viu só?

6.5 Inserindo mais Informações em Gráficos

Existem diversas informações que se pode incluir em um gráfico. Pode-se colocar uma letra para mostrar que este é o painel “a” e ao lado é o painel ”b”; pode-se colocar asteriscos para mostrar quais relações são significativas; pode-se desenhar flechas, outros pontos, uma infinidade de coisas. Tudo isto pode ser feito, mas requer funções comandos separados daqueles já passados pelo par() e plot(), boxplot() ou barplot(). Dentre as várias funções existentes para se inserir informações em gráficos, existem sete que são bastante úteis. Use:

Exercício 4

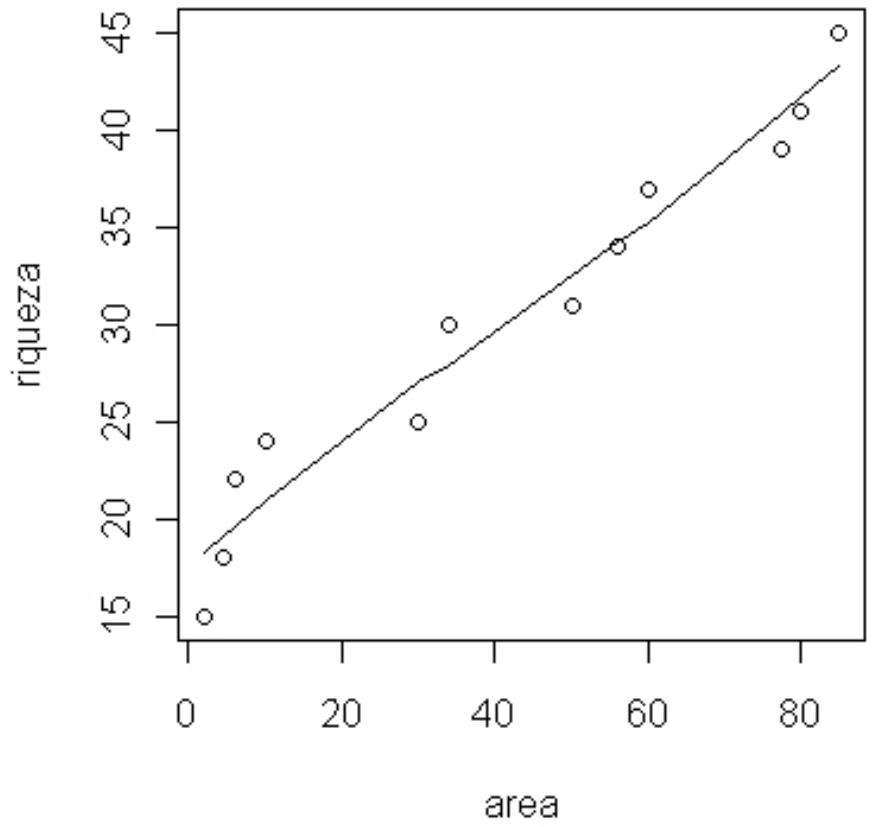
Usando as variáveis:

```
riqueza <- c(15,18,22,24,25,30,31,34,37,39,41,45)
area <- c(2,4.5,6,10,30,34,50,56,60,77.5,80,85)
abundancia <- rev(riqueza)
```

Crie gráficos inserindo os parâmetros abaixo.

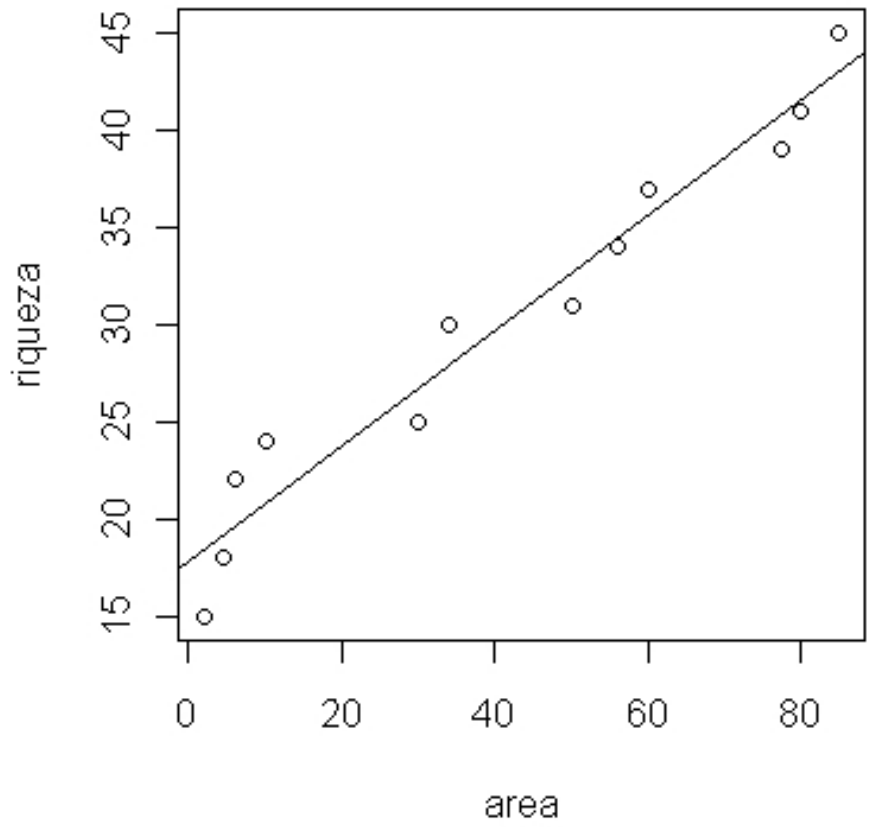
lines() Para inserir linhas retas ou curvas não-paramétricas (como `lowess`, `loess`, `gam`, etc).

```
plot(riqueza~area)
lines(lowess(area, riqueza))
```



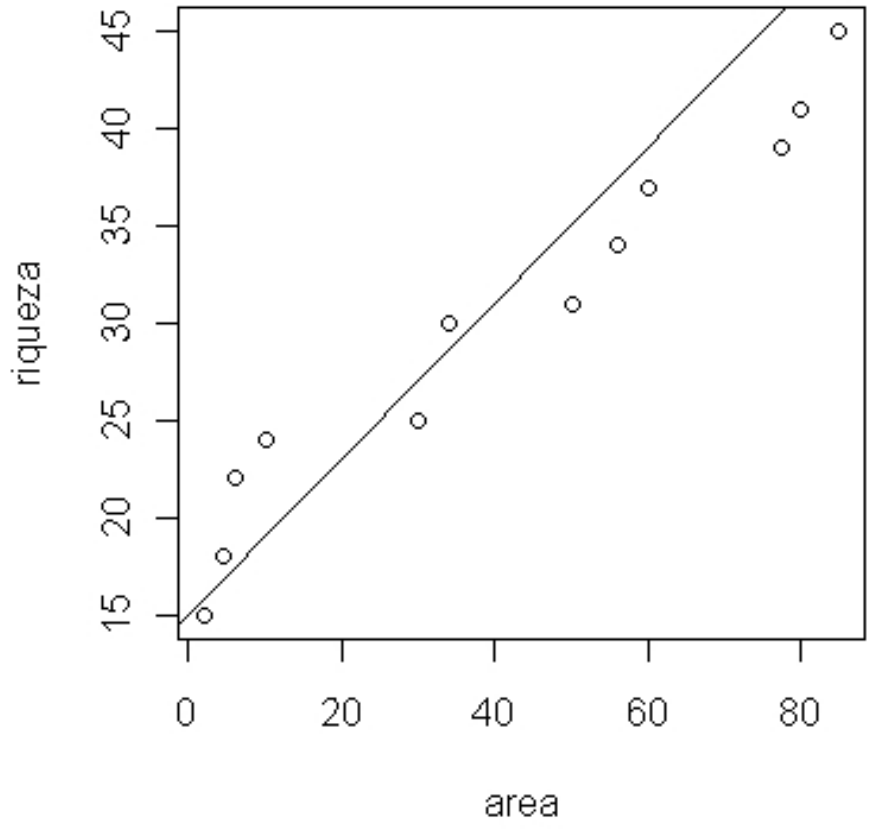
abline() Para inserir linhas de tendência criadas a partir de um modelo linear. Para isso é primeiro necessário criar o modelo, para depois criar a linha.

```
model <- lm(riqueza~area)
plot(riqueza~area)
abline(model)
```



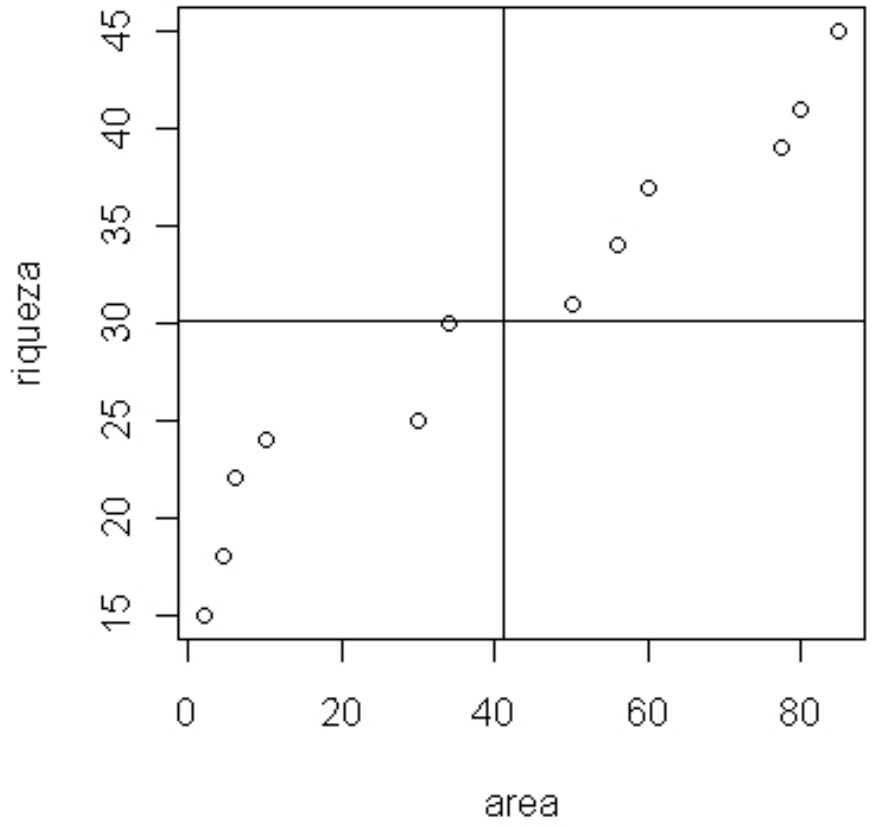
Com a função `abline` você pode também inserir uma linha reta com intercepto e inclinação definidos por você, com os dois primeiros argumentos:

```
plot(riqueza~area)
abline(15,0.4)
```



A função **abline** também serve para acrescentar linhas verticais e horizontais, com os argumentos **v** e **h**. No código abaixo traçamos estas linhas passando pelas médias das duas variáveis do diagrama de dispersão:

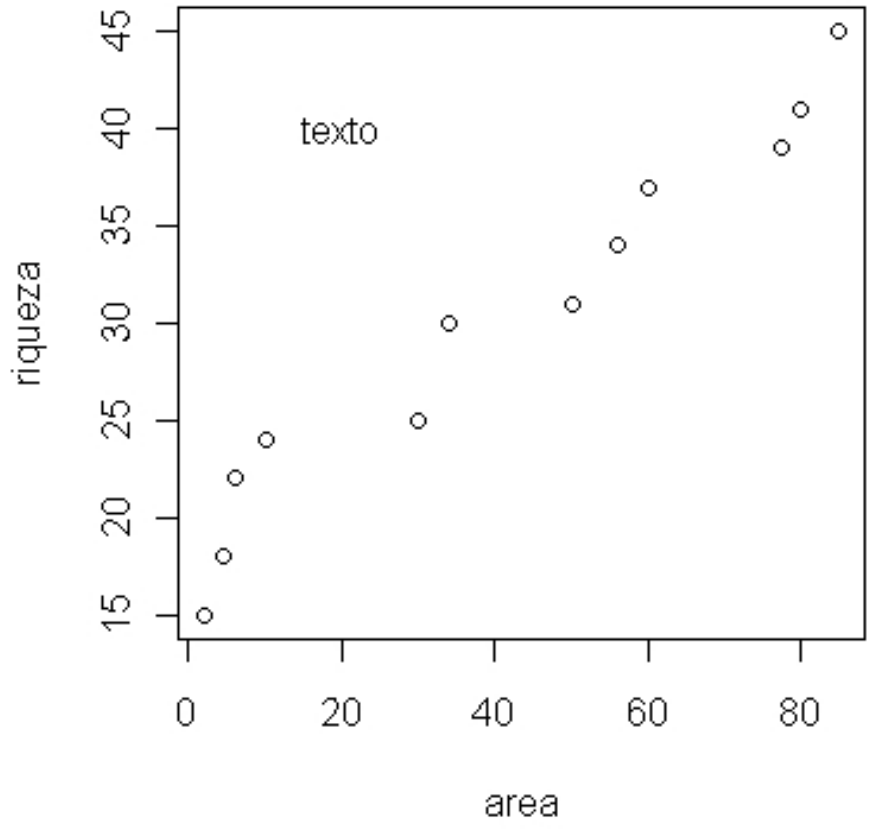
```
plot(riqueza~area)
abline(v=mean(area))
abline(h=mean(riqueza))
```



Você sabia? A reta da regressão linear simples sempre passa pelo ponto que é a interseção destas duas linhas.

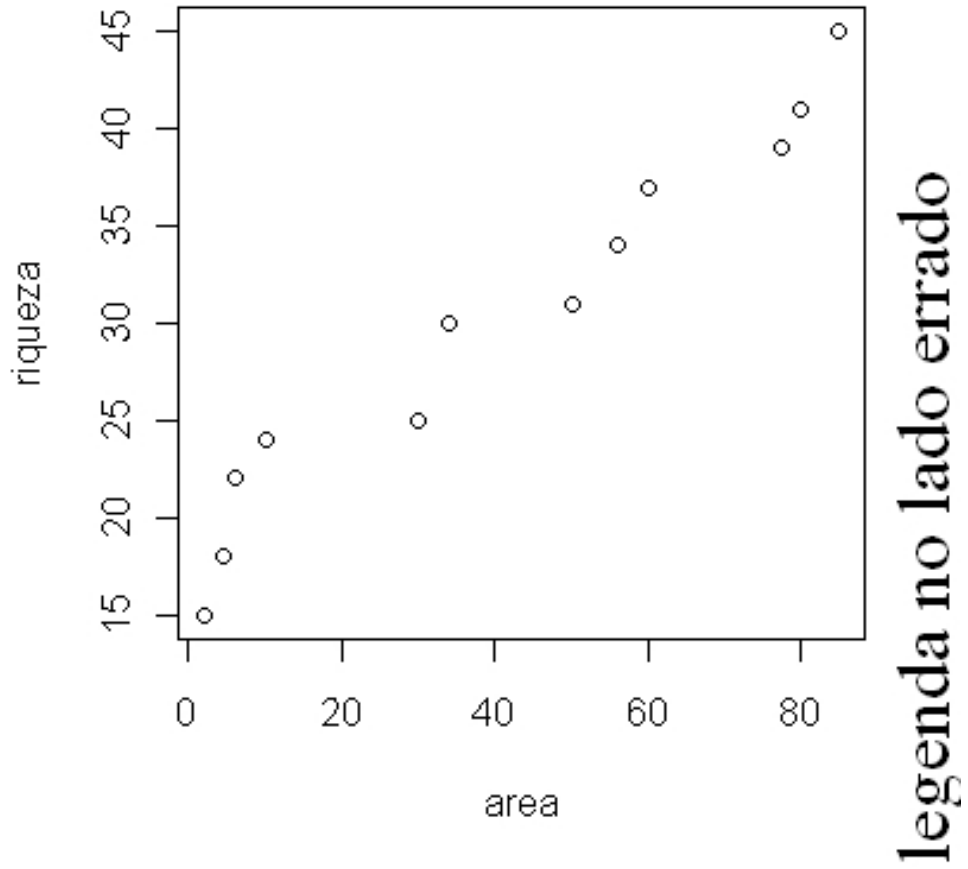
text() Para inserir texto dentro do gráfico. O texto pode ser uma letra, um símbolo (muito usado para mostrar diferenciar classes no gráfico), uma palavra ou até mesmo uma frase.

```
plot(riqueza~area)
text(20,40,"texto")
```



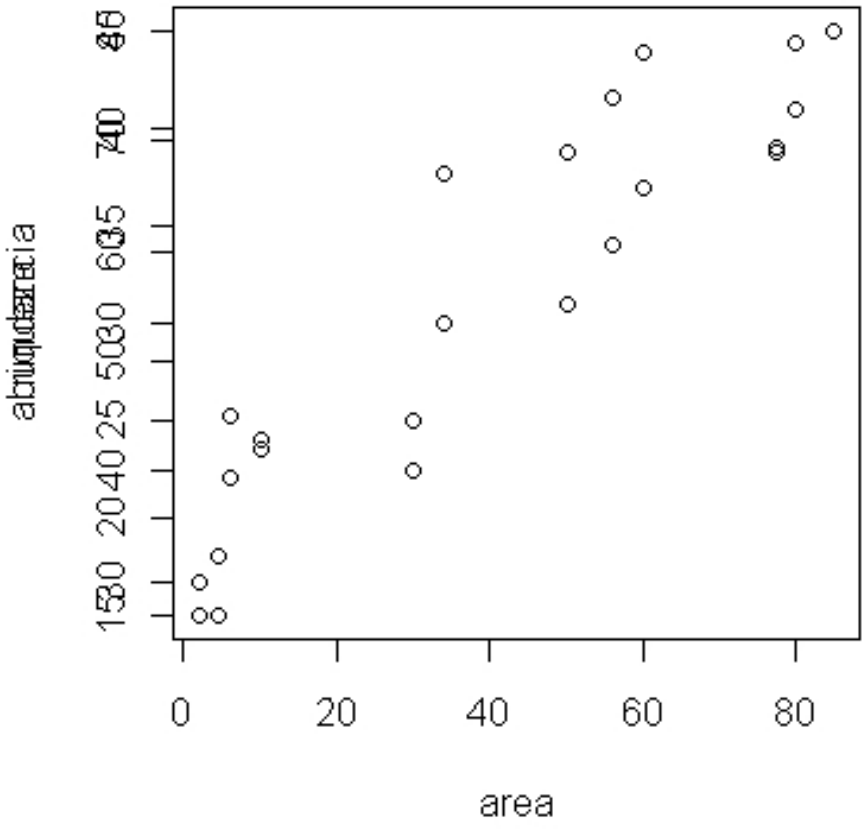
mtext() Este comando acrescenta texto nas margens do gráfico ou da janela gráfica. Seu uso mais freuquente é inserir legendas dos eixos. Apesar de ser possível controlar as legendas por dentro das funções `plot`, `boxplot` e `barplot`, o numero de parâmetros que se pode mudar é limitado. Quando se deseja um controle mais fino dos parâmetros, como posição, alinhamento, cor, tamanho da fonte, etc, é necessário usar `mtext()`.

```
plot(riqueza~area)
mtext("legenda no lado errado", side=4, line=0.9, at=20,cex=2, family="
    serif")
```



par(new=TRUE) Para sobrepor um novo gráfico a um gráfico já existente. Em vez de criar gráficos lado-a-lado, como em `par(mfrow=c())`, este argumento irá desenhar o novo gráfico sobre o gráfico anterior.

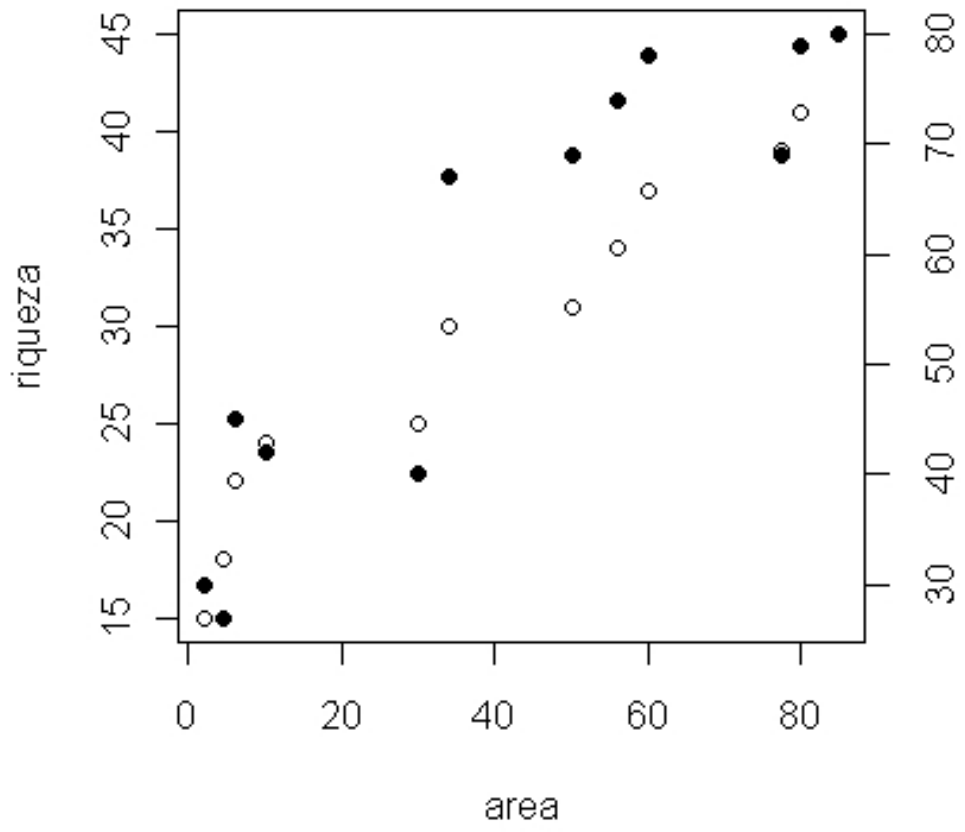
```
plot(riqueza~area)
par(new=TRUE)
plot(abundancia~area)
```



Mas reparem que aqui será necessário alguns ajustes para suprimir eixos e legendas. Em muitos casos quando se está inserindo informações será necessário suprimir parâmetros.

axis() Para se inserir um eixo novo. Esta função é bastante usada nos casos em que se deseja ter dois gráfico dentro de uma mesma figura (ver `par(new=TRUE)`), ou então se deseja controlar muitos dos parâmetros dos eixos (como em `mtext()`).

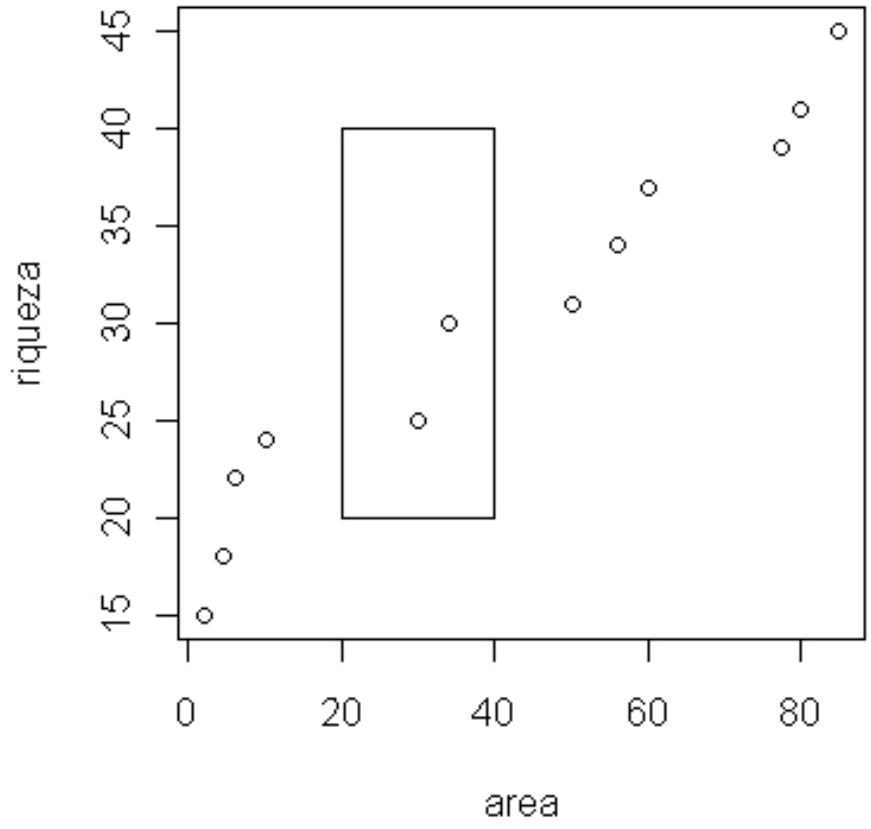
```
plot(riqueza~area)
par(new=TRUE)
plot(abundancia~area, axes=FALSE, ann=FALSE, pch=16)
axis(4)
```



Aqui no caso será necessário usar `axes=F` para suprimir a criação dos eixos do gráfico inicial de abundância e `ann=F` para suprimir a legenda de abundância do lado direito. Para para diferenciar os pontos entre os dois plots usar `pch=16`, ou qualquer outro número. Para inserir a legenda de abundância do lado direito será necessário usar `mtext()`, mas daí será necessário mudar outros parâmetros como distância da margem.

arrows(), rect(), polygon() Para inserir flechas ou **barras de erros** use `arrows()`. Já para inserir retângulos, polígonos e outros formatos use `rect()` e `polygon()`.

```
plot(riqueza~area)
rect(20,20,40,40)
```



6.6 Salvando Gráficos

Após ter feitos todos os gráficos desejados, é possível salvá-los em vários formatos, como jpeg, png, postscript, pdf. Consulte a ajuda do pacote grDevices para a lista completa e mais informações.

Após chegar ao gráfico final, ajustando todos os parâmetros desejados, você pode usar a função do R para criar o arquivo no formato desejado. Há funções para cada formato de arquivo, todas elas com o primeiro argumento `filename`, que especifica o nome do arquivo a salvar. Para criar um arquivo *jpg*, por exemplo, há a função `jpeg`:

```
jpeg(filename = "Algumnome.jpg")
```

Feito isso, o R agora irá enviar todos os resultados de comandos gráficos para este arquivo, que é fechado com a função `dev.off()`.

Exemplo

```
jpeg(filename = "Rplotaula.jpg", width = 480, height = 480,
     units = "px", pointsize = 12, quality = 100,
     bg = "white",  res = NA, restoreConsole = TRUE)

par(mfrow=c(1,2))
par(mar=c(14,4,8,2))
plot(riqueza~area)
boxplot(riqueza~area.cate)

dev.off()
```

6.6.1 Quatro Fatos Importantes sobre Arquivos de Figuras (e uma dica)

- Seu arquivo de figura só terá os parâmetros desejados se você executar todo o código após abrir o arquivo da figura, incluindo todos os comandos `par()`.
- Seu arquivo de figura só será salvo quando você executar o comando `dev.off()`. Até que isso aconteça, **todos os resultados de comandos gráficos continuarão a ser enviados para este arquivo**.
- Por isso, se você criar dois gráficos, o segundo substituirá o primeiro.
- Ao executar o comando `dev.off()`, o arquivo será gravado no diretório de trabalho. Se você quiser gravá-lo em outro lugar, terá que especificar o caminho completo no comando de criação do arquivo, no argumento `filename`.

DICA

Para gravar uma sequência de gráficos sem precisar dar vários comandos de abertura de arquivos, use no argumento do nome do arquivo a notação `nome%[número]d.extensão`, onde [número] normalmente são os números 01, 02 ou 03:

```
png("meugrafico%02d.png")
```

Com essa notação os gráficos gerados serão gravados com uma numeração sequencial, que tem `[numero]` algarismos. Por exemplo `[número] = 01` indica numeração sequencial de um algarismo, e você pode gravar até nove gráficos (*meugrafico1.png* a *meugrafico9.png*). Se `[número] = 02` a numeração sequencial tem dois algarismos, portanto você pode gravar até 99 figuras (*meugrafico01.png* a *meugrafico99.png*).

Se você gera mais gráficos do que este valor máximo, os excedentes sobrepõem os primeiros, na sequência. Para evitar isso, normalmente usamos `[número] = 03`, que permite gerar até 999 arquivos, o que é mais do que suficiente na maioria dos casos.

O padrão das funções de arquivos gráficos do tipo bitmap no R é gerar arquivos com o nome `Rplotxxx.extensão`, com numeração sequencial com 3 algarismos, ou seja:

```
bmp(filename = "Rplot%03d.bmp")
jpeg(filename = "Rplot%03d.jpg")
png(filename = "Rplot%03d.png")
tiff(filename = "Rplot%03d.bmp")
```

Para arquivos do tipo *postscript* e *pdf* o padrão é um pouco diferente, consulte a ajuda.

6.6.2 Dispositivos Gráficos

Para operar bem com arquivos gráficos no R, é preciso entender o conceito de **dispositivo gráfico** (*graphical device*).

Um dispositivo (*device*) é qualquer unidade de entrada (**I**=*input*) ou saída (**O**=*output*) em um computador. O teclado é um dispositivo de entrada (**I**), o monitor de vídeo de saída (**O**) e o disco rígido de ambos (**I/O**).

Em termos bem gerais, o R usa o padrão de sistemas da família UNIX e abre arquivos especiais (*device files*) para controlar cada dispositivo de saída gráfica [14]. Na prática podemos podemos pensar que o R cria uma "rota"para cada dispositivo, e que você pode ter várias destas "rotas"abertas ao mesmo tempo, **mas apenas uma ativa por vez**.

Abrindo e trocando de dispositivos Quando executamos um comando com resultado gráfico, o padrão do R é enviar seu resultado para um dispositivo de tela. Se não há nenhum aberto, este dispositivo de tela será aberto. Por exemplo, se iniciamos o R e executamos um comando `plot`:

```
plot(riqueza~area)
```

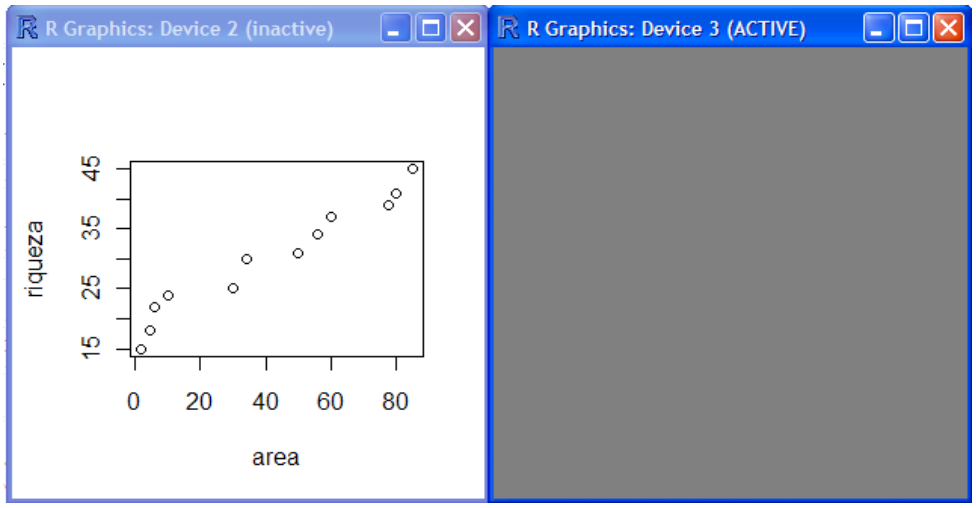
uma janela se abrirá para exibir a figura nele. Este é um **dispositivo gráfico de tela**, normalmente do tipo `X11` em sistemas UNIX, e do tipo `windows` em sistemas Microsoft Windows.

É possível também abrir mais dispositivos de tela, com o comando:

```
> x11()
```

que funciona nos dois tipos de sistemas operacionais. Ao executar este comando, uma nova janela gráfica em branco se abrirá, na margem superior da qual você verá a indicação de que agora este é o dispositivo ativo:

[14] veja este link para uma explicação mais precisa



O dispositivo ativo é que receberá o resultado de todos os comandos gráficos. Assim,você pode criar um novo gráfico neste dispositivo, ou alterar seus parâmetros, sem afetar o gráfico que está na outra janela. Quando você abre um novo dispositivo, os parâmetros são os mantidos por padrão no R, que você obtém com o comando `par()`. Para mudá-los, é preciso usar o comando `par`, como explicado nas seções anteriores.

Para trocar de dispositivo ativo, use a função `dev.set(which=)`, cujo o argumento `which` é o número do dispositivo que você quer tornar ativo. Por exemplo, para voltar à janela do dispositivo 2 execute:

```
> dev.set(which=2)
```

Qual o Dispositivo Ativo? Enquanto temos apenas dispositivos de tela abertos, é fácil descobrir qual é o ativo, pois esta informação é exibida na janela de cada um. Mas quando abrimos um ou mais dispositivos de arquivo, como:

```
> png("figura%02d.png")
> pdf("figura%02d.pdf")
```

é fácil se perder. Para que isso não aconteça há as funções `dev.list`, para listar todos os dispositivos abertos, e `dev.cur`, que retorna o dispositivo ativo:

```
> dev.list()
        windows         windows png:figura%02d.png             pdf
              2               3                   4               5
> dev.cur()
pdf
  5
> dev.set(3)
windows
      3
> dev.cur()
windows
      3
```

Você fecha um dispositivo com o comando `dev.off(which=n)`, em que *n* é o número do dispositivo. Se este argumento é omitido, o valor padrão é o dispositivo ativo:

```
> dev.list()
          windows             windows png:figura%02d.png               pdf
                2                   3                   4                 5
> dev.cur()
pdf
  5
> dev.off()
windows
      2
> dev.list()
          windows             windows png:figura%02d.png
                2                   3                   4
> dev.cur()
windows
      2
```

Quando o dispositivo ativo é fechado, o seguinte na lista de dispositivos torna-se o ativo. O último comando acima ilustra isto.

Exercício 5

Crie diferentes gráficos em diferentes dispositivos. Por fim salve-os em jpeg.

Para saber mais como salvar gráficos em jpeg use a função `jpeg`.

7 Modelos Lineares

Links externos para o material referente a esse tópico no wiki da disciplina:

- Tutorial
- Exercícios
- Apostila

São chamados modelos lineares aqueles que apresentam uma relação entre variáveis que seja **linear nos parâmetros**. Essa linearidade implica que **matematicamente** a variação de cada um dos parâmetros é independente dos demais parâmetros do modelo.

Em termos gerais, podemos reconhecer dois grandes grupos clássicos de modelos lineares:

- **Modelos de Regressão**.

- **Modelos de Análise de Variância**.

Nesse tópico utlizaremos os arquivos de dados:

- Levantamento em caixetais: caixeta.csv (apagar extensão .pdf)
- Dados de biomassa de árvores: esaligna.csv (apagar extensão .pdf)
- Inventário em florestas plantadas:egrandis.csv (apagar extensão .pdf)
- Experimento sobre o crescimento de mudas em viveiro. altura-mudas.csv (apagar extensão .pdf)

7.1 Regressão Linear

Os modelos lineares de regressão são utilizados para modelar a relação entre variáveis quantitativas:

- **Variável Resposta** (y): variável quantitativa (também chamada de variável dependente).
- **Variáveis Preditoras** (x): variáveis quantitativas (também chamadas de variáveis independentes).

7.1.1 A função "lm"

A função utilizada para construir modelos lineares de regressão é a função `'lm` que tem os seguintes argumentos principais:

```
lm(formula, data, weights, subset, na.action)
```

- `formula` - é uma **fórmula estatística** que indica o modelo a ser ajustado. Possui a mesma forma básica que foi vista na funções gráficas.
- `data` - o conjunto de dados (`data.frame`).
- `weights` - são os *pesos* para regressão ponderada.
- `subset` - um vetor com as condições que definem um **sub-conjunto** dos dados.
- `na.acation` - função que especifica o que fazer no caso de observações perdidas (`NA`). O valor default é `na.omit` que elimina as linhas (observações) que possuem observações perdidas nas variáveis definidas na fórmula.

Vejamos um exemplo simples:

```
> egr = read.csv("egrandis.csv",header=T)
> egr[1,]
    especie rot   regiao  inv faz proj talhao parcela arv fuste cap ht hdom
1 E.grandis   1 Botucatu 1995  36   33      1       1   1     1  57 27   NA
    idade carac      dap
1 7.49315     N 18.14366
>
> hipso1 = lm( ht ~ dap, data=egr )
> class(hipso1)
[1] "lm"
>
```

Uma Palavra sobre o Argumento "formula" O argumento fórmula nos modelos lineares é bastante diverso das fórmulas matemáticas usuais. Nesse argumento, sinais de mais e de menos, símbolos como circunflexo (^) e asterisco (*) têm significado bastante diferente dos significados usuais matemáticos.

Apresentaremos agora alguns aspectos básicos do argumento:

- `y ~ x` indica: *modele y como* ***função estatística*** *de x*;
- `y ~ x1 + x2` indica: *modele y como* ***função estatística*** *das variáveis x1 e x2* (efeito aditivo dos modelos lineares);

Se quisermos utilizar os símbolos matemáticos no sentido matemático usual **dentro** de uma fórmula estatística, temos que utilizar a função `I()`:

- `y ~ I( x1^2 * x2^3 )` indica: *modele y como função estatística* ***da variável*** *(x1^2 * x2^3)*;
- `y ~ I( x1 / x2 )` indica: *modele y como função estatística* ***da variável*** *(x1/x2)*;

No caso de utilizarmos **funções matemáticas** específicas a função `'I()` torna-se desnecessária:

- `log(y) ~ log(x)` indica: *modele o* ***log(y)*** *com função estatística da variável* ***log(x))***;
- `log(y) ~ log(x1^2 * x2)` indica: *modele o* ***log(y)*** *com função estatística da variável* ***log(x1^2 * x2))***;

Mais detalhes sobre o argumento `formula` serão apresentados mais adiante.

7.1.2 Exercícios

Exercício: Relação Diâmetro-Altura em Florestas Plantadas

Ajuste um modelo de regressão linear simples da altura (`ht`) em função do DAP (`dap`) das árvores de floresta plantada (:dados:dados-egrandis) para cada uma das rotações (`rot`).

Exercício: Equação de Biomassa para Árvores de Eucalyptus saligna

Utilizando o conjunto de dados de *E. Saligna* (:dados:dados-esaligna), construa um modelo de regressão da biomassa do tronco das árvores (`tronco`) em função do diâmetro (`dap`) e altura (`ht`), utizando dois modelos:
$b_i = \beta_0 + \beta_1(d_i^2\, h_i) + \varepsilon_i$
e
$\ln(b_i) = \beta_0 + \beta_1 \ln(d_i) + \beta_2 \ln(h_i) + \varepsilon_i$
onde:

- b_i é biomassa do tronco;
- d_i é o DAP da árvore;
- h_i é a altura toral da árvore;

7.1.3 Funções que Atuam sobre Objetos "lm"

O objeto produzido pela função `lm` tem classe `lm` (*linear model*), ou seja é um modelo linear. Como modelo linear, esse objeto receberá tratamento particular se utilizarmos algumas funções básicas sobre ele.

- **summary:** a função `summary` apresenta um resumo do modelo linear com:
  1. estatísticas descritivas dos resíduos;
  2. teste *t* dos coeficientes de regressão;
  3. erro padrão da estimativa;
  4. coeficiente de determinação e coef. de det. ajustado;
  5. teste *F* geral do modelo.

```
> summary(hipso1)

Call:
lm(formula = ht ~ dap, data = egr)

Residuals:
     Min       1Q   Median       3Q      Max
-12.9306  -2.1109  -0.5408   1.6642  20.9390

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.79604    0.12120   6.568 5.63e-11 ***
dap          1.27232    0.01006 126.459  < 2e-16 ***
---
```

```
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1

Residual standard error: 3.093 on 4800 degrees of freedom
Multiple R-Squared: 0.7691,     Adjusted R-squared: 0.7691
F-statistic: 1.599e+04 on 1 and 4800 DF,  p-value: < 2.2e-16
```

- **anova:** a função `anova` apresenta a Tabela de análise de variância, tendo as variáveis preditoras como fatores:

```
> anova(hipso1)
Analysis of Variance Table

Response: ht
           Df Sum Sq Mean Sq F value    Pr(>F)
dap         1 153020  153020   15992 < 2.2e-16 ***
Residuals 4800  45929      10
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1
```

- **plot:** a função `plot` apresenta uma série de gráficos para análise do modelo. Ela possui o argumento `which` que define quais dos seis gráficos pré-definidos se deseja ver:

which	O que a função faz	Verificação do modelo
`which=1`	gráfico de dispersão resíduo versus valor ajustado	Linearidade na relação x-y
`which=2`	gráfico quantil-quantil normal dos resíduos	Normalidade
`which=3`	gráfico de dispersão da raiz quadrada do valor absoluto do resíduo padronizado versus valor ajustado	Homogeneidade de variâncias
`which=4`	distância de Cook por observação	Observações influentes
`which=5`	gráfico de dispersão do resíduo padronizado versus *leverage* (medida de influência) , com a distância de Cook	Observações influentes
`which=6`	gráfico de dispersão da distância de Cook versus *leverage* (medida de influência)	Observações influentes

O valor default do argumento `which` é `which = c(1:3, 5)`.

```
> plot( hipso1 )
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
>
```

- **coef:** a função `coef` retorna os coeficientes de regressão do modelo linear:

```
> coef( hipso1 )
(Intercept)         dap
  0.7960402   1.2723242
>
```

- **residuals:** a função `residuals` (também pode ser evocada por `resid`) retorna os resíduos do modelo linear.
- **fitted:** a função `fitted` (também pode ser evocada por `fitted.values`) retorna os *valores ajustados* do modelo linear.

```
> plot( resid( hipso1 ), fitted( hipso1 ) )
```

- **predict:** a função `predict` retorna os valores *preditos* para novas observações:

```
> predict(hipso1, data.frame(dap=c(10,50,100)))
        1         2         3
 13.51928  64.41225 128.02846
>
> predict(hipso1, data.frame(dap=range(egr$dap)))
        1         2
 6.060955 38.055431
>
```

7.1.4 Exercícios

Exercício: Analisando os Modelos de Regressão

Utilizando as funções apresentadas acima analise os modelos de regressão construídos nos exercícios anteriores com relação a:

1. adequação das pressuposições dos modelos lineares;
2. significância das estimativas dos coeficientes de regressão;
3. qualidade dos modelos para uso em predições.

Exercício: Realizando Predições

Considere as árvores da tabela abaixo:

Árvore	DAP	Altura
1	10 cm	12 m
2	20 cm	25 m
3	30 cm	40 m

Faça a predição da biomassa do tronco das árvores com base nos dois modelos de equação de biomassa ajustados.

7.1.5 Regressão Ponderada

A regressão ponderada é utilizada para corrigir o problema de heterogeneidade de variâncias.

Consideremos o exercício do modelo de equação de biomassa do tronco em função do diâmetro e altura. O modelo original apresenta claramente problemas com a pressuposição de homogeneidade de variâncias:

```
> esa = read.csv("esaligna.csv",header=T)
> plot( lm( tronco ~ I(dap^2 * ht), data=esa ) , which=c(1,3) )
Hit <Return> to see next plot:
Hit <Return> to see next plot:
>
```

Se o modelo for ponderado por uma potência do inverso da variável preditora (`1/(dap^2 * ht)`), talvez se torne um modelo com variância homogênea.

```
> plot( lm( tronco ~ I(dap^2*ht), data=esa, weights=1/(dap^2*ht)^0.5 ),
    which=3 )
> plot( lm( tronco ~ I(dap^2*ht), data=esa, weights=1/(dap^2*ht)^0 ), which
    =3 )
> plot( lm( tronco ~ I(dap^2*ht), data=esa, weights=1/(dap^2*ht)^1 ), which
    =3 )
> plot( lm( tronco ~ I(dap^2*ht), data=esa, weights=1/(dap^2*ht)^2 ), which
    =3 )
> plot( lm( tronco ~ I(dap^2*ht), data=esa, weights=1/(dap^2*ht)^3 ), which
    =3 )
>
```

Qual dos valores de potência (0.5; 0; 1; 2; 3) lhe parece mais adequado?

7.1.6 Exercícios

Exercício: Modelos de Biomassa para Eucalyptus saligna

Utilizando o mesmo modelo do exemplo acima, ajuste modelos de equação de biomassa para

1. **biomassa total** (`total`) e
2. **biomassa de ramos** (`sobra`),

de modo que os modelos não possuam problema de heterogeneidade de variância.

7.1.7 Variável Factor como Variável Indicadora (dummy)

Uma forma de incluirmos variáveis categóricas em modelos de regressão é através do uso de **variáveis indicadoras**. Para isso, as variáveis categóricas no R devem ser vistas como uma variável `'factor`.

A variável `'factor` indica uma variável que possui **níveis** (`levels`) sendo, portanto, uma variável categórica típica dos modelos estatísticos.

Esse tipo de variável existe no R para tornar mais fácil a modelagem estatística. Assim, quando o R lê um conjunto de dados e encontra uma variável **alfa-numérica**, ele automaticamente assume que se trata de uma variável `'factor`.

Vejamos como exemplo os dados de inventário floresta em floresta plantada:

```
> egr = read.csv("egrandis.csv",header=T)
>
> names(egr)
 [1] "especie" "rot"     "regiao"  "inv"     "faz"     "proj"    "talhao"
 [8] "parcela" "arv"     "fuste"   "cap"     "ht"      "hdom"    "idade"
[15] "carac"   "dap"
>
> class( egr$regiao )
[1] "factor"
> class( egr$especie )
[1] "factor"
> class( egr$rot )
[1] "integer"
> class( egr$faz )
[1] "integer"
>
```

Note que as variáveis `regiao` e `especie` foram assumias como `factor`.

Note também que as variáveis `rot` (rotação) e `faz` (fazenda) embora também sejam variáveis categóricas, elas foram codificadas por números inteiros e, consequentemente, o R assumiu tratar-se de variáveis quantitativas.

Nos **modelos lineares de regressão**, as variáveis `factor` podem ser assumidas automaticamente como variáveis indicadoras (variáveis dummy).

```
> hipso2 = lm( ht ~ dap + regiao , data=egr )
> summary( hipso2 )

Call:
lm(formula = ht ~ dap + regiao , data = egr)

Residuals:
     Min       1Q   Median       3Q      Max
-10.6196  -1.6235  -0.3575   1.2476  19.5109

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     5.079650   0.145714   34.86   <2e-16 ***
dap             1.116119   0.009403  118.70   <2e-16 ***
regiaoBotucatu -3.627353   0.100221  -36.19   <2e-16 ***
regiaoItatinga -3.827592   0.100715  -38.00   <2e-16 ***
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1

Residual standard error: 2.647 on 4798 degrees of freedom
Multiple R-Squared: 0.831,      Adjusted R-squared: 0.8309
F-statistic:  7866 on 3 and 4798 DF,  p-value: < 2.2e-16
```

Nesse caso (`ht ~ dap + regiao`) a variável região entrou alterando o **intercepto** da regressão entre altura (`ht`) e diâmetro (`dap`).

Note que além do coeficiente de inclinação para variável `dap` aparecem coeficientes de regressão associados às variáveis `regiaoBotucatu` e `regiaoItatinga`. O que significa isso?

O modelo ajustado pela fórmula `ht ~ dap + regiao` é:

onde:

- `h_i` é a altura das árvores;
- `d_i` é o DAP das árvores;
- `I_{\textrm{Botucatu}}` é a variável indicadora para região Botucatu, isto é, ela tem valor **1** se a região for Botucatu e valor **0** (zero) se a região não for Botucatu;
- `I_{\textrm{Itatinga}}` é a variável indicadora para região Itatinga.

A variável região tem três níveis (`levels`)

```
> levels(egr$regiao)
[1] "Bofete"   "Botucatu" "Itatinga"
>
```

O R cria automaticamente 2 variáveis indicadoras, uma para Botucatu e outra para Itatinga, pois a região de Bofete (primeiro nível do fator) é assumida como o *default*.

Como se utiliza o modelo para predição?

- Para Bofete a predição é: `\widehat{h}_i = \beta_0 + \beta_2 d_i`
- Para Botucatu a predição é: `\widehat{h}_i = (\beta_0 + \beta_3) + \beta_2 d_i`
- Para Itatinga a predição é: `\widehat{h}_i = (\beta_0 + \beta_4) + \beta_2 d_i`

É possível ajustar um **modelo de interação completo** do diâmetro com a variável região, alterando o *intercepto* **e** a *inclinação* do modelo em cada regiões:

```
> hipso3 = lm( ht ~ dap * regiao, data=egr )
> summary( hipso3 )

Call:
lm(formula = ht ~ dap * regiao, data = egr)

Residuals:
     Min       1Q   Median       3Q      Max
-12.8439  -1.5492  -0.2357   1.2582  19.3736

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)         7.99813    0.20736  38.570  < 2e-16 ***
dap                 0.90370    0.01436  62.935  < 2e-16 ***
regiaoBotucatu     -9.03699    0.25969 -34.799  < 2e-16 ***
regiaoItatinga     -5.74018    0.29200 -19.658  < 2e-16 ***
dap:regiaoBotucatu  0.45060    0.01981  22.750  < 2e-16 ***
dap:regiaoItatinga  0.10746    0.02503   4.293 1.80e-05 ***
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1

Residual standard error: 2.508 on 4796 degrees of freedom
Multiple R-Squared: 0.8484,     Adjusted R-squared: 0.8483
F-statistic:  5369 on 5 and 4796 DF,  p-value: < 2.2e-16
```

Note que se quisermos usar uma variável como indicadora, mas ela foi codificada como variável numérica, teremos que **forçar** sua transformação em variável `factor`:

```
 coef( lm( ht ~ dap * rot, data = egr ) )
 (Intercept)          dap          rot      dap:rot
 3.790676759  1.206524873 -1.556650365  0.004740839
>
>
> coef( lm( ht ~ dap * as.factor(rot) , data = egr ) )
        (Intercept)                 dap     as.factor(rot)2 dap:as.factor(
           rot)2
        2.234026394         1.211265712        -1.556650365
           0.004740839
```

7.1.8 Exercícios

Exercício: Altura das Árvores Dominantes em Floresta Plantada

Considere o seguinte modelo (*modelo de Schumacher*):
$\ln(y_i) = \beta_0 + \beta_1(1/x_i) + \varepsilon_i$
Ajuste esse modelo de regressão à altura das árvores dominantes (`hdom`) em função da idade (`idade`) das árvores da floresta plantada (:dados:dados-egrandis).
Compare um modelo geral (ajustado a todos os dados) com um modelo ajustado por região.
Compare os resíduos do modelo geral com os resíduos do modelo por região, analisando **a distribuição do resíduo por região**.

Exercício: Relação Altura-Diâmetro de Árvores de Caixeta

Ajuste modelos de regressão linear da altura (`h`) em função do diâmetro (`dap`) **somente para árvores de caixeta** (*Tabebuia cassinoides*) nos diferentes caixetais (`local`).
Considere nessa tarefa os seguintes modelos:

- **Modelo A:** $h_i = \beta_0 + \beta_1 d_i + \varepsilon_i$
- **Modelo B:** $\ln(h_i) = \beta_0 + \beta_1 \ln(d_i) + \varepsilon_i$
- **Modelo C:** $h_i = \beta_0 + \beta_1 d_i + \beta_2 d_i^2 + \varepsilon_i$

7.2 Análise de Variância

Os objetivos dos modelos lineares de análise de variância são bem diferentes dos modelos lineares de regressão. Nos modelos de regressão a questão central é estimar parâmetros, seja para explicar relações, seja para fazer predições.

Nos modelos de análise de variância a questão é comparar a importância de **fatores** sobre o comportamento da variável resposta.

7.2.1 Experimento em Blocos Casualizados

Tomemos como exemplo os dados do experimento de mudas no viveiro (dados:dados-mudas). Nesse experimento temos os seguintes fatores:

- Espécie (`especie`),
- Bloco (`bloco`), e
- Substrato (`substrato`).

O experimento deseja saber se existe diferença entre os substratos no crescimento em altura (**altura**) das mudas. O delineamento foi em blocos casualizados (**bloco**).

Para visualizar esse experimento podemos ler os dados do arquivo altura-mudas.csv (apagar extensão .pdf), através da função **plot**:

```
> mudas = read.csv("dados/altura-mudas.csv",header=T)
> summary(mudas)
     especie        bloco        substrato         altura
 paineira:60   Min.   :1.0   Min.   : 1.0   Min.   : 15.40
 tamboril:60   1st Qu.:2.0   1st Qu.: 3.0   1st Qu.: 32.21
               Median :3.5   Median : 5.5   Median : 46.00
               Mean   :3.5   Mean   : 5.5   Mean   : 49.20
               3rd Qu.:5.0   3rd Qu.: 8.0   3rd Qu.: 65.00
               Max.   :6.0   Max.   :10.0   Max.   :105.12
>
>
> plot( altura ~ bloco + substrato , data=mudas , subset=especie=="paineira
    ")
Hit <Return> to see next plot:
Hit <Return> to see next plot:
>
> plot( altura ~ bloco + substrato , data=mudas , subset=especie=="tamboril
    ")
Hit <Return> to see next plot:
Hit <Return> to see next plot:
>
```

Para ajustar um modelo linear num experimento, podemos utilizar a função **lm** como no caso da regressão linear:

```
> muda.pai = lm( altura ~ as.factor(bloco) + as.factor(substrato), data=
    mudas, subset= especie=="paineira" )
> class(muda.pai)
[1] "lm"
>
> muda.tam = lm( altura ~ as.factor(bloco) + as.factor(substrato), data=
    mudas, subset= especie=="tamboril" )
> class(muda.tam)
[1] "lm"
>
```

Para analisar as pressuposições do modelo utilizamos a função **plot**, da mesma forma que se faz na regressão linear.

Sendo um experimento, o interesse principal é verificar a importância dos **fatores**: os tratamentos (**substrato**) e os blocos (**bloco**). Para isso utilizamos a função **anova**:

```
> anova( muda.pai )
Analysis of Variance Table

Response: altura
                    Df  Sum Sq Mean Sq F value    Pr(>F)
as.factor(bloco)     5  3822.1   764.4  13.664 4.016e-08 ***
```

```
as.factor(substrato)  9 27204.4  3022.7  54.030 < 2.2e-16 ***
Residuals            45  2517.5    55.9
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1
>
> anova( muda.tam )
Analysis of Variance Table

Response: altura
                     Df Sum Sq Mean Sq F value    Pr(>F)
as.factor(bloco)      5 2582.6   516.5  5.1933 0.0007594 ***
as.factor(substrato)  9 9181.3  1020.1 10.2570 2.177e-08 ***
Residuals            45 4475.6    99.5
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1
>
```

7.2.2 Exercícios

Exercício: Altura dos Caixetais

Verifique se existe diferenças estatísticamente significativas na **altura média** dos caixetais (caixeta.csv (apagar extensão .pdf)).
Será que os caixetais diferem em termos de **altura máxima** ou **área basal**?

7.2.3 Outros Delineamentos Experimentais

Considere o experimento de mudas de espécies arbóreas. Se ao invés de trabalhar com a **espécie** em duas análises separadas, houvesse interesse em fazer uma análise conjunta das duas espécies, verificando a interação entre espécie e substrato.

Nesse caso, o experimento se torna um **experimento fatorial 2 x 10**:

```
> muda.sp = lm( altura ~ as.factor(bloco) + especie * as.factor(substrato),
    data=mudas )
> anova(muda.sp)
Analysis of Variance Table

Response: altura
                           Df Sum Sq Mean Sq F value    Pr(>F)
as.factor(bloco)            5   3956     791  7.9598 2.673e-06 ***
especie                     1   3183    3183 32.0265 1.606e-07 ***
as.factor(substrato)        9  32910    3657 36.7909 < 2.2e-16 ***
especie:as.factor(substrato) 9   3476     386  3.8853 0.0003163 ***
Residuals                  95   9442      99
---
```

```
Signif. codes:  0     ***     0.001     **     0.01     *     0.05     .     0.1
          1
```

No que a fórmula para o experimento é apresentada de modo diferente:

```
altura ~ as.factor(bloco) + especie * as.factor(substrato)
```

O elemento **especie * as.factor(substrato)** inclui todos os efeitos ligados a interação entre espécie e substrato, e que aparecem na tabela de análise de variância:

- **especie** (com 1 grau de liberdade) se refere ao **efeito principal** da espécie;
- **as.factor(substrato)** (com 9 graus de liberdade) se refere ao **efeito principal** do substrato;
- **especie:as.factor(substrato)** (com 9 graus de liberdade) se refere à **interação** espécie x substrato.

O elemento chave nessa fórmula é o asterisco (*) que representa todos os efeitos ligados a interação entre dois fatores. Como foi dito, na fórmula estatística os sinais convencionais de operações matemáticas tem outro significado. A tabela abaixo detalha os símbolos utilizados para definir diferentes delineamentos experimentais.

Símbolos utilizados nas Fórmulas Estatísticas para definir diferentes Delineamentos Experimentais

Expressão	Significado
Y ~ X	Modele *Y* como função estatística de *X*
A + B	inclui ambos os fatores *A* e *B*
A - B	inclui todos os efeitos em *A*, exceto os que estão em *B*
*A * B*	*A + B + A:B*
A / B	*A + B %in% (A)* modelos hierárquicos
A:B	efeito da interação entre os fatores *A* e *B*
B %in% A	efeitos de *B* dentro dos níveis de *A*
A^m	todos os termos de *A* cruzados até à ordem *m*

Aliadas a esses símbolos, o R possui uma série de funções que permitem a análise de virtualmente qualquer delineamento experimental. Esse tópico requer, no entanto, um curso específico de análise experimental utilizando o R, e vai muito além do objetivo desse curso introdutório.

7.2.4 Exercícios

Exercício: Fatores que Influenciam a Altura em Florestas Plantadas I

Utilizando os dados de árvores de floresta plantada (dados:dados-egrandis), tome a altura média das árvores dominantes (média de `hdom` por `parcela`) como variável resposta e verifique a influência dos fatores: região (`regiao`) e rotação (`rot`). Para discussão: a relação entre esses fatores deve ser de **interação** ou **hierárquica**?

Exercício: Fatores que Influenciam a Altura em Florestas Plantadas II

Utilizando os dados de árvores de floresta plantada (dados:dados-egrandis), tome a altura média das árvores dominantes (média de `hdom` por `parcela`) como variável resposta e verifique a influência dos fatores: região (`regiao`) e projeto (`proj`). Para discussão: a relação entre esses fatores deve ser de **interação** ou **hierárquica**?

7.2.5 Explorando a Interação entre Fatores

Freqüentemente, a interpretação da **interação** entre dois ou mais fatores é espinhosa e chegar a conclusões baseado apenas na tabela de análise de variância pode gerar equívocos. Uma análise gráfica de interações é sempre instrutiva.

Existe no R a função `interaction.plot` que permite construir gráficos de interação entre fatores que facilitam a interpretação dos resultados estatístico. Seus argumentos principais são:

```
function (x.factor, trace.factor, response, fun = mean)
```

- `x.factor` é o fator que ficará nas abscissas (eixo-x);
- `trace.factor` é o fator que será usado para distinguir diferentes linhas no gráfico;
- `response` é a variável resposta que será grafada nas ordenadas (eixo-y);
- `fun` é a função da estatística a ser utilizada.

Vejamos a interação entre espécies e substrato no experimento do crescimento de mudas de espécies arbóreas:

```
> interaction.plot( mudas$substrato, mudas$especie, mudas$altura , col=c("
    red","blue"))
```

7.2.6 Exercícios

Exercício: Fatores que Influência a Altura em Florestas Plantadas III

Tomando a altura média das árvores dominantes (média de `'hdom` por `'parcela`) como variável resposta (dados de árvores de floresta plantada — dados:dados-egrandis), verifique **graficamente** a interação entre região (`'regiao`) e rotação (`'rot`).

Exercício: Variabilidade de Caixetais

Considere os dados do levantamento em três caixetais (dados:dados-caixeta). Pergunta-se: em qual dos caixetais (`local`), o diâmetro das árvores (`dap`) é mais variável entre as parcelas (`parcela`) quando se considera:

- diâmetro **médio**;
- diâmetro **mediano**;
- diâmetro **máximo**?

Responda com base numa análise gráfica.

7.3 Comparação de Modelos

7.3.1 Função anova: mais do que ANOVA

Um aspecto essencial à construção de modelos lineares é a comparação entre modelos.

A função `"anova"`, apesar do nome, é uma função muito utilizada para comparar modelos lineares que pertençam a uma certa *hierarquia* de modelos.

Vejamos o exemplo de construção de uma equação de biomassa com o seguinte forma:

$$b_i = \beta_0 + \beta_1 d_i + \beta_{d_i^2} + \beta_3 h_i + \beta_4 (d_i\, h_i) + \beta_5 (d_i^2\, h_i) + \beta_6 (d_i\, h_i^2) + \varepsilon_i$$

Embora esse seja um modelo muito problemático, ele serve para ilustrar o problema de seleção de modelos. Vejamos o que acontece utilizando os dados de árvores de *E. saligna* (dados:dados-esaligna):

```
> biom = lm( total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 * ht) + I(
    dap * ht^2), data=esa )
> summary(biom)

Call:
lm(formula = total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 *
    ht) + I(dap * ht^2), data = esa)
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-38.670  -7.688  -1.022   8.561  45.191

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)   -51.60332   65.40689  -0.789    0.437
dap             7.68140   11.24405   0.683    0.500
I(dap^2)        0.35420    0.53985   0.656    0.517
ht              7.48430    5.55867   1.346    0.189
I(dap * ht)    -1.42975    0.82821  -1.726    0.095 .
I(dap^2 * ht)   0.03763    0.03461   1.087    0.286
I(dap * ht^2)   0.01105    0.01593   0.694    0.493
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
           1

Residual standard error: 15.13 on 29 degrees of freedom
Multiple R-Squared: 0.9728,     Adjusted R-squared: 0.9672
F-statistic: 172.9 on 6 and 29 DF,  p-value: < 2.2e-16

>
```

Veja que nenhuma das variáveis preditoras se mostrou significativa (nível de probabilidade de 5%) para estimar a biomassa total das árvores. Mas esse resultado é razoável?

Vejamos o que a função **"anova"** nos mostra:

```
> anova(biom)
Analysis of Variance Table

Response: total
              Df Sum Sq Mean Sq  F value    Pr(>F)
dap            1 218121  218121 952.7442 < 2.2e-16 ***
I(dap^2)       1  17791   17791  77.7108 1.063e-09 ***
ht             1    352     352   1.5375   0.22493
I(dap * ht)    1    312     312   1.3648   0.25222
I(dap^2 * ht)  1    816     816   3.5633   0.06911 .
I(dap * ht^2)  1    110     110   0.4811   0.49343
Residuals     29   6639     229
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
           1
>
```

Nesse caso parece que o diâmetro e o diâmetro ao quadrado são significativos, mas os demais termos não. Por que os testes geram resultados diferentes?

As funções `summary` e `anova` realizam testes de forma distinta:

- O teste *t* da função `summary` testa cada variável preditora assumindo que **todas as demais variáveis já estão presentes no modelo**.

- O teste *F* da função `anova` testa as variáveis preditoras na seqüência apresentada no modelo, assumindo que as **variáveis anteriores** já estavam no modelo.

Desta forma, a função `anova` realiza teste de um modelo contra outro numa seqüência definida. O mesmo resultado se obtem partindo o modelo original numa série de modelos:

- **Modelo 0:**
  $b_i = \beta_0 + \varepsilon_i$
- **Modelo 1:**
  $b_i = \beta_0 + \beta_1 d_i + \varepsilon_i$
- **Modelo 2:**
  $b_i = \beta_0 + \beta_1 d_i + \beta_2 d_i^2 + \varepsilon_i$
- **Modelo 3:**
  $b_i = \beta_0 + \beta_1 d_i + \beta 2_{d_i^2} + \beta_3 h_i + \varepsilon_i$
- **Modelo 4:**
  $b_i = \beta_0 + \beta_1 d_i + \beta 2_{d_i^2} + \beta_3 h_i + \beta_4 (d_i * h_i) + \varepsilon_i$
- **Modelo 5:**
  $b_i = \beta_0 + \beta_1 d_i + \beta 2_{d_i^2} + \beta_3 h_i + \beta_4 (d_i * h_i) + \beta_5 (d_i^2 * h_i) + \varepsilon_i$
- **Modelo 6:**
  $b_i = \beta_0 + \beta_1 d_i + \beta 2_{d_i^2} + \beta_3 h_i + \beta_4 (d_i * h_i) + \beta_5 (d_i^2 * h_i) + \beta_6 (d_i \ * h_i^2) + \varepsilon_i$

Podemos ajustar esses modelos e utilizar a função `anova` para testá-los numa seqüência:

```
> m0 = lm( total ~ 1 , data=esa )
> m1 = lm( total ~ dap , data=esa )
> m2 = lm( total ~ dap + I(dap^2) , data=esa )
> m3 = lm( total ~ dap + I(dap^2) + ht , data=esa )
> m4 = lm( total ~ dap + I(dap^2) + ht + I(dap * ht), data=esa )
> m5 = lm( total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 * ht), data=
    esa )
> m6 = lm( total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 * ht) + I(
    dap * ht^2), data=esa )
>
>
> anova(m0, m1, m2, m3, m4, m5, m6)
Analysis of Variance Table

Model 1: total ~ 1
Model 2: total ~ dap
Model 3: total ~ dap + I(dap^2)
Model 4: total ~ dap + I(dap^2) + ht
Model 5: total ~ dap + I(dap^2) + ht + I(dap * ht)
```

```
Model 6: total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 * ht)
Model 7: total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 * ht) + I(dap
    *
     ht^2)
  Res.Df    RSS Df Sum of Sq        F    Pr(>F)
1     35 244142
2     34  26021  1    218121 952.7442 < 2.2e-16 ***
3     33   8230  1     17791  77.7108 1.063e-09 ***
4     32   7878  1       352   1.5375   0.22493
5     31   7565  1       312   1.3648   0.25222
6     30   6749  1       816   3.5633   0.06911 .
7     29   6639  1       110   0.4811   0.49343
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
            1
>
```

É importante lembrar que a função `anova` realiza o teste **na ordem que os modelos são apresentados**, e que isso pode ter forte influência nos resultados obtidos.

```
> anova( m0, lm(total ~ I(dap^2*ht),data=esa), lm( total ~ I(dap^2*ht) +
    dap, data=esa) )
Analysis of Variance Table

Model 1: total ~ 1
Model 2: total ~ I(dap^2 * ht)
Model 3: total ~ I(dap^2 * ht) + dap
  Res.Df    RSS Df Sum of Sq        F    Pr(>F)
1     35 244142
2     34  22160  1    221982 473.534 < 2.2e-16 ***
3     33  15470  1      6690  14.272 0.0006292 ***
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
            1
>
```

7.3.2 Exercícios

Exercício: Biomassa do Tronco de Árvores de E. saligna

Com base nos modelos apresentados acima, construa vários modelos para biomassa do tronco (`tronco`) de *E. salgina* (:dados:dados-esaligna).

Exercício: Modelo Polinomial

Construa um modelo polinomial (até quarto grau) da altura (`ht`) em função do diâmetro (`dap`) de árvores em caixetais (dados:dados-caixeta). Verifique os termos significativos.

7.3.3 Algumas Funções para Comparação de Modelos

Existem várias outras funções para auxiliar na construção e comparação de modelos.

As funções `add1` e `drop1` permitem adicionar ou retirar **um-a-um** os termos dos modelos lineares:

```
> add1( object = m1, scope = . ~ dap + ht + I(dap*ht) + I(dap^2*ht) , test=
    "F" )
Single term additions

Model:
total ~ dap
              Df Sum of Sq     RSS     AIC F value     Pr(F)
<none>                     26020.8   241.0
ht             1       1.0 26019.7   243.0  0.0013   0.97162
I(dap * ht)    1    2507.0 23513.8   239.3  3.5184   0.06956 .
I(dap^2 * ht)  1   10551.1 15469.6   224.3 22.5077 3.912e-05 ***
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1
>
```

```
> drop1( object = m6, scope = . ~ .,  test="F" )
Single term deletions

Model:
total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 * ht) + I(dap *
    ht^2)
              Df Sum of Sq    RSS    AIC F value   Pr(F)
<none>                     6639.3  201.8
dap            1     106.8 6746.1  200.4  0.4667 0.49993
I(dap^2)       1      98.6 6737.8  200.4  0.4305 0.51693
ht             1     415.0 7054.3  202.0  1.8128 0.18860
I(dap * ht)    1     682.3 7321.5  203.3  2.9801 0.09493 .
I(dap^2 * ht)  1     270.7 6910.0  201.3  1.1824 0.28582
I(dap * ht^2)  1     110.2 6749.4  200.4  0.4811 0.49343
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1
          1
>
>
> drop1( object = m6, scope = . ~ dap + ht,  test="F" )
Single term deletions

Model:
total ~ dap + I(dap^2) + ht + I(dap * ht) + I(dap^2 * ht) + I(dap *
    ht^2)
       Df Sum of Sq    RSS    AIC F value  Pr(F)
<none>              6639.3  201.8
dap     1     106.8 6746.1  200.4  0.4667 0.4999
ht      1     415.0 7054.3  202.0  1.8128 0.1886
>
```

Nessas duas funções, o ponto (".") na fórmula significa todos os termos do modelo. Ou seja, para um dado modelo, o `scope` igual a `". ~ . "` significa todos os termos da fórmula do modelo.

Outras funções úteis para construção e comparação de modelos são:

- `"step"` que realiza *regressão stepwise*; e
- `"AIC"` que calcula o *Akaike Information Criterion*.

```
> aic.tab = AIC( m0, m1, m2, m3, m4, m5, m6 )
> aic.tab$AIC.d = abs( c(0, diff(aic.tab$AIC)) )
> aic.tab
    df       AIC       AIC.d
m0   2 423.7551   0.0000000
m1   3 345.1563  78.5987781
m2   4 305.7148  39.4414506
m3   5 306.1411   0.4262982
m4   6 306.6842   0.5430302
m5   7 304.5765   2.1077173
m6   8 305.9841   1.4076291
>
```

7.3.4 Exercícios

Exercício: Biomassa do Tronco de Árvores de E. saligna II

Utilizando os modelos para biomassa do tronco (`tronco`) de *E. salgina* construidos no exercício acima, utilize as funções `drop1` e `add1` para analisar a importância dos termos individuais do modelo.

Exercício: Relação Altura-Diâmetro em Florestas Plantadas

Utilize a função `AIC` para analisar a importância das variáveis indicadoras nos modelos de relação altura-diâmetro ajustados para florestas de *E. grandis*.

- Tutorial
- Exercícios
- Apostila

8 Teste de significância

Em construção
— *Alexandre Adalardo de Oliveira 2014/03/17 10:16*

9 Reamostragem e Simulação

Links externos para o material referente a esse tópico no wiki da disciplina:

- Apostila
- Tutorial
- Exercícios

O R é uma ferramenta poderosa para simular situações e compará-las com padrões obeservados. O assunto é extenso, e aqui apresentaremos apenas exemplos simples das modalidades mais usadas dem ecologia. Para quem quiser se se aprofundar, um bom começo é Manly (1997 [15]).

9.1 A Função "sample"

Esta função reamostra os elementos de um vetor:

```
> meu.vetor
 [1] "A" "A" "B" "B" "C" "C" "D" "D" "E" "E"
> meu.vetor.bagunçado <- sample(meu.vetor)
> meu.vetor.bagunçado
 [1] "C" "A" "A" "D" "B" "E" "E" "C" "D" "B"
```

Por *default* a função retorna um novo vetor de mesmo comprimento, com os elementos permutados ao acaso. Portanto, é uma amostragem **sem reposição**. A função tem argumentos que definem se a amostragem é ou não com reposição, o tamanho do vetor resultante, entre outros. Consulte a página de ajuda.

9.1.1 Exercícios

Exercício: Quem Precisa da Caixa Econômica?

Você pode simular sorteios das loterias com a função sample. Consulte a ajuda da função para descobrir como simular um sorteio como o feito para o jogo da sena.

[15] Manly B. F. J., 1997 Randomization, bootstrap and Monte Carlo methods in biology. 2nd Ed., Chapman and Hall, London

9.2 Reamostragens sem Reposição: Permutações

Nesse procedimentos apenas "embaralhamos"os elementos de um ou mais vetores, o que podemos fazer gerando um vetor de mesmo tamanho, cujos elementos são amostrados ao acaso **sem reposição**. Esse procedimento é chamado de permutação ao acaso.

9.2.1 Testes de Permutação

Uma das aplicações simples das permutações ao acaso são testes de comparação de médias entre grupos, como o teste t, como mostramos abaixo.

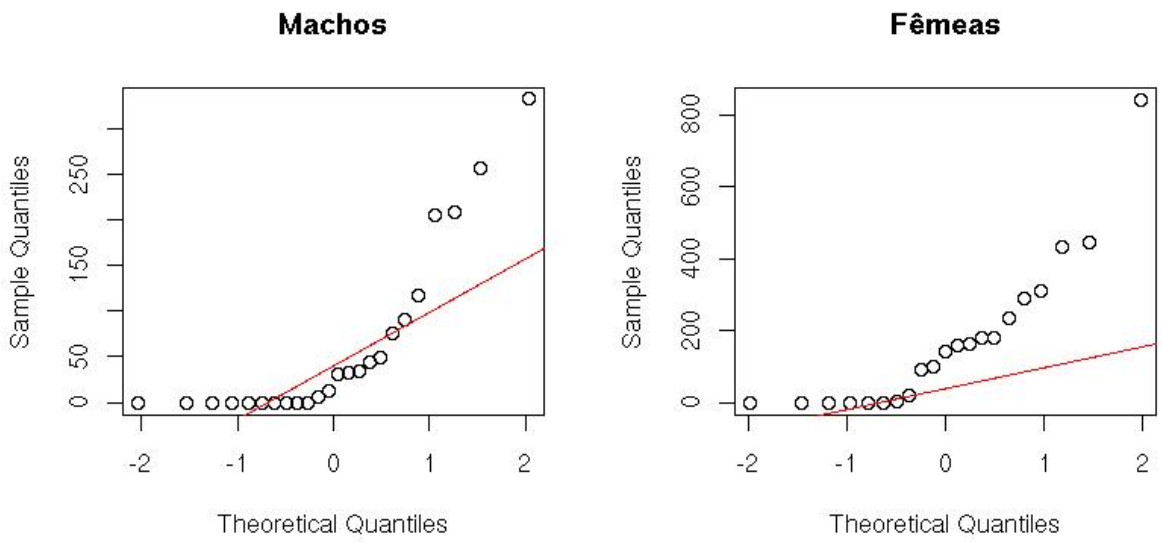
Os dados que usaremos são numero de besouros no conteúdo estomacal de lagartos *Phrynosoma brevirostrae* machos e femeas (Manly,1997):

```
> lagartos <- data.frame(sexo=factor(rep(c("m","f"),c(24,21))),
+ ncoleop=c(256,209,0,0,0,44,49,117,6,0,0,75,34,13,
+ 0,90,0,32,0,205,332,0,31,0,0,89,0,0,0,163,289,3,
+ 843,0,158,443,311,232,179,179,19,142,100,0,432))
```

Uma análise exploratória dos dados mostra que as médias amostrais são muito diferentes, mas as variâncias também, e a variação é muito grande, devido ao grande número de animais sem sinais de besouros no estômago:

```
> tapply(lagartos$ncoleop,lagartos$sexo,summary)
$f
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.0     0.0   142.0   170.6   232.0   843.0

$m
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    0.00   22.00   62.21   78.75  332.00

##Médias
> medias <- tapply(lagartos$ncoleop,lagartos$sexo,mean)
> medias
       f        m
170.57143  62.20833
##Diferença absoluta entre as médias
> dif.obs <- abs( diff( as.vector(medias) ) )
> dif.obs
[1] 108.3631

##Variâncias
> tapply(lagartos$ncoleop,lagartos$sexo,var)
       f        m
43533.557  8855.911
```

Os gráficos de quantis téoricos indicam um forte desvio em relação à distribuição normal:



Como os dados não atendem as premissas de um teste t, uma alternativa é substituí-lo por um teste de permutação. Como a hipótese nula é que as duas amostras vêm de populações com a mesma média, ela pode ser simulada permutando-se ao acaso os valores entre os sexos.

Calculamos então um índice de diferenças entre as médias das amostras que deve ser similar ao obtido, caso a hipótese nula esteja correta. Repetindo esse procedimento milhares de vezes, podemos estimar a chance de um valor igual ou maior que o observado ter ocorrido mesmo se as duas amostras vêm de populações com médias diferentes.

A permutação é feita com a função **sample**, que pode ser repetida com um *loop*, por meio da função **for**:

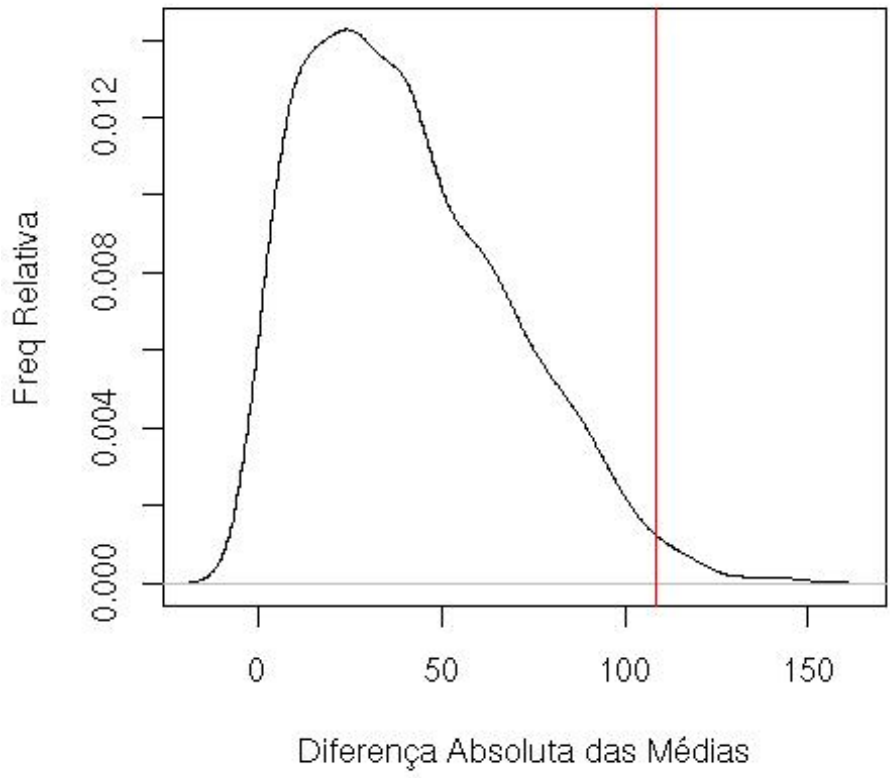
```
> ##cria um vetor para armazenar os resultados
> results <- c()
> ##Permuta os valores das medidas, calcula a diferença absoluta entre as m
    édias e
> ##armazena no vetor "results". Repete a operação mil vezes
> for(i in 1:1000){
+ results[i] <- abs( diff( tapply(sample(lagartos$ncoleop),lagartos$sexo,
    mean) ) )
+ }
```

Em apenas 15 das mil permutações a diferença absoluta das médias foi igual ou maior do que a observada:

```
> sum(results >= dif.obs)
[1] 15
```

Logo, a diferença observada é pouco provável sob a hipótese nula, o que nos permite rejeitá-la. Um gráfico de densidade probabilística dos valores das permutações ilustra isso:

```
> plot(density(results),xlab="Diferença Absoluta das Médias",ylab="Freq
    Relativa", main="")
> abline(v = dif.obs, col="red")
```



9.2.2 Modelos Nulos em Ecologia de Comunidades

Modelos nulos em ecologia de comunidades [16] muitas vezes envolvem permutação de elementos de uma matriz, em geral de ocorrência ou abundância de espécies por local.

Vamos imaginar uma dessas matrizes, com seis espécies (linhas) e seis locais (colunas):

```
> cc2
    [,1] [,2] [,3] [,4] [,5] [,6]
sp1    1    1    0    0    0    0
sp2    1    1    1    0    0    0
sp3    0    0    1    1    0    0
```

[16] ver GOTELLI, N. J. & G. R. GRAVES. 1996. Null models in ecology. Washington and London, Smithsonian Institution Press

```
sp4    0    0    1    1    1    0
sp5    0    0    0    0    1    1
sp6    0    0    0    0    1    1
> #Numero de especies por local
> S.obs <- apply(cc2,2,sum)
> S.obs
[1] 2 2 3 2 3 2
> ##Maximo de especies em coexistencia
> max(S.obs)
[1] 3
```

Podemos nos perguntar se há um limite ao número de espécies que podem coexistir. Se isso ocorre, o número máximo de espécies em co-ocorrência deve ser menor do que se distribuirmos as espécies ao acaso pelos locais. Há vários cenários nulos possíveis, mas ficaremos aqui com permutação dos locais em cada espécie ocorreu.

Para isso, basta permutar os valores dentro de cada linha:

```
> apply(cc2,1,sample)
     sp1 sp2 sp3 sp4 sp5 sp6
[1,]   1   0   0   1   0   1
[2,]   0   1   0   0   1   0
[3,]   0   1   1   1   0   0
[4,]   0   0   0   1   1   0
[5,]   0   1   0   0   0   1
[6,]   1   0   1   0   0   0
```

Ops! a função **apply** retorna sempre seus resultados em colunas, transpondo a matriz original. Para evitar isso, transpomos o resultado, voltando as espécies para as linhas:

```
> t(apply(cc2,1,sample))
    [,1] [,2] [,3] [,4] [,5] [,6]
sp1    0    1    0    0    1    0
sp2    1    1    0    0    1    0
sp3    0    1    0    1    0    0
sp4    0    1    0    0    1    1
sp5    1    0    0    0    0    1
sp6    1    0    1    0    0    0
```

Agora basta repetir essa permutação muitas vezes e calcular o número máximo de espécies em co-ocorrência. Para esse cálculo, podemos criar uma função:

```
> ##Uma funcao para calcular o maximo de especies em co-ocorrencia
> max.c <- function(x){ max(apply(x,2,sum)) }
> max.coc <- max.c(cc2)
> max.coc
[1] 3
```

E então repetimos as permutações 1000 vezes, inserindo-as em um *loop*:

```
##Um vetor para guardar os resultados
> results <- c()
```

```
## Mil simulações em um loop
> for(i in 1:1000){
+   result.c[i] <- max.c( t(apply(cc2,1,sample)))
+ }
```

O número de randomizações que teve um máximo de espécies por local menor ou igual ao observado foi alto (21,6%), portanto a observação de que no máximo três espécies ocorrem no mesmo local não é evidência de que haja um limite a esse número:

```
> sum(result.c<=max.coc)
[1] 216
```

9.3 Reamostragem com Reposição: Bootstrap

Como a maioria das boas idéias, o conceito geral do bootstrap é muito simples:

> Se aceitamos que nossa amostra é a melhor informação disponível que temos sobre uma população estatística, podemos simular uma nova coleta de dados reamostrando os elementos de nossa amostra, com reposição.

A aplicação mais simples deste princípio é estimar intervalos de confiança por reamostragem. Para exemplificar, usaremos o conjunto de dados BCI, um dataframe de abundâncias de espécies de árvores na parcela permanente de Barro Colorado nas (linhas), por subparcelas de 1 ha:

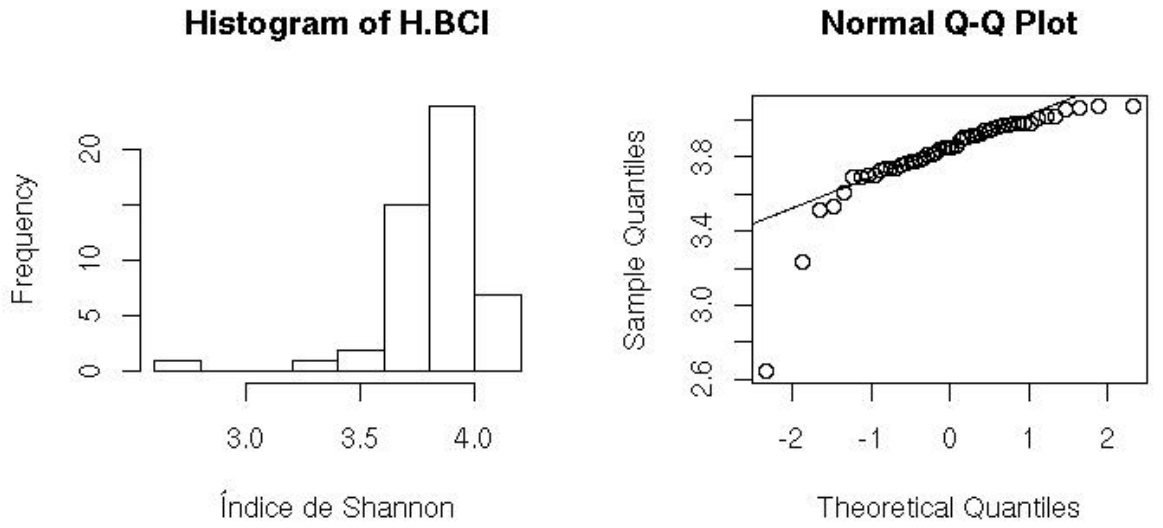
```
##Esse objeto de dados está no pacote "vegan"
> data(BCI, package="vegan")
##Visão de três linhas e três colunas
> BCI[1:3,25:28]
  Brosimum.guianense Calophyllum.longifolium Casearia.aculeata Casearia.
     arborea
1                  0                        0                 0
                   1
2                  0                        2                 0
                   1
3                  0                        0                 0
                   3
```

Vamos criar uma função para calcular o índice de Shannon a partir de um vetor de abundâncias de espécies, e aplicar a cada parcela:

```
> H <- function(x){
+   y <- x[x>0]
+   pi <- y/sum(y)
+   -sum(pi*log(pi))
+ }
> H.BCI <- apply(BCI,1,H)
```

O vetor H.BCI contém os índices de Shannon de cada parcela de 1 ha. Sua distribuição não parece ser normal:

```
> hist(H.BCI,xlab="  ndice   de  Shannon")
> qqnorm(H.BCI);qqline(H.BCI)
```



Vamos simular uma amostra deste conjuntos de parcelas, e estimar o intervalo de confiança da diversidade média por parcela:

```
> ##Uma amostra ao acaso de 20 parcelas
> bci.sample <- BCI[sample(1:nrow(BCI),20),]

> ##Estimativas da média e intervalo de confiança baseados na distribuição
    de t de Student:
> t.test(H.am)

        One Sample t-test

data:  H.am
t = 114.8506, df = 19, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 3.783456 3.923914
sample estimates:
mean of x
 3.853685
```

A função `t.test` quando aplicada a um vetor de números sem nenhum outro argumento retorna o teste t com a hipótese nula de que o vetor é uma amostra de uma população com distribuição normal e média zero. O intervalo de confiança da média da população de onde veio a amostra também é calculado.

No caso, esse intervalo não inclui o zero, o que rejeita a hipótese nula. Mas o que

interessa aqui é que o intervalo inclui a média da população de parcelas de onde veio a amostra, que é :

```
> mean(H.BCI)
[1] 3.82084
```

Se a amostra vem de uma população de valores com distribuição normal, o intervalo deve incluir a média da população em 95% das vezes que uma amostragem for realizada [17].

Como a premissa de normalidade parece não ser verdadeira, uma solução é calcular um intervalo por *bootstrap*. Para isso, basta re-amostrar com reposição a amostra, e calcular a estatística de interesse, no caso a média dos índices de Shannon de cada parcela:

```
##Um vetor para armazenar os resultados
> results <- c()
##Reamostrando 1000 vezes com reposição
> for(i in 1:1000){ results[i] <- mean( sample(H.am,replace=T) )}
```

A diferença entre a média da amostra original e a média das aleatorizações estima o viés, no caso muito pequeno:

```
> mean(results)
[1] 3.853739
> mean(H.am) - mean(results)
[1] -5.396726e-05
```

O intervalo de confiança por quantis a 95% está compreendida entre o valor abaixo do qual há apenas 2,5% das randomizações e o valor acima do qual restam apenas 2,5% das randomizações. Portanto, são os quantis a 2,5% e 97,5% dos mil valores simulados. Usamos a função **quantile** para obtê-los:

```
> quantile(results , probs=c(0.025,0.975))
    2.5%    97.5%
3.788261 3.912513
```

No caso, não houve diferença importante entre os intervalos de confiança estimados por *bootstrap* e pela distribuição de t de Student.

9.4 Simulações com Distribuições Teóricas

O R gera valores amostrados de distribuições teóricas de probabilidades (ver), o que pode ser usado em simulações de diversas maneiras.

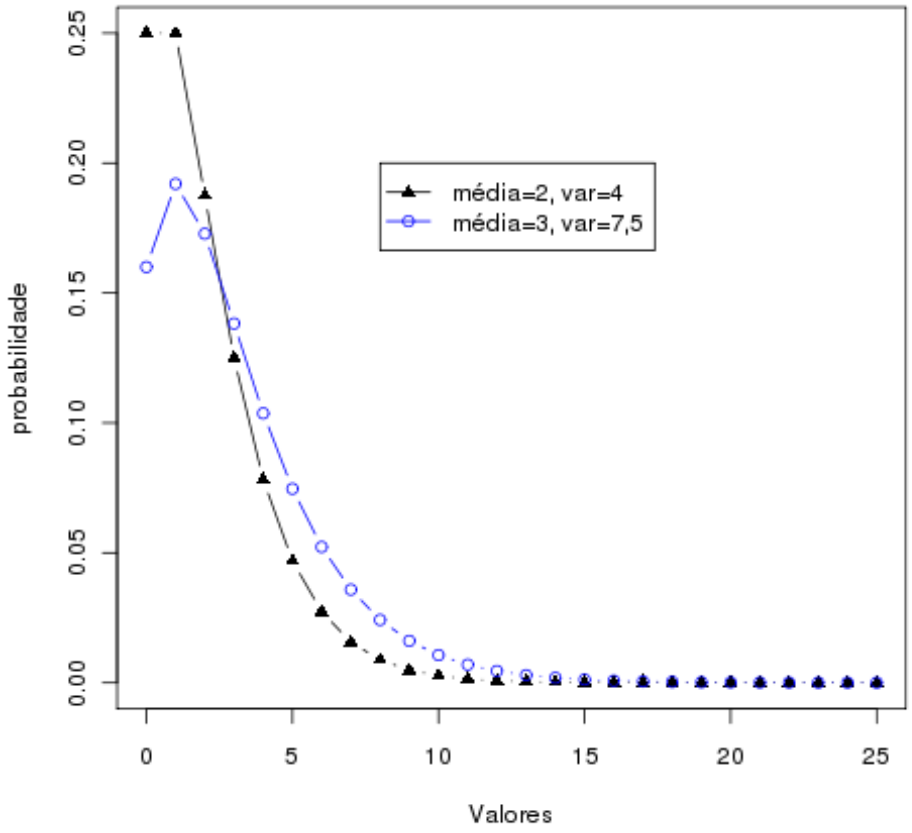
Um delas é avaliar a eficácia de um procedimento, como a força de um teste de significância em diferentes situações. O exemplo a seguir avalia a força de um teste

[17] Obs: é um erro comum pensar que o intervalo de confiança é fixo, e a média tem 95% de chance de estar dentro dele. Na verdade, a média populacional é fixa, pois é um parâmetro. O intervalo varia a cada vez que você repetir a amostragem, sendo uma *estimativa por intervalo*

t comparando duas amostras tomadas de duas populações com distribuição binomial negativa.

Para ilustrar a forma dessas distribuições, podemos fazer o gráfico dos valores teóricos de probabilidade, no intervalo de zero a 25:

```
> x <- 0:25
> y1 <- dnbinom(x,mu=2, size=2)
> y2 <- dnbinom(x,mu=3, size=2)
> plot(y1~x, type="b", xlab="Valores",ylab="probabilidade",
+      ylim=c(0,max(c(y1,y2))), pch=17 )
> points(y2~x, col="blue", type="b")
> legend(8,0.2,c("média=2, var=4","média=3, var=7,5"),lty=1,
+        col=c("black","blue"), pch=c(17,1))
```



As premissas do teste t são que as amostras vêm de populações com distribuições normais que têm variâncias iguais (pelo menos aproximadamente). A distribuição binomial negativa tem uma forma muito diferente da normal, e nela a variância é uma função da média. Assim, duas distribuições binomiais negativas com médias diferentes, como as ilustradas acima, terão variâncias diferentes. Além disso, a distribuição binomial negativa é **discreta**, enquanto a normal é **contínua**.

Em resumo, usar o teste t para comparar as médias de amostras tomadas de populações com distribuição binomial negativa implica em desconsiderar as premissas do teste, o que pode comprometer sua força. Mas como avaliar isso? Podemos simular amostras tomadas de populações com distribuição binomial negativa com médias diferentes, aplicar o teste t e contar o número de simulações em que o teste indicou uma diferença significativa entre as médias, a 5%.

Para simular a comparação de duas amostras de tamanho dez, usando um *loop* temos:

```
> ##Um vetor fator com 20 elementos, para representar cada elemento das
    duas amostras
> tratamento <- factor(x=rep(c("a","b"),each=10))
> ##Um vetor para armazenar os resultados da simulação
> results <- c()
> ##Dez mil simulações, com um loop e a função t.test para realizar o teste
    t
> for(i in 1:10000){
+   variaveis <- c(rnbinom(n=10,size=2,mu=2),rnbinom(n=10,size=2,mu=3))
+   results[i] <- t.test(variaveis~tratamento)$p.value
+ }
> ##Quantas das simulações indicaram uma diferença significativa das médias
    ?
> sum(results<0.05)
[1] 1148
```

O teste detectou corretamente que as amostras vêm de populações com médias diferentes em 1148 das dez mil simulações, o que estima a força do teste neste caso em = 0,1148 [18].

Vamos agora avaliar o que acontece se tomamos amostras de populações com distribuição normal, mantendo as mesmas variâncias:

```
> for(i in 1:10000){
+    variaveis <- c(rnorm(n=10,mean=2,sd=2),
+                   rnorm(n=10,mean=3,sd=sqrt(7.5)))
+    results[i] <- t.test(variaveis~tratamento)$p.value
+  }
> sum(results<0.05)
[1] 1389
```

Como poderíamos esperar, a força estimada foi maior, pois uma das premissas do teste agora é satisfeita. No entanto, a premissa de igualdade de variâncias ainda não é cumprida. Se simulamos amostras com variâncias iguais à média das duas variâncias usadas nas simulações anteriores, temos um ganho na força do teste, embora pequeno:

```
> ## Desvio-padrão médio correspondente  s  variâncias de 7,5 e 4
> sd1 <- mean(sqrt(c(7.5,4)))
> ## Verificando o valor
```

[18] Este valor irá variar a cada repetição da simulação, pois novos valores serão sorteados, mas ficará em torno do obtido. Para repetir uma simulação e obter os mesmos valores defina a mesma semente de números aleatórios com a função **set.seed**.

```
> sd1
[1] 2.369306
> for(i in 1:10000){
+     variaveis <- c(rnorm(n=10,mean=2,sd=sd1),
+                    rnorm(n=10,mean=3,sd=sd1))
+     results[i] <- t.test(variaveis~tratamento)$p.value
+   }
> sum(results<0.05)
[1] 1419
```

Mesmo nesta situação ideal com todas as premissas cumpridas, em apenas 14% das tentativas o teste detectará a diferença que existe entre as populações. Isto acontece pelo pequeno tamanho amostral. As simulações podem prosseguir para avaliar o efeito de diferentes tamanhos amostrais.

9.4.1 Exercícios

Exercício: Que simulação é esta?

Descubra que simulação é feita com código abaixo:

```
> m1 <- matrix(data=rnbinom(n=10000,size=2,mu=2),nrow=10,ncol=1000)
> m2 <- matrix(data=rnbinom(n=10000,size=2,mu=3),nrow=10,ncol=1000)
> matrizona <- rbind(m1,m2)
> tratamento <- factor(x=rep(c("a","b"),each=10))
> signif.t <- function(valores,fator){(t.test(formula=valores~fator))$
    p.value}
> sum(apply(X=matrizona,MARGIN=2,FUN=signif.t,fator=tratamento)<=0.05)
```

10 Noções de Programação

Links externos para o material referente a esse tópico no wiki da disciplina:

- Tutorial
- Exercícios
- Apostila

10.1 R: Um Ambiente Orientado a Objetos

10.1.1 Atributos

Até esse ponto do curso, foi visto que existem no R funções, variáveis e vetores. Todos esses ítens são chamados genericamente de **objetos**.

Veremos no decorrer do curso vários outros **objetos** do R. A importância do conceito de **objeto** num ambiente de trabalho de análise de dados é que os objetos possuem **atributos**, os quais podem variar em função do tipo de objeto.

Vejamos um exemplo.

```
> zoo
 onça  anta  tatu guará
    4    10     2    45
> class( zoo )
[1] "numeric"
> length( zoo )
[1] 4
> names( zoo )
[1] "onça"  "anta"  "tatu"  "guará"
>
```

O vetor `zoo` é um vetor de classe `numeric`, de comprimento (`length`) 4 e com nomes (`names`): onça, anta, tatu e guará. Classe, comprimento e nomes são os atributos típicos de vetores.

Qualquer vetor sempre terá uma classe e um comprimento, mas o atributo `names` é opcional:

```
> b
[1] 1 2 3 4 5 6 7 8
> class( b )
[1] "integer"
> length( b )
[1] 8
> names( b )
NULL
>
```

A função `attributes` nos mostra os atributos de um objeto, mas é de uso limitado no caso de vetores:

```
> zoo
 onça  anta  tatu guará
    4    10     2    45
> attributes( zoo )
$names
[1] "onça"  "anta"  "tatu"  "guará"

> b
[1] 1 2 3 4 5 6 7 8
> attributes( b )
```

```
NULL
>
```

10.1.2 Funções

As funções do R também são objetos, mas da classe `function`:

```
> class( ls )
[1] "function"
> class( log )
[1] "function"
> class( sin )
[1] "function"
>
```

No caso das funções, podemos associar a elas os **argumentos** que elas necessitam para serem executadas:

```
> args( ls )
function (name, pos = -1, envir = as.environment(pos), all.names = FALSE,
    pattern)
NULL
> args( log )
function (x, base = exp(1))
NULL
>
```

Algumas funções matemáticas, no entanto, tem sempre apenas um argumento e são consideradas **funções primitivas**:

```
> args( sin )
NULL
> sin
.Primitive("sin")
>
> args( exp )
NULL
> exp
.Primitive("exp")
>
```

10.1.3 Mundo dos Objetos

Um aspecto importante num ambiente orientado a objetos é que **tudo** o que o ambiente trabalha são objetos e o ambiente não pode trabalhar com nada que não seja um objeto conhecido. Inclui nessa categoria tudo aquilo que o R apresenta na tela, por isso toda saída do R pode ser guardada num objeto:

```
> length( zoo )
[1] 4
> zoo.comp = length( zoo )
> zoo.comp
[1] 4
> class( zoo )
[1] "numeric"
> zoo.class = class( zoo )
> zoo.class
[1] "numeric"
> class( zoo.class )
[1] "character"
> names( zoo )
[1] "onça"  "anta"  "tatu"  "guará"
> class( names( zoo ) )
[1] "character"
> length( names( zoo ) )
[1] 4
>
```

Quando o R nos mostra, como resultado de uma operação, valores como `NULL` e `integer(0)` ele está dizendo que o resultado é **vazio**, isto é, não há resultado:

```
> b
[1] 1 2 3 4 5 6 7 8
> names( b )
NULL
> b[ b > 10 ]
integer(0)
>
```

Veja que o valor `NULL` é um valor válidos que podem ser utilizados.

```
> zoo2 = zoo
> zoo2
 onça  anta  tatu guará
    4    10     2    45
> names( zoo2 )
[1] "onça"  "anta"  "tatu"  "guará"
> names( zoo2 ) = NULL
> zoo2
[1]  4 10  2 45
> names( zoo2 )
NULL
>
```

Exercício 7.1: Freqüência de Espécies Considere o vetor com nome de espécies:

```
> sp

[1] "Myrcia sulfiflora"      "Syagrus romanzoffianus" "Tabebuia
    cassinoides"
[4] "Myrcia sulfiflora"
```

Para obter a freqüência das espécies podemos usar a função `table`:

```
> table(sp)
sp
     Myrcia sulfiflora Syagrus romanzoffianus   Tabebuia cassinoides
                     2                       1                      1
>
```

Qual a classe do objeto que a função `table` retorna? Quais são os seus attributos?

Exercícios

Exercício 7.2: Classe da Classe

Qual a classe do objeto produzido pelo comando `class(x)`?

10.2 Construindo Funções Simples

10.2.1 A Estrutura Básica de uma Função

Toda manipulação de dados e análises gráficas e estatísticas no R são realizadas através de funções. Entretanto, você não precisa ser um programador experimentado para construir algumas funções simples para facilitar a atividade de manipulação de dados.

A estrutura básica de uma função é:

```
> minha.funcao <- function( argumento1, argumento2, argumento3, . . .)
                                    {

                                            comando 1

                                            comando 2
                                            comando 3
                                            . . .
                                            comando n
                                            return("resultado")
                                    }
```

Os elementos dessa expressão são:

- `minha.funcao` é o nome que a nova função receberá;
- `function` é a expressão no R que cria uma nova função;

- **entre as chaves "{}"** são listados os comandos da função, sempre com um comando por linha;
- **entre os parênteses "()"** são listados (separados por vírgula) os argumentos necessários a função;
- **comando return("resultado")** retorna os resultados, caso falte, será apresentado o resultado do último comando (comando n).

Vejamos alguns exemplos simples:

```
##criar um vetor de dados com 20 valores aleatórios de uma distribuição
    Poisson

dados.dens<-rpois(20,lambda=6)

##funcão para calcular média

media.curso <-function(x,rmNA=TRUE)
  {
  soma=sum(x)
  nobs=length(x)
  media=soma/nobs
  return(media)
  }

##Vamos agora preparar uma função mais elaborada, considerando a
##presença e excluíndo NA por padrão, e lançando mensagem na tela
##sobre o número de NAs removidos. Note que é uma função com dois
    argumentos
##que permite ao usuário tomar a decisão de remover ou não NAs e avisando,
##diferente da função mean()


media.curso <-function(x,rmNA=TRUE)
  {
  if(rmNA==TRUE)
    {
    dados=(na.omit(x))
    dif=length(x)-length(dados)
    cat("\t", dif," valores NA excluídos\n")
    }
    else
    {
    dados=x
    }
  soma=sum(dados)
  nobs=length(dados)
  media=soma/(nobs-1)
  return(media)
  }
```

```
###calcular a média do objeto dados
media.curso(dados.dens)


########################################
###função para calcular variância

var.curso<-function(x)
  {
  media=media.curso(x)
  dados=na.omit(x)
  disvquad=(dados-media)^2
  variancia=sum(disvquad)/(length(dados)-1)
  return(variancia)
  }

###Calcular a variância de dados
var.curso(dados.dens)

###Tomando dados.dens como a contagem de uma espécie em uma amostra de 20
    parcelas de 20x20m,
###podemos verificar o padrão de dispersão dessa espécie, utilizando o
     ndice  de Dispersão (razão variância / média)

ID.curso<-function(x)
  {
  id=var.curso(x)/media.curso(x)
  return(id)
  }

##Calcular o  coeficiente de dispersão

ID.curso(dados.dens)

## quando o valor é próximo a 1 a distribuição é considerada aleatória.
## podemos fazer um teste de significância  pela aproximação com o valor
    Qui-Quadrado
para verificar a significância dos dados

test.ID <- function(x)
        {
         dados=na.omit(x)
     med=media.curso(x)
     dev.quad=(dados-med)^2
     qui=sum(dev.quad)/med
         critico.qui<-qchisq(c(0.025,0.975),df=(length(dados)-1))
          if(critico.qui[1]<=qui & critico.qui[2]>=qui)
      { cat("\t distribuição aleatória para alfa=0.05\n")}
        else{}
    if(qui < critico.qui[1])
              { cat("\t","distribuição agregada, p<0.025 \n")}
      else{}
```

```
    if(qui>critico.qui[2])
    { cat("\t","distribuição regular, p>0.975 \n")}
     resulta=c(qui,critico.qui)
     names(resulta)<-c("qui-quadrado", "critico 0.025", "critico 0.975")
         return(resulta)
         }

############
```

10.2.2 Exercícios

Exercício 7.3: QUE FRIO!

Construa uma função que calcula automaticamente o valor de graus Celsius, sabendo-se a temperatura em Fahrenheit.
C = $\frac{5}{9}$ * (F(temperatura dada) - 32)

Exercício 7.4: Somatório do Primeiros Números Naturais

Construa uma função que calcula o somatório dos primeiros *n* números naturais. Por exemplo se *n=4* a função deve retornar o valor: *1+2+3+4*.

Exercício 7.5: Índices de Dispersão I

Existe uma série de índices de dispersão baseados em dados de contagem para verificar o padrão espacial de uma espécie.
Alguns deles são:

- **Razão Variância-Média**: ID = variância / média;
- **Coeficiente de Green**: IG = (ID-1)/(n-1);
- **Índice de Morisita**:

$$I_\delta = n \frac{\left(\sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i\right)}{\left(\sum_{i=1}^{n} x_i\right)^2 - \sum_{i=1}^{n} x_i}$$

onde:
n = tamanho da amostra; xi= número de indivíduos na i-ésima unidade amostral
Construa uma função para cada um desses índices, assumindo como argumento os valores de xi. Aplique aos dados de caixetais, verificando a dispersão da árvores de caixeta em cada caixetal.

10.2.3 Definindo Argumentos

Todos argumentos de uma função tem seu respectivo nome. Ao evocar a função podemos fazê-lo de duas formas:

- utilizando o **nome** dos argumentos **em qualquer ordem**;
- utilizando a **ordem** dos argumentos, mas **omitindo** os nomes.

```
> plot(col="red", pch=2, y=egr$ht, x=egr$dap)
> plot(egr$dap, egr$ht)
```

Para qualquer argumento podemos definir um **valor default** apresentando esse valor junto com argumento na definição da função:

```
> myplot <- function(..., col="red") { plot(..., col="red") }
> myplot(cax$dap, cax$h)
```

```
> myplot(ht ~  dap, data=egr)
```

O exemplo acima também mostra a função do argumento **". . ."**. Esse argumento representa **qualquer argumento adicional** que desconhecemos, mas que desejamos que seja passado para as funções dentro da função que estamos construindo.

10.2.4 Exercícios

Exercício 7.6: Gráfico de Whittaker

>Faça uma função para construir o gráfico de diversidade de espécies de Whittaker: logaritmo da abundância contra a ordem (descrescente) da abundância das espécies. Construa essa função de forma que qualquer parâmetro gráfico possa ser alterado.

10.3 Trabalhando com Funções mais Complexas

10.3.1 Um Aspecto Prático

Para saber qual é o editor padrão do R use o comando:

```
> getOption("editor")
[1] "vi"
>
```

Para alterar o editor padrão use o comando:

```
> options(editor= "gedit")       # Faz o editor "gedit" ser o editor padrão
    do R
```

No caso de editar sua função num editor externo ao R (p.ex., no arquivo `minhas-funcoes.R`), você traz o código para dentro do R utilizando o comando **"source"**:

```
> source("minhas-funcoes.R")
```

É importante que o arquivo editado externamente (`minhas-funcoes.R`) seja um arquivo **ASCII** sem qualquer símbolo especial.

10.3.2 Exercícios

Exercícios: Editando Funções Externamente

Experimente definir um editor com o qual você consiga trabalhar (`gedit`?) e refaça os exercícios anteriores salvando todos os códigos num arquivo externo.

Exercícios: Índices de Diversidade de Espécies

Construa funções para computar os seguintes índices de diversidade de espécies:

- Índice de Shannon: $H' = -\sum(p_i * \ln(p_i))$
- Índice de Simpson: $D = \sum({p_i}^2)$

onde p_i é a proporção da espécie i
Considere que o argumento de sua função será uma matriz com a abundância das espécies sendo as parcelas amostradas nas colunas. Considere a possibilidade de haver NA nessa matrix e a remoção dele.

10.3.3 Realizando Loops

Em linguagem de programação um **loop** é quando você força um programa a executar uma série de comandos repedidas vêzes.

A estrutura de loop no R é:

```
for( "variável"  in  "vetor de valores")
{
        comando 1
        comando 2
        comando 3
        . . .
        comando n
}
```

A palavra `for` é o chamado do loop. Dentro dos parênteses se define uma variável seguida da palavra `in` e um vetor de valores que a variável deverá assumir. Dentro das chaves se lista os comandos que devem ser repeditos a cada passo do loop.

Vejamos um exemplo: *Convergência da distribuição t de Student para distribuição Normal Padronizada*:

```
> #
> # Convergência da distribuição t de Student para distribuição Normal
    Padronizada
> #
> curve(dnorm(x), from=-4, to=4, col="red", lwd=6)
> for(gl in 1:200)
+ {
+       curve(dt(x, gl), -4, 4, add=TRUE, col="green")
+ }
>
```

No exemplo acima temos:

- `gl` é a variável definida para o loop;

- 1:200 é o vetor de valores que a variável assumirá, logo, o loop será repetido 200 vêzes.

10.3.4 Exercícios

Exercícios: Loop para Demonstrar o TCL

Construa uma função para demonstrar o Teorema Central do Limite.

10.3.5 Solução Vetorial x Loop

Sendo um **ambiente vetorial** os *loops* não são uma opção muito eficiente para computação dentro do R. Em geral, o R é mais eficiente se encontrarmos uma **solução vetorial** para problemas de computação que aparentemente exigem um loop. A solução vetorial, entretanto, costuma ser mais exigente em termos do tamanho de da memória RAM do computador.

Considere o problema o seguinte problema: temos a localização espacial de plantas num plano cartesiano com coordenadas **(x,y)**. Por exemplo:

```
> x = runif(100)
> y = runif(100)
> plot(x,y)
```

O objetivo é obter as **distâncias** entre as plantas duas-a-duas. Primeiro consideremos uma solução através de loop:

```
inter.edist = function(x, y)
{
  n = length(x)
  dist  <- c()
  for(i in 1:(n-1))
  {
    for(j in (i+1):n)
    {
      dist  <- c(dist, sqrt( (x[i] - x[j])^2 + (y[i] - y[j])^2 ))
    }
  }
  dist
}
```

Consideremos agora uma solução vetorial:

```
inter.edist.v = function(x, y)
{
  xd <- outer( x, x, "-" )
  yd <- outer( y, y, "-" )
  z <- sqrt( xd^2 + yd^2 )
  dist <- z[ row(z) > col(z) ]
  dist
```

```
}
```

Qual dessas soluções é mais eficiente em termos do uso do tempo?

```
> x = runif(100)
> y = runif(100)
>
> system.time( inter.edist( x, y ) )
[1] 0.140 0.008 0.149 0.000 0.000
>
> system.time( inter.edist.v( x, y ) )
[1] 0.008 0.000 0.009 0.000 0.000
```

Não tente rodar o exemplo acima com 1000 ou mais observações, pois o tempo fica **realmente longo** para versão em loop.

CONCLUSÃO:

Use apenas **pequenos loops** no R!!!

10.3.6 Exercícios

Exercícios: Tabela de Fitossociologia

Construa uma função que gera uma tabela de fitossociologia. Utilize os dados de caixeta (dados:dados-caixeta) como teste.

APÊNDICE: Tabela de Operadores do R

Outro aspecto formal importante da linguagem R é a ordem de prioridade de seus operadores. Além das regras de precedência usuais para as operações matemáticas, é essencial conhecer a prioridade dos outros operadores:

OPERADOR	DESCRIÇÃO	PRIORIDADE
$	seleção de componentes	ALTA
[[[	indexação	|
^	potência	|
–	menos unitário	|
:	operador de seqüência	|
%nome%	operadores especiais	|
/	multiplicação, divisão	|
< > <= >= == !=	comparação	|
!	não	|
& && | ||	e, ou	|
~	fórmula estatística	↓
«- <- -> =	atribuição	Baixa