Statistics Software Lab Report - 4

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

IIT Kharagpur
Statistics Software Lab

Generation of Random Permutation

A random permutation is a random ordering of a set of objects, that is, a permutation-valued random variable. The use of random permutations is often fundamental to fields that use randomized algorithms such as coding theory, cryptography, and simulation. A good example of a random permutation is the shuffling of a deck of cards: this is ideally a random permutation of the 52 cards.

There are several methods used in the generation of random permutations:

The Brute Force method

The entry-by-entry brute force method for the generation of random permutations involves assigning random values to each entry of a permutation one by one. Here's a basic outline of the process:

Algorithm 1 Entry-by-Entry Brute Force for Random Permutations

```
procedure GenerateRandomPermutationBruteForce(Set S)
    Initialize an empty permutation P
    while P is not a complete permutation do
        Choose a random element x from S that has not been used in P
        Add x to the next available position in P
    end while
    return P                                   ▷ Resulting random permutation
end procedure
```

Fisher Yates Shuffle

A simple algorithm to generate a permutation of n items uniformly at random without retries, known as the Fisher–Yates shuffle, is to start with any permutation (for example, the identity permutation), and then go through the positions 0 through $n-2$, and for each position i swap the element currently there with a randomly chosen element from positions i through $n-1$ (the end), inclusive. It's easy to verify that any permutation of n elements will be produced by this algorithm with probability exactly $\frac{1}{n!}$, thus yielding a uniform distribution over all such permutations.

Algorithm 2 Fisher-Yates Shuffle

```
procedure FisherYatesShuffle(Array A)
    for i ← length(A) downto 2 do
        Choose a random index j such that 1 ≤ j ≤ i
        Swap A[i] and A[j]
    end for
end procedure
```

Random Number Generation in R

The following algorithm is used to generate random permutation in R:

Algorithm 3 Generating a Random Permutation

procedure GENERATERANDOMPERMUTATION(n)
  Let $p_1, p_2, \ldots, p_n$ be any permutation of $1, 2, \ldots, n$ (e.g., $p_j = j$ for $j = 1, \ldots, n$).
  Set $k = n$.
  while $k > 1$ **do**
    Generate a random number $U \sim U(0, 1)$ and let $I = \lfloor kU \rfloor + 1$.
    Interchange the values of p_I and p_k.
    Let $k = k - 1$.
  end while
  return $p_1, p_2, \ldots, p_n$ ▷ Desired permutation
end procedure

The above algorithm can be shown by the implementation of the following code in R with input permutation taken as: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

```
set.seed(67)

generate_random_permutation <- function (n, p) {
  k <- n

  while(k>=1){
    u <- runif(1, 0, 1)
    I <- floor(k*u) + 1
    temp <- p[I]
    p[I] <- p[k]
    p[k] <- temp
    k <- k-1
  }

  return(p)
}

n <- 10
p <- seq(1, n, 1)
cat("Initial Permutation is: \n")
cat(p)
q <- generate_random_permutation(n, p)
cat("\nFinal Permutation is: \n")
cat(q, "\n")
plot(q, col="red", xlab="Values", ylab="Indices")
```

We can illustrate the following in a plot with X-axis representing the values that are obtained and y-axis represents the indices (Original Positions in this case).

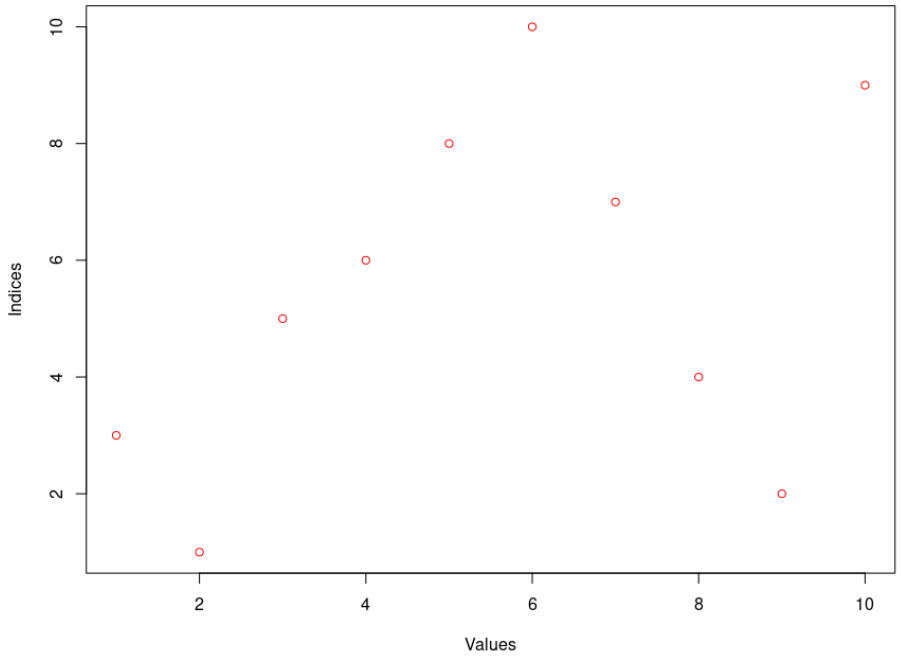

Figure 1: This is plot of a random permutations

Generation of Stationery Poisson Process

A Stationary Poisson Process is a stochastic process that models the occurrence of events in continuous time. It is characterized by the following properties:

- **Rate Parameter λ:** The process has a constant rate of events per unit time, denoted by $\lambda > 0$.
- **Counting Process $N(t)$:** Let $N(t)$ represent the number of events that have occurred by time t. The counting process increments by 1 whenever an event occurs.
- **Memoryless Property:** The time until the next event follows an exponential distribution, meaning that the process has no memory of past events. This property is expressed by the formula $P(T > t + s|T > s) = P(T > t)$ for all $s, t \geq 0$, where T is the time until the next event.

The Stationary Poisson Process is often used to model rare events that occur randomly in time, such as the arrival of customers at a service point, radioactive decay, or the arrival of packets in a computer network.

Probability Density Function (PDF)

The probability density function of a stationary Poisson process is given by:

$$f_N(n;\lambda,t) = \frac{e^{-\lambda t}(\lambda t)^n}{n!}, \quad n = 0, 1, 2, \ldots$$

where n is the number of events, λ is the rate of events per unit time, and t is the length of the time interval.

Cumulative Distribution Function (CDF)

The cumulative distribution function of a stationary Poisson process is given by:

$$F_N(n;\lambda,t) = \sum_{k=0}^{n} \frac{e^{-\lambda t}(\lambda t)^k}{k!}, \quad n = 0, 1, 2, \ldots$$

This represents the probability that there are at most n events in the time interval $[0, t]$.

Let the rate be $\lambda > 0$, t refer to time, I is the number of events that have occurred by time t, and $S(I)$ is the most recent time.

We shall be using the following algorithm for the generation of Stationary Poisson Process:

Algorithm 4 Generation of Stationary Poisson Process

1: **Initialization:** Set $t = 0$ and $I = 0$.
2: **Generate a Random Number:** Generate a random number U.
3: **Update Time:** Update t by $t = t - \frac{1}{\lambda} \ln U$. If $t > T$, stop.
4: **Update Event Count:** Increment I by 1, and update $S(I) = t$.
5: **Repeat:** Go back to step 2.

```r
generate_stationery_poisson <- function(T, S, lambda){
  t <- 0
  I <- 0

  while(t < T){
    u <- runif(1, 0, 1)
    t <- t - (1/lambda)*log(u)
    if (t>T)
      break
    I <- I + 1
    S[I] <- t
  }
  return(S)
}

T <- 10
S <- c()
lambda <- 5

S <- generate_stationery_poisson(T, S, lambda)
cat("The resultant array is as follows: ", S)
plot(S, col="lightblue", xlab="Values", ylab="Indices")
```

We can illustrate the following in a plot with X-axis representing the values that are obtained and y-axis represents the indices (Original Positions in this case).

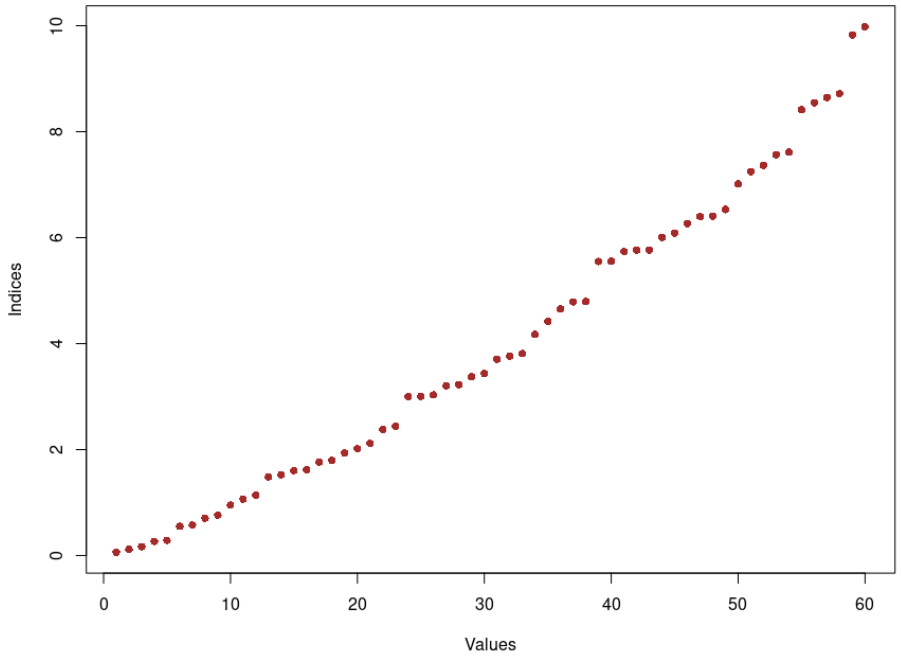

Figure 2: This is the plot of the above calculated S function

Generation of Non-stationary Poisson Process

A Non-Stationary Poisson Process is a stochastic process that models the occurrence of events in continuous time, similar to the Stationary Poisson Process. However, in the non-stationary case, the rate parameter $\lambda(t)$ is allowed to vary with time.

- **Time-Varying Rate $\lambda(t)$:** Unlike the constant rate in a stationary process, the rate parameter $\lambda(t)$ in a non-stationary process is a function of time. It can vary according to different patterns or external factors.
- **Counting Process $N(t)$:** Similar to the stationary case, let $N(t)$ represent the number of events that have occurred by time t. The counting process increments by 1 whenever an event occurs.
- **Varying Intensity:** The non-stationary nature allows for variations in the intensity of events over time. This can be useful in modeling scenarios where the rate of events changes due to external influences.

The probability mass function (PMF) for the number of events k occurring in a time interval of length t is still given by the Poisson distribution, but with the time-varying rate:

$$P(N(t) = k) = \frac{[\lambda(t)t]^k e^{-\lambda(t)t}}{k!}, \quad k = 0, 1, 2, \ldots$$

Modeling non-stationary processes is valuable in situations where the rate of events is subject to change, such as in dynamic systems or processes influenced by external factors.

The first algorithm for generation of a non stationery Poisson Process can be stated as follows:

Let the rate be $\lambda(t)$, where λ is the maximum intensity and $\lambda(t) \leq \lambda$. The final value of I represents the number of events that occurred by time T, and $S(1), \ldots, S(I)$ are the event times.

Algorithm 5 Non-Stationary Poisson Process

1: **Initialization:** Set $t = 0$ and $I = 0$.
2: **Generate a Random Number:** Generate a random number U.
3: **Update Time:** Update t by $t = t - \frac{1}{\lambda} \ln U$. If $t > T$, stop.
4: **Generate Another Random Number:** Generate another random number U.
5: **Check Intensity:** If $U \leq \frac{\lambda(t)}{\lambda}$, set $I = I + 1$ and $S(I) = t$.
6: **Repeat:** Go back to step 2.

The following is the code for the implementation of the above algorithm in R:

```r
# c)
# Algorithm - 1 : Generation of a non-stationery Poisson Process
intensity_function <- function (t, lambda) {
  return(lambda*exp(-t))
}

generate_non_stationery_poisson_algo_1 <- function(T, S, lambda_upper_bound,
    fn){
  t <- 0
  I <- 0

  while(t < T){
    u <- runif(1, 0, 1)
    t <- t - (1/lambda)*log(u)
    if (t>T)
      break
    if(u <= fn(t, lambda)/lambda_upper_bound){
      I <- I + 1
      S[I] <- t
    }
  }
  return(S)
}

T <- 1000
S <- c()
lambda <- 40

S <- generate_non_stationery_poisson_algo_1(T, S, lambda, intensity_function
   )
cat("The resultant array is as follows: ", S)
plot(S ,  col="blue", xlab="Values", ylab="Indices", pch=16)
```

We can illustrate the following in a plot with X-axis representing the values that are obtained and y-axis represents the indices (Original Positions in this case).

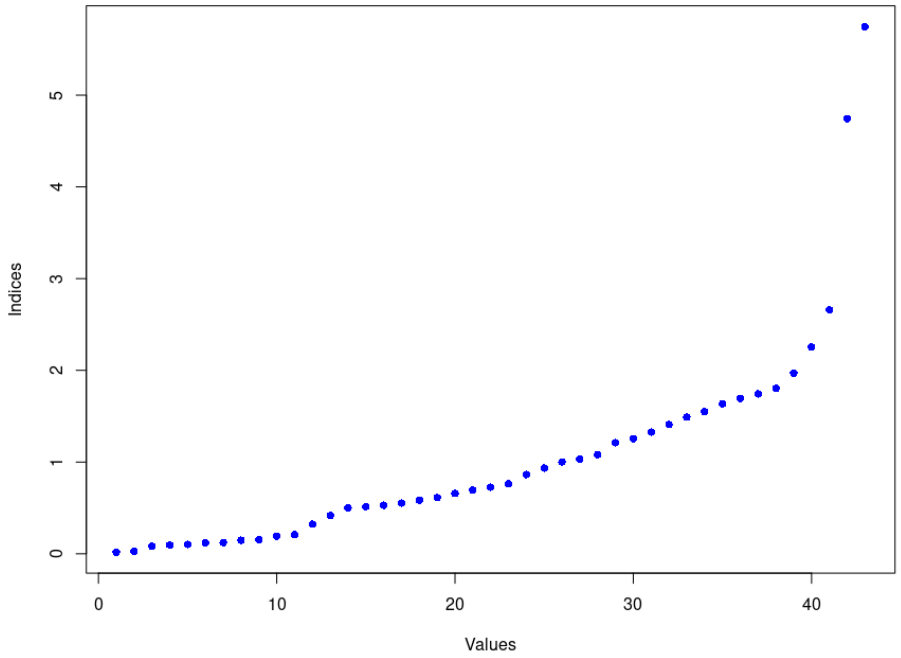

Figure 3: This is the plot of the above calculated S function

In another version of the algorithm, t represents the present time, J the present interval ($J = j$ when $t_{j-1} \leq t \leq t_j$), I is the number of events so far, and $S(1), \ldots, S(I)$ are the event times.

Algorithm 6 Generation of Non Stationery Poisson Process Algorithm

1: **Initialization:** Set $t = 0$, $J = 1$, $I = 0$.
2: **while** true **do**
3: **Generate a Random Number:** Generate a random number U and set $X = -\frac{1}{\lambda_J} \ln U$.
4: **Check Time:** If $t + X > t_J$, go to step 8.
5: **Update Time:** $t = t + X$.
6: **Generate Another Random Number:** Generate another random number U.
7: **if** $U \leq \frac{\lambda(t)}{\lambda}$ **then**
8: **Update Event Count:** Set $I = I + 1$, $S(I) = t$.
9: **end if**
10: **Repeat:** Go back to step 2.
11: **Check Interval:** If $J = k + 1$, stop.
12: **Update X and Interval:** $X = (X - t_J + t)\frac{\lambda_J}{\lambda_{J+1}}$, $J = J + 1$.
13: **Repeat:** Go back to step 3.
14: **end while**

The above algorithm can be implemented on R using the following code:

```
# Algorithm - 2 : Generation of a non-stationery Poisson Process
intensity_function <- function (t, lambda) {
  return(lambda*exp(-t))
}

generate_non_stationary_poisson_algo2 <- function(fn, intervals, l, k){
  t <- 0
  J <- 1
  I <- 0
  S <- numeric(0)
  flag <- FALSE

  while(flag == FALSE){
    u1 <- runif(1)
    X <- (-(1/l[J])*log(u1))
    while(TRUE){
      if(t+X <= intervals[J]){
        t<- t+X
        u2 <- runif(1)
        if(u2 <= fn(t,l[J])/l[J]){
          I <- I+1
          S[I] <- t
        }
        break
      }
      if(J == k+1){
        flag <- TRUE
        break
      }
      X <- (X-intervals[J]+t)*l[J]/l[J+1]
```

```
      J <- J + 1
    }
  }
  return(S)
}

intervals <- seq(10,100,5)
k <- length(intervals)-1
l <- numeric(0)

for(i in 0:k+1){
  l[i] = runif(1,1,50)
}
getAns <- generate_non_stationary_poisson_algo2(intensity_function,
    intervals, l, k)
print(getAns)
plot(getAns,  col="orange", xlab="Values", ylab="Indices", pch=16)
```

We can illustrate the following in a plot with X-axis representing the values that are obtained and y-axis represents the indices (Original Positions in this case).

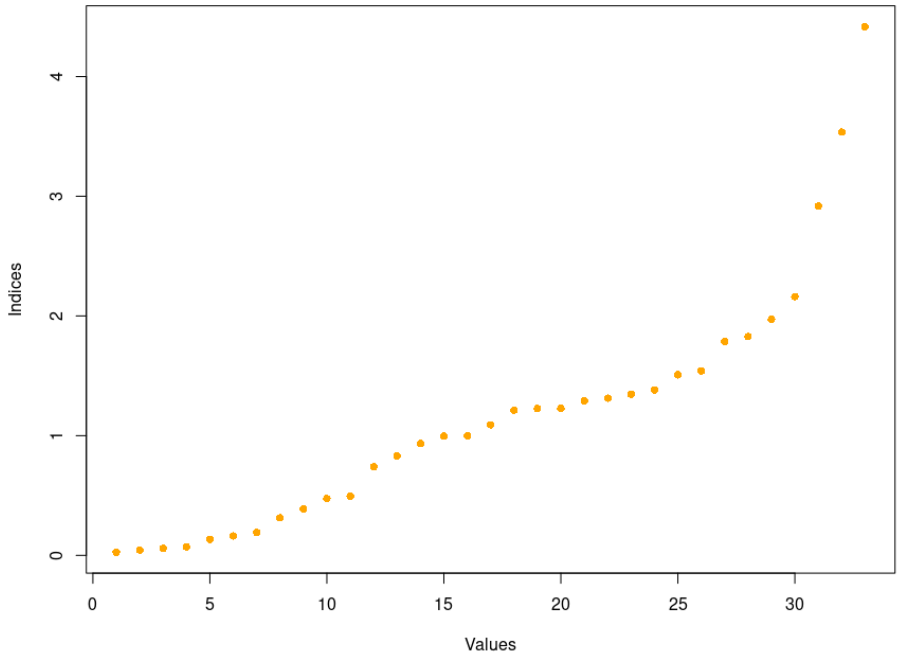

Figure 4: This is the plot of the above calculated S function