Nivelamento:

Introdução ao R

Vítor Wilher
Mestre em Economia
analisemacro.com.br

25 de outubro de 2018

Sumário

Lista de Tabelas

Lista de Figuras

Sobre o Nivelamento

Ementa

O nivelamento se divide em 5 seções. Elas se propõem a uma introdução qualificada ao R, desde a instalação e apresentação dos programas utilizados até a criação de funções, passando pelas principais estruturas de dados, importação de dados e construção de gráficos.

Programa

- Primeiros Passos
- Estruturas de Dados
- Importação de Dados
- Gráficos
- Ambientes Controlados

Sobre o Suporte do Curso

Os cursos da Análise Macro são segmentados por planos. Abaixo as características de cada um deles:

- **No plano básico**, o aluno tem acesso a todo o material do curso, podendo fazer download dos materiais impressos e assistir às videoaulas gravadas dentro do período da sua turma. **Nesse plano, não há suporte do professor**;
- **No plano intermediário**, além do acesso ao material do plano básico, o aluno pode receber feedbacks das listas de exercícios e dos demais conteúdos do curso;
- **No plano premium**, além do acesso ao material do plano básico e de poder receber feedbacks das listas de exercícios e dos demais conteúdos do curso, o aluno pode ainda marcar uma conversa ao vivo com o professor.

O suporte nos planos intermediário e premium funcionará da seguinte forma. Você deve acessar o arquivo `respostas.Rnw`, disponível no arquivo zipado do link *Exercícios*, na seção Primeiros Passos do Nivelamento em R. Ao final, você deve compilar o arquivo `respostas.Rnw`, gerando um pdf, conforme foi ensinado na seção Primeiros Passos do Nivelamento em R. Uma vez gerado o pdf, você deve enviá-lo para o professor via o botão azul no canto inferior direito da plataforma, conforme a figura abaixo. Demais dúvidas também devem ser enviadas por esse canal.

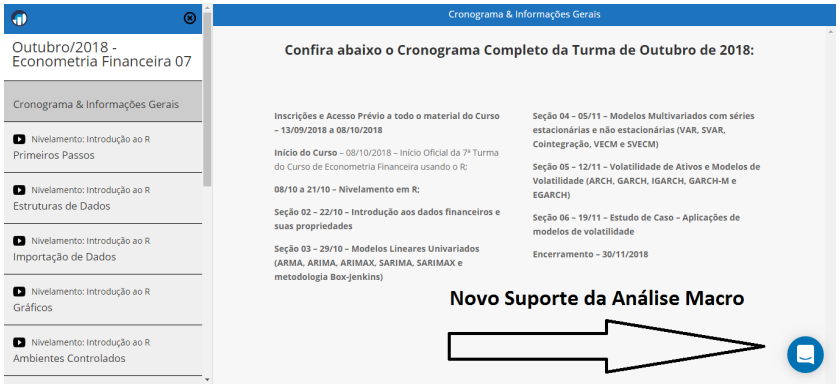


Figura 1: Suporte de Dúvidas da Análise Macro.

Sobre a distribuição desse material

Material Complementar

Essa **apostila** conta com material complementar, que será disponibilizado na página do nosso curso na Área do Aluno da Análise Macro. O conjunto desses materiais faz parte do curso de **Análise de Conjuntura usando o** $\mathbf{R}$.

Cada seção do nosso Curso de Análise de Conjuntura usando o $\mathbf{R}$ é composta por uma videoaula, uma apostila e uma lista de exercícios. A depender da seção, pode estar disponível também um slide. Todo o material impresso pode ser baixado pelo aluno, para que possa melhor absorver o conteúdo do curso.

Como você verá na seção *Primeiros Passos*, todo o material do curso é produzido em LaTeX, no $\mathbf{RStudio}$. Os arquivos .Rnw das listas de exercício serão disponibilizados, de modo que o aluno possa respondê-las para o professor, caso o mesmo seja dos planos premium e intermediário. Para melhor compreender como lidar com esses arquivos, veja a seção *Primeiros Passos*.

Pacotes utilizados nessa apostila

```
### Pacotes
library(fUnitRoots)
library(stargazer)
library(ggplot2)
library(ggthemes)
library(easyGgplot2)
library(forecast)
library(gridExtra)
library(astsa)
library(Cairo)
library(BMR)
library(xtable)
library(dynlm)
library(XLConnect)
library(base)
library(xts)
library(urca)
library(zoo)
library(mFilter)
library(TStools)
library(BETS)
library(seasonal)
library(reshape2)
library(scales)
```

Sobre o Autor

Vítor Wilher é Bacharel e Mestre em Economia, pela Universidade Federal Fluminense, tendo se especializado na construção de modelos macroeconométricos, política monetária e análise da conjuntura macroeconômica doméstica e internacional. Tem, ademais, especialização em Data Science pela Johns Hopkins University. Sua dissertação de mestrado foi na área de política monetária, titulada "Clareza da Comunicação do Banco Central e Expectativas de Inflação: evidências para o Brasil", defendida perante banca composta pelos professores Gustavo H. B. Franco (PUC-RJ), Gabriel Montes Caldas (UFF), Carlos Enrique Guanziroli (UFF) e Luciano Vereda Oliveira (UFF). Já trabalhou em grandes empresas, nas áreas de telecomunicações, energia elétrica, consultoria financeira e consultoria macroeconômica. É o criador da Análise Macro, startup especializada em treinamento e consultoria em linguagens de programação voltadas para data analysis, sócio da MacroLab Consultoria, empresa especializada em cenários e previsões e fundador do hoje extinto Grupo de Estudos sobre Conjuntura Econômica (GECE-UFF). É também Visiting Professor da Universidade Veiga de Almeida, onde dá aulas nos cursos de MBA da instituição, Conselheiro do Instituto Millenium e um dos grandes entusiastas do uso do R no ensino. Leia os posts de Vítor Wilher no Blog da Análise Macro. Caso queira, mande um e-mail para ele: *vitorwilher@analisemacro.com.br*.

1 Primeiros Passos

Eu comecei a utilizar o $\mathbf{R}$ há alguns anos, influenciado por amigos. Minha introdução ao mundo dos programas estatísticos foi através do Eviews, ainda nos tempos da graduação em economia, como provavelmente muitos dos matriculados nesse curso. Ainda que seja possível *programar* no Eviews e em outros pacotes estatísticos fechados (que precisam de licença), as vantagens do $\mathbf{R}$ são inúmeras, como comentarei mais à frente. Por ora, talvez seja necessário tecer algumas palavras sobre por que afinal é preciso aprender uma linguagem de programação.

Eu poderia falar que o mundo está mudando, que cada vez mais empregos e empresas têm exigido conhecimentos de programação. E isso de fato é verdade, o que por si só gera uma necessidade de saber programação. Mas, aqui entre nós, acho que é meio chato aprender algo por necessidade, não é mesmo?

Minha motivação para aprender a programar foi de fato outra. Eu estava um pouco cansado de *apertar botões* e fazer tarefas repetitivas em pacotes estatísticos como o Eviews, então parecia natural aprender uma forma de *automatizar* as coisas. Essa, afinal, era uma baita motivação para mim e talvez também seja para você. Mas por que o $\mathbf{R}$, você pode perguntar.

Essa é de fato uma boa pergunta. Por que não aprender a programar no próprio Eviews? E a resposta é bastante simples. Você já tentou encontrar alguma coisa de programação em Eviews na internet? E sobre $\mathbf{R}$? Pois é. Entre as inúmeras vantagens do $\mathbf{R}$, posso destacar:

- A existência de uma comunidade grande e bastante entusiasmada, que compartilha conhecimento todo o tempo;
- o $\mathbf{R}$ é gratuito, *open source*, de modo que você não precisa comprar licenças de software para instalá-lo;
- Tem inúmeras bibliotecas (pacotes) em estatística, *machine learning*, visualização, importação e tratamento de dados;
- Possui uma linguagem estabelecida para *data analysis*;
- Ferramentas poderosas para comunicação dos resultados da sua pesquisa, seja em

forma de um website ou em pdf.

Ao aprender $\mathbf{R}$, você conseguirá integrar as etapas de coleta, tratamento, análise e apresentação de dados em um único ambiente. Você vai esquecer ter de abrir o excel, algum pacote estatístico, depois o power point ou o word, depois um compilador de pdf para gerar seu relatório. Todas essas etapas serão feitas em um único ambiente. E essa talvez seja a grande motivação para você entrar de cabeça nesse mundo.

Certamente não será simples, tenho de lhe dizer. Haverá muitos momentos em que você pensará em desistir. Um erro inesperado em algum *script* poderá lhe tirar horas do seu tempo. No início, mais especificamente, qualquer pequeno problema pode parecer uma barreira intransponível. Nesses momentos, minha dica é *não se demorar muito nessas dificuldades*. Vá para outro problema ou mesmo descanse. Faça outras coisas. Depois volte mais tranquilo, procure nos canais de ajuda que colocaremos aqui e as coisas irão se acertar. Acredite: passado esse tempo inicial, com persistência, você certamente se beneficiará muito em ter aprendido uma linguagem de programação como o $\mathbf{R}$.

Nosso objetivo na Análise Macro, com nossos Cursos Aplicados de R é lhe proporcionar uma experência inovadora. Todos os nossos cursos apresentam sólida teoria, mas sempre recheada de exercícios e exemplos práticos. Nosso objetivo é trazer para o Brasil algo que já é bastante corriqueiro no resto do mundo: *você deve aprender fazendo*. Isso vai tornar o aprendizado mais interessante, mais divertido até.

Isso dito, vamos começar então? Nessa primeira seção, você verá alguns procedimentos básicos, sobre programas e um *overview* do $\mathbf{R}$. A partir da seção 02 começa o seu curso propriamente dito. Seja bem vindo a esse mundo e espero que não sai mais dele!

1.1 Instalando os programas

Antes de tudo, é preciso que você tenha os programas que utilizaremos ao longo das aulas instalados em seu computador. Serão três programas: $\mathbf{R}$, $\mathbf{RStudio}$ e $\mathbf{MikTex}$. Não se preocupe, posto que são todos programas gratuitos e com *download* seguro. Desse modo, para que não tenhamos problemas, siga a sequência abaixo:

1. Baixe o $\mathbf{R}$ em http://cran-r.c3sl.ufpr.br/;

2. Baixe o RStudio em https://www.rstudio.com/products/rstudio/download/;

3. Baixe o MikTex *se você for usuário de Windows* em http://miktex.org/download;

4. Baixe o MacTex *se você for usuário de Mac* em http://www.tug.org/mactex/.

Para que você não tenha problemas no futuro, instale a versão do **MikTex** correspondente à versão do seu Windows. Isto é, se o seu Windows for 64 bits, instale a versão 64 bits do **MikTex**, se for 32 bits, instale a versão 32 bits do **MikTex**.

Por fim, alguns pacotes que utilizaremos fazem uso do **JAVA**. Assim, é importante que você o tenha instalado em sua máquina, **na versão correspondente do seu sistema operacional**, assim como o **MikTex**. Para instalar o JAVA, vá em Oracle.

1.2 Trabalhando com o R a partir do RStudio

Ao longo desse curso, nós não trabalharemos diretamente no R. Ao invés disso, usaremos o RStudio, uma interface mais amigável, que nos permite emular todos os códigos do R, visualizar gráficos, ver o histórico de nossas operações, importar dados, criar scripts, etc. Com o RStudio, poderemos otimizar o nosso curso, de maneira que o aluno terá mais facilidade para interagir com a linguagem.

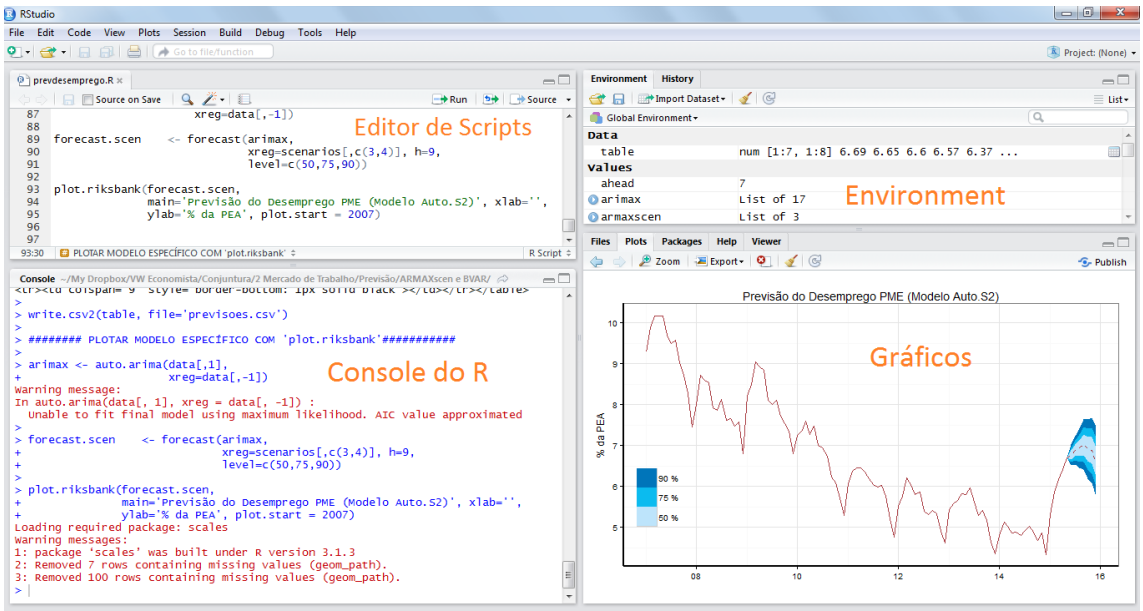


Figura 2: Ambiente do RStudio.

A figura acima resume as quatro principais partes de uma tela do $\mathbf{RStudio}$. Na parte superior esquerda é onde ficará o nosso *editor de scripts*. Um *script* é uma sequência de comandos com um determinado objetivo. Por exemplo, você pode estar interessado em *construir um modelo univariado para fins de previsão do índice BOVESPA*. Para isso, terá de primeiro importar os dados do Ibovespa, bem como fazer uma análise descritiva inicial dos dados. Depois, com base nessa análise inicial, você terá de decidir entre alguns modelos univariados distintos. De posse da sua decisão, você enfim construirá um modelo de previsão para o índice BOVESPA. Essa sequência de *linhas de comando* pode ficar armazenada em um *script*, com extensão .R, podendo ser acessada posteriormente por você mesmo ou compartilhada com outros colegas de trabalho. Para abrir um novo *script*, vá em *File*, *New File* e clique em *R Script*.

Na parte inferior esquerda, está o *console do R*, onde você poderá executar comandos rápidos, que não queira registrar no seu script, bem como será mostrados os *outputs* dos comandos que você executou no seu script.

Já na parte superior direita, está o *Environment*, onde ficam mostrados os objetos que você cria ao longo da sua seção no RStudio. Por fim, na parte inferior direita, ficarão os gráficos que você solicitar, bem como pacotes que você instalou, alguma ajuda sobre as funções e os arquivos disponíveis no seu diretório de trabalho.

1.3 Definindo seu diretório de trabalho

Agora que você já instalou os programas e já conhece um pouco do ambiente do $\mathbf{RStudio}$, podemos começar a brincar um pouco. Para isso, antes de mais nada é preciso definir o seu *diretório de trabalho* ou a pasta onde ficará salvo o seu *script*. Uma vez definido, você poderá importar arquivos, colocar figuras no seu documento LaTeX, etc. Logo, dois comandos são importantes para isso. O primeiro é o **getwd**, para você ver o seu atual *working directory*. O segundo é o **setwd**, para você *setar* o seu diretório de trabalho[1]

[1]Como você verá ao longo do nosso curso, as funções dentro do $\mathbf{R}$ são todas em inglês e bastante intuitivas, como *get something* ou *set something*.

```
getwd()
```

[1] "C:/Users/Vítor Wilher/Dropbox/VW Economista/Análise Macro/01 - Cursos/Nivelamento em R/Apostila"

```
setwd('C:/Users/Vítor Wilher/Dropbox/VW Economista')
```

Uma vez *setado* o seu diretório de trabalho, você poderá importar dados contidos naquela pasta. Assim, é um ponto importante realizar isso antes de qualquer coisa. Caso já tenha um *script* de R, por suposto, uma vez abrindo-o, o RStudio setará automaticamente o diretório onde o mesmo esteja.

1.4 Uma linguagem orientada a objetos

O $\mathbf{R}$ é uma linguagem orientada a objetos, de modo que o que você fará dentro do programa será, basicamente, manipulá-los. Seja lidando com objetos criados por terceiros, seja criando seus próprios objetos. As principais estruturas de dados dentro do $\mathbf{R}$ envolvem *vetores*, *matrizes*, *listas* e *data frames*. Abaixo colocamos um exemplo da estrutura mais simples do $\mathbf{R}$: um vetor que exprime a sequência de 1 a 10.

```
vetor <- c(1:10)

vetor
```

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

Ao longo do nosso curso, aprenderemos na prática a lidar com essas diferentes estruturas de dados. Em particular, aprenderemos a fazer o *subsetting* dessas estruturas, de modo a capturar subelementos das mesmas.

1.5 Milhares de pacotes a sua disposição

O $\mathbf{R}$ é uma linguagem aberta, onde qualquer pessoa em qualquer parte do mundo pode dar a sua contribuição. Em geral, elas fazem isso através de *pacotes*, que são coleções de funções que fazem algum coisa dentro do $\mathbf{R}$. Veremos muitos desses pacotes ao longo

do nosso curso. A instalação de pacotes é feita primariamente pelo CRAN, através da seguinte função:

```
install.packages('fBasics')
```

O pacote é instalado corretamente quando aparece a mensagem *package 'fBasics' successfully unpacked and MD5 sums checked* no seu console. Ademais, no processo de instalação de um pacote, pode ser necessário instalar outros pacotes, chamados de *dependentes*, porque uma ou mais funções do pacote que você quer instalar fazem uso de funções de outros pacotes. Assim, **para que a instalação seja efetuada com sucesso, é preciso que todos os pacotes dependentes sejam instalados corretamente.** Por isso, procure verificar as mensagens no console de forma a verificar se o pacote foi instalado corretamente, como mostrado na figura 2.

```
> install.packages('fBasics')
Warning in install.packages :
  cannot open URL 'http://www.stats.ox.ac.uk/pub/RWin/src/contrib/PACKAGES.rds'
: HTTP status was '404 Not Found'
Installing package into 'C:/Users/Vitor Wilher/Documents/R/win-library/3.4'
(as 'lib' is unspecified)
Warning in install.packages :
  cannot open URL 'http://www.stats.ox.ac.uk/pub/RWin/bin/windows/contrib/3.4/P
ACKAGES.rds': HTTP status was '404 Not Found'
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.4/fBasics_3011.87.zi
p'
Content type 'application/zip' length 1559813 bytes (1.5 MB)
downloaded 1.5 MB

package 'fBasics' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Vitor Wilher\AppData\Local\Temp\Rtmp6rMrmM\downloaded_packages
```

Figura 3: Instalando pacotes.

Uma vez instalado, os seus pacotes ficam armazenados na pasta *library* da versão correspondente do seu R.[2] Você pode ver a lista de pacotes naturamente, diretamente no RStudio, através da aba *Packages* no canto inferior direito.[3]

Uma outra forma muito comum de instalar pacotes é através do **GitHub**, uma plataforma bem bacana utilizada por desenvolvedores para compartilhar códigos. Ali ficam armazena-

[2]Toda vez que você instalar uma nova versão do R, uma dica é pegar os pacotes da sua pasta library e copiar os mesmos para a pasta library da nova versão, de modo a não ter que instalar tudo novamente.

[3]Uma lista dos pacotes disponíveis pode ser encontrada aqui.

dos pacotes *em desenvolvimento*, que ainda não estão disponíveis no CRAN. Para instalar um pacote via GitHub, você deve ter instalado primeiro o pacote *devtools*. O código abaixo exemplifica com a instalação do pacote brasileiro BETS.[4]

```
install.packages('devtools')
require(devtools)
install_github("pedrocostaferreira/BETS")
```

No código acima, nos instalamos o pacote *devtools*, depois **carregamos** o mesmo com a função `require` - também pode ser utilizado a função `library` - e então utilizamos a função `install_github` para instalar um pacote armazenado no GitHub.

Às vezes pode ser necessário instalar uma versão antiga de um pacote, seja porque a versão atual tem algum *bug* - acontece! - seja porque ela é incompatível com o pacote que queremos usar. Para instalar versões antigas, você tem duas opções. Uma é subir o arquivo fonte zipado do pacote diretamente no RStudio, na opção *Tools* e *Install Packages*. Outra é utilizar a função `install_version` do pacote *devtools*, como abaixo.

```
install_version("DBI", version = "0.5", repos = "http://cran.us.r-project.org")
```

É muito comum os alunos receberem o erro de *caminho inválido* para a biblioteca de pacotes. Nesse caso, você pode especificar o caminho da biblioteca com o código abaixo.

```
library(withr) # É instalado e carregado junto com o devtools
withr::with_libpaths(new = "C:/Program Files/R/R-3.4.1/library", install_github("Stat
```

Por fim, vale ressaltar que todo pacote disponível no CRAN tem uma página com suas informações. Veja o exemplo do *devtools* aqui. Lá você pode ver um resumo do pacote, um pdf com as principais funções, versões antigas, etc.

[4] Para saber mais sobre o BETS, vá em https://github.com/pedrocostaferreira/BETS.

1.6 Obtendo ajuda

Uma parte importante de aprender uma nova linguagem é saber conseguir resolver os pepinos que irão surgir ao longo do caminho. E, acredite: eles serão muitos! Mas não se desespere. Como há uma comunidade incrível trabalhando com **R**, existem inúmeros sites, blogs, tutoriais, apostilas, etc, que podem lhe ajudar. No próprio ambiente do **RStudio**, você pode invocar ajuda com os comandos abaixo.[5]

```
help.start() # Você terá acesso à página de ajuda do R.
help("read.csv") # Uma ajuda sobre a função 'read.csv'
?read.csv # A mesma coisa do comando acima.
```

Caso continue com dúvidas, entretanto, nada melhor do que o bom e velho Google. Recomendo que você jogue sempre sua dúvida lá, antes de qualquer coisa. Uma dica: jogue ela em inglês e verá logo na primeira página um monte de gente com o mesmo problema que você! Em particular, um fórum bastante conhecido é o Stackoverflow, onde pessoas mais experientes procuram ajudar os mais novos. Caso sua dúvida não seja resolvível via google, considere jogá-la nesse fórum. A versão em português do fórum está disponível em https://pt.stackoverflow.com/questions/tagged/r.

Outra coisa que você deve fazer a partir de agora é acompanhar blogs que falam de **R**. Modéstia ao largo, não se esqueça de acompanhar o meu próprio site www.analisemacro.com.br/blog. Lá faço alguns exercícios interessantes de como usar o **R** para resolver nossos problemas diários de análise de dados. No Brasil, há ainda os ótimos blogs do Cláudio Shikida e do Carlos Cinelli, que estão há bastante tempo nessa batalha também. No exterior, há uma infinidade de blogs reunidos no famoso R-bloggers.

[5] Observe que utilizamos o 'jogo da velha' # para colocar um comentário após a função. Isso é extremamente útil para que você lembre seus passos em um script no **R**, bem como no momento que você compartilhar seu script com outra pessoa.

1.7 Produzindo relatórios

1.7.1 Ei, você já instalou o MikTex/MacTex e o knitr?

Agora que você já conseguiu instalar e carregar pacotes no $\mathbf{R}$, vamos partir para um outro assunto. Uma das coisas mais legais do $\mathbf{R}$ é poder esquecer que o Word e o Power Point existem! Em um único ambiente você pode coletar, tratar, analisar e ... **apresentar** seus dados! Toda essa apostila, a propósito, é feita diretamente no $\mathbf{RStudio}$, dada a integração com o LaTeX. Para que isso seja possível, precisamos antes preparar o nosso ambiente. Siga os passos abaixo, de modo a fazer a integração entre o $\mathbf{R}$ e o LaTeX.[6]

1. Instale o pacote **knitr** com a `função` *install.packages*;
2. Vá em Tools, Global Options, Sweave e mude a opção *Weave Rnw files using*: to **knitr**;
3. Pronto, agora você está pronto para criar códigos LaTeX.

1.7.2 Tudo se resume a scripts

Uma vez que esteja tudo corretamente instalado, você deve abrir o $\mathbf{RStudio}$. Como vimos anteriormente, ao longo do nosso curso trabalharemos com *scripts*, sequências de comandos que têm por objetivo lidar com algum dado. Nesse caso, trabalharemos com extensões .R, enquanto no caso de produzir um documento, trabalharemos com extensões .Rnw. Criaremos muitos scripts para lidar com dados a partir da próxima seção - principalmente nas listas de exercícios -, de modo que por hora vamos nos concentrar nos scripts que geram documentos.

Para gerar um script que cria um documento, você deve ir em *File*, *New File* e clicar em *R Sweave*. Um *script* se abrirá com os seguintes comandos em LaTeX:

```
\documentclass{article}
```

[6] Caso queira conhecer desde já o ambiente do LaTeX, sugiro uma boa apostila do pessoal do PET de Telecomunicações da UFF aqui.

```
\begin{document}

\end{document}
```

Na primeira linha, nós definimos a classe do documento, pode ser um artigo, livro, apresentação, etc. Entre `\documentclass{article}` e `\begin{document}`, temos o *preâmbulo* do nosso documento, onde podemos listar pacotes que iremos utilizar, bem como outras definições de estilo do documento. Você irá escrever o seu documento entre `\begin{document}` e `\end{document}`, onde estará o *corpo* propriamente dito de tudo o que você fará. Uma referência para que você compreenda um documento LaTeX, desde a concepção do documento até diferentes comandos e ambientes que você pode utilizar, pode ser obtida aqui. Nessa nossa seção, tudo o que faremos é *facilitar* a sua vida, mostrando um exemplo simples de como gerar um relatório e uma apresentação, com códigos de R dentro.

1.7.3 Criando o seu primeiro documento LaTeX

Abra um arquivo .Rnw no RStudio seguindo as instruções da subseção anterior. Uma vez lá, coloque a sequência abaixo:

```
\documentclass[12pt, a4paper]{article}

\usepackage[T1]{fontenc}

\usepackage[portuguese]{babel}

\begin{document}

\begin{center}

\LARGE{Meu primeiro arquivo \LaTeX}

\end{center}

\subsection{Seção 1}

\subsubsection{Subseção 1.1}
```

Esse é o meu primeiro arquivo LaTeX.

```
\end{document}
```

Uma vez que tenha colocado essa sequência, clique em *Compile PDF* no cabeçalho do *script*. Caso não tenha salvo, o RStudio pedirá para salvar o arquivo e depois gerará o seu pdf. Caso você tenha conseguido gerar, é porque está tudo certo com os programas que recomendamos você instalar. Agora, vamos melhorar isso e colocar um código de R no seu documento LaTeX?

1.7.4 Adicionando códigos de R ao seu documento LaTeX

Para isso, precisamos utilizar o pacote **knitr**. É ele que vai emular códigos de R no seu documento. Você poderá controlar se quer mostrar os seus códigos ou se quer apenas o resultado dele, poderá controlar o espaço das figuras e inúmeras outras facilidades. Tudo isso é feito nos *chunks*, como o exemplo simples abaixo.

```
<<"chunk_example", eval=FALSE>>=
@
```

Ao longo das nossas listas de exercícios, vamos explorar algumas opções de *chunks*, mas você já pode conferir algumas aqui. Para adicionar um código de R no seu documento LaTeX, copie o *chunk* abaixo e coloque no arquivo que você acabou de criar acima.

```
<<"exemplo01", eval=T, echo=T, results='hide'>>=
@
```

Nessa área cinza, antes do @, adicione os seguintes comandos:

```
vetor <- 1:10
vetor
```

E agora, clique em *Compile PDF*. Você deverá ter gerado um PDF que mostra o código acima. Para mostrar o resultado do código, isto é, a sequência de 1 a 10, basta que você substitua *hide* por *asis* em `results`.

As listas de exercício do nosso curso são feitas nesse estilo. Procure utilizar o arquivo .Rnw da lista para respondê-la e enviar para mim - caso você seja aluno dos planos intermediário e premium. Isso vai facilitar muito a nossa interação.

1.7.5 Nosso curso já começou...

Pode não parecer, mas nosso curso já começou. Nossa ideia é mostrar para você como lidar com dados de forma real. Nas próximas semanas, tudo o que você fará será basicamente praticar, sempre. Mesmo que haja alguma teoria envolvida, em todas as seções, você será obrigado a fazer alguma coisa prática. Nessa seção não é diferente, você já tem um monte de coisa para fazer!

Senão, vejamos. Você já pode dar uma boa olhada naquela apostila de LaTeXque referenciamos. Você também já pode dar uma boa olhada nas opções de *chunk* no site do criador do pacote **knitr**.

A próxima seção, por suposto, é a primeira seção do seu curso propriamente dito. Bem vindo e mãos à obra!

2 Estruturas de dados

O R possui cinco classes básicas (ou *atômicas*) de objetos, a saber: *character*, *numeric*, *integer*, *complex* e *logical*. Os objetos no R, por sua vez, são divididos em algumas `estruturas de dados`, a saber: *vetores atômicos*, *matrizes*, *arrays*, *listas* e *data frames*. Cada uma dessas estruturas possui características próprias, servindo a determinados tipos de dados. Elas podem, entretanto, ser organizadas, como propõe Wickham (2014), pela *dimensionalidade* e pelo fato de poderem receber dados homogêneos (todos os elementos precisam ser do mesmo tipo) ou heterogêneos, da seguinte forma:

Organização de estruturas de dados no R

	Homogêneos	Heterogêneos
1d	Atomic Vector	List
2d	Matrix	Data frame
nd	Array	

2.1 Vetores

O mais básico tipo de objeto no R é o `vetor`. Vetores podem ser de dois tipos: `vetores atômicos` ou `listas`. Eles possuem três propriedades comuns:

- Tipo, `typeof()`, o que é;
- Tamanho, `length()`, quantos elementos possui;
- Atributos, `attributes()`, metadados arbitrários adicionais.

A diferença entre eles, como visto acima, é que todos os elementos em um vetor atômico precisam ser da mesma classe, enquanto na lista, não necessariamente. Abaixo um exemplo de vetor numérico de tamanho 10 no R.

```
x = seq(1:10) # criando uma sequência de 1 a 10
vetor = rnorm(x, 34, 12) # geração aleatório de números
class(vetor) # verificando a classe do objeto
```

```
## [1] "numeric"

length(vetor) # verificando o tamanho do objeto

## [1] 10

typeof(vetor) # o tipo do objeto

## [1] "double"

str(vetor) # A estrutura do objeto

##  num [1:10] 57.9 34.9 34.6 23.6 33.3 ...
```

Sobre um vetor, é possível fazer operações de *redução* ou de *vetorização*, como abaixo.

```
sum(vetor)/length(vetor) # Obtendo a média dos dados

## [1] 35.27531

vetor - mean(vetor) # O quanto cada observação se distancia da média

##  [1]  22.6306821  -0.3500676  -0.6905102 -11.6791533  -2.0131874
##  [6]  10.8941518   1.6481332 -15.0456823  -1.4398437  -3.9545226
```

No primeiro caso, estamos fazendo uma operação que irá resultar em único número. Já no segundo, estamos fazendo uma operação com cada elemento do vetor, o que naturalmente irá resultar em um novo vetor.

Vamos avançar um pouco no entendimento e manipulação dessa estrutura básica no **R** por meio de um exemplo mais real. Para isso, precisamos instalar o pacote `UsingR`, que acompanha o livro Verzani (2014), nossa referência bibliográfica básica nesse curso. O código abaixo ilustra a instalação e posterior carregamento do pacote.[7]

```
install.packages('UsingR')
require(UsingR)
```

Feito isso, podemos acessar todos os `data sets` do pacote com o comando abaixo.

```
data(package='UsingR')
```

Escolhemos *brincar* com o *data set* `coldvermont`, que traz a temperatura mínima diária, em *Fahrenheit*, em Woodstock Vermont entre os anos de 1980 e 1985.

[7] Caso tenha dificuldades nesse processo, dê uma olhada na subseção **??**.

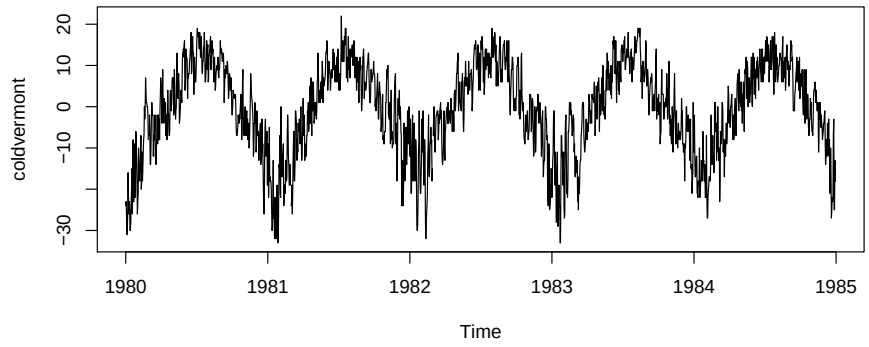
```
library(UsingR)
class(coldvermont)

## [1] "ts"
```

Observe que a classe do objeto é `ts`, de série temporal. Podemos, para facilitar a interpretação, passar de graus Fahrenheit para graus Celsius, através da fórmula abaixo.

```
coldvermont = round((coldvermont-32)/1.8,0)
```

E abaixo, plotamos um gráfico rápido do nosso conjunto de dados.

```
plot(coldvermont)
```



Observe que as temperaturas mínimas apresentam uma *sazonalidade* ao longo do ano. Isto é, elas aumentam durante a primeira metade e diminuem na segunda metade do ano.

2.1.1 Sequências e repetições

Dois comandos que facilitam o dia-a-dia são sequências e repetições.

```
seq(1:10) # cria uma sequência de 1 a 10.
```

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

```
seq(2,20, 2) # cria uma sequência de 2 a 20, com razão 2.
```

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

```r
rep(10, 10) # repete o número 10, dez vezes.
```

[1] 10 10 10 10 10 10 10 10 10 10

```r
rep(3:6, 3) # repete a sequência de 3 a 6, três vezes.
```

[1] 3 4 5 6 3 4 5 6 3 4 5 6

2.2 Matrizes

Um vetor é um caso especial de matriz posto que só possui uma única coluna. De forma geral, entretanto, podemos ter n colunas. No R, o comando básico para representar essa estrutura é dado abaixo.

```r
x <- rep(1:4)
matriz <- matrix(x, nrow=2, ncol=2)
```

	1	2
1	1	3
2	2	4

Tabela 1: Exemplo de Matriz

Abaixo um exemplo de matriz com três variáveis.

```
##       selic inflacao desemprego
## [91,] 10.90     0.01        4.7
## [92,] 10.90     0.25        5.0
## [93,] 10.90     0.57        4.8
## [94,] 10.92     0.42        4.6
## [95,] 11.15     0.51        5.2
## [96,] 11.58     0.78        5.0
```

Para acessar partes de uma matriz, você também utiliza []. Nesse caso, entretanto, temos linhas e colunas, logo você deve especificar a linha e a coluna que quer de acordo com os exemplos abaixo.

```r
data[1,] # Primeira linha da matriz
data[1:6,] # Mostra as seis linhas de uma matriz
data[,3] # Mostra a terceira coluna
data[,c(2,3)] # Mostra a segunda e a terceira coluna
```

Por fim, você pode atribuir nomes às colunas e linhas de uma matriz, conforme os comandos abaixo.

```r
colnames(data) <- c('IPCA', 'SELIC', 'DESEMPREGO')
rownames(data) <- c(1:nrow(data))
```

2.3 Listas

Listas no $\mathbf{R}$ são um caso especial de vetor, dado que é possível ter nessa estrutura diferentes tipos de dados. Em outras palavras, é possível ter nomes e números, por exemplo.

```r
lista <- read.csv2('arquivo01.csv', header=F)


lista <- list(Nome=lista[,1], Curso=lista[,2])
```

No exemplo acima, criamos uma lista com os nomes e os cursos dos alunos da turma. Desse modo, o objeto *lista* possui duas partes: uma parte de cursos e outra de nomes. Para ver a estrutura do objeto, usamos o comando abaixo.

```r
str(lista)
```

Você pode acessar o componente Nome das listas com $, como no exemplo abaixo.

```r
lista$Nome
```

Como você verá no decorrer do curso, objetos com vários componentes sempre podem ser desmembrados, para que você possa manipulá-los de acordo com as suas necessidades. Ademais, você também pode acessar os componentes de uma lista com os comandos abaixo.

```r
lista[[2]] # Acessa os cursos dos alunos da lista.
```

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

```
##  [4] Economia           Ciências Atuariais Ciências Atuariais
##  [7] Economia           Ciências Atuariais Ciências Atuariais
## [10] Economia           Economia           Economia
## [13] Ciências Atuariais
## Levels: Ciências Atuariais Economia
```

Além disso, é possível que você queira ver apenas um item do componente Nomes. Isso é possível com o comando abaixo.

```
lista[[1]][1] # Vai lhe dar o primeiro nome do componente Nomes.

## [1]  Leandro Moares Franco de Barros
## 13 Levels:  Leandro Moares Franco de Barros ...

lista$Nome[1] # Outra forma de ver a mesma coisa.

## [1]  Leandro Moares Franco de Barros
## 13 Levels:  Leandro Moares Franco de Barros ...
```

2.4 Data frames

`Data frames` se situam na fronteira entre matrizes e listas de modo que nós temos duas interfaces para trabalhar dentro da mesma estrutura. Ademais, é possível utilizar determinadas funções para tornar o gerenciamento dessa estrutura de dados ainda mais conveniente. Vamos organizar nossa discussão em alguns aspectos, como se segue.

2.4.1 Construindo um data frame

Data frames são listas de variáveis. Eles podem ser construídos como se segue.

```
df = data.frame(nm1=vec1, nm2=vec2)
```

A combinação `key=value` equivale às colunas, enquanto as linhas são criadas automaticamente. Observe, por suposto, que os vetores precisam ter o mesmo tamanho para estarem no mesmo data frame. Com a função `as.data.frame` é possível ainda forçar uma estrutura a ser um data frame.

2.4.2 Pesquisa de dados

É possível acesssar as variáveis de um data frame através da notação $, da seguinte forma:

```
data(mtcars)
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

É possível ainda pesquisar dados como em matrizes, da seguinte forma:

```
mtcars[,1]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

Ou fazer operações dentro do mesmo data frame com a função `with`:

```
with(mtcars, mean(mpg) - sd(mpg))
```

```
## [1] 14.06368
```

2.4.3 Modificando linhas, colunas e nomes

A notação de matriz é útil para modificar blocos de linhas em um data frame.

```
library(UsingR)
d = Cars93[1:3,1:4]
d[1,1] = d[3,4] = NA
d
```

```
##   Manufacturer   Model    Type Min.Price
## 1         <NA> Integra   Small      12.9
## 2        Acura  Legend Midsize      29.2
## 3         Audi      90 Compact        NA
```

Observe que o objeto d acima é composto por quatro colunas, sendo três delas recebendo *Factors*. Observe abaixo:

```
str(d)

## 'data.frame': 3 obs. of  4 variables:
##  $ Manufacturer: Factor w/ 32 levels "Acura","Audi",..: NA 1 2
##  $ Model       : Factor w/ 93 levels "100","190E","240",..: 49 56 9
##  $ Type        : Factor w/ 6 levels "Compact","Large",..: 4 3 1
##  $ Min.Price   : num  12.9 29.2 NA
```

Assim, se quisermos, por exemplo, adicionar uma linha precisamos primeiro criar os *levels* para o *Factor* envolvido:

```
levels(d$Model) = c(levels(d$Model), c('A3', 'A4', 'A6'))
d[4,] = list('Audi', 'A4', 'Midsize', 35)
d

##   Manufacturer   Model    Type Min.Price
## 1         <NA> Integra   Small      12.9
## 2        Acura  Legend Midsize      29.2
## 3         Audi      90 Compact        NA
## 4         Audi      A4 Midsize      35.0
```

Uma outra forma de adicionar uma linha, por suposto, é com a função `rbind`:

```
d = rbind(d, list('Audi', 'A6', 'Large', 45))
d

##   Manufacturer   Model    Type Min.Price
## 1         <NA> Integra   Small      12.9
## 2        Acura  Legend Midsize      29.2
## 3         Audi      90 Compact        NA
## 4         Audi      A4 Midsize      35.0
## 5         Audi      A6   Large      45.0
```

Adicionar uma coluna pode ser feito da seguinte forma:

```
d[,5] = d$Min.Price*1.3 # Preço em Euro
d

##   Manufacturer   Model    Type Min.Price    V5
## 1         <NA> Integra   Small      12.9 16.77
## 2        Acura  Legend Midsize      29.2 37.96
## 3         Audi      90 Compact        NA    NA
## 4         Audi      A4 Midsize      35.0 45.50
## 5         Audi      A6   Large      45.0 58.50
```

Ou, de forma a nomear a coluna:

```
d = d[,-5]
d$Min.Price.Euro = d$Min.Price*1.3
d

##   Manufacturer   Model    Type Min.Price Min.Price.Euro
## 1         <NA> Integra   Small      12.9          16.77
## 2        Acura  Legend Midsize      29.2          37.96
## 3         Audi      90 Compact        NA             NA
## 4         Audi      A4 Midsize      35.0          45.50
## 5         Audi      A6   Large      45.0          58.50
```

Podemos colocar todos os nomes em minúsculo com o código abaixo:

```
names(d) = tolower(names(d))
d

##   manufacturer   model    type min.price min.price.euro
## 1         <NA> Integra   Small      12.9          16.77
## 2        Acura  Legend Midsize      29.2          37.96
## 3         Audi      90 Compact        NA             NA
## 4         Audi      A4 Midsize      35.0          45.50
## 5         Audi      A6   Large      45.0          58.50
```

E podemos modificar o nome de uma coluna como abaixo:

```
names(d)[3] = 'car type'
d

##   manufacturer   model car type min.price min.price.euro
## 1         <NA> Integra    Small      12.9          16.77
## 2        Acura  Legend  Midsize      29.2          37.96
## 3         Audi      90  Compact        NA             NA
## 4         Audi      A4  Midsize      35.0          45.50
## 5         Audi      A6    Large      45.0          58.50
```

2.4.4 A função `subset`

Como vimos acima, podemos extrair blocos de um data frame especificando as linhas e colunas do mesmo. Isso, entretanto, pode ser complicado se o seu data frame possuir muitas linhas e colunas. Um modo mais interessante de fazê-lo pode ser utilizando a função `subset` como abaixo:

```
aq = airquality[1:5,]
aq
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
```

```
subset(aq, select=Ozone:Wind)
```

```
##   Ozone Solar.R Wind
## 1    41     190  7.4
## 2    36     118  8.0
## 3    12     149 12.6
## 4    18     313 11.5
## 5    NA      NA 14.3
```

```
subset(aq, select=-c(Month,Day))
```

```
##   Ozone Solar.R Wind Temp
## 1    41     190  7.4   67
## 2    36     118  8.0   72
## 3    12     149 12.6   74
## 4    18     313 11.5   62
## 5    NA      NA 14.3   56
```

```
subset(aq, subset=!is.na(Ozone), select=Ozone:Wind)
```

```
##   Ozone Solar.R Wind
## 1    41     190  7.4
## 2    36     118  8.0
## 3    12     149 12.6
## 4    18     313 11.5
```

2.4.5 is.na, complete.cases

Quando estamos extraindo dados de um data frame, pode ser interessante ignorar linhas com NA. Observe abaixo:

```
DF = data.frame(a=c(NA,1,2), b=c('one', NA, 'three'))
DF
```

```
##    a     b
## 1 NA   one
## 2  1  <NA>
## 3  2 three
```

```
subset(DF, !is.na(a))
```

```
##   a     b
## 2 1  <NA>
## 3 2 three
```

```
subset(DF, complete.cases(DF))
```

```
##   a     b
## 3 2 three
```

3 Importação de dados

Nas seções anteriores, iniciamos nossa aventura pelo maravilhoso mundo do $\mathbf{R}$. Aprendemos como instalar os programas, fazer a integração do $\mathbf{R}$ com o LaTeX, se situar dentro do ambiente do $\mathbf{RStudio}$, instalar e carregar pacotes, reconhecer as principais estruturas de dados, etc. Chegamos agora a uma seção crucial para nossas pretensões: a importação de dados para o $\mathbf{R}$. Essa etapa é estratégica porque é precisamente nela que moram quase todos os dramas de quem está iniciando na linguagem.[8]

Vamos, nesse contexto, bem devagar, para que você consiga absorver as principais etapas de importação de dados, do caso mais simples aos mais complexos, ok?

3.1 Importando arquivos .csv

O primeiro caso é de arquivos .csv. Esses arquivos são comuns em databases, como o SIDRA-IBGE e o IPEADATA, então é preciso estar atento ao processo. Vamos importar o arquivo02.csv, disponível na página do curso. Antes de importar, baixe o arquivo e dê uma olhada em como ele é. Veja que ele está separado por vírgulas, como é típico em arquivos desse tipo. Para fazer a importação, vamos usar a função *read.csv2*, conforme o exemplo abaixo.

```
datacsv <- read.csv2('arquivo02.csv', header = T, sep=',')
```

Você pode usar *?read.csv2* para ver todos os argumentos dessa função. Observe que no exemplo acima colocamos T em *header* e ',' em *sep*. Estamos dizendo para o $\mathbf{R}$ que o arquivo tem cabeçalho e que está separado por vírgulas. Feito isto, podemos dar uma olhada nas primeiras linhas do objeto criado com a função *head*, como já vimos na seção anterior.

```
head(datacsv, 4)

##   X selic inflacao desemprego
```

[8] Sobre isso, recomendo esse podcast aqui.

```
## 1  1 13.13     0.44        9.7
## 2  2 12.93     0.44        9.6
## 3  3 12.74     0.37        9.6
## 4  4 12.58     0.25        9.6
```

Observe que ao importar foram criadas quatro colunas, ao invés de três. Você pode ver no *Environment* do **RStudio** que o nosso objeto *data* possui 4 variáveis. Para corrigir isso, use o comando abaixo.

```
datacsv <- datacsv[,-1] # Retira a primeira coluna do objeto.
tail(datacsv)

##    selic inflacao desemprego
## 91  10.9     0.01        4.7
## 92  10.9     0.25          5
## 93  10.9     0.57        4.8
## 94 10.92     0.42        4.6
## 95 11.15     0.51        5.2
## 96 11.58     0.78          5

class(datacsv)

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

Observe ainda que o nosso objeto está classificado como um *data frame*. Você pode mudar isso depois, não se preocupe. Com efeito, os seus dados estão prontos para serem utilizados dentro do **R**. Para que isso fique claro para você, sugiro que você pare agora e comece a praticar. Dê uma olhada nos argumentos da função *read.csv2*, pegue arquivos .csv em bases de dados como o IPEADATA ou no Banco Central. Reitero: não tenha pressa aqui, posto que esse ponto é crucial para nossas pretensões! No script dessa seção há outros exemplos para você praticar.

3.2 Importando arquivos .xls e .xlsx

Existem diversas formas de importar arquivos .xls e .xlsx para o R.[9] Dessas, a que achei mais intuitiva foi utilizar o pacote *XLConnect*. Caso não tenha, instale com o comando *install.packages*. Abaixo segue um exemplo simples de como importar um arquivo .xls com esse pacote.

```
library(XLConnect) # Carregar o pacote XLConnect.
dataxls <- loadWorkbook('arquivo03.xls')
dataxls <- readWorksheet(dataxls, sheet = "IBC", header = TRUE,
                         colTypes = c('POSIXct','numeric',
                                      'numeric'))
```

Primeiro, você carregar a pasta do Excel com a função *loadWorkbook*. Feito isto, escolhe uma das planilhas que quer com o argumento *sheet*. Observe ainda que colocamos no argumento *colTypes* o tipo de dado que está em cada coluna.[10]

3.3 Importando arquivos da internet

Uma outra forma de importar arquivos para o R é diretamente pela internet. Você pode fazer isso com a função *download.file*.[11] Abaixo colocamos um exemplo de um arquivo .xls que está no site do Banco Central. Também utilizamos aqui o pacote *XLConnect* e a única diferença para o processo anterior são as duas primeiras linhas.

```
temp <- tempfile()
download.file("http://www.bcb.gov.br/pec/Indeco/Port/IE1-01.xlsx",
              destfile=temp,mode='wb')
datanet <- loadWorkbook(temp)
datanet <- readWorksheet(datanet, sheet = 1, header = TRUE)
rm(temp) # Deletar o arquivo temporário
```

[9] Você pode ver um resumo dessas formas aqui.

[10] Os tipos possíveis são lógicos, datas, numérico e caractere.

[11] Use *?download.file* para ver os argumentos dessa função.

No código acima, nós criamos um arquivo temporário com a função *tempfile* e fazemos o download do arquivo .xls para esse arquivo.[12] O processo posterior é igual ao de cima. Uma outra forma bem interessante é pegar arquivos zipados. Isso é feito com o código abaixo.

```
temp <- tempfile()
download.file("http://www.bcb.gov.br/ftp/notaecon/Divggnp.zip",temp)
datazip <- unzip(temp, files='Divggnp.xls')
datazip <- loadWorkbook(datazip)
datazip <- readWorksheet(datazip, sheet = "% PIB", header = TRUE,
                         colTypes = 'numeric')
rm(temp)
```

3.4 Importando dados irregulares

Dados irregulares são geralmente um baita problema para programas estatísticos. No **R**, você terá que fazer alguns ajustes para tornar esses dados analisáveis. Com o código abaixo, nós importamos o 'arquivo04.csv', disponível na página do nosso curso. Observe que colocamos ';' no argumento *sep* e ',' no argumento *dec*.

```
datairregular <- read.table(file='arquivo04.csv', sep=';',
                            dec = ',', header=T)
```

Feito isto, vamos dar uma olhada nos dados.

```
tail(datairregular)

##            Date Selic Expinf NeutroMin NeutroMax
## 3469 21/08/2015 14.15   5.66       4.5       5.5
## 3470 24/08/2015 14.15   5.66       4.5       5.5
## 3471 25/08/2015 14.15   5.66       4.5       5.5
## 3472 26/08/2015 14.15   5.66       4.5       5.5
```

[12] Observe o argumento *destfile*.

```
## 3473 27/08/2015 14.15   5.66       4.5       5.5
## 3474 28/08/2015 14.15   5.66       4.5       5.5
```

O arquivo contém a taxa básica de juros (a Selic), as Expectativas de Inflação 12 meses a frente e dois valores de mínimo e máximo para o juro neutro. A ideia é obter o que os economistas chamam de juro real ex-ante, isto é, os juros nominais deflacionados pela expectativa de inflação. Se esse juro se encontrar abaixo do valor mínimo, dizemos que a política monetária é expansionista. Se estiver acima do valor máximo, dizemos que ela é contracionista. Veremos isso melhor mais a frente, mas o importante agora é deixar o nosso dado analisável. Para isso, vamos utilizar dois pacotes: *xts* e *zoo*. O primeiro utilizaremos para ordenar as séries e o segundo para torná-las séries de tempo.[13] O código abaixo faz precisamente isso.

```
### Primeira coluna como datas
datairregular$Date = as.Date(datairregular$Date,
                             format="%d/%m/%Y")

### Ordenar séries de acordo com datas
library(xts)
selic <- xts(x = datairregular$Selic,
                       order.by = datairregular$Date)
expinf <- xts(x = datairregular$Expinf,
              order.by = datairregular$Date)
neutromin <- xts(x = datairregular$NeutroMin,
                 order.by = datairregular$Date)
neutromax <- xts(x = datairregular$NeutroMax,
                 order.by = datairregular$Date)

### Gerar série de juro real
juroreal <- selic-expinf
```

[13] Para uma rápida leitura sobre séries temporais no **R**, ver Lundholm (2011).

```
### Transformar em série temporal
library(zoo)
juroreal <- as.zoo(juroreal)
neutromin <- as.zoo(neutromin)
neutromax <- as.zoo(neutromax)
```

Com esse código, os dados estão prontos para serem analisados. Falaremos mais a frente sobre tudo isso, não se preocupe.

3.5 Importando dados de *databases*

Atualmente, existem alguns *hubs* de dados bem interessantes na internet, que fazem conexão com o **R**. Veremos alguns deles nessa subseção. A começar pelo Quandl, que possui um pacote de mesmo nome no **R**.[14] Instale o pacote e rode o código abaixo para baixar uma série.

```
library(Quandl)
selic.mensal <- Quandl('BCB/4189', type='zoo',
                       start_date = '1999-07-01')
```

Outra plataforma interessante é a *quantmod*.[15] Novamente aqui, antes de qualquer coisa, você precisa instalar o pacote de mesmo nome. Feito isso, está apto a rodar o código como abaixo. A função *getFX* pega taxas de câmbio do provedor **Oanda**. Para séries gerais, utilize a função *getSymbols*. Atualmente, estão disponíveis as bases de dados yahoo, google, FRED St. Louis e a própria Oanda.

```
library(quantmod)
# Downloading data from FRED
datalines <- cbind("GDPC96", "CPILFESL",
                   "FEDFUNDS", "CPIAUCSL")
```

[14] Para maiores detalhes sobre isso, vá em https://www.quandl.com/tools/r.

[15] Informações sobre ela, vá em http://www.quantmod.com/.

```
getSymbols(datalines, src = "FRED")
```

```
## [1] "GDPC96"   "CPILFESL" "FEDFUNDS" "CPIAUCSL"
```

3.6 Pacotes brasileiros para importação de dados

É com enorme felicidade que tenho visto a maior penetração do R no país. Ao longo das últimos meses, por onde passo, sou abordado por pessoas que conhecem a Análise Macro e estão interessadas em aprender o R. Passados quase três anos do início desse projeto - sim, estamos quase comemorando três anos! -, não deixa de ser bastante auspicioso verificar o avanço do uso da linguagem entre estudantes, profissionais de mercado e professores. Uma das formas de avaliar esse avanço, diga-se, é na produção de pacotes para coleta de dados nas principais bases do país. Nessa subseção, vou falar dos pacotes atualmente disponíveis para pegar dados do IPEADATA, FGV, IBGE e Banco Central.

3.6.1 BETS

O pacote que mais tenho utilizado atualmente para pegar dados dessas bases é o BETS, pacote produzido pelo pessoal da FGV. Está disponível no CRAN e tem se mostrado bastante estável, à medida que alguns bugs foram corrigidos. Ele pode ser utilizado para pegar dados do Banco Central, IBGE e da FGV. Um vignette do pacote está disponível aqui.

```
library(BETS)
ipca = BETS.get(433)
```

3.6.2 rbcb

Outro pacote que tenho utilizado é o rbcb, produzido pelo Wilson Freitas. Ele serve, como o próprio nome já entrega, para coletar dados do Banco Central.

```
library(rbcb)
ipca = get_series(433)
```

3.6.3 ribge

Um pacote que conheci recentemente foi o ribge, que ainda está em fase de desenvolvimento, disponível no GitHub. Tomare que ao longo dos próximos meses esteja plenamente funcional!

3.6.4 sidrar

O pacote sidrar que coleta dados do SIDRA, base de dados do IBGE, diretamente para o R. Pacote bem legal para quem precisa trabalhar com as pesquisas da instituição, tais como a PNAD Contínua, PME, PMC, Contas Nacionais, etc. Abaixo, coloco um exemplo de como utilizar o pacote.

```
library(sidrar)
tabela = get_sidra(api='/t/6381/n1/all/v/4099/p/all/d/v4099%201')
times = seq(as.Date('2016-01-01'),
            as.Date('2017-08-01'), by='month')
desemprego = data.frame(time=times,
                        desemprego=tail(tabela$Valor, 20))
```

3.6.5 ecoseries

Por fim, outro pacote que também coleta dados do SIDRA IBGE, Banco Central e IPEADATA é o ecoseries, disponível no CRAN.

```
library(ecoseries)
ipca = window(ts(series_ipeadata('36482',
 periodicity = 'M')$serie_36482$valor,
 start=c(1979,12), freq=12), start=c(1999,08))
```

4 Gráficos

A próxima etapa no nosso aprendizado é criar gráficos. Já vimos uma série de objetos dentro do R, mas ainda não colocamos os mesmos de forma gráfica. Bom, antes disso e daqui para frente, nós vamos trabalhar com a ideia de *série temporal*. Definir a mesma, em termos precisos, não é o objetivo desse curso, mas para nossas pretensões, basta que o leitor saiba que uma série temporal é dada por sequência de variáveis aleatórias distruídas no tempo.[16]

A função básica para gerar gráficos no R é a função *plot*. Com ela, você poderá gerar gráficos simples, mas bastante interessantes para suas análises e apresentações diárias.[17] Como dito acima, antes de mais nada, precisamos transformar os dados que coletamos até aqui em séries temporais. Para isso, aplica-se o código abaixo sobre, por exemplo, o objeto *data*.

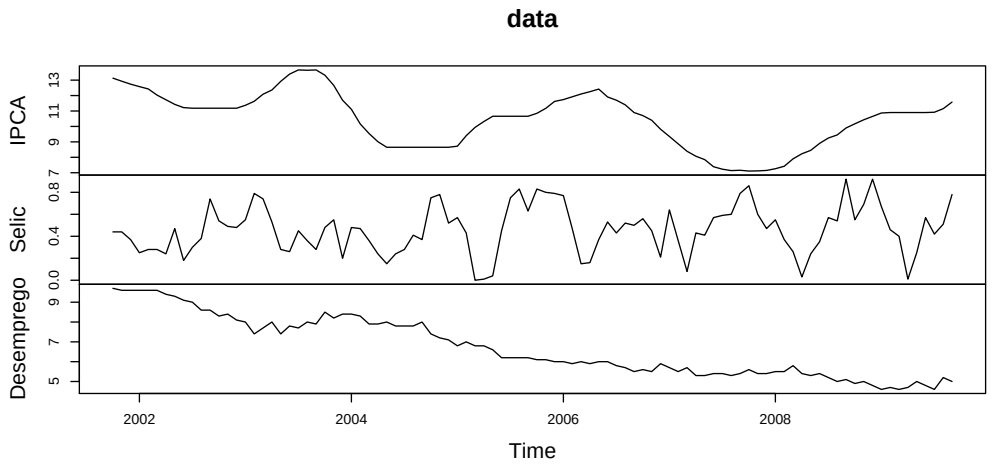
```
data = read.csv2('arquivo02.csv', header=TRUE, sep=',',
                 dec='.')[,-1]
data <- ts(data, start=c(2001, 10), freq=12)
```

A função *ts* transforma o objeto *data* em uma série temporal mensal (frequência=12), iniciada em outubro de 2001. Com efeito, podemos pedir o gráfico com a função *plot*, sem maiores detalhamentos. Apenas para que você veja como fica.

```
colnames(data) <- c('IPCA', 'Selic', 'Desemprego')
plot(data)
```

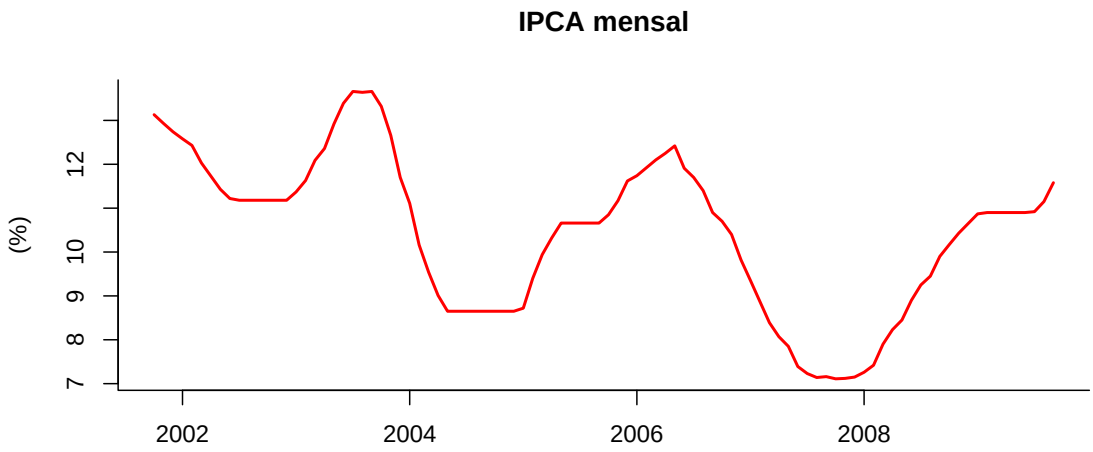
[16] Como dito anteriormente, para uma rápida introdução à séries temporais no R, ver Lundholm (2011). Para maiores aprofundamentos, ver Lima (2014), Cowpertwait and Metcalfe (2009) e Pfaff (2008).

[17] Para uma referência sobre gráficos no R, ver Murrel (2006). O material on-line está disponível aqui.



Caso queiramos apenas a série IPCA, devemos dar o seguinte comando.

```
plot(data[,1], xlab='', ylab='(%)', bty='l', col='red',
     main='IPCA mensal', lwd=2, lty=1)
```



Observe que nesse caso, nós já acrescentamos alguns argumentos à função *plot*. Você pode ver alguns deles aqui. No caso acima, plotamos o IPCA, retiramos o *label* do eixo das abscissas (xlab), colocamos (%) no eixo das ordenadas (ylab), deixamos apenas a linha da esquerda (bty), colorimos de vermelho (col), colocamos um título (main), alteramos a intensidade da linha (lwd) e mantivemos a linha contínua (lty).[18]

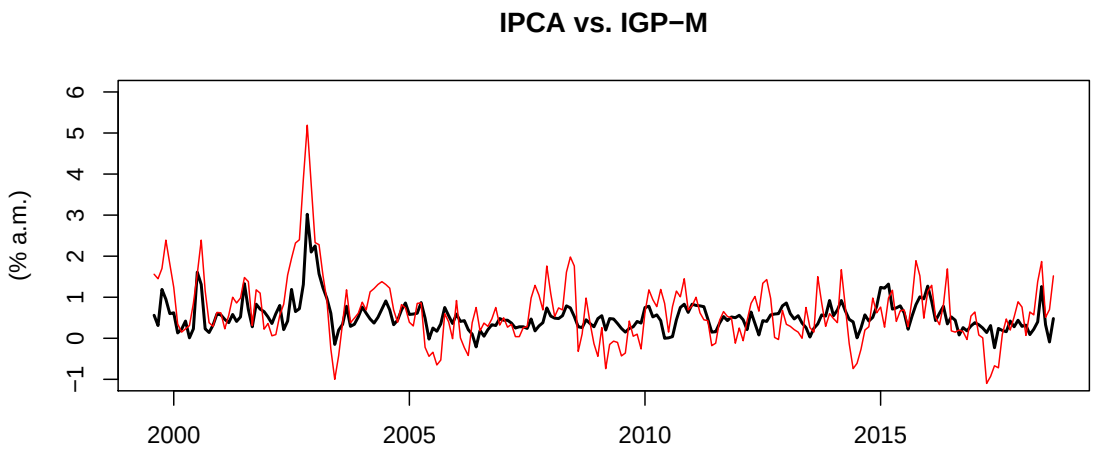
[18]Para um código de cores, ver, por exemplo, aqui.

4.1 Duas séries no mesmo gráfico

Para colocar duas séries no mesmo gráfico de forma simples, use a função *lines*. Primeiro, vamos importar os dados.

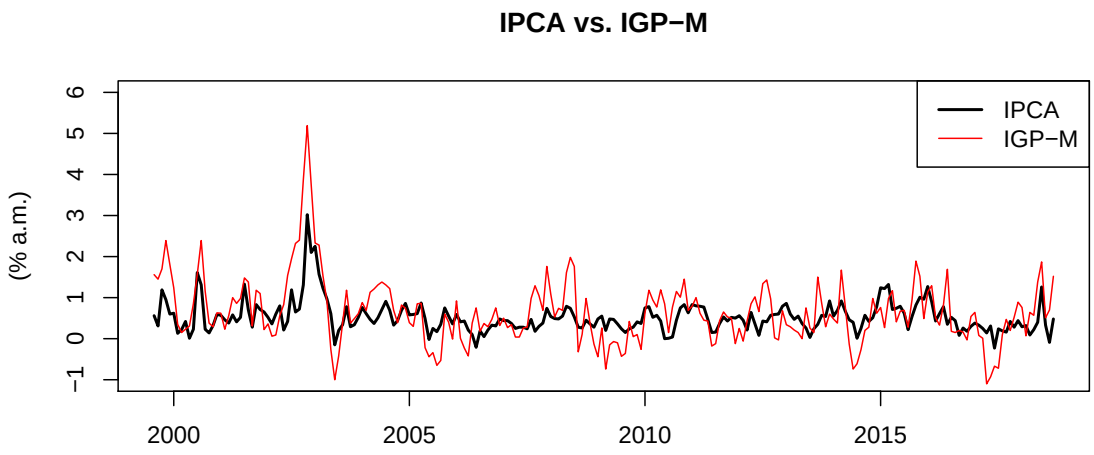
```
library(BETS)
ipca = BETS.get(433, from='1999-08-01')
igpm = BETS.get(189, from='1999-08-01')
```

```
plot(ipca, lwd=2, xlab='',
     ylab='(% a.m.)', main='IPCA vs. IGP-M',
     ylim=c(-1,6))
lines(igpm, col='red', ylim=c(-1,6))
```



Para colocar legenda no gráfico, utilize a função *legend*.

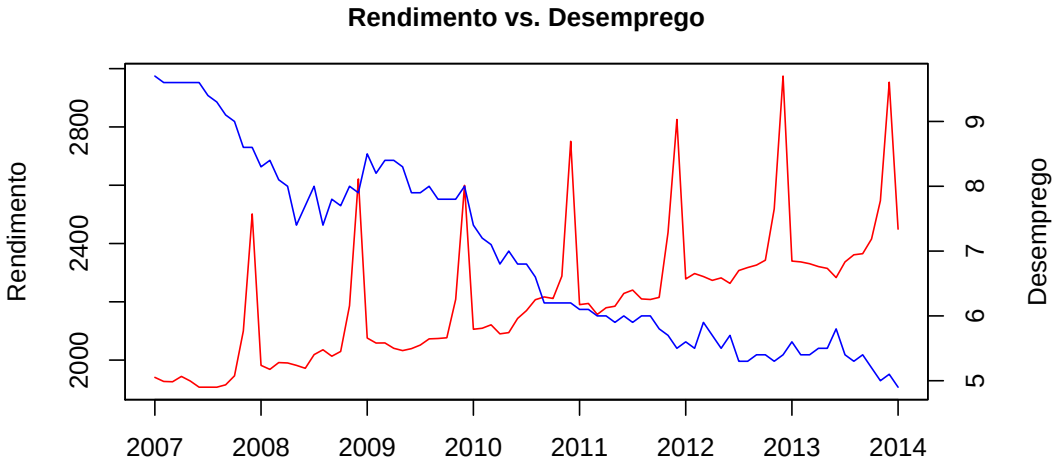
```
plot(ipca, lwd=2, xlab='', ylab='(% a.m.)',
     main='IPCA vs. IGP-M',
     ylim=c(-1,6))
lines(igpm, col='red', ylim=c(-1,6))
legend('topright', col=c('black','red'), lty=c(1,1), lwd=c(2,1),
       legend=c('IPCA', 'IGP-M'))
```



4.2 Colocar eixo secundário no gráfico

Algumas séries possuem escalas muito diferentes, logo o melhor é colocar uma no eixo principal e outra no eixo secundário. Isso é feito com o código abaixo.

```
rendimento = BETS.get(10790, from='2007-01-01',
                      to='2014-01-01')
desemprego = BETS.get(10777, from='2007-01-01',
                      to='2014-01-01')
par(mar=c(5,4,4,5)+.1)
plot(rendimento, xlim=c(2007,2014),
     xlab='', ylab='Rendimento',
     col='red')
par(new=T)
plot(desemprego, xlim=c(2007,2014), xlab='', ylab='',
     xaxt='n',yaxt='n', col='blue')
axis(4)
mtext('Desemprego',side=4,line=3)
mtext('Rendimento vs. Desemprego', side=3, line=1, font=2)
mtext('Vermelho: rendimento, Azul: desemprego', side=1, line=3)
```

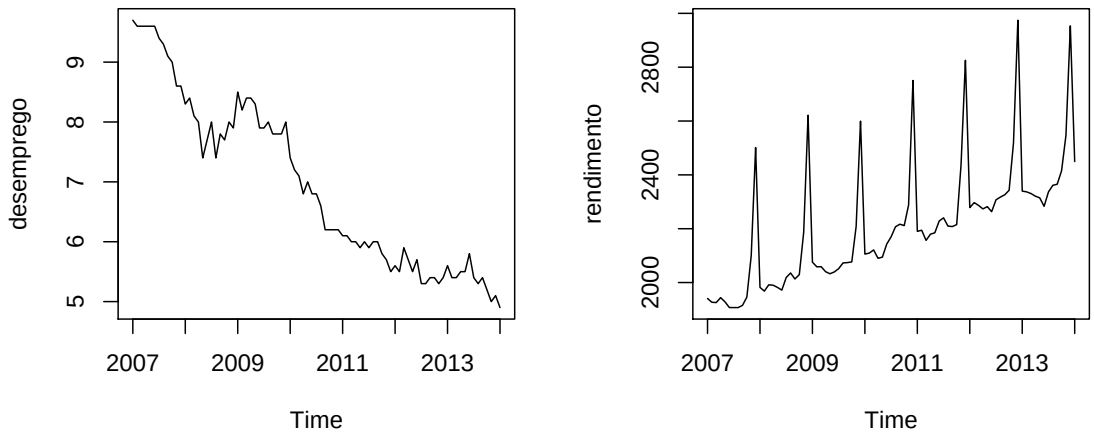


Vermelho: rendimento, Azul: desemprego

4.3 Gráficos lado a lado

Uma opção gráfica interessante no $\mathbf{R}$ é colocar um gráfico ao lado do outro. Isso é possível com a função *par*. No código abaixo, setamos o nosso ambiente gráfico com o parâmetro *mfrow* igual a uma linha e duas colunas, de forma que um gráfico fique do lado do outro.

```
par(mfrow=c(1,2))
plot(desemprego)
plot(rendimento)
```



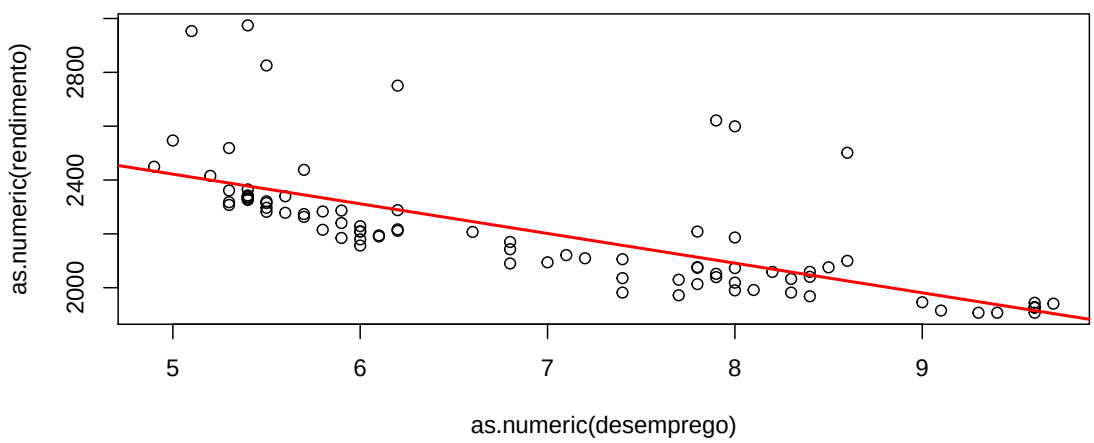
Você pode, nesse contexto, colocar vários gráficos, setando o ambiente adequado com o

parâmetro *mfrow*. Faremos isso na lista de exercícios, não se preocupe.

4.4 Gráfico de Correlação

Às vezes, pode ser interessante fazer um gráfico para ver a correlação entre duas variáveis, colocando uma linha de regressão entre elas. Isso é feito no código abaixo.

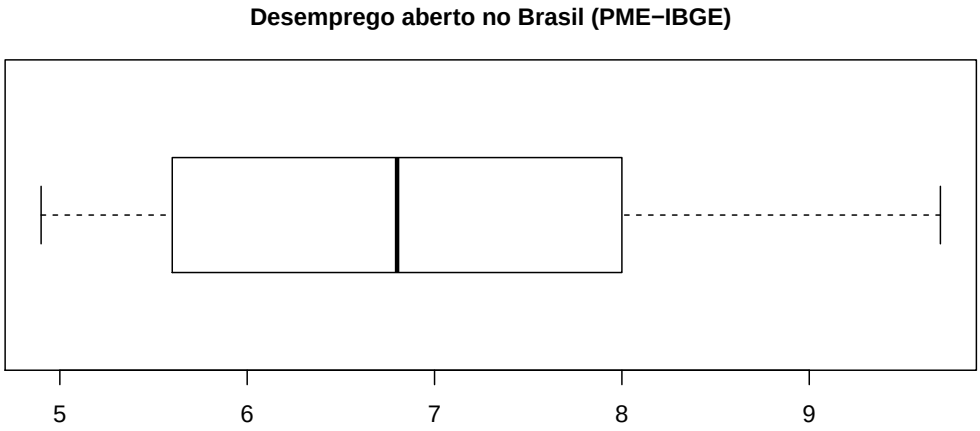
```
plot(as.numeric(desemprego), as.numeric(rendimento))
fitline <- lm(rendimento~desemprego)
abline(fitline, col='red', lwd=2)
```



4.5 Boxplots

Um boxplot também é bastante útil em análises simples e rápidas. Com uma linha de código, você faz isso no R. Claro, você pode querer colocar um título com a função *mtext*.

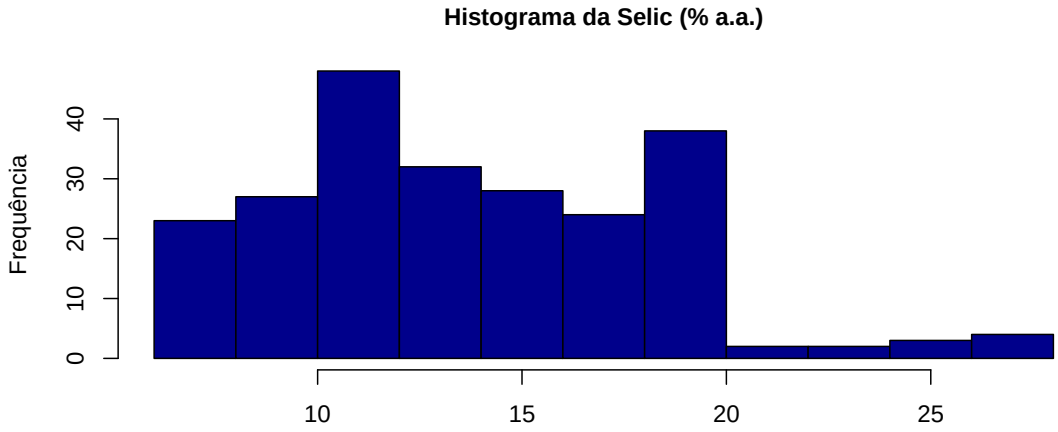
```
boxplot(desemprego, horizontal=TRUE)
mtext('Desemprego aberto no Brasil (PME-IBGE)',
      side=3, line=1, font=2)
```



4.6 Histogramas

Histogramas são muito úteis para ver a distribuição de uma determinada série.

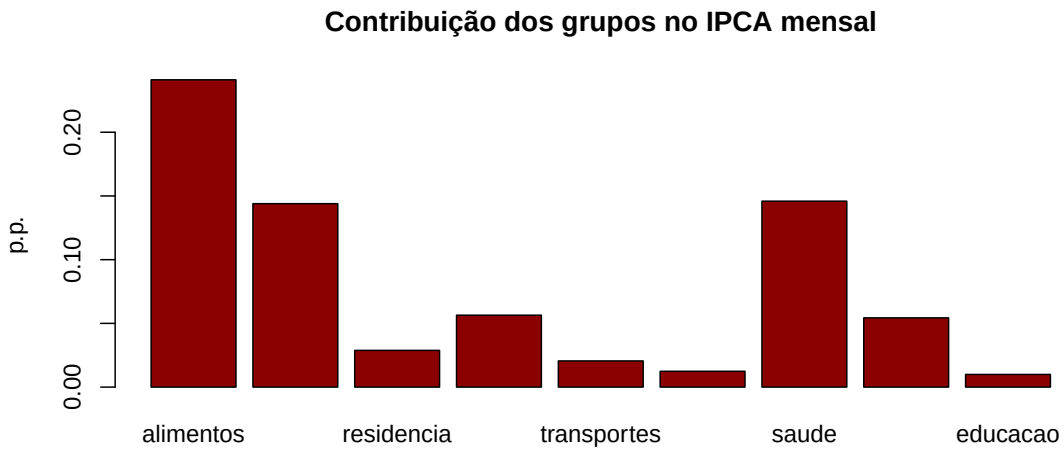
```
selic = BETS.get(4189, from='1999-08-01')
hist(selic, xlab='', ylab='Frequência', col='darkblue', main='',
     breaks=10)
mtext('Histograma da Selic (% a.a.)',
      side=3, line=1, font=2)
```



4.7 Barplots

Um gráfico de barras pode ser interessante em alguns casos. Abaixo coloco a contribuição dos nove grupos do IPCA no índice mensal.[19]

```
barplot(as.matrix(exemplo), names=colnames(exemplo),
        col='darkred',
        main='Contribuição dos grupos no IPCA mensal',
        ylab='p.p.')
```



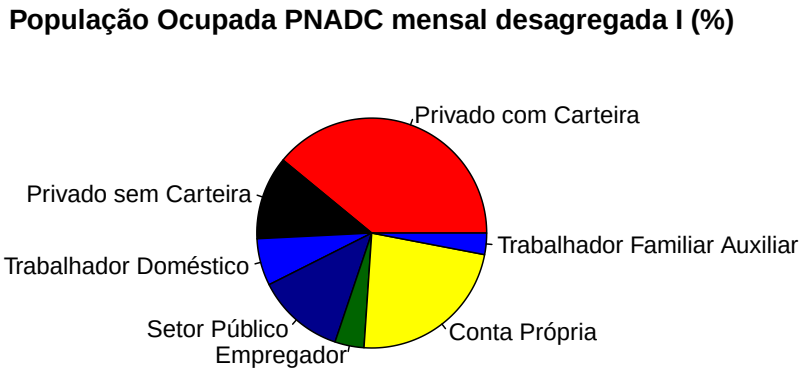
4.8 Pie Charts

Gráficos de torta são interessantes para ver a participação de um determinado subsetor. No código abaixo fazemos isso.[20]

```
pie(colMeans(pesos.po1), col=c('red','black','blue','darkblue',
                               'darkgreen', '367', '556'),
  main='População Ocupada PNADC mensal desagregada I (%)')
```

[19] Esse gráfico, em particular, precisa de um código auxiliar para ser gerado. Para vê-lo, você deve abrir o script dessa seção.

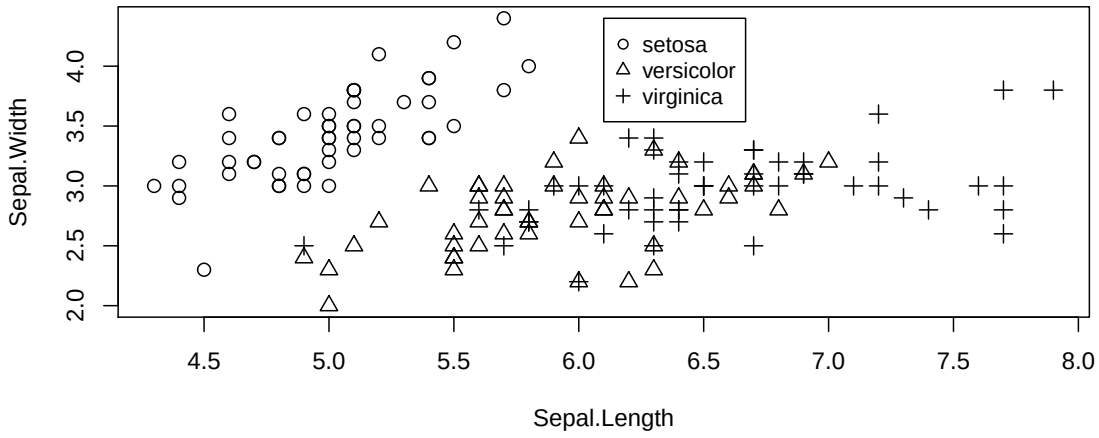
[20] O código, entretanto, é apenas parcial. Ele está no script dessa seção para que você possa gerá-lo.



4.9 Gráficos multivariados

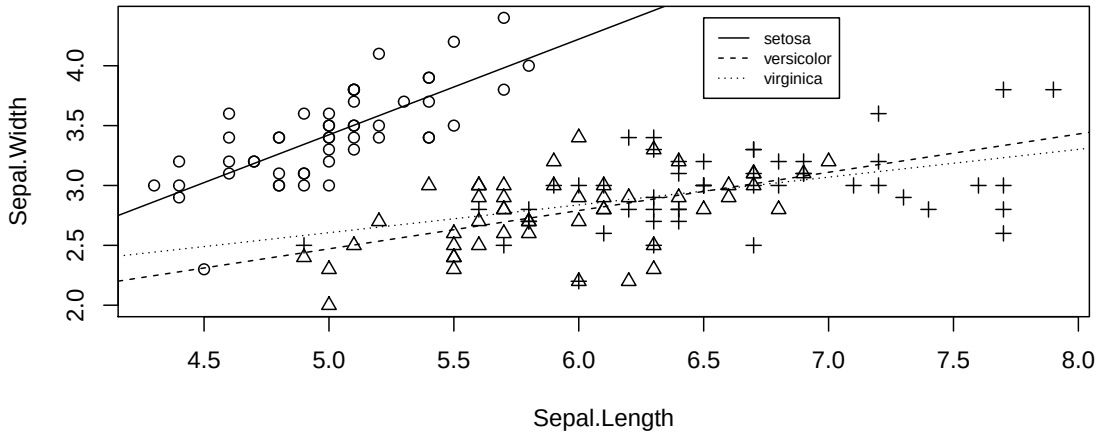
Para ilustrar a construção de gráficos multivariados, vamos utilizar o conjunto de dados `iris`. O código abaixo cria um gráfico de correlação entre duas das variáveis do *dataset*, diferenciando ainda pelas três espécies de iris.

```
library(UsingR)
with(iris, plot(Sepal.Length, Sepal.Width,
                pch=as.numeric(Species), cex=1.2))
legend(6.1, 4.4, c('setosa', 'versicolor', 'virginica'),
       cex=0.9, pch=1:3)
```



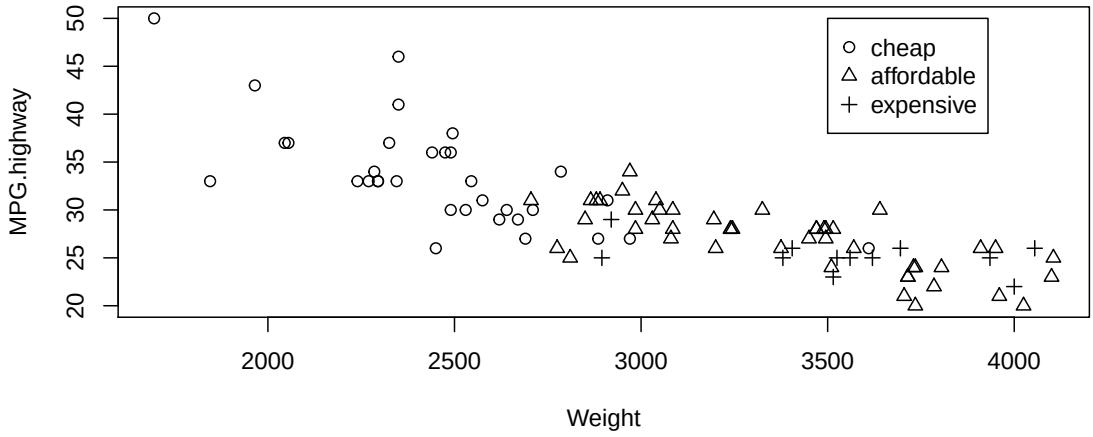
Isto é, além de plotar os pares (x, y), o código acima utiliza o argumento `pch=as.numeric(Species)` de modo a forçar diferentes formatos para cada uma das três espécies. A ideia aqui é adicionar uma terceiro variável ao gráfico, que neste caso seriam as variáveis categóricas representadas pelas espécies. Ademais, é adicionada uma legenda ao gráfico, de acordo com as três espécies. Podemos aumentar a complexidade do nosso gráfico ao adicionar uma *reta de regressão*:

```
fm = Sepal.Width ~ Sepal.Length
plot(fm, iris, pch=as.numeric(Species))
out = mapply(function(i, x) abline(lm(fm, data=x), lty=i),
             i=1:3, x=split(iris, iris$Species))
legend(6.5, 4.4, levels(iris$Species), cex=0.7, lty=1:3)
```



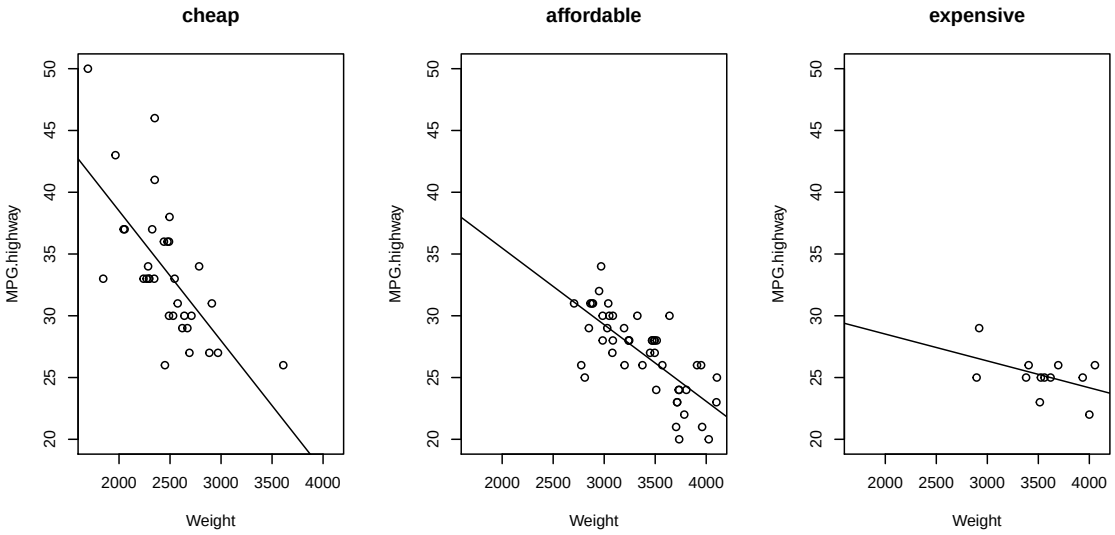
O gráfico torna visível a diferença na relação para a espécie *setosa*, dado que a inclinação é bastante distinta. Ainda nessa linha, a propósito, variáveis numéricas podem ser transformadas em variáveis categóricas para propósitos de agrupamento. Para ilustrar, vamos utilizar abaixo o conjunto de dados `Cars93`.

```
Cars93 = transform(Cars93, price=cut(Price, c(0,15,30,75),
                                     labels=c('cheap', 'affordable',
                                              'expensive')))
plot(MPG.highway ~ Weight, Cars93, pch=as.numeric(price))
legend(3500, 50, levels(Cars93$price), pch=1:3)
```



O problema do gráfico acima é que fica difícil identificar a tendência dos diferentes grupos. Isso pode ser resolvido se plotamos um gráfico para cada um. O código abaixo operacionaliza.

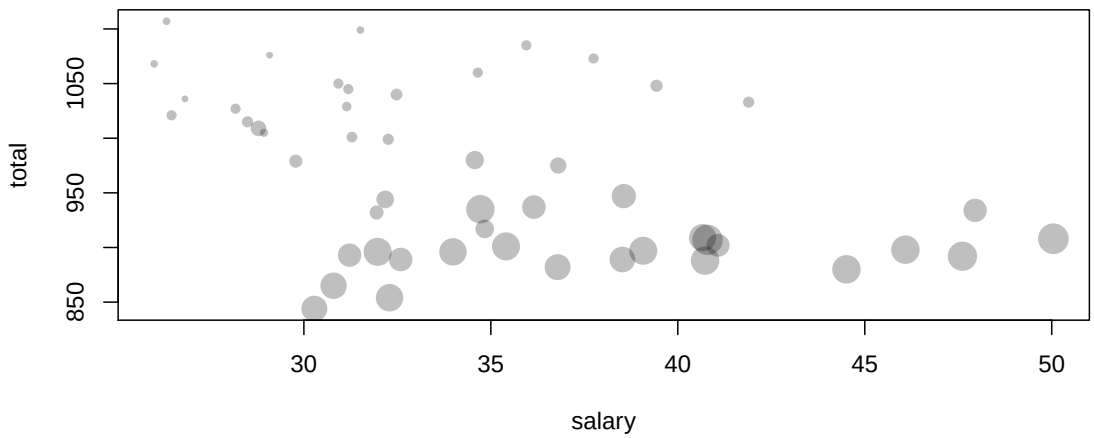
```
l = split(Cars93, Cars93$price)
par(mfrow=c(1, length(l)))
fm = MPG.highway ~ Weight
xlim = range(Cars93$Weight)
ylim = range(Cars93$MPG.highway)
mapply(function(x, nm) {
  plot(fm, data=x, main=nm, xlim=xlim, ylim=ylim)
  abline(lm(fm, data=x))
}, l, names(l))
```



4.9.1 Gráfico de bolhas

Vamos mostrar agora uma outra forma bastante comum de inserir uma terceira variável numérica em um gráfico. Para isso, vamos utilizar o conjunto de dados SAT no código abaixo.

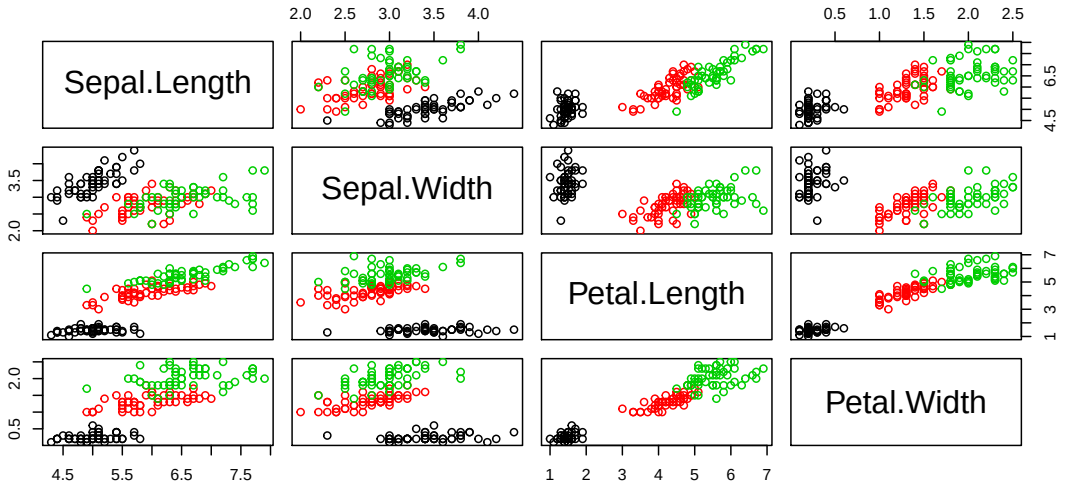
```
plot(total ~ salary, data=SAT, cex=sqrt(perc/10),
     pch=16,
     col=rgb(red=0, green=0, blue=0, alpha=0.250))
```



4.9.2 Pairs plots

Um *pairs plot* mostra gráficos de correlação para cada *pair* de uma lista de variáveis numéricas. Por exemplo,

```
species = iris$Species
values = Filter(is.numeric, iris)
pairs(values, col=species)
```



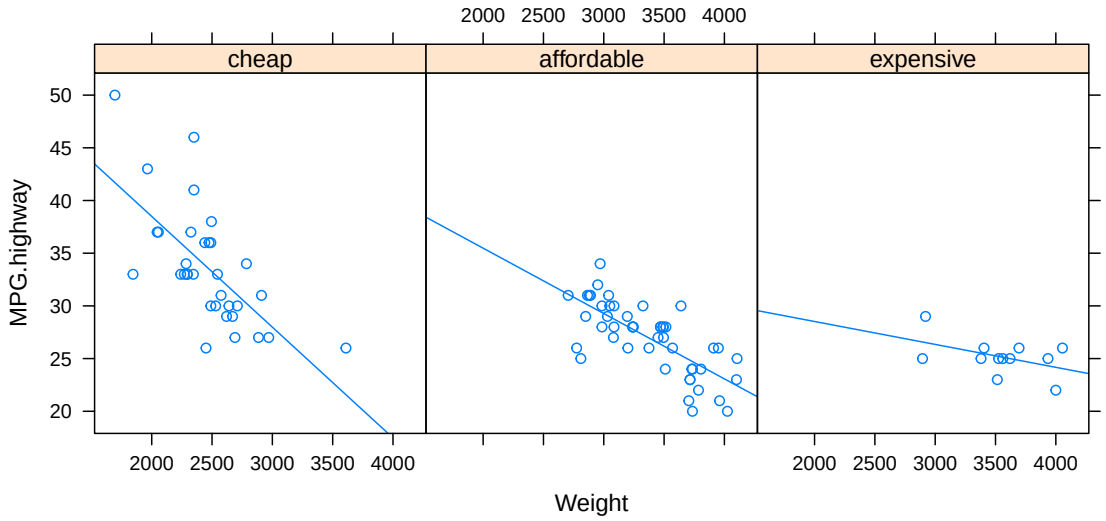
4.10 Gráficos com lattice

O pacote `lattice` é um poderoso e elegante sistema de visualização de dados multivariados. A elegância provém do uso de fórmulas de R de modo a especificar dados *splitados*. A fórmula básica é extendida de momo a incluir variáveis condicionais como

$$y \quad \sim \quad x \quad | \quad g1 + g2 + ...$$

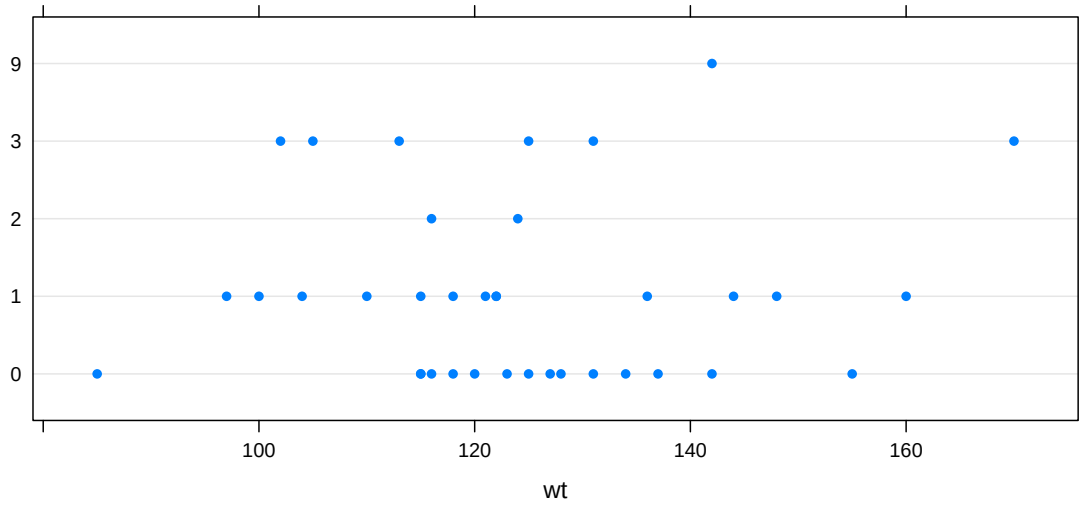
de modo que as variáveis condicionais $g1, g2, ...$ são utilizadas para dividir os dados. Por exemplo, o gráfico que fizemos acima pode ser facilmente replicado com o código abaixo.

```
require(lattice)
Cars93 = transform(Cars93, price=cut(Price, c(0,15,30,75),
                                     labels=c('cheap', 'affordable',
                                              'expensive')))
xyplot(MPG.highway ~ Weight | price, data=Cars93,
       layout=c(3,1), type=c('p', 'r'))
```



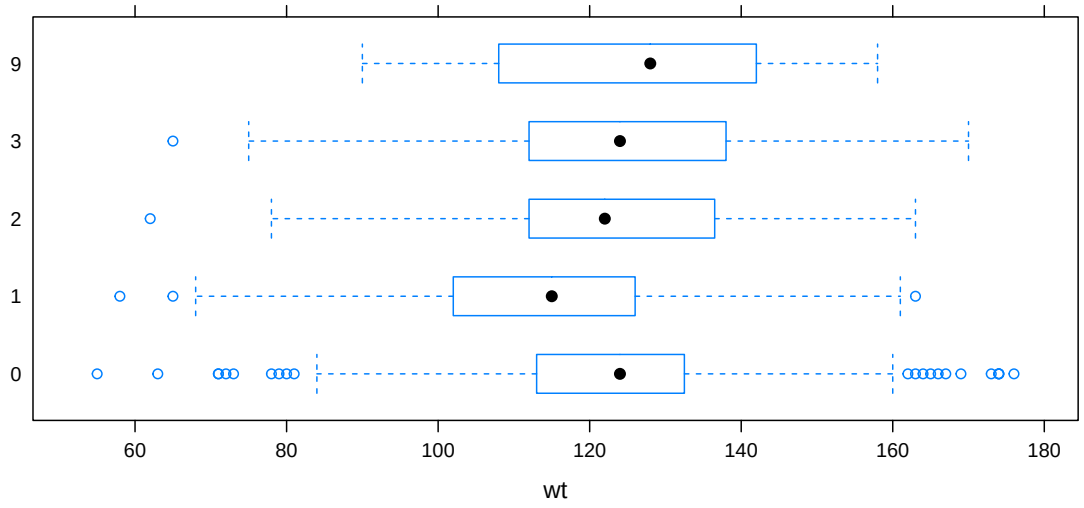
4.10.1 Dot charts

```
dotplot(factor(smoke) ~ wt, data=babies, subset=wt<999 & ded==3)
```



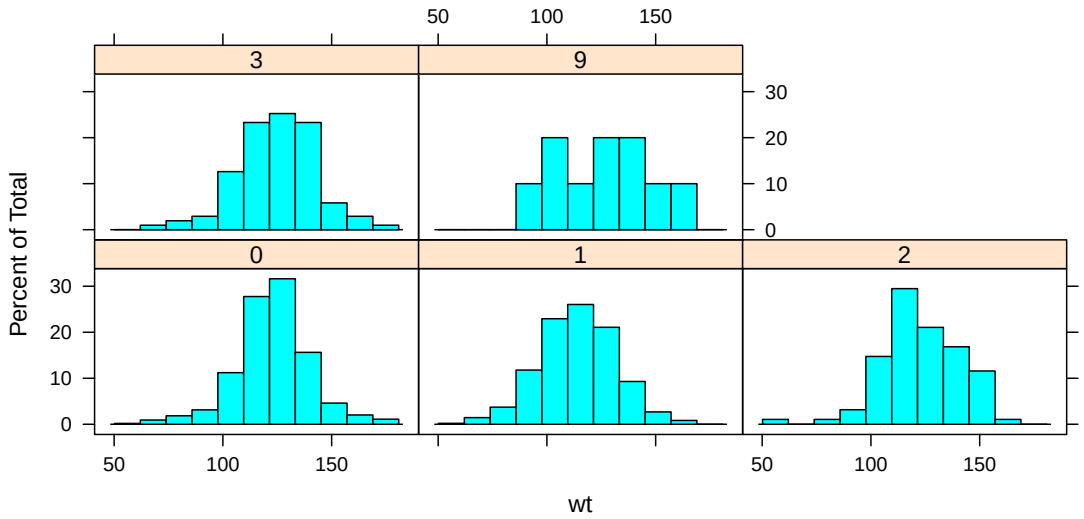
4.10.2 Boxplots

```
bwplot(factor(smoke) ~ wt, data=babies, subset=wt<999)
```



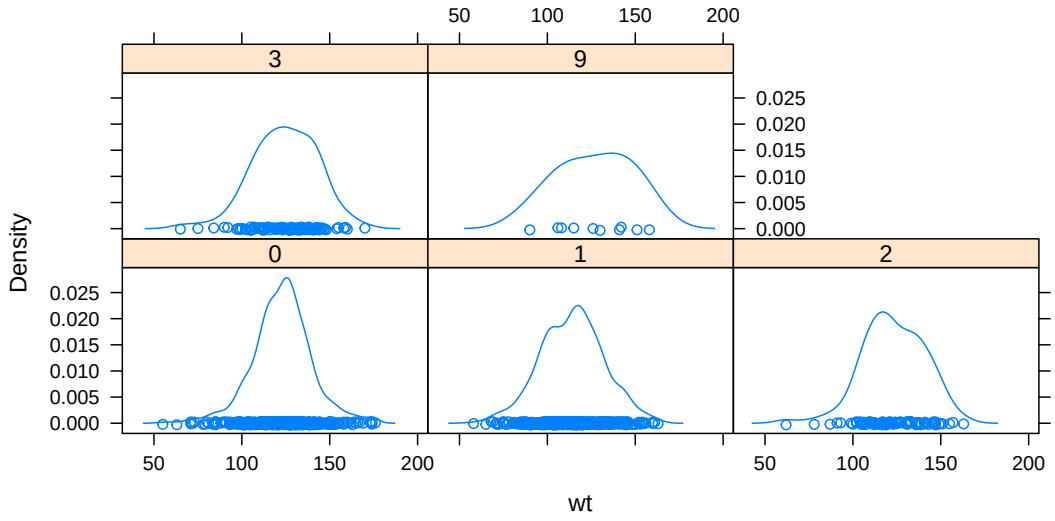
4.10.3 Histogramas

```
histogram(~wt|factor(smoke), data=babies, subset=wt<999)
```



4.10.4 Gráficos de densidade

```
densityplot(~wt|factor(smoke), data=babies, subset=wt<999)
```



4.11 Gráficos com ggplot2

O pacote `ggplot2` (ver Wickham (2009)) utiliza a *gramática dos gráficos* de Wilkinson (ver Wilkinson (2005)), de modo que um gráfico é o resultado da combinação de diversos componentes. Vamos introduzir o pacote com um exemplo detalhado.

Aesthetics ou *Atributos Estéticos*: Começamos definindo um objeto `ggplot` para um conjunto de dados.

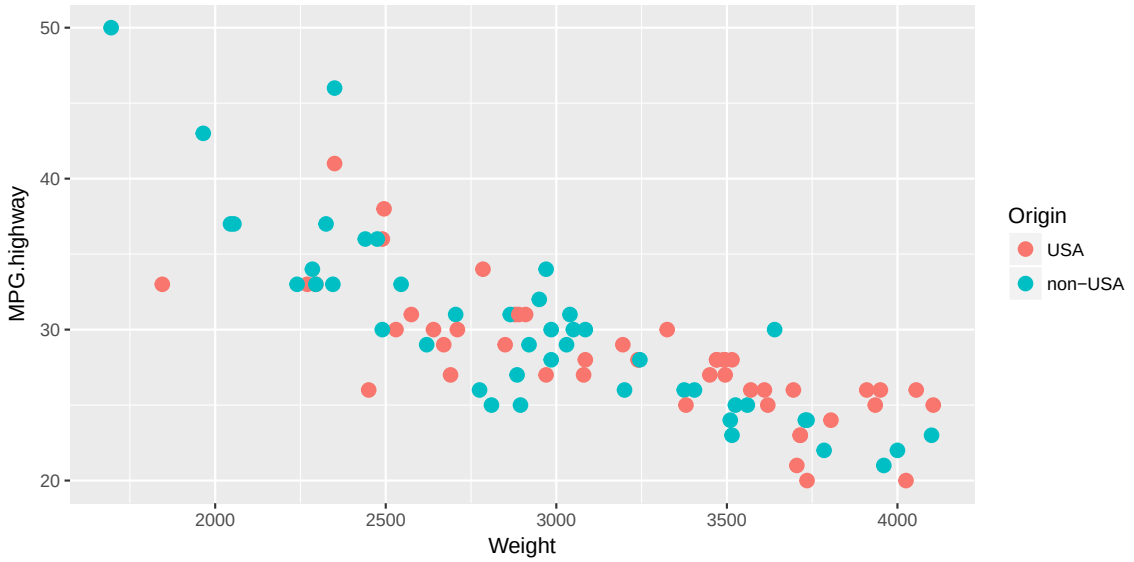
```
require(ggplot2)
p = ggplot(Cars93)
```

Criamos acima apenas um objeto, nenhum gráfico ainda é gerado. Para fazê-lo, precisamos de dois componentes principais: **aesthetics** e **geometries** - objetos geométricos. Aesthetics mapeiam as variáveis em um conjunto de dados de modo que suas propriedades possam ser colocadas em um gráfico. Por exemplo, tamanho, forma e cor são aesthetics. Ademais, valores para x e y também serão aesthetics. Abaixo declaramos.

```
p = p + aes(x=Weight, y= MPG.highway, color=Origin)
```

Geoms: uma vez definidos os atributos estéticos, precisamos definir o objeto geométrico que estamos interessados. Com efeito, já temos um gráfico para mostrar.

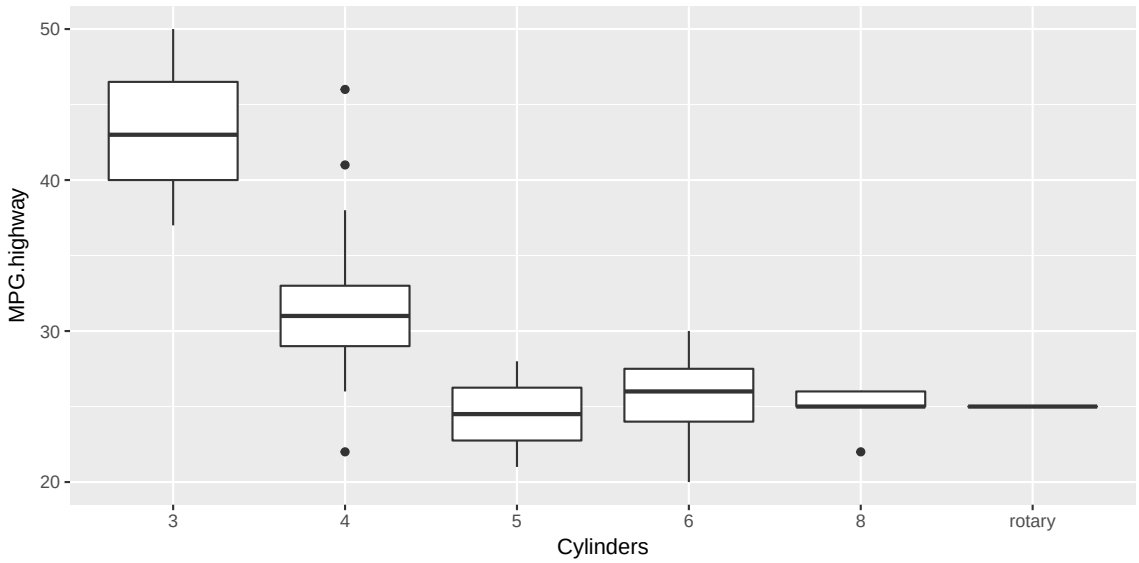
```
p + geom_point(cex=3)
```



4.11.1 Grouping

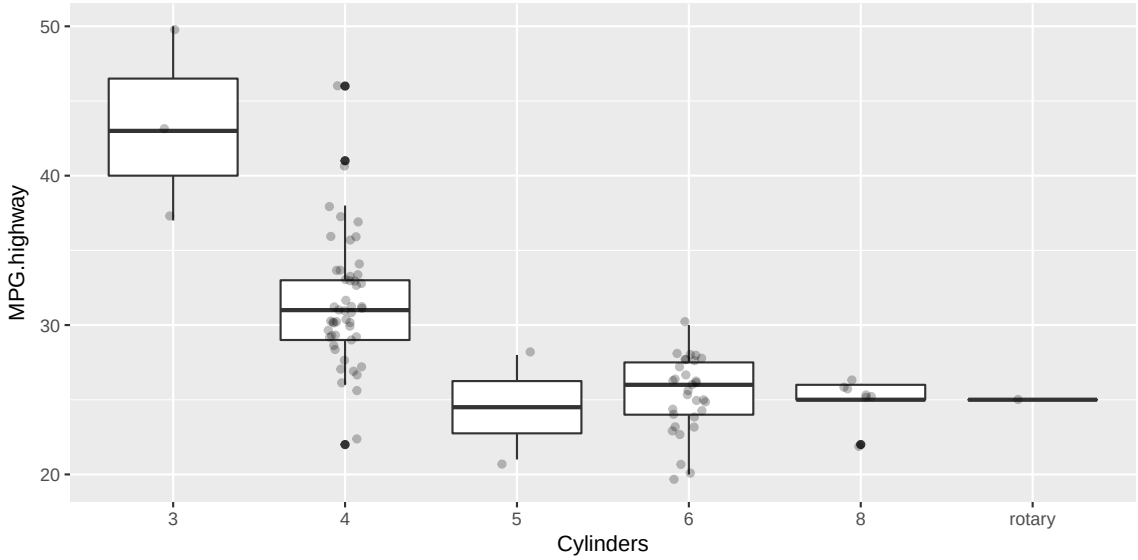
O objeto geométrico `geom_point` mapeia cada ponto de dados em uma representação gráfica. Esse, entretanto, não é o caso para gráficos que reduzem a visualização de dados, tal como histogramas ou boxplots. Para esses, os objetos correspondem a grupos de dados. Abaixo um exemplo.

```
p = ggplot(Cars93, aes(x=Cylinders, y=MPG.highway))
p + geom_boxplot()
```



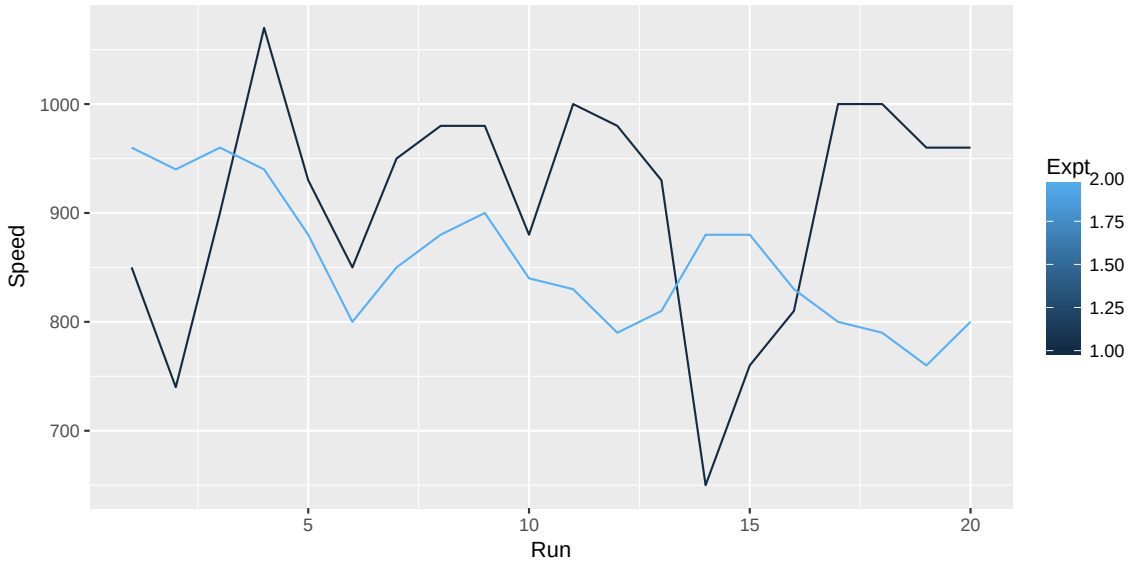
Ou

```
p + geom_boxplot() +
  geom_jitter(position=position_jitter(w=0.1), alpha=.25)
```



Dados longitudinais, por suposto, podem gerar uma situação onde o agrupamento padrão pode não ser suficiente. O conjunto de dados `morley`, por exemplo, guarda 100 diferentes medidas de velocidade da luz. Há 20 rodadas para cada um dos cinco experimentos. Nós podemos então traçar as medidas de cada rodada ao plotar um gráfico de linha com `Run` no eixo x e `Speed` no eixo y. Nós podemos então construir o gráfico por diferentes grupos de acordo com a variável `Expt`. Veja abaixo.

```
m = subset(morley, Expt %in% 1:2) # apenas os dois primeiros exper
p = ggplot(m, aes(x=Run, y=Speed, group=Expt, color=Expt))
p + geom_line()
```

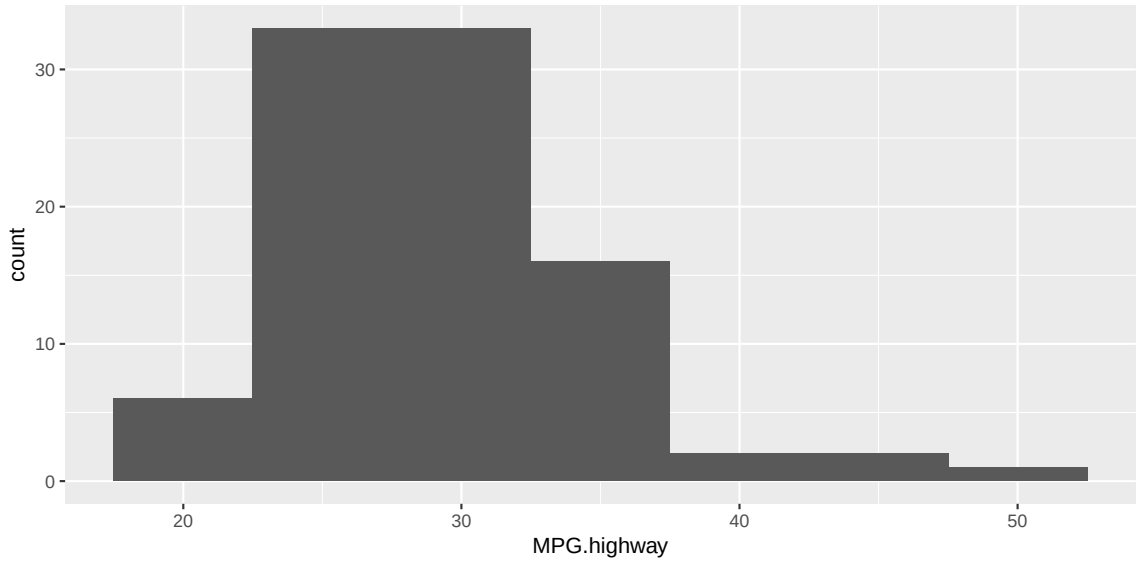


4.11.2 Transformações estatísticas

Estatísticas sumarizam os dados, usualmente reduzindo sua dimensão. Por exemplo, a média sumariza um conjunto de dados com n valores para um único número.

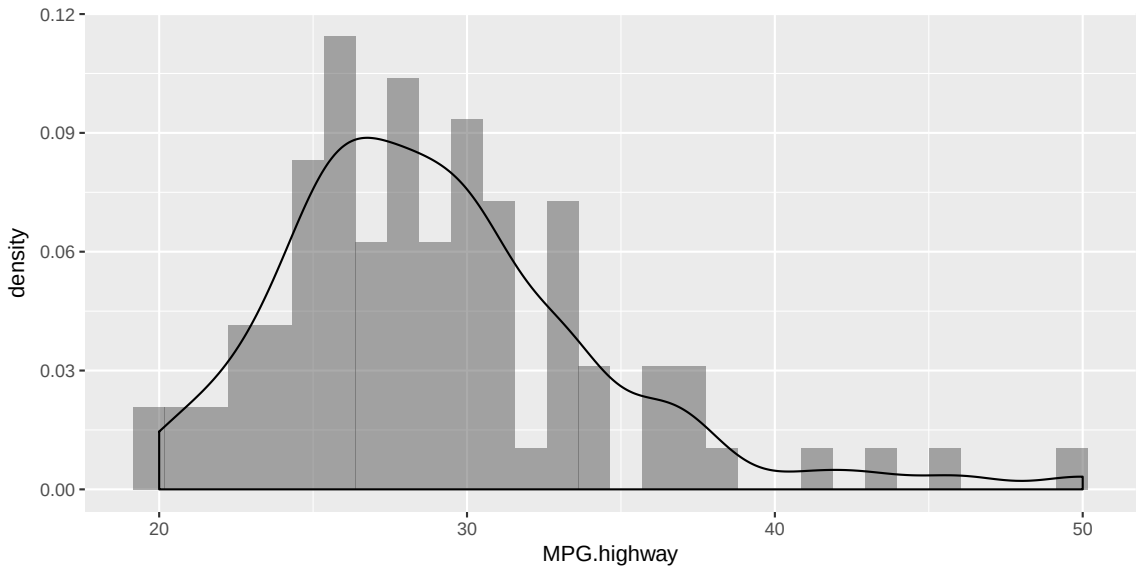
stat_bin: Na gramática do `ggplot2`, transformações estatísticas (**stats**) sumarizam os dados antes deles serem processados. Abaixo um exemplo.

```
p = ggplot(Cars93, aes(x=MPG.highway))
p + stat_bin(binwidth = 5)
```



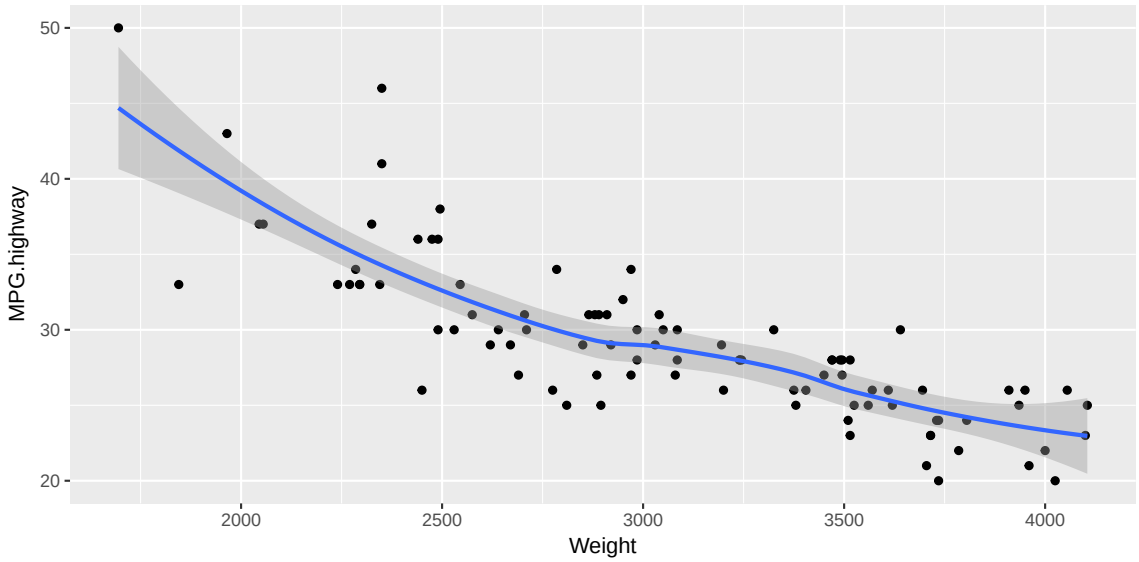
stat_density: Gráficos de densidade são feitos de maneira similar.

```
p = ggplot(Cars93, aes(x=MPG.highway, y=..density..))
p + geom_histogram(alpha=.5)+geom_density()
```



stat_smooth: Uma linha de tendência é um sumário estatístico de uma relação bivariada. Abaixo um exemplo com o `ggplot2`.

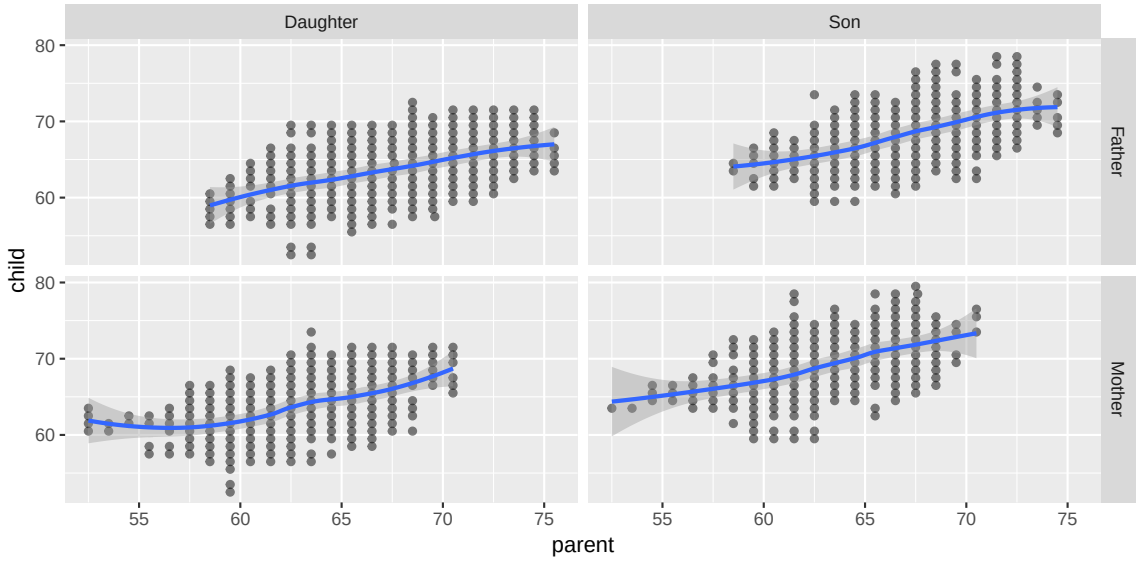
```
p = ggplot(Cars93, aes(x=Weight, y=MPG.highway))
p + geom_point() + geom_smooth()
```



4.11.3 Faceting

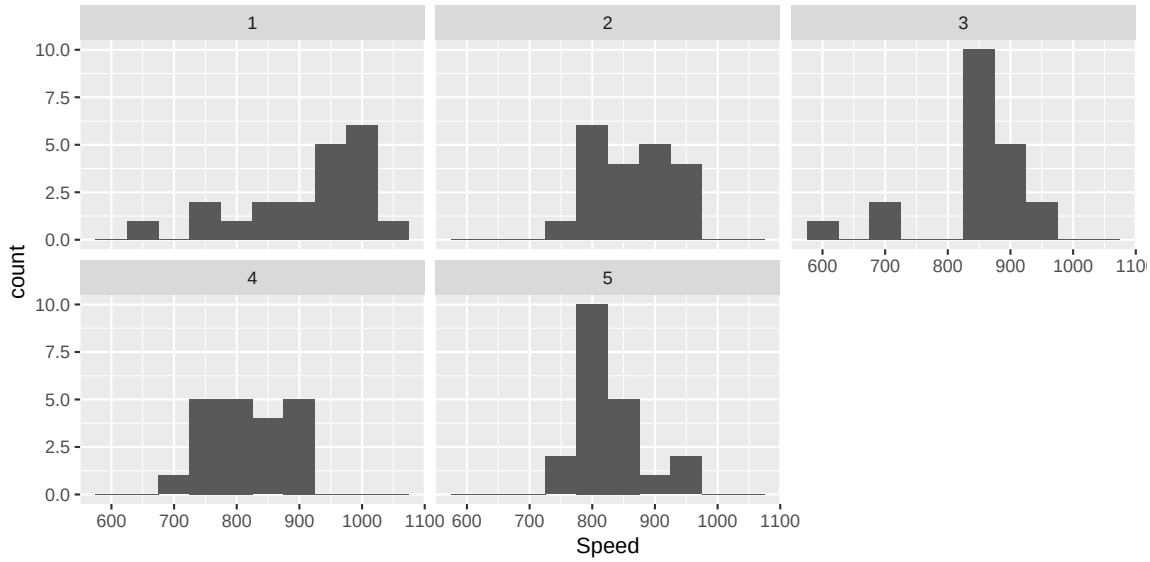
No pacote `lattice` podemos fazer comparações através de grupos por meio de painéis. No pacote `ggplot2`, essa facilidade está implementada através de *facets*. Há dois tipos: `facet_grid` e `facet_wrap`. Abaixo, exemplos.

```
p = ggplot(PearsonLee, aes(y=child, x=parent))
p + geom_point(alpha=.5) + geom_smooth(method='loess') +
  facet_grid(par ~ chl)
```



Ou,

```
p = ggplot(morley, aes(x=Speed)) + geom_histogram(binwidth=50)
p + facet_wrap(~Expt)
```



5 Ambientes Controlados

Essa seção encerra nossa Introdução ao R e de certa maneira, ela é um pouco mais avançada do que tudo o que vimos até aqui. De fato, as coisas que veremos aqui são mais voltadas para quem quer usar o $\mathbf{R}$ para programar e não apenas para aplicar alguns comandos sobre dados. Daí a ideia de um ambiente controlado, onde tudo o que foi feito parte de um objetivo pré-definido pelo programador. A despeito disso, algumas dessas estruturas, notadamente as de repetição, bem como a noção de funções dentro $\mathbf{R}$, podem ser bastante úteis nas nossas análises diárias de dados. Daí a motivação para apresentá-las aqui, mesmo em um curso de introdução à linguagem.

5.1 If-else

A primeira estrutura que veremos aqui é a famosa "If-Else", que todo usuário de EXCEL conhece. A ideia dessa estrutura é testar uma condição: se ela for satisfeita, então faça alguma coisa, se não for, faça outra coisa. Abaixo, um exemplo bem simples.

```
x <- 5
if(x> 0){
  print('Número Positivo')
} else {
  print('Não positivo')
}
```

[1] "Número Positivo"

Outro exemplo também bem simples.

```
x <- 10
if(x>4){
  z <- 4000
} else {
  z <- 300
}
```

5.2 For loops

A função *for* é particularmente interessante para nos poupar tempo na hora de fazer coisas repetitivas. A ideia aqui é automatizar certas rotinas que nos consumiria muito tempo se fóssemos fazer "na mão". Abaixo um primeiro exemplo simples, só para vermos a estrutura da função.

```
codigos <- c('Inflação', 'Nível de Atividade',
             'Política monetária', 'Política Fiscal',
             'Setor Externo', 'Economia Internacional')


for(i in 1:length(codigos)){
  print(codigos[i])
}
```

```
## [1] "Inflação"
## [1] "Nível de Atividade"
## [1] "Política monetária"
## [1] "Política Fiscal"
## [1] "Setor Externo"
## [1] "Economia Internacional"
```

O que fizemos aqui foi basicamente ir em cada um dos elementos do objeto *codigos* e executar a função *print*, ao invés de fazer isso seis vezes, "na mão".

5.3 While loops

A função *while* é similar à função *if-else*, com a diferença que ela repete a ação enquanto a condição estiver satisfeita e não apenas em um ponto do tempo. Veja o exemplo abaixo.

```
x <- 2
while(x < 6) {
 print(x)
 x <- x + 1
}
```

[1] 2 [1] 3 [1] 4 [1] 5

Enquanto x for menor do que 6, o *loop* executa a função *print* e soma x mais 1. Observe que só executássemos a função *print(x)*, o loop seria infinito, não é mesmo?

5.4 Repeat, break e next

A função *repeat* serve para manter uma ação dentro de um loop. Já a função break interrompe o loop. Veja o exemplo abaixo.

```
x <- 1


repeat {
   print(x)
   x = x+1
   if (x == 6){
       break
   }
}
```

[1] 1 [1] 2 [1] 3 [1] 4 [1] 5

A função *next*, por seu turno, serve para ignorar uma interação, sem, entretanto, terminá-la.

```
x <- 1:5


for (val in x) {
    if (val == 3){
        next
    }
    print(val)
}
```

[1] 1 [1] 2 [1] 4 [1] 5

5.5 Funções no R

O ambiente *function* permite criarmos funções dentro do R, customizando e automatizando nossas necessidades de coleta, tratamento, análise e apresentação de dados. Desde questões muito simples até programas completos. Nessa subseção apresentaremos exemplos bem simples de funções, para que você possa se ambientar a elas.[21]

```
x <- 1:4
Quadrado <- function(x) {
  return(x^2)
}
Quadrado(x)
```

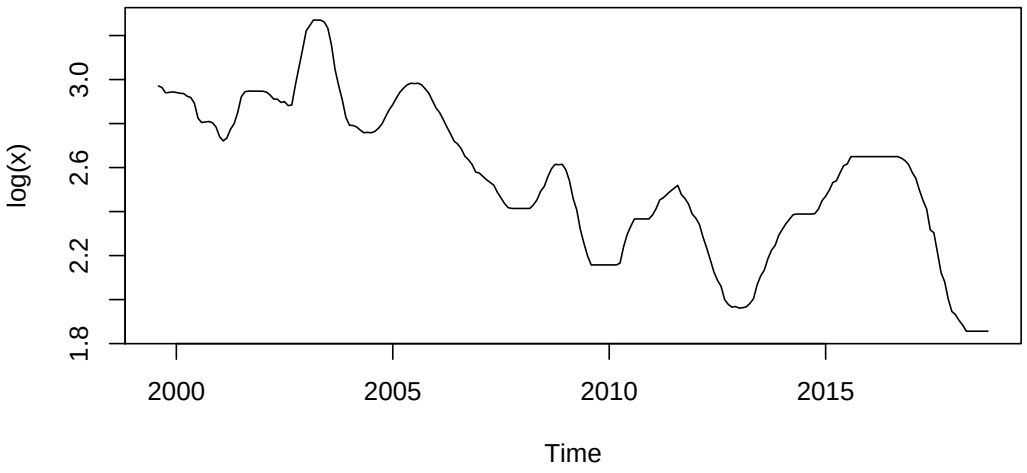
[1] 1 4 9 16

```
Soma <- function(x, y) {
  x+y
}
Soma( 3, 4)
```

[1] 7

```
get.log <- function(x){
          plot(log(x))
}
get.log(selic)
```

[21] Os exemplos foram retirados do Coursera e do site programiz.com.




```
x <- 1:30


above <- function(x, n){
          use <- x > n
          x[use]
}
above(x, 12)
```

[1] 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

```
columnmean <- function(z, removeNA=T) {


               nc <- ncol(z)
               means <- numeric(nc)
               for(i in 1:nc) {
                       means[i] <- mean(z[,i], na.rm=removeNA)
               }
               means
}
```

Um ambiente controlado é o espaço para que o usuário consiga programar, de fato, o que quiser dentro da linguagem. Você poderá praticar um pouco na nossa lista de exercícios. Para, entretanto, maiores informações sobre essa seção, recomendamos fortemente que

você leia Matloff (2009).

Referências Bibliográficas

Cowpertwait, P. S. P. and Metcalfe, A. V. *Introductory Time Series with R*. Springer, 2009.

Lima, R. S. P. Séries temporais no **R**. *mimeo*, 2014.

Lundholm, M. *Introduction to R´s time series facilities*. 2011.

Matloff, N. *The Art of R Programming*. 2009.

Murrel, P. *R Graphics*. Chapman e Hall/CRC, 2006.

Pfaff, B. *Analysis of Integrated and Cointegrated Time Series with R*. Springer, 2008.

Verzani, J. *Using R for Introductory Statistics*. CRC Press, 2014.

Wickham, H. *ggplot2 Elegant Graphics for Data Analysis*. Springer, 2009.

Wickham, H. *Advanced R*. CRC Press, 2014.

Wilkinson, L. *The Grammar of of Graphics*. Springer, 2005.