A PORTABLE RANDOM NUMBER GENERATOR WITH BUILT-IN WELL-SPREAD SEEDS

Graham Birtwistle
Computer Science Department
University of Calgary, Alberta, Canada T2N 1N4

ABSTRACT

This paper discusses the techniques employed in implementing pseudo-random number generating routines in Demos (a discrete event simulation extension to Simula). We give a basic routine for generating random numbers uniform in the interval (0, 1) and show how this was built out to produce the other standard distributions such as randint, Poisson, negexp, etc.; then we show how the basic routine can be adapted to generate many well-separated seeds; and how to effect automatic reporting. The techniques can be employed with other random number generators and in languages other than Simula.

BACKGROUND

This paper describes how random number routines are implemented in Demos [1]. Demos is a discrete event simulation extension to the general purpose programming language Simula [2]. Simula is implemented on IBM, UNIVAC, Cyber, Dec 10/20, ICL 2900, Seimens, Honeywell GCOS systems amongst others. Demos is process based and includes its own event list routines; built-in process-resource, process-process, and interrupt synchronisations; automatic reporting; tracing; and its own random number routines. Demos is coded entirely in Simula.

Simula itself contains several random drawing procedures: draw, randint, uniform, normal, poisson, negexp, erlang, discrete, linear and histd. Whilst these library routines are well-implemented and fast, there are some drawbacks to using them:

(a) the basic generators differ from Simula implementation. Thus experiments run on one hardware cannot be duplicated closely on another.

(b) distribution profiles and usages are not automatically reported.

(c) although an arbitrary number of distributions can be employed in Simula programs, their start seeds are chosen by the user. There is no neat, general way of ensuring that they are well separated.

(c) the distributions are all implemented as subroutines and calls contain several parameters. It is easy to make textual slips when calling the same distribution from different program points (e.g. randint (1, 6, U1) from one place, and randint (i, 6, U1) from one another). Whilst not frequent, these errors are difficult to spot.

At the cost of some loss of speed and little loss of flexibility, Demos attempts to overcome these deficiencies. The design aims for the Demos random number library were that it should run without change on any machine with a word length of 32 bits or more, that it be written entirely in Simula, permit textually neat and descriptive calls, allow for the automatic generation of well-spread seeds, and automatically echo back all distribution profiles and their use.

THE CHOICE OF GENERATOR

Most Simula compilers have used Lehmer generators of the form

$$U_0 := \text{some constant} \qquad \{1 \le U_0 < p\}$$

$$U_{k+1} := a * U_k \bmod p \qquad \{k \ge 1\}$$

and found them satisfactory. It is easy to show that the period of the generator is given by the minimum n satisfying $a^n \equiv 1 \bmod p$. It follows from the Fermat-Euler theorem that if a and p have no factors in common then $a^{p-1} \equiv 1 \bmod p$. But (p-1) is not necessarily the smallest n for which this holds; however, the smallest n must divide (p-1). If the smallest value of n is p-1, then a is a primitive root mod p and the cycle of length p-1 can be attained. Hardy and Wright [3] have shown that a primitive root exists for every prime p and that the number of primitive roots is $\phi(p-1)$ the number of integers less than and prime to (p-1). There is no general rule for finding a primitive root for a given prime. However, sufficient conditions for 2 to be a primitive root mod p have been established (see for example, Roberts [4]). They are

(a) (p-1)/2 is prime

(b) p = 3 mod 8.

For such a prime p, an appropriate primitive root a can be found by $a = 2^s \bmod p$, hcf(s, p-1) = 1.

When choosing a and p we have to bear in mind that Simula is a high level language and will not tolerate arithmetic overflow (it results in a run time error). At first sight it would seem that in order to prevent overflow, we would have to restrict our choice of a and p so that $ap \le 2^{31}$. Knuth [5] advises further that $\sqrt{p} < a < p - \sqrt{p}$, i.e. $p \sim 2^{20}$ (if a is small) and $p \sim 2^{15}$ (if a is large), which conditions are a shade too restrictive in large simulations. A way to lengthen the period is to find an a which factorises (say $a = a_1a_2a_3$), then U_{k+1} can be computed by

$$X \leftarrow a_1 * U_k \bmod p$$

$$X \leftarrow a_2 * X \bmod p$$

$$U_{k+1} \leftarrow a_3\, X \bmod p$$

three times slower but with an extended range for p, $(p \sim 2^{31}/\max(a_1, a_2, a_3))$.

We chose a published generator ([6]) with good overall properties which fits the above demands:

$$U_{k+1} := 8192 * U_k \bmod 67099547$$

By splitting 8192 into 32*32*8, the generator can be coded as

```
FOR K := 32, 32, 8 DO
BEGIN
  U := K * U;
  IF U >= 67099547 THEN U := U -
    U//67099547 * 67099547;
END;
```

which will not cause overflow on a 32 bit machine since 67099547 < 67108864 = 2**26 and 32 = 2**5.

When it comes to generating well-spread seeds automatically, we take advantage of a neat trick described by Mats Ohlin [7]. Note that if

$$U_{k+1} = a * U_k \bmod p,$$

$$= a * a * U_k \bmod p,$$

$$U_{k+3} = a * U_{k+2} \bmod p,$$

$$= a * a * a * U_k \bmod p, \text{ etc.}$$

It follows that if we use a^r as multiplier, we move through the basic cycle in steps of r at a time. Not all values of r are suitable as the period may be drastically shortened through an unfortunate choice. We have chosen r = 120633 which has the multiplier 8192**120633 mod 67099547 = 36855 = 7*13*15*27. Again this can be coded in Simula so as not to give overflow on a 32 bit machine. The period of this (seed) generator is also 67099546.

WELL-SPREAD SEEDS

Well-spread seeds are automatically generated in Demos by calls on NEXTSEED which works in conjunction with the global variable SEED.

```
INTEGER SEED, M;    {initially SEED = 907,
                               M = 67099547}
INTEGER PROCEDURE NEXTSEED;
BEGIN
  INTEGER K;
  FOR K := 7, 13, 15, 27 DO
    SEED := mod(SEED * K, M);
  NEXTSEED := SEED;
END***NEXTSEED***;

PROCEDURE SETSEED(N); INTEGER N;
BEGIN
  IF N < 0 THEN N := -N;
  N := mod(N, M);
  IF N = 0 THEN N := M//2
  SEED := N;
END***SETSEED***;
```

PROCEDURE SETSEED(N); INTEGER N; normally sets SEED to N, but if N lies outside the range [1, 67099546] - which would cause NEXTSEED to overflow (or always return 0 if SEED became 0) - N is manipulated into a safe value by

```
N := REMAINDER(ABS(N), M);
IF N = 0 THEN N := M//2,
```

Initially SEED is set by Demos to 907; thereafter every time a DIST object is created (or a subclass object), its U and USTART variables are set by calling NEXTSEED. Each such call produces the next 'well-spread-seed' separated from the last by 120633 drawings.

When a Demos program is run, it generates well-spread-seeds in the predetermined order (these default values may be overridden):

0	907
1	33427485
2	22276755
3	46847980
4	43847980
5	64042082
6	44366385
7	41357879
8	11320893
9	6528269
10	47478000

each of which has its 'own' portion of the basic cycle of length 120633 drawings. After 120633 drawings, the underlying rth distribution will start to overlap with the r+1st. This separation holds for over 500 distributions.

DISTRIBUTIONS IMPLEMENTED

Nine distributions are defined in Demos:

Returning real values we have: CONSTANT (A) which always returns the same value A; ERLANG (A, B) which returns a drawing from an ERLANG distribution with mean A and standard deviation A/SQRT(B); EMPIRICAL (N) which reads in a cumulative probability distribution in the form of a table of 'size' N and returns samples from it; NEGEXP (A) which returns drawings from an exponential distribution with an arrival rate of A; NORMAL (A, B) which returns drawings from a normal distribution with mean A and standard deviation B; UNIFORM (A, B) which returns drawings from a uniform distribution with lower bound A and upper bound B;

Returning integer values we have: POISSON (A) which returns drawings from a POISSON distribution with a mean of A; RANDINT (A, B) which returns drawings from a distribution randomly distributed amongst the integers A, A+1, ..., B.

Returning Boolean values we have: DRAW (P) which returns true with probability P.

These nine distributions are not implemented as procedures, but as Simula classes. A Simula class is a template for a data structure. An arbitrary number of instances of a class ('objects' in Simula parlance) may be created. The code below declares two pointers U1 and U2 and a class UNIFORM (to which more will be added), then creates two objects 'representing' distributions uniform over (0, 100) and (-10, 10) respectively.

```
REF(UNIFORM) U1, U2;

CLASS UNIFORM (LB, UB); INTEGER LB, UB;
BEGIN
  INTEGER U;
  REAL RESETAT;

  RESETAT := TIME;
  U       := NEXTSEED;
  IF LB > UB THEN error;
END***UNIFORM***;

U1 :- NEW UNIFORM (0.0, 100.0);
U2 :- NEW UNIFORM (-10.0, 10.0);
```

When a UNIFORM object is created (say, by NEW UNIFORM (0.0, 100.0), its parameter values are initialised (here LB := 0.0, UB := 100.0), and its local data values set to standard Simula defaults (U := 0; RESETAT := 0.0). Then the actions of the class body are carried out. These set RESETAT to the current simulation clock time, the start seed for this distribution U is initialised by a call on global NEXTSEED (here to 33427485); and then the sense of the parameters is checked (we make sure that LB ≤ UB). When the class body actions are exhausted, control returns to the generating call and the assignment to U1 is completed. Then, and in the same way, U2 is created.

The parameters and local values may be accessed from outside an object via the dot notation

```
"pointer.attribute" - e.g. U1.LB = 0.0,
                           U1.U = 33427485,
                           U2.U = 22276755.
```

In addition to data and actions, a class declaration may also contain procedures which operate upon its attributes (parameters and local quantities) and globals. A more complete definition of UNIFORM is:

```
CLASS UNIFORM (LB, UB); REAL LB, UB;
BEGIN
  INTEGER U, OBS;
  REAL RESETAT;

  PROCEDURE RESET;
  BEGIN
    RESETAT := TIME;
    OBS     := 0;
  END***RESET***;

  REAL PROCEDURE NEXT;
  BEGIN
    INTEGER K;
    FOR K := 32, 32, 8 DO
      U := mod (K * U, 67099547);
    NEXT := U/67099547;
    OBS  := OBS + 1;
  END***NEXT***;

  REAL PROCEDURE SAMPLE;
    SAMPLE := LB + (UB - LB) * NEXT;

  PROCEDURE REPORT;.....;

  RESET;
  U := NEXTSEED;
  IF LB > UB THEN error;
END***UNIFORM***;
```

Given CHECKOUT :- NEW UNIFORM (50.0, 100.0) with this new definition of UNIFORM, CHECKOUT.RESET sets CHECKOUT.RESETAT to the current clock time and CHECKOUT.OBS (the number of calls on CHECKOUT. NEXT in this epoch) to zero; CHECKOUT.NEXT operates upon CHECKOUT.U and returns a real value uniform in (0, 1). It also increments CHECKOUT. OBS; and CHECKOUT.SAMPLE which calls CHECKOUT. NEXT and then transforms the latter's value into one uniform over (50.0, 100.0). Note that if we implement UNIFORM as a procedure, each call would require three parameters (lower bound, upper bound, seed) AND we miss the opportunity to associate an informative name with the call, as with CHECKOUT.SAMPLE. CHECKOUT.REPORT will be covered later.

When we come to write other distribution definitions (negexp, Erlang, Poisson, ...), much of this work is duplicated. Each of them makes good use of RESETAT, U and OBS; and for each of them RESET and NEXT are identical. When class declarations have properties in common like this, the shared attributes may be defined separately and built into later definitions by 'prefixing'. This property which is unique to Simula enables class declarations to be built up in a hierarchical fashion.

In our case we are going to define nine distinct distributions: 6 of which will return real values, 2 will return integer values, and one a boolean. We first extract their common portion and define it in a separate class:

```
CLASS DIST;
BEGIN
  INTEGER U, OBS;
  REAL RESETAT;
  PROCEDURE RESET; as above;
  REAL PROCEDURE NEXT; as above;
  RESET;
  U := NEXTSEED;
END***DIST***;
```

We could now define a class equivalent to the previous UNIFORM by:

```
DIST CLASS UNIFORM (LB, UB); INTEGER LB, UB;
BEGIN
  REAL PROCEDURE SAMPLE;
    SAMPLE := LB + (UB - LB) * NEXT;

  PROCEDURE REPORT;.....;

  IF LB > UB THEN error;
END***UNIFORM***;
```

Prefixing CLASS UNIFORM by DIST, builds all the attributes and actions of DIST into this new declaration. Notice how by writing everything that is common to all distributions as a separate class enables us to concentrate at the new level on what is specific to a UNIFORM distribution. The writing of all other eight distributions is similarly simplified. For example, class constant is merely

```
DIST CLASS CONSTANT (X); REAL X;
BEGIN
  REAL PROCEDURE SAMPLE;
    SAMPLE := X;

  PROCEDURE REPORT;.....;

END***CONSTANT***;
```

and class draw

```
DIST CLASS DRAW (P); REAL P;
BEGIN
  BOOLEAN PROCEDURE SAMPLE;
    SAMPLE := P > NEXT;

  PROCEDURE REPORT;.....;

END***DRAW***;
```

The hierarchy actually implemented in Demos deviates slightly from the above. Demos has several built-in devices which can be created in arbitrary numbers in Demos programs: resources, queues, data collection devices; besides distributions. These devices need to be reset and reported from time to time. Accordingly, Demos defines CLASS REPORTQ and the Demos system automatically generates special REPORTQs for each possible type of device. Thus we have built-in REPORTQ's called DISTQ, QUEUEQ, HISTOQ, etc. Each time a device is created in a program, it is automatically entered into an appropriate REPORTQ. This makes automatic reporting trivially easy; the global report routine is

```
REPORT:
  print heading and time of this report;
  FOR each reportq R DO
    IF R is not empty THEN
    BEGIN
      print an appropriate heading;
      FOR each object X in R DO
        call X.REPORT;
    END;
```

To permit membership of REPORTQs, Demos contains CLASS TAB (outlined below: the parameter TITLE is used in reports):

```
CLASS TAB (TITLE); VALUE TITLE; TEXT TITLE;
  VIRTUAL: PROCEDURE RESET, REPORT;
BEGIN
  REF(TAB) NEXT;  {chain to next in its
                   REPORTQ}
  PROCEDURE JOIN (R); REF(REPORTQ) R;...;
END***TAB***;
```

The VIRTUAL specification is a device unique to Simula which makes declarations completed at inner levels accessible at the TAB level. This makes the coding of global REPORT (and RESET) very neat and simple, yet still secure. To build these capabilities into each distribution, we alter our previous heading of CLASS DIST to

```
TAB CLASS DIST;
```

and add in as a final class body action

```
JOIN (DISTQ);
```

On creation, each and every distribution object is automatically entered into DISTQ where it can be found whenever there is a call on one of the global routines REPORT and RESET.

A final trick used was to define three classes RDIST, IDIST and BDIST as below:

```
DIST CLASS RDIST; VIRTUAL: REAL PROCEDURE
                                SAMPLE;;
DIST CLASS IDIST; VIRTUAL: INTEGER PROCEDURE
                                SAMPLE;;
DIST CLASS BDIST; VIRTUAL: BOOLEAN PROCEDURE
                                SAMPLE;;
```

and to use RDIST to prefix NORMAL, NEGEXP,..., UNIFORM: IDIST to prefix POISSON and RANDOM; BDIST to prefix DRAW. In this way, the sample routines defined at the bottom level are made accessible at the ?DIST levels (? = R, I or B). Again use of VIRTUAL makes the report on all distributions much neater to code. Also, now a REF (DIST) pointer can be used to reference ANY NORMAL, NEGEXP,..., UNIFORM object AND has access to its real valued function SAMPLE.

REPORTING

Except for Empirical distribution reports, all occupy one line and echo back the distribution's title, creation time (or last reset time), number of recorded observations, type, parameters (xbar and sigma if a NORMAL, etc.), and start seed value. A typical set of reports is tabulated below:

```
               D I S T R I B U T I O N S
               *************************
TITLE      OBS/TYPE      /       A/       B/       SEED
LOAD       1000 CONSTANT  50.000
WAITS      1000 NORMAL    10.000   1.000  22276255
SERVICE    1000 UNIFORM    1.000   3.000  46847980
BULB LIFE  1000 ERLANG      .750       3  43859043
NEXT BUS   1000 NEGEXP     1.000          64042082
KICKS      1000 POISSON    0.600          41357879
THROWS     1000 RANDINT        1       6  11320893
CHANCE     1000 DRAW       0.400           6528269
```

The odd man out in these distributions is CLASS EMPIRICAL whose report is spread over several lines. Because of this, Empirical objects are not entered into DISTQ on creation; they are chained into a separate REPORTQ, named EMPQ. A typical empirical report is:

```
                  E M P I R I C A L S
                  *******************
   TITLE        /     (RE)SET/   OBS/      SEED
   WEIGHTS               0.000   1000   33427485

        K/    DIST. X(K)/  PROB. P(K)
        1       58.00000     0.00000
        2       63.00000     0.10000
        3       68.00000     0.45000
        4       70.00000     0.55000
        5       75.00000     0.90000
        6       80.00000     1.00000
```

RDIST, IDIST, BDIST AND THEIR SUB-CLASSES

The declaration of RDIST is simply

```
    DIST CLASS RDIST; VIRTUAL: REAL PROCEDURE
                                   SAMPLE;;
```

It has an empty body, but makes the matching sample routine defined in its sub-classes available to a REF(RDIST) variable. Each sub-class of RDIST contains parameters which specify the particular distribution, a real procedure sample which uses these parameters and the objects own report procedure. The initialising actions check (where possible) for unlawful parameter values. For example,

```
    RDIST CLASS NEGEXP (LAMBDA); REAL LAMBDA;
    BEGIN
      REAL PROCEDURE SAMPLE;
        SAMPLE := -LN(NEXT)/LAMBDA;

      PROCEDURE REPORT;.....;

      IF LAMBDA <= 0.0 THEN error;

    END***NEGEXP***;
```

Classes returning integer values (only POISSON and RANDINT are implemented) are prefixed by IDIST (e.g. IDIST CLASS RANDINT (LB, UB); INTEGER LB, UB; etc.) and again all that needs to be added are an integer procedure sample, a report procedure and actions checking the parameter values.

Classes returning boolean values are prefixed by BDIST.

READDIST

Distribution definitions need not be fixed by the program, they can be read in from the keyboard or a file. Calls on READDIST require two parameters - a text string which must be matched by the input, and a reference variable. Typical external definitions are:

```
     LOAD        CONSTANT    50.0
     WAITS       NORMAL      10.0    1.0
     SERVICE     UNIFORM      1.0    3.0
     BULB.LIFE   ERLANG       0.75   3
     NEXT BUS    NEGEXP       1.0
     KICKS       POISSON      0.6
     THROWS      RANDINT      1      6
     CHANCE      DRAW         0.4
     WEIGHTS     EMPIRICAL    6

           0.00 58.0
           0.10 63.0
           0.45 68.0
           0.55 90.0
           0.90 75.0
           1.00 80.0
```

READDIST operates by skipping leading blanks and matching the text parameter with the input file. If that succeeds, it then reads in the type of the distribution and appropriate parameter values. Then an object of the specified type is created (its TITLE is taken as the first parameter to READDIST) and assigned to the second parameter to READDIST.

SOURCE CODE

The source for Demos is obtainable from the author. (Also it forms an appendix to [8]).

The Simula code (counted in lines) devoted to distribution declarations and definitions is as tabulated below:

	Source	Error checking
TAB	32	2
SEED GENERATORS	16	3
DIST	85	28
RDIST, IDIST, BDIST	3	0
CONSTANT	8	0
NORMAL	24	5
NEGEXP	13	5
UNIFORM	15	6
ERLANG	24	10
EMPIRICAL	91	45
RANDINT	16	6
POISSON	22	5
DRAW	8	0
READDIST	54	21
sundries	13	0
Totals:	424	136

Thus code for Demos distributions accounts for roughly 20% of the total Demos source code, and Error checking and correcting accounts for over 30% of this code for the distributions.

CONCLUSIONS

The implementation aims for the distribution library of DEMOS have all been achieved: the code is written entirely in Simula and runs on IBM, UNIVAC, DEC 10, DEC 20, ICL Systems 4 without modification (on Cyber hardware, keywords are quoted); well-spread seeds are automatically generated; and the calls are neat (calls on SAMPLE require no parameters).

Another important advantage is the ease of adding in new distribution types (as evidenced by the quoted declarations of CONSTANT and DRAW). Since queue membership and RESET are built in, it remains to ask first what type of result is to be returned (real, integer or boolean?) and prefix with RDIST, IDIST or BDIST accordingly. Secondly, the appropriate SAMPLE function must be written (the compiler will give a fault if its type is inconsistent with that implied by the prefix). Finally, a report has to be written. Keep its pattern consistent - other reports already defined spell this out.

Because RESET, REPORT and SAMPLE are all specified as VIRTUAL it is very easy to change any existing definitions. Suppose for example, very high accuracy is required in the extreme ends of a NORMAL distribution. The user can write simply:

```
NORMAL CLASS VHA_NORMAL;
BEGIN
  REAL PROCEDURE SAMPLE;
  BEGIN
    new definition;
  END***REPORT***;
END***NORMAL***;
```

We can now use NORMAL or VHA_NORMAL distributions. NORMAL is untouched. But if the user now uses a VHA_NORMAL object, his new REPORT replaces the one at the NORMAL level (indeed the one at the NORMAL level logically no longer exists in VHA_NORMAL objects). In all other respects a VHA_NORMAL object behaves as a NORMAL one. It will be entered into DISTQ on generation and its old RESET and REPORT are still valid (but standard definitions for the latter may also be overwritten since they too are specified as virtual).

If this piecemeal approach is not good enough, one can of course add in complete new definitions to co-exist along with standard ones. Or more drastically, since Demos is delivered in source, one can throw out old definitions, add new ones, and recompile.

Finally distributing Demos in source has turned maintenance into a non-problem. In three years, 5 errors have been reported. In all cases, corrections have been sent along with the notification of the error. Each repair has meant inserting or amending one line.

REFERENCES

1. G. M. Birtwistle, "Discrete event modelling on Simula", Macmillan, 1979. Distributed in North America by Gage Publishing Ltd., Agincourt, Ontario.

2. G. M. Birtwistle, O. J. Dahl, B. Myhrhaud and K. Nygaard, "Simula begin", Studentlitteratur, Lund, Sweden, 1973.

3. G. H. Hardy and E. M. Wright, "The theory of numbers", Clarendon Press, Oxford, 1960.

4. F. D. K. Roberts, "Pseudo random number generators", M.Sc. Thesis, Liverpool, 1966.

5. D. E. Knuth, "The art of computer programming", vol. 2, Addison-Wesley, 197 .

6. D. Y. Downham and F. D. K. Roberts, "Multiplicative congruential pseudo-random number generators", Computer Journal, vol. 10, no. 1, 1967.

7. M. Ohlin, "Next random - a method of fast access to any number in the random generator cycle", Simula Newsletter, vol. 6, no. 2, 1977.

8. G. M. Birtwistle, "Demos Reference Manual", 1979. Available from the author. Supplied in machine readable form with the Demos system.