Report for Computer Lab 3 in Computational Statistics

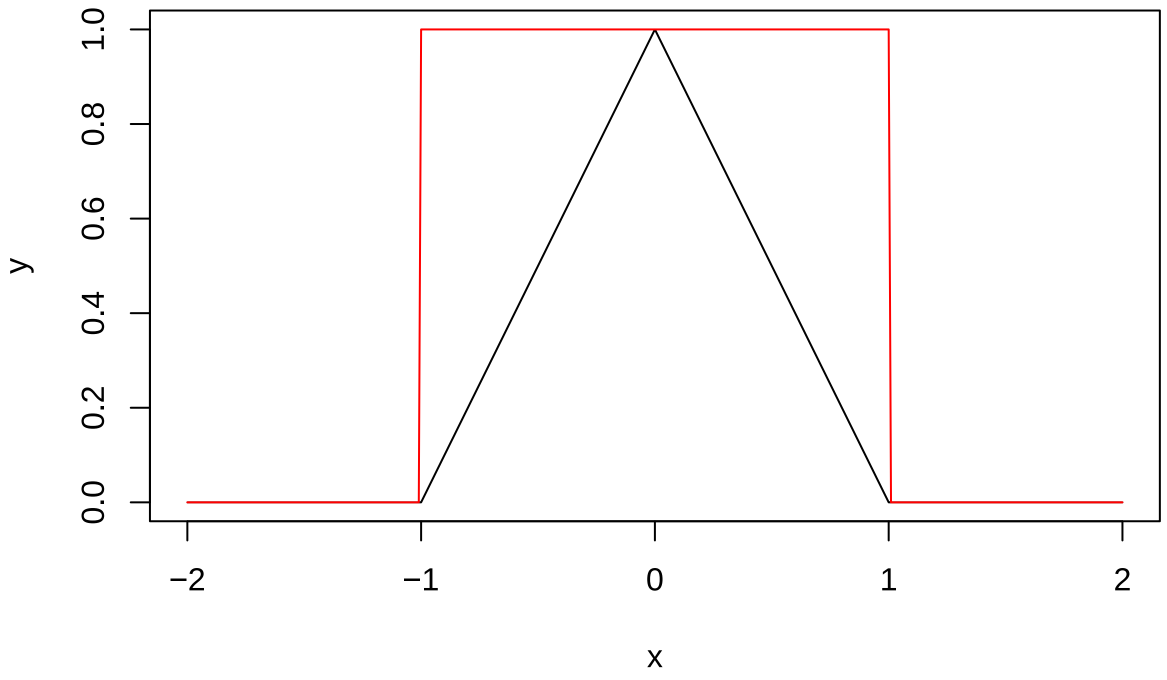
Question 1: Sampling algorithms for a triangle distribution

$$f(x) = \begin{cases} 0, & \text{if } x < -1 \text{ or } x > 1 \\ x+1, & \text{if } -1 <= x <= 1 \\ 1-x, & \text{if } 0 <= x <= 1 \end{cases}$$

a. Choose an appropriate and simple envelope e(x) for density and program a random generator for X using rejection sampling. Generate 10000 random variables and plot a histogram.

As envelope we choose a constant function on the intervall [-1,1].

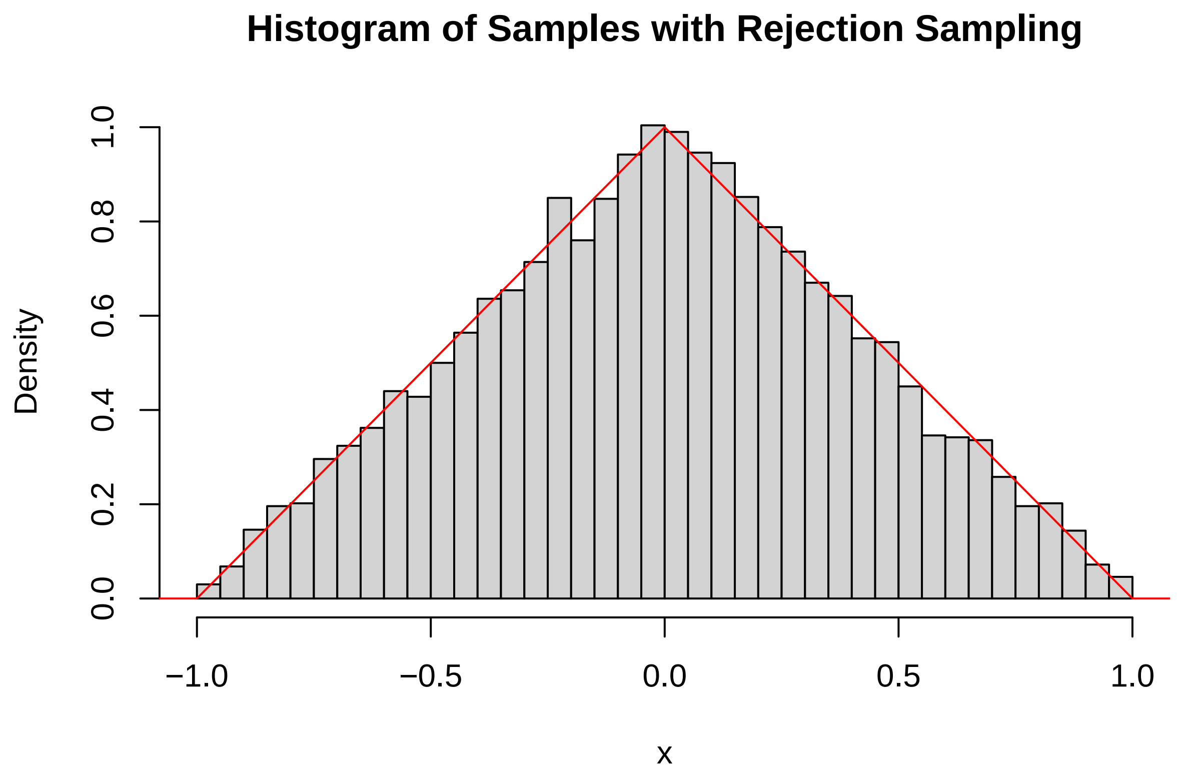
$$e(x) = \begin{cases} 1, & \text{if } x >= -1 \text{ or } x <= 1 \\ 0, & \text{otherwise} \end{cases}$$



The following function was implemented according to the algorithm in the book (Givens). It allows the constant α to be <= 1. We chose $\alpha = 1$ for simplicity, therefore are g(x) and e(x) equal. It still fulfills the condition $g(x)/\alpha >= f(x)$.

```r
rejection_sampling <- function(n) {
  samples <- numeric(n)
  c <- 1
  while(c < n) {
    sample_Y <- runif(1, -1 ,1)
    sample_U <- runif(1, 0, 1)

    # check for U <= f(Y) / e(Y), e(Y) = 1
    if(sample_U <= f(sample_Y)){
      samples[c] <- sample_Y
      c <- c + 1
    }
  }
  return(samples)
}
```

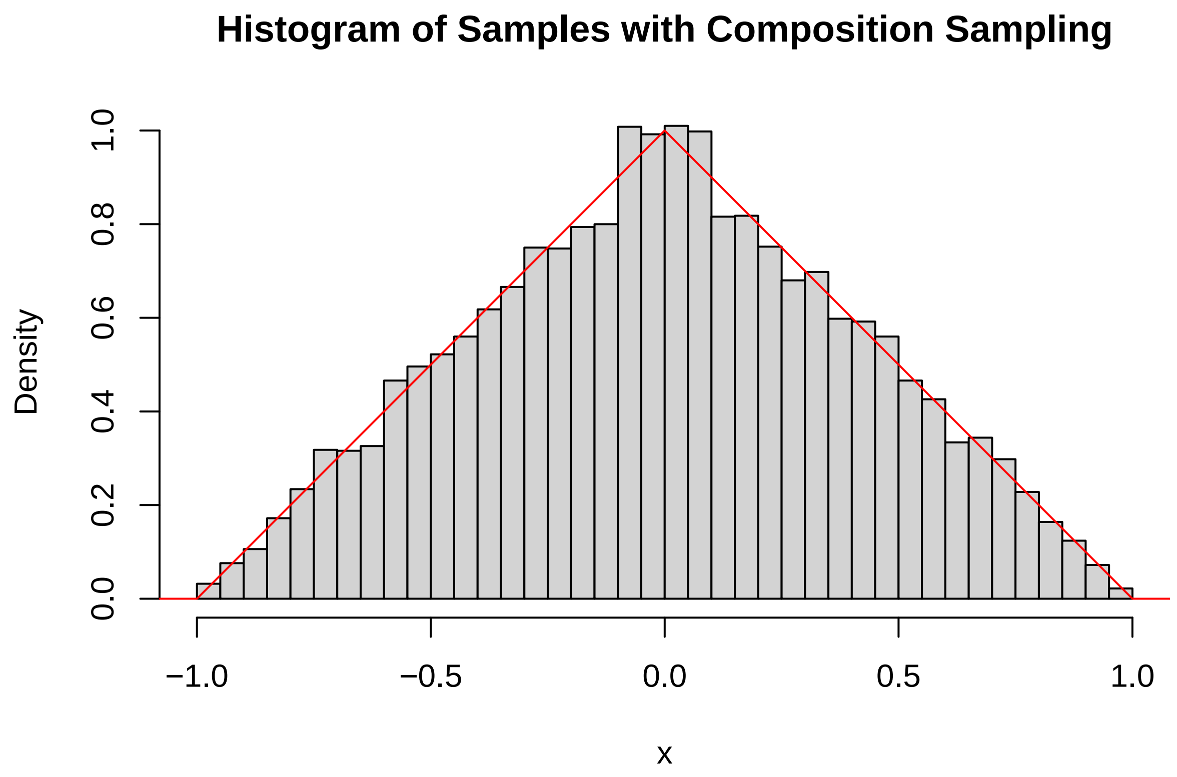
The plot of the histogram of a sample of 10,000 random variables using rejection sampling shows a triangle similar to the original function:



b. In Lecture 3, another triangle distribution was generated using the inverse cumulative distribution function method; see pages 9-10 of the lecture notes. Let Y be a random variable following this distribution. A random variable -Y has a triangle distribution in the interval [-1, 0]. Program a random generator for X using composition sampling based on Y and -Y . You can use the code from the lecture to generate Y . Generate 10000 random variables and plot a histogram.

In this approach two random variables are used to sample from f(x). Sampling from the uniform distribution decides for every sample from which distribution it is drawn.

```
composition_sampling <- function(n) {
  u <- runif(n)
  criterion <- runif(n)

  samples <- ifelse(criterion < 0.5, 1-sqrt(1-u), -(1-sqrt(1-u)))

  return(samples)
}
```

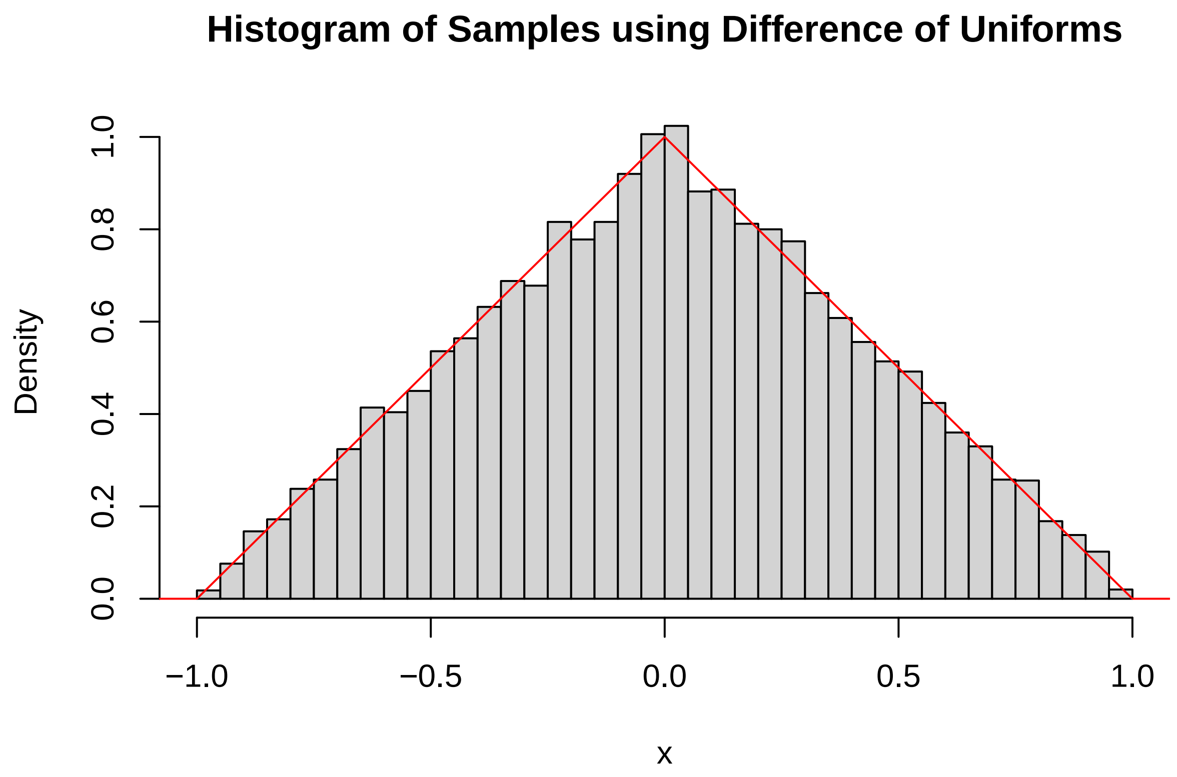


The resulting histogram shows a similar shape as the triangle from the original distribution.

c. Sums or differences of two independent uniformly distributed variables can also have some triangle distribution. When U1, U2 are two independent Unif [0, 1]-distributed random variables, U1 - U2 has the same distribution as X. Use this result to program a generator for X. Generate 10000 random variables and plot a histogram.

This function has a simple implementation, as it only returns the difference of two uniform distributions.

```
diff_sampling <- function(n){
  u1 <- runif(n)
  u2 <- runif(n)

  samples <- u1 - u2

  return(samples)
}
```



Using the function to sample from X results in a histogram with a similar shape to the original distribution.

d. Consider the amount of programming for the three methods and compare the number of random value generation and other time consuming operations needed by the methods to judge expected running time. Which of the three methods do you prefer if you had to generate samples of X? Use data from one method to determine the variance of X.

The programming effort is the lowest for diff_sampling(), as it simply returns the difference of two vectors sampled from two independent uniform distributions. The complexity that composition_sampling() adds is only the ifelse-clause to decide from which distribution to sample. Rejection sampling requires the most effort, as a while-loop is required. It cannot be solved with a vectorized approach, as it is not known upfront how many samples will be rejected. Hence the lengths of vectors cannot be predefined.

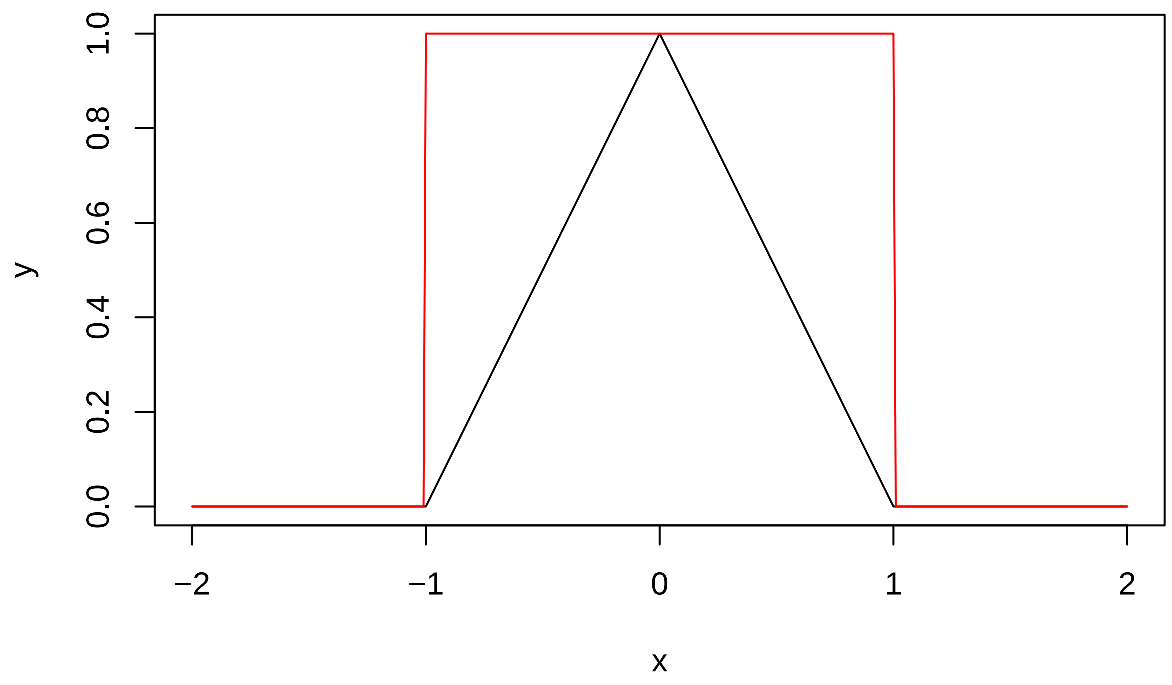
The rejection of samples also makes this method the slowest. We generate a random value twice for each sample in all three algorithms but for rejection sampling we do not know how often we have to run the loop until a sample is accepted. Additionally we have to use a while loop, which is slower than working on vectors. Composition sampling and sampling from the difference of two uniform distributions have the same amount of random value generations but calculating the square root in composition sampling is a bit more costly. Because of the lower computational cost and the simplicity of the code we would choose diff_sampling() to generate samples of X.

Using the latter function to get 10,000 samples of X gives the following variance:

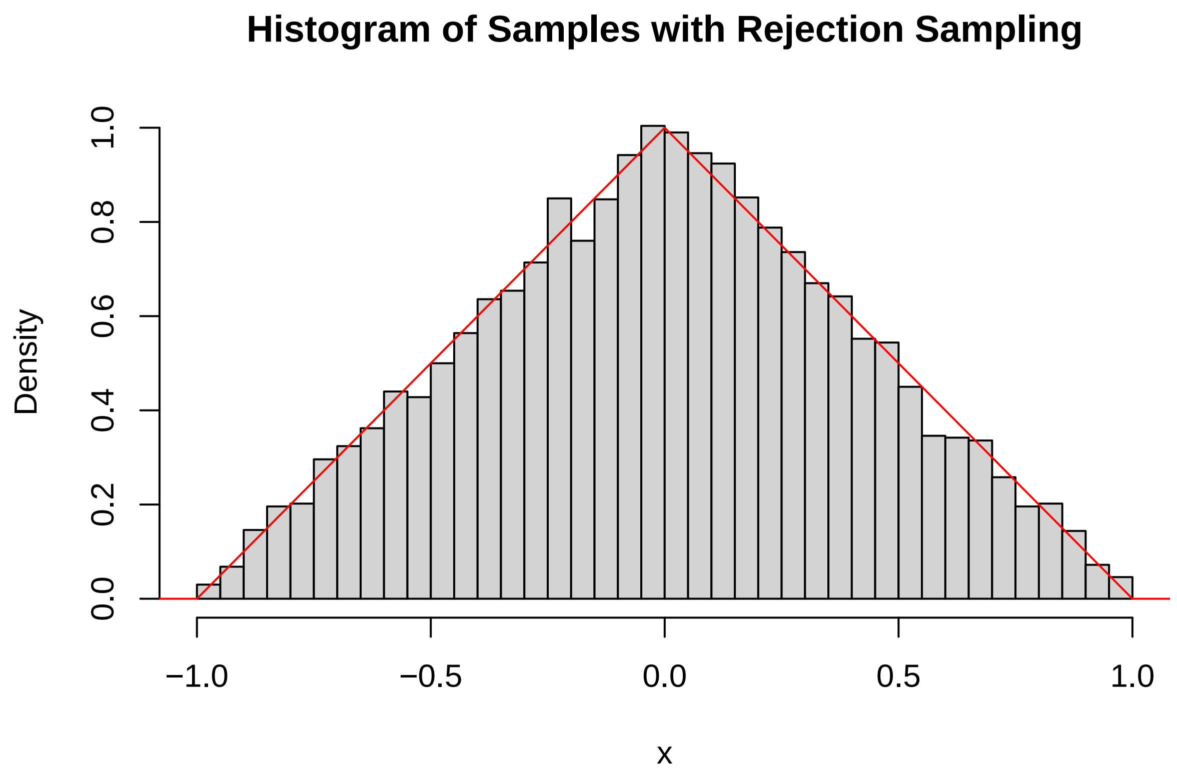
```
## [1] 0.1685393
```

Appendix

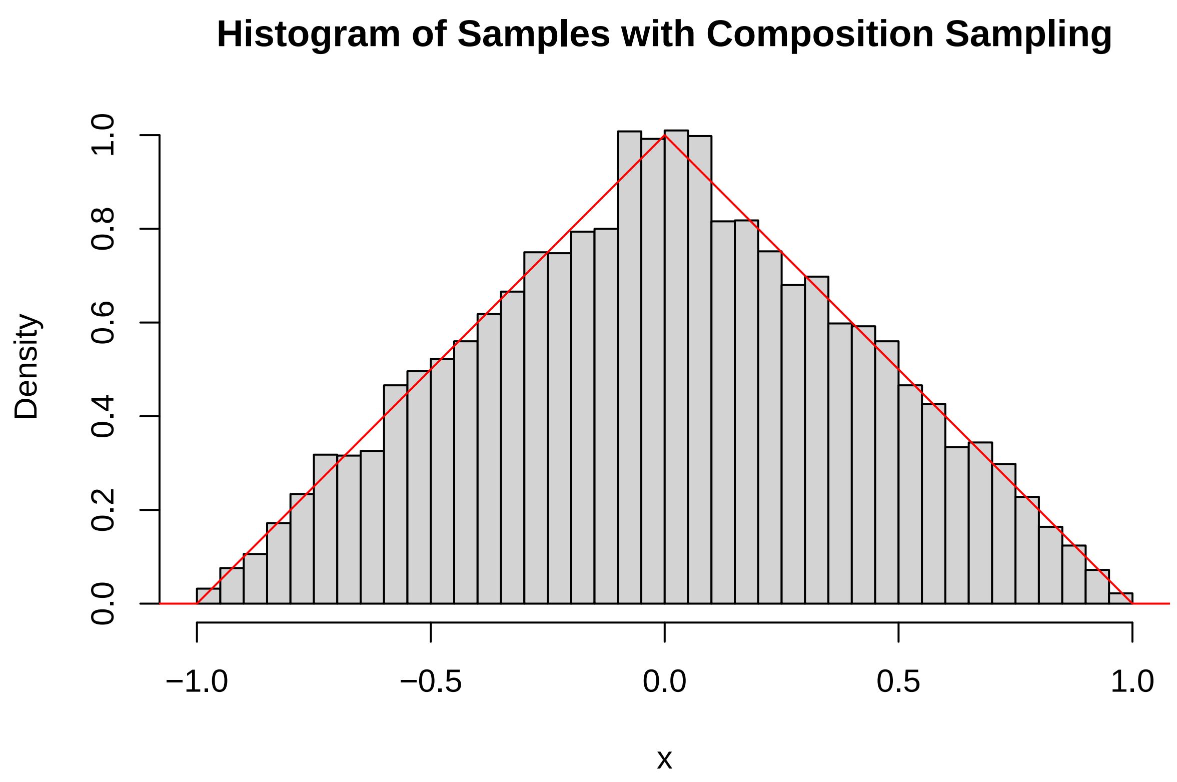
```r
# Question 1
f <- function(x) {
  if (x < -1 | x > 1){
    return(0)
  }
  else if (-1 <= x & x <= 0) {
    return(x+1)
  }
  else {
    return (1-x)
  }
}

envelope <- function(x) {
  if (x < -1 || x > 1){
    return(0)
  }
  else{
    return(1)
  }
}


x <- seq(-2,2,0.01)
y <- sapply(x, f)
plot(x, y, type="l")
y_env <-  sapply(x, envelope)
lines(x, y_env, col="red")
```



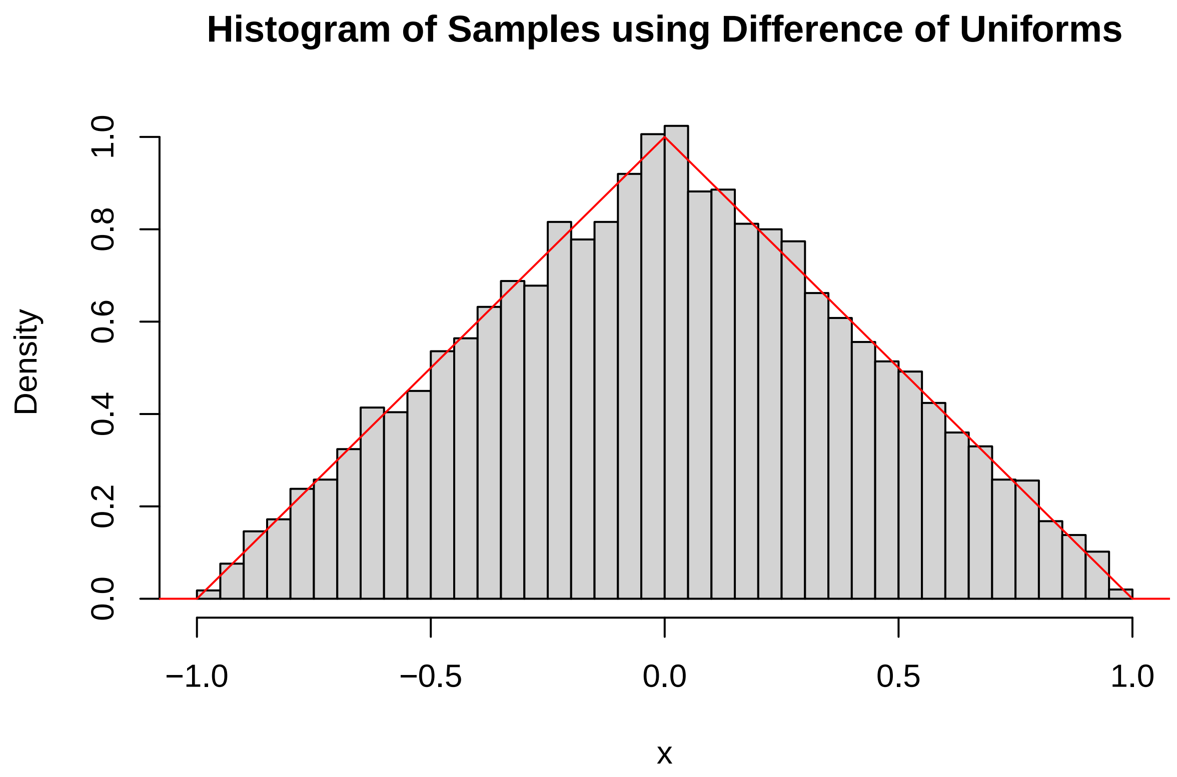
```r
rejection_sampling <- function(n) {
  samples <- numeric(n)
  c <- 1
  while(c < n) {
    sample_Y <- runif(1, -1 ,1)
    sample_U <- runif(1, 0, 1)

    # check for U <= f(Y) / e(Y), e(Y) = 1
    if(sample_U <= f(sample_Y)){
      samples[c] <- sample_Y
      c <- c + 1
    }
  }
  return(samples)
}

set.seed(42)
samples <- rejection_sampling(10000)
hist(samples, breaks=50,probability = TRUE, main = "Histogram of Samples with Rejection Sampling", xlab
lines(x, y, col="red")
```



```r
composition_sampling <- function(n) {
  u <- runif(n)
  criterion <- runif(n)

  samples <- ifelse(criterion < 0.5, 1-sqrt(1-u), -(1-sqrt(1-u)))

  return(samples)
}

set.seed(42)
samples <- composition_sampling(10000)
hist(samples, breaks=50,probability = TRUE, main = "Histogram of Samples with Composition Sampling", xla
lines(x, y, col="red")
```



```r
diff_sampling <- function(n){
  u1 <- runif(n)
  u2 <- runif(n)

  samples <- u1 - u2

  return(samples)
}

set.seed(42)
samples <- diff_sampling(10000)
hist(samples, breaks=50,probability = TRUE, main = "Histogram of Samples using Difference of Uniforms",
lines(x, y, col="red")
```



```r
set.seed(42)
samples <- diff_sampling(10000)
var(samples)
```

```
## [1] 0.1685393
```

Question 2: Bivariate normal and normal mixture distribtution

a) Generate a two dimensional random vector $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ that has a two-dimensional normal distribution with mean vector $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and covariance matrix $\begin{pmatrix} 0.6 & 0 \\ 0 & 0.6 \end{pmatrix}$ using the Box-Muller method and using only *runif* as random number generator.

Since we are asked to generate a two random vector from a two dimensiona normal distribution with the given mean vector and covariance matrix, we simply sample from the uniform distribution and perform the Box-Muller transformation. To achieve the desired covariance matrix, we scale the random vector with $\sqrt{(}0.6)$.

```r
box_muller <- function(input_vec){
  u_1 <- input_vec[1]
  u_2 <- input_vec[2]
  z_1 <- sqrt(-2*log(u_1))*cos(2*pi*u_2)
  z_2 <- sqrt(-2*log(u_1))*sin(2*pi*u_2)
  return(c(z_1, z_2))
}

univariate_vector <- function(){
  return(c(runif(1), runif(1)))
}

start_time <- proc.time()
for(i in (1:10000000)){
  uniform_vec <- univariate_vector()
  normal_vec <- box_muller(uniform_vec) * sqrt(0.6)
}
end_time <- proc.time()

print(end_time - start_time)
```

```
##        User      System verstrichen
##       22.67        0.18       37.67
```

b) Generate again 10000000 random vectors with this distribution, but now using either the one-dimensional function *rnorm* or the package *mvtnorm*. Explain why you have chosen to generate it the way you did. Measure the computation time and compare it to your result from a)

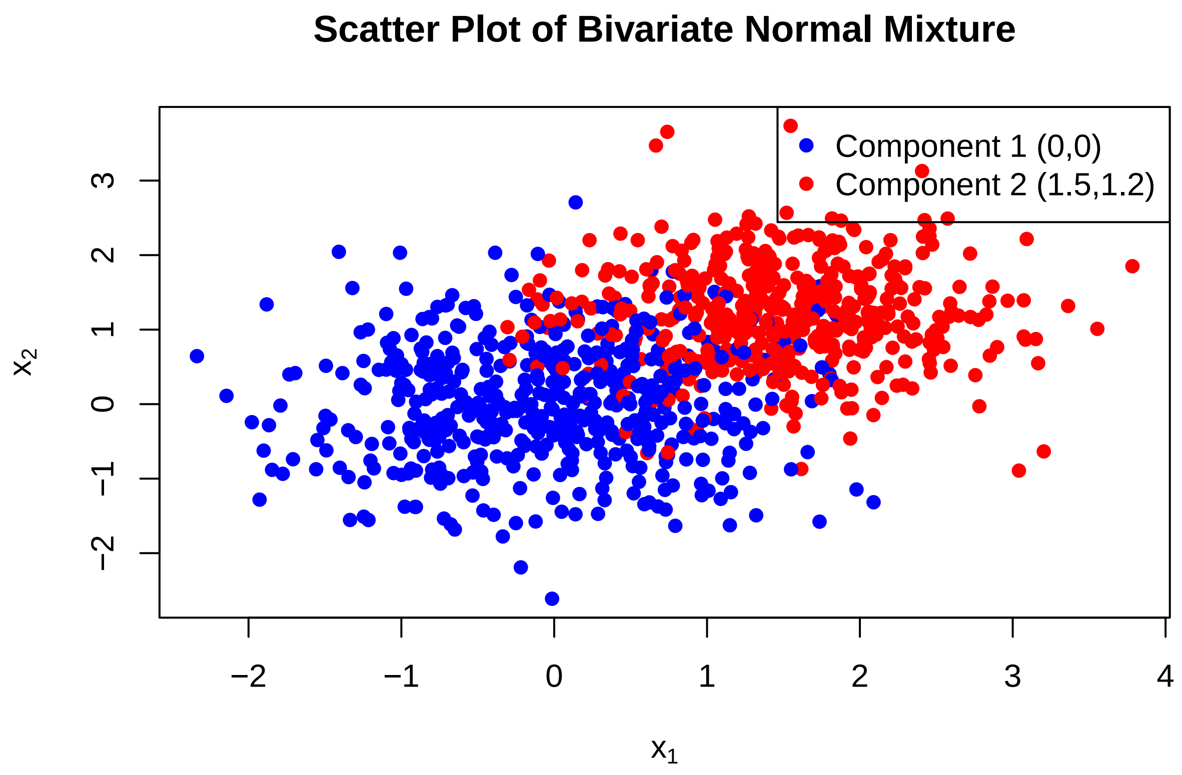
We decided to use *rnorm* as the covariance matrix is diagonal, hence the different random variables are uncorrelated and thus we do not need any dependency structure which *mvtnorm* utilizes.

```r
start_time <- proc.time()

n <- 10000000  # Number of vectors

# Generate independent normal samples with mean 0 and variance 0.6
x_1 <- rnorm(n, mean = 0, sd = sqrt(0.6))
x_2 <- rnorm(n, mean = 0, sd = sqrt(0.6))

end_time <- proc.time()

print(end_time - start_time)
```

```
##        User      System verstrichen
##        0.81        0.34        1.62
```

As we can observe, the using *rnorm* instead of *runif* and then transforming using Box-Muller is many magnitudes faster. This is because rnorm utilizes vectorization, which is very efficient in R, especially when compared to the loop approach in a). c) Now, we consider the density of the bivariate normal mixture in Lecture 2. There, observations follow with 50 probability one of the following two distributions: - normal distribution with mean vector $\begin{pmatrix}0\\0\end{pmatrix}$ and covariance matrix $\begin{pmatrix}0.6 & 0\\0 & 0.6\end{pmatrix}$ - normal distribution with mean vector $\begin{pmatrix}1.5\\1.2\end{pmatrix}$ and covariance matrix $\begin{pmatrix}0.5 & 0\\0 & 0.5\end{pmatrix}$

```r
set.seed(42)
n <- 1000

# Define mean
mu1 <- c(0, 0)
mu2 <- c(1.5, 1.2)

# Define covariance matrices
sigma1 <- matrix(c(0.6, 0, 0, 0.6), nrow = 2)
sigma2 <- matrix(c(0.5, 0, 0, 0.5), nrow = 2)

# Randomly assign samples to components
component <- rbinom(n, size = 1, prob = 0.5)

# Generate samples based on the assigned component
samples <- matrix(0, nrow = n, ncol = 2)
for (i in 1:n) {
  if (component[i] == 0) {
    samples[i, 1] <- rnorm(1, mean = mu1[1], sd = sqrt(sigma1[1,1]))
    samples[i, 2] <- rnorm(1, mean = mu1[2], sd = sqrt(sigma1[2,2]))
  } else {
    samples[i, 1] <- rnorm(1, mean = mu2[1], sd = sqrt(sigma2[1,1]))
    samples[i, 2] <- rnorm(1, mean = mu2[2], sd = sqrt(sigma2[2,2]))
  }
}

# Plot the generated samples
plot(samples, col = ifelse(component == 0, "blue", "red"), pch = 16,
     main = "Scatter Plot of Bivariate Normal Mixture",
     xlab = expression(x[1]), ylab = expression(x[2]))
legend("topright", legend = c("Component 1 (0,0)", "Component 2 (1.5,1.2)"),
       col = c("blue", "red"), pch = 16)
```



The plot looks as one would expect, the blue dots cluster around the coordinates $(0, 0)$ which is determined by the mean-vector. The same is true for the red data points, which cluster around 1.5 for x_1 and 1.2 for x_2. According to the empirical rule 95 of random samples from a normal distribution fall within 2 standard deviations of the mean. Going off visuals alone, this seems to be true for both distributions as well. Lastly, while it might appear that there is an imbalance between the blue dots and red dots, this is misleading as the blue distribution has a higher standard deviation causing the blue samples to be more spread out, having less overlap and thus appearing to be more frequent than the red samples.