Statistics Software Lab Report - 3

Name of the Student: Shatansh Patnaik
Roll No: 20MA20067

IIT Kharagpur
Statistics Software Lab

Rejection Method for Generation of Random Variables

The rejection method, also known as the acceptance-rejection method, is a powerful technique in the field of stochastic simulation for generating random variables from a specified probability distribution. This method is particularly useful when it is challenging to directly sample from a desired distribution, and an alternative, simpler distribution is available.

Overview

The rejection method involves two main distributions: the target distribution (which we want to sample from) and a proposal distribution (which is easier to sample from but may not match the target distribution). The goal is to generate random variables that follow the target distribution by accepting or rejecting samples from the proposal distribution.

Algorithm

The rejection method algorithm can be described as follows:

Algorithm 1 Rejection Method

Input: Target distribution $f(x)$, proposal distribution $g(x)$
Output: Random variable X following the target distribution
while True **do**
    Generate a random sample x from the proposal distribution $g(x)$
    Generate a uniform random number u between 0 and 1
    Calculate acceptance probability: $A = \frac{f(x)}{M \cdot g(x)}$ ▷ M is a constant such that $f(x) \leq M \cdot g(x)$
    if $u \leq A$ **then**
        Accept the sample: $X = x$
        Break out of the loop
    end if
end while

The rejection method involves repeatedly generating samples from the proposal distribution and accepting or rejecting them based on a calculated acceptance probability. The loop continues until a sample is accepted, providing a random variable that follows the target distribution.

We will be generating random variables for Standard Normal Distribution using Rejection Method, out of which two of the distributions would be continous distributions and one would be a discrete distribution.

Beta Distribution

The probability density function (PDF) of a Beta-distributed random variable X is given by:

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

Here, $B(\alpha, \beta)$ is the beta function, a normalizing constant that ensures the total area under the probability density function is equal to 1. The beta function is defined as:

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\, dt$$

We shall generate random samples having beta density using the algorithm given below $f(x) = 2x(1-x)^3$ for $0 < x < 1$:

Algorithm 2 Generation of Beta-Distributed Random Variable

1: **Step 1:** Generate random numbers U_1 and U_2 following Uniform Distribution.
2: **Step 2:** If $U_2 \leq \frac{256}{27} U_1(1-U_1)^3$, stop and set $X = U_1$. Otherwise, return to **Step 1**.

This algorithm follows a rejection-based approach, where random numbers are generated from the uniform distribution, and a condition is checked for acceptance or rejection based on the Beta density function. If the condition is met, the generated value U_1 is the desired Beta-distributed random variable X.

Same code can be illustrated for 1000 random samples, we will be using the builtin function *runif()* to obtain the Uniform Distribution, following which we will use the above algorithm to generate the random sample following Beta Distribution, the code in R is provided below for the same:

```r
# Generation of a single random variable
generate_beta_random_variables <- function() {
  while(1){
    u1 <- runif(1,0,1)
    u2 <- runif(1,0,1)

    if(u2 <= (256/27)*u1*((1-u1)^3)){
      return(u1)
    }
  }
}

# Generation of 1000 random samples
generate_b_rv <- function (n) {
  X <- numeric(0)
  for (i in 1:n){
    X <- append(X, generate_beta_random_variables())
  }
  return(X)
}

beta <- function(x) {
  return (20*x*((1-x)^3))
}
```

We can illustrate the following in a histogram with X-axis representing the Values in the Distribution that are generated and y-axis represents the Frequencies of those values in the distribution.

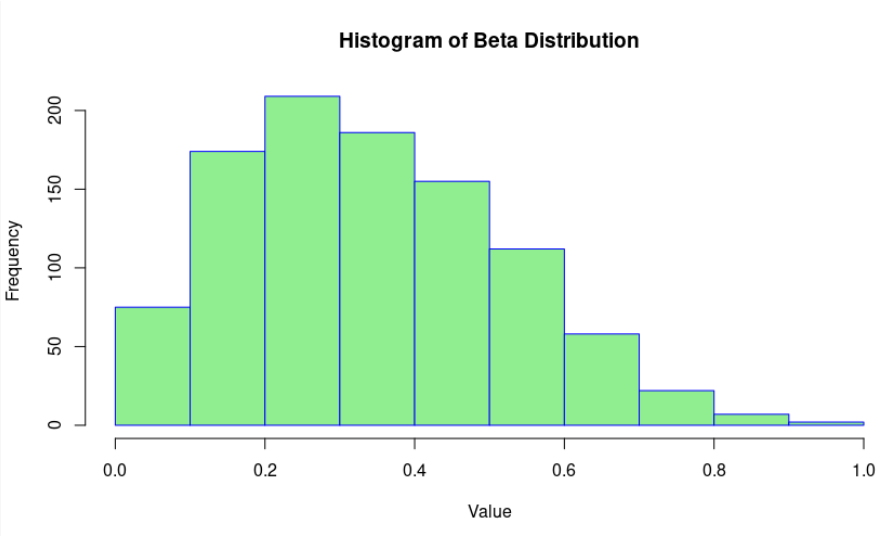

Figure 1: The given figure is a histogram plot of Beta Distribution

Now we shall implement the Chi Square Goodness of Fit test to check if the required distribution is indeed Beta Distribution. The following code illustrates the usage of Chi Square Goodness of Fit:

```
do_chi_sq_test <- function(O, E, s){
  W <- sum(((O-E)^2)/E)
  critical_value <- qchisq(0.95, 9)

  sprintf("The following is the result for %s DIstribution", s)
  sprintf("W: %f", W)
  sprintf("Critical Value: %f", critical_value)

  if(W > critical_value){
    sprintf("The given distribution doesnt follow %s Distribution", s)
  } else {
    sprintf("The given distribution follows %s Distribution", s)
  }
}

do_the_tests <- function(X, O, breaks, s) {
  E <- numeric(0)
  for(i in 1:10){
    y <- simpsons_rule_integral(beta, breaks[i], breaks[i+1], 1000)
    E <- append(E, 1000*y)
  }
  do_chi_sq_test(O, E, s)
}

# We are assuming the number of buckets is 10
X <- generate_b_rv(1000)
breaks <- seq(0, 1, 0.1)
O <- hist(X, breaks=breaks, main = "Histogram of Beta Distribution", xlab =
```

```
    "Value", ylab = "Frequency", col = "lightgreen", border = "blue")$count
do_the_tests(X, 0, breaks, "Beta")
```

Standard Normal Distribution

The Standard Normal Distribution, often denoted as $Z \sim \mathcal{N}(0, 1)$, is a special case of the normal distribution with a mean (μ) of 0 and a standard deviation (σ) of 1.

Probability Density Function (PDF)

The probability density function (PDF) of the standard normal distribution is given by:

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}, \quad -\infty < z < \infty$$

This bell-shaped curve is symmetric around the mean ($z = 0$).

Cumulative Distribution Function (CDF)

The cumulative distribution function (CDF) of the standard normal distribution is expressed using the error function (erf):

$$\Phi(z) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right]$$

Here, $\Phi(z)$ represents the probability that a standard normal random variable is less than or equal to z.

We shall use the following algorithm for the generation of random variables following Standard Normal Distribution

Algorithm 3 Generate Standard Normal Variable

1: **Step 1:** Generate Y_1, an exponential random variable with rate 1.
2: **Step 2:** Generate Y_2, another exponential random variable with rate 1.
3: **Step 3:** If $Y_2 > \frac{1}{2}(Y_1 - 1)^2$, set $Y = Y_2 - \frac{1}{2}(Y_1 - 1)^2$ and go to **Step 4**. Else, go to **Step 1**.
4: **Step 4:** Generate a uniform random number $U \sim \mathcal{U}[0, 1]$ and set

$$Z = \begin{cases} Y_1, & \text{if } U \leq 0.5, \\ -Y_1, & \text{if } U > 0.5. \end{cases}$$

The random variables Z and Y generated by the above algorithm are independent, with Z being standard normal and Y being exponential.

Same code can be illustrated for 1000 random samples generated via Exponential Distribution, we will be using the builtin function *rexp()* to obtain the Exponential Distribution, following which we will use the above algorithm to generate the random sample following Standard Normal Distribution, the code in R is provided below for the same:

```
generate_normal_rv <- function() {
  while(1){
```

```
    y1 <- rexp(1,1)
    y2 <- rexp(1,1)

    if(y2 > (0.5)*(y1-1)^2){
      y <- y2 - (0.5)*(y1-1)^2
      u <- runif(1,0,1)

      if(u<=0.5){
        return(y1)
      }else{
        return(-y1)
      }
    }
  }
}

generate_normal <- function(n) {
  X <- numeric(0)
  for(i in 1:n){
    X <- append(X, generate_normal_rv())
  }
  return (X)
}

normal_pdf <- function (x){
  return((1/sqrt(2*pi))*exp(-x^2/2))
}
```

We can illustrate the following in a histogram with X-axis representing the Values in the Distribution that are generated and y-axis represents the Frequencies of those values in the distribution.

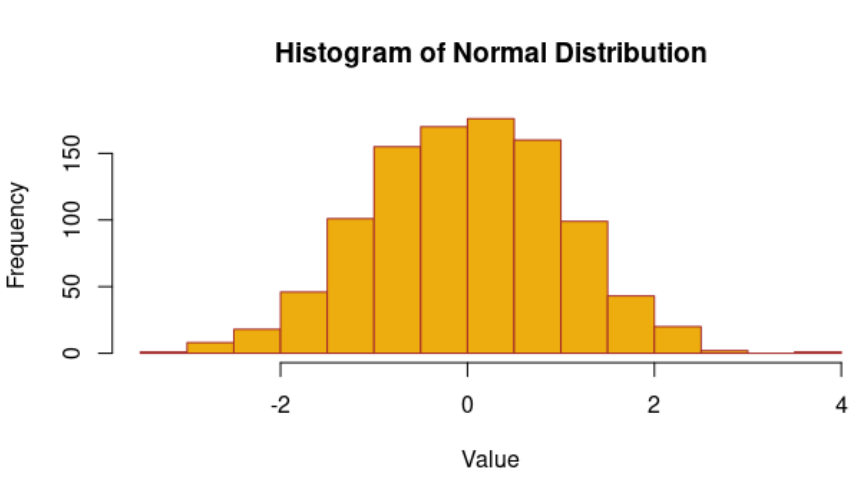

Figure 2: The given figure is a histogram plot of Standard Normal Distribution

Now we shall implement the Chi Square Goodness of Fit test to check if the required distribution

is indeed Standard Normal. The following code illustrates the usage of Chi Square Goodness of Fit:

```
t1 <- Sys.time()
X <- generate_normal(1000)
hist(X, main = "Histogram of Normal Distribution", xlab = "Value", ylab = "
    Frequency", col = "darkgoldenrod2", border = "brown")
maxine <- max(X)
minine <- min(X)

n <- 10
k <- (maxine-minine)/n
ints <- numeric(0)
E <- numeric(0)
O <- numeric(0)

for(i in 1:n){
  ints <- append(ints, minine +(i-1)*k)
}

ints[n+1] <- maxine

for(i in 1:n){
  cdfValue <- simpsons_rule_integral(normal_pdf,ints[i],ints[i+1],1000)
  E <- append(E, cdfValue*1000)
  O <- append(O, length(X[X<=ints[i+1]])-length(X[X<ints[i]]))
}
W
critical_value
W <- sum((O-E)^2/E)
critical_value <- qchisq(0.95,n-1)
print(critical_value)

if(W > critical_value){
  sprintf("The given distribution doesnt follow Standard Normal Distribution
      ")
} else {
  sprintf("The given distribution follows Standard Normal Distribution")
}

t2 <- Sys.time()
print(t2-t1)
```

Rejection Method for Discrete Probability Distributions

Suppose we have a method for simulating a random variable having probability mass function $\{q_j, j \geq 0\}$. We can use this to simulate from a discrete distribution having probability mass function $\{p_j, j \geq 0\}$.

Then X has density p_j.

We will be using this above algorithm to solve an exercise involving a discrete distribution.

Algorithm 4 Rejection Method for Discrete Probability Distributions

1: **Step 1:** Simulate Y having density q_j and simulate a random number U.
2: **Step 2:** If $U \leq \frac{p_j}{cq_j}$, set $X = Y$. Otherwise, return to **Step 1**.

Algorithm: Generating a Random Variable with Given Probabilities

To generate a random variable X that takes one of the values $1, 2, \ldots, 10$ with respective probabilities $0.11, 0.12, 0.09, 0.08, 0.12, 0.10, 0.09, 0.09, 0.10, 0.1$:

Algorithm 5 Generate Random Variable X

1: **Step 1:** Generate a random number U_1 and set $Y = \lfloor 10U_1 \rfloor + 1$.
2: **Step 2:** Generate a second random number U_2.
3: **Step 3:** If $U_2 \leq \frac{P_Y}{0.12}$, set $X = Y$ and stop. Otherwise, return to **Step 1**.

Here, P_Y represents the probability associated with the value Y.

Same code can be illustrated for 1000 random samples generated via Uniform Distribution, we will be using the builtin function *runif()* to obtain the Uniform Distribution, following which we will use the above algorithm to generate the random sample,the code in R is provided below for the same:

```
p <- c(0.11, 0.12, 0.09, 0.08, 0.12, 0.10, 0.09, 0.09, 0.10, 0.10)

generate_random_element <- function(){
  while(TRUE){
    u1 <- runif(1,0,1)
    y <- floor(10*u1)+1
    u2 <- runif(1,0,1)
    if(u2<=(p[y]/(0.12))){
      return(y)
    }
  }
}

generate_sample <- function(n){
  X <- numeric(0)
  for(i in 1:n){
    X <- append(X, generate_random_element())
  }
  return(X)
}
```

We can illustrate the following in a histogram with X-axis representing the Values in the Distribution that are generated and y-axis represents the Frequencies of those values in the distribution.

Now we shall implement the Chi Square Goodness of Fit test to check if the required distribution. The following code illustrates the usage of Chi Square Goodness of Fit:

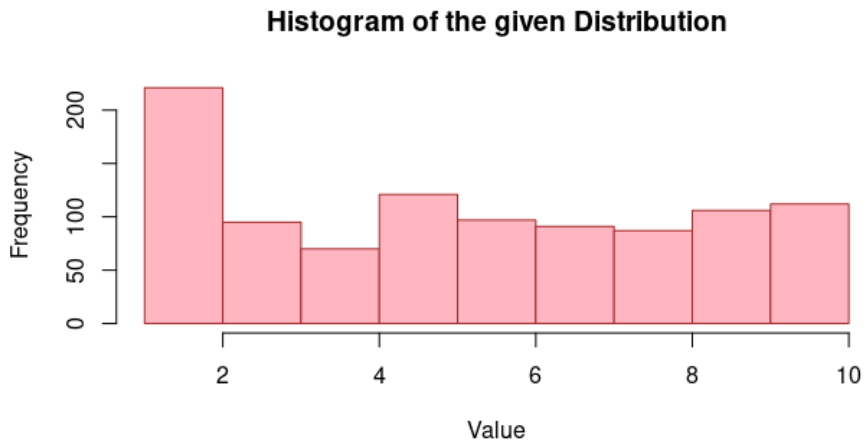

Figure 3: The given figure is a histogram plot of Standard Normal Distribution

```r
# Now we shall calculate the frequency of each element in X
O <- numeric(0)
E <- numeric(0)

# O and E calculations:
for(y in unique(X)){
  O <- append(O, length(X[X == y]))
  E <- append(E, (1000 * p[y]))
}

# Calculations of values
W <- sum(((O-E)^2)/E)
criticial_value <- qchisq(0.95, length(E)-1)

if(W > criticial_value){
  print("The given distribution doesnt follow the given Distribution")
} else {
  print("The given distribution follows the given Distribution")
}
```

Comments on the which algorithm has a better computational efficiency

The algorithms involved in this problem are the ones used in Assignment-3 and Assignment-1. Computationally, we observe that the first algorithm takes a time difference of 0.01368785 seconds to run, while the second algorithm takes 0.04322815 seconds to run. This is because, in the first case, we are simply generating random variables and then using the corresponding sin and cos functions to transform the random variables. In the second case, we are running a while loop to generate a

random variable that satisfies a given condition. The while loop will keep running until the condition is met, leading to a longer runtime.