Stanford Exploration Project, Report 73, November 18, 1997, pages 463–471

Short Note

Random sequences: a review

Francis Muir[1]

keywords: *sequence, random, algorithm, computing*

INTRODUCTION

Notions of randomness are not new in exploration seismology – *white noise* has been with us since the genesis of digital signal processing – but it is likely that they will find increasing use as we move from a deterministic to a probabilty basis for seismic data processing.

In this Short Note I describe the desirable properties of computer-generated random sequences, and present some simple type algorithms.

REQUIREMENTS

The requirements listed below are in the order of their importance.

1. **Random**. The generated sequence should have as much the appearance of randomness as is possible, and as many of the statistical properties as are required of it; e.g., independence, stationarity, distribution characteristics, &c. Since the generator will, in fact, be deterministic, it should be simple enough that its shortcomings are well-known. In the case of linear congruential sequences, for example, it is their periodicity and Marsaglia's complaint that *points lie mainly in the planes*.

2. **Repeatable**. Although we seek to model indeterminate effects, our schemes must not themselves involve indeterminacy. Reproducibility is of the essence in any scientific experiment.

3. **Portable**. In the case of integer sequences, we can insist on identity of results across platforms. For floating-point sequences this is impractical, but individual numbers should differ by no more than round-off dictates.

[1]**email:** not available

4. **Fast**. All the schemes I have in mind are now so fast that speed is no longer an issue. The only problem might be with real-time generators for VHF acoustic signals in laboratory work with rock samples.

GENERATING RANDOM SEQUENCES

Someone – probably Knuth (1981) – has remarked that a sequence with Uniform distribrution can form the basis for generating sequences with any other distribution. A corollary is that a well-shuffled bunch of consecutive integers is the ideal primary ingredient for a Uniform distribution sequence. So, the three steps needed to make a sequence with specified distribution are,

- **Shuffle** a set of consecutive integers.
- **Transform** these integers into floating-point numbers covering the range $\{0, 1\}$.
- **Map** this Uniform distribution over to the required (e.g. Normal) distribution.

Shuffling integers

Lehmer's Method for shuffling is the simplest, and is completely analysable.

1. **Multiply** the current value, x_j, by *magic1*, a constant.
2. **Reduce** the result modulo *magic2*, another constant, and post in $x_j + 1$.

In FORTRAN this might look like:

```
x(j) = MOD(x(j - 1)*magic1,magic2)
```

This scheme leads to a complete, shuffled set of the integers from 1 through (*magic2*-1) if certain constraints on *magic1* and *magic2* are met.

A simple shuffle

If the product of *magic1* and *magic2* is less than the maximum sized integer that the computer can represent, then the code is straightforward, and is listed in the Appendix as SUB1. The modulo, *magic2*, is the Mersenne prime $(2^{19} - 1)$, and the multiplier, *magic1*, is 727, which is primitive modulo *magic2*. These two conditions guarantee that the cycled sequence is complete (see Knuth).

Schrage's method

SUB1 will have too short a cycle for many applications, and in that case we may turn to Schrage's Method (Bratley &al). Here, the *modulo* must still be less than the largest possible integer which the computer can represent (which I take to be $2^{31}-2$), but the product is never explicitly evaluated. In this version, *magic1* and *magic2* are the tried and true 7^5 and $(2^{31}-1)$ respectively – values which are to be found in IMSL and other FORTRAN subroutine sources, and which are discussed by Park & Miller (1988). This algorithm is listed as SUB2 in the Appendix.

An alternative to Schrage

It is possible that Schrage's Method may be too slow for some real-time applications. In this case SUB3 is an alternative. It is identical in structure to SUB1, but uses DOUBLE PRECISION for the internal arithmetic. It depends on the fact that floating point arithmetic precisely emulates integer arithmetic if all numbers can be kept inthe mantissa. On any particular machine, SUB3 should be first tested against SUB2 for identity of result.

Uniform distribution

Sequences with Uniform distribution can be directly created from shuffled, consecutive integers by shifting and scaling. However, many authors, including Park & Miller, introduce a bias into the scaling operation. SUB4 combines this shifting and scaling with SUB1 to provide a simple Uniform generator. It can be modified to take advantage of the longer integer sequences available from SUB2 and SUB3.

Normal Distribution

Knuth (pp. 117–127) discusses several methods of generating sequences with Normal distribution, and I prefer the *Polar Method* due to Box, Muller & Marsaglia, and referenced by Knuth. In SUB5 I combine integer shuffling, Uniform scaling and shifting, and Gaussian mapping into a single subroutine.

Better multipliers

It is almost certain that there are better multipliers than 7^5. Fishman (Park & Miller) has suggested 48721 and 69621, and they are both more random in a certain sense of the word. Anyone interested in a random comparison project might like to test one or both of these two numbers as seeds.

Choice of seed

If only one sequence is required, then setting SEED to unity is always a good choice. if there are multiple sets of sequences required, then it will depend on whehther they are to be generated serially or in parallel. If serially, then the output SEED can be used as the input to the next subroutine call. If in parallel, then seeding each set with consecutive integers, 1,2,3 &c. is an ideal solution. It is certainly difficult to see why one would ever want to surprise oneself by using a lottery scheme such as the system clock. The seeds, like the algorithms, should be reproducible.

ACKNOWLEDGMENTS

I thank Ray Abma for pointing me to the Park & Miller article.

REFERENCES

Bratley, P., Fox, B. L., and Schrage, E. L., 1983, A Guide to Simulation: Springer-Verlag, New York, 180–213.

Knuth, D. E., 1981, The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, 2nd ed: Addison-Wesley, Reading, MA.

Park, S. K., and Miller, K. W., 1988, Random number generators: good ones are hard to find: Commun. ACM, **31**, no. 10, 1192–1201.

APPENDIX: SOME FORTRAN SUBROUTINES

A simple shuffle

```
SUBROUTINE sub1(npts,seed,output)

INTEGER npts,seed,output(npts)
INTEGER magic1,magic2,j

magic1 = 727
magic2 = 524287
output(1) = seed
DO j = 2,npts
   output(j) = MOD(magic1*output(j - 1),magic2)
END DO
```

```
      seed = MOD(magic1*output(npts),magic2)

      RETURN
      END
```

Schrage's Method

```
      SUBROUTINE sub2(npts,seed,output)

      INTEGER npts,seed,output(npts)
      INTEGER magic1,magic2,j,hi,lo,test

      magic1 = 7**5
      magic2 = 2*(2**30 - 1) + 1
      quot = magic2/magic1
      rem = MOD(magic2,magic1)
      output(1) = seed
      DO j = 2,npts
         hi = output(j - 1)/quot
         lo = MOD(output(j - 1),quot)
         test = magic1*lo - rem*hi
         IF (test.GT.0) THEN
            output(j) = test
         ELSE
            output(j) = test + magic2
         END IF
      END DO
      hi = output(npts)/quot
      lo = MOD(output(npts),quot)
      test = magic1*lo - rem*hi
      IF (test.GT.0) THEN
         seed = test
      ELSE
         seed = test + magic2
      END IF

      RETURN
      END
```

An alternative to Schrage

```
SUBROUTINE sub3(npts,seed,output)

INTEGER npts,seed,output(npts)
INTEGER j
DOUBLE PRECISION dmagio1,dmagio2,doubt

dmagio1 = DBLE(7**5)
dmagio2 = DBLE(2*(2**30 - 1) + 1)
doubt = DBLE(seed)
output(1) = doubt
DO j = 2,npts
   doubt = MOD(dmagio1*doubt,dmagio2)
   output(j) = INT(doubt)
END DO
doubt = MOD(dmagio1*DBLE(output(npts)),dmagio2)
seed = INT(doubt)

RETURN
END
```

Uniform distribution

```
SUBROUTINE sub4(npts,seed,output)

INTEGER npts,seed
REAL output(npts)

INTEGER magic1,magic2,temp,j
REAL alfa,beta

magic1 = 727
magic2 = 2**19 - 1
beta = 1.0/FLOAT(magic2 - 1)
alfa = - beta/2.0
temp = seed
DO j = 1,npts
   output(j) = alfa + beta*FLOAT(temp)
   temp= MOD(magic1*temp,magic2)
END DO
```

```
      seed = temp

      RETURN
      END
```

Normal distribution

```
      SUBROUTINE sub5(npts,seed,output)

C        SUB5 generates gaussian deviates in OUTPUT.

C        independent
C        zero mean
C        unity variance

      INTEGER npts,seed
      REAL output(npts)

      INTEGER           j
      DOUBLE PRECISION  midpoint,scale,v1,v2,sumsq,squirt
      DOUBLE PRECISION  dmagio1,dmagio2,dj1,dj2

      dmagio1 = DBLE(7*7*7*7*7)
      dmagio2 = DBLE(1 + 2*(2**30 - 1))
      dj2 = DBLE(seed)
      midpoint = DBLE(dmagio2)/2.0D0
      scale = 2.0D0/(DBLE(dmagio2) - 1.0D0)

      DO j = 2,npts,2
111      dj1 = MOD(dmagio1*dj2,dmagio2)
         dj2 = MOD(dmagio1*dj1,dmagio2)

C           dj1 and dj2 cycle irregularly through the
C           integers 1 through (dmagio2 - 1)

         v1 = scale*(dj1 - midpoint)
         v2 = scale*(dj2 - midpoint)

C           v1 and v2 are the dj's scaled and shifted so that
C           they are uniformly distributed between -1 and 1

         sumsq = v1**2 + v2**2
```

```
            IF(sumsq.GE.1.0D0) THEN
               GO TO 111
            ELSE
            END IF

C              this IF tosses out all points {v1,v2} not inside
C              the unit circle

            squirt = SQRT((-2.0D0*LOG(sumsq))/sumsq)
            output(j - 1) = SNGL(v1*squirt)
            output(j) = SNGL(v2*squirt)
         END DO
         RETURN
         END
```