10
Deterministic Randomness

Some people are attracted to computing by its deterministic nature; it is nice to have something in life where nothing is left to chance. Barring random machine errors or undefined variables, you must get the same output every time you feed your program the same input. Nevertheless, many computer cycles are used for Monte Carlo *calculations that at their very core strive to include some elements of chance. These are calculations in which random numbers generated by the computer are used to* simulate *naturally random processes, such as thermal motion or radioactive decay, or to solve equations on the average. Indeed, much of the recognition of computational physics as a specialty has come about from the ability of computers to solve previously intractable thermodynamic and quantum mechanics problems using Monte Carlo techniques.*

Problem: Your **problem** in this chapter is to explore how computers can generate random numbers and how well they can do it. To check whether it really works, in Chap. 11 you *simulate* some random walks and spontaneous decays, and evaluate some multidimensional integrals. Other applications, such as thermodynamics and lattice quantum mechanics, are considered in later chapters.

10.1
Random Sequences (Theory)

We define a sequence of numbers $r_1, r_2, \ldots$ as *random* if there are no correlations among the numbers in the sequence. Yet randomness does not necessarily mean all numbers in the sequence are equally likely to occur. If all numbers in a sequence are equally likely to occur, then the sequence is *uniform*. To illustrate, $1, 2, 3, 4, \ldots$ is uniform but probably not random, while $3, 1, 4, 2, 3, 1, 3, 2, 4, \ldots$ may be random but does not appear to be uniform. In addition, it is possible to have a sequence of numbers that, in some sense, are random but have very short range correlations, for example, $r_1(1-r_1)r_2(1-r_2)r_3(1-r_3)\cdots$.

Mathematically, the likelihood of a random number occurring is described by a distribution function $P(r)$. This means the probability of finding r_i in the interval $[r, r+dr]$ is $P(r)dr$. A *uniform* distribution means that $P(r) =$

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu
Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim
ISBN: 978-3-527-40626-5

constant. The standard random-number generator on computers generates uniform distributions ($P = 1$) between 0 and 1. In other words, the standard random-number generator outputs numbers in this interval, each with an equal probability yet each independent of the previous number. As we shall see, numbers can also be generated nonuniformly and still be random.

By their very construction we know computers are deterministic and so they cannot truly create a random sequence. Although it may be a bit of work, if we know r_m and its preceding elements, it is always possible to figure out r_{m+1}. For this reason, computers generate *"pseudo"-random numbers*. By the very nature of their creation, computed random numbers must contain correlations and in this way are not truly random. (Yet with our incurable laziness we would not bother saying "pseudo" all the time.) While the more sophisticated generators do a better job at hiding the correlations, experience shows that if you look hard enough, or use these numbers enough you will notice correlations. A primitive alternative to generating random numbers is to read in a table of true random numbers, that is, numbers determined by naturally random processes such as radioactive decay. While not an attractive way to spend one's time, it may provide a valuable comparison.

10.1.1
Random-Number Generation (Algorithm)

The *linear congruent* or *power residue* method is the most common way of generating a pseudo-random sequence of numbers $\{r_1, r_2, \ldots, r_k\}$ over the interval $[0, M-1]$. You multiply the previous random number r_{i-1} by the constant a, add on another constant c, take the *modulus* by M, and then keep just the fractional part (remainder)[1] as the next random number r_i:

$$r_i \overset{\text{def}}{=} (a\, r_{i-1} + c) \bmod M = \text{remainder}\left(\frac{a\, r_{i-1} + c}{M}\right) \tag{10.1}$$

The value for r_1 (the *seed*) is frequently supplied by the user, and *mod* is a built-in function on your computer for *remaindering* (it may be called *amod* or *dmod*). This is essentially a bit-shift operation that ends up with the least-significant part of the input number, and thus counts on the randomness of roundoff errors to produce a random sequence.

1 You may obtain the same result for the modulus operation by subtracting M until any further subtractions would leave a negative number; what remains is the *remainder*.

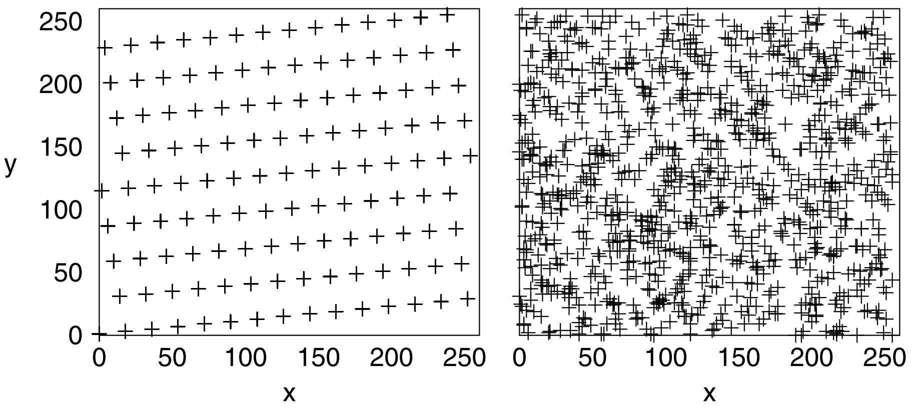

Fig. 10.1 *Left*: A plot of successive random numbers $(x, y) = (r_i, r_{i+1})$, generated with a deliberately "bad" generator. *Right*: a plot with the library routine *drand48*.

As an example, if $c = 1, a = 4, M = 9$, and you supply $r_1 = 3$, then you obtain the sequence

$$r_1 = 3, \tag{10.2}$$

$$r_2 = (4 \times 3 + 1) \bmod 9 = 13 \bmod 9 = \text{rem}\tfrac{13}{9} = 4 \tag{10.3}$$

$$r_3 = (4 \times 4 + 1) \bmod 9 = 17 \bmod 9 = \text{rem}\tfrac{17}{9} = 8 \tag{10.4}$$

$$r_4 = (4 \times 8 + 1) \bmod 9 = 33 \bmod 9 = \text{rem}\tfrac{33}{9} = 6 \tag{10.5}$$

$$r_{5-10} = 7, 2, 0, 1, 5, 3 \tag{10.6}$$

We get a sequence of length $M = 9$, after which the entire sequence repeats. If we want numbers in the range $[0, 1]$, we divide the r's by $M = 9$

$$0.333,\ 0.444,\ 0.889,\ 0.667,\ 0.778,\ 0.222,\ 0.000,\ 0.111,\ 0.555,\ 0.333.$$

This is still a sequence of length 9, but is no longer one of integers. As a general operating procedure

Before using a random-number generator in your programs, you may check its range and that it is producing numbers that "look" random.

Although this is not a strict mathematical test, your visual cortex is quite refined at recognizing patterns, and, in any case, it is easy to perform. For instance, Fig. 10.1 shows results from "good" and "bad" generators; it is really quite easy to tell them apart.

The rule (10.1) produces integers in the range $[0, M - 1]$, but not necessarily every integer. When a particular integer comes up a second time, the whole cycle repeats. In order to obtain a longer sequence, a and M should be large numbers, but not so large that ar_{i-1} overflows. On a scientific computer using 48-bit integer arithmetic, the built-in random-number generator may use M values as large as $2^{48} \simeq 3 \times 10^{14}$. A 32-bit machine may use $M = 2^{31} \simeq 2 \times 10^9$. If your program uses approximately this many random numbers, you may need to reseed the sequence during intermediate steps to avoid repetitions.

Your computer probably has random-number generators that are better than the one you will compute with the power residue method. You may check this out in the manual or help pages (try the *man* command in Unix) and then test the generated sequence. These routines may have names like *rand*, *rn*, *random*, *srand*, *erand*, *drand*, or *drand48*.

We recommend a version of *drand48* as a random-number generator. It generates random numbers in the range $[0, 1]$ with good spectral properties by using 48-bit integer arithmetic with the parameters[2]

$$M = 2^{48} \qquad c = B\ (\text{base}\,16) = 13\ (\text{base}\,8), \tag{10.7}$$

$$a = 5DEECE66D\ (\text{base}\,16) = 273673163155\ (\text{base}\,8). \tag{10.8}$$

To initialize the random sequence you need to plant a seed in it. In Fortran you would call the subroutine *srand48* to plant your seed, while in Java you issue the statement `Random randnum = new Random(seed);` (see `RandNum.java` for details).

Definition (10.1) will generate r_i values in the range $[0, M]$ or $[0, 1]$ if you divide by M. If random numbers in the range $[A, B]$ are needed, you need to only **scale**; for example

$$x_i = A + (B - A)r_i \qquad 0 \le r_i \le 1 \qquad \Rightarrow \qquad A \le x_i \le B \tag{10.9}$$

10.1.2 Implementation: Random Sequence

For scientific work we recommend using an industrial-strength random-number generator. To see why, here we assess how *bad* a careless application of the power residue method can be.

1. Write a simple program to generate random numbers using the linear congruent method (10.1).
2. For pedagogical purposes, try the unwise choice: $(a, c, M, r_1) = (57, 1, 256, 10)$. Determine the *period*, that is, how many numbers are generated before the sequence repeats.
3. Take the pedagogical sequence of random numbers and look for correlations by observing clustering on a plot of successive pairs $(x_i, y_i) =$

[2] Unless you know how to do 48-bit arithmetic and how to input numbers in different bases, we do not recommend that you try these numbers yourself. For pedagogical purposes, large numbers such as $M = 112,233$ and $a = 9999$ work well.

(r_{2i-1}, r_{2i}), $i = 1, 2, \ldots$. (Do *not* connect the points with lines.) You may "see" correlations (Fig. 10.1), which means that you should not use this sequence for serious work.

4. Test the built-in random-number generator on your computer for correlations by plotting the same pairs as above. (This should be good for serious work.)

5. Test the linear congruent method again with reasonable constants like those in (10.7)–(10.8). Compare the scatterplot you obtain with that of the built-in random-number generator. (This, too, should be good for serious work.)

Listing 10.1: `RandNum.java` calls the random-number generator from the Java utility class. Note that a different seed is needed for a different sequence.

```java
// RandNum.java: random numbers via java.util.Random.class

import java.io.*;                                  // Location of PrintWriter
import java.util.*;                                    // Location of Random

public class RandNum {

  public static void main(String[] argv)
                          throws IOException, FileNotFoundException {

  PrintWriter q = new PrintWriter(
                          new FileOutputStream("RandNum.DAT"), true);
                                             // Initialize 48 bit generator
  long seed = 899432;
  Random randnum = new Random(seed);
  int imax = 100, i = 0;

                       // generate random numbers and store in data file
   for ( i=1; i <= imax; i++ ) q.println(randnum.nextDouble());
    System.out.println(" ");
    System.out.println("RandNum Program Complete.");
    System.out.println("Data stored in RandNum.DAT");
    System.out.println(" ");
  }
}                                                          // End of class
```

10.1.3 Assessing Randomness and Uniformity

Because the computer's random numbers are generated according to a definite rule, the numbers in the sequence must be correlated to each other. This can affect a simulation that assumes random events. Therefore, it is wise for you to test a random-number generator before you stake your scientific rep-

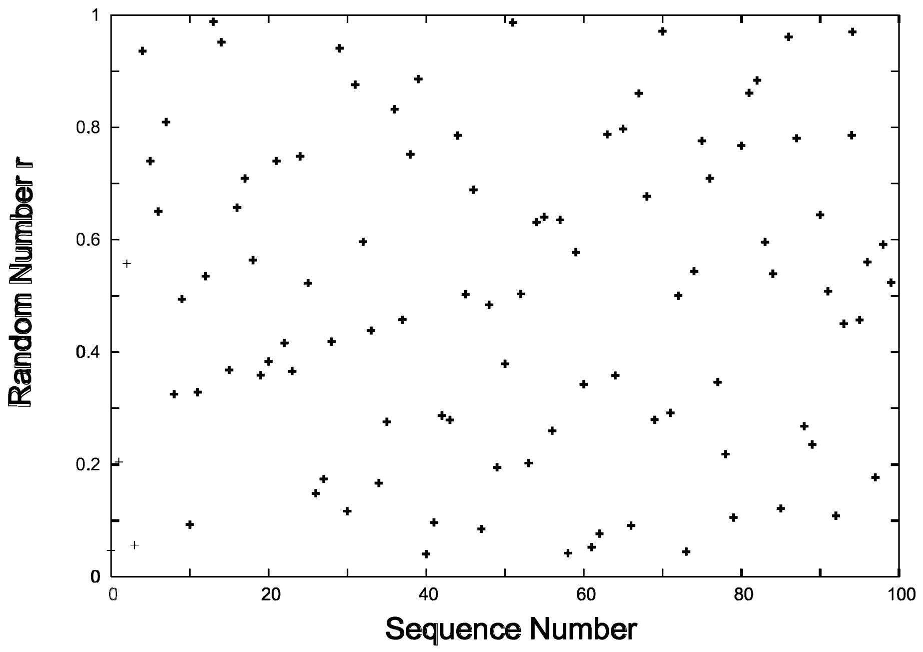

Fig. 10.2 A plot of a uniform, pseudo-random sequence r_i versus i.

utation on results obtained with it. In fact, some tests are simple enough that you may make it a habit to run them simultaneously with your simulation.

In the examples to follow, we test for either randomness or uniformity.

1. Probably the most obvious, but often neglected, test for randomness and uniformity is to look at the numbers generated. For example, Tab. 10.1 is some output from `RandNum.java`. If you just look at these numbers, you know immediately that they all lie between 0 and 1, that they appear to differ from each other, and that there is no obvious pattern (like 0.3333).

2. If you now take this same list and plot it, the ordinate will be r_i and the abscissa, even though we do not state it, will be i (Fig. 10.2). Observe how there appears to be a uniform distribution between 0 and 1 and no particular correlation between points (although your eye and brain will connect the points and create some types of figures).

3. One simple test of uniformity evaluates the kth moment of the random-number distribution:

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k \tag{10.10}$$

Tab. 10.1 A table of a uniform, pseudo-random sequence r_i generated by `RandNum.java`.

0.04689502438508175	0.20458779675039795	0.5571907470797255	0.05634336673593088
0.9360668645897467	0.7399399139194867	0.6504153029899553	0.8096333704183057
0.3251217462543319	0.49447037101884717	0.09307712613141128	0.32858127644188206
0.5351001685588616	0.9880354395691023	0.9518097953073953	0.36810077925659423
0.6572443815038911	0.7090768515455671	0.5636787474592884	0.3586277378006649
0.38336910654033807	0.7400223756022649	0.4162083381184535	0.3658031553038087
0.7484798900468111	0.522694331447043	0.14865628292663913	0.1741881539527136
0.41872631012020123	0.9410026890120488	0.1167044926271289	0.8759009012786472
0.5962535409033703	0.4382385414974941	0.166837081276193	0.27572940246034305
0.832243048236776	0.45757242791790875	0.7520281492540815	0.8861881031774513
0.04040867417284555	0.09690149294881334	0.2869627609844023	0.27915054491588953
0.7854419848382436	0.502978394047627	0.688866810791863	0.08510414855949322
0.48437643825285326	0.19479360033700366	0.3791230234714642	0.9867371389465821

If the random numbers are distributed with a *uniform* probability distribution $P(x)$, then (10.10) is approximately the moment of $P(x)$:

$$\frac{1}{N}\sum_{i=1}^{N} x_i^k \simeq \int_0^1 dx\, x^k P(x) + O(1/\sqrt{N}) \simeq \frac{1}{k+1} \tag{10.11}$$

If (10.11) holds for your generator, then you know that the distribution is uniform. If the deviation from (10.11) varies as $1/\sqrt{N}$, then you *also* know that the distribution is random.

4. Another simple test determines the near-neighbor correlation in your random sequence by taking sums of products for small k:

$$C(k) = \frac{1}{N}\sum_{i=1}^{N} x_i x_{i+k} \qquad (k = 1, 2, \ldots) \tag{10.12}$$

If your random numbers x_i and x_{i+k} are distributed with the joint probability distribution $P(x_i, x_{i+k})$ and are independent and uniform, then (10.12) can be approximated as an integral:

$$\frac{1}{N}\sum_{i=1}^{N} x_i\, x_{i+k} \simeq \int_0^1 dx \int_0^1 dy\, xy\, P(x,y) = \tfrac{1}{4} \tag{10.13}$$

If (10.13) holds for your random numbers, then you know that they are not correlated. If the deviation from (10.13) varies as $1/\sqrt{N}$, then you *also* know that the distribution is random.

5. An effective test for randomness is performed visually by making a scatterplot of $(x_i = r_{2i}, y_i = r_{2i+1})$ for many i values. If your points have noticeable regularity, the sequence is not random. If the points are random, they should uniformly fill a square with no discernible pattern (a cloud) (Fig. 10.1).

6. Another test is to run your calculation or simulation with the sequence $r_1, r_2, r_3, \ldots$, and then again with the sequence $(1 - r_1), (1 - r_2), (1 - r_3), \ldots$. Because both sequences should be random, if your results differ beyond statistics, then your sequence is probably not random.

7. Yet another test is to run your simulation with a sequence of true random numbers from a table and compare it to the results with the pseudo-random-number generator. In order to be practical, you may need to reduce the number of trials being made.

8. Test your random-number generator with (10.11) for $k = 1, 3, 7$, and $N = 100, 10{,}000, 100{,}000$. In each case print out

$$\sqrt{N} \left| \frac{1}{N} \sum_{i=1}^{N} x_i^k - \frac{1}{k+1} \right| \tag{10.14}$$

to check that it is of order 1.

9. Test the mildly correlated series $r_1(1 - r_1)r_2(1 - r_2)r_3(1 - r_3) \cdots$ with (10.13) for $N = 100, 10{,}000, 100{,}000$. Again print out the deviation from the expected result and divide the deviation by $1/\sqrt{N}$ to check that it is of order 1.