Random Number Generation through CDF and acceptance-rejection sampling

23-10-2024

Linear Congrential Random Number

Task 1: Summarize the concept of pseudo-random number generation with Linear Congruential Random Number and simulate one pseudo-random sample

The subsection below contains the following code:
- Linear Congruential Random Number Generation Algorithm
- Plot Function to generate histogram and scatter plot

```
## Plot function that creates histogram and scatter plot

plot_sample <- function(sample){
  hist(sample, breaks = 30, main = "Random Sample",
     xlab = "Random Numbers", ylab = "Frequency")
  plot(sample[-n], sample[-1], main = "Scatter_plot",
     xlab = expression(u[i]), ylab = expression(u[i+1]), pch = 16)
}
```

A Pseudorandom Number Generator is a function that takes a short random seed and generates sequences of numbers that appears random.

Linear Congruential Random Number Generation Algorithm is commonly used Pseudorandom Number Generator, because of its simplicity and efficiency. It uses the linear equation:

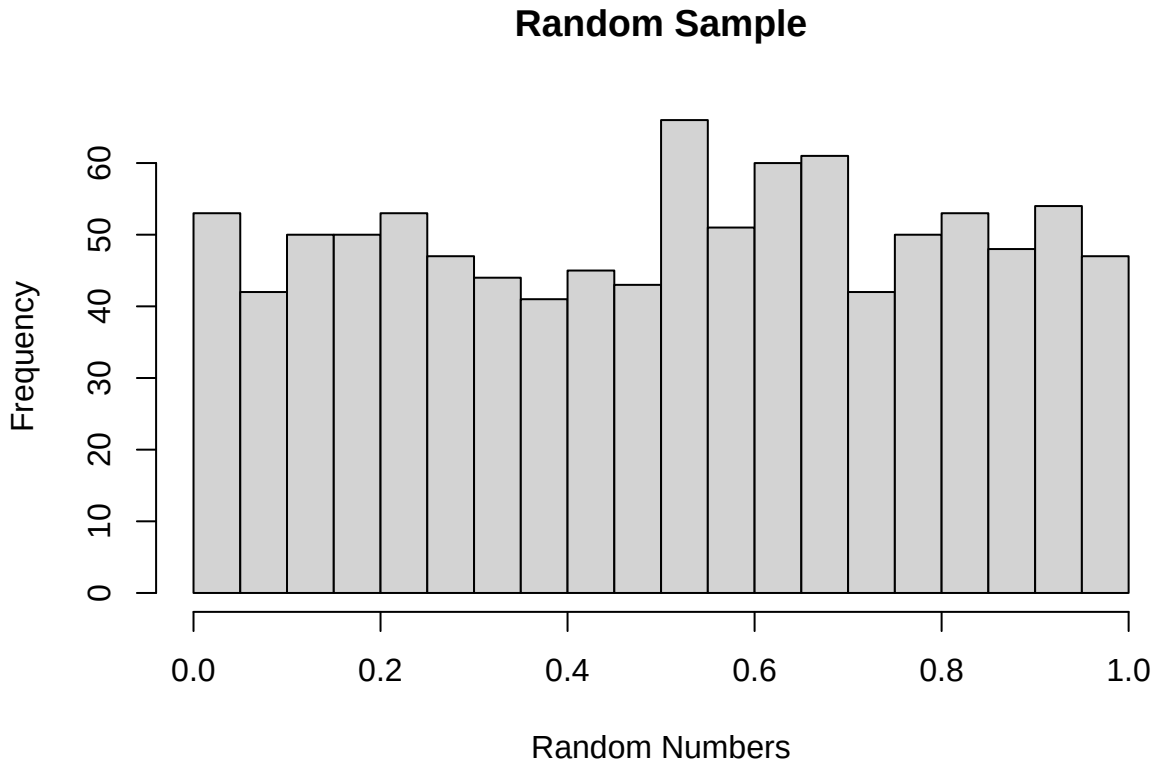
$$X_{n+1} = (a X_n + c) \bmod m$$

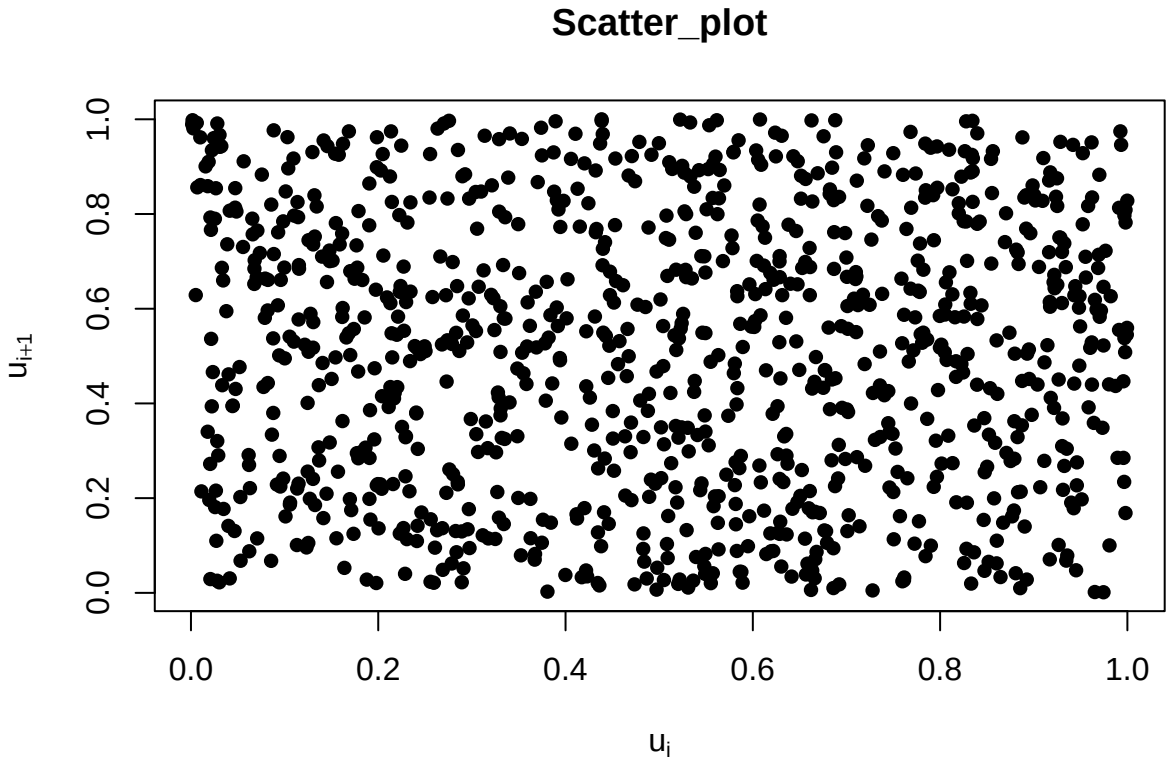
For the algorithm, we require:
- m : a large integer modulus
- a : sqrt(m), an integer multiplier
- c : an integer increment, there ($0 <= c <= m$)

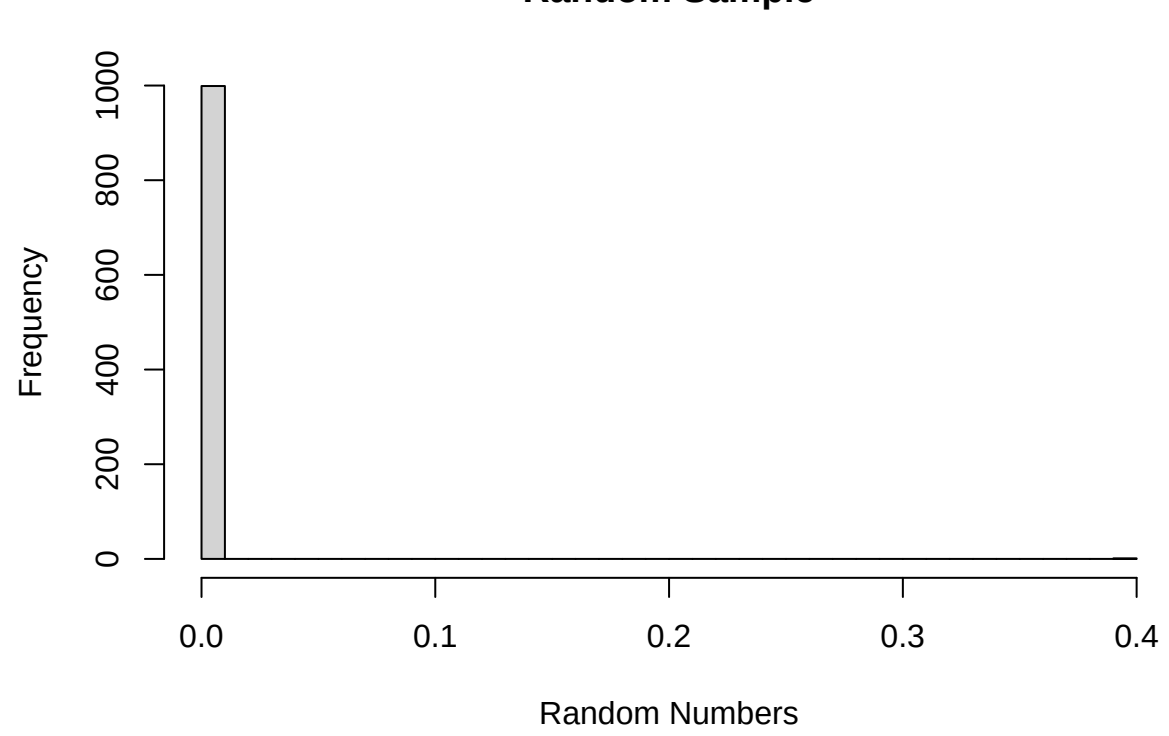
The sequence generated, u1, u2,. . . . are uniform pseudo random numbers.

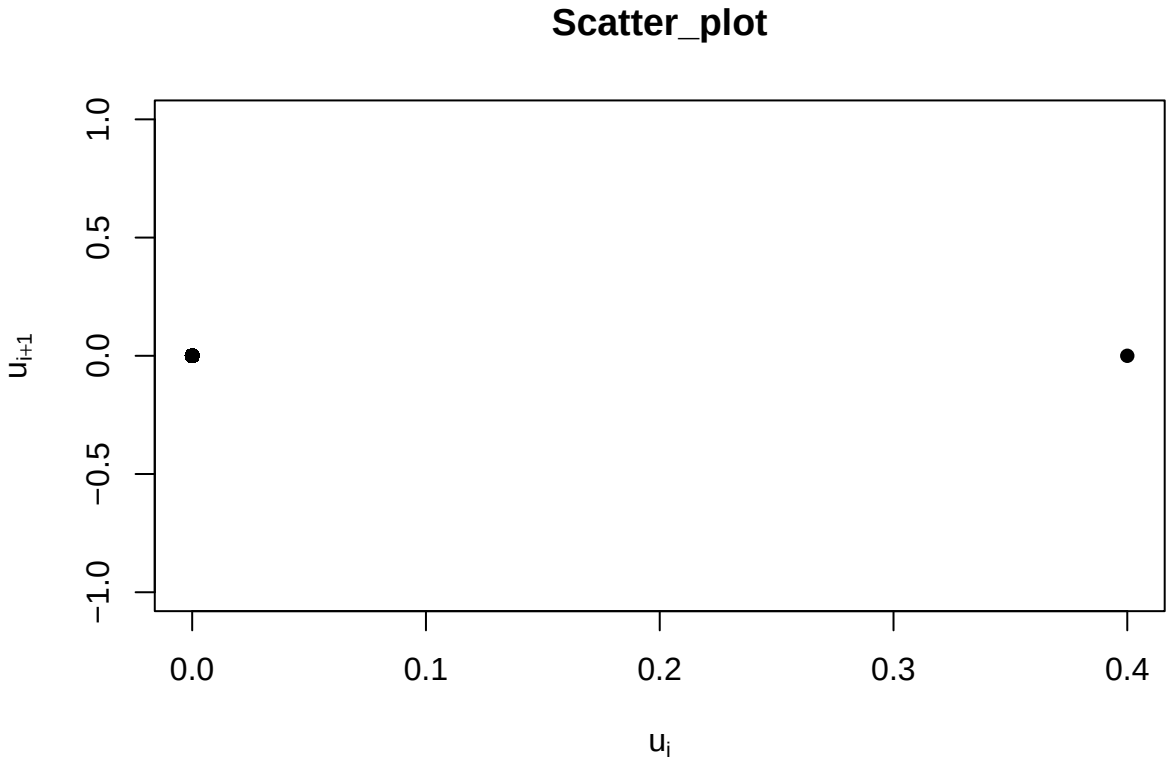
```
## Linear Congruential Random Number Generation Algorithm

LCG <- function(a, c, m, n, x0) {
  us <- numeric(n)

  for (i in 1:n){
    x0 <- (a * x0 + c) %% m
    us[i] <- x0 / m

  }
  return(us)
}

x <- 1234
```

```r
n <- 1000

## Parameters 1

m1 <- 2^31 - 1
a1 <- 46340
c1 <- 12345

sample_1 <- LCG(a1, c1, m1, n, x)

# Parameters 2

m2 <- 100
a2 <- 10
c2 <- 0

sample_2 <- LCG(a2, c2, m2, n, x)

plot_sample(sample_1)
plot_sample(sample_2)
```




Scatter_plot
u_{i+1}
0.0
0.2
0.4
0.6
0.8
1.0
u_i


Random Sample
Frequency
0
200
400
600
800
1000
0.0
0.1
0.2
0.3
0.4
Random Numbers



There are two samples provided above with different parameters.

Parameter 1: Good parameters
- m = 2^31 - 1 (large modulus)
- a = 46340 (approximate sqrt(m))

Observation: The histogram shows a uniform distribution and the scatter plot does not show any obvious patterns. This shows that the parameters were good enough to produce random numbers.

Parameter 2: Bad parameters
- m = 100 (small modulus)
- a = 100 (sqrt(m))

Observation: The historgram shows a non-uniform distribution and the scatter plot show an obvious pattern. This shows that the parameters were very poor.

Exponential Distribution

Task 2: Obtain a random sample from the exponential distribution.

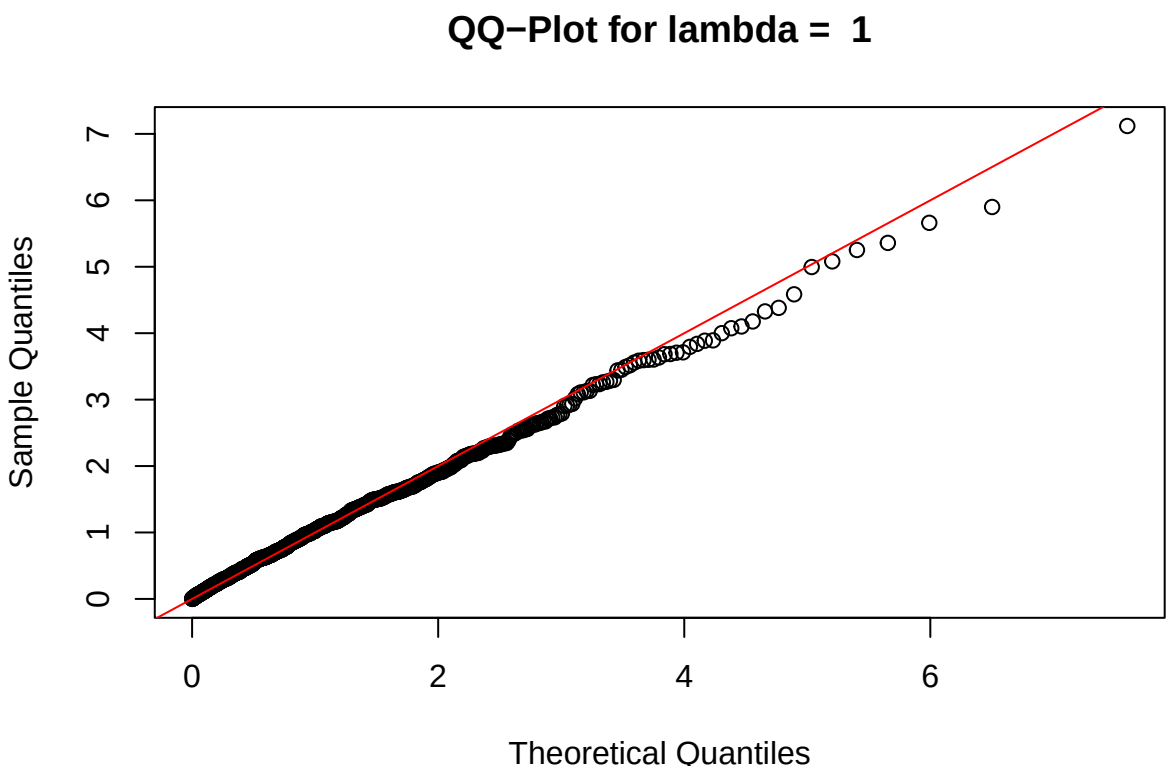
The subsection below contains the following code:
- The mathematical expression for exponential distribution.
- Function to calculate exponential distribution using runif.
- QQ-Plots for three different values of lambda.

The exponential distribution has the following cdf:

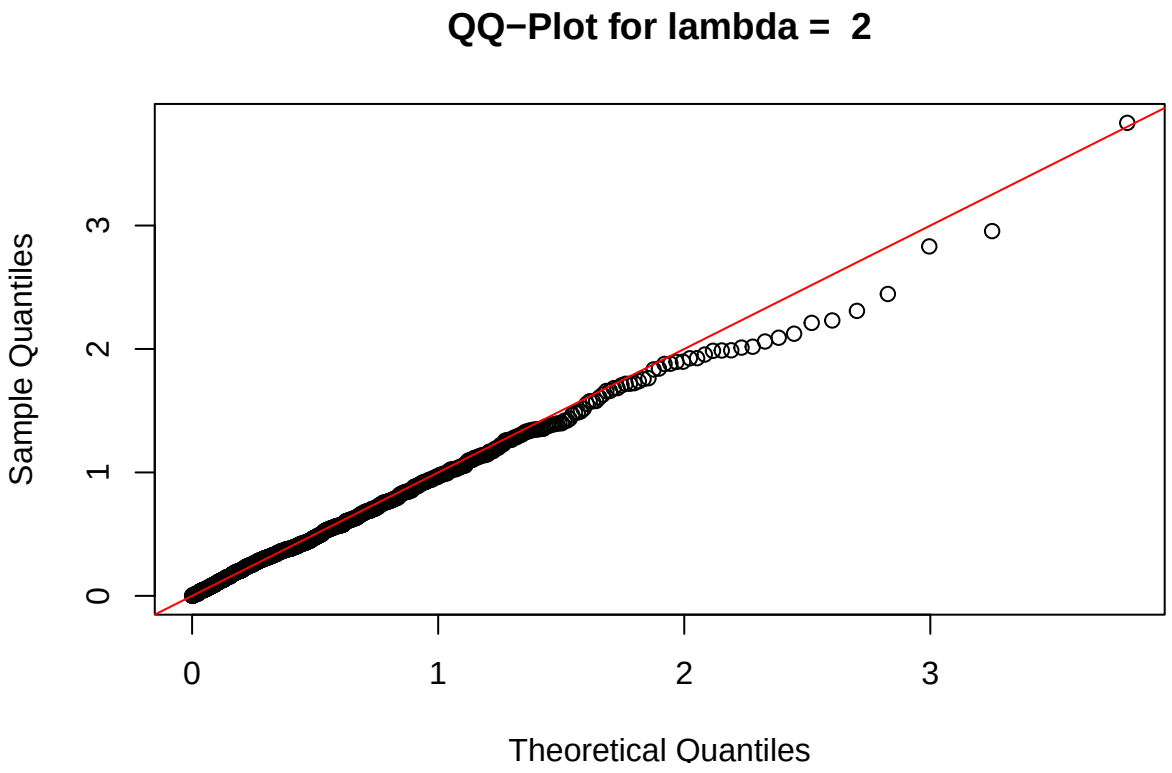
```
F(x) = 1 - exp(- ld * x),       ld > 0 and ld: lambda
```

if F(x) = u, then the mathematical expression to generate exponential random sample is: x = (- 1/ ld) * log(1-u)

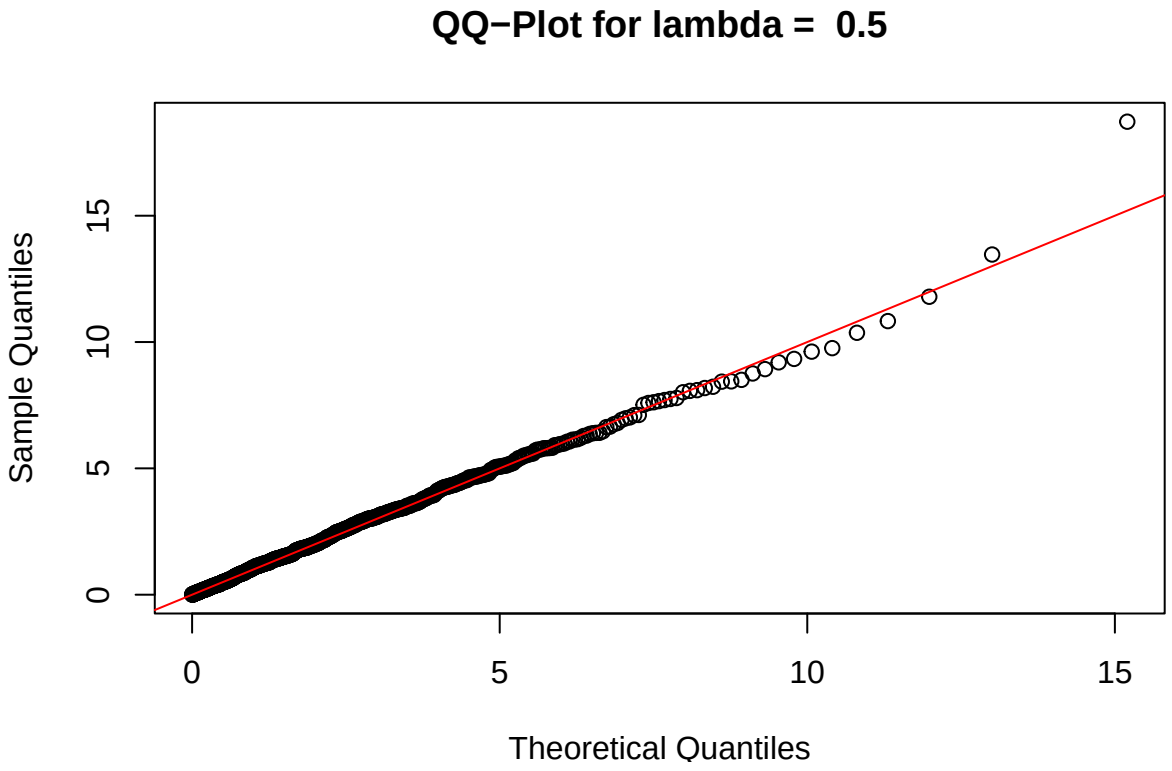
```r
## exponential distribution

exp <- function(l, n){
  u <- runif(n)
  exp_sample <- -log(1 - u) / l
  return(exp_sample)
}

## QQ-Plot function

qq_plot <- function(sample, l){
  qqplot(qexp(ppoints(length(sample)), rate = l), sample, main = paste("QQ-Plot for lambda = ", l),
         xlab = "Theoretical Quantiles", ylab = "Sample Quantiles")
  abline(0, 1, col = 'red')

}

## Setting three different lambda values

l1 <- 1
l2 <- 2
l3 <- 0.5
n <- 1000

sample1 <- exp(l1, n)
sample2 <- exp(l2, n)
sample3 <- exp(l3, n)

qq_plot(sample1, l1)
```



```
qq_plot(sample2, l2)
```




```
qq_plot(sample3, l3)
```



Observation:

Lambda = 0.5: For smaller values of lambda, the random samples tend to be larger, meaning that the decay of the distribution is slower.

Lambda = 1.0: The qq-plot shows a perfect fit for this lambda. This is the standard decay rate.

Lambda = 2.0: The decay of the distribution is faster and the samples are close to zero.

Beta Distribution

Task 3: Use acceptance-rejection approach to sample from a beta distribution

The subsection below contains the following code:
- Function for Beta distribution (2, 2)
- Function for Beta distributions where parameters are larger than 1 - Visualizations through histograms and QQ-plots

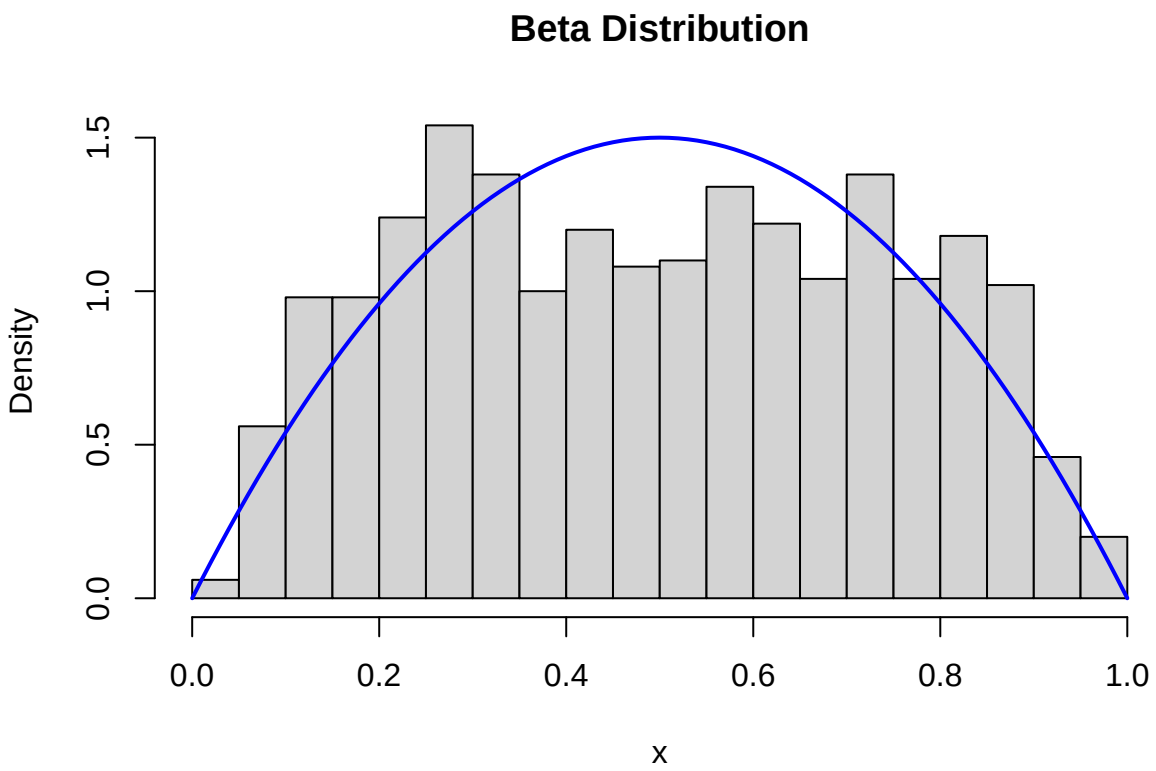
The function below (sam_beta) implements the acceptance-rejection method to a sample from a Beta(2,2) distribution.The sample is accepted if u <= 6 * x * (1-x)

The maximum value of the Beta PDF occurs at

```
x0 = (a-1) / (a + b - 2)   , where a, b > 1
```

For Beta(2, 2), a good constant to keep the rejection rate small is the maximum value of 6 * x * (1-x). At x = 0.5, the constant c = 1.5.

```
## Function sam_beta

sam_beta <- function(n, c) {
  sample <- numeric(n)

  while(c < n) {
    x <- runif(1)
    u <- runif(1)
    fx <- 6 * x * (1-x)

    if (u <= fx) {
      c <- c+1
      sample[c] <- x
    }
  }
  return(sample)
}

n <- 1000
c <- 1.5
b_sam <- sam_beta(n, c)

hist(b_sam, breaks = 30, probability = TRUE, main = "Beta Distribution", xlab = 'x')
curve(dbeta(x, 2, 2), col = 'blue', add = TRUE, lwd = 2)
```

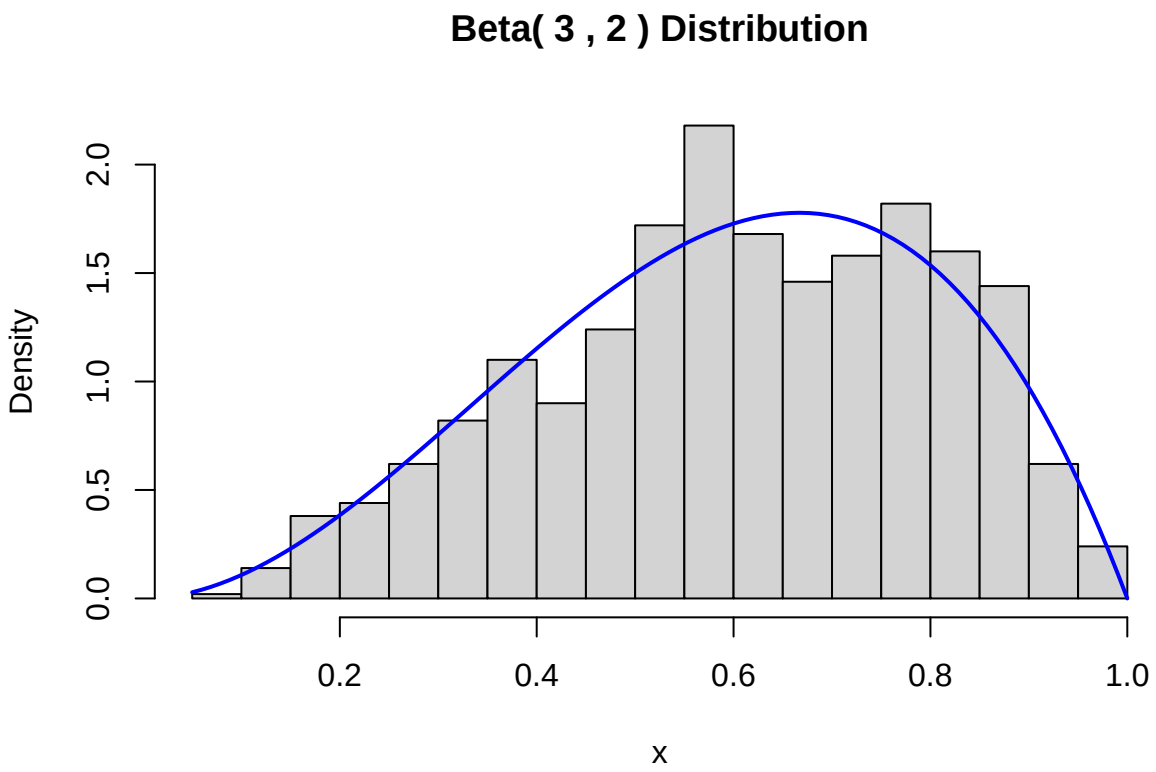


The function below (sample_alpbet) uses the acceptance-rejection method to a sample from an arbitrary Beta(a, b) distribution [where a <- alpha, b <- beta]. The constant c is dynamically calculated as the maximum value of Beta(a, b). The sample is accepted if u <= f(x: a, b) / c.

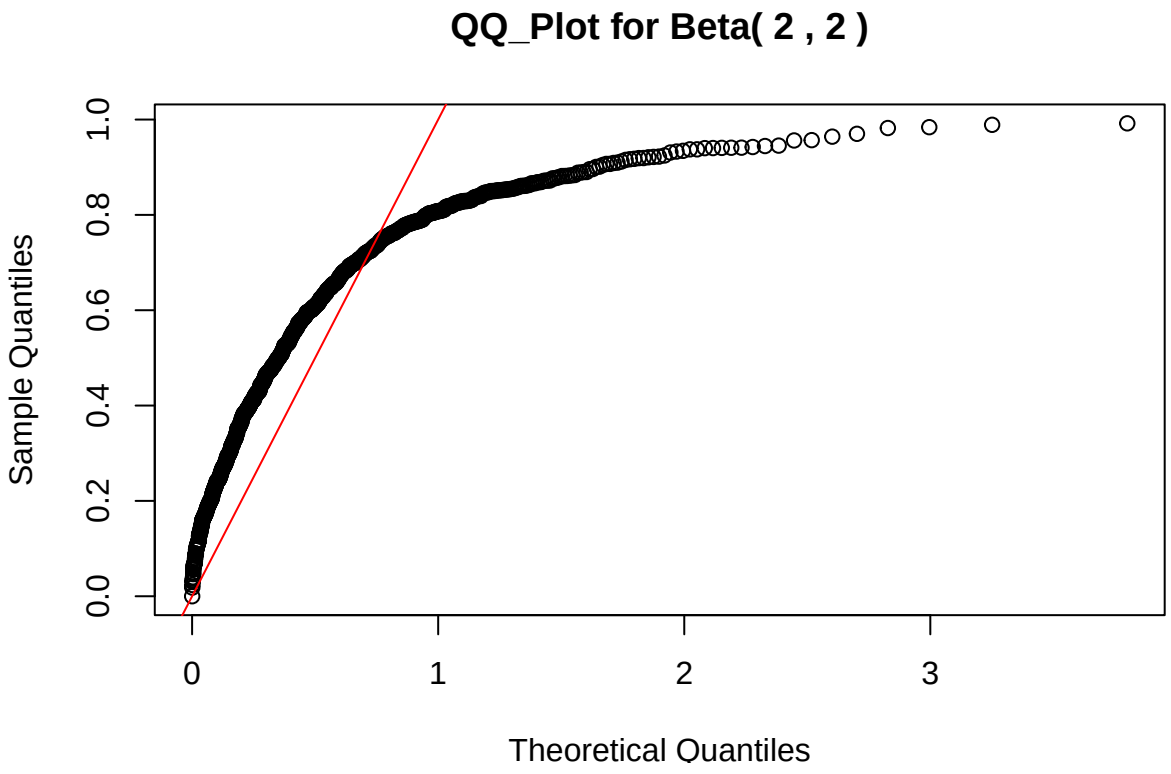
For Beta(3, 2), the constant c is calculated dynamically as the maximum value of dbeta.

```
## Function sample_alpha_beta

sample_alpbet <- function(a, b, n){
  sample <- numeric(n)
  c0 <- 0
  c <- max(dbeta(seq(0, 1, length.out = 1000), a, b))

  while (c0 < n) {
    x <- runif(1)
    u <- runif(1)
    fx <- dbeta(x, a, b)

    if (u <= fx / c){
      c0 <- c0 + 1
      sample[c0] <- x
    }
  }
  return(sample)
}

a <- 3
b <- 2
sam_ab <- sample_alpbet(a, b, n)

hist(sam_ab, breaks = 30, probability = TRUE,
```

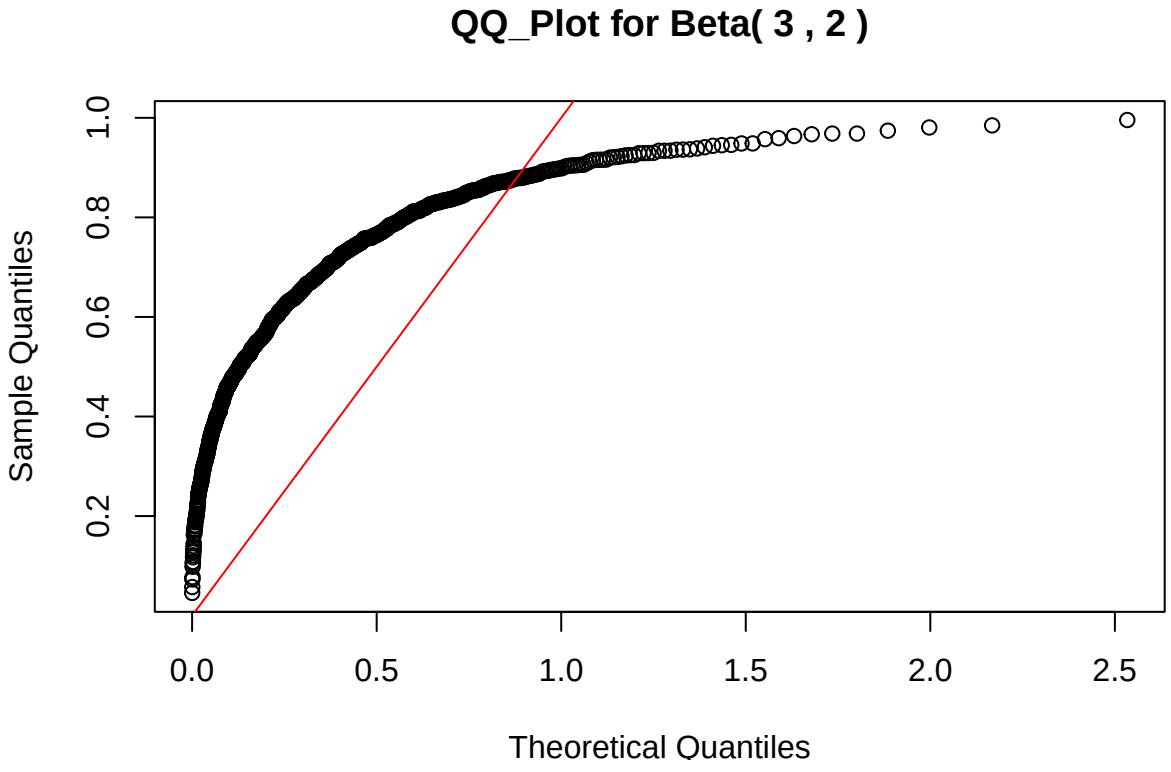
```
     main = paste("Beta(", a, ",", b, ") Distribution"), xlab = "x")
curve(dbeta(x, a, b), col = "blue", add = TRUE, lwd = 2)
```




```
qq_plot_a_b <- function(sample, a, b){
  qqplot(qexp(ppoints(length(sample)), a, b), sample, main = paste("QQ_Plot for Beta(", a, ",", b, ")")
         xlab = "Theoretical Quantiles", ylab = "Sample Quantiles")
  abline(0, 1, col = 'red')
}

n <- 1000
c <- 1.5
beta_2_2 <- sam_beta(n, c)

beta_3_2 <- sample_alpbet(3, 2, n)

qq_plot_a_b(beta_2_2, 2, 2)
```




```
qq_plot_a_b(beta_3_2, 3, 2)
```



Observation:

Beta(2, 2) Distribution:
- The histogram shows a good match between the sampled distribution and the theoretical curve, meaning

that the method works well.
- The QQ plot shows some deviation from the theoretical distributions, especially in the tails, meaning that the method doesn't work well with tail sampling.

Beta(3, 2) Distribution:
- The histogram shows a good match between the sampled distribution and the theoretical curve; even the general shape of the distribution and the peak aligns.
- The middle range in the QQ plot has a higher deviation (higher rejection rates), suggesting that a uniform distribution isn't perfectly suited for Beta(3, 2).