Introdução à Inferência Bayesiana em R usando JAGS

André Silva de Queiroz*

Brasília, 25 de novembro de 2015

Sumário

1 Primeiros Passos **3**

1.1 Instalação 3

1.2 Interface do JAGS com o R 6

2 Utilizando o R **7**

2.1 Conceitos básicos 7

2.2 Gráficos 10

2.3 Lendo seus próprios dados 15

3 Utilizando o JAGS via R **20**

3.1 Definição do modelo 20

3.2 Definição do modelo Bayesiano 22

3.3 Um exemplo aplicado 25

*Mestrando em Estatística pela Universidade de Brasília sob a orientação da Prof.ª Cibele Queiroz da Silva, Ph.D. E-mail para contato: andrequeiroz@live.com.

Introdução

A utilização da estatística Bayesiana como ferramenta prática de análise de dados foi alavancada pelo desenvolvimento computacional nas últimas décadas do século XX. Hoje, existem diversos programas computacionais que implementam os algoritmos de Monte Carlo em cadeia de Markov (MCMC, do inglês *Markov chain Monte Carlo*) que permitem o uso da inferência Bayesiana nos mais diversos tipos de problemas. Os principais são o pacote `MCMCpack`, programado em R, o módulo PyMC em Python, além dos programas independentes JAGS, Stan e WinBUGS.

Nesse documento, não será apresentada a teoria que envolve a análise Bayesiana de dados, tão pouco o funcionamento do MCMC em si. A ideia aqui é mostrar passo a passo como utilizar uma ferramenta bastante flexível de análise Bayesiana, no caso o JAGS (Plummer, 2003), desde a sua instalação até a geração e análise dos resultados. Contudo, para alcançar esse objetivo, será apresentado também um breve guia de como utilizar o R (Ihaka and Gentleman, 1996) em tarefas comuns. Para detalhes sobre a metodologia recomendo inicialmente Sivia and Skilling (2006) e Kruschke (2014); e no nível intermediário Gelman et al. (2014).

1 Primeiros Passos

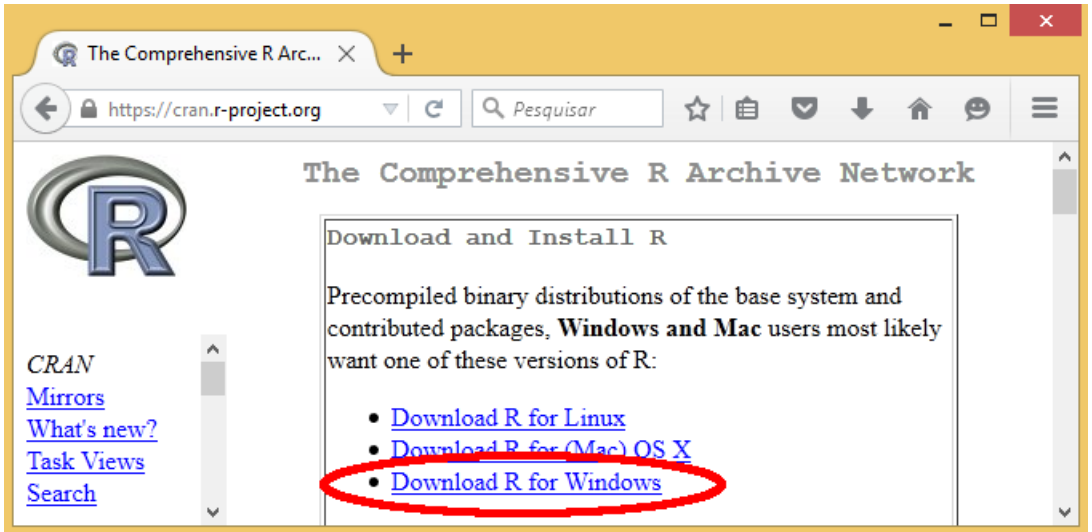
O JAGS é uma implementação multiplataforma de algoritmos de simulação para a análise Bayesiana de modelos hierárquicos utilizando MCMC. JAGS é um acrônimo em inglês para ***J**ust **A**nother **G**ibbs **S**ampler*. A sua escolha foi baseada em dois principais motivos. O primeiro é devido ao seu estágio contínuo de desenvolvimento. Isso faz do JAGS um *software* em constante aprimoramento, seja na melhora da performance dos algoritmos ou na correção de falhas de implementação. O segundo motivo de sua escolha é graças à ótima interação com o R, através do pacote `R2jags` (Su and Yajima, 2015), que será demonstrada ao longo desse documento. Um outro motivo, não menos importante, é em relação ao direito de uso irrestrito sem a necessidade de aquisição de licença. O JAGS, como o R, é licenciado pela *GNU General Public License*, sendo desse modo um *software* livre.

1.1 Instalação

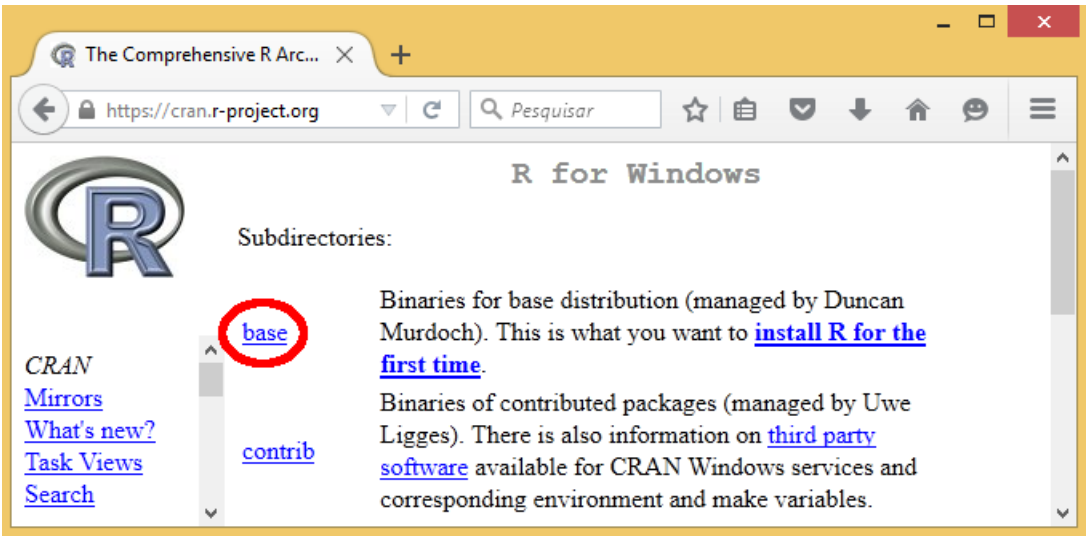
O processo de instalação envolve 2 etapas: a instalação do R e a instalação do JAGS, evidentemente. Os processos são indepententes, ou seja, instalar o JAGS antes ou depois do R não faz diferença alguma. Ambos os programas são multiplataforma, isso quer dizer que existem versões para Windows, Mac OS X e Linux. Todavia, a instalação será demonstrada apenas para os usuários em ambiente Windows, tendo em vista que essa é a plataforma mais recorrente. Usuários em outros sistemas não devem ter grandes dificuldades, já que trata-se de um processo bem simples.

1.1.1 R

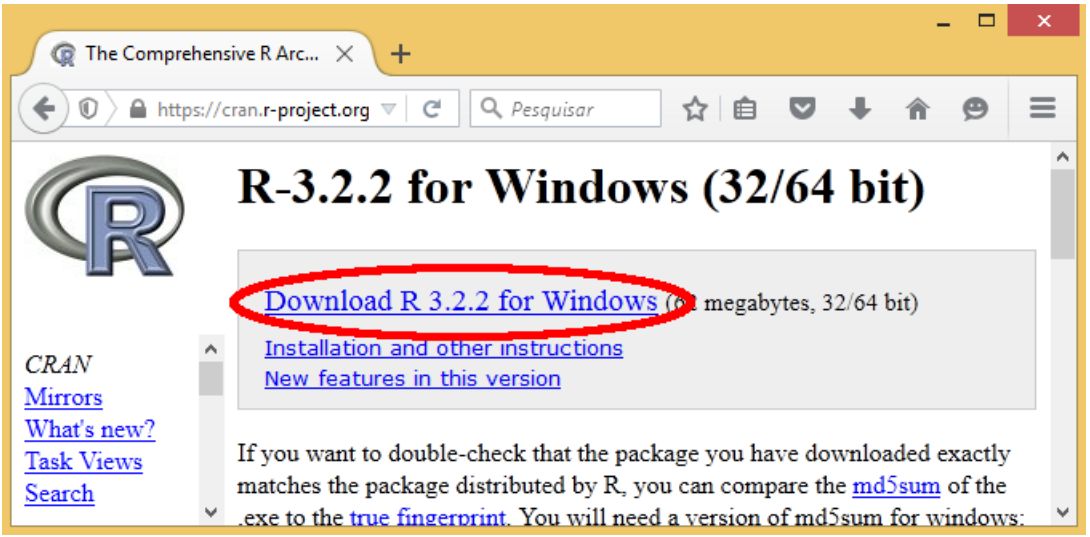
- Entre no *site* https://cran.r-project.org/ e clique em *Download R for Windows*, como mostra a figura.



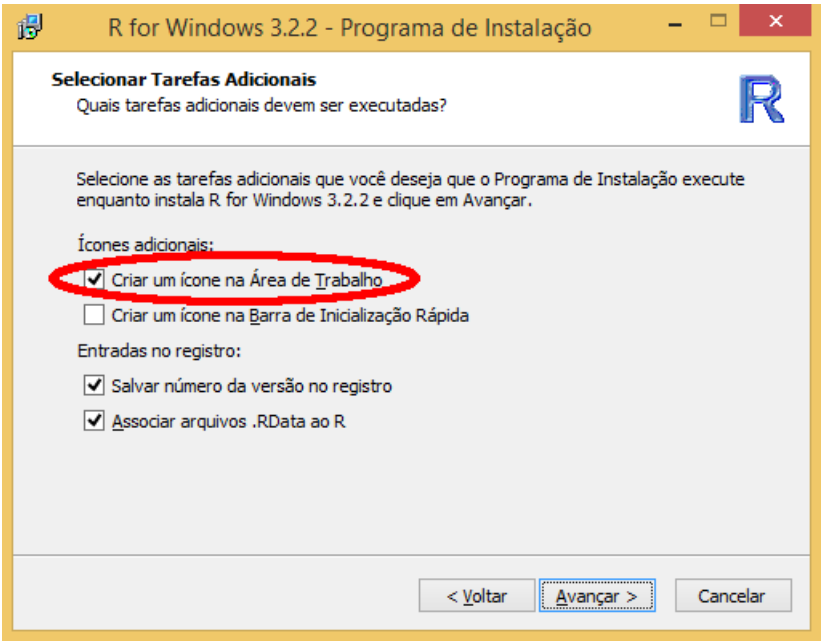
- Na página seguinte clique em *base*.



- Então clique em *Download R 3.2.2 for Windows* para iniciar o *download*.



- Após o término do *download* execute o arquivo baixado e instale o programa. As configurações padrão de instalação atendem perfeitamente o uso típico do R. Então, siga clicando em "próximo". Apenas certifique-se de que seja adicionado um ícone do programa à área de trabalho, como mostra a próxima figura.



- Pronto! O R foi instalado com sucesso. Atualmente ele está na versão 3.2.2 (*Fire Safety*).

1.1.2 JAGS

- Entre no *site* http://sourceforge.net/projects/mcmc-jags/files/ e baixe o arquivo indicado pelo caminho na figura.

- Quando o *download* chegar ao fim, execute o arquivo baixado e instale o programa. Nessa etapa também não há complicações. Clique em "*next*" até o fim e aguarde a finalização da instalação.

- Pronto! O JAGS está instalado, sua versão mais atual é a recente 4.0.0.

1.2 Interface do JAGS com o R

O JAGS se comunica com R através da biblioteca[1] `R2jags`. Porém ela não vem instalada na versão básica do programa.

A instalação de bibliotecas no R é um processo fácil, e que pode ser feito pelo menos de duas maneiras distintas. O modo mais geral e preferível de se instalar pacotes é abrir o programa e digitar no console:

```
install.packages("R2jags")
```

Esse comando irá instalar a biblioteca selecionada e todas as suas dependências. Eventualmente, abrirá uma janela para a seleção do repositório de onde os arquivos serão baixados. Escolha um repositório qualquer (levando em consideração que são desejáveis aqueles fisicamente mais pertos).

A outra opção é atráves do método "*point & click*". Na versão recém instalada, por exemplo, basta ir ao menu "Pacotes" e depois clique em "Instalar pacote(s)...". Se for solicitado, selecione novamente o repositório. Então surgirá uma lista com todas as bibliotecas disponíveis. Escolha o pacote `R2jags` e clique "Ok".

Independente do método, instalar um pacote que porventura já esteja presente no sistema implicará apenas na reinstalação de sua versão mais recente. Para manter **todas as bibliotecas instaladas** atualizadas, mantenha o hábito de abrir o programa e digitar diretamente no console:

```
update.packages(ask = FALSE)
```

Esse comando verifica a situação de todos os pacotes disponíveis no sistema e então baixa e instala automaticamente as atualizações necessárias. O parâmetro `ask = FALSE` é opcional, ele indica que sempre que uma versão mais nova de uma biblioteca estiver disponível, ela será instalada **sem** a necessidade de confirmação do usuário.

[1] De uma forma simplificada, em R, uma biblioteca, ou pacote, é um conjunto de códigos disponibilizados de uma maneira padronizada para facilitar o compartilhamento de programas.

2 Utilizando o R

O R é uma linguagem de programação desenvolvida inicialmente para fins estatísticos. Porém, hoje ela apresenta uma infinidade de possibilidades que a permitem ser classificada como uma linguagem de multiuso. Naturalmente, o seu foco principal permanece na implementação de técnicas estatísticas para diversos finalidades.

O R é uma linguagem interpretada, pois ao entrar na plataforma de programação, o usuário inicia um processo interativo onde são digitados os comandos linha a linha e a resposta é obtida de imediato. Por isso mesmo, é totalmente recomendável que o usuário não digite os comandos diretamente no console, mas adote a prática de programar a partir de *scripts*, que são arquivos de texto que salvam os códigos em R e facilitam o desenvolvimento e reprodução do programa.

Entre no R a partir do ícone[1] na área de trabalho do computador. A sessão do R foi iniciada num programa chamado RGui, que não é o R propriamente dito. O RGui é um ambiente de desenvolvimento bem simples. Ele oferece alguns atalhos e uma ferramenta de editor de texto. Existem outros ambientes de desenvolvimento. Um, por exemplo, que tem ganhado bastante fama nos últimos anos é o Rstudio, que pode ser facilmente baixado e instalado. Na minha opinião, a melhor ferramenta de desenvolvimento é chamada Emacs.

Dentro do programa, abra um novo *script* (na verdade, torne isso também um hábito, sempre programe através de *scripts!*[2]). Isso pode ser feito indo ao menu "Arquivo" e clicando em "Novo script". Agora você está pronto para começar.

2.1 Conceitos básicos

Tudo o que é manipulado em R pode ser salvo em objetos. Seja:

$$X \sim N(0, 1). \tag{1}$$

Como gerar uma amostra de 10 observações dessa distribuição em R? No *script* recém aberto digite:

```
x <- rnorm(n = 10, mean = 0, sd = 1)
```

Bem, o R ainda não sabe que foi digitado esse comando. O código apenas faz parte de

[1] Pode ser que tenham sido criados dois ícones. Isso acontece se o sistema for de 64 bits. Um ícone se refere à versão 32 bits e outra a de 64 bits. Escolha a que achar conveniente.

[2] Programar a partir de *scripts* é altamente recomendável, porém também é importante manter um padrão de escrita da linguagem de programação para facilitar a utilização posterior do arquivo por você e outros usuários. A equipe de desenvolvedores do Google criou uma estilização padronizada do R que pode ser acessada em https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml. São poucas regras que ajudam bastante a manter o código limpo, bonito e organizado.

um arquivo de texto. Para enviar esse comando à sessão do R, selecione a linha e tecle **Ctrl + r**. O comando então será reproduzido no console como foi digitado. Agora o R gerou uma amostra de 10 observações da distribuição de X e salvou num vetor `x`. Apesar de ser solicitado que a linha fosse **selecionada** no *script*, isso não é bem necessário. Basta que o cursor do editor de texto esteja posicionado em **qualquer lugar** dessa linha. Para conferir, volte o cursor para a linha digitada no *script* e tecle **Ctrl + r**.

O comando digitado, `rnorm()`, é uma função, assim como todos os objetos seguidos de parêntesis no R. Ela gera amostras aleatórias da distribuição normal segundo três parâmetros: `n` (o tamanho da amostra), `mean` (a média) e `sd` (o desvio padrão). Apesar de explícitos no exemplo, o nome dos parâmetros podiam ser omitidos. Assim, o comando

```
x <- rnorm(10, 0, 1)
```

é equivalente ao anterior.

E quais foram os valores gerados? Digite e execute[1]:

```
print(x)
```

```
 [1] -0,7251606 -0,3627288  0,7731297  0,9596509  0,2754099  0,4480328
 [7] -0,6028297  0,5814426 -0,3481204  0,5520237
```

ou apenas:

```
x
```

```
 [1] -0,7251606 -0,3627288  0,7731297  0,9596509  0,2754099  0,4480328
 [7] -0,6028297  0,5814426 -0,3481204  0,5520237
```

Quando um objeto, no caso `x`, é passado isoladamente para a linha de comando, o R invoca implicitamente a função `print()`, que é imprimir na tela do computador.

Digite e execute:

```
mean(x)  # média
```

```
[1] 0,155085
```

```
sd(x)  # desvio padrão
```

```
[1] 0,6091485
```

As funções `mean()` e `sd()` retornam a média e o desvio padrão, respectivamente, de um conjunto de dados. Uma coisa que elas têm em comum é que ambas são aplicadas ao vetor `x` e retornam um único valor, a estatística calculada.

Além dessas funções, nesse bloco de comando foi apresentado algo novo. O símbolo # em R é um indicador de comentários. Qualquer coisa escrita após o # até o fim da linha será

[1] Para evitar a redundância ao longo do documento, desse ponto em diante, digite e execute significará digite no *script* e tecle **Ctrl + r** com o cursor sob a linha digitada.

ignorada pelo interpretador do R. Use-o sempre, com bom senso, para comentar o *script* com lembretes, dicas, ou qualquer outra coisa que facilite o entendimento e/ou a leitura futura do arquivo.

Algumas funções são aplicadas elemento a elemento num vetor, executando uma transformação nos dados. Digite e execute:

```
abs(x)  # calcula o valor absoluto dos elementos
```

```
 [1] 0,7251606 0,3627288 0,7731297 0,9596509 0,2754099 0,4480328 0,6028297
 [8] 0,5814426 0,3481204 0,5520237
```

```
sqrt(abs(x))  # calcula a raiz quadrada do valor absoluto dos elementos
```

```
 [1] 0,8515636 0,6022697 0,8792779 0,9796177 0,5247951 0,6693526 0,7764211
 [8] 0,7625238 0,5900173 0,7429830
```

O resultado dessas funções inclusive podem alimentar outra função, como no segundo exemplo onde inicialmente é aplicada a função "valor absoluto" e então é calculada a raiz quadrada desses valores. Outros exemplos de funções matemáticas úteis implementadas em R são:

- $f(x) = e^x \rightarrow$ `exp()`;
- $f(x) = x! \rightarrow$ `factorial()`;
- $f(x) = \ln x \rightarrow$ `log()`.

Operações como somar, subtrair, multiplicar, dividir ou elevar os valores a uma potência podem ser feitas diretamente através dos operadores `+`, `-`, `*`, `/` e `**`, respectivamente. Por exemplo, digite e execute:

```
4 + x  # adiciona 4 a x
```

```
 [1] 3,274839 3,637271 4,773130 4,959651 4,275410 4,448033 3,397170
 [8] 4,581443 3,651880 4,552024
```

```
2 * x  # multiplica x por 2
```

```
 [1] -1,4503212 -0,7254576  1,5462593  1,9193017  0,5508199  0,8960657
 [7] -1,2056595  1,1628851 -0,6962407  1,1040475
```

```
x / 7  # divide x por 7
```

```
 [1] -0,10359437 -0,05181840  0,11044709  0,13709298  0,03934428
 [6]  0,06400469 -0,08611853  0,08306322 -0,04973148  0,07886053
```

```
x ** 3  # eleva x ao cubo
```

```
 [1] -0,38133143 -0,04772502  0,46212239  0,88377107  0,02089002
 [6]  0,08993517 -0,21907056  0,19657147 -0,04218794  0,16821830
```

Seja agora uma amostra de 1.000 elementos de X.

```
x <- rnorm(1000)
```

Os valores da média e da variância foram omitidos, pois `mean` = 0 e `sd` = 1 são os valores padrão da função `rnorm()`. Digite e execute:

```
summary(x)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2,90670 -0,67718  0,05692  0,04496  0,72453  3,24329
```

A função `summary()` na verdade é um **método** em R. Simplificando, ela tem um comportamento diferente dependendo do tipo de objeto ao qual ela é aplicada. Como `x` é um vetor de dados, ela mostra as principais estatísticas desses dados, como média, mediana, mínimo e máximo. Ela aparecerá em outros contextos com um comportamento distinto.

2.2 Gráficos

Um dos pontos fortes do R é a incrível versatilidade em relação a representação visual dos dados. Existem muitas bibliotecas que produzem uma enorme variedade de gráficos. Nesse documento serão apresentados algumas maneiras de produzir gráficos com a biblioteca padrão do R para essa finalidade, `graphics`. Esse pacote já vem na instalação básica do programa e é carregado automaticamente sempre que o R inicia uma sessão.

O R possui internamente algumas bases de dados para fins didáticos. Uma delas é a base de dados `cars`. Ela é carregada num objeto do tipo `data.frame`. Digite e execute:

```
head(cars)
```

```
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10
```

A variável `speed` é a velocidade do carro em milhas por hora e a variável `dist` é a distância em pés percorrida até a frenagem.

A função `head()` retorna, por padrão, os primeiros 6 elementos de um objeto como um vetor ou `data.frame`. Essa quantidade de valores retornados pode ser alterada acrescentando-se à função o parâmetro `n = c`, onde `c` é igual ao número de elementos desejados. Experimente retornar as 20 primeiras observações da base `cars`, por exemplo.

E se for necessário retornar os últimos elementos ao invés dos primeiros? A função `tail()` é a solução e sua utilização é idêntica a da função `head()` apresentada.

As funções `head()` e `tail()` são muito úteis, pois é comum existirem bases de dados com milhares de observações. Assim, imprimir a base toda no console do R irá somente gerar poluição visual e dificilmente agregará alguma informação relevante.

Agora, digite e execute:

```
summary(cars)
```

```
     speed           dist
 Min.   : 4,0   Min.   :  2,00
 1st Qu.:12,0   1st Qu.: 26,00
 Median :15,0   Median : 36,00
 Mean   :15,4   Mean   : 42,98
 3rd Qu.:19,0   3rd Qu.: 56,00
 Max.   :25,0   Max.   :120,00
```

O resultado da função `summary()` mostra que a velocidade média dos carros é 15,4 mph e a distância média de frenagem é 42,98 ft. A primeira coisa a ser feita é levar esses dados para uma escala mais popular no Brasil. Antes "clone" esse banco de dados para um outro objeto no R. Digite e execute:

```
# esse comando clona a base dados cars num novo objeto chamado carros
carros <- cars
```

Para converter a velocidade de milhas por hora em kilometros por hora é necessário multiplicar o valor original por 1,6093. Já a conversão da distância em pés para metros é possível multiplicando o valor original por 0,3048. No R, digite e execute:

```
carros <- transform(carros, speed = 1.6093 * speed, dist = 0.3048 * dist)
```

A função `transform()`, como o nome sugere, transforma as variáveis dentro de um banco de dados. Nesse caso, não há problemas em sobrescrever os dados originais, porém nem sempre isso é uma boa prática. Para salvar os dados transformados numa nova variável dentro desse mesmo `data.frame` defina a variável de destino com um novo nome que não exista na base de dados. Por exemplo, para calcular também a distancia de frenagem em kilometros, digite e execute:

```
carros <- transform(carros, dist.km = dist / 1000)
```

Nesse caso, a variável `dist.km` recebeu os valores da variável `dist` divididos por 1000, transformando assim metros em kilometros.

A função `summary()` traz as estatísticas para os dados transformados.

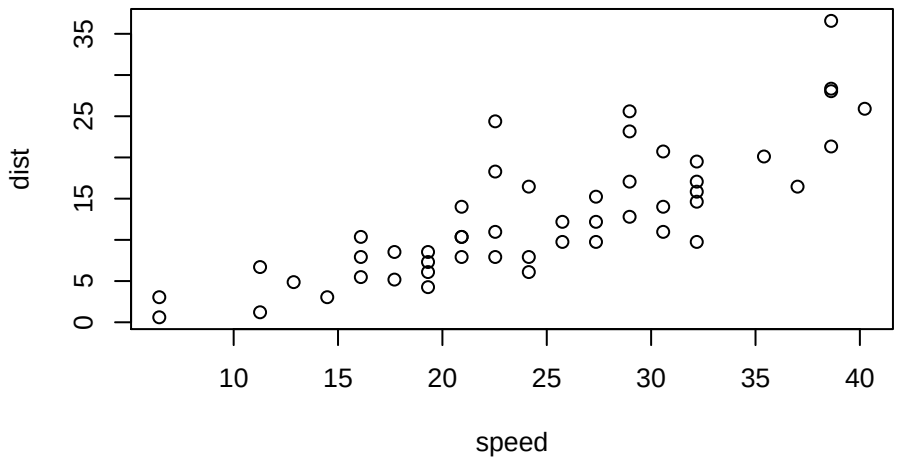
```
summary(carros)
```

```
     speed             dist            dist.km
 Min.   : 6,437   Min.   : 0,6096   Min.   :0,0006096
 1st Qu.:19,312   1st Qu.: 7,9248   1st Qu.:0,0079248
 Median :24,140   Median :10,9728   Median :0,0109728
 Mean   :24,783   Mean   :13,1003   Mean   :0,0131003
 3rd Qu.:30,577   3rd Qu.:17,0688   3rd Qu.:0,0170688
 Max.   :40,233   Max.   :36,5760   Max.   :0,0365760
```

A velocidade média dos carros é de 24,8 km/h e a distância de frenagem média é de 13,1 m.

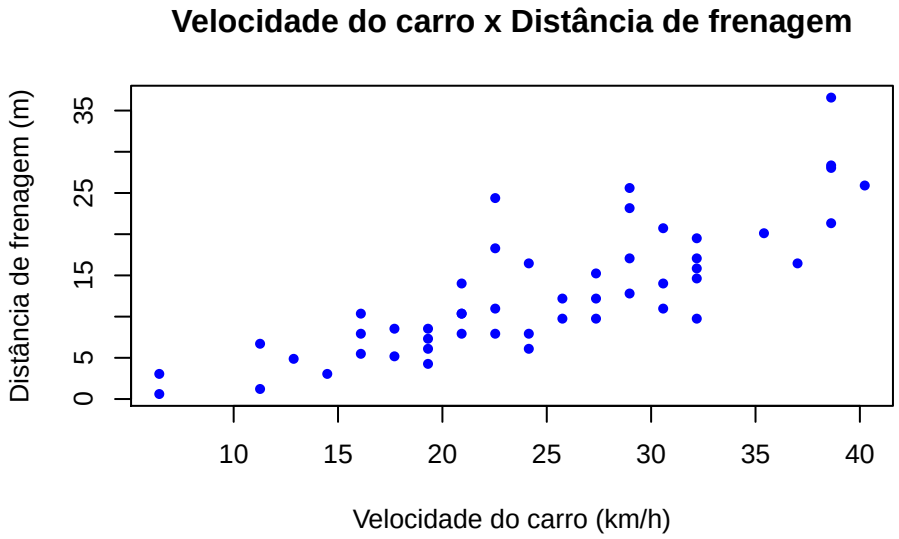
Lembre-se que o tema principal são os gráficos. Uma visualização que naturalmente vem à mente é o gráfico de dispersão. Digite e execute:

```
plot(dist ~ speed, data = carros)
```



Essa é a execução e o resultado mais simples da função `plot()`, que também, na verdade, é um método em R. Ao lado esquerdo de $\sim$ está a variável a ser representada no eixo y, e ao lado direito está a variável do eixo x. Existem muitas maneiras de personalizar esse gráfico. Isso é feito a partir de parâmetros opcionais. Digite e execute:

```
plot(dist ~ speed, data = carros, pch = 20, col = "blue",
     xlab = "Velocidade do carro (km/h)",
     ylab = "Distância de frenagem (m)",
     main = "Velocidade do carro x Distância de frenagem")
```



O parâmetro `pch` recebe um número inteiro e representa o símbolo a ser impresso no gráfico. O número 1 (padrão) são aquelas bolinhas vazias vistas no primeiro gráfico. O número 4 são marcas em formato de x. O número 20, na figura, são bolinhas preenchidas. Teste qualquer número inteiro (até o 127, pelo menos). Cada valor produz um símbolo diferente.

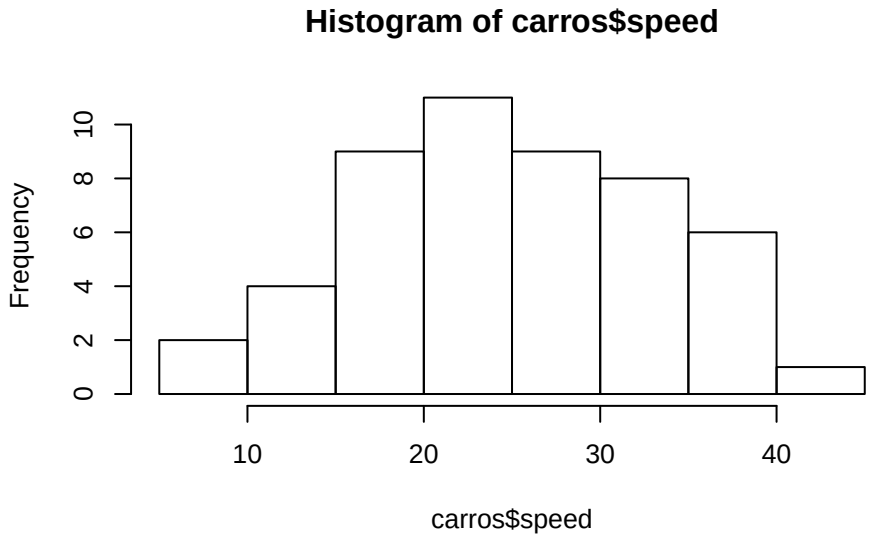
O parâmetro `col`, como aparenta, é a cor do símbolo. São aceitos os nomes das cores em inglês, números inteiros, e se for necessária uma cor bem particular, ela pode ser definida a partir do seu código hexadecimal. Por exemplo, a cor *Sienna* é definida pelo seu código A0522D, para representar essa cor no gráfico, troque o *blue* por *#A0522D*.

Os parâmetros `xlab`, `ylab` e `main` definem os títulos do eixo x, do eixo y e o título principal, respectivamente.

Para uma lista extensa de possíveis personalizações, digite no terminal do R `?par`. Qualquer função do R digitada no terminal precedida pelo ponto de interrogação abre a documentação completa sobre aquela função. Ler essa documentação é a melhor maneira para se aprender recursos novos e diferentes sobre qualquer função do R.

Um gráfico bastante tradicional é o histograma. A biblioteca `graphics` possui uma função dedicada para esse tipo de representação. Digite e execute:

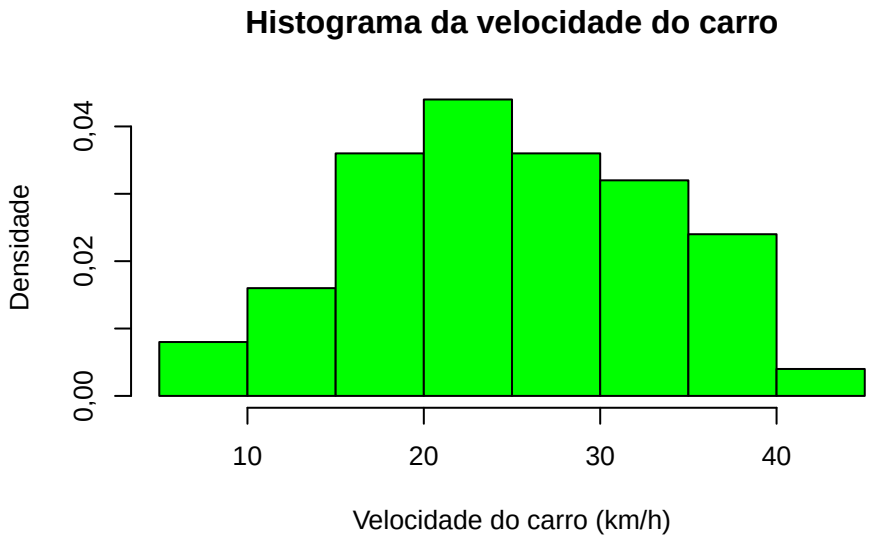
```
hist(carros$speed)
```



O único parâmetro necessário da função `hist()` é a própria série de dados. Os valores da variável `speed`, ou qualquer variável num `data.frame`, podem ser acessados através do operador $, como no comando anterior.

Os parâmetros opcionais que personalizam o gráfico de dispersão funcionam de forma semelhante no histograma. Dessa vez, digite e execute:

```
hist(carros$speed, freq = FALSE, col = "green",
     xlab = "Velocidade do carro (km/h)", ylab = "Densidade",
     main = "Histograma da velocidade do carro")
```

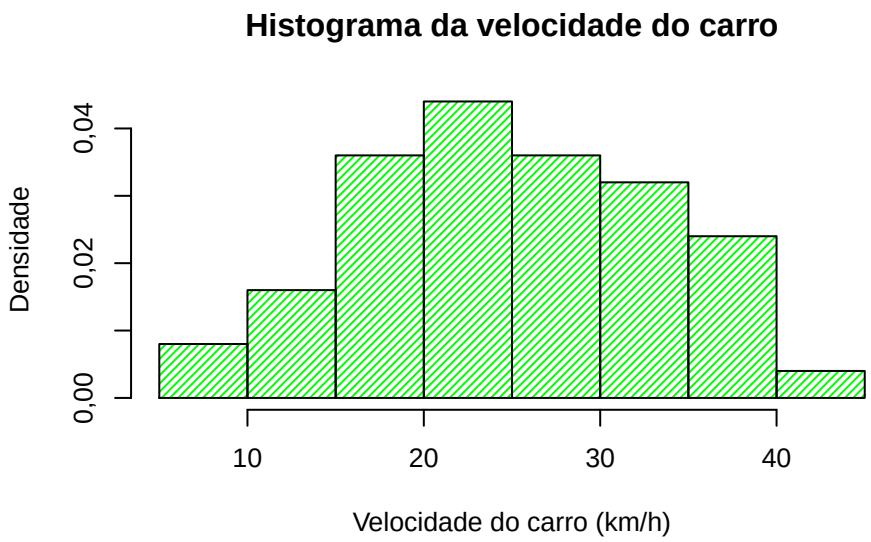


A mudança mais importante, porém pouco perceptiva à primeira vista, é na escala do gráfico. Por padrão, os histogramas são construídos com base nas frequências **absolutas** das observações. Então o parâmetro `freq = FALSE` indica que o gráfico deve ser construído a partir das frequências **relativas** dos dados.

Os parâmetros `col`, `xlab`, `ylab` e `main` têm a mesma função descrita anteriormente na contrução do gráfico de dispersão.

Um efeito interessante e que deixa o aspecto do gráfico mais leve é obtido por meio do parâmetro `density`. Digite e execute:

```
hist(carros$speed, freq = FALSE, col = "green", density = 33,
     border = "black", xlab = "Velocidade do carro (km/h)",
     ylab = "Densidade", main = "Histograma da velocidade do carro")
```



O parâmetro `density` cria um padrão de preenchimento no gráfico de modo que o valor igual a cem cria uma figura totalmente colorida e zero uma figura sem cores. No exemplo, o valor igual 33 equivale a um padrão de cores correspondente a aproximadamente um terço preenchido. Há também um novo parâmetro, `border`. Ele determina a cor da borda do histograma.

O pacote `graphics` permite fazer gráficos de barras, linhas, polígonos e curvas, apenas citando alguns exemplos. Entretanto para ter uma ideia do verdadeiro poder do R nesse quesito recomendo Wickham (2009). A biblioteca `ggplot2`, apresentada na referência anterior, possui uma curva de aprendizado maior, mas os resultados produzidos são extremamente recompensadores.

2.3 Lendo seus próprios dados

O R é estigmatizado por muitos como um *software* que não é bom quando o assunto são grandes bases de dados. Isso não é verdade. A razão principal é que o R nem se propõe a ser um gerenciador de banco de dados. Para essa função já existem programas exclusivos com essa finalidade e que fazem o trabalho com bastante competência, como é o caso dos sistemas de gerenciamento de bases de dados relacionais (MySQL, PostgreSQL, SQLite, e etc). Apesar

da utilização desses sistemas em conjunto com o R fugirem ao escopo desse documento, o ponto fundamental aqui é destacar que o fluxo de trabalho com o R consiste em ler os dados de uma fonte externa e então prosseguir com as análises estatísticas.

Uma das formas mais comuns de armazenamento de dados para leitura em R são arquivos de texto, e também essa é uma das maneiras mais práticas de se disponibilizar dados publicamente.

A Administração de Seguridade Social dos Estados Unidos disponibiliza a contagem de nomes de bebês registrados anualmente em seu território. O arquivo compactado encontra-se em https://www.ssa.gov/oact/babynames/names.zip. Ele possui diversos documentos de texto nomeados "yobAAAA.txt" onde AAAA é o ano de registro. Cada um deles possui três colunas (separadas por vígulas): o nome de registro, o sexo do bebê e a quantidade de registros. Ele pode ser baixado e os documentos extraídos numa pasta qualquer, porém os próximos passos darão ênfase como tudo pode ser feito no R, evidênciando sua característica de linguagem multiuso. Digite e execute:

```
temp <- tempfile()
```

A função `tempfile()` retorna um nome que pode ser utilizado como um arquivo temporário pelo sistema, no caso, o arquivo compactado com as bases de dados anuais. Agora, digite e execute:

```
download.file("https://www.ssa.gov/oact/babynames/names.zip", temp)
```

O arquivo `names.zip` é baixado através da função `download.file()`[1] que o salva no arquivo temporário, `temp`, criado no passo anterior. Para ler os dados de 1980, digite e execute:

```
conexao <- unz(temp, "yob1980.txt")
registros <- cbind(1980, read.table(conexao, sep = ","))
```

A função `unz()` cria uma conexão virtual com o arquivo `yob1980.txt`. Então, a função `read.table()` lê os dados através dessa conexão e salva numa base de dados denominada `registros`. A função `cbind()` adiciona uma coluna com a constante 1980, que é o ano da base.

Se o arquivo `names.zip` foi baixado diretamente pelo navegador de Internet e depois descompactado para alguma pasta qualquer no computador, então digite e execute:

```
registros <- cbind(1980, read.table(file.choose(), sep = ","))
```

Após a execução do comando acima, uma janela (resultante da chamada da função `file.choose()`) será aberta e o aquivo `yob1980.txt` deve ser selecionado.

[1] Essa função requer acesso à Internet e o tempo de execução varia conforme a velocidade de conexão.

Como o arquivo original não tem a informação do nome das colunas, o R criou nomes automáticos para as variáveis no `data.frame`. Esses nomes devem ser editados manualmente para facilitar o trabalho com a base. Digite e execute:

```
names(registros) <- c("ano", "nome", "sexo", "contagem")
```

Para saber se deu tudo certo, digite e execute:

```
head(registros)
```

```
   ano     nome sexo contagem
1 1980 Jennifer    F    58385
2 1980   Amanda    F    35820
3 1980  Jessica    F    33920
4 1980  Melissa    F    31631
5 1980    Sarah    F    25741
6 1980  Heather    F    19971
```

Como pode ser observado, Jennifer é o nome feminino com o maior número de registros em 1980 nos Estados Unidos[1]. E em relação aos meninos? Primeiro, digite e execute:

```
table(registros$sexo)
```

```

    F     M
12157  7282
```

A função `table()` tabula os dados e mostra que há duas categorias da variável `sexo`: F - *female* e M - *male* (do inglês, feminino e masculino, respectivamente). Então, para obter a base apenas com os registros masculinos, digite e execute:

```
meninos <- subset(registros, sexo == "M")
```

A função `subset()` claramente retorna apenas os registros com sexo igual a `M` da base original. O operador == corresponde ao igual comparativo no R. Os primeiros 10 registros dessa nova base são:

```
head(meninos, n = 10)
```

```
       ano        nome sexo contagem
12158 1980     Michael    M    68666
12159 1980 Christopher    M    49086
12160 1980       Jason    M    48176
12161 1980       David    M    41913
12162 1980       James    M    39312
12163 1980     Matthew    M    37857
12164 1980      Joshua    M    36058
12165 1980        John    M    35263
12166 1980      Robert    M    34273
12167 1980      Joseph    M    30189
```

[1] É possível afirmar isso, pois a base já vem em ordem decrescente.

O número à esquerda se refere à linha de registro na base original. Mesmo depois da divisão da base em categorias, essa informação inicial permanece inalterada.

A etapa seguinte consiste em ler o conjunto de documentos de texto de 1981 a 2014 de forma sistemática. O R é uma linguagem de programação, e portanto possui ferramentas que permitem a execução de comandos de forma estruturada. Uma dessas ferramentas é o laço de repetição (mais popularmente conhecido como *loop*). Para ler os registros usando um laço, digite e execute:

```
for (i in 1981:2014) {
    conexao <- unz(temp, paste("yob", i, ".txt", sep = ""))
    base <- setNames(cbind(i, read.table(conexao, sep = ",")),
                     names(registros))
    registros <- rbind(registros, base)
}
unlink(temp)
```

O laço é iniciado em `i = 1981` e segue até `i = 2014`. Em cada iteração, a conexão com a base do respectivo ano é criada por meio da função `unz()`, então os dados são lidos numa base temporária (chamada aqui de `base`). Como os documentos não possuem a informação do ano de referência, esse dado é inserido novamente pela função `cbind()` em cada rodada. A função `setNames()` salva na base temporária os nomes das variáveis provenientes da base principal (`registros`). Em seguida, a base temporária é adicionada à base principal através da função `rbind()`. Por fim, a função `unlink()` libera o arquivo temporário da memória do computador.

Se o arquivo foi baixado diretamente do *link* fornecido e o seu conteúdo extraído para uma pasta no computador, a leitura dos dados pode ser feita por meio dos comandos:

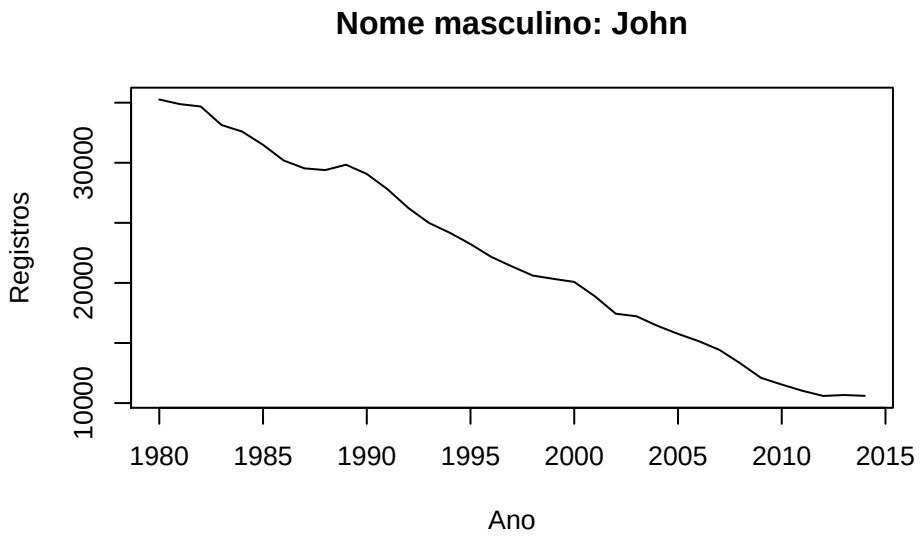
```
setwd(choose.dir())
for (i in 1981:2014) {
    arquivo <- paste("yob", i, ".txt", sep = "")
    base <- setNames(cbind(i, read.table(arquivo, sep = ",")),
                     names(registros))
    registros <- rbind(registros, base)
}
```

A função `setwd()` determina a pasta de trabalho do R, e a função `choose.dir()` abre uma janela onde deve ser selecionada a pasta onde os arquivos de texto foram descompactados.

Feito isso, a base de dados foi carregada na memória RAM do sistema e está pronta para ser analisada.

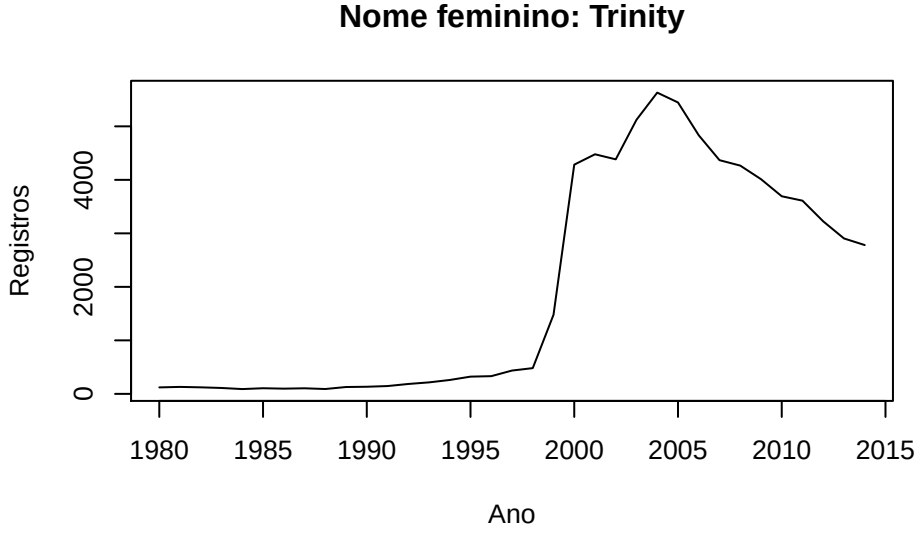
Para saber se tudo funcionou como o esperado, digite e execute:

```
nome <- subset(registros, sexo == "M" & nome == "John")
plot(contagem ~ ano, data = nome, type = "l", xlab = "Ano",
     ylab = "Registros", main = "Nome masculino: John")
```



Pelo visto, o nome masculino "John" tem perdido popularidade a cada ano que passa nos Estados Unidos. Um último exemplo, mudando apenas os parâmetros da função `subset()`, digite e execute:

```
nome <- subset(registros, sexo == "F" & nome == "Trinity")
plot(contagem ~ ano, data = nome, type = "l", xlab = "Ano",
     ylab = "Registros", main = "Nome feminino: Trinity")
```



3 Utilizando o JAGS via R

A seção anterior introduziu, direta ou indiretamente, alguns conceitos básicos de utilização do R que serão necessários para a execução de análises Bayesianas utilizando a dupla R/JAGS. A familiarização com os comandos em R promovida pelos exemplos até agora, também, serão importantes para facilitar o entendimento do JAGS, pois ambos os programas possuem uma sintaxe bem parecida.

Com a finalidade de facilitar a apresentação do uso do JAGS, o conteúdo dessa seção será introduzido através da implementação de um modelo linear simples. A base de dados `carros`, apresentada na subseção 2.2, será de novo objeto de estudo.

3.1 Definição do modelo

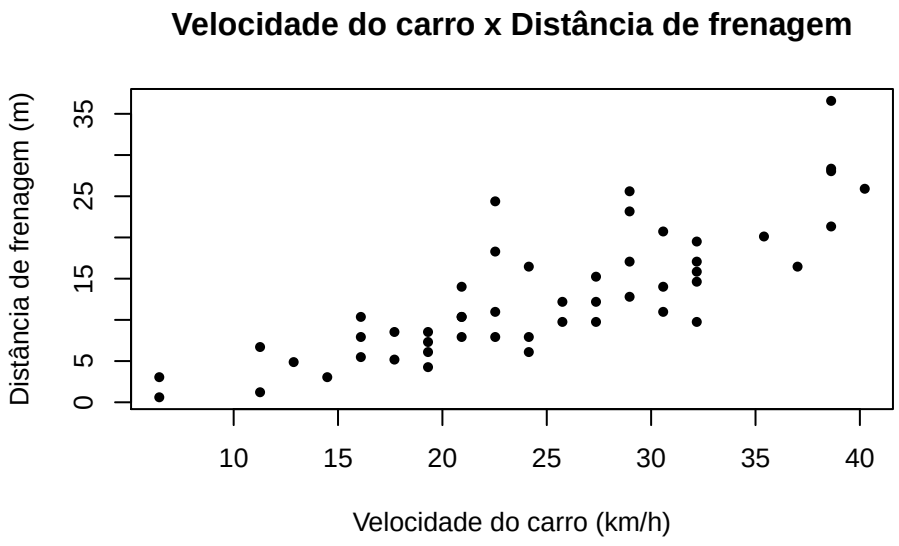
Seja o modelo a ser estudado:

$$y_i = \alpha + \beta x_i + \varepsilon_i, \tag{2}$$

onde y_i é a variável dependente, x_i a variável independente, α e β são parâmetros a serem estimados, ε_i é um erro gaussiano e independente e $i = 1, \ldots, n$ é o índice que caracteriza os n elementos da amostra.

A recapitulação do gráfico de dispersão sugere que talvez não seja uma má ideia ajustar um modelo desse tipo aos dados.

```
plot(dist ~ speed, data = carros, pch = 20,
     xlab = "Velocidade do carro (km/h)",
     ylab = "Distância de frenagem (m)",
     main = "Velocidade do carro x Distância de frenagem")
```



O modelo linear sugerido é estimado de forma bem simples no R através da função `lm()`. Digite e execute:

```
modelo <- lm(dist ~ speed, data = carros)
```

A função `lm()` calculou as estimativas dos parâmetros do modelo:

$$\texttt{dist}_i = \alpha + \beta * \texttt{speed}_i + \varepsilon_i, \tag{3}$$

e salvou todas as informações relevantes no objeto `modelo`. A função `summary()` apresenta de forma sintetizada algumas dessas informações. Digite e execute:

```
summary(modelo)


Call:
lm(formula = dist ~ speed, data = carros)

Residuals:
    Min      1Q  Median      3Q     Max
-8,8603 -2,9033 -0,6925  2,8086 13,1678

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -5,3581     2,0600  -2,601   0,0123 *
speed         0,7448     0,0787   9,464 1,49e-12 ***
---
Signif. codes:  0 '***' 0,001 '**' 0,01 '*' 0,05 '.' 0,1 ' ' 1

Residual standard error: 4,688 on 48 degrees of freedom
Multiple R-squared:  0,6511,Adjusted R-squared:  0,6438
F-statistic: 89,57 on 1 and 48 DF,  p-value: 1,49e-12
```

Os valores estimados dos parâmetros foram: α = -5,3581 e β = 0,7448, ambos significativos ao nível de 5%. Os intervalos de 95% de confiança dos parâmetros podem ser calculados também diretamente. Digite e execute:

```
confint(modelo)

                2,5 %     97,5 %
(Intercept) -9,4999606 -1,2162557
speed        0,5865623  0,9030272
```

3.2 Definição do modelo Bayesiano

O modelo (2) é reescrito de forma Bayesiana como:

$$\begin{aligned} y_i &= \alpha + \beta x_i + \varepsilon_i, \\ \alpha &\sim N(\mu_\alpha, \sigma^2_\alpha), \\ \beta &\sim N(\mu_\beta, \sigma^2_\beta). \end{aligned} \tag{4}$$

onde $\varepsilon_i \sim N(0, \sigma^2)$. Matemáticamente é um modelo bastante similar ao modelo linear simples, porém o paradigma Bayesiano assume que os parâmetros α e β são variáveis aleatórias. Dessa maneira, suas distribuições *a priori* são, nesse caso, duas distribuições normais independentes com hiperparâmetros conhecidos.

Antes de partir para a codificação do modelo, digite e execute:

```
library(R2jags)
```

Se a instalação do JAGS foi feita corretamente, ao chamar a biblioteca com o comando anterior, o R encontrou uma versão compatível do JAGS instalada no sistema que está agora disponível na sessão atual do R.

O modelo escrito na linguagem do JAGS é da seguinte forma:

```
lm.jags <- function() {

    # verossimilhança
    for (i in 1:50) {
        y[i] ~ dnorm(alpha + beta * x[i], 1 / sigma2)
    }

    # prioris
    alpha ~ dnorm(0, 0.01)
    beta ~ dnorm(0, 0.01)
    sigma2 ~ dgamma(2, 0.5)
}
```

Para o R ler o modelo, ele deve ser programado dentro de uma função, por isso o uso da função `function()` na primeira linha.

O primeiro bloco é a parte da verossimilhança. Um laço é escrito varrendo os 50 elementos da amostra através da distribuição condicional $(y_i|\alpha, \beta, x_i) \sim N(\alpha + \beta x_i, \sigma^2)$. Um detalhe importante aqui é que a implementação da distribuição normal em JAGS é feita com o **inverso da variância**, por isso o termo `1 / sigma2` no código.

O segundo e último bloco contém as especificações das distribuições *a priori* dos parâmetros α e β, além da distribuição *a priori* de σ^2. Em todos os casos foram tomadas distribuições

não informativas.

A próxima etapa é definir o conjunto de dados a ser lido pela JAGS. Os valores devem ser transmitidos por meio de uma lista[1] de valores.

```
dados.jags <- list(y = carros$dist, x = carros$speed)
```

A partir desses elementos é possível ajustar o modelo (4). Digite e execute:

```
modelo.jags <- jags(data = dados.jags, model.file = lm.jags,
                    parameters.to.save = c("alpha", "beta"), n.chains = 1,
                    n.iter = 15000, n.burnin = 5000, n.thin = 10)
```

A função `jags()` gera observações das distribuições *a posteriori* de α e β. Para observar os resultados sintetizados, digite e execute:

```
print(modelo.jags)
```

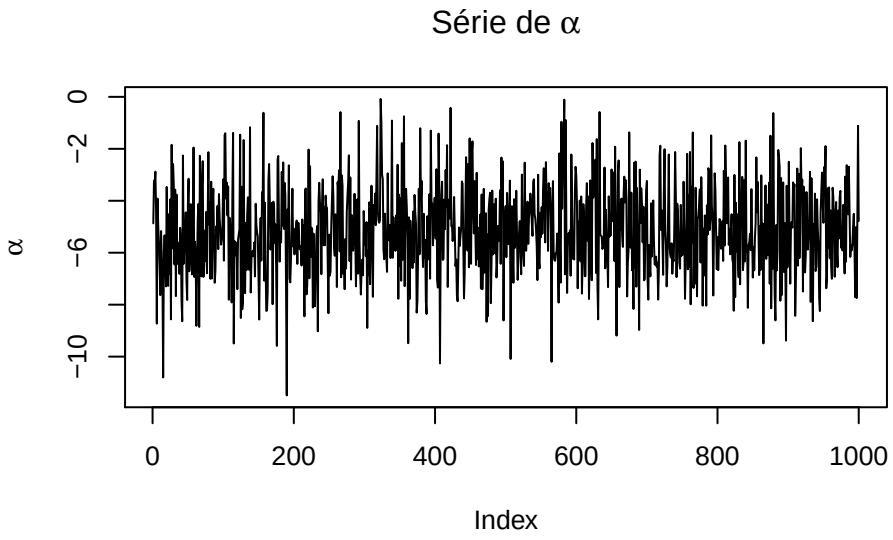
```
Inference for Bugs model at "./jags/lm.jags", fit using jags,
 1 chains, each with 15000 iterations (first 5000 discarded), n.thin = 10
 n.sims = 1000 iterations saved
         mu.vect sd.vect    2,5%     25%     50%     75%   97,5%
alpha     -5,143   1,761  -8,514  -6,322  -5,203  -3,934  -1,481
beta       0,737   0,068   0,602   0,692   0,738   0,784   0,870
deviance 298,020   2,887 294,631 295,950 297,365 299,315 305,386

DIC info (using the rule, pD = var(deviance)/2)
pD = 4,2 and DIC = 302,2
DIC is an estimate of expected predictive error (lower deviance is better).
```

Os valores estimados estão na coluna `mu.vect` e são bem próximos daqueles calculados pela função `lm()`, que ajusta o modelo linear clássico.

[1] Em R, uma lista é um objeto específico criado pela função `list()`.

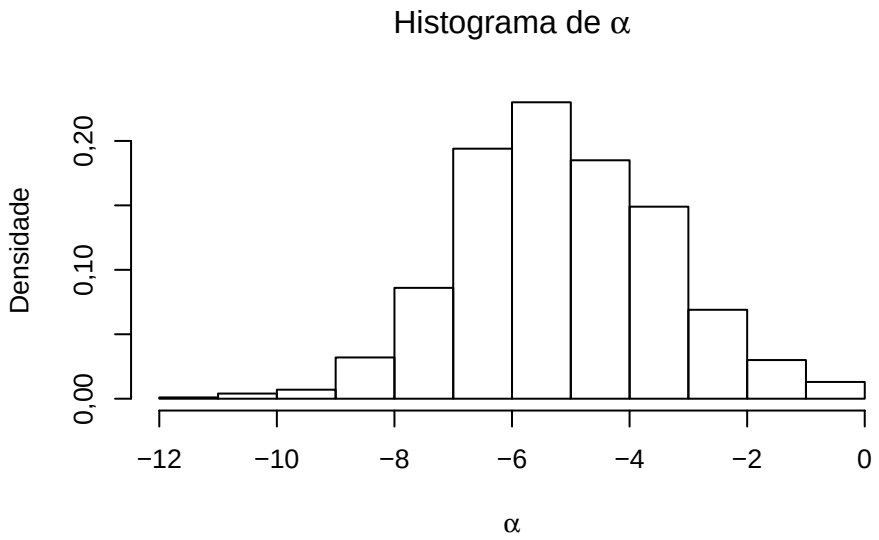
Para investigar a convergência dos parâmetros, digite e execute:

```
plot(modelo.jags$BUGSoutput$sims.list$alpha, type = "l",
     ylab = expression(alpha), main = expression("Série de " * alpha))
```



Outro gráfico interessante é o histograma da distribuição dos valores gerados. Digite e execute:

```
hist(modelo.jags$BUGSoutput$sims.list$alpha, freq = FALSE,
     xlab = expression(alpha), ylab = "Densidade",
     main = expression("Histograma de " * alpha))
```



3.3 Um exemplo aplicado

O seguinte exemplo foi inspirado em Andreon and Weaver (2015).

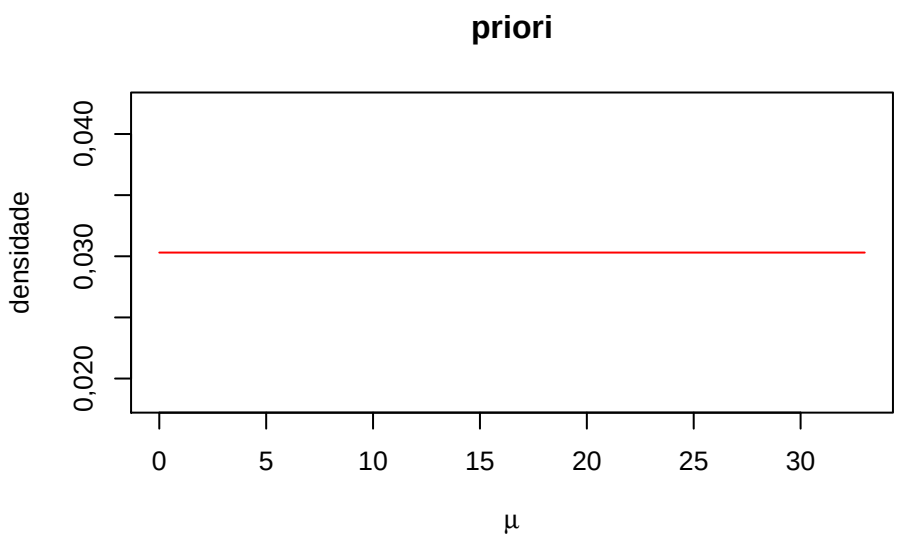
O neutrino é uma partícula elementar sem carga e com a massa bem pequena, portanto é praticamente indetectável com os atuais instrumentos de medição. Um aparelho foi construído para medir a massa, μ, do neutrino com erro instrumental conhecido de $3{,}3\frac{\text{eV}}{\text{c}^2}$[1], que é o desvio padrão esperado dos valores medidos.

Apesar de muito pequena, foi postulada uma distribuição *a priori* uniforme não informativa para a massa do neutrino, tal que:

$$\pi(\mu) = \text{U}(0, 33). \tag{5}$$

No R, essa distribuição pode ser visualizada através da função `curve()`. Digite e execute:

```
curve(dunif(x, min = 0, max = 33), from = 0, to = 33, col = 2,
      xlab = expression(mu), ylab = "densidade", main = "priori")
```



A função `curve()` recebe como parâmetro principal uma função qualquer, $f(x)$, no caso, `dunif(x, min = 0, max = 33)` que é a implementação em R da densidade uniforme definida em (5).

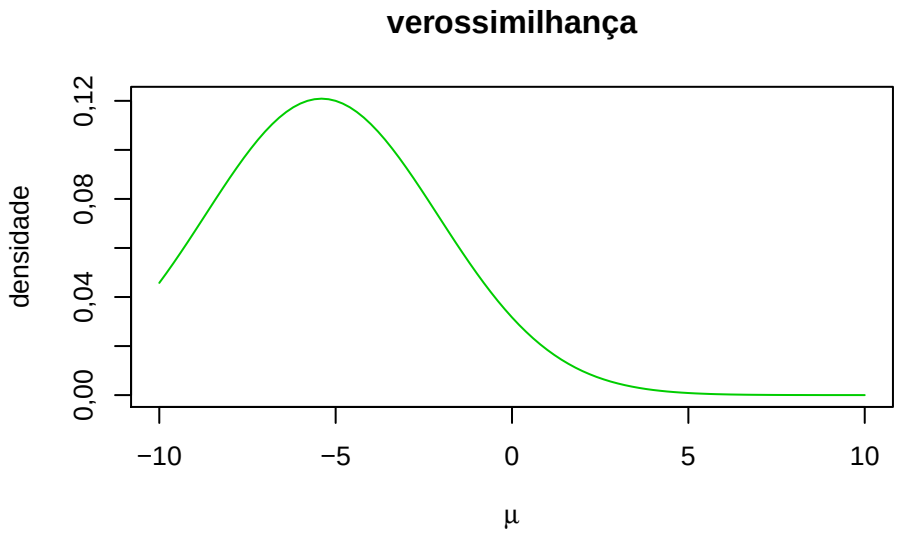
O valor medido da massa do neutrino num experimento foi de $-5{,}4\frac{\text{eV}}{\text{c}^2}$! A distribuição da verossimilhaça dessa observação é dada por:

$$g(x|\mu) = \text{N}(\mu, 3{,}3^2). \tag{6}$$

[1] A massa das partículas elementares são medidas a partir da equivalência massa-energia dada pela famosa equação $E = mc^2$.

A verossimilhança pode ser visualizada no R também através da função `curve()`.

```
curve(dnorm(-5.4, x, 3.3), from = -10, to = 10, col = 3,
      xlab = expression(mu), ylab = "densidade", main = "verossimilhança")
```



Naturalmente, um número negativo não é um valor esperado para uma grandeza física como a massa. A inferência Bayesiana, entretanto, permite fazer inferências sobre a massa do neutrino, mesmo com essa observação atípica! É necessário determinar a distribuição *a posteriori* da massa, μ, do neutrino dada a observação x, tal que:

$$f(\mu|x) \propto g(x|\mu)\pi(\mu). \tag{7}$$

É possível, utilizando o JAGS, obter amostras da distribuição *a posteriori* definida em (7). O modelo implementado em JAGS é dado por:

```
neutrino.jags <- function() {

    # verossimilhança
    x ~ dnorm(mu, 1 / 3.3 ** 2)

    # priori
    mu ~ dunif(0, 33)
}
```

Basta definir as distribuições *a priori* e de verossimilhança do modelo, conforme (5) e (6). A observação deve ser definida num objeto do tipo `list` do R. Assim, digite e execute:

```
dados.jags <- list(x = -5.4)
```

As estimações são feitas pela função `jags()`, como na seção anterior.

```
modelo.jags <- jags(data = dados.jags, model.file = neutrino.jags,
                    parameters.to.save = "mu", n.chains = 1,
                    n.iter = 15000, n.burnin = 5000, n.thin = 10)
```

Foram realizadas 15.000 iterações. Para ver os resultados, digite e execute:

```
print(modelo.jags)
```

```
Inference for Bugs model at "./jags/neutrino.jags", fit using jags,
 1 chains, each with 15000 iterations (first 5000 discarded), n.thin = 10
 n.sims = 1000 iterations saved
         mu.vect sd.vect  2,5%   25%   50%   75%  97,5%
mu         1,396   1,243 0,046 0,463 1,059 1,988  4,747
deviance   8,608   1,774 6,949 7,382 8,057 9,238 13,681

DIC info (using the rule, pD = var(deviance)/2)
pD = 1,6 and DIC = 10,2
DIC is an estimate of expected predictive error (lower deviance is better).
```

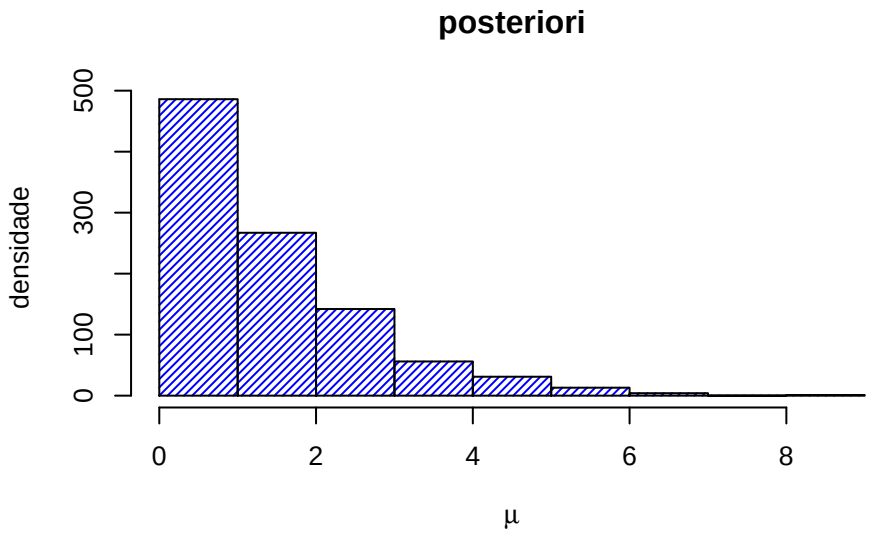
A média *a posteriori* de μ foi estimada em $1{,}396 \frac{eV}{c^2}$.

O objeto `modelo.jags` é da classe `rjags` e possui uma estrutura complexa. Para extrair a série de valores estimados da distribuição *a posteriori* de μ, digite e execute:

```
mu.jags <- as.numeric(modelo.jags$BUGSoutput$sims.list$m)
```

Desse modo, é possível visualizar a distribuição *a posteriori* da massa, μ, do neutrino no R pela função `hist()`. Digite e execute:

```
hist(mu.jags, col = 4, density = 33, border = "black",
     xlab = expression(mu), ylab = "densidade", main = "posteriori")
```



Assim, a inferência Bayesiana, através do JAGS, resolve o problema de estimar a massa do neutrino.

Referências

Andreon, S. and Weaver, B. (2015). *Bayesian Methods for the Physical Sciences: Learning from Examples in Astronomy and Physics*. Springer, New York, 1st edition.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2014). *Bayesian Data Analysis*. Chapman & Hall/CRC, Boca Raton, 3rd edition.

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphics Statistics*, 5(3):299–314.

Kruschke, J. (2014). *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan*. Academic Press, San Diego, 2nd edition.

Plummer, M. (2003). JAGS: A program for analysis of bayesian graphical models using Gibbs sampling. In *Distributed Statistical Computing, Vienna University of Technology*.

Sivia, D. and Skilling, J. (2006). *Data Analysis: A Bayesian Tutorial*. Oxford University Press, Oxford, 2nd edition.

Su, Y. and Yajima, M. (2015). *R2jags: Using R to Run JAGS*. R package version 0.5-6.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer, New York, 1st edition.