Escola Nacional de Saúde Pública – Fiocruz
Departamento de Epidemiologia e Métodos Quantitativos

Aprendendo R

Antonio Guilherme Fonseca Pacheco
Geraldo Marcelo da Cunha
Valeska Lima Andreozzi

O objetivo desse material é introduzir o ambiente R para alunos de pós-graduação em Saúde Pública e mostrar suas vantagens e desvantagens. Gostaríamos de iniciar a apresentação do R a partir de algumas perguntas que são comuns (e que na maioria das vezes foram feitas por nós mesmos antes de termos nos tornados "amantes do R")
;-)

O que é o R?

O R é um sistema desenvolvido a partir da linguagem S (que também é usada numa versão comercial – o S-Plus), que tem suas origens nos laboratórios da AT&T no final dos anos 80. Em 1995 dois professores de estatística da Universidade de Auckland iniciaram o "Projeto R", com o intuito de desenvolver um programa estatístico poderoso baseada em S, e de domínio público.

Com o R posso utilizar menus para fazer análises estatísticas, como no SPSS, SAS e S-Plus?

Não. O R em versão para Windows é até provido de menus, mas todos são usados para realizar tarefas não estatísticas (como atualizar a versão ou salvar um gráfico). Todas as funções estatísticas que acompanham o R devem ser chamadas a partir do cursor do programa (seja digitando um comando ou copiando e colando um comando previamente digitado).

O fato do R não possuir menus não seria uma desvantagem em relação a outros pacotes estatísticos?

Depende. Muitos irão certamente interpretar esse fato como uma desvantagem, mas a gente entende que na verdade esta é uma vantagem forte do R. A utilização do R para realizar análises estatísticas exige muito mais do que simplesmente apertar alguns botões em série e dar alguns cliques no mouse: para trabalhar dados com o R é preciso PENSAR e ENTENDER o que se está fazendo. Ao contrário de muitos pacotes estatísticos clássicos, o R permite uma grande flexibilidade em relação às funções estatísticas pré-existentes, i.e. as funções são "editáveis", além da possibilidade de você mesmo poder criar as suas próprias funções personalizadas (como será mostrado mais tarde).

Quanto custa para ter uma cópia oficial do R?

Não custa nada: ele é de graça MESMO, ou seja, ninguém precisa gastar US$ 1.349, o que seria necessário para comprar o módulo básico do SPSS, por exemplo; nem ser obrigado a cometer um pequeno delito para usar o R.

Se ninguém está ganhando dinheiro para manter o R atualizado, como posso ter certeza que se trata de um produto confiável?

Esta é uma outra vantagem do R: o Projeto R é de uma colaboração internacional de vários pesquisadores que se comunicam através de uma eficiente lista de discussão pela Internet. Com isso, não só "bugs" (defeitos de programação) são detectados e corrigidos, como também novos módulos contendo métodos estatísticos recentemente implementados são regularmente disponibilizados e atualizados na rede.

O que são esses módulos adicionais?

Os módulos adicionais funcionam da seguinte forma: um pesquisador em algum lugar do mundo precisou desenvolver uma aplicação numa área que não é coberta nem pelo módulo básico nem pelos módulos de colaboradores existentes. O que esse pesquisador faz é desenvolver o que é chamada de uma biblioteca para o R com as funções que ele criou e utilizou, disponibilizando-a na rede. A vantagem é que a biblioteca pode ser usada por diferentes pessoas, que irão eventualmente reportar erros nas funções, que podem então ser atualizadas pelo seu criador.

Que plataformas (sistemas operacionais) suportam o R?

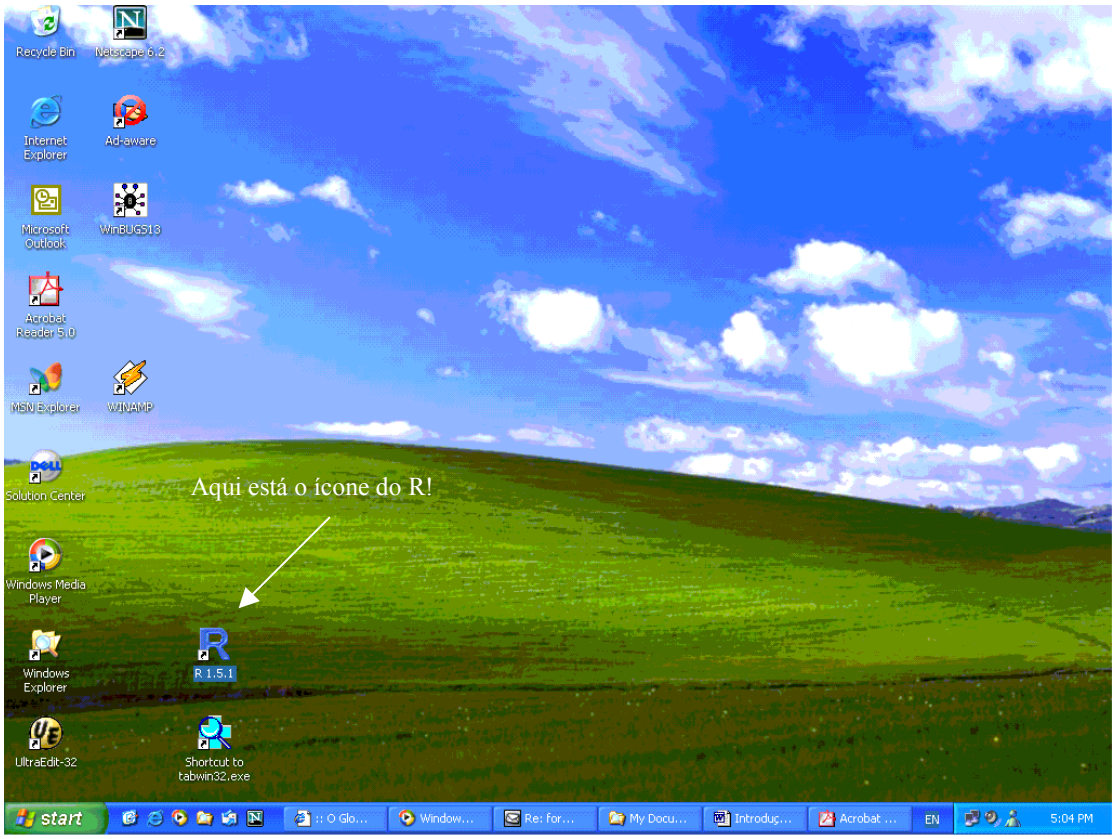
Atualmente o R está disponível para a família UNIX (incluindo LINUX), a maior parte dos Mac OS e ainda Windows 95, 98, NT, 2000, Me, XP.

Onde posso conseguir o R?

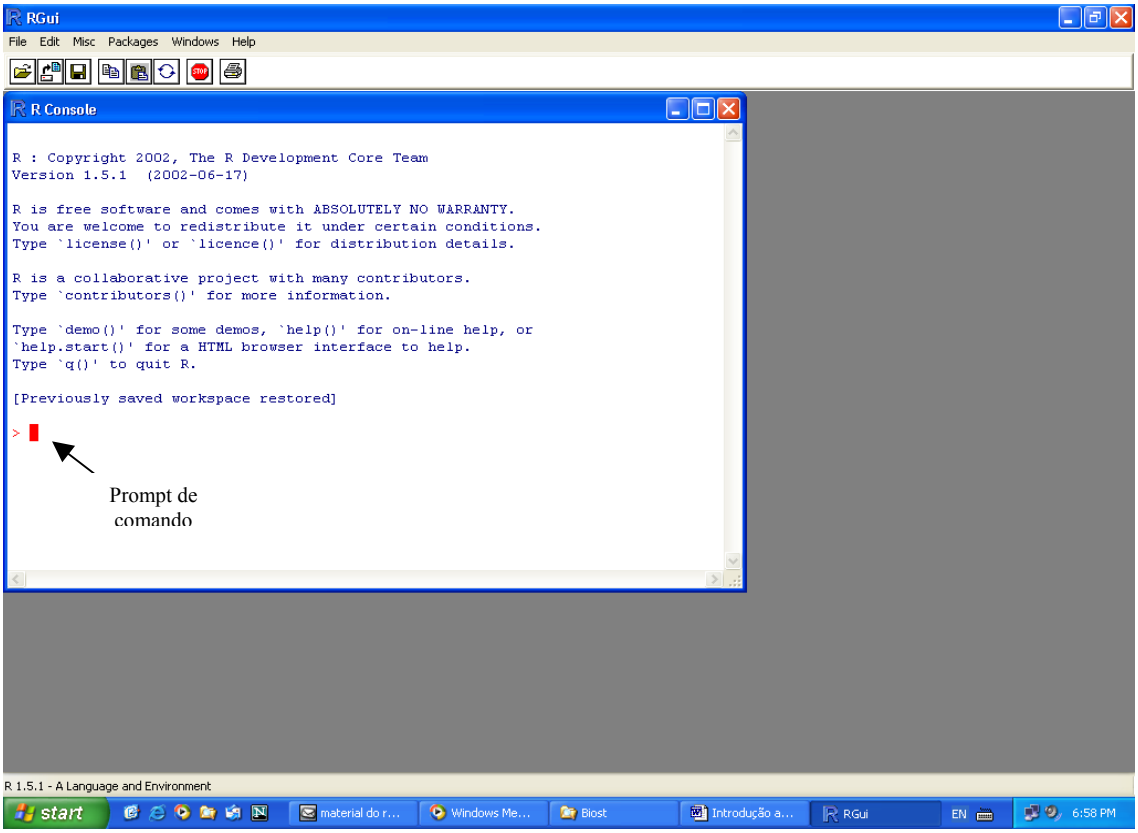
O R está disponível na internet no seguinte endereço:
http://www.r-project.org/

Módulo Básico

Antes de iniciarmos o uso do R é conveniente que algumas noções básicas sejam abordadas. Em primeiro lugar, você deve saber como executar um programa no Windows (o sistema operacional escolhido para ministrar essa introdução). Por ora, assumimos que ele já está instalado no seu computador, e que um ícone já aparece na área de trabalho. Portanto, dê um duplo clique na área de trabalho... Olhe um exemplo aqui em um computador qualquer:



Uma vez iniciado o programa, uma janela contendo o *prompt* de comando é aberta:



Observe que há uma janela dentro do programa com o *prompt* de comando. As saídas não gráficas no R aparecerão nesta mesma janela enquanto que as gráficas serão geradas em uma janela separada.

Cálculos e Funções

Vamos iniciar mostrando o R como calculadora. Por exemplo, tente fazer uma conta simples:

```
> 3+7
[1] 10
```

Repare que o resultado da operação é precedido por um `[1]`. Isto é apenas para indicar que o resultado "10" é o primeiro (e nesse caso único) elemento retornado pelo programa. Bem, essa é uma excelente deixa para dizer que o R pode funcionar como uma calculadora, como a que você já deve ter usado em cursos passados. Porque a gente não experimenta outras operações simples? Tente:

```
> 3-7
[1] -4

> 3*7
[1] 21

> 3/7
[1] 0.4285714
> 3^7
[1] 2187
```

Como em outros programas (e.g. Excel), os operadores matemáticos simples são "+", "-", "*", "/" e "^" – este último para potenciação. Você pode estar se perguntando agora, depois de algumas aulas de matemática: como é o inverso do operador potência? Ou melhor, como descobrir a que potência um certo número foi elevado para obtermos um determinado resultado?

Bom, nesse caso temos que lançar mão de uma **função**, já que como você deve se lembrar o operador inverso da potenciação é a função logarítmica. Recordando:

Se $3^7 = 2187 \Rightarrow \log_3(2187) = 7$, ou seja, o logaritmo de 2187 na base 3 é igual a 7. Lembrou? Será complicado obter esse resultado no R, sem a ajuda de um menuzinho? Bem, vamos tentar escrever exatamente como estamos interpretando a função?

```
> log(2187, base=3)
[1] 7
```

E não é que funcionou??? Quer dizer, precisamos elevar 3 à sétima potência para obter o resultado de 2187.

Vamos aproveitar esse exemplo para explicar algumas características básicas das funções disponíveis no R. Todas têm a forma:

```
> função (argumento(s) obrigatório(s), argumento(s) opcional(is))
```

Sendo que os argumentos opcionais podem ter um valor padrão pré-estabelecido ou não. Os argumentos estarão sempre entre parênteses sendo separados por vírgula. Se você deixar o primeiro argumento em branco, vai receber uma mensagem de erro:

```
> log(, base=3)
Error in log(, base = 3) : Argument "x" is missing, with no default
```

Repare que o programa já indica que o argumento (que ele chama de "x") obrigatório não foi fornecido e que não existe nenhum padrão (*default*). Agora, deixe o segundo argumento em branco:

```
> log(2187,)
[1] 7.690286
```

Dessa vez não houve mensagem de erro, mas o resultado é diferente do que a gente obteve anteriormente... O que terá acontecido? Bem, se não houve queixa do programa quanto ao segundo argumento, ele deve ter um default pré-estabelecido... Por que esse *default* seria 3?

Vamos descobrir, aproveitando para aprender a consultar a ajuda do R? Por que não tentamos assim:

```
> ?log
```

Note que uma nova janela se abriu no seu programa com a informação de ajuda sobre a função "log". Se você olhar a entrada "Usage", vai ver:

```
log(x, base = exp(1))
```

Isso quer dizer que a função `log()` precisa de um argumento `x` que não tem default e também de um argumento `base` que tem um valor pré-estabelecido, `exp(1)` que nada mais é que o número `e` (conhecido como algarismo Neperiano e aproximadamente igual ao valor 2.718282). Vamos tentar? Volte para a janela do prompt e veja o valor:

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

Lembrou dele? Vamos conferir então se foi isso mesmo que aconteceu. Vamos tentar assim:

```
> log(2187, base=2.718282)
[1] 7.690286
```

É o mesmo resultado!!! Parece ser isso mesmo!!!

A próxima pergunta que você pode estar fazendo é se existe uma maneira de economizar digitação omitindo o argumento `base =`. A resposta é sim, neste caso. Vamos tentar:

```
> log(2187, 3)
[1] 7
```

Neste caso deu certo porque a função só tem 2 argumentos possíveis e eles entraram na ordem certa. Se você trocar a ordem, o resultado é diferente:

```
> log(3,2187)
[1] 0.1428571
```

Se você trocar a ordem, porém especificando quem é quem, não haverá confusão:

```
> log(base=3,2187)
[1] 7
```

Moral da história: é sempre melhor especificarmos o que estamos escrevendo (embora muitas vezes nós tenhamos preguiça de fazer isso também...).

;-)

Vamos aproveitar e ver rapidamente algumas funções bastante usadas no R. Veja a tabela abaixo:

Função	Descrição
`sqrt()`	raiz quadrada
`abs()`	valor absoluto
`exp()`	exponencial
`log10()`	logaritmo na base 10
`log()`	Logaritmo na base e
`sin() cos() tan()`	funções trigonométricas
`asin() acos() atan()`	funções trigonométricas inversas
`sin() cos() tan()`	funções trigonométricas

Curioso para saber como essas funções funcionam? Tente usar a ajuda do R para descobrir. Por exemplo:

```
> help(sqrt)
```

Vai se abrir uma janela de ajuda que na verdade é genérica para a função "`abs()`" também. Esperamos que você tenha percebido que esta é uma forma alternativa de chamar a ajuda (em vez de `?sqrt`)

Como o R armazena Objetos

Muito bem, até agora ganhamos uma noção de como o R realiza operações matemáticas e de como as funções pré-definidas (e também as que um dia você mesmo vai criar!!!) funcionam.

Mas você deve estar se perguntando: e os dados? Eu preciso mesmo é analisar dados com este programa... Como é que funciona? Como é que o R armazena dados, sejam eles digitados a partir do teclado, sejam eles importados de um arquivo externo?

Bom, antes de entrar nessa parte mais interessante, vamos começar a falar sobre a maneira diferente como o R guarda um objeto (que pode ser um banco de dados, a saída de uma função ou até mesmo uma fórmula) no seu sistema. Ao contrário de outros programas que você já deve conhecer (como Excel, Word, etc), o R não armazena cada um dos seus objetos como um arquivo que fica em uma determinada localização no seu disco rígido. Ficou difícil de entender? Vamos tentar explicar de novo.

Quando você usa um editor de texto como o Word, por exemplo, e você salva o seu trabalho pela primeira vez, o programa pede para você dar um nome ao seu arquivo e o salva fisicamente em um diretório (que em geral é em "Meus Documentos" – "My Documents" se versão em inglês). Uma vez salvo, aquele único arquivo estará naquela localidade com o nome que você deu e portanto você poderá abri-lo no Word para futura edição ou para imprimir, etc.

O R trabalha um pouco diferente. Na verdade, quando você cria uma série de objetos, eles podem ser salvos também, porém EM GRUPO, ou seja, o R salva TODOS OS OBJETOS num mesmo arquivo. Mais tarde, você poderá abrir este arquivo e trabalhar com TODOS os objetos que foram salvos.

Você deve estar se perguntando se isso não vai atrapalhar muito a sua vida... Depende. Apesar do R salvar o arquivão com o mesmo nome sempre (".RData" – o nome é esquisitão assim mesmo), você pode salvá-lo com um nome diferente (mas mantendo a extensão) em diretórios (ou pastas, o nome moderno...;-) diferentes. A vantagem é que você pode armazenar todos os objetos relativos a um determinado projeto num diretório próprio do projeto. Vamos ver um exemplo. Iremos salvar o nosso arquivão num diretório que já foi criado para você dentro de "Meus Documentos" chamado "Curso R". Faça assim:

- Na barra de menus do R, clique em "File" e em seguida em "Change dir"
- Uma pequena janela chamada "Change directory" se abrirá. Clique em "Browse"
- A janela "Browse for folder" se abrirá. Dê um duplo clique em "Meu Computador", depois em "C:" e em seguida em "CursoR"
- Clique em "OK" e em "OK" novamente na outra janela.
- Agora, vá em "File" de novo, mas clique em "Save Workspace"
- Note que o arquivo ".RData" está para ser salvo no nosso diretório "CursoR"
- Clique "Save"

Vamos agora verificar se o nosso arquivo está mesmo lá... (Para isso é melhor primeiro determinar em que versão de Windows vai ser ministrado o curso...). Experimente salvá-lo com outro nome, mas não se esquecendo de amnter a extensão (exemplo: "aluno.Rdata")

Vetores

Certo. Não fique ansioso ainda... Antes de vermos a questão de ler dados externos, seria interessante ver como podemos entrar com algumas formas simples de dados a partir do teclado diretamente.

A forma mais simples de armazenamento de dados (i.e. o mais simples objeto) no R é um vetor. Um vetor é um seqüência em uma ordem específica de valores de um mesmo tipo de dados. Como os dados são mais freqüentemente números, os vetores numéricos são os mais usuais. Vetores não numéricos formados por caracteres (como nomes) são também bastante utilizados e podem também ser construídos no R.

Vamos criar um objeto chamado `x`, mas colocando diferentes coisas nesse objeto.

Um número:

```
> x <- 3
```

Você já deve ter duas perguntas prontinhas. A primeira é: que diabo é esse <− que apareceu entre o objeto que eu queria criar – o `x` – e o número 3? Esse é o símbolo

de atribuição (*assignment*) no R. Ele significa que o número 3 foi "colocado dentro" do objeto `x`. Toda vez que a gente quiser colocar alguma coisa em um objeto, esse símbolo vai ter que ser usado.

A segunda pergunta é: muito bem, colocamos o número 3 no objeto `x`, mas como é que eu verifico se o objeto `x` realmente possui o número 3? É simples. Tente assim:

```
> x
[1] 3
```

Hummm... Quer dizer que para visualizar um objeto basta digitar o seu nome? É isso aí!!! Vamos aproveitar para mostrar um detalhe a mais do R. Para ele maiúsculas são DIFERENTES de minúsculas. Digite `X` em vez de `x`:

```
> X
Error: Object "X" not found
```

O programa informa que o objeto `X` não existe (existe o `x`).

Mas não era isso que estávamos falando. Vamos voltar para os diferentes conteúdos de um vetor... Que tal entrarmos com um caracter, por exemplo, com a palavra banana?

```
> x <- "banana"
> x
[1] "banana"
```

Percebeu a diferença deste caso para o caso numérico? Sempre que estivermos trabalhando com caracteres devemos usar aspas duplas.Você deve estar se perguntando o que teria ocorrido se não tivéssemos utilizado aspas na palavra banana... O problema é que se não fizermos isso, o R vai pensar que banana é o nome de um outro objeto...

```
> x <- banana
Error: Object "banana" not found
```

Viu só? O R reclama que o objeto banana não existe. Mas e se esse objeto existisse? O que aconteceria? O R copiaria o conteúdo do objeto banana para o objeto x. Estranho? Não. Essa é a maneira de se fazer uma cópia de um objeto no R. Vamos criar um objeto chamado banana, contendo o número 7 e copiar o seu conteúdo para o objeto x:

```
> banana <- 7
> banana
[1] 7
> x <- banana
> x
[1] 7
```

Pegou? Legal. Agora, você deve ter notado um probleminha... a gente tinha colocado no objeto x primeiramente o número 3, depois a palavra banana e por fim o número 7, quando copiamos o conteúdo do objeto banana para x. Repare que as substituições de conteúdo do objeto x foram feitas sem nenhuma cerimônia pelo R. Pois é, isso pode ser um problema no R: ele não pergunta se você quer ou não substituir o conteúdo de um objeto com um nome, por outro. Para evitar esses acidentes, vamos aprender mais tarde uns macetes para economizar digitação... Aguardem...

;-)

Até agora só vimos exemplos que raramente usaremos, né? Quem é que vai entrar UM valor em um vetor??? Embora façamos isso muitas vezes no caso de programação avançada, vamos ver algo mais interessante. No R, para entrar com vários números (ou nomes, ou qualquer outro grupo de coisas), precisamos usar uma função para dizer ao programa que os valores serão *combinados* em um único vetor. Vamos tentar:

```
> x <- c(1,2,3,4,5)
> x
[1] 1 2 3 4 5
```

A essa altura, você já notou que a função usada foi `c()` e serve evidentemente para combinar elementos. Vamos ver com nomes:

```
> x <- c("banana", "laranja", "tangerina")
> x
[1] "banana"    "laranja"   "tangerina"
```

Por último, vamos abordar uma outra função que vai ser muito usada no R, e que a gente queria apresentar para você no contexto de vetores. É a função usada para gerar uma `seq`üência de números. Imagine como seria esta função...Acertou:

```
>x <- seq(from=1, to=12)
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Usamos a função `seq` para criar uma seqüência de (`from`) 1 a (`to`) 12. Bom, lembra quando a gente disse que era preguiçoso? Pois é: dá para fazer a mesma coisa de duas maneiras mais rápidas. Uma é omitindo os argumentos `from` e `to`:

```
> x<- seq(1, 12)
> x
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Agora, a campeã da preguiça mesmo seria:

```
> x<-1:12
```

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

É muita mesmo, não é???

Uma outra facilidade oferecida pelo R é a capacidade de selecionar valores armazenados em posições específicas dentro de um vetor (na verdade, como veremos adiante, essa capacidade se estende para outros objetos do R como matrizes e *data frames*). Vamos, por exemplo, criar o vetor `bicho` (só para treinar):

```
> bicho<-c("macaco","pato","galinha","porco")
> bicho
[1] "macaco"  "pato"    "galinha" "porco"
```

Que tal visualizarmos o conteúdo da posição 3? Vamos fazer assim:

```
> bicho[3]
[1] "galinha"
```

E se quiséssemos selecionar os conteúdos das posições 1, 3 e 4?

```
> bicho[c(1,3,4)]
[1] "macaco"  "galinha" "porco"
```

A gente utilizou a função `c()` para combinar os números correspondentes às coordenadas dos elementos (na verdade criamos um vetor com esses valores). E podemos ainda selecionar os conteúdos através de uma seqüência de posições:

```
> bicho[2:4]
[1] "pato"    "galinha" "porco"
```

Geração de Distribuições e Gráficos

Vamos ver agora algumas funções no R para a geração de distribuições estatísticas. Esta evidentemente é uma das especialidades do R, já que ele é voltado para funções estatísticas principalmente.

Comecemos com as distribuições contínuas. A mais badalada de todas, naturalmente é a Normal, que você já deve estar cansado de ouvir falar. Vamos gerar então um vetor com 100 valores de uma distribuição Normal, digamos com média 10 e variância 4 (estes são os parâmetros de uma distribuição Normal):

```
> x <- rnorm(100, mean=10, sd=2)
```

Repare que a função se chama `rnorm` e que os argumentos usados foram o número de valores a serem gerados, a média (`mean`) e o desvio-padrão (`sd` – de *standard deviation* em inglês). Cabem dois comentários: primeiro o nome da função.

Bom, o `norm` vem de Normal, é claro e o `r` vem de *random*, aleatório em inglês, pois a gente está gerando aleatoriamente 100 valores de uma distribuição Normal(10, 4). A segunda observação é que a rigor (embora nem todos sejam rigorosos) o segundo parâmetro da Normal é a sua variância (e não o seu desvio-padrão). De qualquer maneira, o R recebe o argumento relativo ao desvio-padrão e não à variância.

Você pode estar pensando em duas coisas. Primeiro: e se eu só tivesse a média e a variância e não o desvio padrão em um problema e quisesse gerar a Normal? Eu teria que fazer a conta e colocar o valor? Você pode fazer isso... Entretanto, o R aceita colocar uma função dentro de outra função. Assim, o mesmo resultado pode ser obtido fazendo:

```
x <- rnorm(100, mean=10, sd=sqrt(4))
```

A segunda questão é: será que a função `rnorm()` tem um *default*? A resposta é sim, pelo menos para os argumentos `mean` e `sd`. Ganha um doce quem adivinhar quais são os valores padrões... Acertou quem pensou na mais famosa das Normais: A Normal Padronizada ou Normal (0, 1), ou seja, média zero e variância (ou desvio-padrão) 1. Para verificar esse fato, vamos usar duas funções estatísticas, para calcular a média e o desvio-padrão de vetores. Vamos começar pelo vetor que a gente conhece:

```
> x <- rnorm(100)
> mean(x)
[1] -0.1358806
> sd(x)
[1] 1.053232
```

Epa!!! A média aqui no nosso exemplo não é exatamente 0. E o desvio-padrão também não é exatamente 1. Será que o *default* da função não é gerar a partir da Normal (0,1)?

Não é nada disso! O problema é que o vetor que nós criamos é gerado de maneira aleatória, de modo que, em média, esses valores convergem assintoticamente para os valores estabelecidos para os parâmetros. Ficou difícil? É o seguinte: se a gente aumentar o número de valores gerados, esses valores devem ir se aproximando cada vez mais dos valores dos parâmetros. Querem ver?

```
> x100 <- rnorm(100)
> mean(x100)
[1] -0.1111455
> sd(x100)
[1] 1.121295

> x1000 <- rnorm(1000)
> mean(x1000)
[1] -0.01045328
> sd(x1000)
[1] 1.024044

> x10000 <- rnorm(10000)
```

```
> mean(x10000)
[1] 0.005090138
> sd(x10000)
[1] 0.9953523
```

Ah, e se for fazer isso no R você mesmo, não digite tudo de novo: use a tecla de setinha para cima (↑) que você chama o último comando e pode editá-lo a partir do teclado... Mas atenção, para mover o curso na linha de comando você deve usar as setinhas para direita e esquerda (→ ou ←)

;-)

Muito bem, você acabou aprendendo a simular uma distribuição Normal com diferentes parâmetros e também a conferir a média e o desvio padrão do vetor que você gerou. Além disso, ganhou uma noção que a média e o desvio padrão de vetores maiores serão em princípio mais aproximados da média e desvio padrão das distribuições que os geraram. Por esse motivo, vamos utilizar o nosso amigo, o objeto `x` com 10000 valores nos próximos exemplos.

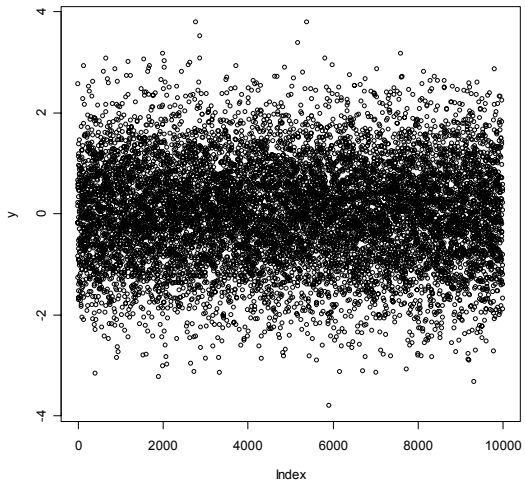
Que tal uma visualização gráfica das nossas distribuições? Você já deve ter ouvido falar de tipos de gráficos comumente usados em estatística, como o *scatter plot* (ou gráfico de dispersão), o histograma, o *boxplot* (ou gráfico de caixas) e ainda um que gostamos muito, o *stem and leaves* (ou ramo e folhas). Por que a gente não aproveita então para dar uma olhada nesses gráficos usando os vetores que foram criados? Vamos lá. Primeiro vamos criar um vetor `y` com 10000 valores de uma Normal Padronizada:

```
> y <- rnorm(10000)
```

Agora vamos "plotar" o nosso vetor:

```
> plot(y)
```

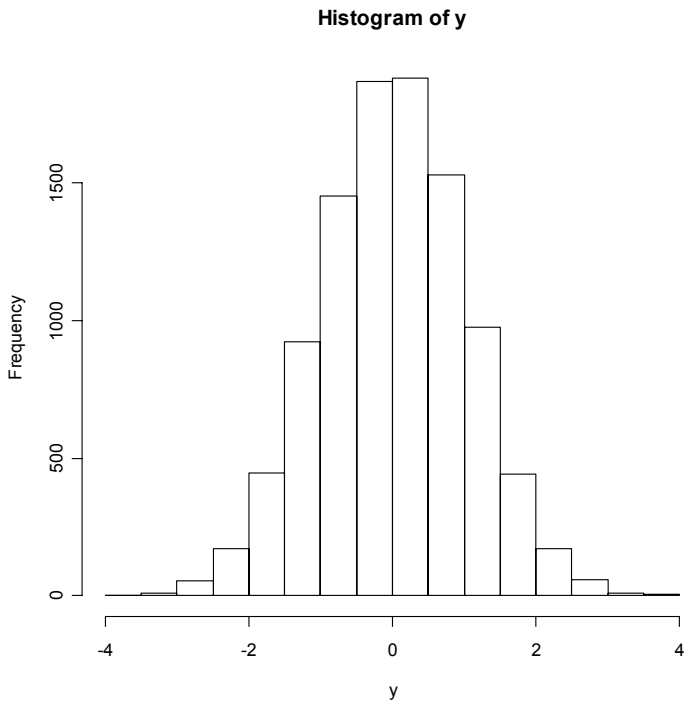
Se você digitou essa função no *prompt* do R, notou que aconteceu algo diferente. A saída dessa função não foi listada na mesma janela, mas uma outra janela diferente foi aberta e um gráfico foi mostrado. Os chamados gráficos de alto nível são sempre plotados nessa janela. O gráfico deve ser mais ou menos assim:



Alguns detalhes devem ser notados nessa figura. Primeiro, apesar da função ser `plot()`, o gráfico mostrado é um *scatter plot*. Esse é o *default* de uma função `plot()` para um vetor, onde o eixo *x* é o índice, ou seja a posição do elemento dentro do vetor, e o eixo do *y* é o valor dos elementos do vetor. O segundo detalhe são os valores do nosso vetor. Observe que, em se tratando de uma Normal Padronizada, cuja média é zero, a maior parte dos pontos se encontra concentrada em torno desse valor (aliás, são tantos os pontos nessa região que eles formam uma verdadeira "nuvem" de pontos). Além disso, note que a esmagadora maioria dos pontos se concentra entre os valores –2 e 2. Você seria capaz de dizer qual a percentagem aproximada de pontos que deve estar entre esses valores?

Outro gráfico muito usado é o `hist`ograma. Vamos ver o jeitão dele:

```
> hist(y)
```



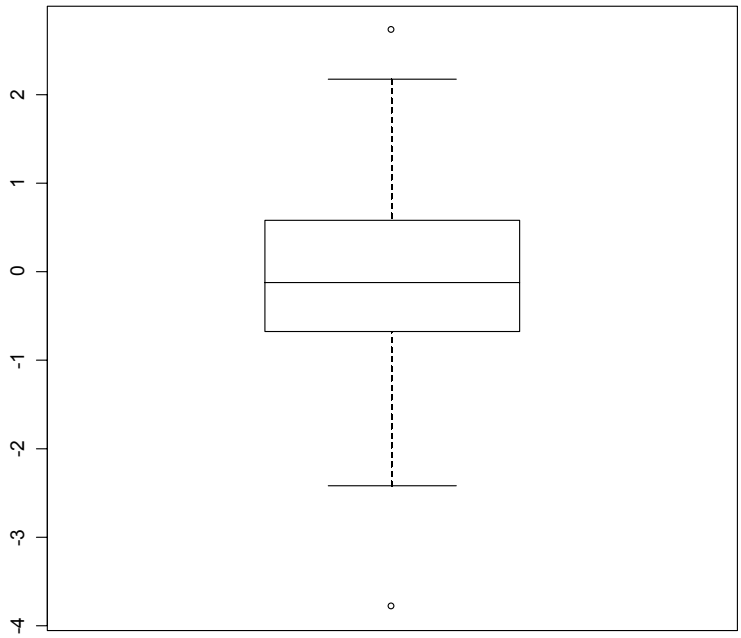
Repare como o histograma lembra bastante a densidade de uma distribuição Normal. Este gráfico tem os valores do vetor *y* no eixo horizontal e as freqüências (absolutas nesse caso) dos valores contidos nos intervalos pré-estabelecidos pelo programa no eixo vertical. Curioso para saber variações da função `hist()`? Consulte a ajuda. Lembra como?

```
> ?hist
```
OU
```
> help(hist)
```

O *boxplot* é também um gráfico interessante principalmente para analisar a dispersão dos dados e também para detectar a presença de *outliers*. Para quem não sabe ou não se lembra, *outliers* são pontos que caem em uma região muito afastada do centro

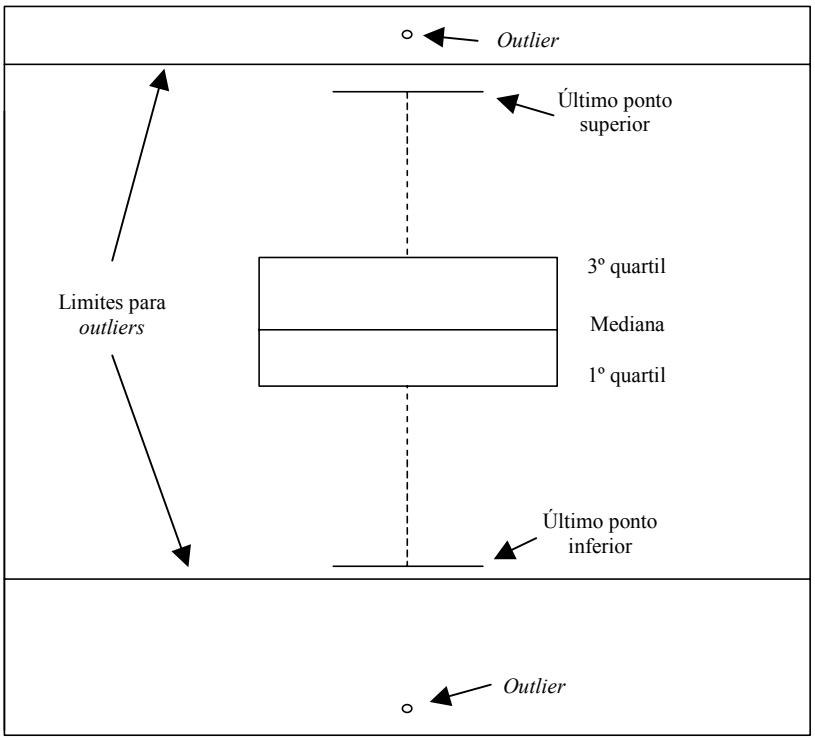
da distribuição (i.e. muito afastados da média e da mediana). Vamos explicar melhor com a visualização de um *boxplot* propriamente dito:

```
> boxplot(y)
```



A linha central do retângulo (que seria a nossa "caixa") representa a **mediana** da distribuição (sabe o que significa mediana? Não? As bordas superior e inferior do retângulo representam o percentis 25 e 75, respectivamente (também conhecidos como primeiro e terceiro quartís, respectivamente). Logo, a altura deste retângulo é chamada de distribuição interquartilar (DI). Os traços horizontais ao final das linhas verticais são traçados sobre o último ponto (de um lado ou de outro) que não é considerado um *outlier*.

A essas alturas você deve estar estranhando a falta de definição de *outlier* não é mesmo? A nossa primeira definição foi algo subjetiva, certo? É isso mesmo! Não há um consenso sobre a definição de um *outlier*. Porém, no caso do *boxplot* em geral, existe uma definição formal. A maior parte das definições considera que pontos acima do valor do 3º quartil somado a 1,5 vezes a DI ou os pontos abaixo do valor do 1º quartil diminuído de 1,5 vezes a DI são considerados *outliers*. Esses pontos são assinalados (no nosso exemplo, tivemos 2 *outiers*, um para cada lado). Vamos ver de novo o *boxplot,* para ficar mais claro:



Para o nosso próximo gráfico, que tal a gente tentar gerar outra distribuição também bastante comum em bioestatística, a binomial. Ao contrário da Normal, essa é uma distribuição *discreta*, ou seja, ela está definida para determinados valores (enumeráveis) num intervalo e não para *todos* os valores (não enumeráveis) em um determinado intervalo.

Assim como a Normal, a Binomial também possui dois parâmetros. O primeiro, *n* corresponde ao número de experimentos que serão realizados (também chamados de experimentos ou processos de Bernoulli); o segundo, *p* é a probabilidade de se obter um sucesso em cada um dos experimentos. A probabilidade de serem observados *k* sucessos é dada por:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

No R, a função para gerar números binomiais chama-se `rbinom()`, por motivos já óbvios. Evidentemente, nós precisamos especificar os dois parâmetros que mencionamos anteriormente:

```
> y <- rbinom(100, size=200, p=0.05)
```

Geramos um número menor de pontos dessa vez (100), para facilitar a visualização do nosso próximo gráfico. Esses parâmetros foram escolhidos

propositadamente para fazer um paralelo com uma situação que é comum em epidemiologia: termos cerca de 200 participantes em um estudo (correspondendo ao nosso `size = 200`) e com uma prevalência de uma determinada doença de 5% (correspondente a uma probabilidade de sucesso de 0.05 por indivíduo).

Para construir o *ramo e folhas* utilizamos a função `stem()`:

```
> stem(y)

  The decimal point is at the |

   2 | 0
   4 | 000000
   6 | 0000000000
   8 | 00000000000000000000
  10 | 0000000000000000000000000000
  12 | 0000000000000000000000000
  14 | 000000000
  16 | 00
  18 | 00
```

A idéia do ramo e folhas é separar um número (como 7,0) em duas partes. Neste caso, a primeiro parte inteira (7) chamada de ramo e a segunda, a parte decimal (0) chamada de folha. Além de separar os números em duas partes (inteira e decimal), o R agrupa os números em classes de tamanho 2. Por exemplo, o ramo 4 leva em conta os números 4 e 5. Não gostou dessa disposição? Não tem problema. Que tal dobrarmos a escala do gráfico para o R considerar cada número separadamente na sua própria classe?

```
> stem(y, scale=2)
  The decimal point is at the |
   3 | 0
   4 | 0000
   5 | 00
   6 | 00000
   7 | 00000000
   8 | 00000000000000000
   9 | 000000000000
  10 | 0000000000000000000
  11 | 0000000000
  12 | 00000000
  13 | 000000
  14 | 0000
  15 | 000
  16 | 00
  17 |
  18 |
  19 |
  20 |
  21 | 0
```

Já percebeu que o argumento extra para dobrar a escala é `scale=2`, né?

E por que a distribuição dos dados parece concentrada em torno de 10 (hum... nem tanto...)?

Bem, o que fizemos aqui foi gerar 100 amostras de 200 pessoas, contando o número de pessoas com a doença em cada amostra. Se a prevalência da doença é 0,05 a gente espera que em uma amostra de 200 pessoas, o número de pessoas com a doença seja igual a 10=200x0,05!! Mas como o processo é aleatório, existe a situação onde apenas 3 pessoas são afetadas pela doença e ainda situações extremas de 21 pessoas (mais que o dobro do esperado).

Vamos fazer um exercício interessante? O exercício consiste em observar o comportamento do número de pessoas afetadas por uma doença na medida em que aumentamos o valor da prevalência. Antes disso, vamos dividir o espaço gráfico em 6 partes através do comando:

```
> par(mfrow=c(2,3))
```

Ih... O Que é essa função `par()`??? Como você deve ter observado, uma janela de gráficos foi aberta. Pois é, essa é a função para atribuir parâmetros gráficos. O argumento `mfrow=` estabelece o número de gráficos que serão visualizados em uma mesma janela gráfica e em que disposição. Quando escrevemos no comando `c(2,3)`, ele implicitamente divide essa janela em 6 partes: 2 linhas e 3 colunas (de gráficos, né?) e que você vai visualizar quando plotarmos 6 gráficos em seqüência.

Escolhendo valores de prevalência 0.02, 0.04, 0.06, 0.08, 0.1 e 0.5 podemos fazer:

```
> hist(rbinom(100, size=200, p=0.02))
> hist(rbinom(100, size=200, p=0.04))
> hist(rbinom(100, size=200, p=0.06))
> hist(rbinom(100, size=200, p=0.08))
> hist(rbinom(100, size=200, p=0.1))
> hist(rbinom(100, size=200, p=0.5))
```

Beleza? Podemos visualizar como se altera a distribuição do número de pessoas atingidas pela doença na medida em que aumentamos o valor da prevalência.

Matrizes

Bem, vamos passar para o próximo tipo de objeto que vamos aprender no R. São as matrizes. Como você deve saber, matrizes são objetos numéricos, que possuem elementos com coordenadas (que são simplesmente a linha e a coluna às quais o elemento pertence). Para construir um objeto que seja uma matriz no R, precisamos usar uma função... Adivinha o nome:

```
> x <- matrix(c(1,2,3,4,5,6,7,8,9,10,11,12), ncol=3)
```

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

Muito bem. Neste momento você deve estar desesperado. Por que usamos uma função que tinha sido usada para criar um vetor *dentro* de uma outra função do R?

É isso mesmo. Na verdade "criamos" um vetor temporário com o a função `c()` como tínhamos visto anteriormente e depois esse vetor foi transformado numa matriz com a função `matrix()`. Quer ver? Vamos fazer por partes. Primeiro, crie o vetor "y":

```
> y <- c(1,2,3,4,5,6,7,8,9,10,11,12)
> y
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Lembre-se que esse vetor poderia ter sido gerado com a nossa seqüência:

```
> y <- 1:12
> y
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Agora, aplique a função `matrix()` ao vetor `y`:

```
> x <- matrix(y, ncol=3)
> x
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

A próxima dúvida é quanto ao argumento `ncol`. Ele representa o número de colunas da nossa gloriosa matriz. Como se trata de um vetor, o R não pode adivinhar qual a dimensão (número de linhas e número de colunas desejada da matriz. Quer ver?

```
> x <- matrix(y)
```

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

A essa altura você já deve ter notado que o *default* do R é transformar o vetor em uma matriz com apenas uma coluna. Um outro detalhe importante a ser comentado é a ordem na qual ele entra com os elementos na matriz. Observe que o preenchimento da mesma é feito pelas colunas. Vamos ver de novo a nossa matriz:

```
> x <- matrix(y, ncol=3)
> x
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Viu? O programa preencheu a coluna 1 com os números 1 a 4, a segunda com os números de 5 a 8 e a terceira com os demais. Mais tarde vamos aprender a mudar este comportamento. Mas se você for curioso, use a ajuda.... e boa sorte!

Agora você deve estar se perguntando como é possível visualizar um elemento (ou um grupo de elementos) contido numa matriz. A lógica é a mesma que com vetores, sendo que no caso das matrizes, os elementos possuem 2 coordenadas: uma para a linha e outra para a coluna. Quer ver um exemplo? Vamos visualizar o elemento da segunda linha, terceira coluna de `x`:

```
> x[2,3]
[1] 10
```

Como você já deve estar pensando, é possível selecionar, como nos vetores, um intervalo de valores. Digamos que você queira visualizar os 3 primeiros elementos da primeira coluna. Para isso façamos:

```
> x[1:3,1]
[1] 1 2 3
```

No caso das matrizes é possível selecionar uma linha (ou coluna) inteira, sem se preocupar em saber o número de colunas (ou linhas) da matriz. Não entendeu nada, né? Nem eu! Vamos tentar um exemplo: vamos selecionar as duas primeiras linhas da matriz, sem levar em conta o número de colunas:

```
x[1:2,]
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
```

Repare que o espaço depois da vírgula, que seria destinado às coordenadas das colunas, ficou em branco, indicando que TODAS as colunas deveriam ser selecionadas. Pode-se fazer um raciocínio semelhante para o caso de selecionar colunas inteiras.

Data Frames

O último tipo de objeto do R que vamos abordar neste material (é isso mesmo, tem mais...) são os chamados *data frames*. Esses objetos são equivalentes a um banco de dados que você provavelmente já viu em outros formatos (e.g. dbf), ou seja, trata-se de uma "tabela de dados" onde as colunas são as variáveis e as linhas são os registros.

Vamos ver um exemplo agora com um banco de dados externo que está no nosso diretório de trabalho "CursoR", chamado "iris.txt". Este arquivo é um banco de dados tipo ASCII. Nós vamos então usar uma função para ler uma tabela externa ao R:

```
dados <- read.table(file="iris.txt", header=T)
```

Já sei: são muitas dúvidas agora... Vamos assumir que você se lembra dos conceitos abordados anteriormente. O que nós fizemos foi criar um objeto `dados`, através da atribuição `<-` pela função `read.table`, que tem um argumento obrigatório (`file`) e um argumento opcional (`header=T`). Bom, `file` é simplesmente o nome do arquivo externo onde os dados estão armazenados, com a extensão inclusive e entre aspas. Se o arquivo estiver no diretório de trabalho basta escrever `"iris.txt",` caso ele esteja em un diretório diferente será necessário dar o caminho (*path*) completo, por exemplo, `C:\\dados\\iris.txt`, ou `C:/dados/iris.txt`. O argumento `header=T` é para indicar para o R que a primeira linha contém o nome das variáveis nesse banco. Se a gente quisesse ver esse banco inteiro, bastaria digitar o nome do objeto `dados` (como você deve se lembrar que acontece com qualquer objeto). Como esse banco tem 150 registros e 5 variáveis, que tal darmos uma olhada somente os 10 primeiros registros? Vamos tentar:

```
> dados[1:10, 1:5]
   slength swidth plength pwidth species
1      5.1    3.5     1.4    0.2  setosa
2      4.9    3.0     1.4    0.2  setosa
3      4.7    3.2     1.3    0.2  setosa
4      4.6    3.1     1.5    0.2  setosa
5      5.0    3.6     1.4    0.2  setosa
6      5.4    3.9     1.7    0.4  setosa
7      4.6    3.4     1.4    0.3  setosa
8      5.0    3.4     1.5    0.2  setosa
9      4.4    2.9     1.4    0.2  setosa
10     4.9    3.1     1.5    0.1  setosa
```

Você deve estar se perguntando por que usamos esse `[1:10, 1:5]`. Se com as matrizes dá para selecionar uma linha inteira, porque não poderia com o *data frame* que nada mais é do que uma matriz com características especiais. Acertou: A mesma lógica funciona aqui. Vamos tentar:

```
> dados[1:10,]
   slength swidth plength pwidth species
1      5.1    3.5     1.4    0.2  setosa
2      4.9    3.0     1.4    0.2  setosa
3      4.7    3.2     1.3    0.2  setosa
4      4.6    3.1     1.5    0.2  setosa
5      5.0    3.6     1.4    0.2  setosa
6      5.4    3.9     1.7    0.4  setosa
7      4.6    3.4     1.4    0.3  setosa
8      5.0    3.4     1.5    0.2  setosa
9      4.4    2.9     1.4    0.2  setosa
10     4.9    3.1     1.5    0.1  setosa
```

Muito bem, agora vamos ler um outro banco de dados externo – só para treinar – e arriscar umas estatísticas descritivas desse banco.

```
fem <- read.table("fem.dat", header=T)
```

Que tal iniciarmos nossa descrição de dados, conhecendo o nome das variáveis que estão contidas neste banco?

```
> names(fem)
[1] "ID" "AGE" "IQ" "ANXIETY" "DEPRESS" "SLEEP" "SEX"
"LIFE" "WEIGHT"
```

A descrição dessas variáveis são:

ID – Identificação do paciente
AGE – Idade do paciente
IQ – QI do paciente
ANXIETY – Ansiedade (1 = não; 2 = leve; 3 = moderada; 4 = grave)

DEPRESS – Depressão (1 = não; 2 = leve; 3 = moderada ou grave)
SLEEP – Dorme normalmente (1 = sim; 2 = não)
SEX – Perdeu o desejo sexual (1 = sim; 2 = não)
LIFE – Pensou em cometer suicídio? (1 = sim; 2 = não)
WEIGHT – Variação de peso em 6 meses (kg)

E se quisermos listar uma das variáveis apenas?

Você naturalmente respondeu que podemos selecionar uma das colunas inteiras do banco usando aquele nosso velho conhecido. Vamos selecionar a variável `IQ`, que corresponde à terceira coluna:

```
> fem[,3]
```

Vamos omitir a saída aqui para poupar espaço... Mas será que existe uma outra maneira de selecionar uma variável em um banco de dados no R? Uma maneira mais convencional, por exemplo, seria chamar o nome da variável e não tendo que saber especificamente a coluna correspondente.

Tente:

```
> fem$IQ
```

Viu? O símbolo `$` serve para indicar o nome de uma variável em um *data frame*.

Digamos agora que você queira listar todos os registros de pacientes que têm um QI acima de 100. Nesse caso, estaremos selecionando todas as colunas do banco e somente as linhas que satisfizerem essa condição. Vamos ver como seria:

```
> fem[fem$IQ >100,]

   ID AGE  IQ ANXIETY DEPRESS SLEEP SEX LIFE WEIGHT
25 25  35 103       2       2     2   1    2  -0.95
39 39  35 102       2       2     2   1    1   1.36
64 64  41 103       2       2     2   1    1  -0.36
87 87  36 106       2       2     2   2    1  -0.45
```

Entendeu o que foi feito? Selecionamos (com os colchetes), do banco `fem` as linhas (primeiro índice dentro dos colchetes) onde a variável `IQ` é > 100, e todas as colunas (o espaço em branco após a vírgula dentro dos colchetes – segundo índice).

Você deve ter observado que quando listamos a variável IQ alguns dos valores são negativos (especificamente `-99`). Esses valores, na verdade são faltantes (*missing*) para esta variável. Para o R, porém ele será computado como um número negativo, e se quisermos, por exemplo, saber o QI médio desses pacientes, esse valor vai alterar (e para baixo) o valor médio. Precisamos então "avisar" ao R que esses valores são *missing*. O R usa uma sigla `NA` (de *not available*) para designar valores *missing*.

Bem, você deve estar se perguntando como é que poderíamos substituir valores em uma variável no R... Esse procedimento envolve elementos que nós já discutimos, mas com uma combinação diferente. Vamos tentar:

```
> fem$IQ[fem$IQ == -99] <- NA
```

Explicando devagar: estamos substituindo (símbolo <-) os valores da variável `IQ` (o primeiro `fem$IQ`, que são iguais a `-99 (fem$IQ == -99)` por `NA`. O símbolo `==` é um operador de comparação e quer dizer "igual a".

Quer ver se funciona? Tente:

```
> fem$IQ
```

Explorando um Data Frame

Vamos agora comparar os valores de QI dos pacientes de acordo com 2 grupos: os que pensaram em suicídio e os que não pensaram. Para isso, lançaremos mão de uma função especial:

```
> by(data=fem$IQ, INDICES=fem$LIFE, FUN=summary)
```

Essa função é especial porque ela aplica uma outra função (no caso a função `summary()` – argumento `FUN`) a uma variável (argumento `data=`) estratificado por uma outra variável (argumento `INDICES`). Pegou? Repare qual é a saída da função `summary()`: ela calcula a média, mediana mínimo, máximo, primeiro e terceiro quartís, além de reportar o número de *missings* (os NA's).

Você deve ter notado que é muito trabalhoso digitar o nome do *data frame* toda vez que a gente quiser trabalhar com uma de suas variáveis. Para contornar esse problema existe uma função no R que permite acessar as variáveis diretamente. Vejamos

```
> attach(fem)
```

Para testar vamos repetir a função `by()` omitindo o nome do *data frame* e também o `$`.

```
> by(IQ, LIFE, summary)
```

Viu como somos preguiçosos? Nem o nome dos argumentos foram usados neste caso. Lembre-se que a omissão dos nomes dos argumentos pode causar uma enorme confusão caso você esqueça a ordem correta (lembra do caso do log?).

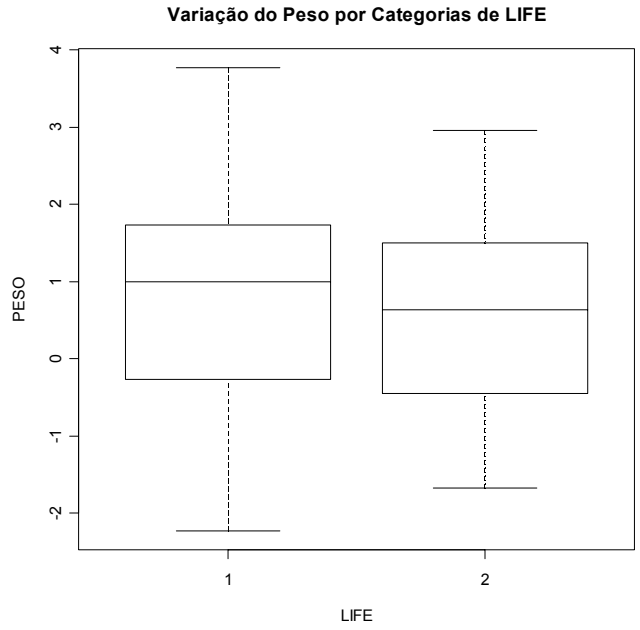
Vamos visualizar os dados com um gráfico que a gente já tinha visto anteriormente, que é o `boxplot()` com alguns argumentos avançados. Para isso escrevamos:

```
> boxplot(WEIGHT ~ LIFE)
```

Entendeu o que foi feito? O símbolo ~ (til) na função `boxplot()` funciona como a função `by()` que vimos anteriormente, ou seja, visualizamos o peso estratificado pela variável `LIFE`

Você pode adicionar alguns nomes aos gráficos para que ele fique mais apresentável

```
> boxplot(WEIGHT ~ LIFE, xlab="LIFE", ylab="PESO",
main="Variação do Peso por Categorias de LIFE")
```



Os grupos não parecem ser diferentes. Suas medianas e distribuições interquartilares são semelhantes. Para ter uma idéia se essas distribuições são aproximadamente normais podemos usar histogramas. Vamos tentar

```
> hist(WEIGHT[LIFE==1])
> hist(WEIGHT[LIFE==2])
```

Como você pode ver na tela, a variável WEIGHT parece ser normal para ambos os grupos.

Um teste muito usado para inferir diferenças entre médias de dois grupos é o teste *t de Student.* Este teste além de assumir normalidade nos dois grupos, pode ser plicado tanto para situações onde a variância dos grupos são homogêneas (homoscedásticas) ou não (heteroscedásticas). A normalidade já foi constatada. Falta agora saber como sas variâncias se comportam. Vamos aplicar o teste de *Bartlett*.

```
> bartlett.test(WEIGHT, LIFE)

        Bartlett test for homogeneity of variances

data:  WEIGHT and LIFE
Bartlett's K-squared = 0.3241, df = 1, p-value =
0.5692
```

A hipótese nula desde teste é que as variâncias são homogêneas. Com o p-valor de 0.5692 não podemos rejeitar esta hipótese nula, e concluímos que as variâncias são de fato homogêneas. Passemos então para o teste t:

```
> t.test(WEIGHT~LIFE,var.equal=TRUE)

        Two Sample t-test

data:  WEIGHT by LIFE
t = 0.5987, df = 104, p-value = 0.5507
alternative hypothesis: true difference in means is
not equal to 0
95 percent confidence interval:
 -0.3382365  0.6307902
sample estimates:
mean in group 1 mean in group 2
      0.7867213       0.6404444
```

Como esperado após a inspeção do gráfico *boxplot*, de fato não existe uma diferença significativa do nível de variação dos pesos estratificados pela variável LIFE

Módulo Entrada e Saída de Dados

Pré-requisitos: Saber como funcionam e como usar funções e pacotes no R. Saber selecionar elementos, linhas e colunas em *dataframes*.

Entrada de Dados

O R é capaz de ler arquivos de dados salvos em diversos formatos diferentes, como ASCII (arquivo texto, delimitado por espaço, tabulação, vírgula, ponto-e-vírgula, entre outros), Excel, SPSS, Epiinfo, etc. No caso do Excel, como veremos adiante, o processo de importação não é direto, tendo que se salvar inicialmente em um formato ".csv".

Vamos iniciar com o mesmíssimo exemplo que usamos no módulo básico do R; portanto, se você já fez esse módulo, provavelmente vai querer pular os próximos parágrafos...

A função mais simples que lê dados externos no R faz parte do pacote básico e se chama `read.table()`. Essa função vai importar dados em formato ASCII para um objeto do tipo *dataframe*. Vamos ver um exemplo:

```
dados <- read.table(file="iris.txt", header=T,
sep="\t")
```

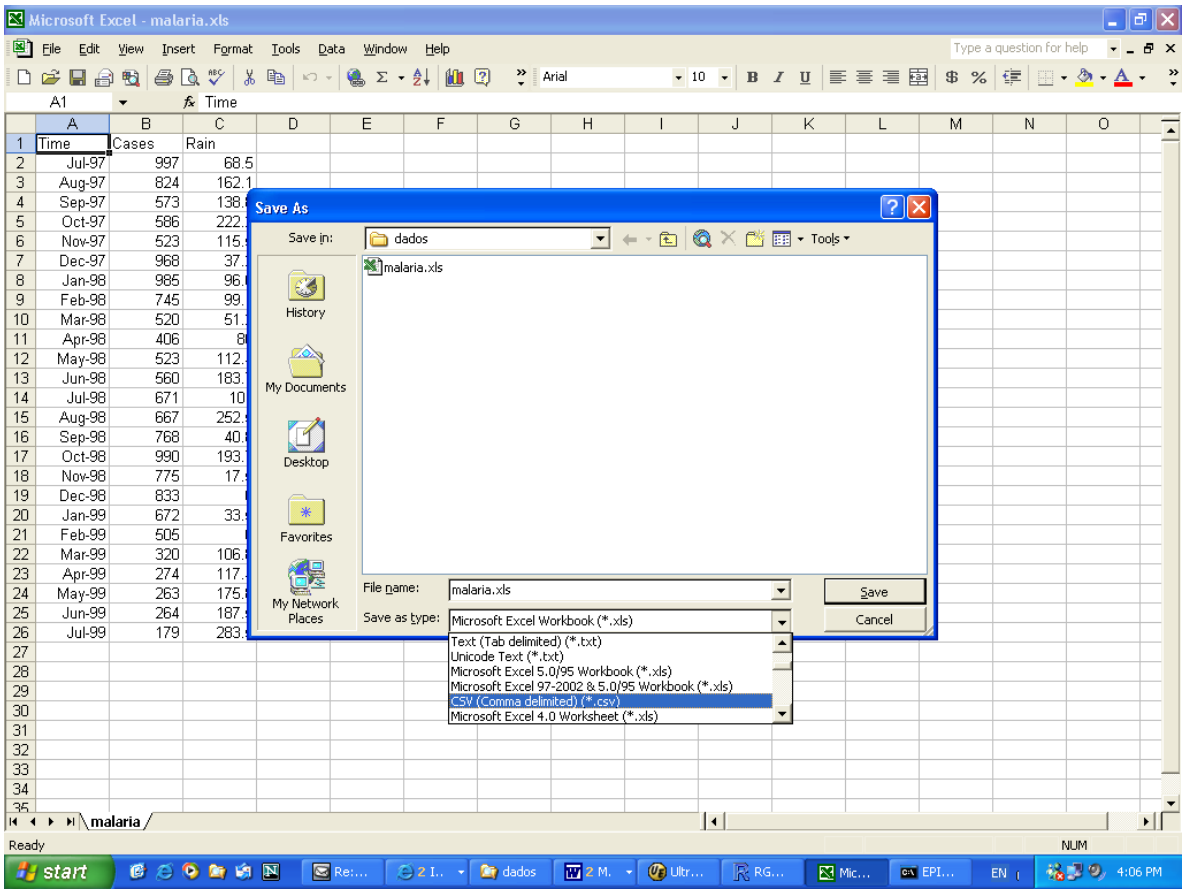
Vamos ver por partes o que nós fizemos: criamos um objeto `dados`, através do *assignment* `<-` pela função `read.table()`, que tem um argumento obrigatório (`file`) e dois argumentos opcionais (`header=T` e `sep="\t"`). Bom, `file` é simplesmente o nome do arquivo externo onde os dados estão armazenados, com a extensão inclusive e entre aspas. O argumento `header=T` é para indicar para o R que a primeira linha contém o nome das variáveis nesse banco. O último argumento que usamos, `sep="\t"` é usado para indicar que a delimitação dos dados nesse caso é feita por tabulação. Uma importante observação é que se o separador fosse tabulação e não espaço, a função leria os dados da mesma forma, apenas mudando-se para `sep=" "`. Que tal dar uma olhadinha na ajuda dessa função? Boa sorte...

Bom, a primeira providência após importar os dados é dar uma olhadinha neles para ver se tudo funcionou a contento. Se a gente quisesse ver esse banco inteiro, bastaria digitar o nome do objeto `dados` (como você deve se lembrar que acontece com qualquer objeto). Como esse banco tem 150 registros e 5 variáveis, que tal a gente ver somente os 10 primeiros registros? Vamos tentar:

```
> dados[1:10,]
   slength swidth plength pwidth species
1      5.1    3.5     1.4    0.2  setosa
2      4.9    3.0     1.4    0.2  setosa
3      4.7    3.2     1.3    0.2  setosa
4      4.6    3.1     1.5    0.2  setosa
5      5.0    3.6     1.4    0.2  setosa
6      5.4    3.9     1.7    0.4  setosa
7      4.6    3.4     1.4    0.3  setosa
8      5.0    3.4     1.5    0.2  setosa
9      4.4    2.9     1.4    0.2  setosa
10     4.9    3.1     1.5    0.1  setosa
```

Agora vamos passar para o caso de um arquivo em Excel, que você gostaria que fosse lido pelo R. Como foi mencionado anteriormente, o R não é capaz de ler um aquivo em formato .xls diretamente; é preciso salvar o arquivo antes em um formato ASCII. Vamos ver como funciona?

No diretório "c:\cursoR" você vai encontrar um arquivo chamado "malaria.xls". Abra esse arquivo no Excel, depois pressione a tecla "F12" no teclado, o que abrirá o menu "Salvar Como". Na parte inferior do menu na opção "salvar arquivo do tipo", selecione "CSV". Na figura abaixo, você verá um exemplo numa versão em inglês do Excel:



Feito isso, esse nosso diretório terá um arquivo chamado “malaria.csv”, que é um arquivo ASCII delimitado por... bem, aqui existe um problema grave: se o seu Excel for em português, provavelmente o delimitador será um ponto-e-vírgula; se for em português. Isso vai fazer diferença porque é preciso especificar o tipo de separador na função `read.table()`.

```
> malaria <-
read.table(file="c:\\curso\\dados\\malaria.csv", header=T,
sep=”,”)
```

Neste caso existem basicamente duas diferenças em relação à leitura de dados feita anteriormente. A primeira delas está no argumento `file=`. Como o arquivo malária não se encontra no diretório de trabalho precisamos especificar o caminho completo do arquivo. Se você já escreveu alguma vez um caminho completo no DOS ou no Windows, deve ter estranhado o uso de duas barras invertidas em vez de uma só. Não se preocupe, o R usa uma sintaxe semelhante ao sistema UNIX. A segunda diferença se encontra na especificação do delimitador, que no caso do Excel em inglês será uma vírgula e no em português será um ponto-e-vírgula.

Vamos agora passar para os casos onde a leitura pode ser feita diretamente de um formato que não seja ASCII. Nesta situação é necessário carregar um pacote específico para importação e exportação de dados chamado `foreign`. No R todas as funções são agrupadas em pacotes específicos. Ao inicializarmos o R automaticamente dois pacotes são carregados: `base` e `ctest` onde se encontram todas as funções até agora utilizadas. Para utilizar funções de outros pacotes precisamos carregá-los através da função `library()`.Para isto digite:

```
> library(foreign)
```

Vamos começar importando dados armazenados em formato “.sav”, que é a extensão de banco de dados do SPSS. No diretório “c:\curso\dados” você vai achar um arquivo chamado “cellular.sav”, que é um arquivo de exemplo que vem com o SPSS. Vamos ler esse arquivo no R:

```
> cellular <- read.spss(file=
"c:\\cursoR\\cellular.sav", to.data.frame=T)
```

A única informação nova aqui é o argumento `to.data.frame=T`, que indica para o R que o formato a ser lido é um *data frame*. Quer saber mais detalhes? Use a ajuda... Ah, e não se esqueça de verificar o arquivo...

Vamos agora ver um outro formato que é bastante usado em saúde pública, que é o “.rec”, extensão usada por bancos armazenados pelo EpiInfo. Se você já usou o EpiInfo, deve conhecer o famosíssimo arquivo “oswego.rec”, usado em muitos exercícios em epidemiologia. Nesse caso vamos acessar o arquivo diretamente do diretório do próprio EpiInfo:

```
> oswego <- read.epiinfo("c:\\epi6\\oswego.rec")
```

Se você tem familiaridade com o EpiInfo, deve se lembrar que antigamente as datas eram armazenadas com os anos em 2 dígitos. No caso desse arquivo, na verdade, o ano foi omitido, já que tudo se passou em apenas dois dias. O problema é que nesses formatos, o R não vai entender esse campo como data, mas como texto. Vamos tentar o seguinte:

```
> oswego$ONSETDATE
```

Note que a saída são as "datas" entre aspas, caracterizando uma variável tipo caracter. Duvida? Então faça:

```
> is.character(oswego$ONSETDATE)
```

A resposta foi TRUE, como esperado. Para contornar esse problema, temos que lançar mão de alguns argumentos extras. Vamos tentar o seguinte:

```
> oswego1 <- read.epiinfo("c:\\epi6\\oswego.rec",
guess.broken.dates=TRUE, thisyear="1972")
```

Bem, você deve ter entendido os dois arguemtos extras: o primeiro pede para o R tentar acertar o ano da data que tenha um ano com 2 dígitos ou que não tenha ano algum (o nosso caso aqui). O segundo é para o R colocar um ano quando ele não estiver especificado. Como nós sabemos que esse surto ocorreu em 1972, esse foi o nosso argumento. Veja que agora temos datas de fato:

```
> oswego1[1:10,1:5]
```

Além desses formatos, o pacote `foreign` também possui funções para ler arquivos em outros formatos como S-Plus, SAS, Minitab e Stata. Fique à vontade para explorar essas funções se for do seu interesse...

Saida de Dados

Se você foi curioso bastante para explorar um pouco mais o pacote `foreign`, notou que ele só possui na verdade uma função para exportar diretamente dados para um formato específico, que é o Stata. Mas o R é capaz de exportar também para o formato ASCII, que pode ser considerado como um formato universal, ou seja, qualquer programa é capaz de ler um arquivo nesse formato. Nessa seção, vamos ver uma função que faz essa tarefa, usando diferentes argumentos para a exportação, convenientes para o programa que vai ler os dados.

O caso mais geral é exportar dados num formato ASCII delimitado por espaço ou tabulação. Vamos começar com a separação por tabulação e mostrar como esse arquivo pode ser facilmente lido pelo SPSS. Vamos usar o mesmo arquivo que importamos anteriormente "iris.txt" e que guardamos no objeto `dados`:

```
> write.table(dados, file="iris.dat", sep="\t",
row.names=F, C)
```

Os 3 primeiros argumentos da função `write.table()` não devem suscitar dúvidas: o primeiro é o nome do objeto a ser exportado, o segundo, o nome do arquivo onde o dado será armazenado e o terceiro o tipo de delimitador a ser usado. O argumento `row.names=F` serve para indicar que as linhas desse objeto não têm nome e ainda previne o R de colocar números como nomes (por default o R vai fazer uma numeração crescente, como se fosse o número do registro, se esse argumento for `T`). O argumento `row.names=F` serve para indicar ao R que variáveis tipo caracter devem ser exportadas sem estar entre aspas (porque se estiverem o SPSS não vai levar isso em conta e vai importar as aspas junto).

O arquivo gerado "iris.dat" pode ser facilmente importado para o SPSS. A técnica de importação para o SPSS não faz parte do escopo desse módulo e deverá ser dominado por pessoas que trabalham com o SPSS de uma forma regular.

O outro formato que vamos ver nessa seção é o separado por vírguas, ou ".csv", um formato que é lido diretamente pelo Excel. A idéia é fundamentalmente a mesma, só diferindo o delimitador:

```
> write.table(dados, file="iris.csv", row.names=F,
sep=",", quote=F)
```

O arquivo "iris.csv" pode ser aberto diretamente no Excel, simplesmente dando um duplo clique sobre o arquivo.

Saída de Resultados

Uma outra necessidade é a saída de resultados de funções usadas no R. Essas saídas são basicamente de dois tipos: Ou um gráfico ou uma saída de texto, que pode conter uma "tabela" ou resultados na forma de texto livre mesmo. Se você já tem alguma familiaridade com o R, deve saber que sua parte gáfica é bastante superior à sua parte de saída tipo tabela. Na verdade essas saídas só podem ser coladas como texto e não têm a mesma facilidade de manuseio como as tabelas do SPSS por exemplo.

Vamos então usar o mesmo exemplo visto no final do módulo básico com o nosso banco `fem` para mostrar como copiar e colar a saída de uma função e também um gráfico do R, no Word.

Você deve estar se lembrando da função inicialmente utilizada para comparar os valores de QI dos pacientes que pensaram ou não em suicídio, a função `by`:

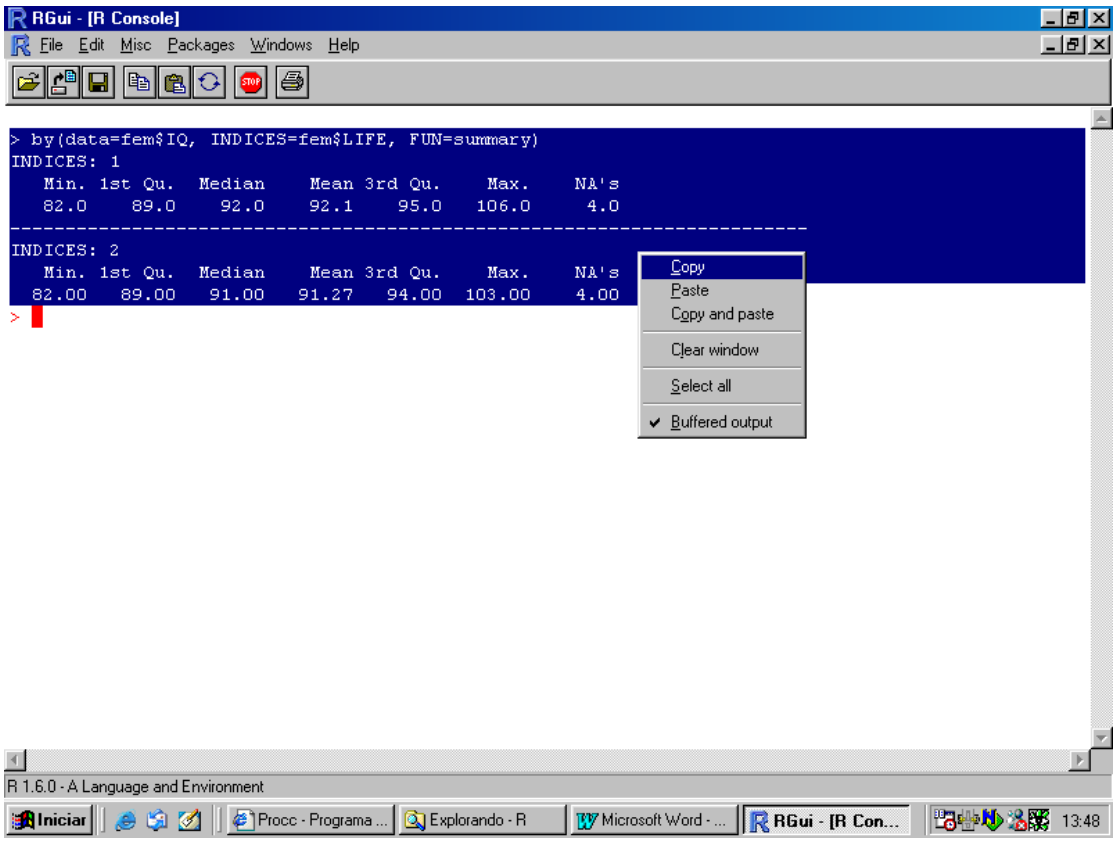
```
> by(data=fem$IQ, INDICES=fem$LIFE, FUN=summary)
```

Temos aqui uma saída do tipo texto.

O primeiro passo para copiar e colar a saída dessa função é selecionar no R a saída ou parte da saída que seja de interesse. Para isso apertamos o botão esquerdo do

mouse e o arrastamos sobre o texto. O tom azul sobre a tela indica a parte que está sendo selecionada.

O próximo passo é copiar a parte do texto selecionada no R. Isso pode ser feito selecionando no menu “Edit” a opção “Copy”, ou simplesmente apertando sucessivamente a tecla “Ctrl” e a tecla “c”. Outra possibilidade ainda é após selecionar o texto no R clicar com o botão direito do mouse e escolher “Copy”:



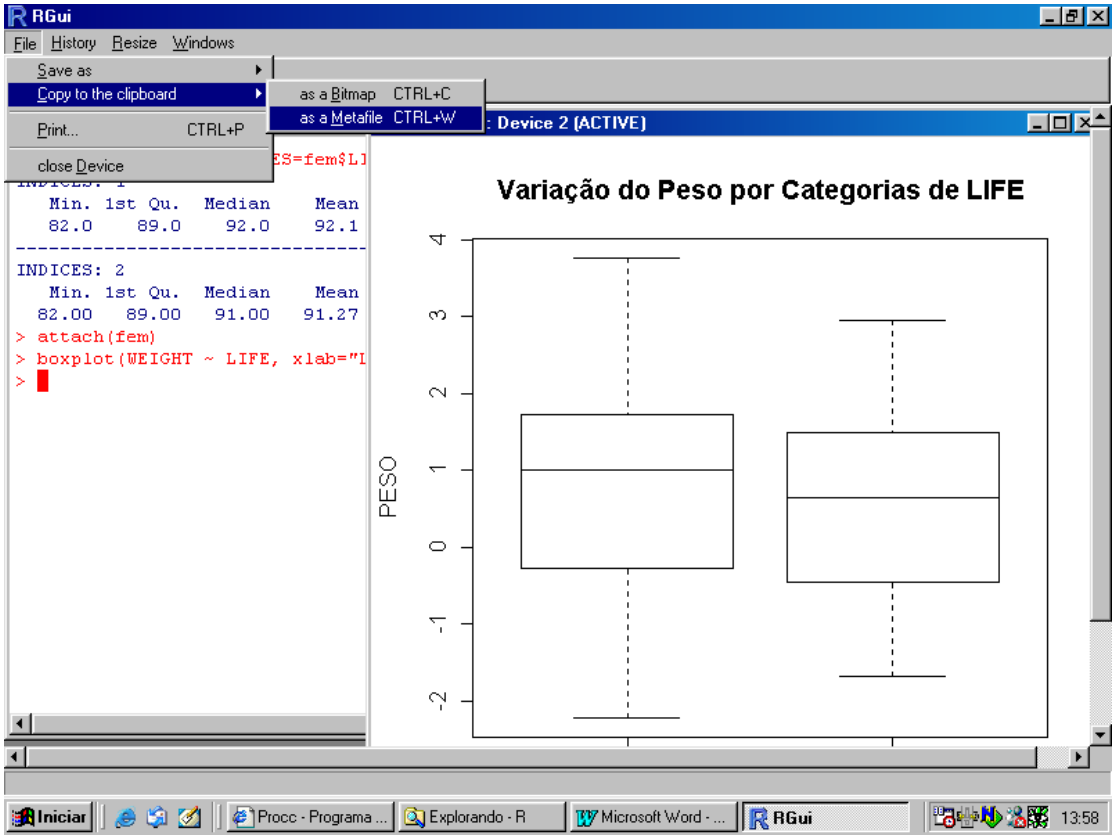
Agora, é só colar o conteúdo no Word ou programa equivalente. No Word a forma de se fazer isso é a mesma utilizada acima no processo de copiar os dados do R bastando trocar a opção “Copy” pela opção “Colar” (ou “Paste”). Para obter uma formatação no Word igual ao que aparece no R troque a fonte do texto colado para “courier new”.

No caso da saída de uma função ser um gráfico, o procedimento de copiar e colar é outro. Para visualizar a diferença dos valores de QI dos pacientes que pensaram ou não em suicídio havíamos construido um boxplot:

```
> attach(fem)
> boxplot(WEIGHT ~ LIFE, xlab="LIFE", ylab="PESO",
main="Variação do Peso por Categorias de LIFE")
```

Um arquivo gráfico pode ser copiado pelo R através de de dois formatos: Bitmap ou Metafile. Esses formatos se diferem entre si por vantagens e desvantagens que cada um possui. Uma dentre outras vantagens do formato Metafile é o fato dele gerar arquivos de tamanho menor.

Para copiar um arquivo no formato Metafile escolha no menu “File” dentro de “Copy to the Clipboard” a opção “as Metafile”:



Depois é só colar no Word. Colando no formato Metafile podemos editar a figura através de um duplo clique sobre a figura permitindo alterar o título do gráfico, valores que aparecem nos eixos, etc.

Uma sugestão: você pode querer que seu gráfico não fique flutuando sobre o texto, facilitando na hora de escrever. Para isso, escolha no menu “Editar” dentro da opção “Colar especial” no formato desejado, desmarcando a opção “Flutuar sobre o texto”.