Random Number Generation through CDF and acceptance-rejection sampling

Kirill Medovshchikov

2022-10-20

Contents

1 Setup

```
set.seed(12144024)
```

2 Linear Random Number Generator

We use the pseudo-random number generators due to the fact that using the actual random number generators are too cost ineffective to use on large scale. The pseudo-random number generators work in such a way that they take in the the number that is used to initialize the whole process. Such a number is called *seed*. The said seed is then recursively processed in such a way that the algorithm outputs the sequence similar to the random sample that could then be used in the statistical problems.

The following code chunk had been borrowed from the lecture slides and represents an implementation of the Linear Congruential Random Number Generator Algorithm.

```
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)
}
```

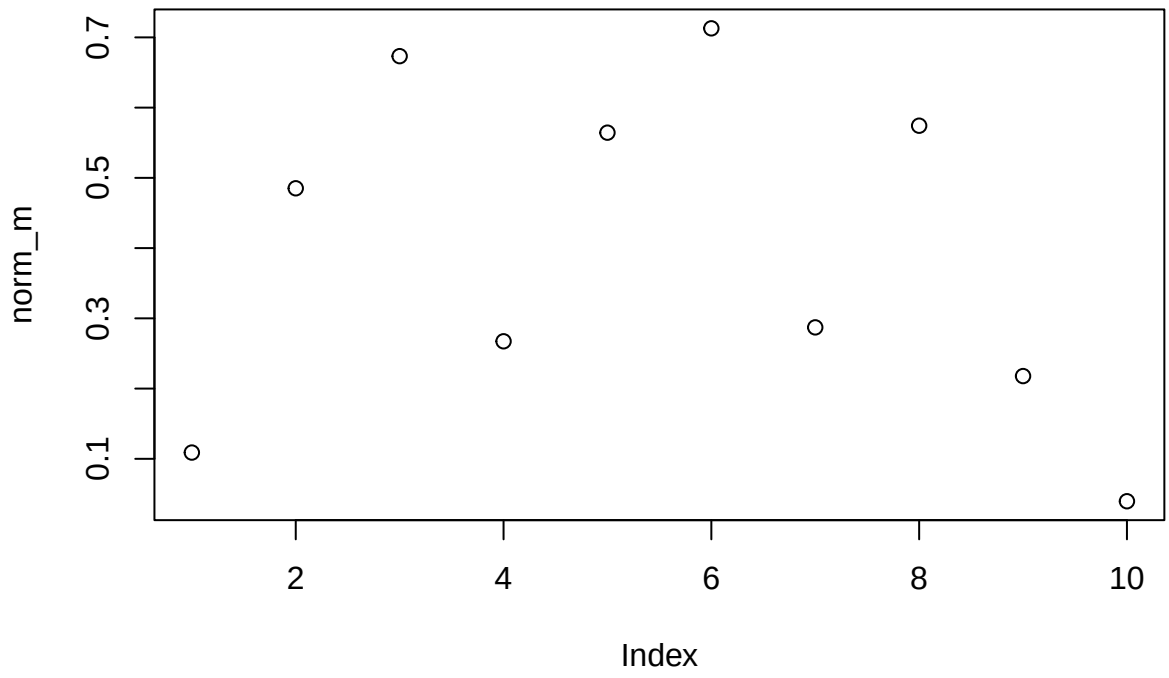
As it can be seen, the function takes in 5 variables: n, m, a, c and x0. Out of this set, the m, c and a properties are the ones that influence the output the most. The m variable is the one that governs the number of the cycles an algorithm makes, therefore the number of function's iterations depend on it. It is also advised for m to be a prime number, due to the nature of the formula above.

The following code chunk demonstrates the results of the results of the algorithm with the m being reasonably big. As it can be seen on a plot, in this case the generated numbers are somewhat evenly distributed among each other.

```
norm_m <- mc.gen(10, 101, 51, 94, 36)
print(norm_m)
```

```
##  [1] 0.10891089 0.48514851 0.67326733 0.26732673 0.56435644 0.71287129
##  [7] 0.28712871 0.57425743 0.21782178 0.03960396
```

```
plot(norm_m)
```

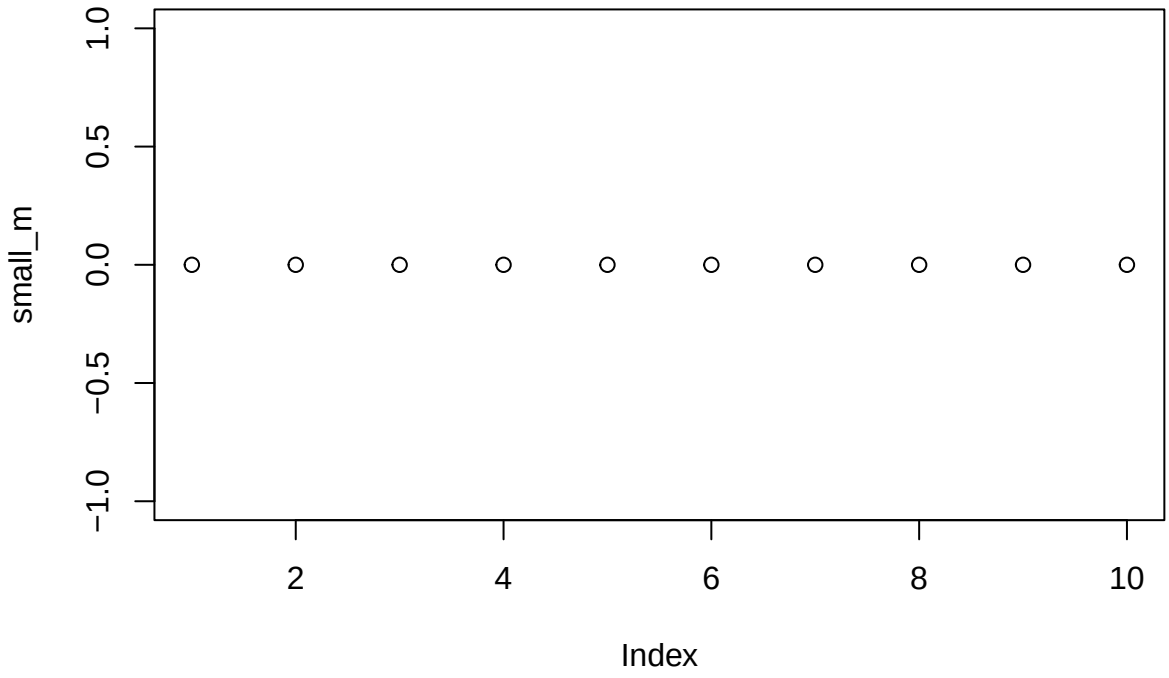


At the same time, if the m value is too small, the whole output turns into 0s and therefore is useless.

```
small_m <- mc.gen(10, 2, 51, 94, 36)
print(small_m)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
plot(small_m)
```

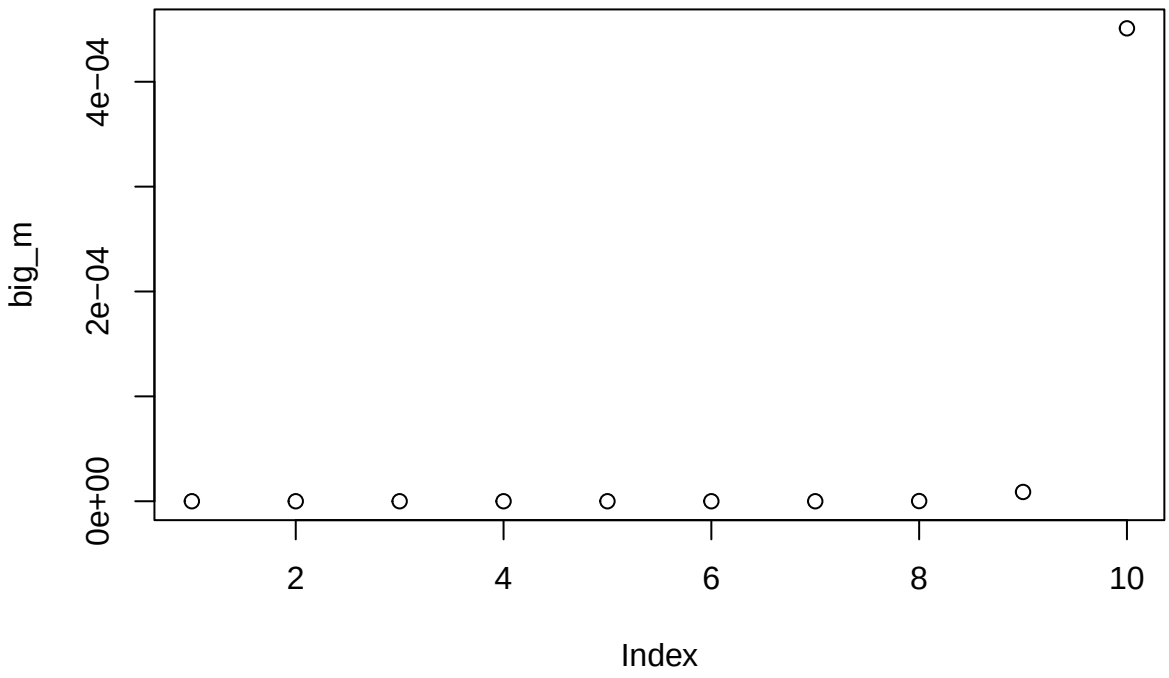


Finally, if the m variable is too big, the output still happens to be of the extremely low quality, all of the generated numbers being on the same axis with just one outlier, with almost no deviation.

```
big_m <- mc.gen(10, 999999999999999999999, 51, 94, 36)
print(big_m)
```

```
##  [1] 1.930000e-19 9.852400e-18 5.024818e-16 2.562658e-14 1.306956e-12
##  [6] 6.665474e-11 3.399392e-09 1.733690e-07 8.841818e-06 4.509327e-04
```

```
plot(big_m)
```



3 Exponential Distribution

First the exponential distribution function is translated into the R code.

```
expo_dist <- function(x, l){
  if(l > 0){
    return(1 - exp(-l*x))
  }
}
```

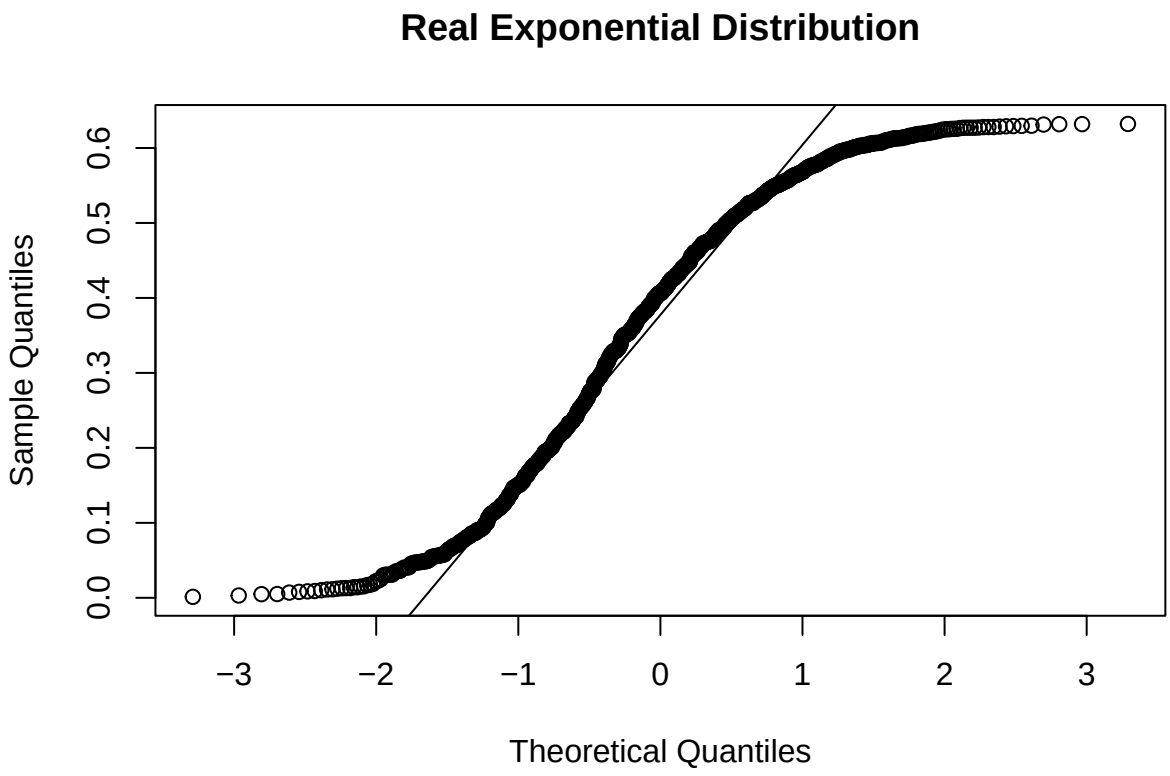
Then another function is created in order to generate the required amount of the uniform random variables and pass them into the exponential distribution function above, along side the proposed lambda. The said function is also able to generate the qqnorm plot of a real exponential distribution, if requested for comparison. For the purposes of the real exponential distribution, a function of pexp() had been chosen.

```
sample_lambda_func <- function(num, l, check=FALSE){
  samples <- runif(num, 0, 1)
  if(check == TRUE){
    real_dist <- pexp(runif(1000, 0, 1), l)
    qqnorm(real_dist, main = "Real Exponential Distribution")
    qqline(real_dist)
  }
  return(expo_dist(samples, l))
}
```

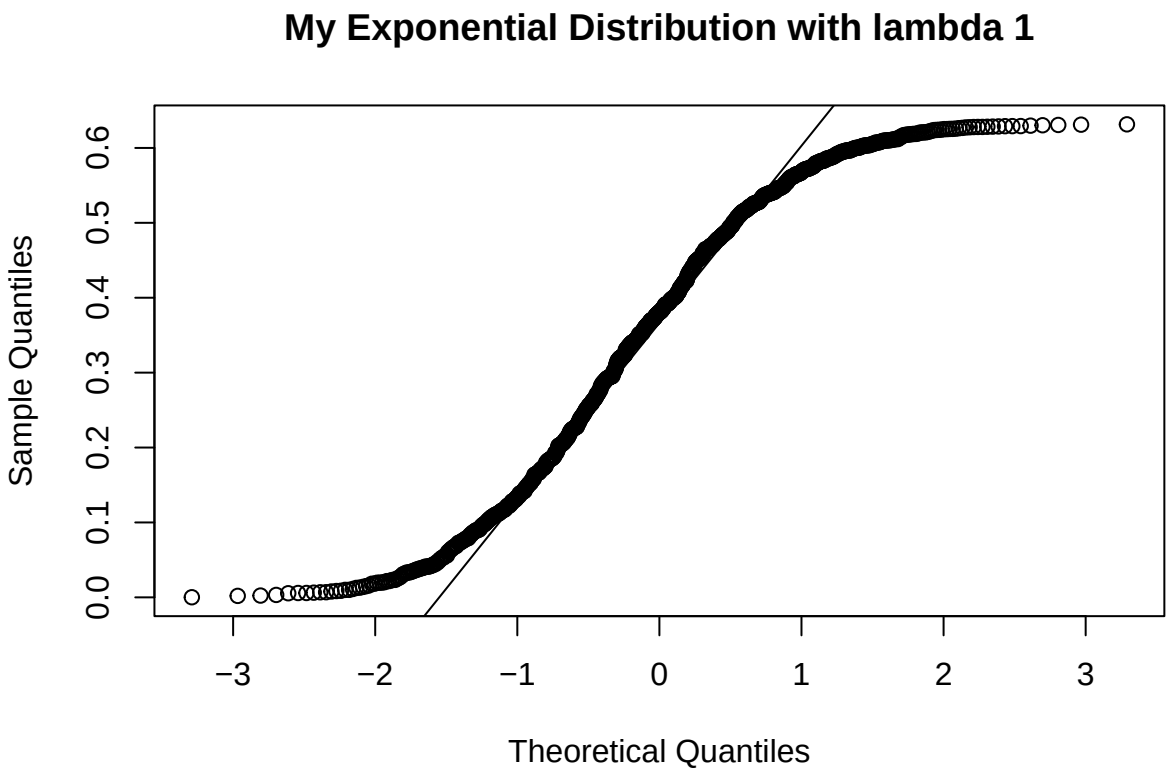
All of the tests are performed with 1000 of the random samples.

As it can be seen from the plots below, the function with lambda of 1 looks almost identically with the real exponential distribution.

```
result1 <- sample_lambda_func(1000, 1, TRUE)
```

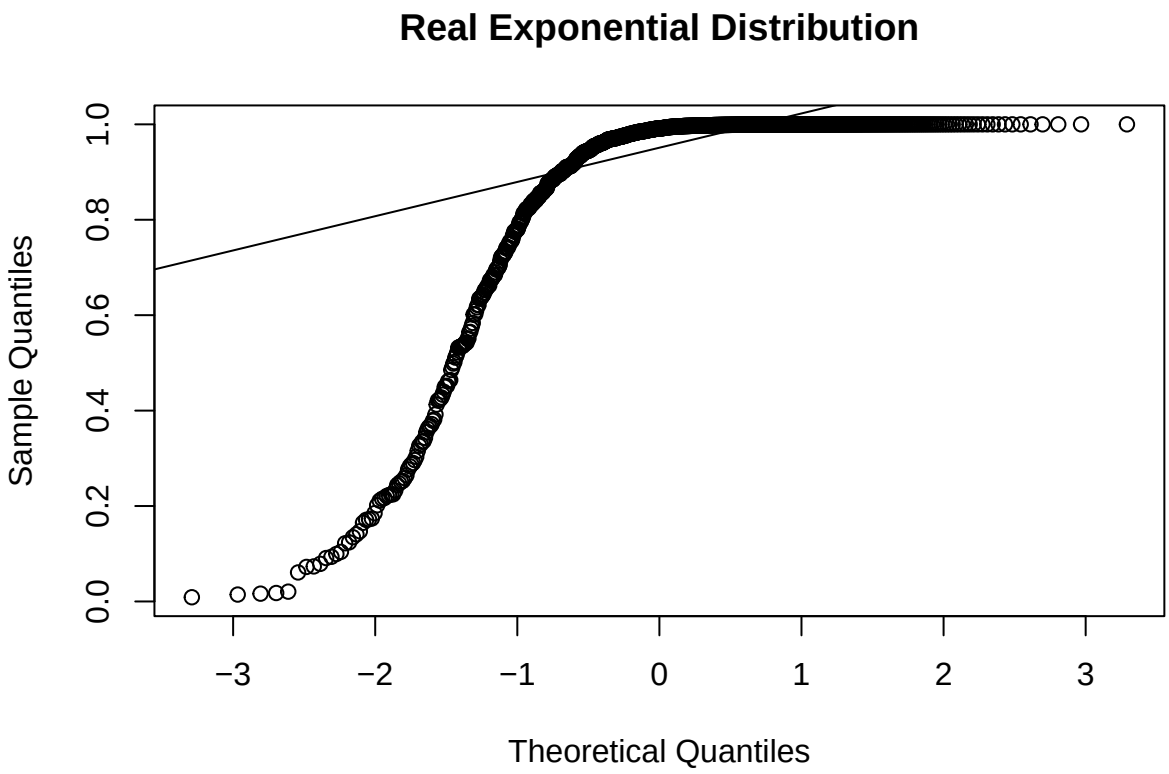



```
qqnorm(result1, main = "My Exponential Distribution with lambda 1")
qqline(result1)
```

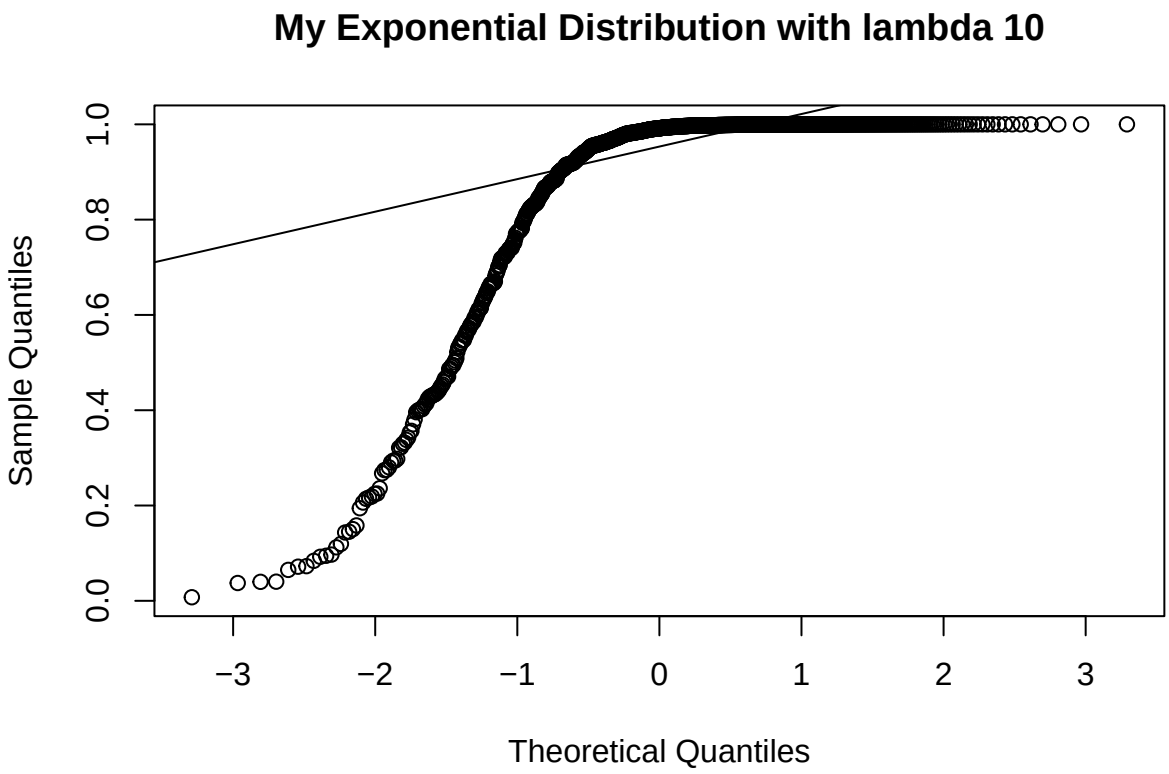


The previously described observations continue to be appearing both with lambda of 10 and 100 as well.

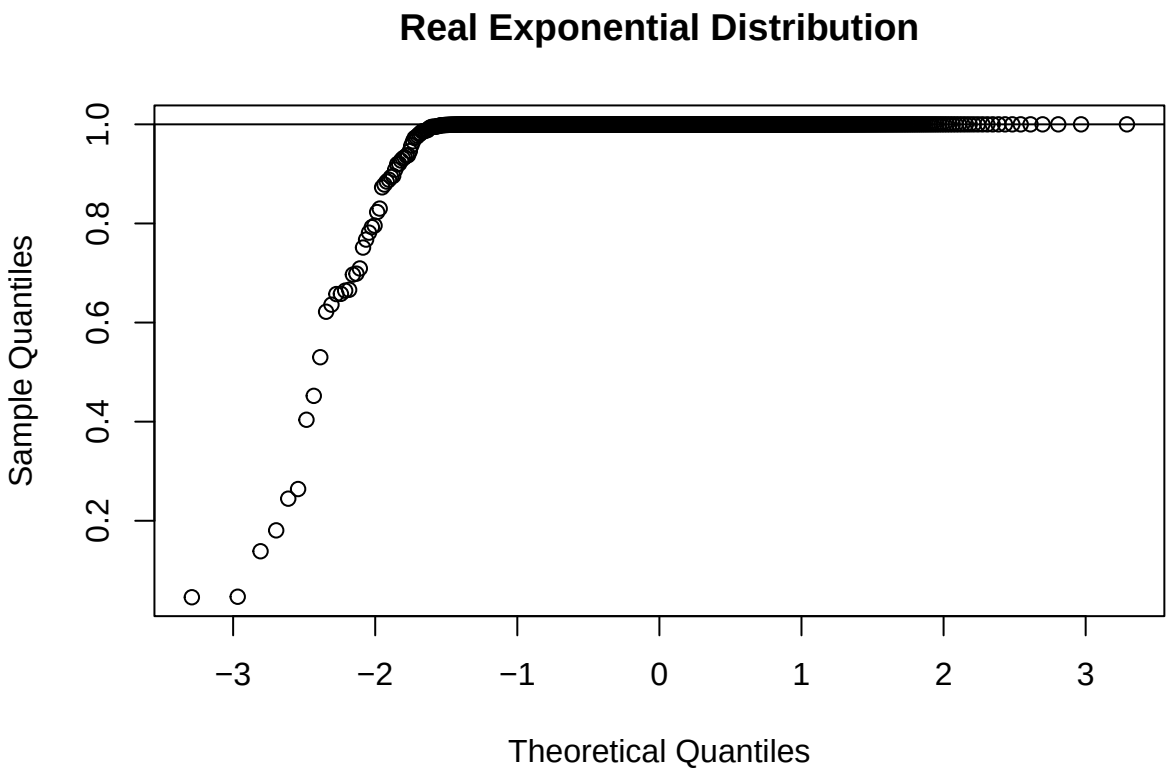
```
result2 <- sample_lambda_func(1000, 10, TRUE)
```



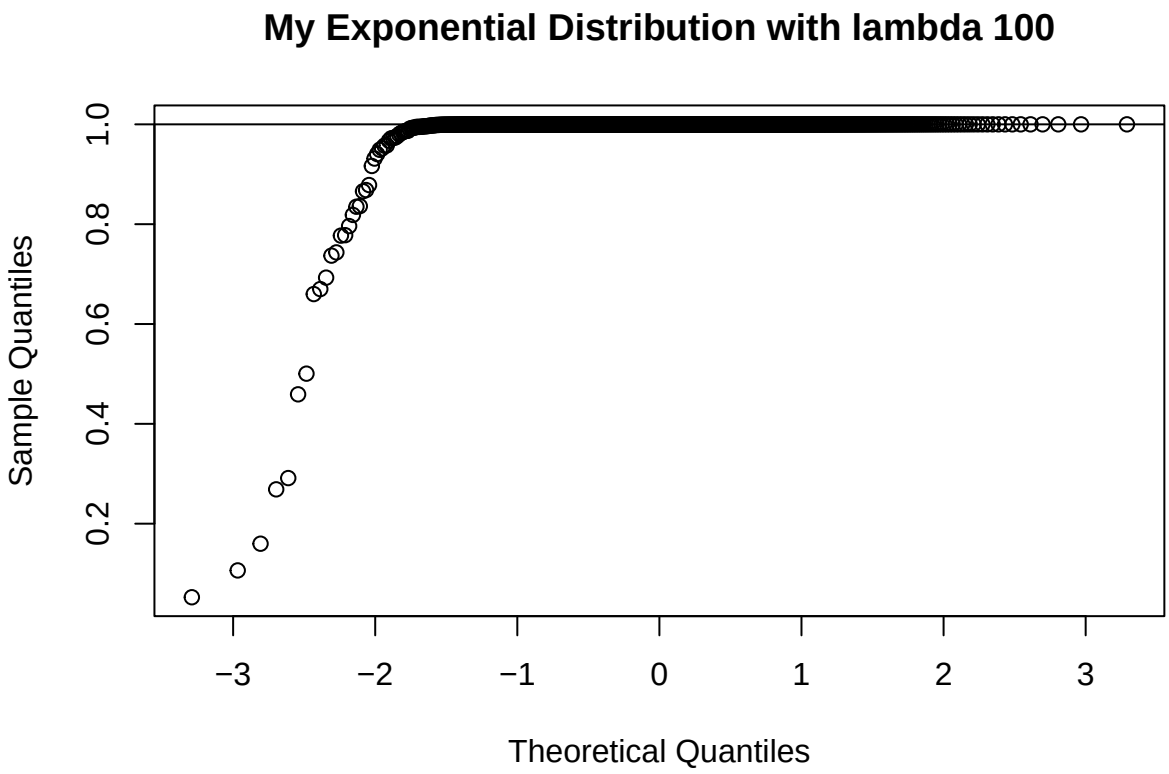

```
qqnorm(result2, main = "My Exponential Distribution with lambda 10")
qqline(result2)
```




```
result3 <- sample_lambda_func(1000, 100, TRUE)
```




```
qqnorm(result3, main = "My Exponential Distribution with lambda 100")
qqline(result3)
```



At the same time lambda influences the pattern in such a way that the curve starts to look much steeper, maxing out more of the output values. In other words, the bigger lambda is, the more values happen to be equal to 1.

4 Beta Distribution

Here the beta distribution algorithm from the lecture is converted into the R code function.

```
beta_dist <- function(x, a, b){
  return((factorial(a + b)/factorial(a) + factorial(b)) * x^(a-1) * (1 - x)^(b-1))
}
```

Then the code from the lecture is used to create the accept reject function in R, incorporating the above beta distribution function into it.

```
accept_reject <- function(a, b){
  n <- 1000; iter <- 0; accepted <- 0
  x <- numeric(n)
  bd <- beta_dist(runif(n, 0, 1), a, b)
  for(beta in bd){
    u <- beta
    iter <- iter + 1
    y <- rt(1,df=1)
    CC <- 1.6
```

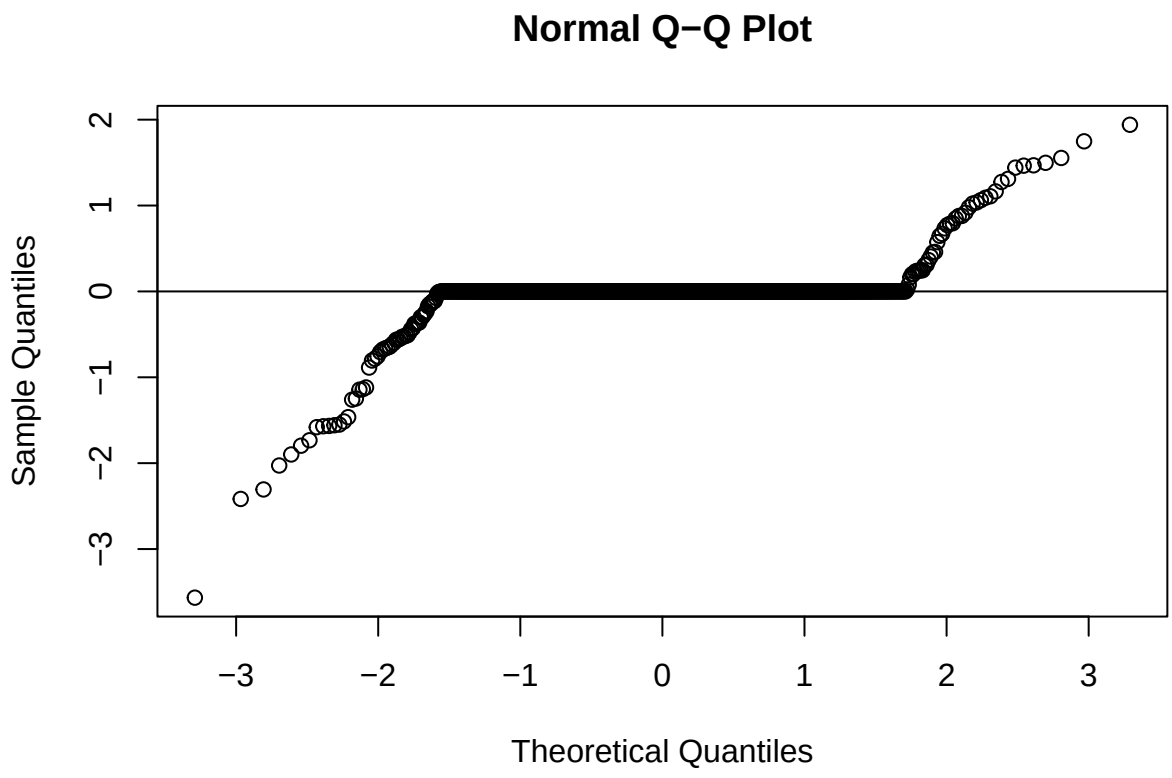
```
    if (dnorm(y)/(CC*dt(y,1)) >= u){
      accepted <- accepted+1
      x[accepted] <- y
    }
  }
  print(accepted)
  return(x)
}
```

Testing the aforementioned acceptance-rejection function with both a and b parameters being equal to 2 accordingly shows that many values are equal to 0, and are not accepted.

```
betaResult1 <- accept_reject(2,2)
```

```
## [1] 102
```

```
qqnorm(betaResult1)
qqline(betaResult1)
```

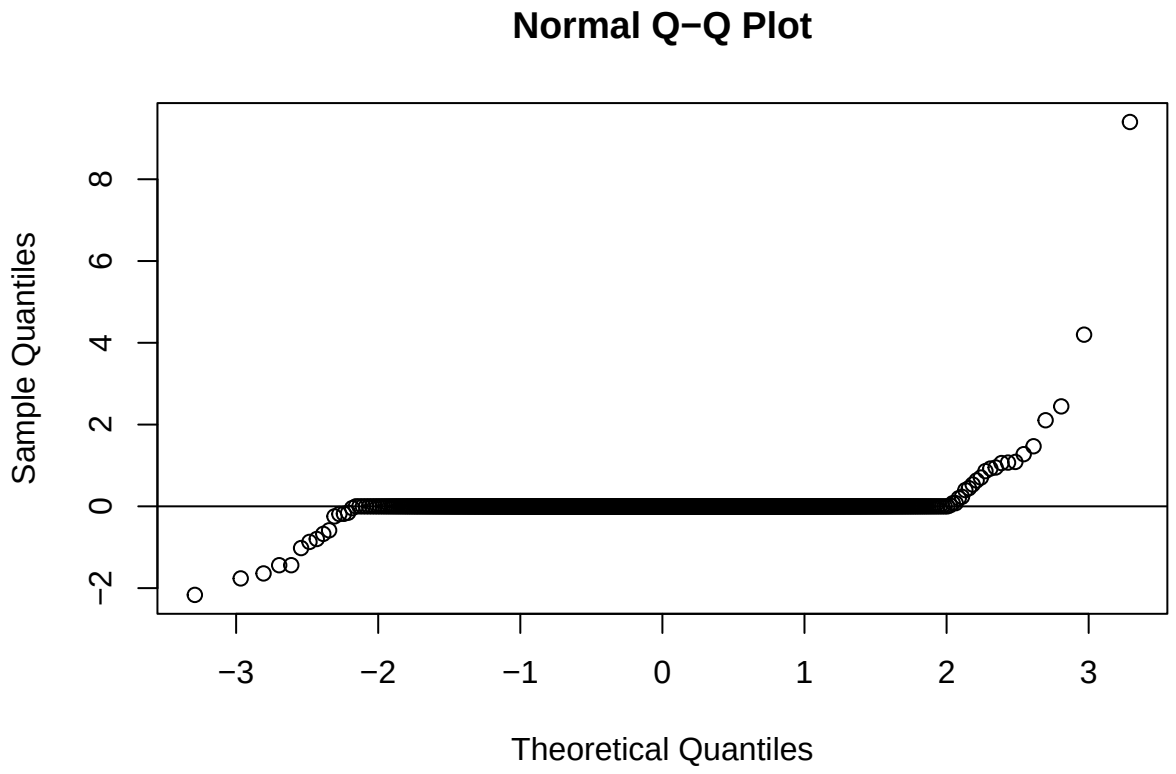


And the bigger the a and b variables become, the less numbers end up being accepted.

```
betaResult2 <- accept_reject(10,25)
```

```
## [1] 37
```

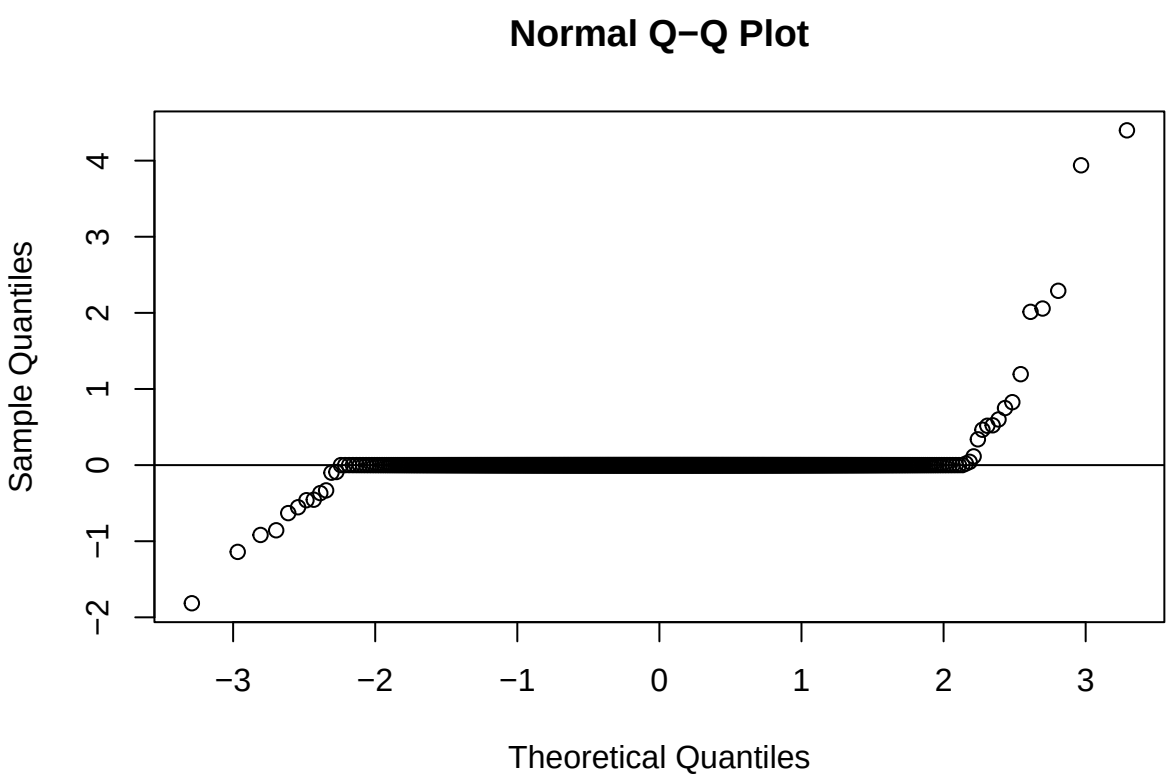
```
qqnorm(betaResult2)
qqline(betaResult2)
```




```
betaResult3 <- accept_reject(50,50)
```

```
## [1] 28
```

```
qqnorm(betaResult3)
qqline(betaResult3)
```


Normal Q−Q Plot
Sample Quantiles
Theoretical Quantiles
−2
−1
0
1
2
3
4
−3
−2
−1
0
1
2
3