Statistical Simulation and Computerintensive Methods

Exercise 2: Random Number Generation

Markus Kiesel | 1228952

11.10.2021

```r
# set seed for all experiments
set.seed(1228952)
```

1. Pseudo-random number generation with Linear Congruential Random Number Generation

Pseudo-random number generation starts with an initializing value called the seed and produces a sequence of numbers of the required length recursively. Thus, the sequence is reproducible for a given seed although the numbers look random.

An implementation of a linear congruential random number generation algorithm can look as follows:

```r
# n <- sample size
# m <- modulus
# a <- multiplier
# c <- increment
# x0 <- seed
lcrnga <- function(n, m, a, c=0, x0) {
  # create empty vector of class numeric (uniform sequence)
  us <- numeric(n)
  # loop to create all values
  for (i in 1:n) {
    #  x_i = (a · x_i-1 + c) (mod m)
    x0 <- (a*x0+c) %% m
    # add random number to sequence
    us[i] <- x0 / m
  }
 return(us)
}
```

Now we can test our algorithm and have a look at our created values. We use a prime number for the parameter m and a value near the square root of m for our parameter a.

We see that the values are in the range of 0 and 1 and all values are different.

```r
# run algorithm
us_first_try <- lcrnga(10,115249,341,0,1)
# range of values
print(paste(round(min(us_first_try), 4), round(max(us_first_try), 4)))
```

```
## [1] "0.003 0.7836"
```

```
# values
print(us_first_try)
```

```
##  [1] 0.002958811 0.008954525 0.053492872 0.241069337 0.204643858 0.783555606
##  [7] 0.192461540 0.629385071 0.620309070 0.525392845
```

Next, we can have a look at the cyclic property of this method. To see this easily we use very small values for **m**, **a** and **c**. We see clearly that for such small parameters the created values are the same after only some iterations. The largest possible cycle length is m, therefore m should be large.

```
# algorithm cycles at 7th value
us_cycle1 <- lcrnga(7,9,2,0,1)
print(us_cycle1)
```

```
## [1] 0.2222222 0.4444444 0.8888889 0.7777778 0.5555556 0.1111111 0.2222222
```

```
# algorithm cycles at 3rd value
us_cycle2 <- lcrnga(3,9,2,0,3)
print(us_cycle2)
```

```
## [1] 0.6666667 0.3333333 0.6666667
```

```
# algorithm cycles at 10th values
us_cycle3 <- lcrnga(10,9,4,1,0)
print(us_cycle3)
```

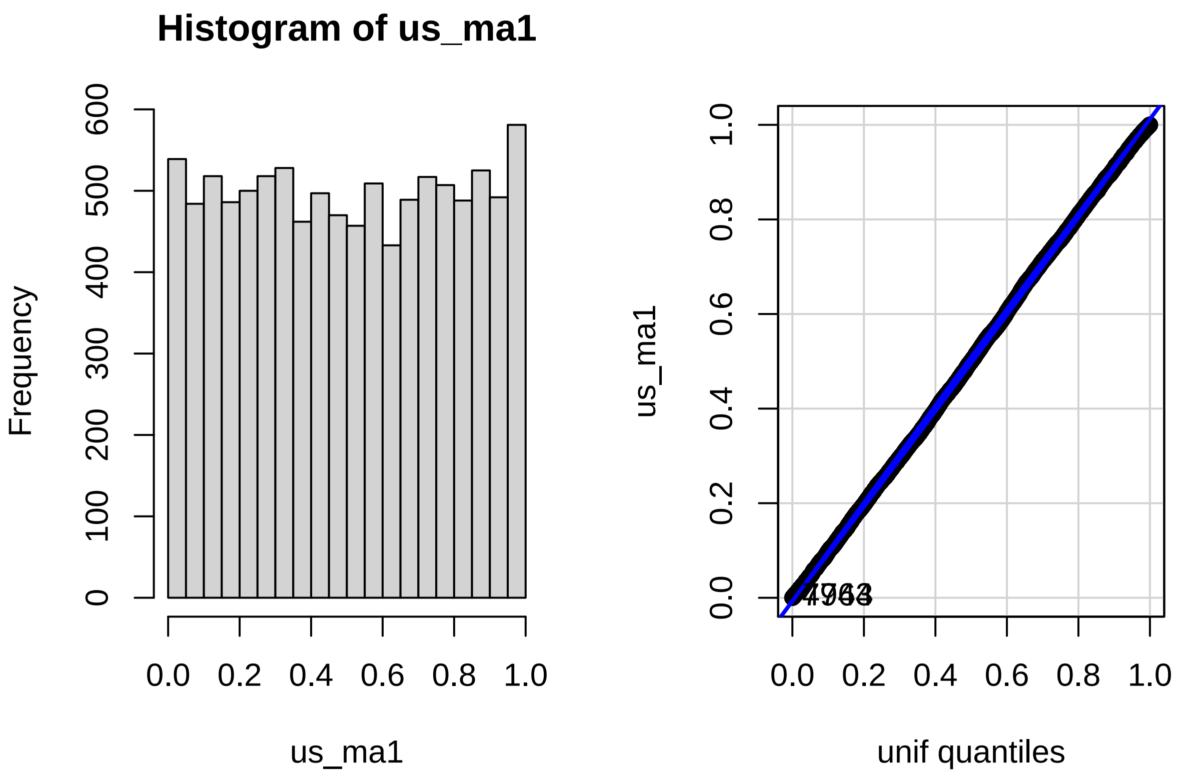
```
##  [1] 0.1111111 0.5555556 0.3333333 0.4444444 0.8888889 0.6666667 0.7777778
##  [8] 0.2222222 0.0000000 0.1111111
```

We inspect the impact of the parameters **m** and **a** further on the algorithm. In our first example we use a large prime number for the parameter **m** and for **a** we choose a value close to the square root of **m**.

The histogram and the QQPlot tell us that the random numbers created by the algorithm are uniformly distributed. So the algorithm works very well with this parameters.

```
# create sequence with:
# m <- large prime number
# a <- value close to square root of m
us_ma1 <- lcrnga(10000,115249,341,0,1)

# create plot of histogram and qqPlot
par(mfrow = c(1,2))
hist(us_ma1)
invisible(qqPlot(us_ma1, distribution = "unif"))
```

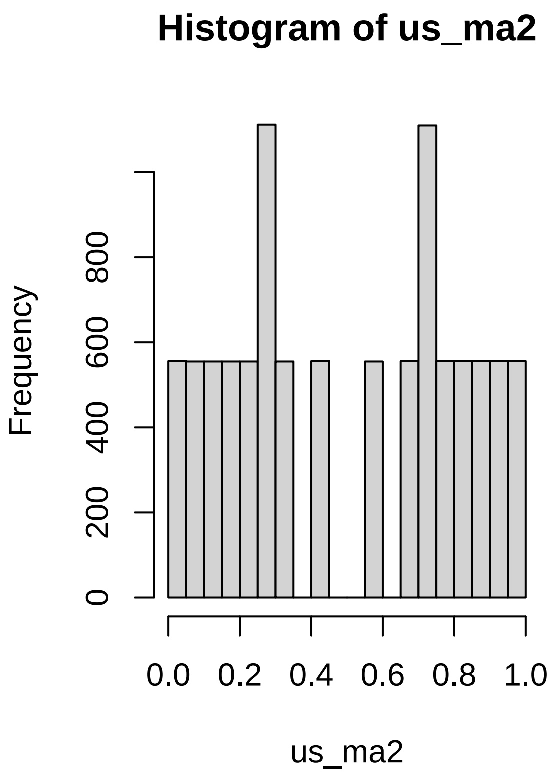


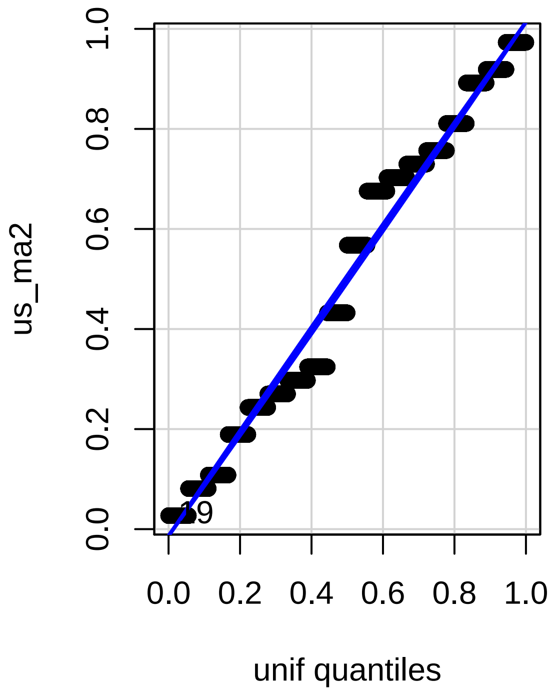
In our second example we do not use a prime number for our parameter **m** and **a** is a divisor of **m**.

The histogram and the QQPlot tell us that the random numbers created by the algorithm are not uniformly distributed. So the parameters have a big impact on the performance of the algorithm.

```r
# create sequence with:
# m <- not a prime number
# a <- divisor of m
us_ma2 <- lcrnga(10000,11877,321,0,1)

# create plot of histogram and qqPlot
par(mfrow = c(1,2))
hist(us_ma2)
invisible(qqPlot(us_ma2, distribution = "unif"))
```





2 The Exponential Distribution & Inversion Method

The exponential distribution has the following cumulative distribution function (cdf) F_x:

$F(x, \lambda) = 1 - \mathrm{e}^{-\lambda x}, x \geq 0$

To create random samples with an exponential distribution we can use the inversion method.

The first step is to compute the inverse function (quantile function) F_x^{-1}:

$F^{-1}(u, \lambda) = \frac{-ln(1-u)}{\lambda}, 0 \leq u < 1$

The second step is to generate a value $u \sim unif[0, 1]$

The third step is to make the transformation $x = F_x^{-1}(u)$

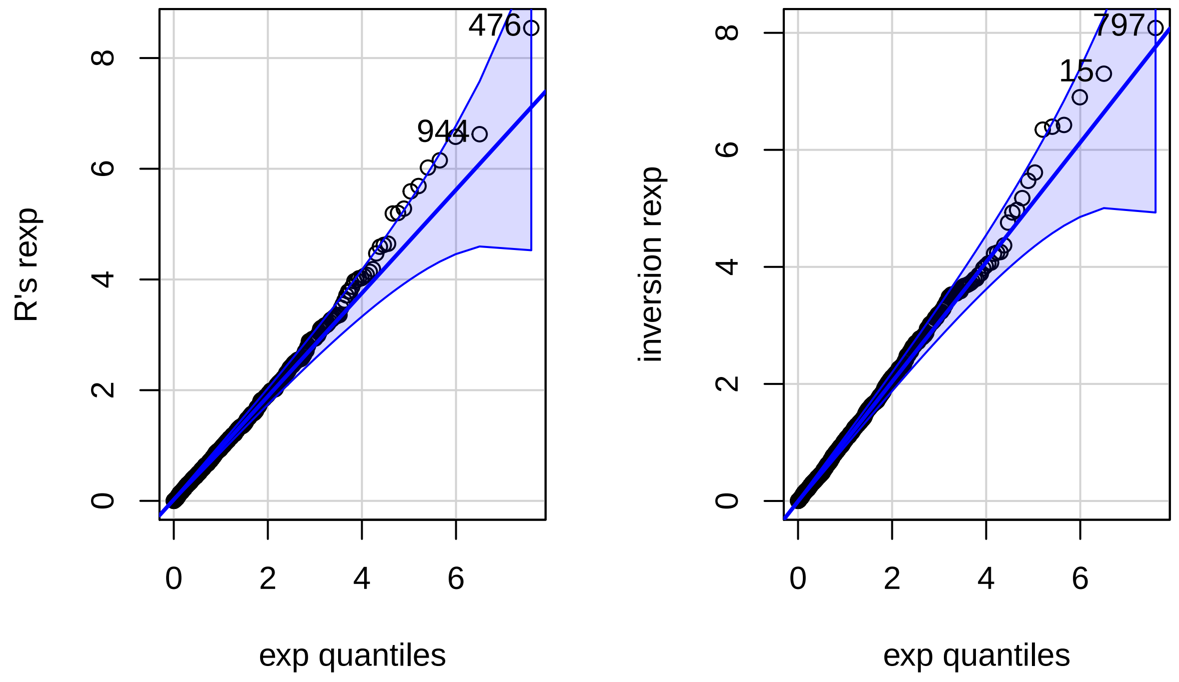
A function which implements this inversion method can look as follows.

```r
# function to create random samples with an exponential distribution
# n <- sample size
# rate <- rate parameter (lambda)
inversion_rexp <- function(n, rate = 1) {
  # create unifor distributet values
  u <- runif(n)
  # plug values into inverse cdf
  es <- -(log(1 - u)) / rate
  return(es)
}
```

Now we can have a look at the quality of our inversion rexp function by comparing it to R's rexp function.

We start with the default rate parameter of 1. For this comparison we see that both algorithms work very well concerning the distribution of the "smaller values" but the "larger values" are not so well distributed because only few values exist.

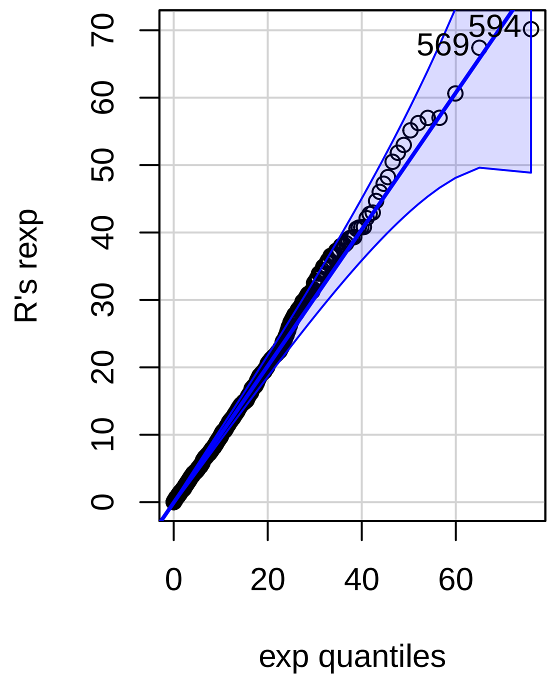
```r
# rate parameter
rate1 <- 1
# create random values with R's rexp function
rexp1 <- rexp(1000, rate = rate1)
# create random values with the inversion rexp function
inversion_rexp1 <- inversion_rexp(1000, rate = rate1)
# create Quantile Quantile Plots
par(mfrow = c(1,2))
invisible(qqPlot(rexp1, distribution = "exp", rate = rate1, ylab = "R's rexp"))
invisible(qqPlot(inversion_rexp1, distribution = "exp", rate = rate1, ylab = "inversion rexp"))
```

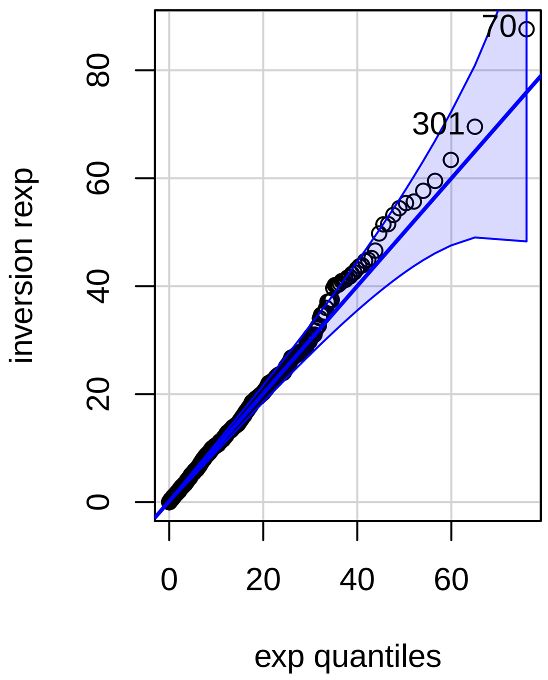


For the second comparison we use a rate parameter of 0.3.

The same observation we did for the first comparison is true for the exponential distribution with $\lambda = 0.3$. Only in the middle the distribution is a little more off.

```r
# rate parameter
rate2 <- 0.1
# create random values with R's rexp function
rexp2 <- rexp(1000, rate = rate2)
# create random values with the inversion rexp function
inversion_rexp2 <- inversion_rexp(1000, rate = rate2)
# create Quantile Quantile Plots
par(mfrow = c(1,2))
invisible(qqPlot(rexp2, distribution = "exp", rate = rate2, ylab = "R's rexp"))
invisible(qqPlot(inversion_rexp2, distribution = "exp", rate = rate2, ylab = "inversion rexp"))
```

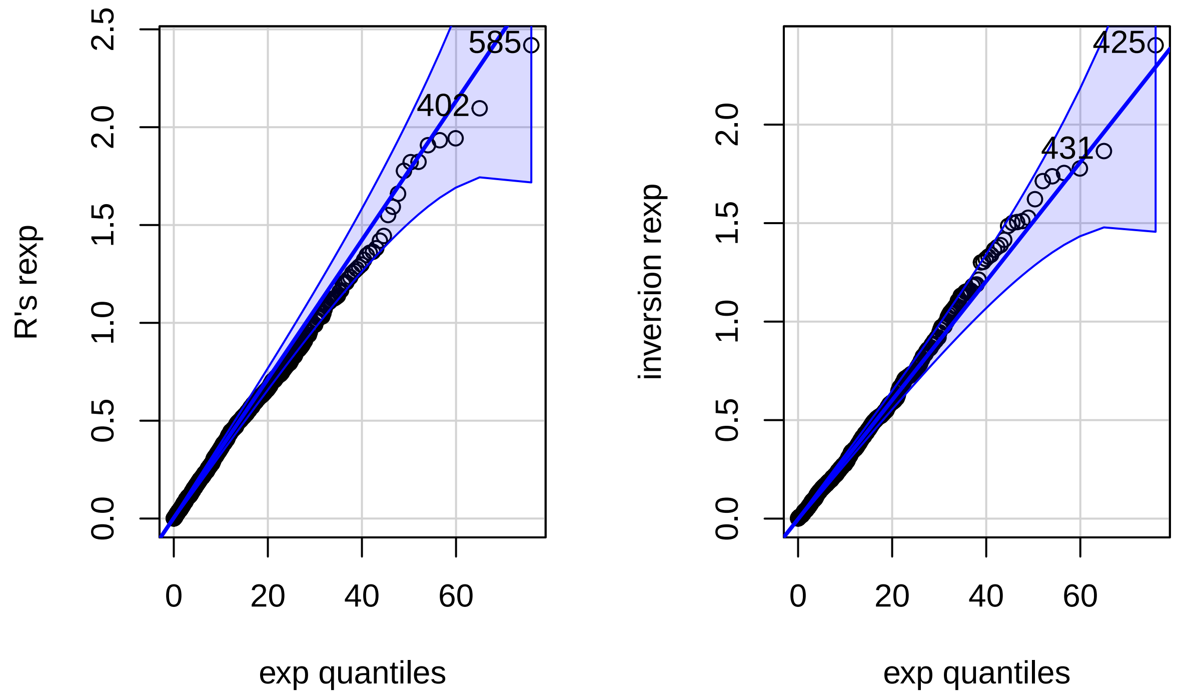




For the third comparison we use a rate parameter of 3.2.

We again observe the same quality of the distribution as for the other parameters we used.

```
# rate parameter
rate3 <- 3.2
# create random values with R's rexp function
rexp3 <- rexp(1000, rate = rate3)
# create random values with the inversion rexp function
inversion_rexp3 <- inversion_rexp(1000, rate = rate3)
# create Quantile Quantile Plots
par(mfrow = c(1,2))
invisible(qqPlot(rexp3, distribution = "exp", rate = rate2, ylab = "R's rexp"))
invisible(qqPlot(inversion_rexp3, distribution = "exp", rate = rate2, ylab = "inversion rexp"))
```



Overall the inversion method works very well compared to R's rexp function. The difference is only minimal. The main problem arises where only few values exist.

3 Beta Distribution & The Acceptance-Rejection Approach

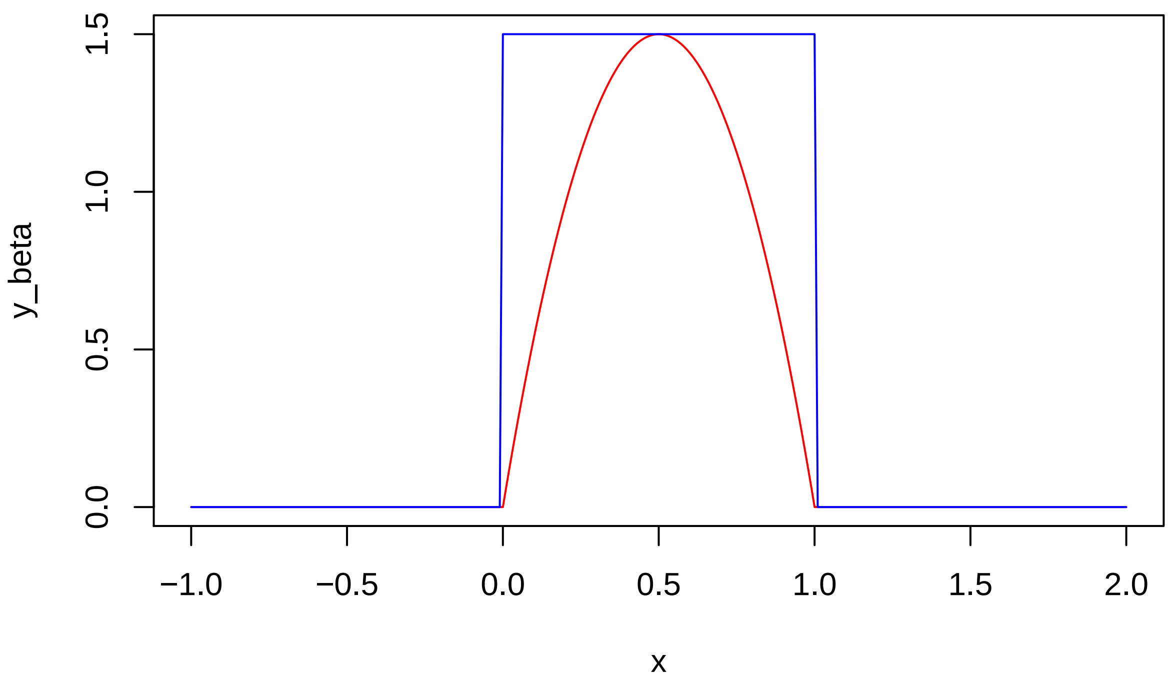
The beta distribution has the following probability density function (pdf):

$f(x; \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$

To create random samples with a beta distribution we use the acceptance-rejection approach. For this we need a function **g** where there exists a constant **c** such that $f(x) \leq cg(x)$ for all **x**. A natural candidate for this distribution is the uniform distribution because all values for both distributions are between 0 an 1. Also we can easily find a constant **c** that satisfies our condition because the pdf of a uniform distribution $unif[0, 1]$ is $g(x) = 1$.

To illustrate this we have a look at a beta distribution with the case $\alpha = \beta = 2$

```
# sequence of values between -1 and 2
x <- seq(-1, 2, by = 0.01)
# create values for beta distribution
y_beta <- dbeta(x, 2, 2)
plot(x, y_beta, type = "l", col = "red")
# constant c
c <- 1.5
# create values for uniform distribution
y_norm <- dunif(x) * c
lines(x, y_norm, col = "blue")
```



With this knowledge we can implement the following function to test the rejection proportion of the algorithm for different values of **c**. For this we use the special case $\alpha = \beta = 2$ again.

```r
# acceptance-rejection approach for beta distribution
# n <- sample size
# c <- constant c
accept_rejct_rp <- function(n, c) {
  x <- numeric(n)
  n_accepted <- 1
  iter <- 0
  while (n_accepted <= n) {
    # generate a random number y coming from g
    y <- runif(1)
    # generate a random number u comfing from unif[0,1]
    u <- runif(1)
    # accept x = y if u <= f(x) / (cg(x))
    if (u <= dbeta(y, 2, 2)/ (c*dunif(y))) {
      # add to x at current index
      x[n_accepted] <- y
      n_accepted <- n_accepted + 1
    }
    iter <- iter + 1
  }
  # return rejection proportion
  return(1 - (n / iter))
}
```

We also write a wrapper function to test the impact of the constant **c** on the rejection proportion.

```r
# wrapper to test the rjection proportion
# m <- number of runs of the algorithm
# n <- sample size
# cs <- values of c to test
wrapper_rej_prop <- function(m, n, cs) {
  n_cs <- length(cs)
  results <- numeric(n_cs)
  # loop for all values of c
  for (i in 1:n_cs) {
    rej_prop <- numeric(m)
    # loop algorithm m number of times
    for (j in 1:m) {
      rej_prop[j] <- accept_rejct_rp(n, cs[i])
    }
    # save mean rejection proportion
    results[i] <- mean(rej_prop)
  }
  return(results)
}
```

Now we can use these functions to test values from 1.5 to 10. We use 1.5 because it is the maximum value of a beta distribution with $\alpha = \beta = 2$ as can be seen by the output of the optimize function.

The results show that as we increase the constant c from its optimum 1.5 the rejection proportion increases. For a value of 1.5 we only reject a third of the samples whereas we reject 90% of all values for a constant of 10.

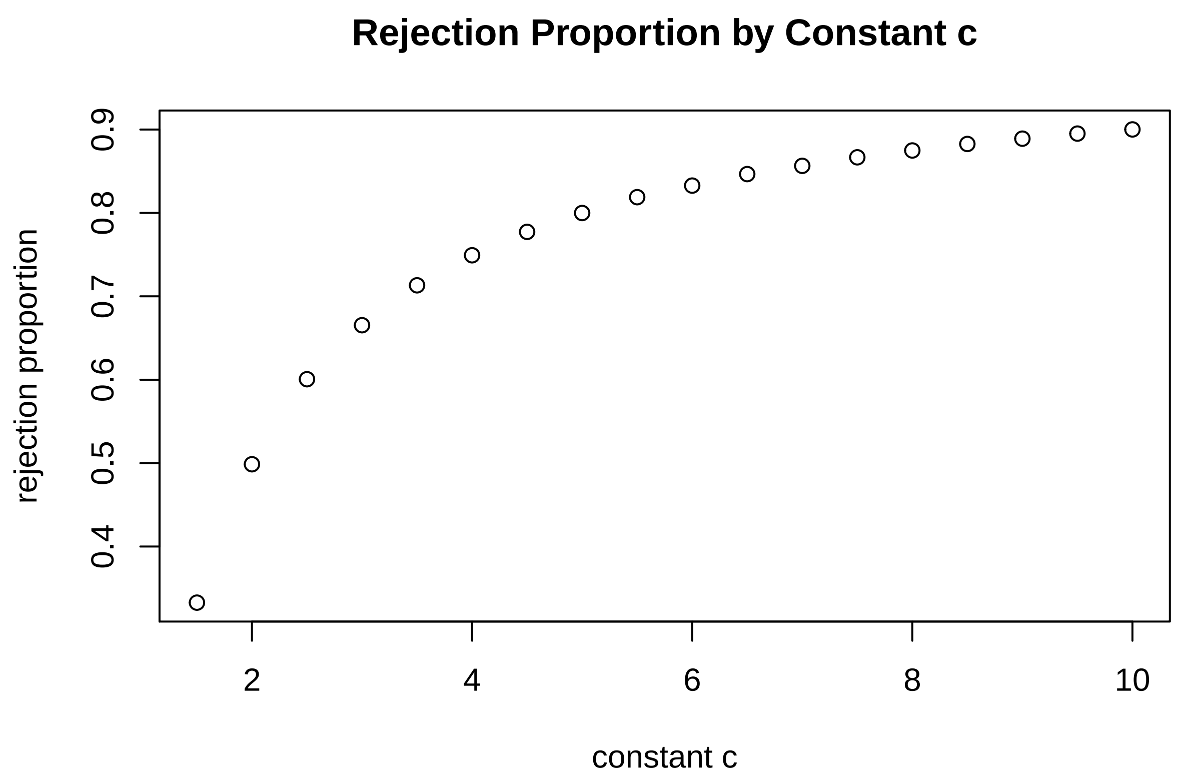
```
# maximum value for beta distribution with alpha = beta = 2
print(optimize(dbeta, interval=c(0, 1), maximum=TRUE, shape1 = 2, shape2 = 2)$objective)
```

```
## [1] 1.5
```

```
# create sequence for constant c
cs <- seq(1.5, 10, 0.5)
# run tests
rejection_proportions <- wrapper_rej_prop(100, 1000, cs)
print(rejection_proportions)
```

```
##  [1] 0.3327112 0.4986268 0.6006140 0.6653574 0.7132061 0.7492333 0.7772689
##  [8] 0.7998752 0.8188919 0.8327668 0.8465523 0.8564482 0.8666663 0.8748721
## [15] 0.8826708 0.8890486 0.8951128 0.9001041
```

```
# plot rejection proportions
plot(cs, rejection_proportions,
     main = "Rejection Proportion by Constant c",
     xlab = "constant c",
     ylab = "rejection proportion")
```



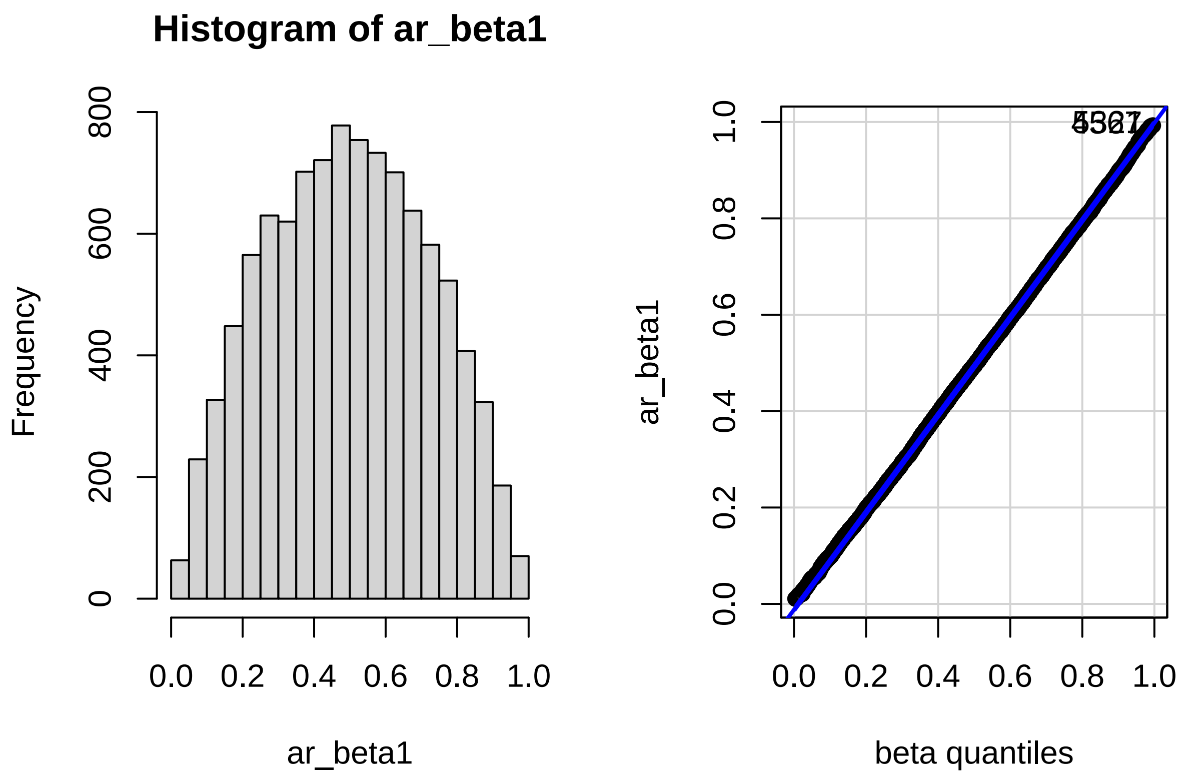
With this knowledge we implement a function to create random values from an arbitrary beta distribution.

```r
# acceptance rejection approach for arbitrary beta distribution
# n <- sample size
# s1 <- alpha or shape1
# s2 <- beta or shape2
accept_rejct_beta <- function(n, s1, s2) {
  x <- numeric(n)
  n_accepted <- 1
  # find the optimal value for c by using the optimize function
  c <- optimize(dbeta, interval=c(0, 1), maximum=TRUE, shape1 = s1, shape2 = s2)$objective
  while (n_accepted <= n) {
    # generate a random number y coming from g
    y <- runif(1)
    # generate a random number u comfing from unif[0,1]
    u <- runif(1)
    # accept x = y if u <= f(x) / (cg(x))
    if (u <= dbeta(y, s1, s2)/ (c*dunif(y))) {
      # add to x at current index
      x[n_accepted] <- y
      n_accepted <- n_accepted + 1
    }
  }
  return(x)
}
```

Now we evaluate the quality of our acceptance-rejection approach by visualizing some distributions with QQplots and Histograms.

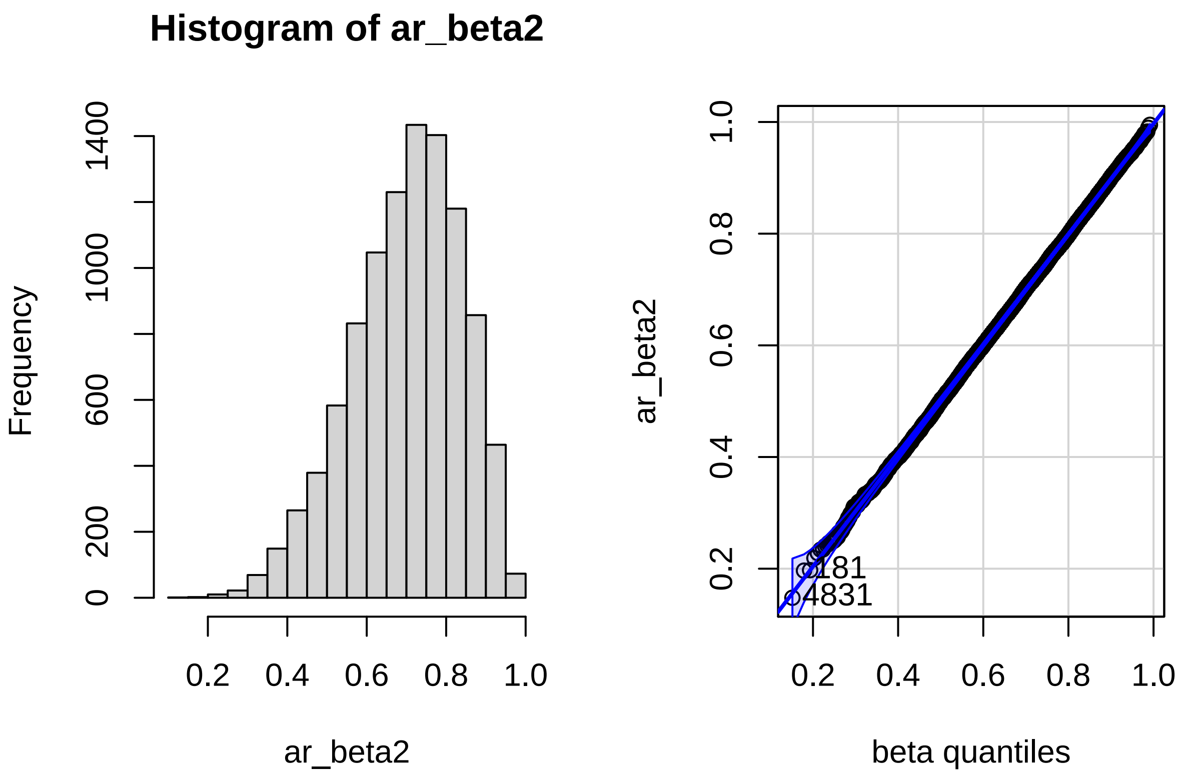
First, we look at the case $\alpha = \beta = 2$. The algorithm works very well and all random values are created according to the distribution.

```r
# set parameters
alpha <- 2
beta <- 2
# create random sample
ar_beta1 <- accept_rejct_beta(10000, alpha, beta)
# create QQPlot
par(mfrow = c(1,2))
hist(ar_beta1)
invisible(qqPlot(ar_beta1, distribution = "beta", shape1 = alpha, shape2 = beta))
```



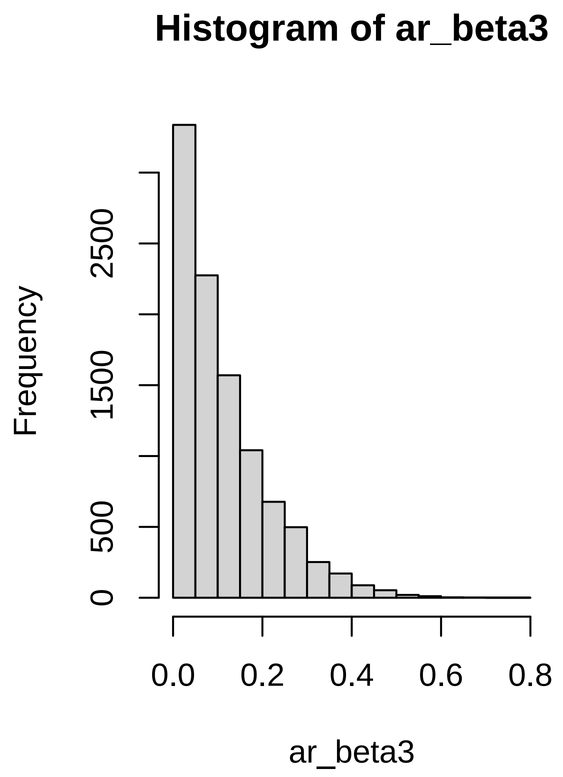
For the next test we use the parameters $\alpha = 7.0, \beta = 3.0$. We see that the algorithm works very well again. Only the tail is not that well distributed where fewer values exist.

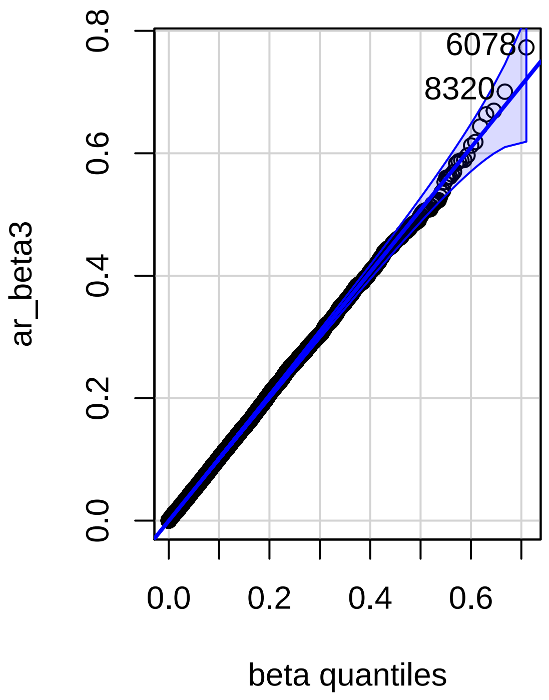
```r
# set parameters
alpha <- 7.0
beta <- 3.0
# create random sample
ar_beta2 <- accept_rejct_beta(10000, alpha, beta)
# create QQPlot
par(mfrow = c(1,2))
hist(ar_beta2)
invisible(qqPlot(ar_beta2, distribution = "beta", shape1 = alpha, shape2 = beta))
```



Last, we use a beta distribution with $\alpha = 1.0, \beta = 8.0$ where we can observe the same phenomenon that the tail with fewer values is not that well distributed but otherwise the algorithm's quality is very good.

```r
# set parameters
alpha <- 1.0
beta <- 8.0
# create random sample
ar_beta3 <- accept_rejct_beta(10000, alpha, beta)
# create QQPlot
par(mfrow = c(1,2))
hist(ar_beta3)
invisible(qqPlot(ar_beta3, distribution = "beta", shape1 = alpha, shape2 = beta))
```


Histogram of ar_beta3
Frequency
0
500
1500
2500
0.0
0.2
0.4
0.6
0.8
ar_beta3


ar_beta3
0.0
0.2
0.4
0.6
0.8
6078
8320
0.0
0.2
0.4
0.6
beta quantiles