



ARL-TR-7928 • JAN 2017



US Army Research Laboratory

The Evolution of Random Number Generation in MUVES

by Joseph C Collins

Approved for public release; distribution is unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.



The Evolution of Random Number Generation in MUVES

by Joseph C Collins
Survivability/Lethality Analysis Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) January 2017		2. REPORT TYPE Technical Report		3. DATES COVERED (From - To) October 2013–December 2016	
4. TITLE AND SUBTITLE The Evolution of Random Number Generation in MUVES				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Joseph C Collins				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-SLB-D Aberdeen Proving Ground, MD 21005-5068				8. PERFORMING ORGANIZATION REPORT NUMBER ARL-TR-7928	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES Author's Email: <joseph.c.collins38.civ@mail.mil>					
14. ABSTRACT The evolution of random number generation in MUVES proceeds from a short-period low-resolution single-threaded legacy implementation with questionable numerical and statistical properties. The development of the modern system is traced through software change requests, resulting in a random number generator that overcomes all shortcomings of the legacy system. This report traces the history of random number generation in MUVES, including the mathematical basis and statistical justification for algorithms used in the code. The working code provided produces results identical to the current implementation. These theoretical and practical details enable the reader to understand the algorithms and ensure that future enhancements to the production code preserve the integrity of the system.					
15. SUBJECT TERMS MUVES, random number generator, parallel processing, thread safe, statistical independence					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 50	19a. NAME OF RESPONSIBLE PERSON Joseph C Collins
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (Include area code) 410-278-6832

Contents

List of Figures	v
1. Introduction	1
2. Legacy	1
2.1 Integers	2
2.2 Linear Congruential Generator	2
2.3 Independent RNG Streams	4
2.4 Jumping Ahead for LCGs	4
2.5 Legacy Issues	6
3. Linear Feedback Shift Register	8
3.1 Definition	8
3.2 Matrix Representation	8
3.3 Recurrence Relations	9
3.4 QuickTaus Trinomial LFSR	10
3.5 Characteristic Polynomial	11
3.6 State Details	11
3.7 Jump	12
3.8 Jump Example	12
4. T258	13
4.1 Implementation	13
4.2 The Uniform(0,1) Distribution	15
4.3 Initialization	15
4.4 Seeding	16
4.5 Parallel Processing	16
4.6 Run Time	17
5. SCR 1049–1050	17
6. SCR 1908	18

6.1	Consequences	19
6.1.1	Jump Computation Failure	19
6.1.2	Stream independence Failure	19
7.	SCR 2138	27
7.1	Identity	28
7.2	Implementation	28
7.3	Jump Verification	29
8.	SCR 2142	30
9.	Demonstration and Verification	31
9.1	T258 Class Code	34
9.2	Driver Code	36
10.	Conclusions and Recommendations	37
11.	References	38
	List of Symbols, Abbreviations, and Acronyms	39
	Distribution List	40

List of Figures

Fig. 1	<i>b</i> series, no offset, same scale.....	21
Fig. 2	<i>b</i> series, no offset, detail	22
Fig. 3	<i>b</i> series, arbitrary offset, same scale	23
Fig. 4	<i>b</i> series, arbitrary offset, detail	24
Fig. 5	<i>b, m, v</i> CDFs, no offset	25
Fig. 6	<i>b, m, v</i> CDFs, arbitrary offset.....	26

INTENTIONALLY LEFT BLANK.

1. Introduction

A random number generator (RNG) is an algorithm with output that is in some sense statistically indistinguishable from a random sample. This is an essential component of any stochastic simulation (such as MUVES), which relies on the availability of independent random quantities for statistical validity. Parallel (distributed) processing adds another layer to this requirement. The sets of random quantities in distinct threads must also be independent of each other. Otherwise, no sets of quantities within or among threads can be claimed to be a random sample.

The evolution of random number generation in MUVES over the past 25 years, from 1990 to 2015, proceeds from a short-period low-resolution single-threaded implementation with questionable numerical and statistical properties. This legacy system is documented in Section 2.

The modern concept presented in Sections 3 and 4 overcomes all the shortcomings of the legacy system and provides independent sets of random numbers for thread-safe parallel processing. Details of the MUVES implementation development through software change requests are seen in Sections 5, 6, 7, and 8. Working algorithm code and a procedure for verification of the statistical independence properties are in Section 9.

This report traces the history of random number generation in MUVES, including the mathematical basis and statistical justification for algorithms used in the code. The working code provided produces results identical to the current implementation. These theoretical and practical details enable the reader to understand the algorithms and ensure that future enhancements to the production code preserve the integrity of the system.

2. Legacy

Parallel processing was not a design consideration when MUVES development began. So the legacy RNG was inherently single-threaded. The main concern was portability, but the legacy RNG is deficient in several other respects. Nonetheless, it is instructive to consider its construction and operation. Sections 2.1–2.5 describe the mathematics behind the legacy RNG implementation, and point out undesirable features and areas that need improvement.

2.1 Integers

Integer addition, subtraction, and multiplication are intrinsic, but division is characterized by the division algorithm: one can divide any a by any nonzero m to get a unique quotient q and the remainder r . To be specific,

$$\forall a \forall m > 0 \exists! q \exists! r : a = qm + r, 0 \leq r < m. \quad (1)$$

(The quotient and remainder are made unique by the bounds on r .) The remainder or “mod” operator “%” expresses this relationship, and we say that “ a mod m equals r ”, denoted $a \% m = r$. When the remainder is 0, we say that “ m divides a ”, denoted $m \mid a$.

Some useful properties of the remainder are

$$(a \pm b) \% m = (a \% m \pm b \% m) \% m \quad (2)$$

which follows from $a \pm b = (q_a \pm q_b)m + (r_a \pm r_b)$,

$$(ab) \% m = (a \% m \cdot b \% m) \% m \quad (3)$$

which follows from $ab = (q_a m + r_a)(q_b m + r_b) = qm + r_a r_b$, and

$$b \mid a \implies \frac{a}{b} \% m = \frac{a \% (bm)}{b} \quad (4)$$

which follows from $a/b = qm + r$, $a = q(bm) + br$, and $a \% (bm) = br$.

2.2 Linear Congruential Generator

The Linear Congruential Generator (LCG) is a basic RNG is defined by a linear recurrence modulo m

$$x_{i+1} = T(x_i) = (ax_i + c) \% m \quad (5)$$

with integer and real output functions $z(x_i)$ and $u(z_i)$. The RNG state x is not necessarily the same thing as the integer output $z(x)$.

An LCG has full period m , obtaining all values in $0, \dots, m - 1$, if and only if

$$\begin{aligned} & m \text{ and } c \text{ are relatively prime} \\ & \forall \text{ prime } q : q \mid m \implies q \mid (a - 1) \\ & 4 \mid m \implies 4 \mid (a - 1). \end{aligned} \tag{6}$$

The MUVES legacy RNG is a full-period LCG with

$$\begin{aligned} m &= 2^{32} \\ a - 1 &= 1103515244 = 2^2 \cdot 13^2 \cdot 613 \cdot 2663 \\ c &= 12345 = 3 \cdot 5 \cdot 823. \end{aligned} \tag{7}$$

The integer output function is

$$z = (x/65536)\%32768 = (x \gg 16) \& 0x7fff, \tag{8}$$

giving a 15-bit integer, $0 \leq z \leq 2^{15} - 1 = 32767$. “Right shift” is “ \gg ” and “bitwise and” is “ $\&$.”

The real output function is

$$u = z/32768 = z/2^{15}, \tag{9}$$

approximating $U(0, 1)$ with 4 fully significant digits and resolution $\varepsilon \approx 3 \times 10^{-5}$.

The integer output z is the same as `rand()`, the C library standard RNG in Kernighan and Ritchie.¹ The “rand” man page offers a warning about this function.

The versions of `rand()` and `srand()` in the Linux C Library use the same random number generator as `random(3)` and `srandom(3)`, so the lower-order bits should be as random as the higher-order bits. However, on older `rand()` implementations, and on current implementations on different systems, the lower-order bits are much less random than the higher-order bits. Do not use this function in applications intended to be portable when good randomness is needed. (Use `random(3)` instead.)

See Collins² for randomness test results, including the failure of this function.

2.3 Independent RNG Streams

Any RNG cycle can be used to create statistically independent RNGs by 2 fundamental methods. The partition method is to divide the cycle into non-overlapping streams of consecutive elements, considered to be mutually independent. This is accomplished by advancing (jumping) the RNG in fixed increments (jumps) to obtain the stream starting points. To get n streams from a RNG with period m , use the jump m/n . The other method is leapfrog, where the n stream starting points are consecutive states s_1, \dots, s_n from the RNG cycle. Then the RNG is used with a jump of n in each stream to obtain non-intersecting interleaved streams, considered to be mutually independent. In either case the effective useful stream length or period is m/n , which is exhausted when one stream collides with another.

MUVES uses the partition method and 8 streams, 4 denoted “Shot Pattern”, “Cluster”, “Shot Assessment”, “BAD”, and 4 unused. First, seed the RNG with a 32-bit integer s_1 , this is stream 1. Then the other 7 streams are obtained by jumping ahead $k = 2^{29} = 2^{32}/8 = 536870912$ in the cycle, starting at $s_i = T^k(s_{i-1})$ for $i = 2, \dots, 8$. To use the streams, apply the single-step (jump 1) transition $T(x)$ in each stream.

To implement the leapfrog method, seed the RNG with a 32-bit integer s_1 for stream 1. Then the single-step (jump 1) transition starts the other 7 streams at $s_i = T(s_{i-1})$ for $i = 2, \dots, 8$. To use the streams, apply the jump 8 transition $T^8(x)$ in each stream. MUVES does not use leapfrog, but the concept appears later in another context.

2.4 Jumping Ahead for LCGs

Either method is useful (and easy to implement) for an LCG, because repeated application of the LCG transition $T(x) = (ax + c) \% m$ of Eq. 5 is another LCG

$$T^k(x) = (a_k x + c_k) \% m. \quad (10)$$

To compute the coefficients of the jump LCG, note that modulo m ,

$$\begin{aligned} x_1 &= T(x_0) = ax_0 + c \\ x_2 &= T^2(x_0) = ax_1 + c = a(ax_0 + c) + c = a^2x_0 + ac + c \\ x_3 &= T^3(x_0) = a^3x_0 + a^2c + ac + c = a^3x_0 + (a^2 + a + 1)c \end{aligned} \quad (11)$$

so in general (mod m),

$$x_k = T^k(x_0) = a_k x_0 + c_k = a^k x_0 + \sum_{i=0}^{k-1} a^i \cdot c = a^k x_0 + \frac{a^k - 1}{a - 1} \cdot c. \quad (12)$$

Apply the properties of Section 2.1 to see that

$$a_k = (a^k) \% m \quad \text{and} \quad c_k = \left[\frac{(a^k - 1) \% [(a - 1)m]}{a - 1} \cdot c \right] \% m. \quad (13)$$

High powers of any w can be computed with “exponentiation by squaring.” One can obtain w^{2^p} by starting with w and recursively squaring p times, each doubling the exponent. The sequence so obtained is $w, w^2, w^4, w^8, \dots, w^{2^p}$ since $w = w^1 = w^{2^0}$ and

$$(w^{2^p})^2 = w^{2 \cdot 2^p} = w^{2^{p+1}}. \quad (14)$$

In other words, square w recursively p times to get w^{2^p} .

The following C code computes LCG coefficients for $T^k = a_k x + c_k$ where $k = 2^p$ and $p = 0, \dots, 32$ based on $T(x) = ax + c$ with MUVES LCG parameters.

```
#include <stdio.h>
#include <stdint.h>
int main() {
    long unsigned a=1103515245, c=12345, m=1L<<32, ak[33], ck[33], p;
    __uint128_t b;
    for(b=ak[0]=a, ck[0]=c, p=1; p<=32; p++) {
        ak[p] = ( ak[p-1] * ak[p-1] ) % m;
        b = ( b * b ) % ((a-1)*m);
        ck[p] = ( (b-1) % ((a-1)*m) / (a-1) * c ) % m;
    }
    for(p=0; p<=32; p++)
        printf("k = 2^%-2d = %10lu : ak = %10u ck = %10u\n",
            p, 1UL<<p, ak[p], ck[p]);
}
```

Note $(a^{2^{p-1}})^2 = a^{2^p}$ in the computation of $(a^{2^p}) \% m$, and $b = (a^{2^p}) \% [(a - 1)m]$. The latter needs 128-bit integers.

The output is

```
k = 2^0 =          1 : ak = 1103515245 ck =          12345
```

k = 2 ¹ =	2	: ak = 3265436265	ck = 3554416254
k = 2 ² =	4	: ak = 3993403153	ck = 3596950572
k = 2 ³ =	8	: ak = 3487424289	ck = 3441282840
k = 2 ⁴ =	16	: ak = 1601471041	ck = 1695770928
k = 2 ⁵ =	32	: ak = 2335052929	ck = 1680572000
k = 2 ⁶ =	64	: ak = 1979738369	ck = 422948032
k = 2 ⁷ =	128	: ak = 387043841	ck = 3058047360
k = 2 ⁸ =	256	: ak = 3194463233	ck = 519516928
k = 2 ⁹ =	512	: ak = 3722397697	ck = 530212352
k = 2 ¹⁰ =	1024	: ak = 1073647617	ck = 2246364160
k = 2 ¹¹ =	2048	: ak = 2432507905	ck = 646551552
k = 2 ¹² =	4096	: ak = 1710899201	ck = 3088265216
k = 2 ¹³ =	8192	: ak = 3690233857	ck = 472276992
k = 2 ¹⁴ =	16384	: ak = 4159242241	ck = 3897344000
k = 2 ¹⁵ =	32768	: ak = 4023517185	ck = 2425978880
k = 2 ¹⁶ =	65536	: ak = 3752067073	ck = 556990464
k = 2 ¹⁷ =	131072	: ak = 3209166849	ck = 1113980928
k = 2 ¹⁸ =	262144	: ak = 2123366401	ck = 2227961856
k = 2 ¹⁹ =	524288	: ak = 4246732801	ck = 160956416
k = 2 ²⁰ =	1048576	: ak = 4198498305	ck = 321912832
k = 2 ²¹ =	2097152	: ak = 4102029313	ck = 643825664
k = 2 ²² =	4194304	: ak = 3909091329	ck = 1287651328
k = 2 ²³ =	8388608	: ak = 3523215361	ck = 2575302656
k = 2 ²⁴ =	16777216	: ak = 2751463425	ck = 855638016
k = 2 ²⁵ =	33554432	: ak = 1207959553	ck = 1711276032
k = 2 ²⁶ =	67108864	: ak = 2415919105	ck = 3422552064
k = 2 ²⁷ =	134217728	: ak = 536870913	ck = 2550136832
k = 2 ²⁸ =	268435456	: ak = 1073741825	ck = 805306368
k = 2 ²⁹ =	536870912	: ak = 2147483649	ck = 1610612736
k = 2 ³⁰ =	1073741824	: ak =	1 ck = 3221225472
k = 2 ³¹ =	2147483648	: ak =	1 ck = 2147483648
k = 2 ³² =	4294967296	: ak =	1 ck = 0

See the MUVES source code `Rn/RnLegacy.cpp`, where a_k and c_k for $k = 2^{29}$ are documented as “magic beans”.

2.5 Legacy Issues

The legacy RNG fails most common tests of randomness. This alone is a reason to reject the legacy RNG outright.

The period of $2^{29} \approx 5 \times 10^8$ is too short. At a rate of 9 million random number draws per second this is exhausted in 60 seconds. In terms of how many random

quantities MUVES uses, consider an analysis with 1000 cells in a view and 1000 BAD fragments per shot using 8 threats and 8 velocities. With 9 iterations, the number of fragments has already exceeded the stream length.

The resolution of 4 digits (15 bits) does not adequately cover the range of double-precision uniform values on the unit interval. Full IEEE double resolution requires 15 digits (53 bits).

Every random quantity needs its own stream for true independence. The 8 streams available in the legacy implementation place an unreasonably low limit on the possible number of independent stochastic quantities.

The legacy implementation provides a single set of random quantities, and as such is not thread-safe or suitable for parallel processing. Independent shotlines need independent sets of independent quantities.

We need a high-quality fast 64-bit RNG with a huge period that can easily be partitioned (into streams) for independent parallel processing and the streams partitioned (into substreams) for independent quantities.

Such a system is the topic of the next section.

3. Linear Feedback Shift Register

Collins² documents the details of T258, the RNG currently implemented in MUVES, based on the linear feedback shift register (LFSR). For clarity some information is repeated in the following along with subsequent developments. Following the definition, the (equivalent) matrix and recurrence relation representations give concrete examples of general LFSR implementation, but MUVES uses neither. Instead, an efficient algorithm (QT) can be used for the particular class of LFSRs in T258.

3.1 Definition

\mathbb{F}_2 is the finite field with 2 elements $\{0, 1\}$ which are equivalent to bits. In \mathbb{F}_2 addition is subtraction as $1 + 1 = 0$ and $1 = -1$. Let x_0, x_1, x_2, \dots be a sequence from \mathbb{F}_2 . An LFSR sequence obeys some recurrence with $c_i \in \mathbb{F}_2$

$$x_{n+k} = \sum_{i=0}^{k-1} c_i x_{n+i} = c_{k-1} x_{n+k-1} + \dots + c_1 x_{n+1} + c_0 x_n \quad (15)$$

so any bit is determined by the previous k bits. Blocks of L bits can yield L -bit integers z with $0 \leq z < 2^L$ via the appropriate output function

$$z = \sum_{i=0}^{L-1} 2^{L-1-i} x_{n+i} = 2^{L-1} x_n + \dots + 2^0 x_{n+L-1} \quad (16)$$

or real numbers u with $0 \leq u < 1$ via the output function $u = z/2^L$

$$u = \sum_{i=0}^{L-1} 2^{-i-1} x_{n+i} = 2^{-1} x_n + \dots + 2^{-L} x_{n+L-1}. \quad (17)$$

3.2 Matrix Representation

The matrix representation uses a k -bit nonzero state vector $X_n = (x_{n,0}, \dots, x_{n,k-1})$ and a $k \times k$ transition matrix A , both with components in \mathbb{F}_2 . The transition recurrence is $X_{n+1} = AX_n$. The shift is $x_{n+1,i} = x_{n,i+1}$ for $i = 0, \dots, k-2$, and the last component $x_{n+1,k-1}$ of X_{n+1} is determined by the recurrence of Eq. 15. Note that in general $X_{n+k} = A^k X_n$. Here, A implements $x_{k+4} = x_{k+2} + x_k$ starting with X_0 ,

where

$$A = \left[\begin{array}{c|ccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 0 \end{array} \right] \quad \text{and} \quad X_0 = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}. \quad (18)$$

Then X_1, \dots, X_5 are seen to be

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_2 + x_0 \end{bmatrix}, \begin{bmatrix} x_2 \\ x_3 \\ x_2 + x_0 \\ x_3 + x_1 \end{bmatrix}, \begin{bmatrix} x_3 \\ x_2 + x_0 \\ x_3 + x_1 \\ x_0 \end{bmatrix}, \begin{bmatrix} x_2 + x_0 \\ x_3 + x_1 \\ x_0 \\ x_1 \end{bmatrix}, \begin{bmatrix} x_3 + x_1 \\ x_0 \\ x_1 \\ x_2 \end{bmatrix}. \quad (19)$$

The matrix A has characteristic polynomial $P(z) = \det(zI - A) = z^4 - z^2 - 1$. Since $P(A) = 0$, we have $A^4 = A^2 + I$ and in general $A^{k+4} = A^{k+2} + A^k$. Note that $X_4 = X_2 + X_0$ and $X_5 = X_3 + X_1$ and in general $X_{k+4} = X_{k+2} + X_k$. So x_n and X_n and A^n follow the same recurrence.

We never use the matrix implementation.

3.3 Recurrence Relations

Any matrix A has characteristic polynomial (CP)

$$C(z) = \det(zI - A) = z^k - c_{k-1}z^{k-1} - \dots - c_1z - c_0. \quad (20)$$

The Cayley-Hamilton theorem assures that $C(A) = 0$, therefore

$$A^k = c_{k-1}A^{k-1} + \dots + c_1A + c_0I. \quad (21)$$

$C(z)$ is also the CP of the recurrence $X_{n+1} = AX_n$, consider $X_{n+k} = A^k X_n$, so

$$X_{n+k} = c_{k-1}X_{n+k-1} + \dots + c_1X_{n+1} + c_0X_n. \quad (22)$$

The recurrence gives the sequence of vectors X_n , where each $c_i \in \mathbb{F}_2$, and applies to each coordinate (bit) of X_n .

We never use the recurrence to implement the RNG.

3.4 QuickTaus Trinomial LFSR

L'Ecuyer's QuickTaus (QT) algorithm³ uses a trinomial recurrence $P(z) = z^k - z^q - 1$ to generate a (horizontal) bit stream in blocks of s bits according to $b_{n+k} = b_{n+q} + b_n$. Word size is L bits. A , B , and C are words. A is the LFSR state. C is a mask of k ones, then $L - k$ zeros.

The QT algorithm updates A .

```
B = A << q;      // q-bit left-shift of A
B = A ^ B;       // A xor B, bitwise
B = B >> (k-s);  // (k-s)-bit right-shift of B
A = A & C;       // A and C, bitwise
A = A << s;      // s-bit left-shift of A
A = A ^ B;       // A xor B, bitwise
```

The C/C++ implementation of QT is succinct.

```
C = -0x1 << (L-k);
A = ( ( A & C ) << s ) ^ ( ( A << q ) ^ A ) >> (k - s) );
```

Note the use of the “twos complement” negative integer representation. A negative integer is stored as the twos complement of its absolute value, which is the sum of its ones complement (all bits reversed) and $0x1$. So $-0x1 = 0xffff\dots fff$.

An example illustrates the action of QT. Consider $P(z) = z^{28} - z^3 - 1$, so $k = 28$ and $q = 3$. The bit recurrence is $b_{n+28} = b_{n+3} + b_n$. The block size is $s = 17$, the word size is $L = 32$. The mask is $C = -0x10 = 0xfffffff0$, which has $k = 28$ ones followed by $L - k = 4$ trailing zeros. So the C/C++ implementation is

```
x = ( ( x & -0x10 ) << 17 ) ^ ( ( x << 3 ) ^ x ) >> 11 );
```

In detail, step by step:

```
x          ; // ( x0..x28 , x29..x31 )  $z^0 = 1$ , initial state satisfies  $P(z)$ 
y = x << 3 ; // ( x3..x31 , 3 @ 0 )  $z^3$ , step forward 3
y = y ^ x  ; // ( x28..x56 , x29..x31 ) by the construction,  $z^3 + z^0 = z^{28}$ 
y = y >> 11; // ( 11 @ 0 , x28..x48 ) tail=new block of s bits, 32 to 48
x = x & C   ; // ( x0..x27 , 4 @ 0 ) make room for new block in x
x = x << 17; // ( x17..x27 , 21 @ 0 ) pop s bits
x = x ^ y   ; // ( x17..x27 , x28..x48 ) combine bitwise: x+y mod 2
```

This generates bits “horizontally” in blocks of 17, and successive words are

```
x      = ( x0 ... x14 , x15 ... x31 )
A(x)  = ( x17 ... x31 , x32 ... x48 )
```

3.5 Characteristic Polynomial

Note that columns are “vertically” leapfrog (by 17) sequences from $P(z)$. Using any bit position $x[b]$ vertically from the sequence $\mathbf{x}=\mathbf{A}(\mathbf{x})$, the characteristic polynomial can be computed as detailed in Collins.²

$$C(z) = \sum_{i=0}^{28} c_i z^i = z^{28} + z^{19} + z^{17} + z^{15} + z^{10} + z^6 + z^3 + z^2 + 1. \quad (23)$$

$C(z)$ is also the characteristic polynomial, hence the recurrence, of the words themselves. Only $2k$ bits are required for this operation, making the LFSR cryptographically useless. From the $2k$ bits x_0, \dots, x_{2k-1} , construct the $k + 1$ vectors $X_i = (x_i, \dots, x_{i+k-1})$ each of length k for $i = 0, \dots, k$. Since X_k is a linear combination of the previous k vectors X_0, \dots, X_{k-1} , the solution of the linear system

$$\begin{bmatrix} x_k \\ x_{k+1} \\ x_{k+2} \\ \vdots \\ x_{2k-1} \end{bmatrix} = \begin{bmatrix} x_{k-1} & \cdots & x_2 & x_1 & x_0 \\ x_k & \cdots & x_3 & x_2 & x_1 \\ x_{k+1} & \cdots & x_4 & x_3 & x_2 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ x_{2k-2} & \cdots & x_{k+1} & x_k & x_{k-1} \end{bmatrix} \begin{bmatrix} c_{k-1} \\ c_{k-2} \\ c_{k-3} \\ \vdots \\ c_0 \end{bmatrix} \quad (24)$$

provides the coefficients of $C(z)$, along with $c_k = 1$ as $c_k X_k = \sum_{i=0}^{k-1} c_i X_i$. The Berlekamp-Massey algorithm is an efficient algorithm for solving this system of equations. Also, in each row ($j = 0, \dots, k - 1$) we see

$$x_{k+j} = \sum_{i=0}^{k-1} c_i x_{i+j}. \quad (25)$$

3.6 State Details

$P(z)$ generates horizontal bit stream computed in 17-bit blocks:

```
101000000111100010011010000110100011101111000100101 ...
10100000011110001 00110100001101000 11101111000100101 ...
```

Shift to fill words, new block on the right, (partial) old block shifted left:

```
010010110000100 10100000011110001
100000011110001 00110100001101000
1110100001101000 11101111000100101
...
```

$C(z)$ generates each vertical bitstream and also the words:

```
010010111000010010100000011110001 = 0x4b0940f1
10000001111000100110100001101000 = 0x81e26868
11010000110100011101111000100101 = 0xd0d1de25
...
```

3.7 Jump

The division algorithm for polynomials is analogous to the integer division algorithm: Divide z^d by $C(z)$ to get the quotient $Q_d(z)$ and remainder $J_d(z)$, which satisfy $z^d = Q_d(z)C(z) + J_d(z)$ where $J_d = 0$ or $0 \leq \deg J_d < \deg C = k$. The remainder is the jump polynomial with coefficients in $\mathbb{F}_2 = \{0, 1\}$: $J_d(z) = z^d \bmod C(z) = \sum_{i=0}^{k-1} j_i z^i$. We know $C(A) = 0$, therefore $A^d = J_d(A)$, and

$$x_{n+d} = A^d x_n = J_d(A) x_n = \sum_{i=0}^{k-1} j_i x_{n+i}. \quad (26)$$

Thus, a future (jump) state is some linear combination of only k successive states. To compute J_{2^p} , start with $m = 0$ and apply p iterations of squaring mod $C(z)$

$$z^{2^{m+1}} \bmod C(z) = \left(z^{2^m} \right)^2 \bmod C(z). \quad (27)$$

Each step is an application of the division algorithm via synthetic division in \mathbb{F}_2 using Knuth's algorithm referenced in Collins.²

3.8 Jump Example

The state transition is $x = A(x)$ where

```
unsigned A ( unsigned x ) {
    return ( ( x & -0x10 ) << 17 ) ^ ( ( ( x << 3 ) ^ x ) >> 11 );
}
```

For $p = 64$, the 2^p jump polynomial for our example is

```
J(z) = z27 + z26 + z25 + z24 + z22 + z19 + z18 + z15 + z11 + z9 + z8 + z6 + z5 + z2 + 1
J = 0x0f4c8b65 = 0000 1111 0100 1100 1000 1011 0110 01012
```

To apply to state x , construct the \mathbb{F}_2 -linear combination of successive states:

```
unsigned j, t=x, y=0;
for(j=J; j; j>>=1, t=A(t)) if(j & 1) y ^= t;
```

Then $y = A^{2^p} x = A^{2^{64}} x = A^{18446744073709551616} x$.

State vectors and polynomials coefficients (0 or 1) are packed into integers.

XOR (^) is \mathbb{F}_2 vector space addition.

4. T258

In practice, a single LFSR is not good enough. MUVES implements the RNG algorithm T258,² an extension of L'Ecuyer's LFSR258⁴ providing probabilistically independent sets of random vectors and suitable for parallel processing (thread-safe).

T258 uses a set of five 64-bit LFSRs, each implementing a recurrence defined by a primitive polynomial of the form $P(z) = z^k - z^q - 1$ in bit blocks of size s by means of the QT algorithm.

Each LFSR is implemented with QT transition

$$x = ((x \& C) \ll s) \wedge ((x \ll q) \wedge x) \gg (k - s);$$

The QT parameters for the 5 components of T258 are

#	L	k	q	s	k-s	L-k	M = -C
0	64	63	1	10	53	1	0x000002
1	64	55	24	5	50	9	0x000200
2	64	52	3	29	23	12	0x001000
3	64	47	5	23	24	17	0x020000
4	64	41	3	8	33	23	0x800000

The LFSR periods are $P_i = 2^{k_i} - 1$ for $k \in \{63, 55, 52, 47, 41\}$,

The P_i are pairwise relatively prime, so the period of T258 is $P = \prod_{i=1}^5 P_i \approx 2^{258}$.

4.1 Implementation

For consistent presentation hereafter, we use a C++ class to implement T258. Complete current code is in Section 9. This is not production MUVES code, but the algorithms are identical.

```
typedef uint64_t uz;
class rng { // T258
public: // ...
    rng(uz seed); // constructor
    rng& init(); // (re)initialize
    rng& t(); // state transition
```

```

    rng& jump(uz p);           // jump 2^p
    rng& jumpk(uz k, uz p);   // jump k*2^p
    uz gen();                 // integer generator
    double u01();            // u(0,1) generator
    static const int JS = 86; // log_2 ( small jump )
    static const int JL = 172; // log_2 ( large jump )
    bool operator!=(const rng& w); // state comparison
    bool operator==(const rng& w); // state comparison
private:
    uz s[5];                 // LFSR state array
    uz seed;                 // this.seed
    void seedx(void);        // set seed
    static const int CPd[5]; // CP degrees
    static const uz JP[63][5]; // jump polynomials
};

```

Representation of the T258 state transition is

```

rng& rng::t() { // T258:      C      s          q          k-s
    s[0] = ((s[0] & -0x000002) << 10) ^ (((s[0] << 1) ^ s[0]) >> 53);
    s[1] = ((s[1] & -0x000200) << 5) ^ (((s[1] << 24) ^ s[1]) >> 50);
    s[2] = ((s[2] & -0x001000) << 29) ^ (((s[2] << 3) ^ s[2]) >> 23);
    s[3] = ((s[3] & -0x020000) << 23) ^ (((s[3] << 5) ^ s[3]) >> 24);
    s[4] = ((s[4] & -0x800000) << 8) ^ (((s[4] << 3) ^ s[4]) >> 33);
    return *this;
}

```

The 64-bit unsigned integer function gen() returns values in $0, \dots, 2^{64} - 1$.

```

uz rng::gen() { // 64-bit integer
    t(); // transition
    return s[0] ^ s[1] ^ s[2] ^ s[3] ^ s[4];
}

```

The function u01() is a discrete version of the continuous $U(0, 1)$ distribution.

```

double rng::u01() { // U(0,1)
    uz z;
    const static double d = 1.0/(1UL<<53); // 1/2^53
    do z = gen() >> 11; while(!z);
    return d * z;
}

```

4.2 The Uniform(0,1) Distribution

The real function `u01()` uses 53 bits of the full 64-bit integer to compute a discrete version of the continuous $U(0, 1)$ distribution with evenly-spaced equiprobable output $i/2^{53}$ for $i = 1, 2, 3, \dots, 2^{53} - 1$. These are the expected values of $U(0, 1)$ order statistics for a sample of size $2^{53} - 1$ and yield the correct discrete approximation. Note the symmetry in the sense that `u01()` and `1-u01()` have the same distribution. In fact, 53 is the largest number with properties, as the internal IEEE representation of `double` uses 53 bits implicitly.

It may be tempting to obtain more “accuracy” by using all 64 bits and dividing by 2^{64} , but this is an error. The resulting distribution will not have equiprobable evenly-spaced values and will not be symmetric. Eventually, some such $x > 0$ will be so small that $1 - x = 1$. The program will crash on something like `log(1 - x)` expecting that both $0 < x < 1$ and $0 < 1 - x < 1$.

4.3 Initialization

L’Ecuyer³ states a condition required for QT, that the initial state must be a valid recurrence element:

“For this algorithm to work properly, A must be initialized correctly with a valid initial S_0 ; that is, which agrees with the recurrence.

[General initialization algorithm omitted.]

If the additional condition $L - k \leq r - s$ is satisfied, then it can be easily verified that after the first pass through the six steps of QuickTaus, A will necessarily contain a valid state, even if the initial state S_0 was not valid. In that case, the above initialization procedure is not necessary for running the generator; just skip the first value.”

For T258, the additional condition is satisfied since $r = k - q$, the general initialization algorithm is not required, and correct operation is obtained by “skipping the first value”, implemented in the initialization code as a single call to QT with its return value discarded.

Without this initialization feature, computation of the $C(z)$ fails. The results are random depending on the (incorrect) initial state. Consequently, computation of the $J(z)$ also fails, since these depend on the $C(z)$. In the current implementation the

$C(z)$ are not used, and the $J(z)$ are pre-computed and stored in tables, so this is not an issue. However, jump computation fails even with correct $J(z)$.

Moreover, if the implementation is ever changed to compute new $J(z)$ for different jump sizes, include dynamic computation of $C(z)$ and $J(z)$, replace the RNG, or perform diagnostic tests or verification and validation, etc., the condition is necessary or else the computations will fail.

4.4 Seeding

L'Ecuyer⁴ presents conditions for correct seeding of LFSR RNGs.

“Before calling `lfsr113` for the first time, the variables z_1 , z_2 , z_3 , and z_4 must be initialized to any (random) integers larger than 1, 7, 15, and 127, respectively. In other words, the k_j most significant bits of z_j must be nonzero, for each j .

Ideally, the vector of initial seeds (z_1, \dots, z_j) would be drawn from a uniform distribution over the set of admissible values.”

Minimum values for T258 seed states are denoted as $M = -C$ in the parameter table on page 13. If $x < M$ then $QT(x) = 0$, and the LFSR is stuck at 0. This is known as the sink condition. Thus, only seed values x with $x \geq M$ are admissible, and ideal seed values x are uniform random with $x \geq M$. Note that $C = -M$ is the QT mask value.

4.5 Parallel Processing

Partition RNG with period $P \approx 2^G$ into 2^S independent streams of length 2^{G-S} for parallel processing or shotlines. Partition each stream into 2^B independent substreams of length 2^{G-S-B} , the effective “period” of any scalar RNG for independent variables.

T258 has $G = 258$, and MUVES uses $S = B = 86$, so $G - S = 172$ and $G - S - B = 86$. The global RNG is seeded once to set base state. Then 2^{86} streams are separated by long jumps of 2^{172} from the base. In each stream, 2^{86} substreams are separated by short jumps of 2^{86} , the effective substream “period.”

4.6 Run Time

Suppose numbers are generated at a rate of 1 billion, or about 2^{30} , per second.

The legacy LCG with a single stream of length 2^{32} has 8 substreams of length 2^{29} . Each will run for 0.5 seconds, and the full cycle runs for 4 seconds.

One year $\approx 60^2 \cdot 24 \cdot 365 \approx 2^{24.91} \approx 2^{25}$ seconds.

For T258: a substream of length 2^{86} will run for $\sim 2^{86-30-25} = 2^{31} \approx 2$ billion years, a stream of length 2^{172} will run for $\sim 2^{172-30-25} = 2^{117} \approx 10^{26}$ billion years, and a full cycle of length 2^{258} will run for $\sim 2^{258-30-25} = 2^{203} \approx 10^{52}$ billion years.

Brute force verification is impractical. The numbers are large:

$$\begin{aligned} 2^{258} &= 463168356949264781694283940034751631413079938662562256157830336031652518559744 \approx 5 \times 10^{77} \\ P &= 463168356949050750352076184268918090343706927944462529355293134289296410279935 \approx 5 \times 10^{77} \\ 2^{258} - P &= 214031342207755765833541069373010718099726802537201742356108279809 \approx 2 \times 10^{65} \\ 2^{172} &= 5986310706507378352962293074805895248510699696029696 \approx 6 \times 10^{51} \\ 2^{86} &= 77371252455336267181195264 \approx 8 \times 10^{25} \\ P < 2^{258} &\text{ so the number of full streams is not } 2^{86} \text{ but } \text{floor}(P/2^{172}) = 77371252455300513717551104 \approx 8 \times 10^{25} \\ &\text{number of streams "lost"} = 35753463644160 \approx 4 \times 10^{13} \end{aligned}$$

A useful approach to verification follows from SCR 2138, Section 7.

5. SCR 1049–1050

This following is equivalent to the initial 2008 implementation of T258 in MUVES as presented in SCR 1049 “Replace existing Uniform random number generator” and SCR 1050 “Provide independent RNG streams for DMUVES”.

The constructor saves the seed and initializes the system.

```
rng::rng(uz seed) {
    this->seed = seed;
    init();
}
```

The initializer seeds the LFSRs and invokes a single transition.

```
rng& rng::init() {
    seedx();
    t();
    return *this;
}
```

The original seeding routine used a single seed integer and an auxiliary LCG of the form $x_{i+1} = c \cdot x_i$ to obtain 5 uniformly-distributed seed values.

```
void rng::seedx() {
    uz c = 69069;
    s[0] = c*seed; if(s[0] < 2) s[0] += 2u;
    s[1] = c*s[0]; if(s[1] < 512) s[1] += 512u;
    s[2] = c*s[1]; if(s[2] < 4096) s[2] += 4096u;
    s[3] = c*s[2]; if(s[3] < 131072) s[3] += 131072u;
    s[4] = c*s[3]; if(s[4] < 8388608) s[4] += 8388608u;
}
```

The system uses an array of 65 polynomials “const uz rng::JP[65][5]” for jumps of 2^p where $p = (86, 172, 173, 174, \dots, 233, 234, 235 = 172 + 63)$. MUVES uses 4 independent substreams (for variables, separated by 2^{86}) in each thread stream (separated by 2^{172}).

Generators for stream k with $0 < k < 2^{64}$ are obtained by applying k large jumps of size 2^{172} using the binary decomposition of k followed by small 2^{86} jumps for substreams. Note that if the n^{th} bit b_n of $k = \sum_{n=0}^{63} b_n 2^n$ is nonzero, then JP[n+1] can be used to advance each LFSR by $2^{171+n} = 2^n \cdot 2^{172}$ for total offset of $k \cdot 2^{172}$. This is efficient even for large k , unlike iterating the 2^{172} jump k times. Then each substream i is advanced by $i \cdot 2^{86}$ to obtain offsets of $k \cdot 2^{172} + i \cdot 2^{86}$ for $i = 0, 1, 2, 3$.

6. SCR 1908

In 2014, SCR 1908 “Changes to Tausworthe T258 Random Number Generator” realized these modifications:

Description: The seeding routine in T258 is more restrictive than necessary and it makes a call to the generator itself before returning. This prevents setting the five states to the same seed and making a call to the generator introduces confusion when comparing to the standalone behind-armor debris model.

and the document `scr.pdf` referenced in the SCR contains:

- Seeding routine is more restrictive than necessary, in two respects:
 - Seed is multiplied by 69069 each time an internal state is set.
 - Makes a call to the RNG itself before the seeding procedure returns.

The current seeding routine (see Figure 1 [omitted]) multiplies the seed by 69069

each time that the five states of the RNG are set. There is no need to do this, and it prevents us from setting the five states to the same seed.

The initialization method implemented by SCR 1908 is equivalent to:

```
rng& rng::init() {
    seedx();
    return *this;
}
```

The seeding method implemented by SCR 1908 is equivalent to:

```
void rng::seedx() {
    s[0] = seed; if(s[0] < 2) s[0] += 2u;
    s[1] = seed; if(s[1] < 512) s[1] += 512u;
    s[2] = seed; if(s[2] < 4096) s[2] += 4096u;
    s[3] = seed; if(s[3] < 131072) s[3] += 131072u;
    s[4] = seed; if(s[4] < 8388608) s[4] += 8388608u;
}
```

6.1 Consequences

6.1.1 Jump Computation Failure

Omission of the valid initial state criterion of Section 4.3 (by removing the initialization call to the RNG) introduces the problems presented in that section. This includes the failure of jump computation even for correct jump polynomials.

The requirement for statistically independent random substreams across all streams is essential for the validity of the simulation. This is obtained by partitioning the overall cycle into *non-overlapping* segments, guaranteed by correct jump computations. When the jump computations fail, the resulting possible overlap makes the claim of independence invalid.

6.1.2 Stream independence Failure

Use of the same single seed for all 5 LFSRs (instead of uniform random seeds derived from a single value) introduces another problem.

Put simply, if you run a simulation with seed x one day, and seed $x + 1$ the next day, the results will not be independent (as is required for a random sample). Sequentially seeded cycles are not independent. If the seeds themselves are generated by some random process (clock time, process id, /dev/random, /dev/urandom, radiation, etc) this is likely not an issue. But if a human needs 3 seeds for a random

sample of runs, she/he just might pick something like 666, 667, and 668. Then the following problem manifests.

A number is Borel normal in base r if every sequence of k symbols in the letters $0, 1, \dots, r-1$ occurs in the base- r expansion of the given number with the expected frequency r^{-k} . Uniform random numbers are Borel normal.

Let x be a 64-bit ($r = 2$) uniform random integer and $b(x) =$ the number of 1s in x . Then $b(x) \sim B(64, 1/2)$, where $B(n, p)$ is the binomial distribution and $q = 1 - p$. Here, $n = 64$ and $p = q = 1/2$. Then $E b(x) = np = 32$ and $\text{Var } b(x) = npq = 16$.

If $(x_i)_{i=1}^{\infty}$, is a sequence of such x , then the $b(x_i)$ are iid $B(n, p)$, asymptotically normal

$$b(x_i) \sim N(np, npq) \quad (28)$$

If $(x_{ji})_{i=1}^{\infty}$ for $j = 1, \dots, k$ are k such sequences, then the $S_i = \{x_{ji} : j = 1, \dots, k\}$ are iid random samples of size k from $B(n, p)$. Then the sample means $m_i = \text{mean}(S_i) = \frac{1}{k} \sum_{j=1}^k b(x_{ji})$ are iid asymptotically normal

$$m_i \sim N\left(np, \frac{npq}{k}\right). \quad (29)$$

and the sample variances $v_i = \text{var}(S_i) = \frac{1}{k} \sum_{j=1}^k (b(x_{ji}) - m_i)^2$ are iid asymptotically normal

$$v_i \sim N\left(npq, (npq)^2 \left(\frac{2}{k-1} + \frac{\kappa}{k}\right)\right) \quad (30)$$

where κ is the excess kurtosis

$$\kappa = \frac{1 - 6pq}{npq}. \quad (31)$$

Graphs follow for $i = 1, \dots, 500$ and $k = 1000$, where k sequential seeds were chosen. First we see the series results for seeds 0 through 999, offset 0, to the same scale in Fig. 1 and with the correct Q method magnified to show detail in Fig. 2. Then in Figs. 3 and 4 we see the same presentation for an arbitrary offset s_o , for seeds s_o through $s_o + 999$. Then, without regard to the series, we see the cumulative distribution functions (CDFs) for offset 0 in Fig. 5 and offset s_o in Fig. 6.

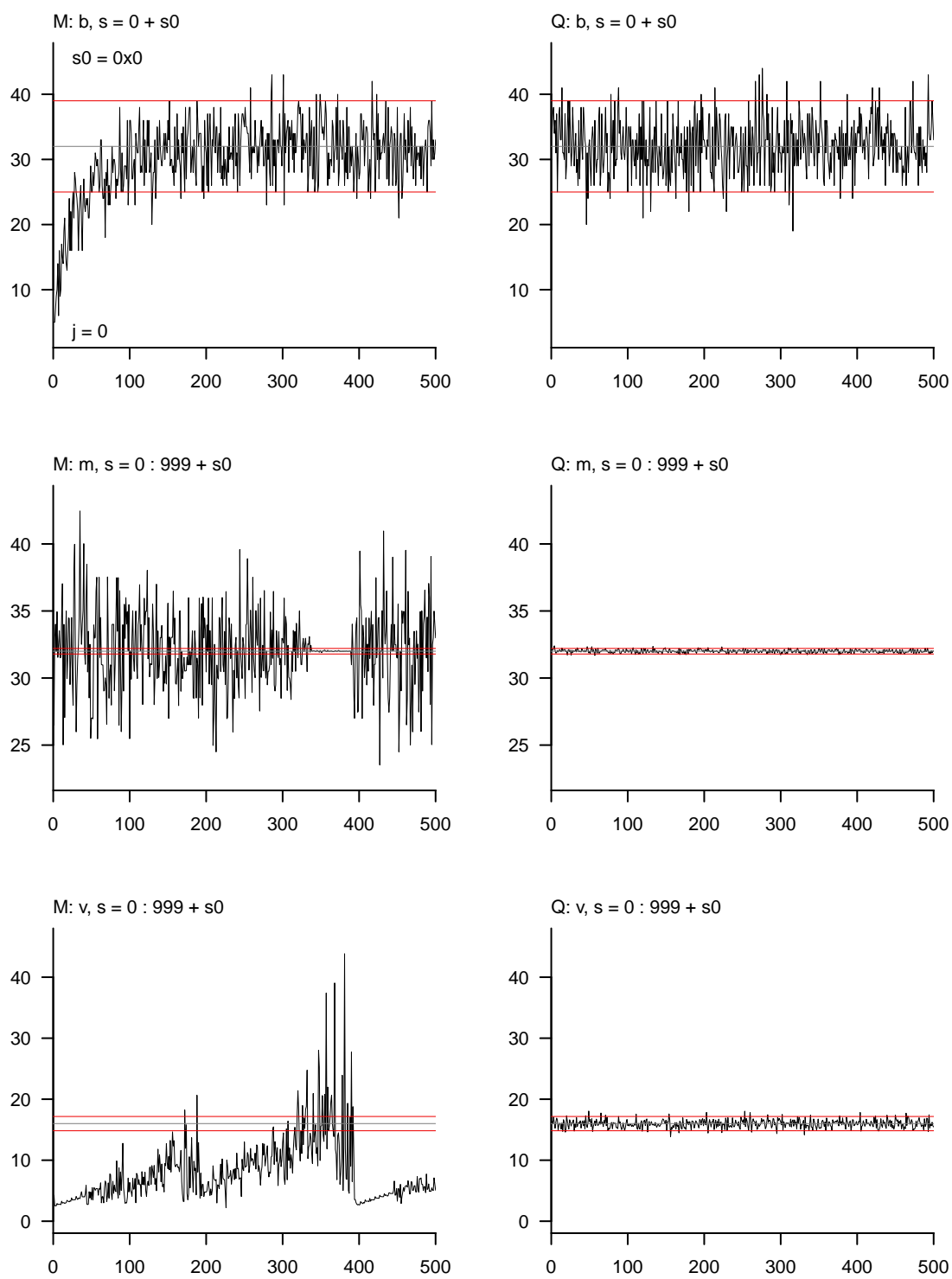


Fig. 1 b series, no offset, same scale

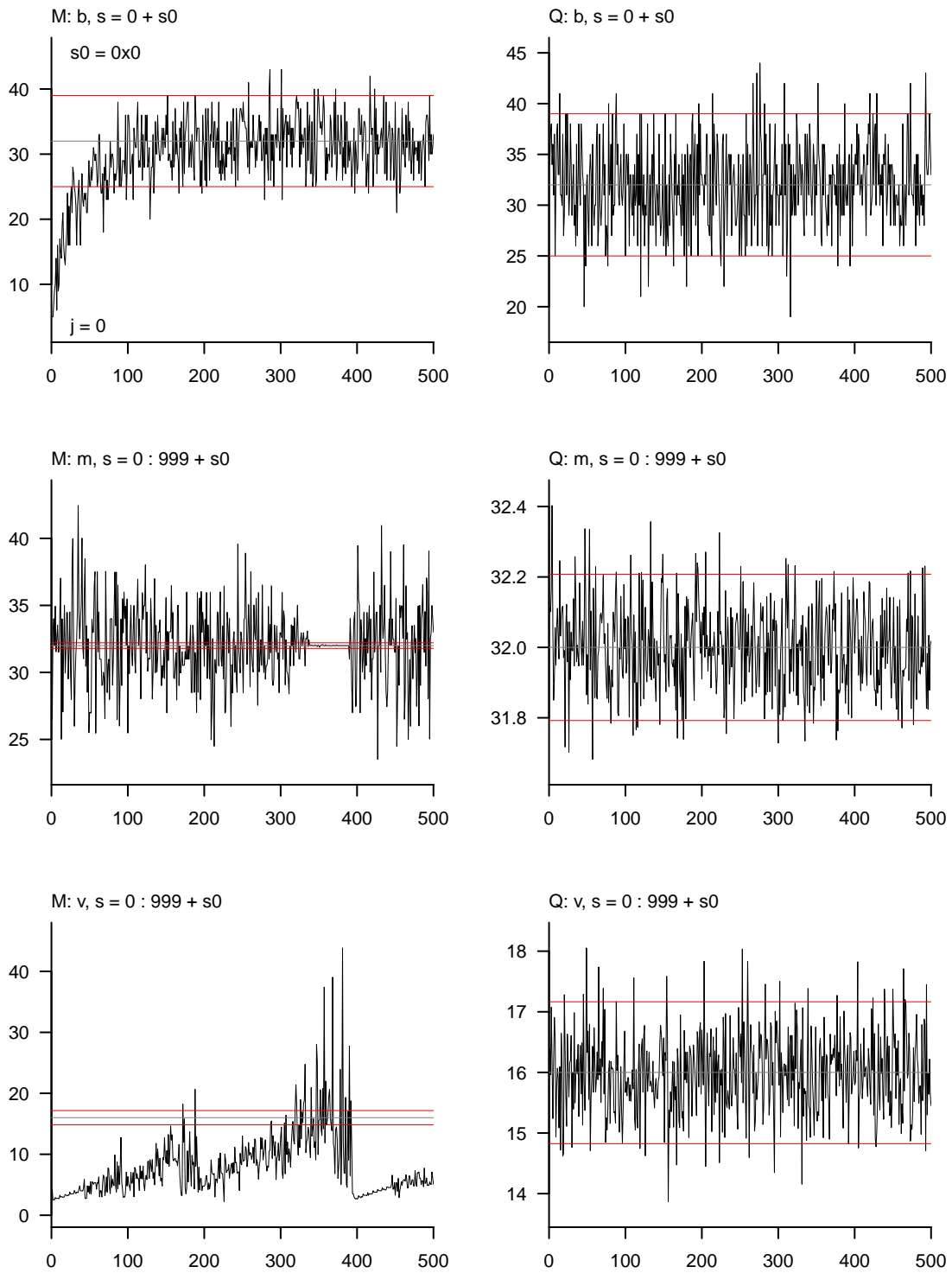


Fig. 2 *b* series, no offset, detail

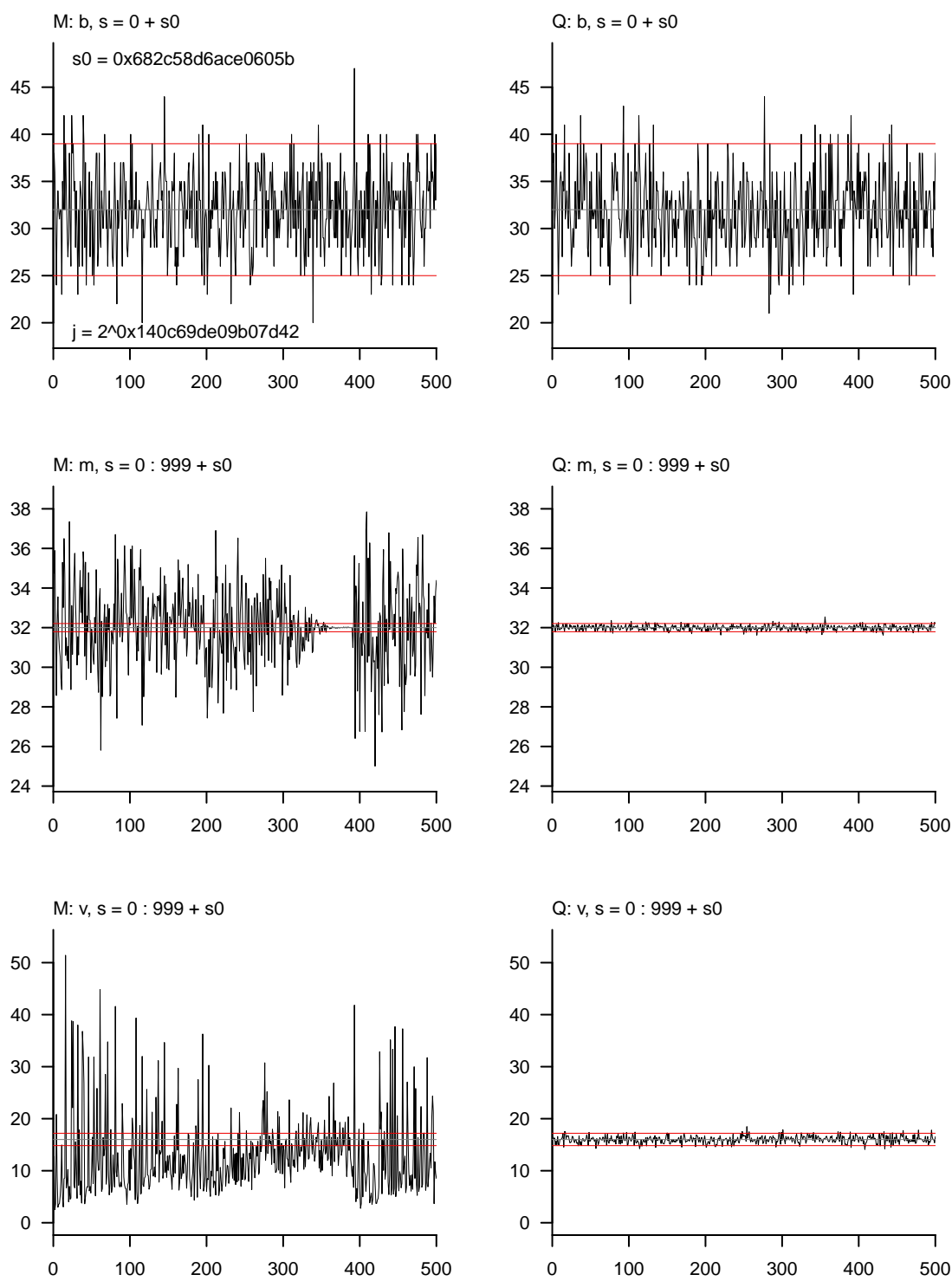


Fig. 3 *b* series, arbitrary offset, same scale

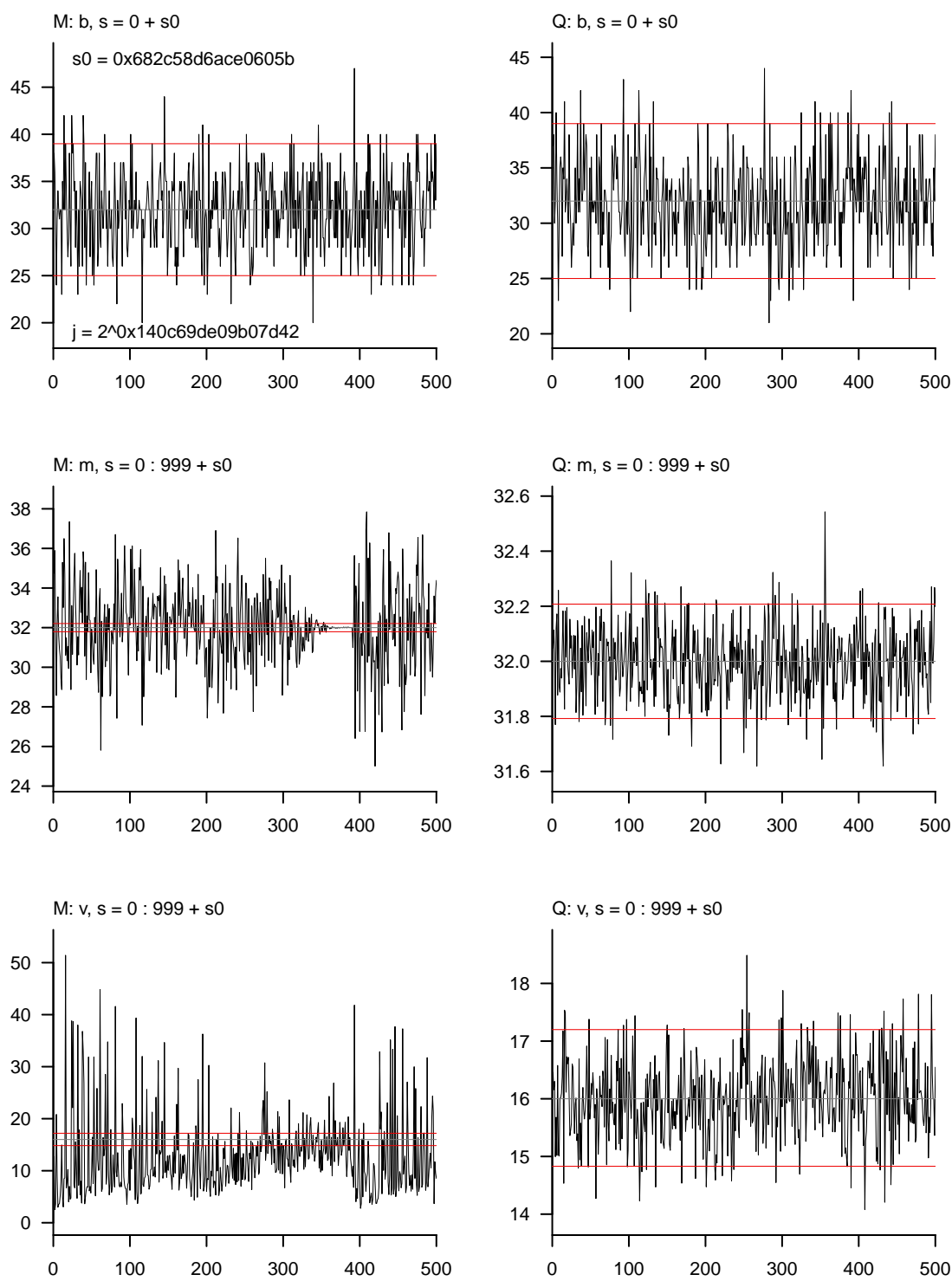


Fig. 4 *b* series, arbitrary offset, detail

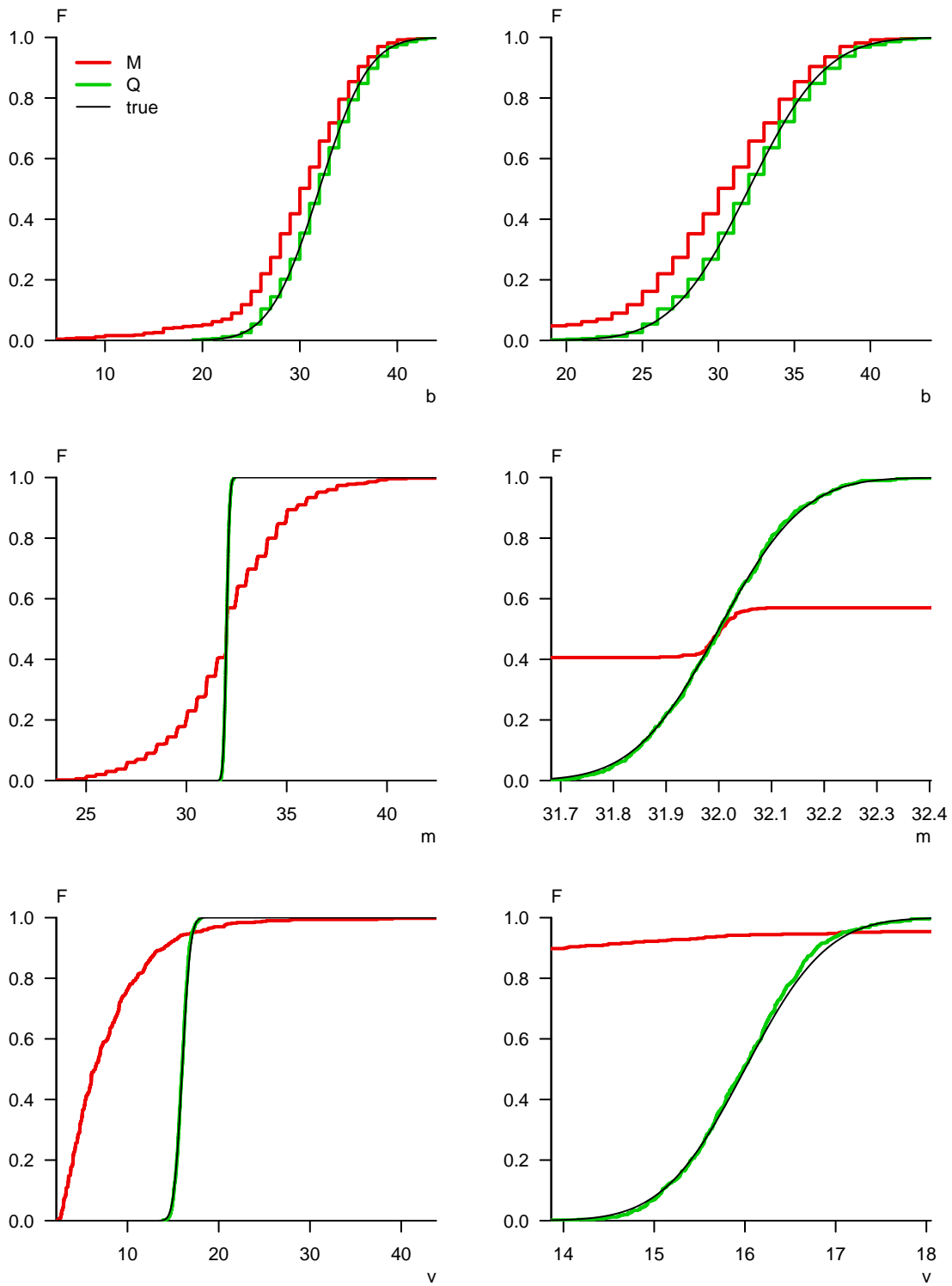


Fig. 5 b, m, v CDFs, no offset

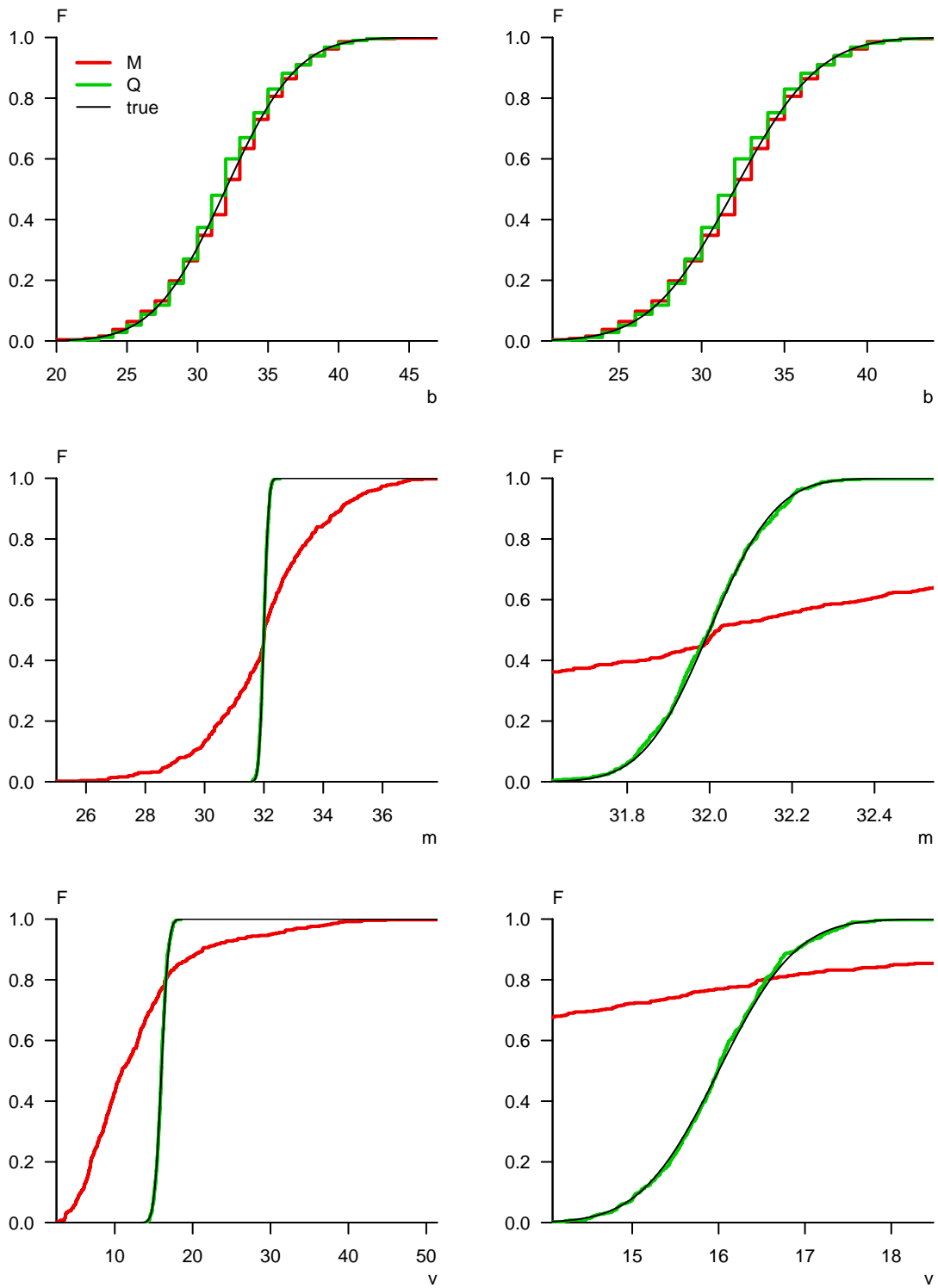


Fig. 6 b, m, v CDFs, arbitrary offset

7. SCR 2138

In 2015, SCR 2138 “MUVES Random Number Generator initialization correction” remediated the effects of SCR 1908. The initialization generator call was reinstated to assure correct jump computations and statistical independence of everything.

```
rng& rng::init() {
    seedx();
    t();
    return *this;
}
```

An improved seeding scheme uses a better LCG that SCR 1049, available in L’Ecuyer.⁵ This reestablishes the independence of sequentially-seeded streams lost in SCR 1908.

```
void rng::seedx() { // set seed
    uz c = 0x27bb2ee687b0b0fd, d = 0x891087b8e3b70cb1;
    do s[0] = c*seed+d; while(s[0] < 0x000002);
    do s[1] = c*s[0]+d; while(s[1] < 0x000200);
    do s[2] = c*s[1]+d; while(s[2] < 0x001000);
    do s[3] = c*s[2]+d; while(s[3] < 0x020000);
    do s[4] = c*s[3]+d; while(s[4] < 0x800000);
}
```

The final element of SCR 2138 implements an improved algorithm for jump computation based on the identity of Section 7.1. Previous computations are a “special case” of the new algorithm, in that the results are identical. The old algorithm could compute jumps of size 2^{86} and $k2^{172}$ for $k = 0, \dots, 2^{64} - 1$. The new algorithm is more transparent but capable of jumps $k2^p$ for both p and k in $\{0, \dots, 2^{64} - 1\}$.

In either case, suppose that $t()$ is a single-step state transition and that $j(p)$ advances the RNG state by 2^p steps. With the old algorithm, one would check that either 2^{86} applications of $t()$ or one application of $j(86)$ to some initial state results in the same final state. At a rate of 1 billion per second this would take 2 billion years. For $j(172)$, this would take 10^{35} years. A major benefit of the new formulation is that the computations can be verified more efficiently due to the availability of arbitrary $j(p)$. One need only check that $t()$ and $j(0)$ give the same state and for $p = 1, 2, 3, \dots, p_{\max}$ that 2 applications of $j(p - 1)$ give the same state as 1 application of $j(p)$, since $2 \cdot 2^{p-1} = 2^p$. For $p_{\max} = 236$ it takes about 5 milliseconds to check that any such $j(p)$ is correct.

7.1 Identity

The LFSRs in T258 generated by polynomials of the form $P(z) = z^k + z^q + 1$ in blocks of size s have periods $m = 2^k - 1$. By the division algorithm a jump size of 2^p can be written as $2^p = Qm + 2^p \% m$ for integer Q , thus 2^p and $2^p \% m$ are the same jump.

In fact, $2^p \% m = 2^{p \% k}$, so only the jumps 2^p for $p = 0, 1, 2, 3, \dots, k - 1$ for each LFSR are required to compute a T258 jump of 2^p for any $p \geq 0$.

Formally, we assert that

$$2^p \% (2^k - 1) = 2^{p \% k} \text{ for } k \geq 2 \text{ and } p \geq 0. \quad (32)$$

Proof:

If $p < k$ then $2^p < m$ and $2^p \% m = 2^p = 2^{p \% k}$.

If $p = k$ then $2^p = m + 1$ and $2^p \% m = 1 = 2^{p \% k}$.

Suppose the claim holds for some $p \geq k$ and all $p' < p$, and consider $p + 1$:

$$\begin{aligned} 2^{p+1} \% m &= (2 \cdot 2^p) \% m = ((2 \% m) \cdot (2^p \% m)) \% m \\ &= (2 \cdot 2^{p \% k}) \% m, && \text{by induction, as it holds for } p \\ &= (2^{p \% k + 1}) \% m = 2^{(p \% k + 1) \% k}, && \text{by induction, as } p \% k + 1 \leq k \leq p \\ &= 2^{(p \% k + 1) \% k} = 2^{(p+1) \% k}. && \text{QED} \end{aligned}$$

If $k = 1$, then $m = 2^k - 1 = 1$ and $2^p \% 1 = 0 \neq 2^{p \% 1} = 2^0 = 1$. So $k \geq 2$ is required. Note that various steps in the proof also rely on $k > 1$.

7.2 Implementation

T258 characteristic polynomial degrees are

```
const int rng::CPd[5] = { 63, 55, 52, 47, 41 };
```

Only the polynomials for 2^0 through 2^{62} , 2^{54} , 2^{51} , 2^{46} , and 2^{40} , are required for the 5 LFSRs, respectively. Then any jump 2^p for $p = 0, \dots, 2^{64} - 1$ can be computed by

```
rng& rng::jump(uz p) { // jump 2^p
    uz i, j, a[5] = { 0 };
    for(i=1; i; i<=1, t())
        for(j=0; j<5; j++)
            if(JP[p % CPd[j]][j] & i) a[j] ^= s[j];
    memcpy(s, a, 5*sizeof(uz));
    return *this;
```

```
}
```

Any jump of size $k2^p$ can be computed by

```
rng& rng::jumpk(uz k, uz p) { // jump k*2^p
    for(; k; k>>=1, p++) if(k & 1) jump(p);
    return *this;
}
```

which uses the binary representations $k = \sum_{i=0}^m b_i 2^i = b_m 2^m + \dots + b_0 2^0$ and $k2^p = \sum_{i=0}^m b_i 2^{i+p} = b_m 2^{m+p} + \dots + b_0 2^p$.

Increments for small (variable) and large (stream) jumps are

```
static const int JS = 86; // log_2 ( small jump )
static const int JL = 172; // log_2 ( large jump )
```

Stream k can be set up by

```
rng *z[4];
for(int i=0; i<4; i++) {
    z[i] = new rng(seed);
    z[i]->jump(k, rng::JL).jump(i, rng::JS);
}
```

7.3 Jump Verification

These operators compare T258 state arrays.

```
bool rng::operator!=(const rng &w) { // state inequality
    uz e = 0;
    for(int i=0; i<5; i++) e |= s[i] ^ w.s[i];
    return e;
}

bool rng::operator==(const rng &w) { // state equality
    return ! ( *this != w );
}
```

Then success of the following code verifies jump computations for the given seed for all jumps 2^p with $p = 0, 1, 2, 3, \dots, p_{\max}$.

```
void Vjump(uz seed, uz pmax) { // inductive jump verification

    rng z0(seed), z1(seed); // T258 objects, same state

    cout << "Vjump(" << Px(seed) << ", " << dec << pmax << "): ";
```

```

    if( z0.jump(0) != z1.t() ) { // base
        cerr << "fail base check" << endl;
        exit(1);
    }

    for(uz p=1; p<=pmax; p++ ) // induction
        if( z0.init().jump(p) != z1.init().jump(p-1).jump(p-1) ) {
            cerr << "fail induction check, p=" << dec << p << endl;
            exit(1);
        }

    cout << "for all p = 0:" << pmax << " jump(p) == t^(2^p)" << endl;
}

```

This checks that $\text{jump}(0)$ is a single transition $t()$ and inductively that $\text{jump}(p)$ is equivalent to 2 applications of $\text{jump}(p-1)$, as $2^p = 2 \cdot 2^{p-1}$. Upon success, this establishes that $\text{jump}(p)$ implements a jump of 2^p for $p = 0, 1, 2, 3, \dots, p_{\max}$.

8. SCR 2142

Released in 2015, SCR 2142 “Modernize Rn package to C++ and STL use” reimplements the RNG system in C++ and provides code cleanup.

The redundant old jump code (available as an option in SCR 2138) was removed, leaving only the new SCR 2138 code. From the SCR text:

For SCR2138, new jumping code was added, but the old jumping code was left in place because it was not understood that the new jumping code is able to provide the same functionality. The old jump code will be removed as part of this SCR as the Rn code is cleaned up.

SCR 2142 also suggests exploiting the Standard Template Library (STL):

Note that the C++ STL contains classes for random number generation and it is possible to adapt the T258 RNG (as a random number engine) for this use. The C++ STL contains a complete set of random distributions and it is highly desirable to use these instead of reimplementing all the distributions in MUVES when standard versions already exist. Therefore, use of the STL should be investigated as part of this SCR (both SCR2061 and SCR2105 will require C++11 and the C++11 STL).

The new C++11 standard library random number generation feature is exposed

by including the `<random>` header. At this time, the MUVES development and distributions platforms do not provide C++11 compilers. So incorporation of the RNG system into the C++11 `<random>` library framework has been deferred.

9. Demonstration and Verification

The code `t258.h` in Section 9.1 is a complete C++ class implementation of T258 equivalent to the current SCR 2138 MUVES version with seeding, initialization, state transition `t()`, integer and real $U(0,1)$ generators, and functions `jump(p)` and `jumpk(k,p)` for jumps of 2^p and $k2^p$, respectively, where p and k are 64-bit unsigned integers in the range $0, \dots, 2^{64} - 1$.

The driver code in Section 9.2 implements the class in a working program. The driver uses a single seed, `0xd8940e83ec602c7b = 15606114568514841723`.

The function `Vtab` generates tables of integer and real output to demonstrate the stream and substream jumps used in MUVES. The resulting tables on pages 32 and 33 were provided to MUVES developers as a sanity check and were incorporated into MUVES unit testing.

The function `Vjump` verifies jump computation for $p = 0, \dots, 236$. The largest jump possible in the current MUVES implementation has $k = 2^{64} - 1$ and $p = 172$, so $k2^p < 2^{64} \cdot 2^{172} = 2^{236}$. On success, one sees

```
Vjump(0xd8940e83ec602c7b, 236): for all p = 0:236 jump(p) == t^(2^p)
```

On failure, one sees either

```
Vjump(0xd8940e83ec602c7b, 236): fail base check
```

if `jump(0)` is not the same as `t()`, or something like

```
Vjump(0xd8940e83ec602c7b, 236): fail induction check, p=7
```

if `jump(p) . jump(p)` is not the same as `jump(p+1)` for some p . The latter occurs if the transition call is removed from the initialization, as in SCR 1908.

Of course, success for a single seed is not proof. But similar code had no failure with billions of different random seeds, which is not proof either but provides reassurance that things are working.

Jump demonstration table, integer:

Vtab(0xd8940e83ec602c7b, uz): jump = J*2¹⁷² + K*2⁸⁶

n	J = 0 K = 0	K = 1	K = 2	K = 3
1	0xc16e6cacb10aafb7	0x8698d7ee73267dfd	0x3b24e9553fe2ab49	0xb501e17f9a22cb5b
2	0x4197d305ac8d14d7	0xa6ba3076faf9803d	0x377b2a9c373cd544	0x51a15d9dfc1a0fb9
3	0xad851fdfedb01c77	0xffa700c1f6894ab6	0xc8af1eff5a013e0	0x641f61b7c5206628
4	0xb1c4c1d91fd598d0	0x4b603c781fcc564a	0x4d016dac0e18f0cf	0xc7b5e5cfb78eca78
5	0xe15f23003cf4a5bd	0x25ed9bbe1d862f17	0xfb244937ce181286	0xc354f0d48b67a904

n	J = 1 K = 0	K = 1	K = 2	K = 3
1	0x4c6be74635dd95f9	0x8da677e1d0c102fc	0x5f94da92d55163fc	0xba2853e0ae2053ed
2	0x37a4ffc68991c454	0x947f071f49deec2a	0x273c05a49c944ba5	0x1c21f190185d6326
3	0xb9f3d6e556357739	0xee429bf79386d48a	0x435a62d0a13dd575	0x55622bd6b0745921
4	0x5923006254cd0202	0x3a0987eaf69515c0	0xd95a9d3985026c3a	0x8a0a27de969e2d40
5	0xb14071ad4ff5297	0xe93c1daf844af7cf	0x9032a81041006403	0x5213f9f085b2d5e8

n	J = 2 K = 0	K = 1	K = 2	K = 3
1	0x71590ce85a42d74d	0x97699306050bd514	0x8d571a13e87d55a9	0x4f21c38777b5f1bb
2	0xde5ff97d8853eea7	0xc221a247176e77d4	0xda7586557e84eaa1	0x64330c2ebf01fe85
3	0x280615a8b54284f1	0x5d33a5b5585a7364	0xa25f2c70d6c3f7e4e	0x861a46a5ff29b9e0
4	0xa5f63cacd6e41d91	0x2c0bc7999bb29ff3	0x617c53df2a932bc8	0x66992fc9c1a53027
5	0xa3c79c2c534b27b4	0x2221ba242669cfc	0xfdd232165791d1d	0x3888d26261dce6c2

n	J = 3 K = 0	K = 1	K = 2	K = 3
1	0xff71ca0b732c4949	0x867d129cd858c058	0xa6ec51930d817e66	0x9dabc7e737ed494f
2	0xbc77a72aaf89753e	0x9d61d14749155805	0x588ae0b6b281981f	0xc7bc1546bc2309c3
3	0x2de469f3f7ddede3	0x1f4e610a2f7263e0	0xd71668e4417bdfc	0x766dbeb7f102b3a7
4	0xff672a0c36125e2c	0xb10593560a59ecd2	0x152646fe75348e59	0xf63d8f4fe717a0b9
5	0x9159bb906bebc03	0xa7a301f2ed7fec8d	0xf4f791cbf9bc6989	0xdfbb782b73b6ae5d

n	J = 4 K = 0	K = 1	K = 2	K = 3
1	0xd9f9bdddff303202	0x0173a3a4f2254c51	0xbfd8aaffd19a665e	0x58a927c37a3602d3
2	0x86d7897db155a745	0x5bdf4b8032d914f9	0x4e09bc87a5c7922c	0x83c947bdc7083b94
3	0xe819eb79b2aa8d0a	0xa8a74e7e6d0c1038	0x505701b38a6bfb90	0x62c86c9c75dc91c
4	0x14c91fcd7c608bfc	0xdb8f97503a9d1574	0x562a180930c36f54	0xb102e6e501b83485
5	0x52785e23c9d880ed	0xbbfc72e609c96ba3	0x4e5031d2d4a0f437	0x9e43932dfac8e9dc

n	J = 5 K = 0	K = 1	K = 2	K = 3
1	0x0c14a3a5d302c989	0x660abd0396ef14a3	0xaf60b1affb3e53a6	0x7ece213ba1ec8811
2	0x8efd1ff4d7e1941b	0xb743ff3460f62e9f	0xa1507842e9da2f5d	0x628c35d2c5c7724f
3	0x623b009e10cc32fe	0x93d3202a8536b686	0xccecf9f154120adb	0x59969f88b7901893
4	0xe3030e94754a8c27	0x2893a81ecb8c6aa7	0x249eb96507002c4f	0xb28a90711088ba68
5	0xe895e1af53cae98e	0x4a19e122b28447be	0x9d92eaad875e312f	0x7227913afc65bb4a

n	J = 6 K = 0	K = 1	K = 2	K = 3
1	0xad7f6af0771f42d0	0xa673517c3e97a375	0xae617e06dcdcfcd2	0x16a04a9eee343931
2	0x7152e34f71b13bdf	0xbf549048b46c344b	0x132d592b77eefaa	0x7a177d8c7427d713
3	0xc91be73b58418123	0xad0064cc27aab021	0xfd91c38a7805f8d7	0x82f297163077e680
4	0x4e55e470f36f030c	0x500ea6baf0d3b074	0x6ea259bd74ef380c	0x5b1f1259957dc675
5	0x085942e71dc8af6b	0xfb5fa80c2bb01aaf	0x92679c69aaf343ff	0xb74e8d82a30ec277

n	J = 7 K = 0	K = 1	K = 2	K = 3
1	0xfbc1292fc680a416	0xe45087ee40f56993	0xef6a5b85ba23a884	0xc0e725b06d4e72fb
2	0x6d9b1af20461b9bc	0x2c4566e9fde33783	0x38e0e42505739ea7	0x90b024dc86178651
3	0x631011d91bf6b2e9	0x75987bb9443e5261	0x6fe6183cf784ebd0	0xed8b48c5ce0a240f
4	0xff48c5924e2590a4	0x6d37fcce4dad574c	0x916a0c91c447963a	0x9515183807fc0f59
5	0x6e62c17ab7b53e40	0x92f288c5f7ec56a6	0x4f119d1ea6a778ea	0xc8700e32f6366368

Jump demonstration table, real:

Vtab(0xd8940e83ec602c7b, u01): jump = $J \cdot 2^{172} + K \cdot 2^{86}$

n	J = 0 K = 0	K = 1	K = 2	K = 3
1	0.7555911943063999	0.5257697064545485	0.2310319741371908	0.7070599495438145
2	0.2562229050495726	0.6512785234908502	0.2167231208215867	0.3188684950547228
3	0.6778125688267099	0.9986420129339751	0.1114951334799346	0.3911038468987740
4	0.6944085269058945	0.2944371979972840	0.3008030457550352	0.7801192886468541
5	0.8803579211789434	0.1481568659924269	0.9810224305384454	0.7630148428828806

n	J = 1 K = 0	K = 1	K = 2	K = 3
1	0.2985214754497660	0.5533213545138587	0.3733650787810965	0.7271778510604474
2	0.2173614368162538	0.5800632907283859	0.1532596136936074	0.1098929383082257
3	0.7263769445327932	0.9307038764414088	0.2630979308462351	0.3335292243140925
4	0.3481903305117164	0.2267079304352431	0.8490389123371728	0.5392174642971602
5	0.7463993492610801	0.9110735467441307	0.5632729568178561	0.3206173145497258

n	J = 2 K = 0	K = 1	K = 2	K = 3
1	0.4427650515811010	0.5914546861359178	0.5521103190044319	0.3091089444858068
2	0.8686519557453064	0.7583257125438025	0.8533557852483856	0.3914039243818549
3	0.1563428437123182	0.3640693252312788	0.6342647338371971	0.5238384394898848
4	0.6482885286303744	0.1720547437686925	0.3808033389892215	0.4007749431199457
5	0.6397645576683257	0.1333271349198305	0.1869680363210801	0.2208377351943116

n	J = 3 K = 0	K = 1	K = 2	K = 3
1	0.9978300359681934	0.5253459580347650	0.6520434364333648	0.6159024180887417
2	0.7362007598126519	0.6147738265072652	0.3458691068685823	0.7802136705105448
3	0.1792665691972314	0.1222897195688772	0.8401856953873755	0.4626120757175254
4	0.9976679115116637	0.6914913258609442	0.0826153155550903	0.9618768282008134
5	0.5677754619209018	0.6548310487828527	0.9569026110636841	0.8739543062290190

n	J = 4 K = 0	K = 1	K = 2	K = 3
1	0.8514670054420896	0.0056707647037416	0.7493998407895588	0.3463311054458345
2	0.5267263347498869	0.3588759601578639	0.3048360663576928	0.5147900427173518
3	0.9066455051530885	0.6588028963715689	0.3138276160556613	0.3858707360408210
4	0.0811939121889528	0.8576597758170588	0.3365798017919254	0.6914505299662046
5	0.3221491658567590	0.7343208133332034	0.3059111728404138	0.6182186114554359

n	J = 5 K = 0	K = 1	K = 2	K = 3
1	0.0471899299473179	0.3986013540042530	0.6850691847459924	0.4953327913844295
2	0.5585498783838203	0.7158813002469847	0.6301341212618828	0.3849519385892373
3	0.3837128053572065	0.5774402717408644	0.8078600134068169	0.3499545773298113
4	0.8867653953978253	0.1585030627425553	0.1430469390534205	0.6974268222843730
5	0.9085370114369585	0.2894573888502388	0.6155230210928202	0.4459162491806613

n	J = 6 K = 0	K = 1	K = 2	K = 3
1	0.6777254902909791	0.6501971176463415	0.9175734446451357	0.0883833539976003
2	0.4426710194545124	0.7473840882779974	0.0749107104419934	0.4769209354309674
3	0.7855820197949068	0.6757872579980055	0.9905054295595831	0.5115141324208419
4	0.3059981132277549	0.3127235609434382	0.4321647727528529	0.3559428662024322
5	0.0326120199440846	0.9819283513749614	0.5718934782007183	0.7160423702180720

n	J = 7 K = 0	K = 1	K = 2	K = 3
1	0.9834161512030755	0.8918538052248174	0.9352166367990679	0.7535270267229011
2	0.4281479683744526	0.1729339905995289	0.2221815672287007	0.5651877439869093
3	0.3869639544536436	0.4593579604445435	0.4371047161908056	0.9279065592691400
4	0.9972041589918688	0.4266355518026803	0.5680244308353208	0.5823531281275650
5	0.4311943935969865	0.5740132792779360	0.3088625144797826	0.7829598307054879

9.1 T258 Class Code

```
typedef uint64_t uz;          // 64-bit unsigned integer

class rng {                  // T258
public:
    rng(uz seed);            // constructor
    rng& init();             // (re)initialize
    rng& t();                // state transition
    rng& jump(uz p);         // jump 2^p
    rng& jumpk(uz k, uz p);  // jump k*2^p
    uz gen();                // integer generator
    double u01();            // u(0,1) generator
    static const int JS = 86; // log_2 ( small jump )
    static const int JL = 172; // log_2 ( large jump )
    bool operator!=(const rng& w); // state comparison
    bool operator==(const rng& w); // state comparison
private:
    uz s[5];                 // LFSR state array
    uz seed;                 // this.seed
    void seedx(void);        // set seed
    static const int CPd[5]; // CP degrees
    static const uz JP[63][5]; // jump polynomials
};

rng::rng(uz seed) {         // constructor
    this->seed = seed;
    init();
}

rng& rng::init() {         // initialize
    seedx();
    t();
    return *this;
}

void rng::seedx() {        // set LFSR seeds
    uz c = 0x27bb2ee687b0b0fd, d = 0x891087b8e3b70cb1;
    do s[0] = c*seed+d; while(s[0] < 0x000002);
    do s[1] = c*s[0]+d; while(s[1] < 0x000200);
    do s[2] = c*s[1]+d; while(s[2] < 0x001000);
    do s[3] = c*s[2]+d; while(s[3] < 0x020000);
    do s[4] = c*s[3]+d; while(s[4] < 0x800000);
}

rng& rng::t() { // T258:      C      s      q      k-s // k
    s[0] = ((s[0] & -0x000002) << 10) ^ (((s[0] << 1) ^ s[0]) >> 53); // 63
    s[1] = ((s[1] & -0x000200) << 5) ^ (((s[1] << 24) ^ s[1]) >> 50); // 55
    s[2] = ((s[2] & -0x001000) << 29) ^ (((s[2] << 3) ^ s[2]) >> 23); // 52
    s[3] = ((s[3] & -0x020000) << 23) ^ (((s[3] << 5) ^ s[3]) >> 24); // 47
    s[4] = ((s[4] & -0x800000) << 8) ^ (((s[4] << 3) ^ s[4]) >> 33); // 41
    return *this;
}

rng& rng::jump(uz p) {     // jump 2^p
    uz i, j, a[5] = { 0 };
    for(i=1; i; i<=1, t())
        for(j=0; j<5; j++)
            if(JP[p % CPd[j]][j] & i) a[j] ^= s[j];
    memcpy(s, a, 5*sizeof(uz));
    return *this;
}

rng& rng::jumpk(uz k, uz p) { // jump k*2^p
    for(; k; k>=1, p++) if(k & 1) jump(p);
    return *this;
}

uz rng::gen() {           // 64-bit integer
    t();
    return s[0] ^ s[1] ^ s[2] ^ s[3] ^ s[4];
}

double rng::u01() {      // U(0,1)
    uz z;
    const static double d = 1.0/(1UL<<53);
    do z = gen() >> 11; while(!z);
    return d * z;
}
```

Approved for public release; distribution is unlimited.

```

bool rng::operator!=(const rng &w) { // state inequality
    uz e = 0;
    for(int i=0; i<5; i++) e |= s[i] ^ w.s[i];
    return e;
}

bool rng::operator==(const rng &w) { // state equality
    return ! ( *this != w );
}

const int rng::CPd[5] = { 63, 55, 52, 47, 41 }; // CP degrees

const uz rng::JP[63][5] = { // JP[p][i] = 2^p jump polynomial for LFSR[i]
    {0x0000000000000002, 0x0000000000000002, 0x0000000000000002, 0x0000000000000002, 0x0000000000000002},
    {0x0000000000000004, 0x0000000000000004, 0x0000000000000004, 0x0000000000000004, 0x0000000000000004},
    {0x0000000000000010, 0x0000000000000010, 0x0000000000000010, 0x0000000000000010, 0x0000000000000010},
    {0x0000000000000100, 0x0000000000000100, 0x0000000000000100, 0x0000000000000100, 0x0000000000000100},
    {0x0000000000010000, 0x0000000000010000, 0x0000000000010000, 0x0000000000010000, 0x0000000000010000},
    {0x0000000010000000, 0x0000000010000000, 0x0000000010000000, 0x0000000010000000, 0x0000000010000000},
    {0x00000008000004006, 0x20000200100200, 0x080805414d000, 0x02009262d390, 0x00004800000},
    {0x0020000000018014, 0x20050881500001, 0x5209f6b064797, 0x54af12d5c26c, 0x00000004920},
    {0x00001801c00c0110, 0x0280d09c100019, 0x0c54a1e621499, 0x3d96aeb24aa5, 0x00010410400},
    {0x70000c007816105, 0x1c8104908c00c1, 0xb05ba8446e76e, 0x3081fec2623c, 0x00100148048},
    {0x7e3fe03e0000e011, 0x002631e50851cf, 0xbba8fed05eb47, 0x702b58a6b6e1, 0x11044801040},
    {0x7fd81f40fbf06105, 0x0d8e231d149309, 0x5c23bbbc5da56, 0x24083a6fe8b7, 0x08481485932},
    {0x0030000000008012, 0x61b24de97c0da3, 0xed52c8defa3df, 0x5065b92cd5bc, 0x160586101cc},
    {0x00001e0e00f0104, 0x00aa994928aec9, 0xf065085eac50, 0x77e8c17a05f8, 0x0db004cf018},
    {0x7e00004007f92013, 0x4ca1b5574dd946, 0x288336225032a, 0xa0b2fb86260, 0x0a0b2fb86260},
    {0x7fe7ffefc006105, 0x23990533099a2d, 0x6e57aa7334925, 0x60057f19fef8, 0x097edfb17f2},
    {0x0007f8ffc3fd6016, 0x275a2982e86c13, 0x1263bf9acdabb6, 0x2a7d2426007b, 0x061deee38fe},
    {0x702fffbefc7e2115, 0x37e54836f269e1, 0x4bfe07b195bb6, 0x017bbf224b71, 0x00000adae2e},
    {0x01f00681200dc116, 0x57eb3124fac097, 0x167a720daf502, 0x7311b420e038, 0x0503e540566},
    {0x0e27feee387c112, 0x04b5b746efc071, 0xc707bfd0e15a6, 0x516e04fd38cc, 0x15c8006ee5c},
    {0x7fd7f8ffc40b6103, 0x08a943bd59e7bf, 0xb0e2f0e8a5511, 0x50f342c7f5dd, 0x0d9bd741142},
    {0x0fc7e63f238c2004, 0x19067d8a48d883, 0x2862328837c1c, 0x70ead9760ba8, 0x07da4b6fcdce},
    {0x0ff80000077e012, 0x1172e358700271, 0x63d6501573b4d, 0x775288793a0, 0x14bb8e21866},
    {0x01ffe03f03ffc104, 0x30a1f72072a741, 0xa4c006bb09430, 0x4f2d355c7f98, 0x0cd36f7b57c},
    {0x000000000012012, 0x5edc4fa65cd612, 0xe88c757d05b63, 0xc8b487d8528, 0x07c2987ce2a},
    {0x0000000104000104, 0x7f0672769e600c, 0x234ecd09c6285, 0x38310a7ff929, 0x04ab84721e4},
    {0x0010008000014016, 0x4a271b24a15c9a, 0x5825170abee2c, 0x4860388bc30a, 0x05c3736f458},
    {0x0020060120028114, 0x173ede9d1dfa0, 0x2816247ec16f1, 0x46447123e58f, 0x159bd06cfd2},
    {0x0e001881c0754116, 0x3a35cab1e5e287, 0xaa7dad2018eed, 0x0ca6a442abe4, 0x0d9f5e2bd96},
    {0x71d81f01fb80e111, 0x48d1a4f8d0e015, 0x43db4dee649bb, 0x2940aa4aea57, 0x169e0af8cee},
    {0x01c00700380ca104, 0x5adaaead5e39a1, 0x5e5d41686b186, 0x2b5379111b18, 0x08e65aa5574},
    {0x0fc7e07f078f0010, 0x680a74c2db72ab, 0xe1e7e9e257df7, 0x70d5e9111b6e, 0x1656b4b58a2},
    {0x01e0000000f8102, 0x526b6108440381, 0x14591771717f79, 0x21735e75c963, 0x09f5d9b72de},
    {0x0007f87fc3fd0004, 0x10d8b20cc2d16d, 0x1a079270e423e, 0x598f11394622, 0x060ebc9bcf4},
    {0x700fffbef87fa011, 0x6ce992764e3e06, 0xed334eb23c6f, 0x5f36f89389ba, 0x01b442b62f8},
    {0x01e01e80e000c106, 0x1251a9fa5ac371, 0x1e0d55467df96, 0x06790d966652, 0x040a2cc1d52},
    {0x71df97edc73e015, 0x0a7d90ee25a348, 0x095ae04d250fb, 0x788a4bb5b473, 0x0180063026c},
    {0x0fe81840c473c114, 0x0ee9c38438b036, 0x813bec99acd9d, 0x7f5d9d9bfe68, 0x000a004140a},
    {0x71e7e63f247d8113, 0x58b2acc1df8c9a, 0x19ef238355322, 0x339c5a67480c, 0x01000220044},
    {0x7017e7be387ae107, 0x1afecbeb09a982, 0x91d1a9b679f82, 0x396e0fc828a2, 0x00080001002},
    {0x71e01901df822011, 0x263665739af91f, 0xf6a6f3cea50b0, 0x21d74a9656e8, 0x0000200004},
    {0x0fd01881c07a6104, 0x75394e09c4b2ca, 0x5cb6b8c1e7ff4, 0x6188995f3157},
    {0x71ddf97edc73e015, 0x0a7d90ee25a348, 0x095ae04d250fb, 0x788a4bb5b473, 0x0180063026c},
    {0x0fdff8ffc78a2116, 0x5286ab7e83ae62, 0xda9574e678f90, 0x06ce5db5ad5c},
    {0x71c007813b8ce115, 0x4d3bcd8f8d85adc, 0x5c23f978ed0f3, 0x1e2c621f971a},
    {0x71e000c0078ce117, 0x017dde9b20b0d5, 0x6cf46e5a6f0f7, 0x7a505dc81443},
    {0x7e381841c7fce115, 0x027f464d66ea0c, 0xac0db23cd9139, 0x2613f7da17b5},
    {0x0fe7e0ff038ec114, 0x13c4979887ba9d, 0x5db01e5be4eee},
    {0x01d01801c0030112, 0x4cb8bb21bf4304, 0xb6f0039dacb41},
    {0x7007e6bf24726101, 0x16b8f3440cf01a, 0xe32af56d4470e},
    {0x70301fc1ff876003, 0x4e392514dce977, 0xd2ae9a043e26a},
    {0x01fffebfe7f08000, 0x005c0f0080a00a, 0xc2b205c5ebe1e},
    {0x7e30004007f98001, 0x29112080522885},
    {0x7fe7e1fe180f6001, 0x3884184171249c},
    {0x7e17f8bfc4044005, 0x0400c0080900e0},
    {0x0fc80640207d4010},
    {0x0ff7fe7ee388a100},
    {0x7ff01e80e3f9e007},
    {0x01d801c11c06a016},
    {0x01ffe73e3bf46112},
    {0x0fc000c0047f6100},
    {0x01c7fffff0e000},
    {0x7ff81fc0fff02001}
};

```

9.2 Driver Code

```
#include <iostream>
#include <iomanip>
#include <sstream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <stdint.h>
#include "t258.h"

using namespace std;

#define Px(x) "0x" << setw(16) << setfill('0') << hex << (x)
#define Pu(x) setw(18) << setprecision(16) << setfill(' ') << fixed << dec << (x)

void Vtab(uz seed, bool qz) {
    int NL = 8, NS = 4, n = 5; // # large, small, sample
    rng *z[NL][NS]; // stream and substream rngs

    for(int i=0; i<NL; i++) // large jumps
        for(int j=0; j<NS; j++) { // small jumps
            z[i][j] = new rng(seed);
            z[i][j]->init().jumpk(i, rng::JL).jumpk(j, rng::JS);
        }

    cout << "Vtab(" << Px(seed) << ", " << (qz ? "uz": "u01") << "): "
         << "jump = J*2^" << dec << rng::JL << " + K*2^" << rng::JS
         << endl << endl;

    for(int i=0; i<NL; i++) { // large jumps
        cout << setw(22) << setfill(' ') << "J = " << setw(3) << i << endl;

        cout << " n ";
        for(int j=0; j<NS; j++)
            cout << setw(16) << "K = " << setw(3) << j;
        cout << endl;

        for(int k=1; k<=n; k++) { // samples
            cout << setw(3) << dec << setfill(' ') << k << " ";
            for(int j=0; j<NS; j++) // small jumps
                if(qz)
                    cout << " " << Px(z[i][j]->gen()); // integer
                else
                    cout << " " << Pu(z[i][j]->u01()); // U(0,1)
            cout << endl;
        }
        cout << endl;
    }
}

void Vjump(uz seed, uz pmax) { // inductive jump verification
    rng z0(seed), z1(seed); // T258 objects, same state

    cout << "Vjump(" << Px(seed) << ", " << dec << pmax << "): ";

    if( z0.jump(0) != z1.t() ) { // base
        cerr << "fail base check" << endl;
        exit(1);
    }

    for(uz p=1; p<=pmax; p++) // induction
        if( z0.init().jump(p) != z1.init().jump(p-1).jump(p-1) ) {
            cerr << "fail induction check, p=" << dec << p << endl;
            exit(1);
        }

    cout << "for all p = 0:" << pmax << " jump(p) == t^(2^p)" << endl;
}

int main (int argc , char* argv[]) {
    uz seed = 0xd8940e83ec602c7b; // rng seed default
    uz pmax = 172 + 64;

    Vtab(seed, true);
    Vtab(seed, false);
    Vjump(seed, pmax);
}
```

Approved for public release; distribution is unlimited.

10. Conclusions and Recommendations

The T258 RNG passes tests for randomness and provides high-resolution 53-bit random real numbers.

T258 class RNG objects are intrinsically thread-safe in the sense that instances do not interact and are thus computationally independent. But this is not sufficient for statistical independence.

The enumeration of threads and initialization of thread RNG streams at offsets of 2^{172} ensures that the 2^{86} available threads are statistically independent and that computation are easily reproducible. Within threads, increments of 2^{86} provide statistical independence of 2^{86} stochastic quantities with stream length 2^{86} . Verification of the jump computations guarantees these independence properties.

When C++11 compilers become widely available, a T258 engine can be incorporated into the C++11 `<random>` library framework. Care must be taken to preserve the independence properties. Otherwise, as noted in Section 1, no sets of quantities within or among threads can be claimed to be independent random samples.

11. References

1. Kernigan BW, Ritchie DM. The C programming language. 2nd ed. Upper Saddle River (NJ): Prentice Hall; 1988. [3](#)
2. Collins JC. Testing, selection, and implementation of random number generators. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2008 Jul. Report No.: ARL-TR-4498. [3](#), [8](#), [11](#), [12](#), [13](#)
3. L'Ecuyer P. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*. 1996;65(312):203–213. [10](#), [15](#)
4. L'Ecuyer P. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*. 1999;68(225):261–269. [13](#), [16](#)
5. L'Ecuyer P. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*. 1999;68(225):249–260. [27](#)

List of Symbols, Abbreviations, and Acronyms

CDF	cumulative distribution function
CP	characteristic polynomial
LCG	linear congruential generator
LFSR	linear feedback shift register
QT	QuickTaus
RNG	random number generator
STL	Standard Template Library

1 DEFENSE TECHNICAL
(PDF) INFORMATION CTR
DTIC OCA

2 DIRECTOR
(PDF) US ARMY RESEARCH LAB
RDRL CIO L
IMAL HRA MAIL & RECORDS MGMT

1 GOVT PRINTG OFC
(PDF) A MALHOTRA

1 DIR US ARMY EVALUATION CTR
(PDF) HQ
TEAE SV
P A THOMPSON

1 DIR USARL
(PDF) RDRL SL
P BAKER

21 DIR USARL
(PDF) RDRL DPW
T HOLDREN
RDRL SL
D BAYLOR
P DISALVO
T STADTERMAN
RDRL SLB
R BOWEN
G MANNIX
RDRL SLB D
J COLLINS
J EDWARDS
R GROTE
L MOSS
E SNYDER
RDRL SLB E
M MAHAFFEY
RDRL SLB G
N ELDREDGE
RDRL SLB S
R DIBELKA
C KENNEDY
D LYNCH
M PERRY
R SAUCIER
G SAUERBORN
RDRL SLB W
S SNEAD
RDRL SLE
R FLORES

