Introdução ao uso do programa R

Victor Lemes Landeiro
Universidade Federal de Mato Grosso
Instituto de Biologia, Departamento de Botânica e Ecologia
vllandeiro@gmail.com

Fabricio B. Baccaro
Universidade Federal do Amazonas
Instituto de Ciências Biológicas, Departamento de Biologia
fbaccaro.ecolab@gmail.com

The R Project for Statistical Computing

Críticas e sugestões envie para Victor Landeiro, vllandeiro@gmail.com.

13 de março de 2015

Conteúdo

Introdução

O objetivo desta apostila é fazer uma breve introdução ao uso do programa R (R Development Core Team 2010). Seu intuito não é ensinar estatística. O formato como esta apostila foi escrita foi planejado para que a apostila seja usada durante disciplinas básicas de introdução ao R (principalmente para pessoas que nunca usaram o R) acompanhados de um professor e monitores. Porem isso não impede que você utilize a apostila sozinho em seus estudos. Ao longo desta apostila você verá que o programa R é extremamente poderoso e versátil e pode ser usado em praticamente todas as etapas da pesquisa científica (menos na mais importante: a definição de hipóteses e perguntas).

Na realidade o "programa R" é uma linguagem para manipulação de dados e análises estatísticas. Ele foi inspirado pela, e continua relativamente compatível com, a linguagem S desenvolvida pela empresa de telecomunicações AT&T. O Nome S, de *statistics*, foi uma alusão a outra linguagem desenvolvida pela mesma empresa: a famosa linguagem C. Posteriormente, a linguagem S foi vendida para uma empresa pequena que adicionou uma interface gráfica (GUI) e renomeou como S-Plus. No entanto, o R ficou mais popular que S ou S-Plus, provavelmente porque é gratuito e conta com um exército de pessoas contribuindo, ampliando e aprimorando seu funcionamento constantemente.

Como usar essa apostila

Como qualquer linguagem, o ideal para aprender a usar o R é "usá-lo!". Então, a melhor forma de se familiarizar com os comandos do R é ler um texto introdutório (como esta apostila) e ao mesmo tempo ir digitando os comandos no R e observando os resultados, gráficos, etc. Apenas ler esta apostila provavelmente não vai ajudar muito. ***Acompanhe-a fazendo os cálculos no R***. A presença de um professor ou monitor pode ajudar, mas sem treinamento diário o aprendizado se perde. Por isso, os resultados dos comandos não aparecem na apostila, você deve executar e conferir o resultado no R! As notas e explicações estão em letra arial e os comandos do R estão em letra `Courier New`. Portanto, use os comandos em `Courier New` para acompanhar a apostila no R. No R o sinal # (*hashtag*, jogo-da-velha) é usado para inserir comentários, é o mesmo que dizer: "a partir do # existem apenas comentários". O R não lê o que vem após o #. No decorrer desta apostila existem vários comentários após um sinal #, explicando o que foi feito, e encorajamos você a expandir esses comentários. Essa é uma excelente forma de aprender a linguagem. Você pode e deve fazer anotações que possam te ajudar a lembrar de um comando ou de uma análise no futuro.

Ler os manuais disponíveis na página do R como o "*An introduction to R*" que vem com o R e o "*Simple R*" de John Verzani [http://www.math.csi.cuny.edu/Statistics/R/simpleR/printable/simpleR.pdf], pode ser de grande ajuda no começo. Outro manual bem curto (49 páginas) e fácil de entender é o *The R Guide* de W. J. Owen disponível em http://www.mathcs.richmond.edu/~wowen/TheRGuide.pdf. Na página do R também existem vários manuais em português, caso não goste de ler em inglês. Além de manuais existem inúmeros fóruns sobre duvidas e tutoriais sobre o R. Um bom tutorial pode ser encontrado no site "https://www.zoology.ubc.ca/~schluter/R/" e um ótimo fórum para tirar dúvidas (depois que você realmente tentou resolvê-la) é página do R Brasil "http://leg.ufpr.br/doku.php/software:rbr".

Aprender a usar o R pode ser difícil e trabalhoso (igual a aprender um novo idioma ou a tocar um novo instrumento), mas lembre-se, o investimento será para você! John Chambers (2008, pp. v) escreveu no prefácio de seu livro: *"Será que é proveitoso gastar tempo para desenvolver e estender habilidades em programação? Sim, porque o investimento pode "contribuir" com sua habilidade em formular questões e na confiança que você terá nas respostas"*. Para os biólogos/ecólogos, veja também no site do ecólogo Nicholas J. Gotelli alguns conselhos para ser um bom pesquisador e para querer aprender a usar o R: http://www.uvm.edu/~ngotelli/GradAdvice.html.

Ao longo da apostila você precisará de alguns arquivos de dados que podem ser baixados no site https://sites.google.com/site/vllandeiror/. Os arquivos estão em formato .txt (p. ex. macac.txt)

Como Instalar o R

O R é um software livre para computação estatística e construção de gráficos que pode ser baixado e distribuído gratuitamente de acordo com a licença GNU. O R está disponível para as plataformas UNIX, Windows e MacOS. Para baixar e instalar o R para o Windows no seu computador:

1. Entre no site do R www.r-project.org
2. Clique em CRAN (Comprehensive R Archive Network)
3. Escolha o espelho de sua preferência (CRAN mirrors)
4. Clique em *Windows 95 or later*
5. Clique em *base* e salve o arquivo do R para Windows.
6. Execute o arquivo.

Se vc usa outro sistema operacional, basta escolher outro sistema no passo 3 e seguir as instruções.

A "cara" do R:

Depois de instalado basta clicar no ícone "R", recém criado. O resultado é uma mensagem dizendo a versão e outros dados do programa e uma tela branca com um *prompt* (sinal >; figura 1). O R possui uma janela com poucas opções para você se divertir clicando (veja figura abaixo). As análises feitas no R são digitadas diretamente na linha de comandos (i.e. você tem controle total sobre o que será feito). Na "linha de comandos" você irá digitar os comandos e funções que deseja.

O sinal **>** (sinal de maior) indica o *prompt* e quer dizer que o R está pronto para receber comandos. Em alguns casos um sinal de **+** aparecerá no lugar do prompt, isso indica que ficou faltando algo na linha de comandos anterior (isso acontece quando houve um erro, ou quando a finalização do comando só ocorrerá nas próximas linhas). **Se tiver errado pressione Esc para retornar ao prompt normal > e sumir com o sinal de +**. Note que na apostila, no começo de cada linha com os comandos do R há um sinal do prompt, >, e em alguns casos um sinal de **+**, não digite estes sinais. Os comandos que você digita aparecem em vermelho e o output do R aparece em azul. Após digitar os comandos tecle Enter para que eles sejam executados!

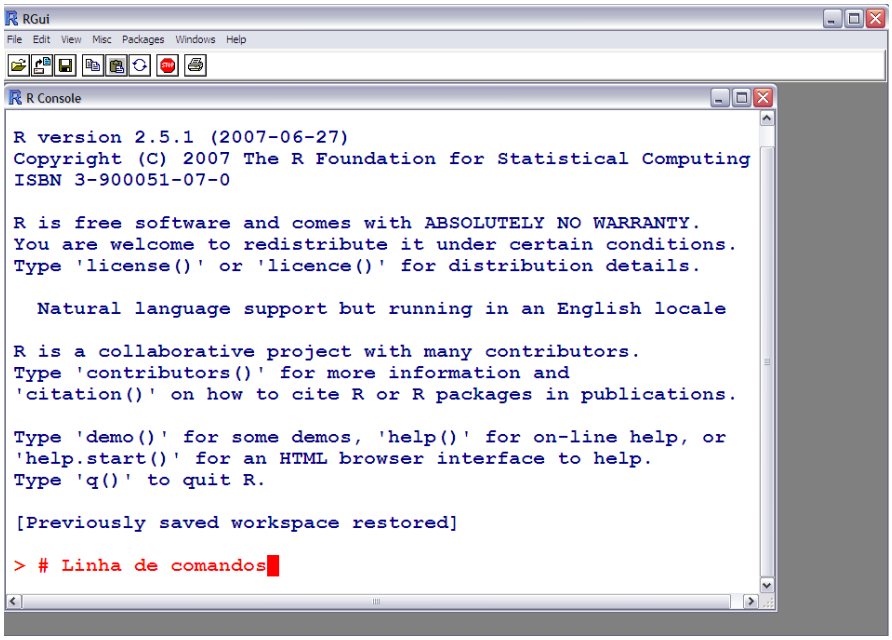


Figura 1 – A "cara" do R em um computador com Windows.

Noções gerais sobre o R

Para usar o R é necessário conhecer e digitar comandos. Alguns usuários acostumados com outros programas notarão de início a falta de "menus" (opções para clicar). Na medida em que utilizam o

programa, os usuários (ou boa parte deles) tendem a preferir o mecanismo de comandos, pois é mais flexível e com mais recursos. Algumas pessoas desenvolveram módulos de “clique-clique” para o R, como o R-commander. Porem, acreditamos que ao usar um módulo de “clique-clique” perdemos a chance de aprender uma das maiores potencialidades e virtudes do R, que é a programação. Clicando você não aprende a linguagem R! Nós iremos começar a apostila apenas usando comandos e funções já prontas no R. Conforme a familiaridade com a linguagem aumenta veremos, na parte final da apostila, como criar e escrever nossas próprias funções.

O R é *case-sensitive*, isto é, ele diferencia letras maiúsculas de minúsculas, portanto `A` é diferente de `a`. O separador de casas decimais é ponto `"."`. A vírgula é usada para separar argumentos (informações). Não é recomendado o uso de acentos em palavras. Acentos são comandos usados em programação e podem causar erros. De forma em geral, evite usar acentos em qualquer nome que for salvar em um computador, não só no R.

O *workspace* do R (área de trabalho).

A cada vez que você abre o R ele inicia uma "área de trabalho" (*workspace*). Neste *workspace* você fará suas análises, gráficos, etc. Ao final, tudo que foi feito durante uma sessão de uso do R pode ser mantido salvando o *workspace* (área de trabalho). Para salvar o *workspace* abra o R vá em "File" e clique em "Save workspace" e salve-o na pasta desejada (o R chama “pasta” de “diretório”), não é necessário nomear o arquivo.

O R é um programa simples e econômico. Ele assume que os arquivos que você quer usar estão na mesma pasta que ele está instalado. Hoje em dia, isso não é comum. Normalmente os programa estão instalados em pastas diferentes (“Arquivos de Programa”) dos nossos arquivos (“Meus Documentos”). Por isso, sempre que for usar o R em um trabalho faça como foi descrito acima. **Abra o programa e salve um *workspace* do R na pasta de trabalho em questão. Feche o R, e abra-o novamente usando o icone que apareceu na pasta.** Isso irá facilitar sua vida, pois não será necessário ficar alterando o diretório de trabalho. No caso do uso desta apostila, **salve um *workspace* na pasta que usará para seu estudo**. Esta pasta será o seu diretório de trabalho. Para facilitar sua vida copie todos os arquivos de dados necessaries para usar esta apostila na mesma pasta. Ao salvar o *workspace* irá aparecer um ícone do R na pasta, a partir do qual você irá abrir o R. Abra o R novamente, mas agora clicando no ícone que apareceu na pasta escolhida, faça os exercícios e no final apenas feche o R, irá aparecer uma caixa perguntando se deseja salvar o trabalho. Como o R tem acesso a todos os arquivos de uma pasta, evite salvar mais de um *workspace* na mesma pasta. Assim você evita que um *workspace* altere objetos de outro *workspace*.

Outra opção para não ter problema com o diretório de trabalho é alterar esta opção usando o Menu do R. Clique em File (Arquivo) e em Change Directory (Mudar diretório) e escolha o diretório onde seus dados estão presentes. No final, quando você salvar o trabalho o R irá salvar um Workspace no diretório escolhido. Abra o R a partir daquela pasta. Quando salvamos um workspace em uma pasta e o abrirmos a partir dela o R imediatamente reconhecerá o endereço da pasta como sendo seu diretório de trabalho. Para conferir o diretório de trabalho use o comando abaixo:

```
> getwd()  # este comando mostra qual diretório está em uso
```

Pacotes do R

O R é um programa leve (ocupa pouco espaço e memória) e geralmente roda rápido, até em computadores não muito bons. Isso porque ao instalarmos o R apenas as configurações mínimas para seu funcionamento básico são instaladas (pacotes que vem na instalação “*base*”). Para realizar tarefas mais complicadas pode ser necessário instalar pacotes adicionais (*packages*). Um pacote bastante utilizado em ecologia é o `vegan`, vegetation analysis, criado e mantido por Jari Oksanen (Oksanen et al. 2011). Para instalar um pacote vá ao site do R (**www.r-project.org**), clique em **CRAN**, escolha o espelho e clique em **packages**, uma lista com uma grande quantidade de pacotes está disponível. Clique no pacote que deseja e depois clique em **windows binary** e salve o arquivo. Abra o R e clique em *Packages*, depois em *Install Package(s) from local zip files* e selecione o arquivo do pacote que você baixou. Baixe e instale o pacote **vegan**.

O R também pode conectar-se diretamente à internet. Desta forma é possível, instalar e atualizar pacotes sem que seja necessário acessar a página do R. Dependendo da sua conexão com a internet é necessário

especificar as configurações de proxy da sua rede. Por exemplo, no Instituto Nacional de Pesquisas da Amazônia é necessário digitar a seguinte linha de comandos para poder acessar a internet com o R.

```
> Sys.setenv("HTTP_PROXY"="http://proxy.inpa.gov.br:3128")
```

Para instalar um pacote direto do R digite, por exemplo:

```
> install.packages("vegan")   ## É necessário estar conectado à internet!
```

Como usar um pacote do R

Não basta apenas instalar um pacote. Para usá-lo é necessário "carregar" o pacote sempre que você abrir o R e for usá-lo. Use a função `library` para rodar um pacote. Por exemplo: Digite `library(vegan)` na linha de comandos do R.

```
> library(vegan)
```

Após executar este comando as funcionalidades do `vegan` estarão prontas para serem usadas. Lembre-se que sempre que abrir o R será necessário carregar o pacote novamente e você pode carregar quantos pacotes quiser. O R oferece uma infinidade de pacotes para as mais diversas funções e análises. Você pode por exemplo, transformar seu R em um programa de geoprocessamento. Veja um exemplo no site: http://blog.revolutionanalytics.com/2012/02/creating-beautiful-maps-with-r.html.

Como citar o R, ou um pacote do R em publicações

No R existe um comando que mostra como citar o R ou um de seus pacotes. Veja como fazer:

```
> citation() # Mostra como citar o R
```

To cite R in publications use:

R Development Core Team (2011). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Veja que na parte com o nome dos autores aparece "R development core team", isso está correto, cite o R desta forma. Algumas pessoas não sabem disso e citam o R com autor Anônimo, isto tira o crédito do time.

Para citar um pacote, por exemplo o vegan, basta colocar o nome do pacote entre aspas.

```
> citation("vegan")
```

To cite package 'vegan' in publications use:
Jari Oksanen, F. Guillaume Blanchet, Roeland Kindt, Pierre Legendre, R. B. O'Hara, Gavin L. Simpson, Peter Solymos, M. Henry H. Stevens and Helene Wagner (2011). vegan: Community Ecology Package. R package version 1.17-6. http://CRAN.R-project.org/package=vegan

Usando o R pela primeira vez

O R como calculadora

A forma de uso mais básica do R é usá-lo como calculadora. Os operadores matemáticos básicos são: **+** para soma, **-** subtração, `*` multiplicação, `/` divisão e `^` exponenciação. Digite as seguintes operações na linha de comandos do R:

```
> 2+2
> 2*2
```

```
> 2/2
> 2-2
> 2^2
```

Use parênteses para separar partes dos cálculos, por exemplo, para fazer a conta 4+16, dividido por 4, elevado ao quadrado:

```
> ((4+16)/4)^2
```

Funções do R

O R tem diversas funções que podemos usar para fazer os cálculos desejados. O uso básico de uma função é escrever o nome da função e colocar os argumentos entre parênteses, por exemplo: `função(argumentos)`. `função` especifica qual função irá usar e `argumentos` especifica os argumentos que serão avaliados pela função. Não se assuste com esses nomes, com um pouco de pratica eles se tornarão triviais.

Antes de usar uma função precisamos aprender como usá-la. Para isso vamos aprender como abrir e usar os arquivos de ajuda do R.

Como acessar a ajuda do R (help)

Em diversos casos você irá querer fazer uma analise cujo nome da função você ainda não conhece. Nestes casos existem três formas básicas para descobrir uma função que faça aquilo que você deseja. A primeira é pesquisar dentro do R usando palavras chave usando a função `help.search()`. Por exemplo, vamos tentar descobrir como calcular logaritmos no R.

```
> help.search("logaritmo")
```

Veja que o R não encontrou nenhuma função, pois o R é desenvolvido em língua inglesa, portanto, precisamos buscar ajuda usando palavras em inglês.

```
> help.search("Logarithms")
```

Com esse comando o R irá procurar, dentro dos arquivos de help, possíveis funções para calcular logaritmos. Uma janela irá se abrir contendo possíveis funções.

Nas versões mais recentes do R a função `help.search()` pode ser substituída por apenas **??** (duas interrogações)

```
> ??logarithms
```

Também é possível buscar ajuda na internet, no site do R, com a função `RSiteSearch()`

```
> RSiteSearch("logarithms") # abre uma página na internet, mas só funcionará se seu computador estiver conectado à internet.
```

Para ver os arquivos de ajuda do R use o comando `help(nome.da.função)` ou `?nome.da.funcão`. Por exemplo, vamos ver o help da função log:

```
> help(log)        # abre o help sobre log
```

ou simplesmente

```
>?log
```

Geralmente, o arquivo de help do R possui 10 tópicos básicos:

1 - ***Description*** - faz um resumo geral sobre o **uso** da função

2 - ***Usage*** - mostra como a função deve ser utilizada e quais **argumentos** podem ser especificados

3 - ***Arguments*** - explica o que é cada um dos **argumentos**

4 - ***Details*** - explica alguns detalhes sobre o uso e aplicação da função (geralmente poucos)

5 - ***Value*** - mostra o que sai no output após usar a função (os resultados)

6 - ***Note*** - notas sobre a função

7 - ***Authors*** - lista os autores da função (quem escreveu os códigos em R)

8 - ***References*** - referências para os métodos usados

9 - ***See also*** - mostra outras funções relacionadas que podem ser consultadas

10 - ***Examples*** - exemplos do uso da função. Copie e cole os exemplos no R para ver como funciona

Quando for usar uma função pela primeira vez será lendo o help que você aprenderá a usá-la. Os tópicos ***Usage*** e ***Arguments*** são os mais importantes, pois mostram como os argumentos devem ser inseridos na função (*Usage*) e caso não saiba o que é algum desses argumentos existe uma explicação para cada um deles (*Arguments*).

Inicialmente, muitas pessoas têm dificuldade em entender o help do R e dizem que ele é ruim. Porém, com a prática os usuários percebem que help possui tudo que você precisa saber. Nada a mais nem nada a menos que o necessário. É **fundamental** aprender a usar o help para um bom uso do R.

Usando algumas funções

Por exemplo, use a função `sqrt` que é a função para calcular a raiz quadrada. Não se esqueça de olhar o help para aprender sobre os detalhes das funções. Além disso você já começa a se familiarizar com o arquivo de help.

```
> sqrt(9)          # Tira a raiz quadrada dos argumentos entre parênteses, no caso 9
> sqrt(3*3^2)      # raiz quadrada de 27
> sqrt((3*3)^2)    # raiz quadrada de 81
```

`prod` é a função para multiplicação

```
> prod(2,2)        # O mesmo que 2x2
> prod(2,2,3,4)    # 2x2x3x4
```

`log` é a função para calcular o logaritmo

```
> log(3)  # log natural de 3
> log(3,10)# log de 3 na base 10
> log10(3)# o mesmo que acima! log 3 na base 10
```

`abs` é a função para pegar os valores em módulo

```
> abs(3-9)         #  abs = modulo, |3-9|
```

Para fazer o fatorial de algum número use `factorial()`

```
> factorial(4)  #4 fatorial (4!)
```

Mais adiante veremos muitas outras funções e comandos do R.

Script do R

Usar o script do R para digitar os comandos

Uma maneira que otimiza o uso do R e que poupa muito tempo é usar um script (um arquivo .txt) para digitar seus comandos. Neste caso, os comandos não são digitados diretamente na linha de comandos do console (workspace) e sim em um editor de texto (por exemplo: R editor, notepad, tinn-R). O R editor já vem com o R e é bem simples de usar. Um script do R é apenas um arquivo .txt onde você digitará todos os comandos e análises.

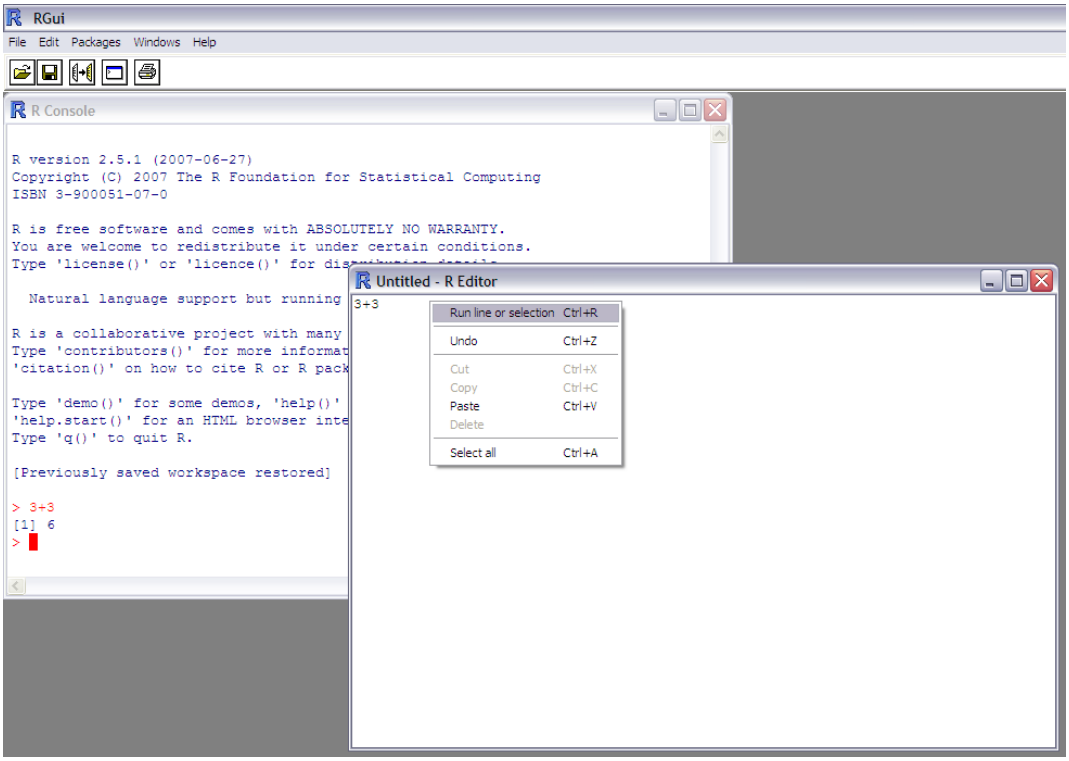


Figura 2. *Workspace* com editor ou *script*

Digite 3+3 no script e aperte Ctrl+R. O 3+3 será enviado para a linha de comandos do R e o resultado aparecerá no workspace. Para fazer outro comando aperte Enter e escreva o novo comando na linha de baixo. (cada comando deve ser digitado em uma linha diferente). No script você também pode inserir observações sobre o que foi feito, usando # para indicar a presença de um comentário. Por exemplo:

```
> x<-sample(1:10,20,replace=TRUE) # Comentários: Aqui eu criei um objeto chamado
```

x composto por 20 valores entre 1 e 10 amostrados ao acaso. O `replace=TRUE` indica que a amostragem foi feita com reposição.

```
> mean(x)          # Aqui eu uso a função mean para calcular a média de x.
```

Pode não parecer, mas o script é a ferramenta mais poderosa do R. O script é o histórico das manipulações de dados e análises que você fez. Em análises mais complexas, que normalmente demandam vários dias de trabalho, você pode recomeçar de onde parou ou fazer correções em suas análises e executa-las novamente com muita rapidez. Se você for organizado, incluirá comentários ao longo dos comandos que indicam em qual parte ou etapa das análises você está. As vantagens de ter um script bem organizado aparecem também durante o processo de publicação de artigos. Quando algum revisor de um de seus artigos solicitar uma mudança em uma de suas analises ou gráficos, basta abrir o script das análises, modificar o que for necessário e executar todo o script novamente. Pronto! Agora todas as modificações foram acrescentadas e os novos resultados ou gráficos estão disponíveis para resubmissão do trabalho. Para executar todo o script de uma só vez, basta clicar crtl+a e depois executar todos os comandos (crt+r). No final desta apostila, incluímos algumas dicas para você criar e manter scripts bem organizados.

Daqui em diante use o script para fazer os cálculos e exercícios desta apostila. Para criar um script clique em *File* no menu do R e clique em New script. Uma janela será aberta (Figura 2). O script aparece com o nome de "R Editor" (editor do R) e a extensão do arquivo é **.R**. Ou seja, para salvar o seu script, após digitar seus comandos, dê um nome ao arquivo e inclua a extensão **.R** no final do nome (p. ex. analises.R).

Para quem gosta de botões...

Hoje em dia a maioria dos programa oferecem ícones para você clicar, mas para usar o R é necessário conhecer e digitar comandos. A princípio essa característica pode parecer um problema, mas o mecanismo

de comandos é mais flexível e com mais recursos. Se você não consegue viver sem um monte de botões, você pode usar um dos IDEs (*integrated development environment*) que foram desenvolvidos para o R. Esses IDEs são gratuitos e precisam do R instalado para funcionar. Os mais conhecidos são:

- O RStudio, que pode ser baixado em: http://rstudio.org/ (tem a vantagem de ser multiplataforma, apresentando a mesma interface gráfica no Windows, Linux e Mac)
- StatET, http://www.walware.de/goto/statet
- ESS (Emacs Speaks Statistics), http://ess.r-project.org
- R Commander: John Fox (2005), "The R Commander: A basic-statistics graphical interface to R", Journal of Statistical Software 14, 1:42.
- O Tinn R pode ser baixado em: http://www.sciviews.org/Tinn-R/

No entanto, ao usar um IDE estilo "clique-clique" você perde a chance de aprender uma das maiores potencialidades e virtudes desse programa, que é a linguagem R! Nessa apostila, usaremos somente linhas de comando, nada de "clique-clique". No início usaremos apenas comandos e funções já prontas no R. Conforme a familiaridade com a linguagem aumenta veremos, na parte final da apostila, como criar e escrever nossas próprias funções.

Introdução a algumas estruturas de dados ou objetos do R

As informações ou dados no R podem ter diferentes estruturas, e existem muitos tipos de estruturas de dados que só passamos a conhecê-los bem com o passar do tempo (ou mais comumente, quando precisamos). O legal do R, e que diferencia da maioria dos outros programas de estatística, é que todos os tipos de dados são tratados da mesma forma. Ou seja, são todos chamados de objeto. Linguagens de programação orientadas a objetos (que é o caso da linguagem R) são extremamente poderosas e estão dominando o mundo virtual. Com certeza seu telefone e seu computador utiliza linguagens desse tipo. Provavelmente a mais famosa é a linguagem Java (www.java.org). Uma das grandes vantagens das linguagens orientadas a objetos é sua flexibilidade e mostraremos para você nas próximas seções. Por enquanto vamos conhecer os tipos básicos de objetos, ou de estruturas de dados:

a) **vetores:** uma sequência de valores numéricos ou de caracteres (letras, palavras).

b) **matrizes:** coleção de vetores em linhas e colunas, todos os vetores dever ser do mesmo tipo (numérico ou de caracteres).

c) **dataframe:** O mesmo que uma matriz, mas aceita vetores de tipos diferentes (numérico e caracteres). Geralmente nós guardamos nossos dados em objetos do tipo data frame, pois sempre temos variáveis numéricas e variáveis categóricas (por exemplo, largura do rio e nome do rio, respectivamente).

d**) listas:** conjunto de vetores, dataframes ou de matrizes. Não precisam ter o mesmo comprimento, é a forma que a maioria das funções retorna os resultados.

e) **funções:** as funções criadas para fazer diversos cálculos também são objetos do R.

No decorrer da apostila você verá exemplos de cada um destes objetos. As funções não são dados. Na realidade as funções são objetos que executam determinado conjunto de operações matemáticas. Ao usar o R, tanto as funções como os dados são tratados como objetos. Você vai entender melhor quais são as vantagens de tratar as funções como objetos mais adiante. Para uma leitura mais aprofundada sobre os tipos de objetos e definições da linguagem R leia o manual "R language definition" e o "An introduction to R" páginas 7 a 12, ambos disponíveis no menu "HELP", "Manuais em PDF". Clique em *help* no menu do R e em *Manuais* (em PDF) para abrir os arquivos.

Demonstrações

Algumas funções do R possuem demonstrações de uso. Estas demonstrações podem ser vistas usando a função `demo()`. Vamos ver algumas demonstrações de gráficos que podem ser feitos no R. Digite o seguinte na linha de comandos:

```
> demo(graphics) # Vai aparecer uma mensagem pedindo que você tecle Enter para
```

prosseguir, depois **clique na janela do gráfico** para ir passando os exemplos.

```
> demo(persp)
```

```
> demo(image)
```

Como criar objetos

Objetos vetores com valores numéricos

Vamos criar um conjunto de dados que contém o número de espécies de aves (riqueza) coletadas em 10 locais. As riquezas são 22, 28, 37, 34, 13, 24, 39, 5, 33, 32.

```
> aves<-c(22,28,37,34,13,24,39,5,33,32)
```

O comando `<-` (sinal de menor e sinal de menos) significa assinalar (*assign*). Indica que tudo que vem após este comando será salvo com o nome que vem antes. É o mesmo que dizer "salve os dados a seguir com o nome de **aves**". A letra `c` significa concatenar (colocar junto). Entenda como "agrupe os dados entre parênteses dentro do objeto que será criado" neste caso no objeto `aves`. Para ver os valores (o conteúdo de um objeto), basta digitar o nome do objeto na linha de comandos.

```
> aves
```

A função `length` fornece o número de observações (n) dentro do objeto.

```
> length(aves)
```

Objetos vetores com caracteres (letras, variáveis categóricas).

Também podemos criar objetos que contêm letras ou palavras ao invés de números. Porem, as letras ou palavras devem vir entre aspas `" "`.

```
> letras<-c("a","b","c","da","edw")
> letras
> palavras<-c("Manaus","Cuiabá","Belém","Brasília")
> palavras
```

Crie um objeto "misto", com letras e com números. Funcionou? Esses números realmente são números? Note a presença de aspas, isso indica que os números foram **convertidos em caracteres**. Evite criar vetores "mistos", a menos que tenha certeza do que está fazendo.

Operações com vetores

Podemos fazer diversas operações usando o objeto `aves,` criado acima.

```
> max(aves)       #valor máximo contido no objeto aves
> min(aves)       #valor mínimo
> sum(aves)       #Soma dos valores de aves
> aves^2          #...
> aves/10
```

Agora vamos usar o que já sabemos para calcular a média dos dados das aves.

```
> n.aves<-length(aves)      # número de observações (n)
> media.aves<-sum(aves)/n.aves        #média
```

Para ver os resultados basta digitar o nome dos objetos que você salvou

```
> n.aves    # para ver o número de observações (n)
> media.aves      # para ver a média
```

Você não ficará surpreso em saber que o R já tem uma função pronta para calcular a média.

```
> mean(aves)
```

Acessar valores dentro de um objeto [colchetes]

Caso queira acessar apenas um valor do conjunto de dados use colchetes `[].` Isto é possível porque o R salva os objetos como vetores, ou seja, a sequencia na qual você incluiu os dados é preservada. Os programadores chamam essa sequencia de índices e você pode usar esses índices para selecionar valores em qualquer objeto. Por exemplo, vamos acessar o quinto valor do objeto `aves.`

```
> aves[5] # Qual o quinto valor de aves?
> palavras[3] # Qual a terceira palavra?
```

O R não mostra os índices quando você executa um objeto para simplificar, mas os índices estão lá. No exemplo de aves acima, o que o R faz pode ser visualizado assim:

```
     [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]   22  28  37  34  13  24  39   5  33   32
```

Como o vetor `aves` só tem uma dimensão, quando você digita `aves[5]`, o R retorna o valor `13` (quinto elemento do vetor `aves`).

Para acessar mais de um valor use c para concatenar dentro dos colchetes `[c(1,3,...)]`:

```
> aves[c(5,8,10)]     # acessa o quinto, oitavo e décimo valores
```

Para excluir um valor, ex: o primeiro, use:

```
> aves[-1]            # note que o valor 22, o primeiro do objeto aves, foi excluído
```

Caso tenha digitado um valor errado e queira corrigir o valor, especifique a posição do valor e o novo valor. Por exemplo, o primeiro valor de aves é 22, caso estivesse errado, ex: deveria ser 100, basta alterarmos o valor da seguinte maneira.

```
> aves[1]<-100 # O primeiro valor de aves deve ser 100
> aves
> aves[1]<-22 # Vamos voltar ao valor antigo
```

Transformar dados

Em alguns casos é necessário, ou recomendado, que você transforme seus dados antes de fazer suas análises. Transformações comumente utilizadas são log e raiz quadrada.

```
> sqrt(aves)      #Raiz quadrada dos valores de aves
> log10(aves)     #log(aves) na base 10, apenas
> log(aves)       # logaritmo natural de aves
```

Para salvar os dados transformados dê um nome ao resultado. Por exemplo:

```
> aves.log<-log10(aves) # salva um objeto com os valores de aves em log
```

Listar e remover objetos salvos

Para listar os objetos que já foram salvos use `ls()` que significa listar.

```
> ls()
```

Para remover objetos use `rm()` para remover o que está entre parênteses.

```
> rm(aves.log)
```

```
> aves.log # você verá a mensagem:
Error: object "aves.log" not found
```

Para apagar todos os objetos do workspace use: `rm(list=ls())`. **Mas atenção, não execute esse comando agora!!!**

Gerar sequências (usando : ou usando seq)

: (dois pontos)

Dois pontos " **:** " é usado para gerar sequências de um em um, por exemplo a sequência de 1 a 10:

```
> 1:10      # O comando : é usado para especificar sequências.
> 5:16      # Aqui a sequência vai de 5 a 16
```

seq

A função `seq` é usada para gerar sequências especificando os intervalos.

Vamos criar uma sequência de 1 a 10 pegando valores de 2 em 2.

```
> seq(from = 1, to = 10, by = 2)    #seq é a função para gerar sequências.
```

Como todas as funções no R, não é necessário especificar os nomes dos argumentos, basta somente digitar os valores:

```
> seq(1,10,2)    #compare o resultado com o comando anterior

> seq(10,1,2)    #e agora? Porque não deu certo?
```

Sempre que você executa uma função sem declarar os nomes dos argumentos (neste caso, “from”, “to” e “by”), o R assume que você está seguindo a mesma ordem em que a função foi criada. Para saber qual é a ordem, basta olhar no help. Por isso é impossível ir de 10 para 1 acrescentando 2! Para esse comando funcionar você precisa fazer algumas modificações:

```
> seq(to=10,from=1,by=2)  #agora sim, mas veja que não é muito intuitivo...

> seq(1,100,5) #sequência de 1 a 100 em intervalos de 5

> seq(1,0.01,-0.02)  #sequência de 1 a 0.01 em intervalos de 0.02
```

A vantagem de não precisar informar os nomes dos atributos é óbvia: simplificar e diminuir o tamanho dos comandos. No entanto, você precisa prestar atenção, principalmente quando está começando a mexer no R. A maioria das funções tem valores padronizados (default) ou nulos em seus atributos. Veja o help da função `seq()`:

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

Veja que tanto o atributo “from” e “to”, tem o valor 1 como padrão. Ou seja, se você não especificar qual valor deseja, o R vai usar o valor 1. Apesar de parecer um pouco complicado a primeira vista, o que está escrito depois do “by” também resulta no default 1. Ou seja, o valor default do atributo “by” também é 1. Compare os resultados abaixo:

```
> seq(to=10,by=2) #....
> seq(to=10) #....
```

Veja também que existem outros argumentos, em que o default é “NULL” (nulo). Nesses casos, os argumentos só serão usados, se você informar o valor. Volte no help da função `seq` e veja o que é e como

funciona o argumento `length.out`, e use esse argumento para criar uma sequência de 10 números pares entre 2 e 20.

Gerar repetições (rep)

rep

Vamos usar a função rep para repetir algo n vezes.

```
> rep(5,10)            # repete o valor 5 dez vezes
```

A função `rep` funciona assim:

`rep(x, times=y)` # rep(repita x, y vezes) # onde x é o valor ou conjunto de valores que deve ser repetido, e times é o número de vezes)

```
> rep(2,5)
> rep("a",5)           # repete a letra "a" 5 vezes
> rep(1:4,2)           # repete a sequência de 1 a 4 duas vezes
> rep(1:4,each=2)      # note a diferença ao usar o comando each=2
> rep(c("A","B"),5)    # repete A e B cinco vezes.
> rep(c("A","B"),each=5)  # repete A e B cinco vezes.
> rep(c("Três","Dois","Sete","Quatro"),c(3,2,7,4))  # Veja que neste
```

caso a primeira parte do comando indica as palavras que devem ser repetidas e a segunda parte indica quantas vezes cada palavra deve ser repetida. Veja o help dessa função para conhecer quais argumentos existem e os valor padrão de cada argumento.

Gerar dados aleatórios

runif (Gerar dados aleatórios com distribuição uniforme)

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

começando em min e terminando em max.

```
> runif(200,80,100)          # gera 200 valores que vão de um mínimo de 80 até um
```

máximo de 100

```
> temp<-runif(200,80,100)
> hist(temp)                 # Faz um histograma de frequências dos valores
```

rnorm (Gerar dados aleatórios com distribuição normal)

```
> rnorm(n, mean=0, sd=1)     # gera n valores com distribuição uniforme, com
```

média 0 e desvio padrão 1.

```
> rnorm(200,0,1)             # 200 valores com média 0 e desvio padrão 1
> temp2<-rnorm(200,8,10)     # 200 valores com média 8 e desvio padrão 10
> hist(temp2)                # Faz um histograma de frequências dos valores
```

Veja o help da função `?Distributions` para conhecer outras formar de gerar dados aleatórios com diferentes distribuições.

Fazer amostras aleatórias

A função `sample`

A função `sample` é utilizada para realizar amostras aleatórias e funciona assim:

`sample(x, size=1, replace = FALSE)` # onde `x` é o conjunto de dados do qual as amostras serão retiradas, `size` é o número de amostras e `replace` é onde você indica se a amostra deve ser feita com reposição (`TRUE`) ou sem reposição (`FALSE`).

```
> sample(1:10,5)  # tira 5 amostras com valores entre 1 e 10
```

Como não especificamos o argumento `replace` o padrão é considerar que a amostra é sem reposição (= `FALSE`).

```
> sample(1:10,15) # erro, amostra maior que o conjunto de valores, temos 10 valores (1 a 10) portanto não é possível retirar 15 valores sem repetir nenhum!
> sample(1:10,15,replace=TRUE)  # agora sim!
```

Com a função `sample` nós podemos criar vários processos de amostragem aleatória. Por exemplo, vamos criar uma moeda e "jogá-la" para ver quantas caras e quantas coroas saem em 10 jogadas.

```
> moeda<-c("CARA","COROA")  # primeiro criamos a moeda
> sample(moeda,10) #ops! Esqueci de colocar replace=TRUE
> sample(moeda,10,replace=TRUE) # agora sim ☺
```

Ordenar e atribuir postos (*ranks)* aos dados

funções: sort, order e rank

Primeiro vamos criar um vetor desordenado para servir de exemplo:

```
> exemplo<-sample(1:100,10)  # amostra ao acaso 10 valores entre 1 e 100
> exemplo  # veja que os valores não estão em ordem. Talvez com muita sorte os seus valores estejam.
 [1] 94 27 89 82 24 51 2 54 37 38 # seus valores serão diferentes
```

sort

A função `sort` coloca os valores de um objeto em ordem crescente ou em ordem decrescente.

```
> sort(exemplo)       # para colocar em ordem crescente
 [1] 2 24 27 37 38 51 54 82 89 94
> sort(exemplo, decreasing=TRUE) # para colocar em ordem decrescente
 [1] 94 89 82 54 51 38 37 27 24 2
```

order

A função `order` retorna a posição original de cada valor do objeto "`exemplo`" caso os valores do objeto "`exemplo`" sejam colocados em ordem.

```
> order(exemplo)     #
 [1]  7  5  2  9 10  6  8  4  3  1
```

Nesta função o R retorna os valores dos índices do objeto. Veja novamente:

```
     [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]   94  27  89  82  24  51   2  54  37   38
```

Note que o primeiro valor acima é 7, isso indica que se quisermos colocar o objeto "`exemplo`" em ordem crescente o primeiro valor deverá ser o sétimo valor do "`exemplo`", que é o valor 2 (o menor deles). Na sequência devemos colocar o quinto valor do objeto "`exemplo`", que é 24, depois o segundo, depois o nono… até que objeto "`exemplo`" fique em ordem crescente.

Você pode descobrir qual a ordem decrescente facilmente:

```
> order(exemplo,decreasing=TRUE)
[1]  1  3  4  8  6 10  9  2  5  7
```

Vamos ver outro exemplo.

```
> valores<-sample(seq(1,100,10))
> valores
> order(valores)
```

Agora vamos usar a informação da ordem e colocar os valores em ordem crescente e decrescente.

```
> valores[order(valores)]
> valores[order(valores,decreasing=TRUE)]
```

É importante entender o comando `order`, pois ele é muito usado para colocar uma planilha de dados seguindo a ordem de alguma de suas variáveis. Essa técnica é muito usada em ecologia de comunidades. Veremos como fazer isso mais adiante.

rank

A função `rank` atribui postos aos valores de um objeto.

```
> exemplo  ## apenas para relembrar os valores do exemplo
[1] 94 27 89 82 24 51 2 54 37 38
```

Agora vamos ver o rank destes valores

```
> rank(exemplo)        # Para atribuir postos (ranks) aos valores do exemplo
[1] 10 3 9 8 2 6 1 7 4 5
```

Veja que 94 é o maior valor do exemplo, portanto recebe o maior rank, no caso 10.

Relembrando o que é o workspace

O workspace é a área de trabalho do R, quando salvamos um workspace em uma pasta e o abrirmos a partir dela o R imediatamente reconhecerá o endereço da pasta como sendo seu diretório de trabalho. Para conferir o diretório de trabalho use o comando abaixo:

```
> getwd()
> [1] "C:/Documents and Settings/Victor Landeiro/My Documents"
```

Caso o diretório que aparecer não for o da pasta que está usando para a apostila salve seu workspace na pasta da apostila. Para isso clique em File (Arquivo) e depois em “Save workspace” (“Salvar área de trabalho”), escolha a pasta que quer e salve. **OBS: 1) não é necessário dar nome ao arquivo 2) não salve mais de um workspace na mesma pasta.**

Exercícios com operações básicas

1- Suponha que você marcou o tempo que leva para chegar a cada uma de suas parcelas no campo. Os tempos em minutos foram: 18, 14, 14, 15, 14, 34, 16, 17, 21, 26. Passe estes valores para o R, chame o objeto de *tempo*. Usando funções do R ache o tempo máximo, mínimo e o tempo médio que você gasta para chegar em suas parcelas.

2. Use as funções `union, intersect e setdiff` para encontrar a união, o intersecto e as diferenças entre os conjuntos A e B. Aprenda no help como utilizar estas funções.

A 1 2 3 4 5

B 4 5 6 7

3. Calcule $|2^3-3^2|$. Módulo de 2^3-3^2

4. Suponha que você coletou 100 amostras em duas reservas, as 50 primeiras amostras foram na reserva A e as 50 ultimas na reserva B. Crie uma variável chamada locais que represente a reserva onde cada uma das 100 amostras foram coletadas.

5. Suponha que você deseja jogar na mega-sena, mas não sabe quais números jogar, use a função `sample` do R para escolher seis números para você. Lembre que a mega-sena tem valores de 1 a 60.

6. Crie uma sequencia de dados de 1 a 30 apenas com números impares. Use a função `seq()`.

7. Simule o resultado de 250 jogadas de um dado. ?`sample`. Descubra como usar a função `table` para saber quantas vezes cada número do dado foi sorteado. ?`table` OBS: a página de help irá conter muitas informações que você ainda não sabe o que é e que não precisará utilizar. Veja os exemplos para ter uma ideia de como funciona.

```
a <- c("A","B","C")
table(sample(a,replace=TRUE))
```

8. Crie um objeto com estes dados: **`9 0 10 13 15 17 18 17 22 11 15`** e chame-o de **temp**. Agora faça as seguintes transformações com esses dados: 1) raiz quadrada de **temp**, 2) log natural de **temp**, 3) log(x+1) de **temp**, 4) eleve os valores de **temp** ao quadrado.

9. Feche o R usando a função `q()` : Ao fechar aparecerá a pergunta se deseja salvar Workspace, diga que sim. Abra o R novamente e confira se seus objetos ainda estão salvos usando o comando `ls()`.

Gráficos do R

PLOTS

Produzir gráficos de qualidade é uma ótima forma para apresentar e explorar dados. O R oferece um output gráfico poderoso e de alta definição, onde você pode mexer virtualmente em tudo! Veja um exemplo:

```
> example(persp)
```

Esse comando mostra uma série de gráficos produzidos pela função `persp()`. A figura 3 mostra o segundo gráfico (precione o Enter no *workspace* quando você estive pronto para passar para o próximo gráfico). Veja que para cada exemplo o *script* é mostrado no console, então você pode testar novas mudanças brincando com os argumentos.

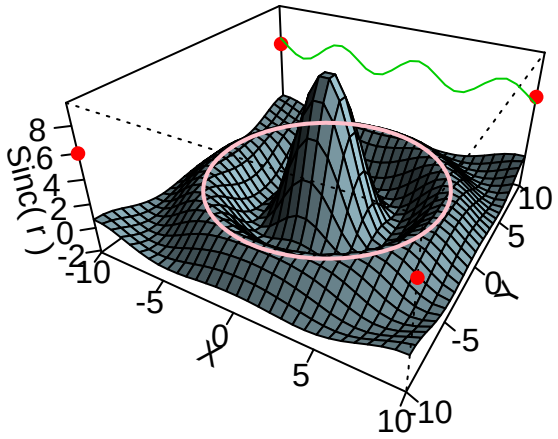


Figura 3. Segundo gráfico da função `persp()`. Isso é só um exemplo, no mundo real evite a qualquer custo fazer um gráfico 3D!

Vamos dar uma olhada como fazer os gráficos mais comuns (gráficos de barras, pizza e dispersão).

Gráficos de barras

Para fazer gráficos de barras no R a função é `barplot`.

```
> barplot(sample(10:100,10))
```

Veja os exemplos de gráficos de barras:

```
> example(barplot) # clique na janela do gráfico para ir passando os exemplos.
```

Gráficos de pizza

Para fazer gráficos de pizza a função `pie`.

```
> pie(c(1,5,7,10))
```

Veja os exemplos de gráficos de pizza

```
> example(pie)
```

Gráfico de pontos (gráficos de dispersão)

Gráficos com variáveis numéricas

Primeiro vamos inserir os dados de duas variáveis numéricas. Lembre que a forma mais simples de inserir dados no R é usando a função de concatenar dados "c".

```
> y<-c(110,120,90,70,50,80,40,40,50,30)
```

```
> x<-1:10
```

y geralmente é a letra usada em livros texto para indicar a variável resposta, a que aparece no eixo Y. Apesar de não ser uma norma, colocar a variável resposta no eixo y (vertical) dos gráficos é um consenso entre a maioria dos estatísticos, daí a letra y para dados resposta. x é chamada de variável independente e aparece no eixo x (horizontal).

É extremamente simples fazer um gráfico de pontos de y contra x no R. A função utilizada é `plot()` e precisa de apenas dois argumentos: o primeiro é o nome da variável do eixo X, o segundo é o da variável do eixo Y.

```
> plot(x,y)
```

Você também pode dizer se quer um gráfico de linhas, de pontos, ou de ambos:

```
> plot(x,y,type="p")
> plot(x,y,type="l")
> plot(x,y,type="b")
```

Ou dizer que não quer plotar nada no gráfico usando "n"

```
> plot(x,y,type="n")
```

Alterando a aparência do gráfico

Para a proposta de apenas explorar os dados, o gráfico acima geralmente é o que você precisa. Mas em publicações é necessário saber melhorar a aparência do gráfico. Sempre é bom ter nomes informativos nos eixos (no R a opção default de nome das legendas é o próprio nome das variáveis). Suponha então que queremos mudar o nome da variável do eixo x para "Variável explanatória". Para isso, o argumento `xlab` ("*x label*") é utilizado. Use as setas do teclado para voltar ao comando anterior e coloque uma "vírgula" após o y e depois da vírgula coloque o comando da legenda do eixo x (`xlab="Variável explanatória"`)

```
> plot(x,y, xlab="Var explanatória")
```

Você pode alterar a legenda do eixo y da mesma forma, porem usando `ylab`. Use as setas e coloque o argumento da legenda do eixo y.

```
> plot(x,y,xlab="Var explanatória",ylab="Var resposta")
```

Também é fácil mudar os símbolos do gráfico, neste momento você está usando a opção default, que é a "bolinha vazia" (`pch=1`). Se você deseja que o símbolo seja um "x" use `pch=4`. Use a seta para cima pra inserir o argumento `pch`.

```
> plot(x,y,xlab="Var explanatória",ylab="Var resposta" ,pch=4)
```

Use outros valores (de 1 a 25) para ver outros símbolos.

Para ver todas as opções de símbolo de uma vez digite:

```
> plot(1:25,  pch=1:25)
> text(1:25,(1:25)-0.9)
```

Para colocar título no gráfico uso o argumento `main`:

```
> plot(x,y,xlab="Var explanatória",ylab="Var resposta" ,pch=4,
main="Título do gráfico")
```

Adicionando linhas a um gráfico de pontos

A função utilizada para inserir linhas é `abline()`. Vamos usar a função abline para inserir uma linha que mostra a média dos dados do eixo Y:

```
> abline(h=mean(y)) # o h é de horizontal. Fará uma linha na horizontal que passa pela média de y.
```

Para passar uma linha que passa pela média de x

```
> abline(v=mean(x)) # o v é de vertical
```

Vamos passar uma linha que passa pelo sétimo valor do eixo x e mudar a cor da linha

```
> abline(v=7, col="red")# pode escrever o nome da cor ou números (abaixo)
```

Também é possível inserir as duas linhas ao mesmo tempo:

```
> plot(x,y)
> abline(h=mean(y), v=mean(x),col=4)
```

Com cores diferentes

```
> abline(h=mean(y), v=mean(x),col=c(2,4))
```

Adicionar mais pontos ao gráfico

Em alguns casos podemos querer inserir pontos de outro local no mesmo gráfico, usando símbolos diferentes para o novo local. Suponha que temos novos valores da variável resposta e da variável explanatória, coletados em outro local, e queremos adicionar estes valores no gráfico. Os valores são

```
v<-c(3,4,6,8,9) ## novos valores da variável explanatória
w<-c(80,50,60,60,70) ## novos valores da variável resposta
```

Para adicionar estes pontos ao gráfico, basta usar a função `points()`, e usar uma cor diferente para diferenciar.

Primeiro vamos refazer o gráfico com x e y e depois adicionar os novos pontos.

```
> plot(x,y)
> points(v,w,col="blue")
```

Gráficos com variáveis explanatórias que são categóricas.

Variáveis categóricas são fatores com dois ou mais níveis (você verá isso no curso de estatística). Por exemplo, sexo é um fator com dois níveis (macho e fêmea). Podemos criar uma variável que indica o sexo como isto:

```
> sex<-c("macho","fêmea")
```

A variável categórica é o fator sexo e os dois níveis são "macho" e "fêmea". Em princípio, os níveis do fator podem ser nomes ou números (1 para macho e 2 para fêmea). Use sempre nomes para facilitar.

Vamos supor que os 5 primeiros valores da nossa variável y eram machos e os 5 últimos eram fêmeas e criar a variável que informa isso.

```
> sexo<-c("Ma","Ma","Ma","Ma","Ma","Fe","Fe","Fe","Fe",
+ "Fe")    # Não digite o sinal de +
```

Para parecer um exemplo mais real vamos supor que y são valores de peso, para isso vamos apenas salvar um objeto chamado peso que é igual a y.

```
> peso<-y  # peso é igual a y
> peso
```

Agora vamos fazer o gráfico:

```
> plot(sexo,peso)
```

Observe que o comando não funcionou, deu erro! Isso ocorreu porque não informamos que sexo é um fator. Vamos verificar o que o R acha que é a variável sexo.

```
> is(sexo)
```

Veja que o R trata a variável sexo como sendo um "vetor de caracteres". Mas nós sabemos que sexo é o nosso fator, então precisamos dar esta informação ao R. A função `factor`, transforma o vetor de caracteres em fator.

```
> factor(sexo)
```

Veja que o R mostra os "valores" e depois mostra os níveis do fator. Agora podemos fazer o nosso gráfico adequadamente:

```
> plot(factor(sexo),peso)
```

Você também pode salvar a variável sexo já como um fator.

```
> sexo.f<-factor(sexo)
```

Gráficos do tipo boxplot são bons quando o número de observações (de dados) é muito grande. Neste caso, um gráfico com pontos seria melhor, pois podemos ver quantas observações foram utilizadas para produzir o gráfico.

Para fazer um gráfico de pontos quando uma variável é categórica precisamos usar a função `stripchart`.

```
> stripchart(peso~sexo) # faz o gráfico, mas na horizontal
```

```
> stripchart(peso~sexo,vertical=TRUE)    # agora o gráfico está na vertical, porém os pontos aparecem nas extremidades. TRUE pode ser abreviado para apenas T.
```

```
> stripchart(peso~sexo,vertical=T,at=c(1.3,1.7))  # agora os pontos estão centralizados, pois com o argumento at, nós especificamos a localização dos pontos no eixo X. Note que agora só há um problema. Eram cinco fêmeas e no gráfico aparecem apenas 4. Isso ocorreu porque duas fêmeas tinham o mesmo peso. Para melhorar o gráfico é necessário usar o argumento method="stack", para que os pontos não fiquem sobrepostos.
```

```
> stripchart(peso~sexo,vertical=T,at=c(1.5,1.7),
method="stack") # os pontos não estão mais totalmente sobrepostos, mas um símbolo ainda está sobre o outro. Usando o argumento offset conseguimos separá-los.
```

```
> stripchart(peso~sexo,vertical=T,at=c(1.3,1.7),method=      +
"stack",offset=1)      ## Não digite o sinal de +
```

Inserir texto em gráficos

Para inserir textos em gráficos a função é `text()`. Ela funciona assim:

```
text(cooX,cooY,"texto")
```

Neste caso cooX e cooY indicam as coordenadas do eixo X e do eixo Y onde o texto deverá ser inserido. Vamos criar um gráfico simples:

```
> plot(1:10,1:10)
```

Agora vamos adicionar "seu nome" no gráfico nas coordenadas 6 e 7. Isso indica que seu nome ficará na posição 6 do eixo X e na posição 7 do eixo Y.

```
> text(6,7,"Seu.Nome")
```

Para inserir duas palavras:

```
> text(c(2,3),c(8,6),c("nome","sobrenome"))
```

Ou seja, a primeira palavra ficará na posição 2-8 e a segunda palavra ficará na posição 3-6.

Também é possível adicionar texto aos gráficos usando a função `locator(n)`. O n indica o número de pontos que você deseja indicar no gráfico. A função `locator` permite que você clique no gráfico com o *mouse* e adicione o texto na posição desejada.

```
> plot(1:10,1:10)
> text(locator(1),"Texto")  # clique com o mouse na posição desejada no gráfico
```

Para mais de um texto:

```
> plot(1:10,1:10)
> text(locator(3),c("texto 1","texto 2","texto 3"))
```

Dividir a janela dos gráficos

Em alguns casos em que se deseja comparar dois ou mais gráficos é possível dividir a janela de gráficos do R. A função `par` controla diversas características dos gráficos. Sempre que quiser melhorar algum aspecto de seu gráfico consulte o help da função (`?par`) e descubra o argumento necessário. Para dividir a janela de gráficos o comando é:

`par(mfrow=c(nl,nc))` # nl indica o número de linhas e nc o número de colunas que a janela deverá ter. Primeiro vamos dividir a janela em duas colunas.

```
> par(mfrow=c(1,2))   # uma linha e duas colunas
> plot(sample(1:10))
> plot(rnorm(10))
```

Agora vamos dividir a janela em duas linhas e uma coluna:

```
> par(mfrow=c(2,1))
> plot(1:10)
> plot(runif(15))
```

Agora vamos dividir a janela em duas linhas e duas colunas (para plotar 4 gráficos)

```
> par(mfrow=c(2,2))
> plot(1:10)
> plot(1:10,1:10)
> plot(1:10, rep(1,10))
> plot(runif(100),rnorm(100))
```

Salvar os gráficos

Existem diversas opções para salvar os gráficos do R. A mais simples é clicar com o botão direito do *mouse* sobre o gráfico e depois em *"save as metafile".* Também é possível salvar os gráficos em PDF, JPEG, BMP, PNG ou outros formatos. Para salvar em pdf ou outros formatos clique sobre o gráfico e depois clique em *Arquivo* e em *salvar como*. Escolha a opção que deseja salvar.

Muitas vezes é preciso salvar gráficos com resolução alta, ou criar painéis grandes com vários gráficos. Você consegue fazer tudo isso no R. Por exemplo, vamos dizer que o editor da revista pediu para você gerar um gráfico na extenção tiff com 300 dpi de resolução. Vc pode facilmente criar esse gráfico, com o comando abaixo:

```
> tiff(filename="figura1.tiff", width=1000, height=1000, res=300)  # cria um arquivo que vai receber seu gráfico, especificando alguns parâmetros
> plot(runif(100),rnorm(100))  # plota o gráfico
> dev.off ()  # encerra o commando, ou seja, produz a figura.
```

A “figura 1” vai aparecer na sua pasta de trabalho. Compare a qualidade da figura 1 produzida pelos comandos acima, com as figuras que vc salvou usando a opção *salvar como*. A figura 1 tem algo em torno de 4 mb (megabites), enquento que a figura gerada pela opção salvar como, tem em torno de 100 kb (kilobites). Veja o help das funções que criam imagens, `?tiff`, `?pdf`, `?png`, `?jpeg` para mais detalhes. O help traz todas as informações necessárias para você produzir figuras de alta qualidade e no tamanho que desejado.

Resumo sobre gráficos

Plot: `plot(xaxis,yaxis)` faz um gráfico de pontos se x é uma variável contínua e um boxplot se x é uma variável categórica (se é um fator).

Lines: `lines(x,y)` traça uma linha reta de acordo com as coordenadas fornecidas. É possível mudar o tipo e a cor das linhas usando os argumentos `lty` para o tipo e `col` para a cor.

Points: `points(x,y)` adiciona mais pontos em um gráfico. É possível colocar os novos pontos com símbolo diferente, usando o argumento `pch=2` ou `pch="H"`.

stripchart: é uma função que faz gráficos de pontos, com a qual é possível separar pontos coincidentes, quando a variável “x” for categórica.

Cores: Gráficos em preto e branco são bons na maioria dos casos, mas cores podem ser mudadas usando `col="red"` (escrevendo o nome da cor) ou `col=2` (usando números).

O comando abaixo mostra os números que especificam algumas cores

```
> pie(rep(1,30),col=rainbow(30))
```

par: a função `par` é utilizada para alterar diversos aspectos dos gráficos, dentre eles, dividir a janela para que caiba mais de um gráfico.

Exercícios com gráficos

1- Um biólogo foi ao campo e contou o número de sapos em 20 locais. Ele também anotou a umidade e a temperatura em cada local. Faça dois gráficos de pontos para mostrar a relação do número de sapos com as variáveis temperatura e umidade. Use a função `par()` para dividir a janela em duas.

Os dados são:

sapos	6-5-10-11-26-16-17-37-18-21-22-15-24-25-29-31-32-13-39-40
umid	62-24-21-30-34-36-41-48-56-74-57-46-58-61-68-76-79-33-85-86
temp	31-23-28-30-15-16-24-27-18-10-17-13-25-22-34-12-29-35-26-19

2- Um biólogo interessado em saber se o número de aves está relacionado ao número de uma determinada espécie de árvore realizou amostras em 10 locais. Os valores obtidos foram:

```
aves<-c(22,28,37,34,13,24,39,5,33,32)
arvores<-c(25,26,40,30,10,20,35,8,35,28)
```

Faça um gráfico que mostra a relação entre o número de aves e o número de árvores.

Um colega coletou mais dados sobre aves e árvores, em outra área, que podemos aproveitar. Os dados são: árvores2 (6,17,18,11,6,15,20,16,12,15),.

```
arvores2<-c(6,17,18,11,6,15,20,16,12,15)
aves2<-c(7,15,12,14,4,14,16,60,13,16)
```

Inclua estes novos pontos no gráfico com um símbolo diferente e cor azul.

Junte o seu arquivo de aves com o arquivo de aves do seu amigo, para que fique em um arquivo completo: `aves.c<-c(aves,aves2)`. Faça o mesmo para árvores.

3- Os dados do exercício anterior foram coletados em regiões diferentes (você coletou no local A e seu amigo no local B). Inclua esta informação no R, use a função `rep` se quiser facilitar.

Faça um gráfico para ver qual região tem mais aves e outro para ver qual tem mais árvores. Lembre que local deve ser um fator para fazer o gráfico. Use função `stripchart`.

```
stripchart(aves.c~locais, vertical=TRUE, pch=c(16,17))
```

Existem locais com o mesmo número de aves, e no gráfico estes pontos apareceram sobrepostos. Faça o gráfico sem pontos sobrepostos:

4 - Existem milhares de outros comandos para alterar a aparência de gráficos, veja esta página do help (`?par`) para ver alguns comandos sobre como alterar a aparência de gráficos. Não se preocupe se você ficar confuso sobre as opções do `help(par)`, apenas praticando você irá começar a dominar estes comandos.

5 – Repita o gráfico do exercício 2 acima e faça as seguintes modificações.

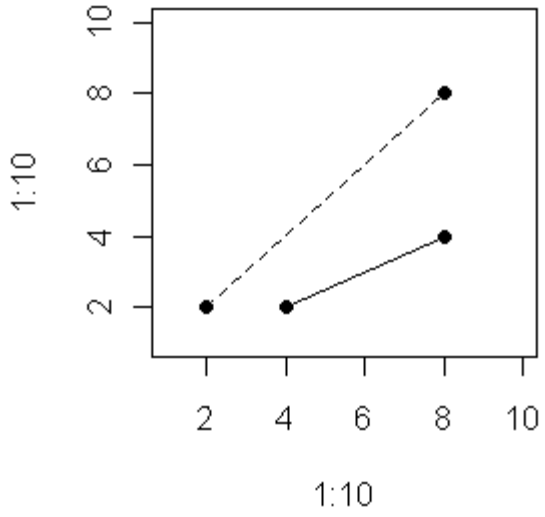
Coloque um título no gráfico

Use Bolinhas cheias e azuis como símbolo.

Coloque as legendas no eixo-x e no eixo-y. Por exemplo: “Número de aves”

O padrão do R é fazer uma “caixa” entorno do gráfico, faça uma alteração para que apareça apenas as linhas do eixo-x e do eixo-y. veja `bty` em `?par`

6 – Faça um gráfico igual ao gráfico abaixo: Use a função `plot`, a função `points` e a função `lines`.



7 – Use a função `rnorm` para gerar dois conjuntos de dados aleatórios com 100 valores e faça um gráfico plotando um dos conjuntos contra o outro.

Manejo de dados

Nesta sessão veremos como importar, exportar, alterar e manejar um arquivo de dados dentro do R. Acessar partes de um conjunto de dados é uma tarefa rotineira dentro do R e da análise de dados em geral. Por exemplo, em diversos casos teremos que selecionar linhas ou colunas dentro de uma planilha, ou em outros casos precisaremos agregar linhas ou colunas. Veremos como fazer coisas desse tipo nesta sessão.

Importar conjunto de dados para o R

Passe a tabela abaixo (locais, abundância de macacos e de árvores frutificando) para o Excel e salve-a (na pasta do curso) em formato de "texto separado por tabulações" (No Excel clique em "salvar como" e em texto (separado por tabulações)). Salve com o nome de `amostras.txt`.

Amostra	reserva	macacos	frutas
A1	A	22	25
A2	A	28	26
A3	A	37	40
A4	A	34	30
A5	A	13	10
A6	A	24	20
A7	A	39	35
A8	A	5	8
A9	A	33	35
A10	A	32	28
B1	B	7	6
B2	B	15	17
B3	B	12	18
B4	B	14	11
B5	B	4	6
B6	B	14	15
B7	B	16	20
B8	B	60	16
B9	B	13	12
B10	B	16	15

Note que esta tabela contém variáveis numéricas e categóricas, portanto este é um exemplo de objeto do tipo *dataframe*. Para importar a tabelas para o R a função é `read.table`.

```
> read.table("amostras.txt",header=TRUE)
```

O argumento `header=TRUE` informa que os dados possuem cabeçalho, ou seja, a primeira linha contém os nomes dos atributos (variáveis) e a primeira coluna possui o nome das amostras.

Não se esqueça de salvar os dados, não basta importar, é preciso salvar também.

```
> macac<-read.table("amostras.txt",header=TRUE)
```

Para ver os dados digite o nome do arquivo.

```
> macac
```

O objeto `macac` é um objeto do tipo `dataframe`. Isso quer dizer que `macac` é um objeto que possui linhas e colunas (observações nas linhas e variáveis (atributos) nas colunas).

Procurar os dados dentro do computador

Caso você não lembre o nome do arquivo de dados que deseja importar existe a opção de procurar os dados no computador com a função `file.choose()`:

```
> macac<-read.table(file.choose(),header=T)# encontre o arquivo macac.txt
```

Você também pode conferir se um determinado arquivo de dados existe no seu diretório:

```
> file.exists("macac.txt")
```

Transformar vetores em matrizes e data frames

Além de importar tabelas, existem opções para juntar vetores em um arquivo dataframe ou matriz. Para criar uma matriz use `cbind` (colum bind) ou `rbind` (row bind). Vamos ver como funciona. Vamos criar três vetores e depois juntá-los em uma matriz.

```
> aa<-c(1,3,5,7,9)

> bb<-c(5,6,3,8,9)

> cc<-c("a","a","b","a","b")

> cbind(aa,bb)    # junta os vetores em colunas

> rbind(aa,bb) # junta os vetores em linhas
```

Lembre que matrizes podem conter apenas valores numéricos ou de caracteres. Por isso, se juntarmos o vetor cc, nossa matriz será transformada em valores de caracteres.

```
> cbind(aa,bb,cc)    # junta os vetores em colunas, mas transforma números em caracteres.
```

Para criar uma dataframe com valores numéricos e de caracteres use a função data.frame:

```
> data.frame(aa,bb,cc)

> data.frame(aa,bb,cc)    #  agora temos números e letras.
```

Acessar partes da tabela de dados (matrizes ou dataframes)

Agora vamos aprender a selecionar (extrair) apenas partes do nosso conjunto de dados `macac` usando `[]` colchetes. Diferente dos vetores, os dataframes apresentam duas dimensões, as linhas e colunas. Da mesma forma que os vetores, as informações nos dataframes são arquivadas na mesma ordem do seu arquivo e cada linha e coluna pode ser acessado pelo seu índice (número dentro de colchetes). Por convenção você sempre deve informar primeiro o valor da linha e depois da coluna, separado por vírgula. O uso de colchetes funciona assim: `[linhas, colunas]`, onde está escrito `linhas` você especifica as linhas desejadas, na maioria dos casos cada linha indica uma unidade amostral. Onde está escrito `colunas`, você pode especificar as colunas (atributos) que deseja selecionar. Veja abaixo:

```
> macac[3,3]          # extrai a terceira linha e a terceira coluna, 1 valor
> macac[1,3]          # extrai o valor da primeira linha e terceira coluna
> macac[,1]           # extrai a primeira coluna e todas as linhas
> macac[,2]           #  extrai a segunda coluna e todas as linhas
> macac[1,]           # extrai a primeira linha e todas as colunas
> macac[c(1:5),c(2,3)] # extrais somente as linhas 1 a 5 e as colunas 2 e 3
```

Existem outras duas maneiras de extrair dados de uma dataframe. Uma é usando a função `attach()`, que torna as variáveis acessíveis apenas digitando o nome delas na linha de comandos

Digite macacos na linha de comandos e veja o que acontece!

```
> macacos
Error: object "macacos" not found
```

Agora use a função `attach`, digite macacos novamente e veja o que acontece.

```
> attach(macac)
> macacos  # agora os dados macacos está disponível
> frutas   # para ver o número de arvores frutificando
> reserva  # para ver as reservas
> plot(macacos,frutas)
```

IMPORTANTE: A função `attach` apesar de parecer "uma mão na roda" pode ser problemática. Se "atacharmos" duas dataframes, e elas tiverem variáveis com o mesmo nome, é "perigoso" usarmos por engano a variável errada. Se você usar a função `attach` é seguro "desatachar" o objeto imediatamente após o seu uso. Para isso use `detach()`.

```
> detach(macac)
macacos   # macacos não está mais disponível
```

Uma melhor forma de acessar colunas pelo nome é usar o comando cifrão `$`, e não usar a função `attach()`. O uso é basicamente o seguinte `dados$variável` (`dados` especifica o objeto e `variável` a variável que deseja extrair). Por exemplo, para extrair os dados de macacos de use:

```
> macac$macacos
```

Ou então usar colchetes e o nome da variável:

```
> macac[,"macacos"]
```

Vamos fazer o gráfico de macacos X frutas usando $.

```
> plot(macac$frutas,macac$macacos)
```

Faça o gráfico de frutas X macacos usando colchetes ao invés de usar $.

Operações usando dataframes

Ordenar a tabela

Os dados de macac estão dispostos na ordem em que foram coletados. Em alguns casos podemos querer colocá-los em outra ordem. Por exemplo, na ordem crescente de quantidade de árvores frutificando. Para isso use a função `order`.

```
> macac[order(macac$frutas),] # ordena os dados macac pela ordem de frutas
```

Para colocar em ordem decrescente use o argumento `decreasing=TRUE`

Calcular a média de uma linha ou de uma coluna

Podemos calcular a média de cada coluna da dataframe usando.

```
> mean(macac[,"macacos"]) # média de macacos ou use
> mean(macac$macacos)
> mean(macac)             # média de cada coluna
```

Repare que resultado para reservas foi NA e em seguida apareceu uma mensagem de aviso. NA indica *Non Available,* pois não é possível calcular a média de variáveis categóricas.

Acima nós calculamos a média de macacos, mas sem considerar a reserva onde as amostras foram coletadas. O que fazer para calcular a média de macacos em cada reserva? Basta selecionar as linhas correspondentes a cada reserva.

```
> mean(macac[1:10,3]) # média de macacos na reserva A
> mean(macac[11:20,3]) # média de macacos na reserva B
```

Para facilitar, principalmente quando tivermos muitas categorias, o melhor é usar a função `tapply()`. Ela funciona assim: `tapply(dados, grupos, função)`. Será calculada uma "`função`" nos "`dados`" em relação aos "`grupos`". Então, vamos calcular a média (`função`) dos macacos (`dados`) em cada reserva (`grupos`).

```
> tapply(macac$macacos,macac$reserva,mean)
```

Veja que agora calculamos a média de macacos na reserva A e a média na reserva B. A função tapply() é uma variante já incorporada ao código R da função apply(). Existem outras como sapply(), lapply()… Entre no help dessas funções e repare as diferenças. Em um primeiro momento a lógica dessas funções pode parecer um pouco estranha, mas depois de um tempo usando o R você verá que essas funções são muito poderosas e versáteis.

Somar linhas e somar colunas

Agora, vamos usar dados sobre a abundância de moluscos que foram coletados em dez parcelas. A tabela também contém informações sobre quantidade de chuva em cada parcela e em qual de duas reservas (A ou B) a parcela estava localizada. O arquivo com os dados é `moluscos.txt` e pode ser encontrado em https://sites.google.com/site/vllandeiror/, importe-o para o R. Chame o arquivo de `mol` para facilitar.

Somar os valores de colunas ou linhas usando as funções `colSums` para somar colunas e `rowSums` para somar linhas.

```
> colSums(mol[,1:6])        #Note que estamos somando apenas as informações sobre as espécies (colunas 1 a 6)
> rowSums(mol[,1:6])
```

Qual informação obteve ao usar os dois comandos acima?

Medias das linhas e colunas

Calcular a média das colunas e linhas. Abundância média por parcela e abundância média por espécie.

```
> colMeans(mol[,1:6])
> rowMeans(mol[,1:6])
```

E se quisermos calcular apenas a abundância média de moluscos na reserva A (linhas de 1 a 5)? Você consegue fazer este cálculo? Veja as quatro opções abaixo e diga qual delas fará o cálculo correto.

```
1 - mean(mol[,1:6])
2 - rowMeans(mol[1:5,1:6])
3 - colMeans(mol[1:5,1:6])
4 - mean(rowSums(mol[1:5,1:6]))
```

Exemplo com dados reais

O arquivo chamado simu.txt pode ser encontrado em https://sites.google.com/site/vllandeiror/. Este arquivo contém amostras de Simuliidae (borrachudos - piuns) coletadas em 50 riachos da Chapada Diamantina - Bahia. Importe este arquivo e vamos ver alguns exemplos de opções que podemos fazer com este conjunto de dados.

```
> simu<-read.table("simu.txt",header=T)
```

Use `names()` para ver o nome das variáveis que estão no arquivo.

```
> names(simu)
```

Note que as 6 primeiras variáveis são ambientais e depois vem os dados abundância de 20 espécies. Estes dados foram coletados em três municípios dentro da chapada diamantina (Lençóis, Mucugê e Rio de Contas).

Primeiro vamos separar os dados das espécies dos dados ambientais:

```
> ambi<-simu[,1:6]

> spp<-simu[,7:26]
```

Vamos ver os gráficos que mostram as relações entre as variáveis ambientais.

```
> plot(ambi[,"temperatura"],ambi[,"altitude"])

> plot(ambi[,"altitude"],ambi[,"pH"])

> plot(ambi[,"condutividade"],ambi[,"pH"])
```

No R a função `pairs()` faz um gráfico com todas essas comparações entre as variáveis ambientais.

```
> pairs(ambi)
```

Vamos calcular a abundância total de borrachudos em cada riacho.

```
> rowSums(spp)   # soma das linhas (riachos)
```

Para salvar a abundância:

```
> abund<-rowSums(spp)
```

E para calcularmos a riqueza de espécies em cada riacho? Primeiro precisamos transformar os dados em presença e ausência (0 para ausência e 1 para presença). Primeiro vamos criar uma cópia do arquivo original.

```
> copia<-spp   # cópia é igual a spp
```

Agora vamos criar o arquivo de presença e ausência:

```
> copia[copia>0]<-1   ## quando o valor de abundância de copia for maior que 0 o valor será substituído por 1

> pres.aus<-copia ## apenas para mudar o nome do arquivo

> pres.aus        # veja que agora os dados estão apenas como 0 e 1
```

Para encontrar a riqueza de espécies basta somar as linhas do arquivo de presença e ausência.

```
> riqueza<-rowSums(pres.aus)  # número de espécies por riacho (riqueza)
```

Para calcular a riqueza média:

```
> riq.media<-mean(rowSums(pres.aus))
```

Vamos calcular a riqueza média por município onde foram realizadas coletas.

```
> riq.muni<-tapply(riqueza,ambi[,"município"],mean)
```

Agora use a função `colSums()` para ver em quantos riachos cada espécie ocorreu e qual a abundância total de cada espécie. Qual a espécie que ocorreu em mais riachos? Qual era a mais abundante?

Para transformar os dados de abundância de espécies em log:

```
> simu.log<-log(spp)
```

Veja o resultado em log:

```
> simu.log        # Note a presença do valor -inf
```

O valor -inf ocorre porque não é possível calcular o log de 0. Veja:

```
> log(0)
```

Por isso é comum você ver em trabalhos os dados transformados em log(x+1):

```
> spp.log<-log(spp+1)

> spp.log
```

Agora vamos retornar ao nosso arquivo completo, `simu,` e ordenar a tabela de acordo com a altitude, de forma que o primeiro riacho seja o que está localizado em menor altitude.

```
> simu[order(simu[,"altitude"]),]# tabela ordenada pela altitude
```

Vamos entender esse comando passo a passo:

Primeiro descobrimos a ordem que os dados devem estar para que a tabela fique em ordem:

```
> order(simu[,"altitude"])      # veja que o primeiro valor é o 18, que é o riacho
```

que ocorre em menor altitude.

Então, para colocar a tabela inteira em ordem de altitude, nós usamos o resultado de `order(simu[,"altitude"]).`

```
> simu[order(simu[,"altitude"]),]  # assim os dados são ordenados pela
```

altitude.

Agora vamos fazer 4 gráficos da riqueza de espécies em relação a altitude, pH, temperatura e condutividade. Primeiro vamos dividir a janela de gráficos em 4 partes.

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

Riqueza X altitude

```
> plot(simu[,"altitude"],rowSums(pres.aus))
```

Faça os outros três gráficos.

Agora vamos fazer um gráfico para ver a riqueza de espécies nas três áreas (Lençóis, Mucugê e Rio de contas).

```
> stripchart(riqueza~factor(simu[,"município"]))
```

Coloque o gráfico na vertical e separe os pontos coincidentes.

As funções aggregate e by

As funções `by` e `aggregate` podem ser utilizadas para aplicar uma função em todas as colunas de uma tabela, enquanto a função `tapply` faz isso apenas uma coluna por vez. As duas funções retornam os mesmos resultados, mas de formas diferentes. Para calcular a média de cada espécie de borrachudo em cada um dos 3 locais use:

```
> aggregate(spp,list(simu[,1]),mean)
```

A função `aggregate` retorna uma dataframe com os valores para cada local.

```
> by(spp,simu[,1],mean)
```

A função `by` retorna uma lista com os valores para cada local. Para acessar os valores de uma lista é preciso usar dois colchetes:

```
> med.local<-by(spp,simu[,1],mean)

> med.local

> med.local[["Lençóis"]]

> med.local[["Mucugê"]]

> med.local[["R.Contas"]]
```

Transpor uma tabela de dados

Em alguns casos é necessário transpor uma tabela de dados, ou seja, colocar as informações das linhas nas colunas e as colunas nas linhas. A função é `t()` e indica *transpose.*

```
> t(spp)
```

Comandos de lógica

Opções para manipular conjunto de dados.

Primeiro vamos ver o significado dos comandos abaixo.

>	Maior que	>=	maior que ou igual a
<	Menor que	<=	menor que ou igual a
==	igualdade	!=	diferença

```
> x<-c(1,2,9,4,5)
> y<-c(1,2,6,7,8)
> x>y    # Retorna TRUE para os maiores e FALSE para os menores
> x>=y
> x<y
> x==y  # Retorna TRUE para os x que são iguais a y
> x!=y  # Retorna TRUE para os x que são diferentes de y
```

Agora vamos selecionar partes dos dados que obedecem a algum critério de seleção.

which

A função `which` funciona como se fosse a pergunta: Quais?

```
> a<-c(2,4,6,8,10,12,14,16,18,20)
> a>10        # Retorna um vetor contendo TRUE se for maior e FALSE se for menor
> which(a>10)    # Equivale a pergunta: "Quais valores de a são maiores que 10?". Note que a resposta é a posição dos valores (o sexto, o sétimo…) e não os valores que são maiores que 10.

> a[6]                    # selecionamos o sexto valor de a

> a[c(6:10)]              # selecionamos do sexto ao décimo valor
```

Tente prever o que ocorrerá usando o comando abaixo.

```
> a[which(a>=14)]
```

Acertou? Selecionamos os valores de a que são maiores ou igual a que 14!

Também podemos usar a função `which` para selecionar partes de uma tabela de dados. Por exemplo, vamos selecionar apenas as parcelas dos dados de moluscos onde a chuva é maior que 1650 mm.

Lembre-se que para selecionarmos partes de uma tabela podemos usar colchetes `[linhas,colunas]` e especificar as linhas e colunas que desejamos usar. Então vamos usar o comando `which` para escolher apenas as linhas (parcelas) onde a chuva é maior que 1650mm.

```
> mol[which(mol$chuva>1650),]
```

Podemos escolher também apenas a parte da tabela que corresponde às amostras da reserva A.

```
> mol[which(mol$reserva=="A"),]
```

Também podemos incluir um segundo critério de escolha usando & (que significa "e") ou | (que significa "ou"). Vamos escolher apenas as parcelas da reserva B e que tenham o valor de chuva maior que 1650mm.

```
> mol[which(mol$reserva=="B" & mol$chuva>1650),]
```

Compare com o resultado do commando abaixo:

```
> mol[which(mol$reserva=="B" | mol$chuva>1650),]
```

O que mudou?

ifelse

Agora vamos aprender a usar o comando `ifelse` que significa: se for isso, então seja aquilo, se não, seja aquilo outro. O comando funciona da seguinte maneira: `ifelse`(aplicamos um teste, especificamos o valor caso o teste for verdade, e o valor caso falso). Complicado? Vamos ver alguns exemplos para facilitar as coisas.

Primeiro crie no R um objeto com o valor do salário de dez pessoas. Os valores são:

```
> salarios<-c(1000, 400, 1200, 3500, 380, 3000, 855, 700,
+ 1500, 500)    ## Não digite o sinal de +
```

As pessoas que ganham menos de 1000 ganham pouco, concorda? Então aplicamos o teste e pedimos para retornar "pouco" para quem ganha menos de 1000 e "muito" para quem ganha mais de 1000.

```
> ifelse(salarios<1000,"pouco","muito")# Se o salário é menor que 1000, seja pouco, se for maior seja muito.
```

Também podemos usar o comando `ifelse` para transformar os dados em presença e ausência. Vamos usar os dados das espécies de borrachudos (spp) da Chapada Diamantina.

```
> ifelse(spp>=1,1,0) # se o valor de spp for maior ou igual a 1 seja 1, se não, seja 0
```

Manipulando dados e formatos de tabela

Como vimos anteriormente, a melhor maneira de armazenar dados coletados em estudos ecológicos é em seu estado mais bruto, onde cada observação equivale a um objeto (ou linha) e seus atributos (colunas) estão descritos para cada observação. Importe a tabela "`tabela_longa.txt`" no R, nomeando-a de "longa". Depois de importa-la, verifique o número de observações (ou de linhas). Você verá que é uma tabela extensa de 1000 observações.

```
> head(tabela_longa) # mostra somente as 6 primeiras linhas dessa tabela
```

Esta é a estrutura de dados mais recomendada para armazenamento porque permite que os dados sejam facilmente manipulados para gerar diferentes tabelas. A partir da tabela de observações, podemos facilmente gerar com o comando `table()` uma tabela de espécies por locais (que é o formato mais usada para análises de comunidades).

```
> table(longa$parcela, longa$especie) # mostra a abundância de cada espécie por parcela
```

Ou calcular qual a abundância média dos indivíduos por espécie usando `tapply()`.

```
> tapply(as.numeric(longa$abundancia), longa$especie, mean)
```

Para ter uma base de dados em que cada observação representa uma linha, informações associadas a uma unidade maior (por exemplo, o nome da parcela) precisam ser repetidas para cada observação. Esse tipo de redundância nos dados facilita a sua manipulação, mas pode não ser igualmente desejável em planilhas de campo ou em artigos científicos. Nestes casos, é mais comum que os dados sejam apresentados de maneira econômica, como uma tabela de contingência apresentada acima.

Combinando tabelas

Até agora vocês criaram bases de dados juntando colunas com o mesmo número de linhas e que estavam na mesma ordem como no exemplo abaixo:

```
> letras<-LETTERS[1:10]

> numeros<-seq(1:10)

> data<-data.frame(cbind(letras,numeros))

> data
```

O mesmo procedimento pode ser feito para qualquer conjunto de dados. Por exemplo, supunha que um colega seu trabalhou nas mesmas parcelas que você. Ele estava investigando outra questão, mas coincidentemente ele coletou informações sobre altitude e pluviosidade por parcela que pode ser útil para

você. Ele disponibilizou esses dados no arquivo “ambiente_2.txt”. Importe essa tabela para o R com o nome de amb2.

```
> amb2 <- read.table(“ambiente_2.txt”, header=T)
```

Note que seu colega não coletou as informações ambientais em todas as parcelas. Mas mesmo assim esses dados são importantes e você precisa incorporá-lo ao seu conjunto de dados. Vimos acima que a função `cbind()` junta colunas de dataframes diferentes. Veja o que acontece quando usamos o `cbind()` para juntar as duas planilhas:

```
> cbind(longa, amb2)
```

As duas tabelas foram combinadas sem problemas, mas mesmo os plots que não tinham dados ambientais em `amb2` agora estão com valores. Mágica? Claro que não. A função `cbind()` foi construída para colar duas colunas ou coluna(a) numa tabela. Esta função assume que se as colunas são de tamanhos diferentes ela deve copiar o conteúdo da coluna menor até que ela fique do mesmo tamanho da coluna maior para que o dataframe não tenha tamanhos diferentes para cada coluna. Isso resolve o problema do dataframe, mas não o seu. Para fazer associação entre tabelas, é preferível usar funções que reconheçam qual é a coluna (ou **chave**) que liga as duas tabelas e fazer a associação por meio desta coluna. Na maioria das vezes, os dados que usamos na vida real não se encaixam perfeitamente. Para juntar duas bases usando o `cbind()`, como no exemplo acima, as bases precisam estar perfeitamente pareadas. Nas bases pareadas, as observações (linhas) devem ter o mesmo número e estarem na mesma ordem. Em alguns casos, conseguimos manter o controle sobre isso, mas dependendo do nível de manipulação de dados que fizermos durante as análises isso pode se tornar bastante complicado. No R, podemos fazer este tipo de associação entre duas tabelas com maior segurança usando a função `merge()`.

```
> longa.amb<-merge(longa,amb2)

> longa.amb
```

Note que o `merge` retornou uma tabela somente com as parcelas que tinham todos os dados, excluindo as parcelas que não tinham dados em `amb2`. Dependendo do tipo de análise este é o resultado esperado, mas também pode ser que você queria ver uma tabela com todas as parcelas, mesmo que elas não tenham dados ambientais adicionais do `amb2`. Neste caso, você deve especificar um parâmetro adicional.

```
> longa.amb.new<-merge(longa,amb2,all.x=T)

> longa.amb.new #Explore as funcionalidades da função merge, olhando o help
```

Como o `merge()` trabalha em cima da associação entre duas colunas, é importante indicar claramente quais são as colunas que ele deve associar. Existem duas maneira de garantir que isso ocorra. A primeira maneira é padronizar o nome da coluna que vai ser associada nas duas tabelas, preferencialmente definindo esta como a primeira coluna em ambas tabelas como fizemos no nosso exemplo anterior. Independente da posição na tabela, a coluna `parcela` é a única coluna que contém elementos em comum nas tabelas `longa` e `amb2`. Esta é forma mais simples e recomendável na maioria dos casos. A outra maneira é indicar como argumento da função `merge()` quais colunas devem ser associadas. Isso é especialmente útil na padronização de nomes, como por exemplo, quando o nome dado a cada parcela pelos diferentes coletores não é o mesmo. Suponha que seu orientador conseguiu as informações das coordenadas geográficas para as parcelas que você trabalhou:

```
# desta vez, vamos criar o dataframe de coordenadas geográficas do seu orientador, em vez de importá-lo!

> coord <- data.frame(matrix(NA,10,3)) # cria uma tabela vazia de 10 linhas e 3 colunas
```

```
> colnames(coord) <-c ("plot","lat","long") # renomeia as colunas
> coord[,1] <- paste("plot_", seq(1,10,1), sep="") # cria os nomes das parcelas seguindo a nomenclatura que seu orientador usou
> coord[,2] <- sample(-5:2,10, replace=T) # gera os número de latitude
> coord[,3] <- sample(-49:-60,10, replace=T) # gera os número de longitute
> coord # Pronto! A tabela com as coordenadas geográficas estão ok!
```

Se você juntar a tabela de coordenadas geográficas com a sua tabela de espécies, o `merge()` não irá retornar nenhum erro. Mas veja que a associação não é feita corretamente porque as duas tabelas não tem nenhuma coluna em comum:

```
> merge(longa.amb.new,coord, all.x=T)
```

Para conseguir associar estes dois conjuntos de dados corretamente é necessário primeiro definir qual é o relacionamento (chave) entre o nome da parcela em uma tabela com o nome do plot na outra tabela. Suponha que as duas tabelas estão relacionadas da seguinte forma (novamente, em vez de importar uma tabela com a chave, vamos criar nossa tabela chave diretamente no R):

```
#definindo associação entre base de campo e coordenadas
> chave <- data.frame(matrix(NA,10,2))
> chave [,1]<-paste("parcela", seq(1,10,1), sep="")
> chave [,2]<-coord[,"plot"]
> colnames(chave)<-c("parcela", "plot")
> chave # tabela de ligação
```

Com a chave pronta fica fácil combinar as tabelas. Só precisamos incluir em uma das tabelas as colunas das tabela chave (que estabelecerá a relação entre as tabelas `coord` e `longa.amb.new`). No nosso exemplo, vamos precisar primeiro estabelecer a associação entre o nome dos plots indicando qual coluna na primeira tabela é comparável com qual coluna na segunda tabela.

```
> coord.ok<-merge(coord,plot.code,by.x="plot",all.x=T)
```

Examine `coord.ok`. Note que nesta tabela que os nomes dos plots da tabela de coordenadas geográficas estão associadas corretamente com os nomes das parcelas que aparecem na tabela `longa.amb.new`. Com a associação entre tabelas estabelecida, poderemos usar o novamente a função `merge()` para fazer a associação entre tabelas da mesma maneira que fizemos anteriormente.

```
> longa.completa <- merge(longa.amb.new,coord.ok)

> longa.completa
```

Note que agora, a tabela está completa com todos as parcelas, as variáveis ambientais coletadas por seu colega e as coordenadas geográficas que seu orientador disponibilizou. Nesse exemplo, criar a chave entre tabelas foi muito fácil e trivial, mas no dia a dia essa tarefa pode ser difícil e em alguns casos a única forma de fazer é manualmente. De qualquer forma, saber como mesclar corretamente tabelas de dados é uma habilidade muito útil nos dias de hoje. Explorarmos aqui somente alguns passos básicos. O R, no entanto tem vários pacotes criados especialmente para manejo de dados, como sql...

Exercícios com dataframes e comandos de lógica:

1-Calcule a média de macacos e frutas dentro de cada reserva do conjunto de dados macac.

2. Quais informações podem ser obtidas da tabela de moluscos quando usamos os quatro comandos abaixo.

```
1 - mean(mol[,1:6])
2 - rowMeans(mol[1:5,1:6])
3 - colMeans(mol[1:5,1:6])
4 - mean(rowSums(mol[1:5,1:6]))
```

3. Use a função `t` para transpor os dados de moluscos (apenas as espécies).

4. Faça um gráfico de pontos para comparar o número de macacos na reserva A com o número de macacos na reserva B. Use o conjunto de dados `macac.` Use a função `stripchart`.

5. Importe para o R o arquivo minhocas.txt que está na pasta do curso. Este arquivo possui dados sobre a densidade de minhocas em 20 fazendas. As variáveis medidas são: área, inclinação do terreno, tipo de vegetação, pH do solo, se o terreno é alagado ou não e a densidade de minhocas.

6.1. Veja a tabela na forma original e depois a ordene de acordo com o tamanho da área.

6.2. Faça um gráfico para ver a relação entre minhocas e área do terreno e outro gráfico para ver a relação entre minhocas e tipo de vegetação.

6.3. Selecione a parte da tabela que contém apenas dados de locais alagados.

6.4. Calcule a densidade média de minhocas, e a densidade média em locais alagados e em locais não alagados.

6.5. Qual a área média das fazendas?

7. Podemos usar a função `ifelse` para transformar dados de abundância em dados de presença e ausência. Transforme os dados de abundância de moluscos (mol) em dados de presença e ausência.

8. Com os dados transformados em presença e ausência, use a função `rowSums` para ver quantas espécies de moluscos ocorrem em cada parcela. Use `colSums` para ver em quantas parcelas cada espécie estava presente.

Linguagem orientada ao objeto

Como o R é uma linguagem de programação orientada ao objeto, você pode combinar vários comandos ou funções usando o resultado do comando ou função anterior. A habilidade de combinar funções no R fornece uma flexibilidade tremenda, e quando usada corretamente, é bem poderosa. Veja o exemplo abaixo:

```
> nrow(subset(cars, speed==18))
```

Primeiro a função `subset()` acessou o *data-frame* cars (que está disponível no R) e selecionou todos os registros que a variável *speed* é igual a 18. Isso resulta em um novo data-frame que alimenta a função `nrow()`. Essa última função conta o número de linhas de qualquer data frame ou matriz. O resultado é reportado pelo workspace: 4. Ou seja existe 4 linhas no data frame cars que a variável speed é igual a 18. Você pode (e deve) testar a função pelos passos descritos acima:

- digite cars e tecle Enter. Esse comando mostrará o data frame cars (que já vem instalado no R);
- digite: subset(cars, speed==18);
- digite a função completa novamente: nrow(subset(cars, speed==18)).

Outra vantagem de uma linguagem direcionada ao objeto é melhor explicada usando outro exemplo. Quando você calcula uma regressão simples com outro programa estatístico, como SAS ou Systat, você acaba com um monte de resultados na tela. Em contraste, quando você usa a função que calcula a regressão no R (`lm()`), a função retorna um objeto contendo todos os resultados (os coeficientes estimados, o erro padrão, resíduos, etc.). Você então pode escolher qual parte desse objeto é útil para você, e usá-la em outra função. Por exemplo:

```
> frenagem <- lm(speed~dist, data=cars)   # faz uma regressão entre speed em função de dist, do dataframe cars, e salva como frenagem.
> summary(frenagem)   # mostra os resultados da regressão os coeficientes estimados, o erro padrão, resíduos, etc.)
> names(frenagem)   # mostra diversos produtos da regressão, como os resíduos
> frenagem$residuals   # mostra os resíduos da regressão
> summary(frenagem)[8]   # retorna somente o r² da regressão
```

Você também verá que o estilo da linguagem R faz a programação ser mais fácil, porque oferece uma certa uniformidade para acessar os dados e resultados. Essa uniformidade confere ao R uma vantagem. A linguagem R é polimórfica, que significa que uma função pode ser aplicada a diferentes tipos de dados, mas o processamento ocorre de forma correta. Funções desse tipo são chamadas de funções genéricas. A função `summary()` é um exemplo de função genérica. Você pode usá-la para ver os resultados de uma regressão simples ou para criar um sumário estatístico dos próprios dados. Veja:

```
> summary(frenagem)  # retorna o resultado da regressão
> summary(cars)   # retorna diversas métricas sobre o dataframe cars
```

Criar Funções (programação)

Enquanto o R é muito útil como uma ferramenta para análise de dados, muitos usuários às vezes precisam crias suas próprias funções. Esta é uma das maiores vantagens do R. Além de ser um programa para análises estatísticas, o R é acima de tudo é uma linguagem de programação, com a qual podemos programar nossas próprias funções. Aprender a usar o R e não aprender a fazer programas básicos em R ainda assim será vantajoso, porém, aprender a programar será extremamente mais vantajoso para você.

As noções de programação são muito simples. Você precisa apenas montar um algoritmo, que inicialmente pode ser montado em claro e bom Português. O algoritmo é nada mais que uma sequência de instruções (regras, operações) que são definidas de maneira precisa e que possuem um fim específico.

Vamos ver como montar um algoritmo sem usar nenhuma linguagem de programação, apenas um "Pseudo-código" em português de quais são as instruções do nosso algoritmo.

Para começar, pense em um conjunto de 20 chaves e o nosso objetivo é descobrir qual chave abre uma determinada porta. Vamos ver então como seria um algoritmo para fazer essa tarefa.

Primeiro, precisamos definir as chaves, depois pegar uma dessas chaves, ver se ela se encaixa na fechadura e, se encaixar, ver se ela abre a porta.

Definindo as chaves:

Chaves = (C_1, C_2, C_3, C_4, C_5, C_7 ..., C_{20}) # Essas são as 20 chaves, veja que cada chave possui um índice C_i e que pode ser utilizado para identificar as chaves. Como são 20 chaves o i poderá ser i=1, 1=2 até i=20. Vamos iniciar nosso algoritmo:

Comece

Pegue a chave *i* e teste

Encaixa?

Se não, volte para o começo e passe para a próxima chave

Se sim, tente abrir

Abriu?

Se não, volte para o começo e passe para a próxima chave

Se sim, termine!

Conhecer a linguagem de programação pode ser difícil, mas criar um algoritmo é fácil pois basta pensar logicamente e saber o que quer fazer. Sempre que for escrever um algoritmo, crie um pseudo-código para "planejar" o seu código.

Exercício: Vamos para outro exemplo. Imagine uma prateleira com livros de diferentes tamanhos toda desorganizada e que você queira programar um robô para coloca-los em ordem decrescente de tamanho na prateleira. Agora crie um pseudocódigo para dar ordens ao robô de forma que o produto final seja uma prateleira com livros maiores na esquerda e os menores na direita. No final da apostila está uma resposta, dentre milhares de opções possíveis. Não olhe antes da resposta antes de tentar muitas vezes.

Sintaxe para escrever funções no R

A sintaxe básica é:

```
function(lista de argumentos){corpo da função}
```

A primeira parte da sintaxe (`function`) é a hora em que você diz para o R, "estou criando uma função".

A lista de argumentos é uma lista, separada por vírgulas, que apresenta os argumentos que serão avaliados pela sua função.

O corpo da função é a parte em que você escreve o "algoritmo" que será utilizado para fazer os cálculos que deseja. Esta parte vem entre chaves.

Ao criar uma função você vai querer salva-la, para isso basta dar um nome

```
minha.função<-function(lista de argumentos){corpo da função}
```

Criando uma função (function)

Comando function

Vamos ver como criar funções começando por uma função simples, que apenas simula a jogada de moedas (cara ou coroa). Neste caso a função terá dois argumentos (x e n). **x** será a "moeda" c("cara", "coroa") e **n** será o número de vezes que deseja jogar a moeda.

Para facilitar a criação de uma função, primeiro nós criamos objetos com o mesmo nome dos argumentos da função. Por exemplo, neste caso precisaremos criar um `x` e um `n`. O `x` será um objeto que represente a moeda e o `n` um objeto que represente o número de vezes que você deseja lançar a moeda. Então vamos criar um x que será composto pelos valore 1 e 2 (1 = Cara, 2= Coroa) e um n que indicará que deverão ser feitos 10 lançamentos:

```
> x<- c(1,2)
> n<-10
```

Agora vamos pensar em uma forma de fazer os lançamentos da moeda. A função `sample` é bem adequada, pois é possível fazer amostras aleatórias com reposição. Portanto

```
> sorteio<-sample(x,n,replace=TRUE)
> sorteio
```

Veja que este comando lança a moeda n vezes, portanto esse é o único comando que precisamos. Porem, isto que fizemos é apenas um comando e não criamos nenhuma função. Para criarmos a função precisamos indicar para o R que criaremos uma função.

Para criar esta função, faríamos um código conforme abaixo. Vamos dar o nome a esta função de `jogar.moeda`

```
> jogar.moeda<-function(x,n){  ## Não digite tudo na mesma linha
+ sorteio<-sample(x,n, replace=T)
+ return(sorteio)
+ }  # Fim da função
```

A primeira chave "{" indica o início do algoritmo da função e a outra indica o fim "}".

O comando `sample(x,n)` indica que desejamos amostrar x, n vezes (jogar a moeda n vezes). Com a função return() nós informamos qual é o resultado que desejamos retornar com esta função, que neste caso será os valores do “sorteio”

Agora vamos usar nossa função, mas primeiro vamos criar a "moeda", usando palavras em vez de números.

```
> moeda<-c("Cara","Coroa")
> jogar.moeda(moeda,2)
> jogar.moeda(moeda,10)
> jogar.moeda(moeda,1000)
```

Veja que jogando 1000 moedas ficou difícil saber quantas caras e quantas coroas saíram. Com a função `table`(), podemos descobrir isso facilmente.

```
> table(jogar.moeda(moeda,1000))
```

Também podemos modificar a nossa função de forma que ela já retorne os valores de forma tabulada. Vamos testar primeiro o código, ver se funciona e depois criamos a função:

```
> sorteio<-sample(x,n,replace=TRUE)# cria a amostragem
> tabela<-table(sorteio)# faz a tabulação do sorteio
> tabela
```

Agora, precisamos recriar a função informando que queremos retornar os resultados já de forma tabulada. Vamos chamar essa função de `sortear`, para diferenciar da anterior. Vamos mudar os nomes

dos argumentos também, em vez de **x,** vamos chamar de **objeto** e em vez de **n** vamos chamar de **n.vezes**.

```
> sortear<-function(objeto,n.vezes){  ## Não digite tudo na mesma linha
+ sorteio<-sample(objeto, n.vezes, replace=T)
+ tabela<-table(sorteio)
+ return(tabela)
+ }  # Fim da função
```

Agora vamos usar a função e ver se está funcionando ok

```
> sortear(moeda, 100)
```

Repita o comando da linha acima 3 vezes para confirmar que os valores mudam.

Veja que também podemos usar a função `jogar.moedas()` ou a função `sortear()` para jogar dados:

```
> dado<-1:6
> jogar.moeda(dado,2)
> table(jogar.moeda(dado,200))


> sortear(dado, 100)
```

Repare que esta função que criamos para jogar moedas é muito simples e nem é necessária, pois podemos jogar moedas sem criar esta função.

```
> sample(c("cara","coroa"),10,replace=T)
```

Fizemos este exemplo apenas para mostrar como criar uma função no R. Agora suponha que você não sabe qual função do R calcula a média, mas você sabe que a fórmula para calcular a média é:

$$média = \frac{\sum Y_i}{n}$$

Conhecendo a fórmula você pode criar sua própria função que calcula a média. Primeiro vamos ver quais argumentos serão necessários e qual código podemos usar para calcular a média. Para testar, vamos usar o conjunto de dados abaixo:

```
> teste<-c(1,4,7,l2,3)
```

Veja na formula que precisamos saber o somátorio dos valores:

```
> somatorio<-sum(teste)
```

E que precisamos saber o n, que é o número de observações:

```
> n<-length(teste)
```

Depois precisamos dividir o somatorio pelo n:

```
> somatorio/n
```

Veja que esta operação calcula corretamente a média. Use a função `mean()` do R para conferir.

Agora que temos o código, podemos criar a função. Repare que o único argumento necessario é aquele que indica os dados que queremos calcular a média. Note também. que podemos inserir comentários dentro da função para depois lembrar o foi feito:

```
> media<-function(dados){                #função chamada media
+ somatorio<-sum(dados)          #Calcula o somatório
+ n<-length(dados)               ## n é o número de observações
+ med<-somatorio/n               ## calcula a média
```

```
+ return(med)                    # return retorna os resultados calculados
+ } # Fim da função
```

Agora use a função criada, `media()` para calcular a média dos valores abaixo:

```
> valores<-c(21, 23, 25, 19, 10,1 ,30, 20, 14, 13)
> media(valores)
```

Use a função do R `mean()` para verificar se o cálculo foi feito corretamente.

Abaixo nós criaremos funções um pouco mais complicadas, mas a estrutura básica para criar uma função é exatamente como estas descritas acima.

Agora vamos ver uma forma de fazer o R repetir uma tarefa o número de vezes que quisermos. No R podemos fazer isso usando funções de controle de fluxo. A forma mais simples é usando o comando `for`.

O comando `for`

O comando `for` é usado para fazer loopings. Ou seja, o R inicia uma tarefa e ao terminar ele reinicia a mesma tarefa o número de vezes que quisermos. O comando `for` funciona da seguinte maneira:

```
"for(i in 1:n){comandos}"
```

Isso quer dizer que: para cada valor **`i`** o R vai calcular os comandos que estão entre as chaves `{comandos}`. O "`i in 1:n`" indica que os valores de `i` serão `i = 1` até `1 = n`. Ou seja, na primeira rodada do `for` o `i` será igual a 1, na segunda `i = 2`, e assim por diante até que `i = n`. Para salvar os resultados que serão calculados no `for` precisamos criar um objeto vazio que receberá os valores calculados. Criamos este objeto vazio assim:

```
> resu<-numeric(0)
> resu # este objeto é um vetor numérico que ainda está vazio
```

Agora vamos usar o `for` para elevar `i` valores ao quadrado:

```
> for(i in 1:5){                    ## o i será i = 1; i = 2; i = 3; i = 4; i = 5
+ resu[i]<-i^2                     ## Na primeira rodada o i = 1, então será 1^2
+ } # Fim do for (i)
> resu                              ## Para ver os resultados
```

Veja que este for não é necessário, pois poderíamos ter elevado esses números ao quadrado sem usar um for.

```
> (1:5)^2
```

Contudo, em diversos casos precisaremos usar um for para fazer uma tarefa especifica para cada valor. Mas antes de continuar, vamos fazer uma espécie de "filminho" mostrando o que o `for` faz:

Primeiro vamos fazer um gráfico com limites xlim =0:10 e ylim=0:10.

```
> plot(0:10,0:10, type="n")
```

Agora vamos usar o `for` para inserir textos no gráfico a cada passo do `for`:

```
> for(i in 1:9){
+ text(i,i, paste("Passo", i))
+ }
```

O R fez tudo muito rápido, de forma que não conseguimos ver os passos que ele deu. Vamos fazer novamente, mas agora inserindo uma função que retarde o R em 1 segundo. Ou seja, cada passo irá demorar 1 segundo.

```
> plot(0:10,0:10, type="n")
```

```
> for(i in 1:9){
+ text(i,i, paste("Passo", i))
+ Sys.sleep(1)                    ## retarda os passos em 1 segundo
+ }
```

Entendeu o que está sendo feito? No primeiro passo do for, o i é igual a 1, portanto apareceu o texto "passo 1" na coordenada x=1, y=1 do gráfico. No segundo passo o i é igual a 2 e aparece o texto "passo 2" na coordenada x=2, y=2, e assim por diante.

O `i` é apenas uma letra qualquer, podemos alterar o `i` para qualquer letra que quisermos, desde que essa letra seja usada consistentemente dentro dos comandos. Por exemplo, vamos substituir o `i` por `k`:

```
> plot(0:10,0:10, type="n")
> for(k in 1:9){
+ text(k,k, paste("Passo", k))
+ Sys.sleep(1)                    ## retarda os passos em 1 segundo
+ }
```

O `for` é um comando bastante utilizado em diversas funções e na simulação de dados. Fique atento ao uso do "`for`" nas funções seguintes e pergunte caso não tenha entendido.

A sequência de Fibonacci é uma sequência famosa na matemática (Braun and Murdoch 2007). Os dois primeiros números da sequência são [1, 1]. Os números subsequentes são compostos pela soma dos dois números anteriores. Assim, o terceiro número da sequência de Fibonacci é 1+1=2, o quarto é 1+2=3, e assim por diante. Vamos usar a função for para descobrir os 12 primeiros números da sequência de Fibonacci (Este exemplo foi retirado e adaptado de: Braun and Murdoch 2007 pp, 48).

```
> Fibonacci<-numeric(0)
> Fibonacci[c(1,2)]<-1    # o 1° e 2° valores da sequência devem ser = 1
> for (i in 3:12){  # 3 a 12 porque já temos os dois primeiros números [1,1]
+ Fibonacci[i]<-Fibonacci[i-2]+Fibonacci[i-1]
+ }

> Fibonacci
```

Diferença entre criar uma função e escrever um código

Existem milhares de formas de se trabalhar em R. Algumas pessoas preferem escrever pedaços de códigos e executá-lo quantas vezes seja necessário, enquanto outras preferem criar funções mais automatizadas, mas ambas produzindo o mesmo resultado. Em alguns casos vocês irão se deparar com códigos e em outros casos com funções. É preciso saber diferenciar os dois tipos, pois seu uso é levemente diferente e pode gerar confusões.

Vamos gerar um código que escolhe números para a mega sena e depois vamos criar uma função que faz a mesma coisa.

```
> njogos<-20 # quantos jogos queremos produzir
```

```
> numeros<-matrix(NA,6,njogos) # arquivo que irá receber números dos jogos
> for(i in 1:njogos){
+ numeros[,i]<-sample(1:60,6)
+ }
> numeros
```

Agora, caso você queira mudar o número de jogos (njogos), é preciso mudar o valor e rodar o código todo novamente.

Crie agora 50 jogos.

Vamos transformar nosso código em uma função e ver o que muda, apesar dos comandos serem quase os mesmos.

```
> megasena<-function(njogos){  # cria a função com nome de megasena
> numeros<-matrix(NA,6,njogos) # cria o arquivo que recebe os jogos
> for(i in 1:njogos){
> numeros[,i]<-sample(1:60,6)
> }
> return(numeros)
> }
> megasena(100)
```

O interessante em criar uma função, é que após ela estar funcionando você precisa mudar apenas um argumento, que nesse caso é o njogos.

```
> megasena(10)
> megasena(5)
```

Em geral, muitas pessoas criam um código, conferem se ele está funcionando e depois transformam o código em uma função para facilitar o processo de repetição.

Exemplos: Criar funções para calcular índices de diversidade

Agora vamos usar o `for` para fazer uma função mais complexa, que calcula o índice de diversidade de Shannon. A fórmula do índice de Shannon é:

$$Shannon\ \ H = -\sum P_i \ln P_i$$

onde P_i é o valor i em proporção e *ln* é o logaritmo natural de i.

O índice de Shannon é usado para criar um valor de diversidade para cada local. Esses valores podem ser utilizados para comparar/verificar quais locais são mais diversos e quais são menos diversos. Verifique a literatura para obter mais informações, aplicações e validade desses índices de diversidade.

Abaixo segue o código da função para calcular o índice de Shannon pronto, mas antes de criar a função por completo, teste os comandos linha por linha. `dados` será um nome genérico para o argumento da função, que servirá para ser usado para representar qualquer conjunto de dados. Para testar a função linha por linha, use os dados de teste abaixo:

```
> dados<-c(20,43,21,14,34,54)
```

Para ir testando as linhas é só digitar os comandos separadamente, por exemplo:

```
> prop<-dados/sum(dados)
> prop
```

Caso você queira testar os passos do `for{}` um de cada vez, sem que tudo seja feito de uma vez, crie um objeto chamado `i`**:**

```
> i=1
```

E teste as linhas de comando dentro do `for{}`

```
> resu[i]<-if(prop[i]>0){prop[i]*log(prop[i])
```

só irá funcionar se você já tiver criado os objetos prop e resu, que aparecem antes do `for{}`

Função para calcular o índice de Shannon:

```
> shannon<-function(dados){      #Criamos a função shannon
+ prop<-dados/sum(dados)         ## calcula as proporções
+ resu<-numeric()     # objeto numérico "vazio" que receberá os valores do for
+ n<-length(prop)
+ for(i in 1:n){
+ resu[i]<-if(prop[i]>0){prop[i]*log(prop[i])}
+ else{0}  # veja abaixo sobre o uso do  if  e do  else
+ }  # Fim do for (i)
+ H<- -sum(resu)
+ return(H)
+ }    # Fim da função
```

Nota: Durante o cálculo do índice de Shannon precisamos pular o calculo quando o valor de proporção é zero, pois `log(0)` não existe (-inf). Por isso usamos o `if` e o `else`, ou seja, se (`if`) o valor for maior que zero faça o calculo, se não (`else`) o valor será 0.

```
> log(0)      # veja o resultado do log de zero.
```

Agora vamos usar a função `shannon` para calcular o índice de Shannon dos dados apresentados no livro da Anne Magurran página 45, tabela 2.1 (Diversidad Ecológica y su Medición, 1988). Os dados são:

```
> magur<-c(235,218,192,0,20,11,11,8,7,4,3,2,2,1,1)# O valor 0 foi alterado propositalmente, no livro o valor é 87.
> shannon(magur)
```

Esta função que criamos para calcular o índice de diversidade de Shannon calcula o índice de apenas um local por vez. Por exemplo, se quisermos calcular o índice de Shannon para o riacho 1 dos dados de borrachudos podemos usar:

```
> spp.m<-as.matrix(spp) #
```

Os dados `spp` são do tipo `data.frame`, por isso precisamos transformar em matrix para podermos acessar apenas os vetores referentes a cada linha (ou riacho).

```
> shannon(spp.m[1,])
```

Ou para o riacho 3:

```
> shannon(spp.m[3,])
```

Daqui em diante, da mesma forma que anteriormente, vá testando cada linha de comandos individualmente para ver o que está sendo feito em cada passo das funções a seguir. Lembre-se de criar um arquivo de teste. Por exemplo: `dados<-spp`, para testar usando os dados de borrachudos da Chapada Diamantina.

Agora vamos criar uma função que calcula o índice de Shannon de vários locais ao mesmo tempo. Neste caso teremos que usar o comando `for` duas vezes.

Função para calcular o índice de Shannon para diversos locais:

```
> shan<-function(dados){ # nome de shan para diferenciar da função acima
+ nl<-nrow(dados)     # número de locais (linhas)
+ nc<-ncol(dados)
+ H<-numeric()        # variável vazia que receberá os valores H de cada local
+    for(i in 1:nl){
+    prop<-dados[i,]/sum(dados[i,]) ## dados do local i em proporção
+    resu<-numeric()
+         for(k in 1:nc){
+         resu[k]<-if(prop[1,k]>0){prop[1,k]*log(prop[1,k])}
+         else{0}
+         }# Fim do for (k)
+    H[i]<--sum(resu)
+    } # Fim do for (i)
+ return(H)
+ } # Fim da função
```

Repare que os espaços deixados acima facilitam a visualização hierárquica dos comandos. Ou seja, veja que o segundo `for`, o do `k`, está inserido dentro do primeiro `for`, o do `i`. Desta forma também é possível ver onde exatamente uma chave está fechando um comando `for`.

Agora vamos calcular o índice de Shannon dos dados de borrachudos (spp).

```
> shan(spp)
```

Para ver se seus cálculos estão corretos compare os resultados usando a função `diversity()` do pacote `vegan`.

```
> library(vegan)
```

> diversity(spp,"shannon",MARGIN=1) # O argumento MARGIN indica se você quer fazer os cálculos por linhas ou por colunas. Use 1 para linhas e 2 para colunas.

Exemplos: Criar funções para calcular matrizes de similaridade

As medidas de similaridade (resemblance measures) são utilizadas, por exemplo, para calcular quão similares dois, ou mais locais, são em relação à sua composição de espécies. As comparações são feitas par-a-par. Por exemplo, se você possui amostras em cinco locais, a similaridade entre todos os locais será calculada, criando uma tabela conforme a seguinte:

	Local 1	Local 2	Local 3	Local 4	Local 5
Local 1	1				
Local 2	0.2	1			
Local 3	0.23	0.6	1		
Local 4	0.34	0.98	0.43	1	
Local 4	0.3	0.4	0.8	0.7	1

Repare que na diagonal principal todos os valores são iguais a 1, pois a similaridade de um local com ele mesmo é total. Acima da diagonal os valores não são mostrados, pois são iguais aos valores abaixo da diagonal. Ou seja, a similaridade do local 1 com o local 3 é igual a similaridade do local 3 com o local 1, por exemplo.

Na ecologia um dos índices de similaridade utilizado é o índice de Jaccard:

$$S(X1, X2) = \frac{a}{a+b+c} \qquad \text{equação 2.10 (Legendre e Legendre 1998, pp 256)}$$

Onde, no caso de uma matriz de espécies: a = número de espécies compartilhadas pelos locais 1 e 2; b = número de espécies presentes apenas no local 1; c = número de espécies presentes apenas no local 2. Veja abaixo como criar uma função para calcular o índice de similaridade de Jaccard. Não se esqueça de testar os comandos linha por linha e ver o que está sendo calculado.

```
> jaccard<-function(dados.spp){
+    # Vamos inserir uma mensagem de erro, caso dados não sejam de presença-ausência
+ if(any(dados.spp>1))
+ stop("Erro: é preciso fornecer uma tabela com dados de   +
presença e ausência")
+ n<-nrow(dados.spp)  # número de locais
+ jac<-matrix(NA,n,n)  ## Matriz que receberá os valores de similaridade
+ colnames(jac)<-rownames(dados.spp) ## Dar os nomes dos locais
+ rownames(jac)<-rownames(dados.spp) ## Dar os nomes dos locais
+    for(i in 1:n){
+          for(j in 1:n){
+          # número de espécies presentes em ambos locais, i e j ( a )
+          a<-sum(dados.spp[i,]==1 & dados.spp[j,]==1)
+          # número de espécies presentes apenas no local i ( b )
+          b<-sum(dados.spp[i,]==1 & dados.spp[j,]==0)
+          # número de espécies presentes apenas no local j ( c )
+          c<-sum(dados.spp[j,]==1 & dados.spp[i,]==0)
+          jac[i,j]<- a /(a+b+c)   ## Fórmula de Jaccard
+          }
+    }
+ return(as.dist(jac))
+ }

> jac<-jaccard(pres.aus)    ## os cálculos demoram um pouco
> jac
```

```
> jac.vegan<-1-vegdist(pres.aus,"jaccard")  ## Cálculo do Jaccard no vegan. OBS: O R não calcula similaridades e sim distâncias (dissimilaridades), por isso usamos o 1-
> jac.vegan
> cbind(jac,jac.vegan)  # coloque os valores lado a lado e compare. Está correto?
```

Outro índice de similaridade muito utilizado em ecologia, mas para dados de abundância é o índice de Bray-Curtis:

$$D(X1, X2) = \frac{\sum|y_{1j} - y_{2j}|}{\sum(y_{1j} + y_{2j})}$$ Equação 7.57 (Legendre & Legendre 1998, pp 287)

Onde: y_{1j} é a abundância da espécie j no local 1 e y_{2j} é a abundância da espécie j no local 2. Esta fórmula calcula a dissimilaridade e não similaridade (diferente do exemplo anterior do Jaccard). Então vamos criar uma função para calcular o índice de similaridade de Bray-Curtis.

```
> bray<-function(dados.spp){
+ n<-nrow(dados.spp)
+ BrayCurtis<-matrix(NA,n,n)
+     for(i in 1:n){
+         for(j in 1:n){
+         numerador<-sum(abs(dados.spp[i,]-dados.spp[j,]))
+         denominador<-sum(dados.spp[i,]+dados.spp[j,])
+         BrayCurtis[i,j]<- numerador/denominador
+         }
+    }
+ return(as.dist(BrayCurtis))
+ }
```

Vamos usar nossa função bray usando os dados de borrachudos e comparar com o resultado obtido usando a função `vegdist` do pacote `vegan`.

```
> bra<-bray(spp)
> bra.vegan<-vegdist(spp,"bray")
> cbind(bra,bra.vegan)
> bra==bra.vegan
```

Exercícios de criar funções

1 - Crie uma função para calcular o índice de diversidade de Simpson dos dados de borrachudos da Chapada Diamantina (spp):

$D = 1 - \sum p_i^2$ onde p_i é o valor i em proporção

Confira o resultado usando a função `diversity` do pacote `vegan`:

```
> library(vegan)
```

```
> diversity(spp,"simpson",MARGIN=1)    # O argumento MARGIN indica se você
```
quer fazer os cálculos por linhas ou por colunas. Use 1 para linhas e 2 para colunas.

2 - Crie uma função para calcular a distância (dissimilaridade) de Hellinger dos dados de abundância de borrachudos. A fórmula é:

$$D(X1, X2) = \sqrt{\sum \left[\sqrt{\frac{y_{1j}}{y_{1+}}} - \sqrt{\frac{y_{2j}}{y_{2+}}} \right]^2}$$ Equação 7.55 (Legendre & Legendre 1998, pp 286)

Onde: y_{1j} são os valores de abundância de cada espécie j no local 1; y_{2j} são os valores de abundância de cada espécie j no local 2; y_{1+} é a soma total das abundâncias do local 1; y_{2+} é a soma total das abundâncias do local 2.

Não se assuste com a fórmula, tente dividi-la em partes dentro da função.

Para calcular a distância de Hellinger no R e conferir com resultado da sua função use:

```
> hel.vegan<-vegdist(decostand(spp,"hellinger"),"euclid")
```

3 – Desafio: Calcule o índice de Shannon usando apenas 3 linhas de comando e o índice de Simpson usando apenas uma linha de comandos. Não será necessário criar nenhuma função.

APENDICE 1)

Como organizar Dados!!!

Os dados ou informações são a chave para a pesquisa científica empírica. Eles podem ser menos importantes para os pesquisadores teóricos, mas assim que os pesquisadores começam a testar as predições de uma teoria, os dados novamente recebem destaque. Apesar da sua importância a maioria dos ecólogos e cursos de ecologia tradicionalmente reservam pouco (ou mais comumente nenhum) tempo para discutir e ensinar técnicas de coleta, arquivamento e manejo de dados. Isso é curioso, dado que muitos dados de campo em ecologia são extremamente custosos para serem coletados. Precisamos tratar os dados como os administradores de empresas tratam os dados deles: **informação é dinheiro (ou informação é tudo)!**

Abaixo segue os oito passos básicos para uma boa coleta e arquivamento de dados organizados.

1) Antes de coletar os dados, determine quais são os objetos e quais são os atributos que serão estudados. A tabela que vai receber os dados deve ter os objetos como linhas e os atributos como colunas. **NUNCA** sumarize os dados originais!!!

Exemplo: Estudo da distribuição de tamanhos de 4 espécies de mamíferos em 2 reservas da Amazônia

OBJETOS ⇨	ATRIBUTOS ⇩			
	Reserva	Amostra	Especie	Tamanho
Indivíduo #1	RAP	1	Lala_lal	2.5
Indivíduo #2	RAP	2	Lela_lel	4.5

2) Quando alguns atributos vão se repetir muitas vezes em uma página da planilha, eles podem ser transformados em cabeçalhos, mas devem voltar a ser colunas da tabela quando esta for digitada no banco de dados.

No campo:

Reserva: A

Data: 20/02/2002

Observador: João Silva

	Amostra	Especie	Tamanho
Indivíduo #1	1	Lala_lal	2.5
Indivíduo #2	2	Lela_lel	4.5

No banco de dados:

	Data	Observador	Reserva	Amostra	Especie	Tamanho
Indivíduo #1	20/02/2002	João Silva	RAP	1	Lala_lal	2.5
Indivíduo #2	20/02/2002	João Silva	RAP	2	Lela_lel	4.5

3) Cada objeto deve ter um identificador único, que não se repete nunca. Este pode ser um número, um nome, ou uma combinação de atributos, desde que as combinações nunca se repitam.

4) Ao criar a tabela, defina claramente qual tipo de dado pode ser aceito em cada coluna da tabela (p.ex. numéricos, alfa-numéricos, data). Isso vai ajudar a evitar erros de digitação.

Atributo	Tipo de dado
Data	Data
Observador	Alfa-numérico (texto)
Reserva	Alfa-numérico (texto)
Amostra	Alfa-numérico (texto)
Espécie	Alfa-numérico (texto)
Tamanho	Numeric com 1 casa decimal

5) Confira os dados após a digitação. Preferencialmente, isso deve ser feito por 2 pessoas, 1 lendo e a outra conferindo.

6) Defina claramente o que cada nome de atributo significa, inclusive especificando as unidades de medida e o tipo de dado.

Atributo	Definição	Tipo de dado
Data	Dia/mes/ano em que a observação foi realizada	Data
Observador	Nome da pessoa que realizou o avistamento	Alfa-numérico (texto)
Reserva	Nome da Reserva, sendo RAP: Reserva Apoio à Pesquisa; RNV:	Alfa-numérico (texto)
Amostra	Número da trilha de observação, de acordo com a nomenclatura do projeto xxxx	Alfa-numérico (texto)
Espécie	Nome do Gênero e espécie, separados por underline.	Alfa-numérico (texto)
Tamanho	Comprimento rostro-anal, em cm, medido com paquímetro de precisão de 0.1 mm	Numerico com 1 casa decimal

7) Antes de coletar os dados escreva os metadados que descrevem a coleta. Leve esta descrição para o campo e atualize-a caso haja qualquer alteração do que foi previsto. Os metadados devem conter pelo menos:

- Quem fez o estudo (inclusive as funções de cada membro da equipe)
- Onde
- Quando
- Como (métodos e instrumentos)
- Nome do arquivo de dados

8) Salve os dados e os metadados em uma mesma pasta do seu computador. É importante que o arquivo de metadados inclua o nome do arquivo de dados ao qual ele se relaciona.

Relembrando. Se seu objeto (sua variável de interesse, seu objeto de estudo) não está individualizada em cada linha da sua tabela, alguma coisa está errada… Use as colunas para cada atributo (variável que vc quer relacionar com seu objeto de estudo). Nunca esqueça de organizar os metadados sobre os dados coletados. O ideal é ter um rascunho dos metadados, antes de sair para coletar os dados!

APENDICE 2

Estatística

Conforme dito na introdução desta apostila, o objetivo proposto aqui não é o de ensinar estatística. Nesta parte iremos ver apenas comandos básicos para realizar algumas das análises mais corriqueiras usadas em análise de dados. Como exemplo, usaremos alguns conjuntos de dados que acompanham alguns pacotes do R. Também usaremos dados e exemplos utilizados no livro Princípios de Estatística em Ecologia (Gotelli e Ellison 2010, "A Primer of Ecological Statistics", 2004). Todos os dados utilizados no livro estão disponíveis no site:

http://harvardforest.fas.harvard.edu/personnel/web/aellison/publications/primer/primer.html

Estatística descritiva

Em geral, as estatísticas descritivas são divididas em dois tipos de medidas, as medidas de dispersão e as medidas de posição. As medidas de posição visam representar a posição onde a maioria dos dados se encontra. A média, a mediana e a moda são as medidas de posição mais comuns. Já as medidas de dispersão representam a variabilidade nos dados. As medias mais comum são a variância, o desvio padrão e o erro padrão.

Nesta sessão usaremos os dados apresentados na Tabela 3.1 do livro. Esses dados são medidas do comprimento do espinho tibial de aranhas Linyphiidae.

Primeiro vamos importar os dados

```
> comp<-read.table("Linyphiidae.txt",header=T)
```

Média aritmética

A media aritmética é calculada somando-se as observações e dividindo-se pelo número de observações. No R a função para calcular a média é a função `mean()`.

```
> mean(comp)  #calcula a média do comprimento dos espinhos
```

Lembre-se que para calcular as médias das linhas ou das colunas de uma tabela você pode usar as funções `rowMeans()` e `colMeans()`, respectivamente.

Exercício: Escreva um código ou uma função para calcular a média geométrica $MG_Y = e^{\left[\sum_{i=1}^{n} \ln(Y_i)\right]}$ e a média harmônica $H_i = \frac{1}{\frac{1}{n}\sum\frac{1}{Y_i}}$ dos dados do comprimento de espinhos.

Variância e o Desvio Padrão

A variância amostral $s^2 = \frac{1}{n-1}\sum(Y_i - \bar{Y})^2$ e o desvio padrão amostral $s = \sqrt{\frac{1}{n-1}\sum(Y_i\bar{Y})^2}$ são as medidas de dispersão mais comuns. No R usamos as funções `var()` e `sd()` para calcular essas medidas.

```
> var(comp)
> sd(comp)
```

Exercício: Escreva seu próprio código, ou uma função, para calcular a variância e o desvio padrão amostral.

Quartis e mediana

O R possui uma função que faz um resumo dos dados apresentando seis medidas de posição que descrevem os dados (os valores mínimo e máximo, a média e a mediana, o primeiro e o terceiro quartis). A função é chamada `summary()`.

```
> summary(comp)
```

Estatística univariada

Os testes e estatísticas univariadas mais conhecidos e comumente aplicados são as regressões, testes-t e anovas. Embora haja uma infinidade de nomes e derivações para estas análises (e.g. ancova, anova em blocos, anova de medidas repetidas, teste-t pareado…), todas são apenas modelos lineares. Veremos que embora haja uma função para fazer teste-t (`?t.test`) e outra para fazer anova (`?aov`) todas essas análises podem ser feitas usando apenas uma função, chamada `lm()`, que significa "linear models". Portanto, **não se preocupe com os nomes de análises**, pois são todos modelos lineares. **Preocupe-se apenas em especificar o seu modelo corretamente**.

Em geral, a maioria das análises é feita especificando-se o modelo que se deseja testar. No R, para especificar um modelo é preciso usar a notação de formulas. Por exemplo, para um modelo de regressão com uma variável resposta **Y** (ou dependente) e uma variável preditora **X** (ou independente) a notação de formula a ser usada no R é: Y ~ X. Onde Y é a variável resposta e X é a variável preditora. O ~ ("til") significa "em relação a" ou "modelado por". Ou seja, **Y em relação a X**, ou **Y modelado por X**. Caso haja duas variáveis preditoras em seu modelo a formula será **Y~X1+X2.** Isso medirá o efeito das duas variáveis preditoras. Caso queira avaliar também a interação entre X1 e X2 a formula será **Y~X1*X2.** Veremos outras possibilidades mais adiante.

Para facilitar o entendimento da abordagem usando modelos lineares iremos começar por análises de regressão.

Regressão Linear Simples

A regressão linear simples é utilizada para analisar relações entre variáveis contínuas. Para exemplificar como fazer uma regressão no R vamos usar dados do número de espécies de plantas em diversas ilhas de Galápagos em relação ao tamanho das ilhas (figura 8.5 e 8.6 do livro de Gotelli e Ellison). Como feito no livro, usaremos os dados transformados em log10 da riqueza de espécies e o log10 da área das ilhas para fazer a regressão. Para isso importe o arquivo de dados galapagos.txt.

Para fazer a regressão no R a função é `lm()`, para linear models. Mais adiante vamos ver que testes-t, anovas tambem são todos modelos lineares e que podem ser analisados usando apenas a função lm().

```
> galap<-read.table…
> riqueza<-galap[,3]
> area<-galap[,2]
> plot(area,riqueza)  # Figura 8.5 do livro
```

Note que a relação não é linear e que um ponto assemelha-se a um outlier. Para linearizar os dados e reduzir o efeito do outlier tranformamos os dados em log.

```
> riqueza.log<-log10(riqueza)
> area.log<-log10(area)
> plot(area.log,riqueza.log) # Figura 8.6 do livro
```

Agora vamos fazer a regressão usando a função `lm()`.

```
> resultado<-lm(riqueza.log ~ area.log)
> resultado
```

Veja que o resultado da regressão mostra os coeficientes a (intercepto) e b (inclinação) da regressão. Para ver mais detalhes sobre o resultado da regressão precisamos usar a função `summary()`.

```
> summary(resultado)
```

Agora, alem dos coeficientes da regressão, também são apresentadas outras estatísticas, como os valores de t, probabilidades, os graus de liberdade e o r^2.

Para apresentar os resultados da mesma forma que Gotelli e Ellison apresentam no livro (tabela 9.2) precisamos pedir uma tabela de anova do resultado da regressão. Para isso use:

```
> summary.aov(resultado)
```

Para inserir a linha de tendência da regressão use a função `abline()`.

```
> plot(area.log,riqueza.log)
> abline(resultado)
```

Regressão Múltipla

A regressão linear com uma única variável preditora *X* pode ser facilmente estendido a duas ou mais variáveis preditoras (regressão múltipla). O exemplo dado no livro de Gotelli e Ellison sobre quando duas ou mais variáveis preditoras são medidas em cada réplica. Por exemplo, em um estudo da variação na densidade de espécies de formigas em florestas de pântanos da Nova Inglaterra, foram medidas a latitude e a altitude em cada sítio amostral. Essas duas variáveis podem ser usadas em uma regressão múltipla. Para fazer, importe o conjunto de dados formigas.txt.

```
> formigas<-read…
> lat<-formigas[,1] # latitude
> alt<-formigas[,2]   # altitude
> dens.formig<-formigas[,3] # densidade de formigas

> lm(log10(dens.formig) ~ lat+alt)

> reg.mult<-lm(log10(dens.formig) ~ lat+alt)
> reg.mult
> summary(reg.mult)
> summary.aov(reg.mult)
```

Caso queira ver o gráfico dos parciais desta regressão múltipla você precisará do pacote “car” e usar a função `avPlots()`.

```
> avPlots(reg.mult)
```

Teste-t e teste-t pareado

O teste-t é utilizado para comparar amostras retiradas de duas populações de dados. No teste-t temos duas variáveis, uma contínua que é a variável resposta e uma categórica que é usada como variável preditora. Para explicar como fazer um teste-t no R usaremos dados de um estudo que avalia a se a taxa de decomposição de folhas em riachos e a abundância de Chironomidae* são afetados pela presença de peixes e camarões (Landeiro et al. 2008). Para isso foi montado um experimento em que peixes e camarões foram excluídos de forragear em determinados pacotes de folhas colocados em 10 locais dentro de 4 riachos. Em cada local foram colocadas duas cercas elétricas, uma ligada e outra não. Desta forma os peixes e camarões tinham acesso apenas às folhas da cerca desligada. Após 18 dias os pacotes de folhas

*Chironomidae são larvas de mosquitos com desenvolvimento aquático que vivem no substrato de riachos (rochas, folhas, etc.)

foram removidos, as larvas de chironomidade foram contadas e o peso seco das folhas remanescentes foi medido.

Para fazer os exemplos abaixo importe o arquivo de dados chamado “chiro.txt”.

```
> dados<-read.table…
> t.test(dados$chiro ~ dados$trat)
```

Veja que o resultado desta analise mostra o valor de t (estatística do teste), os graus de liberdade (df) e o valor de p (significância). Alem disso, o resultado do teste ainda mostra as médias para cada grupo.

Note que existe um efeito significativo de peixes e camarões sobre a abundância de chironomidae, onde a média no grupo C (controle) é bem menor que no grupo E (exclusão)

```
> t.test(dados$degrad ~ dados$trat)
```

Em relação a taxa de decomposição (porcentagem de peso remanescente) note que não há diferença significativa entre os tratamentos.

Porem, como o delineamento experimental foi feito de forma pareada, esses dados são melhor analisados com um teste-t pareado. Para fazer o teste-t pareado basta inserir o argumento paired=TRUE.

```
> t.test(dados$chiro ~ dados$trat, paired=TRUE)
> t.test(dados$degrad ~ dados$trat, paired=TRUE)
```

Note que a estatística do teste-t pareado não é baseada na média dos tratamentos, e sim na diferença entre os pares de tratamentos. Veja que agora os dois testes são significativos. Ou seja, os peixes e camarões possuem um efeito negativo sobre a abundância de chironomidae e sobre a taxa de decomposição de folhas.

Conforme dito anteriormente, vamos analisar esses dados usando a função `lm()`. O teste-t é apenas uma análise onde uma variável contínua é predita/modelada por uma variável categórica com dois níveis (neste exemplo os tratamentos Controle e Exclusão). O modelo para um teste-t usando a função `lm()` seria especificado da mesma forma como anteriormente dados$chiro ~ dados$trat.

Portanto:

```
> resu<-lm(dados$chiro~dados$trat)
> resu
> summary(resu)
> summary.aov(resu)
```

Repare que todos os resultados são os mesmos, embora os resultados obtidos com `lm()` são mais completos. Faça o mesmo para analisar os dados de degradação das folhas.

Agora vamos especificar um modelo linear que seja equivalente ao teste-t pareado usado acima. O teste-t pareado é analisado em blocos, ou seja, alem do tratamento precisamos informar o bloco (O local onde os pares estão). O modelo linear será dados$chiro~dados$trat+dados$local

```
> resu.par<-lm(dados$chiro~dados$trat+dados$local)
> summary(resu.par)
> summary.aov(resu.par)
```

Veja que os resultados são os mesmos obtidos com `t.test(, paired=T)`. Mas agora também podemos avaliar o efeito do bloco. Faça a mesma análise para os dados de degradação das folhas.

Análise de Variância (Anova)

Essencialmente, a análise de variância é igual ao teste-t, porem a variável categórica da anova possui mais de dois níveis.

Vamos fazer uma anova usando dados hipotéticos apresentados na Tabela 10.1 do livro do Gotelli. Os dados são:

```
> resposta<-c(10,12,12,13,9,11,11,12,12,13,15,16)
>   preditor<-c("n.m","n.m","n.m","n.m","c","c","c","c","t","t","t","t")
```

As siglas significam, n.m = "não manipulado", c = "controle" e t = "tratamento"

Para fazer a anova use:

```
> resu.aov<-aov(resposta~preditor)
> summary(resu.aov)
```

Para fazer as análises usando a função lm basta trocar pela função `lm()`.

```
> lm(resposta~preditor)
> summary.aov(lm(resposta~preditor))
```

Árvore de regressão

A árvore de regressão é uma alternativa interessante, ou um complemento, à análise de regressão múltipla. Como na regressão, na árvore de regressão a variável resposta é uma variável contínua, porém as variáveis preditoras podem ser categóricas e/ou contínuas.

No R, as árvores de regressão podem ser construídas usando o pacote `rpart`. Para exemplificar o uso de uma árvore de regressão vamos usar os dados `mite` e `mite.env` do pacote `vegan`. Iremos calcular a riqueza de espécies presentes no arquivo `mite` e relacionar com as variáveis ambientais do conjunto `mite.env`.

```
> library(vegan)
> library(rpart)
> data(mite); data(mite.env)
> riq<-rowSums(decostand(mite,"pa")) # 'pa' transforma a variável em presença ou ausência
> rpart(riq ~., data=mite.env)
```

No modelo acima o "`.`," indica que todas as variáveis do conjunto `mite.env` serão utilizadas como preditores. Seria o equivalente a escrever o modelo da seguinte forma: riq~SubsDens+WatrCont+Substrate+Shrub+Topo. Para ver o significado das variáveis entre no help que descreve o conjunto de dados.

```
> ?mite.env
```

Para maiores detalhes sobre os métodos de árvores de regressão veja De'Ath & Fabricius (2000). Agora vamos salvar o resultado e ver o *output* da análise.

```
> regtree<-rpart(riq ~., data=mite.env)
> summary(regtree)
```

Para fazer o gráfico veja o help abaixo.

```
> ?plot.rpart
> plot(regtree)
> text(regtree)
```

Note que parte do gráfico não aparece. Para corrigir, precisamos definir a margem do gráfico.

```
> plot(regtree,margin=0.03)
> text(regtree)#  se o problema persistir, mude o valor de  margin
```

Estatística multivariada

Transformações e padronizações de dados

Diversas transformações e padronizações nos dados podem ser realizadas de acordo com o tipo de dados e objetivo das análises. Para maiores informações sobre o uso adequado dessas transformações veja (Legendre and Gallagher 2001; Legendre and Legendre 1998). No R a função `decostand` do pacote `vegan` faz diversos tipos de transformações. As transformações mais comuns são:

total: divide pelo total da linha ou da coluna (default MARGIN = 1, que é por linha, MARGIN = 2 é por coluna)

max: divide pelo valor máximo da linha ou da coluna (default MARGIN = 1)

freq: divide pelo valor máximo da linha ou da coluna e multiplica pelo pelo número de itens diferentes de zero (default MARGIN = 2).

normalize: torna a soma dos quadrados das linhas ou das colunas igual a 1 (default MARGIN = 1).

range: padroniza os valores de forma que a amplitude varie de 0 a 1

standardize: Coloca o x em escala com média zero e variância 1 (default MARGIN = 2).

pa: Transforma x em presença e ausência (0/1).

chi.square: divide pelo total das linhas e pela raiz quadrada da soma das colunas (default MARGIN = 1).

hellinger: raiz quadrada do método "total"

log: Transformação logaritmica conforme sugerido por Anderson et al. (2006): *log_b (x) + 1* para *x > 0*, onde b é a base do logaritmo; zeros são mantidos como zeros. Bases maiores dão menos peso para quantidades e mais peso para presenças

Mais detalhes sobre a aplicação e o significado dessas transformações podem ser encontrados em {Legendre, 2001 815 /id&. O pacote `vegan` possui a função `decostand()` que faz todas essas transformações.

```
> decostand(spp,"total",MARGIN=1)
```

Use a função `decostand` para fazer todas as transformações acima usando os dados das espécies de borrachudos da chapada diamantina.

Índices de associação/similaridade/dissimilaridade/distância

A primeira coisa a notar no R em relação aos índices de similaridade é que o R trabalha basicamente com índices de dissimilaridade ou distância (o complemento da similaridade). Os índices de dissimilaridade mais comuns são o Euclideano, Bray-Curtis, Jaccard e Sorensen, dentre outros. No R existem duas funções principais para calcular dissimilaridades/distância. A função `?dist` e a função `?vegdist`, que é uma função do vegan. Para exemplificar como calcular essas matrizes de dissimilaridade/distancia vamos usar o conjunto de dados `data(mite)` e o conjunto `data(mite.xy)`. O conjunto `mite` possui 35 especies coletadas em 70 locais e o conjunto `mite.xy` possui as coordenadas geográficas dos 70 locais amostrados.

Vamos começar pelo índice mais básico, o Euclideano. A distância Euclideana geralmente é usada para calcular a distância entre os locais amostrados. Usando o conjunto `mite.xy` vamos calcular a distancia entre os 70 locais amostrados.

```
> data(mite.xy)
> vegdist(mite.xy,"euclidean")# Não se esqueça de carregar o library(vegan).
```

Veja que a matriz é muito grande e de difícil visualização. Apenas como exemplo, vamos calcular apenas a distância entre os cinco primeiros locais.

```
> dist(mite.xy[1:5,])
```

Veja, por exemplo, que a distância do local 1 até o local 2 é de 0.8, do local 3 ao 4 é 0.28...

Agora, vamos usar os dados “mites” para calcular a dissimilaridade na composição de espécies usando diversos índices.

```
> data(mites)
> vegdist(mites, "bray")#
> vegdist(mites, "jaccard")#
> vegdist(mites, "manhattan")#
> vegdist(mites, "canberra")#
```

Veja no help `?vegdist` outras possibilidades. Detalhes sobre o uso e aplicações dos diversos índices podem ser encontrados no capitulo 7 do livro Numerical Ecology (Legendre & Legendre 1998).

Uma aplicação interessante que podemos fazer aqui é verificar se a dissimilaridade entre as comunidades aumenta com a distância geográfica. Para isso, basta plotar a matriz de distancia contra a matriz de dissimilaridade.

```
> plot(vegdist(mite.xy,"euclidean"),vegdist(mite,"bray"))
```

Repare que há uma tendência de as comunidades mais distantes serem mais dissimilares. Geralmente, os estudos chamam essa relação de “distance decay of similarity” (decaimento da similaridade com a distância). Para avaliar a similaridade, em vez da dissimilaridade basta usar 1-dissimilaridade, desde que o índice que você esteja usando varie de zero a 1, como é o caso do índice de Bray-Curtis.

```
> plot(vegdist(mite.xy,"euclidean"),1-vegdist(mite,"bray"))
```

Note que agora a interpretação é diferente, conforme a distancia aumenta, as comunidades tendem a ser menos similares. Mais adiante veremos como testar essas relações usando, por exemplo, o teste de Mantel.

Ordenações:

Iremos usar os dados que vem no R para fazer as ordenações. Em geral, a maior parte das ordenações feitas em ecologia pode ser feita usando o pacote vegan. O livro mais consultado sobre estatísticas multivariadas é o “Numerical Ecology” de Pierre Legendre e Louis Legendre (1998). Recentemente, Daniel Borcard, François Gillet e Pierre Legendre publicaram o “Numerical Ecology with R” que pode ser uma boa referencia sobre como fazer análises multivariadas no R.

Análise de componentes principais (PCA)

Existem diversas funções no R que fazem PCA. Aqui nós usaremos a função `prcomp()`. Em geral, dois tipos de PCA são feitas, a PCA de covariância e a PCA de correlação. Para exemplificar iremos usar os dados de variáveis químicas do solo amostradas em 24 locais. O conjunto de dados vem com o vegan e chama “varechem”. Para usar esses dados digite `data(varechem)` no R.

Antes de tudo, dê uma olhada no help da função, `?prcomp`. Veja que é possível especificar o primeiro argumento em forma de formula, mas o mais comum é usar o segundo exemplo, onde o primeiro argumento é “x”. Esse “x” pode ser uma matriz ou uma data frame que contem os dados que será usados na pca. No nosso exemplo, o “x” será a matriz de dados “varechem”.

Primeiro vamos fazer uma PCA de covariância.

```
> resu.pca<-prcomp(varechem)
> resu.pca # Mostra os desvios dos componentes principais e os loadings.
> summary(resu.pca)# Mostra a porcentagem de variância capturada por cada eixo.
```

Para salvar os scores da PCA (os eixos) use:

```
> resu.pca$x
```

Para salvar os loadings use:

```
> resu.pca$loadings
```

Para opções gráficas veja `?biplot.prcomp` e `?screeplot`.

```
> biplot(resu.pca) #plota os scores dos locais e
> screeplot(resu.pca)
```

Agora vamos fazer uma PCA de correlação, que nada mais é que fazer a PCA usando dados que foram padronizados para ter média 0 e desvio 1 (Isto é, colocar os dados na mesma escala de variação). Fazendo isso todas as variáveis terão o mesmo peso na análise, pois todas terão a mesma variância. A PCA de correlação é adequada para quando as variáveis foram medidas em unidades diferentes ou quando a variância de cada variável é muito diferente umas das outras. Vamos ver os dados de variáveis químicas "varechem".

```
> round(apply(varechem,2,var),4)  # veja que a variância de cada variável é muito
```

diferente. Então temos que usar uma pca de correlação para que a variável com maior variância não "domine" a análise.

Podemos fazer a pca de correlação de duas formas: 1) padronizando as variáveis, ou 2) mudar o argumento `scale` da função `prcomp` para `scale=TRUE`.

Para padronizar os dados:

```
> varechem.P<-scale(varechem)
```

Agora basta refazer a PCA usando os dados padronizados.

```
> prcomp(varechem.P)
```

Ou apenas mude o argumento `scale=T`

```
> prcomp(varechem, scale=T)
```

Salve os resultados da PCA de correlação e compare com os resultados da pca de covariância. A primeira diferença a reparar é a diminuição de explicação do primeiro componente principal.

Análise de Coordenadas Principais (PCoA)

A análise de coordenadas principais é uma alternativa à PCA. Enquanto a PCA preserva apenas distância Euclideana, a PCoA preserva qualquer distância (Legendre and Legendre 1998). A PCoA se tornou bastante comum na ecologia por causa das diversas medidas de distância (ou dissimilaridade), como Bray-Curtis, Sorensen e Jaccard. Para fazer uma PCoA no R também existem diversas funções, a mais comum é a função `?cmdscale()`. Entre no help para ver os detalhes.

Para fazer a PCoA, vamos usar os dados das espécies presentes nos mesmos locais onde as amostras de solo (variáveis químicas) usadas acima foram tiradas. Os dados são:

```
> data(varespec)
```

A primeira diferença entre fazer uma PCA e uma PCoA no R é que em vez de entrar com a matriz de dados na análise, você entrará com uma matriz de dissimilaridade. Vamos usar a matriz de Bray-Curtis.

```
> spp.bray<-vegdist(varespec,"bray")
```

Para fazer a PCoA:

```
> cmdscale(spp.bray)
```

Veja que o resultado mostra apenas 2 eixos da PCoA. Neste caso, o default da função `cmdscale` é calcular apenas dois eixos, caso você queira mais eixos é preciso especificar o número de eixos que deseja, usando o argumento `k=2`. Se desejar mais de dois eixos aumente o número de k.

```
> cmdscale(spp.bray,k=3)
```

Outra informação que geralmente queremos obter é a porcentagem de explicação dos eixos. Para isso precisamos alterar o argumento `eig=FALSE` para `eig=TRUE`.

```
> cmdscale(spp.bray,k=2,eig=TRUE)
```

Veja que agora aparecem mais quatro resultados, eig, x, ac e GOF. eig são os autovalores (eigenvalues) e GOF (Godness of fit) é a porcentagem de explicação de todos os eixos determinados com o

argumento k. Por exemplo, se k = 1 o GOF indica a porcentagem de explicação desse um eixo, se k = 3 o GOF indicará a porcentagem de explicação dos 3 eixos. Veja que sempre aparecem 2 valores de GOF. Isso ocorre devido ao fato de que na PCoA podem haver autovalores negativos, dessa forma, o primeiro valor de GOF indica a porcentagem de explicação sem considerar os autovalores negativos, enquanto o segundo valor considera os autovalores negativos.

Escalonamento Multimenssional Não Métrico (NMDS)

Para fazer um NMDS no R, use a função `?metaMDS()`. Entre no R e veja os argumentos que precisam ser especificados ao usar essa função.

Use os dados "varespec" ou "mites" para fazer o NMDS.

```
> metaMDS(varespec,"bray")
```

Geralmente há duas coisas importantes para considerar ao usar a função `metaMDS`. Uma é o argumento `autotransform=TRUE`. Caso esse argumento não seja modificado para FALSE é possível que internamente seus dados sejam transformados (Veja no help a explicação sobre essa transformação). Portanto, é preciso estar ciente dessa possível transformação, que segundo o autor da função, ajudam a melhorar o resultado do NMDS.

Outra importante consideração é em relação ao argumento `zerodist`. Quando dois locais possuem exatamente as mesmas espécies e as mesmas abundâncias (para dados de abundância) a distancia/dissimilaridade entre esses locais será zero (ou seja, os locais são iguais). Distâncias iguais a zero não são permitidas e uma mensagem de erro aparecerá. Para solucionar esse problema existem duas opções. A primeira é mudar o argumento `zerodist` para `zerodist="add"`. Isso fará com que um valor desprezível seja somado a dissimilaridade entre os dois locais, de forma que ela não seja zero. A segunda, e conceitualmente mais correta, é remover um dos locais da análise e fazer o NMDS sem um dos locais. Qualquer conclusão que você tirar sobre aquele local que permaneceu na análise também valerá para o local que foi excluído.

Análise de Correspondência Canônica (CCA)

A análise de correspondência canônica é usada para relacionar uma matriz de dados de espécies em relação a uma matriz de dados ambientais. Neste exemplo vamos usar os conjuntos de dados `varespec` (dados de espécies) e `varechem` (dados do ambiente). A função cca do pacote vegan pode ser usada para fazer uma análise de correspondência canônica.

```
> ?cca
> cca(varespec, varechem)
```

Veja no resultado que aparecem 3 partes. A primeira mostra a Inércia (a variação nos dados) e a proporção da inércia que é explicada pelos dados ambientais (Constrained). A segunda parte mostra os autovalores dos eixos *constrained* (considerando as variáveis ambientais) e a terceira mostra os autovalores dos eixos *unconstrained* (sem considerar as variáveis ambientais). Vamos salvar os resultados da cca para ver outras opções.

```
> resu.cca<-cca(varespec, varechem)
> resu.cca
> summary(resu.cca)
```

Veja que os resultados agora contém varias partes. Vá até o inicio dos resultados (usando a barra de rolagem). O primeiro resultado mostra a proporção de inércia explicada. Depois, uma tabela mostra o autovalor e a porcentagem de explicação de cada eixo da CCA e dos eixos da CA (unconstrained). Depois são mostrados os scores das espécies, os weighted scores dos locais (WA scores, que são ponderados pelos scores das espécies), os scores lineares dos locais (LC scores, que são combinações lineares da matriz ambiental) e os scores das variáveis ambientais usadas para fazer o biplot.

```
> ?plot.cca
> plot(resu.cca)
```

Existem diversas formas para alterar esse gráfico da CCA. Veja o help e os exemplos abaixo.

```
> plot(resu.cca, display=c("wa")) # apenas os scores dos locais
> plot(resu.cca, display=c("wa","cn")) # scores dos locais e centroide das var amb.
> plot(resu.cca, display=c("sp")) # apenas os scores das espécies
> plot(resu.cca, display=c("sp","cn")) # scores das espécies e centroide das var amb.
> plot(resu.cca, display=c("wa"),type="n")
> text(resu.cca, display=c("cn"))
```

Para testar se o modelo da cca é significativo podemos usar a função anova.

```
> anova(resu.cca)
```

Análise de Redundância (RDA)

A análise de redundância é uma extensão da regressão múltipla para o caso multivariado. É bastante similar à CCA e pode ser usada para responder às mesmas questões. Detalhes de quando usar RDA ou CCA podem ser encontrados no capítulo 11 do livro Numerical Ecology (Legendre e Legendre 1998). Para fazer uma análise de redundância, repita os mesmos comandos realizados para fazer a CCA, porém, em vez de usar a função `cca`, use a função `?rda()`.

Classificações (Análises de Cluster)

Em geral, o objetivo das análises de cluster é reconhecer subconjuntos dos dados. Portanto, a análise de cluster consiste em dividir o conjunto de dados em subconjuntos. Em geral existem diversos métodos para fazer uma análise de cluster. Os métodos aglomerativo\divisivos, sequenciais\simultâneos, monotéticos\politéticos, hierárquicos\não hierárquicos, probabilísticos/não probabilísticos. Mais recentemente também há disponível as árvores de regressão multivariadas.

Cluster hierárquico

Para fazer um cluster hierárquico no R, a função mais usada é `hclust()`. Para fazer um cluster dos dados `varespec` use:

```
> spp.dist<-vegdist(varespec,"euclidean")
> hclust(spp.dist,method="single")
```

Outros métodos que podem ser usados são: "ward", "single", "complete", "average", "mcquitty", "median" ou "centroid". A definição de qual método usar varia com o objetivo do estudo e com o tipo de matriz de distância usada.

```
> cluster<- hclust(spp.dist,method="complete")
> plot(cluster)
```

Cluster aglomerativo (UPGMA e outros)

```
> clusterUPGMA<- hclust(spp.dist,method="average")
> plot(clusterUPGMA)
```

Outros métodos de cluster aglomerativos são:

Para (UPGMC) use o método "`centroid`"

Para (WPGMA) use o método "`mcquitty`"

Para (WPGMC) use o método "`median`"

Ward's Minimum Variance Clustering

```
> clusterWard<- hclust(spp.dist,method="ward")
> plot(clusterWard)
```

Correlação cofenética

Uma matriz cofenética pode ser calculada para representar as distâncias cofenéticas (distância de um nodo do dendrograma até outro) entre todos os pares de amostras. Calculando a correlação de Pearson entre a distância original (no exemplo spp.dist, que é a distancia de Bray-Curtis) e a distância cofenética ajuda a decidir qual método é melhor para ser usado com seus dados. A matriz coférnetica gerada pelos diversos métodos que tiver a maior correlação de Pearson com a distância original pode ser considerada o melhor método para descrever seus dados. A função `cophenetic()` do R calcula a matriz cofenética de cluster.

```
> cophenetic(clusterWard)
> matrizCofWard<-cophenetic(clusterWard)
> cor(spp.dist,matrizCofWard)
```

Para fazer um gráfico mostrando a relação entre as duas matrizes use:

```
> plot(spp.dist,matrizCofWard)
> abline(0,1)
> lines(lowess(spp.dist, matrizCofWard),col=2)# tendência
```

Exercício: Calcule a matriz cofenética dos clusters calculados com os outros métodos e diga qual deles é melhor para representar os dados "varespec" (distância calculada com o índice de Bray-Curtis).

Árvore de regressão multivariada

A árvore de regressão multivariada é uma extensão da árvore de regressão univariada para o caso multivariado. A análise funciona a partir de uma partição dos dados ("*constrained*" pelos dados ambientais) seguida por uma validação cruzada dos resultados (cross validation). O processo de partição é bastante simples. Para cada variável ambiental (preditora) todas as partições dos dados em dois grupos é realizada. Para cada uma dessas partições calcula-se a soma dos quadrados das distancias para a média de cada um dos dois grupos. Após salvar todos os resultados a solução que minimiza a soma dos quadrados é retida, dividindo os dados em dois grupos a partir da variável explanatória que produziu o resultado com menor soma dos quadrados dos desvios. Como os dados estão divididos em dois grupos, inicia-se uma nova partição para cada um dos dois grupos e assim por diante.

Em geral, usa-se uma árvore de regressão multivariada com dois objetivos. Um em que o objetivo é preditivo e outro em que o objetivo é explanatório. Para isso deve-se escolher a forma de validação cruzada da árvore gerada. Uma forma é minimizando o erro residual e a outra é maximizando o R^2. Maximizando o R^2 você prioriza um modelo mais explanatório, enquanto minimizando o erro residual um modelo preditivo será priorizado (ver De'Ath 2002 para mais detalhes).

Para fazer uma árvore de regressão no R precisamos do pacote `mvpart`. A função `mvpart` usa internamente em seu algoritmo a função `rpart` usada no caso univariado. Uma importante consideração a respeito de como fazer a árvore de regressão multivariada é que a variável resposta precisa ser um objeto da classe "`matrix`" enquanto as variáveis preditoras precisam estar em um objeto da classe "`data frame`". No exemplo vamos usar os dados `varespec` (resposta) e `varechem` (preditores)

```
> data(varespec)
> data(varechem)
> library(mvpart)
```

```
> mvpart(data.matrix(varespec)~.,varechem,xv="pick", xvmult
=100) # O "." indica que as variáveis resposta são todas dentro do objeto varechem.
xvmult indica o número de validações cruzadas que serão realizadas.
```

Este gráfico que aparecerá apresenta o erro residual relativo e a medida de relativa do erro preditivo (CVRE). A solução com o menor CVRE é apontada pelo ponto vermelho, bem como as barras de erro do CVRE

As barras verdes indicam o numero de vezes que a solução foi selecionada como a melhor durante as iterações da validação cruzada.

Para selecionar uma árvore, clique no gráfico sobre o ponto que representa o número de ramos desejado.

O gráfico que representa a árvore de regressão irá aparecer.

```
> resu.MVpart<-mvpart(data.matrix(varespec)~.,varechem, xv=
"pick",xvmult=100)
> summary(resu.MVpart)
> printcp(resu.MVpart)
```

Para mais detalhes sobre árvores de regressão veja Borcard et al., (2011) e De'Ath (2002).

Bibliografia

Borcard, D.. et al. 2011. Numerical Ecology with R. — Springer.

Braun, W. J. and Murdoch, D. J. 2007. A first course in statistical programming with R. — Cambridge University Press.

Chambers, J. M. 2008. Software for data analysis: Programming with R.

De'Ath, G. 2002. Multivariate regression trees: a new technique for modeling species-environment relationships. — Ecology 83: 1105-1117.

De'Ath, G. and Fabricius, K.E. 2000. Classification and regression trees: A powerful yet simple technique for ecological data analysis. — Ecology 81: 3178-3192.

Landeiro, V. L. et al. 2008. Responses of aquatic invertebrate assemblages and leaf breakdown to macroconsumer exclusion in Amazonian "terra firme" streams. — Fundamental and Applied Limnology 172: 49-58.

Legendre, P. and Gallagher, E.G. 2001. Ecologically meaningful transformations for ordinations of species data. — Oecologia 129: 271-280.

Legendre, P. and Legendre, L. 1998. Numerical ecology. — Elsevier.

Oksanen, J.. et al., vegan: Community Ecology Package. [R package version 1.17-6]. 2011. R package version 1.17-6.

R Development Core Team., R: A language and environment for statistical computing. [2.12.1]. 2010. Vienna, Austria, R Foundation for Statistical Computing.

Pseudocódigo para ordenar livros em uma estante.

Livros = livros que devem ser ordenados

Para cada livro, da esquerda para direita, **repita**
Memorize a ***posição*** do livro ***atual***
Pegue o livro ***atual*** e segure-o na ***mão***
Memorize o ***lugar*** do livro na ***mão***
Para cada livro, a direita da ***posição*** do livro ***atual***, **repita**
Se *este* livro for maior que o livro na ***mão***
Coloque o livro da ***mão*** de volta em seu ***lugar***
Pegue ***este*** livro e segure-o na ***mão***
Memorize o ***lugar*** do livro na ***mão***
Próximo livro
Coloque o livro da ***mão*** na ***cadeira***
Pegue o livro ***atual*** e coloque-o no ***lugar*** do livro da ***cadeira***
Pegue o livro da ***cadeira*** e coloque-o na ***posição*** do livro ***atual***
Próximo livro