

# R aplicado à suinocultura

Marcio Valk

2020-07-26



# Contents

		<b>5</b>
Objetivo . . . . .		5
Sobre o autor . . . . .		5
<b>1 Tutorial básico R e RStudio</b>		<b>9</b>
1.1 Apresentação da linguagem R . . . . .		9
1.2 Instalando o R . . . . .		10
1.3 Instalando o RStudio . . . . .		10
1.4 Diretório de trabalho . . . . .		10
1.5 RStudio Cloud . . . . .		11
1.6 Instalação de pacotes . . . . .		11
1.7 Ajuda . . . . .		15
1.8 Funcionalidades básicas . . . . .		17
1.9 Estrutura dos dados . . . . .		22
1.10 Programação básica . . . . .		35
1.11 Funções . . . . .		36
1.12 Operador Pipe . . . . .		37
<b>2 Importando dados</b>		<b>39</b>
2.1 Dados de diferentes formatos . . . . .		39
2.2 Data Frames . . . . .		44
2.3 Análise exploratória . . . . .		45
2.4 Gráficos simples ( <code>scatter plot</code> ) . . . . .		48
2.5 Análise estatística de dados - Modelagem . . . . .		56
<b>3 Visualização de dados com o R</b>		<b>61</b>
3.1 Gráficos com o <code>ggplot2</code> . . . . .		62
3.2 Funções geom e aes . . . . .		62
3.3 Segmentação e junção de gráficos do <code>ggplot</code> . . . . .		67
3.4 Diferentes tipos de gráficos . . . . .		70

<b>4 Análise de delineamento de experimentos</b>	<b>77</b>
4.1 Delineamento inteiramente casualizados . . . . .	77
4.2 Delineamento em blocos ao acaso . . . . .	77
4.3 Fatorial . . . . .	77
4.4 Parcada subdividida . . . . .	77
<b>5 Relatórios e artigos científicos com o rmarkdown</b>	<b>79</b>
5.1 Relatório dinâmico com o Rmarkdown . . . . .	79
5.2 Relatórios em word, pdf ou html . . . . .	79
5.3 Apresentações em Power Point . . . . .	79

## R Aplicado à Suinocultura

Marcio Valk



Figure 1: book cover



## Objetivo

A era da *ciência de dados* tem mudado a maneira como fazemos as coisas. A integração entre as áreas é necessária para acompanhar a dinâmica do desenvolvimento de novas tecnologias. Não é diferente na área da suinocultura, onde a realização de pesquisas têm demandado cada vez mais por ferramentas e profissionais com essa característica de *interdisciplinar*.

O R tem se caracterizado por ser uma ferramenta completa para quem trabalha com pesquisa, seja aplicada ou teórica. As diversas áreas do conhecimento acabam convergindo para o R, por se tratar de uma linguagem moderna, dinâmica, colaborativa e integradora.

A pesquisa e o desenvolvimento tecnológico na área da suinocultura geram um volume considerável de informações que necessitam ser analisadas. O R é uma ferramenta que permite, ler, integrar e tratar grandes bancos de dados, é bem desenvolvido na área de visualização de dados e possui uma diversidade de métodos estatísticos implementados. Além disso, ferramentas para gerar relatórios, apresentações e até mesmo a criação de aplicativos que possibilitam a interação do usuário final, tornam o R uma ferramenta completa.

## Sobre o autor

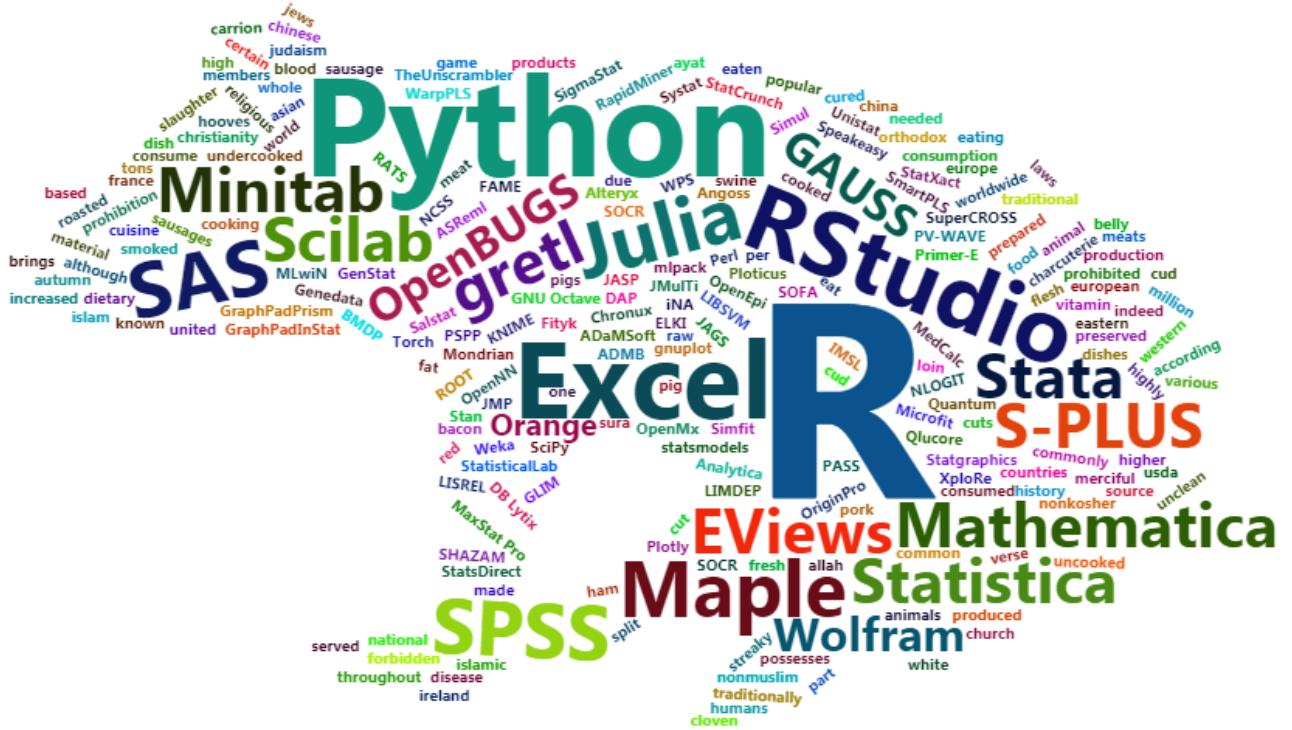
Meu nome é Marcio, sou professor do Departamento de Estatística da UFRGS e sou um grande fã do R. Encarei esse desafio pois gosto de pensar que outras áreas, além da Estatística, possam usufruir das fantásticas ferramentas que essa linguagem oferece. Além disso, contribuindo para o desenvolvimento tecnológico da área da suinocultura , estaremos contribuindo para o desenvolvimento do país.

## R Aplicado à Suinocultura

Marcio Valk



Figure 2: book cover





# Chapter 1

# Tutorial básico R e RStudio

Existe uma ampla variedade de softwares relacionados à estatística e análise de dados. Uma lista generosa pode ser encontrada em List of statistical software. A seguinte ilustração apresenta os mais populares na visão deste autor.

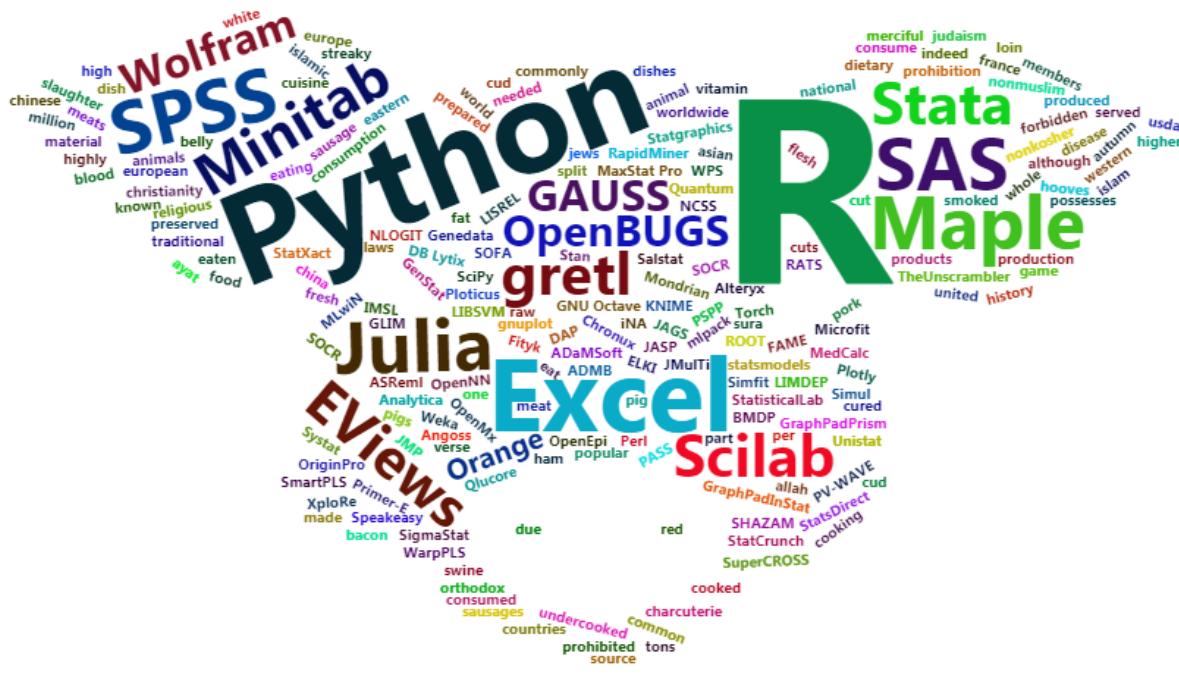


Figure 1.1: Lista de softwares mais populares na análise estatística na visão do autor

## 1.1 Apresentação da linguagem R

R é uma linguagem de programação caracterizada como Software Livre sob os termos da *General Public License (GNU)* da *Free Software Foundation* no formato *open source*. É voltada a manipulação, análise e vizualização de dados e tem como característica o aspecto colaborativo, sendo que as ferramentas desenvolvidas são compartilhadas online pelos desenvolvedores, podendo ter acesso a elas qualquer pessoa, sem restrições. Uma breve história do R pode ser encontrada no wikipedia.

## 1.2 Instalando o R

Para instalar no computador, o R deve ser baixado do CRAN.

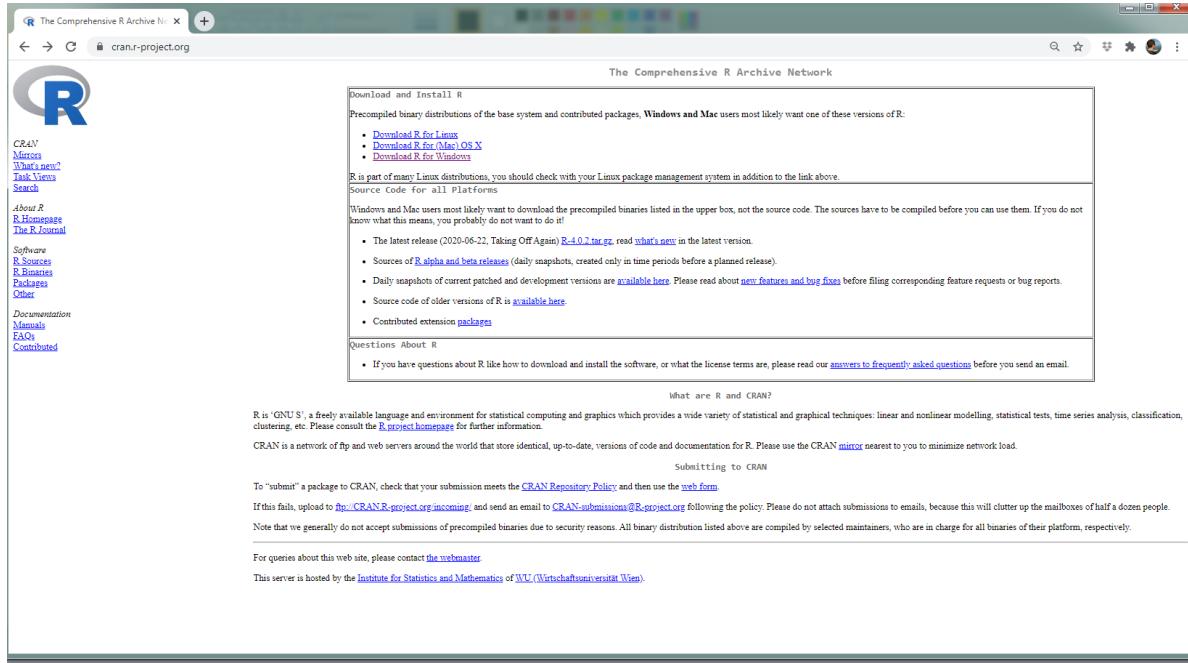


Figure 1.2: Comprehensive R archive network (CRAN)

Se o sistema operacional for Linux, uma versão *base* do R já vem instalada. No caso de outros sistemas operacionais, como o Windows, é necessário instalar o R base.

## 1.3 Instalando o RStudio

Como quase toda linguagem *Open Source* a utilização se dá por meio de linhas de comando. Para tornar a linguagem mais amigável aos usuários, várias IDEs (*integrated development environment*) são utilizadas. No caso do R, a mais desenvolvida e utilizada é o RStudio.

Uma versão *Free* do RStudio para o seu desktop pode ser baixado de <https://rstudio.com/products/rstudio/download/>.

Depois de instalar o R, o RStudio já estará integrado ao R e terá uma interface intuitiva e amigável ao usuário.

*Ainda assim, é importante ressaltar que na linguagem R não encontraremos botões para realizar as análises.*

## 1.4 Diretório de trabalho

Um ação importante que deve ser realizada pelo usuário é *setar o diretório* de trabalho. Para isso existem diferentes formas. Uma delas é usando a função `setwd()`.

```
setwd("~/R aplicado a suinocultura 2")
```

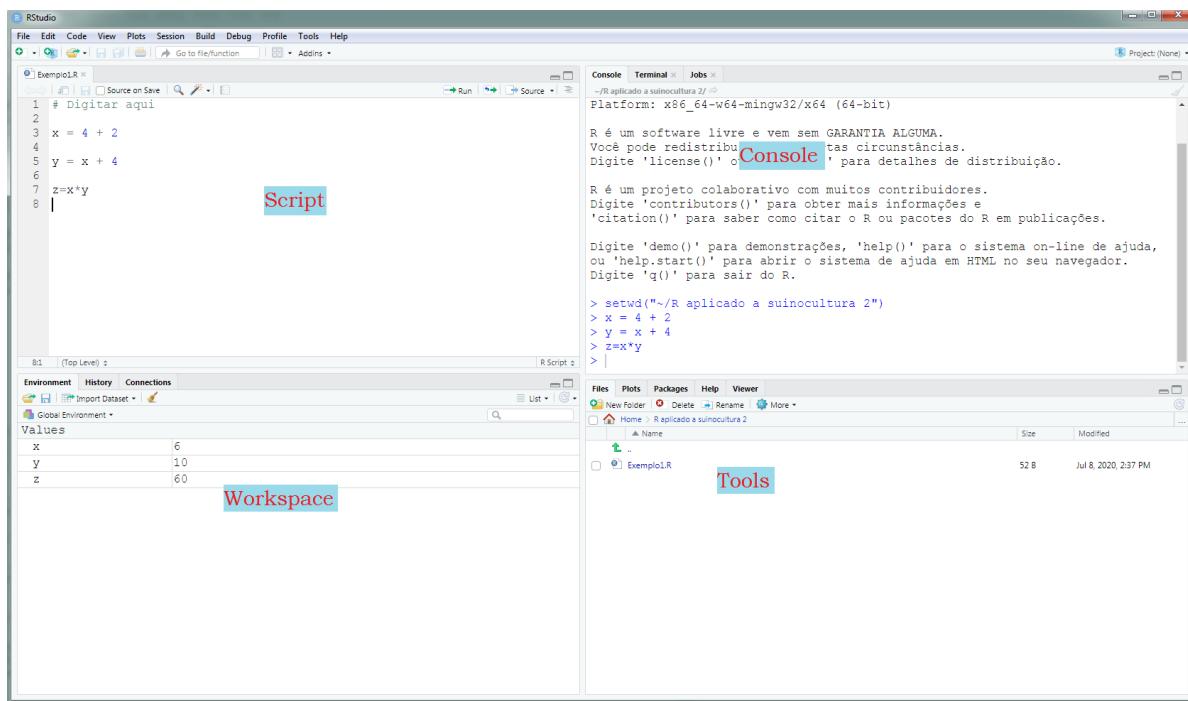


Figure 1.3: Interface do RStudio

Outra opção é através do *Go to directory* que está disponível no Workspace do RStudio, conforme figura ???. Nessa opção o usuário escolhe o diretório de trabalho e depois usando a opção *set as working directory* esse diretório será “*settado*” como diretório de trabalho.

*Todas os arquivos gerados, como figuras serão salvos nesse diretório. Para abrir um conjunto de dados, por exemplo, será muito mais simples se este também estiver salvo no mesmo diretório ou um subdiretório. Além disso, toda vez que o RStudio for reinicializado, esse procedimento terá que ser refeito.*

## 1.5 RStudio Cloud

Outra forma simples e prática para usar o R e o RStudio é usar o RStudio Cloud.

*THE MISSION We created RStudio Cloud to make it easy for professionals, hobbyists, trainers, teachers and students to do, share, teach and learn data science. (RStudio-Cloud 2020)*

Na nuvem é possível usar o RStudio utilizando um login através da conta google, ou criar gratuitamente uma conta. Uma vez logado, escolhendo a opção *project* o usuário terá uma versão do RStudio perfeitamente funcional, que pode ser utilizada até no smartphone. Obviamente, é necessário ter conexão com a internet para que a ferramenta possa ser utilizada.

## 1.6 Instalação de pacotes

Na versão *base* do R, uma série de ferramentas, funções e métodos estatísticos são disponibilizados. Além disso, alguns pacotes também compõe a versão *base* do R. Depois de instalado, o usuário pode verificar quais pacotes estão instalados acessando o ícone *Packages* em *Tools* ou digitando *installed.packages()* no *console*.

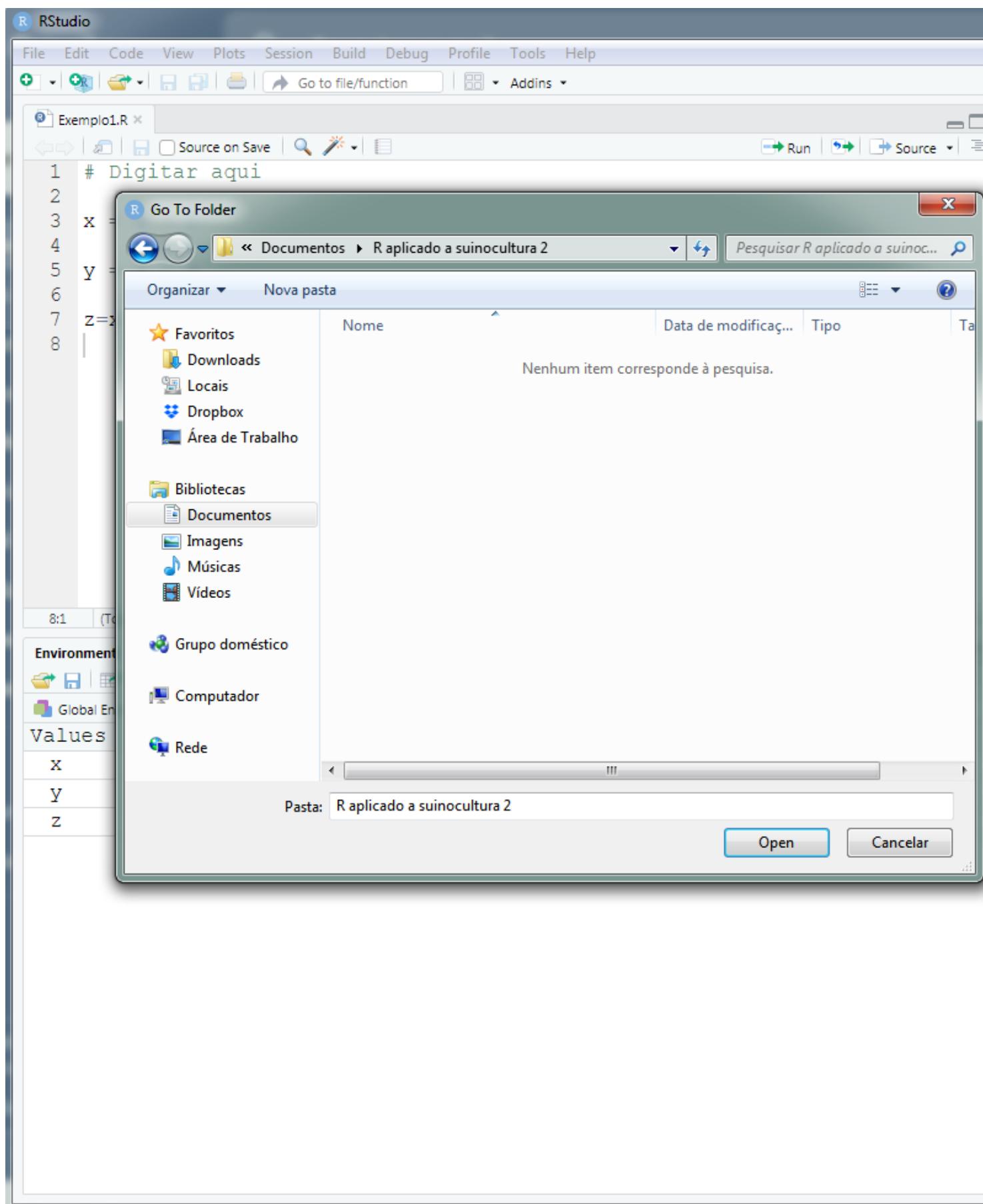


Figure 1.4: Diretório de trabalho

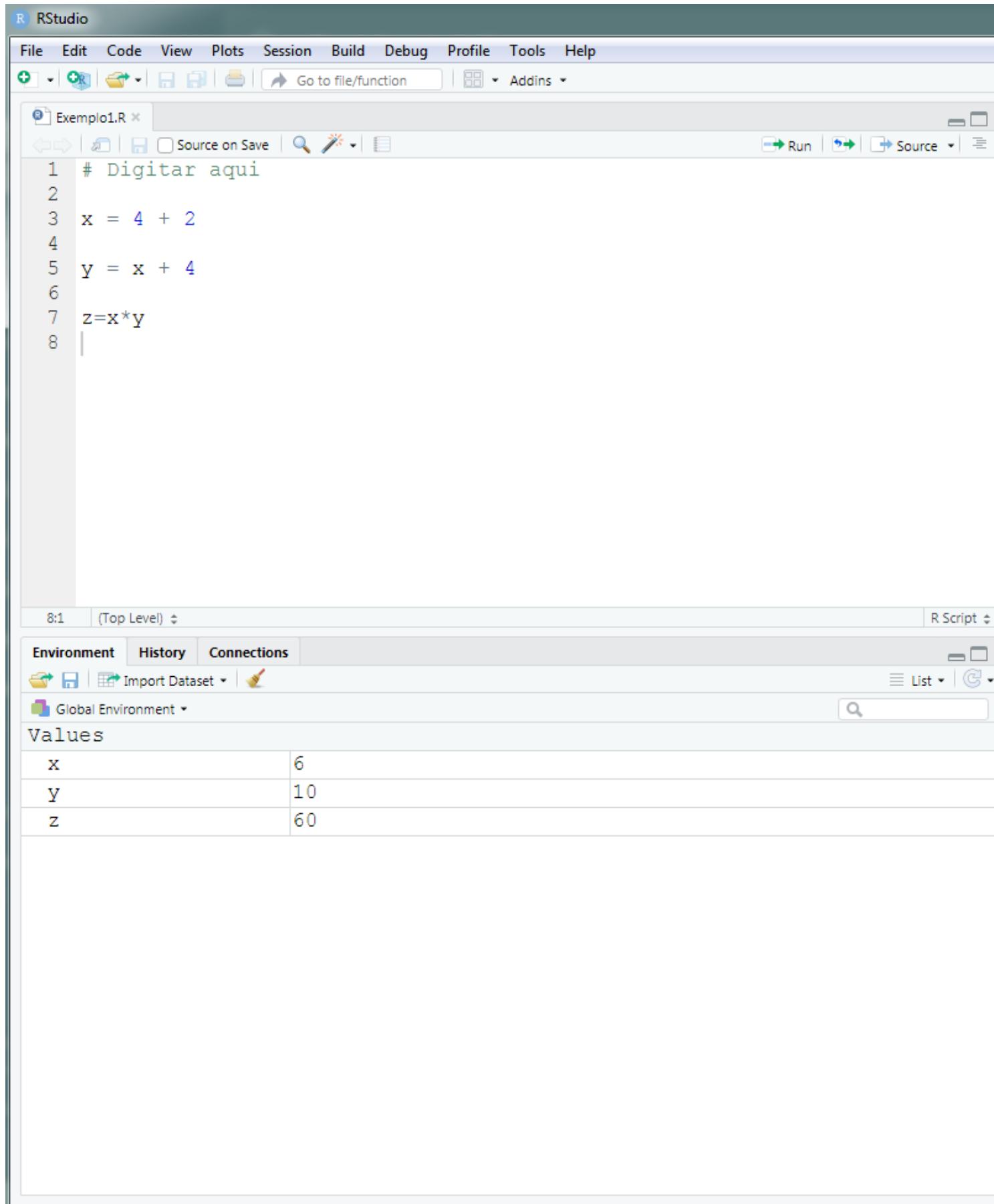
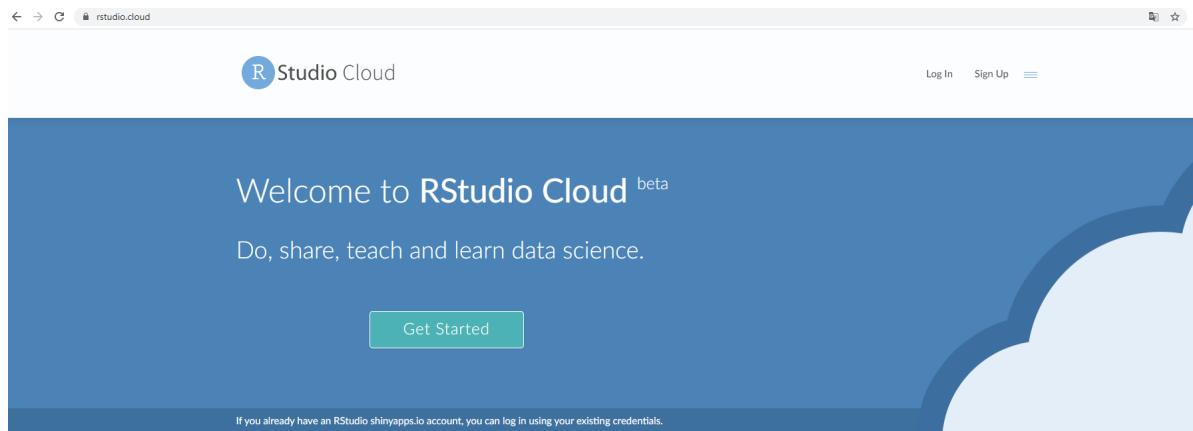


Figure 1.5: Diretório de trabalho



### THE MISSION

We created RStudio Cloud to make it easy for professionals, hobbyists, trainers, teachers and students to do, share, teach and learn data science.



Figure 1.6: Rstudio Cloud

Figure 1.7: Rstudio Cloud

Para instalar um novo pacote, o usuário pode acessar o ícone *Packages* e depois clicar em *Install* ou digitar no console `install.packages("nome do pacote")`. Como exemplo, podemos instalar o pacote usado para vizualização de dados chamado `ggplot2`.

```
install.packages("ggplot2")
```

### 1.6.1 Carregando pacotes

Importante para usuários iniciantes na linguagem R é entender a diferença entre instalar pacotes e carregar pacotes. Uma vez instalado, o pacote estará à disposição do usuário sempre que ele precisar, mas é necessário carregá-lo. É comum deixar um comando nos *scripts* para que cada vez que seja necessário usar alguma função específica de um pacote, ele primeiro seja carregado. O comando usado é o `library()` mas pode ser feito acessando o espaço que chamando de **Tools**, clicar em *Packages* e marcar o pacote desejado para carregá-lo.

```
library(ggplot2)
```

## 1.7 Ajuda

Para um usuário iniciante no R é fundamental saber como resolver problemas diversos que certamente vão surgir durante a instalação de um pacote, uso de uma função, criação de um gráfico, manipulação de dados, etc. Muitas coisas no R são feitas por tentativa e erro, mas o conhecimento é acumulativo e problemas similares, poderão ter uma solução mais rápida. Uma das grandes vantagens da utilização do R é que a comunidade é bastante ativa. Existem diferentes formas de conseguir ajuda e vou elencá-las em ordem de importância, segundo a forma utilização desse autor:

- Google
- Stack Overflow
- Help do R (`help()` ou `?`)

### 1.7.1 Google

Para um iniciante em R coisas simples como calcular a raíz quadrada de um número pode ser difícil. Nesses casos o google é muito útil.

Um pouco de conhecimento de inglês aumenta consideravelmente as opções de ajuda no google.

### 1.7.2 Stack Overflow

Quando o problema parecer um pouco mais complexo, uma opção é colocar a pergunta no google junto com *Stack Overflow*. Isso provavelmente direcionará o usuário para Stack Overflow em Português ou Stack Overflow que são sites de Pergunta e Resposta utilizados por todas as linguagens de programação.

### 1.7.3 Help do R

Se o usuário já sabe qual função deve ser usada, então a documentação do R é bem útil.

```
?sqrt  
help(sqrt)
```

Dicas para uso do Help

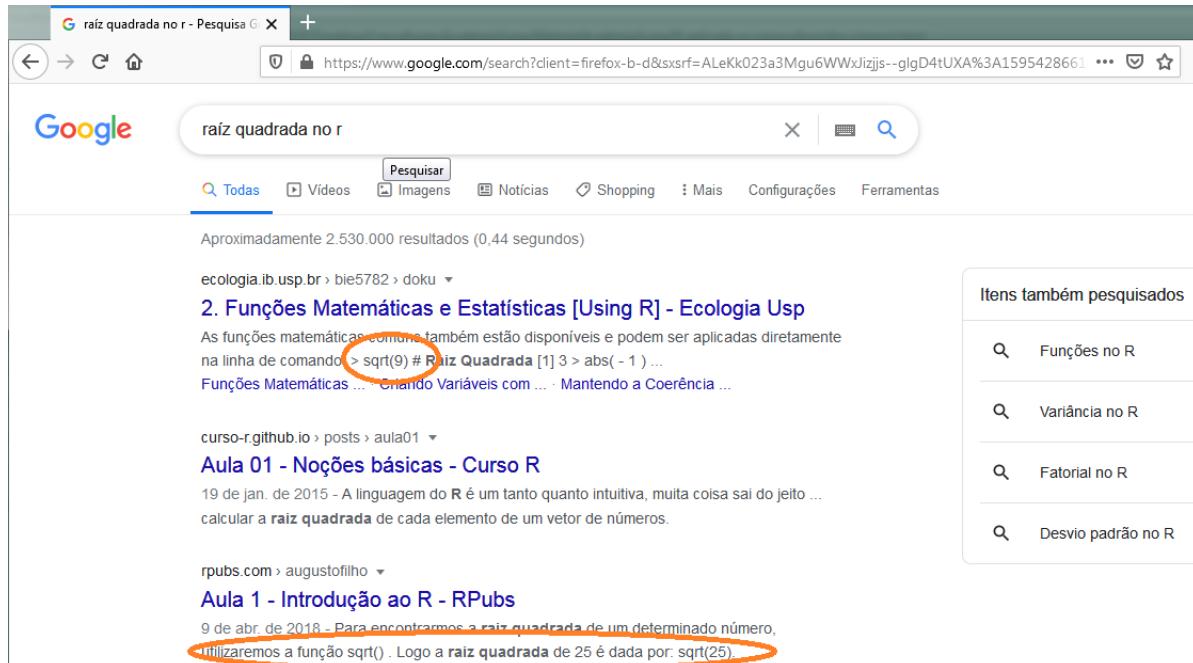


Figure 1.8: Rstudio Cloud

```
> sqrt("2")
Error in sqrt("2") : non-numeric argument to mathematical function
> |
```

Figure 1.9: Rstudio Cloud

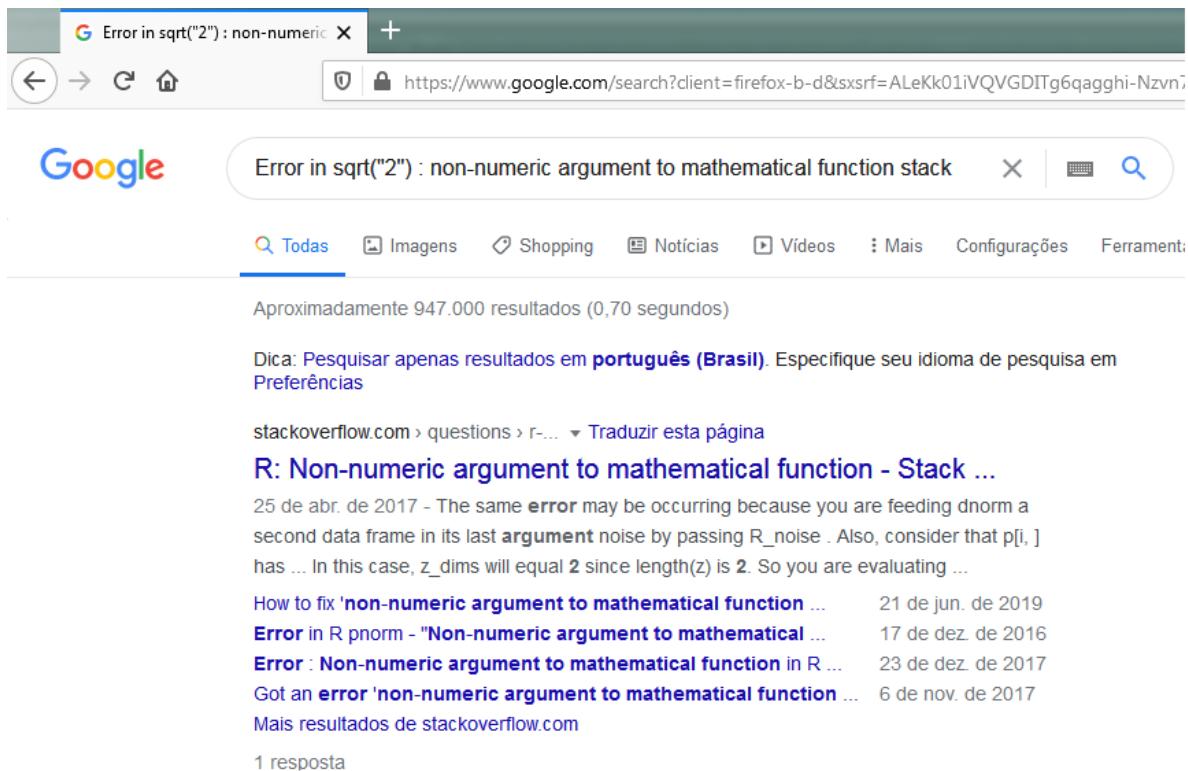


Figure 1.10: Rstudio Cloud

- Pode-se ir direto aos exemplos que estão no final;
- Identificar os parâmetros (*Arguments*);
- Funções relacionadas podem ajudar, dependendo da necessida.
- *Vignettes*, que são tutorias mais completos, mas somente alguns pacotes possuem. Esses textos podem ser acessados com a função vignette(package = 'nomeDoPacote'). Por exemplo, vignette(package = 'ggplot2')

## 1.8 Funcionalidades básicas

Para maior agilidade é importante que o usuário conheça algumas teclas de atalho. Apertando simultaneamente ***Alt + Shift + K*** o usuário tem acesso à uma grande quantidade de atalhos. Ou clicado em *Tools* no menu de ícones do RStudio. Um comando de atalho que destacaria por ser extremamente útil é o ***Ctrl + Enter***. Esse comando executa a linha do *script* em que o cursor está.

Para começar, usaremos o R como uma calculadora simples. Execute o código a seguir diretamente do console do RStudio ou no RStudio, escrevendo-os em um script e executando-os usando ***Ctrl + Enter***.

### 1.8.1 Operações básicas

#### 1.8.1.1 As quatro operações

Operação	código R	Resultado
$3 + 2$	<code>3 + 2</code>	5
$3 - 2$	<code>3 - 2</code>	1
$3 \cdot 2$	<code>3 * 2</code>	6
$3 / 2$	<code>3 / 2</code>	1.5

```
x <- 10
y <- 2
```

Operação	código R	Resultado
$3^2$	<code>3 ^ 2</code>	9
$2^{(-3)}$	<code>2 ^ (-3)</code>	0.125
$100^{1/2}$	<code>100 ^ (1 / 2)</code>	10
$\sqrt{100}$	<code>sqrt(100)</code>	10

```
x^y      # x=10 e y=2
```

```
## [1] 100
```

### 1.8.1.3 Logarítmicos

Observe que não existe `ln()` no R. Usa-se `log()` para significar logaritmo natural. Para as demais bases, é necessário especificar a base desejada.

Operação	código R	Resultado
$\log(e)$	<code>log(exp(1))</code>	1
$\log_{10}(100)$	<code>log10(100)</code>	2
$\log_2(16)$	<code>log2(16)</code>	4
$\log_4(16)$	<code>log(16, base = 4)</code>	2

```
log(x)      # x=10
```

```
## [1] 2.302585
```

```
log(x,base=y) # y=2
```

```
## [1] 3.321928
```

### 1.8.1.4 Constantes matemáticas

Constante	código R	Resultado
$\pi$	<code>pi</code>	3.1415927
$e$	<code>exp(1)</code>	2.7182818

```
log(exp(1))
```

```
## [1] 1
```

```
exp(1)^y      # y=2
```

```
## [1] 7.389056
```

## 1.8.2 Operadores lógicos

Operador	Significado	Exemplo	Resultado
<code>x &lt; y</code>	x menor do que y	<code>3 &lt; 42</code>	TRUE
<code>x &gt; y</code>	x maior do que y	<code>3 &gt; 42</code>	FALSE
<code>x &lt;= y</code>	x menor ou igual à y	<code>3 &lt;= 42</code>	TRUE
<code>x &gt;= y</code>	x menor ou igual à y	<code>3 &gt;= 42</code>	FALSE
<code>x == y</code>	x igual à y	<code>3 == 42</code>	FALSE
<code>x != y</code>	x diferente de y	<code>3 != 42</code>	TRUE
<code>!x</code>	não x	<code>!(3 &gt; 42)</code>	TRUE
<code>x   y</code>	x ou y	<code>(3 &gt; 42)   TRUE</code>	TRUE
<code>x &amp; y</code>	x e y	<code>(3 &lt; 4) &amp; (42 &gt; 13)</code>	TRUE

### 1.8.2.1 Operador lógico `if()`

Dentro dessa classe de operadores, podemos destacar o operador `if()`. É comum usar esse operador para testar condições únicas ou múltiplas na instrução `if()` ou `ifelse()`. Operadores lógicos em R podem ser aplicados a vetores numéricos ou complexos ou objetos booleanos, que são **TRUE** ou **FALSE** (à eles são reservados os *atalkhos* **T** e **F**).

```
knitr:::include_graphics("Figuras/pig_if.png")
```

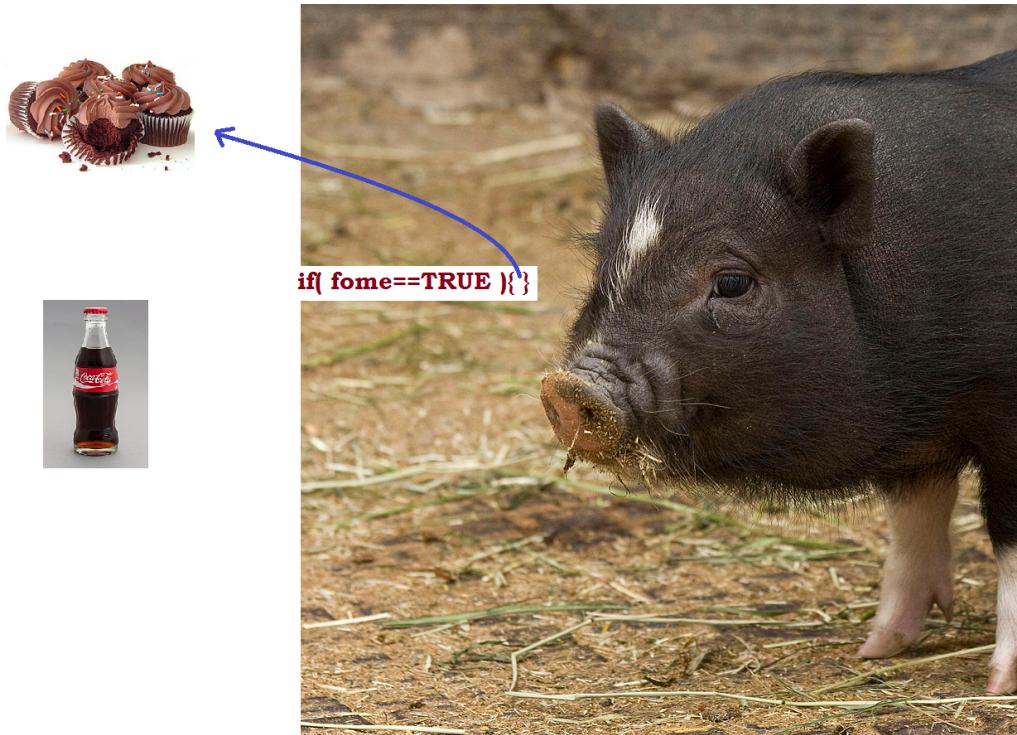


Figure 1.11: Operador lógico \*if()\* - Fonte Imagens: Wikipedia

```
A=4
B=2
if(A>B){
  print("A é maior do que B")
}else{
  print("A não é maior do que B")
}
```

```
## [1] "A é maior do que B"

A=4
B=6
if(A>B){
  print("A é maior do que B")
}else{
  print("A não é maior do que B")
}

## [1] "A não é maior do que B"
```

Outra forma simples de usar o operador é:

```
A=4
B=6
A>B

## [1] FALSE
```

### 1.8.2.2 Operador lógico *ifelse()*

O operador *ifelse( “1”, “2”, “3”)* possui 3 entradas. Na primeira “1”, deve-se colocar a condição a ser testada. Em “2” o resultado caso a condição testada seja verdadeira e em “3” o resultado, caso a condição testada seja falsa.

```
A=4
B=6
# ifelse(A>=B, "Verdadeiro", "FALSO")
```

### 1.8.2.3 Operadores de atribuição

Os operadores de atribuição são provavelmente a família de operadores que você mais usará enquanto trabalha com R. Como o nome desse grupo implica, eles são usados para atribuir objetos, como valores numéricos, *strings*, vetores, modelos e plotagens para um nome (variável). Isso inclui operadores como a seta para trás (<-) ou o sinal de igual (=)

```
str <- "Em Brasília, 19 horas!" # String
int <- 10 # Inteiro
vet <- c(1,2,3,4) # Vetor
```

Para visualizar a variável, podemos simplesmente digitá-la ou usar a função *print()*

```
str

## [1] "Em Brasília, 19 horas!"

#ou
print(str)

## [1] "Em Brasília, 19 horas!"
```

Para visualizar mais de uma variável, podemos usar “c()” para “juntar as variáveis”

```
c(str,int)

## [1] "Em Brasília, 19 horas!" "10"

# ou
print(c(str,int))
```

```
## [1] "Em Brasília, 19 horas!" "10"
```

*Esse exemplo pode ser repetido usando “=” no lugar de “<-”*

### 1.8.3 Tipos de dados

R possui um número básico de *tipos* de dados. Enquanto o R é uma *linguagem fortemente tipada* (não exige do usuário muito conhecimento sobre diferentes tipos de dados) é útil conhecer os tipos disponíveis.

- Numeric
  - Também conhecido como duplo. O tipo padrão ao lidar com números.
  - Exemplos: 1,1,0, 42,5
- Integer
  - Exemplos: 1L,2L, 42L
- Complex
  - Exemplo: 4 + 2i
- Logical
  - Dois valores possíveis: TRUE e FALSE
  - Você também pode usar T e F, mas isso *não* é recomendado.
  - NA também é considerado lógico.
- Character
  - Exemplos: " a ", " Statistics ", " 1 mais 2. "
- Categorical or factor
  - Uma mistura de número inteiro e caractere. Uma variável **fator** atribui um rótulo a um valor numérico.
  - Por exemplo, **fator** (x = c (0,1), labels = c (" male "," female ")) atribui a string *male* aos valores numéricos 0 e a string *female* ao valor 1.

### 1.8.4 Valores especiais

Assim como TRUE e FALSE, existem outros *valores* reservados à situações específicas.

- NA (*Not Available*) representa dado faltante (não disponível), ou que é chamado em estatística de *missing data*.
- NaN (*Not a Number*) são gerados quando temos uma indefinições matemáticas, como sqrt(-1) e 0/0.
- Inf (*Infinito*) é usado quando o valor numérico é muito grande (limite). Por exemplo, exp(2000).
- NULL significa ausência de informação. Parecido com o NA, mas conceitualmente mais usado na lógica de programação.

As funções **is.na()**, **is.nan()**, **is.infinite()** e **is.null()** podem ser usadas para verificar se um objeto possui algum valor com essa característica.

## 1.9 Estrutura dos dados

R também possui um número básico de *estrutura* de dados. Essa estrutura de dados pode ser **homogênea** (setodos os elementos são do mesmo tipo de dados) ou **heterogênea** (se os elementos podem ter mais de um tipo de dados).

Dimensão	Homogênea	Heterogênea
1	Vector	List
2	Matrix	Data Frame
3+	Array	nested Lists

### 1.9.1 Vetores

Muitas operações em R usam **vetores**. Um vetor contém um conjunto de objetos de tipos identicos e são indexados começando na posição 1. A maneira mais comum para se criar um vetor é usar a função `c()`, o que é uma abreviação de `combine()`. Ela combina uma lista de elementos separados por `,`. Por exemplo,

```
c(1,2)
```

```
## [1] 1 2
```

Ou podemos combinar outros objetos, desde que sejam do mesmo tipo.

```
A=c(1,2)
B=c(3,4,5)
c(A,B)
```

```
## [1] 1 2 3 4 5
```

Os objetos não precisam ser numéricos. Por exemplo,

```
A="Amarelo" # objeto da classe "character"
class(A)
```

```
## [1] "character"
```

```
B="Azul"
c(A,B)
```

```
## [1] "Amarelo" "Azul"
```

Embora tenham muitas **letras** nesse objeto combinado `c(A,B)`, na verdade só existem dois elementos, {"Amarelo", "Azul"}. Para acessá-los, usamos o colchete `[]`.

#### 1.9.1.1 Subconjuntos de vetores

Para subconjunto de um vetor, ou seja, para escolher apenas alguns elementos dele, usamos colchetes, `[]`. Aqui vemos que `AA[1]` retorna o primeiro elemento e `AA[2]` retorna o segundo elemento:

```
A="Amarelo"
B="Azul"
AA=c(A,B)
AA[1]
```

```
## [1] "Amarelo"
```

```
AA[2]
```

```
## [1] "Azul"
```

Quando o vetor é maior, podemos usar um conjunto de índices para acessar os valores correspondentes no vetor.

```
x=c(35,42,47,54,70,75)
x[c(1,3,6)]      # Acessa as posições 1, 3 e 6 do vetor
```

```
## [1] 35 47 75
```

```
x[-4]           # Acessa todas as posições do vetor, menos a posição 4
```

```
## [1] 35 42 47 70 75
```

Muitas vezes queremos criar um vetor baseado em uma sequência de números. No R o operador `:` é usado como uma opção `de : até`. Dessa forma é possível usá-lo em diferentes situações.

```
c(1:10) # vetor com números de 1 à 10
```

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

```
1:10      # vetor com números de 1 à 10
```

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

Outros vetores de objetos são possíveis de criar

```
LETTERS[1:4] # maiúsculas
```

```
## [1] "A" "B" "C" "D"
```

```
letters[1:4] # minúsculas
```

```
## [1] "a" "b" "c" "d"
```

### 1.9.1.2 Função `paste()`

Com a função `paste()` podemos concatenar objetos de tipos diferentes e armazená-los em um vetor.

```

paste("Porco_", letters[1:4], sep="")    # Underline no Porco e sem espaço no sep

## [1] "Porco_a" "Porco_b" "Porco_c" "Porco_d"

paste("Porco", letters[1:4], sep="_")    # Underline no sep

## [1] "Porco_a" "Porco_b" "Porco_c" "Porco_d"

p=paste("Porco", letters[1:4], sep="_") # Armazenando no vetor p
                                         # Acessando a posição 2 do vetor p

## [1] "Porco_b"

```

Note que escalares não existem no R. Eles são vetores de tamanho 1.

```
2
```

```
## [1] 2
```

### 1.9.1.3 Função seq()

Se quisermos criar uma sequência que não se limite a números inteiros e que aumente 1 por vez, podemos usar a função `seq()`.

```

seq(from = -0.5, to = 1.8, by = 0.1)

## [1] -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9
## [16] 1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8

```

### 1.9.1.4 Função rep()

Outra operação comum para criar um vetor é `rep()`, que pode repetir um único valor várias vezes.

```

rep("A", times = 10)

## [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"

rep(1, 10)

## [1] 1 1 1 1 1 1 1 1 1 1

```

A função `rep()` pode ser usada para repetir um vetor várias vezes.

```

x=c("O", "A")
rep(x, 3)

## [1] "O" "A" "O" "A" "O" "A"

```

### 1.9.1.5 Função length()

Uma função importante é à que identifica o tamanho do vetor, que é a função `length()`.

```
x=1:5
length(x)
```

```
## [1] 5
```

```
y=rep(x,6)
length(y)
```

```
## [1] 30
```

### 1.9.1.6 Resumo

temos quatro formas de criar vetores

- `c()`
- `:`
- `seq()`
- `rep()`

### 1.9.1.7 Operações com vetores

O R é capaz de executar muitas operações em vetores e escalares.

```
x = 1:10 # Um vetor
x + 1      # Soma um escalar à cada elemento do vetor
```

```
## [1] 2 3 4 5 6 7 8 9 10 11
```

```
2 * x      # Multiplica todos os elementos por 2
```

```
## [1] 2 4 6 8 10 12 14 16 18 20
```

```
2 ^ x      # Eleva 2 na potência correspondente a cada elemento de x
```

```
## [1] 2     4     8    16    32    64   128   256   512 1024
```

```
sqrt(x)    # Calcula a raiz quadrada de cada elemento de x
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
## [9] 3.000000 3.162278
```

```
log(x)      # Calcula o log natural de cada elemento de x
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
## [8] 2.0794415 2.1972246 2.3025851
```

We see that when a function like `log()` is called on a vector `x`, a vector is returned which has applied the function to each element of the vector `x`.

### 1.9.1.8 Operadores lógicos com vetores

Em R, operadores lógicos também funcionam com vetores:

```
x = c(5,3,1,9,27,90)

x == 9

## [1] FALSE FALSE FALSE TRUE FALSE FALSE

x != 9

## [1] TRUE TRUE TRUE FALSE TRUE TRUE

x > 9

## [1] FALSE FALSE FALSE FALSE TRUE TRUE

x < 9

## [1] TRUE TRUE TRUE FALSE FALSE FALSE

x == 9 & x != 9

## [1] FALSE FALSE FALSE FALSE FALSE FALSE

x == 9 | x != 9

## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

Outra operação importante que podemos destacar

```
x[x > 9]

## [1] 27 90

x[x != 9]

## [1] 5 3 1 27 90

sum(x > 9)

## [1] 2

as.numeric(x > 9)

## [1] 0 0 0 0 1 1
```

Usamos a função `sum()` em um vetor de valores lógicos TRUE e FALSE ( resultado de `x>3`) e resultado foi um valor numérico. A operação apenas *contou* quantos vezes `x>3` resultou em TRUE. Durante a chamada de `sum()`, o R automaticamente coerçou (força) automaticamente o lógico para numérico, em que TRUE é 1 e FALSE é 0. Essa coerção do lógico para o numérico acontece na maioria das operações matemáticas.

### 1.9.1.9 Função `which()`

```
# which (dondição de x) retorna verdadeiro / falso
# cada índice de x em que a condição é verdadeira
x = c(5,3,1,9,27,90)
which(x > 9)
```

```
## [1] 5 6
```

```
x[which(x > 9)]
```

```
## [1] 27 90
```

```
max(x)
```

```
## [1] 90
```

```
which(x == max(x))
```

```
## [1] 6
```

```
which.max(x)
```

```
## [1] 6
```

## 1.9.2 Tarefa 2

1. Crie um vetor preenchido com 10 números sorteados na distribuição uniforme discreta em  $\{1,2,3,4,5,6\}$  (dica: use a função `sample()`) e armazene-os em `x`.
2. Usando o subconjunto lógico como acima, obtenha todos os elementos de `x` maiores que 2 e armazene-os em `y`.
3. Usando a função `which`, armazene os *índices* de todos os elementos de `x` que são maiores que 2 em `iy`.
4. Verifique se `y` e `x[iy]` são idênticos.

## 1.9.3 Matrizes

O R também pode ser usado para cálculos de **matriz**. Matrizes têm linhas e colunas contendo um único tipo de dados. Em uma matriz, a ordem das linhas e colunas é importante. (Isso não se aplica à *dataframe*, que é um outro tipo de dado que veremos mais adiante).

Matrizes podem ser criadas usando a função `matrix`.

```
x = 12:1
x
```

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

```
X = matrix(x, nrow = 3, ncol = 4)
X
```

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

Note que o R é **case sensitive** (x vs X).

Por padrão, a função `matrix` preenche seus dados na matriz coluna por coluna. Mas também podemos dizer ao R para preencher as linhas:

```
W = matrix(x, nrow = 3, ncol = 4, byrow = TRUE)
W
```

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

Também podemos criar uma matriz de uma dimensão especificada onde cada elemento é o mesmo, neste caso, 0.

```
Y = matrix(0, 2, 3)
Y
```

```
##      [,1] [,2] [,3]
## [1,]     0    0    0
## [2,]     0    0    0
```

### 1.9.3.1 Matriz diagonal

Para criar uma matriz diagonal podemos usar a função `diag()`

```
diag(4) # cria matriz identidade 4x4
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```
diag(4,5) # cria uma matriz digonal 4x4, em que os elementos da diagonal são 5
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4    0    0    0    0
## [2,]    0    4    0    0    0
## [3,]    0    0    4    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    4
```

Como vetores, matrizes podem ser acessadas usando colchetes, `[]`. No entanto, como as matrizes são bidimensionais, precisamos especificar uma linha e uma coluna ao fazer o subconjunto.

```
X
```

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

```
X[1, 2] # primeira linha e na segunda coluna
```

```
## [1] 9
```

Também podemos acessar uma linha ou coluna inteira.

```
X[1, ]
```

```
## [1] 12 9 6 3
```

```
X[, 2]
```

```
## [1] 9 8 7
```

### 1.9.3.2 Elementos da matriz

Também podemos usar vetores para acessar subconjunto com mais de uma linha ou coluna por vez. Aqui acessamos à primeira e terceira coluna da segunda linha:

```
X[2, c(1, 3)] # segunda linha, primeira e terceira coluna
```

```
## [1] 11 5
```

```
X[c(2,1), c(1, 3)] # segunda e priemira linha, primeira e terceira coluna
```

```
##      [,1] [,2]
## [1,]    11    5
## [2,]    12    6
```

### 1.9.3.3 Funções rbind() e cbind

As matrizes também podem ser criadas combinando vetores como colunas, usando `cbind`, ou combinando vetores como linhas, usando `rbind`.

```
x = 1:4
rev(x)
```

```
## [1] 4 3 2 1
```

```
rep(1,4)
```

```
## [1] 1 1 1 1
```

```
rbind(x, rev(x), rep(1, 4))
```

```
##      [,1] [,2] [,3] [,4]
## x     1     2     3     4
##       4     3     2     1
##       1     1     1     1
```

```
cbind(col_1 = x, col_2 = rev(x), col_3 = rep(1, 4))
```

```
##      col_1 col_2 col_3
## [1,]     1     4     1
## [2,]     2     3     1
## [3,]     3     2     1
## [4,]     4     1     1
```

Ao usar `rbind` e `cbind`, você pode especificar nomes de “argumentos” que serão usados como nomes de colunas.

#### 1.9.3.4 Operações com Matrizes

O R pode então ser usado para realizar cálculos de matriz.

```
x = 1:12
y = 12:1
X = matrix(x, 3, 4)
Y = matrix(y, 3, 4)
X
```

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

```
Y
```

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

```
X + Y # Soma elemento por elemento
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    13    13    13    13
## [2,]    13    13    13    13
## [3,]    13    13    13    13
```

```
X - Y # Subtração elemento por elemento
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   -11    -5     1     7
## [2,]    -9    -3     3     9
## [3,]    -7    -1     5    11
```

```
X * Y # Multiplicação elemento por elemento
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    12    36    42    30
## [2,]    22    40    40    22
## [3,]    30    42    36    12
```

```
X / Y # Divisão elemento por elemento
```

```
## [,1]     [,2]     [,3]     [,4]
## [1,] 0.08333333 0.44444444 1.1666667 3.333333
## [2,] 0.18181818 0.62500000 1.6000000 5.500000
## [3,] 0.30000000 0.8571429 2.2500000 12.000000
```

Note que `X * Y` não é multiplicação de matrizes. É multiplicação de *elemento por elemento*. (O mesmo para `X/Y`).

Para a multiplicação de matrizes usa-se `%*%`. Outras funções para operações com matrizes são `t()`, que fornece a transposição de uma matriz e `solve()`, que retorna a inversa de uma matriz quadrada, se for invertível.

```
x = 1:9
y = 9:1
X = matrix(x, 3, 3)
Y = matrix(y, 3, 3)
X %*% Y
```

```
## [,1] [,2] [,3]
## [1,] 90   54   18
## [2,] 114  69   24
## [3,] 138  84   30
```

```
t(X)
```

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

#### 1.9.4 Arrays

Um vetor é um array unidimensional. Uma matriz é um array bidimensional. Em R, o usuário pode criar arrays de dimensionalidade arbitrária N. Por exemplo:

```
A = 1:16
B = array(data = A, dim = c(4, 2, 2))
C = array(data = A, dim = c(4, 2, 2, 3)) # Recicla A=1:16
B
```

```
## , , 1
##
## [,1] [,2]
## [1,] 1   5
## [2,] 2   6
## [3,] 3   7
## [4,] 4   8
##
## , , 2
##
## [,1] [,2]
## [1,] 9   13
## [2,] 10  14
## [3,] 11  15
## [4,] 12  16
```

Note que B são simplesmente *duas* matrizes (4,2) armazenadas como *páginas* em um mesmo objeto/variável. Similarmente, C teria duas *páginas* e mais 3 registros em uma quarta dimensão. E assim por diante.

Para acessar esses elementos, o procedimento é similar ao de uma matriz ou um vetor, cuidando para indexar cada dimensão:

```
B[,1]      # Todos os elementos da col 1, página 1

## [1] 1 2 3 4

B[2:3, , ] # Linhas 2:3 de todas as páginas e colunas

## , , 1
##
## [,1] [,2]
## [1,]    2    6
## [2,]    3    7
##
## , , 2
##
## [,1] [,2]
## [1,]   10   14
## [2,]   11   15

B[2,2, ]    # linha 2, coluna 2 de todas as páginas.
```

```
## [1] 6 14
```

#### 1.9.4.1 Tarefa 3

1. Crie um vetor contendo 1,2,3,4,5 chamado-o de v.
2. Crie uma matriz (2,5) m contendo os dados 1,2,3,4,5,6,7,8,9,10. A primeira linha deve ser “1,2,3,4,5”.
3. Realize a multiplicação da matriz de m com v. Use o comando %\*%. Qual a dimensão da saída?
4. Por que v%\*% m não funciona?

#### 1.9.5 Listas

Uma lista é uma estrutura de dados unidimensional *heterogênea*. Portanto, é indexado como um vetor com um único valor inteiro (ou com um nome), mas cada elemento pode conter um elemento de qualquer tipo. As listas são semelhantes a um objeto python ou julia.

Muitas estruturas e saídas do R são listas. As listas são objetos extremamente úteis e versáteis; portanto, é interessante entender sua utilização:

```
# Listas simples sem nomes nas entradas
list(12, "Bom dia", TRUE)

## [[1]]
## [1] 12
##
## [[2]]
## [1] "Bom dia"
##
## [[3]]
## [1] TRUE
```



- Café
- Milho
- Chocolate
- Jabuticabas
- Jujubas
- Sorvete
- Pé de moleque
- Doce de leite
- Bolo de fubá
- Canjica

Rabicó, personagem de Monteiro Lobato.

Fonte da imagem: Wikipedia

Figure 1.12: Lista de supermercado do Rabicó

```
# Listas com nome para cada entrada
lista1 = list(
  a = c(1, 2, 3, 4),
  b = TRUE,
  c = "Bom dia!",
  d = function(arg = 10) {print("Em Brasília, 19h!")},
  e = diag(4)
)
```

Os elementos das listas podem ser acessados usando duas sintaxes, o operador `$` e colchetes `[]`. O operador `$` retorna um elemento **nomeado** de uma lista. A sintaxe `[]` retorna uma **lista**, enquanto a `[[[]]]` retorna um **elemento** de uma lista.

- `lista1[1]` retorna uma lista que contém o primeiro elemento
- `lista1[[1]]` retorna o primeiro elemento da lista, neste caso, um vetor.

```
# elementos da lista
```

```
lista1$e
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```
lista1[[1:2]]
```

```
## [1] 2
```

```
lista1[c("e", "a")]
```

```
## $e
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
##
## $a
## [1] 1 2 3 4
```

```
lista1["e"]
```

```
## $e
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```
lista1[["e"]]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```
lista1$d
```

```
## function(arg = 10) {print("Em Brasília, 19h!")}
```

```
lista1$d(arg = 1)
```

```
## [1] "Em Brasília, 19h!"
```

#### Tarefa 4

1. Copie e cole o código acima para `list1` na sua sessão do R. Lembre-se de que `list` pode conter qualquer tipo de objeto R. Como por exemplo, outra lista! Portanto, crie uma nova lista `lista2` que tenha dois campos: o primeiro campo chamado "`Isso`" com o conteúdo da string "`é impressionante`" e um segundo campo chamado "`lista`" que contém `list1`.
2. Acessar os elementos é como em uma lista simples, apenas com várias camadas agora. Obtenha o elemento `c` de `list1` em `lista2`!
3. Componha uma nova string com primeiro elemento em `lista2`, o elemento sob o rótulo "`Isso`". Use a função `paste` para imprimir R `é impressionante` na tela.

## 1.10 Programação básica

Nesta seção, ilustramos alguns conceitos gerais relacionados à programação.

### 1.10.1 Variáveis

Na programação, uma variável denota um *objeto*. Outra maneira de dizer isso é que uma variável é um nome ou um *label* para algo:

```
x = 2
y = "doce"
z = function(x){log(x)}
```

Aqui, `x` se refere ao valor 2, `y` refere-se a *string* “doce” e `z` é o nome de uma função que calcula  $\log x$ . Observe que o argumento `x` da função é diferente do `x` que acabamos de definir.

```
x
## [1] 2

z(9)
## [1] 2.197225

z(x)
## [1] 0.6931472
```

### 1.10.2 Laços (*Loops*), `for` e `while`

Laços (*Loops*) são uma construção de programação muito importante. Como o nome sugere, em um *loop*, a programação *repetidamente* circula um conjunto de instruções, até que alguma condição diga para parar. Uma construção muito poderosa, porém simples, é que o programa pode *contar quantas etapas* já executou - o que pode ser importante saber para muitos algoritmos. Uma forma de fazer um *loop* é o usando o `for`.

#### 1.10.2.1 Uso do `for()`

Por exemplo, para executar uma tarefa 3 vezes, usamos

```
for (i in 1:3){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

A iteração pode ser feita sobre um vetor qualquer de objetos

```
for (i in c("milho","soja","trigo")){
  print(paste("Produção de ",i)) # a função paste() conecta as strings
}
```

```
## [1] "Produção de milho"
## [1] "Produção de soja"
## [1] "Produção de trigo"
```

Podemos fazer *loops* dentro de *loops* (*nested loops*)

```
for (i in 2:3){
  # primeiro: para cada i
  for (j in c("milho","soja","trigo")){
    # segundo: para cada j
    print(paste("Usar",i,"Kg de",j,"na fórmula."))
  }
}

## [1] "Usar 2 Kg de milho na fórmula."
## [1] "Usar 2 Kg de soja na fórmula."
## [1] "Usar 2 Kg de trigo na fórmula."
## [1] "Usar 3 Kg de milho na fórmula."
## [1] "Usar 3 Kg de soja na fórmula."
## [1] "Usar 3 Kg de trigo na fórmula."
```

### 1.10.2.2 Uso do while()

A função `while()`, como o próprio nome sugre, é usada para executar uma tarefa até que uma certa condição aconteça, ou enquanto certa condição não aconteça.

```
Kg=27
while(Kg<30){
  print(paste("Peso é",Kg)) # Enquanto peso é menor do que 30, executa
  Kg=Kg+1 # Enquanto peso é menor do que 30, soma 1 no peso atual
}
```

```
## [1] "Peso é 27"
## [1] "Peso é 28"
## [1] "Peso é 29"
```

## 1.11 Funções

Até agora, usamos funções, mas na verdade não discutimos alguns de seus detalhes. Uma função é um conjunto de instruções que o R executa para nós, como as coletadas em um arquivo *script*. O bom é que as funções são muito mais flexíveis que os scripts, pois podem depender de *argumentos de entrada*, que alteram a maneira como a função se comporta. Aqui está como definir uma função:

```
nome_da_funcao <- function(arg1,arg2=valor_padrao){
  # corpo da função
  # faça coisas com arg1 e arg2
  # a última linha será retornada
}
```

Um exemplo de uma função simples

```
fun1<- function(seu_nome = "Baby"){
  paste("Seja bem vindo ao R,",seu_nome)
  # pode-se escrever também
  # return(paste("Seja bem vindo ao R,",seu_nome))
}
#chamando a função sem nenhum argumento, o argumento seu_nome padrão será Baby
fun1()

## [1] "Seja bem vindo ao R, Baby"
```

Agora tente com o seu nome:

```
fun1("Rabicó")

## [1] "Seja bem vindo ao R, Rabicó"
```

Just typing the function name returns the actual definition to us, which is handy sometimes:

```
fun1

## function(seu_nome = "Baby"){
##   paste("Seja bem vindo ao R,",seu_nome)
##   # pode-se escrever também
##   # return(paste("Seja bem vindo ao R,",seu_nome))
## }
## <bytecode: 0x0000000013e841c0>
```

Funções são maneiras de dizer ao R o que deve ser feito. É como se ensinássemos uma tarefa ao software para que ele passe a executá-la sempre que solicitado. Um dica para não acumular erros durante a programação é dividir o *script* em várias funções. Quando se tem um conjunto de funções que realiza um conjunto de tarefas mais amplas, pode-se juntar tudo em um pacote (*package*), submeter ao CRAN para disponibilizar à comunidade.

### Tarefa 6

1. Escreva um *loop for* para escrever o seu nome, letra por letra. Comece com um vetor com as letras do seu nome.
2. Modifique esse *loop* para que cada iteração leve aproximadamente um segundo. Você pode conseguir isso adicionando o comando `Sys.sleep(1)` abaixo da linha que imprime “iteração i nome\_parcial”, em que `i` é a correspondente iteração no *loop*.

## 1.12 Operador Pipe

Para manipulação de dados é interessante conhecer a lógica do operador pipe, `%>%`. Vários pacotes são construídos em cima dessa lógica, que busca encadear ações do R de uma forma mais compreensível. Vamos instalar o pacote `magrittr` para começar a usar `%>%`.

```
install.packages("magrittr")
library(magrittr)
```

Vejamos como funciona.

```
sin(cos(pi))
pi%>%cos%>%sin
```

Ambas as expressões acima são equivalentes. Parece redundante e que não se tem nenhuma vantagem ao usar o `%>%`. No entanto, para um caso envolvendo dados reais, podemos verificar sua utilidade

```
head(id)
```

```
## # A tibble: 6 x 16
##   Baia Tratamento Idade Dieta Grupo Contagem0 Contagem13 PerdasCreche
##   <dbl>     <dbl> <dbl> <chr>  <chr>    <dbl>      <dbl>
## 1 109       1     16 A     Grp4     26        25        1
## 2 116       1     16 A     Grp3     26        24        1
## 3 118       1     16 A     Grp3     26        20        1
## 4 208       1     16 A     Grp4     26        23        2
## 5 213       1     16 A     Grp4     26        20        4
## 6 222       1     16 A     Grp3     26        26        0
## # ... with 8 more variables: PerdasTerminação <dbl>, PerdasTotais <dbl>,
## #   Peso0 <dbl>, Peso13 <dbl>, GPDacumulado <dbl>, ConsumoAcumulado <dbl>,
## #   Conversão <dbl>, Medicados <dbl>
```

```
id%>%na.omit%>%lm(GPDacumulado ~ id$Dieta+ConsumoAcumulado, data = .) %>% summary
```

```
##
## Call:
## lm(formula = GPDacumulado ~ id$Dieta + ConsumoAcumulado, data = .)
##
## Residuals:
##       Min        1Q        Median         3Q        Max
## -0.0154999 -0.0046727 -0.0003187  0.0052794  0.0177905
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 0.192787  0.013699 14.073   <2e-16 ***
## id$DietaB -0.002336  0.001817 -1.286   0.203    
## ConsumoAcumulado 0.317334  0.009098 34.879   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.007638 on 69 degrees of freedom
## Multiple R-squared:  0.9478, Adjusted R-squared:  0.9463 
## F-statistic: 626.3 on 2 and 69 DF,  p-value: < 2.2e-16
```

# Chapter 2

# Importando dados

Objetivo deste capítulo é mostrar algumas formas de importação de dados. Para usar as ferramentas que o R disponibiliza é importante que o usuário consiga carregar os dados e deixá-los em um formato adequado para que o R faça a sua mágica.



Figure 2.1: Loading

Em um primeiro momento, trabalharemos com **dados delimitados**, que facilita para o usuário iniciante, manipular, selecionar e remover colunas, agrupar, filtrar, mudar a estrutura e o formato dos dados para que seja possível fazer operações básicas, gráficos e análises.

## 2.1 Dados de diferentes formatos

### 2.1.1 Dados no formato .txt

Dados no formato .txt podem ser facilmente importados. Os dados que serão carregados para exemplo estão dentro de uma pasta chamada Dados no diretório de trabalho setado previamente usando o `setwd()`. A primeira

forma de carregar um arquivo .txt é usando a função `read.table()`.

```
MG_txt=read.table("Dados/MorningGlory.txt")
head(MG_txt)
```

```
##      V1   V2     V3     V4     V5     V6     V7     V8
## 1 Treatment Pen Intake021 Intake2142 gain021 gain2142 Mornglor021 mornglor2142
## 2          A   1    106,1    140,3    64,6    75,25        0        0
## 3          A   6    102,9    163,0    60,65   84,95        0        0
## 4          A   7    103,3    162,6    61,95   83,9        0        0
## 5          A  12    109,8    178,3    65,35   87,85        0        0
## 6          A  13    109,3    175,1     63    83,8        0        0
##           V9      V10     V11     V12
## 1 Totintake021 Totintake2142 Days021 Days2142
## 2          106,1    140,3    84,0    84,0
## 3          102,9    163,0    84,0    84,0
## 4          103,3    162,6    84,0    84,0
## 5          109,8    178,3    84,0    84,0
## 6          109,3    175,1    84,0    84,0
```

Podemos observar que as colunas não foram nomeadas adequadamente. Podemos resolver isso usando o argumento `header = TRUE`

```
MG_txt=read.table("Dados/MorningGlory.txt",header = TRUE)
head(MG_txt)
```

```
##   Treatment Pen Intake021 Intake2142 gain021 gain2142 Mornglor021 mornglor2142
## 1          A   1    106,1    140,3    64,6    75,25        0        0
## 2          A   6    102,9    163,0    60,65   84,95        0        0
## 3          A   7    103,3    162,6    61,95   83,9        0        0
## 4          A  12    109,8    178,3    65,35   87,85        0        0
## 5          A  13    109,3    175,1     63    83,8        0        0
## 6          B   2     77,3    121,0    49,95   64,25        0        0
##   Totintake021 Totintake2142 Days021 Days2142
## 1          106,1    140,3    84,0    84,0
## 2          102,9    163,0    84,0    84,0
## 3          103,3    162,6    84,0    84,0
## 4          109,8    178,3    84,0    84,0
## 5          109,3    175,1    84,0    84,0
## 6          77,3     121,0    84,0    84,0
```

Nesse exemplo os dados estão separados por tabulação. Caso a separação fosse por , poeríamos usar o argumento `sep=","`. Os dados do exemplo MorningGlory estão publicados em Sreng et al. (2020).

Podemos manipular os dados que estão na variável `MG_txt` e salvar em um arquivo .txt. Vamos retirar algumas colunas da variável `MG_txt`.

```
colnames(MG_txt)[5:12]
```

```
## [1] "gain021"      "gain2142"      "Mornglor021"    "mornglor2142"
## [5] "Totintake021" "Totintake2142" "Days021"       "Days2142"
```

```
MG1<-MG_txt[,-c(5:12)]
colnames(MG1)
```

```
## [1] "Treatment"   "Pen"          "Intake021"    "Intake2142"
write.table(MG1, file="Dados/MG1.txt",sep=";")
```

A função `write.table()` irá salvar a variável MG1 em um arquivo MG1.txt em que os dados estarão separados por ;.

### 2.1.2 Dados no formato .csv

A função `read.csv` funciona de forma bem semelhante à `read.table()`. Ambas podem ser usadas para carregar um arquivo .csv.

```
#gas2=read.table("Dados/gas2.csv",header = TRUE,sep=";")
gas2_csv=read.csv("Dados/gas2.csv",header = TRUE,sep=";")
head(gas2_csv)
```

```
##      col Treat   X0   X20   X40   X60   X90   X120   X180   X240
## 1 Flash 1 Dry 0,075 0,265 0,316 0,351 0,41 0,491 0,632 0,671
## 2 Flash 1 Dry 0,076 0,267 0,32 0,367 0,421 0,497 0,6 0,685
## 3 Flash 10 Dry 0,106 0,32 0,348 0,417 0,516 0,519 0,603 0,747
## 4 Flash 10 Dry 0,1 0,314 0,35 0,408 0,521 0,525 0,61 0,74
## 5 BMR 1000 Dry 0,067 0,249 0,325 0,34 0,415 0,461 0,532 0,553
## 6 BMR 1000 Dry 0,07 0,24 0,324 0,347 0,435 0,472 0,54 0,557
```

Os dados do arquivo `gas2.csv` podem ser encontrados em Brambillasca (2019).

### 2.1.3 Dados no formato .xls e .xlsx

Para ler dados nos formatos .txt e .csv o R possui funções nativas. No entanto, para demais formatos, será necessário instalar pacotes.

*A maneira de importar dados não necessariamente será única. Apresentaremos uma forma de trabalhar com os diferentes formatos. No entanto, o pacote `readr`, por exemplo, pode fornecer recursos mais otimizados.*

Para importar dados nos formatos .xls e .xlsx será necessário instalar o pacote `readxl` e então carregá-lo.

```
install.packages("readxl")
library(readxl)
```

Para ler os dados no formato .xls ou .xlsx a função `read_excel()` pode ser usada.

```
library(readxl)
gas_xlsx=read_excel("Dados/gas.xlsx",sheet = 3)
head(gas_xlsx)
```

```
## # A tibble: 6 x 8
##   Sample Trat   var treat repet tannin     HI     pGI
##   <chr>  <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Flash 1 Dry     1     1     1     1 61.7 62.5
## 2 Flash 1 Dry     1     1     2     1 61.0 61.8
## 3 Flash 10 Dry    2     1     1     2 65.4 66.3
## 4 Flash 10 Dry    2     1     2     2 65.8 66.7
## 5 Jfood   Dry    3     1     1     2 73.6 74.5
## 6 Jfood   Dry    3     1     2     2 74.0 75.0
```

Nesse exemplo carregamos a *aba 3* da planilha *gas.xlsx*.

### 2.1.4 Dados de um repositório web

Dados em repositório da web podem ser baixados diretamente com o R. Os dados do exemplo anterior (Brambillasca (2019)) podem ser carregados usando-se a função `download.file()`

```
temp = tempfile(fileext = ".xlsx")
dataURL <- "http://data.mendeley.com/datasets/33shf74zn6/1/files
/89c793ba-231e-45c9-9425-95da36fc8c14/data%20in%20vitro%20starch%20and%20gas.xlsx?dl=1"
download.file(dataURL, destfile=temp, mode='wb')
gasw_xlsx <- read_excel(temp, sheet =3)
head(gasw_xlsx)
```

```
## # A tibble: 6 x 8
##   Sample   Trat    var treat repet tannin     HI    pGI
##   <chr>    <chr>  <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Flash 1 Dry      1     1     1     1  61.7  62.5
## 2 Flash 1 Dry      1     1     2     1  61.0  61.8
## 3 Flash 10 Dry     2     1     1     2  65.4  66.3
## 4 Flash 10 Dry     2     1     2     2  65.8  66.7
## 5 Jfood   Dry      3     1     1     2  73.6  74.5
## 6 Jfood   Dry      3     1     2     2  74.0  75.0
```

### 2.1.5 Dados do SAS, SPSS e STATA

Dados do SAS podem ser carregados usando as funções do pacote `haven`.

```
install.packages("haven")
library("haven")

comp_sas <- read_sas("Dados/companies-2.sas7bdat")
head(comp_sas)

## # A tibble: 6 x 11
##   Type Types symbol obsno ror5    de salesgr5 eps5  npm1    pe payoutr1
##   <dbl> <chr>  <chr>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1 Chem dia      1 13    0.7   20.2  15.5  7.2    9  0.426
## 2 1 Chem dow     2 13    0.7   17.2  12.7  7.3    8  0.381
## 3 1 Chem stf     3 13    0.4   14.5  15.1  7.9    8  0.407
## 4 1 Chem dd       4 12.2  0.2   12.9  11.1  5.4    9  0.568
## 5 1 Chem uk       5 10    0.4   13.6  8      6.7    5  0.325
## 6 1 Chem psm      6 9.8   0.5   12.1  14.5  3.8    6  0.511
```

Dados *companies-2* são usados no livro Afifi et al. (2019).

Se os dados estiverem no formato *.sav* do SPSS, uma opção é usar a função `read_sav()` do pacote `haven`.

```
comp_spss<- read_sav("Dados/companies-2.sav")
head(comp_spss)
```

```
## # A tibble: 6 x 7
##       PE    ROR5     DE SALESGR5   EPS5   NPM1 PAYOUTR1
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     9  13  0.700  20.2  15.5  7.20  0.430
## 2     8  13  0.700  17.2  12.7  7.30  0.380
## 3     8  13  0.400  14.5  15.1  7.90  0.410
## 4    9 12.2  0.200  12.9  11.1  5.40  0.570
## 5     5  10  0.400  13.6   8.0  6.70  0.320
## 6    6  9.80  0.5   12.1  14.5  3.80  0.510
```

Para dados do STATA salvos no formato .dta pode-se usar a função `read_dta()` também do pacote `haven`.

```
comp_stata<- read_dta("Dados/companies-2.dta")
head(comp_stata)
```

```
## # A tibble: 6 x 11
##   type types symbol obsno ror5 de salesgr5 eps5 npm1 pe payoutr1
##   <dbl> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1 Chem dia 1 13 0.700 20.2 15.5 7.20 9 0.426
## 2 1 Chem dow 2 13 0.700 17.2 12.7 7.30 8 0.381
## 3 1 Chem stf 3 13 0.400 14.5 15.1 7.90 8 0.407
## 4 1 Chem dd 4 12.2 0.200 12.9 11.1 5.40 9 0.568
## 5 1 Chem uk 5 10 0.400 13.6 8 6.70 5 0.325
## 6 1 Chem psm 6 9.80 0.5 12.1 14.5 3.80 6 0.511
```

Usando o RStudio é possível carregar os dados a partir de `File>Import Dataset` conforme a próxima figura

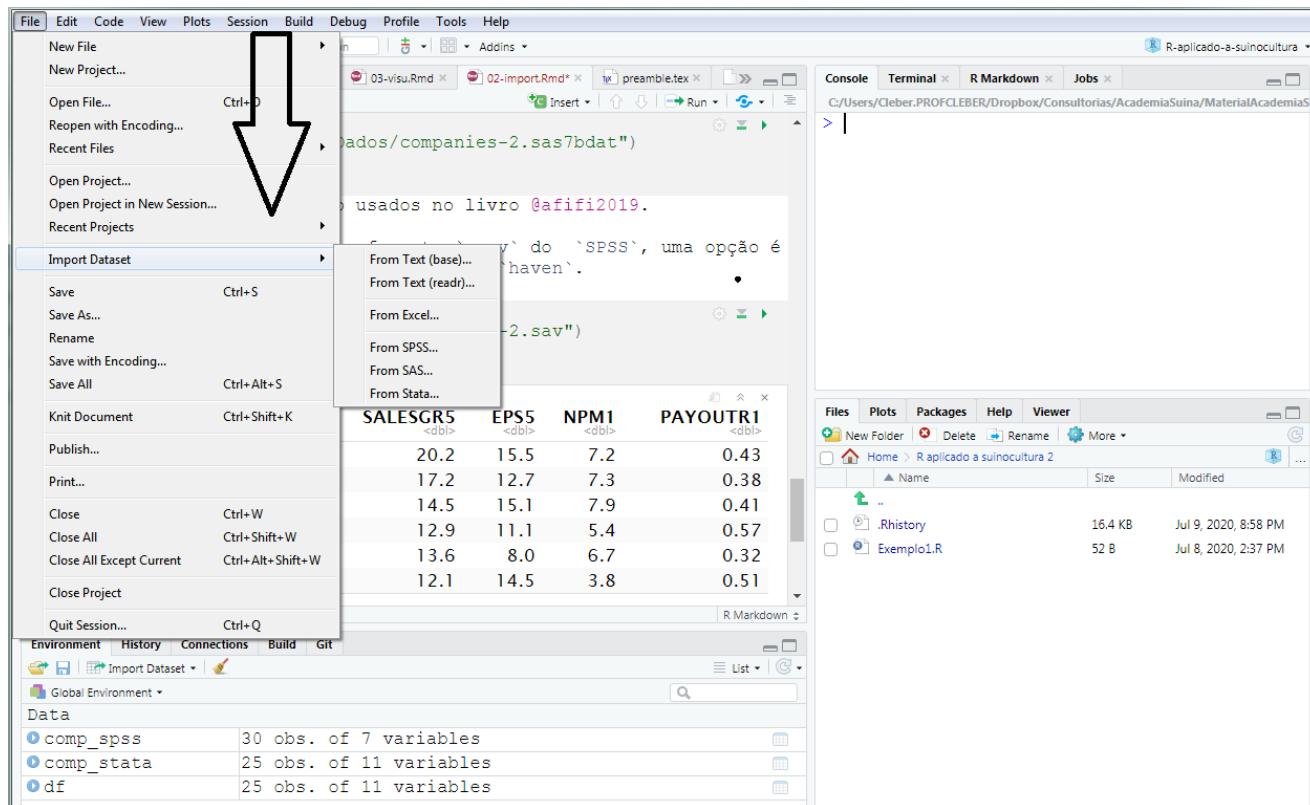


Figure 2.2: Carregando dados a partir do ‘RStudio’

A desvantagem de fazer através desse procedimento é que o comando não ficará registrado em um *script*, e sempre que reiniciar o R esse ação terá que ser repetida manualmente.

## 2.2 Data Frames

Dados do tipo não homogêneo chamado **Data Frames** é fundamental para o bom funcionamento de uma ferramenta muito útil para a visualização de dados, o `ggplot2`. Quase todas as formas de importação de dados carregam os dados para uma variável da classe `data.frame()`.

Por exemplo, para os formatos de dados importados,

- .txt
- .csv
- .xlsx
- .sas
- .sav
- .dta

a classe da variável criada pelo R é `data.frame`, como podemos observar

```
class(MG_txt)

## [1] "data.frame"

class(gas2_csv)

## [1] "data.frame"

class(gasw_xlsx)

## [1] "tbl_df"      "tbl"        "data.frame"

class(comp_sas)

## [1] "tbl_df"      "tbl"        "data.frame"

class(comp_spss)

## [1] "tbl_df"      "tbl"        "data.frame"

class(comp_stata)

## [1] "tbl_df"      "tbl"        "data.frame"
```

*Observe que enquanto as funções `read.table()` e `read.csv` carregaram os dados para variáveis da classe `data.frame`, as demais funções geraram objetos “tbl\_df”, “tbl”, “data.frame” . O termo “tbl” vem de *Tibbles* que é uma maneira mais esperta de criar `data.frames`. Mais adiante iremos explorar o *Tibbles*.*

*Por suportar dados não homogêneos, esse formato permite um flexibilidade e dinamismo nas análises ou aplicação de metodologia estatística. Por exemplo, um `data.frame` pode ter a seguinte característica*

```
df1 = data.frame(x = c(1, 3, 5, 7, 9, 1, 3, 5, 7, 9),
                  y = c(rep("Bacon", 9), "Costela"),
                  z = rep(c(TRUE, FALSE), 5))
```

Observe que linhas correspondentes a observações e colunas correspondentes a vetores contendo dados para uma variável específica.

Vamos visualizar o data.frame criado é comum usar duas formas, a função `head()` e a função `str()`.

```
str(df1)
```

```
## 'data.frame': 10 obs. of 3 variables:
## $ x: num 1 3 5 7 9 1 3 5 7 9
## $ y: Factor w/ 2 levels "Bacon","Costela": 1 1 1 1 1 1 1 1 1 2
## $ z: logi TRUE FALSE TRUE FALSE TRUE FALSE ...
```

Ao contrário de uma lista, que tem mais flexibilidade, os elementos de um data.frame devem ser todos vetores. Novamente, acessamos qualquer coluna com o operador `$`:

```
df1$y
```

```
## [1] Bacon Bacon Bacon Bacon Bacon Bacon Bacon Bacon Bacon
## [10] Costela
## Levels: Bacon Costela
```

Nesse trabalho estamos focando em uma primeira abordagem ao R. Em um segundo momento, podemos avançar mais em questões relacionadas à manipulação de dados. Nesse contexto podemos citar os pacotes `tidyverse`, `tibble` e `dplyr` que permitem trabalhar com dados cuja formatação e estrutura não sejam tão organizadas. Claro, que eles não são como uma *varinha mágica* que irão resolver toda a bagunça de um bano de dados, mas ajuda bastante.

## 2.3 Análise exploratória

Análise exploratória de dados é uma etapa importante para identificar padrões nas variáveis, descobrir erros e problemas nos conjuntos de dados, revelar relações entre as variáveis, observar comportamento médio, dispersão. Usa-se um conjunto de medidas estatísticas aliadas à métodos de visualização de dados.

### 2.3.1 Dados faltantes

Como visto no Capítulo (???), se um banco de dados estiver incompleto, com alguma observação faltante, o R irá identificá-la como um `NA`. Isso pode ser identificado usando-se a função `is.na()`.

```
MG2=read_excel("Dados/MorningGlory2.xlsx",n_max =6)
head(MG2)
```

```
## # A tibble: 5 x 12
##   Treatment  Pen Intake021 Intake2142 gain021 gain2142 Mornglor021 mornglor2142
##   <chr>     <dbl>    <dbl>      <dbl>    <dbl>    <dbl>      <dbl>      <dbl>
## 1 A          1       106.      140.     64.6     75.2       0        0
## 2 B          6       103.      163.     60.6     85.0      NA        0
## 3 B          7       103.      163.     62.0     83.9       0        NA
```

```

## 4 C          12      110.       NA     65.4     87.8      0      0
## 5 A          13      109.      175.      63     83.8      0      0
## # ... with 4 more variables: Totintake021 <dbl>, Totintake2142 <dbl>,
## #   Days021 <dbl>, Days2142 <dbl>

sum(is.na(MG2)) # quantos NAs temos no banco de dados

## [1] 3

is.na(MG2) #retorna quais valores são NAs

##    Treatment Pen Intake021 Intake2142 gain021 gain2142 Mornglor021
## [1,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [2,] FALSE FALSE FALSE FALSE FALSE FALSE TRUE
## [3,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [4,] FALSE FALSE FALSE TRUE FALSE FALSE FALSE
## [5,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##   mornglor2142 Totintake021 Totintake2142 Days021 Days2142
## [1,] FALSE FALSE FALSE FALSE FALSE FALSE
## [2,] FALSE FALSE FALSE FALSE FALSE FALSE
## [3,] TRUE  FALSE FALSE FALSE FALSE FALSE
## [4,] FALSE FALSE FALSE FALSE FALSE FALSE
## [5,] FALSE FALSE FALSE FALSE FALSE FALSE

```

Algumas funções não funcionam corretamente na presença de `NA`. Por exemplo, a função `mean()` que calcula a média

```
mean(MG2$Intake2142)
```

```
## [1] NA
```

Podemos calcular a média forçando a remoção do `NA` através do argumento `na.rm=TRUE`

```
mean(MG2$Intake2142,na.rm = TRUE)
```

```
## [1] 160.2582
```

Ou podemos remover as linhas em que aparecem os `NA`. No entanto assim estaremos retirando todas as informações, o que não é recomendável. Mas poderá ser feito da forma

```
library(tidyr)
MG2 %>% drop_na()
```

```

## # A tibble: 2 x 12
##   Treatment Pen Intake021 Intake2142 gain021 gain2142 Mornglor021
##   <chr>     <dbl>    <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
## 1 A          1      106.     140.     64.6    75.2     0
## 2 A          13     109.     175.     63      83.8     0
## # ... with 4 more variables: Totintake021 <dbl>, Totintake2142 <dbl>,
## #   Days021 <dbl>, Days2142 <dbl>
```

### 2.3.2 Fatores

Se o objetivo é identificar, por exemplo, quantos tratamentos diferentes temos na variável `Treatment`, pode-se usar a função `unique()`

```
unique(MG2$Treatment)
```

```
## [1] "A" "B" "C"
```

### 2.3.3 Medidas de tendência central

Além da média, que pode ser obtida através da função `mean()`, outras medidas são facilmente obtidas com o R. No entanto vamos dar destaque para a média em caso dos dados em que não temos NAs.

#### 2.3.3.1 Média

```
colnames(MG2)
```

```
## [1] "Treatment"      "Pen"           "Intake021"       "Intake2142"
## [5] "gain021"         "gain2142"        "Mornglor021"     "mornglor2142"
## [9] "Totintake021"    "Totintake2142"   "Days021"         "Days2142"
```

```
mean(MG2$Intake021)
```

```
## [1] 106.2858
```

Caso os dados estejam armazenados em uma matriz é possível usar a função `colMeans()` para obter as médias de todas as colunas de uma única vez.

```
M=matrix(ncol=2,c(1,2,3,4))
M
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
colMeans(M)
```

```
## [1] 1.5 3.5
```

Quando os dados estão no formato de uma matriz a função `apply` pode ser bastante útil.

```
apply(M,1,mean) # média das linhas
```

```
## [1] 2 3
```

```
apply(M,2,mean) # média das colunas
```

```
## [1] 1.5 3.5
```

Se o objetivo for calcular outra medida, como por exemplo o desvio padrão, basta trocar `mean` por `sd`.

### 2.3.3.2 Variância e Desvio Padrão

A variância pode ser obtida usando-se a função `var` e o desvio padrão, a função `sd`.

```
var(MG2$Intake021) #Variância
```

```
## [1] 10.45042
```

```
sd(MG2$Intake021) #Desvio Padrão
```

```
## [1] 3.23271
```

### 2.3.3.3 Mediana e quantis

```
median(MG2$Intake021) # Mediana
```

```
## [1] 106.0697
```

```
quantile(MG2$Intake021) # min + Quartis + max
```

```
##      0%      25%      50%      75%     100%
## 102.9103 103.3225 106.0697 109.3440 109.7825
```

```
quantile(MG2$Intake021,c(0.1,0.9)) # percentil 10% e 90%
```

```
##      10%      90%
## 103.0752 109.6071
```

### 2.3.3.4 Range

```
range(MG2$Intake021) # mínimo e máximo do conjunto
```

```
## [1] 102.9103 109.7825
```

## 2.4 Gráficos simples (`scatter plot`)

Visualização de dados é uma etapa prazerosa e importante na análise de dados. Nesse primeiro momento vamos fazer gráficos mais simples, de uma ou duas variáveis, sem usar o pacote `ggplot2` que será introduzido no próximo capítulo.

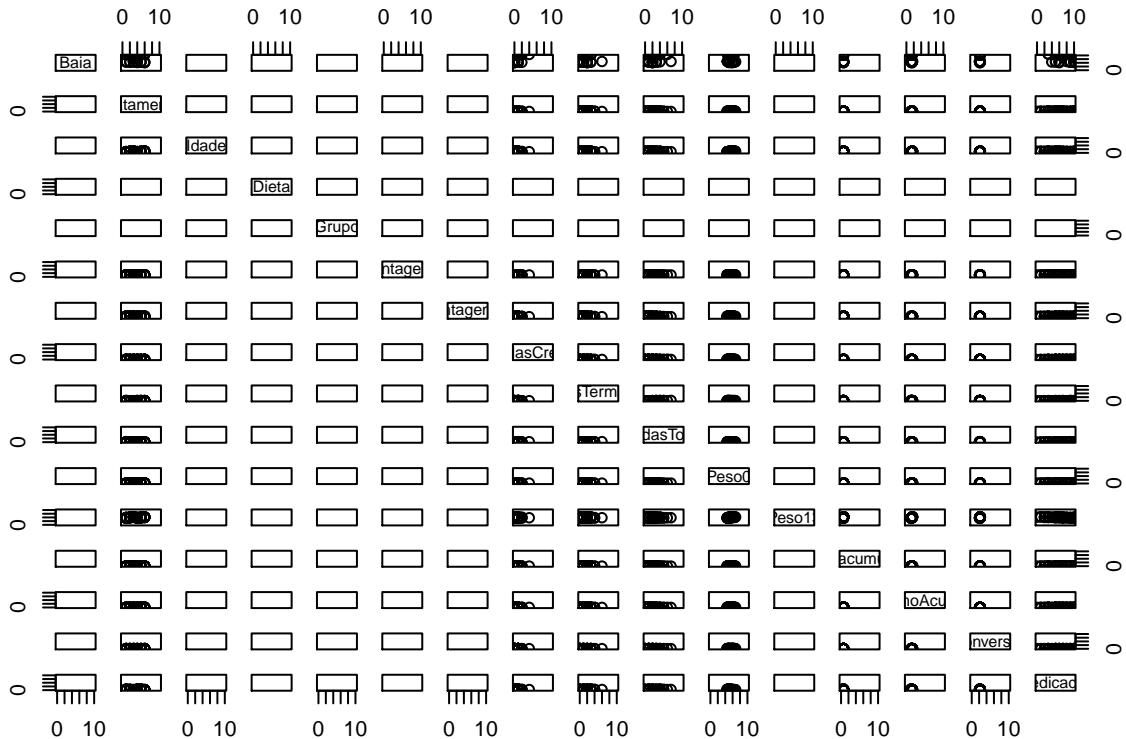
### 2.4.1 Plot simples

O R tem como nativo a função `plot()`. Com ela é possível fazer uma série de gráficos que permitem uma melhor compreensão dos dados. Podemos simplesmente aplicar `plot()` à todos os dados.

```
library(readxl)
idades_dietas <- read_excel("Dados/dados_idades_dietas.xlsx")
plot(idades_dietas, ylim=c(0,200), xlim=c(0,10))
```

## Warning in data.matrix(x): NAs introduzidos por coerção

## Warning in data.matrix(x): NAs introduzidos por coerção

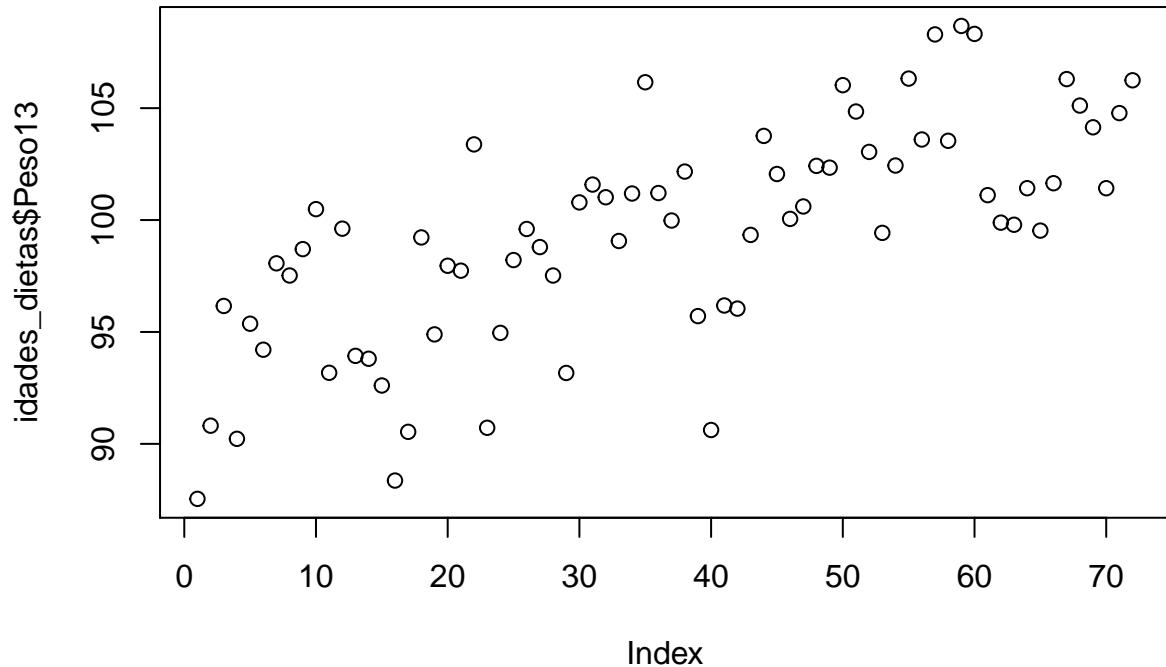


Claramente, não foi uma boa opção. Podemos restringir à uma ou duas variáveis. Assim, fica mais fácil vizualizar. A função `plot()` tem como principal argumento o `y`. Então se plotarmos uma única variável, para o eixo x será criado um índice que vai de 1 até o tamanho do vetor `y`. Por exemplo,

```
colnames(idades_dietas)
```

```
## [1] "Baia"           "Tratamento"      "Idade"          "Dieta"
## [5] "Grupo"          "Contagem0"       "Contagem13"     "PerdasCreche"
## [9] "PerdasTerminação" "PerdasTotais"    "Peso0"          "Peso13"
## [13] "GPDacumulado"   "ConsumoAcumulado" "Conversão"      "Medicados"
```

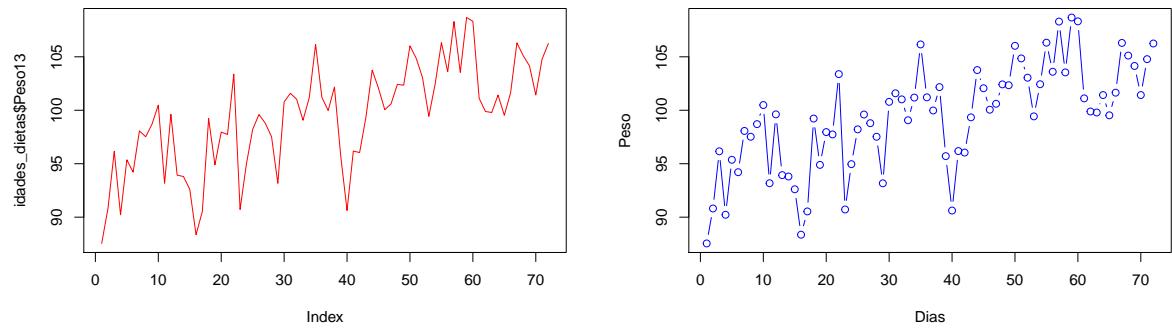
```
plot(idades_dietas$Peso13)
```



#### 2.4.1.1 Parâmetros do `plot()`

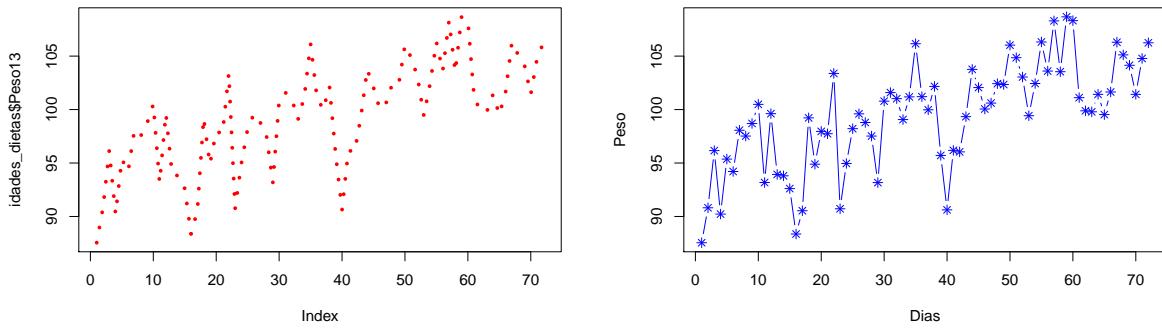
Podemos mudar o tipo (veja opções para `type`), a cor do gráfico (veja opções para `col`) e os *labels*.

```
plot(idades_dietas$Peso13,type="l", col='red')
plot(idades_dietas$Peso13,type="b", col='blue',xlab = "Dias",ylab="Peso")
```



Podemos alterar outros parâmetros para customizar o gráfico, como a grossura da linha `lwd`, o tipo de linha `lty` e o `pch` que é o formato do marcador.

```
plot(idades_dietas$Peso13,type="l", col='red',lwd=4,lty=3)
plot(idades_dietas$Peso13,type="b", col='blue',xlab = "Dias",ylab="Peso",pch=8)
```

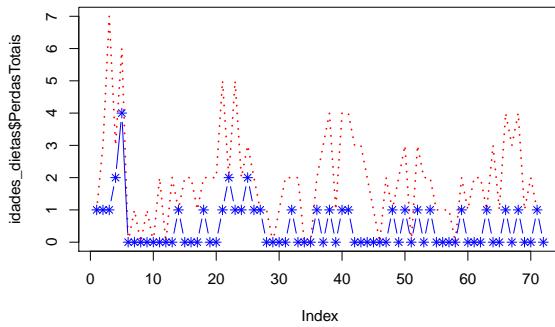


Outros parâmetros do `plot()` podem ser encontrado em r-graph-gallery.

#### 2.4.1.2 Multiplas variáveis em um gráfico

Podemos adicionar uma segunda variável à um gráfico existente. Nesse caso, para o segundo gráfico devemos usar a função `lines`.

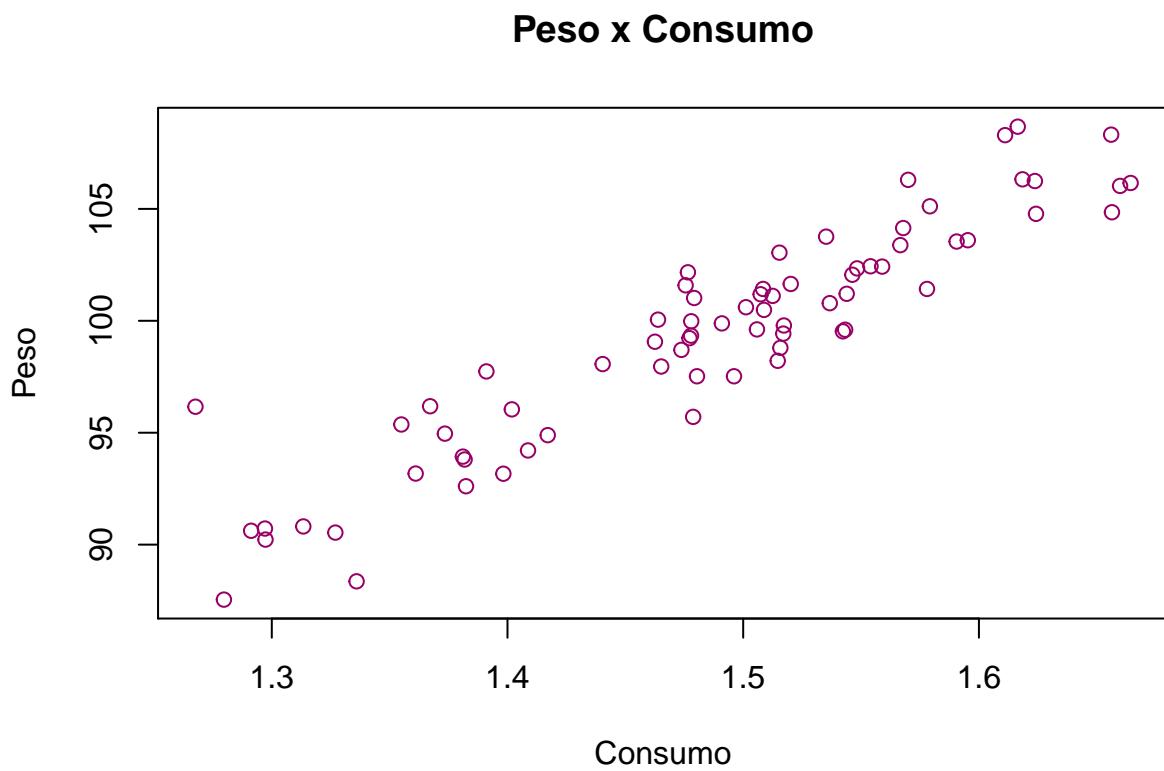
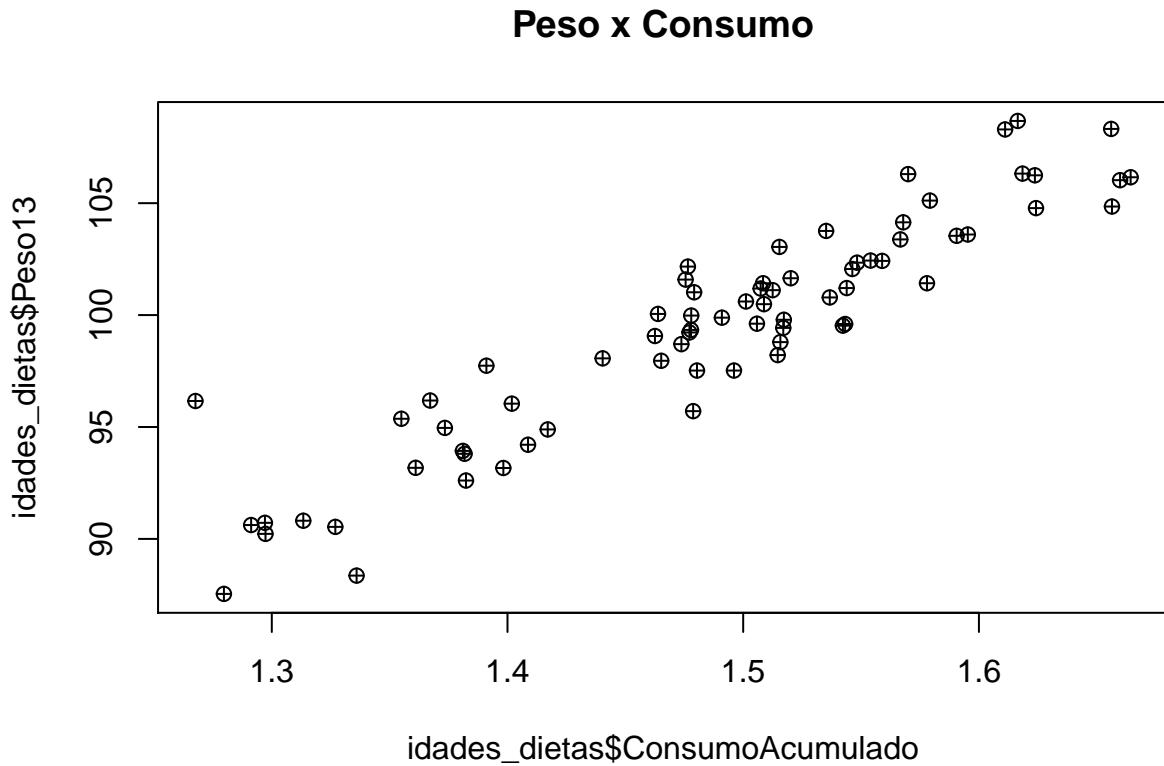
```
plot(idades_dietas$PerdasTotais,type="l", col='red',lwd=2,lty=3,pch=4)
lines(idades_dietas$PerdasCreche,type="b", col='blue',xlab = "Dias",ylab="Peso",pch=8)
```



#### 2.4.2 Scatterplot

Para ver a relação entre duas variáveis podemos usar a função `plot()` atribuindo um `x` e um `y`. Além disso a cor do gráfico pode ser especificada em uma escala *RGB*. Nesse exemplo usamos a numeração “#990066” para especificar uma cor. Além disso, adicionamos um título ao gráfico usando o parâmetro `main()`.

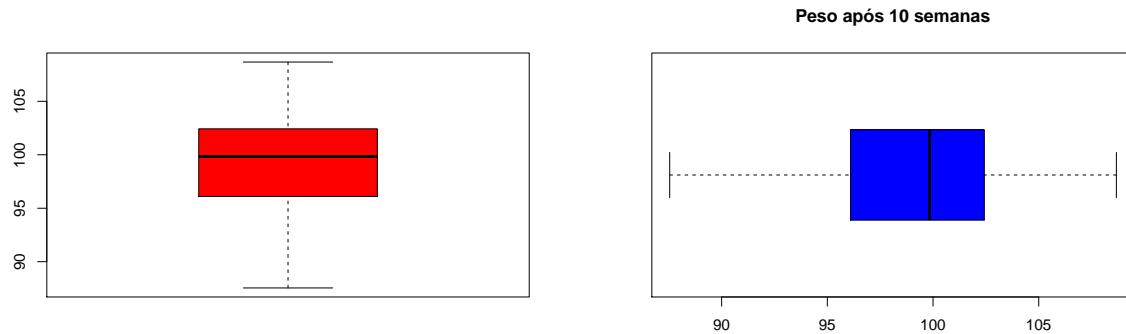
```
plot(idades_dietas$ConsumoAcumulado,idades_dietas$Peso13,main="Peso x Consumo",pch=10)
plot(idades_dietas$ConsumoAcumulado,idades_dietas$Peso13,col="#990066",type="p" ,ylab="Peso",xlab="Consumo")
```



### 2.4.3 Boxplot simples

O boxplot é uma das ferramentas mais usadas para vizualizar como os dados se distribuem. Vamos começar pela versão mais simples usando a função `boxplot()`

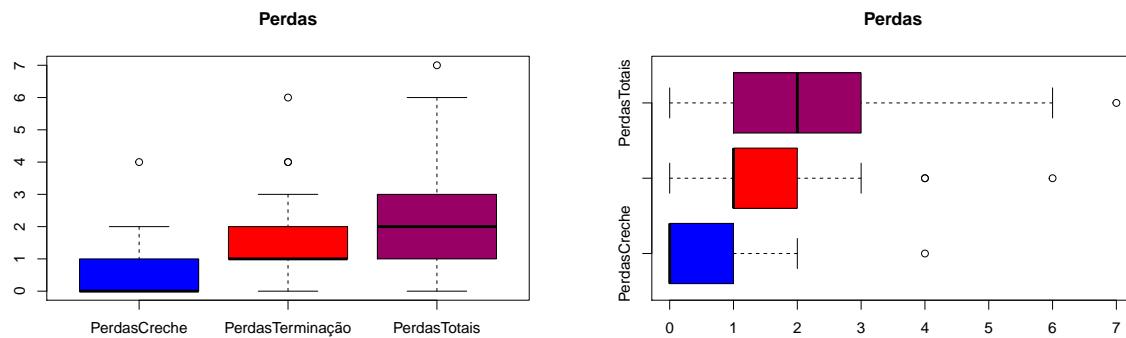
```
boxplot(idades_dietas$Peso13, col='red')
boxplot(idades_dietas$Peso13, type="b", col='blue', horizontal=TRUE, las=1, main="Peso após 10 semanas")
```



### 2.4.4 Box plot multiplas variáveis

Podemos colocar várias variáveis na mesma janela. Nesse caso podemos usar a função `c()` para determinar os parâmetros para cada gráfico. Nesse exemplo usamos as colunas “*PerdasCreche*”, “*PerdasTerminação*” e “*PerdasTotais*” do banco de dados `idades_dietas`.

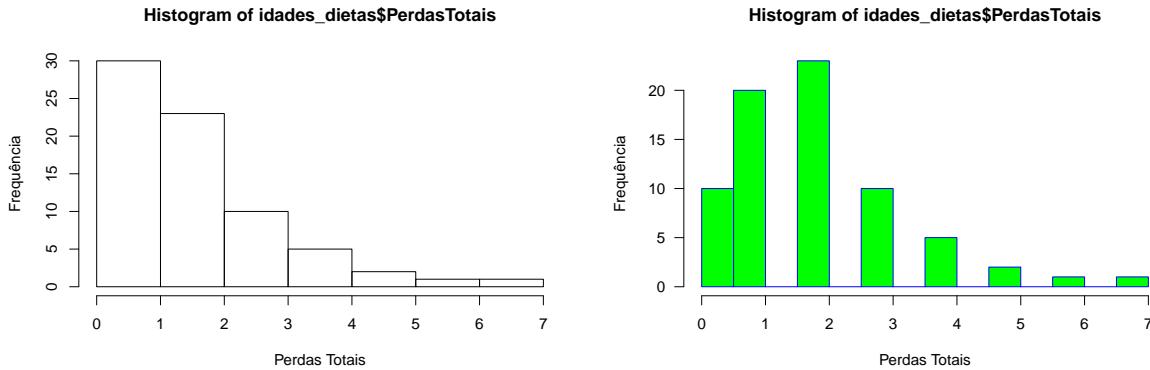
```
boxplot(idades_dietas[,c("PerdasCreche", "PerdasTerminação", "PerdasTotais")], col=c('blue', 'red', "#990066"))
boxplot(idades_dietas[,c("PerdasCreche", "PerdasTerminação", "PerdasTotais")], col=c('blue', 'red', "#990066"))
```



### 2.4.5 Histogramas

Para fazer um histograma a função nativa do R é `hist()`. Como *default* ela retorna a frequência dos dados em um número de classes determinada pela regra de Sturges. No entanto é possível alterar o número de classes através do parâmetro `breaks`.

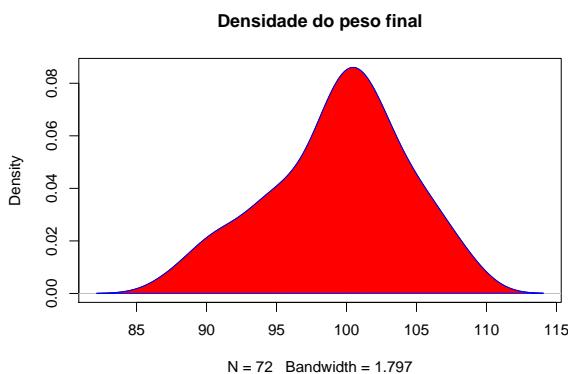
```
hist(idades_dietas$PerdasTotais, xlab="Perdas Totais", ylab="Frequência")
hist(idades_dietas$PerdasTotais, xlab="Perdas Totais", ylab="Frequência", border='blue', col="green", las=1, bre
```



## 2.4.6 Gráfico de densidades

Para ver a distribuição dos dados podemos plotar a densidade e assim ter uma ideia do comportamento dos dados em termos da distribuição de probabilidade. A função `density()` pode ser usada para esse fim.

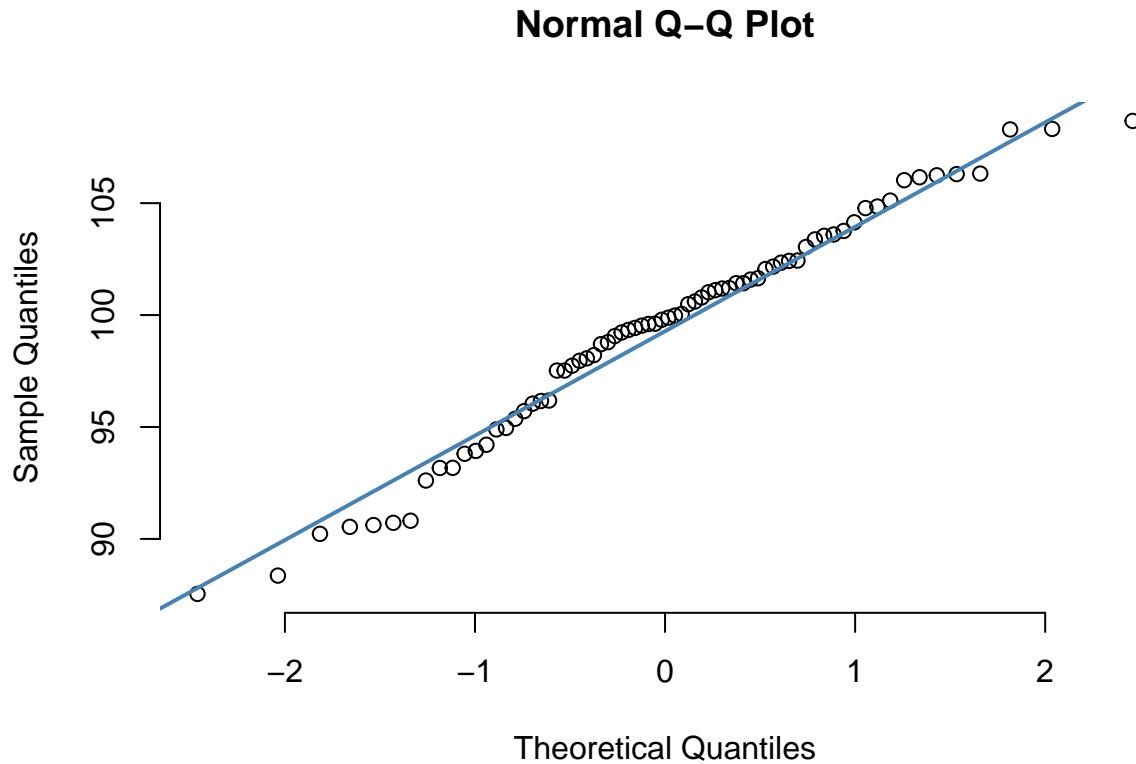
```
d <- density(idades_dietas$Peso13) # retorna a densidade dos dados
plot(d,main="Densidade do peso final") #
plot(d,main="Densidade do peso final")
polygon(d, col="red", border="blue")
```



## 2.4.7 QQplot

O gráfico chamado *quantile quantile plot* ou *qqplot* tem por objetivo comparar os quantis de um distribuição de probabilidade com os quantis amostrais de um conjunto de dados. Assim, se ao plotar os quantis amostrais contra os quantis teóricos da distribuição correta dos dados, teremos pontos próximos à uma reta com um ângulo de 45 graus. Nesse exemplo vamos comparar os quantis da variável Peso13 com os quantis teóricos da distribuição Normal.

```
qqnorm(idades_dietas$Peso13, pch = 1, frame = FALSE)
qqline(idades_dietas$Peso13, col = "steelblue", lwd = 2)
```

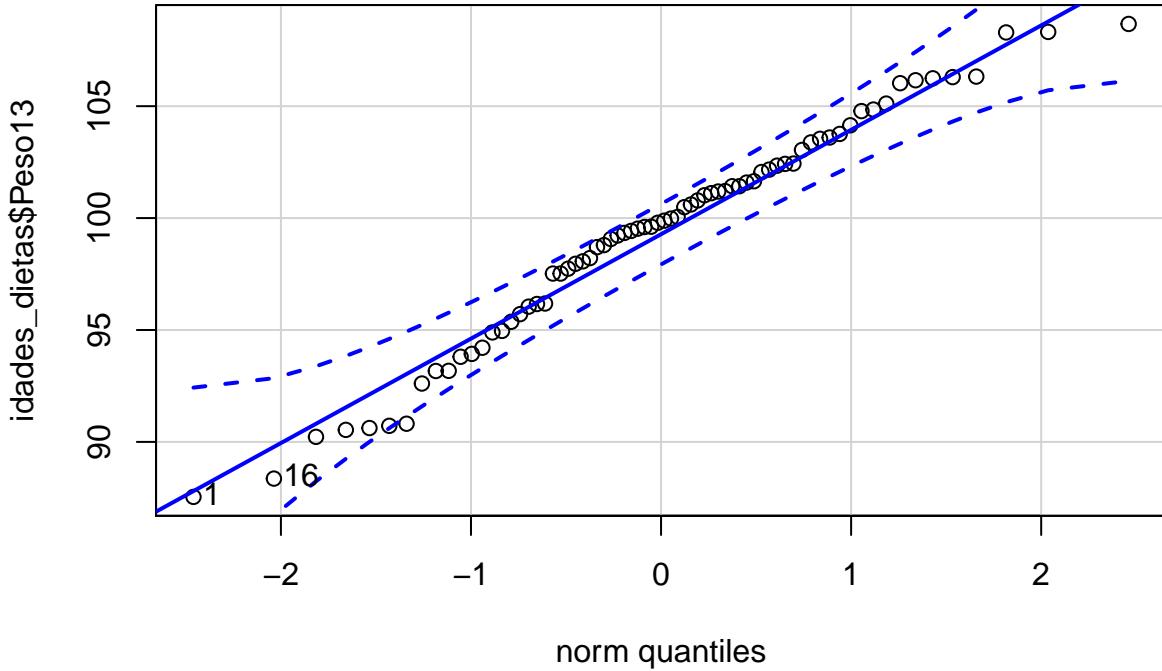


A assimetria vista no gráfico de densidades fica visível no qqplot nos pontos fora da reta, principalmente no quantis mais extremos (próximos à -3 e 3).

Se usamos o pacote `car` podemos colocar um intervalo de confiança para o qqplot normal e podemos perceber que estamos no limiar entre normalidade e não normalidade dos dados.

```
library("car")
qqPlot(idades_dietas$Peso13)
```

```
## [1] 1 16
```



## 2.5 Análise estatística de dados - Modelagem

Faremos uma breve introdução aos modelos mais usados em estatística. Modelos mais complexos e avançados serão abordados posteriormente.

### 2.5.1 Regressão linear simples

Na regressão linear é importante destacar que temos uma variável dependente, geralmente chamada de  $y$  e uma variável independente,  $x$ . Objetivo é ver como a variação em  $x$  afeta  $y$ . O modelo matemático de regressão linear pode ser escrito como

$$y_i = b_0 + b_1 x_i + e_i$$

em que  $b_0$  é o intercepto,  $b_1$  é o coeficiente da regressão, parâmetro importante para entender a relação ou o efeito que  $x$  tem sobre  $y$ . O termo de erro  $e_i$  é assumido ser aleatório e por questões de inferência (teste de hipóteses) assumosição de distribuição normal é requerida.

```
modelo=lm(Peso13~ConsumoAcumulado,data=idades_dietas)
modelo
```

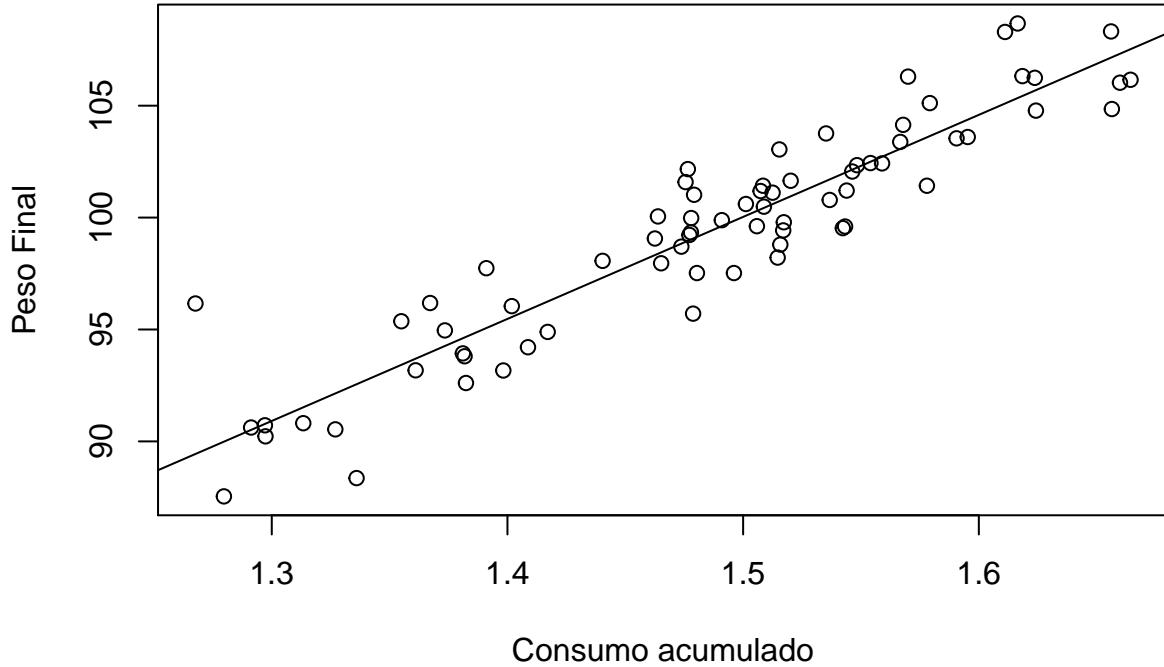
```
##
## Call:
## lm(formula = Peso13 ~ ConsumoAcumulado, data = idades_dietas)
##
## Coefficients:
```

```

##      (Intercept) ConsumoAcumulado
##            31.65          45.59

plot(idades_dietas$ConsumoAcumulado,idades_dietas$Peso13,ylab='Peso Final',xlab='Consumo acumulado')
abline(modelo)

```



Para mais informações sobre o modelo, como significância do parâmetro, estatística  $t$  e  $F$ , podemos usar o `summary()`

```
summary(modelo)
```

```

##
## Call:
## lm(formula = Peso13 ~ ConsumoAcumulado, data = idades_dietas)
##
## Residuals:
##    Min     1Q   Median     3Q    Max 
## -4.1928 -1.3089 -0.0593  0.9056  6.7256 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 31.648     3.262   9.703 1.38e-14 ***
## ConsumoAcumulado 45.588     2.190  20.813 < 2e-16 ***
## --- 
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
## 
## Residual standard error: 1.855 on 70 degrees of freedom
## Multiple R-squared:  0.8609, Adjusted R-squared:  0.8589 
## F-statistic: 433.2 on 1 and 70 DF,  p-value: < 2.2e-16

```

No `summary()` estão várias informações a respeito do ajuste do modelo. Entre elas, o  $R^2$  (*Multiple R-squared*) e o Erro Padrão Residual (Residual standard error)

```
summary(modelo)$r.squared # R^2
## [1] 0.8608836
summary(modelo)$sigma      # Erro Padrão Residual
## [1] 1.855462
```

### 2.5.1.1 Coeficientes

Para acessar os coeficientes do modelo pode-se usar a função `coefficients()`

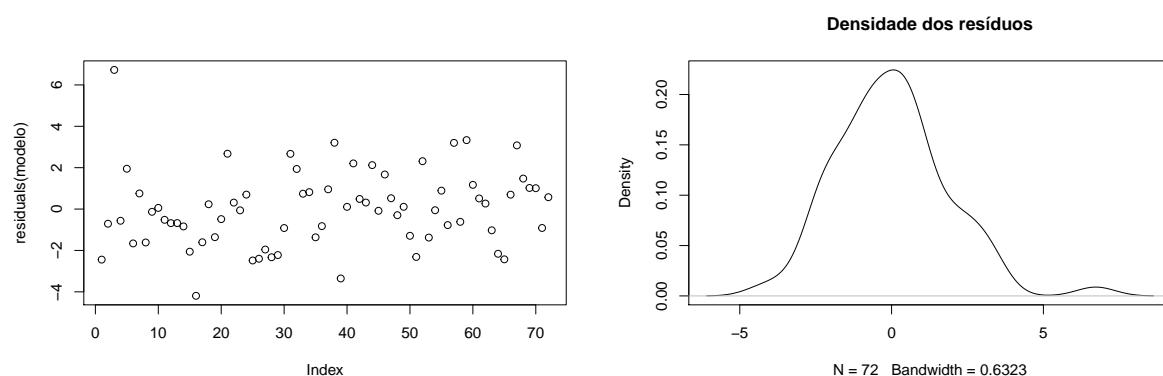
```
coefficients(modelo)
##           (Intercept) ConsumoAcumulado
##             31.64777        45.58832
```

### 2.5.1.2 Intervalos de confiança dos parâmetros

```
confint(modelo)
##                   2.5 %   97.5 %
## (Intercept)    25.14237 38.15318
## ConsumoAcumulado 41.21972 49.95691
confint(modelo, level=0.9)
##                   5 %   95 %
## (Intercept)    26.21067 37.08487
## ConsumoAcumulado 41.93712 49.23951
```

### 2.5.1.3 Resíduos

```
plot(residuals(modelo))
plot(density(modelo$residuals), main="Densidade dos resíduos")
```

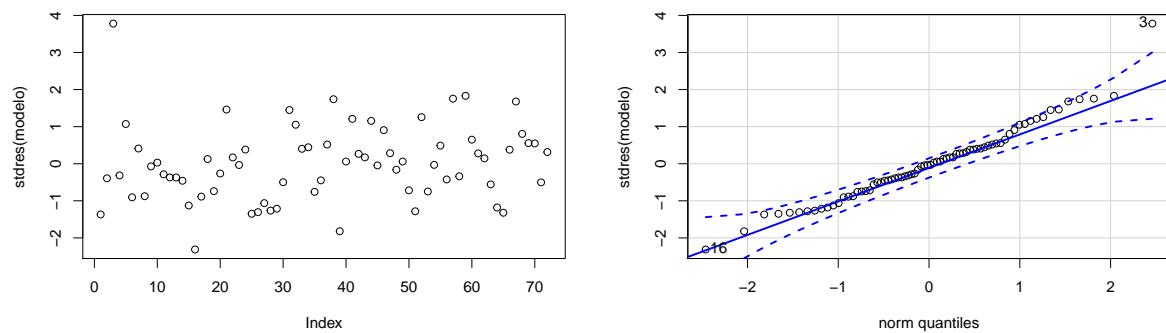


### 2.5.1.4 Resíduos padronizados

Resíduos padronizados podem ser comparados à normal padrão, que possui média zero e desvio padrão 1. Sabe-se que nessa distribuição os valores estão entre -3 e 3 com uma probabilidade de 0.9973. Assim, espera-se que se o modelo está bem ajustado, a maioria dos valores do resíduo padronizado esteja entre -3 e 3.

```
library(MASS)
library("car")
plot(stdres(modelo))
qqPlot(stdres(modelo))
```

```
## [1] 3 16
```



### 2.5.2 Regressão Múltipla

Quando temos mais de uma variável independente ou preditoras  $x_1$  e  $x_2$ , podemos usar a função `lm()` para verificar o efeito dessas variáveis sobre a variável dependente  $y$ . O modelo matemático para essa situação pode ser descrito como

$$y_i = b_0 + b_1 x_{i1} + b_2 x_{i2} + e_i$$

em que os coeficientes/parâmetros  $b_j$ ,  $j = 1, 2$  medem o efeito das variáveis preditoras sobre a variável dependente  $y$ . Os erros são assumidos sendo independentes e tendo distribuição normal padrão para efeito de inferência sobre os parâmetros  $b_j$ ,  $j = 1, 2$ .

```
modelo2=lm(Peso13~ConsumoAcumulado+Dieta,data=idades_dietas)
summary(modelo2)
```

```
##
## Call:
## lm(formula = Peso13 ~ ConsumoAcumulado + Dieta, data = idades_dietas)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -4.2515 -1.2540 -0.0244  0.9001  6.8204 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 31.4282    3.3492   9.384 6.01e-14 ***
## ConsumoAcumulado 45.6867    2.2244  20.538 < 2e-16 ***
## 
```

```

## DietaB           0.1468      0.4441    0.331     0.742
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.867 on 69 degrees of freedom
## Multiple R-squared:  0.8611, Adjusted R-squared:  0.8571
## F-statistic: 213.9 on 2 and 69 DF,  p-value: < 2.2e-16

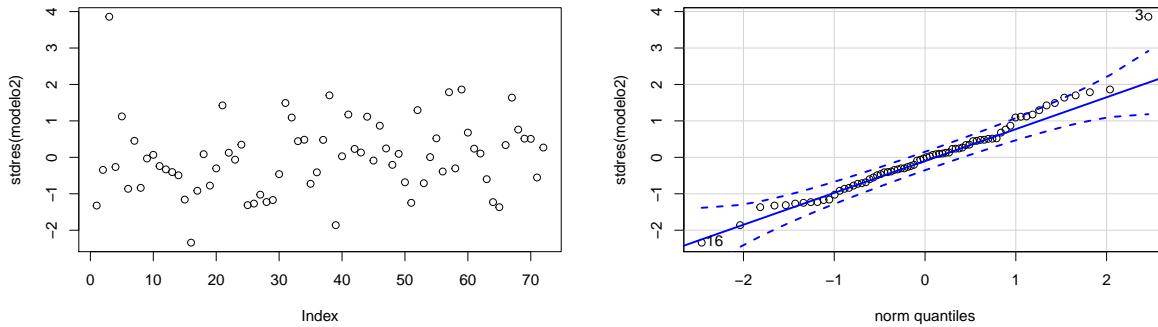
```

```

library(MASS)
library("car")
plot(stdres(modelo2))
qqPlot(stdres(modelo2))

```

```
## [1] 3 16
```



Uma outra forma mais resumida de apresentar os resultados do modelo de regressão múltipla é usando o pacote `pander`.

```

library(pander)
sm2 <- summary(modelo2)
pander(sm2, keep.line.breaks = TRUE)

```

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	31.43	3.349	9.384	6.005e-14
ConsumoAcumulado	45.69	2.224	20.54	4.131e-31
DietaB	0.1468	0.4441	0.3306	0.7419

Table 2.2: Fitting linear model: Peso13 ~ ConsumoAcumulado + Dieta

Observations	Residual Std. Error	R <sup>2</sup>	Adjusted R <sup>2</sup>
72	1.867	0.8611	0.8571

# Chapter 3

## Visualização de dados com o R

Uma revolução na vizualização de dados aconteceu com a criação do pacote `ggplot2` de autoria de Hadley Wickham. A partir da definição de gráfico apresentada em Wilkinson and others (2005), uma nova concepção de gráfico foi proposta em Wickham (2010), que basicamente divide o gráfico em camadas que podem ser construídas por etapas.

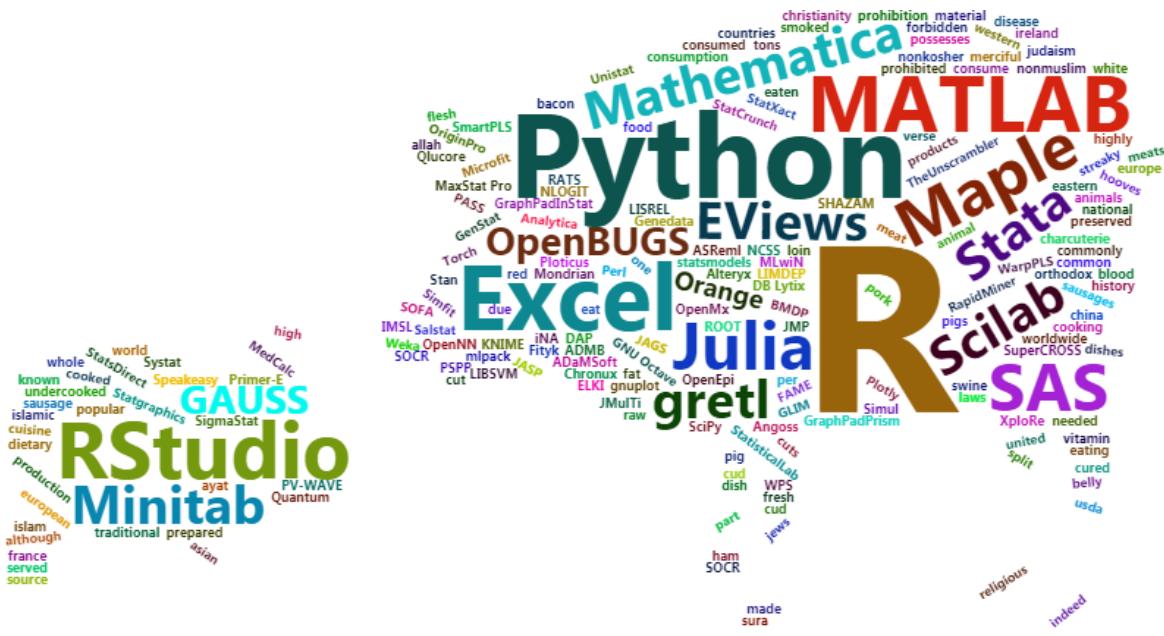


Figure 3.1: Veja bem

Essa abordagem torna a construção do gráfico intuitiva e permite a criação de gráficos bonitos, padronizados, flexíveis e com poucas linhas de código. Para isso é necessário a instalação do pacote `ggplot2`.

```
install.packages("ggplot2")
library(ggplot2)
library(tidyverse)
```

### 3.1 Gráficos com o ggplot2

Os gráficos do ggplot2 são construídos camada por camada. Assim, a ideia é ir adicionando novos elementos através do operador `+`. Para exemplificar o uso do ggplot2 vamos usar os dados de *dietas* e *idades*.

```
g1=ggplot(data = idades_dietas)
g1
```

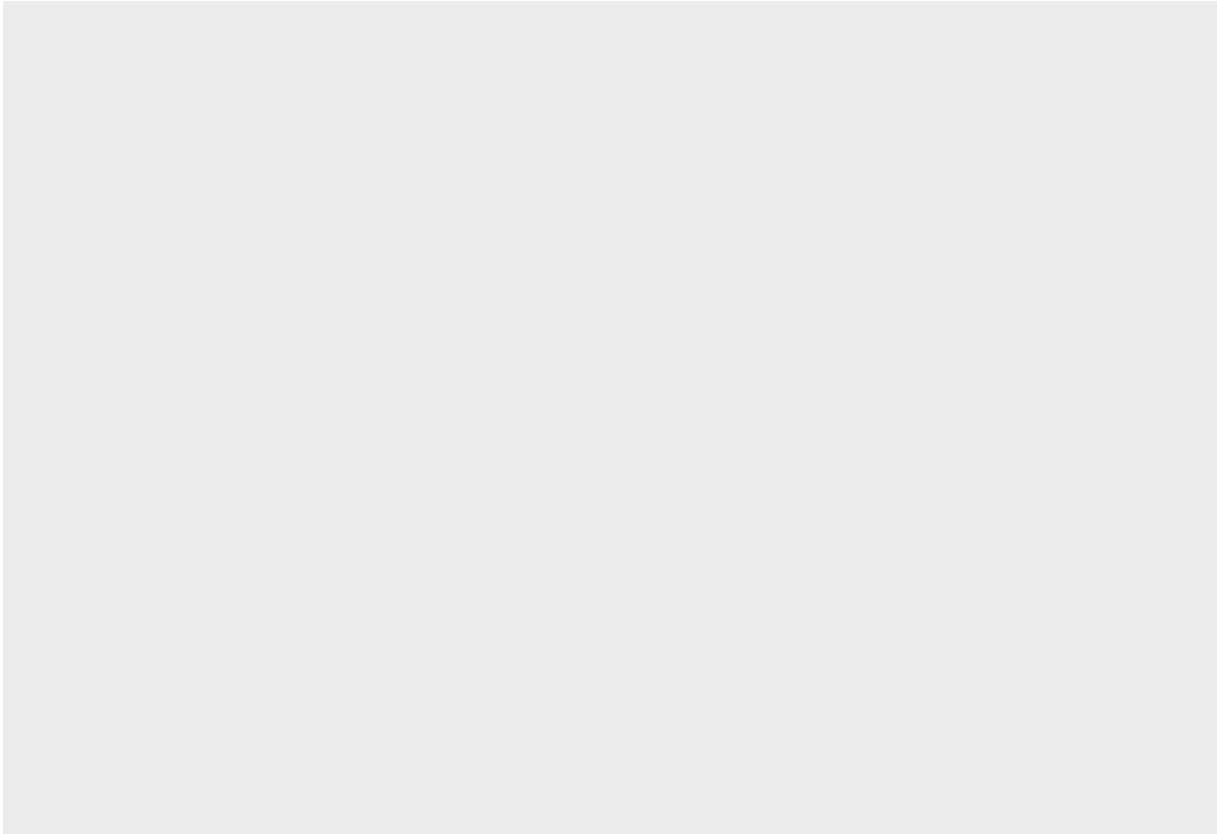


Figure 3.2: Base de um gráfico no ggplot

Essa é a primeira camada do gráfico gerada pela função `ggplot`. Agora novos elementos podem ser adicionados ao gráfico `g1`.

### 3.2 Funções geom e aes

Duas funções fundamentais no conceito `ggplot` são a função `geom()` responsável pela especificação da forma geométrica utilizada no mapeamento das observações e a função `aes()` que especifica quais variáveis serão mapeadas no gráfico.

```
g1 <- g1 + geom_point(aes(x=ConsumoAcumulado, y=Peso13))
g1
```

Com a segunda camada é feita usando-se o `geom_point` a qual adiciona pontos ao gráfico `g1` de acordo com as variáveis especificadas no `aes()`.

Podemos personalizar o gráfico mexendo nas cores, lables, etc.

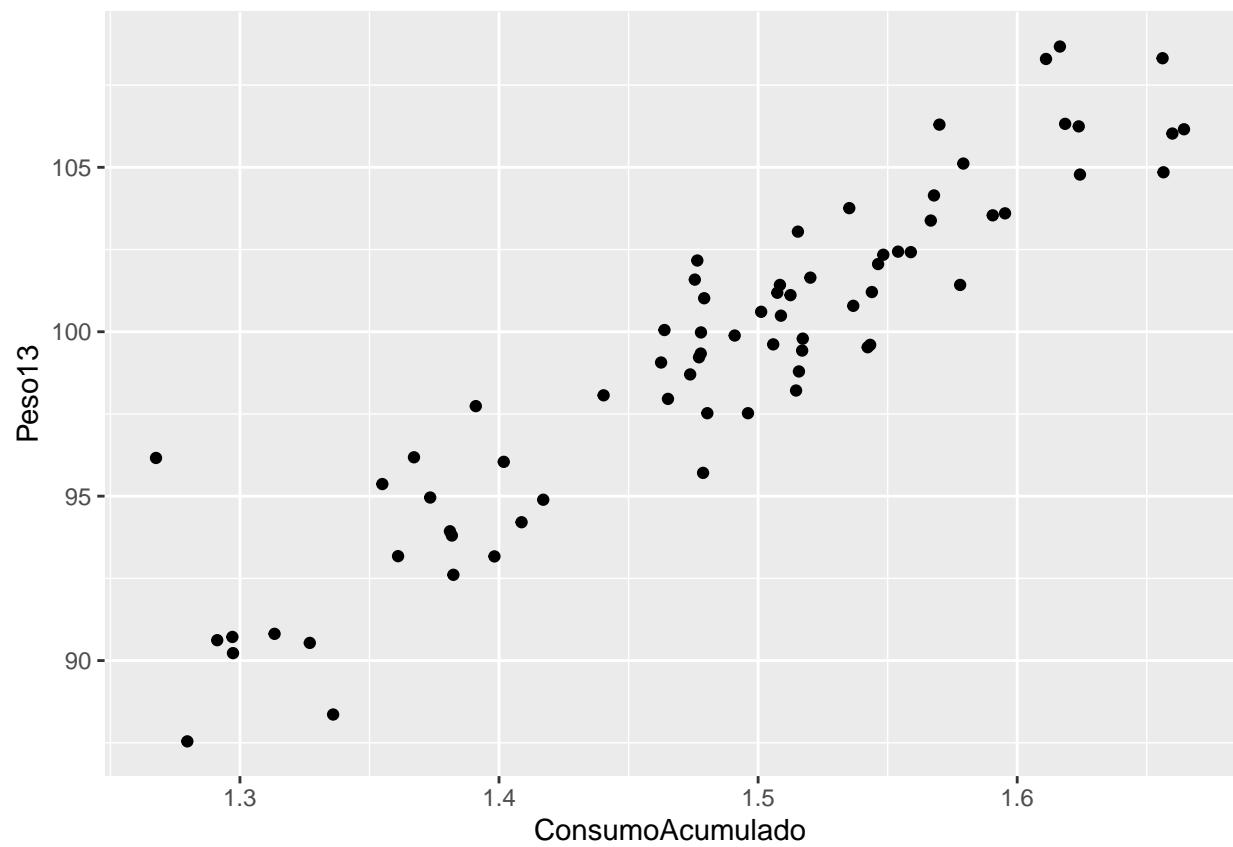
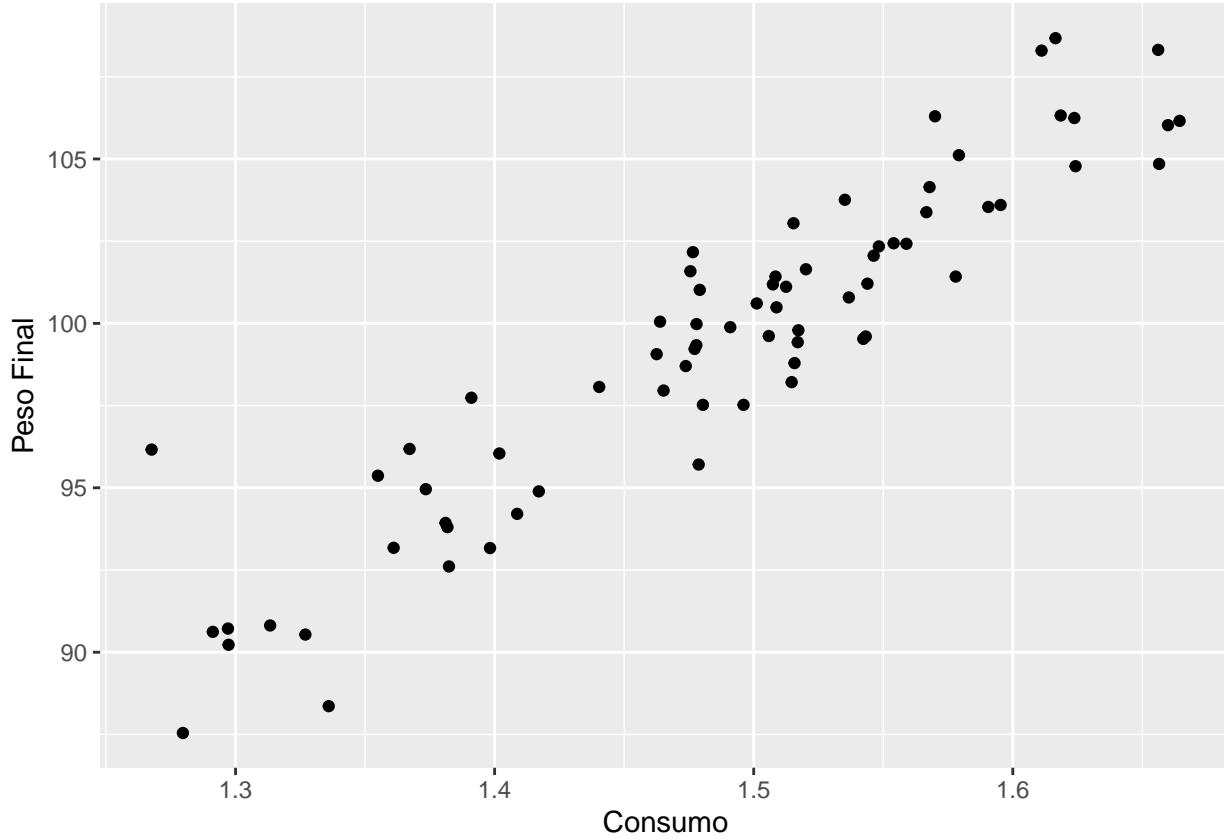


Figure 3.3: Scatter plot do Peso pelo Consumo

```
g1 <- g1 + labs(x="Consumo",y="Peso Final")
g1
```



Pode-

se alterar parâmetros como `colour` que muda a cor do objeto plotado, `shape` que altera o formato do objeto plotado, `size` que determina o tamanho do objeto e `alpha` que muda a intensidade da cor.

```
ggplot(data = idades_dietas,aes(x=ConsumoAcumulado, y=Peso13))+
  geom_point(colour = "blue", shape = 4, size = 2, alpha = 2.5)+  labs(x="Consumo",y="Peso Final")

ggplot(data = idades_dietas,aes(x=ConsumoAcumulado, y=Peso13))+
  geom_point(colour = "blue", shape = 4, size = 8, alpha = 0.5)+  labs(x="Consumo",y="Peso Final")
```

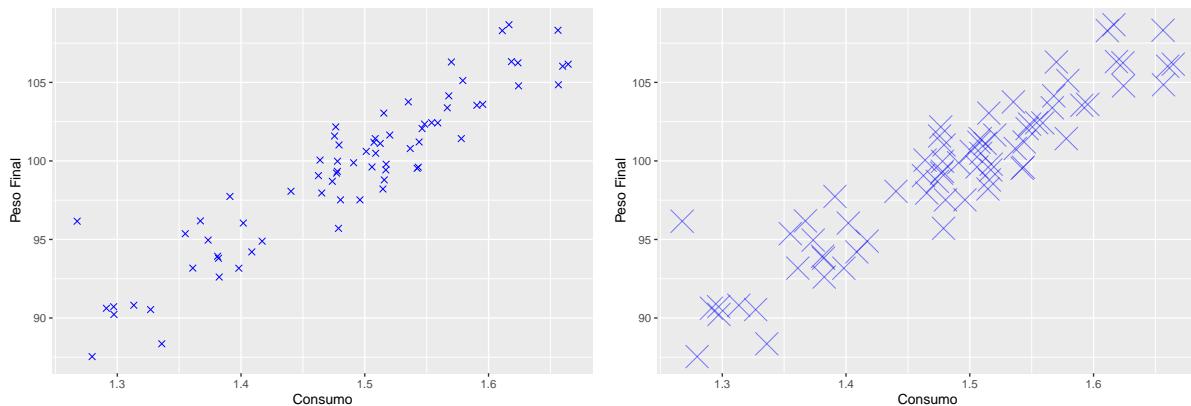


Figure 3.4: Colorindo e alterando os objetos plotados

### 3.2.1 Estética

Com a função `aes()` (de *aesthetics*) podemos alterar a cor, tipo de gráfico, relacionar as variáveis no gráfico à fatores no conjunto de dados. Esses parâmetros também podem (recomendável) ser passados diretamente para a função `ggplot()`. Por exemplo, podemos colorir o gráfico de acordo com o fator *Dieta*.

```
g2 <- ggplot(data=idades_dietas,aes(x=ConsumoAcumulado, y=Peso13,color=Dieta)) +
  geom_point()+
  labs(x="Consumo",y="Peso Final")
g2
```

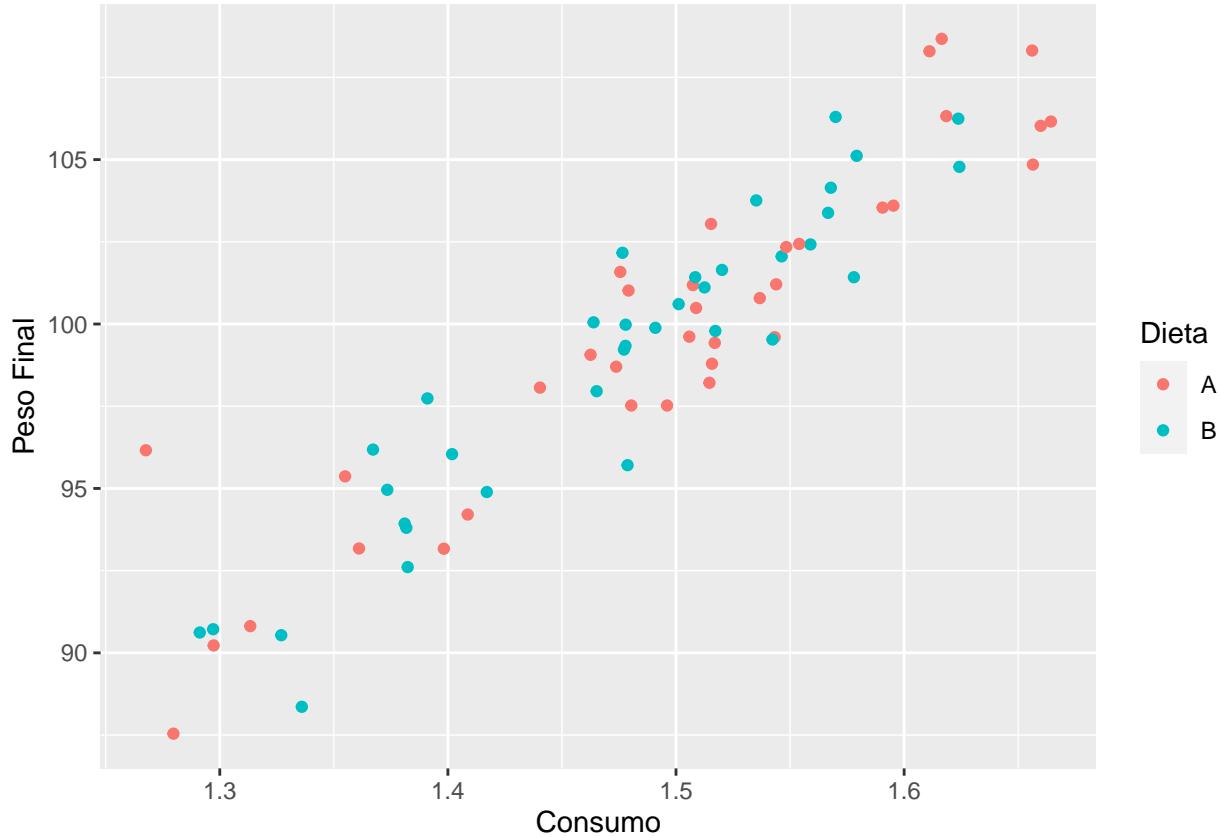


Figure 3.5: Colorindo o gráfico de acordo com os níveis do fator Dieta

Agora vamos colorir pelo fator *Grupo*.

```
g3 <- ggplot(data=idades_dietas,aes(x=ConsumoAcumulado,y=Peso13,color=Grupo)) +
  geom_point()+
  labs(x="Consumo",y="Peso Final")
g3
```

### 3.2.2 Curva de tendência

Uma curva de tendência, com ou sem margem de erro, pode ser adicionada usando o método de regressão linear.

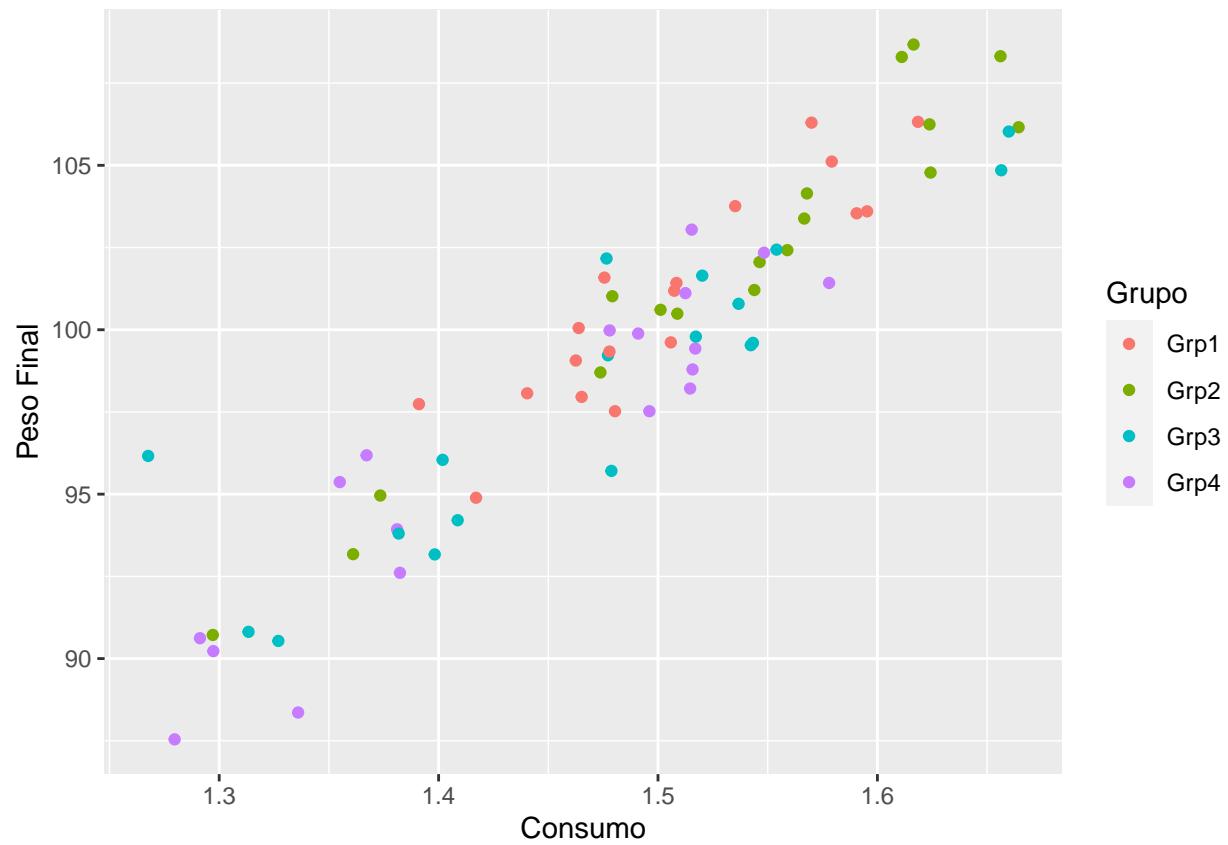


Figure 3.6: Colorindo o gráfico de acordo com os níveis do fator Grupo

```
g2 <- g2 + geom_smooth(method = "lm", se = TRUE)
g2
g3 <- g3 + geom_smooth(method = "lm", se = FALSE)
g3
```

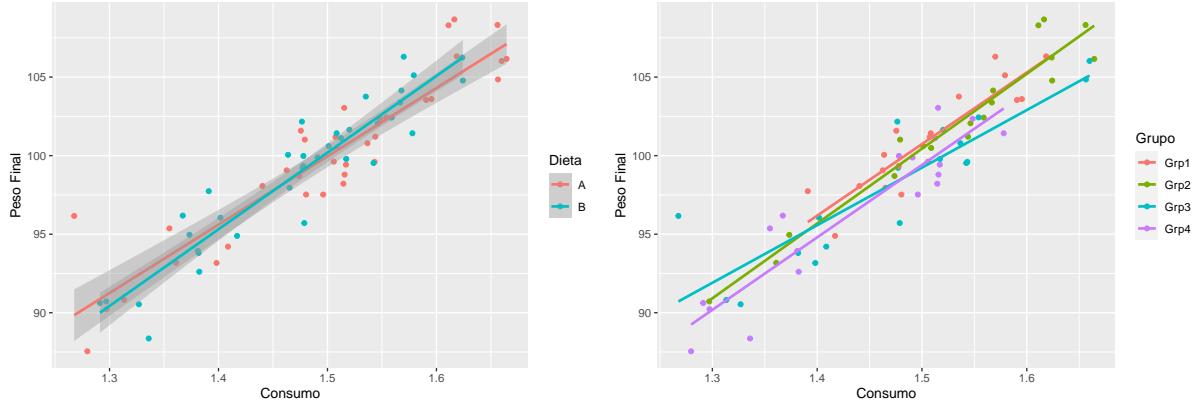


Figure 3.7: Gráfico com tendência com e sem intervalo de confiância

### 3.3 Segmentação e junção de gráficos do ggplot

#### 3.3.1 Segmentação

Para separar o gráfico de *Peso x Consumo*, por Dieta, por exemplo, usamos a função `facet_wrap()`

```
g4 <- ggplot(data=idades_dietas,aes(x=ConsumoAcumulado,y=Peso13,color=Dieta)) +
  geom_point()+
  labs(x="Consumo",y="Peso Final")+
  facet_wrap(~ Dieta)
g4
```

#### 3.3.2 Junção de gráficos

O pacote `patchwork` facilita imensamente a junção de dois ou mais gráficos.

```
install.packages("patchwork")
library("patchwork")
```

```
g1 + g2
```

Para colocar os gráfico em colunas, pode-se usar a função `plot_layout(ncol = 1)`

```
g3 + g4 + plot_layout(ncol = 1)
```

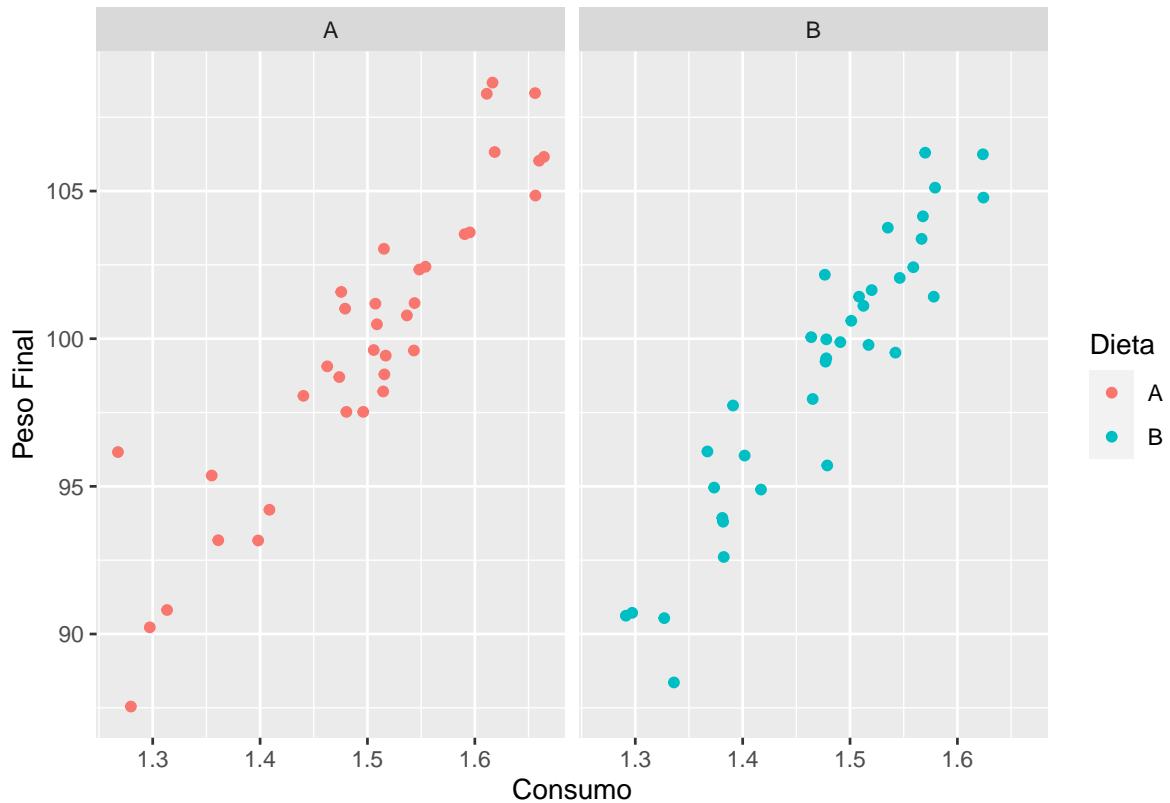
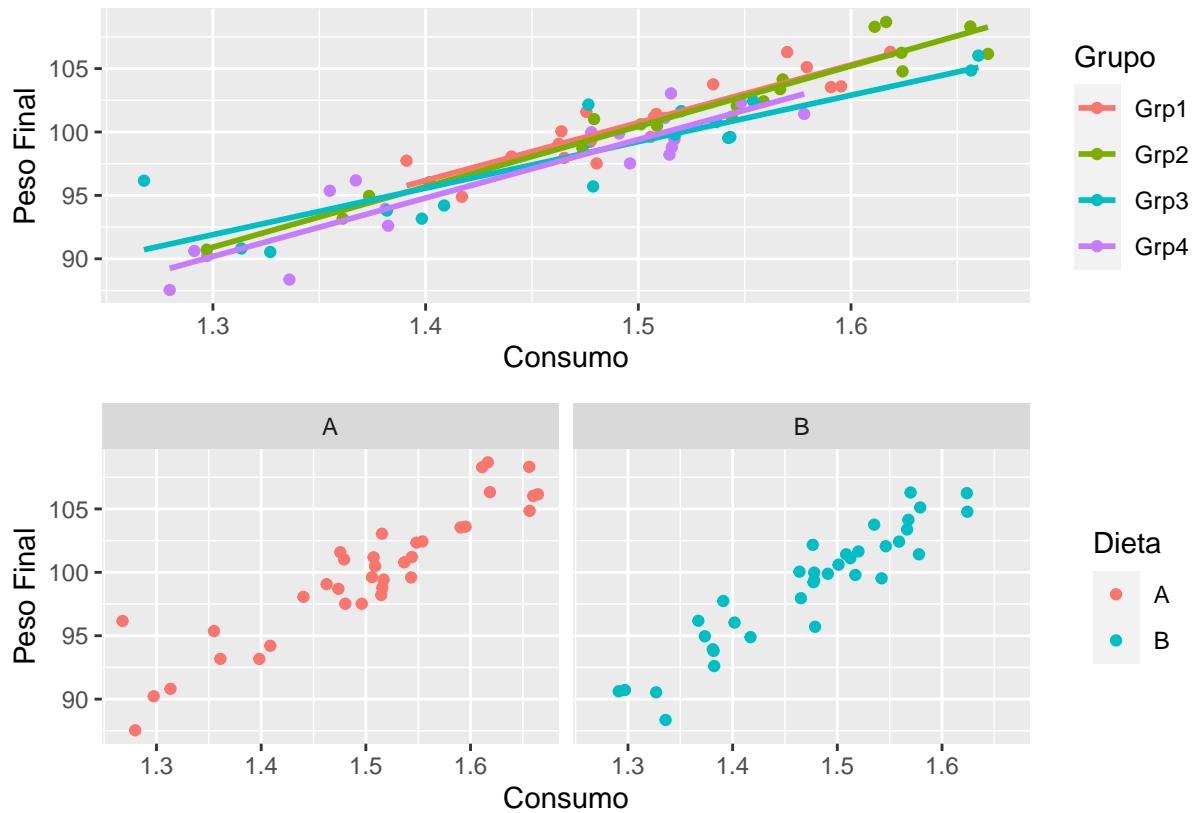


Figure 3.8: Separando gráficos de acordo com a Dieta



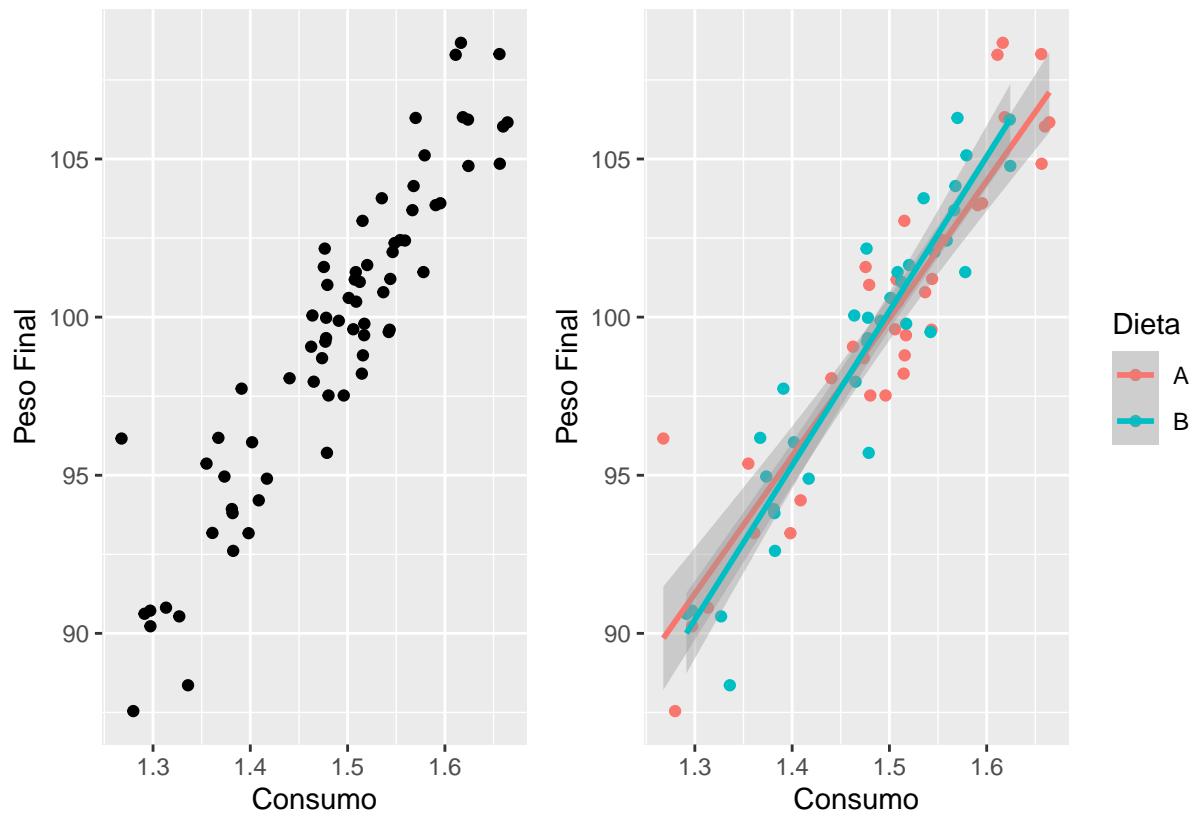


Figure 3.9: Juntando dos gráficos de maneira fácil

### 3.3.3 Salvando um gráfico

Além da opção *export* disponível no Tools, podemos usar uma linha de comando para salvar um gráfico no diretório de trabalho. Algumas opções podem ser modificadas, como `scale`, `width` e `height`. No seguinte exemplo, essas opções ficaram `default`.

```
ggsave(filename = "meugrafico.png", plot = g4)
```

## 3.4 Diferentes tipos de gráficos

Existem várias variações da função `geom`. dentre as mais usadas podemos destacar:

- `geom_abline` - para retas definidas por um intercepto e uma inclinação.
- `geom_hline` - para retas horizontais.
- `geom_boxplot` - para boxplots.
- `geom_density` - para densidades.
- `geom_area` - para áreas.
- `geom_bar` - para barras.
- `geom_histogram` - para histogramas.

### 3.4.1 Linhas, retas horizontais e verticais

Nesse exemplo é *plotado* inicialmente um gráfico de linhas representando a relação entre *Peso13* e *ConsumoAcumulado*. Em cima desse gráfico, foi construído retas horizontal e vertical que representam as médias das respectivas variáveis nos eixos x e y. Nesse caso, a linha horizontal em vermelho representa a média da variável *Peso13* enquanto a reta vertical em azul representa a média da variável *ConsumoAcumulado*.

```
ggplot(data=idades_dietas)+  
  geom_line(aes(x=ConsumoAcumulado,y=Peso13))+  
  geom_vline(aes(xintercept = mean(ConsumoAcumulado)),size=1,col='blue') +  
  theme(legend.position = "none") + #remove legenda  
  geom_hline(aes(yintercept = mean(Peso13)),size=1.5,col='red')
```

### 3.4.2 Box plot

Para fazer um box plot usa-se a função `geom_boxplot()`. Parâmetros como `fill` e `col` ajudam a colorir o gráfico.

```
bp1=ggplot(data=idades_dietas)+  
  geom_boxplot(aes(x=Dieta,y=Peso13,fill=Dieta))  
bp2=ggplot(data=idades_dietas)+  
  geom_boxplot(aes(x=Grupo,y=Peso13,fill=Grupo))  
bp1  
bp2
```

#### 3.4.2.1 Legendas do Box plot

```
bp2+guides(fill = FALSE) #remove legenda  
bp2+ scale_fill_manual(values=c("#CC9900", "#FF3399", "#CCFF00", "#339999"),name="GRUPO", labels=c("G1", "G2"))
```

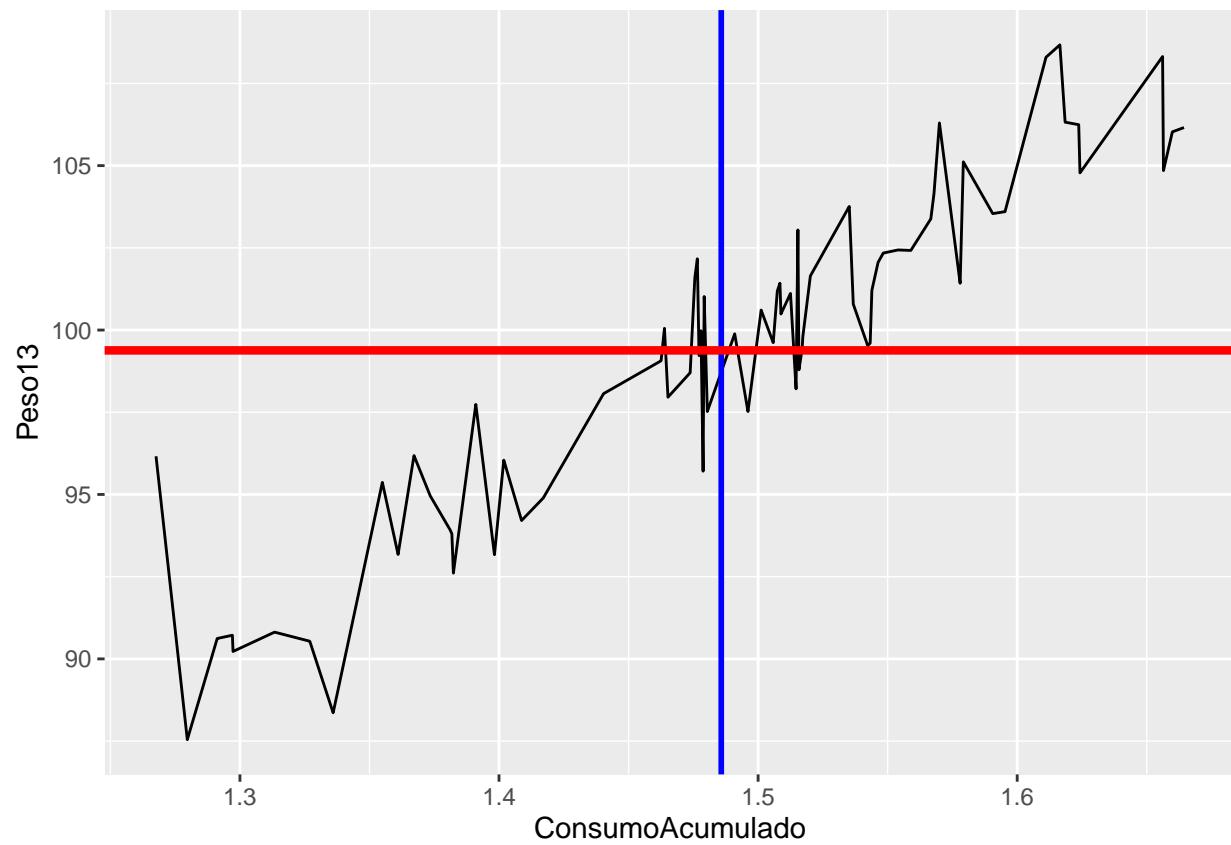


Figure 3.10: Linha Horizontal e linha vertical em um scatter plot conectado por linhas

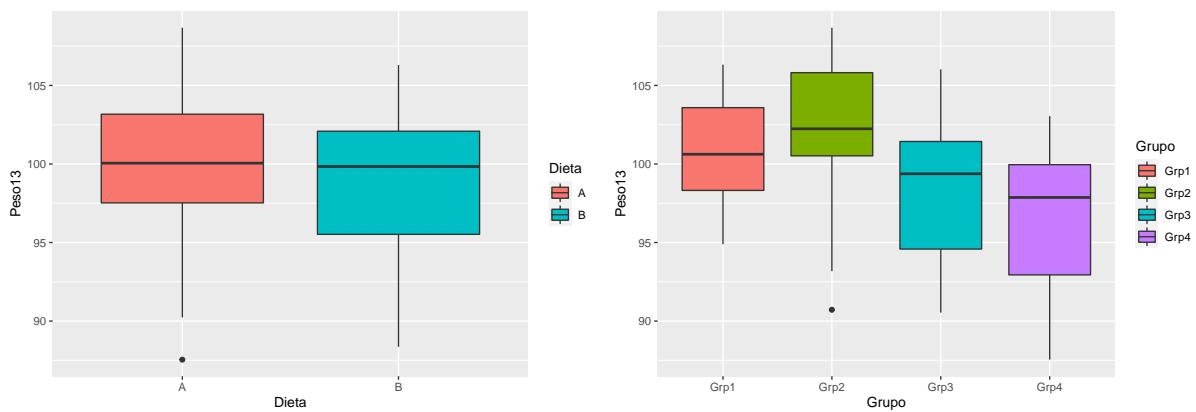


Figure 3.11: Box plot por Dieta e Box plot por Grupo

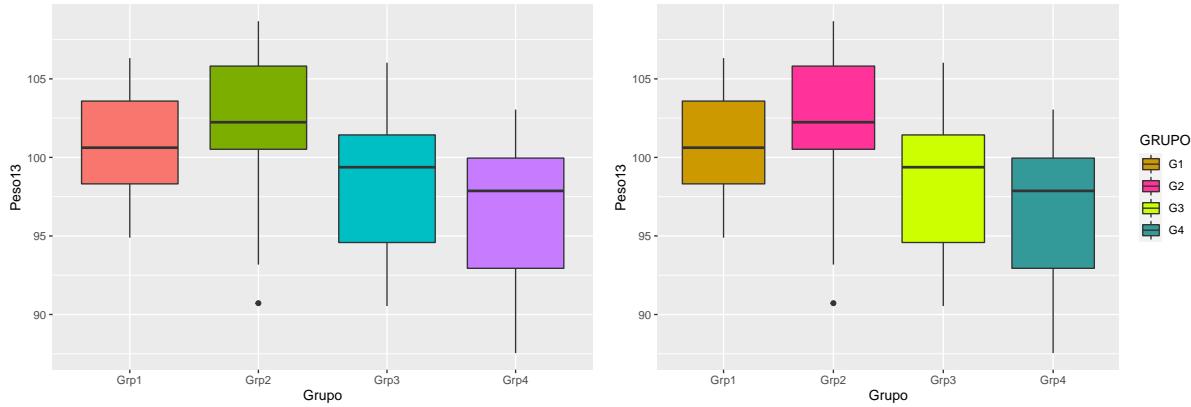


Figure 3.12: Alterando a legenda do Box plot

### 3.4.2.2 Escala de cinza

Em muitas situações é exigido figuras em escala cinza. Isso pode ser facilmente feito no `ggplot`.

```
# Box plot
bp2 + scale_fill_grey() + theme_classic()
# Scatter plot
g2 + scale_color_grey() + theme_classic()

## `geom_smooth()` using formula 'y ~ x'
```

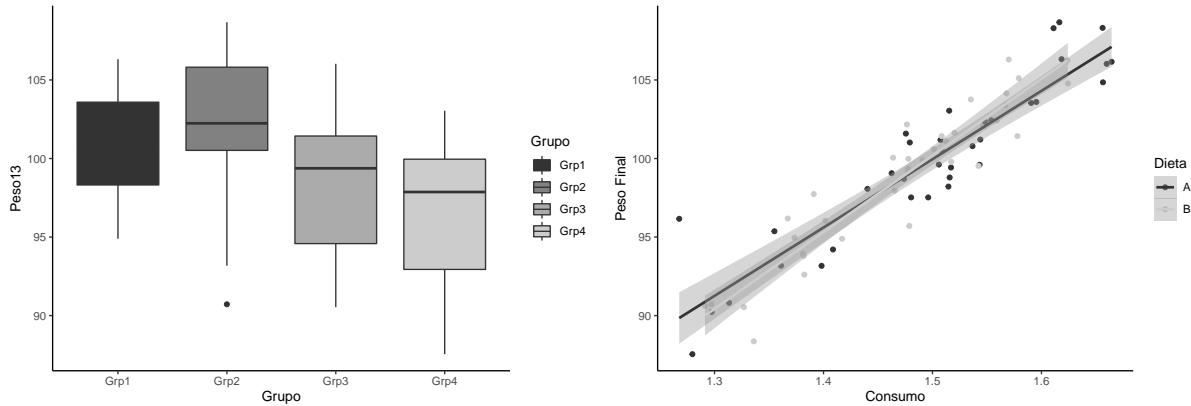


Figure 3.13: Box plot em escala cinza

### 3.4.3 Gráficos de barras

Gráficos de barras são interessantes para comparar as variáveis, identificar proporções, etc. Nesse exemplo, mostramos como fazer um gráfico de barras de uma variável, separando por níveis de um fator e colorí-lo.

```
ggplot(data=idades_dietas, aes(x = Grupo, y = Peso13)) +
  geom_col()
ggplot(data=idades_dietas, aes(x = Grupo, y = Peso13, fill=Grupo)) +
  geom_col()
```

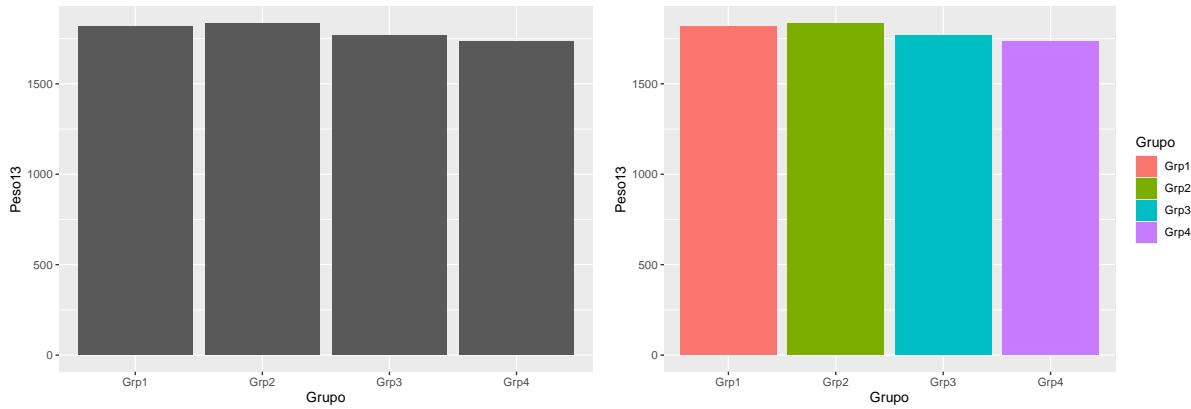


Figure 3.14: Gráfico de barras simples e gráfico de barras colorido por Dieta

### 3.4.4 Histogramas

Para construir histogramas mais elaborados no R, podemos usar o `ggplot` alterando o parâmetro `fill`. Ao colocarmos o parâmetro `fill=Dieta`, teremos o histograma colorido por Dieta.

Para esse exemplo vamos usar um banco de dados simulados especialmente para melhor visualização.

```
# load(dados_sim)
str(dados_sim)

## 'data.frame': 4400 obs. of  4 variables:
## $ Id   : int  1 2 3 4 5 6 7 8 9 10 ...
## $ Dias : int  1 1 1 1 1 1 1 1 1 1 ...
## $ Dieta: Factor w/ 4 levels "A","B","C","D": 3 3 3 2 3 2 2 2 3 1 ...
## $ Peso : num  -4.131 5.205 -3.672 7.457 -0.408 ...

filter(dados_sim,Dias>100)%>%ggplot( aes(x = Peso)) + geom_histogram(bins=20)

filter(dados_sim,Dias>100)%>%ggplot(aes(x=Peso,fill=Dieta))+geom_histogram(bins=25)
```

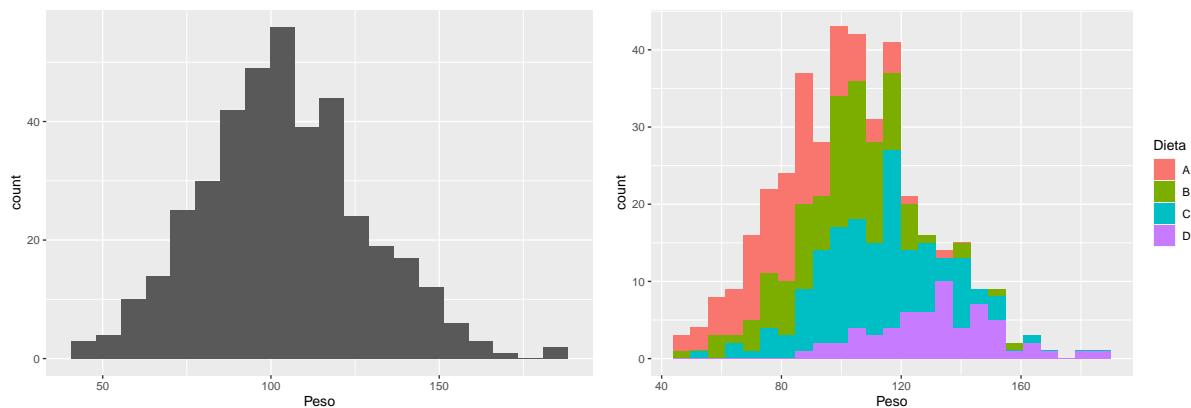


Figure 3.15: Histograma simples e separado por Dieta

### 3.4.5 Gráfico de séries temporais

```
dados_sim %>%
  ggplot( aes(x=Dias, y=Peso, group=Id, color=Dieta)) +
  geom_line() +
  ggtitle("Desempenho por indivíduo para diferentes dietas")
```

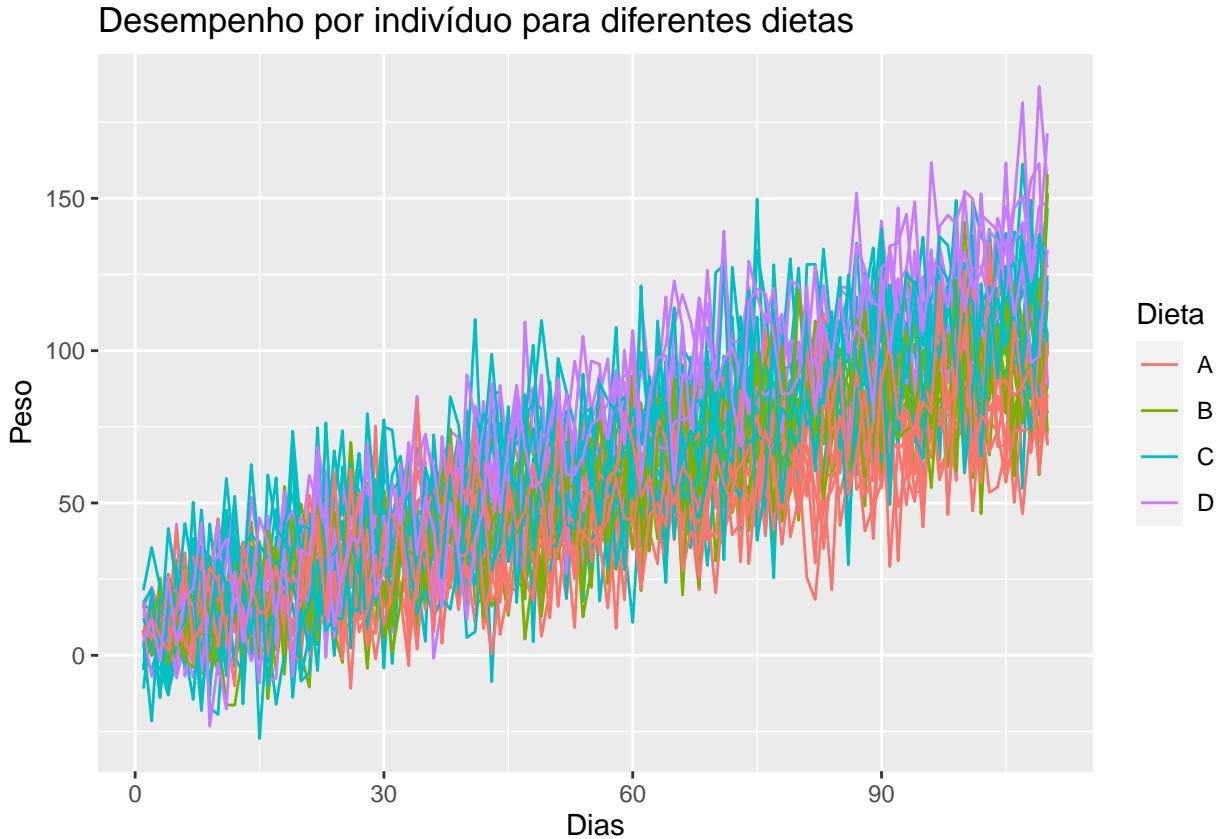


Figure 3.16: Desempenho dos indivíduos ao longo do tempo, colorido por Dieta.

Para fazer o gráfico evoluído no tempo dos indivíduos somente da Dieta A, usamos o comando `pipe` para filtrar os dados. Além disso, usamos o `scale_color_discrete` para refazer a legenda do gráfico;

```
dados_sim%>%filter(Dieta=="A")%>%ggplot( aes(x=Dias, y=Peso,group=Id,col="blue")) +
  geom_line() +
  ggtitle("Desempenho por indivíduo somente dieta A") +
  scale_color_discrete(name = "DIETA", labels = c("A"))
```

Usando o `facet_wrap()` podemos plotar o desempenho dos indivíduos ao longo do tempo, separando por Dietas.

```
ggplot(data=dados_sim) +
  geom_line( aes(x=Dias, y=Peso,group=Id,colour=Dieta)) +
  ggtitle("Sepando por dietas") +
  facet_wrap(~Dieta, ncol=2)
```

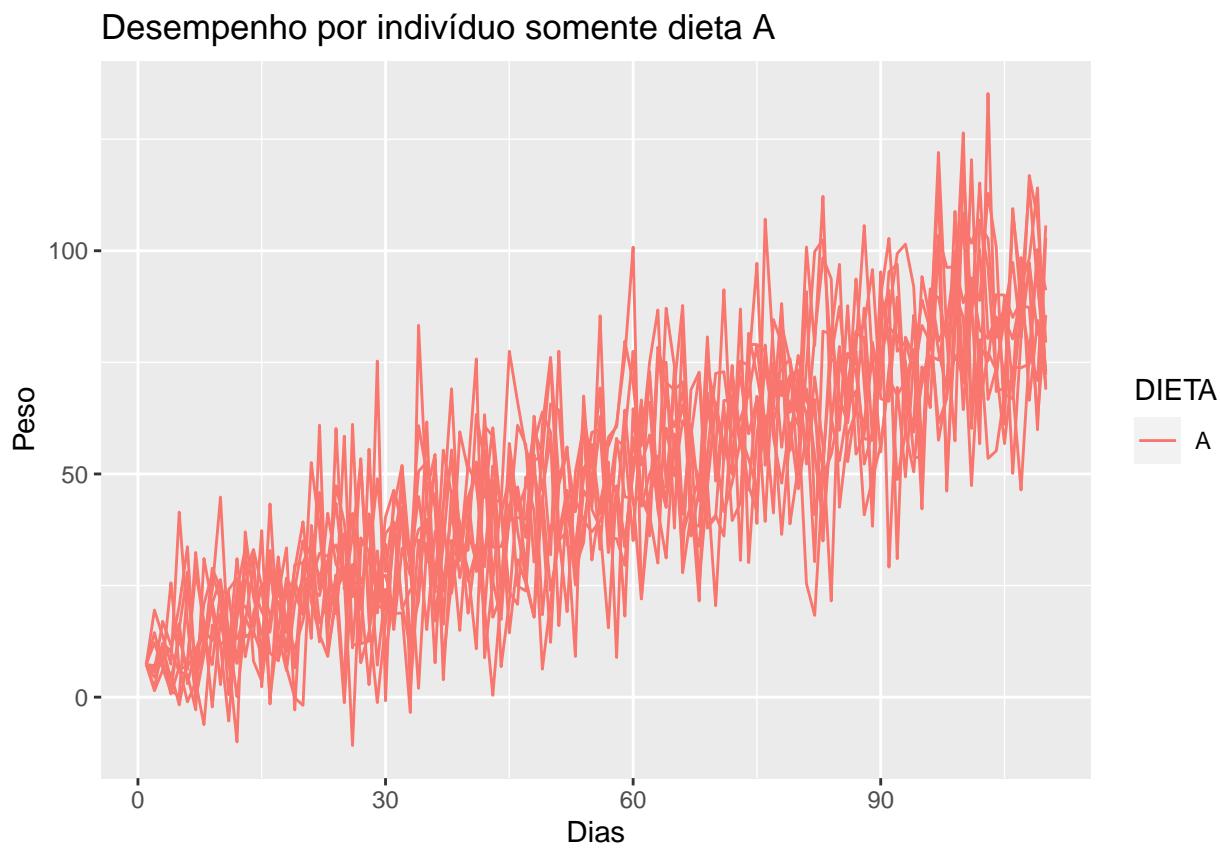


Figure 3.17: Desempenho dos indivíduos ao longo do tempo somente para a dieta A.

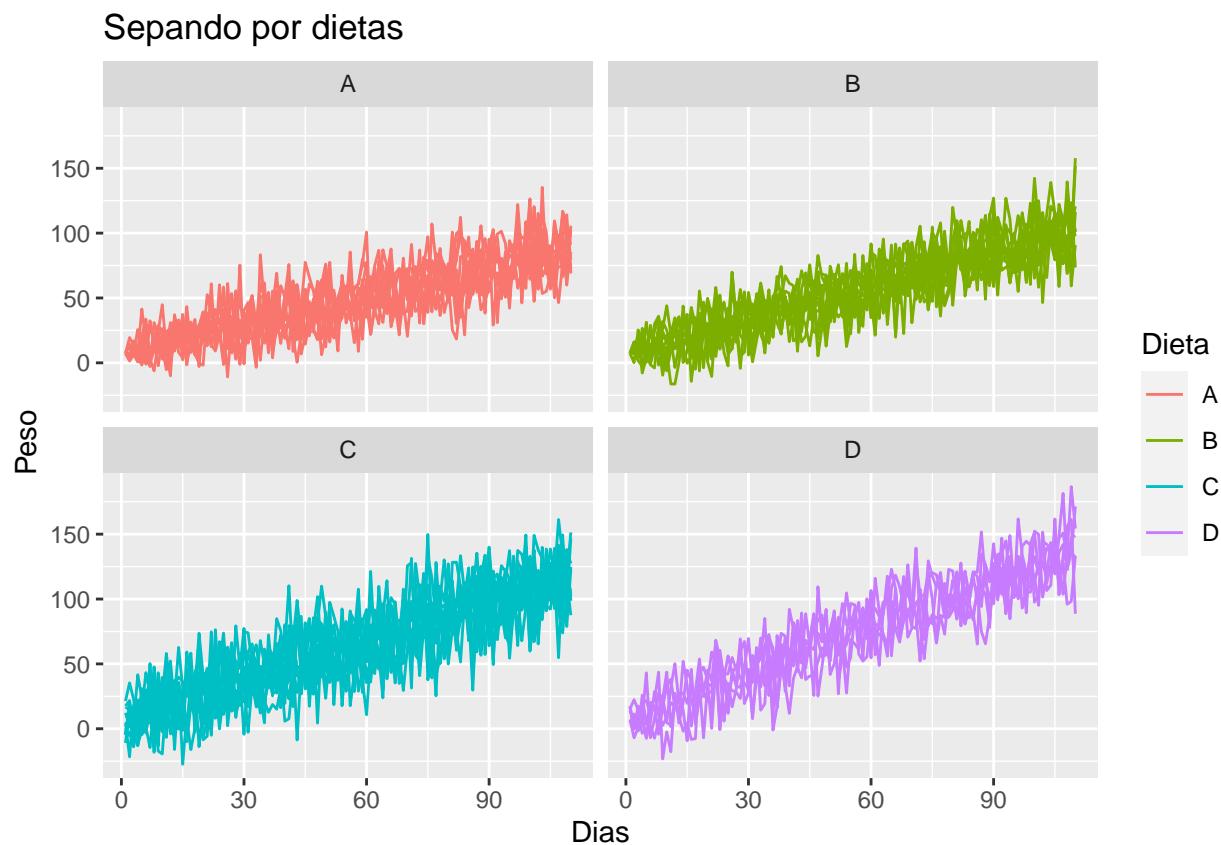


Figure 3.18: Desempenho dos indivíduos ao longo do tempo para todas as dietas em gráfico separados.

## **Chapter 4**

# **Análise de delineamento de experimentos**

**4.1 Delineamento inteiramente casualizados**

**4.2 Delineamento em blocos ao acaso**

**4.3 Fatorial**

**4.4 Parcera subdividida**



# Chapter 5

## Relatórios e artigos científicos com o Rmarkdown

### 5.1 Relatório dinâmico com o Rmarkdown

### 5.2 Relatórios em word, pdf ou html

### 5.3 Apresentações em Power Point

Afifi, Abdelmonem, Susanne May, Robin Donatello, and Virginia A Clark. 2019. *Practical Multivariate Analysis*. CRC Press.

Brambillasca, Sebastián. 2019. “Data for: Characterization of the in Vitro Digestion of Starch and Fermentation Kinetics of Sorghum Grains Soaked or Reconstituted and Ensiled to Be Used in Pig Nutrition.” Mendeley. <https://doi.org/10.17632/33SHF74ZN6.1>.

RStudio-Cloud. 2020. “RStudio Cloud.” <https://rstudio.cloud/>.

Sreng, Samorn, Sath Keo, JM DeRouchey, MD Tokach, Lyda Hok, JL Vipham, and others. 2020. “Effect of Complete Feed Feeding Level and Morning Glory on Growing Pig Performance.” *Open Journal of Animal Sciences* 10 (03): 493.

Wickham, Hadley. 2010. “A Layered Grammar of Graphics.” *Journal of Computational and Graphical Statistics* 19 (1): 3–28.

Wilkinson, Leland, and others. 2005. “The Grammar of Graphics.” Springer,