International Journal of Advances in Engineering and Management (IJAEM)
Volume 2, Issue 2, pp: 248-251 www.ijaem.net ISSN: 2395-5252

Optimizing the library function rand() in C/C++ to increase the degree of randomness

Pushpam Kumar Sinha[1,*]
[1]*Department of Mechanical Engineering, NetajiSubhas Institute of Technology,Amhara, Bihta, Patna, India*

Date of Submission: 08-07-2020 Date of Acceptance: 23-07-2020

ABSTRACT

Almost all the Random Number Generators known are pseudo-Random Number Generators (pRNG) because the numbers generated by them are not truly random but based on recursive use of some mathematical formulae. The random number generator I am concerned with in this paper is the library function rand() employed most prominently in C/C++. What I do in this work is vary several constants appearing in the mathematical formulae of rand() between certain fixed limits and then measure the relative degree of randomness for each of the variation. I use the statistical technique of correlation coefficients, as explained in detail in my paper [1], to measure the relative degree of randomness. I find that only for one particular value of each of these constants, this particular value being different for different constants, the degree of randomness for this variation of rand() is the greatest. Henceforth, what I have achieved in this work is the optimization of rand(), to come up with the recursive mathematical formulae giving better random number generation than the conventional rand().

KEYWORDS: Optimizing problem, pseudo-Random Number Generator, Library function, Correlation Coefficient

I. INTRODUCTION

How do we know that a certain sequence of numbers that we have generated are random or not ? There are several tests of randomness available in literature to do that [2,3,4,5,6]. I have gone a step ahead in [1] and tell quantitatively how random a given pseudo-random number generator (pRNG) is, i.e. with my methodology to test randomness one is able to measure the relative degree of randomness when comparative analysis is needed between severalpRNGs. In order to appreciate the above statement I will write here briefly about one of the difficult to pass tests of randomness as given by Marsaglia and Tsang [6]; and though I have written in detail about my test of randomness in [1], I will introduce it here again for the sake of convenience.

Amongst the tests of randomness proposed by different people [2,3,4,5,6], I will be dealing here with ‘the gcd test’. In the gcd test, one makes a random choice of any two numbers, say a and b, from a sample of supposedly random non-negative integers. Following are the steps of operations to be performed with a and b

Step 1: Divide a by b, or b by a. Suppose one performs a/b, i.e. a is the dividend and b is the divisor

Step 2: The divisor b in step 1 is the new dividend, and the remainder of operation in step 1 is the new divisor

Step 3: Repeat step 2 till the remainder is zero

Let the number of iterations till last step be k, and the gcd (the divisor in the last step) be g. We take a large number of combinations of any two numbers from the sample of supposedly random non-negative integers in the range $\{1,2,...,2^{32}-1\}$ and perform the operations outlined in steps 1-3 above, and plot the distribution of k and g. These distributions, that of k and g, may be compared with the standard to determine whether a given sample is random or not. How do we fix the standard for the distributions of k and g ? If there are a large number of pRNGs yielding distributions very close to each other and to one single pRNG, that pRNG may be a taken as a good random number generator and its distributions be fixed as standard. However, there is other option also, that is, to compare the distributions of k and g with that of the true distributions for a truly random sample.

There are no known true distributions of k, but empirical study suggests that the k distribution for a truly random sample must be normal with mean = 18.5785 and standard deviation = 3.405. The true distribution of g is Probability[g = j]= c/j^2, with c = $6/\pi^2$. It is clear from the description of the gcd test that it does not quantify the relative degree of randomness, but that it qualitatively compares the distributions of k and g with the standard and or true and or empirical. Moreover, the test is also computationally expensive.

International Journal of Advances in Engineering and Management (IJAEM)
Volume 2, Issue 2, pp: 248-251 www.ijaem.net ISSN: 2395-5252

I use in this work the statistical technique of correlation coefficient to test randomness, which unlike others [2,3,4,5,6] is not an absolute statement of randomness but a relative one and that too quantified [1]. In this method I partition the sequence of random numbers a[i], (i=1,2,….,2n) into two data sets x[i] = a[i], (i=1,2,….,n) and y[i] = a[n+i], (i=1,2,….,n). From these two data sets, the correlation coefficient r is calculated as

$$r = \frac{\sum X\,Y}{\sqrt{\sum X^2 \sum Y^2}}$$

where $X = x[i] - \frac{\sum x[i]}{n}$, and $Y = y[i] - \frac{\sum y[i]}{n}$

The correlation coefficient r varies between -1 and +1. However, in my method I will consider the absolute value of r. The closer the absolute value of r is to zero, the more random a particular sequence of random numbers is, and thus this method is a quantitative measurement of the relative degree of randomness.

1.1 The optimizing problem of rand()

rand() is the library function in C/C++ used most frequently to generate random numbers, i.e. the integers in the
range 0 to 32767. One of the routines of rand() [7] is

```
unsigned long int next = 1;
int rand(void)
{
next = next * 1103515245 + 12345;
return (unsigned int)(next/65536) % 32768;
}
```

Note in the routine above the two constants 65536 and 32768. Seeing these two constants, I have a natural curiosity: Cannot I vary these two constants, i.e. play around with them to come up with new pRNGs different from rand(), and, if I do this which one amongst them including the conventional rand() is the most random ? To quench my curiosity, I modify the line

return (unsigned int)(next/65536) % 32768;

in above routine as

return (unsigned int)(next/den) % range;

where den is an integer which varies from 1 to 65536 in steps of 1 and range is also an integer which varies from 10001 to 32768 in steps of 1. This way I generate several new pRNGs, and amongst these I optimize for the one that has the least value of |r|.

A few of the other random number generators available in literature are [8, 9, 10].

II. THE COMPUTER PROGRAM

2.1 The Computer Program for variable range and fixed den

The computer program in C++ for the above mentioned optimization problem for rand() for den = 65536 and range varying from 10001 to 32768 in steps of 1 is

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;

int main()
{
unsigned long int next = 1, a[200001],
x[100001],y[100001];
unsigned long int range[25000];
inti,ii,j;
double sum1=0.0,sum2=0.0,sum3=0.0;
doublexavg,yavg;
double r[25000],smallest=1.0;
for(ii=1;ii<=22768;ii++)
{
range[ii]=10001+(ii-1);
next=1;
for(i=1;i<=200000;i++)
{
next = next * 1103515245 + 12345;
a[i]= (unsigned int)(next/65536) % range[ii];
}
for(j=1;j<=100000;j++)
{
x[j]=a[j];
y[j]=a[100000+j];
sum1=sum1+x[j];
sum2=sum2+y[j];
}
xavg=sum1/100000.0;
yavg=sum2/100000.0;
sum1=0.0;
sum2=0.0;
sum3=0.0;
for(j=1;j<=100000;j++)
{
sum1 = sum1+(x[j]-xavg)*(y[j]-yavg);
sum2 = sum2+pow(x[j]-xavg,2);
sum3 = sum3+pow(y[j]-yavg,2);
}
r[ii] = fabs(sum1/sqrt(sum2*sum3));
cout<<endl<<"  r["<<ii<<"] = "<<r[ii];
sum1 = 0.0;
sum2 = 0.0;
sum3 = 0.0;
}
for(ii=1;ii<=22768;ii++)
{
```

```
if(r[ii]<smallest) smallest = r[ii];
}
cout<<endl<<" smallest="<<smallest;
for(ii=1;ii<=22768;ii++)
{
if(smallest == r[ii])cout<<endl<<" ii="<<ii<<" range="<<range[ii];
}
return 0;
}
```

2.2 The Computer Program for variable den and fixed range

Note that the variation of den from 1 to 65536 in steps of 1 has been partitioned into two (because of memory limitation to store large sized arrays), variation from 1 to 40000 in steps of 1 and that from 40001 to 65536 in steps of 1. The computer program in C++ for the above mentioned optimization problem for rand() for range = 32768 and den varying from 40001 to 65536 in steps of 1 is

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;

int main()
{
unsigned long int next = 1, a[200001], x[100001],y[100001];
unsigned long int denominator[30000];
inti,ii,j;
double sum1=0.0,sum2=0.0,sum3=0.0;
doublexavg,yavg;
double r[30000],smallest=1.0;
for(ii=1;ii<=25536;ii++)
{
denominator[ii]=40000+ii;
next=1;
for(i=1;i<=200000;i++)
{
next = next * 1103515245 + 12345;
a[i]= (unsigned int)(next/denominator[ii]) % 32768;
}
for(j=1;j<=100000;j++)
{
x[j]=a[j];
y[j]=a[100000+j];
sum1=sum1+x[j];
sum2=sum2+y[j];
}
xavg=sum1/100000.0;
yavg=sum2/100000.0;
sum1=0.0;
sum2=0.0;
sum3=0.0;
for(j=1;j<=100000;j++)
{
sum1 = sum1+(x[j]-xavg)*(y[j]-yavg);
sum2 = sum2+pow(x[j]-xavg,2);
sum3 = sum3+pow(y[j]-yavg,2);
}
r[ii] = fabs(sum1/sqrt(sum2*sum3));
cout<<endl<<" r["<<ii<<"] = "<<r[ii];
sum1 = 0.0;
sum2 = 0.0;
sum3 = 0.0;
}
for(ii=1;ii<=25536;ii++)
{
if(r[ii]<smallest) smallest = r[ii];
}
cout<<endl<<" smallest="<<smallest;
for(ii=1;ii<=25536;ii++)
{
if(smallest == r[ii]) cout<<endl<<" denominator="<<denominator[ii];
}
return 0;
}
```

III. RESULTS AND CONCLUSION

All the results are for sequence of generation of 200000 random numbers.

The result for computer program in section 2.1 is optimized range = 31700, and the value of |r| for this range is $2.49623x10^{-8}$. For a computer program similar to that in section 2.2 but with den varying from1 to 40000 in steps of 1, the result is optimized den = 20193, and the value of |r| for this den is $1.6376x10^{-8}$. The result for computer program in section 2.2 is optimized den = 56928, and the value of |r| for this den is $1.19703x10^{-7}$. For a computer program similar to that in section 2.2 but with den varying from 1 to 40000 in steps of 1 and a previously found optimized range of 31700, the result is optimized den = 20868, and the value of |r| for this den is $1.64894x10^{-7}$. For a computer program similar to that in section 2.2 but with a previously found optimized range of 31700, the result is optimized den = 65536, and the value of |r| for this den is $2.49623x10^{-8}$. For a computer program similar to that in section 2.1 but with a previously found optimized den of 20193, the result is optimized range = 32768, and the value of |r| for this range is $1.6376x10^{-8}$.

From above results I conclude that the randomness is the greatest for den = 20193 and range = 32768. Henceforth, the routine for the library function rand() in C/C++ gets modified (to yield the sequence of numbers with the greatest randomness) as

```
unsigned long int next = 1;
int rand(void)
{
next = next * 1103515245 + 12345;
return (unsigned int)(next/20193) % 32768;
}
```

REFERENCES

[1]. Sinha PK, Sinha S. The better pseudo-random number generator derived from the library function rand() in C/C++. I.J. Mathematical Sciences and Computing 2019; 4:13-23.

[2]. Knuth Donald E. The Art of Computer Programming. Volume II. 3rd ed. Reading Mass.: Addison Wesley; 1998

[3]. MacLaren D,Marsaglia G. Uniform random number generators. Journ. Assoc. for Computing Machinery 1965; 12: 83–89.

[4]. Marsaglia G. A current view of random number generators. Keynote Address. Statistics and Computer Science: XVI Symposium on the Interface. Atlanta Proceedings 1985. Elsevier.

[5]. The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness, produced at Florida State University under a grant from The National Science Foundation 1985. Access available at www.stat.fsu.edu/pub/diehard.

[6]. Marsaglia G, Tsang WW. Some difficult-to-pass tests of randomness. Journal of Statistical Software 2002; Vol 7: Issue 3.

[7]. Kernighan Brian W, Ritchie Dennis M. The C Programming Language. 2nd ed. New Delhi: Pearson-Prentice Hall; 1988.

[8]. Park SK, Miller KW. Commun, ACM 1988; **31**: 1192-1201

[9]. Press WH,Teukolsky SA. Portable random number generators. Comput. Phys. 1992; **6**: 521-524

[10]. Marsaglia G,Zaman A. Some very-long-period portable random number generators. Computers in Physics 1995; 8: 117–121.

International Journal of Advances in Engineering and Management
ISSN: 2395-5252

IJAEM

Volume: 02 Issue: 01 DOI: 10.35629/5252

www.ijaem.net
Email id: ijaem.paper@gmail.com