3 Crisp Random Numbers and Vectors

3.1 Introduction

In this chapter we first discuss different ways to generate sequences of "random" numbers in some interval $[a, b]$. Usually the random numbers are first produced in $[0, 1]$ and then we perform a linear transformation to get them into $[a, b]$. Next we consider making sequences of random non-negative integers. We wish to produce sequences of random vectors $v = (x_1, ..., x_n)$ where the x_i are real numbers, and the randomness here means that the v will uniformly fill the space $[a, b]^n$. These random vectors will be used in the next chapter to generate sequences of random fuzzy numbers.

Subsequently, vectors of so-generated random fuzzy numbers are used for streams to feed fuzzy Monte Carlo optimization. As is shown in Chapter 4, with a 5-tuple we can generate a fuzzy number with quadratic membership functions. In some cases we evaluate using a vector of two or three fuzzy numbers generated from 5-tuples. In Chapters 6 and 9, we generate pairs of fuzzy numbers from Sobol quasi-random 10-tuples. In Chapters 7 and 8, vectors of three fuzzy numbers generated from Sobol 15-tuples are used. Other applications are in Chapters 10-16.

3.2 Random Numbers

We could have chosen to generate fuzzy numbers whenever they are needed; however, we wish to study the crisp numbers from which they are made, and we wish to study fuzzy numbers generated in various ways from those streams of crisp numbers. We expand upon a computer program from [2] to create streams of crisp numbers for which we simultaneously evaluate randomness. Our application from [2], `RNGenerator`, has several new features noted below. `RNGenerator` may be linked with any of several random number (RN) generator subroutines. The ones which we used were:

1. True Random: A million 8-bit (in binary notation) true random numbers were downloaded from `http://www.random.org`. This routine supplies one

J.J. Buckley et al.: Monte Carlo Meth. in Fuzzy Optimization, STUDFUZZ 222, pp. 29–34, 2008.
springerlink.com

16-bit true random integer (concatenate two 8-bit bytes), sequentially from that list, with each call.

2. Pseudo-Random: This routine supplies one 16-bit pseudo-random integer. From the C library's `rand()` function (Visual C++) with each call, we first obtain a pseudo-random integer in the interval [0,32767]. We multiply that value times 2 to create an even integer in [0,65534]. Since we will be scaling further to [0,1) for a χ^2 test, having even pseudo-random integers is not a concern.
3. Quasi-Random: Several quasi-random number routines from Burkardt [12] were used as the bases for quasi-random integer generators (Section 7.7, "Quasi-Random Sequences," from [10] provides background to Sobol sequences).
   The routines are designed to create n-tuples of crisp 16-bit integers, where n is user specified. To make their use compatible with the other random number generators, our generators release integers one at a time with each call. We are particularly interested in Sobol quasi-random integers because of our prior work ([1],[2]), and because Sobol sequences are reasonably well known and we have used them with MATLAB [9].

3.2.1 Quasi-random Sequences

Quasi-random numbers are also known as Low Discrepancy Points (LDP) or low discrepancy sequences. They are called quasi-random because they possess many attributes of random numbers, but they are truly not random. Rather they are designed to be less random and more uniformly distributed than Linear Congruential Generated (LCG) pseudo-random numbers. Their other name, Low Discrepancy Points, may be more appropriate though less catchy. The following excerpt from [7] is instructive to the goal of LDPs:

> [Begin] with a unit hypercube that is, a cube of more than three dimensions. Each edge of the cube has a length of 1 unit, so its volume is 1. Lets assume a large number of points are to be distributed within the cube. How can these points be distributed in such a way that, if any volume in the cube is selected, the proportion of the points within the volume is as close to the volume itself? . . . Points that provide, on average, a close fit between the volume and proportion numbers provide a *low discrepancy* thus, their name.

Many quasi-random number algorithms have been designed. The Van der Corput Sequence (1935) [11] generates LDPs in just one dimension [6]. Others have since been designed to provide LDPs in higher dimensions. Some of the best known are Halton (1960), Hammersley (1960), Sobol (1967), Faure (1980), and Niederreiter (1987) [11] Because quasi-random numbers provide more uniform coverage to a space than pseudo-random numbers, they are "at the forefront of financial mathematics" [8]. Algorithms for them are available from various sources including the Association for Computing Machinery (ACM).

Table 3.1. Random Number Generator χ^2 Tests, N=500,000, bins=10, 9 df

type	tuples	χ^2	Min	Max	Equal Pairs
Pseudo	1	9.793000	0	65534	10
True	1	7.073160	0	65535	6
Faure	2	0.007280	0	65534	829
Halton	2	0.011480	0	65534	10
Neiderreiter	2	0.007520	0	65534	864
Sobol	2	0.007600	0	65534	865
Faure	3	0.020880	0	65534	352
Halton	3	0.017680	0	65534	10
Neiderreiter	3	0.007400	0	65534	529
Sobol	3	0.006840	0	65534	530
Faure	5	0.082480	0	65534	339
Halton	5	0.017280	0	65534	10
Neiderreiter	5	0.003800	0	65534	332
Sobol	5	0.006520	0	65534	349
Faure	6	0.060400	0	65533	0
Halton	6	0.031800	0	65534	14
Neiderreiter	6	0.009360	0	65534	332
Sobol	6	0.014720	0	65534	411
Faure	9	0.700920	0	65531	0
Halton	9	0.057960	0	65533	11
Neiderreiter	9	0.007440	0	65534	356
Sobol	9	0.021720	0	65534	423
Faure	10	0.456000	0	65531	0
Halton	10	0.067400	0	65533	11
Neiderreiter	10	0.007840	0	65534	365
Sobol	10	0.013440	0	65534	411
Faure	15	4.328640	0	65534	0
Halton	15	0.203560	0	65533	13
Neiderreiter	15	0.020840	0	65534	411
Sobol	15	0.039640	0	65533	470

3.2.2 Random Number Generator

`RNGenerator` does statistics on the stream of RNs it generates. The 16-bit integers are scaled to [0,1) by division by 65536. A chi-square test is done for 10 bins (9 degrees of freedom) on 500,000 random numbers generated by each method. Though we do not use many of the streams later in this book, we provide our findings for comparison. In Table 3.1: (1) "type" is the type of generator used for the stream; (2) "tuple" is the number of integers the generator creates at a time; (3) "χ^2" is the value of the chi-square statistic; (4) "Min" is the smallest random number produced; (5) "Max" is the value of the largest random number generated; and (6) "Equal Pairs" means that two consecutively generated random numbers were equal.

The chi-square test was the standard randomness test applied to sequences of real numbers. The null hypothesis is H_0 that the sequence is random and the alternate hypothesis is H_1 that the sequence is not random. The significance level of the test was $\gamma = 0.05$. We place the random numbers into 10 equally spaced bins where, assuming H_0 is true, the expected number in each bin would be $500,000/10$. The critical value is $\chi^2 = 16.9190$ for 9 degrees of freedom. So the true random numbers and the pseudo-random numbers pass the randomness test (do not reject H_0). Moreover, the quasi-random numbers also pass the randomness test (do not reject H_0). The use of the quasi-random numbers will be explained in Section 3.4.

3.3 Random Non-negative Integers

Suppose we want a random sequence $z_1, z_2, z_3, ...$ of non-negative integers in some interval $[a, b]$, $a \geq 0$. We can use a pseudo-random number generator to first get a random sequence $x_1, x_2, x_3, ...$ in $[0, 1)$. We next transform the x_i into the interval $[a-0.5, b+0.5]$. Let $y_i = (b-a+1)x_i + (a-0.5)$, $i = 1, 2, 3, ...$ If the decimal part of y_i is less than 0.5 round y_i down to z_i and if the decimal part of y_i is greater than 0.5 round y_i up to z_i, $i = 1, 2, 3, ...$ In case the decimal part of y_i equals 0.5 round y_i to the nearest even integer for z_i. Even non-negative integers are $0, 2, 4, ...$ Random sequences of vectors whose components are non-negative integers, is discussed in Section 3.5.

3.4 Random Vectors: Real Numbers

Using quasi-random numbers in $[0, 1]$ we make vectors $v = (x_1, ..., x_n)$ that should uniformly fill the region $[0, 1]^n$. We can then easily adjust these vectors so that they uniformly fill the space $[a, b]^n$. It is well known ([3],[4],[10]) that if a pseudo-random number generator is used to produce sequences of vectors $v \in [a, b]^n$, and we plot their values, then there will be clusters and vacant regions in $[a, b]^n$. Quasi-random number generators are designed to avoid this problem and uniformly fill the space $[a, b]^n$.

Put "quasi-Monte Carlo simulation" into your search engine and get almost 700 web sites to visit. Another search phrase "low discrepancy numbers" could be used. We downloaded a MATLAB program for Sobol quasi-random vectors from [12]. When you run this program with small initial seeds it obviously does not start off "random". You need to discard the first few vectors and in [13] it is recommended that you delete the first 64 vectors. With large initial seeds we did not have this problem. We now generated N quasi-random vectors of length 7 using this MATLAB program, to be used in Chapter 4 to produce a random sequence of quadratic fuzzy numbers, and tested them for randomness.

We divided $[0,1]$ up into four equal intervals, in each of the seven dimensions in $[0,1]^7$, and we call these intervals $I_1 = [0.00, 0.25)$, $I_2 = [0.25, 0.5)$, $I_3 = [0.5, 0.75)$ and $I_4 = [0.75, 1.00]$. We then construct $K = 4^7$ boxes

$$B(ijklmnp) = I_i \times I_j \times ... \times I_p, \tag{3.1}$$

in $[0,1]^7$. Each vector $v = (x_1, ..., x_7)$ will fall into a unique box and for N vectors let $O(ijklmnp)$ be the number of vectors that were in box $B(ijklmnp)$. In this statistical test the null hypothesis is H_0 that the sequence of vectors is random and the alternative hypothesis H_1 is that the sequence is not random. Let the significance level γ of the test be 0.05. This will be a chi-square goodness of fit test.

Under the randomness assumption of the null hypothesis the probability of an x_i in v being in an interval I_q, $q = 1, 2, 3, 4$, is $\frac{1}{4}$, for $i = 1, ..., 7$. So the expected number of vectors in any box is

$$E = \frac{N}{4^7}. \tag{3.2}$$

For the chi-square test we would like E to be at least five so we choose $N = 100,000$.

Let θ be the degrees of freedom of the test and then the critical value for the test will be cv so that the probability of a chi-square random variable χ^2, with degrees of freedom θ, exceeding cv is equal to $\gamma = 0.05$. The chi-square random variable for this test is

$$\chi^2 = \sum_{boxes} \frac{(O(ijklmnp) - E)^2}{E}. \tag{3.3}$$

Next we need to determine the degrees of freedom θ and the critical value cv. Now [5]

$$\theta = (4^7 - 1) - [(7)(3)] = 16362, \tag{3.4}$$

because: (1) we loose one degree of freedom because the sum of the $O(ijklnmp)$ must equal N; and (2) we loose three degrees of freedom for each dimension because we must specify three of the probabilities, since their sum is one, $p_i =$ the probability of x_i being in I_i, $i = 1, 2, 3$. Since the degrees of freedom is so large we must use the approximation [5]

$$cv = 0.5(z + \sqrt{2\theta - 1})^2, \tag{3.5}$$

where z is the corresponding critical value of the standard normal distribution. We calculate $cv = 16718$.

We wrote a program in MATLAB to run this test. The value we obtained for the test statistic χ^2 was $9935.7 < cv$ and we do not reject H_0. We ran the program again but this time the seed used to produce the vector of length seven was computed from the clock in the computer. The result was $\chi^2 = 10,036 < cv$ with no rejection of the null hypothesis. This does not prove "randomness" but it gives us confidence to use this MATLAB program to produce sequences of vectors in $[0,1]^n$, $n \geq 2$, for our fuzzy Monte Carlo studies.

3.5 Random Vectors: Non-negative Integers

Now we want sequences of vectors $v = (x_1, ..., x_n)$, where each x_i is a non-negative integer in $[a, b]$, so that the v should uniformly fill the region $I \cap [a, b]^n$, $a \geq 0$, where I denotes integers. This may be used in Chapters 18, 20-23, and 25-26. Use a quasi-random number generator to get v with each $x_i \in [0, 1)$. Set $w = (a - b + 1)v + (a - 0.5)$ which puts each $x_i \in [a - 0.5, b + 0.5]$. Round the x_i to integers as follows: (1) if the decimal part of x_i is less than 0.5 round down to y_i; (2) if the decimal part of x_i is greater than 0.5 found up to y_i; and (3) if the decimal part of x_i equals 0.5 round to the nearest even integer y_i. The vector $u = (y_1, ..., y_n)$ is what we want.

References

1. Abdalla, A., Buckley, J.J.: Monte Carlo Methods in Fuzzy Linear Regression. Soft Computing 11, 991–996 (2007)
2. Buckley, J.J.: Fuzzy Probability and Statistics. Springer, Heidelberg (2006)
3. Deley, D.W.: Computer Generated Random Numbers, `http://world.std.com/~franl/crypto/random-numbers.html`
4. Henderson, S.G., Chiera, B.A., Cooke, R.M.: Generating "Dependent" Quasi-Random Numbers. In: Proceedings of the 2000 Winter Simulation Conference, Orlando, FL, December 10-13, 2000, pp. 527–536 (2000)
5. Lindgren, B.W.: Statistical Theory, 3rd edn. MacMillan, New York (1976)
6. Kimura, S., Matsumura, K.: Genetic algorithms using low-discrepancy sequences. In: Proceedings of GECCO 2005: 2005 conference on Genetic and evolutionary computation, New York, pp. 1341–1346 (2005)
7. Lord, G., Paskov, S., Vanderhoof, I.T.: Using Low-Discrepancy Points to Value Complex Financial Instruments, Contingencies, pp. 52–56 (September/October 1996)
8. Mango, D.: Random Number Generation Using Low Discrepancy Points. In: Proceedings of CAS Forum: 1999 Spring, Reinsurance Call Papers, Arlington, VA, USA, pp. 335–362 (1999)
9. MATLAB, The MathWorks, `http://www.MathWorks.com`
10. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: Numerical Recipes in C: The Art of Scientific Computing, 2nd edn. Cambridge Univ. Press, Cambridge (2002)
11. Reuillon, R., Hill, D.R.C.: Research Report LIMOS/RR-05-09: Unrolling optimization technique for quasi-random number generators. In: Proceeding of OICMS 2005: 1st Open International Conference on Modeling & Simulation, June 12-15, 2005, Blaise Pascal University, France (2005)
12. `http://www.csit.fsu.edu/~burkardt/cpp_src/cpp_src.html`
13. `http://www.puc-rio.br/marco.ind/quasi_mc.html`