Exercise 2 - Random Number Generation

Teodor Chakarov

2023-10-17

Contents


```r
set.seed(12141198)
```

Task 1. Linear Congruential Random Number Generation Algorithm

Why we need pseudo number generation in statistics? **Reproducibility -** Pseudo-random numbers allow for reproducibility in experiments and by applying same seed, will guarantee **control** over the experiment. **Control -** Pseudo-random sequences can be controlled and manipulated using seeds and algorithms, which is often necessary for testing and development purposes.

The **Linear Congruential Random Number Generation Algorithm** emits a sequence of not actually random but share many properties with completely random numbers. It works based on the following formula: *xn+1 = (a ∗ xn + c)mod m*

The parameters of the method are:

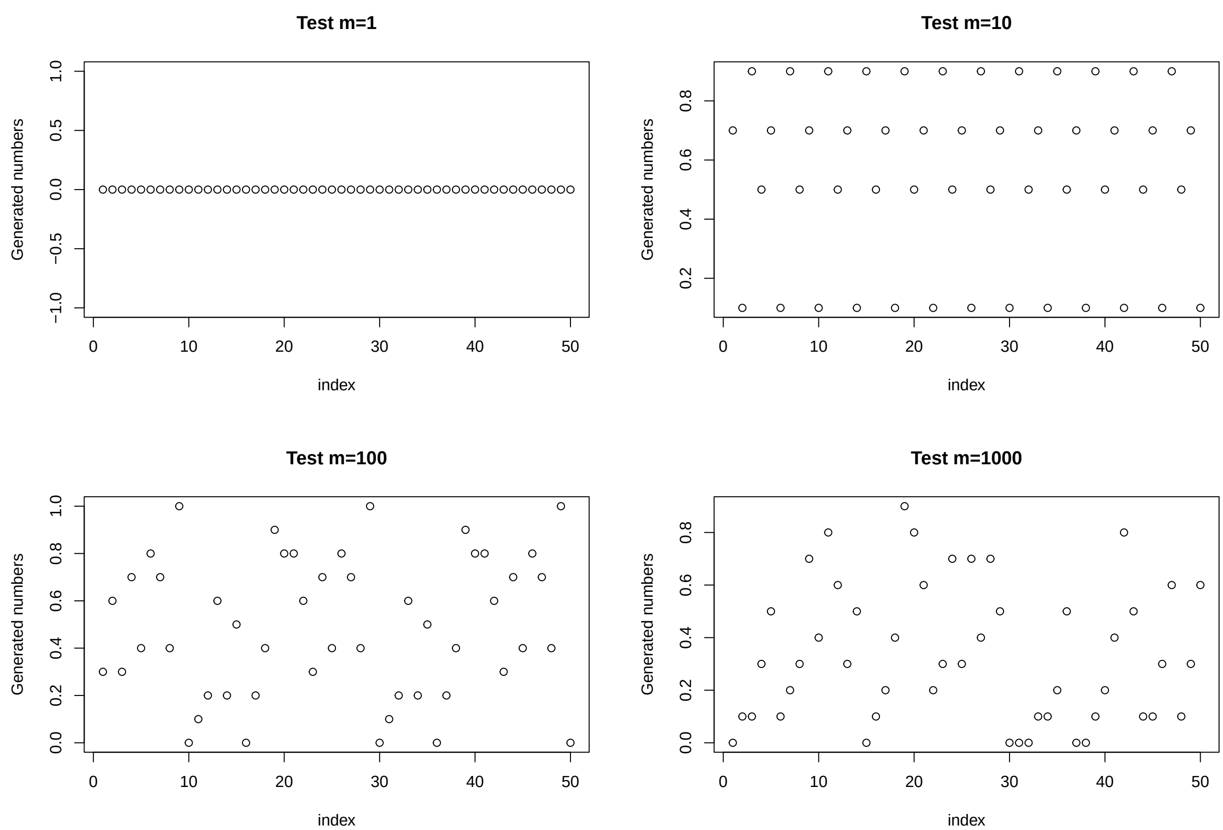
- xn+1: Next number in the sequence
- Modulus (m): (large integer) defines the upper bound of the generated numbers, and it also determines the period of the cycle of the generated sequence.
- Multiplier (a): The multiplier is a factor used in the formula to create the next number in the sequence.
- Increment (c): The value is factor which introduce additional randomness into the sequence.

```r
# Code from lecture slide #19

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)
}
```

Experiments for m

Lets see how the generated numbers are being generated with different value of m:

```
n <- 50

m_values <- c(1, 10, 100, 1000)

for(m in m_values) {
    plot(round(mc.gen(n, m=m, a=2, c=7, x0=10), 1), main=sprintf("Test m=%d", m), xlab="index", ylab="G
}
```



We have to keep in mind that the algorithm is cyclic, which means that, at some point, it will start to repeat itself (e.g. m=1, m=10) However, if the values chosen for 'a,' 'm,' and 'c' are well-tuned, the sequence can be quite long before it repeats.

```
# Generated numbers for m=10

mc.gen(n,m=10,a=2,c=7,x0=10)
```

```
##  [1] 0.7 0.1 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1 0.9
## [20] 0.5 0.7 0.1 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1
## [39] 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1 0.9 0.5 0.7 0.1
```

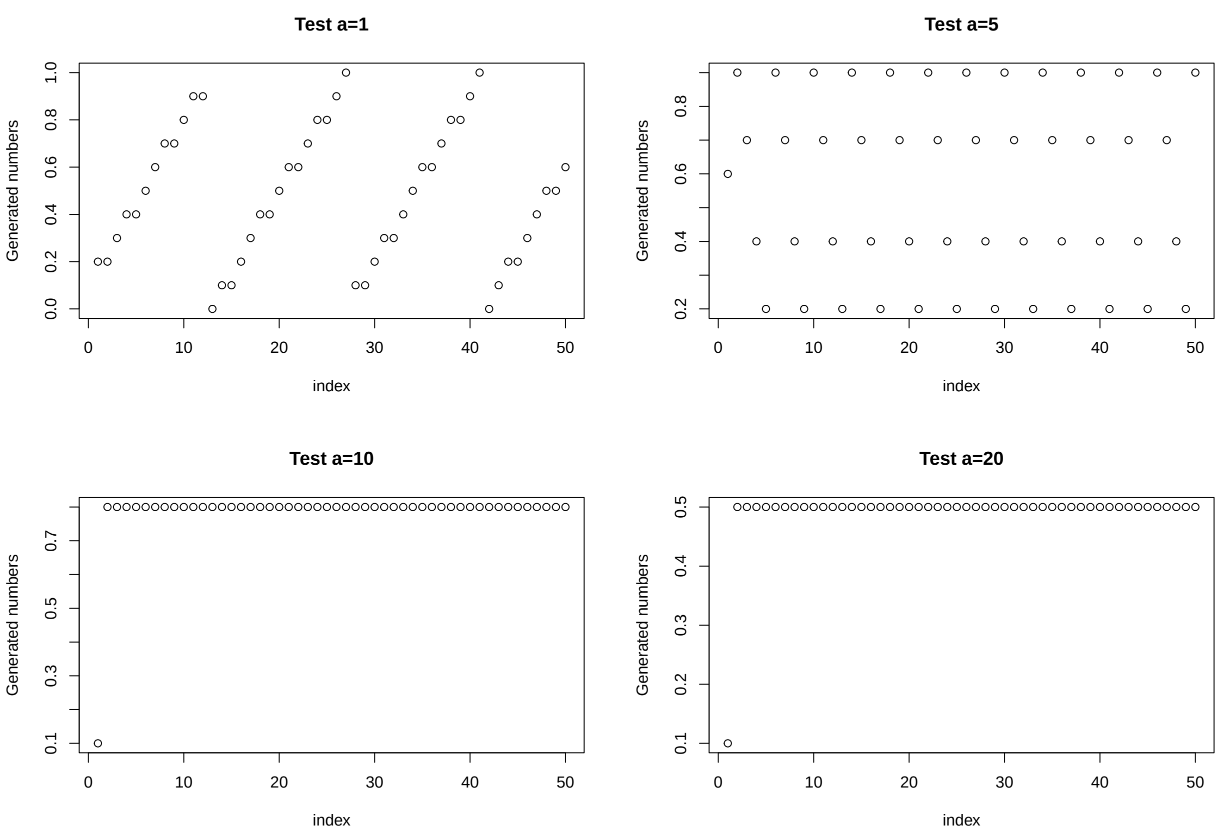
For m=100000 we can see the values are not equal so as long m is big we don't have repetitiveness.

```r
# Generated numbers for m=100000
mc.gen(n,m=100000,a=2,c=7,x0=10)
```

```
##  [1] 0.00027 0.00061 0.00129 0.00265 0.00537 0.01081 0.02169 0.04345 0.08697
## [10] 0.17401 0.34809 0.69625 0.39257 0.78521 0.57049 0.14105 0.28217 0.56441
## [19] 0.12889 0.25785 0.51577 0.03161 0.06329 0.12665 0.25337 0.50681 0.01369
## [28] 0.02745 0.05497 0.11001 0.22009 0.44025 0.88057 0.76121 0.52249 0.04505
## [37] 0.09017 0.18041 0.36089 0.72185 0.44377 0.88761 0.77529 0.55065 0.10137
## [46] 0.20281 0.40569 0.81145 0.62297 0.24601
```

Experiments for *a*

```r
a_values <- c(1, 5, 10, 20)

for(a in a_values) {
    plot(round(mc.gen(n, m=100, a=a, c=7, x0=10), 1), main=sprintf("Test a=%d", a), xlab="index", ylab=
}
```



When we experiment with the "a" multiplier we can still see the cyclic behavior of the number generator.As big the multiplier is we can see the same numbers generated for a=10 and a=20.

```r
mc.gen(n,m=100,a=10,c=7,x0=10)
```

```
##  [1] 0.07 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77
## [16] 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77
## [31] 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77
## [46] 0.77 0.77 0.77 0.77 0.77
```

Conclusion

The efficacy and reliability of the Linear Congruential Random Number Generation Algorithm (LCRNG) are relying on good pick for the given hyper-parameters. Well tuned values will give us better sparsity in the numbers, different numbers and representing all the region of possible values, not only partially.

Task 2. Exponential distribution

In order to generate random numbers in specific distribution, in this case **Exponential distribution**, we need to follow those steps:

- Begin by drawing random samples from a uniform distribution (between 0 and 1).
- The quantile function of the cumulative density function $F(x) = 1 - \exp(-\lambda x)$, where $\lambda > 0$, can be inverted to obtain the following expression: $F_x^{-1}(u) = \frac{-\ln(1-u)}{\lambda}$.
- Use the inverse CDF to transform each uniform random number to an exponentially distributed random number
- To obtain the random sample drawn from the exponential distribution we can use quantile function of the exponential distribution: $F_x^{-1}(u) = \frac{-ln(1-p)}{\lambda}$

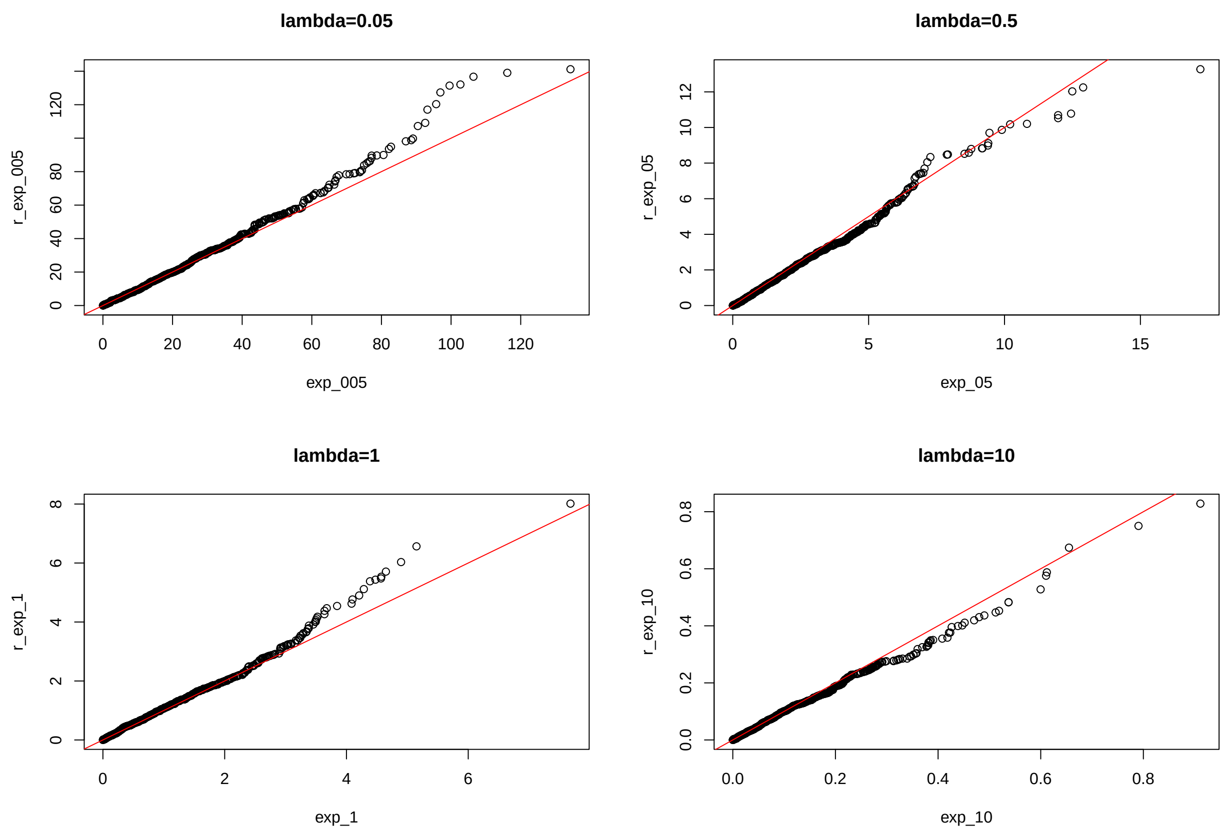
Implementation of the inversion method for exponential distribution With runif we obtain uniform random numbers.

```r
exp_func <- function(n_samples, lambda)
{
  dist <- -log(1-runif(n_samples))/lambda
  return(dist)
}
```

Now lets test the function.

```r
# Generate 1000 samples from our function
exp_005 <- exp_func(1000, 0.05)
exp_05 <- exp_func(1000, 0.5)
exp_1 <- exp_func(1000, 1)
exp_10 <- exp_func(1000, 10)
# Generate 1000 exponential distributed samples from r function
r_exp_005 <- rexp(1000, 0.05)
r_exp_05 <- rexp(1000, 0.5)
r_exp_1 <- rexp(1000, 1)
r_exp_10 <- rexp(1000, 10)

qqplot(exp_005, r_exp_005, main="lambda=0.05")
abline(a=0,b=1, col = "red")
qqplot(exp_05, r_exp_05, main="lambda=0.5")
```

```r
abline(a=0,b=1, col = "red")
qqplot(exp_1, r_exp_1, main="lambda=1")
abline(a=0,b=1, col = "red")
qqplot(exp_10, r_exp_10, main="lambda=10")
abline(a=0,b=1, col = "red")
```



When we are compare our exponensial numbers generated function with R function for sample generation, with help of the quantile-quantile plot (probability plot for comparing two probability distributions) we can observe the following things: * We can see how the values are becoming more sparced in the end of the line. * We can see for 2 different lambda values how our X-axis range is changing and the distribution of the data, how it has a slope.

Conclusion:

We can say that our values are following the red line which represents the exponential distribution. With values of lambda=0.5 we can see that the values are getting a bit under the line which can come from the randomness in the sample and the reproducibility value that we set.

Task 3. Acceptance-rejection approach to sample from a beta distribution

In order to generate beta distribution numbers (in interval (0,1)), we can use a uniform distribution values (since they are also distributed (0 to 1)).

We will use the Acceptance-Rejection method to approach a beta distribution. The Beta distribution has the following pdf: $f(x;\alpha,\beta)=\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{\alpha-1}(1-x)^{\beta-1}$

To define the proposal distribution, we will need a function that, when its multiplied by a constant C (C > 0), will be greater than the target function. This function should only be larger than the target function. A larger difference between the two functions will result in a lower acceptance rate.

General properties of acceptance-rejection approach:

- Constant c is always larger than 1.
- Instrumental distribution should be chosen that has the smallest value c.
- To obtain n samples from f requires then roughly cn random samples from g and from instrumental distribution.

We can also create a function that employs an acceptance-rejection approach to sample from a beta distribution:

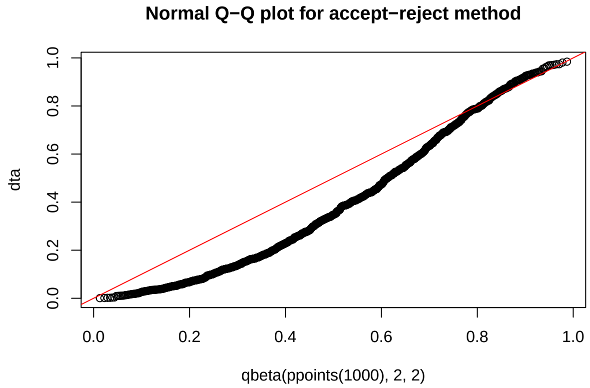
```
accept_or_reject <- function(n, alpha, beta, const){
  set.seed(000001)
  iter <- 0
  accepted <- 0
  data <- numeric(n)

  while(accepted<n)
  {
    iter <- iter+1
    #Candidate from the proposal distribution
    u <- runif(1)
    #Candidate from the proposal distribution
    y <- runif(1)

    if (dbeta(u, alpha, beta)/(const*dunif(u)) >= y)
    {
      accepted <- accepted + 1
      data[accepted] <- y
    }
  }
  acceptance_prop <- round(accepted/iter, 4)*100
  print(paste("Acceptance proportion in %: ", acceptance_prop))
  return(data)
}
```

First we set when alpha and beta are 2, we will take 1.5 as the **c** value:

```
dta <- accept_or_reject(1000,2,2,1.5)
```

```
## [1] "Acceptance proportion in %:  67.16"
```

```
qqplot(qbeta(ppoints(1000), 2, 2),dta, main="Normal Q-Q plot for accept-reject method")
abline(a=0, b=1, col='red')
```



Based on the qq plot we can see the generated sample vs the theoretical one. Our sample has a S shape, but the values are in range as expected from 0 to 1.

I will do a sample search in order to pick a good hyper-parameter for our model. For each constant c generate 1000 samples based on the acceptance-rejection algorithm implemented above with alpha = 2, beta = 2 and calculate the acceptance rate

```
test_c <- seq(from = 1, to = 5, by = 0.1)

acc_rate <- c()
for(i in 1:length(test_c)) {
  k <- 0
  print(paste("Given C: ", i))
  k <- accept_or_reject(1000, 2, 2, test_c[i])
  acc_rate <- append(acc_rate, k[[1]])
}
```

```
## [1] "Given C:  1"
## [1] "Acceptance proportion in %:  79.94"
## [1] "Given C:  2"
## [1] "Acceptance proportion in %:  77.34"
## [1] "Given C:  3"
## [1] "Acceptance proportion in %:  75.93"
## [1] "Given C:  4"
## [1] "Acceptance proportion in %:  72.99"
## [1] "Given C:  5"
## [1] "Acceptance proportion in %:  70.77"
## [1] "Given C:  6"
## [1] "Acceptance proportion in %:  67.16"
## [1] "Given C:  7"
## [1] "Acceptance proportion in %:  62.31"
## [1] "Given C:  8"
## [1] "Acceptance proportion in %:  57.67"
## [1] "Given C:  9"
## [1] "Acceptance proportion in %:  54.02"
## [1] "Given C:  10"
## [1] "Acceptance proportion in %:  51.52"
## [1] "Given C:  11"
## [1] "Acceptance proportion in %:  48.97"
## [1] "Given C:  12"
## [1] "Acceptance proportion in %:  46.88"
## [1] "Given C:  13"
```

```
## [1] "Acceptance proportion in %:  45.56"
## [1] "Given C:  14"
## [1] "Acceptance proportion in %:  43.88"
## [1] "Given C:  15"
## [1] "Acceptance proportion in %:  42.41"
## [1] "Given C:  16"
## [1] "Acceptance proportion in %:  40.13"
## [1] "Given C:  17"
## [1] "Acceptance proportion in %:  38.14"
## [1] "Given C:  18"
## [1] "Acceptance proportion in %:  36.74"
## [1] "Given C:  19"
## [1] "Acceptance proportion in %:  35.3"
## [1] "Given C:  20"
## [1] "Acceptance proportion in %:  33.76"
## [1] "Given C:  21"
## [1] "Acceptance proportion in %:  32.4"
## [1] "Given C:  22"
## [1] "Acceptance proportion in %:  31.44"
## [1] "Given C:  23"
## [1] "Acceptance proportion in %:  30.66"
## [1] "Given C:  24"
## [1] "Acceptance proportion in %:  29.5"
## [1] "Given C:  25"
## [1] "Acceptance proportion in %:  28.4"
## [1] "Given C:  26"
## [1] "Acceptance proportion in %:  27.7"
## [1] "Given C:  27"
## [1] "Acceptance proportion in %:  26.85"
## [1] "Given C:  28"
## [1] "Acceptance proportion in %:  25.82"
## [1] "Given C:  29"
## [1] "Acceptance proportion in %:  25.57"
## [1] "Given C:  30"
## [1] "Acceptance proportion in %:  24.99"
## [1] "Given C:  31"
## [1] "Acceptance proportion in %:  24.45"
## [1] "Given C:  32"
## [1] "Acceptance proportion in %:  23.83"
## [1] "Given C:  33"
## [1] "Acceptance proportion in %:  23.03"
## [1] "Given C:  34"
## [1] "Acceptance proportion in %:  22.51"
## [1] "Given C:  35"
## [1] "Acceptance proportion in %:  21.79"
## [1] "Given C:  36"
## [1] "Acceptance proportion in %:  21.08"
## [1] "Given C:  37"
## [1] "Acceptance proportion in %:  20.78"
## [1] "Given C:  38"
## [1] "Acceptance proportion in %:  20.26"
## [1] "Given C:  39"
## [1] "Acceptance proportion in %:  19.82"
## [1] "Given C:  40"
```

```
## [1] "Acceptance proportion in %:  19.34"
## [1] "Given C:  41"
## [1] "Acceptance proportion in %:  19.14"
```

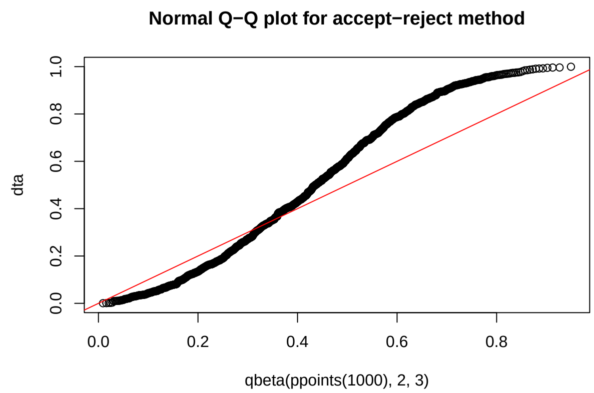
It's to be expected that the higher the c, the lower is the acceptance rate because of the relations between the target and the instrumental distribution.

Lets experiment with other values of beta and alpha.

```r
dta <- accept_or_reject(1000,2,3,1)
```

```
## [1] "Acceptance proportion in %:  72.36"
```

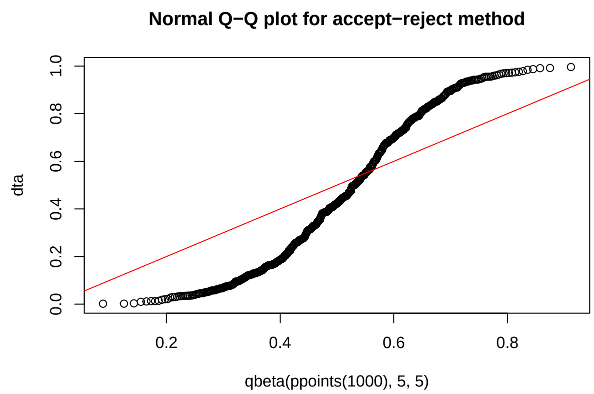
```r
qqplot(qbeta(ppoints(1000), 2, 3),dta, main="Normal Q-Q plot for accept-reject method")
abline(a=0, b=1, col='red')
```




```r
dta <- accept_or_reject(500,5,5,1)
```

```
## [1] "Acceptance proportion in %:  58.21"
```

```r
qqplot(qbeta(ppoints(1000), 5, 5),dta, main="Normal Q-Q plot for accept-reject method")
abline(a=0, b=1, col='red')
```

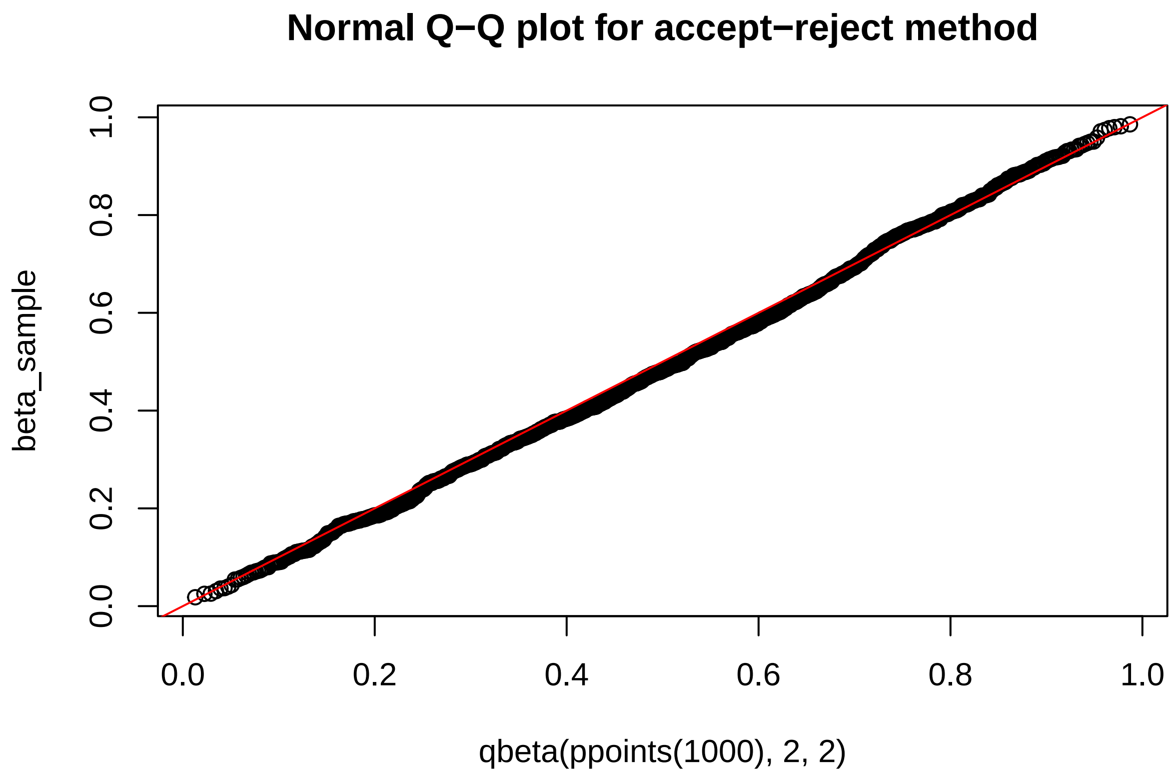


We can see that the data follows the theoretical line and also the values stay between 0 and 1. To sample from arbitrary beta distribution which adapts the constant from the given parameters,

```r
ada_accept_reject <- function(n, alpha, beta) {
  iter <- 0
  if(alpha >= 1 || beta >= 1) {
    mode <- (alpha - 1) / (alpha + beta - 2)

    # Find the peak value of the Beta
    peak <- dbeta(mode, alpha, beta)

    data <- numeric(n)
    accepted <- 0

    while(accepted < n) {
      iter <- iter+1
      x <- runif(1)
      u <- runif(1)
      # This ensures that C*g(x) is always >= f(x)
      C <- peak

      # Acceptance criterion: U <= f(x) / (C * g(x))
      # Given g(x) is Uniform[0,1], g(x) = 1, so criterion becomes U <= f(x) / C
      if(u <= dbeta(x, alpha, beta) / C) {
        accepted <- accepted + 1
        data[accepted] <- x
      }
    }
  }
  print(paste("Selected mode is ", peak))
  print(paste("Acceptance proportion in %: ", round(accepted/iter, 4)*100))
  return(data)
}


beta_sample <- ada_accept_reject(1000, 2, 2)
```

```
## [1] "Selected mode is  1.5"
## [1] "Acceptance proportion in %:  65.62"
```

```r
qqplot(qbeta(ppoints(1000), 2, 2),beta_sample, main="Normal Q-Q plot for accept-reject method")
abline(a=0, b=1, col='red')
```


Normal Q−Q plot for accept−reject method
beta_sample
qbeta(ppoints(1000), 2, 2)
0.0
0.2
0.4
0.6
0.8
1.0