Versão 13: 16 setembro de 2014

Conversando com o R usando 57 palavras:
Introdução à programação com exemplos em Ecologia

Adriano S. Melo asm.adrimelo láNo gmail.com
Dep. Ecologia, ICB
Universidade Federal de Goiás

Sumário

1 Introdução

1.1 O que é programação?

De forma simplificada, programação é a redação de um programa de computador para automatizar tarefas. De maneira mais formal, 'Programação é o processo de escrita, teste e manutenção de um programa de computador' (Wikipedia: http://pt.wikipedia.org/wiki/Programação_de_computadores). Programação é isto e apenas isto (Fig. 1.1).

Figura 1.1 Cartaz eletrônico feito por Luciano F. Sgarbi e Carolina R.C. Gomes para disciplina oferecida por Adriano S. Melo em Pirenópolis em 2012.

Quando falamos de programas de computador pensamos em algo como um navegador de internet (Chrome, Firefox, Internet Explorer etc), redatores de texto (MS Word, Write, Bloco de Notas etc) ou, mais recentemente, programas que acessamos na nuvem (Facebook, Dropbox, Google Drive etc). Além destes, também usamos programas mais simples rotineiramente, como aqueles embutidos em calculadoras ou letreiros eletrônicos.

Um programa de computador é feito a partir de um texto contendo instruções de como uma máquina (computador, calculadora) deve se comportar. Ou seja, o que deve fazer (por ex. somar) quando receber uma informação (por ex. números fornecidos por um usuário). Este texto é escrito numa linguagem de programação e existem dezenas de linguagens disponíveis. Estas diferentes linguagens geralmente possuem muitas coisas em comum, embora cada uma tenha sido criada para atender um determinado propósito e, portanto, também possuam características exclusivas. Esta relação (de parentesco) entre linguagens de programação não é muito diferente daquela existente entre as línguas (Português, Inglês, Espanhol etc). Após redigir um programa, normalmente precisamos compilar nosso programa, ou seja, transformá-lo numa linguagem que o computador entenda. Um programa faz isto, um compilador. Boa parte dos programas que você conhece são

programas compilados.

Além destas linguagens compiladas, existem linguagens que funcionam dentro de um determinado programa ou ambiente de programação. São chamadas de linguagens de extensão ou, mais frequentemente, como script. Estes scripts não são tão eficientes como as linguagens compiladas, ou seja, em geral demoram muito mais para processar uma informação. Por outro lado, em geral são mais fáceis e rápidas de se programar. Ainda, geralmente possibilitam trabalhar de forma interativa com o programa ou ambiente da qual fazem parte. Esta interatividade significa que você pode saber o que um determinado comando faz simplesmente mandando para o ambiente de programação (por ex. com um ENTER).

Existem dezenas de linguagens do tipo script. Em estatística e análise de dados, a mais comum hoje em dia é a linguagem R. Mas antes de falarmos de R, vamos falar um pouco de educação.

1.2 Você precisará de pouca matemática, mas terá que pensar!

"Eu posso até te ajudar, aliás, eu vou te ajudar! Eu quero te ajudar!
Mas agora você tem que me ajudar a te ajudar."
Sargento Rocha

Já ofereci muitas disciplinas na graduação e na pós-graduação envolvendo programação e análise de dados em Ecologia usando R. Na pós-graduação, ofereci as disciplinas "Introdução a Programação em R" [assunto deste livro], "Introdução ao Uso de Modelos Lineares em Ecologia" e "Métodos para Quantificar a Diversidade Biológica" para exatamente 32 turmas em 8 Programas de Pós-Graduação (até agosto de 2014). A maioria destas turmas tinha entre 10 e 30 alunos e, portanto, já conheci um bom número de pós-graduandos. Na graduação, minha experiência com disciplinas deste tipo é menor, embora seja suficiente para notar que o comportamento de graduandos e de pós-graduandos não é muito diferente quando se fala em estatística, análise de dados ou programação. Eu não tenho dúvida dos dois maiores problemas que tenho nestas disciplinas:

i) o aluno acreditar que não é capaz de aprender algo que envolva números, e

ii) o aluno esquecer que seu cérebro foi feito para pensar.

Você pode achar que estou exagerando. Afinal de contas, se o aluno está em sala de aula é porque deve estar querendo aprender. Mas a vida não é tão simples assim. Aqueles neurônios do seu córtex cerebral responsáveis pelo seu consciente estão te impelindo a ir para sala de aula e aprender, pois sabem que tais ferramentas quantitativas são importantes para seu futuro profissional. O problema é que eles não são ditadores; eles devem discutir e negociar com os neurônios do seu

subconsciente e aí o problema geralmente aparece.

Nossos colegas, pais e, por incrível que pareça, até mesmo nosso sistema educacional martelam o tempo inteiro que Matemática é algo difícil. Fazem tanto isto ao longo de nossa infância e adolescência que acabamos incorporando isto como verdade no nosso subconsciente. Para piorar, acabamos incluindo neste saco de coisas complicadas chamada "Matemática" qualquer coisa que envolva números.

Costumo iniciar minhas disciplinas dizendo quais operações matemáticas usaremos: soma, substração, multiplicação, divisão, logaritmo, exponenciação, radiciação e módulo. Alguma dúvida? Talvez os nomes "radiciação" e "módulo" já estejam sumindo de sua memória. O primeiro é aquele que envolve raiz quadrada e o segundo é aquele composto por duas barras verticais que colocamos ao redor de um número para simplesmente retirar seu sinal. Assumo que você conhece estas operações matemáticas e portanto possui o fundamental para conseguir seguir este livro. Não estou brincando. Só vamos precisar disto e um pouco de raciocínio, e aqui entra o segundo problema.

Espero que esteja claro que por mais superficial que seja sua Matemática, ela é suficiente para seguir este texto e realizar a maioria das análises usadas em Ecologia. Quanto ao raciocínio, vamos admitir, nosso treinamento escolar geralmente é péssimo. Se você fez graduação em Ciências Biológicas ou algo do tipo, tenho certeza que seus professores ficaram durante quatro anos ou mais jogando terra sobre seu raciocínio. Tenha certeza que ele ainda está ai dentro de seu cérebro, embora bem enterrado. Não é muito difícil entender como fizeram isto com você.

Nas Ciências Biológicas, praticamente todas disciplinas exigiram que você memorizasse palavras, imagens e historinhas de como algo biológico funciona. Isto e apenas isto. Você deve ter ficado craque nisto, mas seu raciocínio sofreu bastante. Nossos professores foram treinados desta forma e acabam aplicando o mesmo em suas disciplinas. Embora um absurdo na era Google, somos treinados a memorizar detalhes. Muitos detalhes. Veja seus livros-textos, aqueles que você usou em suas disciplinas de graduação. Te exigem pensar em algo? É claro que não. Aliás, já notou que os professores sempre gostam daqueles livros mais pesados, com muitas páginas? Dizem que é mais "completo". Completo de detalhes. Os autores de livros sabem disto e quase infalivelmente tornam seus livros cada vez mais volumosos em edições posteriores (pois sempre descobrem detalhes não incluídos nas edições anteriores; chegam até a agradecer aqueles que os lembram dos detalhes não inclusos nas edições iniciais) (Rigler & Peters 1995).

Pode acreditar em mim quanto ao que falei acima sobre Matemática: você possui o fundamental. Por mais difícil que seja, acredite nisto. Eu quero te ajudar, mas você tem que colaborar. O primeiro passo é este: acredite, simples assim.

Sobre o raciocínio, muitos de vocês terão que fazer força para desenterrá-lo. Posso te dar algumas dicas:

i) Você terá dúvidas ao longo do texto, principalmente quando estiver fazendo exercícios. Absolutamente normal. Alguns alunos acham isto ruim. Eu acho ótimo. Muitos alunos não querem ficar em dúvida, se sentem desconfortáveis. Eu adoro quando isto acontece. Não sou sádico. O problema de você se sentir desconfortável nesta situação é que raramente passou por isto. Você não aprendeu a lidar com isto. Você foi treinado a memorizar e não pensar. Minha sugestão: faça esquemas num pedaço de papel, veja se já reuniu todas informações necessárias para resolver o problema e, finalmente, pense! Caso esteja em sala de aula, te sugiro fortemente que jamais solicite ajuda do seu colega ao lado. Você estará passando o problema para ele (ruim para você) e ainda por cima distraindo-o do que estava fazendo (ruim para ele). É claro que você não vai ficar parado no problema eternamente. Você deve tentar resolver o problema durante certo tempo. Procure ajuda apenas após este período. Se estiver em sala de aula, peça ajuda ao professor ou monitor (não do seu colega) de forma objetiva. Não adianta muito dizer: não entendi. Diga o que não entendeu, onde está o problema. Por fim, espero que você perceba que a melhor coisa que pode te acontecer é ganhar um pouco de experiência na resolução de problemas. Qualquer problema. Todos temos problemas. A diferença é que alguns aprenderam a resolvê-los. Outros não. Escolha o seu time.

ii) Estamos na era multitarefa, fazendo várias coisas simultaneamente. Isto pode ser muito legal, pois você pode aproveitar sua vida. Mas também pode ser uma desgraça, pois corre o risco de não fazer nada de forma correta e intensa. É uma grande besteira ficar pensando em sua dissertação/tese quando está com sua namorada(o) no cinema ou se deliciando com uma cerveja na praia. Esqueça o trabalho e aproveite intensamente o momento. Da mesma forma, tenha concentração no seu trabalho. Não fique tentando responder e-mail, facebook, skype e coisas do tipo durante a aula. Você não conseguirá dividir satisfatoriamente seu processador cerebral. Desligue todas brincadeiras de internet quando for estudar ou estiver em sala de aula. Elas podem parecer importantes, mas não são.

iii) Cada um tem seu tempo de aprendizagem. Algumas pessoas têm maior facilidade em aprender determinadas coisas do que outras. Não se preocupe. Isto não é relevante. Relevante mesmo é ter persistência: "Ter só aptidão não adianta; é necessário muito esforço e dedicação para transformá-la em sucesso e realização profissional. Isso é tanto verdade para as pessoas normais como para os ditos 'gênios'. A diferença, talvez, seja a quantidade de esforço requerida. É como se os gênios

tivessem um barco a motor para atravessar um rio e os normais, um barco a remo. Claro, é mais fácil para os gênios. Mas eles precisam saber pilotar o barco e ter gasolina suficiente para chegar do outro lado. Os remadores têm de trabalhar mais duro, mas também podem chegar lá. O importante é querer atravessar o rio, ter determinação para isso, mesmo que os braços doam, a sede aperte e você veja outros barcos adiante, deixando você para trás." Marcelo Gleiser (2007).

iv) A culpa não é do professor. Temos uma cultura pedagógica extremamente perversa em que tudo é culpa do professor. Se o aluno vai mal ou não tem interesse, a culpa é do professor. Implícito nesta cultura está a ideia de que o aluno é um ser passivo, alimentado pela regurgitação do professor. Você pode continuar pensando assim. Será mais confortável, mas também será sua desgraça. Se você quer aprender, deve considerar, como dizia um professor na minha graduação, que seu professor e seu livro são apenas muletas. Eles te ajudam, mas quem anda é você. Não há como as muletas fazerem você andar. Você tem que querer. De verdade! Neste sentido, uma das coisas mais gratificantes como professor é quando consigo fazer o aluno ter raiva o suficiente de mim para que ele queira demonstrar que é capaz de fazer algo bom na disciplina. Novamente, não sou sádico e não quero ficar te perturbando. Quero apenas que você tome as rédeas do processo de aprendizagem. Fique com raiva deste livro ou do seu autor, mas use esta raiva de maneira proveitosa (não vá riscar meu carro!).

Todo livro ou apostila é baseado em um monte de outros livros e apostilas. É reciclagem com um novo tempero. Caso tenha curiosidade sobre o funcionamento do seu cérebro, veja o fascinante livro do Nicolelis (2011). Ainda nesta linha, veja uma narração das brigas do seu consciente com seu subconsciente em Eagleman (2011). Você vai descobrir que conhece menos de você mesmo do que (seu consciente) imagina. Mais do que genialidade, você precisa ter foco e persistência no trabalho científico. Se você acha que seu cérebro consegue fazer várias coisas ao mesmo tempo, como ter aula e conversar com seu colega ao lado, veja se consegue ver o gorila de Chabris & Simons (2009). Para um biólogo, a autobiografia do Wilson (1994) é inspiradora, enquanto o livrinho (no melhor sentido da palavra) do Gleiser pode te dar um norte no meio do turbilhão de suas dúvidas. Reconhecer o problema é o primeiro passo para sua solução e neste sentido o Cap. 10 de Rigler & Peters (1995) é avassalador quanto ao nosso sistema de ensino. Fiquei um tanto deprimido quando percebi o tanto de inutilidades que "aprendi" na graduação. Aliás, se você está na pós-graduação e ainda não se sente um cientista (já deveria se sentir!), aproveite os três capítulos iniciais de Rigler & Peters (1995). É uma maravilha de objetividade. Se gostou destes três iniciais do Rigler & Peters, também vai gostar do livrinho (novamente, no melhor sentido da palavra) do Rubem Alves (2011).

Finalizando, o texto desta seção pode parecer auto-ajuda. E é mesmo! Aproveite!

1.3 Programação como ferramenta de aprendizado

Se sempre te disseram que estatística é algo difícil e não tem experiência com programação, você deverá achar estranho eu afirmar que programação é uma excelente ferramenta para aprender estatística ou análise de dados. Pois é. Trabalho com R desde 2004 e isto foi uma das coisas mais bacanas que percebi ao longo desta década de trabalho. O motivo disto é bastante simples.

Caso não tenha um treinamento formal com ciências exatas, você provavelmente vai olhar para um problema (uma análise) e tentar resolvê-lo sem qualquer estratégia. Vai ser difícil. Isto pois existem várias informações que precisam ser reunidas, passos intermediários que precisam ser entendidos, resultados de passos intermediários que não são óbvios, detalhes analíticos específicos que nunca vimos etc. Olhar para o problema (ou análise) como um todo vai parecer Grego. Você reconhecerá uma letra ou outra, mas não deixará de ser Grego.

A programação ajuda no sentido de prover métodos para resolvermos um problema ou entender uma análise: i) temos que reunir os ingredientes (ou informações) necessários; ii) identificar (dar nomes) os ingredientes necessários; iii) decompor nosso problema em partes e iv) colocar estas partes em sequência lógica. Quando fazemos nossos programas, estes passos acabam aparecendo de forma natural. Com o tempo, você vai achá-los óbvios.

Linguagens tipo script, como o R, ainda proveem auxílio de outra forma. Podemos testar cada passo de nosso programa de forma automática, interativa. Muitas vezes não vale a pena ficar perdendo tempo procurando informações para ver se algo funciona assim ou assado. Faça e veja. É muito rápido. Em geral basta um ENTER. Isto possibilita testar partes do seu programa e ver se estão fazendo o que deveriam fazer.

Neste livro usaremos várias análises tradicionais em Ecologia, particularmente sobre Diversidade Biológica (meu viés...). Espero que análises aparentemente difíceis se revelem triviais após você ter escrito um programa para calculá-las.

1.4 O ambiente R

O R é tanto um programa estatístico quanto um ambiente de programação. Você pode usá-lo para executar uma análise já disponível ou programar uma. Em relação à execução de análises, ele é atualmente o mais completo de todos. Na maioria das vezes, a pergunta adequada não é se o R faz algo (sim, ele faz), mas onde encontrar. Para a última, o Google é uma grande ajuda.

Trabalhamos no R usando linhas de comando. Para quem nasceu depois de 1975, isto é um

tanto estranho pois "já nasceram" na era dos cliques. Os mais experientes (eufemismo para velhos) certamente lembram do MS-DOS e dos programas baseados em linhas de comando. Posto deste jeito, você pode pensar sobre a razão de usarmos algo do passado. Na verdade, trabalhar com linhas de comando nunca desapareceu. Os profissionais da Ciência da Computação geralmente usam sistemas operacionais baseados em linhas de comando. Mas qual a razão de nós, simples mortais, usarmos linhas de comando? Amplitude de possibilidades e eficiência.

Num programa com janelas temos um limite para o número de opções disponíveis. Chega um ponto em que perdemos muito tempo procurando as coisas. Este problema simplesmente não existe com linhas de comandos. Basta dar um nome novo. Em relação à eficiência, podemos executar análises muito rapidamente, desde, é claro, que você conheça os comandos. Depois de apanhar um pouco no início e passar a ter certa familiaridade, você vai achar uma grande vantagem trabalhar com linhas de comandos. De fato, usuários de R acabam se tornando discípulos de uma religião. Eles não conseguem entender como alguém consegue não trabalhar com R... Mais importante ainda no caso do R, você poderá escrever análises ou rotinas de análises para automatizar tarefas repetitivas. Este livro trata deste último tipo de atividade.

O R descende do S. A linguagem S surgiu na década de 1970 e tinha como objetivo principal ser uma linguagem para análise de dados (Chambers 1998). A linguagem foi implementada num ambiente de programação chamado S+, programa comercial ainda disponível (mas com outro nome). A linguagem S atraiu muitos usuários e foi fortemente ampliada com pacotes (ou bibliotecas) para análises específicas de diversas áreas do conhecimento. Na década de 1990, Ross Ihaka e Robert Gentleman (University of Auckland, Nova Zelândia) iniciaram o desenvolvimento de um ambiente de programação para ser usado em aulas. Este ambiente era baseado em S e foi chamado de R (inicial dos nomes dos autores). Com o passar do tempo, o projeto cresceu e foi adotado por outros pesquisadores. No final da década de 1990 os autores decidiram abrir o desenvolvimento do ambiente para um seleto grupo de desenvolvedores. Uma versão estável e relativamente completa ficou disponível em 2000. A partir de então, o número de usuários cresceu exponencialmente. Este sucesso deve-se, em parte, pelo fato do R ser software livre e conter milhares de pacotes que ampliam de forma incrível seu leque de opções de análises.

1.5 Filosofia de programação em R: rápida e confiável

A ideia de programação no R é que ela seja implementada de forma rápida, mas correta. Nas palavras de um dos criadores da linguagem S, basicamente a mesma usada no R, 'To turn ideas into software,

quickly and faithfully' (Chambers 1998). Em relação a linguagens de programação tradicionais (por ex. C, C++), programar no R é muito fácil. Primeiro, você não precisa se preocupar com uma série de detalhes. Você pode ir direto ao ponto sem perder muito tempo com detalhes periféricos. A segunda facilidade é que já existem centenas de milhares de comandos (ou melhor, funções) já implementadas. Você não terá que sair do zero. Você simplesmente combinará as funções já disponíveis para alcançar seu objetivo. Quase um quebra-cabeça. Mas existem alguns poréns: em relação a programas feitos com linguagens compiladas (por ex. C++), o R é muito lento. Este 'muito lento' pode significar tempo de processamento 20 vezes maior em R. Esta lentidão não é relevante em 99% das situações, mas acaba sendo um problema com conjuntos de dados muito grandes e análises que demandam muito processamento.

1.6 Opção do autor: simplicidade

Este livro é fortemente baseado na minha experiência docente. Durante muitos anos montei cursos com uma estrutura um tanto tradicional. Eu explicava nas primeiras aulas uma longa lista de comandos (ou melhor, funções) básicos. Estas aulas refletiam bem os capítulos iniciais da maioria dos livros de estatística ou programação em R. O objetivo principal destas aulas iniciais era prover uma lista de funções (um vocabulário) e alguma noção de como usar e combinar estas funções. Eu costumava dizer que esta lista compunha mais de 70% do que veríamos na disciplina. O restante do curso, portanto, seria focado no aprofundamento do uso destas funções básicas.

Este formato tradicional funcionava relativamente bem para parte dos alunos, que com 2-3 aulas já conseguiam ter domínio razoável das funções mais comuns e partir para coisas mais interessantes. O problema é que parcela considerável da turma acabava levando mais tempo para absorver este novo vocabulário. Isto gerava alguns problemas nas partes finais do curso. Enquanto parte dos alunos já estavam voando, outros estavam começando a entender para que asas servem.

Com o passar do tempo, passei a reduzir o vocabulário. Por exemplo, deixei de falar das cinco ou mais formas de importar dados. Passei a falar e usar apenas uma. Outra coisa fundamental foi falar menos e fazer mais exercícios. Sobre os exercícios, passei a dar maior atenção ao nível de dificuldade. Muito do que é obvio para o professor não é para o aluno. OK, sei que isto é obvio, mas de vez em quando esquecemos isto.

Como resultado destas experiências docentes anteriores, este livro possui formato um pouco distinto em relação a várias apostilas e livros disponíveis. Alguns leitores com um pouco mais de familiaridade com o R vão se perguntar a razão de numa determinada passagem não se usar a função X que poderia substituir facilmente 3-4 linhas de código. Vão notar que escrevi em cinco linhas algo

que poderia ter escrito em apenas uma. Ainda, vão notar que eu simplesmente não menciono algumas funções de uso frequente e presentes em livros introdutórios. Por outro lado, com apenas 15 funções bem simples faremos várias funções para objetivos bem diversos (ver Capítulo 5).

Estas coisas estranhas foram propositais. Espero que este livro cumpra seu papel junto a usuários sem qualquer experiência de R ou programação. Espero que estes leitores sejam atraídos e não repelidos.

2 Primeiros passos no R: I

2.1 Interface gráfica

Podemos trabalhar diretamente no R (obtenha gratuitamente em um sítio espelho no Brasil, por exemplo, http://cran-r.c3sl.ufpr.br/) ou usando outra interface gráfica (GUI, Graphical User Interface). Existem várias opções para os diferentes sistemas operacionais disponíveis. No momento em que escrevo este livro, uma das mais interessantes parece ser o RStudio, que está disponível gratuitamente para Linux, Windows e Mac (http://www.rstudio.com/). O RStudio parece ser um pouco mais amigável para iniciantes do que a interface básica do R. Você precisa instalar o R antes para que o RStudio funcione.

2.2 Área de trabalho

O R trabalha com um arquivo chamado de área de trabalho (workspace), onde armazenamos todos nossos 'dados'. Coloquei a palavra entre aspas pois veremos que isto pode significar várias coisas. O arquivo normalmente é gravado sem nome e com extensão .RData. Um segundo arquivo é gravado automaticamente, geralmente sem nome e com extensão .Rhistory. Este arquivo guardará tudo o que você digitar (mas não os resultados) e pode ser aberto com um editor de texto simples (e.g. Bloco de notas no Windows, gedit no Linux). Em distribuições Linux, caso grave o arquivo .RData e não consiga ver na pasta, escolha no menu a opção 'ver arquivos ocultos' (ou Ctrl+h ou Alt+.).

Quando você abrir o R (veja instruções para RStudio abaixo) ele automaticamente carregará um arquivo .RData que está no diretório onde você instalou o R (no Windows) ou na raiz dos seus arquivos pessoais (Linux). Não é um bom lugar para guardar seus arquivos pessoais. Portanto, procure no menu a opção "Salvar área de trabalho". Grave numa pasta onde pretenda armazenar todos arquivos para acompanhar este livro. Após gravar, feche o programa. Ele perguntará se quer salvar a área de trabalho. Diga que não (pois acabou de salvar). Vá com seu gerenciador de arquivos no diretório onde gravou o .RData e clique com o botão do mouse direito. Escolha opção de abrir com o R. O arquivo .RData em uso é este que você criou. É claro que você pode abrir um arquivo .RData pelo menu, mas muitas vezes esquecemos de fazer isto no início da sessão e acabamos gerando confusão. Abra o R sempre pelo arquivo que quer editar. Esqueça o ícone na sua área de trabalho ou menu de programas. O R pergunta se quer salvar quando for fechar o programa. Diga que sim e ele vai gravar exatamente neste arquivo que abriu.

Com o RStudio, escolha a opção 'Session' no menu e depois 'Save Workspace As'. Selecione o diretório onde quer gravar. Feche o programa. Caso pergunte se quer gravar, diga que não (pois acabou de gravar). Assim como no uso do R direto, vá no diretório onde gravou o arquivo .RData e clique com o botão direito e escolha opção abrir com RStudio. Ele vai carregar o que existe neste arquivo e gravará novos 'dados' dentro deste arquivo.

Geralmente usamos uma área de trabalho para cada projeto que temos. Por exemplo, você pode abrir um diretório no seu computador ou num pen-drive e gravar um arquivo .RData para este livro ou curso (o .Rhistory é gravado automaticamente). Neste diretório, adicione arquivos de dados ou outros relevantes ao seu projeto. Em particular, é interessante gravar um arquivo de texto contendo os comandos e comentários de trabalho no R. Chamamos este arquivo de script. Vamos falar dele no próximo capítulo.

2.3 Uso como calculadora

Abra o R com o arquivo .RData que vai usar para acompanhar este livro. Vamos ver como funciona. Caso esteja com o RStudio, clique no espaço onde o R foi aberto. É fundamental acompanhar os comandos abaixo com o programa.

O R pode ser usado muito facilmente como uma calculadora. Digite:

```
> 2+2
```

Clique ENTER e aparecerá:

```
[1] 4
```

Note que na linha em que digitou 2+2 existe um símbolo de maior '>'. É o R te dizendo que você pode digitar. O "[1]" simplesmente indica que o primeiro número da linha está na posição 1. Aqui esta informação não é tão útil visto que só temos um número. Agora tente outras possibilidades e veja o resultado (o símbolo '>' será omitido no resto do texto por simplicidade):

```
2-2

2/2

2*2

2^2

3^3
```

Muitas vezes esquecemos de completar um comando e apertamos Enter. Nestes casos, o R fica esperando informações para completar a ação. Para mostrar isto, o símbolo de maior é substituído por

um símbolo de adição '+'. Por exemplo:

```
> 3*
+
```

Neste caso, basta completar com um número e apertar Enter para efetuar a multiplicação. Isto também acontece quando abrimos e não fechamos parênteses, colchetes e chaves. Vamos ver como usar estes símbolos nas próximas seções e capítulos.

2.4 Uso geral: `maria( )`

Cerca de 99% do uso do R é feito por comandos, ou para ser mais específico, funções. Cada função faz uma determinada operação. Por exemplo:

```
log(x=8, base=2)
[1] 3
```

A função **`log()`**, como o próprio nome diz, calcula logaritmos. Cada função obrigatoriamente é seguida de parênteses, onde colocamos os argumentos da função separados por vírgulas. Argumentos são as informações que passamos para a função para indicar o que queremos fazer. Cada função pode ter de zero a dezenas de argumentos. No exemplo acima, temos um argumento "**`x`**" que é o valor(es) do(s) qual(is) queremos tirar o logaritmo. O argumento "**`base`**" indica qual base do logaritmo usar. Portanto, podemos ter:

```
log(x=100, base=10)
[1] 2
```

O R pode executar a função mesmo que você não escreva o nome do argumento. Neste caso, o R interpreta que os argumentos dados seguem a ordem utilizada na função:

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

Muitas funções possuem opções padrões (default) para alguns de seus argumentos. No caso da função **`log()`**, não temos como não incluir o argumento "**`x`**". Temos que tirar log de algum número! Entretanto, não precisamos incluir o argumento que indica a base do logaritmo:

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

Qual base foi usada? Como saberei os argumentos disponíveis para cada função ou mesmo a ordem destes argumentos? Podemos obter estas informações no arquivo de ajuda.

2.5 Ajuda

Cada função possui um arquivo de ajuda, onde podemos saber a ordem dos argumentos na função, o que é cada argumento e uma série de outras informações. Para abrir uma página de ajuda, basta digitar um ponto de interrogação antes do nome da função:

```
?log
```

Todas páginas de ajuda possuem a mesma estrutura. Examine a página da função **log()**. Note que na seção "Usage" a função **log()** possui como opção padrão do argumento "**base**" a expressão **exp(1)**. Ou seja, caso não inclua o argumento "**base**" na função **log()**, **exp(1)** será usada. Note que **exp(1)** parece ser uma função e, de fato, é:

```
exp(1)
[1] 2.718282
```

Esta é a constante de Euler. **exp(x)** é uma função para calcular e^x, onde e = constante de Euler e x = expoente da constante. No caso de **exp(1)**, estamos elevando "e" a 1, que é o próprio valor de "e".

Diferente de outros programas, usamos muito as páginas de ajuda. Visto estas páginas estarem em formato html, você pode navegar pelas páginas de ajuda. No menu do RStudio, use a opção de 'Ajuda' e depois a opção 'Ajuda do R' para acessar a página inicial em html e vários arquivos PDF de uso básico e avançado.

Também podemos procurar palavras nos títulos das páginas de ajuda com dois pontos de interrogação:

```
??logarithm
```

2.6 Atribuição (assign)

Vimos acima que o resultado da função **log()** apareceu na tela. Podemos guardar este resultado para uso posterior. Isto é feito atribuindo um nome ao resultado ou, mais especificamente, ao objeto produzido pela função **log()**. Fazemos isto com uma 'seta' (símbolos de menor e hífen; com ou sem espaços antes do símbolo de menor e depois do hífen). Por exemplo:

```
marcos <- log(100, 10)
```

Note que o resultado não sai na tela. Entretanto, você pode obtê-lo simplesmente digitando:

```
marcos
[1] 2
```

Aqui criamos um objeto chamado marcos que contém o valor 2. Podemos usar este objeto em outras operações:

```
marcos*5
[1] 10
```

2.7 Meia dúzia de funções básicas

Vamos praticar um pouco o uso do R com 6 funções bem básicas. Muitas vezes precisamos juntar 'coisas', na maioria das vezes, números. A função para isto é:

```
a <- c(1, 2, 5, 8)
a
[1] 1 2 5 8
```

A função **c()** concatena ('junta') coisas. O R é uma linguagem feita para trabalhar com vetores, que são muito simplisticamente um conjunto de números. Portanto podemos aplicar muitas operações diretamente sobre eles:

```
a*2
[1] 2 4 10 16
```

Veja estas outras funções:

```
sum(a)
length(a)
sum(a)/length(a)
mean(a)
sqrt(a)
```

2.8 Lista de conceitos e funções usadas

área de trabalho, .RData e .Rhistory

funções: uso geral
uso de: parênteses
argumentos, separação de argumentos com vírgula
uso de ordem dos argumentos em vez dos nomes dos argumentos
argumentos padrão (ou default)
atribuição

operações básicas: +; -; /; *; ^ log() exp() ? ?? c() sum() length() mean() sqrt()	

3 Primeiros passos no R: II

3.1 script (arquivo de roteiro)

Podemos trabalhar no R digitando diretamente em seu console (o espaço que recebe o que você digita). Entretanto, o uso de scripts é muito útil e será incentivado neste livro. O script é simplesmente um arquivo de texto simples onde escrevemos tudo o que queremos fazer. Ele pode ter comandos para funções, comentários ou texto simples. O próprio R pode abrir um script e dentro deste pode-se mandar um comando para o console simplesmente com Ctrl+R (Ctrl+ENTER para RStudio ou Command+ENTER para Mac). Desta forma, podemos trabalhar apenas no script e, quando necessário, mandar algo para o R executar. Outras interfaces gráficas (ou GUI = Graphical User Interface), como o RStudio, também possibilitam o uso de scripts. Para abrir um script no R ou RStudio basta usar o menu Arquivo (ou File). É comum usar a extensão .R para gravar arquivos de script do R no computador.

Algumas pessoas praticamente não armazenam nada além do conjunto de dados básico no arquivo .RData. Quando precisam refazer algo, simplesmente mandam os comandos já escritos no script para o console e rapidamente conseguem tudo o que precisam. É uma forma de ter sua área de trabalho com poucos objetos.

Quando fazemos um trabalho científico, é muito comum fazermos uma série de análises, algumas simplesmente equivocadas, outras não importantes e outras que acabam indo para o manuscrito. O script é um excelente local para documentar toda esta história, tanto do que deu certo quanto do que deu errado. Quando o manuscrito volta da revista, após 2-4 meses, é bastante comum ter que refazer alguma análise. Caso tenha feito um bom script, isto será muito fácil. Muitas vezes, bastará mudar uma ou outra função ou ainda um ou outro argumento de uma função.

O R interpreta sua linha de comandos até encontrar um símbolo de jogo-da-velha "#". Este símbolo é usado para inserir comentários:

```
log(100, 10) # aqui estamos tirando o log de 100 na base 10.
```

Comentários são muito importantes para registrar o que estamos fazendo, o que deu certo, o motivo de fazer isto e não aquilo etc. Muito do código de comandos que você escreve hoje vai virar grego na semana que vem, principalmente na fase inicial de uso do R. Caso tenha colocado comentários, uma revisão posterior será grandemente facilitada. Use a vontade para acompanhar este

livro.

No capítulo 2 havia dito que é interessante ter um arquivo de área de trabalho (.RData) por projeto, e um diretório no seu computador para este projeto. Além disto, sugiro ter um arquivo de texto para o script, com extensão .R, neste mesmo diretório. Para acompanhar este livro, sugiro que use apenas um arquivo de script e uma área de trabalho. Use o arquivo de exercícios que acompanha esta apostila. No script, anote todos seus comentários e dúvidas. Para deixar mais organizado, você pode fazer cabeçalhos do tipo:

############## Capítulo 2 ##############

Isto facilita bastante a navegação pelo arquivo e ajuda no encontro de funções ou comentários feitos anteriormente. A partir do próximo capítulo criaremos algumas funções. Seria bom guardá-las num único local (não se esqueça de fazer cópias de segurança deste arquivo).

3.2 Objetos: **vector** e duas funções relacionadas

Tudo dentro do R são objetos. Precisamos conhecer estes tipos de objetos pois algumas funções são específicas quanto ao tipo de objeto que usam. Vamos iniciar com **vector**. De maneira bem simplista, é um conjunto de números ou caracteres. Quando criamos o objeto abaixo, estamos criando um vetor de comprimento 1.

```
a <- 3
```

Para saber este comprimento, havíamos visto no capítulo anterior a função **length()**. Para criar um vetor, usamos no capítulo anterior a função **c()**. Estas duas funções também são úteis:

```
rep(x=10, times=3)  #repete o número 10 três vezes.
seq(from=5, to=10, by=0.2) # cria sequência de 5 até 10 em passos de 0,2.
```

Muitas vezes precisamos criar sequências de números inteiros. Podemos usar o **seq()** com o argumento **by=1**. Mas também temos uma forma abreviada do **seq()**:

```
1:10
10:1
-3:2
```

3.3 Gerenciando objetos na área de trabalho

Muitas vezes esquecemos dos nomes dados aos objetos que criamos. Podemos listar todos na tela:

```
ls()# note que eles também estão listados na janela superior direita do RStudio.
```

Caso queira remover algum deles:

```
rm(a) # caso tenha o objeto 'a' na sua área de trabalho.
```

3.4 Pacotes

As funções do R estão organizadas dentro de pacotes. Cada pacote pode ter de uma a centenas de funções. Quando instalamos o R também estamos instalando alguns pacotes de uso generalizado. Além destes, entretanto, podemos instalar pacotes disponíveis no sítio eletrônico do R. Existem alguns milhares de pacotes, muitos para coisas que você nem imagina. Na prática, baixamos pacotes conforme nossa necessidade. Para quem trabalha com Ecologia de Comunidades, um pacote fundamental é o "vegan". Na área de Filogenia de Comunidades, muitas funções de interesse podem ser encontradas nos pacotes "ape" e "picante".

Para instalar um pacote num computador que esteja na internet basta acessar o menu 'Pacotes' e escolher a opção de instalar pacotes. No Rstudio veja a aba 'Packages' na janela que normalmente fica no canto inferior direito. O R abrirá uma janela para você escolher o repositório de onde quer baixar. Existem vários ao redor do mundo e todos possuem exatamente os mesmos arquivos. Escolha o mais próximo de você. Uma segunda janela é aberta com a lista de pacotes. Basta selecionar um deles e o próprio R baixa e instala. Note que um pacote instalado desta forma não estará disponível automaticamente. Para usar, precisamos carregar o pacote. A possibilidade mais simples é clicando na caixinha do pacote desejado (janela a direita embaixo do RStudio).

Você precisará instalar um pacote apenas uma vez, mas precisará carregar o pacote cada vez que iniciar uma sessão do R. Os pacotes de uso generalizado do R já são carregados automaticamente quando você inicia uma sessão do R. Para ver descrições dos pacotes do R, acesse: http://cran.r-project.org/web/packages/available_packages_by_name.html. Os pacotes instalados no seu computador são aqueles relacionado na janela a direita embaixo do RStudio.

Qualquer pessoa pode fazer um pacote para o R e submeter ao CRAN (o sítio eletrônico onde os pacotes são armazenados). Isto é uma das coisas que fez o R bastante popular. A comunidade de pesquisadores não precisa esperar que uma empresa implemente uma dada análise em seu programa. A comunidade de pesquisadores pode assumir isto e fazer de maneira bem rápida. Cada pacote possui um ou mais pesquisadores responsáveis por sua manutenção. Visto ser software livre, o uso ou mal uso é por sua conta.

3.5 Lista acumulada de conceitos e funções usadas

Conceitos e funções vistos pela primeira vez neste capítulo estão em negrito.

área de trabalho, .RData e .Rhistory
funções: uso geral
uso de: parênteses
argumentos, separação de argumentos com vírgula
uso de ordem dos argumentos em vez dos nomes dos argumentos
argumentos padrão (ou default)
atribuição
`script`
`comentários`
objetos: **`vector`**
`pacotes`

operações básicas: +; -; /; *; ^
`log()`
`exp()`
`?`
`??`
`c()`
`sum()`
`length()`
`mean()`
`sqrt()`
`ls()`; **`rm()`**	

4 Seus primeiros programas no R

4.1 Uma função para fazer funções

Vimos até agora uma dúzia de funções e os operadores aritméticos básicos. Pouco, mas é suficiente para já programarmos várias funções! Vamos ilustrar o procedimento fazendo uma função para o Índice de Diversidade de Simpson. Índices de diversidade combinam informações de riqueza de espécies e equabilidade. Maior o número de espécies, maior o valor do índice. Maior a semelhança na abundância (ou outra medida de importância), maior o valor do índice. A formula é 1-D ou 1/D, onde $D = \Sigma p_i^2$, onde p_i é a proporção da abundância total representada por indivíduos da espécie i.

Uma função é um objeto do R. Vimos no capítulo anterior o objeto **`vector`**. Função é o segundo tipo que vemos neste livro. Criamos uma função com a função **`function()`**. Vamos fazer a função usando a forma 1/D:

```
f1.simp <- function(abund){#1 cria a função. Note que possui um argumento: abund
total <- sum(abund)        #2 abundância total na amostra.
prop.i <- abund/total      #3 proporção de cada espécie.
prop.i.2 <- prop.i^2       #4 proporções elevadas ao quadrado.
D <- sum(prop.i.2)         #5 soma das proporções elevadas ao quadrado.
indice <- 1/D              #6 divisão para obter o índice.
return(indice)             #7 para mandar para fora da função.
}                          #8 fecha a função.
```

4.2 Ingredientes e receita de uma função. **`f1.simp()`**: Índice diversidade de Simpson

As funções que criaremos neste livro possuem basicamente a mesma estrutura:

nome: note que gravamos uma função da mesma forma que gravamos outros tipos de objetos, com uma atribuição '**`<-`**'. Nomeamos nossa função de '**`f1.simp`**'. Poderia ser outro nome. Você escolhe. Existem algumas restrições. Não pode iniciar com números ou '_', não pode ter hífen (usado para subtração) e não é conveniente usar nomes de outras funções já existentes no R (de maneira simplista, pois 'esconde' a função do R). Acentos e cedilha também podem gerar problemas. Pode usar '**`.`**' e '_' no meio do nome. Maiúsculas e minúsculas são interpretadas como caracteres distintos. Para

não se confundir, nomeie suas funções e outros objetos apenas com letras minúsculas. Caso sejam duas palavras, una-as com ponto ou use a notacaoCamelo.

{ }: uma função (ou programa) nada mais é do que uma sequência de instruções para o computador. Para que o R entenda que todas instruções fazem parte de uma mesma função, usamos chaves '**{ }**'. Sempre feche a chave após abri-la. Escreva entre as chaves. Não deixe para fechar depois, pois vai esquecer. Idem para parênteses. Note que no RStudio e outros editores o fechamento é automático e isto evita muitos erros quando estamos iniciando no R.

argumento: nossa função possui um único argumento, que chamamos de '**abund**'. Um índice de diversidade precisa de um vetor com abundâncias das espécies. Este argumento serve para isto. É a forma do usuário enviar uma informação para a função. Note que esta informação é usada dentro da função (linhas 2 e 3). Note também que seu objeto com dados não precisa chamar '**abund**'. Isto é o nome do argumento (assim como '**x**' e '**base**' quando vimos a função **log()**). Não importa o nome que use para seus dados, ele será passado para o interior da função como '**abund**'.

corpo: conforme dito acima, uma função nada mais é do que uma sequência de instruções para o computador. Esta sequência segue uma lógica. Por exemplo, não podemos inverter a linha 6 com a 2. Não temos como calcular D na linha 2 pois precisamos de outras informações que ainda não estão disponíveis. Portanto, geralmente iniciamos obtendo os 'ingredientes' ou informações básicas necessárias para outras partes da função.

saída: os argumentos são formas de entrada de informações. Servem para instruir a função como se comportar. Precisamos do resultado final, a saída. Fazemos isto com a função **return()**. Você notará que nada do que é criado dentro da função (por ex. '**total**', '**prop.i**', '**prop.i.2**' na função **f1.simp()**) fica disponível para o usuário. Usamos o **return()** para enviar para o usuário o que interessa. Na função **f1.simp()**, apenas um valor, chamado de '**indice**' é enviado para o usuário.

Não é interessante escrever funções diretamente no R. Geralmente usamos editores de texto, como aquele disponível no RStudio. Isto pois precisamos testar passos intermediários para ver se fazem o que queremos. Se você abre uma chave no R diretamente, ele só mostrará o resultado quando ela for fechada. Isto inviabiliza o teste de um passo em particular. Escreva a função acima num arquivo (pode ser o script fornecido com esta apostila). Mande para o R com Ctrl+R (se usar no R) ou Ctrl+Enter (RStudio).

Agora podemos usar ou 'rodar' a função. Suponha que Adorinda coletou uma parcela numa mata no Paraná e encontrou 4 espécies de serpentes com estas abundâncias:

```
mata.parana <- c(3, 7, 10, 2)
```

Podemos usar nossa função para calcular o índice de diversidade de Simpson para a mata:

```
f1.simp(abund=mata.parana)
[1] 2.987654
```

Note que nossa função não serve apenas para a mata do Paraná. Ela serve para qualquer conjunto de dados que atenda algumas restrições. Por exemplo, os cálculos da função necessitam de um vetor e, portanto, nossos dados devem ser um vetor. Conseguimos fazer com que nossa função possa ser usada para outros conjuntos de dados pois não inserimos no corpo do seu código informações específicas de um conjunto de dados. Por exemplo, precisamos da soma dos valores para posteriormente calcularmos as proporções. Não colocamos este valor de soma dentro do corpo da função. Calculamos este valor dentro do corpo da função. Colocar valores de um dado conjunto de dados dentro de uma função é um erro comum (não caia nele!).

O exemplo que vimos resume muito, mas muito mesmo, da criação de funções dentro do R. Passe um tempo trabalhando com a função. Tenha certeza que entendeu tudo, mas tudo mesmo.

4.3 Exercício resolvido. `f2.correl()`: Coeficiente de correlação de Pearson

É provável que você já tenha visto ou mesmo calculado o coeficiente de correlação de Pearson. Ele serve para avaliar o quanto duas variáveis quantitativas estão associadas. O coeficiente varia de -1 a 1. Valores próximos de 1 indicam que quando uma variável aumenta a outra também aumenta. Valores próximos de -1 indicam que quando uma variável aumenta a outra diminui. Quanto mais próximo de 1 ou -1, mais forte é esta relação. Valores, positivos ou negativos, próximos de 0 indicam ausência de associação. A fórmula do coeficiente pode parecer complicada à primeira vista.
Não é.

$$r = \frac{\sum[(X_i - \overline{X}) * (Y_i - \overline{Y})]}{\sqrt{[(\sum(X_i - \overline{X})^2) * (\sum(Y_i - \overline{Y})^2)]}}$$

X_i e Y_i = valores i das variáveis X e Y.
X e Y com barras = médias de X e Y.

Podemos fazer algo bastante útil para iniciarmos, um pseudo-código. Um código em palavras e não em funções. Ou seja, nos preocupamos inicialmente com a lógica da resolução. Depois nos preocupamos com as funções que fazem cada passo.

#1 dar nome e criar função. Abrir chaves.
#2 reunir 'ingredientes'. O primeiro poderia ser a média de x.
#3 o segundo poderia ser a média de y.
#4 diferenças de cada xi em relação à média de x.
#5 idem acima, mas para y.
#6 multiplicar os dois termos dentro de parênteses no numerador.
#7 somar valores obtidos no passo anterior. Isto é o numerador.
#8 já temos as diferenças de x em relação à média (passo 4). Elevar ao quadrado.
#9 somar valores obtidos no passo anterior.
#10 idem passo 8, mas para y.
#11 idem passo 9, mas para y.
#12 multiplicar as somas obtidas em 9 e 11.
#13 tirar raiz quadrada do passo anterior. Já temos o denominador.
#14 calcular o coeficiente r.
#15 enviar o coeficiente para o usuário.

Veja os passos de nosso pseudo-código acima. Quais funções precisaremos? Estas: **`function()`**; **`mean()`**; **`sum()`**; **`sqrt()`**; **`return()`** e as operações aritméticas básicas: **`-`**; **`*`**; **`/`**; **`^`**

Vamos completar a segunda fase, fazer a função:

```
f2.correl <- function(x, y){#1 nome, criar função, chaves. Função precisa de 2 argumentos.
x.media   <- mean(x)        #2 reunir os 'ingredientes'. O primeiro poderia ser a média de x.
y.media   <- mean(y)        #3 o segundo poderia ser a média de y.
x.dif     <- x-x.media      #4 diferenças de cada xi em relação a média de x.
y.dif     <- y-y.media      #5 idem acima, mas para y.
xy.dif    <- x.dif*y.dif    #6 multiplicar os dois termos dentro de parênteses no numerador.
numerador <- sum(xy.dif)    #7 somar valores obtidos no passo anterior. Isto é o numerador.
x.dif2    <- x.dif^2        #8 temos as difs de x – média de x (passo 4). Elevar ao quadrado.
x.dif2.sum <- sum(x.dif2)   #9 somar valores obtidos no passo anterior.
y.dif2    <- y.dif^2        #10 idem passo 8, mas para y.
y.dif2.sum <- sum(y.dif2)   #11 idem passo 9, mas para y.
denom.sem.sqrt <- x.dif2.sum*y.dif2.sum   #12 multiplicar as somas obtidas em 9 e 11.
denom     <- sqrt(denom.sem.sqrt)         #13 tirar raiz quadrada para ter denominador.
r         <- numerador/denom              #14 calcular o coeficiente r.
```

```
return(r)                    #15 enviar o coeficiente para o usuário.
}                            #16 fecha função
```

Agora mandamos todo o código para o R. Caso receba uma mensagem de erro, leia a próxima seção.

Suponha que Albina queira avaliar a associação entre riqueza de anfíbios e tamanho da população humana. Ela obteve 8 quadrículas de 1º latitude x 1º longitude no Cerrado e em cada uma obteve a riqueza de espécies de anfíbios e o tamanho da população humana (vejo que estamos avaliando uma associação e não inferindo causa-e-efeito; veja Araújo [2003] para um estudo real deste assunto). Existe associação entre riqueza de anfíbios e população humana no Cerrado?

```
riq.anfibios <- c( 50,  83,  79,  41, 33, 29,  45,  36)
pop.humana <-  c(300, 105, 990, 250, 77, 82, 190, 120) # vezes mil habitantes
f2.correl(x=riq.anfibios, y=pop.humana)
[1] 0.5780908
```

Nossa função produziu o valor de 0,578. Indica que existe associação positiva razoável. Mas, antes de mais nada, podemos confiar neste valor de 0,578? Vamos criar uma função para testá-lo (ver se um valor igual ou superior a este não poderia ser gerado facilmente pelo acaso com 8 pares de valores) por aleatorização no capítulo 8.

4.4 Erros: temos que resolvê-los (ou, os sutis são os piores)

É comum aparecerem erros quando fazemos funções. Não se desespere. Veja (com muita atenção) a mensagem de erro e tente localizar o que digitou errado. Corrija e mande a função novamente para o R. Se ainda não deu certo, veja novamente a mensagem e quebre um pouco a cabeça procurando o problema. Você será um bom programador quando conseguir descobrir rapidamente boa parte dos seus erros. Isto é uma parte fundamental do seu treinamento. Não fique triste nem perca a paciência com erros. Eles fazem parte.

Geralmente, os erros podem aparecer em três fases. A primeira é quando mandamos o código da função para o R. O R avalia cada linha de código e confere parênteses, chaves, colchetes (veremos para que serve em outro capítulo), espaços em locais inadequados etc. É relativamente comum haver erros na primeira vez que mandamos a função para o R. Mas não é tão difícil corrigir. Basta um pouco de prática e atenção às mensagens de erros que o R mostra.

A segunda fase é quando rodamos (esta palavra é um tanto informal, uma gíria derivada do

Inglês, mas é tão usada que nem vou colocar aspas...) nossa função. Se tiver, novamente, veja o erro mostrado e tente consertar. Erros comuns aqui são: i) uso de funções não disponíveis (por ex. funções em pacotes não carregados), ii) grava objeto com um nome, mas usa com outro nome abaixo e iii) operações não permitidas (por ex. divide Y por X, mas um elemento de X é zero).

As duas fases acima em geral são tranquilas. A terceira fase é a mais complicada. É quando o R roda sua função e não te dá nenhuma mensagem de erro. Aqui, o problema é saber se fizemos a coisa certa. Na seção anterior, Albina obteve o valor de 0,578 com a função **`f2.correl()`**. O fato do R não apontar erros indica que não existem erros na ortografia e gramática. Não quer dizer que o significado esteja correto. Podemos ter feito uma operação matemática válida, mas diferente do que deveríamos fazer. Achar estes erros pode ser um pouco mais complicado, em particular em situações onde a função funciona corretamente (produz valor correto) com alguns conjuntos de dados, mas não com outros.

Para erros nas duas primeiras fases, uma solução simples mas um pouco trabalhosa é rodar cada linha do seu código separadamente e em cada uma conferir o objeto produzido. Ele está de acordo com o que você espera?

Para erros na terceira fase, quando o R não aponta erros e produz um resultado, podemos comparar os resultados com algum programa confiável já disponível. Para nossa função **`f2.correl()`**, podemos usar:

```
cor(riq.anfibios, pop.humana)
[1] 0.5780908
```

O mesmo resultado de nossa função. Você não tem 100% de certeza que ela está correta, mas já pode ficar mais feliz. Digo que não é 100% de certeza pois podem aparecer erros com outros conjuntos de dados. É por isto que disse acima que as duas primeiras fases eram mais tranquilas. Isto pois sabemos que erramos e podemos consertar nossos erros. E quando não sabe que existe um erro? É também por isto que desenvolvedores geralmente mandam seus programas para usuários testarem antes de distribuírem amplamente.

4.5 Lista acumulada de conceitos e funções usadas

Conceitos e funções vistos pela primeira vez neste capítulo estão em negrito.

área de trabalho, .RData e .Rhistory
funções: uso geral
uso de: parênteses, chaves
argumentos, separação de argumentos com vírgula
uso de ordem dos argumentos em vez dos nomes dos argumentos

argumentos padrão (ou default)
atribuição
script
comentários
objetos: vector, **function**
pacotes
funções: estrutura,
funções: pseudo-código
funções: trabalhando com erros

operações básicas: +; -; /; *; ^ log() exp() ? ?? c() sum() length() mean() sqrt() rep(); seq(); : ls(); rm() **function()**; **return()** **cor()**	**f1.simp()** **f2.correl()**

5 Exercícios (quase) resolvidos

5.1 Como aprendemos programação?

'Missão dada, parceiro, é missão cumprida!
Estamos entendidos?'
Coronel Nascimento

Programando! Fazendo exercícios, praticando. Não tem outra forma. Estudantes da grande área de ciências biológicas estão acostumados a ler para estudar. Com exceção de alguns raros gênios, não funciona quando colocamos um número junto com uma letra. Temos que praticar com exercícios. Isto não é uma descoberta minha... É extremamente comum disciplinas na área de exatas terem listas de exercícios em quase todas aulas. Você até pode achar que entendeu a aula expositiva do professor, mas não conseguirá fazer a prova se não fizer as listas de exercícios.

Até este momento vimos 15 funções e os operadores aritméticos básicos. É um número bem restrito. Entretanto, isto está de acordo com o que havia dito no capítulo introdutório. Apesar disto, já podemos fazer funções interessantes. Este capítulo é fundamental. Você deve dominá-lo bem antes de prosseguir.

5.2 **f3.dp()**: Desvio padrão de uma amostra

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

Faça uma função para calcular o desvio padrão de uma amostra.

DP = desvio padrão;

x_i = valor da observação i;

x-barra = média das observações;

n = número de observações.

```
f3.dp <- function(amostra){# cria função com 1 argumento
                           # média da amostra (media)
                           # diferenças dos valores em amostra em relação à média (dif)
                           # eleva as diferenças ao quadrado (dif2)
```

```
                                    # soma as diferenças ao quadrado (numerador)
                                    # obtém número de observações (n)
                                    # calcula o denominador
                                    # faz a divisão
                                    # tira raiz quadrada
                                    # retorna resultado para usuário
}
```

Calcule o desvio padrão para as abundâncias das espécies na parcela obtida na mata do Paraná por Adorinda:

```
mata.parana <- c(3, 7, 10, 2)
f3.dp(amostra=mata.parana)
```

Resultado: 3,696846.

5.3 `f4.shannon()`: Índice de diversidade de Shannon

Adorinda, aquela pesquisadora do Paraná, estava interessada em calcular o índice de diversidade de Shannon para sua parcela:

$$H' = -\sum p_i \ln p_i$$

H' = valor do índice;
p_i = abundância relativa da espécie i;
ln = logaritmo natural.

Crie uma função:

```
f4.shannon <- {# inicie a função. Ela deverá ter 1 argumento, para abundâncias.
               # some abundâncias para obter abundância total
               # obtenha as abundâncias relativas de cada espécie (prop_i)
               # obtenha o logaritmo natural das prop_i
               # multiplique as prop_i pelo ln das prop_i
```

```
                    # some a multiplicação anterior
                    # coloque o sinal negativo e depois a soma obtida antes
                    # devolva para o usuário
}

 mata.parana <- c(3, 7, 10, 2)
 f4.shannon(mata.parana)
```

Resultado: 1,212435.

5.4 `f5.hill()`: Série de diversidade de Hill

Existem dezenas de índices de diversidade na literatura. Na verdade, podemos ter infinitos índices de diversidade. Isto pois um índice de diversidade basicamente combina riqueza com equabilidade. A diferença entre eles é o peso que se dá para cada componente. Portanto, podemos pensar que riqueza de espécies é um índice de diversidade; aquele em que o peso para equabilidade é zero. Todos índices de diversidade são, portanto, muito parecidos entre si e podemos expressar isto com uma série ou perfis de diversidade (Tóthmérész 1995, Melo 2008). Existem várias séries. Uma delas é a de Hill e tem a forma:

$$N_a = \left(p^{a}_{1} + p^{a}_{2} + p^{a}_{3} \ldots + p^{a}_{n}\right)^{1/(1-a)}$$

N_a = Índice de diversidade 'a';
p_i = Abundância relativa da espécie i até n;
a = Parâmetro da série. Valores maiores dão maior peso para equabilidade. Valores devem ser iguais ou maiores que zero exceto 1 (pode ser 0,9999 ou 1,0001).

```
 f5.hill <- function(abunds, a){       #
                    # já deve estar ficando fácil...
                    # deixo para você fazer sua própria 'receitinha'.
                    # use quantas linhas quiser.
 }
```

Adorinda (novamente ela...) quer saber qual seria o índice de diversidade na série de Hill com a=0 para aqueles dados do Paraná:

```
 mata.parana <- c(3, 7, 10, 2)
```

Ela também ficou curiosa para saber os valores com a=0,9999 e a=2. Poderia calcular para ela? Respostas: (a=0) 4, (a=0,9999) 3,361711 e (a=2) 2,987654. Alguma semelhança com isto:

```
length(mata.parana)
shannon.parana <- f4.shannon(mata.parana)
exp(shannon.parana)
f1.simp(mata.parana)
```

Qual o motivo de não poder usar a=1? Não é muito difícil descobrir olhando na formula. Se pudéssemos colocar **a=1**, o valor produzido com a série de Hill seria idêntico àquele obtido com o **exp()** da nossa função **f4.shannon()**, e não apenas um valor aproximado.

5.5 **f6.cole()**: Modelo de amostragem passiva em Biogeografia de Ilhas

A Teoria de Biogeografia de Ilhas, proposta por MacArthur & Wilson (1967), prevê que a riqueza de espécies numa ilha é um balanço entre imigrações e extinções. Segundo a teoria, este balanço deve ser afetado positivamente por proximidade do continente (fonte de colonizadores) e tamanho da ilha (maior disponibilidade de recursos, maior número de habitats). O modelo se tornou bastante popular imediatamente após sua proposição e foi estendido para vários sistemas insulares. Wilson (1994) conta algumas histórias bacanas sobre o desenvolvimento (com MacArthur) e teste inicial da teoria (com Simberloff) em sua fascinante autobiografia. Outros relatos, não menos fascinantes, podem ser encontrados em Quammen (2008).

Poderíamos pensar num modelo alternativo em que apenas área seja importante. Ainda, que seja importante de maneira proporcional à sua área em relação à área de um arquipélago (Coleman et al. 1982). Este exercício foi baseado em Gotelli (2007; pp. 173-175), onde explicação mais detalhada pode ser encontrada. Neste modelo, a riqueza de espécies numa ilha não só é proporcional a sua área, como a presença de uma espécie numa ilha é proporcional a sua abundância total. Espécies comuns, com muitos indivíduos, ocuparão muitas ilhas. O objetivo deste modelo é estimar quantas espécies vivem numa ilha de um dado tamanho. O modelo é:

$$E(S_i) = \sum_{j=1}^{S} [1 - (1 - x_i)^{n_j}]$$

$E(S_i)$ = Expectativa da riqueza de espécies na ilha i;

S = Total de espécies na região e que podem colonizar uma dada ilha;

x_i = Área relativa (proporção em relação a área total do arquipélago);

n_j = Abundância total da espécie j no arquipélago.

A função deverá ter como argumentos:

areas: áreas das ilhas;

abunds: abundâncias totais das espécies;

area.i: área da ilha i, ou seja, da ilha para a qual queremos estimar a riqueza.

Crie a função usando este pseudo-código:

```
f6.cole <- {     # crie a função. Deve ter argumentos para areas, abunds, area.i
            # obtenha área total do arquipélago
            # obtenha a área relativa da ilha i (xi) (aquela que tem interesse)
            # obtenha complemento da área relativa: 1-xi
            # eleve o complemento da área a todas abundâncias das espécies (1-xi)^nj
            # faça a subtração 1 - o passo anterior
            # some o obtido no passo anterior.
            # passe o resultado (passo anterior) para o usuário
}
```

Alba estudou lagartos em 5 ilhas de um arquipélago. As ilhas tinham as seguintes áreas: 10, 28, 32, 4, 2. Suponha que existam 7 espécies de lagartos no arquipélago e estas tinham como abundância total: 100, 900, 380, 1500, 50, 800, 20. Quantas espécies Alba deve encontrar na ilha com área 28? E na ilha de área 2? Respostas: 6,999898 e 6,080284 espécies.

5.6 **f7.F2()**: Estatística F na comparação das média de duas amostras (teste t)

Normalmente pensamos na estatística t quando falamos numa comparação de duas médias. De fato, chamamos a comparação de teste t. Muitos alunos imaginam que análise de variância (quando temos mais do que duas médias a comparar) seja completamente distinta de um teste t (afinal, os nomes são). Na verdade, as duas e muitas outras análises que você conhece ou ouviu falar (regressão linear simples ou múltipla, análise de covariância etc) são todas 'farinha do mesmo saco'. São modelos lineares. Aliás, $F = t^2$.

Faça uma função para calcular o valor de F de uma comparação entre duas amostras. Para tanto, obtenha os componentes de uma tabela de análise de variância.

$$SQT = \sum_{i=1}^{k}\sum_{j=1}^{ni}\left(y_{ij} - \bar{y}\right)^2 \quad SQE = \sum_{i=1}^{k}\left[\sum_{j=1}^{ni}\left(y_{ij} - \bar{y}_i\right)^2\right]$$

k = número de tratamentos (2 no teste t);

n_i = número de observações no tratamento i;

y_{ij} = cada uma das observações;

y-barra = média geral (observações de todos tratamentos);

y_i-barra - média do tratamento i.

Fazemos uma análise de variância decompondo a variação total em dois componentes, uma devido ao modelo de regressão e outra devido ao erro: SQT = SQR + SQE. SQ significa soma de quadrados. Dizemos 'modelo de regressão' pois sempre temos um modelo. No caso de uma regressão linear, este modelo é uma reta. No caso de variáveis explanatórias categóricas (teste t, anova) o modelo é, simplesmente, a média de cada tratamento.

Note que você obteve uma soma de quadrado quando você programou a f3.dp(). Visto que tendo 2 valores conseguimos o terceiro, vamos calcular apenas SQT e SQE. Após calcular SQT, SQE e SQR (por diferença dos outros dois), precisamos calcular os quadrados médios (que nada mais são do que variâncias): QMR e QME, respectivamente quadrado médio do modelo de regressão e do erro. Para tanto, basta dividir cada soma de quadrados por seu grau de liberdade. Na fórmula da variância de uma amostra, o valor é n-1. No caso da SQR para um teste t, o grau de liberdade sempre será 1 e para o SQE será n-2 (Tab. 5.1).

Tabela 5.1 Tabela de Análise de Variância.

Fonte	Soma de quadrados	Graus de liberdade	Quadrados médios	F
Regressão	SQR	1	QMR=SQR/1	QMR/QME
Resíduo (erro)	SQE	n-2	QME=SQE/n-2	
Total	SQT=SQR+SQE	n-1		

Esta sugestão de pseudo-código apenas lista os blocos fundamentais. Use quantas linhas achar necessário para obter cada bloco:

```
f7.F2<-function(k1, k2){
                    # junte as duas amostras.
                    # obtenha a média geral (ybarra).
```

```
                # obtenha a SQT.
                # obtenha a média do primeiro tratamento (y1).
                # obtenha a soma de SQE parcial para y1.
                # obtenha a média do segundo tratamento (y2).
                # obtenha a soma de SQE parcial para y2.
                # obtenha a SQE.
                # obtenha a SQR.
                # obtenha a QMR.
                # obtenha a QME (obtenha n dentro da função).
                # obtenha F.
                # retorne F para usuário.
}
```

A professora Assunta estava interessada em saber qual de duas estratégias era mais efetiva para ensinar programação no R. A primeira estratégia foi aplicada a 8 alunos de uma turma e consistia em explicar 40 funções na primeira aula. As aulas subsequentes eram usadas para treinar o uso das funções. A segunda estratégia foi aplicada a 8 alunos de outra turma (mas semelhante à primeira turma) e consistia em expor 10 funções na primeira aula, usá-las em exercícios nas aulas seguintes e só nas aulas finais expor gradualmente as outras 30 funções. As notas dos alunos numa prova aplicada no final dos dois cursos foram:

```
porrada <- c(2, 6, 3, 10, 9, 1, 0, 3)
maciota <- c(5, 7, 4, 6, 10, 10, 8, 9)
```

A professora obteve um valor relativamente alto, F=4,163. Conseguiu achar isto? Podemos testar este valor por aleatorização no Capítulo 8. Embora não faça parte do assunto deste livro, percebeu que esta análise é baseada em pseudo-replicação?

5.8 Lista acumulada de conceitos e funções usadas

Conceitos e funções vistos pela primeira vez neste capítulo estão em negrito.

área de trabalho, .RData e .Rhistory
funções: uso geral
uso de: parênteses, chaves
argumentos, separação de argumentos com vírgula

uso de ordem dos argumentos em vez dos nomes dos argumentos
argumentos padrão (ou default)
atribuição
script
comentários
objetos: `vector`, `function`
pacotes
funções: estrutura
funções: pseudo-código
funções: trabalhando com erros

operações básicas: +; -; /; *; ^
`log()`
`exp()`
`?`
`??`
`c()`
`sum()`
`length()`
`mean()`
`sqrt()`
`rep(); seq(); :`
`ls(); rm()`
`function(); return()`
`cor()`
	`f1.simp()`
	`f2.correl()`
	`f3.dp()`
	`f4.shannon()`
	`f5.hill()`
	`f6.cole()`
	`f7.F2()`

6 Testes booleanos e avaliação de condições

6.1 Operadores relacionais, lógicos e funções relacionadas

Testes booleanos são aqueles que oferecem respostas binárias, geralmente FALSE ou TRUE. Os operadores mais comuns para estes testes são:

```
acacia <- 2
acacia > 5
[1] FALSE

acacia < 5
[1] TRUE

acacia == 2
[1] TRUE

acacia > 2
[1] FALSE

acacia >= 2 # Note que o sinal de igualdade sempre vem depois.
[1] TRUE

acacia != 2 # Note que o sinal de igualdade sempre vem depois.
[1] FALSE
```

Note que o sinal "=" não faz uma comparação. Ele é um substituto do comando para atribuir um nome a um objeto:

```
adosinda = 3*3
adosinda
[1] 9
```

O uso da flechinha "<-" é muito mais comum do que o sinal de igualdade e, portanto, vamos

usá-la neste livro. O R opera sobre vetores. Portanto, podemos ampliar os operadores acima para vetores com comprimento maior que 1:

```
 acacia.seq <- 1:5
 acacia.seq > 3
[1] FALSE FALSE FALSE  TRUE  TRUE

 acacia.seq >= 3  # Note que o sinal de igualdade sempre vem depois.
 [1] FALSE FALSE TRUE  TRUE  TRUE

 acacia.seq <= 3
 [1] TRUE TRUE TRUE  FALSE  FALSE
```

Também podemos avaliar dois testes simultaneamente, ou seja, um teste do tipo 'e':

```
 acacia.seq
 [1] 1 2 3 4 5

 adosinda.seq <- 11:15

 acacia.seq > 3 & adosinda.seq < 20

[1] FALSE FALSE  FALSE TRUE TRUE
```

O teste acima avalia cada elemento em cada um dos vetores. Para que seja TRUE, o resultado tem que ser verdadeiro nos dois testes. Fica implícito que os dois vetores usados nos testes devem ter o mesmo comprimento (experimente com **`adosinda.seq <- 11:16`**). Além dos testes tipo 'e', temos os testes do tipo 'ou':

```
 acacia.seq
 [1] 1 2 3 4 5
 adosinda.seq
 [1] 11 12 13 14 15

 acacia.seq > 2 | adosinda.seq > 14
 [1] FALSE  FALSE TRUE  TRUE TRUE
```

Também podemos usar estas funções para saber se pelo menos um ou todos elementos são

verdadeiros:

```
acacia.seq
[1] 1 2 3 4 5
any(acacia.seq==2)
[1] TRUE
all(acacia.seq==2)
[1] FALSE
```

6.2 Coerção de dados booleanos

'Pede pra sair, senão vai sair debaixo de porrada!"

(Capitão Nascimento).

Frequentemente precisamos transformar os resultados de testes booleanos em números. A função **as.numeric()** faz uma coerção. Poderíamos ler como 'seja numérico'. No R, por convenção, a coerção de FALSE produz 0 e TRUE produz 1.

```
acacia.seq
[1] 1 2 3 4 5
acacia.seq.resu <- acacia.seq > 2
acacia.seq.resu
[1] FALSE FALSE  TRUE  TRUE  TRUE
as.numeric(acacia.seq.resu)
[1] 0 0 1 1 1
```

6.3 Avaliação de condições

Muitas vezes precisamos aplicar uma dada operação em apenas algumas situações especiais. Por exemplo, para uma função que usa como argumento de entrada um vetor contendo apenas valores zero e um, podemos transformar todos valores acima de zero em um:

```
aldara <- c(0,1,0,4)
ifelse(aldara>0, yes=1, no=0)
[1] 0 1 0 1
```

Outra possibilidade é manter o valor original caso não queiramos fazer qualquer mudança. Por

exemplo:

```
aldara <- c(0,1,2,3)
ifelse(aldara>2, 100, aldara)
[1] 0 1 2 100
```

A função **ifelse()** é uma forma abreviada de um sistema mais amplo de avaliação de condições. A função geral é **if()** e é usada assim:

```
 acacia <- 2
if(acacia==42){
   print('Era um número mágico...')
   acacia <- log(acacia+1)
   print('... agora não é mais')
}
acacia
```

Podemos ler os comandos acima como: Se acacia é igual a 42, imprima (função **print()**) na tela 'Era um número mágico...', some 1 ao valor 42, tire log e finalmente imprima na tela '... agora não é mais'. Essa sequência de comandos só será aplicada se o resultado da avaliação **acacia==42** for **TRUE**. Caso contrário, se for **FALSE**, a função não faz nada. Com o exemplo abaixo, a mensagem só será colocada na tela se houver um valor negativo:

```
if(acacia<0){
   print('Atenção: o valor é negativo')
}
```

Nos casos acima, uma ação é tomada apenas no caso da condição ser verdadeira. Podemos estender isto para duas ações, ou seja, aplicar uma ação se for verdadeiro e outra ação se for falso:

```
if(acacia<0){
   print('Atenção: havia um(vários) valor(es) negativo (s)')
   acacia <- abs(acacia)# módulo
    print('Mas ok, todos foram passados para positivo')
}  else{
      print('Não haviam valores negativos')
   }
```

Note que o uso do **`else`** é um pouco diferente. Ele não é exatamente uma função no sentido de não usarmos parênteses para ele. Ele é usado para indicar o que fazer quando o teste lógico usado no **`if()`** é **`FALSE`**.

Note também que usamos chaves **`{}`** para a função **`if()`**. Já havíamos usado chaves quando fizemos funções. De maneira geral, chaves servem para agrupar várias linhas de comando. Uma última coisa no exemplo acima é que em algumas linhas usamos espaços antes das ações. Em R, estes espaços não fazem diferença para o resultado das funções. Entretanto, eles são bastante úteis para organizarmos nossas funções. No exemplo acima, sabemos que a função **`print()`** está dentro das chaves do comando **`if()`**. Pode parecer um detalhe aqui, mas faz uma boa diferença quando temos várias linhas de código.

6.4 Exercício resolvido. **`f8.dissi.par.quali()`**: Índice de dissimilaridade de Sorensen ou Jaccard

Nascemos com a mania de comparar. Comparamos coisas relativamente simples como preços. Neste caso, temos apenas uma variável quantitativa e, portanto, as coisas são fáceis. Ou os preços são iguais ou diferentes. Mas também comparamos coisas um pouco mais complexas, como dieta e aparência das pessoas. Nestes últimos, temos mais de uma variável a comparar. Dieta não se resume a uma variável (por ex. leite), mas muitas (leite, carne, cereais etc). É algo multivariado. Idem para aparência. Várias áreas da Ciência usam índices de similaridade para expressar o quanto dois objetos são parecidos. Em nutrição, podemos comparar as dietas das pessoas e ver que par de pessoas possui dieta mais semelhante. Em ecologia de comunidades, podemos comparar áreas em relação a espécies de plantas. Podemos comparar espécies de roedores em relação a fauna de parasitas.

Antes de mais nada, é fundamental ter claro o que é objeto e o que são os descritores (ou variáveis). Nos exemplos multivariados acima, os objetos eram pessoas, áreas e espécies de roedores. Para estes objetos, usamos como descritores, respectivamente, itens da dieta, espécies de plantas e espécies de parasitas.

Índices de similaridade geralmente variam entre 0 e 1. Para estes, podemos chamar seu complemento de dissimilaridade. Ou seja, dissimilaridade = (1 – similaridade). Em ecologia de comunidades, as duas formas são usadas, embora análises dentro do R geralmente usem dissimilaridades. Vamos usar dissimilaridades neste livro. O importante, entretanto, é saber que tendo um temos o outro.

Índices de dissimilaridade podem usar dados qualitativos ou quantitativos. Dados qualitativos geralmente são expressos como 0 (descritor ausente) ou 1 (descritor presente). Entretanto, 0 e 1 também podem ser usados para expressar dois estados diferentes de uma variável. Em ecologia de comunidades, geralmente estamos querendo comparar áreas em relação à composição de espécies. Para simplificar, vamos usar apenas duas áreas. Os dados geralmente tem este formato:

Áreas	sp1	sp2	sp3	sp4	sp5
area.go	1	0	1	0	1
area.rs	0	1	1	1	0

```
area.go <- c(1,0,1,0,1)
area.rs <- c(0,1,1,1,0)
```

Vamos criar uma função para calcular dois dos índices de dissimilaridade mais comuns em ecologia de comunidades, Jaccard e Sorensen. Estes índices, e outros qualitativos, usam as seguintes informações:

		Objeto 2	
		presente	ausente
Objeto 1	presente	a	b
	suente	c	d

a = número de variáveis compartilhadas pelos dois objetos;
b, c = número de variáveis presentes em um objeto, mas não no outro;
d = número de variáveis ausentes nos dois objetos.

Os índices de Sorensen e Jaccard, assim como a maioria daqueles usados em ecologia de comunidades não usam a informação d. Ela indica, por exemplo, que o fato da Mata Atlântica e a Amazônia não terem pinguins as tornam semelhantes. O índice de dissimilaridade de Sorensen pode ser obtido como Sor = 1 – (2a/(2a+b+c)) e o de Jaccard como Jac = 1 - (a/(a+b+c)). Vamos fazer uma função com possibilidade de calcular os dois índices para um par de objetos. A entrada dos dados será

feita separadamente, um argumento para cada objeto. Ainda ela terá a opção de calcular um índice ou o outro.

```
f8.dissi.par.quali <- function(obj1, obj2, qual='jac'){
    # caso o usuário não coloque nada no argumento 'qual', a função usará como padrão o 'jac'.
    # precisamos encontrar as informações a, b e c. Para a, uma solução seria:
soma <- obj1 + obj2 # note que sum() não funcionaria aqui. Experimente e veja a razão.
quais2 <- soma==2 # 2 indica as variáveis descritoras que estão presentes nos dois objetos
quais2.num <- as.numeric(quais2) # fazendo coerção para transformar em 0 e 1.
a <- sum(quais2.num)
  # agora precisamos de b e c, mas note na formula que não precisamos separadamente:
quais1 <- soma==1 # 1 indica que está presente num objeto ou no outro, mas não nos dois.
quais1.num <- as.numeric(quais1)
b.c <- sum(quais1.num)
  # agora temos nossos 'ingredientes' para calcular. Só falta saber o que calcular:
if(qual == 'sor'){
  dois.a <- 2*a
  indice <- dois.a/(dois.a + b.c)
}
if(qual == 'jac'){
  indice <- a/(a+b.c)
}

return(1-indice) #para ficar como dissimilaridade
}
```

A dissimilaridade de Jaccard para o par de áreas descritas acima foi 0,8. A dissimilaridade de Sorensen foi 0,67.

6.5 Exercícios

6.5.1 **f9.ind.div()**: Índice de diversidade de Simpson ou Shannon. Crie uma função que calcule o índice de diversidade de Simpson ou de Shannon. Você pode aproveitar parte dos códigos das funções **f1.simp()** e **f4.shannon()** se quiser.

```
f9.ind.div <- function(dados, indice='simp'){ # complete o restante do código
}
```

6.5.2 **f10.nodf.par()**: Índice de aninhamento NODF. Dizemos que A é aninhado em B se A for um subconjunto de B. Em ecologia de comunidades, podemos usar o índice NODF (Nested Overlap and Decreasing Fill; Almeida-Neto et al. 2008) para avaliar quanto uma comunidade está aninhada em outra. O índice possui dois passos. No primeiro avaliamos se a comunidade supostamente mais rica em espécies (vamos chamá-la de supost.rica) de fato possui mais espécies do que aquela supostamente mais pobre (supost.pobre). Se isto for falso, o valor do índice é 0. Se for verdadeiro, passamos para o segundo passo que é basicamente a proporção de espécies presentes em supost.pobre que estão presentes em supost.rica. Se todas presentes em supost.pobre estiverem em supost.ricas, o valor é 1. Se nenhuma estiver em supost.rica, o valor é 0.

```
f10.nodf.par <- function(supost.rica, supost.pobre){
}
```

6.6 Lista acumulada de conceitos e funções usadas

Conceitos e funções vistos pela primeira vez neste capítulo estão em negrito.

área de trabalho, .RData e .Rhistory
funções: uso geral
uso de: parênteses, chaves
argumentos, separação de argumentos com vírgula
uso de ordem dos argumentos em vez dos nomes dos argumentos
argumentos padrão (ou default)
atribuição
script
comentários
objetos: `vector`, `function`
pacotes
funções: estrutura
funções: pseudo-código
funções: trabalhando com erros
coerção

operações básicas: +; -; /; *; ^ operadores relacionais e lógicos: **>**; **<**; **==**; **>=**; **<=**; **!=**; **&**; **|** **any()**; **all()** log() exp() ?	f1.simp() f2.correl() f3.dp() f4.shannon() f5.hill() f6.cole()

?? c() sum() length() mean() sqrt() rep(); seq(); : ls(); rm() function(); return() cor() **as.numeric()** **ifelse()**; **if()**; **if() else** **abs()** **print()**	f7.F2() **f8.dissi.par.quali()** **f9.ind.div()** **f10.nodf.par()**

7 Objetos: **matrix, data.frame e list**

7.1 **matrix** e **data.frame**

matrix: Até este momento trabalhamos com dois tipos de objetos, **function** e **vector**. Podemos pensar em **matrix** como uma extensão de **vector**, onde temos duas dimensões. Estas dimensões são linhas e colunas. Podemos criar uma **matrix** com:

```
matrix(data=1:10, nrow=2) # sequência de 1 a 10 com 2 linhas e, portanto, 5 colunas.
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Note que a matrix foi preenchida 'por linhas'. Na ajuda você pode ver que existe um argumento para preenchimento 'por colunas'. Note também que a **matrix** foi impressa na tela com números e vírgulas entre colchetes. Vamos ver isto na próxima seção. Outra forma de construir uma **matrix** é criar uma estrutura e depois preenche-la:

```
a <- matrix(, nrow=2, ncol=5)
a[1,] <- c(0,1,0, 3, 5)
a[2,] <- c(1,0,2,1,1)
```

data.frame: possui 2 dimensões e portanto é parecido com uma **matrix**. De fato, a maioria das funções que servem para um também servem para outro. Mas existem diferenças, é claro. Uma **matrix** pode conter números ou caracteres, mas não os dois. O **data.frame** é mais flexível e uma coluna pode ser de números enquanto outra é de caracteres. Se colocar números e caracteres numa mesma coluna o R interpretará os números como caracteres. Uma tradução em Português para **data.frame** poderia ser conjunto de dados. Geralmente criamos **data.frame** quando importamos dados de planilhas (vamos ver isto no capítulo 9). Mas também podemos criar um **data.frame** a partir de outras funções:

```
a <- matrix(data=1:10, nrow=2)
b <- as.data.frame(a)
  V1 V2 V3 V4 V5
```

```
  1  1  3  5  7  9
  2  2  4  6  8 10
```

Note que a forma de impressão na tela é um pouco diferente de **matrix**. No local dos colchetes aparecem nomes como legendas das colunas e números sem colchetes como legendas das linhas.

A função **as.data.frame()** faz uma coerção. Vimos o que era coerção no capítulo anterior. No exemplo acima, o objeto **a** é uma **matrix** e estamos forçando-o para que se torne um **data.frame**. Em geral conseguimos. Podemos fazer o contrário também:

```
as.matrix(b)
```

Esta última coerção, de **data.frame** para **matrix**, em geral será feita sem problemas se o **data.frame** possuir apenas números ou caracteres. Estas funções abaixo servem tanto para **matrix** quanto para **data.frame**. Não é muito difícil entender o que fazem:

```
nrow(a)
ncol(a)
rowSums(a)
rowMeans(a)
colSums(a)
colMeans(a)
```

7.2 Extraindo e gravando elementos de **vector**, **matrix** e **data.frame**

Vimos no capítulo 2 que podemos descobrir o comprimento (ou número de posições) de um vector com:

```
a <- c(3, 10, 7, 18, 56)
length(a)
[1] 5
```

Também havíamos visto no capítulo 2 que o '**[1]**' simplesmente indica que o primeiro número da linha está na posição 1. De fato, como vamos ver a seguir, colchetes servem para indicar posição num vector e isto tem implicações extremamente importantes. Primeiro, podemos usar colchetes para extrair elementos de um vector:

```
a[2] # retira elemento na segunda posição do vector a.
```

```
[1] 10
a[1:3] # retira elementos nas posições 1:3.
[1] 3 10 7
posicoes <- c(1,2,4)
a[posicoes]
[1] 3 10 18
```

Assim como podemos extrair elementos em posições específicas, também podemos gravar elementos em posições específicas:

```
a
[1] 3 10 7 18 56
a[1] <- 200
a
[1] 200 10 7 18 56
a[3:5] <- c(1000, 2000, 11)
a
[1] 200 10 1000 2000 11
```

O procedimento de extração e gravação de elementos em posições específicas de um vector pode ser facilmente estendido para **`matrix`** e **`data.frame`**. A diferença aqui é que temos duas dimensões, linhas e colunas. Para tanto, temos que especificar as posições de linhas e colunas. Fazemos isto separando as posições das duas dimensões com uma vírgula. Podemos entender isto observando a forma de impressão de uma **`matrix`** na tela:

```
a <- matrix(1:10, 2)
a
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Note que aparece **`[1,]`** na primeira linha e **`[,1]`** na primeira coluna. Eles indicam, respectivamente, a linha 1 e coluna 1. De fato, indica o que aparece a direita e abaixo, respectivamente, do que foi impresso na tela. Portanto:

```
a[1,]
```

```
[1] 1 3 5 7 9
a[,1]
[1] 1 2
```

Nos exemplos acima, extraímos uma linha e depois uma coluna. Em ambos os casos, o resultado é um **vector**. Note que antes da vírgula sempre indicamos as posições das linhas. Depois da vírgula vêm as posições das colunas. Nos exemplos acima, selecionamos uma linha OU uma coluna. Ou seja, indicamos apenas uma dimensão. Nestes casos, toda a linha ou coluna é retirada. Podemos ser mais específicos:

```
a[1, 3] # elemento na linha 1 e coluna 3
[1] 5
a[1, 2:4] # elementos na linha 1 e colunas 2 até 4.
[1] 3 5 7
maria <- c(1,3,5)
a[2, maria] # elementos na linha 2 e colunas definidas pelo objeto maria.
[1] 2 6 10
```

Assim como podemos extrair, também podemos usar os colchetes para gravar elementos dentro de **matrix** ou **data.frame**. O procedimento é semelhante ao que fizemos com **vector**:

```
a[1,2] <- 100 #gravar o valor 100 na posição [1,2] (linha 1 e coluna 2)
a
a[1,2:3]<-c(100, 200) # colocamos 100 e 200 na linha 1 e colunas 2 e 3, respectivamente.
a
a[2,] <- 55 # não especificamos em qual(is) coluna(s) preencher. O R colocará em todas.
a
a[2,] <- 1:10 # erro bastante comum. Não temos como colocar 10 elementos em 5 espaços.
```

7.3 list

De maneira informal, uma **list** é uma coleção de outros objetos. Podemos pensar em **list** como um vetor, mas cujas posições podem ser preenchidas com qualquer tipo de objeto. Por exemplo:

```
a <- 1:5
```

```
b <- matrix(1:10, 2)
a.b <- list(a,b)
a.b
[[1]]
[1] 1 2 3 4 5

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

Note que a lista possui dois elementos. O primeiro é designado como **[[1]]** e é um vector. O segundo é uma **matrix**. Sabemos disto pela forma como o R imprime na tela (e também pois acabamos de criar **a** e **b**!).

7.4 Extraindo e gravando elementos de **list**

Assim como fizemos com **vector**, **matrix** e **data.frame**, podemos tentar usar colchetes para extrair elementos de uma list:

```
a.b[1]
[[1]]
[1] 1 2 3 4 5
```

Conseguimos retirar uma parte da lista, mas o objeto que retiramos ainda é uma lista. Sabemos disto pois o R imprimiu na tela o **[[1]]**. Se quisermos retirar o vector desta lista temos que usar colchetes duplos:

```
a.b[[1]]
[1] 1 2 3 4 5
```

Assim como vimos para outros objetos, podemos gravar elementos em posições específicas de uma lista. O procedimento é o mesmo:

```
a.b[1] <- 8
a.b
[[1]]
```

```
[1] 8

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

7.5 Nomes

Geralmente é muito útil saber os nomes das colunas e linhas em um **`data.frame`** ou dos elementos de um **`list`**. Em menor grau, ás vezes também precisamos disto para **`vector`** e **`matrix`**. O procedimento é:

```
a <- c(1,5,10)
a
names(a) # não possui nomes.
names(a) <- c('maria', 'joao', 'alfredo')
a
names(a) # agora com nomes.
```

Duas coisas importantes a notar. Primeiro, note que usamos aspas (simples ou duplas, não importa) para designar os nomes. O motivo é simples. Caso os nomes não tenham aspas, o R buscará na área de trabalho objetos com aqueles nomes. Não é isto o que queremos. No exemplo acima, queremos apenas usar o próprio nome (e não um objeto do R). Indicamos isto ao R usando aspas.

A segunda coisa é que a função **`names()`** foi usada de duas formas. O primeiro uso é semelhante ao uso de 99% das funções em R e é o que havíamos feito até agora. O segundo uso, com a função **`names()`** à esquerda da setinha, é um pouco distinto e serve para gravar os nomes dentro do objeto.

O uso de nomes para **`matrix`** e **`data.frame`** é semelhante ao uso com vector, a diferença sendo a necessidade de especificar qual dimensão usar:

```
a <- matrix(1:10,2)
a
rownames(a)
colnames(a)
rownames(a) <- c('sem.peixe', 'com.peixe' )
```

```
a
colnames(a) <- c('sp1', 'sp2', 'sp3', 'sp4', 'sp5')
a
```

Visto que **list** tem apenas uma dimensão, o uso de **names()** é semelhante ao uso com **vector**.

```
a <- 1:5
b <- matrix(1:10, 2)
a.b <- list(a,b)
a.b
names(a.b)
names(a.b) <- c('vetor.a', 'matriz.b')
a.b
$vetor.a
[1] 1 2 3 4 5

$matriz.b
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Note que os nomes, após o símbolo $, saíram nos locais onde antes haviam colchetes duplos. Este símbolo de cifrão pode ser usado para retirar elementos de um **list**:

```
a.b$vetor.a
a.b$ve  # mesmo acima. No R, geralmente bastam caracteres suficientes para definir o nome.
        # note que dois caracteres não seriam suficiente caso existisse um segundo nome vetor.b.
```

7.6 Exercícios

7.6.1 **f11.jack1()**: Estimador de riqueza Jackknife 1. Raramente fazemos um censo sobre a composição de espécies numa comunidade. A rigor, teríamos que coletar ou pelo menos observar todos indivíduos, o que raramente é possível. Portanto, fazemos uma amostragem para estimar características da comunidade, por exemplo, a riqueza de espécies. Entretanto, um problema no uso da

riqueza de espécies observada numa amostra é que este valor é uma estimativa enviesada negativamente da comunidade. Primeiro, nunca podemos coletar mais espécies que existem. Segundo, qualquer pessoa com um pouco de experiência de campo sabe que quanto mais se coleta, mais espécies são encontradas. Com exceção de algumas comunidades muito pobres e ou com fácil detecção das espécies, sempre descobrimos espécies ainda não vistas (raras) conforme aumentamos nosso esforço amostral. Uma forma de tentar contornar este problema é usar estimadores de riqueza (Colwell & Coddington 1994; mas veja Melo 2004). Um dos estimadores mais simples é o Jackknife de primeira ordem. Ele é calculado como Jack1 = Sobs + Q_1*((m-1)/m), onde:

Sobs = número de espécies observadas na amostra;

Q_1 = número de espécies que apareceram em apenas 1 unidade amostral;

m = número de unidades amostrais.

Vamos fazer uma função para calcular o Jackknife 1 a partir de uma **`matrix`** ou **`data.frame`**. As linhas serão unidades amostrais e as colunas serão as espécies. Por exemplo, os dados poderiam ser:

```
campo[1,] <- c(0, 0, 5, 2, 1)
campo[2,] <- c(1, 0, 8, 1, 1)
campo[3,] <- c(0,50, 0, 5, 8)
campo[4,] <- c(0, 0, 0, 9, 7)
campo
colnames(campo) <- c('sp1', 'sp2', 'sp3', 'sp4', 'sp5')
```

Complete o restante desta função:

```
f11.jack1 <- function(dados){ # dados não podem ter colunas inteiramente vazias (apenas zeros).
                              # obtenha Sobs com ncol().
                              # obtenha m com nrow().
dados.01 <- ifelse(dados>0,1,0)#precisamos obter Q1. Transformamos em 0 e 1.
Q1.TF <- colSums(dados.01)==1  # espécies aparecem em apenas 1 ua?
Q1.01 <-                       # pode usar uma coerção para transformar TRUE em 1...
Q1 <-                          # ... e depois somar.
                               # já possui todos ingredientes. Calcule a formula.
                               # use mais de uma linha se quiser.
                               # não esqueça de mandar resultado para usuário.
}
```

7.6.2 **f12.chao1()**: Estimador de riqueza Chao 1. O estimador Chao 1 é semelhante ao Jackknife 1, mas com uma diferença básica. No Jackknife, espécies raras são aquelas que apareceram em apenas 1 unidade amostral (Q_1). Quanto maior o Q_1, maior será a riqueza estimada. No Chao 1, espécies raras são aquelas que apareceram com apenas 1 ou 2 indivíduos. Note que no Jackknife não importa com quantos indivíduos uma espécie rara apareceu, desde que esteja presente em apenas 1 unidade amostral. A formula é Chao1 = Sobs + ($F_1^2/2*F_2$), onde:

F_1 = Número de espécies que apareceram com 1 indivíduo;

F_2 = Número de espécies que apareceram com 2 indivíduos.

Complete:

```
f12.chao1 <- function(dados){ # dados não podem ter colunas inteiramente vazias (apenas zeros).
                    # já sabe como obter o Sobs.
                    # F1 é fácil. Em relação ao Q1 (visto acima), não precisa transformar em 0 e 1.
                    # F2: fácil também
                    # termine e mande resultado para usuário.
}
```

7.6.3 **f13.jacks()**: Estimadores de riqueza Jackknife 1 e 2. O Jackknife 1 usa como informação de espécies raras apenas aquelas que apareceram em 1 unidade amostral. O Jackknife 2 usa, além desta informação, o número de espécies que apareceram em apenas 2 unidades amostrais (Q_2). A fórmula é:

$$S_{Jack2} = S_{obs} + \left[Q_1 \left(\frac{2m-3}{m} \right) - Q_2 \left(\frac{(m-2)^2}{m(m-1)} \right) \right]$$

A formula é um pouco mais longa, mas fácil. Faça uma função que calcule o Jackknife 1 e 2:

```
f13.jacks<-function(dados){ # dados não podem ter colunas inteiramente vazias (apenas zeros).
                    # o código é apenas uma sugestão. Pode fazer de outra forma.
                    # obtenha os 'ingredientes'
j1 <-               # calcule o Jack1. Use parte do código da f11.jack1()
                    # calcule o que está no primeiro parênteses da formula acima.
                    #multiplique o que obteve acima por Q1.
                    #calcule o que está no segundo parênteses. Use mais de uma linha se quiser.
```

```
                    #multiplique por Q2.
                    # já consegue calcular o que está dentro dos colchetes.
  j2 <-             # calcule o Jack2.
  resu <- list(j1, j2) # entenda o que estamos fazendo aqui...
  names(resu) <- c('Jackknife 1', 'Jackknife 2') # ... e aqui
  return(resu)
  }
```

7.7 Lista acumulada de conceitos e funções usadas

Conceitos e funções vistos pela primeira vez neste capítulo estão em negrito.

área de trabalho, .RData e .Rhistory
funções: uso geral
uso de: parênteses, chaves e colchetes (simples e duplos)
argumentos, separação de argumentos com vírgula
uso de ordem dos argumentos em vez dos nomes dos argumentos
argumentos padrão (ou default)
atribuição
script
comentários
objetos: `vector`, `function`, **`matrix`**, **`data.frame`**, **`list`**
pacotes
funções: estrutura
funções: pseudo-código
funções: trabalhando com erros
coerção
uso de nomes

operações básicas: +; -; /; *; ^
operadores relacionais e lógicos: >; <; ==; >=; <=; !=; &; |
`any(); all()`
`log()`
`exp()`
`?`
`??`
`c()`
`sum()`
`length()`
`mean()`
`sqrt()`
`rep(); seq(); :`
`ls(); rm()`
`function(); return()`
	`f1.simp()`
	`f2.correl()`
	`f3.dp()`
	`f4.shannon()`
	`f5.hill()`
	`f6.cole()`
	`f7.F2()`
	`f8.dissi.par.quali()`
	`f9.ind.div()`
	`f10.nodf.par()`
	`f11.jack1()`
	`f12.chao1()`
	`f13.jacks()`

cor() as.numeric() ifelse(); if(); if() else abs() print() **matrix()**; **list()** **as.data.frame()**; **as.matrix()** **nrow()**; **ncol()** **rowSums()**; **rowMeans()**; **colSums()**; **colMeans()** **names()**; **rownames()**; **colnames()**	

8 Amostragem e laço **for()**

8.1 Amostragem

Em alguns lugares do Brasil se usa a expressão 'fazer miséria' para expressar algo espetacular ou surpreendente. Por exemplo, 'Lucrécio fez miséria no jogo de futebol' significa que ele jogou muito bem, mais do que o esperado. Neste capítulo vamos ver duas funções que 'fazem miséria'. A primeira é bastante simples e é usada para amostrar elementos de um vetor de dados. O procedimento é:

```
avelina <- 1:5
sample(avelina)
[1] 5 4 3 1 2
```

É provável que sua sequência seja diferente desta mostrada acima. Isto pois cada vez que se usa o **sample()** um novo sorteio é feito. Faça algumas vezes para comprovar. Este é o uso mais simples da função. No arquivo de ajuda da função podemos ver que existem dois argumentos interessantes. O primeiro é o tamanho da amostra:

```
sample(avelina, size=3)
[1] 2 4 3 # obtenção de subamostra de tamanho 3
```

Assim como no exemplo anterior, é provável que seu resultado seja diferente deste mostrado acima. Repita o comando e uma nova subamostra será obtida. O segundo argumento de interesse aqui é o de reposição:

```
sample(avelina) #note que cada elemento é sorteado apenas 1 vez.
sample(avelina, replace=TRUE)
[1] 4 4 1 5 2
```

Note que o número 4 apareceu 2 vezes enquanto o número 3 não apareceu. Repita o comando e uma nova sequência será obtida. Também podemos usar os dois argumentos juntos:

```
sample(avelina, size=4, replace=TRUE)
sample(avelina, size=6) #erro: não é possível obter 6 elementos de um vetor com 5...
sample(avelina, size=6, replace=TRUE) #... mas com reposição podemos.
```

O uso básico do **sample()** é simples assim. Existe um outro argumento que controla a probabilidade de cada elemento ser sorteado. Não vamos usá-lo aqui, mas pode ser interessante ver como funciona no arquivo de ajuda. E o 'fazer miséria' que falei acima? O uso da função é simples. O interessante é a ampla gama de situações em que podemos usá-la. Vamos ver alguns casos nas próximas seções. Um exemplo simples é a aleatorização de linhas de uma **matrix** ou **data.frame**:

```
ana <- matrix(1:12, 4, 3)
ana
ordem <- sample(1:4)
ana[ordem, ]
ana[sample(1:4), ] #equivalente ao comando acima.
```

8.2 Laço **for()**

Uma das coisas mais úteis em computação é automatizar tarefas repetitivas. Elas podem até ser simples, mas tornam-se tediosas quando devem ser repetidas 1ooo vezes. Ainda, a chance de erro é alta quando fazemos algo muitas vezes. O uso básico do **for()** é:

```
for(i in 1:5){
   print(i)
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

O resultado aparece de maneira automática na tela, mas na verdade houve 5 ciclos e em cada ciclo um número foi impresso na tela. Note que dentro dos parênteses existe uma sequência de inteiros entre 1 e 5. O comando **for()** faz ciclos ou repetições ou laços ou 'rodadas' (loops). No primeiro ciclo, o **i** vale 1. Portanto, quando o R chega na segunda linha o **i** (que vale 1) é impresso na tela. Chega-se então na chave que fecha o código e o R volta ao início para um segundo ciclo. Neste segundo ciclo, entretanto, o valor de **i** já não é 1, mas 2. Portanto, o número 2 será impresso na tela. O procedimento continua até se esgotar o vetor 1:5. Podemos usar qualquer sequência dentro do **for()**. Por exemplo:

```
maria <- c(6, 7, 1, 4)
```

```
 for(i in maria){
    print(i)
 }
[1] 6
[1] 7
[1] 1
[1] 4
```

Nos dois exemplos acima usamos o valor de **i** diretamente nas ações contidas dentro das chaves. Isto não é necessário:

```
 for(i in 1:10){
    print('não vou conversar sobre novelas durante aulas')
 }
```

Nos três exemplos acima usamos apenas a letra **i** para indicar a posição do vetor. Na verdade, podemos usar qualquer letra ou nome. Ainda, usamos apenas uma função dentro das chaves, o **print()**. Entretanto, podemos usar qualquer função ou grupo de funções:

```
 abunds <- c(5, 8, 9, 1, 0, 7, 6, 12)#abundâncias de uma sp. em 8 parcelas.
 resu <- numeric(10) #cria um vetor de tamanho 10 preenchido com zeros.
 for(j in 1:10){
    amostra <- sample(abunds, replace=TRUE)
    resu[j] <- mean(amostra)
 }
 resu
```

Visto que não especificamos o argumento de tamanho para a função **sample()**, as amostras obtidas serão do mesmo tamanho de **abunds**. Note que as amostras não serão idênticas pois estamos fazendo o sorteio com reposição. O código acima pode parecer simples (e é!), mas acabamos de estimar uma média da população por meio de um bootstrap, procedimento bastante importante na Estatística. Note também como guardamos cada média em cada ciclo dentro do vetor **resu**. Finalmente, note que usamos o **j** em vez do **i** (também poderíamos ter usado, por exemplo, joana). Tenha certeza que entendeu cada linha do código acima. Vamos ver vários exemplos no restante deste

capítulo, todos eles baseados nos conceitos expressos pelo código acima.

8.3 Exercício resolvido. **f14.correl.teste()**: Teste de correlação por aleatorização.

Programamos no capítulo 4 uma função para calcular uma correlação de Pearson. Demos o nome de **f2.correl()**. Podemos agora testar por aleatorização se um dado valor de correlação é diferente do esperado ao acaso. O procedimento abaixo é muito semelhante a qualquer teste por aleatorização. Caso consiga entender este, saiba que já entendeu 90% de todos testes de aleatorização!

Todos testes de aleatorização são compostos basicamente por 6 etapas:

1) Escolha a estatística do teste;
2) Calcule a estatística para os dados observados;
3) Aleatorize os dados;
4) Calcule a estatística para os dados aleatorizados;
5) Repita as etapas 3 e 4 muitas vezes (por exemplo, 9999 vezes);
6) Calcule a probabilidade como a proporção de valores obtidos nas aleatorizações que são iguais ou mais extremos que a estatística observada.

A diferença básica entre testes de aleatorização é o passo 1. Cada teste será baseado em uma estatística diferente. Quando falamos em estatística, pense em um número que resuma um conjunto de informações. A média é uma estatística. O coeficiente de correlação de Pearson é uma estatística. A diferença entre duas médias pode ser usada como uma estatística num teste para avaliar se as médias de dois grupos diferem.

Nossa função para testar um coeficiente de correlação pode ser:

```
f14.correl.teste <- function(x, y, aleats=999){
r.obs <- f2.correl(x, y) #Passos 1 e 2. Podemos aproveitar uma função que já fizemos.
r.aleats <- numeric(aleats)      #para guardar os r aleatorizados
for(maria in 1:aleats){
   x.aleat <- sample(x)          #Passo 3. Basta aleatorizar um dos vetores.
   r.aleats[maria] <- f2.correl(x.aleat, y) #Passo 4.
}                                #fecha o for(). Passos 5.

   #Agora só precisamos fazer o Passo 6.
```

```
r.obs.abs <- abs(r.obs) #pois estamos interessados em saber se o observado é um valor
                        #extremo, independente da cauda (positiva ou negativa)
r.aleats.abs <- abs(r.aleats) #idem
maiores <- r.aleats.abs >= r.obs.abs #vai criar um vetor com TRUE e FALSE
extremos <- sum(as.numeric(maiores)) #transforma TRUE em 1 e FALSE em 0 e soma.
p <- (extremos+1) / (aleats+1) #temos que somar 1 no numerador pois existe a
                               #possibilidade de não existir nenhum valor extremo.
                               #Se não somarmos, o p seria 0 e não existe
                               #probabilidade 0 (seria uma certeza).
return(list(r.observado=r.obs, Probabilidade=p, r.aleats=r.aleats))
}
```

Vamos ser se funcionou. Havíamos visto no capítulo 4:

```
riq.anfibios <- c( 50,  83,  79,  41, 33, 29,  45,  36)
pop.humana <- c(100, 300, 320, 250, 77, 82, 190, 120) # vezes mil habitantes
f2.correl(x=riq.anfibios, y=pop.humana)
[1] 0.8375786
```

Agora:

```
f14.correl.teste(x=riq.anfibios, y=pop.humana)
$r.observado
[1] 0.8375786

$Probabilidade
[1] 0.007
```

e um monte de valores aleatorizados... Lembrando que criamos uma lista para os resultados, melhor usar:

```
resu <- f14.correl.teste(x=riq.anfibios, y=pop.humana)
resu$r.observado
resu$Prob  # basta o número de caracteres suficientes para o R saber qual é.
resu$r.aleats
```

Note que não especificamos o número de aleatorizações e, portanto, o valor padrão que colocamos no argumento **aleats** da função foi usado (999 vezes). Note também que seu valor de probabilidade pode ser um pouco diferente. Fizemos um teste usando sorteios. Cada vez que fazemos sorteios obtemos valores distintos. Nossa função também retorna os valores de r aleatorizados. Podemos ver o quanto o nosso valor observado é extremo desta forma:

```
hist(resu$r.aleats) #para fazer um histograma. Deixe a janela do gráfico aberta.
abline(v=c(resu$r.obs, -resu$r.obs)) #coloca linhas verticais nas posições v do eixo X.
```

O valor de probabilidade nada mais é do que a proporção de valores de r aleatorizados que são iguais ou mais extremos do que o valor observado. Ou seja, a quantidade de valores que estão antes da primeira linha vertical e depois da segunda linha vertical no histograma que fizemos.

Finalmente, podemos comparar nossa probabilidade com aquela produzida por uma função já disponível no R:

```
cor.test(riq.anfibios, pop.humana)
```

Veja que a probabilidade foi 0,009. Parecida com a nossa. Bom indicativo (mas não certeza...) de que não cometemos erros.

8.4 Exercícios

8.4.1 **f15.medias2.teste()**: Teste de comparação entre duas médias por aleatorização.

Programe uma função para testar por aleatorização se duas médias são diferentes. Ou seja, se a diferença entre elas pode ser devido ao acaso simplesmente.

```
f15.medias2.teste <- function(g1, g2, aleats=999){
        #g1 e g2 são vetores com observações. De preferência, do mesmo tamanho.
        #obtenha a estatística observada (diferença das médias).
        #crie um vetor para guardar as estatísticas aleatorizadas. Qual tamanho?
        #inicie um for
            #pode-se aleatorizar um dos vetores apenas? Aleatorizar os dois?
            #ou juntá-los, aleatorizar, e depois partir em dois vetores?
            #obtenha a estatística aleatorizada.
```

```
        #feche o for.
        #obtenha os absolutos do observado e dos aleats.
        #calcule o número de aleats extremos.
        #calcule o p
        #retorne para o usuário os valores da estatística observada e a probabilidade.
}
```

8.4.2 **`f16.par.teste()`**: Teste de comparação entre duas médias com dependência por aleatorização. No exercício anterior vimos um teste de aleatorização equivalente a um teste t de duas amostras. Agora vamos fazer um equivalente ao teste t pareado. A diferença crucial é a estatística a ser usada e a forma de aleatorizar. Sobre a estatística, devemos lembrar que o teste é feito 'por pares', ou seja, a comparação é feita dentro de cada par. Portanto, podemos usar como estatística a média das diferenças de cada par. A aleatorização deve respeitar o esquema de par. Ou seja, deve-se fazer a aleatorização dentro do par. Pense um pouco na lógica deste teste para entender a razão de fazermos isto.

```
f16.par.teste <- function(dados, aleats=999){
    # dados é um data.frame ou matrix com duas colunas. Cada linha indica um par.
    # calcule a estatística para...
    # crie vetor...
    # abra um for...
        # preste atenção em como deve fazer a aleatorização. Crie uma estratégia.
        # calcule estatística, guarde...
    # calcule p, mande para usuário valor da estatística observada e probabilidade.
}
```

Para ver se fez certo, confira com este (valor de p pode ser levemente diferente). Schneck et al. (2011) avaliaram, entre outras coisas, o efeito da rugosidade sobre a riqueza de algas do perifíton em riachos. Um problema detectado no planejamento do estudo é que a diferença em riqueza deveria ser pequena e portanto difícil de ser detectada. Para complicar, riachos geralmente são bastante heterogêneos quanto a velocidade da água e isto tem efeitos fortes sobre as algas. Em outras palavras, isto geraria variação indesejada que poderia impedir a detecção de um efeito (diferença de riqueza), mesmo que este exista. Para contornar este problema, eles decidiram usar um experimento em pares

(ou blocos), onde cada par era constituído por uma lâmina de microscopia. Metade da lâmina era lisa e a outra metade rugosa. Visto que a lâmina é pequena, as condições ambientais em ambas metades são praticamente iguais. Use estes dados fictícios para testar sua função:

```
rugosa <- c(10, 13,  8, 12,  6, 11, 14)
lisa   <- c( 7, 11,  7, 13,  5,  8, 11)
algas <- matrix[, nrow=7, ncol=2)
algas[,1] <- rugosa
algas[,2] <- lisa)
    #Importante: neste desenho experimental, a ordem dos valores nos vetores não é aleatória.
    #O primeiro valor de um vetor faz par com o primeiro valor no outro vetor.
f16.par.teste(dados=algas, aleats=999)
```

8.4.3 **`f17.coletor()`**: Curva do coletor. América foi ao campo e registrou anfíbios anuros em várias poças. Ela estava interessada em saber se seu esforço amostral estava bom. Ou seja, se seria interessante continuar coletando. Os dados eram estes:

```
america <- matrix(, nrow=5, ncol=11)
america[1,] <- c(1, 3, 2, 1, 0, 0, 0, 1, 0, 0, 0)
america[2,] <- c(1, 0, 2, 0, 1, 0, 0, 1, 0, 0, 0)
america[3,] <- c(1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0)
america[4,] <- c(5, 1, 0, 1, 0, 0, 1, 5, 0, 1, 0)
america[5,] <- c(0, 0, 0, 7, 0, 0, 0, 1, 0, 0, 8)
colnames(america) <-c ('sp1','sp2','sp3','sp4','sp5','sp6','sp7',
                       'sp8','sp9','sp10','sp11')
```

Ela então fez um gráfico onde registrou no eixo X o número de poças visitadas e no eixo Y o número acumulado de espécies encontradas (uma Curva do Coletor). A ordem das poças seguiu a ordem de coleta no campo. Um colega de pós-graduação viu seu gráfico e disse que a ordem das poças utilizada era arbitrária e que poderia ser feito com qualquer outra ordem. Vamos criar uma função para calcular curvas de coletor feitas com ordem arbitrária das poças.

```
f17.coletor <- function(comunidade){
    #comunidade é uma matrix ou data.frame com poças nas linhas e espécies nas colunas.
    #transforme em 0 ou 1 (pois usuário pode ter inserido valores de abundância).
    #obtenha o número de poças. Chame de n.
```

```
    # obtenha o número de espécies. Chame spp.
ordem <- sample(1:n) #crie uma ordem aleatória para as poças.
temp <- matrix(0, nrow=n, ncol=spp) # uma matriz com zeros. Veja como será preenchida.
resu <- numeric() #um vetor numérico, mas sem conteúdo.
  #inicie um for com o indexador i. Ele irá de 1 até n.
    temp.po <-  # use ordem para obter 1 poça (a 1ª no ciclo 1). Lembre dos colchetes.
     temp[i,] <- temp.po #junte as poças
                #obtenha a riqueza de espécies. Talvez precise de várias linhas.
                #guarde o valor de riqueza em resu.
  # fecha for
    # retorne sequência de valores de riqueza acumulada para o usuário.
}
```

Para testar, faça um gráfico:

```
riquezas <-
plot(1:nrow(comunidades),  riquezas)
```

Calcule outra curva e faça outro gráfico. Cada curva deve ter vários degraus irregulares. O ponto final deve ser o mesmo (riqueza total nas poças).

8.4.4 **f18.caspp()**: Curva de acumulação de espécies. A curva do coletor pode ter diferentes formas dependendo da ordem das amostras utilizada para construí-la. Uma curva de acumulação de espécies é a média de várias curvas do coletor para uma dada amostra.

```
f18.caspp(comunidade, aleats){
    #use a função f17.coletor() para fazer cada curva do coletor.
    # pense um pouco antes de fazer. Tente escrever um pseudo-código. Não é tão difícil :-)

return(riquezas) #vetor de riqueza acumuladas médias
}
```

Para testar, faça um gráfico:

```
riquezas <-
plot(1:nrow(comunidades),  riquezas)
```

Calcule outra curva e faça outro gráfico. As curvas devem ser suaves e muito parecidas entre si. O ponto final deve ser o mesmo (riqueza total nas poças).

8.5 Lista acumulada de conceitos e funções usadas

Conceitos e funções vistos pela primeira vez neste capítulo estão em negrito.

área de trabalho, .RData e .Rhistory
funções: uso geral
uso de: parênteses, chaves e colchetes (simples e duplos)
argumentos, separação de argumentos com vírgula
uso de ordem dos argumentos em vez dos nomes dos argumentos
argumentos padrão (ou default)
atribuição
script
comentários
objetos: `vector`, `function`, `matrix`, `data.frame`, `list`
pacotes
funções: estrutura
funções: pseudo-código
funções: trabalhando com erros
coerção
uso de nomes

operações básicas: +; -; /; *; ^ operadores relacionais e lógicos: >; <; ==; >=; <=; !=; &; | any(); all() log() exp() ? ?? c() sum() length() mean() sqrt() rep(); seq(); :; **numeric()** ls(); rm() function(); return() cor() as.numeric(); as.data.frame(); as.matrix() ifelse(); if(); if() else **for()** abs() print() matrix(); list() nrow(); ncol() rowSums(); rowMeans(); colSums(); colMeans() names(); rownames(); colnames() **sample()** **plot()**; **hist()** **abline()**	f1.simp() f2.correl() f3.dp() f4.shannon() f5.hill() f6.cole() f7.F2() f8.dissi.par.quali() f9.ind.div() f10.nodf.par() f11.jack1() f12.chao1() f13.jacks() **f14.correl.teste()** **f15.medias2.teste()** **f16.par.teste()** **f17.coletor()** **f18.caspp()**

9 Outras ferramentas úteis para Ecólogos

9.1 Importação e exportação de objetos

Em capítulos anteriores digitamos nossos conjuntos de dados diretamente no R. Eram pequenos e, portanto, o procedimento era fácil e rápido. As coisas complicam um pouco quando usamos dados reais. Nestes casos, o ideal é digitar os dados num programa de planilha eletrônica (MS Excel, LibreOffice Calc etc) e depois importar para o R. Uma sugestão é digitar em sua planilha eletrônica apenas os dados originais ou brutos. Qualquer transformação, combinação de dados, remoção de amostras etc pode ser feita dentro do R. A grande vantagem disto é que, caso use um script, você terá um registro fácil de tudo que foi feito. Qualquer alteração poderá ser repetida muito rapidamente.

A função mais básica de importação de planilhas (o tipo de dados mais comum) é **`read.table()`**. Para ilustrar o seu uso, digite na sua planilha eletrônica preferida isto:

	clima	sp1	sp2	sp3
a1	trop	1	1	3
a2	trop	1	20	0
b1	temp	15	0	0
b2	temp	0	22	0

A primeira célula (linha 1 e coluna 1) deverá ficar em branco. Não deve haver espaço entre nomes ou números. Não deverá haver células em branco exceto pela primeira da planilha. Evite acentos. Grave o arquivo como 'Texto separado por tabulações'. Leia e procure entender as mensagens que sua planilha eletrônica provavelmente te mostrará. Note que perderá formatações. Isto não é um problema. Grave no mesmo diretório que contém seu .RData. Use o nome dados.txt. Depois:

```
acaros <- read.table('dados.txt', header=TRUE)
```

Alguns erros comuns. Nome do arquivo possui diferença em relação ao que foi escrito como argumento da função. Arquivo não está no diretório de trabalho. Existe um ou mais espaços na planilha original.

O primeiro argumento é o endereço e nome do arquivo. Caso tenha colocado o arquivo no diretório de trabalho, basta colocar o nome. Caso contrário, terá que escrever o endereço completo. Geralmente algo do tipo:

```
acaros <- read.table('C:/doc/dados.txt', header=TRUE) # MS Windows
acaros <- read.table('/home/maria/doc/dados.txt', header=TRUE) # Linux
acaros
```

Também podemos obter um arquivo hospedado em um servidor de internet:

```
japi <- read.table('http://www.ecoevol.ufg.br/adrimelo/div/japi.txt',
header=TRUE)
japi
```

O segundo argumento utilizado informa se existe ou não cabeçalho na planilha a ser importada. Note que usamos cabeçalho no exemplo e esta é a razão da primeira célula da planilha ficar em branco. Caso use **header=FALSE**, a importação será feita a partir da primeira linha e primeira coluna e, portanto, não haverá nomes para linhas e colunas. Existem outros argumentos da função **read.table()**. Entre eles, pode-se especificar um caractere específico para marcar a separação dos dados. A opção padrão é "espaço em branco' e isto inclui tabulação.

Existem outras opções de importação, algumas muito semelhantes ao 'copiar' e 'colar'. Você vai descobri-las com o tempo. Entretanto, quando estiver fazendo um trabalho científico, eu recomendo fortemente que você documente tudo o que fizer. Usar o **read.table()** na forma mostrada acima é uma excelente prática de documentação.

Um problema comum é ter várias versões das planilhas conforme vamos coletando mais informações ou corrigindo erros. Isto por si só já exige uma boa organização para não usar planilhas antigas, por exemplo, usando sufixos no nome do arquivo (ex. dados-v2.txt). Um problema adicional é realizar várias operações sobre estas planilhas (transformações, padronizações, separações, agregações). Nestes casos, caso algo mude nos dados originais, todas estas outras planilhas deverão ser atualizadas. Isto te exige ainda mais organização e trabalho. Sugiro fortemente que faça isto dentro do R. Deixe no diretório de sua área de trabalho apenas 1 (ou se necessário 2 ou 3) arquivo(s) de dados. Faça o restante dentro do seu script. Sua vida ficará mais simples.

A função básica de exportação de planilhas (**data.frame** ou **matrix**) é

```
write.table(acaros, 'acaros.txt')
```

O primeiro argumento é o objeto que quer exportar. O segundo é o nome que quer usar para gravar. O arquivo será gravado no diretório de trabalho. Você pode gravar em outro local usando o

endereço completo da pasta (conforme usamos para **`read.table()`**).

9.2 Gráficos

O R pode fazer praticamente qualquer gráfico. Além das funções já disponíveis na sua instalação básica, você pode instalar vários pacotes específicos. O comando básico é:

```
x <- 1:9
y <- c(1:5, 4 :1)
plot(x, y)
plot(x, y, col=2, type='l', xlim=c(-10, 20), xlab='Eixo X')
```

A função **`plot()`** é dita ser 'genérica'. Isto significa que ela produzirá diferentes gráficos dependendo do tipo de dados utilizados:

```
x <- c(rep(1,5), rep(2,5))
x.fac <- factor(x)
x.fac  # note que não temos mais números, mas níveis de um fator. Portanto:
x.fac*2  # Isto não vai funcionar.
y <- 1:10
plot(x, y)
plot(x.fac, y)  # Visto que X é um fator, um diagrama de caixas é produzido.
```

Pode-se incluir letras, pontos etc em um gráfico. Para tanto, deixe a janela do gráfico aberta e:

```
x <- 1:9
y <- c(1:5, 4:1)
plot(x, y, xlim=c(0,10), ylim=c(1,10))
text(5, 9, 'Seu texto')
points(5, 6, col='red', pch=2, cex=4)
```

Note que os dois primeiros argumentos indicam as coordenadas x e y. Pode-se usar vetores de mesmo tamanho para colocar vários textos ou pontos simultaneamente. Também podemos inserir linhas. Ainda com a janela do gráfico aberta:

```
lines(c(2, 2), c(2, 3), col='blue')
```

```
lines(c(3, 3, 5), c(2,3, 4), col='green')
```

Uma função mais simples (mas também mais limitada) para incluir linhas (ainda com a janela de gráfico aberta):

```
abline(v=5, col='orange')
abline(h=4, col='yellow')
abline(a=3, b=1, col='violet', lwd=2)
```

Para saber com maior exatidão as coordenadas de 1 ponto no gráfico (com a janela aberta):

```
locator(1)
```

Note que o símbolo de maior de sua área de seu console desapareceu. Se mover o cursor do seu mouse para a janela de gráfico notará que ele mudará de forma. Clique em algum ponto dentro do gráfico e você terá as coordenadas na área de trabalho. Você também pode (com a janela do gráfico aberta) digitar isto e depois clicar no gráfico:

```
text(locator(1), 'Cliquei aqui')
```

Esta é uma pequena demonstração de gráficos que o R pode fazer. Conforme dito acima, o R pode fazer praticamente qualquer gráfico. Entretanto, geralmente temos que especificar muitos detalhes, tais como espessura de linhas, tamanho de números nas legendas, cores, margens da janela etc. Não é difícil fazer isto, mas pode ser um pouco demorado no início, pois você terá que ver as páginas de ajuda para descobrir os muitos argumentos disponíveis. O importante, entretanto, é que você faça bons gráficos, por exemplo, que sejam compreensíveis e com boa proporcionalidade entre seus elementos. Isto nem sempre é produzido com poucos argumentos e, portanto, geralmente você terá que estudar um pouco a ajuda das funções gráficas. Você só não pode fazer um gráfico ruim e dar aquela desculpa esfarrapada 'o R fez assim'. Estude! Ou vá procurar outro programa de gráficos de sua preferência.

9.3 Mapas

Instale o pacote '**maps**' usando o menu do RStudio e em seguida carregue-o (clique na caixinha). Podemos fazer mapas simples com:

```
map('world')
```

```
map('world', 'brazil')
sa <- c('brazil', 'argentina', 'uruguay', 'paraguay', 'chile',
       'bolivia', 'peru', 'ecuador', 'colombia', venezuela',
      'suriname', 'guyana', 'french guiana')
map('world', sa)
map('world', sa, col='red')
map('world', sa, col='red', fill=TRUE)
map('world', sa, boundary=FALSE)
map('world', sa, interior=FALSE)
```

O primeiro argumento da função **map()** é a base de dados. Veja o arquivo de ajuda do pacote '**maps**' para ver outras. O segundo é a região para a qual quer fazer o mapa.

9.4 Arquivos shape

Baixe os arquivos shape (são 3 arquivos) dos biomas do Brasil no IBGE:

```
e1<-'ftp://geoftp.ibge.gov.br/mapas_tematicos/mapas_murais/shapes/biomas/Biomas5000.shp'
e2<-'ftp://geoftp.ibge.gov.br/mapas_tematicos/mapas_murais/shapes/biomas/Biomas5000.shx'
e3<-'ftp://geoftp.ibge.gov.br/mapas_tematicos/mapas_murais/shapes/biomas/Biomas5000.dbf'
download.file(e1, 'biomas.shp')
download.file(e2, 'biomas.shx')
download.file(e3, 'biomas.dbf')
```

Os arquivos serão baixados no seu diretório de trabalho. Agora instale um pacote necessário (**'maptools'**), carregue-o, importe os dados para o R e faça um mapa rápido:

```
biomas <- readShapePoly('biomas.shp')
plot(biomas) # o plot() também faz isto!
```

9.5 Árvores filogenéticas

Um dos formatos mais usados para codificar árvores filogenéticas em arquivos de texto é o Newick, cuja descrição pode ser vista aqui: http://evolution.genetics.washington.edu/phylip/newicktree.html. Vamos ver alguns exemplos obtidos no sitio eletrônico acima. Instale e carregue o pacote '**ape**'

```
arv <- '( ( (A,B),(C,D) ), E );'
 filo <- read.tree(text=arv)
 plot(filo)
```

Veja que dentro dos dois parênteses mais internos existem 2 pares de objetos (A com B em amarelo; C com D em verde). Um parêntese os une (cinza). Estes 4 objetos são então ligados a um quinto objeto (E em azul) pelo parêntese mais externo (roxo). Note que os elementos são separados por vírgulas e estes podem ser objetos isolados (E) ou grupos de objetos dentro de parênteses (A com B). O final da árvore é indicado com um ponto e vírgula. Podemos especificar comprimentos para os ramos da árvore:

```
 arv.c <- '( ( (A:1,B:1):2,(C:2,D:2):1 ):1, E:4 );'
 filo.c <- read.tree(text=arv.c)
 plot(filo.c)
```

Outro exemplo (do sítio eletrônico citado acima):

```
 arv.p <- '(Bovine:0.69395,(Gibbon:0.36079,(Orang:0.33636,(Gorilla:0.17147,
(Chimp:0.19268, Human:0.11927):0.08386):0.06124):0.15057):0.54939,
Mouse:1.21460):0.10;'
 filo.p <- read.tree(text=arv.p)
 plot(filo.p)
```

Podemos examinar o objeto que criamos e editá-lo:

```
 filo.p # para informações básicas da árvore
 names(filo.p)
 filo.p$tip.label
 filo.p$tip.label[6] <- 'Eu!'
 outros <- c('Bovine', 'Mouse')
 filo.primos <- drop.tip(filo.p, outros)
 plot(filo.primos)
```

Muitos pacotes do R incluem conjuntos de dados. Para obtê-los, basta usar a função **`data()`**. Por exemplo, para algo menos didático e mais real (disponível no pacote '**ape**'):

```
 data(bird.families)
```

```
plot(bird.families)
```

É bastante provável que você não tenha que digitar sua árvore, pois geralmente vai obtê-la em outro pacote do R ou programa externo. Um dos pacotes fundamentais para manipulação de árvores filogenéticas é o '**ape**'. Este pacote é acompanhado pelo excelente livro do Paradis (2012). Análises de ecologia de comunidades com informação filogenética estão disponíveis no '**picante**'. Instale e carregue-o. Por exemplo, podemos calcular a diversidade filogenética bem como a distância (soma dos comprimentos de ramos) entre espécies presentes numa comunidade:

```
data(phylocom)
names(phylocom)
phylocom$sample
phylocom$phylo
plot(phylocom$phylo)
pd(phylocom$sample, phylocom$phylo) # Phylogenetic Diversity
mpd(phylocom$sample, cophenetic(phylocom$phylo))
    # Média das distâncias entre taxa em cada uma das comunidades (linhas da matriz).
```

Calculamos no capítulo 6 o índice de dissimilaridade de Sorensen baseado em presença e ausência de espécies. Podemos interpretá-lo como uma situação em que cada espécie é igualmente original e, portanto, a árvore de relacionamento é uma única politomia. Mas o mundo não precisa ser preto-no-branco. Podemos ponderar a importância de cada espécie para a dissimilaridade total de acordo com sua originalidade filogenética e calcular o phyloSor (Graham & Fine 2008) bem como seus componentes de substituição (turnover) e diferença de diversidade filogenética (Leprieur et al. 2012). Vamos usar alguns exemplos de dados disponíveis no pacote '**CommEcol**'. Ele não está no repositório oficial do R (CRAN), mas num repositório de desenvolvimento e, portanto, temos que especificar o endereço:

```
install.packages("CommEcol", repos=c("http://r-forge.r-project.org",
"http://cran-r.c3sl.ufpr.br"), dep=TRUE)
data(sites5.6)
sites5.6
data(tree6)
plot(tree6)
phylosor(sites5.6, tree6)# no 'picante'. É uma similaridade.
```

```
1 - phylosor(sites5.6, tree6)# para dissimilaridade.
part.p.tree(sites5.6, tree6, index.family = "sorensen") # no 'CommEcol'.
```

Para entender um pouco melhor o phylosor e análises relacionadas (como o treeNodf; ver abaixo):

```
poli <- read.tree(text="(taxon_1:1,taxon_2:1,taxon_3:1,
                    taxon_4:1,taxon_5:1, taxon_6:1):1;")
plot(poli)
1- phylosor(sites5.6[1:2, ], poli)# veja que é igual a:
f8.dissi.par.quali(obj1=sites5.6[1,], obj2=sites5.6[2,], qual='sor')
```

Vimos no capítulo 6 como calcular o NODF, uma medida de aninhamento para comunidades baseada em presença e ausência de espécies. Podemos usar o relacionamento filogenético para calcular o treeNodf (Melo et al. 2014):

```
treeNodf(sites5.6, col.tree=tree6, order.rows=FALSE)
```

9.6 Lista acumulada de conceitos e funções usadas

Conceitos e funções vistos pela primeira vez neste capítulo estão em negrito.

área de trabalho, .RData e .Rhistory
funções: uso geral
uso de: parênteses, chaves e colchetes (simples e duplos)
argumentos, separação de argumentos com vírgula
uso de ordem dos argumentos em vez dos nomes dos argumentos
argumentos padrão (ou default)
atribuição
script
comentários
objetos: `vector`, `function`, `matrix`, `data.frame`, `list`
pacotes
funções: estrutura
funções: pseudo-código
funções: trabalhando com erros
coerção
uso de nomes

operações básicas: +; -; /; *; ^ operadores relacionais e lógicos: >; <; ==; >=; <=; !=; &; | any(); all() log() exp() ? ?? c() sum() length() mean() sqrt() rep(); seq(); :; numeric() ls(); rm() function(); return() cor() as.numeric(); as.data.frame(); as.matrix() ifelse(); if(); if() else for() abs() print() matrix(); list() nrow(); ncol() rowSums(); rowMeans(); colSums(); colMeans() names(); rownames(); colnames() sample() **data()** **read.table()**; **write.table()** **install.packages()** **readShapePoly()** **download.file()** **read.tree()** **drop.tip()** **phylosor()**; **treeNodf()** **pd()**; **mpd()** plot(); hist() abline(); **points()**; **text()**; **lines()** **locator()** **map()**	f1.simp() f2.correl() f3.dp() f4.shannon() f5.hill() f6.cole() f7.F2() f8.dissi.par.quali() f9.ind.div() f10.nodf.par() f11.jack1() f12.chao1() f13.jacks() f14.correl.teste() f15.medias2.teste() f16.par.teste() f17.coletor() f18.caspp()

10 Literatura Citada

Almeida-Neto, M., P. Guimarães, P.R. Guimarães, R.D. Loyola & W. Ulrich. 2008. A consistent metric for nestedness analysis in ecological systems: reconciling concept and measurement. Oikos 117: 1227–1239.

Alves, R. 2011. Filosofia da ciência: introdução ao jogo e a suas regras. Loyola.

Araújo, M.B. 2003. The coincidence of people and biodiversity in Europe. Global Ecology & Biogeography 12: 5-12.

Chabris, C. & D. Simons. 2009. The invisible gorilla. Broadway Paperbacks. New York.

Chambers, J.M. 1998. Programing with data: a guide to the S language. Springer.

Coleman, B.D., M.A. Mares, M.R. Willig & Y-H. Hsieh. 1982. Randomness, area, and species richness. Ecology 63: 1121-1133.

Colwell, R.K. & J.A. Coddington. 1994. Estimating terrestrial biodiversity through extrapolation. Philosophical Transactions of the Royal Society B 345: 101-118.

Eagleman, D. 2011. Incógnito: As vidas secretas do cérebro. Rocco.

Gleiser, M. 2007. Cartas a um jovem cientista: o universo, a vida e outras paixões. Campus.

Gotelli, N.J. 2007. Ecologia. 3a edição. Planta.

Graham, C.H. & P.V.A Fine. 2008. Phylogenetic beta diversity: linking ecological and evolutionary processes across space in time. Ecology Letters 11: 1265-1277.

Leprieur, F., C. Albouy, J.D. Bortoli, P.F. Cowman, D.R. Bellwood & D. Mouillot. 2012. Quantifying Phylogenetic Beta Diversity: Distinguishing between 'true' turnover of lineages and phylogenetic diversity gradients. PlosOne 7(8) e42760.

MacArthur, R.H. & E.O. Wilson. 1967. The theory of island biogeography. Princeton Univ. Press.

Melo, A.S. 2004. A critique of the use of jackknife and related non-parametric techniques to estimate species richness. Community Ecology 5: 149-157.

Melo, A.S. 2008. O que ganhamos 'confundindo' riqueza de espécies e equabilidade em um índice de diversidade? Biota Neotropica 8: 21-27. Disponível em: (http://www.biotaneotropica.org.br/v8n3/pt/fullpaper?bn00108032008+pt).

Melo, A.S., M.V. Cianciaruso & M. Almeida-Neto. 2014. treeNODF: nestedness to phylogenetic, functional and other tree-based diversity metrics. Methods in Ecology and Evolution 5: 563-572.

Nicolelis, M. 2011. Muito além do nosso eu - A nova neurociência que une cérebro e máquinas e como

ela pode mudar nossas vidas. Cia das Letras.

Paradis E. 2012. Analysis of phylogenetics and evolution with R. 2nd ed. Springer, New York.

Quammen, D. O. 2008. O canto do dodô: Biogeografia de ilhas numa era de extinções. Cia das Letras.

Rigler, F.H. & R.H. Peters. 1995. Science in Limnology. Ecology Institute. Oldendorf. PDF em http://www.int-res.com/articles/eebooks/eebook06.pdf

Schneck, F., A. Schwarzbold & A.S. Melo. 2011. Substrate roughness affects stream benthic algal diversity, assemblage composition, and nestedness. Journal of the North American Benthological Society 30: 1049-1056.

Tóthmérész, B. 1995. Comparison of different methods for diversity ordering. Journal of Vegetation Science 6: 283-290.

Wilson, E.O. 1994. Naturalista. Nova Fronteira.