Generation and Analysis of the Random Numbers by GCC Compiler

SAKSHAM SRIVASTAVA

Department of Computer Engineering, Malaviya National Institute of Technology, Jaipur, India.

(Received: February 14, 2013; Accepted: February 27, 2013)

ABSTRACT

The random numbers have been generated by the GCC compiler and tested for their randomness. The code used by the library function of GCC compiler to generate random numbers is also presented. Chi-square test, Runs test below and above median, and Reverse arrangement test have been conducted to test the randomness. It has been shown that the Random numbers generated by GCC compiler have successfully passed these tests.

Key words: Random numbers, GCC compiler, Chi-square Test, rand() function, randomness.

INTRODUCTION

A random number is a number generated by certain process, whose outcome is unpredictable. Random numbers are required to be independent, so that there is no correlation between successive numbers. Random numbers are subject to intensive investigation due to their application in simulation of random numbers, statistical sampling, cryptography, computer programming, numerical analysis, decision making and recreation\textsuperscript{1-2}.

All the sources of random numbers behave in the same way, and some are better than others for different applications. The tests are usually conducted by empirical tests and theoretical tests. Empirical tests are conducted on the sequence generated by a Random Number Generator (RNG), and require no knowledge of how the RNG produces the sequence. Theoretical tests are better and require a knowledge of the structure of the RNG.

Computer-generated random numbers are referred as "pseudo random" numbers, on the other hand random numbers generated by physical processes are called "true random" numbers. Many algorithms have been developed to generate random numbers\textsuperscript{3-4} and also many tests to check their randomness. Commonly used pseudo random number generators are Linear Congruential generator, Blum Blum Shub and Lagged Fibonacci Generator. Most random number generators require an initial number called seed as a starting value. True random numbers are generated by flipping coins, rolling dice, keyboard latency, atmospheric noise picked up by a radio receiver, decay of radioactive materials and by the motions of lava lamps.
Among all the random number generators, the method used by the library function of GCC to generate random numbers is Linear Congruential generator.

**Linear Congruential Method for generation of Random numbers**

Linear congruential method was originally proposed by Lehmer in 1948, but reported by Knuth in 1951. It generates the sequence of random numbers by the following formula:

\[ Y_i = (a \times Y_{i-1} + c) \mod m \]

where \( Y_i \) is the \( i \)th term of the sequence and \( m \leq 0, 0 \leq a < m, 0 \leq c < m \) and \( 0 \leq Y_0 \) (seed) \(< m \)

For example if \( a=2, c=3, m=7, \) seed = 1, then the sequence generated by Linear Congruential generator is: 5, 6, 1, 5, 6, 1, .......

Here, the sequence gets repeated after an interval called the period of the sequence. Therefore constants \( a, c, m \) should be chosen carefully for getting a good generator.

**Random number generated by GCC compiler**

The GCC compiler has an inbuilt library function rand() for generating random numbers based on the non-additive feedback. The GCC compiler uses srand() function to initialise the random function by passing a seed as its argument. Mathematically, exact algorithm used by the rand() of GCC library to generate random numbers with a seed 's' is given below.

The sequence with \( i_{th} \) term denoted by \( R_i \) can be calculated as:

\[ R_i = R_{i-3} + R_{i-31} \mod 4294967296 \]  

for \( i \leq 34 \)

\[ R_i = R_{i-31} \mod 31 \leq 33 \)

\[ R_i = (16807 \times (\text{signed int}) R_{i-1}) \mod 2147483647 \]  

for \( 1 \leq i \leq 30 \)

where \( 2147483647 = 2^{31} - 1 \) and \( 4294967296 = 2^{32} \)

The first 343 terms of this sequence are ignored, and the first term obtained by the rand() function is equal to the 344th term of this sequence. If the \( i_{th} \) term obtained by rand() function be denoted by \( X_i \), then:

\[ X_i = R_{i+34} >> 1. \]

It is a 31-bit number and the least significant bit of \( R_{i+34} \) has been ignored. The multiplication by 16807 is done in a large signed integer type so that there is no overflow before the modulo operation. Further \( R_i \) is converted to a signed 32-bit value before the multiplication. Value obtained can only be negative in the case of \( i=1 \), if \( s \leq 2^{31} \). The modulo operation is mathematical and result obtained is in between 0 and 2147483646. Almost linear output sequence is obtained even after ignoring the least significant bit.

The C code for generating the random numbers same as that generated by rand() can be written as:

```c
#include <stdio.h>
#define MAX 1000
#define seed 1

int main() {
    int r[MAX], i;
    r[0] = seed;
    for (i=1; i<31; i++) {
        r[i] = (16807LL * r[i-1]) % 2147483647;
        if (r[i] < 0) {
            r[i] += 2147483647;
        }
    }
    for (i=31; i<34; i++) {
        r[i] = r[i-31];
    }
    for (i=34; i<344; i++) {
        r[i] = r[i-3] + r[i-31];
    }
    for (i=344; i<MAX; i++) {
        r[i] = r[i-31] + r[i-3];
        printf("%d\n", ((unsigned int)r[i]) >> 1);
    }
    return 0;
}
```

The terms \( X_i \) generated by the above algorithm using seed 1, are same as generated by the rand() function of GCC and are recorded in Table 1.
The various different sequences can be generated by giving different initialising seed values each time to rand() of GCC. The varying seed value can be obtained by using time() function present in the time.h header file of GCC library to initialise the random function. It returns the number of seconds passed from January 1, 1970 till the current time. The GCC Library function time() can be used with srand() as follows: srand(time(NULL))

The rand() can be used to obtain numbers in any certain range by a slight modification. E.g. numbers between 0 to m can be obtained by using modulo (m+1) with the rand(). The rand() can further be modified to obtain negative values by subtracting appropriate number after modulo operation. The following C code has been used to generate first 1000 terms with values between 0-999 by rand() of GCC compiler which will be used for conducting the randomness test, and some of the generated terms are mentioned in Table 2.

```c
#include<stdio.h>
define seed 1
define MAX 1000
int main()
{
    int i,r[MAX];
srand(seed);
    for(i=0;i<MAX;i++)
    {
        r[i] = rand()%1000;
        printf("%d
",r[i]);
    }
    return 0;
}
```

**Table 1: X<sub>i</sub> generated by the above C code**

<table>
<thead>
<tr>
<th>i</th>
<th>X&lt;sub&gt;i&lt;/sub&gt;</th>
<th>i</th>
<th>X&lt;sub&gt;i&lt;/sub&gt;</th>
<th>i</th>
<th>X&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1804289383</td>
<td>20</td>
<td>1303455736</td>
<td>30</td>
<td>468703135</td>
</tr>
<tr>
<td>2</td>
<td>846930886</td>
<td>21</td>
<td>35005211</td>
<td>50</td>
<td>2038664370</td>
</tr>
<tr>
<td>3</td>
<td>1681692777</td>
<td>22</td>
<td>521595368</td>
<td>75</td>
<td>1998898814</td>
</tr>
<tr>
<td>4</td>
<td>1714636915</td>
<td>23</td>
<td>294702567</td>
<td>100</td>
<td>1956297539</td>
</tr>
<tr>
<td>5</td>
<td>1957747793</td>
<td>24</td>
<td>1726956429</td>
<td>200</td>
<td>1784685329</td>
</tr>
<tr>
<td>6</td>
<td>424238335</td>
<td>25</td>
<td>336465782</td>
<td>300</td>
<td>1176911340</td>
</tr>
<tr>
<td>7</td>
<td>719885386</td>
<td>26</td>
<td>861021530</td>
<td>500</td>
<td>1081174232</td>
</tr>
<tr>
<td>8</td>
<td>1649760492</td>
<td>27</td>
<td>27872862</td>
<td>600</td>
<td>553160358</td>
</tr>
<tr>
<td>9</td>
<td>596516649</td>
<td>28</td>
<td>233665123</td>
<td>800</td>
<td>1981208324</td>
</tr>
<tr>
<td>10</td>
<td>1189641421</td>
<td>29</td>
<td>2145174067</td>
<td>1000</td>
<td>1143565421</td>
</tr>
</tbody>
</table>

**Table 2: T<sub>i</sub> generated by the above C code**

<table>
<thead>
<tr>
<th>i</th>
<th>T&lt;sub&gt;i&lt;/sub&gt;</th>
<th>i</th>
<th>T&lt;sub&gt;i&lt;/sub&gt;</th>
<th>i</th>
<th>T&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>383</td>
<td>15</td>
<td>763</td>
<td>100</td>
<td>539</td>
</tr>
<tr>
<td>2</td>
<td>886</td>
<td>20</td>
<td>736</td>
<td>200</td>
<td>529</td>
</tr>
<tr>
<td>3</td>
<td>777</td>
<td>25</td>
<td>782</td>
<td>300</td>
<td>340</td>
</tr>
<tr>
<td>4</td>
<td>915</td>
<td>30</td>
<td>135</td>
<td>400</td>
<td>868</td>
</tr>
<tr>
<td>5</td>
<td>793</td>
<td>40</td>
<td>42</td>
<td>500</td>
<td>232</td>
</tr>
<tr>
<td>6</td>
<td>335</td>
<td>50</td>
<td>370</td>
<td>600</td>
<td>358</td>
</tr>
<tr>
<td>7</td>
<td>386</td>
<td>60</td>
<td>281</td>
<td>700</td>
<td>775</td>
</tr>
<tr>
<td>8</td>
<td>492</td>
<td>70</td>
<td>857</td>
<td>800</td>
<td>324</td>
</tr>
<tr>
<td>9</td>
<td>649</td>
<td>80</td>
<td>750</td>
<td>900</td>
<td>587</td>
</tr>
<tr>
<td>10</td>
<td>421</td>
<td>90</td>
<td>399</td>
<td>1000</td>
<td>421</td>
</tr>
</tbody>
</table>
**Statistical Overview of the data**

The statistical overview of the data obtained from `rand()` of GCC compiler has been calculated by taking different seeds from 1 to 10. The parameter includes mean, median, maximum value, minimum value, probability of occurrence of even numbers, number of duplicate entries and non-occurring numbers in the data set. The summary statistics for the first 1000 random numbers obtained from `rand()` of GCC compiler are recorded in Table 3.

<table>
<thead>
<tr>
<th>Seed</th>
<th>Mean</th>
<th>Median</th>
<th>Max</th>
<th>Min</th>
<th>P(Even)</th>
<th>No of Duplications</th>
<th>No of non-Ocurring Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>499.495</td>
<td>497</td>
<td>999</td>
<td>0</td>
<td>.507</td>
<td>268</td>
<td>362</td>
</tr>
<tr>
<td>2</td>
<td>494.843</td>
<td>494.5</td>
<td>999</td>
<td>1</td>
<td>.471</td>
<td>257</td>
<td>367</td>
</tr>
<tr>
<td>3</td>
<td>504.442</td>
<td>496</td>
<td>998</td>
<td>2</td>
<td>.512</td>
<td>259</td>
<td>358</td>
</tr>
<tr>
<td>4</td>
<td>493.549</td>
<td>494</td>
<td>998</td>
<td>0</td>
<td>.511</td>
<td>258</td>
<td>370</td>
</tr>
<tr>
<td>5</td>
<td>489.946</td>
<td>488</td>
<td>999</td>
<td>0</td>
<td>.480</td>
<td>263</td>
<td>369</td>
</tr>
<tr>
<td>6</td>
<td>512.263</td>
<td>531.5</td>
<td>999</td>
<td>2</td>
<td>.495</td>
<td>266</td>
<td>371</td>
</tr>
<tr>
<td>7</td>
<td>500.201</td>
<td>505</td>
<td>999</td>
<td>1</td>
<td>.489</td>
<td>260</td>
<td>365</td>
</tr>
<tr>
<td>8</td>
<td>511.306</td>
<td>519.5</td>
<td>998</td>
<td>9</td>
<td>.506</td>
<td>266</td>
<td>390</td>
</tr>
<tr>
<td>9</td>
<td>508.331</td>
<td>510</td>
<td>998</td>
<td>1</td>
<td>.475</td>
<td>258</td>
<td>358</td>
</tr>
<tr>
<td>10</td>
<td>504.028</td>
<td>523.5</td>
<td>999</td>
<td>0</td>
<td>.476</td>
<td>270</td>
<td>372</td>
</tr>
</tbody>
</table>

**Analysis of data**

There are many techniques used for analysing randomness of the sequence. Out of these three have been taken for visual analysis of the randomness and are graphical in nature. Random numbers obtained by GCC compiler by taking seed one will be used for analysis.

**The Run Sequence Plot**

Run sequence plot is a graph of each observation against the random numbers in the sequence. Figure 1 is the run sequence plot of 1000 numbers obtained from `rand()` of GCC compiler which shows a random pattern. The plot fluctuates around 500, the expected mean of the numbers, and these fluctuations appear random.

![Run Sequence Plot](image-url)

Fig. 1: Run sequence plot of random numbers obtained by `rand()`
Lag plot

Lag plot is an interesting graph for detecting outliers. If there are chance outliers or significant outliers, this indicates that there may be something wrong with the generator. Figure 2 shows no outliers and the data points are spread evenly across the whole plain. This is a good indication of randomness.

![Lag Plot of random numbers](image)

Fig. 2: Lag plot of random numbers ($T_i$ against $T_{i-1}$)

The Histogram

The histogram plot is the count of observations that occur in each subgroup. The expected number of observations would be same in each category. In Figure 3 the random numbers are divided into 20 categories with 50 observations in each, which confirms property of uniformity.

![Histogram of rand()](image)

Fig. 3: Histogram of rand()
Theoretical test of randomness

The following three tests have been taken to test the randomness of the numbers:

Chi-square test

Chi-square test was given by Karl Pearson in 1900. It is a test of distributional accuracy. The chi-square test is a very common statistical test and is widely used in the analysis of random numbers\(^6\). The given number of observations is divided into \(n\) categories and the chi-square test \((\chi^2)\) is given by:

\[
\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}
\]

where \(O_i\) is the observed frequency and \(E_i\) is the expected frequency of \(i\)th observation.

The C code for determining chi-square test can be written as:

```c
#include<stdio.h>
define MAX 1000
define seed 1
int main()
{
    int i,j=0,r[1000],freq[1000]={0},count[10]={0};
    float median;
    srand(seed);
    for(i=0;i<MAX;i++)
    {
        r[i] = rand()%1000;
    }
    for(i=0;i<MAX;i++)
    {
        freq[r[i]]++;
    }
    printf("Oi are\n");
    for(i=0;i<1000;i++)
    {
        count[j] = count[j] + freq[i];
        if(%100 == 99) {
            printf("%d\n",count[j]);
            j++;
        }
    }
    printf("Oi-Ei  and their squares are \n");
    float v=0;
    for(i=0;i<=9;i++)
    {
        int y = count[i]-100;
        printf("%d \t %d\n",y,y*y);
        v= v+ (float)(y*y)/100;
    }
    printf("value of Chi-square variable is %f\n",v);
    return 0;
}
```

This test has been conducted by dividing the 1000 observations obtained in 10 equal categories having 100 observations in each. The expected frequency in each category of observation is 100. The results of the expected and observed frequencies are shown in Figure 4 in the form of a histogram. The value of \(\chi^2\) is calculated by the above formula and the calculations are summarised in Table 4.

![Fig. 4: Histogram of observed and expected frequency](image-url)
The chi-square value ($\chi^2$) comes out to be 5.04. The level of Significance ($\chi$) is taken as 0.05. The critical value ($\chi^2_{0.05,9} = 16.92$) is determined from the Chi-square table. If $\chi^2 > \chi^2_{0.05,9}$, then the given sequence fails the chi-square test. Whereas, if $\chi^2 \leq \chi^2_{0.05,9}$, then the given sequence passes the chi-square test of random numbers.

Since the $\chi^2$ value is less than the critical value so the null hypothesis is accepted at the 5% significance level. Therefore the numbers follow a uniform distribution which is one of the properties of random numbers.

### Runs test above and below median

The runs test is a common, non-parametric, distribution free test. Runs test is a good test to determine any fluctuating trends in the given sequence. This test is based on the number of runs of consecutive values above and below the median. A run is defined as a series of increasing or decreasing values. And the number of such increasing or decreasing patterns is defined as length of a run. In a random data set, the probability that the $(i+1)_{th}$ value is larger or smaller than the $i_{th}$ value follows a binomial distribution, which forms the basis of the runs test.

Those values of the given sequence which are greater than the median are assigned ‘a’, while those which are less are assigned ‘b’. Values which are equal to the median are ignored. Let the number of ‘a’s be defined as $N_1$ and ‘b’s as $N_2$. The first step would be to calculate the number of runs in the sequence, and the values of $N_1$ and $N_2$. This can be well illustrated as:

Sequence : 25 34 45 29 37 48
median = 35.5
Values assigned: a a b a b b
So the number of runs is 4 and value of $N_1 = 3$ and $N_2 = 3$.

The C code to find the number of runs and the values of $N_1$ (number of a’s) and $N_2$ (number of b’s) for the random numbers generated by the rand() function is written as:

```c
#include<stdio.h>
#define MAX 1000
#define seed 1
int main()  {
  int i,k,swap,counta=0,countb=0,j,r[MAX],s[MAX];
  char ch[MAX]={0};
  float median;
  srand(seed);
  for(i=0;i<MAX;i++) {
    r[i] = rand()%1000;
    s[i]=r[i];
  }
  for ( i = 0 ; i < ( MAX - 1 ) ; i++ ) {
    r[i] = rand()%1000;
    s[i]=r[i];
  }
  for ( i = 0 ; i < ( MAX - 1 ) ; i++ ) {
    k = i;
    for ( j = i + 1 ; j < MAX ; j++ ) {
      if ( s[k] > s[j] )
        k = j;
    }
    if ( k != i ) {
  }
```

### Table 4: Calculation of $\chi^2$ for the data

<table>
<thead>
<tr>
<th>Interval</th>
<th>$O_i$</th>
<th>$E_i$</th>
<th>$O_i-E_i$</th>
<th>$(O_i-E_i)^2$</th>
<th>$(O_i-E_i)^2 / E_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-99</td>
<td>103</td>
<td>100</td>
<td>3</td>
<td>9</td>
<td>.09</td>
</tr>
<tr>
<td>100-199</td>
<td>92</td>
<td>100</td>
<td>-8</td>
<td>64</td>
<td>.64</td>
</tr>
<tr>
<td>200-299</td>
<td>103</td>
<td>100</td>
<td>3</td>
<td>9</td>
<td>.09</td>
</tr>
<tr>
<td>300-399</td>
<td>114</td>
<td>100</td>
<td>14</td>
<td>196</td>
<td>1.96</td>
</tr>
<tr>
<td>400-499</td>
<td>92</td>
<td>100</td>
<td>-8</td>
<td>64</td>
<td>.64</td>
</tr>
<tr>
<td>500-599</td>
<td>103</td>
<td>100</td>
<td>3</td>
<td>9</td>
<td>.09</td>
</tr>
<tr>
<td>600-699</td>
<td>90</td>
<td>100</td>
<td>-10</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>700-799</td>
<td>101</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>.01</td>
</tr>
<tr>
<td>800-899</td>
<td>96</td>
<td>100</td>
<td>-4</td>
<td>16</td>
<td>.16</td>
</tr>
<tr>
<td>900-999</td>
<td>106</td>
<td>100</td>
<td>6</td>
<td>36</td>
<td>.36</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>0</td>
<td>5.04</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The chi-square value ($\chi^2$) comes out to be 5.04. The level of Significance ($\chi$) is taken as 0.05. The critical value ($\chi^2_{0.05,9} = 16.92$) is determined from the Chi-square table. If $\chi^2 > \chi^2_{0.05,9}$, then the given sequence fails the chi-square test. Whereas, if $\chi^2 \leq \chi^2_{0.05,9}$, then the given sequence passes the chi-square test of random numbers.
swap = s[i];
s[i] = s[k];
s[k] = swap;
}
}
if(MAX %2==0)
median = (s[MAX/2] + s[MAX/2 -1])/2.0;
else
median = s[(MAX-1)/2];
for(i=0;i<(MAX);i++) {
    if(r[i]<median) {
        ch[i] = 'a'; counta++;
    } else if(r[i]>median) {
        ch[i] = 'b'; countb++;
    }
}
int previous=ch[0],runs=0;
for(i=1;i<(MAX-1);i++) {
    if(ch[i] != 'a' & & ch[i] != 'b')
        continue;
    if((previous == 'a' & & ch[i]== 'b') ||
        (previous == 'b' & & ch[i] == 'a'))
        runs++;
    previous = ch[i];
}
runs++;
printf("No of runs is %d \n Value of N1 is %d \n Value of N2 is %d\n",runs,counta,countb);
return 0;
}

The mean (µ_u) of the distribution of µ is

\[ u_\mu = \frac{2 N_1 N_2}{N} + 1 \]

and the standard deviation of µ is

\[ \sigma_\mu^2 = \frac{2 N_1 N_2 (2 N_1 N_2 - N)}{N^2 (N - 1)} \]

where N = N_1 + N_2. And the test statistic Z is given as

\[ Z = \frac{(u \pm 0.5) - u_\mu}{\sigma_\mu} \]

This test has been carried out on 1000 random numbers and the result obtained are given in Table 5.

<table>
<thead>
<tr>
<th>Table 5: Test characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu )</td>
</tr>
<tr>
<td>491</td>
</tr>
</tbody>
</table>

The level of significance (α) is taken as 0.05 and critical value of \( Z_{u_0/2} \) is determined from the standard table. The sequence is only excepted as random number, if \(|Z| < Z_{u_0/2}\).

Critical value = \( Z_{u_0/2} = Z_{0.025} = 1.96 \)

Therefore for successfully passing the test:

\[ |Z| < Z_{u_0/2} \]
\[ -1.96 < Z < 1.96 \]
\[ -1.96 < Z < 1.96 \]

As the current value for Z lies between ±1.96 hence the null hypothesis is accepted at the 5% significant level.

Reverse arrangement test

Let us consider the number of observations to be N and \( i_u \) observation is denoted by \( T_i \), for \( i = 1,2,3...N \). Count the number of times \( T_i > T_j \) for each \( i < j \). Thus a reverse arrangement can be defined as the occurrence of a number smaller than \( T_i \) after \( T_i \) in the sequence. Total number of reverse arrangements is denoted by A. Then the variable \( h_{ij} \) is defined as:

\[ h_{ij} = \begin{cases} 
  1 & \text{if } x_i > x_j \\
  0 & \text{else}
\end{cases} \]
Then, 
\[
A_i = \sum_{j=i+1}^{N} h_{ij}
\]
and
\[
A = \sum_{i=1}^{N-1} A_i
\]

First, \(h_{ij}\) is calculated for each number and summing these \(h\)'s gives \(A_i\). Then \(A_i\) is calculated for each observation and the sum of these \(A_i\)'s gives \(A\), the total number of reverse arrangements. If the sequence of \(N\) observations are independent observations on the same random variable, then the number of reverse arrangements \(A\) is a random variable with a mean \((\mu_A)\)

\[
\mu_A = \frac{N(N-1)}{4}
\]

and a variance

\[
\sigma_A^2 = \frac{N(2N-1)(N-1)}{72}
\]

This test has been applied on first 100 terms obtained by \texttt{rand()} i.e. with \(N=100\). This test has been performed by using the C code as:

```c
#include<stdio.h>
define MAX 100
#define seed 1
int main()
{
    int i,h,j,r[MAX];
    srand(seed);
    for(i=0;i<(MAX);i++)
    {
        r[i] = rand()%1000;
    }
    h=0;
    for ( i = 0 ; i <(MAX-1) ; i++ )
    {
        for ( j = i + 1 ; j <MAX ; j++ )
        {
            if ( r[i] > r[j] )
            {
                h++;
            }
        }
    }
    printf("value of A is %d",h);
    return 0;
}
```

The value of \(A\) obtained after calculation is 2501. Level of significance is taken as 0.05, and the critical values \(A_{N;1-\alpha/2}\) and \(A_{N;\alpha/2}\) are obtained from the standard table. The given sequence is accepted as random numbers, only if the number of reverse arrangements \(A\) lies between the critical values. Hence for the sequence to pass the test:

- \(A_{N;1-\alpha/2} < 2501 < A_{N;\alpha/2}\)
- \(A_{100;0.975} < 2501 < A_{100;0.025}\)
- 2145 < 2501 < 2804

As the value for \(A\) in this experiment lies between 2145 and 2804 hence the null hypothesis is accepted at the 5% significance level. The mean and variation of the distribution are calculated by the above formula:

\[
\mu_A = 2475
\]
\[
\sigma_A^2 = 27362.5
\]

**CONCLUSIONS**

The random numbers generated by GCC compiler satisfies the test of uniformity, independence, summation and duplication. Uniformity is conformed by the histogram and the chi-square test. The property of summation and duplication are satisfied by the results of the summation statics. The properties of distribution and duplication are much more binding than those of uniformity and independence. This means that a set of numbers that display uniformity and independence are not random unless they have the properties of summation and duplication.
REFERENCES