Chapter 4

RANDOM NUMBERS GENERATION

© M. Ragheb
9/16/2013

4.1. INTRODUCTION

The use and generation of random numbers uniformly distributed over the unit interval: [0, 1] is a unique feature of the Monte Carlo method. These are used as building blocks for constructing and sampling probability density functions that represent any of the processes or phenomena that are under investigation. Meaningful sequences of random numbers must be generated for valid results to be generated. Otherwise the infamous gigo (garbage in, garbage out) adage of numerical computations would apply. A user of Monte Carlo is well advised to the check the validity, including length and period and randomness of his sequence of random numbers before embarking on a major simulation.

Random numbers can be obtained or generated in many different ways. The last digits in phone numbers; but not the first numbers, in a phone directory can be used as random numbers. Tables of random numbers that were statistically tested for randomness have been published, just like trigonometric functions tables. The white noise from electronic equipment and the decay of radioactive isotopes, being random phenomena have been used to generate random numbers. Spinning a roulette wheel with its perimeter divided into ten sections can generate a random sequence of digits from 0 to 9. This particular way of generating random digits sequences suggested the name of Monte Carlo, alluding to the famous gambling casino at Monte Carlo in the municipality of Monaco, by Nice in Southern France.

4.2 THE MID-SQUARE METHOD

Computer usage depends on the mathematical generation of sequences of random numbers that are long enough, or having a long period, so that they do not repeat themselves in a given simulation. If the sequence starts repeating itself, the sampled points would be repetitions of previous points. The repeated samples are useless and would not yield any new information.

In the mid-square method, due to John von Neumann, an initial number n_0 is raised to its second power. Let us consider a number of four significant digits. Then the four middle digits of the ensuing number are kept to constitute the next number in the sequence, the two digits to the left and the two digits to the right are discarded, and the process repeated.

We show an example of such a generated sequence as follows:

$$n_0 = 0.9876$$

$$n_0^2 = 0.97535376$$

$n_1 = 0.5353$

$n_1^2 = 0.28654609$

$n_2 = 0.6546$

$n_2^2 = 0.42850116$

$n_3 = 0.8501$

$n_3^2 = 0.72267001$

$n_4 = 0.2670$

…………..……

This method is marginally satisfactory, and the multiplicative congruential method has universally replaced it.

4.3 THE MULTIPLICATIVE CONGRUENTIAL METHOD

This method is the best studied and most widely used method for random sequences generation. It generates pseudo-random sequences of random numbers uniformly distributed over the unit interval. It depends on the use of the recursive relation:

$$x_i \equiv \{ax_{i-1} + c\}\text{modulo}(m) \quad (1)$$

The notation "modulo" or sometimes "mod" signifies that x_i is the remainder when $\{ax_{i-1} + c\}$ is divided into m. Here m is a large integer determined by the design of the computer, usually a large power of 2 or 10, and a, c, and x_i are integers between 0 and (m-1).

The initial value x_0 is designated as the "seed" of the sequence.

In many applications of the method the constant c is taken as zero, yielding a simpler form of Eqn. 1:

$$x_i \equiv \{ax_{i-1}\}\text{modulo}(m) \quad (2)$$

The value of m is normally taken as the largest number that can be generated on a computer, which depends on the number of bits used in its processor and its data busses. In this case for a design of n bits:

$$m = 2^n - 1 \quad (3)$$

For instance, for a hypothetical n = 4 bits machine, the largest number that can be generated would be according to Eqn. 3:

$$m = 2^4 - 1 = 16 - 1 = 15.$$

This number expressed in the binary notation is:

$$\begin{aligned} m &= 1111 \\ &= 1x2^0 + 1x2^1 + 1x2^2 + 1x2^3 \\ &= 1x1 + 1x2 + 1x4 + 1x8 \\ &= 1 + 2 + 4 + 8 \\ &= 15 \end{aligned}$$

The numbers:

$$\xi_i = \frac{x_i}{m}, i = 1,2,3,...(m-1), \qquad (4)$$

and *not* just x_i, are taken as the pseudorandom sequence over the unit interval.

The advantages of using pseudorandom sequences are that a calculation can always be repeated, starting from the same seed number, for comparison and testing purposes of programs, a few simple operations are needed, and the program uses a few memory positions.

The only disadvantage is that the sequence must satisfy certain conditions for not repeating itself after a long length or period.

As an example of a random sequence using Eqn. 3:

Let: Seed $x_0 = 2$

$m = 2^4 = 16$

$c = 1$

$a = 3$

Then: $x_0 = 2 \Rightarrow \xi_0 = \frac{2}{16} = 0.1250$

$x_1 = \{3x\ 2 + 1\} \bmod 16 = 7 \bmod 16 = 7 \Rightarrow \xi_1 = \frac{7}{16} = 0.4375$

$$x_2 = \{3x\ 7 + 1\} \bmod 16 = 22 \bmod 16 = 6 \Rightarrow \xi_2 = \frac{6}{16} = 0.3750$$

$$x_3 = \{3x\ 6 + 1\} \bmod 16 = 19 \bmod 16 = 3 \Rightarrow \xi_3 = \frac{3}{16} = 0.1875$$

$$x_4 = \{3x\ 3 + 1\} \bmod 16 = 10 \bmod 16 = 10 \Rightarrow \xi_4 = \frac{10}{16} = 0.6250$$

$$x_5 = \{3x10 + 1\} \bmod 16 = 31 \bmod 16 = 15 \Rightarrow \xi_5 = \frac{15}{16} = 0.9375$$

$$x_6 = \{3x15 + 1\} \bmod 16 = 46 \bmod 16 = 14 \Rightarrow \xi_6 = \frac{14}{16} = 0.8750$$

$$x_7 = \{3x14 + 1\} \bmod 16 = 43 \bmod 16 = 11 \Rightarrow \xi_7 = \frac{11}{16} = 0.6875$$

$$x_8 = \{3x11 + 1\} \bmod 16 = 34 \bmod 16 = 2 \Rightarrow \xi_8 = \frac{2}{16} = 0.1250$$

$$x_9 = \{3x\ 2 + 1\} \bmod 16 = 7 \bmod 16 = 7 \Rightarrow \xi_9 = \frac{7}{16} = 0.4375$$

…………………………………………………………………

We notice that the sequence obtained for the x_i's is:

2, 7, 6, 3, 10, 15, 14, 11, 2, 7, …..

so that the sequence started repeating itself with a period of 8.

If we would have chosen:

$$m = 2^4 - 1 = 15,$$

the generated sequence would become:

$$x_1 = \{3x2 + 1\} \bmod 15 = 7 \bmod 15 = 7 \Rightarrow \xi_1 = \frac{7}{15} = 0.466666$$

$$x_2 = \{3x7 + 1\} \bmod 15 = 22 \bmod 15 = 7 \Rightarrow \xi_2 = \frac{7}{15} = 0.466666$$

…………………………………………………………………..

In this case, the generated sequence is a single number that would repeat itself indefinitely, the period of the sequence is 1, and the sequence is useless for any meaningful calculations.

4.4 COMPUTER IMPLEMENTATION AND TESTING

Usually the sequence repeats itself after at most m steps. It must be ensured for a given simulation that the period is longer than the needed number of random numbers. The value of m is usually chosen large enough to permit this.

If a compiler does not provide a satisfactory random number generator, writing one's own generator is advisable. Figure 1 shows a random number generator subroutine, rand, which can be embedded and called from any other application. It could be reprogrammed as a function instead of a subroutine.

```
!       pseudo_random f90
!       Visualizing the randomness of our own pseudo random
!       number generator by generating uniformly distributed
!       points on the unit square, for plotting with a plotting
!       routine, e.g. Excel.
!       The multiplicative congruential method:
!       x(i)={a*x(i-1)+c}(modulo m)
!       is used where:
!       The (modulo m) notation signifies that x(i) is the remainder
!       when {a*x(i-1)} is divided by m.
!       m is a large integer determined by the design of the computer,
!       usually a large power of 2.
!       a, c, and x(i) are integers between 0 and m-1
!       The numbers     x(i)/m are used as the pseudo-random sequence.
!       M. Ragheb
!
        program pseudo_random
        real x, y, rr
        integer :: trials = 1000
!       Initialize output file of uniformly distributed random
!       numbers on the unit square
        open(44, file = 'random_out1')
        do i= 1, trials
                call rand(rr)
                x=rr
                call rand(rr)
                y=rr
                write (44,100) x, y
        end do
100     format (2f10.3)
        end

        subroutine rand(rr)
        real rr, xx1, xm
        integer x1
        integer :: x0 = 2
        integer :: c  = 1
!       integer :: a  = 3
```

```
      integer :: a  = 3*17
      integer :: m  = 2**20
      xm = m
      x1 = (a*x0 + c)
      x1 = mod (x1, m)
!     write(*,*) x1
      xx1 = x1
      rr = xx1 / xm
!     write(*,*) rr
      x0 = x1
      return
      end
```

Figure 1. Procedure for generating and visual testing a pseudorandom sequence on the unit square.

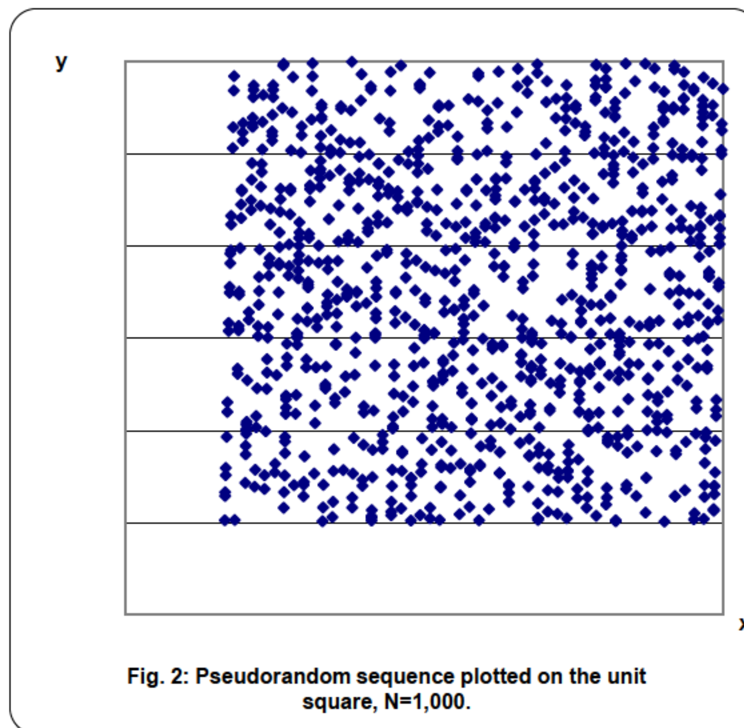

Fig. 2: Pseudorandom sequence plotted on the unit square, N=1,000.

There are formal statistical tests that are available for testing the randomness of the generated sequence. These must be used whenever possible. A fast test would be to consider each pair of consecutive points in the sequence as points in the unit square. A scatter display of these points would visually show a random pattern. A bad sequence would visually display a banding pattern whenever the period of the sequence is short and the sequence starts repeating itself. Figure 2 shows 1,000 points that display a satisfactory choice of the parameters:

$$x_0 = 2,\ c=1,\ a = 51,\ m = 2^{20}.$$

On the other hand, just changing the value of a in the routine into:

$$a = 3,$$

leads to a short sequence, which can be detected visually as banding as shown in Fig. 3.

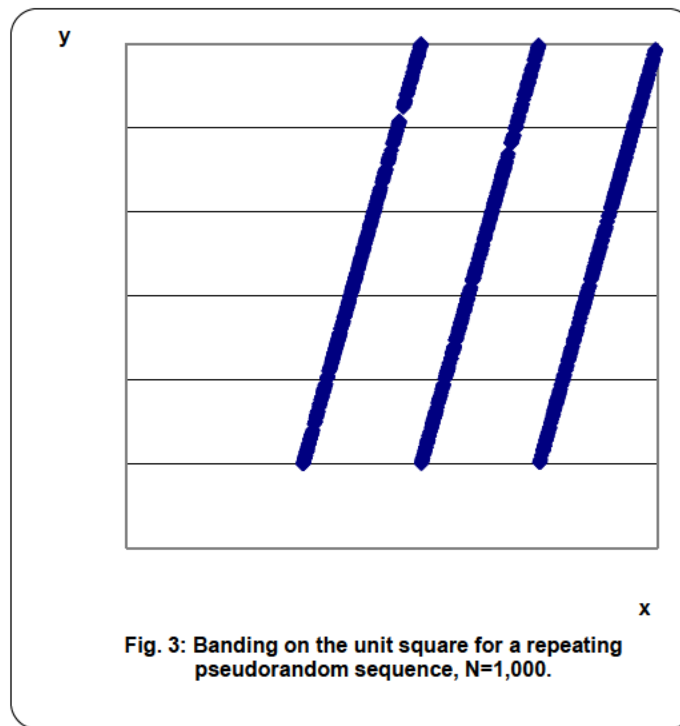

Fig. 3: Banding on the unit square for a repeating pseudorandom sequence, N=1,000.

4.5 PACKAGED PSEUDORANDOM SEQUENCES GENERATORS

Most compilers and program libraries contain well-tested pseudorandom sequences generators. The International Mathematical and Statistical program library, IMSL, contains a variety of these programs. It is still advisable to test these generators whenever they are used, to make sure that they are being implemented correctly.

The procedure in Fig. 4 calls the compiler's pseudorandom number generator, random, and places each two consecutive number as a vector of points in the plane. The file on which they are written can then be loaded into a plotting routine and displayed in the scatter mode.

```
!       plot_random_numb.f90
!       Visualising the randomness of the compiler's random
!       number generator by generating uniformly distributed
!       points on the unit square, for visualization in a
!       plotting routine, e.g. Excel.
!       M. Ragheb
!
        program plot_random_numb
        real x, y
        integer :: trials = 1000
!       Initialize output file of uniformly distributed random
!       numbers on the unit square
        open(44, file = 'random_out1')
```

```
      do i= 1, trials
            call random(rr)
            x=rr
            call random(rr)
            y=rr
      write (44,100) x, y
      end do

100   format (2f10.3)
      end
```

Figure 4. Procedure to visualize a compiler's random number generator.

4.6 QUASI RANDOM SEQUENCES

Rather than using a pseudorandom sequence, a good sampling of the unit interval may be possible by using quasi random sequences as suggested by Zaremba and Halton. In this case an example of a sequence of the unit interval being halved, and then each subdivision is halved again can generate the following quasi random sequence:

0.5	0.25	0.125	…..
	0.75	0.325	…..
		0.625	…..
		0.825	…..

Sampling the unit interval uniformly is not a goal by itself. Beyond allowing us to uniformly sample the unit interval, pseudorandom or quasi random sequences allow us to go the extra step of sampling any discrete or continuous probability density function and thus allow us to simulate any needed effect or process that can be represented by a probability density function.

4.7 DISCUSSION

The incentive to design and build computer platform with a larger number of bits in its word length is not mandated just by the need to obtain higher accuracies by retaining a large number of significant digits, the need to generate long encryption cipher keys, or by the need to generate large magnitudes in the address registers so as to manipulate long vectors and large matrices. It is also mandated by the need to generate unrepeated long sequences of random numbers in Monte Carlo simulations. This affects the costs of the computing machinery. A 32 bits or 64 bits desktop computer or workstation costs in the range of the thousands of dollars, whereas a supercomputer with 128 bits of word length costs in the range of the millions of dollars. Desktops and workstations are satisfactory for most practical simulations. If more ambitious computations are contemplated with a need of a large number of simulations, one should consider the migration of the work from a desktop machine or a workstation to a computing platform with a larger word length. A word of caution must be stated here about processors that could process data with a reported word length of say 64 bits, but

then send the data on 32 bits data buses. In this case half the bits are lost, and the computer becomes effectively a 32 bits machine. The need to check the capabilities of the computing platform used, the adequacy of the random number generator adopted, and the randomness and periods of the generated random sequences in a given computation cannot be overemphasized.

EXERCISES

1. Instead of visualizing the pseudorandom sequence on the unit square, modify the procedure given above and use each three consecutive numbers in the sequence to generate a point (x_i, x_{i+1}, x_{i+2}) in the unit cube. Display both a good and a bad sequence of pseudorandom numbers.
2. Use the simpler form of the congruential multiplicative where $c = 0$, and investigate the conditions under which good and bad random sequences are generated.