Mais um manual básico de R
versão 1.0.0

Prof. Dr. Marcelo de Oliveira Rosa

22 de abril de 2021

Resumo

Este documento apresenta principais funções do R para uso em diversas aplicações. Não tenho a pretensão de prover uma apostila completa dessa extensa e flexível ferramenta.

A intenção aqui é ajudar o aluno que tem seu primeiro contato com R, apresentando os conceitos fundamentais que sustentam seus tipos de dados e funções associados.

Espero que goste e me envie quaisquer sugestão ou comentário (críticas também são sempre bem vindas) para aprimorar este material.

Prof. Marcelo de Oliveira Rosa

Sumário

Capítulo 1

Introdução

Certamente há uma quantidade significativo de apostilas e guias para a linguagem `R`. Pode-se dizer que constantemente reinventa-se a roda. Claro que cada material tem um objetivo específico, orientando o leitor para uma ou outra função ou problema em particular.

O presente material visa atender alunos dos cursos que ministro e que usam `R` na resolução de problemas. A primeira vista, parece uma linguagem estranha, começando pela forma como as atribuições são feitas. Devemos nos lembrar que `R` é um produto livre (não confundir com gratuito, que é um conceito bem diferente) concebido para a comunidade de estatística que conhecia o produto comercial S.

Há uma comunidade bastante ativa na manutenção desta linguagem e seu software, bem como uma rica e extensa quantidade de bibliotecas que cobre, eu diria, qualquer problema que exijam um tratamento estatístico.

Você poderia perguntar: mas por que não usar linguagens modernas como Python (este texto está sendo escrito em 2020)? Na verdade, sim, poderia ser usado, assim como outras como Perl, Julia e, até mesmo, MathWorks Matlab ou Statistica. Ou seja, ferramenta não falta.

Entretanto, meu objetivo é duplo: ensinar uma nova ferramenta, para ampliar nosso leque e fazer com que saíamos de uma zona de conforto, e mostrar que a ferramenta não importa: o que importa é você entender os princípios de uma linguagem de programação e adaptar seu conhecimento adquirido em programação e estruturas de dados para uma nova realidade que o `R` lhe apresenta.

Assim, esta apostila lhe apresentará funções básicas que `R` fornece, as estruturas de dados que estão intimamente ligadas às análises estatísticas e alguns conceitos de estatística que ajudem a entender essas funções e estruturas de dados.

Capítulo 2

Linha de Comando

Quando você baixa o aplicativo `R` da internet (lembre-se de usar sites confiáveis). A fonte principal de informação (incluindo *download*) é `https://www.r-project.org/`. Procure usar sempre a versão mais estável do aplicativo.

Uma alternativa bastante interessante é o RStudio, aplicativo desenvolvido pela empresa RStudio (mesmo nome do produto), que é um ambiente integrado de desenvolvimento (*IDE* ou *Integrated Development Environment*). Ele é encontrado em `https://rstudio.com/`. Permite que você trabalhe com projetos, gerencie intuitivamente as variáveis que cria (inclusive editando-as graficamente), organize as figuras, e controle o ambiente de `R` mais facilmente. Além disso, seu ambiente para criação e edição de *scripts* é muito similar àqueles existentes para linguagens de programação como C ou Python.

Uma vez instalado, você notará que sua interface (*GUI*) é bastante simples: há uma janela para que você digite comandos e obtenha respostas imediatas (*shell*), menu de opções para criação e abertura de arquivos de comandos (*scripts*), janela para visualização de gráficos (só aparece quando você efetivamente gera um gráfico - você verá isso a seguir) e menus para gestão de pacotes/bibliotecas.

Toda a operação em `R` é feita a partir de comandos digitados no *shell* (um *script* é um conjunto de comandos que você digitaria nessa janela). Pense que você está interagindo com `R`: você digita um comando e ele responde.

`R` expressa sua disponibilidade no *shell* apresentando o caracter `>`. Nesse momento podemos digitar qualquer comando.

Exemplos de operações com `R`:

```
> 5 + 5
[1] 10
> plot(rnorm(1000))
> q()
```

O comando `q()` permite que você encerre a execução de `R`. Note que o aplicativo pergunta se deve salvar o ambiente em arquivo ou não. Basicamente significa que você pode persistir todas as variáveis que você criou e atualmente estão disponíveis no aplicativo em um arquivo, para poder retormar o trabalho em outra oportunidade.

Capítulo 3

HELP, I need somebody's help

Talvez a função mais importante para qualquer usuário de `R` (iniciantes e experientes) seja a função `help()`, também acessível pela função simbólica `?`. Basicamente indicamos a função ou objeto cujo conteúdo de auxílio desejamos acessar:

```
> help()
> help(sin)
> ?sin
> help(tapply)
> help(t.test)
```

Outra forma interessante de obter informações sobre uma função exige que apenas digitemos seu nome, sem os parênteses ou parâmetros.

```
> help
> sin
> tapply
> t.test
```

Algumas funções retornam inclusive um *script* em `R` mostrando a função que será executada. Em sua maioria elas são apenas o código-fonte associado a um código binário efetivamente executado. Outras informações são dadas.

No caso de funções que criamos, a digitação do seu nome faz com que `R` mostre o código-fonte que escrevemos (posteriormente veremos que esse código é aquele que carregamos para execução via função `source()`).

Capítulo 4

Operações básicas

4.1 Variáveis

Primeiramente vamos considerar o termo objeto como uma área de memória cujo conteúdo pode ser um valor, um vetor, ou qualquer outra estrutura de dados. Esta área de memória tem um endereço inicial que não é observada diretamente em `R`.

Este objeto tem um nome, que é atribuído ao objeto através de operadores de atribuição. Assim, em `R`, variáveis são objetos com nomes (uma função também pode ser entendido como um objeto - sua implementação, o que a função faz - com um nome - o nome da função).

A criação de variáveis em `R` (atribuição de um nome a um objeto) é feita de modo peculiar, usando o símbolo composto `<-`. Versões mais recentes, incluindo a atual, já permitem o uso alternativo de `=`. Em ambos os casos não há necessidade de definição de tipos (como fazemos em C/C++, por exemplo, apesar de algumas situações específicas exigirem isso), pois `R` deduz o tipo mais apropriado para o objeto.

```
> uma_variavel <- 5 + sin(3)
> uma_variavel <- uma_variavel + 1
> outra_variavel <- c(3, 4, 2, 4)
> outra_variavel <- exp(outra_variavel)
> resultado.total <- sum(outra_variavel)
```

Alguns lembretes na hora de criar nomes os objetos (variáveis):

1. Seus nomes não podem começar com números ou ponto (`.`) seguindo de número;
2. Não deve haver espaços "internos";
3. O uso de ponto (`.`) no início do seu nome tem significado especial (basicamente inibe sua visualização, apesar da variável existir).
4. Se o nome da variável for o mesmo de uma variável que já exista (ou até mesmo seja interna ao `R`) ou de uma função, ambas serão ignoradas enquanto a nova variável existir.

Atente para o último lembrete, pois você pode inibir o uso de funções ou valores como `TRUE` ou `FALSE`, e seus respectivos sinônimos `T` ou `F`.

É **FRISAR**: `R` fazem distinção entre maiúsculas e minúsculas (ou seja: Sin $\neq$ sin). O mesmo é válido para variáveis (descritas na seqüência).

Tecnicamente, a operação de atribuição liga (*bind*) um nome a um objeto. O `binding` permite que conectemos vários nomes a um mesmo espaço de memória. Isso significa que duas variáveis podem ter o mesmo conteúdo.

```
> install.packages('lobstr')
> x <- c(1, 2, 3, 4)
> y <- x
> obj_addr(x)
> obj_addr(y)
```

Assim, dois conceitos importantes aparecem: quando atribuímos um nome a outro, `R` simplesmente cria um novo nome que aponta para o mesmo objeto apontado por outro nome. Isso porque todos os objetos em `R` são imutáveis. Quando um objeto perde seu nome, o gerenciador de memória (*garbage collector*) libera a memória ocupada pelo objeto. Esta conceito é chamado *copy-on-modify*.

Qualquer operação que leve a alteração do objeto faz com que `R` gere uma cópia desse objeto com a versão modificada do objeto, com duas exceções:

- Quando o objeto recebeu apenas um nome e `R` até o momento da modificação do seu conteúdo, permitindo que `R` evite uma nova alocação de memória para manter esse objeto (nesse caso, a imutabilidade do objeto é desconsiderada);
- Para ambientes (que são locais que armazenam os nomes dos objetos), que são sempre mutáveis.

```
> x <- c(1, 2, 3, 4)
> y <- x
> obj_addr(x)
> obj_addr(y)
> x[[1]] <- 10
> obj_addr(x)
> obj_addr(y)
> v <- c(1, 2, 3, 4)
> obj_addr(v)
> v[[1]] <- 10
> v[2] <- 10
> obj_addr(v)
```

Uma modificação do objeto obedecendo essas exceções é chamada de *modify-in-place*.

Os nomes de objetos criados (doravante chamados de variáveis) podem ser listadas através da função `ls()` (a função `objects()` faz exatamente a mesma coisa, inclusive tem os mesmos parâmetros). O ambiente RStudio automaticamente mostra as variáveis existentes em janela específica.

```
> ls()
> ls(all.names = TRUE)
> objects()
> objects(all.names = T)
```

Note que estas funções retornam um vetor de *strings* contendo nomes de objetos existentes.

Há um tipo de objeto - chamado ambientes (*environments*) - que `R` usa para agrupar coleções de nomes de objetos (valores, funções, expressões, etc.) de modo compartimentalizado. Isso permite a criação de "grupos" de objetos acionados de acordo com a necessidade. Efetivamente as funções `ls()` e `objects()` listam as variáveis do ambiente corrente ou de um ambiente específico (veja a lista de seus parâmetros). Veremos isso mais a frente.

Para remover uma variável, usamos a função `rm()`. Para isso precisamos saber o nome da variável ou variáveis que queremos eliminar do ambiente.

```
> rm(uma_variavel)
> rm(list = ls())
> objects()
> a <- 3
> b <- 4
> c <- 5
> d <- 7
> ls()
> rm(a)
> rm('b')
> rm(c, 'd')
```

O parâmetro mais importante dessa função é o nome do objeto. Caso desejese remover mais de uma variável, basta indicar um vetor de *strings* contendo o nome dos seus objetos, usando o parâmetro nomeado `list`, ou indicando os nomes das variáveis, um a um, com ou sem aspas. Posteriormente explicaremos o que são parâmetros nomeados para funções.

Como falamos, nomes de objetos podem "camuflar" outros já existentes, criando situações chatas, como esconder a existência dos valores `T` e `F` (abreviações existentes em `R` para `TRUE` e `FALSE`). Outra situação é ter funções com mesmo nome em pacotes distintos.

Para saber o caminho usado por `R` para chegar a um objeto contido em um pacote, usamos a função `search()`. Quando executado, `R` retorna uma lista das ambientes de pacotes que ele acessará para encontrar objetos e funções que queremos usar, por ordem de prioridade, ou seja, ele comecará suas buscas pelo que digitamos a partir do primeiro ambiente listado, até a último. Caso esses objetos e funções não sejam encontrados em nenhuma desses ambientes, `R` retorna um indicativo apropriado.

4.2 Pacotes

Como mencionado anteriormente, R possui um grande número pacotes (ou bibliotecas) para variados usos. Alguns pacotes são armazenados no computador no momento de sua instalação. Para usá-las, é necessárias carrega-las. Isso é feito através das funções `library()` ou `require()`. A diferença entre é como elas lidam quando o pacote que se deseja carregar não está no computador: `library()` gera uma mensagem de erro enquanto `require()` gera uma mensagem de advertência e retorna `FALSE`. Se tudo der certo, o pacote é carregado e suas funções e variáveis são inseridas no caminho de busca de R

```
> installed.packages()
> search()
> require(MASS)
> library(nnet)
> search()
```

Neste exemplo, listamos os pacotes disponíveis no computador e na sequência carregamos dois pacotes específicos.

A instalação de pacotes é feita pela função `install.package()`, tendo como parâmetro o nome do pacote desejado. A partir do nome do pacote, essa função acessa um repositório específico (CRAN, a princípio, mas pode ser alterado inclusive para acessar um diretório onde pacotes estão armazenados) para baixar o pacote (incluindo dependências, se quisermos). No LINUX, os códigos-fontes dos pacotes costumam ser baixados e compilados no computador, enquanto que no Windows, os códigos-binários é que são baixados.

Para exemplificar o processo, vamos baixar o pacote `ggplot2`, responsável por gerar gráficos usando uma abordagem de camadas, e atualizar os pacotes existentes em nossa máquina antes de carregar as funções e objetos desse pacote.

```
> install.packages("ggplot2")
> update.packages()
> require(ggplot2)
```

Note a diferença entre instalar um pacote e carregá-lo para uso em R. O primeiro caso consiste em trazer o pacote para a máquina (os arquivos ficam em um diretório particular que o R usa para tal fim), enquanto que no segundo caso, vincula-o (*attach*) ao caminho de busca. Falaremos mais sobre essa vinculação (e desnviculação) na próxima seção.

4.3 Controle do caminho de busca

Como mencionado nas Seções 4.1 e 4.2, R encontra objetos seguindo um caminho de busca. A criação ou remoção desses objetos e funções manipula indiretamente esse caminho. Entretanto, as funções de manipulação `attach()` e `detach()` manipulam diretamente esse caminho.

A função `detach()` remove um objeto (um pacote, um quadro de dados (*data frame*), ou um ambiente de variáveis) do caminho de busca de R. Aqui trataremos de pacotes e posteriormente lidaremos com quadro de dados.

Adicionalmente, a função `find()` localiza o pacote ou ambiente no qual uma função ou objeto está armazenado e disponível para `R`.

```
> detach(ggplot2)
> detach(nnet)
> detach(MASS)
> search()
> a <- 1
> find("a")
> find("t.test")
> find("cars")
> detach("package:datasets")
> find("cars")
> library("datasets", pos = 7)
```

4.4 Ambientes (*Environments*)

Como mencionado anteriormente, `R` agrupa nomes de objetos dentro de estruturas chamadas ambientes. Podemos pensar nelas como *namespaces* de C/C++ ou JAVA. Os objetos estão na memória mas o seu nome é sempre armazenado em um ambiente.

O ambiente *default* em `R` é o ambiente global, identificado pelo nome `.GlobalEnv` no caminho de busca (note o ponto antes do nome, para "esconde-lo" de uma listagem, mas ele está disponível se quisermos acessá-lo). Há também uma função para acessá-lo (`globalenv()`).

Outros dois ambientes existentes são `base` (ou básico, do pacote de mesmo nome) e `empty` (ou ambiente vazio). Estes são acessíveis através das funções `baseenv()` e `emptyenv()`, respectivamente.

Além desses ambientes, os pacotes carregados trazem ambientes com seu nome (vimos inclusive que podemos alterar a posição de um pacote dentro do caminho de busca de `R`). Se nos aprofundarmos em pacotes, veremos que pacotes em `R` na verdade contém dois tipos de ambientes: aquele ambiente que é colocado no caminho de busca (chamado de ambiente do pacote) e um ambiente especial (chamado de ambiente de espaço de nomes do pacote) para manter as dependências entre suas funções (isso é para garantir que a criação de uma função de mesmo nome que uma de uso interno do pacote não interfira com o funcionamento de outras funções do pacote que dependem dela). É bom reforçar que um ambiente não precisa ter ou ser associado a um pacote.

Os três ambientes básicos (global, básico e vazio) são organizados em uma fila simplesmente encadeada, nesta sequência. Internamente há um ponteiro (chamado `parent`) que aponta o próximo ambiente no caminho de busca. O primeiro ambiente é SEMPRE o global (não há como alterar isso), o último é SEMPRE o ambiente vazio, que é SEMPRE antecedido do básico (o ambiente vazio funciona como um indicador de fim da fila). Usamos a função `parent.env()` Para identificar o próximo ambiente de um dado ambiente.

Quando um pacote é carregado, seu ambiente é inserido sempre depois do ambiente global, salvo desejarmos inseri-lo em algum outro ponto entre o global e o básico (vimos isso em um exemplo da Seção 4.3.

$$\texttt{globalenv()} \rightarrow \cdots \rightarrow \texttt{baseenv()} \rightarrow \texttt{emptyenv()}$$

Na literatura de R, essa flecha indica o pai (ou mãe) (*parent*) do pacote (ou `parent.env(baseenv()) = emptyenv()`).

Para criar um ambiente específico (para inserir seus objetos dentro de um contexto particular), usamos a função `new.env()`. E para listar seu conteúdo, basta usar a função `ls()`

```
> globalenv()
> parent.env(globalenv())
> baseenv()
> parent.env(baseenv())
> emptyenv()
> identical(parent.env(baseenv()), emptyenv())
> parent.env(emptyenvy())
> ambiente <- new.env()
> parent.env(ambiente)
> as.environment("package:stats")
> ls(ambiente)
```

Para criar um objeto em um ambiente, usamos a operação de atribuição que já conhecemos, com uma atenção especial para nomeá-lo. Usamos o operador de referência `$` da seguinte forma:

`<ambiente>$<nome da variável> <- <valor>`,

que semanticamente significa "atribua `<valor>` ao objeto cujo nome é `<nome da variável>` pertencente ao ambiente `<ambiente>`". A função `assign()` também permite efetuar tal operação, só que ela usa uma *string* como nome do objeto. Isso é interessante, pois permite geremos dinamicamente objetos que são reconhecidos como se fossem parte do código executável.

Além dessas funções, é possível verificar a existência e localizar objetos pelos seus nomes dentro do caminho de busca de R. Isso é feito, respectivamente, pelas funções `exists()` e `where()`.

```
a <- 1
ambiente$a <- 4
ambiente$b <- ambiente$a
ambiente$e <- 8
assign('c', 2, ambiente)
assign('b', 3)
d <- 5
ls()
ls(ambiente)
where('d', env = ambiente)
where('e')
exists('e')
exists('d')
```

Capítulo 5

Tipos de variáveis

Neste capítulo apresentaremos os tipos de variáveis suportados por `R`, incluindo algumas estruturas de dados importante, como o *data frame*.

5.1 Básicos

Há 7 tipos básicos visíveis ao usuário (há outros internos, cujo acesso é indireto e não serão tratados aqui), a saber:

- Caracter (`character`), que compreende sequencias de caracteres entre aspas (duplas ou simples);
- Inteiros (`integer`), ou qualquer valor pertencente ao conjunto dos números inteiros (com ou sem sinal);
- Numérico (`numeric` ou `double`), ou qualquer valor pertencente ao conjunto dos números reais (com ou sem sinal);
- Complexo (`complex`), ou qualquer valor pertencente ao conjunto dos números complexos (com ou sem sinal), sempre no formato `<parte real>+<parte imaginária>i`;
- Lógico (`logical`), ou valores de álgebra booleana;
- "Cru" (`raw`), que é um tipo para armazenar e trabalhar com *bytes* (e *bits*);
- Nulo (`NULL`), que representa a ausência de objeto, usado em situações para indicar erro, por exemplo.

Lembrando que é possível fazer conversões (*cast*) entre esses tipos, assumindo possíveis perdas por truncamento. Note também que o tipo `character` é a denominação de *string* em `R` (veja a Seção 5.2).

```
> rm(list = ls())
> a <- "um texto (character)"
> b <- 2.0  # um valor numérico
> c <- 2L   # um valor inteiro
> d <- 2+0i # um valor complexo
```

```
> e <- T    # um valor lógico
> f <- raw(4) # um valor cru de 4 bytes (todos nulos)
> # uma conversão para vermos o conteúdo de g
> g <- charToRaw("bom dia")
> h <- as.raw(d)
```

Há outros tipos visíveis para representar objetos internos (que são implementados dentro do núcleo do aplicativo e não dependem de pacotes), para representar ambientes (estruturas que mantém um conjunto de variáveis, para agrupá-los segundo algum contexto do usuário), para representar as próprias funções (sejam elas presentes em pacotes ou descritas pelo usuário), para representar expressões (representação geralmente matemática que não é imediatamente interpretada/executada por `R`) e símbolos (ou nomes, que são variáveis não interpretadas/executadas por `R`), e objetos da classe S4, que representam uma estrutura forma de orientação a objetos em `R` (você também ouvirá falar de objetos da classe S3, que é/foi uma tentativa de incorporar orientação a objetos em `R` usando `list()` e `class()`).

```
> meu_seno <- sin
> typeof(meu_seno)
> typeof(sin)
> funcao <- function(x) sqrt(x\%*\%x)
> typeof(funcao)
> lista <- list(3, 4, 5)
> typeof(list)
> expressao <- expression(sin(x) + y)
> typeof(expressao)
> typeof(pi)
> meu_pi <- quote(pi)
> typeof(meu_pi)
> expressao_mastigada <- substitute(expression(sin(x)+y))
> typeof(expressao_mastigada)
> novo_ambiente <- new.env(parent = baseenv())
> typeof(novo_ambiente)
> setClass("aluno", slots = list(nome = "character",
 idade = "integer", nota = "numeric"))
> um_aluno <- new("aluno", nome = "Marcelo",
 idade = 20, nota = 6.5)
> typeof(um_aluno)
```

A grosso modo, expressões e símbolos/nomes são parte do que se chama interpretação/execução não padronizada (ou *non-standard evaluation*). Isso significa que `R` aplica suas análises sintáticas sobre expressões sem executá-las. Posteriormente essas expressões são efetivamente executadas a partir de atribuições apropriadas para as variáveis "prometidas" (`promisse`). Parece com função, mas é mais versátil, pois podemos criá-las em tempo de execução a partir de *strings*.

A função `typeof()` é usada para determinar o tipo dito de baixo nível. Ou seja, ele retorna uma *string* indicando um dos tipos básicos de `R`.

Outra função interessante é `mode()`, que permite identificar a estrutura interna de armazenamento dos objetos. Sugiro substituir `typeof` por `mode` para conferir.

`R` também possui funções que testam o tipo de um objeto. Todas elas são escritas como `is.<tipo>()`, retornando um valor booleano. Já as funções escritas como `as.<tipo>()` são úteis para fazer conversão (*casting*) entre diferentes tipos (podendo resultar em algum truncamento) ou até mesmo impossibilidade da conversão.

```
> is.double(2.5)
> as.double("5.2")
> is.integer(2L)
> is.logical(TRUE)
> is.character("meu nome")
> as.character(1000L)
```

5.2 *Strings*

Uma *string* é um conjunto de caracteres "cercado" de um par de aspas simples ou duplas. A ideia de usar tanto aspas simples quanto duplas é permitir redigir textos com aspas simples ou duplas sem muita dificuldade. Caracteres especiais são inseridos com a ajuda da barra invertida (\). Abaixo seguem exemplos de *strings* em atribuições com `R`.

```
> nome       <- "meu nome"
> outro.nome <- 'meu nome'
> com.aspas.duplas  <- '"nome com aspas duplas"'
> com.aspas.simples <- "'nome com aspas duplas'"
> com.tabulador <- "nome\tidade\tcidade"
```

Caracteres especiais são inseridos com a ajuda da barra invertida (no exemplo, o tabulador - `\t`).

5.3 `NULL`, `NaN`, `Inf` e `NA`

`R` dispõe de alguns valores especiais para representar algumas situações importantes, sejam elas ligadas à programação ou a eventos estatísticos. Vimos anteriormente que o valor `NULL` representa a inexistência de um objeto. Pode ser usado quando alguma condição de uma função exige tal indicação (por exemplo, quando pesquisamos por um nome inexistente em uma lista). Como já vimos, o tipo desse objeto é específico (também chamado `"NULL"`)

Já os valores `NaN` (do inglês *Not A Number*) são valores numéricos usados para representar situações bem específicas. No primeiro caso, `NaN` é um resultado comum para situações como:

- 0/0
- arcsin(2)

- $\log(-1)$

Nessas e em outras situações, a solução é indefinida, fazendo com que `R` e outras linguagens tenham um valor especial para representar essa solução, ou seja, `NaN`. Note a grafia desse símbolo.

O valor `Inf` (novamente, atenção a sua grafia) representa o valor ∞, podendo ser antecedido pelo sinal negativo (como $-$`Inf` representando $-\infty$). Ele aparece como resultado de operações matemáticas como:

- $1/0$
- $\log(0)$

Finalmente, o valor `NA` representa a falta de um valor (*Not Available*). Sua existência deve-se a necessidade de registrar a falta de um valor em uma coleta de dados para análise estatística (por exemplo: pode ser que um aluno tenha faltado a uma prova, fazendo com que o professor defina sua nota como não disponível - até que haja uma segunda chamada). Posteriormente veremos que muitas funções estatísticas de `R` permitem que ignoremos esses valores no cálculo feito sobre um vetor, por exemplo. Esse controle do uso de `NA` é feito por parâmetros de função `na.rm`.

É recomendável em `R` que se use algumas funções específicas para checar se o valor de um objeto (expressão matemática, variável simples, etc.) é um desses valores especiais ou não. Essas funções são geralmente grafadas como `is.<algo>()`.

```
> 0/0
> log(-1)
> 1/0
> log(0)
> is.null(NULL)
> is.nan(NaN)
> is.nan(0/0)
> is.infinite(Inf)
> is.infinite(log(0))
> NA + 3
> -1 * Inf
> 3 * Inf
> is.na(NA)
> notas <- c(NA, 3, 3, NA)
> is.na(notas)
> mean(notas)
> mean(notas, na.rm = FALSE)
```

Note particularmente que `NA` e `NaN` acabam "absorvendo" qualquer outro valor quando realizamos operações matemáticas com eles. O caso de `Inf` é ligeiramnete diferente, pois podemos alterar seu sinal por multiplicação com números negativos. Todas as operações matemáticas com esses valores especiais são matematicamente consistentes.

5.4 Tipos "estruturais"

Usei o termo estruturais para definir tipos que são considerados estruturas de dados em outras linguagens de programação. Por exemplo, uma lista é considerada um tipo básico, mas prefiro defini-lo como um tipo estrutural.

Em `R`, todos os tipos estruturais podem ser identificados usando a função `class()`. Essa função recupera uma *string* contendo o nome do tipo estruturado do objeto.

Os principais tipos estruturais são:

- Vetor de concatenação, ou `c()`;
- Lista de dados, ou `list`;
- Arranjo bidimensional ou matriz, ou `matrix()`;
- Arranjo multidimensional, ou `array()`;
- Fatores, ou `factor()`
- Quadro de dados, ou `data.frame()`;

Internamente, a estrutura fundamental de `R` para suporte ao tipos estruturados é o vetor, que é um bloco de memória dividido em N partes iguais (tamanho do vetor) e cada parte do vetor é capaz de armazenar um valor de um único tipo. Caso esse tipo seja um dos tipos básicos já mencionados, esse vetor interno passa a ser um tipo estruturado (disponível para o usuário) chamado vetor atômico. Esse termo vem do fato de que o tipo dos valores que ele armazena deve ser atômico (forma de `R` dizer que se trata de um tipo básico). Para saber se um tipo é atômico ou não, basta usar a função `is.atomic()`.

Entretanto, se o vetor interno conter endereço de memória para outras áreas de memória que podem ter tamanhos de alocação diferentes (suportando tipos diferentes de objetos), o vetor interno passa a ser um tipo estruturado chamado lista.

Qualquer vetor de `R` pode ter atributos (metadados associados ao objeto), sendo dois atributos particularmente importantes: o atributo de dimensão (que permite a caracterização de matrizes como arranjo lógico estruturado sobre conjunto "contínuo" de objetos) e o atributo de classe (que permite uma construção simples de orientação a objetos - modelo S3 já comentado anteriormente). É importante salientar que qualquer objeto pode receber atributos, de acordo com as necessidades do usuário. As funções `attr()` e `attributes()` recuperam e definem atributos para os objetos. A função `structure()` também pode ser usada para esse fim.

A função `c()` é frequentemente utilizada para definir vetores. O processo é chamado concatenação, na qual todos os objetos listados como parâmetros dessa função são armazenados em um vetor atômico.

O tipo do vetor resultante é definida pelo maior tipo dos objetos a serem concatenados, com ordinalidade definida por:

$$\texttt{NULL} < \texttt{raw} < \texttt{logical} < \texttt{integer} < \texttt{double} < \texttt{complex} < \texttt{character} < \texttt{list} < \texttt{expression}$$

Isso significa que se um dos objetos a serem concatenados for uma lista, o resultado será uma lista, a menos que outro objeto seja uma expressão. Esse ajuste de tipos é chamado de coersão (*coersion*) em R.

```
> rm(list = ls())
> vetor <- c(1, 2, 3, 4)
> vetor
> str(vetor)
> vetor_com_nomes <- c(a = 1, b = 2, c = 3, d = 4)
> vetor_com_nomes
> matriz <- vetor
> dim(matriz) <- c(2, 2)
> matriz
> str(matriz)
> attributes(matriz)
> attributes(matriz) <- NULL
> matriz
> dim(matriz) <- c(4, 1)
> attributes(matriz)\$temperatura <- 20
> attr(matriz, 'pressao') <- 'alta'
> colnames(matriz) <- 'B'
> rownames(matriz) <- c('C1', 'C2', 'C3', 'C4')
> outro <- structure(2, temp = 3, pres = 'alta')
> str(outro)
> matriz <- matrix(vetor, nrow = 2)
> str(matriz)
> outra_matriz <- array(vetor, dim = c(2, 2))
> str(outra_matriz)
> matriz <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9),
 ncol = 3, byrow = TRUE)
> matriz <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9),
 ncol = 3, byrow = F)
```

Note que as funções `array()` e `matrix()` basicamente reestruturam (como uma *view*) os dados (com coersão, se necessário) para representarem matrizes bidimensionais ou arrays multidimensionais. Internamente os objetos são dos tipos básicos vetor e lista.

Note que usamos a função `str()` para visualizar muitos dos objetos de tipo estruturado. Essa função mostra a estrutura interna do objeto, serve para entendermos aspectos técnicos de R e ajuda na depuração de *scripts*.

Uma lista é definida pela função `list()`. Seus parâmetros são aglutinados em uma lista, que internamente é formada por um vetor de endereços apontando para a memória alocada para cada desses parâmetros.

```
> lista1 <- list(c(1, 2, 3, 4))
> lista1
> lista2 <- list(1, 2, 3, 4)
> lista2
> names(lista2)
> lista3 <- list(x = 1, y = 2, z = 3, w = 4)
```

```
> lista3
> names(lista3)
> lista4 <- c(lista3, lista3)
> lista5 <- list(lista3, lista3)
> lista6 <- list(list(list(list(3))))
```

Podemos considerar fatores e *data frames* como versões especializadas de vetores e matrizes, respectivamente.

Fatores são vetores contendo apenas valores pré-definidos sem repetição (os níveis ou *levels*). Quando usamos algumas funções que importam planilhas de outros aplicativos, é comum que elas recebam colunas com valores do tipo *string* e que haja poucos valores únicos com grande repetição. O tipo fator otimiza o armazenamento desse tipo de dado. Internamente os vetores contém apenas índices para um segundo vetor contendo os níveis. A função `factor()` cria objetos desse tipo estruturado.

```
> viagem <- factor(c('CWB', 'CWB', 'NYC',
 'CDR', 'GRU', 'CWB', 'NOR'))
> letters
> quebra_letras <- substring('estatistica',
 first = 1:11, last = 1:11)
> letras_texto <- factor(quebra_letras, level = letters)
> letras_texto
> str(letras_texto)
> letras_texto[1]
```

É comum termos funções de `R` que inibem a conversão de dados do tipo *string* para fatores: elas trazem o parâmetro booleano `stringsAsFactor`, que deve ser fixado em `FALSE`.

Data frame é um tipo estrutural muito significativo para `R`. É implementado internamente como uma lista nomeada de vetores de mesmo tamanho (outra visão simples para esse tipo é uma matriz, com cada coluna tendo um nome). Do ponto de vista estatístico, cada coluna representa uma característica do espaço amostral, enquanto que cada linha é uma amostra (com todas as suas características). Isso permite aplicar métodos estatísticos para traçar correlações ou teste de hipóteses sobre tais dados a partir de funções de `R` que consideram os dados amostrais estruturados em *data frames*. Naturalmente é possível efetuar conversões de matrizes para *data frames* e vice-versa.

```
> turma <- data.frame(id = c(1, 2, 3, 4, 5),
 turno = c('N', 'N', 'D', 'D', 'D'), nota1 = c(5, 7, 10, 8, 8),
 nota2 = c(7, 4, 8, 9, 7), stringsAsFactors = T)
> names(turma)
> row.names(turma)
> rownames(turma)
> colnames(turma)
> length(turma)
> ncol(turma)
> nrow(turma)
```

```
> turma
> summary(turma)
> str(turma)
```

A construção do *data frame* é feita pela função `data.frame()`. Basicamente incluimos quantos vetores (agregador `c()`) quiseremos, nomeando-os diretamente na chamada da função: obrigatoriamente eles têm que ter o mesmo tamanho.

As funções `names()` e `colnames()` recuperam os nomes das colunas (retornam vetor de *strings*), sendo sinônimas para este tipo estruturado. Já a função `rownames()` (e seu sinônimo `row.names()` - há várias funções que possuem pontos internos para representar espaços e facilitar sua leitura).

A função `length()` recupera o número de observações (nome técnico em `R` para número de amostras). Também pode ser utilizada para vetores ou listas. As funções `ncol()` e `nrow()`

Finalmente, introduzimos uma nova função (`summary()`) que calcula um resumo estatístico de um objeto. No caso de *data frames*, valores mínimos, médios e máximos, desvio padrão, mediana, 1^o e 2^o quartis. No caso de fatores, computa-se o número de amostras para cada nível do fator.

Para adicionar novas amostras/observações e novas características a um *data frame*, usamos as funções `rbind()` e `cbind()`, do inglês *row binding* e *column binding*, respectivamente. Elas podem ser usadas em vetores.

```
> nova_observacao <-
 data.frame(id = 6, turno = 'D', nota1 = 7.5, nota2 = 8)
> turma <- rbind(turma, nova_observacao)
> nova_caracteristica <- data.frame(final = c(6, 7, 10, 5, 4, 7))
> turma <- cbind(turma, nova_caracteristica)
> summary(turma)
```

5.5 Obtendo subconjuntos

Subconjuntos são partes de objetos estruturados, que possuem o mesmo tipo do objeto de origem. Assim, um subconjunto de um vetor também é um vetor, de uma lista é uma lista, etc. A geração de subconjuntos também é conhecida como *slicing* ou *subsetting*.

Os operadores de subconjunto são `[]`, `[[]]` e `$` (já usamos este último na Seção 4.4).

Os parâmetros dos dois primeiros operadores indexam elementos de um objeto. Sua diferença reside no tipo do objeto retornado: enquanto o operador `[]` retorna um objeto contendo elementos indexados e seu tipo igual ao do objeto indexado, o operador `[]` retorna apenas um elemento do objeto indexado (só permite um índice) cujo tipo é igual ao tipo dos elementos do objeto estruturado. Os índices são valores inteiros ou booleanos (se forem valores reais, serão truncados para valores inteiros).

```
> rm(list = ls())
> vetor <- c(1.1, 2.2, 3.3, 4.4, 5.5)
```

```
> lista <- list(1.1, 2.2, '3.3', 4.4, '5.5')
> vetor
> summary(vetor)
> lista
> summary(lista)
> lista[]
> vetor[1]
> vetor[[1]]
> lista[1]
> lista[[1]]
> lista[c(1, 3, 5)]
> lista[c(3, 1, 5, 3)]
> vetor[-1]
> vetor[-2]
> vetor[c(-1, -4)]
> vetor[c(T, T, F, F, T)]
> lista[c(T, T, F, F, T)]
> vetor[vetor>4]
> lista[lista>4]
> vetor[0]
> lista[0]
> vetor[1:3]
> lista[-5:-3]
> vetor[seq(from=1, to=5, by=2)]
> vetor[1:10]
> vetor[1:8] <- c(1, 2)
> vetor
```

Note que a diferença entre esses operadores é mais significativa para listas. Isso ocorre porque o resultado da indexação é uma nova lista contendo os elementos indexados.

Para os índices, valores positivos significam "selecione este elemento" e valores negativos significam "ignore este elemento" (não é possível misturar índices positivos e negativos). O índice zero é apenas para garantir consistência com índices negativos e positivos (retorna um objeto vazio).

Índices booleanos indicam se determinado elemento (de acordo com a posição ordinal no índice) fará parte do resultado ou não. Se o comprimento do vetor de índices booleano for menor do que o comprimento do objeto indexado, `R` replica circularmente esses índices booleanos até o comprimento do objeto indexado. No caso do comprimento do vetor de índices booleano for maior do que o comprimento do objeto indexado, `R` retorna `NA` para aquelas posições inexistentes do objeto indexado.

Índices booleanos também pode ser construídos a partir de lógica booleana aplicada ao objeto. Quando realizamos uma operação lógica sobre um vetor, por exemplo, `R` aplica essa lógica a cada um de seus elementos, retornando `TRUE` e `FALSE`.

Note que podemos gerar índices usando algumas funções e operadores específicos de `R`. A função `seq()` gera um vetor a partir das definições de valor inicial, final e intervalo. Já o operador `:` gera um vetor numérico sequencial, do

valor esquerda dos dois-pontos até o valor a direita (R detecta automaticamente se a sequência é crescente ou decrescente).

```
> lista_lista <- list(lista, list(lista, lista))
> lista_lista
> summary(lista_lista
> lista_lista[[1]][1]
> lista_lista[[1]][[1]]
> lista_lista[2][1]
> lista_lista[2][1][1]
> lista_lista[2][1][1][1]
> lista_lista[2][1][[1]]
> lista_lista[[2]][[1]][[1]]
```

Devemos lembrar que subconjunto de listas são listas, enquanto que seus elementos podem ter tipos diversos. Assim, devemos ter cuidado para não confundir o uso de [] e [[]].

```
> matriz <- matrix(1:9, ncol = 3)
> transposta <- t(matriz)
> turma <- data.frame(id = c(1, 2, 3, 4, 5),
 turno = c('N', 'N', 'D', 'D', 'D'), nota1 = c(5, 7, 10, 8, 8),
 nota2 = c(7, 4, 8, 9, 7), stringsAsFactors = T)
> matriz
> transposta
> matriz[c(2:1),]
> matriz[, 2:3]
> matriz[,2:3] <- matrix(c(10:15), ncol = 3)
> matriz
> matriz[1:5]
> matriz[2,2] <- 20
> matriz[2:3,] <- c(1, 2)
> rownames(matriz) <- c('dia', 'mes', 'ano')
> colnames(matriz) <- c('cdb', 'di', 'acoes')
> str(matriz)
> matriz['dia',]
> matriz[, 'acoes']
> matriz['mes', 'di']
> matriz[,c('acoes', 'di')]
> turma[3]
> turma['nota1']
> turma[c(1,3)]
> turma[1:2,]
> turma[1:2, c('id', 'nota1')]
> turma$nota1
> turma[1:3,]$id
> turma$final <- (turma$nota1 + turma\$nota2)/2
> turma[2:4, 'final']
> turma[2:4, 'final', drop = FALSE]
> turma <- transform(turma,
```

```
 resultado = ifelse(final>6.0, 'aprovado', 'reprovado'))
> turma
> subset(turma, final>=6)
> subset(turma, final>=6, select = c(id, final))
> subset(matriz, matriz[,1]==1, select = 2:3)
```

As regras de indexação de matrizes e arranjos multidimensionais são equivalentes às de vetores: podemos indexá-los linearmente (já que os elementos de matrizes e arranjos são visões de vetores) ou indexá-los por linha, coluna, ou por qualquer dimensão. Particularmente perceba que deixando uma dimensão vazia implica em considerar todos os elementos daquela dimensão.

Podemos também substituir valores dentro de vetores e arranjos multidimensionais, valendo-se inclusive da capacidade de reciclagem de `R` (isso ocorre quando a quantidade de elementos indexados é maior do que a quantidade de valores substitutos, fazendo com que `R` repita os valores substitutos até preencher exigido).

A indexação de *data frames* é igualmente simples, considerando-os como matrizes cujas linhas e colunas são nomeadas. Adicionalmente, podemos empregar o operador de referência `$` para acessar colunas desse tipo estrutural: é uma questão de nível de inteligibilidade, particularmente para *scripts*. Como a seleção de linhas ou colunas individuais (independente do tamanho) pode resultar em vetores ao invés de *data frames* (como uma otimização em `R`), podemos usar o parâmetro `drop` para inibir tal otimização.

Outra forma flexível de obter subconjunto de tipos estruturais é através da função `subset()`, que permite aplicar expressões com resultado booleano para indexar as linhas desejadas - e selecionar as colunas de interesse.

Finalmente, a função `transform()` permite a criação de novas colunas a partir de operações realizadas com as colunas anteriores de um *data frame*.

5.6 Alguns vetores padronizados

Como vimos, alguns vetores podem ser criados automaticamente por `R`. Eles evitam a criação de algoritmos para sua criação. Já vimos alguns deles na Seção 5.5, como o `seq()` e o operador `:`.

```
> integer(10)
> double(15)
> logical(3)
> complex(5)
> character(4)
> raw(6)
> vector('integer', 10)
> rep(c(T, F), times = 4)
> rep(c(T, F), times = 1, each = 4)
> rep(c(T, F), times = 3, each = 5)
> rep(list(list('a', 'b'), F), times = 2, each = 2)
```

As funções `integer()`, `double()`, `logical()`, `complex()`, `character()` e `raw()` criam vetores com uma quantidade estipulada de elementos "nulos" (que

depende do tipo). Uma versão mais genérica - `vector()` permite que se forneça a *string* descritiva do tipo cujo vetor se deseja criar.

Já a função `rep()` permite que se crie um vetor ou uma lista (alguns tipos como matrizes e *data frames* são convertidos internamente para vetor ou lista), a partir de repetições controladas dos elementos de um objeto.

Capítulo 6

Funções

Neste capítulo detalharemos mais aspectos de funções. `R` dá muita flexibilidade na construção de funções, dado alguns conceitos importantes que queremos reforçar:

- Todas as funções são objetos com nome
- Todas as funções possuem três elementos importantes (compatíveis como serem objetos):
  - Argumentos (ou argumentos formais);
  - Corpo (que é o *script* propriamente dito - o que é ela faz);
  - Ambiente (escopo) das variáveis usadas pelo corpo da função.

Criamos uma função usando o operador `function()`, que define os argumentos formais (entre parênteses) e o corpo da função. Para funções longas, podemos envolver o corpo com chaves. Ou seja:

```
nome <- function(parametros) corpo
```

ou

```
nome <- function(parametros) { corpo }
```

Toda vez que `R` executa uma função, ele cria um ambiente *environment* temporário (que é eliminado na saída da função) para lidar com o escopo de variáveis locais e globais. Note também que o nome da função é o próprio nome de um objeto, unificando a visão de objetos

```
> rm(list = ls())
> media_ponderada <- function(x, y) sum(x * y)/length(x)
> copia_media <- media_ponderada
> media_ponderada(c(1, 2, 3), c(1, 1, 1))
> copia_media(c(1, 2, 3), c(0.5, 5, 2))
> media_ponderada
> formals(media_ponderada)
> body(media_ponderada)
```

```
> body(copia_media) <- quote(mean(x * y))
> copia_media(c(1, 2, 3), c(0.5, 5, 2))
> outra_media <- function() sum(x)/length(x)
> formals(outra_media) <- alist(x =)
> outra_media(c(1, 2, 3)
> do.call(outra_media, list(c(1, 2, 3)))
> estatistica <- list(media = function(x) mean(x),
 dp = function(x) sd(x))
> estatistica$media(c(1, 2, 3))
> estatistica$dp(c(1, 2, 3))
> body(sum)
> formals(sum)
```

As funções `formals()` e `body()` recuperam e atributem lista de parâmetros e seu conjunto de instruções (corpo), respectivamente. Note que para atribuir esses elementos a uma função, empregamos funções que obrigam `R` a postergar sua avaliação (expressões, como já vimos na Seção 4.1). A função `quote()` cria uma estrutura interna de execução sem que qualquer objeto seja executado ou avaliado (avaliar uma expressão ou objeto significa aplicar valores atuais de objetos nomeados e obter o seu resultado final). Já a função `alist()` gera uma versão de lista na qual seus parâmetros não são avaliados, sob medida para substituir os parâmetros formais de uma função. Note também que podemos executar uma função chamando `do.call()`, apenas lembrando que os parâmetros da função a ser executada devem ser descritos em uma lista (reforçando que argumentos formais são "listas" de objetos).

Você verá que algumas funções (ditas internas ou primitivas em `R`, como *sin*()) retornam `NULL`, pois que estão implementadas em linguagem de máquina a partir de codificação em linguagem C, por questões de performance.

Note que a última operação realizada no corpo define o valor de retorno da função. `R` possui a função `return()`, cujo parâmetro é retornado na saída da função, encerrando-a (a falta de um parâmetro resulta no retorno de `NULL`). Caso a última operação a ser executada do corpo da função não seja `return()`, o resultado dessa operação é tido como valor a ser retornado.

Os objetos usados pelo corpo da função obedecem regras de escopo (similares àquelas que encontramos em linguagem C), considerando a estrutura de ambientes existentes: primeiramente a busca de um nome de objeto é feito no ambiente criado especificamente para o corpo da função (que armazena o que conhecemos normalmente por variáveis locais), seguindo para seu ambiente "pai", até chegar no ambiente global de `R`. Assim, nomes de objetos de um nível podem ser mascarados pelos nomes de outro nível, de acordo com nossas necessidades.

```
cat('ambiente global\n')
str(environment())
str(parent.env(environment()))
x0 <- 1
func1 <- function() {
        cat('ambiente de func1\n')
        str(environment())
        str(parent.env(environment()))
        x1 <- 2
```

```
        func2 <- function() {
                cat('ambiente de func2\n')
                str(environment())
                str(parent.env(environment()))
                cat('ambiente de func2 -> func1\n')
                str(environment(func2))
                x1 <- -2
                x2 <- 3

                c(x0, x1, x2)
        }

        return(c(func2(), x0, x1))
}

c(func1(), x0)
func1
func2
```

Como uma função é um objeto nomeado (depois trataremos de funções anônimas), eventualmente ela pode ter o mesmo nome de outro objeto. Se isso acontecer, `R` preferirá usar a função caso o contexto exija a chamada de uma função (presença de `()`, por exemplo).

```
f <- function(x) {
  f <- function(x) {
    f <- function() {
      x ^ 2
    }
    f() + 1
  }
  f(x) * 2
}
f(10)
```

Outra característica de `R` é que ele avalia as operações a medida que as necessidade. Essa característica (*lazy evaluation*) otimiza a execução dos *scripts* direcionando os esforços de `R` para o que realmente interessa.

```
stop("Mensagem de erro!")

funcao_constante <- function(parametro) {
  10
}

funcao_constante(stop("Outra mensagem de erro!"))

x_ok <- function(x) {
  !is.null(x) && length(x) == 1 && x > 0
```

```
}

x_ok(NULL)
x_ok(1)
x_ok(1:3)
```

6.1 Mais sobre parâmetros

Como visto em outras linguagens de programação, `R` permite atribuir valores *default* para parâmetros de funções. Devemos apenas lembrar de que o corpo da função opera sobre um ambiente distinto do ambiente externo.

```
a <- 20
b <- 100
uma_funcao <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100

  c(x, y, z)
}
uma_funcao()
```

Em uma chamada de função, `R` permite que a ordem de atribuição dos valores aos parâmetros seja alterada desde que os parâmetros sejam nominalmente identificados. Alguns exemplos desta capacidade já foram mostradas neste documento. Note que os valores são atribuídos aos parâmetros na ordem definida

```
rnorm(5)
rnorm(mean = 1, n = 5, 3)
rnorm(5, sd = 3)

uma_funcao <- function(x, y, z) {
        c(x = x, y = y, z = z)
}
uma_funcao(1, 2, 3)
uma_funcao(y = 2,
uma_funcao(y = 2, z = 3, 1)
uma_funcao(x = 1, y = 2, z = 3)
```

Para auxiliar, a função `rnorm()` gera números "aleatórios" segundo uma distribuição normal (especificada por sua média e desvio padrão).

6.2 Funções anônimas e funcionais

Como o nome diz, são funções sem nome (apenas com corpo e parâmetros). Elas são usadas no que `R` chama de funcionais. Isso porque ela trata funções como objetos, que podem ser armazenados nomeados (variáveis, como conhecemos) ou estruturas de dados (como listas). Além disso, permite a criação de

funções ditas puras (ou com alta coesão): elas produzem as mesmas saídas para as mesmas entradas. Tais características mudam nossa forma de pensar um programa de computador, exigindo a decomposição de problemas mas permitindo manipulação paralela de dados (*trade-offs* na vida de um programador).

De modo geral, um funcional é uma função que tem como parâmetro uma função e retorna um vetor. Há alguns funcionais usados em operações matemáticas específicas, como cálculo de integral (`integrate()`), otimização (`optim()` e busca de uma raiz (`uniroot()`), que podem retornar um objeto diferente de vetores. Neste caso, a ideia é operar sobre uma função de acordo com uma estratégia algorítmica.

```
rm(list = ls())
mtcars
typeof(mtcars)
as.vector(mtcars)
apply(mtcars, 2, mean)
apply(mtcars, 1, mean)
apply(mtcars, 2, summary)

minha_stat <- function(x, a) c(mean(x+a), sd(x+a))
apply(mtcars, 2, minha_stat, 3)
apply(mtcars, 2, function(x, a) c(mean(x+a), sd(x+a)), 3)
funcoes <- c(mean, sd)
apply(mtcars, 2, funcoes[[1]])
apply(mtcars, 2, funcoes[[2]])
```

A função `apply()` é um funcional, pois recebe como parâmetro uma função. Essa função será aplicada sobre cada elemento (linha, coluna ou bloco de um *array* multi-dimensional) de uma estrutura de dados (sempre convertida para um *array*, que no final é um vetor interno de `R`, cuja dimensão depende da função usada no funcional). Perceba também que o primeiro parâmetro da função é atribuído internamente pelo funcional com base nos elementos do *array* a ser processado.

Note que a função pode ser anônima (sem um nome): isso dependerá do contexto em que é usado, do interesse do programador, etc.

O objeto `mtcars` é um *data frame* contido no pacote `datasets@datasets`, que descreve características de alguns automóveis de 1974. Há outros conjuntos de dados disponíveis para uso em `R` para uso de quem está aprendendo estatística ou testando alguma ideia.

```
as.list(mtcars)
lapply(mtcars, mean)
lapply(mtcars, 'mean')

meu_lapply <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
```

```
}
meu_lapply(mtcars, mean)

vapply(mtcars, mean, double(1))
sapply(mtcars, mean)
vapply(mtcars, minha_stat, double(2), 0)
vapply(mtcars, 'minha_stat', double(2), 0)
```

As funções `lapply()`, `vapply()` e `sapply()` são variantes de `apply()`. Sua flexibilidade reside no fato de que operam sobre listas ao invés de *arrays*. Objetos passados que serão processados por elas são adaptados para listas (um vetor sofre coersão para se tornar uma lista, por exemplo). Já o resultado pode ser uma lista (`lapply`), um vetor (`vapply()` e `sapply()`). No caso das duas últimas, `vapply()` é mais eficiente por exigir a definição do tipo básico dos elementos do vetor retornado.

Note que as funções a serem executadas pelos funcionais podem, inclusive, ser referenciadas por *strings*, o que permite termos *scripts* altamente flexíveis.

```
iris
area <- mapply(function(x, y) x*y,
 iris$Sepal.Length, iris$Sepal.Width)
area
```

Para lidar com elementos de múltiplos objetos, podemos usar a função `mapply()`, que aplica uma função elementos de mesmo índice desses objetos.

```
area <- Map(function(x, y) x*y,
 iris$Sepal.Length, iris$Sepal.Width)
Reduce('+', area, 0)
Filter(function(x) x<5, iris$Sepal.Length)
Filter(function(x) x>5, iris$Sepal.Length)
```

Finalmente, os funcionais `Map()` e `Reduce()` são usadas para operações de mapeamento e redução (ou *map/reduce*). O funcional de mapeamento aplica uma função a um conjunto de dados (equivalente `mapply()`), gerando um conjunto de resultados parciais que serão reduzidos a um através da função de redução. A redução a partir de uma função $f()$ pode ser vista, matematicamente, como $f(f(f(f(x[[1]], i), x[[2]]), x[[3]]), x[[4]])$ no qual i é o valor de início da redução.

O importante aqui é que as funções, sendo puras, exigem que a estratégia de solução considere que a redução ignore a ordem de execução (apesar do exemplo usado considerar uma ordem dos elementos de x, isso não deve ser considerado como verdade, pois em operações paralelas, não há garantia dessa ordem, sob pena de perda de performance).

O funcional `Filter()` permite uma filtragem paralela dos elementos de um objeto, sendo que a função deve ser booleana para garantir a seleção de alguns elementos desse objeto.

6.3 Controle de laços

A função `if()` desvia o fluxo de execução do *script* de acordo com uma expressão lógica, algo comum em linguagens de programação.

`R` também possui laços condicionais. O principal laço é definido pela construção `for()`, mas também há laços tradicionais como `while()`, `repeat()`.

No caso de `for()`, um vetor (pode ser lista) é iterado totalmente e seus elementos são atribuídos a um objeto para uso dentro do corpo do laço.

Adicionalmente, as funções `break` e `next` possibilitam alteração do comportamento do laço, auxiliando na programação dos *scripts*. A primeira função faz com que o fluxo de execução saia do laço que a contém, enquanto que a segunda transfere o fluxo de execução para o controle de decisão do laço.

```
soma <- 0
for (t in 1:10) {
  if (t > 5)
    soma <- soma + 3
  else if (t > 2)
    soma <- soma - 1
  else {
    l <- 0
    while( l<10 )
      l <- l + 1
  }

  a <- 1
  repeat {
     a <- a * 2
     if (a < 100)
       break
  }
  soma + a
}
```

Expressões multilinhas devem ser acomodadas entre chaves (`{}`), como ocorre em linguagem de programação C/C++. Essas expressões retornam valor (última operação executada internamente), exceto para os laços `while()` e `repeat()`, que retornam sempre `NULL`.

6.4 Algumas funções úteis

Nesta seção listarei algumas funções úteis que certamente usaremos em nossos *scripts*. As funções `print()` e `cat()` organizam a impressão de resultados na tela, sendo muito usados em *scripts*.

Podemos enxergar a função `print()` como um método geral (chamado de função genérica em `R`) que cada objeto pode (e geralmente o faz) especializar. Quando acionado, `R` checa se o objeto possui uma especialização dessa função. Se não houver, a função `print.default()` é acionada. Para descobrir os métodos especializados de uma função genérica (ou seja, como alguns objetos podem ter

especializado esse método), usamos a função `methods()` (reforçando que uma função também é um objeto).

```
print
print(print)
methods(print)
print.data.frame
print(mtcars)
class(mtcars)
is.object(mtcars)
```

Note que esta abordagem segue a implementação S3 de orientação a objetos em R, perceba a composição do nome da função que efetivamente imprime o conteúdo de um *data frame*. O padrão é observado para outros métodos/funções especializadas de `print`. Cada especialização pode agregar parâmetros específicos, dependendo do contexto.

A função `cat()` concatena diversos objetos (`print()` é restrita a um objeto), incluindo automaticamente separadores (se desejado) e os imprime na tela ou em arquivo. Como todas as funções retornam algum resultado, `cat()` retorna `NULL`. Essa função exige que os objetos

```
matriz <- matrix(sample(c(0, 1), 16, replace = T), nrow = 4)
cat("total de 1's =", sum(matriz))
cat(matriz, sep = ', ')
```

A função `format()` possibilita formatar objetos para uma impressão mais organizada. Já as funções `paste()` e `paste0()`. Elas efetuam concatenação de objetos (de modo diferente de `cat()`) e retornam uma `string` com o resultado dessa concatenação para algum usso posterior.

Em caso de tipos simples, ela simplesmente os concatena, permitindo a inclusão (`paste()`) ou não (`paste0()` de separadores. Para objetos vetores, seus elementos são concatenados individualmente, podendo inclusive haver reciclagem de elementos caso os parâmetros de entrada sejam vetores de diferentes tamanhos.

```
format(letters[1:10], justify = 'left', width = 5)
format(1:10, width = 5)
format(13.7)
format(13.7, nsmall = 3)
format(c(6.0, 13.1), digits = 2)
format(c(6.0, 13.1), digits = 2, nsmall = 1)
format(2^31-1)
format(2^31-1, scientific = TRUE)
format(matriz, width = 3)
paste('bom', 'dia', 'alunos')
paste0('bom', 'dia', 'alunos')
paste(c('bom', 'dia', 'alunos'))
cat(paste(c('bom', 'dia', 'alunos')))
paste(1:4, 4:1, sep = '-')
```

```
paste(1:4, 1:2, sep = '-')
cat(paste(1:4, 1:2, sep = '-'), sep = '+')
```

Capítulo 7

Gráficos

Quanto analisamos dados, indubitavelmente usaremos gráficos. `R` possui funções bastante versáteis para geração de gráficos mas essa versatilidade se traduz em uma certa dificuldade de uso no primeiro contato (isso quando comparados a aplicativos como Matlab).

A função principal é `plot()`. A partir de pares de valores x-y que são traduzidos em coordenadas cartesianas, um gráfico é gerado em uma janela específica (novas chamadas da função `plot()` geram gráficos que sobrepõe o atual enquanto que o aplicativo RStudio mantém os sucessivos gráficos gerados para posterior visualização). Se usarmos um gráfico de linhas, os pares de valores x-y geram linhas conectando coordenadas sucessivas, podendo inclusive termos curvas que "retrocedem" no eixo x.

```
rm(list = ls())
x <- runif(50, 0, 2)
y <- runif(50, 0, 2)
plot(x, y, main='Título principal',
 sub='Subtitulo', xlab='abscissa', ylab='ordenada')
text(0.6, 0.6, 'texto em (0.6, 0.6)')
abline(h = 0.6, v = 0.6)
for (side in 1:4) mtext(-1:4, side=side, at=0.7, line=-1:4)
mtext(paste('lado',1:4), side=1:4, line=-1, font=2)
```

No exemplo apresentado (mostrado na Figura 7.1), vemos algumas funções auxiliares, que geram camadas visuais sobre o gráfico gerado por `plot()`. Note que as dimensões do gráfico são determinadas automaticamente pela função com base no intervalo de valores dos objetos `x` e `y`que armazenam os valores.

A função `text()` coloca um texto em uma coordenada do gráfico enquanto a função `abline()` gera linhas (no exemplo, horizontais e verticais) a partir de informações paramétricas dessas linhas.

Todo gráfico gerado por `R` possui margens em volta da área de representação visual dos dados. A escrita nessas margens é feita indiretamente pela própria função `plot()` - no caso, escrita de títulos e outros acessórios de um gráfico científico - enquanto a função `mtext()` escreve diretamente nessas margens. As margens são especificadas em número de linhas, assim como cada lado redebe um valor numérico para indexá-lo corretamente.

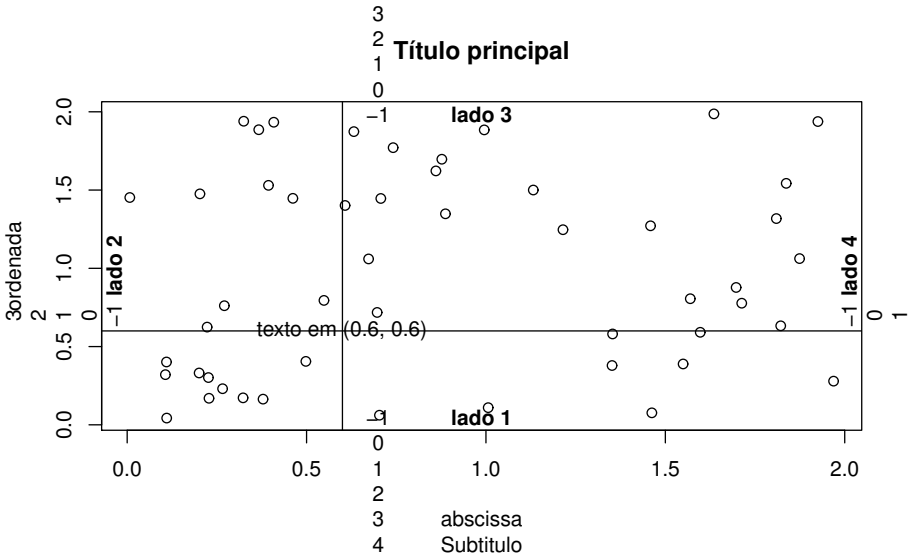

Figura 7.1: Exemplo do uso de `plot()` para geração de gráfico de pontos (*scatter plot*) incluindo indicativo de título, subtítulo, textos para abscissa e ordenada e descritivo de margens externas

Cabe lembrar que esta forma de geração de gráficos não é única em `R`. Um pacote bastante poderoso e flexível para isso é `ggplot2`. Ele opera sob o conceito chamado "gramática de gráficos" na qual os dados são mapeados para primitivas estéticas (*aesthetics*) que permitem sobreposições de tipos de gráficos distintos usando múltiplos gráficos e outras sofisticações.

Podemos usar a função `par()` para alterar parâmetros dos gráficos, como alterar o tamanho das margens, alterar cores de alguns elementos gráficos, alterar corpo/fonte de letras, etc. Ele recupera o valor dos parâmetros gráficos atuais (em uma lista nomeada). Ao invés de exigir que todos os parâmetros de um gráfico sejam informados em cada chamada de `plot()`, `R` mantém uma lista com parâmetros que são sempre usados para a geração dos gráficos e configuração de *layout* de um dispositivo gráfico, ou *device* (um dispositivo gráfico pode ser uma janela gráfica ou um arquivo em formatos como PDF e EPS, por exemplo). Lembre-se que a alteração de parâmetros somente é efetivada para os gráficos gerados posteriormente à alteração.

```
par()
config_antiga <- par(mfcol=c(2, 1))
par(mfg=c(1, 1), mar=c(4, 4, 0, 0)+0.1)
plot(x, y, xlab='abscissa', ylab='ordenada')
par(mfg=c(2, 1), mar=c(5, 4, 4, 2)+0.1)
plot(x, y, xlab='abscissa', ylab='ordenada', pch=2)
par(config_antiga)
```

Alguns parâmetros gráficos podem ser dados para um gráfico específico ou para todos os gráficos que compartilham o mesmo dispositivo gráfico, como é

o caso do parâmetro `pch` do exemplo: pode-se definir o valor para uma curva específica ou para todas as curvas através de `par()`.

Um jeito de resetar a configurações é encerrar o dispositivo (através da função `dev.off()`), que força `R` a abrir um novo dispositivo quando um novo gráfico é gerado.

```
dev.off()
t = seq(1, 3, length.out = 1000)
config_antiga <- par(mar = c(4.1, 4.1, 0.3, 0.3))
plot(t, sin(2*pi*5*t), type = 'l', lwd = 2, col = 'red',
 xlab = 't', ylab = 'amplitude')
lines(t, 1+cos(2*pi*2*t), lty = 'dotted', lwd = 5, col = 'blue')
curve(cos(2*pi*2*x - pi/3)-1, add = T, lwd = 3, col = 'yellow')
par(config_antiga)
```

Note que a definição da estrutura básica dos gráficos (margens, intervalos para abcissas e ordenadas, etc.) é feita pela função `plot()`. As demais curvas usam esse espaço para plotagem.

A função `line()` gera um gráfico de linha a partir dos pares de pontos x e y. Já a função `curve()` é mais propícia para plotagem de expressões matemáticas, podendo inclusive iniciar um gráfico (com `add = F` que é o valor default para esse seu parâmetro).

Capítulo 8

Dados para brincar

R dispõe de um conjunto de dados que é geralmente usado para ensino, teste e validação de algoritmos e procedimentos matemáticos. Um deles - datasets - é carregado automaticamente (vide search()).

Uma forma de listá-los é usando a função data(). Ele fornece a lista dos conjuntos de dados já carregados por R. Combinando seu uso com a função que lista todos os pacotes disponíveis, ou seja, que não precisam ser instalados (.packages(), podemos conhece todos os conjuntos de dados disponíveis.

```
search()
data()
data(package = .packages(all.available = TRUE))
```

Alguns conjuntos de dados interessantes

- airquality: Medições diárias da qualidade de ar de Nova York, de maio a setembro de 1973.
- cars: Velocidade e tempo de frenagem de veículos de 1920.
- iris: Dados de sépala, pétalas e espécie da planta (flor) iris.
- JohnsonJohnson: Lucro por ações da empresa Johnson e Johnson, de 1960 a 1980.
- morley: Dados usados por Michelson e Morley para medir a velocidade da luz.
- mtcars: Características de desenho e performance de 32 veículos em 1973.
- pressure: Medições da pressão de vapor do mercúrio para diferentes temperaturas.
- sunspots: Média mensal do número de pontos solares obervados entre 1749 e 1983.
- swiss: Indicadores socio-econômicos e de fertilidade dos suiços em 1888.
- tree: Medições geométricas de troncos de cerejeiras negras.

- `UCBAdmissions`: Dados agregados dos candidatos a pós-graduação na UC Berkeley em 1973.
- `women`: Altura e peso de mulheres americanas de 30 a 39 anos, de 1975.
- `WWWusage`: Números de usuários conectados à servidores da Internet a cada minuto.

Abaixo seguem algumas operações que podemos realizar com esses dados (usando como exemplo `airquality`). Alguns trabalhos da disciplina usarão esses dados para reforçar nosso aprendizado.

```
airquality
head(airquality)
tail(airquality)
copia <- airquality
summary(airquality)
plot(airquality)
```

Capítulo 9

Conclusão

Esta apostila teve como objetivo apresentar as ferramentas de computação numérica usadas tanto em ambiente acadêmico, como em ambiente de pesquisa e desenvolvimento (seja público ou privado).

Espera-se que o aluno possa, sozinho ou sob orientação, conseguir construir programas nessas ferramentas para solucionar seus problemas matemáticos. Com essa premissa, procurou-se então assumir que o aluno nunca teve contato com tais ferramentas, mas têm conhecimento de programação básica.

Existe muito trabalho para ampliação desse manual, focando sempre na generalização das ferramentas de computação numérica para alunos de graduação. Os autores esperam receber críticas e sugestões (quaisquer que sejam elas).

Índice Remissivo