UNIVERSIDADE FEDERAL DE LAVRAS
DEPARTAMENTO DE CIÊNCIAS EXATAS

Estatística Computacional Utilizando R

Daniel Furtado Ferreira
danielff@ufla.br

LAVRAS
Minas Gerais - Brasil
11 de dezembro de 2009

Prefácio

Nestas notas de aula tivemos a intenção de abordar o tema de estatística computacional que é tão importante para a comunidade científica e principalmente para os estudantes dos cursos de pós-graduação em estatística. Podemos afirmar sem medo de errar que a estatística computacional se tornou e é hoje em dia uma das principais áreas da estatística. Além do mais, os conhecimentos desta área podem ser e, frequentemente, são utilizados em outras áreas da estatística, da engenharia e da física. A inferência Bayesiana é um destes exemplos típicos em que geralmente utilizamos uma abordagem computacional. Quando pensamos nestas notas de aulas tivemos muitas dúvidas do que tratar e como abordar cada tópico escolhido. Assim, optamos por escrever algo que propiciasse ao leitor ir além de um simples receituário, mas que, no entanto, não o fizesse perder em um emaranhado de demonstrações. Por outro lado buscamos apresentar os modelos e os métodos de uma forma bastante abrangente e não restritiva.

Uma outra motivação que nos conduziu e nos encorajou a desenvolver este projeto, foi a nossa experiência pessoal em pesquisas com a estatística computacional. Também fizemos isso pensando no benefício pessoal, não podemos negar, que isso nos traria ao entrarmos em contato direto com a vasta publicação existente neste ramo da estatística. Não temos, todavia, a intenção de estudarmos todos os assuntos e nem mesmo pretendemos para um determinado tópico esgotar todas as possibilidades. Pelo contrário, esperamos que estas notas sejam uma introdução a estatística computacional e que sirvam de motivação para que os alunos dos cursos de graduação em estatística possam se adentrar ainda mais nessa área.

Estas notas são baseadas em um livro que estamos escrevendo sobre a estatística computacional utilizando a linguagem Pascal. A adaptação para o R de algumas das rotinas implementadas neste livro foi uma tarefa bastante prazerosa e reveladora.

Aproveitamos esta oportunidade para desvendar um pouco dos inúmeros segredos que este poderoso programa possui e descobrir um pouco sobre seu enorme potencial e também, por que não dizer, de suas fraquezas. Nessa primeira versão não esperamos perfeição, mas estamos completamente cientes que muitas falhas devem existir e esperamos contar com a colaboração dos leitores para saná-las. Estamos iniciando uma segunda versão revisada e ampliada, utilizando ainda o Sweave, que integra o R ao LaTeX. Essas notas serão constantemente atualizadas na internet. Esperamos que este manuscrito venha contribuir para o crescimento profissional dos estudantes de nosso programa de pós-graduação em Estatística e Experimentação Agropecuária, além de estudantes de outros programas da UFLA ou de outras instituições. Dessa forma nosso objetivo terá sido atingido.

Daniel Furtado Ferreira
11 de dezembro de 2009

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução ao Programa R

O programa R (R Development Core Team, 2006[14]) foi escolhido para ministrar este curso por uma série de razões. Além de ser um programa livre, no sentido de possuir livre distribuição e código fonte aberto, pode ser utilizado nas plataformas Windows e Unix. Além do mais, o R possui grande versatilidade no sentido de possuir inúmeros pacotes já prontos e nos possibilitar criar novas rotinas e funções. O R é a versão livre baseada na linguagem S, cuja versão comercial é o S-Plus. O programa SAS, por outro lado, é o programa estatístico mais comum, mas o R é o mais popular. O programa R por ser genuinamente um programa orientado por objeto nos possibilita programar com muita eficiência e versatilidade. Outro aspecto que é bastante atrativo no R refere-se ao fato de o mesmo receber contribuições de pesquisadores de todo o mundo na forma de pacotes. Essa é uma característica que faz com que haja grande desenvolvimento do programa em relativamente curtos espaços de tempo e que nos possibilita encontrar soluções para quase todos os problemas com os quais nos deparamos em situações reais. Para os problemas que não conseguimos encontrar soluções, o ambiente de programação R nos possibilita criar nossas próprias soluções.

Nestas notas de aulas pretendemos apresentar os conceitos básicos da estatística computacional de uma forma bastante simples. Inicialmente obteremos nossas próprias soluções para um determinado método ou técnica e em um segundo momento mostraremos que podemos ter a mesma solução pronta do programa R quando esta estiver disponível. Particularmente neste capítulo vamos apresentar algumas características do ambiente e da linguagem R para implementarmos nossas soluções.

Nosso curso não pretende dar soluções avançadas e de eficiência máxima para os problemas que abordaremos, mas propiciar aos alunos um primeiro contato com a linguagem R (ou S) e com os problemas da estatística computacional. A linguagem R é um dialeto da linguagem S que foi desenvolvida em 1980 e se espalhou pela comunidade científica desde então.

A desvantagem é que o R não é um programa fácil de aprender. Alguns esforços iniciais são necessários até que consigamos obter algum benefício. Não temos a intenção de apresentar neste curso os recursos do R para análises de modelos lineares de posto completo ou incompleto, de modelos não-lineares, de modelos lineares generalizados ou de gráficos. Eventualmente poderemos utilizar algumas destas funções como um passo intermediário da solução do problema que estaremos focando. Este material será construído com a abordagem teórica do tópico e associará exemplificações práticas dos recursos de programação R para resolver algum problema formulado, em casos particulares da teoria estudada.

Este material é apenas uma primeira versão que deverá ter muitos defeitos. Assim, o leitor que encontrá-los ou tiver uma melhor solução para o problema poderá contribuir enviando um e-mail para *danielff@ufla.br*.

1.1 Introdução ao R

No R os símbolos ou variáveis são objetos e podem ter as mais variadas estruturas, tais como matrizes, vetores, *data frames*, listas, funções, expressões e muitos outras. Vamos descrever de forma sucinta e gradativa algumas destes objetos.

Os vetores são compostos de células contíguas de dados homogêneos. Os vetores podem ser de vários tipos como, por exemplo, lógicos, inteiros, reais e caracteres. O modo mais simples de criar um vetor é utilizar $>$ *x <- c(5, 1.5, 4.5, 6.7, 3.1)*. Esta instrução faz com que concatenemos os 5 elementos anteriores com a função c (concatenação) e o resultado é um vetor de tamanho 5. Podemos acessar qualquer elemento deste vetor utilizando a instrução $> x[j]$, em que $j = 1, 2, \cdots, 5$ e o tamanho do vetor por $> length(x)$. Se queremos ver o resultado do objeto na tela, devemos digitar o nome dele no *prompt* do R da seguinte forma $> x$. Podemos criar um vetor combinando dois vetores x, intercalados por zero da seguinte forma: $> y$ <- $c(x,0,x)$. Outro aspecto que devemos esclarecer é que as atribuições no R podem ser feitas por <- ou por =. Vamos utilizar neste material a maneira menos comum

para os usuários do R, atribuição com o =, por uma questão de emprego de outras linguagens de programação, em que o sinal de igualdade é utilizado para atribuições.

As instruções R devem ser preferencialmente escritas em um editor de texto como o bloco de notas. O R nos possibilita abrir tais arquivos textos, denominados de *scripts*, nos quais podemos executar os códigos digitados marcando-os e teclando simultaneamente *ctrl r*. Nossos programas exemplos não apresentarão o *prompt* > do programa R, por considerar que foram digitados em um arquivo texto. Cada linha deve conter uma função, uma atribuição ou uma instrução. Podemos digitar mais de uma instrução por linha se as separarmos por ponto e vírgula. Vamos ilustrar o uso de vetores com o exemplo a seguir. Neste exemplo aproveitamos o ensejo para apresentar alguns outros comandos úteis da linguagem R.

```
> # programa R demonstrando o uso de vetores e de intruções
> # úteis para lidarmos com eles. Todos os textos após #
> # serão considerados comentários
> x <- c(2, 3, 4, 5.1, 3.2)       # cria um vetor de tamanho 5
> x                               # imprime o vetor

[1] 2.0 3.0 4.0 5.1 3.2

> y <- c(x,0,1,x)                 # concatena x com o vetor [0; 1] e x
> y                               # imprime o vetor

 [1] 2.0 3.0 4.0 5.1 3.2 0.0 1.0 2.0 3.0 4.0 5.1 3.2

> z <- c(2.4, 1.3, 3.4, 2.1, 5.7) # cria um vetor de tamanho 5
> w <- 2*x+z+1                    # operações elementares com os vetores
> w                               # imprime o vetor w

[1]  7.4  8.3 12.4 13.3 13.1

> rx <- range(x)                  # vetor de tamanho 2: c(min(x); max(x))
> rx                              # imprime o vetor rx

[1] 2.0 5.1

> lx <- length(x)                 # tamanho do vetor x em lx
> lx                              # imprime lx

[1] 5

> x1 <- seq(from=1, to=10, by=0.5)# gera uma sequência 1 a 10 de 0.5 em 0.5
> x1                              # imprime x1

 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5
[11]  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

```
> xr1 <- rep(x,times=5)          # replica x na íntegra 5 vezes
> xr2 <- rep(x,each=5)           # replica cada elemento de x, 5 vezes
> xr1                            # imprime xr1

 [1] 2.0 3.0 4.0 5.1 3.2 2.0 3.0 4.0 5.1 3.2 2.0 3.0
[13] 4.0 5.1 3.2 2.0 3.0 4.0 5.1 3.2 2.0 3.0 4.0 5.1
[25] 3.2

> xr2                            # imprime xr2

 [1] 2.0 2.0 2.0 2.0 2.0 3.0 3.0 3.0 3.0 3.0 4.0 4.0
[13] 4.0 4.0 4.0 5.1 5.1 5.1 5.1 5.1 3.2 3.2 3.2 3.2
[25] 3.2
```

Os vetores podem ser utilizados em expressões aritméticas, sendo as operações realizadas elemento a elemento. As operações elementares são $+$, $-$, $*$, / e ^ para potenciação. Vários outros operadores aritméticos são disponíveis, como por exemplo, *log*, *exp*, *sin*, *cos*, *tan* e *sqrt*. Funções como *range* retornam um vetor de tamanho dois composto pelos valores $c(min(x), max(x))$ e o operador $length(x)$ retorna o tamanho do vetor x.

Podemos gerar sequências regulares no programa R facilmente. A instrução x <- $1:5$ gera um vetor x que seria obtido de forma equivalente por x <- c(1,2,3,4,5). O operador dois pontos possui a maior prioridade entre todos os operadores aritméticos dentro de uma expressão. Assim, $2 * 1:5$ irá gerar a sequência 2,4,6,8,10. Se queremos uma sequência que se inicia em 2 (exemplo óbvio) e termina em 5 com passo igual a 1, devemos usar a instrução $(2 * 1):5$, que irá produzir 2,3,4,5. A instrução $x1$ <- $seq(from = 1, to = 10, by = 0.5)$ gera uma sequência que vai de 1 a 10 utilizando um passo igual a 0,5. Podemos simplesmente utilizar $x1$ <- $seq(1, 10, by = 0.5)$. Se o limite superior da sequência não for um múltiplo exato do passo, então o R irá produzir uma sequência que irá finalizar no múltiplo mais próximo, mas inferior, ao limite superior. Uma outra instrução que é bastante útil nos permite dizer o tamanho da sequência, o valor inicial e o tamanho do passo, da seguinte forma: $seq(length = 19, from = 1, by = 0.5)$. Esta instrução irá produzir a mesma sequência do programa anterior. Podemos ainda utilizar tamanhos de passos negativos para a sequência, como por exemplo, seq(length=19, from=1, by=-0.5), que irá produzir uma sequência de 1 a -8 com tamanho de passo $-0{,}5$.

Um outro importante comando é o *rep*() que podemos utilizar para replicar um objeto das mais diferentes formas. No exemplo anterior utilizamos duas delas.

Utilizamos o comando $rep(objeto, times)$ e o comando $rep(objeto, each)$.

Podemos gerar vetores lógicos por condições ($>, >=, <, <=, ==, !=$). As condições == indicam igualdade exata e ! =, diferenças. Assim, o programa a seguir ilustra o uso de vetores lógicos. A combinação de duas condições é feita utilizando o operador & para *and* (interseção), | para *or* (união) e ! para negação de uma das condições. Podemos utilizar este operadores lógicos para realizarmos muitas tarefas complicadas. No exemplo apresentado a seguir formamos um novo vetor w contendo os elementos de x que são inferiores a 5. O segundo exemplo cria um vetor $w2$, cujos elementos são a soma dos elementos do vetor x que são maiores ou iguais a 3 e adicionados de 1. Os vetores são os tipos mais importantes em R, mas outros tipos importantes devem ser considerados. Várias outras operações vetoriais são possíveis e serão descritas oportunamente, na medida que necessitarmos deles.

```
> # programa R demonstrando o uso de vetores lógicos
> x <- c(2.2, 1.3, 4.7, 5.1, 3.2) # cria vetor de tamanho 5
> x                # imprime o vetor

[1] 2.2 1.3 4.7 5.1 3.2

> y <- x > 5.0      # cria vetor lógico
> y                # imprime o vetor

[1] FALSE FALSE FALSE  TRUE FALSE

> z <- !y           # vetor lógico de negação de y
> z                # imprime o vetor

[1]  TRUE  TRUE  TRUE FALSE  TRUE

> w <- x[x<5.0]     # vetor w com elementos de x < 5
> w                # imprime o vetor w

[1] 2.2 1.3 4.7 3.2

> w2 <- (x+1)[x>=3.0] # elementos de x >= 3, adicionados de 1
> w2                 # imprime o vetor w2

[1] 5.7 6.1 4.2
```

O R nos possibilita lidar com arranjos de várias dimensões. Vamos utilizar neste material apenas o arranjo mais simples de duas dimensões que são as matrizes. A forma mais simples de gerarmos uma matriz é utilizar o comando $matrix(0, n, p)$. Este comando nos permite gerar uma matriz composta de elementos iguais a zero e de dimensões linha e coluna iguais a n e p, respectivamente. Os elementos desta

matriz (A) são preenchidos atribuindo, por exemplo, na linha i e coluna j, o valor desejado 10 da seguinte forma $A[i,j]$ <- 10. Variando-se o índice da linha e da coluna podemos atribuir valores a quaisquer outras posições da matriz A.

Alguns operadores aplicados a matriz são executados elemento a elemento. Por exemplo, se fizermos $A * B$, teremos uma matriz resultante do produto dos elementos de A com os elementos correspondentes da matriz B. Se desejarmos o produto matricial entre duas matrizes, devemos utilizar o operador $\%*\%$. Assim, $A\%*\%B$ retorna a matriz resultante da multiplicação matricial de A por B. Os operadores $nrow(A)$ e $ncol(A)$ retornam o número de linhas e de colunas de A, respectivamente. O operador $diag$ depende do argumento. Se $diag(x)$ é aplicado a um vetor x, o resultado é uma matriz diagonal cujos elementos são iguais aos elementos do vetor x. Se por outro lado utilizarmos $diag(A)$, em que A é uma matriz, obteremos um vetor cujos elementos são os mesmos da diagonal principal de A. O comando $solve(A)$ retorna a inversa de A, ou seja, A^{-1}. Também pode ser utilizado para resolver sistemas de equações lineares do tipo $y = Ax$, da seguinte forma x <- $solve(A,y)$. Autovalores e autovetores de uma matriz simétrica A podem ser computados por ev <- $eigen(A)$. Os autovalores são armazenados no objeto (vetor) $ev\$values$ e os autovetores correspondentes, no objeto (matriz) $ev\$vec$. Cada coluna desta matriz contém um autovetor correspondente ao elemento associado do vetor de autovalores. Uma matriz qualquer pode ser decomposta da seguinte forma $A = UDV^\top$. Esta decomposição é conhecida de decomposição do valor singular e a função R é dada por udv <- $svd(A)$. O resultado é uma lista com os objetos u, d e v com significados óbvios. Uma lista possui objetos de diferentes tipos que podem ser acessados por $objeto\$componente$. No exemplo, obtivemos os elementos da matriz diagonal D utilizando o comando $udv\$d$. Os exemplos apresentados no programa a seguir ilustram as funções e os comandos retro mencionados.

```
> # programa R demonstrando o uso de matrizes e de alguns de seus operadores
> A <- matrix(0,2,2) # cria uma matriz 2 x 2 de zeros
> A[1,1] <- 4; A[1,2] <- 1; A[2,1] <- 1; A[2,2] <- 2 # atribui valores a matriz
> A # imprime a matriz

     [,1] [,2]
[1,]    4    1
[2,]    1    2

> B <- matrix(c(8,-1,-1,1),2,2) # cria uma matriz 2 x 2 de outra forma
> B # imprime a matriz
```

```
     [,1] [,2]
[1,]    8   -1
[2,]   -1    1
```

```
> C <- A * B # multiplicação elemento por elemento
> C # imprime a matriz
```

```
     [,1] [,2]
[1,]   32   -1
[2,]   -1    2
```

```
> nrow(C);ncol(C)# número de linhas e colunas de C
```

```
[1] 2
```

```
[1] 2
```

```
> D <- A%*%B     # multiplicação matricial A por B
> D              # imprime a matriz resultante
```

```
     [,1] [,2]
[1,]   31   -3
[2,]    6    1
```

```
> v <- diag(D)   # vetor com a diagonal principal de D
> v              # imprime o vetor resultante
```

```
[1] 31  1
```

```
> AI <- solve(A) # inversa de A
> AI             # imprime a matriz resultante
```

```
           [,1]       [,2]
[1,]  0.2857143 -0.1428571
[2,] -0.1428571  0.5714286
```

```
> ev <- eigen(A) # autovalores e autovetores de A
> ev$vec         # imprime os autovetores
```

```
           [,1]       [,2]
[1,] -0.9238795  0.3826834
[2,] -0.3826834 -0.9238795
```

```
> ev$values      # imprime os autovalores
```

```
[1] 4.414214 1.585786
```

```
> udv <- svd(A)  # decomposição do valor singular de A
> udv            # imprime todos os resultados da lista: DVS
```

```
$d
[1] 4.414214 1.585786

$u
           [,1]       [,2]
[1,] -0.9238795 -0.3826834
[2,] -0.3826834  0.9238795

$v
           [,1]       [,2]
[1,] -0.9238795 -0.3826834
[2,] -0.3826834  0.9238795
```

Muitos outros objetos existem no R e serão descritos oportunamente, na medida que aparecerem nas funções que desenvolveremos posteriormente. Vamos somente descrever, para finalizar, como podemos ler um arquivo de dados no formato texto e associá-lo a um objeto R. Vamos imaginar que tenhamos um arquivo de dados com duas variáveis X_1 e X_2 e $n = 20$ observações. Devemos criar um arquivo com algum editor de texto, onde na primeira linha colocamos a identificação de cada coluna (variável). Neste exemplo vamos criar um arquivo com o nome "dados.txt".

Para ler estes dados do arquivo e associá-lo a um objeto R do tipo *data frame*, devemos utilizar o comando *read.table()*. O comando é usado da seguinte forma: *read.table("dados.txt",header=TRUE)*. A opção *header=TRUE* indica ao R que nosso arquivo possui na primeira linha a identificação das variáveis. No exemplo a seguir apresentamos detalhes destes comandos em um pequeno programa R. É conveniente mudarmos o diretório de trabalho para o local onde se encontra o arquivo "dados.txt" no menu *File\Change dir* do R. Se quisermos editar os dados e atribuí-los a um novo objeto (*data frame*), devemos utilizar o comando > *edit(dados)*. Um planilha será aberta para que sejam efetuadas as trocas necessárias.

O arquivo de dados é

X1	X2
13.4	14
14.6	15
13.5	19
15.0	23
14.6	17

14.0	20
16.4	21
14.8	16
15.2	27
15.5	34
15.2	26
16.9	28
14.8	24
16.2	26
14.7	23
14.7	9
16.5	18
15.4	28
15.1	17
14.2	14

e o programa

```
> # programa R demonstrando a leitura de arquivos de dados
> dados <- read.table("dados.txt",header=TRUE) # lê o arquivo no data frame dados
> dados.novo <- edit(dados)                    # editando o data frame
```

Finalmente, devemos enfatizar que o R diferencia entre comandos digitados em maiúsculo e minúsculo. Por exemplo na opção *header=TRUE*, a palavra *TRUE* deve estar em maiúsculo. Se for digitado *header=True* ou *header=true* o R acusará um erro e o objeto *dados* não será criado. No entanto, para esta opção, o R aceita a simplificação *header=T*.

1.2 Estruturas de Controle de Programação

O R é um ambiente de programação cujos comandos são expressões. Estas expressões podem ser organizadas em grupos com uso de chaves {*expr_1; expr_2;* $\cdots$*; expr_m*}. O valor do grupo é o resultado da última expressão avaliada. Os comandos do R são separados por ponto e vírgula ou por uma nova linha. O ponto e vírgula sempre indica o fim do comando, enquanto uma nova linha pode indicar ou não o fim do comando. O comando pode ser associado a um objeto ou não. Quando não estiver associado a um objeto, o valor do comando é retornado imediatamente. Assim, $> x <- 2*3$ e $> 2*3$ farão a mesma coisa, mas no primeiro caso o resultado é armazenado no objeto x e no segundo o seu valor é retornado na janela de *console* do

R. Vejamos o exemplo de alguns comandos com uso de agrupamentos no programa a seguir. Observe que nos comandos que se estendem por mais de uma linha, o prompt $>$ se altera para $+$, aguardando que a continuação do comando seja digitada.

```
> # programa R demonstrando comandos agrupados e comandos
> # em mais de uma linha e em apenas uma linha
> x <- 1; x <- x+10
> x
[1] 11
> x <- 1; x + 10
[1] 11
> {
+   x <- 1
+   x + 10
+ }
[1] 11
> z <- {x=0; y <- x + 1; x <- y + 1} # quanto vale z?
> z
[1] 2
```

O $if/else$ é um importante comando de controle que avalia condições para realizar um determinado comando ou grupo de comandos, se o resultado for verdadeiro, ou outro comando ou grupo, se o resultado for falso. Múltiplos comandos podem ser hierarquizados. O comando formal é feito da seguinte forma:

> if (expr_1) expr_2 else expr_3

A expr_1 é avaliada e se o resultado lógico for verdadeiro, então expr_2 é executada; se o resultado for falso a expr_3 será avaliada. Se o expr_2 não for um bloco de comandos o else deve obrigatoriamente ficar na mesma linha de comando de expr_1. O programa a seguir ilustra algumas das possibilidades para o uso do comando $if/else$. O programa não faz nada de útil e algumas comparações não fazem nenhum sentido, a não ser o de exemplificar o comando.

```
> # programa R demonstrando alguns usos do if/else
> x <- 1
> if (x > 0.5) y <- 5 else y <- x
> y
[1] 5
```

```
> z <- c(0.5,-0.5, 4.5,10) # experimente mudar valores do vetor z
> if (any(z <= 0) ) w1 <- log(1+z) else w1 <- log(z)
> w1

[1]  0.4054651 -0.6931472  1.7047481  2.3978953

> w2 <- if (any(z <= 0) ) log(1+z) else log(z)
> w2

[1]  0.4054651 -0.6931472  1.7047481  2.3978953

> if (any(z <= 0)) w3 <- z[z <= 0]
> w3

[1] -0.5

> # comandos compostos junto ao if
> if (any(z <= 0))
+ {
+
+    w4 <- z[z <= 0]
+    w4 <- z[z <= 0]^2 + w4
+ } else w4 <- z^0.5  # aqui poderia ter comando composto também
> w4

[1] -0.25
```

Neste programa os comandos > *if(any(z <= 0)) w1 <- log(1+z) else w1 <- log(z)* e > *w2 <- if(any(z <= 0)) log(1+z) else log(z)* são equivalentes. Se houver no vetor algum elemento menor ou igual a zero, o logaritmo neperiano será tomado elemento a elemento do vetor adicionado de 1 e em caso contrário, será tomado diretamente nos valores originais do vetor. Assim, os vetores $w1$ e $w2$ são iguais. O comando > *if(any(z <= 0)) w3 <- z[z <= 0]*, por sua vez, criará o objeto $w3$ contendo o subvetor de z, cujos elementos são inferiores ou iguais a zero. No entanto, se nenhum valor de z for inferior ou igual a zero, então o subvetor $w3$ não será criado. No caso de vários comandos aninhados ou hierarquizados temos a seguinte estrutura. Um dos comandos numerados por números pares será executado.

```
if (expr_1)
    expr_2
  else if (expr_3)
      expr_4
    else if (expr_5)
        (expr_6)
```

else
expr_8

O R possui alguns comandos apropriados para realizar *loops*, ou seja, as chamadas estruturas de repetições. Os comandos para isso são o *for*, *while* e o *repeat*. Adicionalmente os comandos *next* e *break* fornecem controles adicionais sobre estas estruturas. O R é tão flexível que estas estruturas retornam o último valor computado e nos permite, embora seja extremamente incomum, associar uma variável aos seus resultados. Uma outra vantagem do R diz respeito ao fato de que muitas operações, especialmente as aritméticas, são vetoriais e, portanto, podemos evitar os *loops*, que são menos eficientes.

O primeiro comando que descreveremos é o *repeat*. Devemos tomar muito cuidado na especificação da condição de parada para evitarmos repetições infinitas (*loops* infinitos). A sintaxe geral do comando é:

> repeat expr

O comando *expr* deve ser necessariamente um grupo de comandos (comando composto). Um destes comandos irá fazer uma avaliação da condição de parada e se esta condição for satisfeita o comando *break* será usado para interromper o *loop*. O comando *break* pode ser usado para cessar qualquer um dos outros comandos de repetição, que é, em geral, não normal, mas no caso do *repeat* constituí-se na única forma. No programa apresentado a seguir, ilustramos o uso do comando *repeat*. Este programa não faz nada de útil e tem apenas a finalidade de ilustrar o comando.

```
> # programa R demonstrando uso do repeat/break
> i <- 1
> repeat
+ {
+    print(i)
+    i <- i + log(i+1)*3
+    if (i > 35) break
+ }
[1] 1
[1] 3.079442
[1] 7.297322
[1] 13.64512
[1] 21.69744
[1] 31.0642
```

O comando *while* é muito parecido com o *repeat* e a sintaxe geral é:

> while (condição) expr

Serão repetidos os comandos do grupo *expr* até que a condição de parada seja satisfeita. Novamente é necessário frisar que deve-se tomar muito cuidado com a condição imposta para evitarmos *loops* infinitos. Nos códigos seguintes repetimos o mesmo programa feito com o comando *repeat*, porém utilizando o comando *while*.

```
> # programa R demonstrando uso do while
> i <- 1
> while (i <= 35)
+ {
+   print(i)
+   i <- i + log(i+1)*3
+ }
[1] 1
[1] 3.079442
[1] 7.297322
[1] 13.64512
[1] 21.69744
[1] 31.0642
```

O comando *for* é usado para repetirmos uma determinada sequência ou grupo de operações em um número fixo de vezes. Os passos do *loop* no R são determinados por um vetor ou sequência do tipo $ini : fim$. A sintaxe geral do *for* para uma dada sequência (seq) e um grupo de comandos (expr) é dada por:

> for (i in seq) expr

Um exemplo de programa R para ilustrar o comando *for* de controle de *loops* é dado por:

```
> # programa R demonstrando uso do for
> x <- matrix(0,10,1)
> for(i in 1:10)
+ {
+   x[i] <- i * 2 + 3 - log(i)
+ }
> x
          [,1]
 [1,]  5.000000
```

```
 [2,]  6.306853
 [3,]  7.901388
 [4,]  9.613706
 [5,] 11.390562
 [6,] 13.208241
 [7,] 15.054090
 [8,] 16.920558
 [9,] 18.802775
[10,] 20.697415
```

Como já salientamos dentro do grupo de comandos do *for* ou do *while* podemos utilizar o *break* se alguma condição for satisfeita. Com o *break* o R irá executar o próximo comando, se houver, após o término do *loop*.

1.3 Funções no R

O R nos possibilita criar nossas próprias funções, que são genuinamente funções R, sendo armazenadas internamente de uma forma especial e podendo ser utilizadas em novas expressões. Desta forma a linguagem ganha grande poder, conveniência e elegância. O aprendizado em escrever funções úteis é uma das muitas maneiras de fazer com que o uso do R seja confortável e produtivo. A sintaxe geral de uma função é dada por:

> nome = function(arg_1, arg_2, ...) expr

em que expr é, normalmente, um grupo de comandos e nome é o objeto que receberá a função. Sua chamada se dará por um comando *nome(a1, a2, ...)*, em que $a1$, $a2$, etc. são os valores que deverão ser passados como argumentos dos objetos *(arg_1, arg_2, ...)*.

Vamos apresentar uma função simples para testar a hipótese $H_0 : \mu = \mu_0$ a partir de uma amostra simples de uma distribuição normal. Dois argumentos serão utilizados: o vetor de dados x de tamanho n e o valor real hipotético μ_0. A função calculará o valor da estatística t_c do teste por:

$$t_c = \frac{\bar{X} - \mu_0}{\frac{S}{\sqrt{n}}}. \tag{1.3.1}$$

A função resultante, em linguagem R, é apresentada a seguir. Neste exemplo uma amostra de tamanho $n = 8$ foi utilizada para obter o valor da estatística para

testar a hipótese $H_0 : \mu = 5{,}0$. Podemos observar que o resultado final da função é igual ao do último comando executado, ou seja o valor da estatística e do valor-p, por meio de um objeto do tipo lista. Se o comando e *return()* não for utilizado, a função retornará sempre como resultado, o valor da última operação ou comando executado. Esta função utiliza no seu escopo três outras funções do R (funções básicas do R). As duas primeiras, *var()* e *mean()* retornam a variância e a média do vetor utilizado como argumento, respectivamente, e a terceira, *pt()*, retorna a probabilidade acumulada da distribuição t de Student para o primeiro argumento da função com ν graus de liberdade, que é o seu segundo argumento.

```
> # programa R demonstrando uso de uma função
> tteste <- function(x,mu0)
+ {
+   n  <- length(x)
+   S2 <- var(x)
+   xb <- mean(x)
+   t <- list(tc=c(0), pr=c(0))
+   t$tc <- (xb-mu0)/sqrt(S2/n)
+   t$pr <- 2 * (1 - pt(abs(t$tc), n - 1))
+   return(t)
+ }
> y <- c(3.4,5.6,4.0,6.0,7.8,9.1,3.4,4.5)
> t <- tteste(y,5.0)
> t

$tc
[1] 0.6428422


$pr
[1] 0.540803
```

Os argumentos x e *mu0* definem as variáveis cujos valores serão fornecidos quando a função for chamada. Os nomes destes argumentos podem ser utilizados dentro do escopo desta função e os valores devem ser fornecidos no momento em que função foi chamada, como neste caso. No exemplo, os valores especificados anteriormente para os componentes do vetor y e o valor 5,0 para o valor hipotético foram utilizados como argumentos da função. Podemos utilizar valores *default* para os argumentos, como por exemplo *variável = valor*. Poderíamos utilizar o *default* *mu0 = 0* na declaração da função. Neste caso, se o valor hipotético não for passado, então a hipótese testada será sempre $H_0 : \mu = 0$.

Assim, poderíamos ter o seguinte programa resultante:

```
> # programa R demonstrando uso de uma função
> tteste <- function(x,mu0 = 0)#usando valor default para mu0
+ {
+   n  <- length(x)
+   if (n <= 1) stop("Amostra deve conter mais de 1 elemento")
+   S2 <- var(x)
+   xb <- mean(x)
+   # acrescentando informações amostrais na lista de saída
+   t <- list(tc=c(0), pr=c(0), média = xb, variância = S2, n = n)
+   t$tc <- (xb-mu0)/sqrt(S2/n)
+   t$pr <- 2 * (1 - pt(abs(t$tc), n - 1))
+   return(t)
+ }
> y <- c(3.4,5.6,4.0,6.0,7.8,9.1,3.4,4.5)
> tc1 <- tteste(x = y, mu0 = 5.0)
> tc1

$tc
[1] 0.6428422

$pr
[1] 0.540803

$média
[1] 5.475

$variância
[1] 4.367857

$n
[1] 8

> tc2 <- tteste(x = y) # testar H0: mu = 0
> tc2

$tc
[1] 7.409602

$pr
[1] 0.0001482082

$média
```

```
[1] 5.475

$variância
[1] 4.367857

$n
[1] 8
```

```
> tc3 <- tteste(y, 5.0) # testar H0:mu = 5
> tc3
```

```
$tc
[1] 0.6428422

$pr
[1] 0.540803

$média
[1] 5.475

$variância
[1] 4.367857

$n
[1] 8
```

Neste programa demonstramos diferentes formas de invocar a função. No caso, os argumentos que possuem *default* podem ser omitidos, desde que a ordem de entrada dos argumentos não seja alterada, ou desde que entremos com os argumentos nomeando-os, como foi o caso dos exemplos *tc1* e *tc2*. O R, diferentemente do S-plus, irá buscar um objeto utilizado em uma função preferencialmente no próprio escopo da função e somente após não encontrá-lo localmente, será buscado o objeto global. O S-plus faz exatamente o contrário, busca preferencialmente os objetos globais de mesmo nome dos objetos locais. Iremos dar preferências às funções neste material para executarmos tarefas e resolvermos problemas que eventualmente defrontaremos daqui por diante. O resultado de uma função pode ser colocado em uma lista e o objeto que irá invocar a função herdará os objetos desta lista. Assim, como vimos nesta função, calculamos, além do valor da estatística, o valor-p (pr). Se não tivéssemos criado a lista t, poderíamos ter calculado o valor-p e o valor da estatística separadamente e terminaríamos a função utilizando o seguinte comando:

> list(tc=t, valor.p=pr). Os objetos *tc1*, *tc2* e *tc3* possuem os componentes *tc* e *pr* como resultados e são chamados, por exemplo, por *tc1$tc* e *tc1$pr*. Outras informações amostrais foram acrescentadas ao objeto do tipo lista t, como a média, a variância e o tamanho.

1.4 Introdução a Estatística Computacional

Os métodos de computação intensiva têm desempenhado um papel cada vez mais importante para resolver problemas de diferentes áreas da ciência. Vamos apresentar algoritmos para gerar variáveis aleatórias de diversas distribuições de probabilidade, para realizar operações matriciais, para realizar inferências utilizando métodos de permutação e bootstrap, etc. Assim, buscamos realizar uma divisão deste material em uma seção básica e em outra aplicada. As técnicas computacionais são denominadas de estatística computacional se forem usadas para realizarmos inferências, para gerarmos variáveis aleatórias ou para compararmos métodos e técnicas estatísticas.

Vamos explorar métodos de geração de variáveis aleatórias de diversos modelos probabilísticos, para manipularmos matrizes, para obtermos quadraturas de funções de distribuição de diversos modelos probabilísticos e de funções especiais na estatística e finalmente vamos apresentar os métodos de computação intensiva para realizarmos inferências em diferentes situações reais. Temos a intenção de criar algoritmos em linguagem R e posteriormente, quando existirem, apresentar os comandos para acessarmos os mesmos algoritmos já implementados no R.

Temos a intenção de apresentar os métodos de bootstrap e Monte Carlo, os testes de permutação e o procedimento jackknife para realizarmos inferências nas mais diferentes situações reais. Assim, este curso tem basicamente duas intenções: possibilitar ao aluno realizar suas próprias simulações e permitir que realizem suas inferências de interesse em situações em que seria altamente complexo o uso da inferência clássica.

Seja na inferência frequentista ou na inferência Bayesiana, os métodos de simulação de números aleatórios de diferentes modelos probabilísticos assumem grande importância. Para utilizarmos de uma forma mais eficiente a estatística computacional, um conhecimento mínimo de simulação de variáveis aleatórias é uma necessidade que não deve ser ignorada. Vamos dar grande ênfase a este assunto, sem descuidar dos demais. Apresentaremos neste material diversos algoritmos desenvolvidos e

adaptados para a linguagem R.

Simular é a arte de construir modelos segundo Naylor et al. (1971) [11], com o objetivo de imitar o funcionamento de um sistema real, para averiguarmos o que aconteceria se fossem feitas alterações no seu funcionamento (Dachs, 1988 [2]). Este tipo de procedimento pode ter um custo baixo, evitar prejuízos por não utilizarmos procedimentos inadequados e otimizar a decisão e o funcionamento do sistema real.

Precauções contra erros devem ser tomadas quando realizamos algum tipo de simulação. Podemos enumerar:

1. escolha inadequada das distribuições;
2. simplificação inadequada da realidade; e
3. erros de implementação.

Devemos fazer o sistema simulado operar nas condições do sistema real e verificar por meio de alguns testes se os resultados estão de acordo com o que se observa no sistema real. A este processo denominamos de validação. Então, a simulação é uma técnica que usamos para a solução de problemas. Se a solução alcançada for mais rápida, de menor custo e de fácil interpretação em relação a outro método qualquer, o uso de simulação é justificável.

1.5 Exercícios

1.5.1 Criar no R os vetores $A^\top = [1, 2, 4, 5]$ e $B^\top = [3, 6, 8, 9]$ e concatená-los formando um único vetor. Obter o vetor $C = 2A + B$ e o vetor $D = A^\top B$. Criar uma sequência cujo valor inicial é igual a 2 e o valor final é 30 e cujo passo é igual a 4. Replicar cada valor da sequência 3 vezes de duas formas diferentes (valores replicados ficam agregados e a sequência toda se replica sem que os valores iguais fiquem agregados).

1.5.2 Selecionar o subvetor de $x^\top = [4, 3, 5, 7, 9, 10]$ cujos elementos são menores ou iguais a 5.

1.5.3 Criar a matriz

$$A = \begin{bmatrix} 10 & -1 \\ -1 & 2 \end{bmatrix}$$

e determinar os autovalores e a decomposição espectral de A.

1.5.4 Construir uma função para verificar quantos elementos de um vetor de dimensão n são menores ou iguais a 0. Utilize as estruturas de repetições *for*, *while* e *repeat* para realizar tal tarefa (cada uma destas estruturas deverá ser implementada em uma diferente função). Existe algum procedimento mais eficiente para gerarmos tal função sem utilizar estruturas de repetições? Se sim, implementá-lo.

1.5.5 Implementar uma função R para realizar o teste t de Student para duas amostras independentes.

Capítulo 2

Geração de Variáveis Uniformes

Neste capítulo vamos considerar a geração de números aleatórios para o modelo probabilístico uniforme. A partir do modelo uniforme podemos gerar variáveis aleatórias de qualquer outro modelo probabilístico. Para gerar realizações de uma distribuição uniforme, precisamos gerar números aleatórios. Isso não pode ser realizado por máquinas. Na verdade qualquer sequência produzida por uma máquina é na verdade uma sequência previsível. Dessa forma, a denominamos de sequência de números pseudo-aleatórios.

Uma sequência de números será considerada "aleatória" do ponto de vista computacional se o programa que a gerar for diferente e estatísticamente não correlacionado com o programa que a usará. Assim, dois geradores de números aleatórios deveriam produzir os mesmos resultados nas suas aplicações. Se isso não ocorrer, um deles não pode ser considerado um bom gerador de números aleatórios (Press et al., 1992 [13]).

Os conceitos de números uniformes e números aleatórios podem ser muitas vezes confundidos. Números uniformes são aqueles que variam aleatoriamente em uma faixa determinada de valores com probabilidade constante. No entanto, devemos diferenciar números aleatórios uniformes de outros tipos de números aleatórios, como por exemplo, números aleatórios normais ou gaussianos. Estes outros tipos são geralmente provenientes de transformações realizadas nos números aleatórios uniformes. Então, uma fonte confiável para gerar números aleatórios uniformes determina o sucesso de métodos estocásticos de inferência e de todo processo de simulação Monte Carlo.

2.1 Números Aleatórios Uniformes

Números uniformes aleatórios são aqueles que, a princípio, se situam dentro de uma determinada amplitude, geralmente entre 0 e 1, para os quais não podemos produzir uma sequência previsível de valores. Em vários programas estes números são gerados utilizando o comando "random" ou comandos similares. Em Pascal, por exemplo, se este comando for utilizado com o argumento n, "random(n)", números aleatórios inteiros U do intervalo $0 \leq U \leq n-1$ são gerados e se o argumento n não for usado, os números gerados são valores aleatórios reais do intervalo $[0;1[$.

Em geral, os programas utilizam o método da congruência. Sejam os números uniformes inteiros $U_1, U_2, U_3, \ldots$ entre 0 e $m-1$, em que m representa um grande número inteiro. Podemos gerar estes números utilizando o método da congruência por meio da relação recursiva:

$$U_{i+1} = (aU_i + c) \mod m \tag{2.1.1}$$

em que m é chamado de módulo, a e c são inteiros positivos denominados de multiplicador e incremento, respectivamente. O operador mod retorna o resto da divisão do argumento $(aU_i + c)$ por m. A sequência recorrente (2.1.1) se repete em um período que não é maior que m, por razões óbvias. Se a, c e m são adequadamente escolhidos, a sequência tem tamanho máximo igual a m. A escolha do valor inicial U_0 é também muito importante. O valor do número uniforme correspondente no intervalo de 0 a 1 é dado por U_{i+1}/m, que é sempre menor que 1 mas que pode ser igual a zero.

Vamos apresentar um exemplo didático para ilustrar o gerador de números aleatórios. Sejam $U_0 = a = c = 7$ e $m = 10$, logo,

$$U_1 = (7 \times 7 + 7) \mod 10 = 56 \mod 10 = 6$$

$$U_2 = (7 \times 6 + 7) \mod 10 = 49 \mod 10 = 9$$

e assim sucessivamente. Obtemos a sequência de números aleatórios:

$$\{7, 6, 9, 0, 7, 6, 9, 0, 7, 6, 9, \cdots\}$$

e verificamos que o período é igual a 4 $\{7,6,9,0,\cdots\}$, que é menor do que $m = 10$.

Este método tem a desvantagem de ser correlacionado em série. Se m, a ou c não forem cuidadosamente escolhidos a correlação pode comprometer a sequência

gerada em série. Por outro lado, o método tem a vantagem de ser muito rápido. Podemos perceber que a cada chamada do método, somente alguns poucos cálculos são executados. Escolhemos, em geral, o valor de m pelo maior inteiro que pode ser representado pela máquina, qual seja, 2^{32}. Um exemplo que foi utilizado por muitos anos nos computadores "IBM mainframe", que representam uma péssima escolha é $a = 65.539$ e $m = 2^{31}$.

A correlação serial não é o único problema desse método. Os bits de maior ordem são mais aleatórios do que os bits de menor ordem (mais significantes). Devemos gerar inteiros entre 1 e 20 por $j = 1 + int(20 \times random(semente))$, ao invés de usar o método menos acurado $j = 1 +$ mod $(int(1000000 \times random(semente))$,20), que usa bits de menor ordem. Existem fortes evidências, empíricas e teóricas, que o método da congruência

$$U_{i+1} = aU_i \mod m \tag{2.1.2}$$

é tão bom quanto o método da congruência com $c \neq 0$, se o módulo m e o multiplicador a forem escolhidos com cuidado (Press et al., 1992[13]). Park e Miller (1988)[12] propuseram um gerador "padrão" mínimo baseado nas escolhas:

$$a = 7^5 = 16.807 \quad m = 2^{31} - 1 = 2.147.483.647 \tag{2.1.3}$$

Este gerador de números aleatórios não é perfeito, mas passou por todos os testes a qual foi submetido e tem sido usado como padrão para comparar e julgar outros geradores. Um problema que surge e que devemos contornar é que não é possível implementarmos diretamente em uma linguagem de alto-nível a equação (2.1.2) com as constantes de (2.1.3), pois o produto de a e U_i excede, em geral, o limite máximo de 32 bits para inteiros. Podemos usar um truque devido a Schrage (1979)[15] para multiplicar inteiros de 32-bits e aplicar o operador de módulo e que permite uma portabilidade para ser implementado em praticamente todas as linguagens e em todas as máquinas. O algoritmo de Schrage baseia-se na fatoração aproximada de m dada por:

$$m = aq + r; \qquad \text{i.e.,} \qquad q = \lfloor m/a \rfloor; \quad r = m \mod a \tag{2.1.4}$$

em que $\lfloor z \rfloor$ denota a parte inteira do número z utilizado como argumento. Para um número U_i entre 1 e $m - 1$ e para r pequeno, especificamente para $r < q$, Schrage

(1979) [15] mostraram que ambos $a(U_i \mod q)$ e $r\lfloor U_i/q \rfloor$ pertencem ao intervalo $0 \cdots m-1$ e que

$$aU_i \mod m = \begin{cases} a(U_i \mod q) - r\lfloor U_i/q \rfloor & \text{se maior que 0} \\ a(U_i \mod q) - r\lfloor U_i/q \rfloor + m & \text{caso contrário.} \end{cases} \tag{2.1.5}$$

Computacionalmente observamos que a relação:

$$a(U_i \mod q) = a(U_i - q\lfloor U_i/q \rfloor)$$

se verifica. No R pode-se optar por usar o operador %% (*mod*), que retorna o resto da operação entre dois inteiros e o operador %/% (*div*), que retorna o resultado do dividendo para operações com inteiros. A quantidade $U_i \mod q = U_i - (U_i \text{ div } q) \times q$ pode ser obtida em R simplesmente com U_i %% q. Atribuímos o resultado a uma variável qualquer definida como inteiro. Para aplicarmos o algoritmo de Schrage às constantes de (2.1.3) devemos usar os seguintes valores: $q = 127.773$ e $r = 2.836$.

A seguir apresentamos o algoritmo do gerador padrão mínimo de números aleatórios:

```
> # gerador padrão mínimo de números aleatórios adaptado de Park and
> # Miller. Retorna desvios aleatórios uniformes entre 0 e 1. Fazer
> # "sem" igual a qualquer valor inteiro para iniciar a sequência;
> # "sem" não pode ser alterado entre sucessivas chamadas da sequência
> # se "sem" for zero ou negativo, um valor dependente do valor do relógio
> # do sistema no momento da chamada é usado como semente. Constantes
> # usadas a = 7^5 = 16.807; m = 2^31 - 1 = 2.147.483.647
> # e c = 0
> gna0 <- function(n,sem)
+ {
+   gnu0 <- function (sem) # função local
+   {
+      k <- sem %/% iq # divisão de inteiros
+      # calculando sem <- mod(ia * sem, im) sem provocar overflow- Schrage
+      sem <- ia * (sem - k * iq) - ir * k
+      if (sem < 0) sem <- sem+im
+      ran0 <- am * sem # converte sem para ponto flutuante
+      return(list(ran0 = ran0, sem = sem))
+   }
+   ia <- 16807;im <- 2147483647;am <- 1.0/im
+   iq <- 127773;ir <- 2836
```

```
+   if (sem<=0)
+   {
+     library(R.utils) #library necessária
+     asem <- GString$getBuiltinTime(format="%H:%M:%S") #depende relógio/sistema
+     sem <- as.numeric(substr(asem,1,2)) * 60 * 60 +
+            as.numeric(substr(asem,4,5)) * 60 +
+            as.numeric(substr(asem,7,8))
+   }
+   u <- matrix(0, n, 1) # inicia o vetor de resultados
+   amostra <- gnu0 (sem) #chama gnu0
+   u[1] <- amostra$ran0 # inicia o primeiro elemento
+   for (i in 2:n)
+   {
+     amostra <- gnu0(amostra$sem)
+     u[i] <- amostra$ran0
+   }
+   return(u)
+ } # função
> #uso da função para gerar o vetor X de n números uniformes 0, 1
> n <- 5
> x <- gna0(n,0)
> x

          [,1]
[1,] 0.4862915
[2,] 0.1004663
[3,] 0.5366946
[4,] 0.2259471
[5,] 0.4922966
```

Devemos apresentar algumas considerações a respeito desse algoritmo escrito em linguagem R: a) a função gna0 (gerador de números aleatórios mínima) retorna um número real entre 0 e 1. Este tipo de especificação determina que as variáveis possuam precisão dupla. A precisão dupla (*double*) possui números na faixa de $5{,}0 \times 10^{-324} \therefore 1{,}7 \times 10^{308}$, ocupa 8 bytes de memória e possui $15-16$ dígitos significantes; b) o valor da semente é definido pelo usuário e é declarado na função como parâmetro de referência. Isso significa que a variável do programa que chama a função e que é passada como semente deve ser atualizada com o novo valor modificado em "gna0". Se o seu valor inicial for zero ou negativo, a função atribui um inteiro dependente da hora do sistema no momento da chamada; c) o sinal "#" indica comentário no

R; d) o comando "> *list(u=amostra$ran0, sem=amostra$sem)* " passa o valor de amostra para o resultado da função em uma lista contendo o novo valor da semente e o número aleatório $ran0$ gerado; e) a função tem dependência do pacote *R.utils*, que por sua vez depende dos pacotes *R.methodsS3* e *R.oo*. Essa dependência se caracteriza por usar uma função para capturar a hora do sistema.

A rotina é iniciada com os valores de n e da semente fornecidos pelo usuário. Se a semente for nula ou negativa, atribuímos um novo valor dependente do relógio do sistema no momento da chamada usando funções do R para isso. A partir deste ponto o programa deve chamar reiteradas vezes a função "gna0", que retorna o valor do número aleatório 0 e 1 utilizando o algoritmo descrito anteriormente, até que a sequência requerida pelo usuário seja completada. Nas sucessivas chamadas desta função, o valor da semente é sempre igual ao valor do último passo, uma vez que "sem" da última chamada deve ser passada como parâmetro de referência para a função "gna0" a partir do bloco do programa que a chamou.

O período de "gna0" é da ordem de $2^{31} \approx 2{,}15 \times 10^9$, ou seja, a sequência completa é superior a 2 bilhões de números aleatórios. Assim, podemos utilizar "gna0" para a maioria dos propósitos práticos. Evitamos o valor zero na função "gna0" devido ao fato de todos os valores da sequência serem nulos para um valor inicial de semente igual a zero. No entanto, para sementes positivas nunca ocorrerá o valor 0 na geração da sequência.

Como já salientamos o gerador padrão mínimo de números aleatórios possui duas limitações básicas, quais sejam, sequência curta e correlação serial. Assim, como existem métodos para eliminar a correlação serial e que aumentam o período da sequência, recomendamos que sejam adotados. Claro que a função apresentada teve por objetivo ilustrar como podemos programar nossas próprias funções para gerarmos números aleatórios uniformes. O R, no entanto, possui seu próprio gerador de números uniformes, que veremos na sequência. Um dos melhores e mais interessantes geradores de números aleatórios é o *Mersenne Twister* (MT). Mersenne Twister é um gerador de números pseudo-aleatórios desenvolvido por Makoto Matsumoto e Takuji Nishimura nos anos de 1996 e 1997 [9]. MT possui os seguintes méritos segundo seus desenvolvedores:

1. foi desenvolvido para eliminar as falhas dos diferentes geradores existentes;

2. possui a vantagem de apresentar o maior período e maior ordem de equi-

distribuição do que de qualquer outro método implementado. Ele fornece um período que é da ordem de $2^{19.937}-1 \approx 4{,}3154 \times 10^{6001}$, e uma equidistribuição 623-dimensional;

3. é um dos mais rápido geradores existentes, embora complexo;

4. faz uso de forma muito eficiente da memória.

Existem muitas versões implementadas deste algoritmo, inclusive em Fortran e C e que estão disponíveis na internet. Como o R nos possibilita incorporar funções escritas nestas linguagem, podemos utilizar este algoritmo. Felizmente, o R já possui este algoritmo implementado, sendo inclusive usado como *default*. Por se tratar de um tópico mais avançado, que vai além do que pretendemos apresentar nestas notas de aulas, não descreveremos este tipo de procedimento para incorporações de funções escritas em outras linguagens.

2.2 Números Aleatórios Uniformes no R

No SAS o comando usado para gerarmos números uniformes aleatórios é "Ranuni(semente)". Neste caso devemos usar o argumento "semente" para especificar uma sequência reproduzível de números aleatórios, sendo que seu valor deve ser um inteiro. Se o valor 0 for especificado como argumento, o programa escolherá automaticamente um valor para semente que depende da hora e da data do sistema. No programa R podemos gerar números aleatórios uniformes contínuos utilizando uma função pré-programada. Os números aleatórios uniformes são gerados pelo comando "runif(n, min, max)", em que n é o tamanho da sequência, "min" e "max" são argumentos que delimitam o valor mínimo e máximo da sequência a ser gerada. O controle da semente para se gerar uma sequência reproduzível de números uniformes é dada pelo comando "set.seed(semente)", onde o argumento "semente" deve ser um número inteiro. O R automaticamente determina a cada chamada uma nova semente. Conseguimos gerar diferentes sequências em cada chamada do comando "runif(n, min, max)", sem nos preocuparmos com a semente aleatória. Devemos reiterar que o programa R utiliza por *default* o algoritmo *Mersenne Twister*.

No programa apresentado a seguir ilustramos como podemos gerar n números aleatórios uniformes entre 0 e 1 utilizando a função *runif* do R. O programa resultante é compacto, simples e eficiente.

```
> # programa R demonstrando o uso de geração de n números aleatórios uniformes
> # por meio do comando runif
> n <- 5      # determina o tamanho do vetor aleatório
> x <- runif(n, 0, 1) # gera o vetor uniforme
> x # imprime o vetor x

[1] 0.3481472 0.6071912 0.7755748 0.5074987 0.2698393
```

Fizemos um programa para comparar o tempo de execução das funções *gna0* e *runif* e observamos que o tempo médio para cada variável gerada no *gna0* é de $2{,}9 \times 10^{-5}$ e no *runif* é de $1{,}7 \times 10^{-7}$. Desta forma verificamos que o algoritmo *runif* é aproximadamente 170 vezes mais rápido do que *gna0*. Obviamente temos que considerar que o algoritmo *runif* foi implementado em outra linguagem e a função compilada é que está associada ao R. O algoritmo *gna0* por sua vez foi implementado em linguagem R, que é interpretada. Assim, naturalmente o algoritmo *runif* deveria ser mais rápido do que qualquer outro algoritmo feito em linguagem interpretada R. Para obtermos este resultado utilizamos um Intel Core 2 CPU, U7700, 1,33 GHz (centrino) e 2 GB de memória RAM. Foram gerados 1.000.000 de números aleatórios do algoritmo *gna0* e 10.000.000 do algoritmo *runif*. O trecho do programa utilizado foi:

```
> # programa R demonstrando a comparação de tempos dos algoritmos
> # de geração de n números aleatórios uniformes
> # a função gna0 deve ser carregada previamente
> # tempo gna0
> n          <- 1000000
> ini        <- proc.time()
> x          <- gna0(n,0)
> fim        <- proc.time()
> tempo1.seg <- (fim-ini)/n
> tempo1.seg

     user    system   elapsed
5.191e-05 6.000e-08 5.307e-05

> # tempo de runif
> n          <- 10000000
> ini        <- proc.time()
> x          <- runif(n,0,1)
> fim        <- proc.time()
> tempo2.seg <- (fim - ini)/n
> tempo2.seg
```

```
   user   system  elapsed
2.81e-07 6.00e-09 2.92e-07

> tempo1.seg / tempo2.seg

   user   system  elapsed
184.7331  10.0000 181.7466
```

2.3 Exercícios

2.3.1 Utilizar o gerador *gna0* para gerar 10 números de uma distribuição exponencial $f(x) = \lambda e^{-\lambda x}$. Sabemos do teorema da transformação de probabilidades, que se U tem distribuição uniforme, $X = F^{-1}(U)$ tem distribuição de probabilidade com densidade $f(x) = F'(x)$; em que $F(x) = \int_{-\infty}^{x} f(t)dt$ é a função de distribuição de X e $F^{-1}(y)$ é a sua função inversa para o valor y. Para a exponencial a função de distribuição de probabilidade é: $F(x) = \int_0^x \lambda e^{-\lambda t}dt = 1 - e^{-\lambda x}$. Para obtermos a função inversa temos que igualar u a $F(x)$ e resolver para x. Assim, $u = 1 - e^{-\lambda x}$ e resolvendo para x temos: $x = -\ln(1-u)/\lambda$. Devido à simetria da distribuição uniforme $1 - u$ pode ser trocado por u. O resultado final é: $x = -\ln(u)/\lambda$. Para gerar números da exponencial basta gerar números uniformes e aplicar a relação $x = -\ln(u)/\lambda$. Fazer isso para gerar os 10 números exponenciais solicitados.

2.3.2 Para gerar números de uma distribuição normal, cuja densidade é dada por $f(x) = 1/(\sqrt{2\pi\sigma^2})\exp\{-(x-\mu)^2/(2\sigma^2)\}$, qual seria a dificuldade para podermos utilizar o teorema anunciado no exercício 2.3.1?

2.3.3 Como poderíamos adaptar o algoritmo apresentados nesse capítulo para gerar números aleatórios uniformes utilizando os valores propostos por Park e Miller, ou seja, $a = 48.271$ e $m = 2^{31} - 1$? Implementar o algoritmo, tomando cuidado em relação aos novos multiplicador q e resto r da fatoração de m?

2.3.4 Comparar a velocidade de processamento das funções apresentadas (*runif* e *rna0*) nesse capítulo para gerar 10.000.000 de números aleatórios uniformes entre 0 e 1. Como você poderia propor um teste estatístico simples para avaliar a aleatoriedade da sequência de números uniformes gerados por esses algoritmos? Implementar sua ideia.

Capítulo 3

Geração de Variáveis Não-Uniformes

Neste capítulo vamos apresentar alguns métodos gerais para gerarmos variáveis aleatórias de outras distribuições de probabilidade, como, por exemplo, dos modelos exponencial, normal e binomial. Implementaremos algumas funções em linguagem R e finalizaremos com a apresentação das rotinas otimizadas e já implementadas no R.

3.1 Introdução

Vamos estudar a partir deste instante um dos principais métodos, determinado pela lei fundamental de transformação de probabilidades, para gerarmos dados de distribuições de probabilidades contínuas ou discretas. Para alguns casos específicos, vamos ilustrar com procedimentos alternativos, que sejam eficientes e computacionalmente mais simples. Esta transformação tem por base o modelo uniforme 0, 1. Por essa razão a geração de números uniformes é tão importante.

Veremos posteriormente nestas notas de aulas algoritmos para obtermos numericamente a função de distribuição $F(x)$ e a sua função inversa $x = F^{-1}(p)$, em que p pertence ao intervalo que vai de 0 a 1. Este conhecimento é fundamental para a utilização deste principal método.

Neste capítulo limitaremos a apresentar a teoria para alguns poucos modelos probabilísticos, para os quais podemos facilmente obter a função de distribuição de

probabilidade e a sua inversa analiticamente. Para os modelos mais complexos, embora o método determinado pela lei fundamental de transformação de probabilidades seja adequado, apresentaremos apenas métodos alternativos, uma vez que este, em geral, é pouco eficiente em relação ao tempo gasto para gerarmos cada variável aleatória. Isso se deve ao fato de termos que obter a função inversa numericamente da função de distribuição de probabilidade dos modelos probabilísticos mais complexos.

3.2 Métodos Gerais para Gerar Variáveis Aleatórias

Podemos obter variáveis aleatórias de qualquer distribuição de probabilidade a partir de números aleatórios uniformes. Para isso um importante teorema pode ser utilizado: o teorema fundamental da transformação de probabilidades.

Teorema 3.1 (Teorema fundamental da transformação de probabilidades)**.** *Sejam U uma variável uniforme $U(0,1)$ e X uma variável aleatória com densidade f e função de distribuição F contínua e invertível, então $X = F^{-1}(U)$ possui densidade f. Sendo F^{-1} a função inversa da função de distribuição F.*

Demonstração: Seja X uma variável aleatória com função de distribuição F e função de densidade f. Se $u = F(x)$, então o jacobiano da transformação é $du/dx = F'(x) = f(x)$. Assim, a variável aleatória $X = F^{-1}(U)$ tem densidade f dada por:

$$f_X(x) = g(u)\left|\frac{du}{dx}\right| = g\left[F_X(x)\right] f(x) = f(x), \quad 0 < u < 1$$

e $g(u) = 0$ para outros valores de u. Em outras palavras a variável aleatória $X = F^{-1}(U)$ possui função densidade $f_X(x)$, estabelecendo o resultado almejado e assim, a prova fica completa. ■

Para variáveis aleatórias discretas, devemos modificar o teorema para podermos contemplar funções de distribuições F em escada, como são as funções de distribuição de probabilidades associadas a essas variáveis aleatórias.

Na Figura 3.1 representamos como podemos gerar uma variável aleatória X com densidade f e função de distribuição F. Assim, basta gerarmos um número uniforme u_0 e invertermos a função de distribuição F neste ponto. Computacionalmente a dificuldade é obtermos analiticamente uma expressão para a função F^{-1} para muitos modelos probabilísticos. Em geral, essas expressões não existem e métodos numéricos são requeridos para inverter a função de distribuição. Neste capítulo vamos

apresentar este método para a distribuição exponencial e nos próximos capítulos estudaremos os métodos numéricos de obtenção das funções de distribuições de inúmeros modelos probabilísticos.

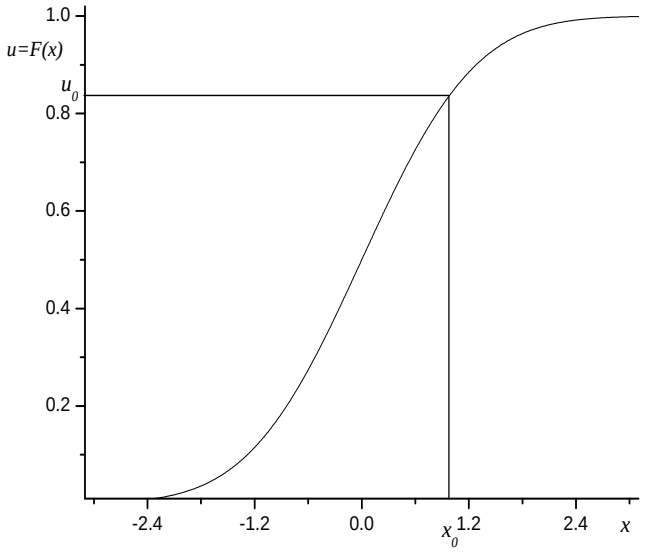


Figura 3.1: Ilustração do teorema fundamental da transformação de probabilidades para gerar uma variável aleatória X com densidade $f(x) = F'(x)$. A partir de um número aleatório uniforme u_0 a função de distribuição é invertida neste ponto para se obter x_0, com densidade $f(x)$.

Um outro método bastante geral que utilizaremos é denominado de método da amostragem por rejeição. Esse método tem um forte apelo geométrico. Procuraremos, a princípio, descrever esse método de uma forma bastante geral. Posteriormente, aplicaremos este método para gerarmos variáveis aleatórias de alguns modelos probabilístico. A grande vantagem deste método contempla o fato de não precisarmos obter a função de distribuição de probabilidade e nem a sua inversa. Estas estratégias só podem ser aplicadas em muitos dos modelos probabilísticos existentes, se utilizarmos métodos numéricos iterativos. Seja $f(x)$ a densidade para a qual queremos gerar uma amostra aleatória. A área sob a curva para um intervalo qualquer de x corresponde à probabilidade de gerar um valor x nesse intervalo. Se pudéssemos gerar um ponto em duas dimensões, digamos (X,Y), com distribuição

uniforme sob a área, então a coordenada X teria a distribuição desejada.

Para realizarmos de uma forma eficiente a geração de variáveis aleatórias com densidade $f(x)$, evitando as complicações numéricas mencionadas anteriormente, poderíamos definir uma função qualquer $g(x)$. Essa função tem que ter algumas propriedades especiais para sua especificação. Deve possuir área finita e ter para todos os valores x densidade $g(x)$ superior a $f(x)$. Essa função é denominada de função de comparação. Outra característica importante que $g(x)$ deve ter é possuir função de distribuição $G(x)$ analiticamente computável e invertível, ou seja, $x = G^{-1}(u)$. Como a função $g(x)$ não é necessariamente uma densidade, vamos denominar a área sob essa curva no intervalo para x de interesse por $A = \int_{-\infty}^{\infty} g(x)dx$. Como G^{-1} é conhecida, podemos gerar pontos uniformes (x,y) que pertencem à área sob a curva $g(x)$ facilmente. Para isso basta gerarmos um valor de uma variável aleatória uniforme u_1 entre 0 e A e aplicarmos o teorema (3.1). Assim, obtemos o primeiro valor do ponto (x_0,y_0) por $x_0 = G^{-1}(u_1)$. Para gerarmos a segunda coordenada do ponto não podemos gerar um valor de uma variável aleatória uniforme no intervalo de 0 a A, sob pena de gerarmos um ponto que não está sob a curva $g(x)$. Assim, calculamos o valor de g no ponto x_0 por $g(x_0)$. Geramos $y_0 = u_2$, sendo u_2 o valor de uma variável aleatória uniforme entre 0 e $g(x_0)$. Assim, obtemos um ponto (x_0,y_0) uniforme sob a curva $g(x)$. A dificuldade deste método é justamente estabelecer essa função $g(x)$ com as propriedades exigidas.

Vamos agora traçar as curvas correspondentes a $g(x)$ e $f(x)$ no mesmo gráfico. Se o ponto uniforme (x_0,y_0) está na área sob a curva $f(x)$, ou seja se $y_0 \leq f(x_0)$, então aceitamos x_0 como um valor válido de $f(x)$; se por outro lado o ponto estiver na região entre as densidades $f(x)$ e $g(x)$, ou seja se $f(x_0) < y_0 \leq g(x_0)$, então rejeitamos x_0. Uma forma alternativa de apresentarmos esse critério é tomarmos y_0 de uma distribuição $U(0,1)$ e aceitarmos ou rejeitarmos x_0 se $y_0 \leq f(x_0)/g(x_0)$ ou se $y_0 > f(x_0)/g(x_0)$, respectivamente. Ilustramos esse método na Figura (3.2), sendo que a A representa a área total sob a curva $g(x)$.

Vamos ilustrar o primeiro método e salientar que o segundo método é o que ocorre na maioria dos casos de geração de variáveis aleatórias. A exponencial é uma distribuição de probabilidade em que facilmente podemos aplicar o teorema 3.1 para gerarmos amostras aleatórias. Assim, optamos por iniciar o processo de geração de números aleatórios nesta distribuição, uma vez que facilmente podemos obter a função de distribuição e a sua inversa. Seja X uma variável aleatória cuja densidade

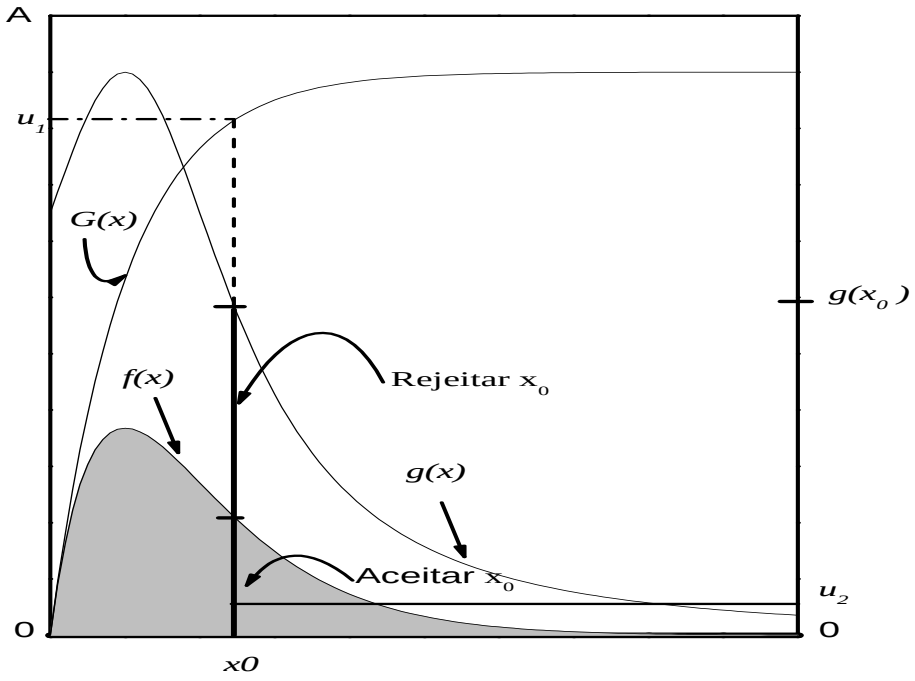


Figura 3.2: Método da rejeição para gerar um valor x_0 da variável aleatória X com densidade $f(x)$ que é menor do que $g(x)$ para todo x.

apresentamos por:

$$f(x) = \lambda e^{-\lambda x} \tag{3.2.1}$$

em que $\lambda > 0$ é o parâmetro da distribuição exponencial e $x > 0$.

A função de distribuição exponencial é dada por:

$$F(x) = \int_0^x \lambda e^{-\lambda t} dt$$

$$F(x) = 1 - e^{-\lambda x}. \tag{3.2.2}$$

E a função de distribuição inversa $x = F^{-1}(u)$ é dada por:

$$x = F^{-1}(u) = \frac{-\ln(1-u)}{\lambda} \tag{3.2.3}$$

em que u é um número uniforme $(0, 1)$.

Devido à distribuição uniforme ser simétrica, podemos substituir $1-u$ na equação (3.2.3) por u. Assim, para gerarmos uma variável exponencial X, a partir de uma variável aleatória uniforme, utilizamos o teorema 3.1 por intermédio da equação:

$$x = \frac{-\ln(u)}{\lambda} \tag{3.2.4}$$

O algoritmo R para gerarmos variáveis aleatórias exponenciais é dado por:

```
> # programa R demonstrando a geração de n realizações de variáveis
> # aleatórias exponenciais com parâmetro lamb, utilizamos
> # o algoritmo \textit{runif} para
> # gerarmos números aleatórios uniformes
> rexpon <- function(n, lamb)
+ {
+   u <- runif(n, 0, 1)  # gera vetor u (U(0,1))
+   x <- -log(u) / lamb  # gera vetor x com distrib. exp.
+   return(x)            # retorna o vetor x
+ }
> # exemplo
> rexpon(5, 1)

[1] 1.4447841 0.7248243 1.0561786 0.3940982 1.5119558
```

Podemos utilizar a função pré-existente do R para realizarmos a mesma tarefa denominada *rexp*. Simplesmente digitamos *rexp(n,lamb)* e teremos uma amostra aleatória de tamanho n de uma exponencial com parâmetro $\lambda < -lamb$. O R faz com que a estatística computacional seja ainda menos penosa e portanto acessível para a maioria dos pesquisadores.

Um método muito utilizado em matemática, física ou estatística para gerarmos dados de uma densidade para a qual é difícil obter amostras diretamente é o Metropolis-Hastings. Este algoritmo é um caso particular do método da rejeição. Na inferência bayesiana é possível utilizar este método na simulação Monte Carlo em cadeias de *Markov* para aproximar distribuições, gerar o histograma, ou calcular integrais para obter, por exemplo, o valor esperado. Este algoritmo recebeu este nome em referência a Nicholas Metropolis e W.K. Hastings que apresentaram este algoritmo para uma situação particular, no primeiro caso, e para uma situação geral, no segundo.

Para gerarmos dados de uma densidade $f(x)$, precisamos apenas obter o valor da densidade no ponto x. Ao aplicarmos o algoritmo, iremos gerar uma cadeia de *Markov* em que a realização da variável aleatória no estágio $t+1$, (x_{t+1}), depende apenas da realização no estágio prévio t (x_t). O algoritmo usa uma função auxiliar, denominada função candidata, $q(x_*; x_t)$, que depende do estado corrente x_t, para gerar o novo valor x_*, candidato a ser o valor do próximo estágio x_{t+1}. Assim, se o valor x_* poderá ser aceito ou rejeitado de acordo com o seguinte critério:

$$\begin{cases} u < \min\left[1, \dfrac{f(x_*)q(x_t; x_*)}{f(x_t)q(x_*; x_t)}\right] & \text{então, } x_{t+1} = x_* \\ \\ u \geq \min\left[1, \dfrac{f(x_*)q(x_t; x_*)}{f(x_t)q(x_*; x_t)}\right] & \text{então, } x_{t+1} = x_t \end{cases} \tag{3.2.5}$$

A função de densidade candidata $q(x_*; x_t)$ deve ser escolhida a partir de algumas características específicas, como possuir o mesmo domínio da distribuição alvo, embora não haja necessidade de ser um "envelope", $q(x) > f(x)$, para todos os valores da variável aleatória, além de possuir função de distribuição e inversa da função de distribuição conhecidas ou algum método simples de gerar observações. Assim, a função candidata, para fins de ilustração, poderia ser a distribuição normal centrada no ponto x_t e com variância σ^2, ou seja,

$$q(x_*; x_t) \sim N(x_t, \sigma^2),$$

que neste caso representa o algoritmo denominado passeio aleatório.

Esta função candidata nos permitiria gerar realizações da variável aleatória em torno do estado atual x_t com variância σ^2. É natural admitirmos que este processo possa ser generalizado para dimensões maiores do que 1. Na sua forma original, o algoritmo Metropolis requeria que a função candidata fosse simétrica, ou seja, $q(x_*; x_t) = q(x_t; x_*)$, mas, a partir do trabalho de Hastings, esta restrição foi eliminada. Além do mais, é permitido que a função $q(x_*; x_t)$ não dependa de x_t, o que é denominado de algoritmo de cadeias de *Markov* independentes para o Metropolis-Hastings, que difere do exemplo anterior, onde se ilustrou o algoritmo passeio aleatório. Este algoritmo é muito eficiente se a escolha da função candidata for adequada, mas pode possuir fraco desempenho se a escolha for inadequada, o que requer algum tipo de conhecimento da $f(x)$. Recomenda-se que a taxa de aceitação esteja em

torno de 60%. Se o seu valor for muito alto, a amostra caminhará lentamente em torno do espaço e convergirá lentamente para $f(x)$; se por outro lado, a aceitação for muito baixa, a função candidata está gerando valores em uma região de densidade muito pequena. Na Figura 3.3 ilustramos o algoritmo Metropolis-Hastings, para um caso particular do passeio aleatório, onde a função de transição inicialmente está centrada em x_1 e, no passo seguinte, está centrada em x_2.

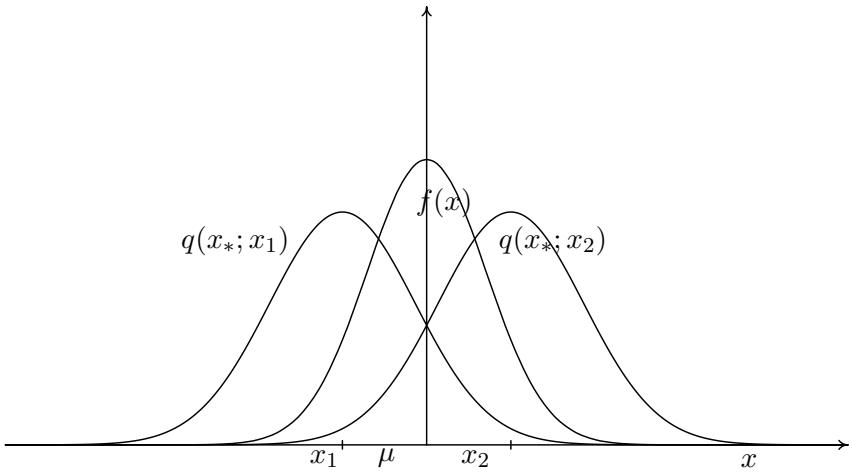


Figura 3.3: Ilustração gráfica do algoritmo Metropolis-Hastings, apresentando a função de transição $q(x_*; x_t)$ e a transição de x_1 para x_2 por meio do algoritmo de passeio aleatório.

No exemplo a seguir ilustramos a geração de realizações de variáveis normais com média μ e variância σ^2 utilizando o algoritmo Metropolis-Hastings. Denominamos o algoritmo de *MH_Norm* e utilizamos a função auxiliar *rtdf* para gerar realizações de variáveis t da distribuição t com média *media*, parâmetro de dispersão *sigma2* e graus de liberdade *df*. Da mesma forma utilizamos a função *dtdf* para calcularmos a densidade da distribuição t, com os parâmetros recém definidos. A função candidata escolhida foi a t e, é importante que se enfatize que utilizamos inclusive recursos redundantes, embora não entraremos neste mérito, uma vez que o objetivo é ilustrarmos o uso do algoritmo Metropolis-Hastings. Devemos também enfatizar que o R é capaz de gerar amostras aleatórias da normal diretamente por algoritmos mais eficiente que este. Novamente reiteramos, que escolhemos o modelo normal pela familiaridade que possuímos com este modelo e com seus resultados e para que houvesse uma maior facilidade de ilustrarmos as características do algoritmo. Neste

caso iniciamos com o valor inicial igual ao parâmetro μ e o valor candidato da segunda realização geramos de uma t com média igual ao valor anterior (inicial no segundo passo) e parâmetro de dispersão igual a variância da normal σ^2. Escolhemos uma t com $\nu = 3$ graus de liberdade como função candidata. Em geral, como já dissemos, o ponto crítico na utilização deste algoritmo é justamente a determinação apropriada da função candidata.

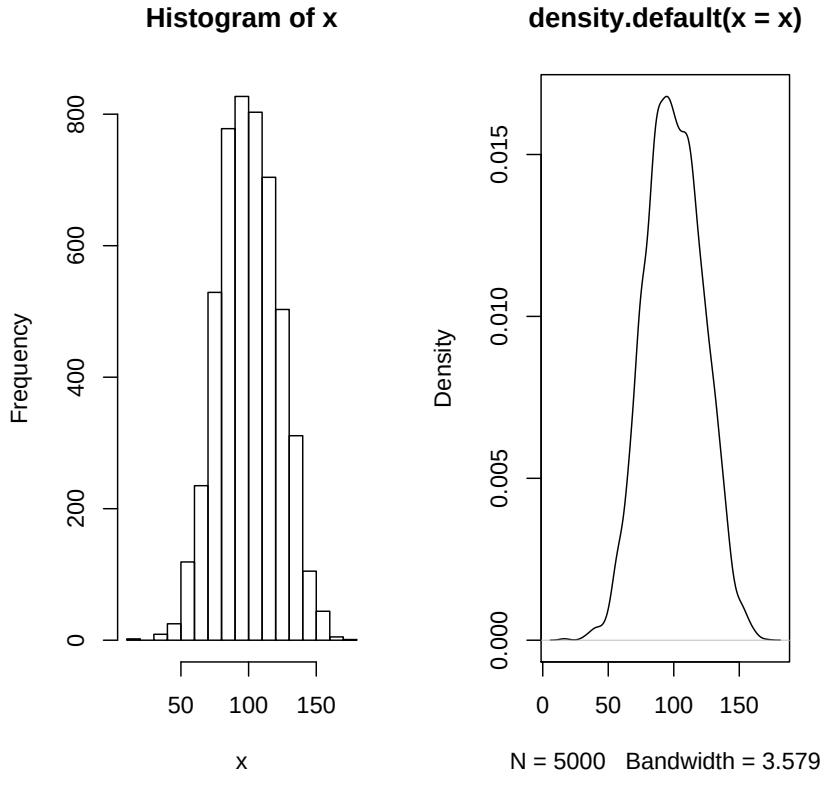
```
> # retorna uma v.a. de uma t univariada com df graus de liberdade, e
> # parâmetros media e sigma2
> rtdf <- function (n, media=0, sigma2 = 1, df = 1)
+ {
+    return( (rmvnorm(n, mean=c(0),sigma=as.matrix(sigma2,1,1))/sqrt(rchisq(n, df)/df))
+    + media)
+ }
> # retorna a densidade de uma t univariada com df graus de liberdade, e par media e sigma2
> dtdf <- function (x, media=0, sigma2 = 1, df = 1)
+ {
+     dt <- gamma((df+1)/2) / ((pi*df)**0.5 * gamma(df)*sigma2**0.5) * (1 +
+           (x-media)**2 / (df*sigma2))**(-(df+1)/2)
+ }
> # Função para exemplificar o uso do Metropolis-Hastings
> # Este exemplo pretende mostrar a geração de normais (mu=0, sig=1) com
> # uso de uma candidata t(nu=3).
> # Vamos utilizar rtdf(n,media,sigma2,df) para gerar dados da t.
> MH_Norm = function(n, mu=0, sig=1)
+ {
+   nu <- 3
+   x0 <- mu # valor inicial da cadeia
+   x  <- x0
+   for (i in 2:n)
+   {
+     y0  <- rtdf(1,x0,sig**2, nu) # valor candidato com random walk
+     qy0 <- dtdf(y0,x0,nu*sig**2/(nu-2),nu) # densidade da t no ponto y0
+     if (i>2) qx0 <- dtdf(x0,x[i-2],nu*sig**2/(nu-2),nu) #dens. da t no ponto x0
+     qx0 <- dtdf(x0,x[1], nu*sig**2/(nu-2), nu) # densidade da t no ponto x0
+     px0 <- dnorm(x0,mu,sig) # densidade da normal no ponto x0
+     py0 <- dnorm(y0,mu,sig) # densidade da normal no ponto y0
+     axy <- min(1,qx0*py0/(px0*qy0))
+     u   <- runif(1) # número uniforme para verificar se aceita ou descarta
+     if (u <= axy)
+     {
```

```
+       x <- rbind(x, y0)
+       x0 <- y0
+     } else x <- rbind(x, x0)
+   }
+   return(x)
+ }
> library(mvtnorm)
> n <- 5000; mu <- 100;sig <- 20
> x <- MH_Norm(n, mu, sig)
> par(mfrow = c(1, 2))
> hist(x)
> plot(density(x))
> var(x)

         [,1]
[1,] 442.5977

> mean(x)

[1] 99.6275
```



3.3 Variáveis Aleatórias de Algumas Distribuições Importantes

Vamos descrever nesta seção alguns métodos específicos para gerarmos algumas variáveis aleatórias. Vamos enfatizar a distribuição normal. Apesar de o mesmo método apresentado na seção 3.2 poder ser usado para a distribuição normal, daremos ênfase a outros processos. Para utilizarmos o mesmo método anterior teríamos que implementar uma função parecida com rexpon, digamos "rnormal". No lugar do comando "x = -log(u)/lamb" deveríamos escrever "x= invnorm(u)*$\sigma + \mu$", sendo que "invnorm(u)" é a função de distribuição normal padrão inversa. A distribuição normal padrão dentro da família normal, definida pelos seus parâmetros μ e σ^2, é aquela com média nula e variância unitária. Esta densidade será referenciada por $N(0,1)$. Essa deve ser uma função externa escrita pelo usuário. Os argumentos μ e σ da função são a média e o desvio padrão da distribuição normal que pretendemos gerar. A função de densidade da normal é:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.3.1}$$

Nenhuma outra função utilizando o teorema 3.1 será novamente apresentada, uma vez que podemos facilmente adaptar as funções "rexpon" se tivermos um eficiente algoritmo de inversão da função de distribuição do modelo probabilístico alvo. A dificuldade deste método é a necessidade de uma enorme quantidade de cálculo para a maioria das densidades. Isso pode tornar ineficiente o algoritmo, pois o tempo de processamento é elevado.

Podemos ainda aproveitar a relação entre algumas funções de distribuições para gerarmos variáveis aleatórias de outras distribuições. Por exemplo se X é normal com densidade (3.3.1), $N(\mu,\sigma^2)$, podemos gerar $Y = e^X$. Sabemos que fazendo tal transformação Y terá distribuição log-normal, cuja densidade com parâmetros de locação α (μ) e escala β (σ) é:

$$f(y) = \frac{1}{y\beta\sqrt{2\pi}} e^{-\frac{1}{2}\left[\frac{\ln(y)-\alpha}{\beta}\right]^2} \qquad y > 0. \tag{3.3.2}$$

Um importante método usado para gerar dados da distribuição normal é o de Box-Müller, que é baseado na generalização do método da transformação de pro-

babilidades para mais de uma dimensão. Para apresentarmos esse método, vamos considerar p variáveis aleatórias X_1, $X_2, \ldots,$ X_p com função de densidade conjunta $f(x_1,x_2,\ldots,x_p)$ e p variáveis Y_1, $Y_2, \ldots,$ Y_p, funções de todos os X's, então a função de densidade conjunta dos Y's é:

$$f(y_1,\ldots,y_p)dy_1\ldots dy_p = f(x_1,\ldots,x_p)\begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \cdots & \frac{\partial x_1}{\partial y_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_p}{\partial y_1} & \cdots & \frac{\partial x_p}{\partial y_p} \end{vmatrix} dx_1\ldots dx_p \tag{3.3.3}$$

em que $J = |\partial()/\partial()|$ é o Jacobiano da transformação dos X's em relação aos Y's.

Vamos considerar a transformação (3.3.3) para gerar dados Y normais com densidade dadas por (3.3.1), devidamente adaptada para acomodar Y e não X. Para aplicarmos a transformação de Box-Müller, vamos considerar duas variáveis aleatórias uniformes entre 0 e 1, representados por X_1 e X_2 e duas funções delas, representadas por Y_1 e Y_2 e dadas por:

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1}\cos(2\pi x_2) \\ y_2 &= \sqrt{-2\ln x_1}\sin(2\pi x_2) \end{aligned} \tag{3.3.4}$$

Ao explicitarmos X_1 e X_2 em (3.3.4) obtemos alternativamente:

$$\begin{aligned} x_1 &= e^{-\frac{1}{2}\left(y_1^2+y_2^2\right)} \\ x_2 &= \frac{1}{2\pi}\arctan\left(\frac{y_2}{y_1}\right) \end{aligned} \tag{3.3.5}$$

Sabendo que a seguinte derivada $dk\arctan(g)/dx$ é dada por $k(dg/dx)/(1+g^2)$, em que g é uma função de X, então o Jacobiano da transformação é:

$$\begin{aligned} \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} &= \begin{vmatrix} -y_1 e^{-0{,}5(y_1^2+y_2^2)} & -y_2 e^{-0{,}5(y_1^2+y_2^2)} \\ -\frac{y_2}{2\pi y_1^2\left(1+\frac{y_2^2}{y_1^2}\right)} & \frac{1}{2\pi y_1\left(1+\frac{y_2^2}{y_1^2}\right)} \end{vmatrix} \\ &= -\frac{1}{2\pi}e^{-0{,}5(y_1^2+y_2^2)} \end{aligned} \tag{3.3.6}$$

Assim, a função de densidade conjunta de Y_1 e Y_2 é dada por $f(x_1,x_2)|J|$, sendo portanto:

$$f(y_1,y_2) = \left[\frac{1}{\sqrt{2\pi}} e^{-\frac{y_1^2}{2}}\right]\left[\frac{1}{\sqrt{2\pi}} e^{-\frac{y_2^2}{2}}\right]. \tag{3.3.7}$$

Desde que a densidade conjunta de Y_1 e Y_2 é o produto de duas normais independentes, podemos afirmar que as duas variáveis geradas são normais padrão independentes, como pode ser visto em Johnson e Wichern (1998)[6] e Ferreira (2008)[5].

Assim, podemos usar esse resultado para gerarmos variáveis aleatórias normais. A dificuldade, no entanto, é apenas computacional. A utilização de funções trigonométricas como seno e co-seno pode limitar a performance do algoritmo gerado tornando-o lento. Um truque apresentado por Press et al. (1992)[13] é bastante interessante para evitarmos diretamente o uso de funções trigonométricas. Esse truque representa uma melhoria do algoritmo de Box-Müller e é devido a Marsaglia e Bray (1964)[8].

Ao invés de considerarmos os valores das variáveis aleatórias uniformes x_1 e x_2 de um quadrado de lado igual a 1 (quadrado unitário), tomarmos u_1 e u_2 como coordenadas de um ponto aleatório em um círculo unitário (de raio igual a 1). A soma de seus quadrados $R^2 = u_1^2 + u_2^2$ é um valor de uma variável aleatória uniforme que pode ser usada como x_1. Já o ângulo que o ponto (u_1, u_2) determina em relação ao eixo u_1 pode ser usado como um ângulo aleatório dado por $\theta = 2\pi x_2$. Podemos apontar que a vantagem da não utilização direta da expressão (3.3.4) refere-se ao fato do co-seno e do seno poderem ser obtidos alternativamente por: $\cos(2\pi x_2) = u_1/\sqrt{R^2}$ e $\sin(2\pi x_2) = u_2/\sqrt{R^2}$. Evitamos assim as chamadas de funções trigonométricas. Na Figura 3.4 ilustramos os conceitos apresentados e denominamos o ângulo que o ponto (u_1, u_2) determina em relação ao eixo u_1 por θ.

Agora podemos apresentar o função *BoxMuller* para gerar dados de uma normal utilizando o algoritmo de Box-Müller. Essa função utiliza a função *Polar* para gerar dois valores aleatórios, de variáveis aleatórias independentes normais padrão Y_1 e Y_2. A função *BoxMuller* é:

```
> # função BoxMüller retorna uma amostra de tamanho n de
> # uma distribuição normal com média mu e variância sigma^2
> # utilizando o método de Box-Müller modificado (polar)
> BoxMuller <- function(n, mu = 0, sigma = 1)
```

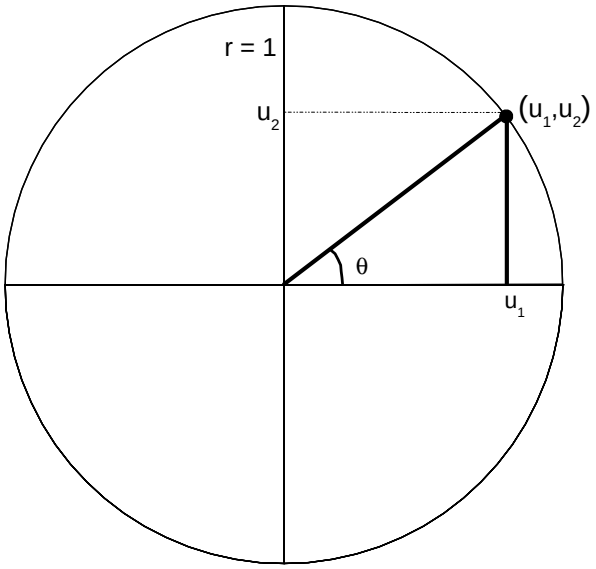


Figura 3.4: Círculo unitário mostrando um ponto aleatório (u_1, u_2) com $R^2 = u_1^2 + u_2^2$ representando x_1 e θ o ângulo que o ponto (u_1, u_2) determina em relação ao eixo u_1.

```
+ {
+   # Polar é a função que retorna dois números normais
+   # padrão em cada chamada do procedimento
+   Polar <- function() #função sem argumento
+   {
+     repeat
+     {
+       u  <- runif(2,-1,1) # gera dois números uniformes U(-1, 1)
+       R2 <- u%*%u # toma o seu quadrado
+       if ((R2 > 0) & (R2 < 1)) break
+     } # fim de repeat
+     ff <- sqrt(-2 * log(R2) / R2)
+     y  <- ff * u # vetor de dim. 2 com v.a. normais padrão indep.
+     return(y)
+   } # fim de Polar
+   if (n %% 2 == 0) # n é par
+   {
+     k <- n %/% 2 # divisão de inteiros
```

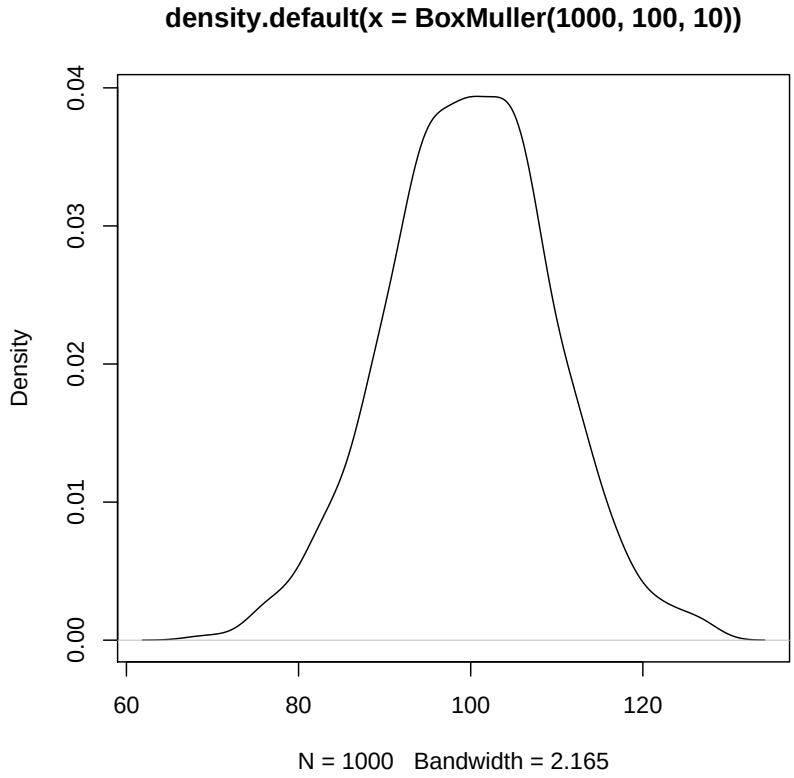
```
+     for (ki in 1:k)
+     {
+       if (ki == 1) x <- c(Polar()) else x <- c(x,Polar())
+     } # for
+   } else # n é ímpar
+   {
+     k <- n %/% 2
+     if (k == 0)
+     {
+       x <- Polar()[1]
+     } else
+     {
+       for (ki in 1:k)
+       {
+         if (ki == 1) x <- c(Polar()) else x <- c(x,Polar())
+       } # for
+       x <- c(x, Polar()[1])
+     } #else interno
+   } #else n par
+   x <- x * sigma + mu
+   return(x)
+ } # fim de BoxMüller
> # Exemplos de utilização
> x <- BoxMuller(12)
> x

 [1]  1.40937701  0.28741937 -0.48862146 -1.20826977
 [5] -0.69290503 -0.44234918  0.94446026  1.05152974
 [9]  0.04192309 -1.95650769 -0.58927306 -0.85146945

> y <- BoxMuller(12,100,10)
> y

 [1]  87.25362  84.14675  93.00117 102.22216  99.74829
 [6]  98.31755 111.57694 133.40115  88.57582  96.78552
[11]  86.89375 115.97198

> par(mfrow = c(1,1))
> plot(density(BoxMuller(1000,100,10)))
```



Algumas aproximações são apresentadas em Atkinson e Pearce (1976) [1] e serão apenas descritas na sequência. Uma das aproximações faz uso da densidade Tukey-lambda e aproxima a normal igualando os quatro primeiros momentos. Esse algoritmo, além de ser uma aproximação, tem a desvantagem de utilizar a exponenciação que é uma operação lenta. Utilizando essa aproximação podemos obter uma variável normal X a partir de uma variável uniforme $U \sim U(0,1)$ por:

$$X = [U^{0,135} - (1-U)^{0,135}]/0{,}1975.$$

Outro método é baseado na soma de 12 ou mais variáveis uniformes ($U_i \sim U(0,1)$) independentes. Assim, a variável $X = \sum_i^{12} U_i - 6$ tem distribuição aproximadamente normal com média 0 e variância 1. Isso ocorre em decorrência do teorema do limite central e em razão de cada uma das 12 variáveis uniformes possuírem média $1/2$ e variância $1/12$.

Estas duas aproximações tem valor apenas didático e não devem ser recomendadas como uma forma de gerar variáveis normais. Muitas vezes esse fato é ignorado

em problemas que requerem elevada precisão e confiabilidade dos resultados obtidos. Quando isso acontece conclusões incorretas ou no mínimo imprecisas podem ser obtidas.

Vamos ilustrar a partir deste instante a geração de variáveis aleatórias discretas. Escolhemos a distribuição binomial para exemplificar. A distribuição binomial é de forma surpreendente extremamente importante nas aplicações da estatística. Esta distribuição aparece nas mais variadas situações reais e teóricas. Por exemplo, o teste não-paramétrico do sinal utiliza a distribuição binomial, a ocorrência de animais doentes em uma amostra de tamanho n pode ser muitas vezes modelada pela distribuição binomial. Inúmeros outros exemplos poderiam ser citados. A distribuição binomial é a primeira distribuição discreta de probabilidade, entre as distribuições já estudadas. A variável aleatória X com distribuição de probabilidade binomial tem a seguinte função de probabilidade:

$$P(X = x) = \binom{n}{x} p^x (1-p)^{n-x}, \qquad x = 0,1,\cdots,n \quad \text{e} \quad n \geq 1, \tag{3.3.8}$$

em que os parâmetros n e p referem-se, respectivamente, ao tamanho da amostra e a probabilidade de sucesso de se obter um evento favorável em uma amostragem de 1 único elemento ao acaso da população. O termo $\binom{n}{x}$ é o coeficiente binomial definido por:

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}.$$

A probabilidade de sucesso p, em amostras de tamanho n da população, ou seja, em n ensaios de Bernoulli, deve permanecer constante e os sucessivos ensaios devem ser independentes. O parâmetro n em geral é determinado pelo pesquisador e deve ser um inteiro maior ou igual a 1. Se $n = 1$, então a distribuição binomial se especializa na distribuição Bernoulli. Assim, a distribuição binomial é a repetição de n ensaios Bernoulli independentes e com probabilidade de sucesso constante. Os seguintes teoremas e lemas são importantes, para a definição de alguns métodos que apareceram na sequência. Estes lemas serão apresentados sem as provas.

Teorema 3.2 (Gênese)**.** *Seja X o número de sucessos em uma sequência de n*

ensaios Bernoulli com probabilidade de sucesso p, ou seja,

$$X = \sum_{i=1}^{n} I(U_i \leq p)$$

em que U_1, U_2,$\cdots$, U_n *são variáveis uniformes* (0,1) *i.i.d. e* $I(\bullet)$ *é uma função indicadora. Então,* X *tem distribuição binomial* (n, p).

Lema 3.1 (Soma de binomiais)**.** *Se* X_1, X_2, $\cdots$, X_k *são variáveis aleatórias binomiais independentes com* (n_1, p), $\cdots$, (n_k, p), *então* $\sum_{i=1}^{k} X_i$ *tem distribuição binomial, com parâmetros* $(\sum_{i=1}^{k} n_i, p)$.

Lema 3.2 (Tempo de espera - propriedade 1)**.** *Sejam* G_1, G_2, $\cdots$ *variáveis aleatórias geométricas independentes e,* X, *o menor inteiro tal que*

$$\sum_{i=1}^{X+1} G_i > n.$$

Assim, X *tem distribuição binomial* (n, p).

As variáveis geométricas citadas no lema 3.2 são definidas a seguir. Se G tem distribuição geométrica com probabilidade de sucesso constante $p \in (0,1)$, então, a função de probabilidade é:

$$P(G = g) = p(1-p)^{g-1}, \qquad g = 1,2\cdots \tag{3.3.9}$$

A geométrica é a distribuição do tempo de espera até a ocorrência do primeiro sucesso no g-ésimo evento, numa sequência de ensaios Bernoulli independentes, incluindo o primeiro sucesso. Assim, supõe-se que venham a ocorrer $g-1$ fracassos, cada um com probabilidade de ocorrência constante $1-p$, antes da ocorrência de um sucesso no g-ésimo ensaio, com probabilidade p. Finalmente, o segundo lema do tempo de espera pode ser anunciado por:

Lema 3.3 (Tempo de espera - propriedade 2)**.** *Sejam* E_1, E_2, $\cdots$ *variáveis aleatórias exponenciais i.i.d., e* X *o menor inteiro tal que*

$$\sum_{i=1}^{X+1} \frac{E_i}{n-i+1} > -ln(1-p).$$

Logo, X *tem distribuição binomial* (n, p).

As propriedades especiais da distribuição binomial, descritas no teorema e nos lemas, formam a base para dois algoritmos binomiais. Estes dois algoritmos são baseados na propriedade de que uma variável aleatória binomial é á soma de n variáveis Bernoulli obtidas em ensaios independentes e com probabilidade de sucesso constante p. O algoritmo binomial mais básico baseia-se na geração de n variáveis independentes $U(0,1)$ e no cômputo do total das que são menores ou iguais a p. Este algoritmo denominado por Kachitvichyanukul e Schmeiser (1988) [7] de BU é dado por:

1. Faça $x = 0$ e $k = 0$
2. Gere u de uma $U(0,1)$ e faça $k = k + 1$
3. Se $u \leq p$, então faça $x = x + 1$
4. Se $k < n$ vá para o passo 2
5. Retorne x de uma binomial (n, p)

O algoritmo BU tem velocidade proporcional a n e depende da velocidade do gerador de números aleatórios, mas possui a vantagem de não necessitar de variáveis de *setup*. O segundo algoritmo atribuído a Devroy 1980 [3] é denominado de BG e é baseado no lema 3.2. O algoritmo BG pode ser descrito por:

1. Faça $y = 0$, $x = 0$ e $c = ln(1 - p)$
2. Se $c = 0$, vá para o passo 6
3. Gere u de uma $U(0,1)$
4. $y = y + \lfloor \ln(u)/c \rfloor + 1$, em que $\lfloor \bullet \rfloor$ denotam a parte inteira do argumento $\bullet$
5. Se $y \leq n$, faça $x = x + 1$ e vá para o passo 3
6. Retorne x de uma binomial (n, p)

O algoritmo utilizado no passo 4 do algoritmo BG é baseado em truncar uma variável exponencial para gerar uma variável geométrica, conforme descrição feita por Devroy (1986) [4]. O tempo de execução desse algoritmo é proporcional a np,

o que representa uma considerável melhoria da performance. Assim, $p > 0{,}5$, pode-se melhorar o tempo de execução de BG explorando a propriedade de que se X é binomial com parâmetro n e p, então $n - X$ é binomial com parâmetros n e $1 - p$. Especificamente o que devemos fazer é substituir p pelo $\min(p,1-p)$ e retornar x se $p \leq \frac{1}{2}$ ou retornar $n - x$, caso contrário. A velocidade, então, é proporcional a n vezes o valor $\min(p,1-p)$. A desvantagem desse procedimento é que são necessárias várias chamadas do gerador de variáveis aleatórias uniformes, até que um sucesso seja obtido e o valor x seja retornado. Uma alternativa a esse problema pode ser conseguida se utilizarmos um gerador baseado na inversão da função de distribuição binomial.

Como já havíamos comentado em outras oportunidades o método da inversão é o método básico para convertermos uma variável uniforme U em uma variável aleatória X, invertendo a função de distribuição. Para uma variável aleatória contínua, temos o seguinte procedimento:

- Gerar um número uniforme u
- Retornar $x = F^{-1}(u)$

O procedimento análogo para o caso discreto requer a busca do valor x, tal que:

$$F(x-1) = \sum_{i<x} P(X = i) < u \leq \sum_{i \leq x} P(X = i) = F(x).$$

A maneira mais simples de obtermos uma solução no caso discreto é realizarmos uma busca sequencial a partir da origem. Para o caso binomial, este algoritmo da inversão, denominado de BINV, pode ser implementado se utilizarmos a fórmula recursiva:

$$\begin{aligned} P(X = 0) &= (1-p)^n \\ P(X = x) &= P(X = x-1)\frac{n-x+1}{x}\frac{p}{1-p} \end{aligned} \tag{3.3.10}$$

para $x = 1{,}2, \cdots, n$ da seguinte forma:

1. Faça $pp = \min(p,1-p)$, $qq = 1 - pp$, $r = pp/qq$, $g = r(n+1)$ e $f = qq^n$

2. Gere u de uma $U(0,1)$ e faça $x = 0$

3. Se $u \leq r$, então vá para o passo 5

4. Faça $u = u - f$, $x = x + 1$, $f = f(\frac{g}{x} - r)$ e vá para o passo 3

5. Se $p \leq \frac{1}{2}$, então retorne x de uma binomial (n, p), senão retorne $n - x$ de uma binomial (n, p)

A velocidade deste algoritmo é proporcional a n vezes o valor $\min(p, 1-p)$. A vantagem desse algoritmo é que apenas uma variável aleatória uniforme é gerada para cada variável binomial requerida. Um ponto importante é o tempo consumido para gerar qq^n é substancial e dois problemas potenciais podem ser destacados. O primeiro é a possibilidade de *underflow* no cálculo de $f = qq^n$, quando n é muito grande e, o segundo, é a possibilidade do cálculo recursivo de f ser uma fonte de erros de arredondamento, que se acumulam e que se tornam sérios na medida que n aumenta (Kachitvichyanukul e Schmeiser, 1988 [7]). Devroy (1986) [4] menciona que o algoritmo do tempo de espera BG baseado no lema 3.2 deve ser usado no lugar de BINV para evitarmos esses fatos. Por outro lado, Kachitvichyanukul e Schmeiser (1988) [7] mencionam que basta implementar o algoritmo em precisão dupla que esses problemas são evitados.

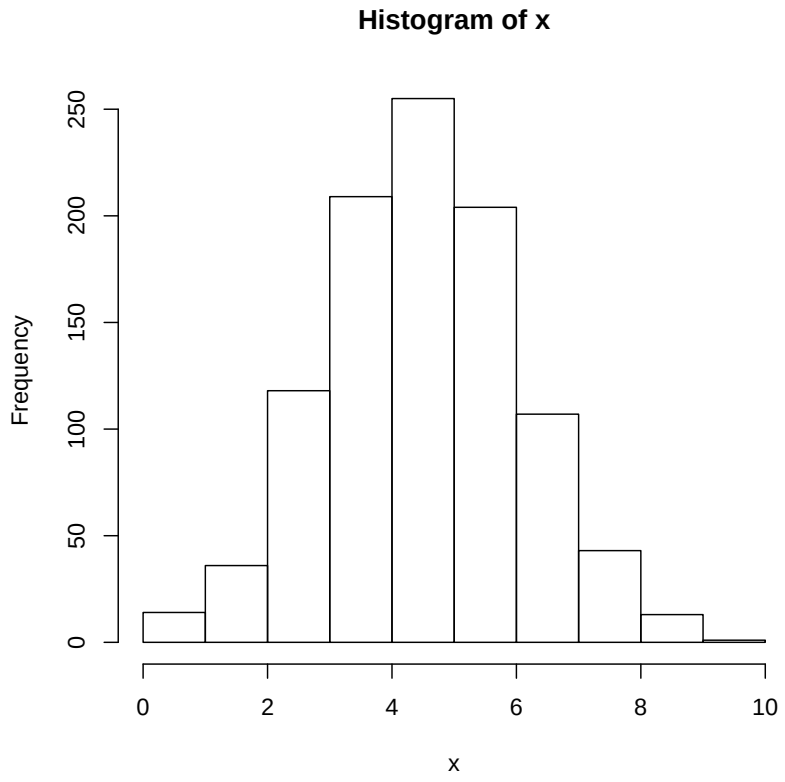
```
> # Exemplificação de algoritmos para gerar variáveis aleatórias binomiais B(n, p).
> # Os algoritmos BU, BG e BINV foram implementados
>
> BU <- function(n,p)
+ {
+   x <- 0; k <- 0
+   repeat
+   {
+     u <- runif(1,0,1)
+     k <- k + 1
+     if (u <= p) x <- x + 1
+     if (k == n) break
+   } # repeat
+   return(x)
+ } # function BU
> BG <- function(n,p)
+ {
+    if (p > 0.5) pp <- 1 - p else pp <- p
```

```
+    y <- 0; x <- 0; c <- log(1 - pp)
+    if (c < 0)
+    {
+      repeat
+      {
+        u <- runif(1, 0, 1)
+        y <- y + trunc(log(u) / c) + 1
+        if (y <= n)
+        {
+          x <- x + 1
+        } else break
+      } # repeat
+      if (p > 0.5) x <- n - x
+    } # if c <= 0
+    return(x)
+ } # function BG
> BINV <- function(n,p)
+ {
+   if (p > 0.5) pp <- 1 - p else pp <- p
+   q  <- 1 - pp
+   qn <- q**n # forma alternativa de obter potências
+   r  <- pp / q
+   g  <- r * (n + 1)
+   u  <- runif(1, 0, 1)
+   x  <- 0; f <- qn
+   while (u > f)
+   {
+     u <- u - f
+     x <- x + 1
+     f <- f * (g / x - r)
+   } # while
+   if (p > 0.5) x <- n - x
+   return(x)
+ } # function BINV
> # Exemplo de utilização da função BINV
> n <- 1000
> x <- BINV(10,0.5)
> for (i in 2:n) x <- c(x, BINV(10, 0.5))
> par(mfrow = c(1,1))
> hist(x)
> mean(x)
```

```
[1] 4.965

> var(x)

[1] 2.594369
```



Procedimentos de geração de números aleatórios poderiam ser apresentados para muitas outras distribuições de probabilidades. Felizmente no R não temos este tipo de preocupação, pois estas rotinas já existem e estão implementadas em linguagem não interpretada. Inclusive para os modelos considerados temos rotinas prontas em R. Veremos uma boa parte delas na próxima seção.

3.4 Rotinas R para Geração de Variáveis Aleatórias

Nesta seção, veremos alguns dos principais comandos R para acessarmos os processos de geração de realizações de variáveis aleatórias de diferentes modelos probabilísticos. Os modelos probabilísticos contemplados pelo R estão apresentados na Tabela 3.1.

Tabela 3.1: Distribuições de probabilidades, nome R e parâmetros dos principais modelos probabilístico.

Distribuição	Nome R	Parâmetros
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
quiquadrado	chisq	df, ncp
exponencial	exp	rate
F	f	df1, df1, ncp
gama	gamma	shape, scale
geométrica	geom	prob
hipergeométrica	hyper	m, n, k
log-normal	lnorm	meanlog, sdlog
logística	logis	location, scale
binomial negativa	nbinom	size, prob
normal	norm	mean, sd
Poisson	pois	lambda
t de Student	t	df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

Se acrescentarmos os prefixos 'd' para densidade, 'p' para *CDF* (função de distribuição de probabilidades), 'q' para quantis e 'r' para simulação (realizações de variáveis aleatórias), então poderemos utilizar as funções básicas da Tabela 3.1 em diferentes contextos. No caso particular deste capítulo temos interesse na geração de realizações de variáveis aleatórias e, portanto, no comando *rmodelo*. O primeiro argumento é x para dmodelo, q para pmodelo, p para qmodelo and n for rmodelo (exceto para rhyper e rwilcox, nos quais o primeiro argumento é nn). Estas exceções se devem ao fato de que n é usado como um argumento da função (parâmetro). Assim, nn representa o tamanho da amostra que queremos gerar. Em algum modelos probabilísticos não centrais existe a possibilidade de utilizar o parâmetro de não-centralidade (ncp).

O uso destas funções para gerarmos números aleatórios é bastante simples. Se,

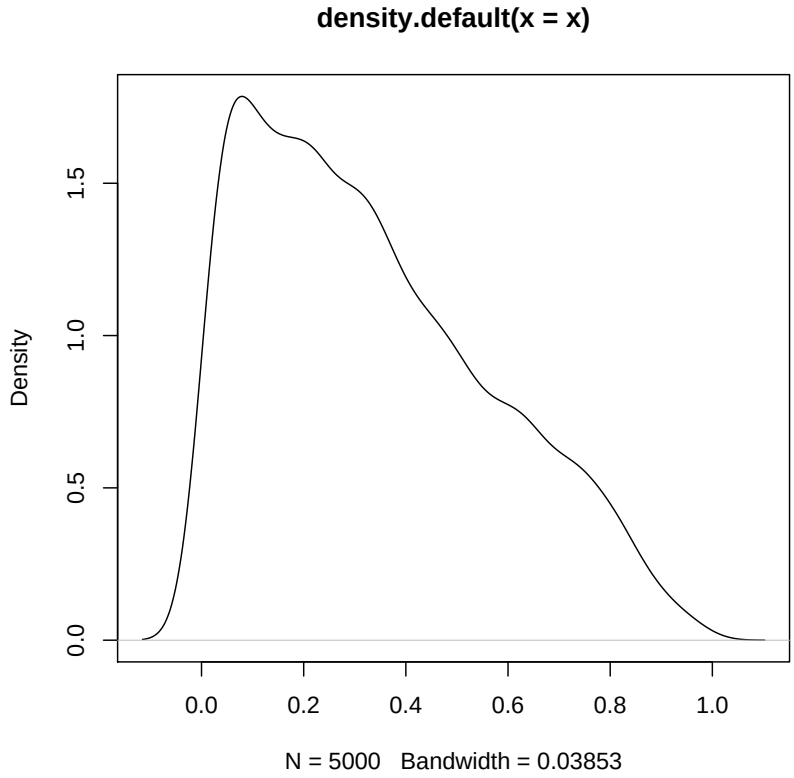
por exemplo, quisermos gerar dados de uma distribuição beta com parâmetros $\alpha = 1$ e $\beta = 2$, podemos utilizar o programa ilustrativo apresentado na sequência. Podemos utilizar funções semelhantes a função *beta*, de acordo com a descrição feita na Tabela 3.1, para gerarmos n dados de qualquer outra função de densidade ou função de probabilidade. Este procedimento é mais eficiente do que utilizarmos nossas próprias funções, pois estas funções foram implementadas em geral em C ou Fortran e são mais eficientes. Se para algum modelo particular desejarmos utilizar nossas próprias funções e se iremos chamá-las milhares ou milhões de vezes é conveniente que implementemos em C ou em Fortran e as associemos ao R. A forma de associarmos as rotinas escritas em C ou Fortran ao R foge do escopo deste material e por isso não explicaremos como fazê-lo.

```
> # Exemplo de algoritmos para gerar dados de
> # variáveis aleatórias Beta(alpha; beta)
> # usando as funções R diretamente.
> #Gera uma amostra de tamanho n e plota o histograma.
> n <- 5000
> alpha <- 1.0
> beta <- 2.0
> x <- rbeta(n, alpha, beta)
> plot(density(x))
> mean(x)
```

```
[1] 0.3327995
```

```
> var(x)
```

```
[1] 0.05410881
```



3.5 Exercícios

3.5.1 Seja $f(x) = 2x$ uma função de densidade de uma variável aleatória contínua X com domínio definido pelo intervalo $[0; 1]$. Aplicar o método da inversão e descrever um algoritmo para gerar variáveis aleatórias dessa densidade. Implementar em R e gerar uma amostra de tamanho $n = 1.000$. Estimar os quantis 1%, 5%, 10%, 50%, 90%, 95% e 99%. Confrontar com os quantis teóricos.

3.5.2 Os dados a seguir referem-se ao tempo de vida, em dias, de $n = 40$ insetos. Considerando que a distribuição do tempo de vida é a exponencial e que o parâmetro λ pode ser estimado pelo estimador de máxima verossimilhança $\hat{\lambda} = 1/\bar{X}$, em que $\bar{X} = \sum_{i=1}^{n} X_i/n$, obter o intervalo de 95% de confiança utilizando o seguinte procedimento: i) gerar uma amostra da exponencial de tamanho $n = 40$, utilizando o algoritmo *rexpon*, considerando o parâmetro igual a estimativa obtida; ii) determinar a estimativa da média $\mu = 1/\lambda$ por $\bar{X}$

nesta amostra simulada de tamanho $n = 40$; iii) repetir 1.000 vezes os passos (i) e (ii) e armazenar os valores obtidos; iv) ordenar as estimativas e tomar os quantis 2,5% e 97,5%. Os valores obtidos são o intervalo de confiança almejado, considerando como verdadeira a densidade exponencial para modelar o tempo de vida dos insetos. Este procedimento é denominado de bootstrap paramétrico. Os dados em dias do tempo de vida dos insetos são:

8,521	4,187	2,516	1,913	8,780	5,912	0,761	12,037
2,604	1,689	5,626	6,361	5,068	3,031	1,128	1,385
12,578	2,029	0,595	0,445	3,601	7,829	1,383	1,934
0,864	8,514	4,977	0,576	1,503	0,475	1,041	0,301
1,781	2,564	5,359	2,307	1,530	8,105	3,151	8,628

Repetir esse processo, gerando 100.000 amostras de tamanho $n = 40$. Compare os resultados e verifique se o custo adicional de ter aumentado o número de simulações compensou a possível maior precisão obtida.

3.5.3 Gerar uma amostra de $n = 5.000$ variáveis normais padrão utilizando as aproximações: $X = [U^{0,135} - (1 - U)^{0,135}]/\ 0{,}1975$ e da soma de 12 ou mais variáveis uniformes ($U_i \sim U(0,1)$) independentes, dada por $X = \sum_i^{12} U_i - 6$. Confrontar os quantis 1%, 5%, 10%, 50%, 90%, 95% e 99% esperados da distribuição normal com os estimados dessa distribuição. Gerar também uma amostra de mesmo tamanho utilizando o algoritmo Polar-Box-Müller. Estimar os mesmos quantis anteriores nesta amostra e comparar com os resultados anteriores.

3.5.4 Se os dados do exercício 3.5.2 pudessem ser atribuídos a uma amostra aleatória da distribuição log-normal, então estimar os parâmetros da log-normal e utilizar o mesmo procedimento descrito naquele exercício, substituindo apenas a distribuição exponencial pela log-normal para estimar por intervalo a média populacional. Para estimar os parâmetros da log-normal utilizar o seguinte procedimento: a) transformar os dados originais, utilizando $X_i^* = \ln(X_i)$; b) determinar a média e o desvio padrão amostral dos dados transformados - estas estimativas são as estimativas de μ e σ. Utilizar estas estimativas para gerar amostras log-lognormais. Confrontar os resultados e discutir a respeito da dificuldade de se tomar uma decisão da escolha da distribuição populacional no processo de inferência. Como você poderia fazer para determinar

qual a distribuição que melhor modela os dados do tempo de vida dos insetos? Justificar sua resposta adequadamente com os procedimentos numéricos escolhidos.

3.5.5 Fazer reamostragens com reposição na amostra do exercício 3.5.2 e estimar o intervalo de 95% para a média populacional, seguindo os passos descritos a seguir e utilizar um gerador de números uniformes para determinar quais elementos amostrais devem ser selecionados: i) reamostrar com reposição os $n = 40$ elementos da amostra original e compor uma nova amostra por: X_i^*; ii) calcular a média desta nova amostra por $\bar{X}^* = \sum_{i=1}^{n} X_i^*/n$; iii) armazenar este valor e repetir os passos (i) e (ii) B vezes; iv) ordenar os valores obtidos e determinar os quantis 2,5% e 97,5% desse conjunto de B valores. Escolher $B = 1.000$ e $B = 100.000$ e confrontar os resultados obtidos com os obtidos nos exercícios anteriores para a distribuição exponencial e log-normal. Os resultados que estiverem mais próximos deste resultado devem fornecer um indicativo da escolha da distribuição mais apropriada para modelar o tempo de vida de insetos. Este procedimento sugerido neste exercício é o bootstrap não-paramétrico.

3.5.6 Uma importante relação para obtermos intervalos de confiança para a média de uma distribuição exponencial ($f(x) = \lambda e^{-\lambda x}$), refere-se ao fato de que a soma de n variáveis exponenciais com parâmetro λ é igual a uma gama com parâmetros $\alpha = n$ e $\beta = 1/\lambda$ $\left(f(y) = \frac{1}{\lambda\Gamma(\alpha)} (y/\beta)^{\alpha-1} e^{-\frac{y}{\beta}}\right)$. Assim, assumir que os dados do exercício 3.5.2 têm distribuição exponencial com parâmetro λ estimado pelo recíproco da média amostral. Considerando que a variável X tem distribuição gama padrão com parâmetro $\alpha = n$, então obtenha $Y = \hat{\beta}X = X/\hat{\lambda}$. Neste caso Y tem distribuição da soma de n variáveis exponenciais, ou seja, distribuição gama. Como queremos a distribuição da média, devemos obter a transformação $\bar{Y} = Y/n$. Gerar amostras de tamanho $n = 1.000$ e $n = 100.000$ e estimar os quantis 2,5% e 97,5% da distribuição de $\bar{Y}$, em cada uma delas. Confrontar os intervalos de confiança, dessa forma obtidos, com os do exercício 3.5.2. Quais são suas conclusões? Qual é a vantagem de utilizar a distribuição gama?

3.5.7 Duas amostras binomiais foram realizadas em duas (1 e 2) diferentes popu-

lações. Os resultados do número de sucesso foram $y_1 = 20$ e $y_2 = 30$ em amostras de tamanho $n_1 = 120$ e $n_2 = 140$, respectivamente, de ambas as populações. Estimar os parâmetros p_1 e p_2 das duas populações por: $\hat{p_1} = y_1/n_1$ e $\hat{p_2} = y_2/n_2$. Para testarmos a hipótese $H_0 : p_1 = p_2$, podemos utilizar o seguinte algoritmo de bootstrap paramétrico: a) utilizar $\hat{p_1}$ e $\hat{p_2}$ para gerarmos amostras de tamanho n_1 e n_2 de ambas as populações; b) estimar $\hat{p_{1j}} = y_{1j}/n_1$ e $\hat{p_{2j}} = y_{2j}/n_2$ na j-ésima repetição desse processo; c) calcular $d_j = \hat{p_{1j}} - \hat{p_{2j}}$; d) repetir os passos de (a) a (c) B vezes; e) ordenar os valores e obter os quantis 2,5% e 97,5% da distribuição bootstrap de d_j; f) se o valor hipotético 0 estiver contido nesse intervalo, não rejeitar H_0, caso contrário, rejeitar a hipótese de igualdade das proporções binomiais das duas populações.

Capítulo 4

Geração de Amostras Aleatórias de Variáveis Multidimensionais

Os modelos multivariados ganharam grande aceitação no meio científico em função das facilidades computacionais e do desenvolvimento de programas especializados nesta área. Os fenômenos naturais são em geral multivariados. Um tratamento aplicado em um ser, a um solo ou a um sistema não afeta isoladamente apenas uma variável, e sim todas as variáveis. Ademais, as variáveis possuem relações entre si e qualquer mudança em uma ou algumas delas, afeta as outras. Assim, a geração de realizações de vetores ou matrizes aleatórias é um assunto que não pode ser ignorado. Vamos neste capítulo disponibilizar ao leitor mecanismos para gerar realizações de variáveis aleatórias multidimensionais.

4.1 Introdução

Os processos para gerarmos variáveis aleatórias multidimensionais são muitas vezes considerados difíceis pela maioria dos pesquisadores. Uma boa parte deles, no entanto, podem ser realizados no R com apenas uma linha de comando. Embora tenhamos estas facilidades, nestas notas vamos apresentar detalhes de alguns processos para gerarmos dados dos principais modelos probabilísticos multivariados como, por exemplo, a normal multivariada, a Wishart e a Wishart invertida, a t de Student e algumas outras distribuições.

Uma das principais características das variáveis multidimensionais é a correlação

entre seus componentes. A importância destes modelos é praticamente indescritível, mas podemos destacar a inferência paramétrica, a inferência Bayesiana, a estimação de regiões de confiança, entre outras. Vamos abordar nas próximos seções formas de gerarmos realizações de variáveis aleatórias multidimensionais para determinados modelos utilizando o ambiente R para implementarmos as rotinas ou para utilizarmos as rotinas pré-existentes. Nossa aparente perda de tempo, descrevendo funções menos eficientes do que as pré-existentes no R, tem como razão fundamental permitir ao leitor ir além de simplesmente utilizar rotinas previamente programadas por terceiros. Se o leitor ganhar, ao final da leitura deste material, a capacidade de produzir suas próprias rotinas e entender como as rotinas pré-existentes funcionam, nosso objetivo terá sido alcançado.

4.2 Distribuição Normal Multivariada

A função de densidade normal multivariada de um vetor aleatório $\boldsymbol{X}$ é dada por:

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = (2\pi)^{-\frac{p}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\} \tag{4.2.1}$$

em que $\boldsymbol{\mu}$ e $\boldsymbol{\Sigma}$ são, respectivamente, o vetor de média e a matriz de covariâncias, simétrica e positiva definida, e p é a dimensão do vetor aleatório $\boldsymbol{X}$.

Um importante resultado diz respeito a combinações lineares de variáveis normais multivariadas e será apresentado no seguinte teorema.

Teorema 4.1 (Combinações lineares)**.** *Seja o vetor aleatório normal multivariado* $\boldsymbol{X} = [X_1, X_2, \cdots, X_p]^\top$ *com média* $\boldsymbol{\mu}$ *e covariância* $\boldsymbol{\Sigma}$ *e seja* $\boldsymbol{C}$ *uma matriz* $(p \times p)$ *de posto* p*, então a combinação linear* $\boldsymbol{Y} = \boldsymbol{C}\boldsymbol{X}$ $(p \times 1)$ *tem distribuição normal multivariada com média* $\boldsymbol{\mu}_{\boldsymbol{Y}} = \boldsymbol{C}\boldsymbol{\mu}$ *e covariância* $\boldsymbol{\Sigma}_{\boldsymbol{Y}} = \boldsymbol{C}\boldsymbol{\Sigma}\boldsymbol{C}^\top$.

Prova: se $\boldsymbol{C}$ possui posto p, então existe $\boldsymbol{C}^{-1}$ e portanto

$$\boldsymbol{X} = \boldsymbol{C}^{-1}\boldsymbol{Y}.$$

O Jacobiano da transformação é $J = |\boldsymbol{C}|^{-1}$ e a distribuição de $\boldsymbol{Y}$ é dada por $f_{\boldsymbol{Y}}(\boldsymbol{y}) = f_{\boldsymbol{X}}(\boldsymbol{x})|J|$. Se $\boldsymbol{X}$ tem distribuição normal multivariada, então

$$
\begin{aligned}
f_{\boldsymbol{Y}}(\boldsymbol{y}) =& (2\pi)^{-p/2} \left|\boldsymbol{\Sigma}\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\boldsymbol{C}^{-1}\boldsymbol{y}-\boldsymbol{\mu})^{\top}\boldsymbol{\Sigma}^{-1}(\boldsymbol{C}^{-1}\boldsymbol{y}-\boldsymbol{\mu})\right\}|\boldsymbol{C}|^{-1} \\
=& (2\pi)^{-p/2}|\boldsymbol{C}|^{-1/2}\left|\boldsymbol{\Sigma}\right|^{-\frac{1}{2}}|\boldsymbol{C}|^{-1/2} \\
& \times \exp\left\{-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{C}\boldsymbol{\mu})^{\top}\left(\boldsymbol{C}^{\top}\right)^{-1}\boldsymbol{\Sigma}^{-1}\boldsymbol{C}^{-1}(\boldsymbol{y}-\boldsymbol{C}\boldsymbol{\mu})\right\} \\
=& (2\pi)^{-p/2}\left|\boldsymbol{C}\boldsymbol{\Sigma}\boldsymbol{C}^{\top}\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{C}\boldsymbol{\mu})^{\top}\left(\boldsymbol{C}\boldsymbol{\Sigma}\boldsymbol{C}^{\top}\right)^{-1}(\boldsymbol{y}-\boldsymbol{C}\boldsymbol{\mu})\right\}
\end{aligned}
$$

que é a densidade normal multivariada do vetor aleatório $\boldsymbol{Y}$ com média $\boldsymbol{\mu}_{\boldsymbol{Y}} = \boldsymbol{C}\boldsymbol{\mu}$ e covariância $\boldsymbol{\Sigma}_{\boldsymbol{Y}} = \boldsymbol{C}\boldsymbol{\Sigma}\boldsymbol{C}^{\top}$. Portanto combinações lineares de variáveis normais multivariadas são normais multivariadas. **C.Q.D.**

O teorema 4.1 nos fornece o principal resultado para gerarmos dados de uma normal multivariada. Assim, como nosso objetivo é gerar dados de uma amostra normal multivariada com vetor de médias $\boldsymbol{\mu}$ e matriz de covariâncias $\boldsymbol{\Sigma}$ pré-estabelecidos, devemos seguir os seguintes procedimentos. Inicialmente devemos obter a matriz raiz quadrada de $\boldsymbol{\Sigma}$, representada por $\boldsymbol{\Sigma}^{1/2}$. Para isso, vamos considerar a decomposição espectral da matriz de covariâncias dada por $\boldsymbol{\Sigma} = \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{\top}$. Logo, podemos definir a matriz raiz quadrada de $\boldsymbol{\Sigma}$ por $\boldsymbol{\Sigma}^{1/2} = \boldsymbol{P}\boldsymbol{\Lambda}^{1/2}\boldsymbol{P}^{\top}$, em que $\boldsymbol{\Lambda}$ é a matriz diagonal dos autovalores, $\boldsymbol{\Lambda}^{1/2}$ é a matriz diagonal contendo a raiz quadrada destes elementos e $\boldsymbol{P}$ é a matriz de autovetores, cada um destes vetor disposto em uma de suas colunas. Desta decomposição facilmente podemos observar que $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}^{1/2}\boldsymbol{\Sigma}^{1/2}$.

Assim, podemos utilizar o seguinte método para gerarmos uma realização p-variada de uma normal multivariada. Inicialmente devemos gerar um vetor aleatório $\boldsymbol{Z} = [Z_1, Z_2, \cdots, Z_p]^{\top}$ de p variáveis normais padrão independentes utilizando, por exemplo, o algoritmo de Box-Müller. Isto que dizer que $\boldsymbol{\mu}_{\boldsymbol{Z}} = \boldsymbol{0}$ e que $\text{Cov}(\boldsymbol{Z}) = \boldsymbol{I}$. Este vetor deve sofrer a seguinte transformação linear:

$$
\boldsymbol{Y} = \boldsymbol{\Sigma}^{1/2}\boldsymbol{Z} + \boldsymbol{\mu}. \tag{4.2.2}
$$

De acordo com o teorema 4.1, o vetor $\boldsymbol{Y}$ possui distribuição normal multivariada com média $\boldsymbol{\mu}_{\boldsymbol{Y}} = \boldsymbol{\Sigma}^{1/2}\boldsymbol{\mu}_{\boldsymbol{Z}} + \boldsymbol{\mu} = \boldsymbol{\mu}$ e matriz de covariâncias $\boldsymbol{\Sigma}^{1/2}\boldsymbol{I}\boldsymbol{\Sigma}^{1/2} = \boldsymbol{\Sigma}$. Este

será o método que usaremos para obter a amostra p-dimensional de tamanho n de uma normal multivariada com média $\boldsymbol{\mu}$ e covariância $\boldsymbol{\Sigma}$. Para obtermos a matriz raiz quadrada no R, podemos utilizar o comando para a obtenção da decomposição espectral *svd* ou alternativamente o comando *chol*, que retorna o fator de Cholesky de uma matriz positiva definida, que na verdade é um tipo de raiz quadrada. Geraremos um vetor de variáveis aleatórias normal padrão independentes $\boldsymbol{Z}$ utilizando o comando *rnorm*. A decomposição do valor singular neste caso se especializa na decomposição espectral, pois a matriz $\boldsymbol{\Sigma}$ é simétrica.

Vamos ilustrar e apresentar o programa de geração de variáveis normais multivariada para um caso particular bivariado ($p = 2$) e com vetor de médias $\boldsymbol{\mu} = [10, 50]^\top$ e matriz de covariâncias dada por:

$$\boldsymbol{\Sigma} = \begin{bmatrix} 4 & 1 \\ 1 & 1 \end{bmatrix}.$$

O programa resultante é dado por:

```
> # Exemplificação e algoritmo para gerar $n$ vetores aleatórios normais
> # multivariados com vetor de médias mu e covariância Sigma.
>
> rnormmv <- function(n,mu,Sigma)
+ {
+   p         <- nrow(Sigma)
+   sig.svd   <- svd(Sigma)
+   Sigmaroot <- sig.svd$u%*%diag(sqrt(sig.svd$d))%*%t(sig.svd$v)
+   z <- rnorm(p,0,1)
+   x <- t(Sigmaroot%*%z+mu)
+   if (n > 1)
+   {
+     for (ii in 2:n)
+     {
+        z <- rnorm(p,0,1)
+        y <- Sigmaroot%*%z+mu
+        x <- rbind(x,t(y))
+     } # for ii in 2:n
+   } # if $n > 1$
+   return(x) # matriz n x p dos dados
+ } # rnormmv
> # exemplo de utilização
>
```

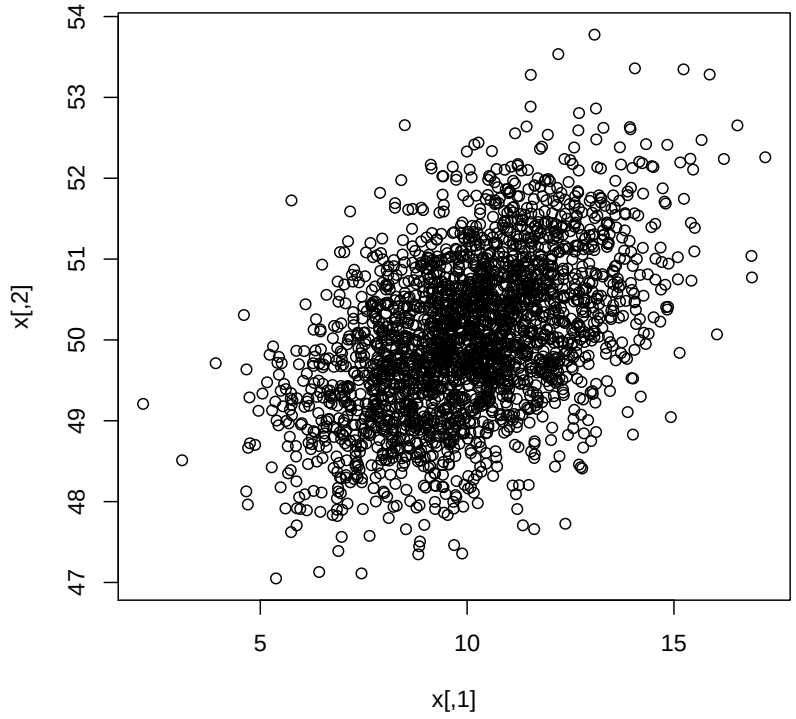
```
> Sigma <- matrix(c(4, 1, 1, 1), 2, 2)
> mu <- c(10,50)
> n <- 2500
> x <- rnormmv(n,mu,Sigma)
> xb <- apply(x, 2, mean) # aplica uma função (mean) às colunas de X - op. 2
> xb

[1] 10.04615 49.98492

> var(x)

          [,1]      [,2]
[1,] 3.9484753 0.9926383
[2,] 0.9926383 0.9932001

> par(mfrow = c(1,1))
> plot(x)
```



Uma alternativa é realizarmos uma implementação que não exige *loops*. Realizamos isso e verificamos um grande ganho em eficiência da função obtida. O resultado é apresentado a seguir.

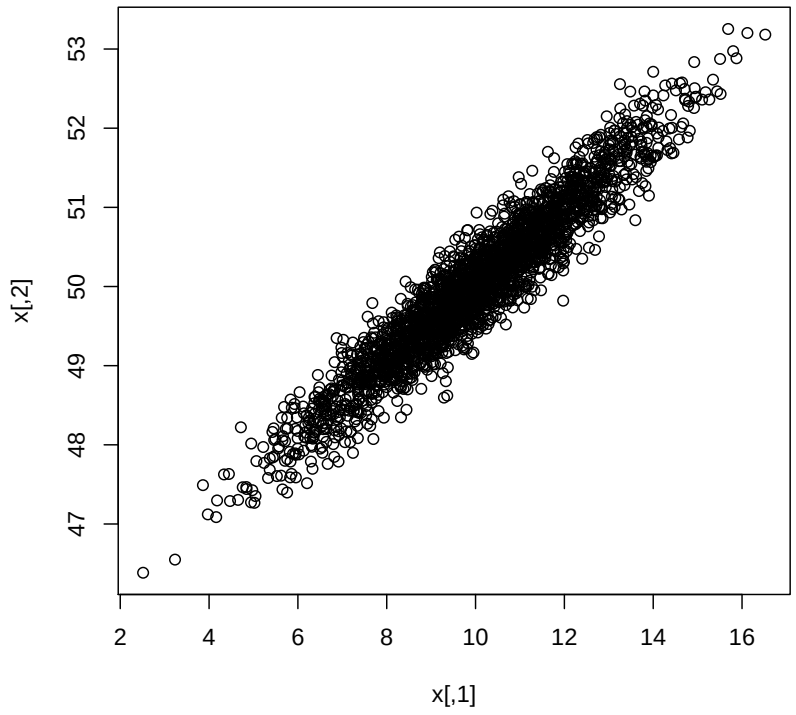
```
> # Exemplificação e algoritmo para gerar $n$ vetores aleatórios normais
> # multivariados com vetor de médias mu e covariância Sigma.
> # versão otimizada, que não usa loops
>
> rnormmultv <- function(n,mu,Sigma)
+ {
+   p         <- nrow(Sigma)
+   sig.svd   <- svd(Sigma)
+   Sigmaroot <- sig.svd$u%*%diag(sqrt(sig.svd$d))%*%t(sig.svd$v)
+   x <- (matrix(rnorm(n * p), n, p) %*% Sigmaroot) +
+        matrix(rep(mu, each = n), n, p)
+   return(x) # matriz n x p dos dados
+ } # rnormmv
> # exemplo de utilização
>
> Sigma <- matrix(c(4, 1.9, 1.9, 1), 2, 2)
> mu <- c(10, 50)
> n <- 2500
> x <- rnormmultv(n, mu, Sigma)
> xb <- apply(x, 2, mean) # aplica uma função (mean) às colunas de X - op. 2
> xb
[1]  9.980753 49.989423
> var(x)
         [,1]      [,2]
[1,] 3.831546 1.8282452
[2,] 1.828245 0.9699694
> plot(x)
> n <- 15000
> time1 <- proc.time()
> x <- rnormmv(n, mu, Sigma)
> time2 <- proc.time()
> tg1 <- (time2 - time1) / n
> n <- 50000
> time1 <- proc.time()
> x <- rnormmultv(n, mu, Sigma)
> time2 <- proc.time()
> tg2 <- (time2 - time1) / n
> tg1 / tg2 # razão dos tempos médios gastos
    user   system  elapsed
178.4848      Inf 186.3636
```

```
> tg1

       user       system      elapsed
0.0003926667 0.0000060000 0.0004100000

> tg2

   user  system elapsed
2.2e-06 0.0e+00 2.2e-06
```



Podemos em vez de utilizar as funções implementadas anteriormente, gerar dados da normal multivariada a partir da função *mvrnorm(n, mu, Sigma)* do pacote *MASS* ou a partir da função *rmvnorm(n, mean, sigma)* do pacote *mvtnorm*. O pacote *MASS* refere-se as funções disponibilizadas em *Modern Applied Statistics with S* e é conhecido também por pacote *VR*. É extremamente simples gerarmos dados de normais multivariadas utilizando tais funções, pois não necessitamos programar os geradores aleatórios. Vamos demonstrar o uso destas duas funções no programa ilustrativo apresentado na sequência. Devemos reiterar que pacotes são carregados para a memória com o comando *library(pacote)*. O programa resultante é:

```
> # Exemplo para gerarmos n vetores aleatórios normais multivariados
> # com vetor de médias mu e covariância Sigma, a partir das funções
> # dos pacotes MASS e mvtnorm.
> # pacote MASS
>
> Sigma <- matrix(c(4, 1.9, 1.9, 1), 2, 2)
> mu <- c(10, 50)
> n <- 30
> library(MASS)
> x <- mvrnorm(n, mu, Sigma)
> library(mvtnorm)
> y <- rmvnorm(n, mu, Sigma)
> xb <- apply(x, 2, mean)# aplica uma função (mean) às colunas de X - op. 2
> xb # média x
[1] 10.21758 50.02263
> var(x) # covariância x
         [,1]      [,2]
[1,] 3.284661 1.4223831
[2,] 1.422383 0.7224305
```

4.3 Distribuição Wishart e Wishart Invertida

As distribuições Wishart e Wishart invertida são relativas às matrizes de somas de quadrados e produtos não-corrigidas $\boldsymbol{W}$ obtidas de amostras de tamanho ν da distribuição normal multivariada com média $\mathbf{0}$. Seja $\boldsymbol{X}_j = [X_1, X_2, \cdots, X_p]^\top$ o j-ésimo vetor ($j = 1, 2, \cdots, \nu$) de uma amostra aleatória de tamanho ν de uma normal com média $\mathbf{0}$ e covariância $\boldsymbol{\Sigma}$, então a matriz aleatória

$$\boldsymbol{W} = \sum_{j=1}^{\nu} \boldsymbol{X}_j \boldsymbol{X}_j^\top$$

possui distribuição Wishart com ν graus de liberdade e parâmetro $\boldsymbol{\Sigma}$ (matriz positiva definida).

Da mesma forma, se temos uma amostra aleatória de tamanho n de uma distribuição normal multivariada com média $\boldsymbol{\mu}$ e covariância $\boldsymbol{\Sigma}$, a distribuição da matriz aleatória

$$\boldsymbol{W} = \sum_{j=1}^{n} (\boldsymbol{X}_j - \bar{\boldsymbol{X}})(\boldsymbol{X}_j - \bar{\boldsymbol{X}})^\top$$

é Wishart com $\nu = n - 1$ graus de liberdade e parâmetro $\boldsymbol{\Sigma}$.

A função de densidade Wishart de uma matriz aleatória $\boldsymbol{W}$ de somas de quadrados e produtos e representada por $W_p(\nu,\boldsymbol{\Sigma})$ é definida por:

$$f_{\boldsymbol{W}}(\boldsymbol{w}|\nu,\boldsymbol{\Sigma}) = \frac{|\boldsymbol{\Sigma}|^{-\nu/2}|\boldsymbol{w}|^{(\nu-p-1)/2}}{2^{\nu p/2}\pi^{p(p-1)/4}\prod_{i=1}^{p}\Gamma\left(\frac{\nu-i+1}{2}\right)} \exp\left\{-\frac{1}{2}tr\left(\boldsymbol{\Sigma}^{-1}\boldsymbol{w}\right)\right\} \qquad (4.3.1)$$

em que $\Gamma(x) = \int_0^\infty t^{x-1}e^{-x}dt$ é função gama.

Assim, para gerarmos variáveis Wishart com parâmetros ν inteiro e $\boldsymbol{\Sigma}$ positiva definida, podemos utilizar um gerador de amostras aleatórias normais multivariadas e obter a matriz de somas de quadrados e produtos amostrais. Esta matriz será uma realização de uma variável aleatória Wishart, que é uma matriz de dimensão $p \times p$. A seguinte função pode ser utilizada para obtermos realizações aleatórias de uma Wishart:

```
> # Exemplificação para gerarmos matrizes de somas de quadrados e produtos
> # aleatórias W com distribuição Wishart(nu, Sigma)
> # utiliza o pacote mvtnorm para gerar amostras normais.
>
> rWishart <- function(nu, Sigma)
+ {
+    p  <- nrow(Sigma)
+    mu <- matrix(0, p)
+    y  <- rmvnorm(nu + 1, mu, Sigma)
+    w  <- nu * var(y) # var(y) retorna a mat. de cov., logo (n-1)var(y) = W
+    return(w)
+ }
> # exemplo de utilização
>
> library(mvtnorm) # Carregar o pacote mvtnorm antes de chamar a função
> Sigma <- matrix(c(4, 1, 1, 1), 2, 2)
> nu    <- 5
> w     <- rWishart(nu, Sigma)
> w

          [,1]     [,2]
[1,] 42.651800 3.560491
[2,]  3.560491 3.816040
```

Outra distribuição relacionada que aparece frequentemente na inferência multivariada é a Wishart invertida. Seja $\boldsymbol{W}$ uma matriz aleatória $W_p(\nu, \boldsymbol{\Sigma})$, então a distribuição de $\boldsymbol{S} = \boldsymbol{W}^{-1}$, dada pela função de densidade

$$f_{\boldsymbol{S}}(\boldsymbol{s}|\nu,\boldsymbol{\Sigma}) = \frac{|\boldsymbol{\Sigma}^{-1}|^{\nu/2}|\boldsymbol{s}|^{-(\nu+p+1)/2}}{2^{\nu p/2}\pi^{p(p-1)/4}\prod_{i=1}^{p}\Gamma\left(\frac{\nu-i+1}{2}\right)} \exp\left\{-\frac{1}{2}tr\left(\boldsymbol{\Sigma}^{-1}\boldsymbol{s}^{-1}\right)\right\} \qquad (4.3.2)$$

é a Wishart invertida que vamos representar por $W_p^{-1}(\nu, \boldsymbol{\Sigma})$.

Se quisermos gerar uma matriz aleatória de uma distribuição Wishart invertida em vez de uma Wishart precisamos simplesmente realizar a transformação $\boldsymbol{S} = \boldsymbol{W}^{-1}$. Assim, vamos alterar o programa anterior para gerar simultaneamente realizações aleatórias de distribuição Wishart e Wishart invertida. É importante salientar que em nossa notação os parâmetros da Wishart invertida são aqueles da Wishart. Isto é relevante, pois alguns autores apresentam os parâmetros da Wishart invertida e não da Wishart e devemos estar atentos para este fato, senão estaremos gerando variáveis aleatórias de densidades diferentes.

```
> # Exemplificação para gerarmos matrizes de somas de quadrados e produtos
> # aleatórias W com distribuição Wishart(nu, Sigma) e Wishart invertida
> # WI(nu, Sigma). Utiliza o pacote mvtnorm para gerar amostras normais.
>
> rWishart <- function(nu, Sigma)
+ {
+    p  <- nrow(Sigma)
+    mu <- matrix(0, p)
+    y  <- rmvnorm(nu + 1, mu, Sigma)
+    w  <- nu * var(y) # var(y): matriz de covariância, logo (n-1)var(y) = W
+    wi <- solve(w)
+    return(list(w = w, wi = wi))
+ }
> # exemplo de utilização
>
> library(mvtnorm) # Carregar o pacote mvtnorm antes de chamar a função
> Sigma <- matrix(c(4, 1, 1, 1), 2, 2)
> nu    <- 5
> x     <- rWishart(nu, Sigma)
> x$w   # Wishart

         [,1]     [,2]
```

```
[1,] 6.171315 1.562092
[2,] 1.562092 2.486979

> x$wi  # Wishart invertida

           [,1]       [,2]
[1,]  0.1926726 -0.1210192
[2,] -0.1210192  0.4781074
```

Um aspecto importante que precisamos mencionar é sobre a necessidade de gerarmos variáveis Wishart ou Wishart invertida com graus de liberdade reais. Para isso podemos utilizar o algoritmo descrito por Smith e Hocking (1972)[16]. Seja a decomposição da matriz $\boldsymbol{\Sigma}$ dada por $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}^{1/2}\boldsymbol{\Sigma}^{1/2}$ que utilizaremos para realizarmos uma transformação na matriz aleatória gerada, que de acordo com propriedades de matrizes Wishart, será Wishart. Devemos inicialmente construir uma matriz triangular inferior $\boldsymbol{T} = [t_{ij}]$ (i,j = 1, 2, $\cdots$, p), com $t_{ii} \sim \sqrt{\chi^2_{\nu+1-i}}$ e $t_{ij} \sim N(0,1)$ se $i > j$ e $t_{ij} = 0$ se $i < j$. Na sequência devemos obter a matriz $\boldsymbol{\Gamma} = \boldsymbol{T}\boldsymbol{T}^\top$, que possui distribuição $W_p(\nu, \boldsymbol{I})$. Desta forma obteremos $\boldsymbol{W} = \boldsymbol{\Sigma}^{1/2}\boldsymbol{\Gamma}\boldsymbol{\Sigma}^{1/2}$ com a distribuição desejada, ou seja, com distribuição $W_p(\nu,\boldsymbol{\Sigma})$. Fica claro que com esse algoritmo podemos gerar variáveis Wishart com graus de liberdade reais ou inteiros.

```
> # Função mais eficiente para gerarmos matrizes de somas de quadrados
> # e produtos aleatórias W com distribuição Wishart(nu, Sigma) e
> # Wishart invertida WI(nu, Sigma)
>
> rWWI <- function (nu, Sigma)
+ {
+     p <- nrow(Sigma)
+     df <- (nu + nu - p + 1) - (nu - p + 1):nu
+     if (p > 1)
+     {
+         T <- diag(sqrt(rgamma(c(rep(1, p)), df/2, 1/2)))
+         T[lower.tri(T)] <- rnorm((p * (p - 1)/2))
+     } else T <- sqrt(rgamma(1, df/2, 1/2))
+     S  <- chol(Sigma)  # fator de Cholesky
+     w  <- t(S) %*% T %*% t(T) %*% S
+     wi <- solve(w)
+     return(list(w = w, wi = wi))
+ } # função rWWI
> # Exemplo de uso
>
```

```
> nu    <- 5
> Sigma <- matrix(c(4, 1, 1, 1), 2, 2)
> rWWI(nu, Sigma)

$w
          [,1]      [,2]
[1,] 8.2251481 0.2097683
[2,] 0.2097683 3.1365051

$wi
             [,1]         [,2]
[1,]  0.121786082 -0.008145008
[2,] -0.008145008  0.319370938
```

Vamos chamar a atenção para alguns fatos sobre esta função, pois utilizamos alguns comandos e artifícios não mencionados até o presente momento. Inicialmente utilizamos o comando *T <- diag(sqrt(rgamma(c(rep(1, p)), df/2, 1/2)))* para preencher a diagonal da matriz $\boldsymbol{T}$ com variáveis aleatórias quiquadrado. Os artifícios usados foram: a) usamos o comando *c(rep(1, p))* para criar um vetor de 1's de dimensão $p \times 1$; b) usamos o comando *rgamma(c(rep(1, p)), df/2, 1/2)* para produzir um vetor de variáveis aleatórias gama com parâmetros $\alpha < -df/2$ e $\beta < -2$; c) o comando *sqrt* é aplicado no vetor formado e o comando *diag* para criar uma matriz com os elementos deste vetor. Resta nos justificar que o gerador de números aleatórios da gama foi utilizado como um artifício para gerarmos variáveis aleatórias quiquadrado aproveitando a relação da gama com a quiquadrado. Um distribuição gama com parâmetros α e β e uma quiquadrado com parâmetro ν têm a seguinte relação: $\chi^2(\nu) = Gama(\alpha = \nu/2, \beta = 2)$. O segundo detalhe, de extrema importância, refere-se ao fato de que no R, o parâmetro β é o recíproco do terceiro argumento da função *rgamma()*. Por isso entramos com 1/2 e não com 2 para este argumento. A escolha do gerador de variáveis aleatórias gama foi feita apenas para ilustrar a relação da distribuição quiquadrado com a gama.

Outro aspecto interessante que merece ser mencionado é o uso do fator de Cholesky no lugar de obter a matriz raiz quadrada de $\boldsymbol{\Sigma}$. O fator de Cholesky utiliza a decomposição $\boldsymbol{\Sigma} = \boldsymbol{S}\boldsymbol{S}^\top$, em que $\boldsymbol{S}$ é uma matriz triangular inferior. O R por meio da função *chol(Sigma)*, retorna a matriz $\boldsymbol{S}^\top$. Finalmente, se o número de variáveis é igual a 1, a distribuição Wishart se especializa na quiquadrado e a Wishart invertida na quiquadrado invertida. Assim, quando $p = 1$ o algoritmo retornará va-

riáveis $\sigma^2 X$ e $1/(\sigma^2 X)$ com distribuições proporcionais a distribuição quiquadrado e quiquadrado invertida, respectivamente.

As mais importantes funções pré-existentes do R são a função *rwishart* do pacote *bayesm* que se assemelha muito com a implementação que realizamos *rWWI* e a função *rwish* do pacote *MCMCpack* que possui também implementação parecida. Em ambos os casos é utilizado o gerador de variáveis aleatórias quiquadrado.

4.4 Distribuição t de Student Multivariada

A família de distribuições elípticas é muito importante na multivariada e nas aplicações Bayesianas. A distribuição t de Student multivariada é um caso particular desta família. Esta distribuição tem particular interesse nos procedimentos de comparação múltipla dos tratamentos com alguma testemunha avaliada no experimento. A função de densidade do vetor aleatório $\boldsymbol{X} = [X_1, X_2, \cdots, X_p]^\top \in \mathbb{R}^p$ com parâmetros dados pelo vetor de médias $\boldsymbol{\mu} = [\mu_1, \mu_2, \cdots, \mu_p]^\top \in \mathbb{R}^p$ e matriz simétrica e positiva definida $\boldsymbol{\Sigma}$ $(p \times p)$ é

$$\begin{aligned} f_{\boldsymbol{X}}(\boldsymbol{x}) =& \frac{g((\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}))}{|\boldsymbol{\Sigma}|^{1/2}} \\ =& \frac{\Gamma\left(\frac{\nu+p}{2}\right)}{(\pi\nu)^{p/2}\Gamma(\nu/2)|\boldsymbol{\Sigma}|^{1/2}} \left[1 + \frac{1}{\nu}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right]^{-\frac{\nu+p}{2}} \end{aligned} \tag{4.4.1}$$

em que a função g é dada por $g(z) = \dfrac{\Gamma\left(\frac{\nu+p}{2}\right)}{(\pi\nu)^{p/2}\Gamma(\nu/2)} \left(1 + \dfrac{z}{\nu}\right)^{-(\nu+p)/2}$. A variável aleatória $\boldsymbol{X}$ tem média $\boldsymbol{\mu}$ e matriz de covariâncias $\nu\boldsymbol{\Sigma}/(\nu-2)$ no caso de $\nu > 2$.

Se efetuarmos a transformação $\boldsymbol{Y} = \boldsymbol{\Sigma}^{-1/2}(\boldsymbol{X} - \boldsymbol{\mu})$ obteremos a distribuição t multivariada esférica simétrica, cuja densidade é dada por:

$$f_{\boldsymbol{Y}}(\boldsymbol{y}) = \frac{\Gamma\left(\frac{\nu+p}{2}\right)}{(\pi\nu)^{p/2}\Gamma(\nu/2)} \left[1 + \frac{1}{\nu}\boldsymbol{y}^\top \boldsymbol{y}\right]^{-\frac{\nu+p}{2}}. \tag{4.4.2}$$

A variável aleatória $\boldsymbol{Y}$ terá vetor de médias nulo e covariâncias $\nu\boldsymbol{I}/(\nu-2)$ e a densidade terá contornos esféricos de mesma probabilidade.

Vamos apresentar a forma geral para gerarmos variáveis aleatórias p-dimensionais t multivariada com ν graus de liberdade e parâmetros $\boldsymbol{\mu}$ e $\boldsymbol{\Sigma}$. Seja um vetor aleatório $\boldsymbol{Z}$ com distribuição $N_p(\boldsymbol{0}, \boldsymbol{I})$ e a variável aleatória U com distribuição quiquadrado

com ν graus de liberdade, então o vetor aleatório $\boldsymbol{Y}$, dado pela transformação

$$\boldsymbol{Y} = \sqrt{\nu}\frac{\boldsymbol{Z}}{\sqrt{U}}, \tag{4.4.3}$$

possui distribuição t multivariada esférica com ν graus de liberdade. O vetor $\boldsymbol{X}$ obtido pela transformação linear

$$\boldsymbol{X} = \boldsymbol{\Sigma}^{1/2}\boldsymbol{Y} + \boldsymbol{\mu}, \tag{4.4.4}$$

possui distribuição t multivariada elíptica com ν graus de liberdade e parâmetros $\boldsymbol{\mu}$ e $\boldsymbol{\Sigma}$.

Assim, devemos aplicar a transformação (4.4.4) n vezes a n diferentes vetores aleatórios $\boldsymbol{Y}$ e variáveis U. Ao final deste processo teremos uma amostra de tamanho n da distribuição t multivariada almejada com ν graus de liberdade. Assim, para gerarmos dados de uma t multivariada com dimensão p, graus de liberdade ν (não necessariamente inteiro), vetor de média $\boldsymbol{\mu}$ nulo e matriz positiva definida $\boldsymbol{\Sigma}$ podemos utilizar a seguinte função, substituindo na expressão (4.4.4) a matriz raiz quadrada pelo fator de Cholesky $\boldsymbol{F}$ de $\boldsymbol{\Sigma}$:

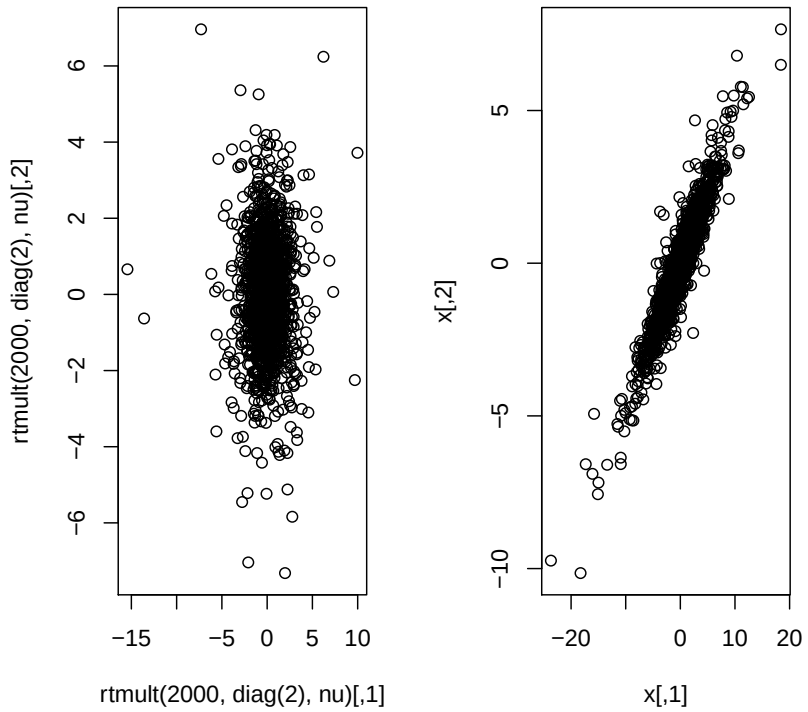
```
> # Função para gerarmos variáveis aleatórias t
> # multivariadas (n, p, nu, Sigma). Devemos carregar o
> # pacote mvtnorm antes de usar a função
> # usa o comando: library(mvtnorm)
>
> rtmult <- function (n, sigma = diag(3), df = 1)
+ {
+    library(mvtnorm)
+    F <- chol(sigma)
+    p <- nrow(sigma)
+    return((rmvnorm(n, sigma = diag(p))/sqrt(rchisq(n, df)/df))%*%F)
+ }
> # Exemplo de uso
>
> nu <- 4
> par(mfrow = c(1,2))
> plot(rtmult(2000,diag(2), nu))
> Sigma <- matrix(c(4, 1.9, 1.9, 1), 2, 2)
> x <- rtmult(3000,Sigma, nu) # t com correlação 0,95 entre X1 e X2
> var(x) # valor esperado é nu Sigma/(nu - 2)
```

```
         [,1]     [,2]
[1,] 7.483107 3.582243
[2,] 3.582243 1.969299
```

```
> nu * Sigma / (nu - 2) # valor esperado
```

```
     [,1] [,2]
[1,]  8.0  3.8
[2,]  3.8  2.0
```

```
> plot(x)
```



Podemos utilizar a implementação determinada pela função *rmvt*, do pacote *mvtnorm*, para gerarmos dados da distribuição t multivariada centrada em zero. Graus de liberdade reais positivos podem ser utilizados como argumento da função. Foram definidos para os parâmetros $\boldsymbol{\Sigma}$ e ν, os valores $\boldsymbol{I}_3$ e 1, respectivamente, como *default*.

4.5 Outras Distribuições Multivariadas

Existem muitas outras distribuições multivariadas. Vamos mencionar apenas mais duas delas: a log-normal e a normal contaminada multivariadas. A geração de um vetor coluna aleatório de dimensão p com distribuição log-normal multivariada é feita tomando-se o seguinte vetor $\boldsymbol{W} = [\exp(Z_1),\ \exp(Z_2),\ \cdots,\ \exp(Z_p)]^\top$, em que $\boldsymbol{Z} \sim N_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Para gerarmos realizações da normal contaminada multivariada consideramos a simulação de um vetor aleatório $\boldsymbol{X}$ cuja densidade de probabilidade é dada por:

$$\begin{aligned} f_{\boldsymbol{X}}(\boldsymbol{x}) =& \delta(2\pi)^{-p/2}|\boldsymbol{\Sigma}_1|^{-1/2}\exp\left\{-\frac{(\boldsymbol{x}-\boldsymbol{\mu}_1)^\top\boldsymbol{\Sigma}_1^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1)}{2}\right\} \\ &+ (1-\delta)(2\pi)^{-p/2}|\boldsymbol{\Sigma}_2|^{-1/2}\exp\left\{-\frac{(\boldsymbol{x}-\boldsymbol{\mu}_2)^\top\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_2)}{2}\right\} \end{aligned}$$

em que $\boldsymbol{\Sigma}_i$ positiva definida, $i = 1, 2$ e $0 \leq \delta \leq 1$.

4.6 Exercícios

4.6.1 Gerar uma amostra de tamanho ($n = 10$) uma distribuição normal trivariada com vetor de médias $\boldsymbol{\mu} = [10{,}100, 50]^\top$ e matriz de covariâncias

$$\boldsymbol{\Sigma} = \begin{bmatrix} 5 & -1 & 2 \\ -1 & 3 & -2 \\ 2 & -2 & 7 \end{bmatrix}.$$

Estimar a média e a covariância amostral. Repetir este processo 1.000 vezes e estimar a média das médias amostrais e a média das matrizes de covariâncias amostrais. Estes valores correspondem exatamente aos respectivos valores esperados? Se não, apresentar a(s) principal(is) causa(s).

4.6.2 Alterar a função *rnormmv* de tal forma que a matriz $\boldsymbol{X}$ $(n \times p)$ de observações multivariadas normais seja obtida sem a necessidade de utilizar o *loop*, ou seja, sem a estrutura de repetição determinada pelo comando *for*, com fizemos por intermédio da função *rnormmultv*. Entretanto utilize o fator de Cholesky, no lugar da raiz quadrada obtida a partir da decomposição espectral de Σ. Verifique se houve melhoria no desempenho em relação ao tempo médio de processamento de cada variável aleatória.

4.6.3 Para gerar variáveis Wishart e Wishart invertida apresentar um programa utilizando as funções *rwishart* do pacote *bayesm* e *rwish* do pacote *MCMCpack*.

4.6.4 Sabemos que variáveis Wishart possuem média $\nu\boldsymbol{\Sigma}$. Apresentar um programa para verificar se o valor esperado, ignorando o erro de Monte Carlo, é alcançado com o uso das funções apresentadas para gerar variáveis Wishart. Utilizar 10.000 repetições de Monte Carlo.

4.6.5 Implementar funções R para gerarmos variáveis aleatórias log-normal e normal-contaminada elípticas multivariadas.

4.6.6 Implementar uma função R para gerarmos variáveis aleatórias t multivariada com parâmetros $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$ e ν de dimensão p. Utilizar vetor de médias $\boldsymbol{\mu}$ não nulo e possibilitar a especificação de valores reais positivos para o parâmetro ν, o que diferenciaria esta função da *rtmult*.

Capítulo 5

Algoritmos para Médias, Variâncias e Covariâncias

Muitos algoritmos para o cálculo de médias, variâncias e covariâncias são imprecisos, podendo gerar resultados finais contendo grandes erros. A necessidade de utilizarmos algoritmos eficientes para realizarmos estas tarefas simples são evidentes e serão descritos neste capítulo.

5.1 Introdução

Felizmente o R utiliza algoritmos precisos para cálculo da média, da variância e de covariância. Vamos buscar esclarecer como a utilização de algoritmo ineficientes podem levar a resultados inconsistentes e imprecisos. Nosso objetivo neste capítulo é apresentar os algoritmos eficientes para estimarmos estes parâmetros e mostrar como as fórmulas convencionais podem falhar se utilizadas nos algoritmos diretamente.

Estes algoritmos eficientes são particularmente úteis quando os dados possuem grande magnitude ou estão muito próximos de zero. Neste caso particular, algumas planilhas eletrônicas, como o Excel nas suas versões mais antigas, podiam falhar (McCullough e Wilson, 1999[10]). O conhecimento de algoritmos que conduzirão a maiores precisões numéricas pode levar o pesquisador a não cometer os mesmos erros encontrados em alguns *softwares*.

5.2 Algoritmos Univariados

A ideia básica de utilizarmos algoritmos para média, para a soma de potências dos desvios em relação a média ou para soma de produtos de desvios é aplicarmos recursivamente as fórmulas existentes. Se desejamos, por exemplo, obter a média de n observações, podemos inicialmente calcular a média da primeira observação, que é a própria. Em um segundo estágio, propor uma expressão para atualizarmos a média da primeira observação, contemplando a segunda e assim sucessivamente. O mesmo procedimento de atualização é aplicado às variâncias ou às covariâncias, no caso de termos mais de uma variável.

Para uma amostra de tamanho n dada por $X_1, X_2, \cdots, X_n$, a média e a variância amostral convencionais são obtidas, respectivamente, por:

$$\bar{X}_{\cdot} = \frac{\sum_{i=1}^{n} X_i}{n}, \tag{5.2.1a}$$

$$S^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n} X_i^2 - \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n}\right]. \tag{5.2.1b}$$

Alguns algoritmos existentes procuraram melhorar os algoritmos dos livros textos que são as fórmulas das equações (5.2.1a e 5.2.1b) procurando fazer adaptações, repassando a amostra duas vezes. Estes algoritmos podem ser eficientes do ponto de vista da precisão, mas não são rápidos, justamente por repassarem duas vezes os dados. West (1979)[17] propôs utilizar um algoritmo que faz uma única passada, atualizando a média e a variância em cada nova observação. Podemos facilmente mostrar que para uma amostra de tamanho n, que a média é igual a X_1 se $n = 1$, $(X_1 + X_2)/2$ se $n = 2$ e assim por diante. No (k-1)-ésimo passo podemos especificar o estimador da média por:

$$\bar{X}_{k-1} = \frac{\sum_{i=1}^{k-1} X_i}{k-1}.$$

No k-ésimo passo teremos observado X_k e a média atualizada é:

$$\bar{X}_k = \frac{\sum_{i=1}^{k} X_i}{k}. \tag{5.2.2}$$

A pergunta que fazemos é "podemos expressar a média do k-ésimo passo em função da média do (k-1)-ésimo passo?" A resposta a esta pergunta é sim e o resultado nos fornece o algoritmo desejado. Então, a partir da equação (5.2.2) obtemos:

$$\begin{aligned}
\bar{X}_k =& \frac{\sum_{i=1}^{k-1} X_i + X_k}{k} \\
=& \frac{(k-1)\sum_{i=1}^{k-1} X_i}{(k-1)k} + \frac{X_k}{k} \\
=& \frac{(k-1)\bar{X}_{k-1}}{k} + \frac{X_k}{k} \\
=& \bar{X}_{k-1} - \frac{\bar{X}_{k-1}}{k} + \frac{X_k}{k}
\end{aligned}$$

resultando na equação recursiva final

$$\bar{X}_k = \bar{X}_{k-1} + \frac{X_k - \bar{X}_{k-1}}{k}, \tag{5.2.3}$$

para $2 \leq k \leq n$, sendo que $\bar{X}_1 = X_1$.

Da mesma forma se definirmos a soma de quadrados corrigidas das k primeiras observações amostrais $1 < k \leq n$ por:

$$W2_k = \sum_{i=1}^{k} X_i^2 - \frac{\left(\sum_{i=1}^{k} X_i\right)^2}{k}$$

veremos que a variância correspondente é dada por $S_k^2 = W2_k/(n-1)$. Se expandirmos esta expressão isolando o k-ésimo termo e simplificarmos a expressão resultante teremos:

$$\begin{aligned} W2_k =& \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{\left(\sum_{i=1}^{k-1} X_i + X_k\right)^2}{k} \\ =& W2_{k-1} + k\left(X_k - \bar{X}_k\right)^2/(k-1) \end{aligned} \tag{5.2.4}$$

A expressão que desenvolvemos (5.2.4) é equivalente a apresentada por West (1979)[17]. Isso pode ser demonstrado facilmente se substituirmos $\bar{X}_k$ obtida na equação (5.2.3) na equação (5.2.4), de onde obtivemos:

$$W2_k = W2_{k-1} + (k-1)\left(X_k - \bar{X}_{k-1}\right)^2/k \tag{5.2.5}$$

para $2 \leq k \leq n$, sendo que $W2_1 = 0$. A variância é obtida por $S^2 = W2_n/(n-1)$.

Para demonstrarmos diretamente a expressão (5.2.5) temos:

$$\begin{aligned} W2_k =& \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{\left(\sum_{i=1}^{k-1} X_i + X_k\right)^2}{k} \\ =& \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2 + 2X_k\sum_{i=1}^{k-1} X_i + X_k^2}{k} \\ =& \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{(k-1)\left(\sum_{i=1}^{k-1} X_i\right)^2 + 2(k-1)X_k\sum_{i=1}^{k-1} X_i + (k-1)X_k^2}{k(k-1)} \\ =& \sum_{i=1}^{k-1} X_i^2 - \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2}{k-1} + X_k^2 + \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2}{k(k-1)} - \frac{2(k-1)X_k\sum_{i=1}^{k-1} X_i}{k(k-1)} - \frac{X_k^2}{k} \\ =& W2_{k-1} + \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2}{k(k-1)} - \frac{2(k-1)X_k\sum_{i=1}^{k-1} X_i}{k(k-1)} + \frac{(k-1)X_k^2}{k} \\ =& W2_{k-1} + \frac{(k-1)\bar{X}_{k-1}^2}{k} - \frac{2(k-1)X_k\bar{X}_{k-1}}{k} + \frac{(k-1)X_k^2}{k}, \end{aligned}$$

resultando em

$$W2_k = W2_{k-1} + (k-1)\left(X_k - \bar{X}_{k-1}\right)^2/k.$$

Podemos generalizar essa expressão para computarmos a covariância $S_{(x,y)}$ entre uma variável X e outra Y. A expressão para a soma de produtos é dada por:

$$W2_{k,(x,y)} = W2_{k-1,(x,y)} + (k-1)\left(X_k - \bar{X}_{k-1}\right)\left(Y_k - \bar{Y}_{k-1}\right)/k \tag{5.2.6}$$

para $2 \leq k \leq n$, sendo $W2_{1,(x,y)} = 0$. O estimador da covariância é obtido por $S_{(x,y)} = W2_{n,(x,y)}/(n-1)$.

Podemos observar, analisando todas estas expressões, que para efetuarmos os cálculos da soma de quadrados e da soma de produtos corrigida necessitamos do cálculo das médias das variáveis no k-ésimo passo ou no passo anterior. A vantagem é que em uma única passagem pelos dados, obtemos todas as estimativas. Podemos ainda estender estes resultados para obtermos somas das terceira e quarta potências dos desvios em relação a média. As expressões que derivamos para isso são:

$$W3_k = W3_{k-1} + \frac{(k^2-3k+2)(X_k - \bar{X}_{k-1})^3}{k^2} - \frac{3(X_k - \bar{X}_{k-1})W2_{k-1}}{k} \tag{5.2.7a}$$

$$W4_k = W4_{k-1} + \frac{(k^3-4k^2+6k-3)(X_k - \bar{X}_{k-1})^4}{k^3} + \\ + \frac{6(X_k - \bar{X}_{k-1})^2 W2_{k-1}}{k^2} - \frac{4(X_k - \bar{X}_{k-1})W3_{k-1}}{k} \tag{5.2.7b}$$

para $2 \leq k \leq n$, sendo $W3_1 = 0$ e $W4_1 = 0$.

Desta forma podemos implementar a função *medsqk()* que retorna a média, a soma de quadrado, cubo e quarta potência dos desvios em relação a média e variância a partir de um vetor de dados $\boldsymbol{X}$ de dimensão n.

```
> # função para retornar a média, somas de desvios em relação a média
> # ao quadrado, ao cubo e quarta potência e variância
> medsqk <- function(x)
+ {
+   n <- length(x)
+   if (n <= 1) stop("Dimensão do vetor deve ser maior que 1!")
+   xb <- x[1]
+   W2 <- 0
+   W3 <- 0
```

```
+   W4 <- 0
+   for (ii in 2:n)
+   {
+     aux <- x[ii] - xb
+     W4  <- W4 + (ii**3 - 4 * ii**2 + 6 * ii - 3) * aux**4 / ii**3
+            + 6 * W2 * aux**2 / ii**2 - 4 * W3 * aux / ii
+     W3  <- W3 + (ii**2 - 3 * ii + 2) * aux**3 / ii**2 - 3 * W2 * aux / ii
+     W2  <- W2 + (ii - 1) * aux**2 / ii
+     xb  <- xb + aux / ii
+   }
+   S2 <- W2 / (n - 1)
+   list(media = xb, variancia = S2, SQ2 = W2, W3 = W3, W4 = W4)
+ }
> x <- c(1, 2, 3, 4, 5, 7, 8)
> medsqk(x)

$media
[1] 4.285714

$variancia
[1] 6.571429

$SQ2
[1] 39.42857

$W3
[1] 22.04082

$W4
[1] 338.4030
```

5.3 Algoritmos para Vetores Médias e Matrizes de Covariâncias

Vamos apresentar nesta seção a extensão multivariada para obtermos o vetor de médias e as matrizes de somas de quadrados e produtos e de covariâncias. Por essa razão não implementamos uma função específica para obtermos a covariância entre duas variáveis X e Y. Seja uma amostra aleatória no espaço $\mathbb{R}^p$ dada por $\boldsymbol{X}_1$, $\boldsymbol{X}_2$, $\cdots$, $\boldsymbol{X}_j$, $\cdots$, $\boldsymbol{X}_n$, sendo que estes vetores serão dispostos em uma matriz $\boldsymbol{X}$ de dimensões $(n \times p)$. Para estendermos os resultados da seção anterior, utilizaremos

as mesmas expressões, tomando o devido cuidado para adaptá-las para lidar com operações matriciais e vetoriais. Assim, para estimarmos o vetor de médias populacionais, devemos em vez de utilizar o estimador clássico dos livros textos dado por

$$\bar{\boldsymbol{X}} = \frac{\sum_{i=1}^{n} \boldsymbol{X}_i}{n},$$

utilizaremos a expressão recursiva dada por

$$\bar{\boldsymbol{X}}_k = \bar{\boldsymbol{X}}_{k-1} + \frac{\boldsymbol{X}_k - \bar{\boldsymbol{X}}_{k-1}}{k}, \tag{5.3.1}$$

para $2 \leq k \leq n$, sendo que $\bar{\boldsymbol{X}}_1 = \boldsymbol{X}_1$.

Da mesma forma, a adaptação para p dimensões da expressão (5.2.5) é direta e o resultado obtido é:

$$\boldsymbol{W}_k = \boldsymbol{W}_{k-1} + (k-1)\left(\boldsymbol{X}_k - \bar{\boldsymbol{X}}_{k-1}\right)\left(\boldsymbol{X}_k - \bar{\boldsymbol{X}}_{k-1}\right)^{\top}/k \tag{5.3.2}$$

para $2 \leq k \leq n$, sendo $\boldsymbol{W}_1 = \boldsymbol{0}$, uma matriz de zeros de dimensões $(p \times p)$. O estimador da matriz de covariâncias é obtido por $\boldsymbol{S} = \boldsymbol{W}_n/(n-1)$.

Implementamos a função *medcov()* apresentada a seguir para obtermos o vetor de médias, a matriz de somas de quadrados e produtos e a matriz de covariâncias. O argumento desta função deve ser uma matriz de dados multivariados com n linhas (observações) e p colunas (variáveis). O programa resultante e um exemplo são apresentados na sequência. Escolhemos um exemplo onde geramos uma matriz de dados de uma normal multivariada.

```
> # função para retornar o vetor de médias, a matriz de somas de
> # quadrados e produtos e a matriz de covariâncias
> medcov <- function(x)
+ {
+   n  <- nrow(x)
+   p  <- ncol(x)
+   if (n <= 1) stop("Dimensão linha da matriz deve ser maior que 1!")
+   xb <- x[1,]
+   W  <- matrix(0, p, p)
+   for (ii in 2:n)
+   {
```

```
+     aux <- x[ii,] - xb
+     W   <- W + (ii - 1) * aux %*% t(aux) / ii
+     xb  <- xb + aux / ii
+   }
+   S  <- W / (n - 1)
+   list(vetmedia = xb, covariancia = S, SQP = W)
+ }
> n <- 1000
> p <- 5
> library(mvtnorm) # Para gerarmos dados da normal multivariada
> x <- rmvnorm(n,matrix(0, p, 1), diag(p)) # simular da normal pentavariada
> medcov(x)

$vetmedia
[1]  4.363227e-02 -5.117586e-03 -2.581255e-05 -6.267990e-02  8.289320e-04

$covariancia
             [,1]         [,2]         [,3]         [,4]        [,5]
[1,]  1.005835821 -0.002244794  0.015719671  0.026609861  0.02347450
[2,] -0.002244794  1.033590238 -0.042214692 -0.007294192 -0.03582641
[3,]  0.015719671 -0.042214692  1.021891307 -0.004031706 -0.04419991
[4,]  0.026609861 -0.007294192 -0.004031706  0.934869709  0.03512129
[5,]  0.023474503 -0.035826415 -0.044199912  0.035121286  1.03653640

$SQP
            [,1]        [,2]        [,3]       [,4]       [,5]
[1,] 1004.829986   -2.242550   15.703952  26.583251   23.45103
[2,]   -2.242550 1032.556647  -42.172477  -7.286898  -35.79059
[3,]   15.703952  -42.172477 1020.869416  -4.027675  -44.15571
[4,]   26.583251   -7.286898   -4.027675 933.934839   35.08617
[5,]   23.451028  -35.790588  -44.155712  35.086165 1035.49986

> var(x)            # comparar com resultado da nossa função

             [,1]         [,2]         [,3]         [,4]        [,5]
[1,]  1.005835821 -0.002244794  0.015719671  0.026609861  0.02347450
[2,] -0.002244794  1.033590238 -0.042214692 -0.007294192 -0.03582641
[3,]  0.015719671 -0.042214692  1.021891307 -0.004031706 -0.04419991
[4,]  0.026609861 -0.007294192 -0.004031706  0.934869709  0.03512129
[5,]  0.023474503 -0.035826415 -0.044199912  0.035121286  1.03653640

> apply(x, 2, mean) # comparar com resultado da nossa função

[1]  4.363227e-02 -5.117586e-03 -2.581255e-05 -6.267990e-02  8.289320e-04
```

Felizmente, devido ao R usar precisão dupla e utilizar algoritmos de ótima qualidade para estas tarefas, não precisaremos nos preocupar com a implementação de funções como as apresentadas neste capítulo. Mas se formos utilizar um compilador da linguagem Pascal, Fortran ou C e C^{++}, deveremos nos atentar para estes algoritmos, pois somente assim teremos elevada precisão, principalmente se tivermos lidando com dados de grande magnitude ou muito próximos de zero.

5.4 Exercícios

5.4.1 Mostrar a equivalência existente entre as expressões (5.2.4) e (5.2.5).

5.4.2 Implementar em R uma função para obtermos a soma de produtos, equação (5.2.6), e covariância entre as n observações de duas variáveis e cujos valores estão dispostos em dois vetores $\boldsymbol{X}$ e $\boldsymbol{Y}$. Criar uma matriz com n linhas e $p = 2$ colunas de algum exemplo e utilizar a função *medcov()* para comparar os resultados obtidos.

5.4.3 Os coeficientes de assimetria e curtose amostrais são funções das somas de potências dos desvios em relação a média. O coeficiente de assimetria é dado por $\sqrt{b_1} = (W3_n/n)/(W2_n/n)^{3/2}$ e o coeficiente de curtose é $b_2 = (W4_n/n)/(W2_n/n)^2$. Implementar uma função R, utilizando a função *medsqk* para estimar o coeficiente de assimetria e curtose univariados.

5.4.4 Utilizar uma amostra em uma área de sua especialidade e determinar a média, variância, soma de quadrados, soma de desvios ao cubo e na quarta potência. Determinar também os coeficientes de assimetria e curtose.

Capítulo 6

Aproximação de Distribuições

Algoritmos para a obtenção de probabilidades foram alvos de pesquisas de muitas décadas e ainda continuam sendo. A importância de se ter algoritmos que sejam rápidos e precisos é indescritível. A maior dificuldade é que a maioria dos modelos probabilísticos não possui função de distribuição explicitamente conhecida. Assim, métodos numéricos sofisticados são exigidos para calcularmos probabilidades. Um outro aspecto é a necessidade de invertermos as funções de distribuição para obtermos quantis da variável aleatória para uma probabilidade acumulada conhecida. Nos testes de hipóteses e nos processos de estimação por intervalo e por região quase sempre utilizamos estes algoritmos indiretamente sem nos darmos conta disso.

Neste capítulo vamos introduzir estes conceitos e apresentar algumas ideias básicas de métodos gerais para realizarmos as quadraturas necessárias. Métodos numéricos particulares serão abordados para alguns poucos modelos. Também abordaremos separadamente os casos discreto e contínuo. Para finalizarmos apresentaremos as principais funções pré-existentes do R para uma série de modelos probabilísticos.

6.1 Introdução

Não temos muitas preocupações com a obtenção de probabilidades e quantis se estamos utilizando o R, pois a maioria dos modelos probabilísticos e funções especiais já está implementada. Nosso objetivo está além deste fato, pois apesar de estarmos utilizando o ambiente R, nossa intenção é buscar os conhecimentos necessários para que possamos ir adiante e para que entendamos como determinada função opera e quais são suas potencialidades e limitações.

Vamos iniciar nossa discussão com a distribuição exponencial para chamarmos a atenção para a principal dificuldade existente neste processo. Assim, escolhemos este modelo justamente por ele não apresentar tais dificuldades. Seja uma variável aleatória X com distribuição exponencial com parâmetro λ, então a função de densidade é dada por:

$$f(x) = \lambda e^{-\lambda x}, \quad x > 0 \tag{6.1.1}$$

e a função de distribuição exponencial é

$$F(x) = 1 - e^{-\lambda x}. \tag{6.1.2}$$

Para este modelo probabilístico podemos calcular probabilidades utilizando a função de distribuição (6.1.2) e obter quantis com a função de distribuição inversa que é dada por:

$$q = F^{-1}(p) = \frac{-\ln(1-p)}{\lambda}. \tag{6.1.3}$$

Implementamos, em R, as funções de densidade, de distribuição e inversa da distribuição e as denominamos *pdfexp*, *cdfexp* e *icdfexp*, respectivamente. Estas funções são:

```
> # função de densidade exponencial
> # f(x) = lambda*exp(-lambda * x)
> pdfexp <- function(x, lambda = 1)
+ {
+    if (any(x < 0)) stop("Elementos de x devem ser todos não-negativos!")
+    fx <- lambda * exp(-lambda * x)
+    return(fx)
+ }
> # função de distribuição exponencial
> # F(x) = 1 - exp(-lambda * x)
> cdfexp <- function(x, lambda = 1)
+ {
+    if (any(x < 0)) stop("Elementos de x devem ser todos positivos!")
+    Fx <- 1 - exp(-lambda * x)
+    return(Fx)
+ }
```

```
> # função inversa da distribuição exponencial
> # q = -log(1 - p) / lambda
> icdfexp <- function(p, lambda = 1)
+ {
+     if (any(p >= 1) | any(p < 0))
+     stop("Elementos de p devem estar entre 0 (inclusive) e 1!")
+     q <- -log(1 - p) / lambda
+     return(q)
+ }
> lambda <- 0.1
> x <- 29.95732
> p <- 0.95
> pdfexp(x, lambda)

[1] 0.005000001

> cdfexp(x, lambda)

[1] 0.95

> icdfexp(p, lambda)

[1] 29.95732
```

Estas três funções foram facilmente implementadas, pois conseguimos explicitamente obter a função de distribuição e sua inversa. Se por outro lado tivéssemos o modelo normal

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \tag{6.1.4}$$

não poderíamos obter explicitamente a função de distribuição e muito menos a função inversa da função de distribuição de probabilidade. Como para a grande maioria dos modelos probabilísticos encontramos os mesmos problemas, necessitamos de métodos numéricos para obter estas funções. Por essa razão iremos apresentar alguns detalhes neste capítulo sobre alguns métodos gerais e específicos de alguns modelos. No caso discreto podemos utilizar algoritmos sequenciais, porém como boa parte dos modelos probabilísticos possuem relação exata com modelos contínuos, encontramos os mesmos problemas.

6.2 Modelos Probabilísticos Discretos

Vamos apresentar algoritmos para obtenção da função de probabilidade, função de distribuição de probabilidade e sua inversa para os modelos discretos que julgamos mais importantes, os modelos binomial e Poisson. Vamos iniciar nossa discussão pelo modelo binomial. Seja uma variável aleatória X com distribuição binomial, então a sua função de probabilidade é dada por:

$$P(X = x) = \binom{n}{x} p^x (1-p)^{(n-x)} \tag{6.2.1}$$

em que n é o tamanho da amostra ou números de ensaios de Bernoulli independentes com probabilidade de sucesso p e x é o número de sucessos, que pertence ao conjunto finito $0, 1, \cdots, n$. A função de distribuição de probabilidade é dada por:

$$F(x) = \sum_{t=0}^{x} \binom{n}{t} p^t (1-p)^{n-t}. \tag{6.2.2}$$

Vimos no capítulo 3, equação (3.3.10) que podemos obter as probabilidades acumuladas de forma recursiva. Sendo $P(X = 0) = (1-p)^n$, obtemos as probabilidades para os demais valores de X, ou seja, para $x = 2, \cdots, n$ de forma recursiva utilizando a relação $P(X = x) = P(X = x-1)[(n-x+1)/x][p/(1-p)]$. Este mesmo algoritmo apropriadamente modificado é utilizado para obtermos as probabilidades acumuladas e a inversa da função de distribuição. Se os valores de n e de p forem grandes, este algoritmo pode ser ineficiente. Para o caso do parâmetro p ser grande (próximo de um) podemos utilizar a propriedade da binomial dada por: se $X \sim Bin(n, p)$, então $Y = n - X \sim Bin(n, 1-p)$. Assim, podemos, por exemplo, obter $P(X = x)$ de forma equivalente por $P(Y = n - x)$ e $P(X \leq x) = P(Y \geq n - x)$, que pode ser reescrito por $F_X(x) = 1 - F_Y(n - x - 1)$, exceto para $x = n$, em que $F_X(x) = 1$. Desta forma trocamos de variável para realizar o cálculo e retornamos o valor correspondente a do evento original.

```
> # função de probabilidade e distribuição da binomial(n, p)
>
> pfcdfbinom  <- function(x, n, p)
+ {
+   if (p>0.5)
```

```
+   {
+     pp <- 1 - p
+     x  <- n - x
+   } else pp <- p
+   q <- 1 - pp
+   qn <- q^n
+   r  <- pp / q
+   g  <- r * (n + 1)
+   if (x >= 0) f <- qn else f <- 0
+   cdf <- f
+   if (x > 0)
+   {
+     u <- 0
+     while (u < x)
+     {
+       u   <- u + 1
+       f   <- f * (g / u - r)
+       cdf <- cdf + f
+     }  # while
+   } # if x > 0
+   if (p > 0.5) cdf <- 1 - cdf + f
+   return(list(pf = f, cdf = cdf))
+ } # função
> # inversa da função distribuição binomial(n, p)
>
> icdfbinom <- function(prob, n, p)
+ {
+   q  <- 1 - p
+   qn <- q^n
+   r  <- p / q
+   g  <- r * (n + 1)
+   x  <- 0
+   f  <- qn
+   u  <- prob
+   while ((u - f) >= 1e-11)
+   {
+     u <- u - f
+     x <- x + 1
+     f <- f * (g / x - r)
+   }  # while
+   return(x)
+ }
```

```
> # Exemplo de uso
> p <- 0.5; x <- 3; n <- 4
> prob <- pfcdfbinom(x, n, p)$cdf
> prob

[1] 0.9375

> icdfbinom(prob, n, p)

[1] 3
```

A função de distribuição binomial possui uma relação exata com a função distribuição beta. Se tivermos algoritmos para obter a beta incompleta podemos utilizar esta relação para obter probabilidades da distribuição binomial. A função de distribuição binomial, $F(x)$, se relaciona com a função beta incompleta, $Ip(\alpha,\beta)$, da seguinte forma:

$$F(x;n,p) = \begin{cases} 1 - I_p(x+1,n-x) & \text{se } 0 \leq x < n \\ 1 & \text{se } x = n. \end{cases} \tag{6.2.3}$$

Desta forma podemos antever a importância de conhecermos algoritmos de integração numérica das distribuições contínuas. Isso fica mais evidente quando percebemos que para grandes valores de n os algoritmos recursivos são ineficientes e podem se tornar imprecisos. Na seção 6.3 apresentaremos alguns métodos gerais de de integração para funções contínuas. No *script* seguinte utilizamos a relação (6.2.3) para a obtenção da função de distribuição da binomial.

```
> # função de probabilidade e distribuição da binomial(n, p)
> # a partir da relação com a função beta incompleta
>
> Fbinbeta <- function(x, n, p)
+ {
+    if ((p <= 0) | (p >= 1)) stop("O parâmetro p deve estar entre 0 e 1!")
+    if (x < n)
+    {
+        alpha <- x + 1
+        beta  <- n - x
+        cdf   <- 1 - pbeta(p, alpha, beta)
+    } else cdf <- 1
+    return(cdf)
+ }
```

```
> # exemplo de uso
> p <- 0.5
> n <- 4
> x <- 3
> Fbinbeta(x, n, p)

[1] 0.9375

> pbinom(x, n, p)

[1] 0.9375
```

A distribuição Poisson é a segunda que consideraremos. Existe relações da função de distribuição da Poisson com a gama incompleta, porém utilizaremos o método recursivo para obtermos função de probabilidade, a função de distribuição e inversa da função de distribuição. Se uma variável aleatória discreta X com valores $x = 1, 2, 3, \cdots$ possui distribuição Poisson com parâmetro λ, então podemos definir a função de probabilidade por

$$P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!} \tag{6.2.4}$$

Podemos obter probabilidades desta distribuição utilizando a fórmula recursiva

$$P(X = x) = P(X = x - 1)\frac{\lambda}{x}, \tag{6.2.5}$$

para valores de $x \geq 1$ e começando de $P(X = 0) = e^{-\lambda}$. Assim, da mesma forma que fizemos com a binomial, implementamos funções em R para obtermos a função de probabilidade, a função de distribuição e a sua inversa utilizando esta fórmula recursiva.

```
> # função de probabilidade e distribuição da Poisson(lamb)
>
> pfcdfpoi <- function(x, lamb)
+ {
+   qn  <- exp(-lamb)
+   if (x >= 0) f <- qn else f <- 0
+   cdf <- f
+   if (x > 0)
+   {
+     u <- 0
+     while (u < x)
```

```
+     {
+        u   <- u + 1
+        f   <- f * lamb / u
+        cdf <- cdf + f
+     }  # while
+   } # if x>0
+   return(list(pf = f, cdf = cdf))
+ }
> # Inversa da função distribuição Poisson(lamb)
>
> icdfpoi <- function(p, lamb)
+ {
+   qn <- exp(-lamb)
+   x  <- 0
+   f  <- qn
+   u  <- p
+   while ((u - f) >= 1e-11)
+   {
+     u <- u - f
+     x <- x + 1
+     f <- f * lamb / x
+   }  # while
+   return(x)
+ }
> # exemplos de utilização
>
> lamb <- 3
> x    <- 5
> p    <- 0.95
> pfcdfpoi(x, lamb)


$pf
[1] 0.1008188

$cdf
[1] 0.916082


> icdfpoi(p, lamb)


[1] 6
```

6.3 Modelos Probabilísticos Contínuos

Para obtermos a função de distribuição ou a inversa da função de distribuição de modelos probabilísticos contínuos via de regra devemos utilizar métodos numéricos. Existem exceções como, por exemplo, o modelo exponencial descrito no início deste capítulo. A grande maioria dos modelos probabilísticos utilizados atualmente faz uso de algoritmos especialmente desenvolvidos para realizar quadraturas em cada caso particular. Estes algoritmos são, em geral, mais precisos do que os métodos de quadraturas, como são chamados os processos de integração numérica. Existem vários métodos de quadraturas numéricas como o método de Simpson, as quadraturas gaussianas e os métodos de Monte Carlo. Vamos apresentar apenas os métodos de Simpson e de Monte Carlo, que embora sejam mais simples, são menos precisos.

Vamos utilizar o modelo normal para exemplificar o uso desses métodos gerais de integração nas funções contínuas de probabilidades. Uma variável aleatória X com distribuição normal com média μ e variância σ^2 possui função densidade dada por:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}. \tag{6.3.1}$$

A função de distribuição de probabilidade não pode ser obtida explicitamente e é definida por:

$$F(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(t-\mu)^2}{2\sigma^2}\right\} dt. \tag{6.3.2}$$

Em geral utilizamos a normal padrão para aplicarmos as quadraturas. Neste caso a média é $\mu = 0$ e a variância é $\sigma^2 = 1$. Assim, representamos frequentemente a densidade por $\phi(z)$, a função de distribuição por $\Phi(z)$ e a sua inversa por $\Phi^{-1}(p)$, em que $0 < p < 1$. A variável Z é obtida de uma transformação linear de uma variável X normal por $Z = (X - \mu)/\sigma$.

Vamos apresentar de forma bastante resumida a regra trapezoidal estendida para realizarmos quadraturas de funções. Seja f_i o valor da função alvo no ponto x_i, ou seja, $f_i = f(x_i)$, então a regra trapezoidal é dada por:

$$\int_{x_1}^{x_n} f(x)dx = \frac{h}{2}(f_1 + f_2) + O(h^3 f''), \tag{6.3.3}$$

em que $h = x_n - x_1$ e o termo de erro $O(h^3 f'')$ significa que a verdadeira resposta difere da estimada por uma quantidade que é o produto de h^3 pelo valor da segunda derivada da função avaliada em algum ponto do intervalo de integração determinado por x_1 e x_n, sendo $x_1 < x_n$.

Esta equação retorna valores muito pobres para as quadraturas da maioria das funções de interesse na estatística. Mas se utilizarmos esta função $n-1$ vezes para fazer a integração nos intervalos (x_1, x_2), (x_2, x_3), $\cdots$, (x_{n-1}, x_n) e somarmos os resultados, obteremos a fórmula composta ou a regra trapezoidal estendida por:

$$\int_{x_1}^{x_n} f(x)dx = h\left(\frac{f_1}{2} + f_2 + f_3 + \cdots + f_{n-1} + \frac{f_n}{2}\right) + O\left(\frac{(x_n - x_1)^3 f''}{n^2}\right). \quad (6.3.4)$$

em que $h = (x_n - x_1)/(n-1)$

A melhor forma de implementar a função trapezoidal é discutida e apresentada por Press et al. (1992)[13]. Nesta implementação inicialmente é tomada a média da função nos seus pontos finais de integração x_1 e x_n. São realizados refinamentos sucessivos. No primeiro estágio devemos acrescentar o valor da função avaliada no ponto médio dos limites de integração e no segundo estágio, os pontos intermediários 1/4 e 3/4 são inseridos e assim sucessivamente. Sejam *func()* a função de interesse, a e b os limites de integração e n o número de intervalos de integração previamente definido, então podemos obter a função *trapzd()* adaptando a mesma função implementada em fortran por Press et al. (1992)[13] da seguinte forma:

```
> # Esta rotina calcula o n-ésimo estágio de refinamento da regra
> # de integração trapezoidal estendida. em que, func é uma função externa
> # de interesse que deve ser chamada para n=1, 2, etc. e o valor
> # de s deve ser retornado a função
> # em cada nova chamada.
>
> trapzd <- function(func, a, b, s, n)
+ {
+     if (n == 1)
+     {
+        s <- 0.5 * (b - a) * (func(a) + func(b))
+     }  else
+     {
+        it  <- 2^(n - 2)
+        del <- (b - a) / it #espaço entre pontos a ser acrescentado.
```

```
+        x   <- a + 0.5 * del
+        sum <- 0.0
+        for (j in 1:it)
+        {
+           sum <- sum + func(x)
+           x   <- x + del
+        }
+        # troca s pelo seu novo valor refinado
+        s <- 0.5 * (s + (b - a) * sum / it)
+     }  # if then else
+     return(s)
+ }
> # função para executar quadraturas de funções definidas em func()
> # até que uma determinada precisão tenha sido alcançada
>
> qtrap <- function (func, a, b)
+ {
+   EPS  <- 1.e-9
+   jmax <- 20
+   olds <- -1.e30 # impossível valor para a quadratura inicial
+   j    <- 1
+   fim  <- 0
+   repeat
+   {
+       s <- trapzd(func, a, b, s, j)
+       # evitar convergência prematura espúria
+       if (j > 5)
+       {
+         if ((abs(s-olds)<EPS*abs(olds)) | ((s<0.0) & (olds==0.0))) fim <- 1
+       }
+       olds <- s
+       if (fim == 1) break
+       j <- j  + 1
+       if (j > jmax) break
+   }
+   if (j > jmax)  stop("Limite de passos ultrapassado!")
+   return(s)
+ }
```

Devemos chamar a função *qtrap()* especificando a função de interesse *func()* e os limites de integração. Assim, para a normal padrão devemos utilizar as seguintes funções:

```
> # função de densidade de probabilidade normal padrão
>
> fdpnorm <- function(x)
+ {
+     fx <- (1 / (2 * pi)^0.5) * exp(-x^2 / 2)
+     return(fx)
+ }
> #função de distribuição de probabilidade normal padrão
>
> cdfnorm <- function(func,x)
+ {
+   Fx <- qtrap(func, 0, abs(x))
+   if (x > 0) Fx <- 0.5 + Fx  else
+   Fx <- 0.5 - Fx
+   return(Fx)
+ }
> # exemplo de uso
>
> z <- 1.96
> p <- cdfnorm(fdpnorm, z)
> p

[1] 0.9750021
```

É evidente que temos métodos numéricos gerais mais eficientes do que o apresentado. As quadraturas gaussianas representam uma classe de métodos que normalmente são mais eficientes que este apresentado . Não vamos nos atentar para estes métodos gerais, por duas razões básicas. A primeira é que existem métodos específicos para obtermos as quadraturas dos principais modelos probabilísticos que são mais eficientes. A maior eficiência destes métodos específicos se dá por dois aspectos: velocidade de processamento e precisão. A segunda razão refere-se ao fato de que no R estas rotinas específicas já estão implementadas. Como ilustração, podemos substituir a função que implementamos *cdfnorm()* pela pré-existente no R *pnorm()*. Vamos ilustrar um destes algoritmos especializados para obtermos a função de distribuição da normal padrão. O algoritmo de Hasting possui erro máximo de 1×10^{-6} e é dado por:

$$\Phi(x) = \begin{cases} G & \text{se } x \leq 0 \\ 1 - G & \text{se } x > 0 \end{cases} \tag{6.3.5}$$

sendo G dado por

$$G = (a_1\eta + a_2\eta^2 + a_3\eta^3 + a_4\eta^4 + a_5\eta^5)\phi(x)$$

em que

$$\eta = \frac{1}{1 + 0{,}2316418|x|}$$

e $a_1 = 0{,}319381530$, $a_2 = -0{,}356563782$, $a_3 = 1{,}781477937$, $a_4 = -1{,}821255978$ e $a_5 = 1{,}330274429$.

O resultado da implementação deste procedimento é:

```
> #  CDF da normal padrão - aproximação de Hasting
>
> hcdfnorm <- function(x)
+ {
+    eta <- 1 / (1 + abs(x) * 0.2316418)
+    a1  <- 0.319381530; a2 <- -0.356563782
+    a3  <- 1.781477937; a4 <- -1.821255978
+    a5  <- 1.330274429
+    phi <- 1 / (2 * pi)^0.5 * exp(-x * x / 2)
+    G   <- (a1*eta + a2*eta^2 + a3*eta^3 + a4*eta^4 + a5*eta^5) * phi
+    if (x <= 0) Fx <- G
+    else        Fx <- 1 - G
+    return(Fx)
+ }
> # exemplo de uso
>
> z <- 1.96
> p <- hcdfnorm(z)
> p
[1] 0.9750022
```

Com o uso desta função ganhamos em precisão, principalmente para grandes valores em módulo do limite de integração superior e principalmente ganhamos em tempo de processamento. Podemos ainda abordar os métodos de Monte Carlo, que são especialmente úteis para integrarmos funções complexas e multidimensionais. Vamos apresentar apenas uma versão bastante rudimentar deste método. A ideia é determinar um retângulo que engloba a função que desejamos integrar e bombardearmos a região com pontos aleatórios (u_1, u_2) provenientes da distribuição uniforme.

Contamos o número de pontos sob a função e determinamos á area correspondente, a partir da proporcionalidade entre este número de pontos e o total de pontos simulados em relação a área sob a função na região de interesse em relação à área total do retângulo. Se conhecemos o máximo da função f_{max}, podemos determinar este retângulo completamente. Assim, o retângulo de interesse fica definido pela base (valor entre 0 e z_1 em módulo, sendo z_1 fornecido pelo usuário) e pela altura (valor da densidade no ponto de máximo). Assim, a área deste retângulo é $A = |z_1| f_{max}$. No caso da normal padrão, o máximo obtido para $z = 0$ é $f_{max} = 1/\sqrt{2\pi}$ e a área do retângulo $A = |z_1|/\sqrt{2\pi}$. Se a área sob a curva, que desejamos estimar, for definida por A_1, podemos gerar números uniformes u_1 entre 0 e $|z_1|$ e números uniformes u_2 entre 0 e f_{max}. Para cada valor u_1 gerado calculamos a densidade $f_1 = f(u_1)$. Assim, a razão entre as áreas A_1/A é proporcional a razão n/N, em que n representa o número de pontos (u_1, u_2) para os quais $u_2 \leq f_1$ e N o número total de pontos gerados. Logo, a integral é obtida por $A_1 = |z_1| \times f_{max} \times n/N$, em que z_1 é o valor da normal padrão para o qual desejamos calcular a área que está compreendida entre 0 e $|z_1|$, para assim obtermos a função de distribuição no ponto z_1, ou seja, para obtermos $\Phi(z_1)$. Assim, se $z_1 \leq 0$, então $\Phi(z_1) = 0{,}5 - A_1$ e se $z_1 > 0$, então $\Phi(z_1) = 0{,}5 + A_1$. Para o caso particular da normal padrão, implementamos a seguinte função:

```
> # Quadratura da normal padrão via simulação Monte Carlo
>
> mcdfnorm <- function(x, nMC)
+ {
+    max <- 1 / (2 * pi)^0.5
+    z   <- abs(x)
+    u1  <- runif(nMC) * z                    # 0 < u1 < z
+    u2  <- runif(nMC) * max                  # 0 < u2 < max
+    f1  <- (1 / (2 * pi)^0.5) * exp(-u1^2 / 2) # f1(u1) - normal padrão
+    n   <- length(f1[u2 <= f1])
+    G   <- n / nMC * max * z
+    if (x < 0) Fx <- 0.5 - G else Fx <- 0.5 + G
+    return(Fx)
+ }
> # exemplo de uso
>
> mcdfnorm(1.96, 1500000)

[1] 0.9752265
```

Outra forma de obtermos uma aproximação da integral

$$\int_0^{abs(z)} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt = \int_0^{abs(z)} \phi(t) dt$$

por Monte Carlo é gerarmos m números uniformes entre 0 e $abs(z)$, digamos z_1, z_2, ..., z_m e obter

$$\int_0^{abs(z)} \phi(t) dt \approx \Delta z \left[\frac{1}{m} \sum_{i=1}^{m} \phi(z_i) \right],$$

em que $\phi(t) = 1/\sqrt{2\pi} \times \exp\{-t^2/2\}$, e a função densidade normal padrão avaliada no ponto t e $\Delta z = abs(z) - 0$. A ordem de erro desse processo é dada por $O(m^{-1/2})$. O programa R para obter o valor da função de distribuição normal padrão $\Phi(z)$ utilizando essas ideias é apresentado a seguir. Por meio de uma comparação dessa alternativa Monte Carlo com a primeira podemos verificar que houve uma grande diminuição do erro de Monte Carlo no cálculo dessa integral, nessa nova abordagem. Muitas variantes e melhorias nesse processo podem ser implementadas, mas nós não iremos discuti-las aqui.

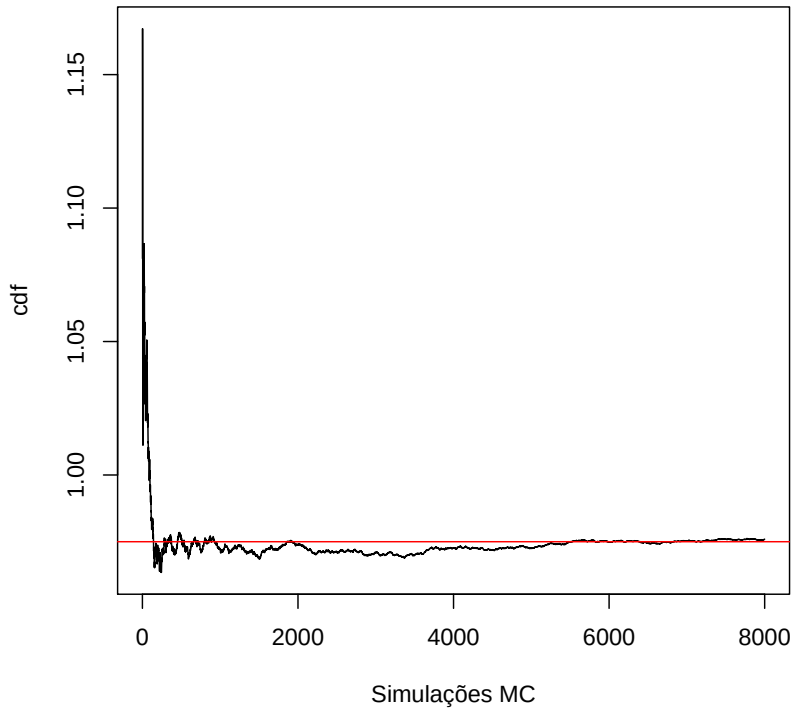
```
> # Quadratura da normal padrão via simulação Monte Carlo
> # Segunda forma de obter tal integral: forma clássica
> cdfnorm <- function(z, m)
+ {
+    x <- runif(m,0,abs(z))
+    cdf <- (1/(2*pi)^0.5)*exp(-(x^2)/2)
+    cdf <- abs(z)*mean(cdf)
+    if (z < 0) cdf <- 0.5 - cdf else
+        cdf <- 0.5 + cdf
+    return(cdf)
+ }
> # Exemplo
> m <- 8000   # número de pontos muito inferior ao caso anterior
> z <- 1.96
> cdfnorm(z, m)  # Estimativa de Monte Carlo

[1] 0.9728296

> pnorm(z)       # valor real utilizando diretamente o R

[1] 0.9750021
```

```
> # Ordem de erro: O(m^(-1/2))
> 1/m^0.5
[1] 0.01118034
> # verificação da convergência
> x <- runif(m, 0, abs(z))
> pdf <- (1/(2*pi)^0.5)*exp(-(x^2)/2)
> if (z < 0) cdf <- 0.5 - abs(z)*cumsum(pdf)/(1:m) else
+         cdf <- 0.5 + abs(z)*cumsum(pdf)/(1:m)
> plot(cdf,type="l",xlab="Simulações MC")
> abline(a=pnorm(z), b=0, col="red") # linha com o valor real
```



Vamos apresentar o método numérico de Newton-Raphson para obtermos a solução da equação

$$z = \Phi^{-1}(p),$$

em que $0 < p < 1$ é o valor da função de distribuição da normal padrão, $\Phi^{-1}(p)$, o valor da inversa da função de distribuição normal padrão no ponto p e z o quantil

correspondente, que queremos encontrar dado um valor de p. Podemos apresentar esse problema por meio da seguinte equação

$$\Phi(z) - p = 0,$$

em que $\Phi(z)$ é a função de distribuição normal padrão avaliada em z. Assim, desejamos encontrar os zeros da função $f(z) = \Phi(z) - p$. Em geral, podemos resolver essa equação numericamente utilizando o método de Newton-Raphson, que se trata de um processo iterativo. Assim, devemos ter um valor inicial para o quantil para iniciarmos o processo e no $(n+1)$-ésimo passo do processo iterativo podemos atualizar o valor do quantil por

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}, \tag{6.3.6}$$

em que $f'(z_n)$ é derivada de primeira ordem da função para a qual queremos obter as raízes avaliada no ponto z_n. Para o caso particular temos que

$$z_{n+1} = z_n - \frac{[\Phi(z_n) - p]}{\phi(z_n)}, \tag{6.3.7}$$

sendo $\phi(z_n)$ a função densidade normal padrão. Como valores iniciais usaremos uma aproximação grosseira, pois nosso objetivo é somente demonstrar o método. Assim, se p for inferior a 0,5 utilizaremos $z_0 = -0{,}1$, se $p > 0{,}5$, utilizaremos $z_0 = 0{,}1$. Obviamente se $p = 0$, a função deve retornar $z = 0$. A função *qPhi* a seguir retorna os quantis da normal, dados os valores de p entre 0 e 1, da média real μ e da variância real positiva σ^2, utilizando para isso o método de Newton-Raphson e a função *hcdfnorm* de Hasting para obtermos o valor da função de distribuição normal padrão.

```
> # função auxiliar para retornar o valor da
> # função densidade normal padrão
> phi <- function(z)
+ {
+    return(1 / (2 * pi)^0.5 * exp(-z * z / 2))
+ }
> # Método de Newton-Raphson para obtermos quantis da distribuição normal
```

```
> # com média real mu e desvio padrão real positivo sig, dado 0 < p < 1
> # utiliza as funções phi(z) e  hcdfnorm(z), apresentadas anteriormente
>
> qPhi <- function(p, mu = 0, sig = 1)
+ {
+   if ((p <= 0) | (p >= 1)) stop("Valor de p deve estar entre 0 e 1!")
+   if (sig <= 0) stop("Desvio padrão deve ser maior que 0!")
+   if (abs(p - 0.5) <= 1e-8) z1 <- 0 else
+   {
+       if (p < 0.5) z0 <- -0.1 else z0 <- 0.1
+       it        <- 1
+       itmax     <- 2000
+       eps       <- 1e-8
+       convergiu <- FALSE
+       while (!convergiu)
+       {
+          z1 <- z0 - (hcdfnorm(z0) - p) / phi(z0)
+          if ((abs(z0 - z1) <= eps * z0) | (it > itmax)) convergiu <- TRUE
+          it <- it + 1
+          z0 <- z1
+       }
+   }
+   return(list(x = z1 * sig + mu, it = it - 1))
+ }
> # Exemplo de uso
>
> p   <- 0.975
> mu  <- 0
> sig <- 1
> qPhi(p, mu, sig)

$x
[1] 1.959963

$it
[1] 7

> qnorm(p, mu, sig) # pra fins de comparação

[1] 1.959964
```

Poderíamos, ainda, ter utilizado o método da secante, uma vez que ele não necessita da derivada de primeira ordem, mas precisa de dois valores iniciais para

iniciar o processo e tem convergência mais lenta. O leitor é incentivado a consultar Press et al. (1992)[13] para obter mais detalhes. Também poderia ter sido usada a função *cdfnorm*, que utiliza o método trapezoidal. Nesse caso a precisão seria maior, mas o tempo de processamento também é maior. Para isso bastaria substituir a linha de comando

```
> z1 <- z0 - (hcdfnorm(z0) - p) / phi(z0)
```

por

```
> z1 <- z0 - (cdfnorm(z0) - p) / phi(z0)
```

Felizmente o R também possui rotinas pré-programadas para este e para muitos outros modelos probabilísticos, que nos alivia da necessidade de programar rotinas para obtenção das funções de distribuições e inversas das funções de distribuições dos mais variados modelos probabilísticos existentes.

6.4 Funções Pré-Existentes no R

Na Tabela 3.1 apresentamos uma boa parte das funções de probabilidade ou densidade contempladas pelo R. Na segunda coluna foi apresentado o nome R, que se for precedido por "d", retornará a densidade (ou função de probabilidade), por "p", a função de distribuição e por "q", a função de distribuição inversa. Na terceira coluna especificamos os parâmetros que devem ser repassados como argumentos das funções. Todas estas funções são vetoriais, indicando que podemos obter tais quantidades dispostas em um vetor de dimensão n. O R também permite que utilizemos o prefixo "r" para gerarmos números aleatórios de cada distribuição, conforme foi apresentado no capítulo 3. Assim, podemos utilizar os comandos *dnorm(1.96)*, *pnorm(1.96)* e *qnorm(0.975)*, para obtermos a densidade, função de distribuição e inversa da distribuição normal padrão, respectivamente, dos argumentos utilizados nos exemplos.

6.5 Exercícios

6.5.1 Comparar a precisão dos três algoritmos de obtenção da função de distribuição normal apresentados neste capítulo. Utilizar a função *pnorm()* como referência.

6.5.2 Utilizar diferentes números de simulações Monte Carlo para integrar a função de distribuição normal e estimar o erro de Monte Carlo relativo e absoluto máximo cometidos em 30 repetições para cada tamanho. Utilize os quantis 1,00, 1,645 e 1,96 e a função *pnorm()* como referência.

6.5.3 A distribuição Cauchy é um caso particular da t de Student com $\nu = 1$ grau de liberdade. A densidade Cauchy é dada por:

$$f(x) = \frac{1}{\pi(1 + x^2)}$$

Utilizar o método trapezoidal estendido para implementar a obtenção dos valores da distribuição Cauchy. Podemos obter analiticamente também a função de distribuição e sua inversa. Obter tais funções e implementá-las no R. Utilize as funções R pré-existentes para checar os resultados obtidos.

Referências Bibliográficas

[1] ATKINSON, A. C.; PEARCE, M. C. The computer generation of beta, gamma and normal random variable. *Journal of the Royal Statistical Society, Series A*, 139(4):431–461, 1976. 46

[2] DACHS, J. N. W. *Estatística computacional. Uma introdução ao Turbo Pascal.* Livros Técnicos e Científicos, Rio de Janeiro, 1988. 236p. 19

[3] DEVROY, L. Generating the maximum of independent identically distributed random variables. *Computers and Mathematics with Applications*, 6:305–315, 1980. 49

[4] DEVROY, L. *Non-uniform random variate generation.* Springer-Verlag, New York, 1986. 843p. 49, 51

[5] FERREIRA, D. F. *Estatística Multivariada.* Editora UFLA, Lavras, 2008. 662p. 43

[6] JOHNSON, R. A.; WICHERN, D. W. *Applied multivariate statistical analysis.* Prentice Hall, New Jersey, 4th edition, 1998. 816p. 43

[7] KACHITVICHYANUKUL, V.; SCHMEISER, B. W. Binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988. 49, 51

[8] MARSAGLIA, G.; BRAY, T. A. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, 1964. 43

[9] MATSUMOTO, M.; NISHIMURA, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. 26

[10] McCULLOUGH, B. D.; WILSON, B. On the accuracy of statistical procedures in Microsoft Excel 97. *Computational Statistics and Data Analysis*, 31:27–37, 1999. 79

[11] NAYLOR, T. H. J.; BALINTFY, J. L.; BURDICK, D. S.; CHU, K. *Técnicas de simulação em computadores*. Vozes, Petrópolis, 1971. 402p. 19

[12] PARK, S. K.; MILLER, K. W. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988. 23

[13] PRESS, W. H.; FLANNERY, B. P.; TEUKOLSKY, S. A.; VETTERLING, W. T. *Numerical recipes in Fortran: the art of scientific computing*. Cambridge University Press, Cambridge, 1992. 994p. 21, 23, 43, 98, 107

[14] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0. 1

[15] SCHRAGE, L. A more portable Fortran random number generator. *ACM Transactions on Mathematical Software*, 5(2):132–138, 1979. 23, 24

[16] SMITH, W. B.; HOCKING, R. R. Algorithm AS 53: Wishart variate generator. *Journal of the Royal Statistical Society - Series C*, 21(3):341–345, 1972. 71

[17] WEST, D. H. D. Updating mean and variance estimates: an improved method. *ACM Transactions on Mathematical Software*, 22(9):532–535, Sept 1979. 80, 82

Índice Remissivo