Universidade de São Paulo (USP)
Faculdade de Economia, Administração, Contabilidade e Atuária (FEA)
Professor Leandro Maciel

EAD830 - IA e ML Aplicados a Finanças

Guia Básico: Programação em R

1. Introdução e Download

O R consiste em um intérprete de tipo *open source* para linguagem de programação, voltado à manipulação, análise e visualização de dados. Atualmente, é mantido por uma comunidade de colaboradores voluntários que contribuem com código fonte da linguagem e com a expansão de funcionalidades por **bibliotecas**, que são uma coleção de subprogramas utilizados no desenvolvimento que resolvem determinados problemas. Um conjunto básico de pacotes (biblioteca) vem embutido na instalação do R, sendo muito outros disponíveis na rede de distribuição do R, o *Comprehensive R Archive Network* (CRAN).

O R, portanto, constitui uma ferramenta eficaz para solução de uma série de problemas (computação estatística e gráfica). Existem outras linguagens de programação, como Java, C++, MatLab, Python, etc, porém, a principal distinção entre elas corresponde a sintaxe, disponibilidade gratuita e composição de bibliotecas para diversas funcionalidades.

Para *download* do R, basta acessar a página oficial do *software*:

`https://www.r-project.org`

Ao baixar a versão, de acordo com o sistema operacional do usuário, a instalação padrão (*default*) inclui uma série de funções. De acordo com suas necessidades, o usuário constrói o banco de funções sob demanda. Ao longo do curso utilizaremos uma série de pacotes, sobretudo que tratam dos problemas de finanças que abordaremos.

A interface do R é muito simples e não apresenta funcionalidades *user-friendly*, portanto, uma alternativa é utilizar um ambiente integrado que facilita a programação, sobretudo para iniciantes, como é caso do **RStudio**, *software* livre de ambiente de desenvolvimento integrado para R. Deve-se notar que, o RStudio constitui de um ambiente de desenvolvimento, em que o R deve estar previamente instalado. De forma simples, o RStudio "roda" o R em uma interface mais amigável, em que as funcionalidades são explícitas (por meio de "botões"), não requerendo codificação, como é o caso, por exemplo, do *software* excel. Assim, o uso do RStudio só se mostra como uma alternativa para os usuários programarem com o R. O RStudio pode ser baixado em:

`https://rstudio.com`

Na condução do curso EAD-737, o RStudio será utilizado para o desenvolvimento das práticas de programação. Aos que não desejam instalar os programas em suas máquinas, uma alternativa consiste em utilize o R no próprio *browser* de internet, como é o caso do **RStudio Cloud**: `https://rstudio.cloud`.

2. Funcionalidades Básicas

Expressões, que descrevem os procedimentos a serem executados pelo R, são digitadas na linha de comando no "Console". No console, o sinal "$>$" indica que o R está pronto para receber e executar um comando. Na ausência desse sinal, o R está executando um comando (em processamento). Caso, na linha de comando, o sinal "$+$" esteja indicado, isso significa que o comando digitado está incompleto e necessita ser finalizado. Se o usuário não deseja finalizar o comando e quer cancelá-lo, pressiona-se "esc" (*escape*) no teclado.

Ao digitar uma expressão e pressionar "enter", o R opera a expressão e mostra o resultado na tela. Os operadores matemáticos básicos são:

- $+ \rightarrow$ soma;
- - $\rightarrow$ subtração;
- $* \rightarrow$ multiplicação;
- $/ \rightarrow$ divisão.

Expressões já executadas podem ser novamente apresentadas na linha de comando de duas formas: i) pressionando a tecla "up" ($\uparrow$) do teclado (quando pressionada sucessivamente os comandos digitados na ordem do mais recente para o mais antigo serão apresentados); ou digitando "history()" na linha de comando e acessando a aba "History" no canto superior direito da tela. Essa ferramenta é muito útil para identificar erros em programas.

Todo valor retornado de uma expressão simples é acompanhado de "[1]", pois todo número no R é interpretado como um vetor, e 1 corresponde ao indexador do elemento do vetor que está apresentado na linha. De forma geral, "[n]" indica que o primeiro número impresso na linha é o n-ésimo elemento do vetor.

A ordem de prioridade das operações é multiplicação e divisão e, depois, soma e subtração. Entre multiplicação e divisão (ou soma e subtração) a ordem se dá de acordo com a ocorrência. Para uma definição específica da ordem de processamento das operações, utiliza-se parênteses. Não se usa "[ ]" ou "{ }", pois estão associados a outras funcionalidades do R.

O R permite associar valores ou expressões à **variáveis** nomeadas pelo usuário. O sinal de associação é o "=" ou "$<$ -". Exemplo:

$> x = 1$
$> x <$- $1 + 4 * 2$

O que significa, na linguagem R, "a variável x recebe (guarda) o valor 1 (ou o resultado da expressão $1 + 4 * 2$)". Quando da criação de uma variável, essa é armazenada no sistema. Pode-se verificar sua criação no "Environment" do sistema (canto superior direito).

O R é *case sensitive*, ou seja: $> x = 1$ é diferente de $> X = 1$.

Com base em variáveis criadas, o R permite escrever expressões com variáveis:

$> y = 4$
$> z <$- $y + 13/4$

Para apagar ou remover uma variável usamos a **função** "rm()". Exemplo:

$> y = 4$
$> rm(y)$
$> rm(list = ls()) \rightarrow$ apaga todas as variáveis do *environment*.

Qualquer objeto em R tem uma classe correspondente, que pode ser combinado em estruturas de dados, veremos as estruturas depois. São essencialmente 5 classes distintas:

- "character" $\rightarrow$ texto, ou uma cadeia de caracteres, chamada de *string*. Ex.: $> x =$ "Palavra"
- "numeric" $\rightarrow$ números reais (decimais - separação decimal é por " . ", ponto). Ex.: $> y = 5.65$
- "integer" $\rightarrow$ números inteiros. Ex.: $> z = 5L$ (usa-se L, maiúsculo, para R reconhecer valor inteiro)
- "logical" $\rightarrow$ expressões lógicas. Ex.: $> w =$ TRUE
- "complex" $\rightarrow$ números complexos. Ex.: $> t = 1 + 4i$

Para saber a classe de uma variável, usamos o comando "class()". Ex.: $>$ class(w).

Importante: Quando nomeamos uma variável, por meio de um símbolo (nome que usuário escolhe), não podemos usar as seguintes expressões: **if, else, repeat, while, function, for, in, next, break, TRUE, FALSE, NULL, Inf, NaN, NA, NA integer, NA real, NA complex, NA character, ..., ..1, ..2, ..3, ..4, ..5, ..6, ..7, ..8 ou ..9**. Essas expressões especiais desempenham função específica no R. Ao longo do curso iremos discutir essas simbologias.

Muitas vezes, precisamos executar uma série de comandos e operações repetidas vezes, de forma que não é prático digitá-los a cada novo uso. Para isso, podemos criar um arquivo que contenha todos os comandos, que chamamos de **script** ou, de forma mais usual, "**código**". Esse *script* conterá todos os comandos que podem ser executados no R. Para criar um *script*, podemos seguir o caminho "Arquivo $>$ Novo $>$ R Script", ou usar o primeiro ícone (botão) à esquerda logo acima do Console.

A vantagem dos *scripts*, como um conjunto de códigos, é que podemos incluir **comentários**, ou seja, mensagens não compiladas ("rodadas") pelo R, que auxiliam o usuário na compreensão dos procedimentos que são realizados. Sempre quando criamos um código é recomendado comentar os objetivos dos procedimentos, tanto para o próprio usuário, quanto para distribuição. Para inserir um comentário, usamos o símbolo "**#**". Tudo à direita do símbolo # não é compilado (o R "ignora"). O *script* é salvo com o nome de preferência e terá extensão ".R".

Para executar ("rodar") cada linha (comando) de um *script*, podemos usar o comando "shift + enter" ou, alternativamente, usar o ícone (botão) "rodar" (seta verde apontando para a direita), localizado na barra de ferramentas de edição dos *scripts*. Os *scripts* são muito úteis para elaborarmos **algoritmos**, como um conjunto de operações de processamento para solução de problemas.

Por fim, um detalhe de visualização importante, antes de avançarmos para a compreensão das próximas funcionalidades, trata-se da cor de fundo da tela do R. Muitos programadores preferem fundo escuro para visualização das operações. Essa é uma questão pessoal e o usuário que define a qual achar mais apropriada. Para alterar a cor de fundo, usamos o caminho: "Tools $>$ Global Options $>$ Appearance $>$ Editor theme $>$ escolher o tema".

3. Vetores, Matrizes, Operações e Indexação

Em muitos problemas, a representação de um conjunto de dados (informações) é usualmente realizada por meio do uso de vetores e matrizes. Para nós, os vetores são um conjunto de informações referenciado em uma única componente (dimensão única). Ex.: $v = [0,1,2,3]$ é um vetor com 4 componentes, na componente 1 temos o valor numérico 0, na 2 o valor 1 e etc. Já as matrizes, a referência dos elementos se dá por duas componentes. Ex.: $V = \begin{bmatrix} 4 & 9 \\ -1 & 0 \end{bmatrix}$ é uma matriz de ordem 2 linhas e 2 colunas (2×2), o elemento referenciado na linha 2 e coluna 1 é 9.

Para criarmos um vetor no R, usamos a função "**c( )**". Toda função no R é composta por seu nome ("c") e seus argumentos entre parênteses - os nomes das funções são únicos e são *case sensitive*. Argumentos são as informações que a função requer para que ela desenvolva (realize) sua funcionalidade. Como a função "c( )" cria um vetor, seus argumentos são os componentes do vetor. Nas funções, os argumentos são sempre separados por vírgula. Portanto a criação do vetor $v = [3, 1.7, -7, 10, 0.9]$ é:

$> v = c(3, 1.7, -7, 10, 0.9)$ # note que o vetor é criado e associado à variável chamada v

As operações básicas $(+, -, *, /)$ podem ser aplicadas para vetores. Temos três casos possíveis:

i) **vetores com mesma dimensão** $\rightarrow$ as operações são realizadas elemento a elemento (correspondentes):

$> v = c(3, 1.7, -7, 10, 0.9)$

$> u = c(1, 0, -88, 0.41, 999)$

$> v + u$

$> v - u$

$> v * u$

$> v/u$ # atente-se ao resultado da divisão por zero!

ii) **vetores com dimensões distintas** $\rightarrow$ as operações são realizadas elemento a elemento (correspondentes) e, depois, a sequência menor é repetida:

$> v = c(1,2,3,4)$

$> u = c(5,6,7)$

$> v + u$ # atente-se a mensagem de aviso que aparece!

$> v * u$

$> v + 5.50$

$> h = c(1, v, 4, 5, u)$ # podemos também combinar vetores

iii) **operações vetoriais** (seguindo regras da álgebra vetorial) $\rightarrow$ os vetores são reconhecidos como vetores coluna. O produto vetorial de dois vetores de mesma dimensão, ou produto interno (soma do produto elemento a elemento), é realizado usando o operador de produto para vetores e matrizes **%*%** (multiplicação tradicional no R, mas entre símbolos de %). Exemplos:

$> v = c(1,2,3,4)$

$> u = c(2,2,2,2)$

$> v * u$ # produto elemento a elemento

$> v \,\%*\%\, u$ # produto interno

Nesse caso, não existem as operações **%+%** ou **%-%**, porque a soma e subtração vetorial/matricial já é elemento a elemento. Além disso, não existe divisão de matrizes ou vetores, portanto, o operador **%/%** tem outro significado no R: **resulta a parte inteira da divisão de dois números**, no caso de vetores elemento a elemento, exemplos:

$> 5/4$ # retorna o valor da divisão

> 5 %/% 4 # retorna o valor inteiro da divisão

$> c(1,2,3,4)/c(2,2,2,2)$ # divisão elemento a elemento

$> c(1,2,3,4)$ %/% $c(2,2,2,2)$ # inteiro da divisão elemento a elemento

Aalternativamente, o operador "**%%**" retorna o resto da divisão entre dois números, exemplos:

> 10 %% 4 # retorna o resto da divisão de 10 por 4

$> c(8,11,13,15)$ %% $c(2,3,4,5)$ # retorna o resto da divisão elemento a elemento

Os elementos de um vetor podem conter variáveis de diferentes classes, ou seja:

$> VetorDiverso = c(3.4, 5L, \text{“}Maria\text{”}, 3-2i)$ # 4 componentes de distintas classes

Sempre que um vetor contém variáveis de diferentes classes, quando há ao menos uma delas do tipo "character", o vetor também o será; se não houver variável do tipo "character", mas há ao menos uma delas do tipo "complex", o vetor também será "complex"; por fim, se houver apenas variáveis de tipo "numeric" e "integer", o vetor será do tipo "numeric". A classe das variáveis é importante porque, em alguns casos, as operações não são realizadas com valores (ou vetores) de classes distintas. Quando operamos variáveis "numeric" e "integer", o resultado é "numeric". Se um dos valores for complexo, a operacionalização gera também valores de tipo "complex". Caso uma variável seja "character" não haverá operação (erro).

Para acessar elementos particulares de um determinado vetor, **indexação**, usamos colchetes "[ ]". Exemplos:

$> v = c(3,4,8,1,0,9,13,6,-1,4.5)$ # criamos um vetor v com 10 componentes

$> v[3]$ # acessa o elemento da componente 3 do vetor

$> v[2:7]$ # acessa os elementos da componente 2 até (":" significa até no R) a componente 7

$> u = v[c(1,5,9)]$ # cria novo vetor u com os elementos nas componentes 1, 5 e 9 do vetor v

$> v[v\%\%3 == 0]$ # acessa elementos divisíveis por 3 no vetor v (cujo resto da divisão é igual a zero)

Observação: o sinal de "=" indica associação, enquanto que o sinal "==" indica a verificação de uma igualdade. O comando, por exemplo, "1 == 2" retorna uma resposta lógica: "TRUE" ou "FALSE".

Para determinar a dimensão de um vetor, usamos a função "**length( )**", ou seja:

$> v = c(0.4,3,-2.6,7)$ # criamos um vetor v com 4 componentes

$> length(v)$ # retorna a dimensão (no. de componentes) do vetor v

$> v[3:length(v)]$ # acessa os elementos da componente 3 até o número de componentes máximo (4)

Note que, no último exemplo, temos uma combinação de funções, ou seja, *length* está "dentro" da função que acessa os elementos. Isso permite que o usuário não tenha a necessidade de criar uma nova variável com o tamanho do vetor v para depois acessá-lo.

Para criarmos matrizes no R, usamos a função **matrix( )**. Essa função tem três argumentos básicos: os dados que comporão a matriz, o número de linhas, e o número de colunas. Por exemplo, para criarmos a matriz $A = \begin{bmatrix} 3 & 6 & 1 & 4 \\ 0 & -1 & 5 & -8 \\ 10 & 4.5 & -0.2 & -2 \end{bmatrix}$, usamos:

$> A = matrix(c(3,0,10,6,-1,4.5,1,5,-0.2,4,-8,-2), nrow = 3, ncol = 4)$ # ou

$> A = matrix(c(3,0,10,6,-1,4.5,1,5,-0.2,4,-8,-2), ncol = 4, nrow = 3)$ # ou

$> A = matrix(c(3,0,10,6,-1,4.5,1,5,-0.2,4,-8,-2),3,4)$ # forma alternativa, nesse caso, ordem dos argumentos deve ser cumprida

Nota: i) quando nomeados, os argumentos não requerem uma ordem, porém, quando não especificados, o R reconhece a ordem da construção da função (essa ordem pode ser acessada usando o **help**); ii) o primeiro argumento da função é um conjunto de valores, por isso estão dentro da função "c( )" - um único argumento com várias componentes; iii) o preenchimento dos valores se dá **por coluna** (atenção!).

Podemos criar matrizes específicas facilmente:

$> A = matrix(1, nrow = 2, ncol = 3)$ # matriz 2×3 com 1 em todas entradas

$> B = matrix(0, nrow = 4, ncol = 6)$ # matriz 4×6 com 0 em todas entradas

$> C = diag(1, nrow = 3, ncol = 3)$ # função que cria matrizes diagonais (ex.: identidade)

$> D = diag(c(1,2,3), nrow = 3, ncol = 3)$ # diagonal com elementos específicos

Para acessar elementos, ou conjuntos de elementos de uma matriz, a indexação é similar aos vetores, porém, matrizes têm duas dimensões (linhas e colunas). Exemplos de indexação:

$> A[1,3]$ # acessa elemento da primeira linha e terceira coluna

$> A[2,]$ # acessa todos elementos da segunda linha

$> A[,4]$ # acessa todos elementos da quarta coluna

$> A[2:3,]$ # acessa todos elementos da segunda até a terceira linha (em todas colunas)

$> A[1:2,3:4]$ # acessa elementos da primeira a segunda linha na terceira até quarta coluna

$> A[,c(2,3)]$ # acessa elementos da segunda e terceira colunas em todas as linhas

Para realizar as operações entre matrizes, elemento a elemento, usamos "$+, -, *, /$", porém, é necessário matrizes de mesma dimensão. Para operações matriciais, usamos os operadores tradicionais entre símbolos "$\%$", mas apenas para multiplicação, uma vez que a adição e subtração de matrizes já são definidas elemento a elemento. Lembre-se que a divisão de matrizes não é definida, e a multiplicação requer a condição que, para $A \times B$ existir, o número de colunas de A deve ser igual ao número de linhas de B. Exemplos:

$> A = matrix(c(1,2,3,4,5,6,7,8,9,10,11,12), nrow = 4, ncol = 3)$

$> B = matrix(c(-1,3,5,10,0.6,7,9,11,14,-5,3,1), nrow = 4, ncol = 3)$

$> C = matrix(c(-4,-5,10,11,0.3,70), nrow = 3, ncol = 2)$

$> A + B$

$> A - C$ # atente-se para a mensagem de erro

$> A * B$ # produto elemento a elemento

$> A/B$ # divisão elemento a elemento

$> A\%*\%C$ # produto **matricial**

$> A\%*\%B$ # atente-se para a mensagem de erro

Para determinar a dimensão de uma matriz, usamos a função "**dim( )**" (diferente de quando tratamos de vetores), ou seja:

$> A = matrix(c(1,2,3,4,5,6,7,8,9,10,11,12), nrow = 4, ncol = 3)$

$> dim(A)$ # são duas dimensões - linha e coluna

Por fim, o R disponibiliza algumas funções muito úteis para matrizes:

$> A = matrix(c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16), nrow = 4, ncol = 4)$

$> B = t(A)$ # guarda na variável B a matriz transposta de A

$> det(A)$ # calcula o determinante da matriz A

$> C = matrix(c(-3,2,5,7), nrow = 2, ncol = 2)$

$> det(C)$ # calcula o determinante da matriz C

$> solve(C)$ # calcula a matriz inversa de C

$> solve(A)$ # pense no motivo da mensagem de erro nesse caso!

A manipulação de vetores e matrizes será muito importante, pois grande parte das práticas de *data mining* operacionaliza informações no formato de matrizes e vetores. Obviamente que tais objetos podem incluir variáveis de classe de caracteres, porém, como nosso objetivo é análise de dados, o foco do desenvolvimento será com exemplos numéricos, uma vez que as operações não são realizadas quando objetos são de classes distintas (sobretudo quando usamos variáveis de classe "character").

4. Objetos em R

Como o R é uma linguagem de programação orientada a objetos, é fundamental a utilização adequada dos seus 5 tipos de objetos: vetores, fatores, matrizes, dataframes e listas. Os vetores e matrizes já foram desenvolvidos.

Fatores

Os fatores são uma sequência de valores, definido por níveis. Um fator é muitas vezes utilizado para expressar uma variável categórica presente em uma base de dados, como sexo por exemplo. Nestes casos, os níveis representam as categorias da variável (masculino e feminino). Suponha que queremos criar uma variável sexo que armazena a informação do sexo de 6 indivíduos. Inicialmente, criamos um vetor:

$>$ *sexo* = c("Feminino", "Feminino", "Masculino","Feminino","Masculino","Masculino")

Em seguida, utilizamos a função "as.factor()" para definir o vetor como um fator:

$>$ *fatorsexo* = as.factor(sexo)

$>$ *fatorsexo*[4] # note que a indexação é a mesma que em vetores

Observe que ao chamar os objetos sexo e o fatorsexo, o R apresenta o conteúdo de maneiras diferentes. Enquanto o vetor sexo contém elementos formados por uma sequência de caracteres, apresentados entre aspas, o fator fatorsexo é formado por níveis de um fator - não apresentados entre aspas. Tal tipo de objeto é fundamental em problemas de classificação que utilizam variáveis categóricas para tratar determinado problema. Por exemplo: desejamos criar um classificador para identificar a incidência de determinada doença. Nesse sentido, podemos usar uma variável do tipo fator como a do exemplo, como variável explicativa da análise.

Dataframe

A forma como os dados estão estruturados pode ser determinante para realizar uma análise. O objeto do tipo dataframe pode ser a melhor forma de armazenar os dados, já que permite trabalhar com vetores alfanuméricos e fatores. O dataframe compartilha de muitas propriedades de matrizes e listas. Em geral, é utilizado com cada coluna representando uma variável e cada linha uma observação do conjunto de dados.

Para criar um dataframe, podemos utilizar a função "data.frame()":

$>$ *idade* = $c(23,55,32,40,29,18)$ # criamos um vetor com idades
$>$ *base* = data.frame(idade,fatorsexo) # criamos um dataframe com o vetor e a variável fator

A forma como se organiza o dataframe pode determinar a viabilidade da utilização de determinadas funções e pacotes do R, que podem exigir um formato pré-especificado para os dados a serem analisados. A indexação em objetos do tipo dataframe muda, exemplo:

$>$ *base*\$*idade* # usamos \$ para acessar todos os valores de uma categoria

$>$ *base*\$*sexo*

$>$ *base*\$*idade*[3] # acessa o elemento 3 da categoria idade

Listas

Objetos da classe lista são estruturas capazes de conter objetos de diversos tipos de classes. Para criar uma lista utilizando a função "list()". Como exemplo, vamos criar uma lista contendo um vetor, um fator, uma matriz e um dataframe criados anteriormente:

$>$ *lista* = list(idade,sexo,fatorsexo,base)

É fundamental notar, no environment do R, a diferenciação dos objetos de diferentes tipos. Mais uma vez, quando o objeto é do tipo lista, a indexação também se modifica:

$>$ *lista*[[1]] # acessa todos os elementos do item 1 da lista

$>$ *lista*[[3]] # acessa todos os elementos do item 3 da lista

$>$ *lista*[[2]][3] # acessa o elemento na posição 3 do item 2 da lista

$>$ *lista*[[4]]$[3,1]$ # acessa o elemento na posição 3 da coluna 1 do item 4 (dataframe) da lista

5. Funções Matemáticas e Criação de Funções

Agora vamos desenvolver algumas funções básicas no R. Retomando, uma função consiste na automação de determinadas operações (calcula a média de uma amostra, o logaritmo de um número, etc). Todas as funções têm o seguinte formato: "NomeDaFunção(Argumento1,Argumento2,...,ArgumentoN)". O nome é sempre próprio, específico, para cada função e, mais uma vez, case sensitive. Os argumentos são sempre separados por vírgula, e cada função tem um número específico de argumentos. Cada função nomeia os argumentos de uma forma específica, assim, os argumentos, quando nomeados, podem ser apresentados em qualquer ordem, porém, quando não nomeados, a sequência faz diferença. Lembre-se da função "matrix" que já desenvolvemos, que tem 3 principais argumentos:

$> A = matrix(data = c(-3,2,5,7,1,2), nrow = 2, ncol = 3)$ # argumentos nomeados

$> A = matrix(nrow = 2, data = c(-3,2,5,7,1,2), ncol = 3)$ # argumentos nomeados - ordem indiferente

$> A = matrix(c(-3,2,5,7,1,2), 2, 3)$ # argumentos não nomeados - sequência da função

$> A = matrix(2, c(-3,2,5,7,1,2), 3)$ # argumentos não nomeados - sequência incorreta (ver erro)

Para saber todos os detalhes de uma função, usamos **help("matrix")**. O help do R é muito útil, além de fornecer toda a descrição da função, utiliza de exemplos para facilitar a compreensão.

As funções mais básicas são as matemáticas. A Tabela 1 apresenta as principais funções para operação de valores. Deve-se notar que cada função requer um número de argumentos distintos e, em alguns casos, operam tanto valores numéricos, vetores ou matrizes.

Tab. 1: Funções Matemáticas no R.

Função	**Descrição**
abs(x)	calcula valor absoluto de x
log(x,b)	logaritmo de x na base b
log(x)	logaritmo natural de x
exp(x)	e (exponencial) elevado a x
sqrt(x)	raiz quadrada de x
sin(x)	seno de x
cos(x)	cosseno de x
tan(x)	tangente de x
round(x,digits = n)	arrendondamento de x com n casas decimais
length(v)	dimensão de um vetor v
max(v)	máximo valor de um vetor v
min(v)	mínimo valor de um vetor v
mean(v)	média aritmética de um vetor v
range(v)	menor e maior valor de um vetor v
seq(from = 1, to = 10, by = 0.2)	cria sequência de 1 a 10 variando 0.2
sum(v)	soma os valores de um vetor v
rep(v, n)	repete elementos de um vetor v n vezes
prod(v)	retorna o produto dos elementos do vetor v

Uma função muito utilizada em programação é a que "imprime" determinada informação/valor/objeto no Console do R, chamada de função "print()". Exemplos:

> print("Hello World") # imprime na tela a mensagem Hello World

> print("Hello World",quote = FALSE) # imprime na tela sem uso das aspas

> $idade = c(22,15,45)$

> print(idade) # imprime na tela os componentes de uma variável chamada idade

> print(idade[1]) # imprime na tela o elemento na componente 1 da variável idade

Se quisermos imprimir na tela uma mensagem que combina texto e números (de objetos), usamos a função "paste()":

> paste("As idades são",idade[1],idade[2],"e", idade[3])

Outra função muito utilizada por programadores é a que recebe valores indicados pelo usuário, chamada no R de **"readline()"**. Essa função, em sua formulação, lê o que foi digitado pelo usuário como texto. Caso queira que o valor digitado seja reconhecido como número, devemos combinar com a função **"as.numeric()"**. Exemplos:

> idade = readline("Digite a sua idade: ") # o valor digitado será reconhecido como character

> idade = as.numeric(readline("Digite a sua idade: ")) # o valor digitado será reconhecido como número

Também podemos criar no R nossas próprias funções, que recebe um determinado número de argumentos e realiza uma certa operação. Para isso, usamos a função **"function()"**. A estrutura básica para criar uma função se dá da seguinte forma:

> NomeDaFunção = function(Argumento1,...,ArgumentoN) { Procedimentos da Função }

O programador determina o nome da função, que não pode ser igual aos nomes das funções já existentes no R, assim como nomeia os argumentos, o número de argumentos, e os procedimentos da função. Exemplo: vamos construir uma função matemática que recebe dois argumentos, x e y, e retorna o valor da função, $f(x,y)$, de acordo com a seguinte relação:

$$f(x,y) = \frac{x^2 - 4y + 2}{y^2 + 1} - 10$$

Vamos chamar essa função de "MinhaFuncao", então, podemos gerá-la da seguinte forma:

> MinhaFuncao = function(x,y){ ((x^2-4*y+2)/(y^2+1))-10 }

Atente-se ao uso dos parênteses. O símbolo " ^ " (circunflexo) é o operador de exponenciação no R. No environment é criada a função correspondente. Seu uso se dá como qualquer função no R:

> MinhaFuncao(x=1,y=2) # identificamos os argumentos

> MinhaFuncao(2,4) # não identificamos os argumentos (reconhece na ordem)

> valor = MinhaFuncao(4,10) # guardamos a saída da função na variável "valor"

> valor2 = MinhaFuncao(4) # verifique a mensagem de erro!

> valores = MinhaFuncao(c(1,2,3),c(2,4,1)) # calcula o valor da função para os pares (1,2) (2,4) e (3,1)

Vamos agora criar uma função que realiza mais de uma operação e, portanto, deve retornar mais de um valor. Suponha uma função que receba 4 argumentos numéricos e deva retornar a soma desses valores, o valor máximo, o valor mínimo e o produto dos valores. Podemos criá-la com o seguinte comando:

```
> MinhaFuncao2 = function(valor1,valor2,valor3,valor4){
  sum(valor1,valor2,valor3,valor4); max(valor1,valor2,valor3,valor4);
  min(valor1,valor2,valor3,valor4); prod(valor1,valor2,valor3,valor4) }
```

Note que usamos o sinal ";" como separador de operações. Quando as operações são informadas na mesma linha usamos esse separador, de outra forma, quando escrevemos por um script basta pular linha para separar as operações. Agora, vamos testar a função para um determinado conjunto de valores:

> resultado = MinhaFuncao2(1,2,3,4)

O resultado será um único valor e igual a 24, que se trata do produto dos valores. Ou seja, sempre as funções retornam o valor da última operação. Para definirmos qual o valor a ser retornado, usamos em conjunto a função **"return"**, ou seja:

```
> MinhaFuncao2 = function(valor1,valor2,valor3,valor4){
  a = sum(valor1,valor2,valor3,valor4); b = max(valor1,valor2,valor3,valor4);
  c = min(valor1,valor2,valor3,valor4); d = prod(valor1,valor2,valor3,valor4); return(c(a,b,c,d)) }
```

> resultado = MinhaFuncao2(1,2,3,4)

Dentro da função return, usamos a função "c()" para indicar que o resultado é um conjunto (vetor) de valores. Além disso, criamos as variáveis a, b, c e d com os resultados para inserir na função return. Note que variáveis criadas dentro de uma função não são salvas no environment.

As funções podem conter centenas de comandos, de acordo com a complexidade do problema tratado pelo programador. No conjunto de comandos das funções, podemos incluir inúmeros comandos já disponíveis no R. Exemplo: vamos criar uma função que o usuário fornece, como argumentos, seu peso e altura, e seja retornado seu IMC (Índice de Massa Corporal). Lembremos que o IMC é o quociente entre o peso e o quadrado da altura. Podemos programar da seguinte forma:

> CalculadoraIMC = function(peso,altura) { peso/(altura^2) }

> CalculadoraIMC(peso = 65, altura = 1.70) # identificamos os argumentos

> CalculadoraIMC(78,1.70) # não identificamos os argumentos

Podemos sofisticar tal função, por exemplo, fazendo com que o usuário digite na tela de comando seu peso e sua altura, e a calculadora imprima uma mensagem com o IMC. Exemplo:

```
> CalculadoraIMC = function(){ # note que não tem argumento - usuário digita na tela
  peso = as.numeric(readline("Digite o seu peso em Kg: "));
  altura = as.numeric((readline("Digite a sua altura em metros: ")));
  IMC = peso/(altura^2);
  paste("O seu IMC é",round(IMC,2)) } # arrendonda para 2 casas decimais
```

> CalculadoraIMC() # chama a função

6. Expressões Lógicas e Condicionais

Iremos agora criar conjuntos de regras, ou estruturas de controle, para desenvolver procedimentos mais sofisticados no R, ou seja, algoritmos/programas capazes de operacionalizar determinados objetivos ou automatizar um conjunto de operações. Para tanto, precisamos antes desenvolver os operadores lógicos no R. A tabela abaixo apresenta a sintaxe dos operadores no R e sua descrição.

Sintaxe	Operador
==	Igualdade
!=	Não Igualdade/Diferente
<	Menor
>	Maior
<=	Menor ou Igual
>=	Maior ou Igual
&&	E
||	Ou

Com bases nos operadores, podemos verificar operações lógicas, que resultam em um resultado dicotômico, Verdadeiro **"TRUE"** ou Falso **"FALSE"**. TRUE e FALSE são expressões especiais no R (têm significado próprio) e não podem ser utilizadas como símbolos (nomear variáveis criadas pelo usuário). Note que "=" significa associação ($x = 2$, associe o valor 2 na variável x), enquanto que "==" é uma expressão lógica que verifica a igualdade ($3 == 1$, 3 é igual a 1, que retorna verdadeiro ou falso). Exemplos de expressões lógicas:

```
> 1 < 10 # 1 é menor que 10

> 2 == 2 # 2 é igual a 2

> 4 > -5 # 4 é maior que -5

> 2  >= 0|| 3 >= 0 # 2 ou 3 é maior ou igual a 0

> 7 < 2 || 9 < 2 # 7 ou 9 são menores que 2

> 10 != 20 && 20 != 20 && 30 != 20 # 10 e 20 e 30 são diferentes de 20

> 10 != 20 || 20 != 20 || 30 != 20 # 10 ou 20 ou 30 são diferentes de 20

> 15%%4 != 0 # o resto da divisão de 15 por 4 é diferente de zero

> 15%/%4 == 2 # o inteiro da divisão de 15 por 4 é igual a 2
```

Conhecidos os operadores lógicos, para desenvolvermos a programação orientada, o primeiro operador importante se trata da implementação da **declaração condicional** no R, ou seja, indicar a execução de determinados procedimentos quando a condição é verdadeira, e outros quando a condição é falsa. Para isso, usamos a função do tipo controle **"if"** ou **"if...else"**. A declaração condicional no R tem a seguinte forma de utilização:

if (*condição*) {
"*Conjunto de Comandos Se Verdadeiro*"
}

Combinando if e else:

if (*condição*) {
"*Conjunto de Comandos Se Verdadeiro*"
} **else** {
"*Conjunto de Comandos Se Falso*"
}

Exemplo: vamos criar um programa em que o usuário fornece um número e é calculada a raiz quadrada desse número, sendo o resultado arredondado com duas casas decimais após a vírgula. Primeiro, vamos considerar que o programa apenas resulte a raiz quando se trata de um número não negativo e, nesse caso, imprima na tela a mensagem "A raiz quadrada de x é y". Assim, podemos fazer:

```
numero = as.numeric(readline("Digite um número: "))
if (numero >= 0){
raiz = sqrt(numero)
paste("A raiz de",numero, "é", round(raiz,2))
}
```

Teste essa função para valores positivos e negativos. Agora, como sabemos, não existe a raiz quadrada de números negativos, portanto, queremos que o programa, quando fornecido um número não negativo, imprima na tela a mensagem "A raiz quadrada de x é y", mas, se é fornecido um número negativo, a seguinte mensagem seja impressa "O número x não possui raiz quadrada". Ou seja:

```
numero = as.numeric(readline("Digite um número: "))
if (numero >= 0){
raiz = sqrt(numero)
paste("A raiz de",numero, "é", round(raiz,2))
} else {
paste("O número",numero,"não possui raiz quadrada")
}
```

Podemos ter problemas que incluam procedimentos distintos para mais de duas condições, ou seja, devemos combinar mais de um comando "if...else", que chamamos de "*nested if...else*". Por exemplo, queremos criar a seguinte função matemática no R:

$$f(x) = \begin{cases} x^2, & se\ x \leq 0, \\ sen(x), & se\ 0 < x \leq 2, \\ 4x+1, & se\ x > 2. \end{cases}$$

A função recebe um valor x e deve fornecer um resultado distinto nos três casos. Portanto, podemos, por exemplo, construir tal função da seguinte forma:

```
funcao = function(x){
  if (x <= 0){
  fx = x^2
  } else if (x>0 && x<=2){
  fx = sin(x)
  } else {
  fx = 4*x + 1
  }
  return(fx)
}
```

Devemos sempre nos atentar ao uso das chaves "{ }", pois elas farão a distinção entre os procedimentos que fazem parte de determinado comando. Para evitar problemas, o uso da identação é importante, ou seja, utilizar-se de espaços para indicar o fechamento ou continuidade de outros procedimentos. Quando escrevemos um script no R, essa identação já é sugerida de forma automática.

O comando "if...else" é aplicado para condições que envolvem apenas um valor. Porém, podemos estar interessados em verificar uma condição para um conjunto de elementos, por exemplo, para os elementos de um vetor. Para isso, usamos a função **"ifelse()"**. Tal função tem a seguinte estrutura:

ifelse("teste","sim","não")

A função tem três argumentos. O primeiro, "teste", consiste na descrição da condição condicional a ser verificada para todos os elementos de um vetor. Os argumentos "sim" e "não" indicam a operação a ser realizada caso a condição seja satisfeita ("sim") ou não ("não"). Por exemplo, suponha que temos um vetor com vários números e estamos interessados em saber quais são divisíveis por 5 (cujo resto da divisão por 5 seja igual a zero). Podemos utilizar a função "ifelse()" da seguinte forma:

```
> x = c(10,12,15,16,22,25,30,34,37,40,45,50,56) # crio um vetor com vários números
> ifelse(x%%5 == 0,"É divisível","Não é divisível")
```

Voltemos agora ao nosso exemplo em que criamos uma função para calcular o IMC de um indivíduo (com peso e altura fornecidos pelo usuário na tela de comando). A OMS (Organização Mundial da Saúde), a partir dos valores de IMC, criou uma tabela para classificar a situação dos indivíduos em termos do seu peso. A tabela abaixo apresenta a categorização do IMC.

IMC	**Classificação**
Abaixo de 18,5	Abaixo do peso ideal
Entre 18,5 e 24,9	Peso ideal
Entre 25 e 29,9	Sobrepeso
Entre 30,0 e 34,9	Obesidade grau 1
Entre 35,0 e 39,9	Obesidade grau 2
40,0 e Acima	Obesidade grau 3

Podemos agora sofisticar nossa função de cálculo do IMC, incluindo a impressão, além do valor do IMC, a classificação do indivíduo de acordo com a categorização da OMS. Uma alternativa para esse problema pode ser obtida de acordo com o seguinte código:

```
CalculadoraIMC = function(){
  peso = as.numeric(readline("Digite seu peso em Kg: "))
  altura = as.numeric(readline("Digite sua altura em metros: "))
  IMC = round(peso/(altura^2),1)
  if(IMC < 18.5){
    paste("O seu IMC é",IMC,"e a classificação é abaixo do peso ideal")
  } else if(IMC>=18.5 && IMC <= 24.9){
    paste("O seu IMC é",IMC,"e a classificação é peso ideal")
  } else if(IMC >= 25 && IMC <= 29.9){
    paste("O seu IMC é",IMC,"e a classificação é sobrepeso")
  } else if(IMC >= 30 && IMC <=34.9){
    paste("O seu IMC é",IMC,"e a classificação é obesidade grau 1")
  } else if(IMC >= 35 && IMC <= 39.9){
    paste("O seu IMC é",IMC,"e a classificação é obesidade grau 2")
  } else {
    paste("O seu IMC é",IMC,"e a classificação é obesidade grau 3")
    }
}
```

Atenção: para esse caso, usamos "&&" (operador lógico "E") ao invés de "||" (operador lógico "Ou"), porque as classes do IMC correspondem ao IMC estar entre um valor (limite inferior) **e** outro valor (limite superior). O uso do operador "Ou" nesse caso iria indicar que basta o IMC estar, ao menos, acima do limite inferior, **ou**, abaixo do limite superior. Assim, se uma só dessas condições fosse verdadeira, a expressão lógica já resultaria em TRUE quando não se deve.

Por fim, em muitos programas, é necessário avaliar a adequação dos dados ao objetivo proposto. Ou seja, nesse nosso exemplo da calculadora do IMC, podemos, quando o usuário inserir um valor negativo, imprimir uma mensagem de erro (não existe peso e altura negativos). Para isso, usamos a função "stop()" que encerra os processos e exibe uma mensagem de erro. Por exemplo, no código acima, logo abaixo das linhas que tomam os valores de peso e altura, podemos inserir os seguintes comandos:

```
if(peso <= 0 || altura <= 0) {
  stop("Não existe peso/altura negativo ou igual a zero")
}
```

Note que agora usamos o operador "Ou", ou seja, basta um dos valores não cumprirem a condição para que a mensagem de erro apareça. Além disso, quando ocorre um erro, para que o R não mova para o modo de "conserto" do erro, que chamamos de *debugging mode*, é preciso ir em "Debug - On Error" e selecionar "Message Only".

7. Loops no R

Em automatização de processos, precisamos realizar uma série de procedimentos repetidamente. Para realizarmos isso em linguagem de programação no R podemos usar as funções **repeat**, **while** ou **for**. Essencialmente, as três funções realizam a mesma funcionalidade, e a escolha irá depender do tipo de automação objetivada, assim como da organização da lógica proposta pelo programador. Vamos considerar um exemplo distinto utilizando cada uma das funções.

A função **repeat** repete um conjunto de procedimentos computacionais a ser definido pelo programador. A sua utilização tem a seguinte estrutura:

```
repeat {
  "Conjunto de Comandos a Serem Repetidos"
}
```

Porém, essa função precisa de um comando adicional: "**break**", que encerra as repetições. Caso esse comando não seja inserido, o R irá repetir os procedimentos infinitamente. Portanto, é fundamental que o programador defina o conjunto de operações mas insira uma condição (usando **if...else**, por exemplo) para o caso em que as repetições devem ser encerradas. Exemplo:

```
repeat {
  "Conjunto de Comandos a Serem Repetidos"
  if("Condição de Parada for Satisfeita") { break }
}
```

Exemplo 1: Suponha que a população de um país A seja da ordem de 80.000 habitantes, com uma taxa anual de crescimento de 3%, e que a população do país B seja de 200.000 habitantes com uma taxa de crescimento anual de 1,5%. Faça um programa que calcule e imprima na tela o número de anos necessários para que a população do país A ultrapasse ou iguale a população do país B, mantidas as taxas de crescimento constantes.

Note que, nesse exemplo, temos que realizar o crescimento das populações nos países A e B e verificar quando a condição é satisfeita. Não sabemos "quantas vezes" as populações devem crescer, ou "quantos anos" devem passar, para atingir a restrição. Portanto, queremos **repetir** o crescimento até um determinado limite e, para isso, podemos usar a função **repeat**. Quando a condição for satisfeita, as repetições devem ser encerradas (populações devem parar de crescer) e, para isso, usamos o comando **break**. Um exemplo de código que resolve esse problema é:

```
PopA = 80000 # população de A
PopB = 200000 # população de B
crescA = 0.03 # crescimento de A
crescB = 0.015 # crescimento de B
n = 0 # inicializamos a variável que guarda o no. de anos

repeat{
  if(PopA >= PopB){
    break
  } else {
    n = n + 1 # passa um ano
    PopA = PopA*(1+crescA) # A cresce
    PopB = PopB*(1+crescB) # B cresce
  }
}

paste("O número de anos é",n)
```

A função **while**, similar a *repeat*, também repete um conjunto de procedimentos computacionais a ser definido pelo programador **enquanto** uma determinada condição não for satisfeita. A sua utilização tem a seguinte estrutura:

while ("*Condição*") {
  "*Conjunto de Comandos a Serem Repetidos*"
}

Nesse caso, não precisamos mais do comando "**break**", pois os procedimentos só serão repetidos enquanto a condição imposta não for satisfeita. Nesse sentido, podemos também solucionar o problema do Exemplo 1 com o comando **while**, ou seja, uma alternativa seria:

```
PopA = 80000 # população de A
PopB = 200000 # população de B
crescA = 0.03 # crescimento de A
crescB = 0.015 # crescimento de B
n = 0 # inicializamos a variável que guarda o no. de anos

while(PopA < PopB){
  n = n + 1 # passa um ano
  PopA = PopA*(1+crescA) # A cresce
  PopB = PopB*(1+crescB) # B cresce
}
paste("O número de anos é",n)
```

Note que a condição muda, ou seja, **enquanto** a população de A for **menor** que a população de B, ambas populações devem crescer. Agora, vamos sofisticar um pouco nossos problemas.

Exemplo 2: Sabemos que, para ser o vencedor da mega sena, é necessário acertar um conjunto de 6 números, não repetidos. Os números variam de 1 a 60. Supondo que o sorteio da mega sena seja aleatório, escreva um programa que, dado um jogo (conjunto de 6 números), determina quantos sorteios são realizados para que esse jogo seja premiado, ou, em outras palavras, em quantas tentativas (sorteios) esse jogo seria finalmente o vencedor.

Para desenvolvermos esse programa, precisamos conhecer duas novas funções no R que serão muito úteis para o problema. A primeira delas é a função "**sample.int()**", que gera aleatoriamente números inteiros dentro de um conjunto de valores. Essa função possui essencialmente três argumentos e tem a forma geral como:

sample.int(x, size $= n$, replace $=$ FALSE)

O argumento x pode ser, por exemplo, um número inteiro positivo. Se assim o for, por exemplo, $x = 60$, como é o caso dos números da mega sena, isso indica que a função irá selecionar números inteiros, de forma aleatória no conjunto de $\{1,2,\ldots,60\}$. Já size $= n$ indica que, nesse conjunto de números, quantos devem ser sorteados. Como o jogo da mega sena tem 6 números, podemos usar size $= 6$. Por fim, o argumento replace indica se podem ser selecionados iguais valores na amostra (se pode haver repetição). Como na mega sena os números não se repetem, mantemos esse argumento igual a FALSE. Então, para sortear, aleatoriamente, 6 números da mega sena, podemos escrever:

sample.int(60, size $= 6$, replace $=$ FALSE)

Ou seja, entre os números inteiros de 1 a 60, sorteie 6 números, sem repetição. Rode essa função diversas vezes e verifique que a cada caso um conjunto diferente é selecionado. Para facilitar a comparação do "nosso jogo" com os números sorteados, deixamos as sequências em ordem crescente de valores e, para isso, usamos a função "**sort()**". Se x é um vetor com n valores, "sort(x)" ordena os valores de x do menor para o maior. Por fim, para comparar os números sorteados com o nosso jogo, usamos a função "**all()**". Ou seja, "all(x == y)" retorna TRUE ou FALSE ao se comparar a igualdade elemento a elemento dos vetores x e y.

Dessa forma, para esse problema, podemos usar o seguinte programa:

```
# Sorteio mega sena:

meujogo = sort(c(10,34,44,21,44,58)) # meu jogo em ordem crescente
nS = 1 # inicializa o no. de sorteios necessários para ser vencedor
sorteio = sort(sample.int(60, size = 6,replace = FALSE))

while(all(sorteio == meujogo) == FALSE){
  nS = nS + 1 # faz-se mais um sorteio, que será:
  sorteio = sort(sample.int(60, size = 6,replace = FALSE))
}
```

Esse código pode levar muito tempo para executar (note o sinal "stop" no console do R). Para simplificar, e termos a ideia de quantos sorteios são necessários para termos nosso jogo sorteado, podemos inserir um critério de parada, por exemplo: faça no máximo 5.000 sorteios. Assim, o código fica:

```
# Sorteio mega sena:

meujogo = sort(c(10,34,44,21,44,58)) # meu jogo em ordem crescente
nS = 1 # inicializa o no. de sorteios necessários para ser vencedor
sorteio = sort(sample.int(60, size = 6,replace = FALSE))

while(all(sorteio == meujogo) == FALSE){
  nS = nS + 1 # faz-se mais um sorteio, que será:
  sorteio = sort(sample.int(60, size = 6,replace = FALSE))
  if(nS > 5000){break}
}
```

Por fim, a função **for** também repete um conjunto de procedimentos computacionais a ser definido pelo programador, porém, diferente das funções repeat e while, o número de vezes em que essa repetição deve ser realizada tem que ser previamente conhecido. A sua utilização tem a seguinte estrutura:

for ("*Indexador*" **in** "*Lista de Valores*") {
"*Conjunto de Comandos a Serem Repetidos*"
}

Nesse caso, mais uma vez, não precisamos do comando "**break**", pois os procedimentos só serão repetidos em um total de vezes que é conhecido pelo usuário. O Exemplo 1 ou 2 não podem ser solucionados com o comando **for**, pois não sabemos quantas vezes os procedimentos deverão ser executados.

Exemplo 3: Uma garagem de estacionamento cobra $ 2,00 de taxa mínima para estacionar até três horas. A garagem cobra um adicional de $ 0,50 por hora ou fração, caso sejam excedidas as três horas. A taxa máxima para qualquer período determinado de 24 horas é $ 10,00. Admita que nenhum carro fica estacionado mais de 24 horas. Escreva um programa que, ao receber um vetor cujas componentes são o tempo de permanência de cada cliente, calcula as taxas de estacionamento para esses clientes, assim como o total arrecadado pelo estacionamento. Você deve fornecer as horas que cada cliente ficou estacionado (fornecer o vetor com o tempo, em horas, de cada cliente).

A resolução desse problema pode ser obtida com o seguinte código:

```
# Criamos um vetor com o tempo de estacionamento de cada cliente em um dia:
x = c(1.5,4.8,6.9,12,15,0.5,23,14,10.8,9.7,6.5,5,4.3)
# Verificamos quantos clientes tivemos naquele dia:
nClientes = length(x)
# Criamos um vetor para guardar os valores pagos por cada cliente:
Gastos = rep(0,nClientes)

# Calcular o gasto de cada cliente:
for(i in 1:nClientes){
  if(x[i] <= 3){
    Gastos[i] = 2
  } else {
    Gastos[i] = 2 + 0.5*(round((x[i] - 3),0))
  }
  if(Gastos[i] > 10){
    Gastos[i] = 10
  }
}

sum(Gastos) # Total arrecadado no dia
```

Nesse exemplo, o número de clientes de um determinado dia define o número de vezes em que o custo respectivo deverá ser calculado, sendo nesse caso mais apropriado o uso da função for, ao invés de while ou repeat.

8. Importação, exportação e visualização de dados

Agora iremos trabalhar com bases de dados no R, para isso, precisamos aprender a construir uma base, importar dados externos e exportar dados para planilhas, por exemplo. Em sequência, trataremos da visualização dos dados, ou seja, construção de gráficos.

A construção de bases de dados no próprio R é realizada diretamente no console, ou usando um *script*. Porém, essa forma é menos usual, uma vez que a grande maioria de dados já é disponibilizada em arquivos com extensão txt, csv e xlsx. Para construir uma tabela de dados usamos a função data.frame. Exemplo:

```
profissao = c("mecânico","vendedor","motorista","advogada","engenheira")
salarios = c(2460.87,1452.10,2356.29,4563.11,10928.00)
idade = c(41,22,38,50,46)
Colaboradores = data.frame(profissao,salarios,idade)
```

Para substituir qualquer valor dentro de uma tabela, basta fazer a mudança usando o operador de associação ("="), indicando o elemento por meio de indexação (note que um data frame é uma matriz, portanto possui duas dimensões, linha e coluna). Exemplo:

> Colaboradores[2,3] = 36

Se quisermos incluir uma coluna em um data frame, usamos a função **cbind()**, ou seja:

> tempo = c(3,2,6,8,7)

> Colaboradores = cbind(Colaboradores,tempo)

Para incluir uma linha, usamos a função **rbind()**:

> linha = data.frame(profissao="marceneiro",salarios=3271.19,idade=30,tempo=7)

> Colaboradores = rbind(Colaboradores,linha)

Construída uma base, ou um conjunto de variáveis, podemos exportá-los de duas formas: i) em arquivo de dados com extensão ".RData"; ii) em arquivos de dados com extensão .txt, .csv, .xlsx. Se quisermos exportar como arquivo do tipo RData, usamos a função **save()**:

> save(Colaboradores, file = "~/Desktop/DadosColaboradores.RData")

Note que, caso deseje salvar todas as variáveis criadas no "Environment", basta usar o botão save (disquete), localizado no canto superior direito. Para toda vez não necessitar nomear o diretório onde os arquivos devem ser salvos, podemos definir um diretório padrão, usando o menu de tarefas: "Session → Set Working Directory → Choose Directory". Ou, alternativamente, no próprio console do R:

> setwd("~/Desktop")

Um detalhe importante: o arquivo .RData, por consequência, só poderá ser aberto no R, e isso pode ser não útil para alguns usuários. Assim, exportar os dados com outra extensão é fundamental. Para exportar dados em extensão .txt ou .csv usamos a função **write.table()**:

> write.table(Colaboradores,"Dados.txt")

> write.table(Colaboradores,"Dados.csv")

Quando não é mencionado o diretório, como nos exemplos acima, os arquivos são salvos no diretório padrão (definido usando a função **setwd()**. Para salvarmos arquivo com extensão .xlsx, em caso do pacote necessário ainda não estar instalado no R, precisamos instalar antes. Pacotes (em inglês *packages*) em R são bibliotecas contendo funções e dados, que não tem uma utilidade geral, mas são importantes para alguma finalidade específica. No nosso caso, usaremos o pacote chamado **"xlsx"**, que contém uma função que faz a exportação de dados para planilhas em excel.

Para instalar um pacote, usamos a função do R **install.packages()**:

> install.packages("xlsx")

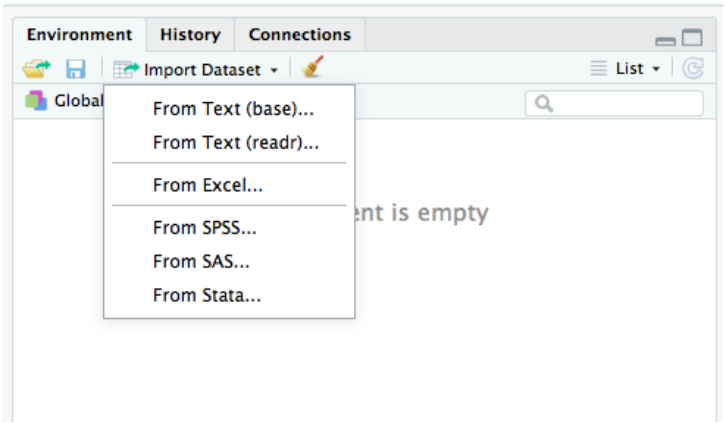
Note que precisamos conhecer o pacote a ser instalado, inserir seu nome correto e entre aspas duplas. Toda vez que um pacote é instalado, ele já faz parte da biblioteca do R na sua máquina, portanto, não precisamos instalar duas vezes um mesmo pacote. Toda vez que necessitarmos usar uma função de um pacote instalado, precisamos indicar para o R deixar tal pacote disponível para uso, por meio do uso da função **library()**:

> library(xlsx)

Perceba que agora o nome do pacote não vai entre aspas duplas. Esse pacote, xlsx, contém uma função "**write.xlsx()**" que faz a exportação de dados para planilhas excel. Assim, um exemplo de uso é:

> write.xlsx(Colaboradores,"Dados.xlsx")

A importação de dados externos é bem simples. Basta acessar o ícone "Import Dataset" no environment:



Em caso da opção "From Excel..." não estiver disponível em sua versão do R, basta instalar o pacote **readxl**.

Com os dados disponíveis, agora vamos construir representações gráficas dos mesmos: construir figuras. Desenvolveremos quatro tipos: linhas, scatter, pizza, barras, e histograma. A planilha "DadosExemplos.xlsx" contém conjuntos de dados que, depois de importados, vamos construir figuras. Na primeira aba, temos os preços históricos das ações MGLU3. Para construir um gráfico de linha, usamos a função **plot()**:

> plot(Dados$Data,Dados$MGLU3, type="l', main="Preços", xlab="Tempo", ylab="Preço")

Os argumentos type, main, xlab e ylab indicam, respectivamente, o tipo do gráfico, o título, o título do eixo x e o título do eixo y. Os tipos de gráfico são "l" (linha), "p" (pontos), "o" ou "b" (linhas e pontos), "s" (escada), e "h" (barras). Note que os dados são importados como data frame, portanto, para fazer a identificação dos dados usamos $ na indexação. Agora, vamos plotar os dados da aba Phillips, que contém um histórico mensal da taxa de inflação e da taxa de desemprego no Brasil, para construirmos a chamada Curva de Phillips, desenvolvida na teoria econômica para observar a dinâmica conjunta de tais variáveis:

> plot(Dados$Desemprego,Dados$Inflacao, type="p', main="Curva de Phillips", xlab="Desemprego", ylab="Inflação",col="blue")

Note que agora incluímos o argumento col, que define a cor do gráfico. Obviamente que a função plot contém mais argumentos, que podem ser verificados usando o help do R.

Outro exemplo de ilustração gráfica, corresponde às formas de barra e pizza. Na planilha "DadosExemplos.xlsx", aba "Carteira", temos um exemplo de uma carteira composta por diferentes ativos em distintas proporções. Podemos construir a representação por meio da função "**barplot()**" ou "**pie()**", por exemplo:

> barplot(DadosExemplos$Participacao,names.arg = DadosExemplos$Ativo,main ="Composição da Carteira",ylab="Pesos",horiz = FALSE,border = "red",col = "blue",density = 10)

> pie(DadosExemplos$Participacao,labels = DadosExemplos$Ativo,main="Composição da Carteira")

Note que, na função barplot() temos argumentos que definem se as barras devem estar na vertical (horiz = FALSE) ou na horizontal (horiz = TRUE), a cor da borda das barras (border) e a inclusão de linha nas barras (density).

Histrogramas também são interessantes para verificar a frequência de ocorrência de determinada informação a ser analisada. Com base nos dados da idade dos passageiros do Titanic, planilha "DadosExemplos.xlsx", aba "Titanic", podemos construir o histograma correspondente usando a função "**hist()**":

> hist(DadosExemplos$Idade,ylab = "Frequência",xlab = "Idade",main = "Histograma",density = 13,nclass = 15,col = "purple",border = "yellow")

Agora, o argumento nclass define o número de barras de frequência. Em caso de ausência desse argumento o número é definido automaticamente de acordo com os dados. Por fim, podemos criar *frames* de figuras, ou seja, várias figuras em uma única imagem. Para isso usamos a função "**par()**" para definir o frame de figuras e, em seguida, inserimos sequencialmente os gráficos. Na planilha "DadosExemplos.xlsx", aba "Acoes", temos os preços históricos de quatro ações, para formarmos um frame com quatro figuras usamos o seguinte conjunto de comandos:

> par(mfrow=c(2,2)) # frame 2 × 2, 2 linhas e 2 colunas

> plot(DadosExemplos$Data,DadosExemplos$GGBR4,type="l", ylab="GGBR4")

> plot(DadosExemplos$Data,DadosExemplos$ITUB4,type="l", ylab="ITUB4")

> plot(DadosExemplos$Data,DadosExemplos$NTCO3,type="l", ylab="NTCO3")

> plot(DadosExemplos$Data,DadosExemplos$CVCB3,type="l", ylab="CVCB3")

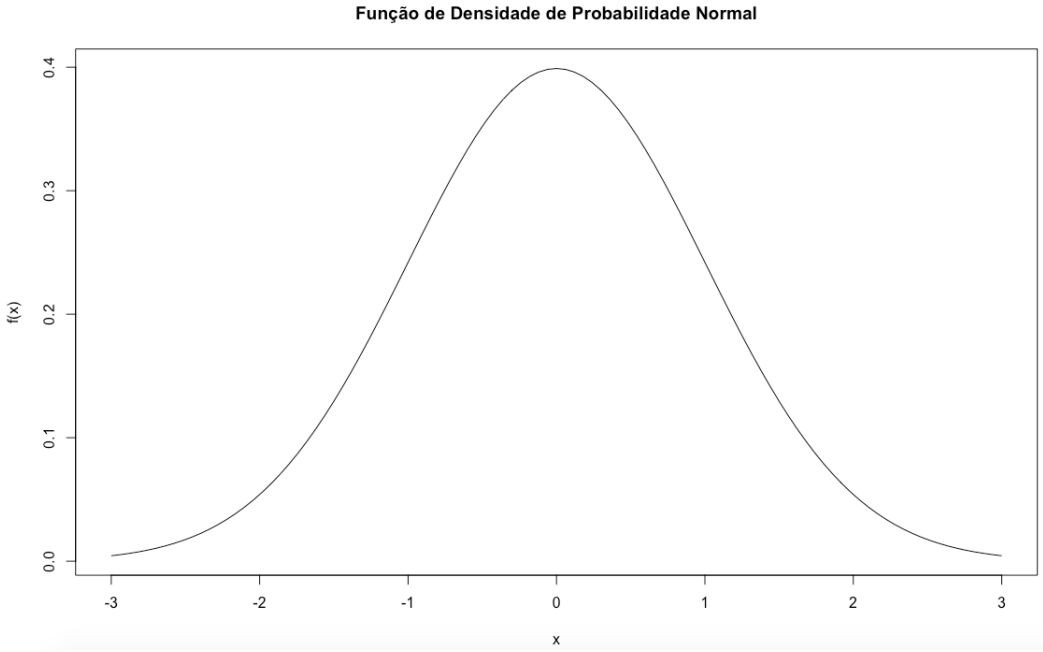
Existem inúmeros pacotes em R que permitem a construção de figuras mais sofisticadas, os principais são: graphics, lattice e ggplot2.

9. Distribuições de Probabilidade e Números Aleatórios

O R apresenta diversas funções para tratar distribuições teóricas de probabilidade. As estruturas das funções básicas são similares para diferentes distribuições. Como exemplo, vamos desenvolver para a distribuição mais conhecida, a distribuição Normal. Uma v.a. X com distribuição Normal, com média μ e variância σ^2, tem a seguinte função densidade de probabilidade:

$$f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}, -\infty < x < \infty$$

A figura a seguir ilustra uma curva Normal com média zero e variância unitária, $X \sim N(0,1)$.



Se estamos interessados em calcular o valor da função de densidade de probabilidade, usamos a função "**dnorm()**", ou seja:

> dnorm(-0.4, mean = 0, sd = 1)

Que calcula o valor da função de densidade de probabilidade para $X = -0.4$, sendo $X \sim N(0,1)$. A partir dessa função, conseguimos construir a função de densidade para qualquer variável aleatória com distribuição Normal. Por exemplo, suponha uma v.a. com distribuição normal com média 3,5 e variância 14, sua função de densidade pode ser construída com os seguintes comandos:

> x = seq(-10,25,0.1)

> fdp = dnorm(x, mean = 3.5, sd = sqrt(14))

> plot(x,fdp,type="l")

A partir da distribuição de densidade de probabilidade, podemos calcular as respectivas probabilidades por integração:

$$P(a < X < b) = \int_a^b f(x)dx$$

No R, para calcularmos a probabilidade de uma v.a. com distribuição Normal usamos a função "**pnorm()**", como por exemplo:

> pnorm(4, mean = 4.5, sd = 2)

Note que, nesse exemplo, a seguinte probabilidade está sendo calculada:

$$P(X < 4), \text{ com } X \sim N(4.5,4)$$

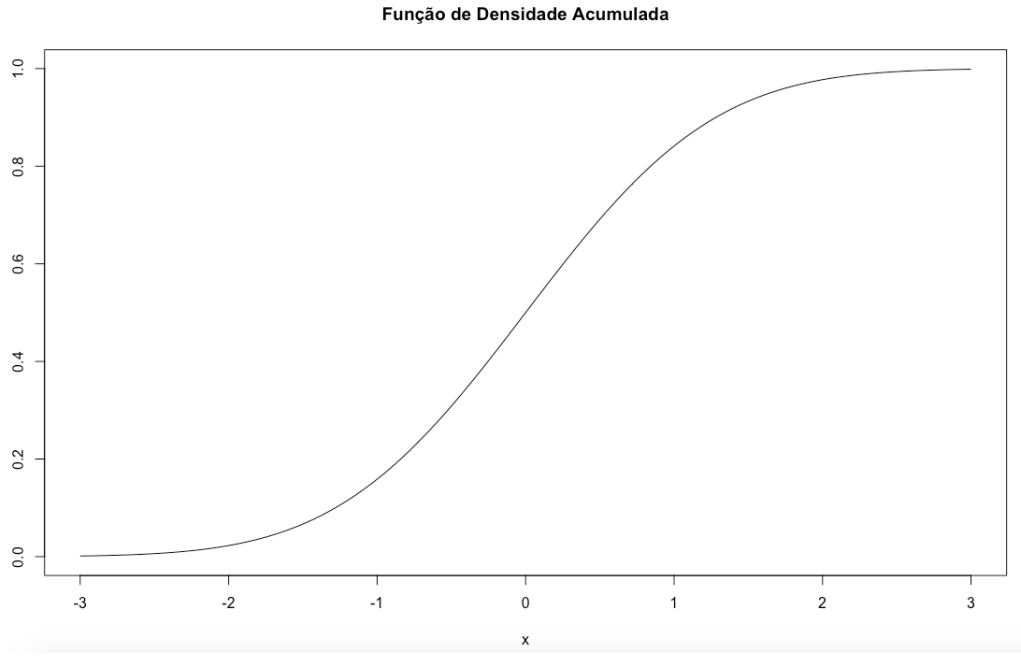
A função recebe três argumentos, o primeiro corresponde ao valor de interesse, o segundo a média da variável aleatória, e o terceiro seu desvio padrão (lembre-se que $\sigma = \sqrt{\sigma^2}$). Ou seja, a função pnorm() calcula a probabilidade acumulada pela cauda esquerda da distribuição. Se quisermos calcular, para esse mesmo exemplo, $P(X > 4)$, ao sabermos que a total área sob a curva soma a unidade, fazemos:

> 1 - pnorm(4, mean = 4.5, sd = 2)

De forma alternativa, se estamos interessados em calcular $P(1.5 < X < 4.6)$, usamos:

> pnorm(4.2, mean = 4.5, sd = 2) − pnorm(1.8, mean = 4.5, sd = 2)

Como a função pnorm() calcula a probabilidade acumulada, podemos construir a função de densidade acumulada que tem a seguinte forma:



Como exemplo, a função de densidade acumulada pode ser construída como:

> x = seq(-10,25,0.1)

> fdc = pnorm(x, mean = 3.5, sd = sqrt(14))

> plot(x,fdc,type="l")

Alternativamente, se estamos interessados em calcular um determinado valor de uma variável aleatória que resulta em uma probabilidade especificada, usamos a função "**qnorm()**":

> qnorm(0.35, mean = 3.5, sd = sqrt(14))

Ou seja, estamos calculando um valor x tal que:

$$P(X < x) = 35\%$$

Por fim, também podemos gerar números aleatórios de uma v.a. com distribuição conhecida utilizando a função "**rnorm()**":

> valoresAleatorios = rnorm(500, mean = 3.5, sd = sqrt(14))

Nesse exemplo, a variável chamada "valoresAleatorios" guardará um total de 500 valores aleatórios de um v.a. com distribuição normal com média 3,5 e variância 14. O primeiro argumento da função é, portanto, a quantidade de números aleatórios que serão gerados.

A partir dessas quatro funções, podemos desenvolver análises estatísticas com a utilização de distribuições teóricas de probabilidade, inclusive simulações. O formato e funcionalidade dessas funções também se estendem para outras distribuições, tais como binomial, chi-quadrado, F de Snedecor, log-normal e t de Student:

- Binomial → dbinom, pbinom, qbinom, rbinom;
- Chi-quadrado → dchisq, pchisq, qchisq, rchisq;
- F de Snedecor → df, pf, qf, rf;
- log-normal → dlnorm, plnorm, qlnorm, rlnorm;
- t-Student → dt, pt, qt, rt.

A seguir, são apresentados alguns exercícios para utilização das funções de distribuições de probabilidades teóricas.

Exemplo 1: Para $X \sim N(100, 100)$, calcule: i) P($X < 115$); ii) P($X > 80$); iii) P($85 < X < 110$).

Exemplo 2: As alturas de um conjunto de alunos têm distribuição aproximadamente normal com média de 170 cm e desvio padrão de 5 cm. Responda: i) qual a probabilidade de se observar um aluno com altura superior a 165 cm?; ii) qual o intervalo simétrico em torno da média que conterá 75% das alturas dos alunos?

Exemplo 3: Suponha que as amplitudes de vida de dois aparelhos elétricos, D_1 e D_2, tenham distribuições $N(42, 36)$ e $N(45, 9)$, respectivamente. Se os aparelhos são feitos para ser usados por um período de 45 horas, qual aparelho deve ser preferido? E se for por um período de 49 horas?

Exemplo 4: Uma máquina de embalar macarrão está regulada de forma que os pesos dos pacotes seguem uma distribuição Normal com média de 235 gramas e desvio-padrão de 7 gramas. Com base nessas informações: i) construa a função de densidade de probabilidade e a função de distribuição cumulativa dessa máquina; ii) qual a probabilidade de um pacote apresentar um peso maior que 242,5 gramas?; iii) qual a probabilidade de um pacote apresentar um peso entre 221,12 e 239,89 gramas?

Outras funções importantes para análise estatísticas também são descritas na tabela a seguir.

Nome da Função	**Funcionalidade**
quantile()	calcula os quantis de um conjunto de valores
boxplot()	construi gráficos de boxplot
summary()	calcula estatísticas descritivas de um conjunto de valores
cor()	calcula a correlação entre variáveis
cov()	calcula a covariância entre variáveis
kurtosis()*	calcula o coeficiente de curtose de um conjunto de valores
skewness()*	calcula o coeficiente de assimetria de um conjunto de valores

As funções skewness e kurtosis estão contidas na biblioteca do pacote "**moments**". Nesse sentido, o objetivo de análise do pesquisador direciona o conhecimento das ferramentas estatísticas a serem utilizadas.