Exercise 2 - Random Number Generation through CDF and acceptance-rejection sampling

Konstantinos Vakalopoulos 12223236

2023-10-18

Contents

Task 1

Summarise the concept of pseudo-random number generation with Linear Congruential Random Number Generation Algorithm using the available code examples from the course to create a working code example and simulate one pseudo-random sample.

Try out and compare different values of m, a to illustrate the behavior of the method as described on slide 18.

According to the available codes examples and instructions from the course about the method Linear Congruential Random Number Generation Algorithm, we have to choose the parameter m to be as large as possible in order to not obtain the first element of the sequence after a number of iterations. Therefore, the whole method is characterized as cyclic and the largest the m parameter the higher the cycle length of the random number generator sequence. Furthermore, there is a second parameter, which is called "multiplier" a and is usually close to square root of m. Finally, the last parameter is the "increment" c, of which the value ranges between [0,m]. It is worth mentioning that first an initial integer value must be chosen in order to start the procedure. The initial value is called "starting value" or "seed". Also, the parameter m must be a prime number, which means that it is a number that can only be divided by itself and 1 without remainders.

Below, the Linear Congruential Random Number Generation Algorithm is presented. The function takes as initial values the sample n, the m and a parameter, the increment c and the starting value x0 and the uniform pseudo random numbers are generated.

```
# function for Linear Congruential Random Number Generator
mc.gen <- function(n,m,a,c=0,x0){
  us <- numeric(n)
  for (i in 1:n){
    x0 <- (a*x0+c) %% m
    us[i] <- x0 / m
    }
  return(us)
}
```

A simulation is performed by taking the values below and calling the function.

```
# Simulation
n <- 100
m <- 7919 # 1000th prime number. I could use the student ID
a <- round(sqrt(m),2)
c <- 1158
x0 <- 25
mc.gen(n,m,a,c,x0)
```

```
##   [1] 0.42716883 0.15998516 0.38331017 0.25700305 0.01693188 0.65299882
##   [7] 0.25659596 0.98070538 0.41920262 0.45107178 0.28710794 0.69596583
##  [13] 0.08023002 0.28589975 0.58844909 0.51231526 0.73716584 0.74661858
##  [19] 0.58781835 0.45618515 0.74214746 0.18993339 0.04840304 0.45361671
##  [25] 0.51358147 0.84984538 0.77397056 0.02187101 0.09253219 0.38066984
##  [31] 0.02203959 0.10753337 0.71562499 0.82969830 0.98108212 0.45272887
##  [37] 0.43457270 0.81885554 0.01618491 0.58652561 0.34114435 0.50466661
##  [43] 0.05651181 0.17521692 0.73878444 0.89065775 0.40586414 0.26408068
##  [49] 0.64677012 0.70230332 0.64420289 0.47384590 0.31377680 0.06922775
##  [55] 0.30680781 0.44905749 0.10785654 0.74438420 0.38898067 0.76162074
##  [61] 0.92286063 0.27159772 0.31571140 0.24138834 0.62737863 0.97665483
##  [67] 0.05874400 0.37385879 0.41592433 0.15933680 0.32561271 0.12250560
```

```
##  [73] 0.04800351 0.41806301 0.34965761 0.26226092 0.48482956 0.29121319
##  [79] 0.06129280 0.60067644 0.60042690 0.57822053 0.60207565 0.72494235
##  [85] 0.65885029 0.77731786 0.31974708 0.60052282 0.58675591 0.36163909
##  [91] 0.32849301 0.37882314 0.85770195 0.47312722 0.24982157 0.37785204
##  [97] 0.77128386 0.78278170 0.80597375 0.86983447
```

Below, three simulation are performed. The values of each parameter are chosen intuitively. According to these simulations, it is proven that the Linear Congruential Random Number Generator is a cyclic method. Also, the method depends extremely on the choice of m, a and c and it is difficult to predict the next value without knowing the m, a, c, and x0.

```
# Simulation 1
n <- 100
m <- 17
a <- 4
c <- 1
x0 <- 3
mc.gen(n,m,a,c,x0)
```

```
##   [1] 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118
##   [8] 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471
##  [15] 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059
##  [22] 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706
##  [29] 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118
##  [36] 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471
##  [43] 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059
##  [50] 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706
##  [57] 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118
##  [64] 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471
##  [71] 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059
##  [78] 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706
##  [85] 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471 0.5294118
##  [92] 0.1764706 0.7647059 0.1176471 0.5294118 0.1764706 0.7647059 0.1176471
##  [99] 0.5294118 0.1764706
```

```
# Simulation 2
n <- 100
m <- 37
a <- 12
c <- 9
x0 <- 7
mc.gen(n,m,a,c,x0)
```

```
##   [1] 0.5135135 0.4054054 0.1081081 0.5405405 0.7297297 0.0000000 0.2432432
##   [8] 0.1621622 0.1891892 0.5135135 0.4054054 0.1081081 0.5405405 0.7297297
##  [15] 0.0000000 0.2432432 0.1621622 0.1891892 0.5135135 0.4054054 0.1081081
##  [22] 0.5405405 0.7297297 0.0000000 0.2432432 0.1621622 0.1891892 0.5135135
##  [29] 0.4054054 0.1081081 0.5405405 0.7297297 0.0000000 0.2432432 0.1621622
##  [36] 0.1891892 0.5135135 0.4054054 0.1081081 0.5405405 0.7297297 0.0000000
##  [43] 0.2432432 0.1621622 0.1891892 0.5135135 0.4054054 0.1081081 0.5405405
##  [50] 0.7297297 0.0000000 0.2432432 0.1621622 0.1891892 0.5135135 0.4054054
##  [57] 0.1081081 0.5405405 0.7297297 0.0000000 0.2432432 0.1621622 0.1891892
##  [64] 0.5135135 0.4054054 0.1081081 0.5405405 0.7297297 0.0000000 0.2432432
```

```
##  [71] 0.1621622 0.1891892 0.5135135 0.4054054 0.1081081 0.5405405 0.7297297
##  [78] 0.0000000 0.2432432 0.1621622 0.1891892 0.5135135 0.4054054 0.1081081
##  [85] 0.5405405 0.7297297 0.0000000 0.2432432 0.1621622 0.1891892 0.5135135
##  [92] 0.4054054 0.1081081 0.5405405 0.7297297 0.0000000 0.2432432 0.1621622
##  [99] 0.1891892 0.5135135
```

```r
# Simulation 3
n <- 100
m <- 9
a <- 2
c <- 0
x0 <- 1
mc.gen(n,m,a,c,x0)
```

```
##   [1] 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222
##   [8] 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222 0.4444444
##  [15] 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222 0.4444444 0.8888889
##  [22] 0.7777778 0.5555556 0.1111111 0.2222222 0.4444444 0.8888889 0.7777778
##  [29] 0.5555556 0.1111111 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556
##  [36] 0.1111111 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111
##  [43] 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222
##  [50] 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222 0.4444444
##  [57] 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222 0.4444444 0.8888889
##  [64] 0.7777778 0.5555556 0.1111111 0.2222222 0.4444444 0.8888889 0.7777778
##  [71] 0.5555556 0.1111111 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556
##  [78] 0.1111111 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111
##  [85] 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222
##  [92] 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222 0.4444444
##  [99] 0.8888889 0.7777778
```

Task 2

The exponential distribution has the following cdf: $F(x) = 1 - e^{-\lambda x}$

Assume you can generate easily uniform random variables. How can you obtain then a random sample from the exponential distribution?

- Write down the mathematical expression for this.
- Write an R function which takes as input the number of samples required and the parameter λ. Use the function runif to create the uniform random variables.
- For three different values of λ create 1000 random samples and evaluate the quality of your random number generator using qq-plots which compare against a real exponential distribution with the specified parameter λ.

In order to obtain a random sample from the exponential distribution, we are going to use the inversion method. We have the cdf F(x) and we proceed to the next 3 steps:

1. Compute the quantile function F_x^{-1}
2. Generate a u ~ unif[0, 1]
3. Make the transformation $x = F_x^{-1}$

Therefore, the mathematical expression for the cdf F(x) will be: $x = -\frac{1}{\lambda}\ln(1-F)$

Consequently, the specific mathematical expression is used in order to generate random numbers. The function, below, takes as input the number of samples required and the parameter λ. Also, the function runif is used to create the uniform random variables.

```
exponential_rng <- function(sample,l){
  set.seed(12223236)
  variables <- runif(sample, min=0, max=1)
  x <- -log(1-variables)/l
  return(x)
}
```

After, 3 different values for the λ are taken and 1000 random samples have been created. For the λ, the values 1, 10 and 25 are used. The choice was done arbitrarily. Furthermore, a real exponential distribution with the specified parameter λ is used by using the function rexp() from R.

```
lambda <- c(1,10,25)
sample <- 1000

# Create an empty list to save the random numbers
simulations <- list()
real_exponential_distribution <- list()

# For loop for the 3 different values of lambda parameter
set.seed(12223236)
for (l in lambda){
  simulations <- c(simulations, list(exponential_rng(sample,l)))

  real_exponential_distribution <- c(real_exponential_distribution,
                                     list(rexp(sample, rate = l)))
}
```

```
par(mfrow = c(1, 2))
qqnorm(simulations[[1]], main="Implementation Exponential Distribution")
qqline(simulations[[1]])

qqnorm(real_exponential_distribution[[1]], main="Real Exponential Distribution")
qqline(real_exponential_distribution[[1]])
```

```
par(mfrow = c(1, 2))
qqnorm(simulations[[2]], main="Implementation Exponential Distribution")
qqline(simulations[[2]])

qqnorm(real_exponential_distribution[[2]], main="Real Exponential Distribution")
qqline(real_exponential_distribution[[2]])
```

```
par(mfrow = c(1, 2))
qqnorm(simulations[[3]], main="Implementation Exponential Distribution")
qqline(simulations[[3]])

qqnorm(real_exponential_distribution[[3]], main="Real Exponential Distribution")
qqline(real_exponential_distribution[[3]])
```

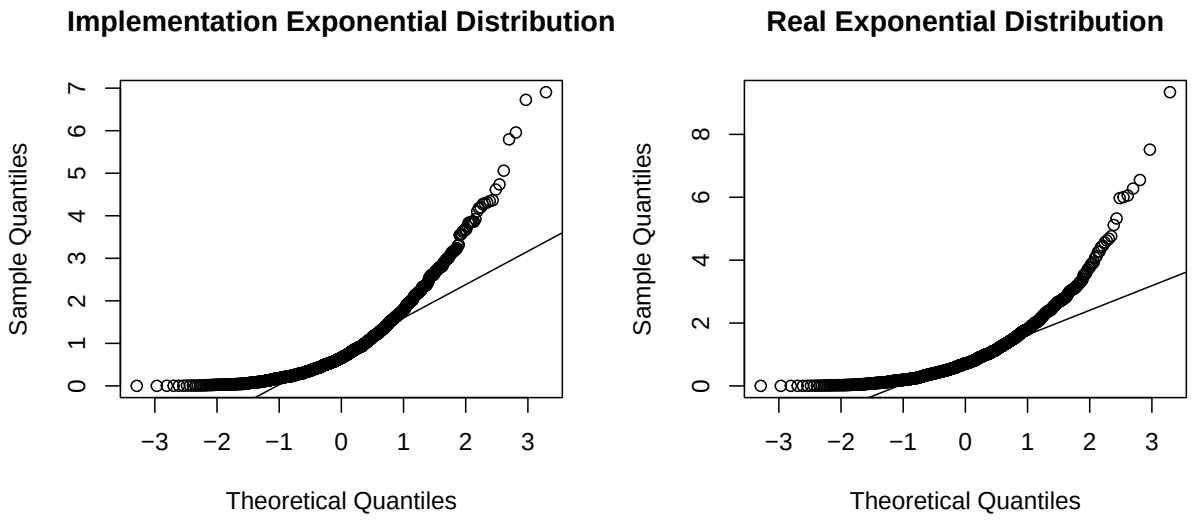

Figure 1: QQ plot for lambda 1

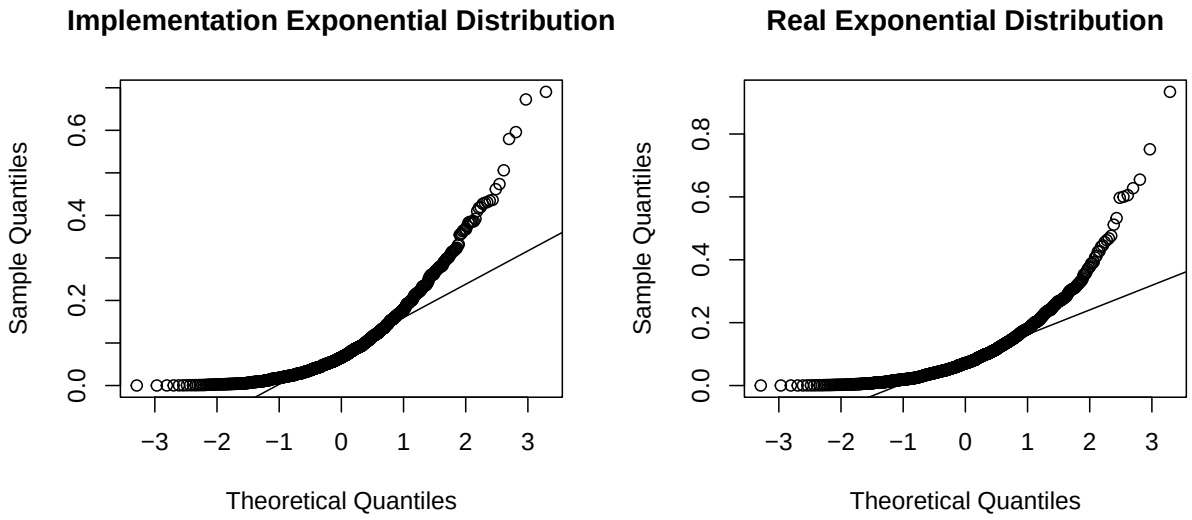

Figure 2: QQ plot for lambda 10

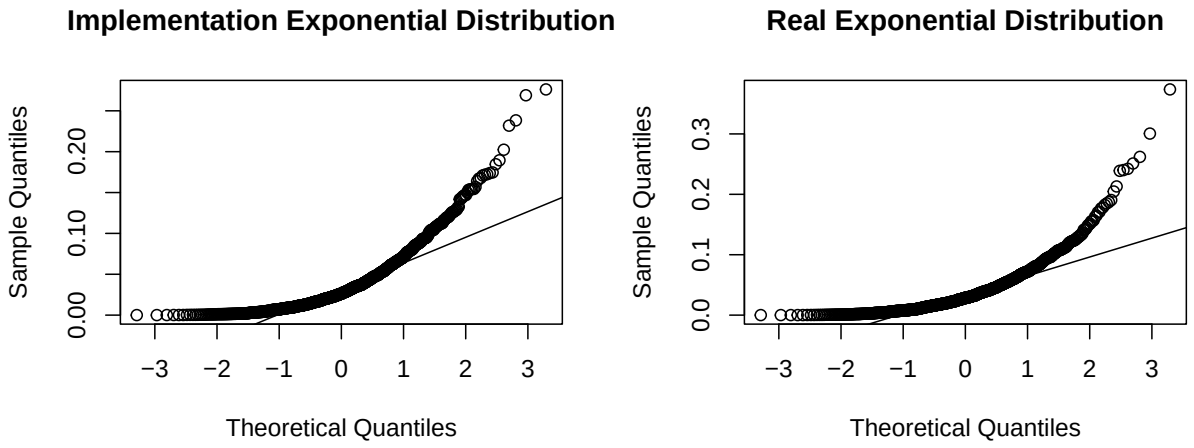

Figure 3: QQ plot for lambda 25

Task 3

The Beta distribution has the following pdf: $f(x;\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$

Write a function which uses an acceptance-rejection approach to sample from a beta distribution. Argue what is a natural candidate for a proposal distribution.

- Focus first on the case where both parameters of the beta distribution are 2. Find what is a good constant to keep the rejection proportion small.
- Write a function based on this approach to sample from an arbitrary beta distribution and which adapts the constant to the user specified parameters assuming that both parameters are larger than 1.

All functions above should take as input the number of samples to be created and if applicable also the parameters of the beta distribution.

Evaluate the quality of your different functions visually.

The f(x) is known, according to the instructions of the exercise, and it is a beta distribution with the following pdf: $f(x;\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$. Now, we have to find a density function g(x) so that the following inequality is fulfilled ($f(x) \leq cg(x)$). Therefore, we are able to perform the acceptance-rejection method and generate the proper values. In our implementation, the uniform distribution has been chosen. Thus, the g(x) will be: $g(x) = \frac{1}{b-a}$ where $x \in [a,b]$.

The uniform distribution, despite not being as efficient as other distributions, is characterized for its simplicity and its flexibility and versatility. In our case, we know the boundaries of the x variable, which are [0,1], since we have the pdf of a beta distribution. Thus, we can easily apply g(x). In terms of flexibility, by slightly changing the parameters a and b from $f(x;\alpha,\beta)$, our curve will be different every time which means that we can easily adapt g(x) by choosing an appropriate value for the parameter c.

Below, the acceptance rejection function is presented.

```
acceptance_rejection_method <- function(sample, a, b, c = 1){
  set.seed(12223236)  # Set a seed for reproducibility

  x <- numeric(sample)
```

```
  accepted <- 0
  rejection <- 0
  while (accepted < sample){

    u <- runif(1) # this is the u
    y <- runif(1) # this is g(x)
    f <- gamma(a+b)/(gamma(a)*gamma(b))*y^(a-1)*(1-y)^(b-1)

    # Check the acceptance
    if (u<=f/(c*y)){
      accepted <- accepted+1
      x[accepted] <- y
    }
    else{
      # Calculate the number of rejections in order to find rejection proportion
      rejection <- rejection + 1
    }
  }
  return (list(x,rejection))
}
```

The acceptance_rejection_method() function takes as input the sample size, the parameters a and b and constant parameter c. We set a seed for reproducibility purposes and we initialize the variables accepted and rejection which are responsible for number of accepted and rejected values, respectively. While the number of accepted values is lower than the sample size, the u and g(x) are generated and the f(x) is calculated. Afterwards, the value is accepted or not if the following condition is true: $u \leq \frac{f(x)}{cg(x)}$. If the condition is true, the accepted variable is increased by one and we accept the value. Otherwise, the rejection variable is increased by one. Finally, we return all the generated values and the number of rejected values in order to calculate then the rejection proportion. Our goal is to find a good constant to keep the rejection proportion small.

First, we focus on the case where both parameters of the beta distribution are 2. Therefore the initial values will be:

```
# initial values
sample <- 1000
a <- 2
b <- 2
```

Regarding the constant c, multiple simulations were performed in order to find the optimal one.

```
# Simulations for constant c
con <- seq(1.5,20,0.1) # the 1.5 value is the maximum for the f function
rejections <- c()
for (c in con){
  rejections <- c(rejections, acceptance_rejection_method(sample, a, b, c)[[2]])
}
cat("The minimum c, for g(x) to be uniform distribution, is:", con[which.min(rejections)])
```

```
## The minimum c, for g(x) to be uniform distribution, is: 1.5
```

It is worth mentioning that the constant c gets values from 1.5 to 20 with a step of 0.1. The starting value is 1.5 and it is chosen on purpose because $f(x;\alpha,\beta)$ has a maximum on y = 1.5 for x = 0.5. Therefore, if we take values for c lower than 1.5 the condition $f(x) \leq cg(x)$ is not fulfilled.

The next plot shows, also, why the $c = 1.5$ is the optimal value. As the c is increased, the number of rejections is increased as well.

```
plot(con,rejections, type = "l", lty = 1, ylab = "Number of rejections", xlab = "Constant c")
```

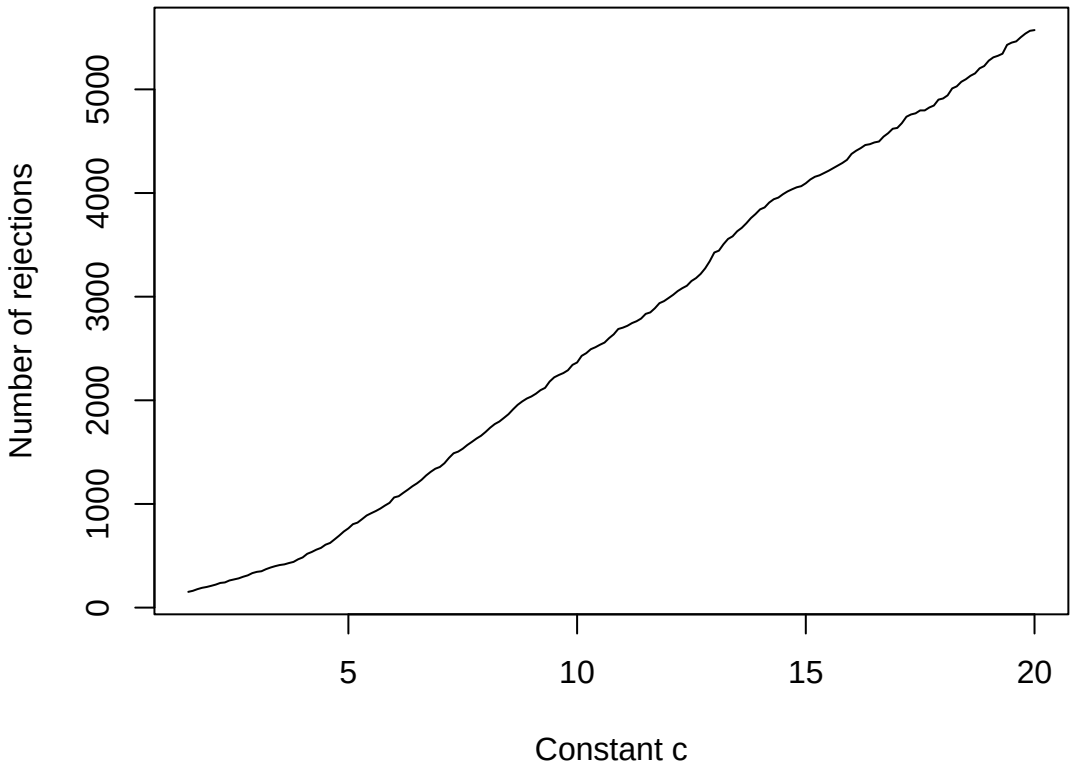

Figure 4: Rejections vs Constant c

In this part of the exercise, it is needed to write a function based on this approach to sample from an arbitrary beta distribution and which adapts the constant to the user specified parameters assuming that both parameters are larger than 1. Basically, we are going to generalize the previous question and perform the acceptance-rejection approach for the beta distribution for every α and β parameters chosen by the user. The function from the previous question will be used.

In a repetitive manner, the uniform distribution as g(x) will be used. In order, however, to find the optimal constant c, we are going to calculate the maximum value of the $f(x;\alpha,\beta)$ for each α and β. Regarding the maximum value, the function optimize() from R will be used.

```
# This is the f(x)
f = function(y,a,b) {
  gamma(a+b)/(gamma(a)*gamma(b))*y^(a-1)*(1-y)^(b-1)
}
```

The α and β will be (these value could change depending on the user's preferences):

```r
# Few demonstrations about the f function/beta distribution
# we take again uniform distribution
# Initial values
a <- 6
b <- 8
```

Then, we calculate the max of the $f(x; \alpha, \beta)$:

```r
# find the maximum of f(x) within the interval [0, 1]
ans = optimize(f, c(0,1), maximum = TRUE, a = a,b = b)

x_max = ans$maximum
y_max = ans$objective

cat("The max value of the beta distribution is: ", y_max)
```

```
## The max value of the beta distribution is:  2.97187
```

The plot below shows the $f(x; \alpha, \beta)$ and the maximum value. The abline() function indicates the pdf of the uniform distribution g(x).

```r
# plot f(x)
x = seq(0, 1, length.out = 100)
plot(x, f(x,a,b), type = 'l')

points(x_max, y_max, pch = 15)
text(x = x_max, y = y_max, labels = 'Maximum',
     pos = 4, col = 'blue')

abline(h = y_max, col = "red")
```

The same procedure, as it was done in the previous question, is repeated but this time the constant c will take values from y_max (maximum value of the f(x)) to 20 with a step of 0.1.

```r
# Simulations for constant c
# We show that for c greater than the max of the function f, the rejections are increased
con <- seq(y_max,20,0.1) # the y_max value is the maximum for the f function
rejections <- c()
for (c in con){
  rejections <- c(rejections, acceptance_rejection_method(sample, a, b, c)[[2]])
}
cat("The minimum c, for g(x) to be uniform distribution, is:", con[which.min(rejections)])
```

```
## The minimum c, for g(x) to be uniform distribution, is: 2.97187
```

The next plot shows, also, why the $c = y_{\max}$ is the optimal value. As the c is increased, the number of rejections is increased as well.

```r
plot(con,rejections, type = "l", lty = 1, ylab = "Number of rejections", xlab = "Constant c")
```

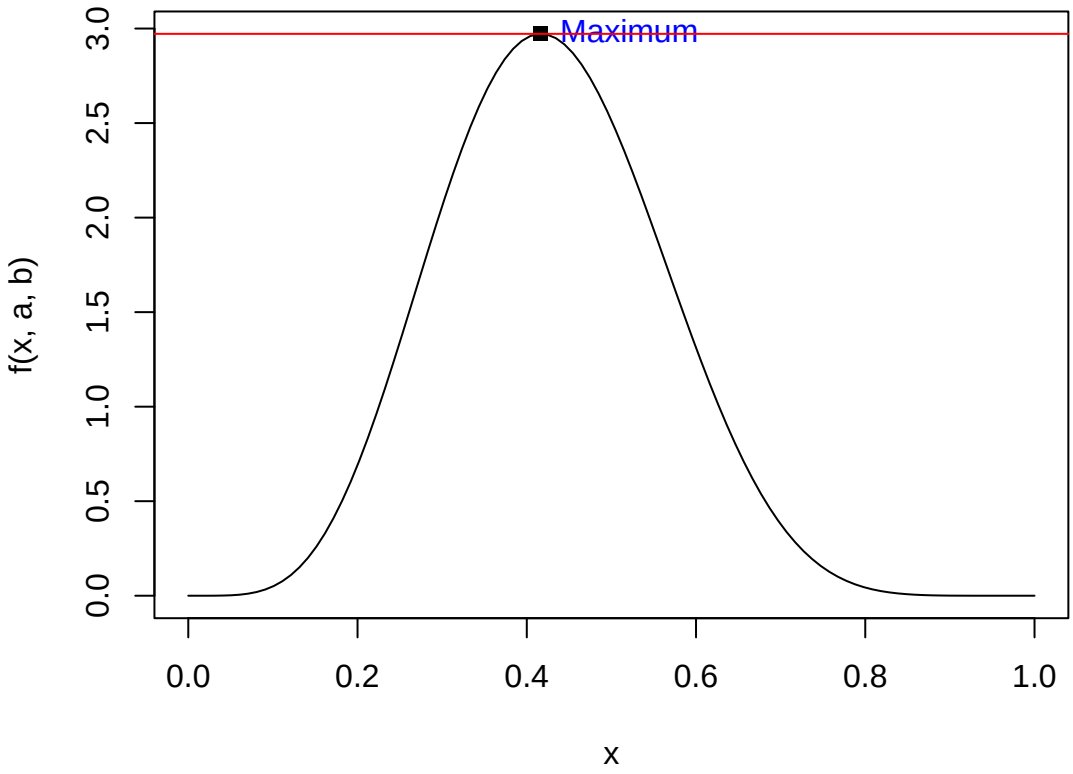

Figure 5: PDF of Beta Distribution

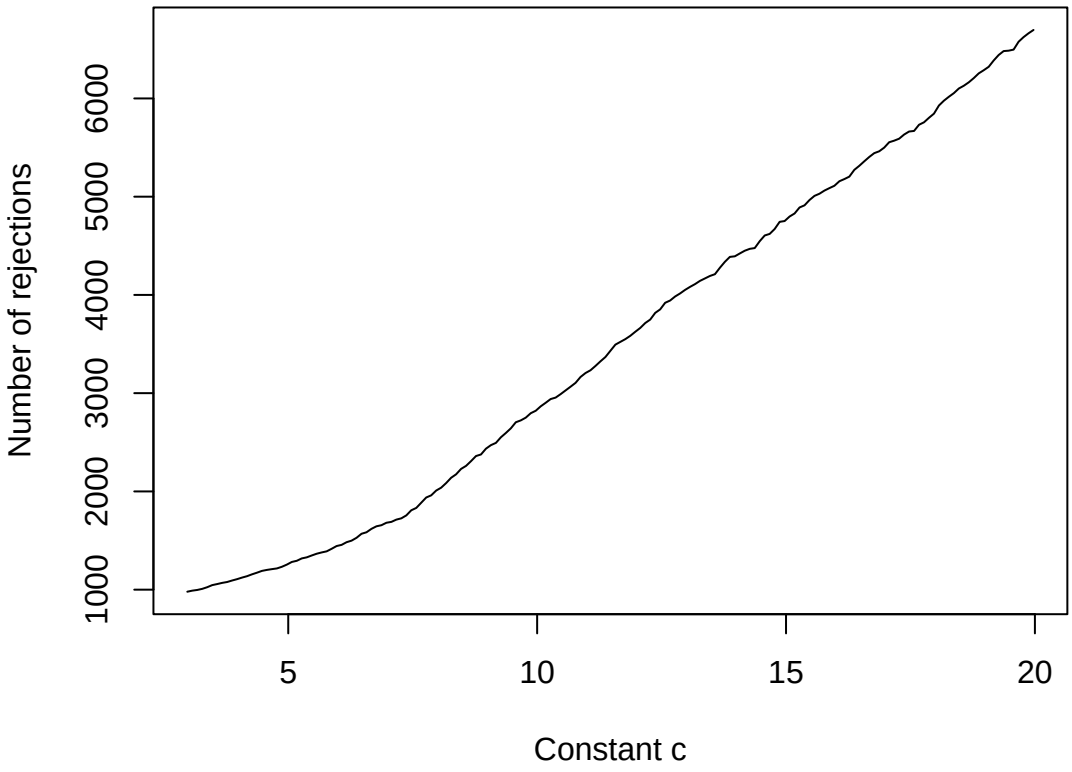

Figure 6: Rejections vs Constant c