```
#define m1 2147483647
#define m2 2145483479
#define a12 63308
#define a13 −183326
#define a21 86098
#define a23 −539608
#define q12 33921
#define q13 11714
#define q21 24919
#define q23 3976
#define r12 12979
#define r13 2883
#define r21 7417
#define r23 2071
#define Invmp1 4.656612873077393e−10;
int x10, x11, x12, x20, x21, x22;

int     Random()
        {
        int h, p12, p13, p21, p23;
        /* Component 1 */
        h = x10/q13; p13 = −a13*(x10−h*q13)−h*r13;
        h = x11/q12; p12 = a12*(x11−h*q12)−h*r12;
        if(p13<0) p13 = p13+m1; if(p12<0) p12 = p12+m1;
        x10 = x11; x11 = x12; x12 = p12−p13; if(x12<0) x12 = x12+m1;
        /* Component 2 */
        h = x20/q23; p23 = −a23*(x20−h*q23)−h*r23;
        h = x22/q21; p21 = a21*(x22−h*q21)−h*r21;
        if(p23<0) p23 = p23+m2; if(p21<0) p21 = p21+m2;
        /* Combination */
        if (x12<x22) return (x12−x22+m1); else return (x12−x22);
        }

double Uniform01()
        {
        int Z;
        Z=Random(); if(Z==0) Z=m1; return (Z*Invmp1);
        }
```

Fig. 2.3. Implementation in C of a combined multiple recursive generator using integer arithmetic. The generator and the implementation are from L'Ecuyer [224].

with all a_i equal to 0 or 1. This method was proposed by Tausworthe [346]. It can be implemented through a mechanism known as a *feedback shift register*. The implementation and theoretical properties of these generators (and also of *generalized* feedback shift register methods) have been studied extensively. Matsumoto and Nishimura [258] develop a generator of this type with a period of $2^{19937} - 1$ and apparently excellent uniformity properties. They provide C code for its implementation.

Inversive congruential generators use recursions of the form

$$x_{i+1} = (a x_i^- + c) \bmod m,$$

where the (mod m)-inverse x^- of x is an integer in $\{1, \ldots, m-1\}$ (unique if it exists) satisfying $xx^- = 1 \bmod m$. This is an example of a nonlinear congruential generator. Inversive generators are free of the lattice structure

```
double s10, s11, s12, s13, s14, s20, s21, s22, s23, s24;

#define norm 2.328306549295728e-10
#define m1 4294949027.0
#define m2 4294934327.0
#define a12 1154721.0
#define a14 1739991.0
#define a15n 1108499.0
#define a21 1776413.0
#define a23 865203.0
#define a25n 1641052.0

double MRG32k5a ()
        {
        long k;
        double p1, p2;
        /* Component 1 */
        p1 = a12 * s13 − a15n * s10;
        if (p1 > 0.0) p1 −= a14 * m1;
        p1 += a14 * s11; k = p1 / m1; p1 −= k * m1;
        if (p1 < 0.0) p1 += m1;
        s10 = s11; s11 = s12; s12 = s13; s13 = s14; s14 = p1;
        /* Component 2 */
        p2 = a21 * s24 − a25n * s20;
        if (p2 > 0.0) p2 −= a23 * m2;
        p2 += a23 * s22; k = p2 / m2; p2 −= k * m2;
        if (p2 < 0.0) p2 += m2;
        s20 = s21; s21 = s22; s22 = s23; s23 = s24; s24 = p2;
        /* Combination */
        if (p1 <= p2) return ((p1 − p2 + m1) * norm);
        else return ((p1 − p2) * norm);
        }
```

Fig. 2.4. Implementation in C of a combined multiple recursive generator using floating point arithmetic. The generator and implementation are from L'Ecuyer [225].

characteristic of linear congruential generators but they are much more computationally demanding. They may be useful for comparing results in cases where the deficiencies of a random number generator are cause for concern. See Eichenauer-Herrmann, Herrmann, and Wegenkittl [110] for a survey of this approach and additional references.

2.2 General Sampling Methods

With an introduction to random number generation behind us, we henceforth assume the availability of an ideal sequence of random numbers. More precisely, we assume the availability of a sequence $U_1, U_2, \ldots$ of independent random variables, each satisfying

$$P(U_i \le u) = \begin{cases} 0, & u < 0 \\ u, & 0 \le u \le 1 \\ 1, & u > 1 \end{cases} \qquad (2.12)$$