Anotações de Aulas

Victor

16 de agosto de 2016

# MATF05 - Técnicas computacionais em estatística

## Aula 1 - 16.08.2016

Essa primeira aula trata de uma introdução ao R, apresentando as principais funções e funcionalidades do R.

Tópicos:

* Sobre o R
* O R como calculadora
* Armazenando objetos no R
* Vetores
* Lendo os dados
* Baixando os dados
* Matrizes
* Análise descritiva: data.frames
* Importando dados: read.table
* Distribuição de probabilidade
* Criando funções
* Análises de frequência
* Testes t
* Testes F

### Sobre o R

Usado para análise manipilação de dados e realização e cálculos e visualização de gráficos.

Presença de muitos pacotes, extensa comunidade estatistica.

Software Livre e disponível para download

### Instalação do R

**R**

<https://cran.r-project.org/bin/windows/base/>

**RStudio**

<https://www.rstudio.com/products/rstudio/#Desktop>

### R como calculadora

#### Calculadora simples

#Somar  
1+1

## [1] 2

#Subtrair  
14-6

## [1] 8

#Multiplicar  
3\*4

## [1] 12

#Dividir  
47/11

## [1] 4.272727

#Exponenciar  
2^5

## [1] 32

#### Calculadora científica

#Trigonométricas  
sin(pi)

## [1] 1.224606e-16

cos(pi)

## [1] -1

#Outras  
sqrt(9)

## [1] 3

abs(-10)

## [1] 10

exp(1)

## [1] 2.718282

log10(100)

## [1] 2

log(2.78)

## [1] 1.022451

### Armazenando valores no R

' <- ' ou ' = '

resultado <- 47/11  
  
resultado

## [1] 4.272727

#É possível operar com valores armazenados no objeto  
sin(resultado)

## [1] -0.9048957

#Distinção entre maiúsculas e minúsculas

**Obs1**.: O R utiliza, por *default*, o ponto '.' como separador de casas decimais.

**Obs2**.: A seta para cima, do console, recupera os comandos antigos

**Obs3**.: Usar Ctrl + l para limpar o console

### Vetores

Usar a função c() para concatenar os elementos em um vetor

#c()  
x <- c(1, 2, 3, 4, 5)  
x

## [1] 1 2 3 4 5

# ':' cria uma sequência  
x <- 1:5  
x

## [1] 1 2 3 4 5

#Vetores não precisam ser numéricos (strings)  
x <- c('Rio', "São Paulo", "Salvador")  
x

## [1] "Rio" "São Paulo" "Salvador"

#Booleans  
x <- c(TRUE, FALSE, FALSE, T)  
x

## [1] TRUE FALSE FALSE TRUE

#### Indexação de vetores

Utiliza-se o operador '[]' para selecionar objetos específicos em posições específicas de um vetor.

Usamos alguns vetores já programados originalmente no R (letters, LETTERS, month.name)

#Letras do alfabeto  
letters[1:6]

## [1] "a" "b" "c" "d" "e" "f"

LETTERS[c(4, 5, 6)]

## [1] "D" "E" "F"

month.name[6:12]

## [1] "June" "July" "August" "September" "October" "November"   
## [7] "December"

Podemos utilizar também operações lógicas para selecionar elementos do vetor, buscando os valores condicionalmente.

x <- c(0, 3, 3, 5, 1, 1:5, 12)  
  
x[x > 3]

## [1] 5 4 5 12

x[x != 3]

## [1] 0 5 1 1 2 4 5 12

#Podemos calcular o número de elementos de um vetor usando a função length()  
length(x[x>3])

## [1] 4

#### Operações com vetores

Por default, o R opera em vetores de forma element-wise.

x <- 1:5  
y <- 6:10  
  
#Soma  
x+y

## [1] 7 9 11 13 15

#Multiplicação  
x\*y

## [1] 6 14 24 36 50

#Divisão  
x/y

## [1] 0.1666667 0.2857143 0.3750000 0.4444444 0.5000000

### Lendo os dados

#### Manualmente

É possível criar um vetor usando as funções scan() e edit(). Para alterar os dados em y, basta usar a função fix().

Para efeito do markdown, não há como rodar essas funções

# y <- scan()  
# y <- edit(data.frame())  
# fix(y)

**Obs**.: Diversos bancos de dados estão disponíveis em <https://archive.ics.uci.edu/ml/datasets.html>

#### De um arquivo diretamente

A função read.table() pode ser utilizada para ler bancos de dados

df <- read.table("c:/Users/duranvr/Google Drive/UFBA/2016.2/Métodos Computacionais/dados/iris.txt", header = TRUE, sep = ",")

#### Criando variáveis usando um banco conhecido

Nomes <- c("João", "Maria", "Roberto", "Bruna", "Dolores", "Daniel", "Vinicius", "Alexandre")  
Peso <- c(70, 42, 91, 56, 58, 72, 81, 98)  
Altura <- c(1.65, 1.61, 1.89, 1.71, 1.68, 1.75, 1.69, 1.88)  
  
dados <- data.frame(Nomes, Peso, Altura)

Vamos criar uma variável BMI, que é dada pela fórmula

$$BMI=\frac{Peso,(Altura^2)}$$

#Criando a variávle IMC  
dados$BMI <- dados$Peso/((dados$Altura)^2)

Agora vamos criar uma variável do tipo fator que categoriza o peso de cada indivíduo

#Criando um vetor vazio  
dados$categoria <- NA  
  
#Preenchendo condicionalmente  
dados$categoria[dados$BMI < 20] <- "Abaixo"  
dados$categoria[dados$BMI <= 25 & dados$BMI >= 20] <- "Normal"  
dados$categoria[dados$BMI > 25] <- "Acima"  
  
#Utilizando a função ifelse()  
system.time(dados$categoria <- ifelse(dados$BMI < 20, "Abaixo", ifelse(dados$BMI <= 25, "Normal", "Acima")))

## user system elapsed   
## 0 0 0

#Utilizando os condicionais de programação  
for(i in 1:length(dados$categoria)){  
 dados$categoria[i] <- if(dados$BMI[i] < 20){  
 "Abaixo"  
 } else if(dados$BMI[i] <= 25){  
 "Normal"  
 } else {  
 "Acima"  
 }  
}

#### Matrizes

Lembramos que a matriz apenas comporta elementos com a mesma classe.

Primeiramente criamos a matriz apenas com os dados. Para substituir a apresentação genérica por uma nomeada, usamos a função dimnames(), que recebe como valor uma lista com um vetor para cada dimensão e cada vetor com um elemento para cada nível da dimensão

#Gerando os dados  
dados2 <- matrix(c(12, 23, 19, 43, 11, 25, 35, 65, 32, 24, 26, 24),  
 nrow = 3, ncol = 4, byrow = TRUE)  
  
#Nomeando as dimensões  
dimnames(dados2) <- list(c("João", "Pedro", "Maria"),  
 c("Exame1", "Exame2", "Exame3", "Exame4"))

Para indexar em uma matriz, precisamos lembrar que ela possui duas dimensões. Utilizamos o operador colchete mesmo, só que dessa vez com dois argumentos. Nesse formato:

matriz['linhas','colunas']

#Exibir todos os dados que se encontram na coluna 1  
dados2[,1]

## João Pedro Maria   
## 12 11 32

#Exibir o primeiro elemento da primeira linha  
dados2[1,1]

## [1] 12

Operações com matrizes

mat1 <- matrix(rep(1, 6), 2, 3)  
mat2 <- matrix(rep(2, 6), 2, 3)  
  
#Soma de elementos da matriz  
mat1+mat2

## [,1] [,2] [,3]  
## [1,] 3 3 3  
## [2,] 3 3 3

#Produtos de matrizes elemento a elemento  
mat1 \* mat2

## [,1] [,2] [,3]  
## [1,] 2 2 2  
## [2,] 2 2 2

#Para fazer multiplicação de matrizes, precisamos usar outro operador e transpor uma das matrizes  
mat1 %\*% t(mat2)

## [,1] [,2]  
## [1,] 6 6  
## [2,] 6 6

### Estatística básica

#### Descritivas básicas

Usando a função summary() em todo o banco

summary(dados)

## Nomes Peso Altura BMI   
## Alexandre:1 Min. :42.0 Min. :1.610 Min. :16.20   
## Bruna :1 1st Qu.:57.5 1st Qu.:1.673 1st Qu.:20.20   
## Daniel :1 Median :71.0 Median :1.700 Median :24.49   
## Dolores :1 Mean :71.0 Mean :1.732 Mean :23.34   
## João :1 3rd Qu.:83.5 3rd Qu.:1.782 3rd Qu.:26.22   
## Maria :1 Max. :98.0 Max. :1.890 Max. :28.36   
## (Other) :2   
## categoria   
## Length:8   
## Class :character   
## Mode :character   
##   
##   
##   
##

#### Correlação e covariância

grupo1 <- rnorm(100)  
grupo2 <- rnorm(100)  
grupo3 <- rnorm(100)  
grupo4 <- rnorm(100)  
  
dados3 <- data.frame(grupo1, grupo2, grupo3, grupo4)  
  
#Matriz de covariância  
cov(dados3)

## grupo1 grupo2 grupo3 grupo4  
## grupo1 0.88664967 0.19907576 -0.06067871 -0.04056088  
## grupo2 0.19907576 1.21755365 0.18165105 -0.04239804  
## grupo3 -0.06067871 0.18165105 1.15442703 0.06711658  
## grupo4 -0.04056088 -0.04239804 0.06711658 1.08227156

#Matriz de correlação  
cor(dados3)

## grupo1 grupo2 grupo3 grupo4  
## grupo1 1.00000000 0.19160124 -0.05997594 -0.04140600  
## grupo2 0.19160124 1.00000000 0.15321829 -0.03693462  
## grupo3 -0.05997594 0.15321829 1.00000000 0.06004523  
## grupo4 -0.04140600 -0.03693462 0.06004523 1.00000000

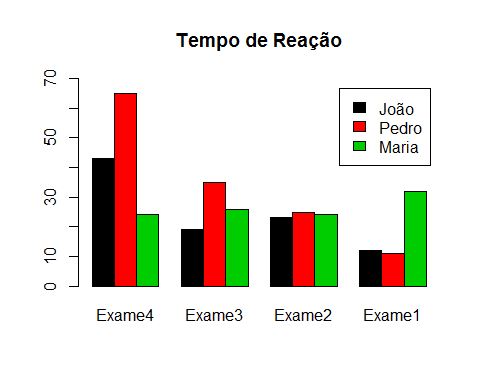
#Arredondando usando o round()  
round(cor(dados3), 3)

## grupo1 grupo2 grupo3 grupo4  
## grupo1 1.000 0.192 -0.060 -0.041  
## grupo2 0.192 1.000 0.153 -0.037  
## grupo3 -0.060 0.153 1.000 0.060  
## grupo4 -0.041 -0.037 0.060 1.000

#### Distribuição de frequência

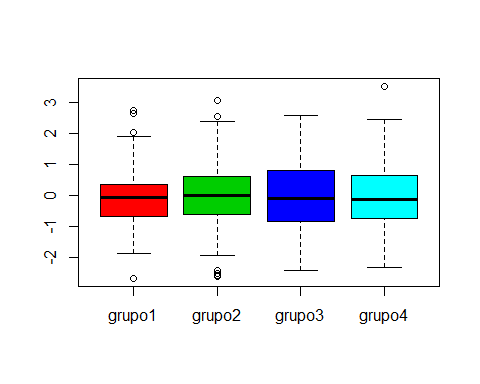
Todas as vantagens de um gráfico

barplot(dados2[,4:1], beside = TRUE, col = 1:3, legend = rownames(dados2), ylim = c(0, 70), main = "Tempo de Reação")



#### Boxplot

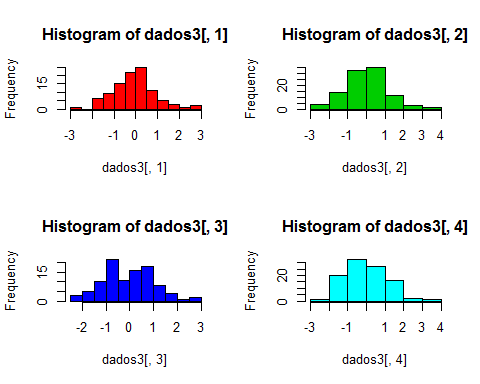
boxplot(dados3, col = 2:5)



#### Histogramas

Vamos atentar para a estrutura dos gráficos usando o par(mfrow = )

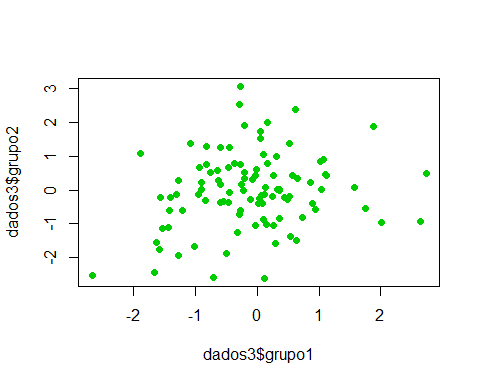
par(mfrow = c(2, 2))  
  
hist(dados3[,1], col = 2)  
hist(dados3[,2], col = 3)  
hist(dados3[,3], col = 4)  
hist(dados3[,4], col = 5)



par(mfrow = c(1, 1))

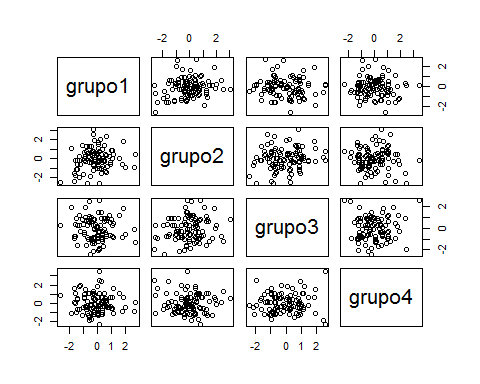
#### Gráficos de dispersão

plot(dados3$grupo1, dados3$grupo2, col = 3, pch = 19)



Para gerar vários gráficos de dispersão ao mesmo tempo por pares de variáveis.

pairs(dados3)



### Estatística inferencial

#### Análise de frequência - tabelas de contingência

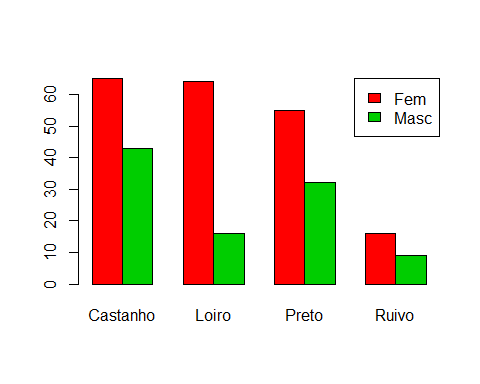
Gerando o banco

cor <- c(rep("Preto", 32+55),  
 rep("Castanho", 43+65),  
 rep("Loiro", 16+64),  
 rep("Ruivo", 9+16))  
  
sexo <- c(rep("Masc", 32), rep("Fem", 55),  
 rep("Masc", 43), rep("Fem", 65),  
 rep("Masc", 16), rep("Fem", 64),  
 rep("Masc", 9), rep("Fem", 16))  
  
dadosCabelo <- data.frame(sexo, cor)  
  
chisq.test(table(dadosCabelo))

##   
## Pearson's Chi-squared test  
##   
## data: table(dadosCabelo)  
## X-squared = 8.9872, df = 3, p-value = 0.02946

Observando as distribuições dos dados

barplot(table(dadosCabelo), beside = TRUE, col = c(2, 3))  
legend("topright", c("Fem", "Masc"), fill = c(2, 3))



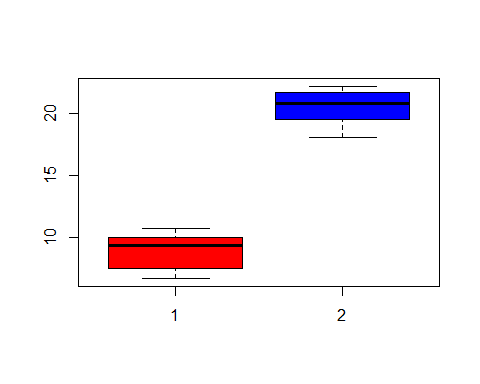
#### Teste T para uma amostra

set.seed(10)  
tempos <- rnorm(10, 45, 1)  
  
t.test(tempos, mu = 45, alternative = "greater")

##   
## One Sample t-test  
##   
## data: tempos  
## t = -2.2169, df = 9, p-value = 0.9731  
## alternative hypothesis: true mean is greater than 45  
## 95 percent confidence interval:  
## 44.10363 Inf  
## sample estimates:  
## mean of x   
## 44.50934

#### Teste T para duas amostras

set.seed(10)  
dieta1 <- rnorm(10, 10, 2)  
dieta2 <- rnorm(8, 20, 2)  
  
boxplot(dieta1, dieta2, col = c(2, 4))



t.test(dieta1, dieta2, alternative = "two.sided", var.equal = FALSE)

##   
## Welch Two Sample t-test  
##   
## data: dieta1 and dieta2  
## t = -17.069, df = 14.9, p-value = 3.44e-11  
## alternative hypothesis: true difference in means is not equal to 0  
## 95 percent confidence interval:  
## -12.99668 -10.10966  
## sample estimates:  
## mean of x mean of y   
## 9.018686 20.571854

### Criando funções

Para criar funções customizadas utilizamos a função 'function()'

Vamos criar uma função que seleciona letras aleatórias.

randomLetter <- function(n){  
 out <- sample(LETTERS, n, replace = TRUE)  
 return(out)  
}

Agora vamos criar uma função que seleciona n sequências de 10 números

randomSeq <- function(n){  
 out <- matrix(NA, n, 10)  
 for(i in 1:n){  
 out[i,] <- sample(0:9, 10, replace = TRUE)  
 }  
 return(out)  
}  
  
randomSeq(2)

## [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]  
## [1,] 8 9 6 5 2 2 0 7 2 1  
## [2,] 0 4 1 8 3 9 2 4 1 5

### Distribuições de probabilidade

O R inclui algumas distribuições básicas em que a gente pode calcular:

* d: densidade
* p: probabilidade acumulada num ponto
* q: o quantil em um determinado ponto
* r: gera amostras da distribuição

#### Distribuição normal

\_norm()

#Densidade no ponto 0  
dnorm(0)

## [1] 0.3989423

#usando a fórmula  
z <- 0  
myDens <- function(z, mu=0, sigma=1){  
 (1/sqrt(2\*pi\*sigma^2))\*exp((-1/(2\*sigma^2))\*(z-mu)^2)  
}  
  
myDens(0)

## [1] 0.3989423

#Caso a normal não seja padrão  
dnorm(-1, 2, 3)

## [1] 0.08065691

myDens(-1, 2, 3)

## [1] 0.08065691

#Probabilidade acumulada no ponto 0  
pnorm(0)

## [1] 0.5

myProb <- function(x){  
 return(x)  
}  
  
#Quantil associado à probabilidade .975  
qnorm(.975)

## [1] 1.959964

#Simula valores de uma normal 0, 1  
rnorm(10)

## [1] -0.10176101 -0.25378053 -1.85374045 -0.07794607 0.96856634  
## [6] 0.18492596 -1.37994358 -1.43551436 0.36208723 -1.75908675

#Para fixar a randomicidade da smilação set.seed()  
set.seed(1)  
rnorm(3)

## [1] -0.6264538 0.1836433 -0.8356286

set.seed(1)  
rnorm(3)

## [1] -0.6264538 0.1836433 -0.8356286

Outras distribuições disponíveis: Binomial (binom), Uniforme (unif), Poisson (pois)

## Aula 2 - 23.08.16

### Geração de números (pseudo)aleatórios - Simulação estocástica

O objetivo desse conteúdo é apresentar como obter números pseudoaleatórios para qualquer função densidade.

Simular é imitar, reproduzir a realidade. Útil para estudos em que não há dados, ou estudos de recuperação de parâmetros para verificar qualidade de estimadores. Fortalece o argumento de artigos.

Um computador digital não é capaz de gerar números aleatórios.

Os sinais podem ser analógicos ou digitais, analógico é contínuo e digital é discreto.

Bit é 0 ou 1, byte são 8 bits.

Trabalharemos na geração de números pseudoaleatórios, um processo determinístico.

John von Neumann (1951): "Anyone who considers arithmetical methods of producing random digits is of course in a state of sin."

Geração de números verdadeiramente aleatórios tem como base processos físicos como ruído térmico, efeito fotoelétrico, decaimento radioativo.

#### Geradores para a distribuição U[0,1]

É a distribuição base da qual pode-se chegar às outras. O objetivo é gerar v.a's i.i.d.

**Métodos**

* **Método congruencial** (linear misto - Lehumer, 1951):

Onde a operação opera divisão de inteiros do valor entre parêntesis por e retorna o resto; , e são parãmetros fixos.

Temos também um que é chamado de semente, ou raiz.

Note que

Quando temos o método congruencial linear multiplicativo.

*Limitações*: Além do problema da geração ser discreta, pode haver também periodicidade da cadeia.

Ex.:

Sendo o **período** o número de iterações que uma sequência leva para se repetir. Desejamos períodos longos para minimizar "buracos".

É importante então pensar em parâmetros , e que encontre o maior período possível.

Para obter os números entre 0 e 1 basta fazer

**Escolha dos parâmetros**

: Número próximo do número máximo da máquina (inteiro)

: Um valor alto qualquer exemplo,

Note que como excluimos o 0 e o m, o período máximo é m-1

congruencial <- function(x0, a, c, m = 2^64 - 1, n=100){  
 n = n+1  
 out <- numeric(n)  
 out[1] <- x0  
 for(i in 2:n){  
 out[i] <- (a\*out[i-1] + c)%%m  
 }  
 return(out[-1])  
}  
  
N <- 5  
checkPrimo <- function(N){  
 for(i in 2:(N-1)){  
 if((N)%%i == 0){  
 return(paste(N, "não é primo. É divisível por: ", i))  
 break  
 } else if(i == (N-1)){  
 return(paste(N, "é primo!"))  
 }  
}  
}  
  
  
checkPrimo2 <- function(N){  
 divs <- numeric(0)  
   
 for(i in 2:(N-1)){  
 if((N)%%i == 0){  
 divs <- c(divs, i)  
 }  
 }  
 if(length(divs) > 0){  
 return(paste(N, "não é primo. É divisível por: ", paste(divs, collapse=", ")))  
 } else {  
 return(paste(N, "é primo!"))  
 }  
}  
  
checkPrimo2(12314)

## [1] "12314 não é primo. É divisível por: 2, 47, 94, 131, 262, 6157"

##### Exemplo 2.1.

Se , , e , quais os 10 primeiros valores?

Os valores observados são: 15, 45, 135, 105, 15, 45, 135, 105, 15, 45

Para obter os valores uniformes basta dividir por .

0.1, 0.3, 0.9, 0.7, 0.1, 0.3, 0.9, 0.7, 0.1, 0.3

Note que embora o período máximo seja 149, temos um período observado de 4. A sequência dos gerados é chamada de sequência de Lehmer.

##### Exercício 2.1.

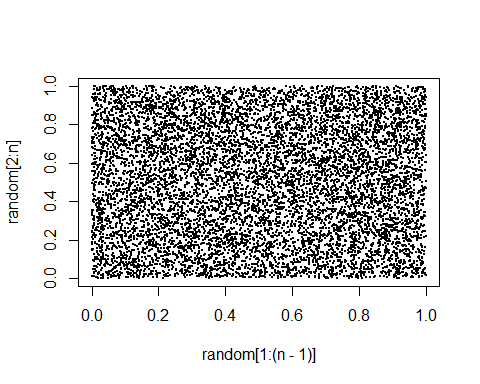
Se , , e , quais os 10 primeiros valores?

Resp.: Valores: 22, 117, 192, 167, 42, 17, 92, 67, 142, 117, na uniforme 0.11, 0.585, 0.96, 0.835, 0.21, 0.085, 0.46, 0.335, 0.71, 0.585

n <- 10000  
  
system.time(random <- congruencial(10, 21983, 239, 49187233, n)/49187233)

## user system elapsed   
## 0.03 0.00 0.03

# system.time(random <- runif(n))  
  
plot(random[1:(n-1)], random[2:n], pch = 19, cex = .1)



#### Aplicações

* Simulações
* Método de monte carlo
* Reamostragem

**Obs**: a é dita "raíz primitiva" mod m se:

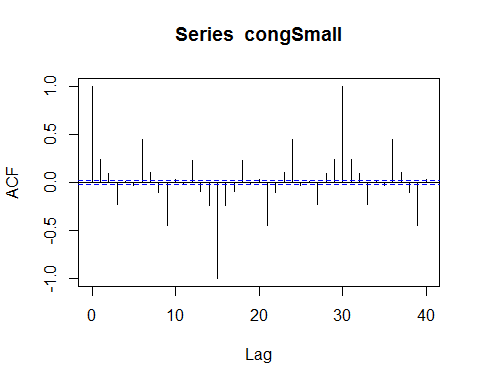
checkPrimRoot <- function(a, m){  
 for(i in 1:(m-2)){  
 if((a^i %% m) == 1){  
 return(paste(a, " NÃO é raiz primitiva de mod", m))  
 }  
 }  
 return(paste(a, "é raiz primitiva de mod", m))  
}  
  
  
checkPrimRoot(3, 31)

## [1] "3 é raiz primitiva de mod 31"

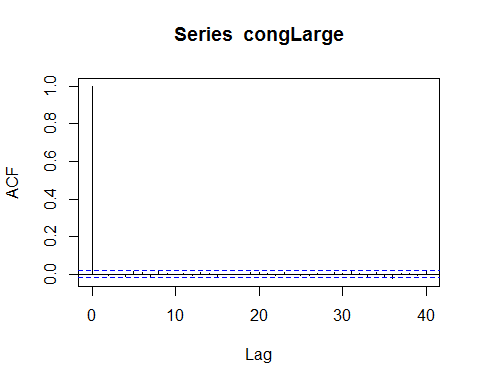
which(congruencial(29138739218, 3, 0, 31, 100) == 26)

## [1] 1 31 61 91

congSmall <- congruencial(29138739218, 3, 0, 31, 10000)/(31)  
congLarge <- congruencial(32139, 7^5, 0, 2^31-1, 10000)/(2^31-1)  
  
#Autocorrelações  
acf(congSmall)



acf(congLarge)



#Média = .5  
mean(congSmall)

## [1] 0.5001065

mean(congLarge)

## [1] 0.497439

#Variância = 1/12 = 0.08333  
var(congSmall)

## [1] 0.07794618

var(congLarge)

## [1] 0.08371179

#Até 10.000.000 vai ...  
length(table(congruencial(32139, 7^5, 0, 2^31-1, 1000)))

## [1] 1000

* Método \* Linear Feedback Shift Register\* (Tausworthe, 1965)

Trata-se de um método binário, também possui problemas de periodicidade, o período máximo é , mas nada garante que seja.

### Distribuições univariadas

* **Método da inversa da f.d.a. ou da transformação inversa**

Método mais fácil e mais rápido de ser implementado.

Seja uma v.a. de forma que , em que é uma f.d.a. inversível e v.a. com distribuição . Temos então que tem f.d.a. e tem f.d.a. . Assim,

É a função de distribuição da v.a. gerada.

**VER!!** Prova do método da inversa (Casella?)

**Exemplo**: Distribuição exponencial

Gerar uma realização onde

**Algoritmo**

1 - Gerar

2 - Calcular

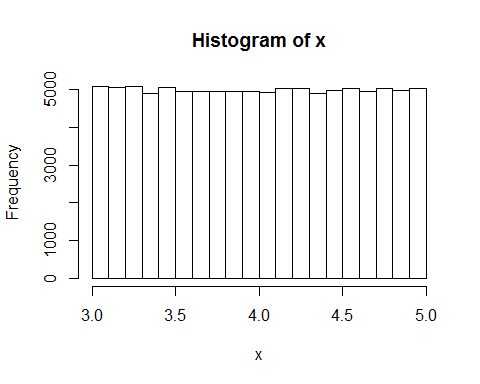
3 - Retornar

Com base nesse resultado eu posso gerar de várias distribuições que tem relação com exponencial, bastando obter apenas as outras inversas.

**Exercícios**

1 - Utilizar método da inversa da função de distribuição acumulada para gerar

u <- runif(100000)  
  
a <- 3; b <- 5 #Alguns exemplos só pra testar  
x <- (b-a)\*u + a #forma da inversa  
  
hist(x)



2 - Utilizar método da inversa da função de distribuição acumulada para gerar

3 - Utilizar método da inversa da função de distribuição acumulada para gerar

Outras distribuições que podem ser geradas pelo método da inversa:

Beta(a, 1); Beta(1, b), Pareto(a, b), Weibull(a, b), etc...

## Aula 4 - 30.08.2016

### Métodos para geração de variáveis aleatórias com distribuições específicas

#### Distribuição normal:

Caso queira outra média e variância, lembrar que, se , então

##### Média uniforme (método aproximado)

Pelo TCL, sabemos que se são v.a.'s iid , então:

Então tem distribuição aproximadamente normal (). Este processo nada mais é do que padronizar a soma das variáveis uniformes.

Tomando , temos:

Recomenda-se a não utilização desse método por ser dependente de uma aproximação e não um método exato.

**Exercício**: Utilizando o gerador de uniformes do R, verifique a aproximação para

mu10 <- replicate(10, sum(runif(12))-6)  
  
mu100 <- replicate(100, sum(runif(12))-6)  
  
mu1k <- replicate(1000, sum(runif(12))-6)  
  
mu10k <- replicate(10000, sum(runif(12))-6)  
  
  
#Testes  
# install.packages("ADGofTest", dependencies = TRUE)  
require(GLDEX); require(ADGofTest)

## Loading required package: GLDEX

## Loading required package: cluster

## Loading required package: ADGofTest

##KS  
ks.gof(mu10, "pnorm")

##   
## One-sample Kolmogorov-Smirnov test  
##   
## data: x  
## D = 0.29715, p-value = 0.2805  
## alternative hypothesis: two-sided

ks.gof(mu100, "pnorm")

##   
## One-sample Kolmogorov-Smirnov test  
##   
## data: x  
## D = 0.098196, p-value = 0.2899  
## alternative hypothesis: two-sided

ks.gof(mu1k, "pnorm")

##   
## One-sample Kolmogorov-Smirnov test  
##   
## data: x  
## D = 0.021729, p-value = 0.7325  
## alternative hypothesis: two-sided

ks.gof(mu10k, "pnorm")

##   
## One-sample Kolmogorov-Smirnov test  
##   
## data: x  
## D = 0.00824, p-value = 0.5056  
## alternative hypothesis: two-sided

#AD  
ad.test(mu10, distr.fun = pnorm)

##   
## Anderson-Darling GoF Test  
##   
## data: mu10 and pnorm  
## AD = 1.1048, p-value = 0.3051  
## alternative hypothesis: NA

ad.test(mu100, distr.fun = pnorm)

##   
## Anderson-Darling GoF Test  
##   
## data: mu100 and pnorm  
## AD = 1.2018, p-value = 0.2668  
## alternative hypothesis: NA

ad.test(mu1k, distr.fun = pnorm)

##   
## Anderson-Darling GoF Test  
##   
## data: mu1k and pnorm  
## AD = 0.46794, p-value = 0.7796  
## alternative hypothesis: NA

ad.test(mu10k, distr.fun = pnorm)

##   
## Anderson-Darling GoF Test  
##   
## data: mu10k and pnorm  
## AD = 1.2034, p-value = 0.2663  
## alternative hypothesis: NA

#Shapiro-wilk  
shapiro.test(mu10)

##   
## Shapiro-Wilk normality test  
##   
## data: mu10  
## W = 0.93212, p-value = 0.469

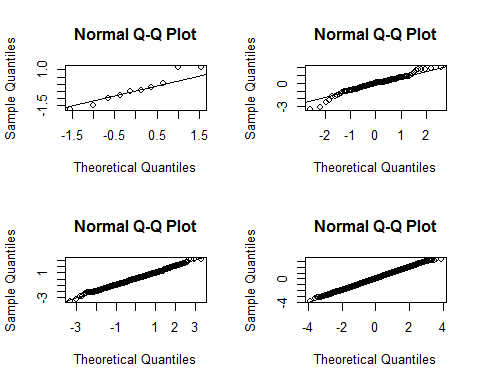
shapiro.test(mu100)

##   
## Shapiro-Wilk normality test  
##   
## data: mu100  
## W = 0.96745, p-value = 0.01416

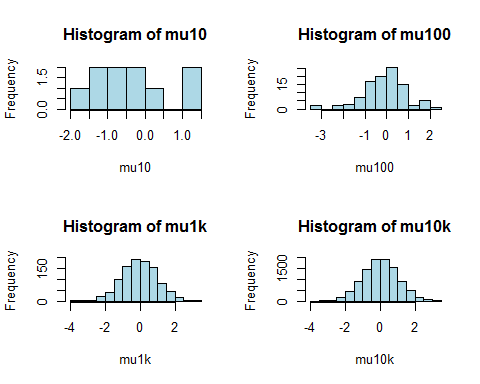
shapiro.test(mu1k)

##   
## Shapiro-Wilk normality test  
##   
## data: mu1k  
## W = 0.9986, p-value = 0.6219

# shapiro.test(mu10k) #Não roda  
  
#QQPlot  
par(mfrow = c(2, 2))  
qqnorm(mu10); qqline(mu10)  
qqnorm(mu100); qqline(mu100)  
qqnorm(mu1k); qqline(mu1k)  
qqnorm(mu10k); qqline(mu10k)



#Histograma  
hist(mu10, col = "light blue")  
hist(mu100, col = "light blue")  
hist(mu1k, col = "light blue")  
hist(mu10k, col = "light blue")



par(mfrow = c(1, 1))  
  
#Testando várias replicações  
mu10.1k <- replicate(1000, replicate(10, sum(runif(12))-6))  
pValues <- sapply(mu10.1k, function(x){  
 ks.gof(x, "pnorm")$p.value  
})  
  
mean(pValues<.05)

## [1] 0.0478

##### Método Box-Muller (1958)

Baseado na geração inicial de duas que são submetidas à seguinte transformação:

Assim, e independentes

Vantagem: é **exato**

Desvantagem: mais **lento**

A base desse algoritmo são coordenadas polares, pensando nas duas realizações da normal como pontos de um plano cartesiano, representando-os em coordenadas polares temos:

Como independentes

Aplicando a transformação usando jacobiano, temos que

Onde o primeiro termo corresponde ao e o segundo ao .

Teremos pelo método da inversa que e $\theta=2\piU\_2$. Assim é possível obter que e

##### Método Polar (Marsaglia, 1962)

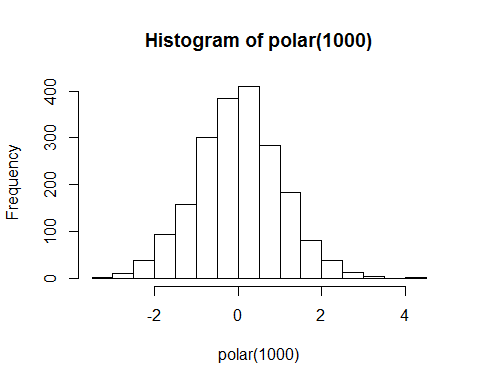
Parte de e independentes.

**Algoritmo:**

1. Até que , repetir:
   * Gerar e
   * Calcular: e
2. Calcular:

Trata-se de um algoritmo de aceitação/rejeição, com alta taxa de aceitação . Também fundamentado em coordenadas polares

polar <- function(n){  
 out <- numeric()  
 for(i in 1:n){  
 v1=1  
 v2=2  
 while((v1^2+v2^2)>1){  
 v1=2\*runif(1)-1  
 v2=2\*runif(1)-1  
 }  
 a <- v1^2+v2^2  
 z1 <- v1\*sqrt((-2\*log(a))/(a))  
 z2 <- v2\*sqrt((-2\*log(a))/(a))  
   
 out <- c(out, z1, z2)  
 }  
 return(out)  
}  
  
  
hist(polar(1000))



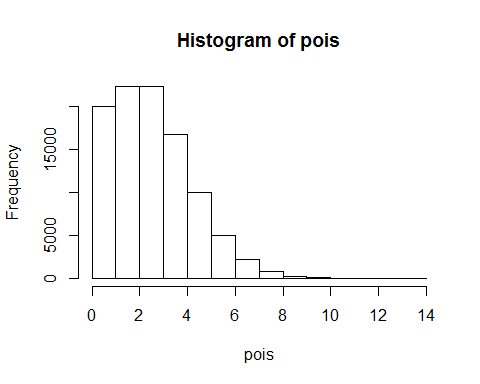
#### Distribuição poisson:

[Ver caderno para detalhes]

myPois <- function(l, n){  
 out <- numeric(n)  
   
 for(i in 1:n){  
 y <- 0  
 k <- exp(-l)  
 p <- runif(1)  
 while(p>k){  
 u <- runif(1)  
 p <- u\*p  
 y <- y+1  
 }  
 out[i] <- y  
 }  
 return(out)  
}  
  
  
pois <- myPois(3, 100000)  
  
ks.gof(pois, "ppois", lambda = 3)

##   
## One-sample Kolmogorov-Smirnov test  
##   
## data: x  
## D = 0.11288, p-value < 2.2e-16  
## alternative hypothesis: two-sided

hist(pois)



## Aula 5 - 2016.09.06

### Métodos de Rejeição

Procurar: acceptance-rejection ou accept-reject ou reject method ou sampling

#### Método de rejeição básico

Considere uma v.a. com f.d.p. proporcional a , ou seja, onde é a f.d.p. de .