Statistical properties of R's parallel RNG

Moustafa Shaker and Yu Liu

4/19/2020

Introduction

This semester we studied Monte Carlo methods, which are a broad class of computational algorithms that depend on repeated random sampling to obtain numerical results. As we learned in class the underlying concept is to use randomness to solve problems that can be deterministic in nature. The methods of Monte Carlo can be used to solve problems in multiple domains such as physical sciences, engineering, computational biology, finance, and more. Some of those problems are computationally intensive in nature. Therefore, using parallel computing to execute calculations and processes simultaneously can be greatly beneficial. Luckily, Monte Carlo computations are easy to parallelize. However, the results can be negatively affected by defects in the parallel pseudo-random number generators used. A parallel pseudo-random number generator is considered defective if one or both of the following conditions apply:

- There is a strong correlation between random numbers in one stream.
- There is a strong correlation between random number streams on different processes.

pseudo-random number generators have finite possible states. Therefore, the sequences will start repeating after a certain period. Sometimes a sequence will actually stop behaving like a truly random sequence before the period is exhausted, and that leads to correlations between different parts of the sequence or between different streams on different processes.

Statistical tests can be used to examine the quality of a random sequence produced by a pseudo-random number generator. Furthermore, statistical tests are designed to compare some statistics obtained using a pseudo-random number generator with what we would obtain from a truly random independent and identically distributed numbers on the unit interval. If the results are similar, then our confidence in the pseudo-random number generator increases.

In this project, we will study the quality of parallel pseudo-random number generators in R by examining their statistical properties. The outline of this project is as follows. We will first introduce methods of parallelizing pseudo-random number generators. Then, we will explain the algorithms used by the pseudo-random number generators that we will test in this project. We will also describe the statistical tests we will use in the project and how they work. Finally, we will conclude with a summary of the results and further recommendations.

Methods to parallelize pseudo-random number generators

Before testing the qualify of pseudo parallel random number generators, it's important to understand the design behind pseudo parallel random numbers generators to see how they work. In this section, we will discuss different methods of designing pseudo parallel random number generators. We will start by defining a few terminologies used in sequential pseudo-random number generators, as this will help us better understand the process of how those sequential pseudo-random number generators are parallelized. As we mentioned in the introduction section, a pseudo-random number generator has a finite set of states. It also has a transition function F that take the pseudo-random number generator from one state to the next. The first state of a pseudo-random number generator is known as a seed. So if we have a state X_i for a pseudo-random number generator, then there is a function G that gives a corresponding random number N_i. We note that each pseudo-random number generator has a finite number of possible states. This means that the random sequence wills start to repeat itself leading to non-randomness. The length of a pseudo-random number generator is known as a period.

Now that we have laid the ground and defined a few concepts with regard to a sequential pseudo-random number generators, we will discuss common methods to parallelize sequential pseudo-random number generators. Here are the methods:

1. Cycle division: In this method, a cycle that corresponds to a single random number generator is divided among different processors such that each processor gets a different portion of the cycle. The cycle division usually happens in one of 3 ways:

- The user can randomly select a different seed for each processor. The user does this in the hope that each seed will take the processors to separate potions of the cycle so that there is no overlap between the random number streams that are used by different processors.
- Sequence splitting scheme: the sequence splitting scheme is another way to divide a cycle of a pseudo-random number generator. Here the user deterministically chooses widely separate seeds for each processor. The problem with this approach is that if the user consumes more random numbers than originally planned, then the streams on different processors may overlap. Furthermore, sometimes pseudo-random number generators have long-range correlation issues. Those long-range correlation issues in sequential pseudo-random number generators become short-range correlation issues between random number streams on different processors.
- Leapfrog scheme: Let's say that we have n processors so that each stream in the leapfrog scheme gets random numbers that are n positions away from each other in the original sequence. For instance, processor 0 will get random numbers x_0,x_n,x_{2n},etc. The issue that can arise here is that if the original sequence of random numbers has long-range correlation issues, this can lead to short-range correlation issues in any of the streams of random numbers.

2. Parameterization: This method provides full-period independent streams for each processor. The user can parameterize a set of streams by using a stream number. So given a stream number j, there is a way of making the jth stream. There are generally two ways of parameterization:

- Seed parameterization: In some generators, the possible states can be divided into a number of smaller cycles. Those cycles can be numbered from 0 to $N-1$. Here the N refers to the total number of cycles. In this approach, we are able to give each processor a seed from a different cycle.
- Iteration function parameterization: In this approach, a different iteration function, which is the function that takes the current state of a sequence and gives the next state, is used for each stream. The iteration function is parameterized so that given i, the ith iteration function can be produced.

Now that we understand the different methods to parallelize pseudo-random number generators, we will next discuss the algorithms of the pseudo parallel random numbers generators that will test their quality in this project.

Pseudo parallel random numbers generators algorithms

In this section, we will focus on defining the algorithms behind the 2 pseudo parallel random number generators that we will test in this project. Furthermore, we will define the method used to parallelize those pseudo-random number generators.

L'Ecuyer Combined multiple-recursive generator

This generator was designed with parallel computations in mind. It works using a separate stream for each parallel computation to make sure that the random numbers don't get into sync. Furthermore, the parallel computations can themselves use substreams. This generator has two components each of order 3. At step n, its state is the pair of vectors $s_{1,n} = (x_{1,n},x_{1,n+1},x_{1,n+2})$ and $s_{2,n} = (x_{2,n},x_{2,n+1},x_{2,n+2})$, which evolve according to the following linear recurrences:

$$x_{1,n} = (1403580 * x_{1,n-2} - 810728 * x_{1,n-3}) \ mod \ m_1$$

$$x_{2,n} = (527612 * x_{2,n-1} - 1370589 * x_{2,n-3}) \ mod \ m_2$$

where $m_1 = 2^{32} - 209 = 4294967087$ and $m_2 = 2^{32} - 22853 = 4294944443$, and its output u_n is defined by the following:

$$z_n = (x_{1,n} - x_{2,n}) mod 4,294,967,087$$

where $u_n = z_n/4294967088$ if $z_n > 0$ or $u_n = 4294967087/4294967088$ if $z_n = 0$.

We note that the period of this generator is 2^{191} and that each stream is a subsequence that has a period of 2^{127}. As we noted earlier this generator allows parallel computations to use substreams where each of those substreams has a period of 2^{76}.

In order to test this generator, we will use the parallel package from R to generate different sequences and test their statistical properties.

64-Bit Linear congruential generator

This generator's recurrence relation for a sequence of random numbers is given by:

$$x_n = ax_{n-1} + p \ mod \ 2^{64}$$

where a is a multiplier and p is a prime number. Using this generator we can obtain different random number streams by choosing different prime number p. The period of this generator is equal to 2^{64}. Also, the number of available distinct streams is more than 2^{24}.

In order to test this generator, we will use the rTRNG (Advanced and Parallel Random Number Generation) package from R to generate different sequences and test their statistical properties.

Statistical tests:

In this section, we will be testing the quality of the random number sequences produced by the generators that we defined in the previous section. We will be testing each sequence for correlations within a stream produced by one process and also we will be testing for correlations between different streams produced by different processes. If the random number streams obtained from our generators have the properties of independent and identically distributed random samples drawn from the uniform distribution, then our pseudo parallel random number generators are of good quality.

The statistical tests are designed such that the distribution of some test statistic is known for the uniform distribution. Then the generated random number stream will be subjected to the same test. The statistic we get from the generated sequence will be compared against the known distribution.

The sequential statistical tests of random numbers can be modified to test parallel pseudo-random number generators. This can be done by interleaving different streams to produce a new random number stream. This new stream can then be tested using the standard sequential tests. For instance, if stream j is give by x_{j0}, x_{j1}, etc where $0 \leq j \leq N$, then the new stream is given by $x_{00}, x_{10}, ..., x_{N-1,0}, x_{01}, x_{11}, etc.$ If each of the individual streams produced by different processes is random and the streams are independent of each other, then the new stream will also be random. On the other hand, if there is any correlation between streams or within one stream, then the test will show that there is a correlation between the random numbers in the interleaved stream.

We are going to use 5 different statistical tests to examine the quality of the streams of random numbers generated by the 2 generators we introduced in the previous section. For each generator, we will test 1000 sequences each of length 100,000 random numbers. Before we start our testing we will first generate the sequences. We will start with the L'Ecuyer Combined multiple-recursive generator:

```
#L'Ecuyer Combined multiple-recursive generator

#Loading the necessary package
library(parallel)
library(randtoolbox)

#Create a cluster with 4 cores
cl <- makeCluster(4)

#Setting the right parallel RNG
iseed <- 0
clusterSetRNGStream(cl = cl, iseed = iseed)

#Confirming that our generator is the L'Ecuyer Combined multiple-recursive generator
parLapply(cl, 1:4, function(n)RNGkind())

#Each stream will have 100,000 random numbers
nSims <- 100000
taskFun <- function(i){
  val <- runif(1)
  return(val)
}

#Defining matrix to hold generated sequences
cmrg_sequences = matrix(nrow=1000, ncol=100000)

#Generate 1000 random sequences in parallel each of length 100,000 using the L'Ecuyer Combined #multipl
for (i in 1:1000){
  cmrg_sequences[i,] <- parSapply(cl, 1:nSims, taskFun)
}

#Stopping the cluster
stopCluster(cl)
```

Now we will use the 64-Bit Linear congruential generator:

```
#64-Bit Linear congruential generator

#Loading the rTRNG library
library(rTRNG)

#Indicating that processes should be done in parallel with 4 threads
RcppParallel::setThreadOptions(numThreads = 4L)

#Defining matrix to hold generated sequences
lcg64_sequences = matrix(nrow=1000, ncol=100000)

#Setting up the right generator
TRNGkind("lcg64")
```

```
#Confirming that the right generator is in use
TRNGkind()

#Generate 1000 random sequences in parallel each of length 100,000 using the 64-Bit Linear congruential
for (i in 1:1000){
   lcg64_sequences[i,] <- runif_trng(100000, min = 0, max = 1, parallelGrain = 100L)
}
```

Next, we will define the statistical tests that we will use to test the qualify of the generated random number sequences as well as carry out the testing.

Gap test:

Gap test examines the significance of the interval between recurrence of the same digit. A gap of length x occurs between the recurrence of some digit. For example, if we are looking for digit 3 in a sequence, and there are a total of ten 3's in the sequence, then only 9 gaps can occur. The probability of a particular gap length can be determined by Bernoulli trail:

$$P(\text{gap of length n}) = P(x \neq 3)P(x \neq 3)...P(x \neq 3)P(x = 3))$$

If we only care about the digits between 0 and 9, then

$$P(\text{gap of length n}) = 0.9^n 0.1$$

The theoretical frequency distribution for randomly ordered digits is given by

$$P(\text{gap} \leq x) = F(x) = 0.1 \sum_{n=0}^{x} (0.9)^n = 1 - 0.9^{x+1}$$

Gap test computes the length of zero gaps. If we denote the number of zero gaps of length j by n_j. The chi-squared statistic of such a test is given by

$$S = \sum_{j=1}^{m} \frac{(n_j - np_j)^2}{np_j}$$

where $p_j = (1-p)^2 p^j$ is the probability that the length of gaps equals to j, and m is the max number of lengths.

Frequency test:

The frequency test is designed to work on a serie of ordered contiguous integers ($J = [i_1, ..., i_l] \cap \mathbb{Z}$). If we denote the sample number of the set I by $(n_i)_{1 \leq i \leq n}$, the expected number of intergers equals to $j \in J$ is $\frac{1}{i_l - i_1 + 1} \times n$, which is independent of j. Next, we can obtained the chi-square statistic from $S = \sum_{j=1}^{l} \frac{(Card(n_i = i_j) - m)^2}{m}$, where $m = \frac{n}{d}$.

Serial test:

In many respects, the serial test is similar to the chi-square test on frequency. To be more specific, the serial test examines whether sequential, non-overlapping pairs of random numbers have a uniform distribution. This test groups the random numbers into a relatively small number of groups. The most intuitive way to

split the unit hypercube $[0,1]^t$ into $k = d^t$ subcubes. It is achieved by splitting each dimension into $d > 1$ pieces. The volume of each cell is just $\frac{1}{k}$. The associated chi-square statistic is defined as

$$S = \sum_{j=1}^{m} \frac{(N_j - \lambda)^2}{\lambda}$$

where N_j denotes the counts and $\lambda = \frac{n}{k}$ denotes their expectation.

Poker test:

The poker test treats numbers grouped together as a poker's hand. Then the hands obtained are compared to what is expected using the chi-square test. Let us consider a hand of k cards from k different cards. The probability to have exactly c different cards is

$$P(C = c) = \frac{1}{k^k} \frac{k!}{(k-c)!} {}_2S_k^c$$

where C is the random number of different cards and ${}_2S_k^c$ is the second-kind Stirling numbers.

Order test:

The order test examines a d-tuple, if its components are ordered equiprobably. For example, if $d = 3$, we should have an equal number of vectors $(u_i, u_{i+1}, u_{i+2})_i$ such that

$$u_i < u_{i+1} < u_{i+2}$$
$$u_i < u_{i+2} < u_{i+1}$$
$$u_{i+1} < u_i < u_{i+2}$$
$$u_{i+1} < u_{i+2} < u_i$$
$$u_{i+2} < u_i < u_{i+1}$$
$$u_{i+2} < u_{i+1} < u_i$$

For some d, we have $d!$ possible orderings of coordinates, which have the same probability to appear $\frac{1}{d!}$. The chi-square statistic for the order test for a sequence $(u_i)_{1 \leq i \leq n}$ is just

$$S = \sum_{j=1}^{d!} \frac{(n_j - m/d!)^2}{m/d!}$$

where n_j's are the counts for different orders and $m = \frac{n}{d}$. Computing $d!$ possible orderings has an exponential cost, so in practive d is small.

Now that we understand the theory behind each statistical test, we will implement each test in R to test the sequences we generated earlier.

Gap test R implementation using sequences generated from the L'Ecuyer Combined multiple-recursive generator

```
r1 = rep(0,1000)
for (i in 1:1000){
   if (gap.test(cmrg_sequences[i,])$p.value < 0.05){
     r1[i] =1
   }
}
cat("Rejection:", sum(r1)/1000)
```

Here is a summary of the gap test results:

Gap test

chisq stat = 26, df = 16, p-value = 0.053

(sample size : 100000)

length observed freq theoretical freq 1 12284 12500 2 6325 6250 3 3216 3125 4 1593 1562 5 803 781 6 383 391 7 221 195 8 100 98 9 51 49 10 18 24 11 9 12 12 5 6.1 13 7 3.1 14 0 1.5 15 0 0.76 16 0 0.38 17 1 0.19

Rejection: 0.17

Out of 1000 sequences, 170 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Frequency test R implementation using sequences generated from the L'Ecuyer Combined multiple-recursive generator

```
r2 = rep(0,1000)
for (i in 1:1000){
   if (freq.test(cmrg_sequences[i,],1:10)$p.value < 0.05){
     r2[i] =1
   }
}
cat("Rejection:", sum(r2)/1000)
```

Here is a summary of the frequency test results:

Frequency test

chisq stat = 5.6, df = 9, p-value = 0.78

(sample size : 100000)

observed number 9867 9973 9955 9931 10045 10106 10096 10049 10041 9937 expected number 10000

Rejection: 0.058

Out of 1000 sequences, 58 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Serial test R implementation using sequences generated from the L'Ecuyer Combined multiple-recursive generator

```
r3 = rep(0,1000)
for (i in 1:1000){
  if (serial.test(cmrg_sequences[i,])$p.value < 0.05){
    r3[i] =1
```

```
  }
}
cat("Rejection:", sum(r3)/1000)
```

Here is a summary of the serial test results:

Serial test

chisq stat = 72, df = 63, p-value = 0.2

(sample size : 100000)

observed number 813 787 811 791 808 815 788 729 744 789 762 790 783 731 786 745 792 801 771 773 753 780 754 742 822 792 737 808 840 770 738 805 743 749 764 791 841 753 790 784 803 761 785 739 732 777 812 770 851 780 760 771 782 799 848 774 777 838 810 751 771 803 764 777 expected number 781

Rejection: 0.048

Out of 1000 sequences, 48 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Poker test R implementation using sequences generated from the L'Ecuyer Combined multiple-recursive generator

```
r4 = rep(0,1000)
for (i in 1:1000){
  if (poker.test(cmrg_sequences[i,])$p.value < 0.05){
    r4[i] =1
  }
}
cat("Rejection:", sum(r4)/1000)
```

Here is a summary of the poker test results:

Poker test

chisq stat = 5.9, df = 4, p-value = 0.21

(sample size : 100000)

observed number 39 1833 9653 7709 766 expected number 32 1920 9600 7680 768

Rejection: 0.054

Out of 1000 sequences, 54 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Order test R implementation using sequences generated from the L'Ecuyer Combined multiple-recursive generator

```
r5 = rep(0,1000)
for (i in 1:1000){
  if (order.test(cmrg_sequences[i,],d=5)$p.value < 0.05){
    r5[i] =1
  }
}
cat("Rejection:", sum(r5)/1000)
```

Here is a summary of the order test results:

Order test

chisq stat = 122, df = 119, p-value = 0.41

(sample size : 100000)

observed number 176 162 172 146 182 160 169 182 165 166 172 187 180 163 205 180 172 167 161 162 143 193 147 185 171 150 156 168 166 158 164 162 150 181 165 166 163 171 172 160 188 173 147 171 194 171 178 162 143 160 173 160 152 171 148 155 172 140 175 174 182 166 153 161 163 139 175 165 169 193 168 169 160 170 148 161 187 164 149 157 169 171 163 160 166 157 154 178 167 162 199 178 175 188 167 141 163 143 162 158 147 153 152 174 164 175 167 167 186 195 166 171 171 159 170 184 155 154 177 166 expected number 167

Rejection: 0.04

Out of 1000 sequences, 40 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Gap test R implementation using sequences generated from the 64-Bit Linear congruential generator

```
l1 = rep(0,1000)
for (i in 1:1000){
   if (gap.test(lcg64_sequences[i,])$p.value < 0.05){
     l1[i] =1
   }
}
cat("Rejection:", sum(l1)/1000)
```

Here is a summary of the gap test results:

Gap test

chisq stat = 33, df = 16, p-value = 0.0067

(sample size : 100000)

length observed freq theoretical freq 1 12750 12500 2 6328 6250 3 3062 3125 4 1571 1562 5 740 781 6 362 391 7 193 195 8 93 98 9 41 49 10 18 24 11 12 12 12 3 6.1 13 5 3.1 14 3 1.5 15 4 0.76 16 0 0.38 17 0 0.19

Rejection: 0.18

Out of 1000 sequences, 180 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Frequency test R implementation using sequences generated from the 64-Bit Linear congruential generator

```
l2 = rep(0,1000)
for (i in 1:1000){
   if (freq.test(lcg64_sequences[i,])$p.value < 0.05){
     l2[i] =1
   }
}
cat("Rejection:", sum(l2)/1000)
```

Here is a summary of the frequency test results:

Frequency test

chisq stat = 11, df = 15, p-value = 0.79

(sample size : 100000)

observed number 6337 6251 6256 6164 6361 6193 6228 6212 6261 6187 6263 6285 6161 6341 6326 6174 expected number 6250

Rejection: 0.06

Out of 1000 sequences, 60 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Serial test R implementation using sequences generated from the 64-Bit Linear congruential generator

```
l3 = rep(0,1000)
for (i in 1:1000){
   if (serial.test(lcg64_sequences[i,])$p.value < 0.05){
     l3[i] =1
   }
}
cat("Rejection:", sum(l3)/1000)
```

Here is a summary of the serial test results:

Serial test

chisq stat = 45, df = 63, p-value = 0.96

(sample size : 100000)

observed number 728 824 764 806 781 773 794 796 770 780 761 769 796 734 770 790 756 741 776 763 775 800 836 811 794 777 772 748 766 747 800 802 793 764 772 784 821 786 789 801 771 763 786 812 781 800 800 788 773 799 777 790 766 784 796 793 759 828 742 785 713 807 782 795 expected number 781

Rejection: 0.047

Out of 1000 sequences, 47 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Poker test R implementation using sequences generated from the 64-Bit Linear congruential generator

```
l4 = rep(0,1000)
for (i in 1:1000){
   if (poker.test(lcg64_sequences[i,])$p.value < 0.05){
     l4[i] =1
   }
}
cat("Rejection:", sum(l4)/1000)
```

Here is a summary of the poker test results:

Poker test

chisq stat = 0.66, df = 4, p-value = 0.96

(sample size : 100000)

observed number 35 1930 9614 7643 778 expected number 32 1920 9600 7680 768

Rejection: 0.051

Out of 1000 sequences, 51 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Order test R implementation using sequences generated from the 64-Bit Linear congruential generator

```
l5 = rep(0,1000)
for (i in 1:1000){
  if (order.test(lcg64_sequences[i,],d=5)$p.value < 0.05){
    l5[i] =1
  }
}
cat("Rejection:", sum(l5)/1000)
```

Here is a summary of the order test results:

Order test

chisq stat = 86, df = 119, p-value = 0.99

(sample size : 100000)

observed number 163 179 165 162 175 168 162 162 173 170 181 174 176 178 160 169 154 160 162 179 162 184 189 190 176 180 167 177 151 171 161 160 182 173 166 174 160 159 162 168 187 172 170 170 176 168 156 169 164 168 157 149 166 165 173 157 164 161 165 157 179 161 152 200 156 183 172 175 176 169 171 168 166 152 131 154 154 155 156 176 163 166 160 157 158 152 162 174 175 162 170 163 166 131 171 180 160 159 181 155 172 152 170 147 182 155 182 164 158 177 173 171 177 166 164 161 187 165 160 148 expected number 167

Rejection: 0.041

Out of 1000 sequences, 41 cases failed the null hypothesis. The null hypothesis states that the random numbers in the sequence are independent and identically distributed.

Conclusion and further recommendations

In this section, we will examine the results of the 5 statistical tests we implemented in the previous section. Furthermore, we will compare the performance of the 64-Bit Linear congruential generator (LCG) with the L'Ecuyer Combined multiple-recursive generator (CMRG). We note that the results of the statistical tests are considered passed if the KS percentile is between 2.5% and 97.5%. Each of the generators, the LCG, and the CMRG generated 1000 sequences and each sequence have 100,000 random number. All of the sequences were generated using 4 parallel cores. We found that 83% of the sequences generated by the CMRG passed the gap test, and 82% of the sequences generated by the LCG passed the gap test. When a sequence passes a test this means that the random numbers in the sequence are independent and identically distributed. So there was no correlation between random numbers in a stream generated on one processor and no correlation between streams generated on different processors. The gap test shows that the quality of the random number sequences generated by the LCG is very similar to those generated by the CMRG. The results of the frequency test also show that the performance of both generators is similar. We found that 94.2% of the sequences generated by CMRG passed the frequency test and 94% of the sequences generated by the LCG passed the frequency test. The serial, poker, and order tests also show a similar story. Out of

the 1000 sequences generated by the CMRG 95.2% passed the serial test, 94.6% passed the poker test, and 96% passed the order test. Furthermore, out of the 1000 sequences generated by the LCG 95.3% passed the serial test, 94.9% passed the poker test, and 95.9% passed the order test. While not all the sequences passed all the tests, the percentage of the sequences that passed the tests is reasonably high. This shows that the random number sequences parallelly generated by CMRG or LCG are of high quality.

As we were learning about parallel pseudo-random number generators, we learned that sometimes applications interact with pseudo-random number generators in unpredictable ways. Moreover, we learned that many applications require not just an absence of correlations in one dimension, but in higher dimensions as well. Given more time we would improve our project by including application-based tests such as the random walk test to examine the quality of the parallel pseudo-random number generators in a manner similar to that of the application in which it will be used.

R Code Appendix

```
#Gap test

gap.test <- function(u,lower =0,upper=1/2,echo=TRUE)
{
    #Compute gaps such as (g_i)=1 if (u_i) > max or (u_i)<min, 0 otherwise
    gap <-   (!((u<=upper)&(u>=lower)))*1
    n <- length(u)
    p <- upper - lower

    #Find index of zeros
    indexzero <- (gap == 1)*1:n
    indexzero <- indexzero[indexzero != 0]
    indexzero <- c(0, indexzero, n+1)
    lindzero <- length(indexzero)

    #Compute sizes of zero lengths
    lengthsize <- indexzero[2:lindzero] - indexzero[2:lindzero-1] -1
    lengthsize <- lengthsize[lengthsize != 0]
    maxlen <- max(lengthsize)
    maxlen <- max(maxlen, floor((log(10^(-1))- 2*log(1-p) - log(n)) / log(p)))

    #Compute observed and theoretical frequencies
    obsnum <- sapply(1:maxlen, function(t) sum(lengthsize == t))
    expnum <- (1-p)^2*p^(1:maxlen)*n

    #Compute chisquare statistic
    residu <- (obsnum - expnum)/sqrt(expnum)
    stat <- sum(residu^2)
    pvalue <- pchisq( stat, maxlen - 1, lower.tail = FALSE)
    options(digits=2)
    if( echo )
    {
        cat("\n\t\t\t Gap test\n")
        cat("\nchisq stat = ", stat, ", df = ",maxlen-1, ", p-value = ", pvalue, "\n", sep="")
        cat("\n\t\t (sample size : ",length(u),")\n\n", sep="")
        cat("length\tobserved freq\t\ttheoretical freq\n")
        for(i in 1:maxlen)
        cat(i,"\t\t\t", obsnum[i],"\t\t\t", expnum[i],"\n")
```

```
    }

    res <- list(statistic=stat, parameter=maxlen-1,
                p.value=pvalue, observed=obsnum,
                expected=expnum, residuals=residu)
    return(invisible(res))
}
```

```
#Frequency test

freq.test <- function(u, seq=0:15, echo=TRUE)
{

    #Compute integers such as min(seq) <= integernum[i] < max(seq)
    integernum <- floor(u*length(seq) + min(seq))

    #Observed numbers equal to seq[i]
    obsnum <- sapply(seq, function(x) sum(integernum == x))

    #Expected number equal to seq[i]
    expnum <- length(u)/length(seq)

    #Compute chisquare statistic
    residu <-  (obsnum - expnum)/sqrt(expnum)
    stat <- sum(residu^2)
    pvalue <- pchisq(stat, length(seq) - 1, lower.tail = FALSE)
    options(digits=2)
    if(echo)
    {
        cat("\n\t\t\t Frequency test\n")
        cat("\nchisq stat = ", stat, ", df = ",length(seq)-1, ", p-value = ", pvalue, "\n", sep="")
        cat("\n\t\t (sample size : ",length(u),")\n\n", sep="")
        cat("\tobserved number\t",obsnum,"\n")
        cat("\texpected number\t",expnum,"\n")
    }

    res <- list(statistic = stat, parameter=length(seq)-1,
                p.value = pvalue, observed = obsnum,
                expected = expnum, residuals =residu )
    return( invisible( res ) )
}
```

```
#Serial test

serial.test <- function(u , d=8, echo=TRUE)
{
    if(length(u)/d-as.integer(length(u)/d) > 0)
    stop("the length of 'u' must be a multiple of d")

   #Compute pairs in {0, ..., d-1}
    pair <- matrix(floor(u*d), length(u)/2, 2)

    #Compute (u_i)*d+(u_i)+1 in {0, ..., d^2-1}
    poly <- pair[,1]*d+pair[,2]
```

```
    #Compute numbers
    obsnum <- sapply(0:(d^2 -1), function(x) sum(poly == x))
    expnum <- length(u)/(2*d^2)

    #Compute chisquare statistic
    residu <- (obsnum - expnum)/sqrt(expnum)
    stat <- sum(residu^2)
    pvalue <- pchisq(stat, d^2-1, lower.tail=FALSE)
    options(digits=2)
    if( echo )
    {
        cat("\n\t\t\t Serial test\n")
        cat("\nchisq stat = ", stat, ", df = ",d^2-1, ", p-value = ", pvalue, "\n", sep="")
        cat("\n\t\t (sample size : ",length(u),")\n\n", sep="")
        cat("\tobserved number\t",obsnum,"\n")
        cat("\texpected number\t",expnum,"\n")
    }

    res <- list(statistic = stat, parameter=d^2-1,
                p.value=pvalue, observed=obsnum,
                expected=expnum, residuals=residu)
    return(invisible(res))
}
```

```
#Poker.test

poker.test <- function(u, nbcard=5,echo=TRUE)
{
    if(length(u)/nbcard-as.integer(length(u)/nbcard) > 0)
    stop("the length of 'u' must be a multiple of nbcard")

    #Number of lines
    nbl <- length(u)/nbcard

    #Compute "nbcard-hands" in {0, ..., k-1}
    hands <- matrix(as.integer(floor(u*nbcard)),nbl, nbcard)

    #Compute observed hands
    obshands <- .Call("doPokerTest", hands, nbl, nbcard)

    #Compute expected hands
    fact <- vector("numeric", nbcard+1)
    fact[1] <- 1
    for(i in 2:(nbcard+1))
    fact[i] <- fact[i-1]*(i-1)

    #Fact contains 0!,1!, 2! ... (2*nbcard)!
    ind <- 1:nbcard
    stirlingNum <- stirling(nbcard)[-1]
    exphands <- 1/nbcard^nbcard*fact[nbcard+1]/fact[nbcard-ind+1]*stirlingNum[ind]*nbl

    stat <- sum((obshands-exphands)^2/exphands)
    pvalue <- pchisq(stat, nbcard-1, lower.tail=FALSE)
```

```
    options(digits=2)
    if( echo )
    {
        cat("\n\t\t\t Poker test\n")
        cat("\nchisq stat = ", stat, ", df = ",nbcard-1, ", p-value = ", pvalue, "\n", sep="")
        cat("\n\t\t (sample size : ",length(u),")\n\n", sep="")
        cat("\tobserved number\t",obshands,"\n")
        cat("\texpected number\t",exphands,"\n")
    }

    res <- list(statistic=stat, parameter=nbcard-1,
                p.value=pvalue, observed=obshands,
                expected=exphands, residuals=(obshands-exphands)^2/exphands)
    return(invisible(res))
}
```

```
#Order test

order.test <- function(u, d=3,echo=TRUE)
{
    if(d > 8)
    stop("too long to compute this exponential cost problem.")
    if(d < 2)
    stop("wrong argument 'd'.")

    if(length(u)/d-as.integer(length(u)/d) > 0)
    stop("the length of 'u' must be a multiple of d")

    #Store u in a matrix
    if(is.vector(u))
    u <- matrix(u, length(u) / d, d)
    if(!is.matrix(u))
    stop("wrong argument u")

    #Compute observed numbers manually to save on computational time
    factOfD <- factorial(d)
    obsnum <- vector("numeric", length=factOfD)
    if(d == 2)
    {
        obsnum[1] <- sum(u[,1] < u[,2])
        obsnum[2] <- sum(u[,2] < u[,1])
    }
    if(d == 3)
    {
        obsnum[1] <- sum(u[,1] < u[,2] & u[,2] <u[,3])
        obsnum[2] <- sum(u[,1] < u[,3] & u[,3] <u[,2])
        obsnum[3] <- sum(u[,2] < u[,1] & u[,1] <u[,3])
        obsnum[4] <- sum(u[,2] < u[,3] & u[,3] <u[,1])
        obsnum[5] <- sum(u[,3] < u[,2] & u[,2] <u[,1])
        obsnum[6] <- sum(u[,3] < u[,1] & u[,1] <u[,2])
    }
    if(d == 4)
    {
```

```
    obsnum[1] <- sum(u[,1] < u[,2] & u[,2] <u[,3] & u[,3] <u[,4])
    obsnum[2] <- sum(u[,1] < u[,3] & u[,3] <u[,2] & u[,2] <u[,4])
    obsnum[3] <- sum(u[,2] < u[,1] & u[,1] <u[,3] & u[,3] <u[,4])
    obsnum[4] <- sum(u[,2] < u[,3] & u[,3] <u[,1] & u[,1] <u[,4])
    obsnum[5] <- sum(u[,3] < u[,2] & u[,2] <u[,1] & u[,1] <u[,4])
    obsnum[6] <- sum(u[,3] < u[,1] & u[,1] <u[,2] & u[,2] <u[,4])
    obsnum[7] <- sum(u[,1] < u[,2] & u[,2] <u[,4] & u[,4] <u[,3])
    obsnum[8] <- sum(u[,1] < u[,3] & u[,3] <u[,4] & u[,4] <u[,2])
    obsnum[9] <- sum(u[,2] < u[,1] & u[,1] <u[,4] & u[,4] <u[,3])
    obsnum[10] <- sum(u[,2] < u[,3] & u[,3] <u[,4] & u[,4] <u[,1])
    obsnum[11] <- sum(u[,3] < u[,2] & u[,2] <u[,4] & u[,4] <u[,1])
    obsnum[12] <- sum(u[,3] < u[,1] & u[,1] <u[,4] & u[,4] <u[,2])
    obsnum[13] <- sum(u[,1] < u[,4] & u[,4] <u[,2] & u[,2] <u[,3])
    obsnum[14] <- sum(u[,1] < u[,4] & u[,4] <u[,3] & u[,3] <u[,2])
    obsnum[15] <- sum(u[,2] < u[,4] & u[,4] <u[,1] & u[,1] <u[,3])
    obsnum[16] <- sum(u[,2] < u[,4] & u[,4] <u[,3] & u[,3] <u[,1])
    obsnum[17] <- sum(u[,3] < u[,4] & u[,4] <u[,2] & u[,2] <u[,1])
    obsnum[18] <- sum(u[,3] < u[,4] & u[,4] <u[,1] & u[,1] <u[,2])
    obsnum[19] <- sum(u[,4] < u[,1] & u[,1] <u[,2] & u[,2] <u[,3])
    obsnum[20] <- sum(u[,4] < u[,1] & u[,1] <u[,3] & u[,3] <u[,2])
    obsnum[21] <- sum(u[,4] < u[,2] & u[,2] <u[,1] & u[,1] <u[,3])
    obsnum[22] <- sum(u[,4] < u[,2] & u[,2] <u[,3] & u[,3] <u[,1])
    obsnum[23] <- sum(u[,4] < u[,3] & u[,3] <u[,2] & u[,2] <u[,1])
    obsnum[24] <- sum(u[,4] < u[,3] & u[,3] <u[,1] & u[,1] <u[,2])
}

if(d == 5)
{
    obsnum[1]<-sum(u[,1]<u[,2]&u[,2]<u[,3]&u[,3]<u[,4]&u[,4] <u[,5])
    obsnum[2]<-sum(u[,1]<u[,3]&u[,3]<u[,2]&u[,2]<u[,4]&u[,4] <u[,5])
    obsnum[3]<-sum(u[,2]<u[,1]&u[,1]<u[,3]&u[,3]<u[,4]&u[,4] <u[,5])
    obsnum[4]<-sum(u[,2]<u[,3]&u[,3]<u[,1]&u[,1]<u[,4]&u[,4] <u[,5])
    obsnum[5]<-sum(u[,3]<u[,2]&u[,2]<u[,1]&u[,1]<u[,4]&u[,4] <u[,5])
    obsnum[6]<-sum(u[,3]<u[,1]&u[,1]<u[,2]&u[,2]<u[,4]&u[,4] <u[,5])
    obsnum[7]<-sum(u[,1]<u[,2]&u[,2]<u[,4]&u[,4]<u[,3]&u[,3] <u[,5])
    obsnum[8]<-sum(u[,1]<u[,3]&u[,3]<u[,4]&u[,4]<u[,2]&u[,2] <u[,5])
    obsnum[9]<-sum(u[,2]<u[,1]&u[,1]<u[,4]&u[,4]<u[,3]&u[,3] <u[,5])
    obsnum[10]<-sum(u[,2]<u[,3]&u[,3]<u[,4]&u[,4]<u[,1]&u[,1] <u[,5])
    obsnum[11]<-sum(u[,3]<u[,2]&u[,2]<u[,4]&u[,4]<u[,1]&u[,1] <u[,5])
    obsnum[12]<-sum(u[,3]<u[,1]&u[,1]<u[,4]&u[,4]<u[,2]&u[,2] <u[,5])
    obsnum[13]<-sum(u[,1]<u[,4]&u[,4]<u[,2]&u[,2]<u[,3]&u[,3] <u[,5])
    obsnum[14]<-sum(u[,1]<u[,4]&u[,4]<u[,3]&u[,3]<u[,2]&u[,2] <u[,5])
    obsnum[15]<-sum(u[,2]<u[,4]&u[,4]<u[,1]&u[,1]<u[,3]&u[,3] <u[,5])
    obsnum[16]<-sum(u[,2]<u[,4]&u[,4]<u[,3]&u[,3]<u[,1]&u[,1] <u[,5])
    obsnum[17]<-sum(u[,3]<u[,4]&u[,4]<u[,2]&u[,2]<u[,1]&u[,1] <u[,5])
    obsnum[18]<-sum(u[,3]<u[,4]&u[,4]<u[,1]&u[,1]<u[,2]&u[,2] <u[,5])
    obsnum[19]<-sum(u[,4]<u[,1]&u[,1]<u[,2]&u[,2]<u[,3]&u[,3] <u[,5])
    obsnum[20]<-sum(u[,4]<u[,1]&u[,1]<u[,3]&u[,3]<u[,2]&u[,2] <u[,5])
    obsnum[21]<-sum(u[,4]<u[,2]&u[,2]<u[,1]&u[,1]<u[,3]&u[,3] <u[,5])
    obsnum[22]<-sum(u[,4]<u[,2]&u[,2]<u[,3]&u[,3]<u[,1]&u[,1] <u[,5])
    obsnum[23]<-sum(u[,4]<u[,3]&u[,3]<u[,2]&u[,2]<u[,1]&u[,1] <u[,5])
    obsnum[24]<-sum(u[,4]<u[,3]&u[,3]<u[,1]&u[,1]<u[,2]&u[,2] <u[,5])
    obsnum[25]<-sum(u[,1]<u[,2]&u[,2]<u[,3]&u[,3]<u[,5]&u[,5] <u[,4])
```

```
obsnum[26]<-sum(u[,1]<u[,3]&u[,3]<u[,2]&u[,2]<u[,5]&u[,5] <u[,4])
obsnum[27]<-sum(u[,2]<u[,1]&u[,1]<u[,3]&u[,3]<u[,5]&u[,5] <u[,4])
obsnum[28]<-sum(u[,2]<u[,3]&u[,3]<u[,1]&u[,1]<u[,5]&u[,5] <u[,4])
obsnum[29]<-sum(u[,3]<u[,2]&u[,2]<u[,1]&u[,1]<u[,5]&u[,5] <u[,4])
obsnum[30]<-sum(u[,3]<u[,1]&u[,1]<u[,2]&u[,2]<u[,5]&u[,5] <u[,4])
obsnum[31]<-sum(u[,1]<u[,2]&u[,2]<u[,4]&u[,4]<u[,5]&u[,5] <u[,3])
obsnum[32]<-sum(u[,1]<u[,3]&u[,3]<u[,4]&u[,4]<u[,5]&u[,5] <u[,2])
obsnum[33]<-sum(u[,2]<u[,1]&u[,1]<u[,4]&u[,4]<u[,5]&u[,5] <u[,3])
obsnum[34]<-sum(u[,2]<u[,3]&u[,3]<u[,4]&u[,4]<u[,5]&u[,5] <u[,1])
obsnum[35]<-sum(u[,3]<u[,2]&u[,2]<u[,4]&u[,4]<u[,5]&u[,5] <u[,1])
obsnum[36]<-sum(u[,3]<u[,1]&u[,1]<u[,4]&u[,4]<u[,5]&u[,5] <u[,2])
obsnum[37]<-sum(u[,1]<u[,4]&u[,4]<u[,2]&u[,2]<u[,5]&u[,5] <u[,3])
obsnum[38]<-sum(u[,1]<u[,4]&u[,4]<u[,3]&u[,3]<u[,5]&u[,5] <u[,2])
obsnum[39]<-sum(u[,2]<u[,4]&u[,4]<u[,1]&u[,1]<u[,5]&u[,5] <u[,3])
obsnum[40]<-sum(u[,2]<u[,4]&u[,4]<u[,3]&u[,3]<u[,5]&u[,5] <u[,1])
obsnum[41]<-sum(u[,3]<u[,4]&u[,4]<u[,2]&u[,2]<u[,5]&u[,5] <u[,1])
obsnum[42]<-sum(u[,3]<u[,4]&u[,4]<u[,1]&u[,1]<u[,5]&u[,5] <u[,2])
obsnum[43]<-sum(u[,4]<u[,1]&u[,1]<u[,2]&u[,2]<u[,5]&u[,5] <u[,3])
obsnum[44]<-sum(u[,4]<u[,1]&u[,1]<u[,3]&u[,3]<u[,5]&u[,5] <u[,2])
obsnum[45]<-sum(u[,4]<u[,2]&u[,2]<u[,1]&u[,1]<u[,5]&u[,5] <u[,3])
obsnum[46]<-sum(u[,4]<u[,2]&u[,2]<u[,3]&u[,3]<u[,5]&u[,5] <u[,1])
obsnum[47]<-sum(u[,4]<u[,3]&u[,3]<u[,2]&u[,2]<u[,5]&u[,5] <u[,1])
obsnum[48]<-sum(u[,4]<u[,3]&u[,3]<u[,1]&u[,1]<u[,5]&u[,5] <u[,2])
obsnum[49]<-sum(u[,1]<u[,2]&u[,2]<u[,5]&u[,5]<u[,3]&u[,3] <u[,4])
obsnum[50]<-sum(u[,1]<u[,3]&u[,3]<u[,5]&u[,5]<u[,2]&u[,2] <u[,4])
obsnum[51]<-sum(u[,2]<u[,1]&u[,1]<u[,5]&u[,5]<u[,3]&u[,3] <u[,4])
obsnum[52]<-sum(u[,2]<u[,3]&u[,3]<u[,5]&u[,5]<u[,1]&u[,1] <u[,4])
obsnum[53]<-sum(u[,3]<u[,2]&u[,2]<u[,5]&u[,5]<u[,1]&u[,1] <u[,4])
obsnum[54]<-sum(u[,3]<u[,1]&u[,1]<u[,5]&u[,5]<u[,2]&u[,2] <u[,4])
obsnum[55]<-sum(u[,1]<u[,2]&u[,2]<u[,5]&u[,5]<u[,4]&u[,4] <u[,3])
obsnum[56]<-sum(u[,1]<u[,3]&u[,3]<u[,5]&u[,5]<u[,4]&u[,4] <u[,2])
obsnum[57]<-sum(u[,2]<u[,1]&u[,1]<u[,5]&u[,5]<u[,4]&u[,4] <u[,3])
obsnum[58]<-sum(u[,2]<u[,3]&u[,3]<u[,5]&u[,5]<u[,4]&u[,4] <u[,1])
obsnum[59]<-sum(u[,3]<u[,2]&u[,2]<u[,5]&u[,5]<u[,4]&u[,4] <u[,1])
obsnum[60]<-sum(u[,3]<u[,1]&u[,1]<u[,5]&u[,5]<u[,4]&u[,4] <u[,2])
obsnum[61]<-sum(u[,1]<u[,4]&u[,4]<u[,5]&u[,5]<u[,2]&u[,2] <u[,3])
obsnum[62]<-sum(u[,1]<u[,4]&u[,4]<u[,5]&u[,5]<u[,3]&u[,3] <u[,2])
obsnum[63]<-sum(u[,2]<u[,4]&u[,4]<u[,5]&u[,5]<u[,1]&u[,1] <u[,3])
obsnum[64]<-sum(u[,2]<u[,4]&u[,4]<u[,5]&u[,5]<u[,3]&u[,3] <u[,1])
obsnum[65]<-sum(u[,3]<u[,4]&u[,4]<u[,5]&u[,5]<u[,2]&u[,2] <u[,1])
obsnum[66]<-sum(u[,3]<u[,4]&u[,4]<u[,5]&u[,5]<u[,1]&u[,1] <u[,2])
obsnum[67]<-sum(u[,4]<u[,1]&u[,1]<u[,5]&u[,5]<u[,2]&u[,2] <u[,3])
obsnum[68]<-sum(u[,4]<u[,1]&u[,1]<u[,5]&u[,5]<u[,3]&u[,3] <u[,2])
obsnum[69]<-sum(u[,4]<u[,2]&u[,2]<u[,5]&u[,5]<u[,1]&u[,1] <u[,3])
obsnum[70]<-sum(u[,4]<u[,2]&u[,2]<u[,5]&u[,5]<u[,3]&u[,3] <u[,1])
obsnum[71]<-sum(u[,4]<u[,3]&u[,3]<u[,5]&u[,5]<u[,2]&u[,2] <u[,1])
obsnum[72]<-sum(u[,4]<u[,3]&u[,3]<u[,5]&u[,5]<u[,1]&u[,1] <u[,2])
obsnum[73]<-sum(u[,1]<u[,5]&u[,5]<u[,2]&u[,2]<u[,3]&u[,3] <u[,4])
obsnum[74]<-sum(u[,1]<u[,5]&u[,5]<u[,3]&u[,3]<u[,2]&u[,2] <u[,4])
obsnum[75]<-sum(u[,2]<u[,5]&u[,5]<u[,1]&u[,1]<u[,3]&u[,3] <u[,4])
obsnum[76]<-sum(u[,2]<u[,5]&u[,5]<u[,3]&u[,3]<u[,1]&u[,1] <u[,4])
obsnum[77]<-sum(u[,3]<u[,5]&u[,5]<u[,2]&u[,2]<u[,1]&u[,1] <u[,4])
obsnum[78]<-sum(u[,3]<u[,5]&u[,5]<u[,1]&u[,1]<u[,2]&u[,2] <u[,4])
```

```
    obsnum[79]<-sum(u[,1]<u[,5]&u[,5]<u[,2]&u[,2]<u[,4]&u[,4] <u[,3])
    obsnum[80]<-sum(u[,1]<u[,5]&u[,5]<u[,3]&u[,3]<u[,4]&u[,4] <u[,2])
    obsnum[81]<-sum(u[,2]<u[,5]&u[,5]<u[,1]&u[,1]<u[,4]&u[,4] <u[,3])
    obsnum[82]<-sum(u[,2]<u[,5]&u[,5]<u[,3]&u[,3]<u[,4]&u[,4] <u[,1])
    obsnum[83]<-sum(u[,3]<u[,5]&u[,5]<u[,2]&u[,2]<u[,4]&u[,4] <u[,1])
    obsnum[84]<-sum(u[,3]<u[,5]&u[,5]<u[,1]&u[,1]<u[,4]&u[,4] <u[,2])
    obsnum[85]<-sum(u[,1]<u[,5]&u[,5]<u[,4]&u[,4]<u[,2]&u[,2] <u[,3])
    obsnum[86]<-sum(u[,1]<u[,5]&u[,5]<u[,4]&u[,4]<u[,3]&u[,3] <u[,2])
    obsnum[87]<-sum(u[,2]<u[,5]&u[,5]<u[,4]&u[,4]<u[,1]&u[,1] <u[,3])
    obsnum[88]<-sum(u[,2]<u[,5]&u[,5]<u[,4]&u[,4]<u[,3]&u[,3] <u[,1])
    obsnum[89]<-sum(u[,3]<u[,5]&u[,5]<u[,4]&u[,4]<u[,2]&u[,2] <u[,1])
    obsnum[90]<-sum(u[,3]<u[,5]&u[,5]<u[,4]&u[,4]<u[,1]&u[,1] <u[,2])
    obsnum[91]<-sum(u[,4]<u[,5]&u[,5]<u[,1]&u[,1]<u[,2]&u[,2] <u[,3])
    obsnum[92]<-sum(u[,4]<u[,5]&u[,5]<u[,1]&u[,1]<u[,3]&u[,3] <u[,2])
    obsnum[93]<-sum(u[,4]<u[,5]&u[,5]<u[,2]&u[,2]<u[,1]&u[,1] <u[,3])
    obsnum[94]<-sum(u[,4]<u[,5]&u[,5]<u[,2]&u[,2]<u[,3]&u[,3] <u[,1])
    obsnum[95]<-sum(u[,4]<u[,5]&u[,5]<u[,3]&u[,3]<u[,2]&u[,2] <u[,1])
    obsnum[96]<-sum(u[,4]<u[,5]&u[,5]<u[,3]&u[,3]<u[,1]&u[,1] <u[,2])
    obsnum[97]<-sum(u[,5]<u[,1]&u[,1]<u[,2]&u[,2]<u[,3]&u[,3] <u[,4])
    obsnum[98]<-sum(u[,5]<u[,1]&u[,1]<u[,3]&u[,3]<u[,2]&u[,2] <u[,4])
    obsnum[99]<-sum(u[,5]<u[,2]&u[,2]<u[,1]&u[,1]<u[,3]&u[,3] <u[,4])
    obsnum[100]<-sum(u[,5]<u[,2]&u[,2]<u[,3]&u[,3]<u[,1]&u[,1] <u[,4])
    obsnum[101]<-sum(u[,5]<u[,3]&u[,3]<u[,2]&u[,2]<u[,1]&u[,1] <u[,4])
    obsnum[102]<-sum(u[,5]<u[,3]&u[,3]<u[,1]&u[,1]<u[,2]&u[,2] <u[,4])
    obsnum[103]<-sum(u[,5]<u[,1]&u[,1]<u[,2]&u[,2]<u[,4]&u[,4] <u[,3])
    obsnum[104]<-sum(u[,5]<u[,1]&u[,1]<u[,3]&u[,3]<u[,4]&u[,4] <u[,2])
    obsnum[105]<-sum(u[,5]<u[,2]&u[,2]<u[,1]&u[,1]<u[,4]&u[,4] <u[,3])
    obsnum[106]<-sum(u[,5]<u[,2]&u[,2]<u[,3]&u[,3]<u[,4]&u[,4] <u[,1])
    obsnum[107]<-sum(u[,5]<u[,3]&u[,3]<u[,2]&u[,2]<u[,4]&u[,4] <u[,1])
    obsnum[108]<-sum(u[,5]<u[,3]&u[,3]<u[,1]&u[,1]<u[,4]&u[,4] <u[,2])
    obsnum[109]<-sum(u[,5]<u[,1]&u[,1]<u[,4]&u[,4]<u[,2]&u[,2] <u[,3])
    obsnum[110]<-sum(u[,5]<u[,1]&u[,1]<u[,4]&u[,4]<u[,3]&u[,3] <u[,2])
    obsnum[111]<-sum(u[,5]<u[,2]&u[,2]<u[,4]&u[,4]<u[,1]&u[,1] <u[,3])
    obsnum[112]<-sum(u[,5]<u[,2]&u[,2]<u[,4]&u[,4]<u[,3]&u[,3] <u[,1])
    obsnum[113]<-sum(u[,5]<u[,3]&u[,3]<u[,4]&u[,4]<u[,2]&u[,2] <u[,1])
    obsnum[114]<-sum(u[,5]<u[,3]&u[,3]<u[,4]&u[,4]<u[,1]&u[,1] <u[,2])
    obsnum[115]<-sum(u[,5]<u[,4]&u[,4]<u[,1]&u[,1]<u[,2]&u[,2] <u[,3])
    obsnum[116]<-sum(u[,5]<u[,4]&u[,4]<u[,1]&u[,1]<u[,3]&u[,3] <u[,2])
    obsnum[117]<-sum(u[,5]<u[,4]&u[,4]<u[,2]&u[,2]<u[,1]&u[,1] <u[,3])
    obsnum[118]<-sum(u[,5]<u[,4]&u[,4]<u[,2]&u[,2]<u[,3]&u[,3] <u[,1])
    obsnum[119]<-sum(u[,5]<u[,4]&u[,4]<u[,3]&u[,3]<u[,2]&u[,2] <u[,1])
    obsnum[120]<-sum(u[,5]<u[,4]&u[,4]<u[,3]&u[,3]<u[,1]&u[,1] <u[,2])
}

#If d is greather than 5, compute all the permutation recursively
if(d > 5 && d <= 8)
{
    mypermut <- permut(d)
    obsnum <- u[, mypermut[,1]] < u[, mypermut[,2]]

    #Compare columns of u
    for(i in 3:d)
    {
```

```
            obsnum <- obsnum & u[,mypermut[,i-1]] < u[,mypermut[,i]]
        }
        if(NCOL(obsnum) == 1)
        obsnum <- sum(obsnum)
        else
        obsnum <- colSums(obsnum)
    }

    #Compute expected numbers
    expnum <- length(u[,1])/factOfD

    #Compute chisquare statistic
    residu <- (obsnum - expnum)/sqrt(expnum)
    stat <- sum(residu^2)
    pvalue <- pchisq(stat,factOfD-1, lower.tail=FALSE)

    options(digits=2)
    if( echo )
    {
        cat("\n\t\t\t Order test\n")
        cat("\nchisq stat = ", stat, ", df = ",factOfD-1, ", p-value = ", pvalue, "\n", sep="")
        cat("\n\t\t (sample size : ",length(u),")\n\n", sep="")
        if(length(obsnum) <= 1000)
        cat("\tobserved number\t",obsnum,"\n")
        else
        cat("\tobserved number\t too many to be printed\n")
        cat("\texpected number\t",expnum,"\n")
    }

    res <- list(statistic=stat, parameter=factOfD-1,
                p.value=pvalue, observed=obsnum,
                expected=expnum, residuals=residu)
    return(invisible(res))
}
```