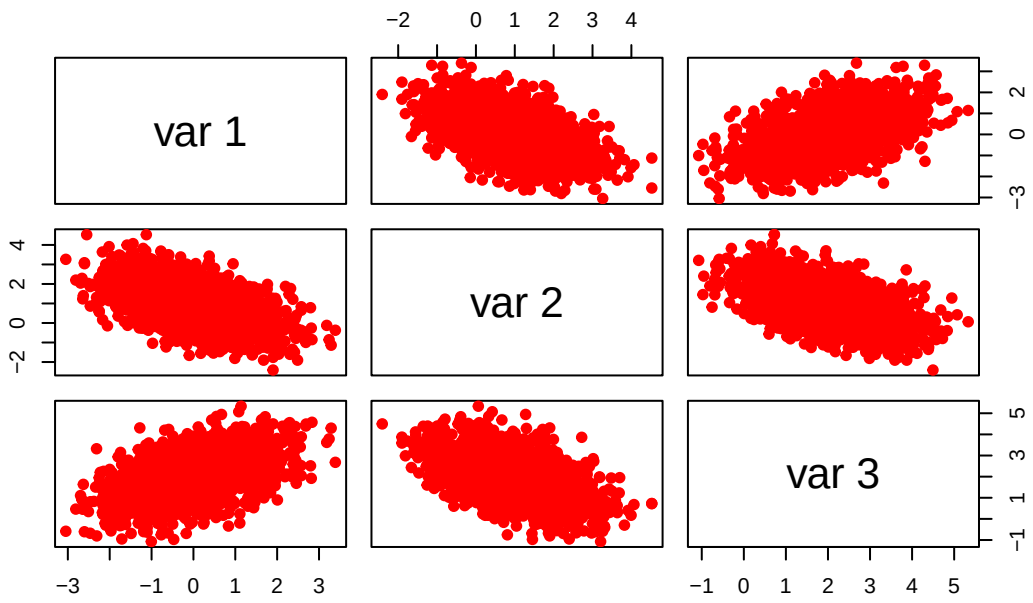
Lista 3

Lucas Menezes e Silva

2023-07-14

Questao 1 da lista 3

```
library(MASS)

# Set the mean vector and covariance matrix
mu <- c(0, 1, 2)
Sigma <- matrix(c(1.0, -0.5, 0.5, -0.5, 1.0, -0.5, 0.5, -0.5, 1.0), nrow = 3)

# Generate random observations
n <- 2000
set.seed(123)

rmvn.cholesky <-
  function(n, mu, Sigma) {
    p <- length(mu)
    Q <- chol(Sigma)
    Z <- matrix(rnorm(n*p), nrow=n, ncol=p)
    X <- Z %*% Q + matrix(mu, n, p, byrow=TRUE)
    X
  }

a3 <- rmvn.cholesky(n, mu, Sigma)
# print(colMeans(a3))
# Create the pairs plot
pairs(a3, main = "Scatter Plots das Variáveis",
      pch = 19, col = "red")
```




```
print(cor(a3))# se aproximam
```

```
##            [,1]       [,2]       [,3]
## [1,]  1.0000000 -0.5141424  0.4997655
## [2,] -0.5141424  1.0000000 -0.4930523
## [3,]  0.4997655 -0.4930523  1.0000000
```

A partir da decomposição de Cholesky construimos via simulação para um n = 2000 e a matriz Sigma convergiu bastante para os valores esperados, dessa forma concluimos que o processo de composição de normais funciona e converge assintoticamente para a matriz desejada.

Questao 2 da lista 3

```
# Definir a função que será integrada
funcao <- function(x) {
  return(exp(1)^x)
}

# Set the number of iterations
N <- 100000


# Metodo de monte carlo
montecarlo<- function(N) {
  u <- runif(N)  # Gera N numeros aleatorios da distribuição uniforme
  theta <- funcao(u)  # Estima  usando a função
  estimate <- mean(theta)  # Calcula a media de
  return(estimate)
}
```

```r
# Antithetic variate method
antitetica <- function(N) {
  u <- runif(N/2)  # Generate N/2 random numbers from a uniform distribution
  u_antithetic <- 1 - u  # Generate the antithetic variates
  u_combined <- c(u, u_antithetic)  # Combine the variates
  theta <- funcao(u_combined)  # Estimate  using the function
  estimate <- mean(theta)  # Calculate the mean of
  return(estimate)
}

set.seed(123)
# Estimar  usando o simples Monte Carlo
estimativa_mc <- montecarlo(N)
set.seed(123)
# Estimate  using the antithetic variate method
estimativa_antitetica <- antitetica(N)

# Print the results
print(paste(" estimando usando o simples método de Monte Carlo:", estimativa_mc))#mais longe
```

```
## [1] " estimando usando o simples método de Monte Carlo: 1.71698509222497"
```

```r
print(paste(" estimando usando método 'antithetic variate' :",estimativa_antitetica))#mais perto
```

```
## [1] " estimando usando método 'antithetic variate' : 1.71808612286583"
```

```r
print(paste("Valore real de  :", integrate(funcao,0,1)[1]))
```

```
## [1] "Valore real de  : 1.71828182845905"
```

Aqui acima foi criada a função de monte carlo para estimar a função assim como o estimado antitético. Após criadas foi estimado com um número N de repetições, onde N = 100000, e foi calculado os estimadores com uma semente fixa padrão "123", dessa forma os estimadores performaram bem por conta do número de repetições mas aparentemente o estimador antitético possuiu uma estimativa consideravelmente mais precisa comparado com o estimador monte carlo. Exite o real valor da integral para referencial comparativo.

Questao 3 da lista 3

```r
library(e1071)# Pacote contendo a implementação do SVM
```

```
## Warning: package 'e1071' was built under R version 4.2.3
```

```r
library(mlbench)
```

```
## Warning: package 'mlbench' was built under R version 4.2.3
```

```r
data(Glass, package="mlbench")
## split data into a train and test set
index <- 1:nrow(Glass)
N <- trunc(length(index)/3)
set.seed(123)
testindex <- sample(index, N)
testset <- Glass[testindex,]
trainset <- Glass[-testindex,]
## svm
svm.model <- svm(Type ~ ., data = trainset, cost = 100, gamma = 0.1)
svm.model
```

```
##
## Call:
## svm(formula = Type ~ ., data = trainset, cost = 100, gamma = 0.1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  100
##
## Number of Support Vectors:  105
```

```
svm.pred <- predict(svm.model, testset[,-10])
svm.pred
```

```
## 159 207 179  14 195 170  50 118  43 211 213 153  90  91 197 201 185  92 137  99
##   3   7   6   1   7   5   1   2   1   7   7   3   2   3   7   7   2   2   1   2
##  72  26   7 209 196 164  78  81 206 103 117  76 143  32 109 192 190 169  74  23
##   2   1   1   7   7   2   2   2   7   1   2   2   2   1   6   7   2   5   2   1
## 155  53 135 173 174 166  34  69 194 183  63 141  97 199 203  38  21  41 202  60
##   3   1   1   5   5   5   1   1   7   5   1   2   3   7   7   1   2   1   2   1
##  16 116  94   6  86 150  39 204 208 168   4
##   1   2   3   2   3   2   1   7   2   5   2
## Levels: 1 2 3 5 6 7
```

```
## compute svm confusion matrix
matriz_confusao = table(pred = svm.pred, true = testset[,10])
matriz_confusao = as.matrix(matriz_confusao)



# install.packages("caret")
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.2.3
```

```
## Carregando pacotes exigidos: ggplot2
```

```
## Carregando pacotes exigidos: lattice
```

```
## Warning: package 'lattice' was built under R version 4.2.3
```

```
# Assuming you have a confusion matrix named "confusion_matrix"

# Set the range of gamma values to evaluate
gammas <- c(1, 10 ,100)

# Perform k-fold cross-validation
set.seed(123)  # For reproducibility
folds <- createFolds(seq(1:6), k = 10)

erros = numeric()
for (gamma in gammas) {
  erro1 <- numeric()  # Variável para armazenar o erro para cada fold
  erro2 <- numeric()  # Variável para armazenar o erro para cada fold
  erro3 <- numeric()  # Variável para armazenar o erro para cada fold
```

```
  for (i in 1:length(folds)) {
    # Divide os dados em conjunto de treinamento e teste para o fold atual
    train_data <- matriz_confusao[-folds[[i]], ]
    test_data <- t(as.matrix(matriz_confusao[folds[[i]], ]))

    # Extrai as variáveis de entrada (X) e de saída (Y) dos dados de treinamento e teste
    train_X <- train_data[, -ncol(train_data)]
    train_Y <- as.vector(train_data[, ncol(train_data)])
    test_X <- as.vector(test_data[, -ncol(test_data)])
    test_Y <- as.vector(test_data[, ncol(test_data)])

    # Treina o modelo SVM com o valor de gamma atual
    modelo1 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[1])
    modelo2 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[2])
    modelo3 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[3])

    # Realiza a predição nos dados de teste
    predicao1 <- predict(modelo1)
    predicao2 <- predict(modelo2)
    predicao3 <- predict(modelo3)

    # Calcula a taxa de erro no fold atual
    erro1[i] <- sum(predicao1 != test_Y) / length(test_Y)
    erro2[i] <- sum(predicao2 != test_Y) / length(test_Y)
    erro3[i] <- sum(predicao3 != test_Y) / length(test_Y)

    }
  # Calcula o erro médio para o valor de gamma atual
  avg_erro <- c(mean(erro1),mean(erro2),mean(erro3))
  erros <- c(erros, avg_erro)
}

# Encontra o valor de gamma com o menor erro
melhor1_gamma <- gammas[which.min(erros)]
melhor1_error <- min(erros)

cat("Melhor valor de gamma:", melhor1_gamma, "\n")
```

```
## Melhor valor de gamma: 1
```

```
cat("Erro de teste correspondente:", melhor1_error, "\n")
```

```
## Erro de teste correspondente: 140.8333
```

```
# Se mudarmos os erros para um valor menor, teremos um erro melhor
gammas = NULL



# novo gamma:
# Set the range of gamma values to evaluate
gammas <- c(0.01, 0.1 ,1)
set.seed(123)  # For reproducibility
folds <- createFolds(seq(1:6), k = 10)
```

```r
erros = numeric()
for (gamma in gammas) {
  erro1 <- numeric()  # Variável para armazenar o erro para cada fold
  erro2 <- numeric()  # Variável para armazenar o erro para cada fold
  erro3 <- numeric()  # Variável para armazenar o erro para cada fold

  for (i in 1:length(folds)) {
    # Divide os dados em conjunto de treinamento e teste para o fold atual
    train_data <- matriz_confusao[-folds[[i]], ]
    test_data <- t(as.matrix(matriz_confusao[folds[[i]], ]))

    # Extrai as variáveis de entrada (X) e de saída (Y) dos dados de treinamento e teste
    train_X <- train_data[, -ncol(train_data)]
    train_Y <- as.vector(train_data[, ncol(train_data)])
    test_X <- as.vector(test_data[, -ncol(test_data)])
    test_Y <- as.vector(test_data[, ncol(test_data)])

    # Treina o modelo SVM com o valor de gamma atual
    modelo1 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[1])
    modelo2 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[2])
    modelo3 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[3])

    # Realiza a predição nos dados de teste
    predicao1 <- predict(modelo1)
    predicao2 <- predict(modelo2)
    predicao3 <- predict(modelo3)

    # Calcula a taxa de erro no fold atual
    erro1[i] <- sum(predicao1 != test_Y) / length(test_Y)
    erro2[i] <- sum(predicao2 != test_Y) / length(test_Y)
    erro3[i] <- sum(predicao3 != test_Y) / length(test_Y)

    }
  # Calcula o erro médio para o valor de gamma atual
  avg_erro <- c(mean(erro1),mean(erro2),mean(erro3))
  erros <- c(erros, avg_erro)
}

# Encontra o valor de gamma com o menor erro
melhor2_gamma <- gammas[which.min(erros)]
melhor2_erro <- min(erros)

cat("Melhor valor de gamma:", melhor2_gamma, "\n")
```

```
## Melhor valor de gamma: 0.1
```

```r
cat("Erro de teste correspondente:", melhor2_erro, "\n")
```

```
## Erro de teste correspondente: 140.8333
```

```r
gammas = NULL



gammas <- c(0.0001,0.001,0.01)
```

```r
set.seed(123)  # For reproducibility
folds <- createFolds(seq(1:6), k = 10)

erros = numeric()
for (gamma in gammas) {
  erro1 <- numeric()  # Variável para armazenar o erro para cada fold
  erro2 <- numeric()  # Variável para armazenar o erro para cada fold
  erro3 <- numeric()  # Variável para armazenar o erro para cada fold

  for (i in 1:length(folds)) {
    # Divide os dados em conjunto de treinamento e teste para o fold atual
    train_data <- matriz_confusao[-folds[[i]], ]
    test_data <- t(as.matrix(matriz_confusao[folds[[i]], ]))

    # Extrai as variáveis de entrada (X) e de saída (Y) dos dados de treinamento e teste
    train_X <- train_data[, -ncol(train_data)]
    train_Y <- as.vector(train_data[, ncol(train_data)])
    test_X <- as.vector(test_data[, -ncol(test_data)])
    test_Y <- as.vector(test_data[, ncol(test_data)])

    # Treina o modelo SVM com o valor de gamma atual
    modelo1 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[1])
    modelo2 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[2])
    modelo3 <- svm(Type ~ ., data = trainset, cost = 100, gamma = gammas[3])

    # Realiza a predição nos dados de teste
    predicao1 <- predict(modelo1)
    predicao2 <- predict(modelo2)
    predicao3 <- predict(modelo3)

    # Calcula a taxa de erro no fold atual
    erro1[i] <- sum(predicao1 != test_Y) / length(test_Y)
    erro2[i] <- sum(predicao2 != test_Y) / length(test_Y)
    erro3[i] <- sum(predicao3 != test_Y) / length(test_Y)

    }
  # Calcula o erro médio para o valor de gamma atual
  avg_erro <- c(mean(erro1),mean(erro2),mean(erro3))
  erros <- c(erros, avg_erro)
}

# Encontra o valor de gamma com o menor erro
melhor3_gamma <- gammas[which.min(erros)]
melhor3_erro <- min(erros)

cat("Melhor valor de gamma:", melhor3_gamma, "\n")
```

```
## Melhor valor de gamma: 0.01
```

```r
cat("Erro de teste correspondente:", melhor3_erro, "\n")
```

```
## Erro de teste correspondente: 142.8333
```

Dado dos valores dos testes realizados onde os betas são os recomendados (1,0.1,0.01) temos os valores abaixo dada as comparações que foram feitas onde os menores erros tendem para valores de beta acima de 0.1, sendo

o valor do teste correspondente a 140.8333 .

Questao 4 da lista 3 - 9.3

```r
# Criar o vetor de valores de x
x <- seq(-10, 10, length.out = 1000)

# Definir a função densidade da distribuição Cauchy

densidade_cauchy <- function(x, locacao, escala) {
  return(1 / (3.141593 * escala * (1 + ((x - locacao) / escala)^2)))
}

# Definir os parâmetros da distribuição Cauchy
locacao <- 0  # Parâmetro de localização
escala <- 1  # Parâmetro de escala = teta

# Número de interações
N <- 10000

# Número de amostras descartadas
descarte <- 1000

# Iniciar a corrente
chain <- numeric(N + descarte)
chain[1] <- 0  # Starting value for the chain

# distribuição proposta incialmente (e.g., normal distribution)
dp_prop <- 1  # Desvio padrao da distribuição proposta incialmente

# Definir a desidade esperada (cauchy padrao)
target_density <- function(x) {
  return(1 / (pi * (1 + x^2)))
}

# Algoritmo de Metropolis-Hastings
for (i in 2:(N + descarte)) {
  # Generate a candidate sample from the proposal distribution
  candidato <- rnorm(1, mean = chain[i - 1], sd = dp_prop)

  # Calculate the acceptance ratio
  acceptance_ratio <- target_density(candidato) / target_density(chain[i - 1])

  # Aceitar ou rejeitar a amostra do candidato
  if (runif(1) < acceptance_ratio) {
    chain[i] <- candidato  # Aceite o candidato
  } else {
    chain[i] <- chain[i - 1]  # rejeitar o candidato e manter o valor anterior
  }
}

# Desccarte das amostras
chain <- chain[(descarte + 1):(N + descarte)]
```

```r
# Calcular os decis gerados
decis_gerado <- quantile(chain, probs = seq(0.1, 0.9, by = 0.1))

# Calcular os decis da distribuição
decis_cauchy <- qcauchy(seq(0.1, 0.9, by = 0.1))

# Valor dos decis
print("Decis das observações geradas:")
```

```
## [1] "Decis das observações geradas:"
```

```r
print(decis_gerado)
```

```
##        10%        20%        30%        40%        50%        60%        70%
## -4.1211493 -1.7405473 -0.9370590 -0.4685379 -0.1157711  0.2174371  0.6304238
##        80%        90%
##  1.2355084  2.7643459
```

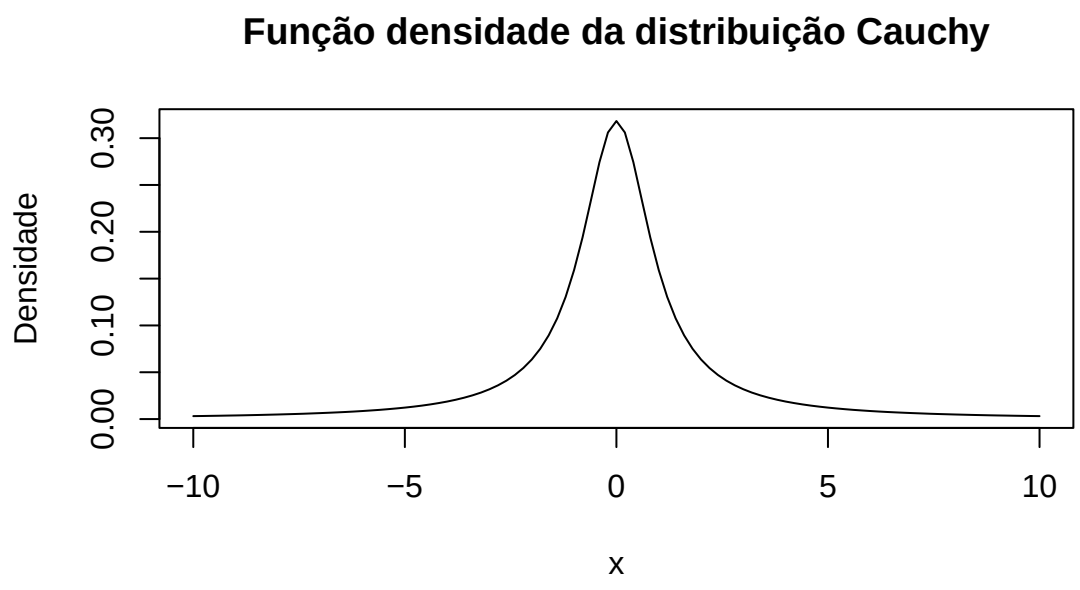
```r
print("Decis da Cauchy padrão:")
```

```
## [1] "Decis da Cauchy padrão:"
```

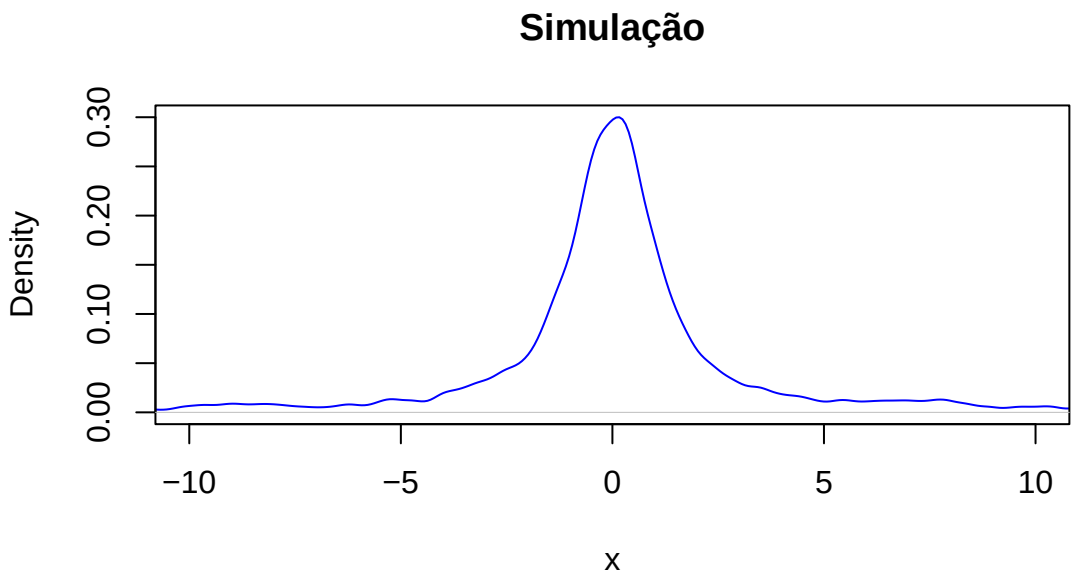
```r
print(decis_cauchy)
```

```
## [1] -3.0776835 -1.3763819 -0.7265425 -0.3249197  0.0000000  0.3249197  0.7265425
## [8]  1.3763819  3.0776835
```

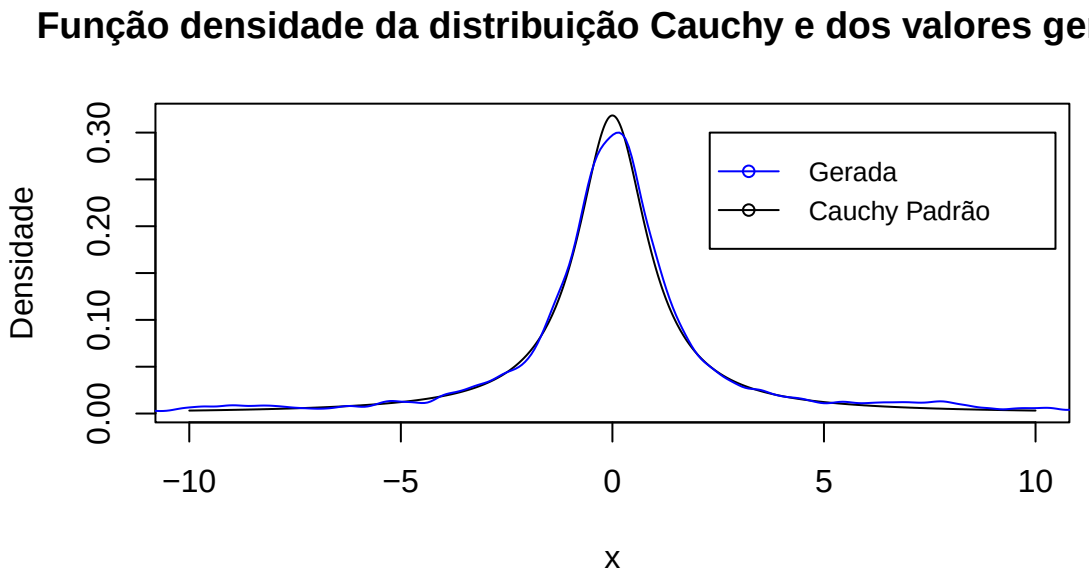
```r
curve(densidade_cauchy(x, locacao, escala), from = -10, to = 10,
      xlab = "x", ylab = "Densidade", main = "Função densidade da distribuição Cauchy")
```




```r
# Create the density plot
plot(density(chain), main = "Density Plot", xlab = "x", ylab = "Density", col = "blue",
     xlim = c(-10,10))
```



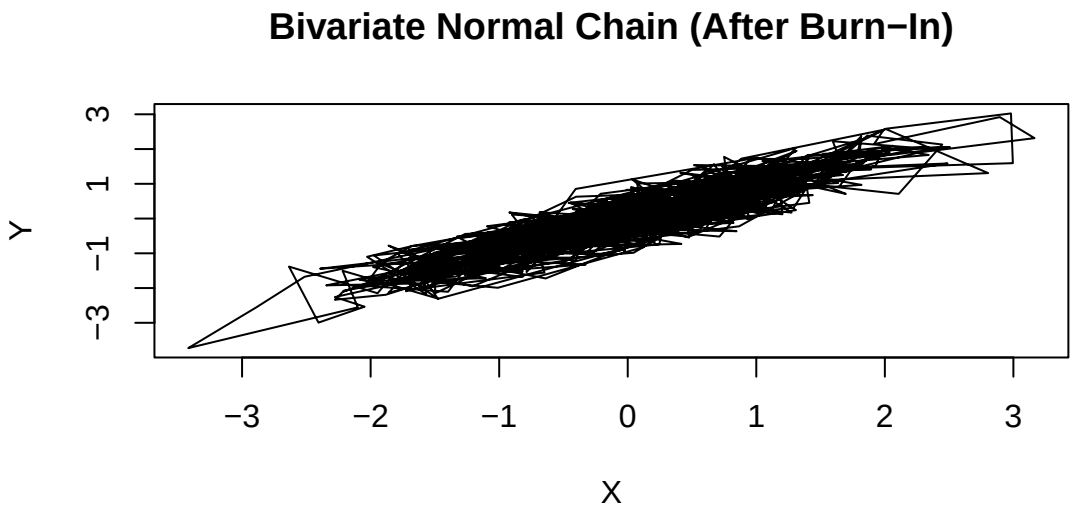

```r
# Calcular os valores da função densidade para cada valor de x
densidades <- densidade_cauchy(x, locacao, escala)

# Criar o gráfico
plot(x, densidades, type = "l",
     xlab = "x", ylab = "Densidade",
     main = "Função densidade da distribuição Cauchy e dos valores gerados")
lines(density(chain), col = "blue")
legend( legend = c("Gerada", "Cauchy Padrão"),
       col = c("blue", "black"), lty = 1, pch =1, cex = 0.8, x = 2.3, y = 0.3)
```



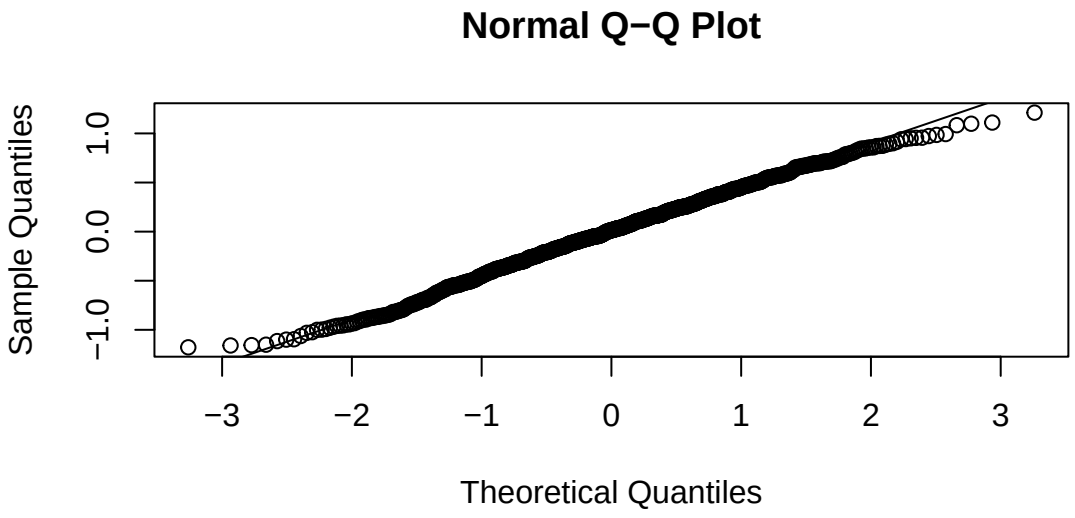
Acima vemos os valores reais da função densidade da cauchy e os valores simulados dentro da ideia do descarte das amostras. Existe o gráfico comparativo para mostrar o quão similares são. A simulada possui caudas mais pesadas devido ao número amostral, dito isso ela convergiria assintoticamente.

Questao 4 da lista 3 - 9.7

```r
# Set the number of iterations
N <- 1000

# Set the correlation coefficient
rho <- 0.9

# Initialize the chains
X <- numeric(N)
Y <- numeric(N)


# Set the initial values for X and Y
X[1] <- 0
Y[1] <- 0

# Run the Gibbs sampler
for (t in 2:N) {
  # Sample X[t] given Y[t-1]
  X[t] <- rnorm(1, mean = rho * Y[t - 1], sd = sqrt(1 - rho^2))

  # Sample Y[t] given X[t]
  Y[t] <- rnorm(1, mean = rho * X[t], sd = sqrt(1 - rho^2))
}

# Set the burn-in length
burn_in <- 100

# Discard the burn-in samples
X_discarded <- X[(burn_in + 1):N]
Y_discarded <- Y[(burn_in + 1):N]

# Plot the bivariate normal chain after burn-in
plot(X_discarded, Y_discarded, type = "l", xlab = "X", ylab = "Y", main = "Bivariate Normal Chain (Afte
```



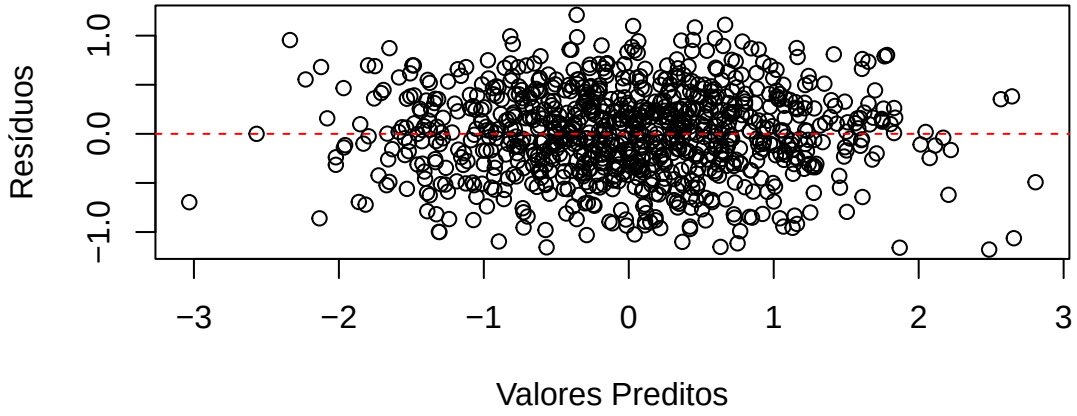

```r
# Aplicando a Regressao

Banco = as.data.frame(Y_discarded)
Banco$X_discarded = X_discarded
names(Banco) <- c("Y","X")

# Passo 2: Criação do modelo linear
modelo <- lm(Y ~ X, data = Banco)

# Passo 3: Conferir a normalidade dos dados pelos quantis
# Q-Q plot
qqnorm(modelo$residuals)
qqline(modelo$residuals)
```



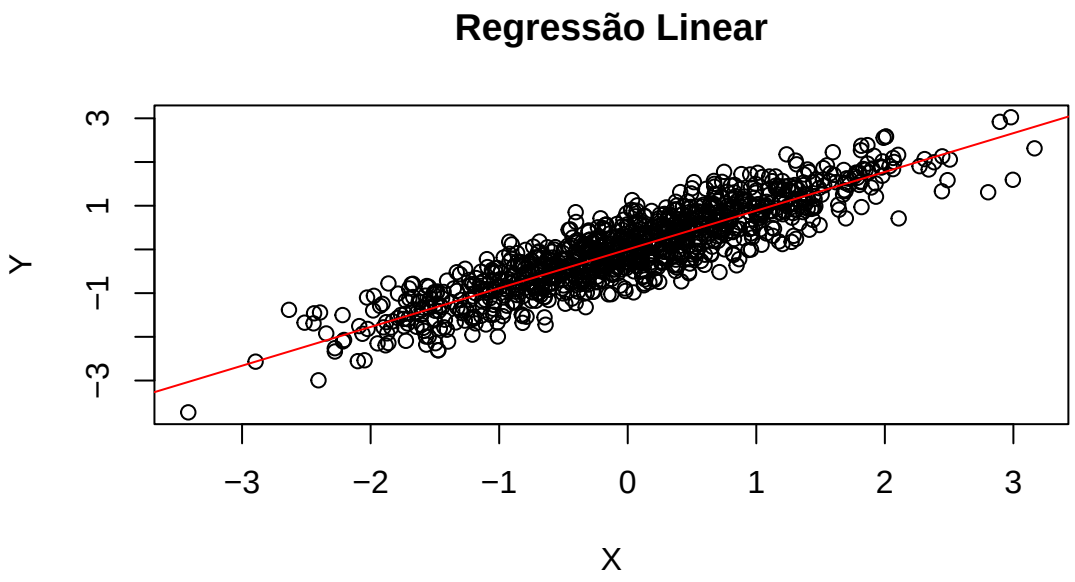
```r
# Shapiro-Wilk teste
shapiro.test(modelo$residuals)# normal
```

```
##
##  Shapiro-Wilk normality test
##
## data:  modelo$residuals
## W = 0.99777, p-value = 0.272
```

```r
# Passo 4: Confira os resídios para avaliar a variancia constante
# Plot dos residuos pelos valores preditos
plot(modelo$fitted.values, modelo$residuals,
     xlab = "Valores Preditos", ylab = "Resíduos")
abline(h = 0, col = "red", lty = 2)
```




```r
# aparentemente existe uma dispersão ok dos dados

# Gráfico de dispersão
plot(X, Y, main = "Regressão Linear", xlab = "X", ylab = "Y")

# Linha de regressão
abline(modelo, col = "red")
```




```
summary(modelo)
```

```
##
## Call:
## lm(formula = Y ~ X, data = Banco)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.28423 -0.30517 -0.00403  0.29795  1.23895
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.008337   0.014980   0.557    0.578
## X           0.885193   0.015419  57.411   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4456 on 898 degrees of freedom
## Multiple R-squared:  0.7859, Adjusted R-squared:  0.7856
## F-statistic:  3296 on 1 and 898 DF,  p-value: < 2.2e-16
```

A partir do princípio de Gibbs fornecido é possível ver os pontos sendo gerados e pelo número de simulações acabaram convergindo a uma normalidade, sendo possível a avaliação a partir dos resídios, que seguem distribuição normal e são independentes.