Research Journal of Mathematical and Statistical Sciences ______________ISSN 2320-6047
Vol. **7(2),** 27-29, May **(2019)** *Res. J. Mathematical and Statistical Sci.*

Short Communication

An efficient method of generation of a random sample using random numbers significantly less than the sample size

Rushali Gupta[1] and Soubhik Chakraborty[2*]
[1]Department of Computer Science and Engineering, BIT Mesra, Ranchi-835215, India
[2]Department of Mathematics, BIT Mesra, Ranchi-835215, India
soubhikc@yahoo.co.in

Available online at: www.iscamaths.com , www.isca.in , www.isca.me
Received 8th December 2018, revised 12th April 2019, accepted 4th May 2019

Abstract

Given that the pseudo-random numbers generated by the computer have a cycle; it is wise not to lose random numbers in simulation studies. For drawing a random sample of size n from a population of size N (n<=N), the existing sampling algorithms require n pseudo-random numbers. If N is large, accordingly n should also be large for better representation of the population. Since most simulation studies require at least 500 samples, we would need 500xn pseudo random numbers which can lead to cycle break. We are therefore motivated to develop an efficient sampling algorithm which generates the desired sample using random numbers significantly less than the sample size. Our algorithm has the facility that a single pseudo-random number can generate the sample of size 60 for a population of size 100000 using a python code. We would of course need more than one pseudo-random number if the sample size exceeds 60 for this population.

Keywords: Sample, population, pseudo-random numbers, algorithm, statistical computing.

Introduction

The generation of pseudo random numbers is a significant and important task in scientific computing. It extends in a wide range of applications in several domains like cryptography, computer games, sampling theory, computer simulation, etc. Random number generation is an important aspect in statistical computing. An important application of random number lies in simulation techniques, where multiple iterations are required to generate random samples. In many cases, the basic process of generating random numbers is deterministic. The techniques to generate a sample of size n require n pseudo-random numbers. Our work focuses on reducing this requirement of n pseudo-random numbers substantially. We will use only a few pseudo random numbers to generate the desired random sample of size n.

Literature review and motivation

Given that we have random samples from the standard uniform distribution, the random samples from other distribution may be obtained by transformations. Some of the techniques of random number generation are congruential methods and feedback shift register methods.

The congruential method was first proposed by Lehmer[1]. He introduced the idea of multiplicative-congruential. A useful source of pseudorandom integers is a linear congruential sequence. This can take many forms.Another idea that sparked the concept was that of feedback shift registers, which was suggested by Tausworthe[2]. It takes successive positional values (mostly bits), as residues congruent to a linear combination.
Allard, Dobell and Hull[3] have given a mixed congruential random number generators for decimal machines.

Pseudo-Random number generator (PRNG) is an algorithm that is used to generate many random numbers. It starts from an arbitrary state called "seed", and generate numbers which are efficient and deterministic.

Monte-Carlo methods, a combination of probability theory and sampling technique, are extensively used in simulation studies. Their main application domains are optimization, numerical integration and drawing variates from probability distribution.

Reservoir sampling is a family of randomized algorithms for randomly choosing a sample of k items from a list S containing n items, where n is either a very large or unknown number.

Given that the pseudo-random numbers generated by the computer have a cycle (once the cycle breaks, the numbers repeat themselves in the same sequence which can seriously violate the simulation requirements); it is wise not to lose random numbers in simulation studies. For drawing a random sample of size n from a population of size N (n<=N), the existing sampling algorithms require n pseudo-random numbers. For a formal discussion on such sampling algorithms, see Kennedy and Gentle[4]. Further literature on statistical computing can be found in Kundu and Basu[5] and Gentle[6]. A recent book on random numbers and computers is by Kneusel[7]. Other useful

books on statistical computing are by Givens[8], Rizzo[9], Martinez and Martinez[10], and Sawitzki[11]. The edited volume by Dirschedl, and Ostermann[12] gives a nice collection of papers on the topic.
If N is large, accordingly n should also be large for better representation of the population. Since most simulation studies require at least 500 samples, we would need 500xn pseudo random numbers which can lead to cycle break. We are therefore motivated to develop an efficient sampling algorithm which generates the desired sample using random numbers significantly less than the sample size. Our algorithm has the facility that a single pseudo-random number can generate the sample of size 60 for a population of size 100000 using a python code. We would of course need more than one pseudo-random number if the sample size exceeds 60 for this population.

Our contribution: We have designed an algorithm which is capable of generating up to 300 digits in a random sequence irrespective of any parameter. The algorithm has been implemented in Python language. We have used Rand () function to generate random digits after the decimal point. The complete methodology is explained in the following steps: i. Input: Population size N and sample size n, ii. Output: A random sample of size n.

Algorithm efficient sampling technique: i. Using R and () function, generate a continuous uniform variate (*a*) in the range of (0,1). ii. Decide the no. of digits (group length) to be taken into consideration which will depend upon the population size N. For example, if N=100, group length = 2. iii. Since we require a random sample size of n, we will require n sets of decimal digits in the uniform variate, each set representing one sampling unit (see the example 1). For N =100, each set will be a doublet. iv. We will get each sampling unit by extracting the digits up to group length to the immediate left side of the decimal point in *a* by repeatedly multiplying *a* with 10^*group_length.* v. We will store the sampling units in a list. vi. To check for repetition, we can scan through the list and if the sampling unit has already been stored, in the list, it will not be stored again.

Example-1: Suppose the population size is 100 and the sample size is 30. That is, we agree to sample about one third of the population. Now we generate a uniform variate with 60 digits after the decimal place. The reason is that we are considering sets of two digits each sequentially to represent the sampling units. That is, the digits 00 to 99 will represent the 100 population units. Every doublet after the decimal will give us the corresponding sampling unit. For example, if our uniform variate takes the value say 0.713026640801….. then the sampling units are 71, 30, 26, 64,08, 01 etc. In the algorithm Efficient Sampling Technique, this will be achieved by repeatedly multiplying the uniform variate by 100 and taking two digits sequentially to the immediate left of the decimal as our sampling unit. These sampling units are to be kept in a list ensuring that there is no repetition in the list items. Since we have generated the uniform variate upto 60 places after the decimal, and each pair (doublet) gives one sampling unit, we obtain the desired random sample of size 30. For higher population size, we may require a triplet or, in general, an m-tuple (x_1, x_2, ….x_m) of the digits 0, 1, 2…9 (this m is the group_length) after the decimal expansion to represent the sampling unit and there has to be n such m-tuples to generate the desired random sample of size n. The value of m in the m-tuple will depend on the population size. In our example, m=2 will suffice as we took the population size to be 100. If the population size is 1000, the digits 000 to 999 will represent the 1000 population units and so we fix m=3. Now if the sample size is say 300, we would need to generate 300x3=900 digits after the decimal place. However, as our python code generates upto 300 digits, so we would require generation of 3 uniform variates using this code. Even then, as 3 <<300, we are done!

Python Code
```
frombuiltinsimportint, str
fromnumpy.polynomial.tests.test_classesimport random
fromtest.test_pdbimportdo_nothing
fromdecimal import *
importsys

#N is population size and n is sample size(Input data)
N=100000
n=600
print("population size ",N)
print("sample size ",n)
#find no of digits in N
group_length=len(str(abs(N)))-1
print("group length ",group_length)
#generation of a random number
r=random()
#creation of a list to store multiple random numbers
l=list()
a=r
print("Generated Random number ",a)
x=n

while(x>0):
   x=x-1
#shifting "group_length" digits to the left of the decimal point
   a=a*(10**group_length)
try:
if(a>0):
#extracting each number with "group_length" digits
num=int(a)%(10**group_length)
#print(num)
ifnumnotin l:
#adding the extracted number to the list
l.append(num)
else:
do_nothing
else:
break
```

```
except:
#print (sys.exc_info()[0])
do_nothing()

print("list of random numbers generated")
print(l)
print('size of list')
print(l.__len__())
```

Experimental results

Table-1 gives the output.

Table-1: Experimental results.

Population size	100000
Sample size	600
Group Length	5
Generated random number	0.028083975896380……..
List of 5 tuples generated	[02808, 39758, 96380, 77952, 99008, 85440, 83264, 20992, 89408, 57536, 98496, 25184, 17984, 8864, 02560, 56320, 51776, 13216, 74656, 03520, 30976, 61088, 05088, 39232, 62656, 92960, 39616, 89024, 6656, 57984, 13760, 37952, 56448, 60192, 9856, 41472, 56480, 50560, 17344, 34048, 82464, 97728, 52576, 77216, 81120, 20480, 50944, 12480, 16128, 84448, 05440, 23488, 80896, 44448, 19872, 42720, 39360, 02880, 76064, 55136, 01024]
Size of the list	61

We are able to get around 300 digits after decimal place. So depending upon the population size N (here 100000), we can generate about 300/5 = 60 sampling units (actually 61) using only one uniform variate. This means, as our random sample has size 600, so we would require 10 uniform variates with decimal expansion upto 300 digits in each, each such variate yielding about 60 (actually 61) sampling units. As 10 << 600, we claim having developed an efficient method of generation of a random sample using random numbers significantly less than the sample size.

One advantage of using this method is that the chance of a number occurring in a periodic manner will be very less.

Discussion, limitation and scope for further improvement: In the general case if our sample size n is less than or equal to a x 60 where a is some positive integer, we would need a uniform U{0, 1} variates with the present python code to generate the desired sample. As $a << a$ x 60, we claim to have developed an efficient method of generation of a random sample using random numbers significantly less than the sample size.

Further enhancement can be made on how to extract more values in decimal precision, so that we can achieve the target of generating larger samples using even fewer random numbers. For this purpose, we are exploring other softwares such as Mathematical.

Conclusion

We have developed an efficient method of generation of a random sample using random numbers significantly less than the sample size. Our python code successfully generates a sample of size n from a large population of size N using only a few (n/60) pseudo-random numbers.

References

1. Lehmer D.H. (1951). Mathematical methods in large-scale computing units. *Annu. Comput. Lab. Harvard Univ*., 26, 141-146.
2. Tausworthe R.C. (1965). Random Numbers Generated by Linear Recurrence Modulo Two. *Math.Comp*., 19, 201-209.
3. Allard J.L., Dobell A.R. and Hull T.E. (1963). Mixed congruential random number generators for decimal machines. *Journal of the ACM (JACM)*, 10(2), 131-141.
4. Kennedy Jr. W.J. and Gentle J.E. (1980). Statistical Computing. Marcel Dekker Inc, 33.
5. Kundu D. and Basu A. (2004). Statistical Computing: Existing Methods and Recent Development. *Alpha Science International Ltd.*
6. Gentle J.E. (2009). Computational Statistics. Springer.
7. Kneusel R. (2018). Random numbers and computers. Springer Publishing Company, Incorporated.
8. Givens G.H. (2005). Computational Statistics. Wiley Interscience.
9. Rizzo M.L. (2007). Statistical Computing with R. Chapman and Hall.
10. Martinez W.L. and Martinez A.R. (2015). Computational Statistics Handbook with MATLAB. Chapman and Hall.
11. Sawitzki G. (2009). Computational Statistics: an introduction to R. Chapman and Hall.
12. Dirschedl P. and Ostermann R. (1994). Computational Statistics, Papers Collected on the Occasion of the 25th Conference on Statistical Computing at Schloß Reisensburg. Springer.