Preprint ver. 1.42, of 97-06-26

# Fast generation of nonuniform random numbers

Martin Mössner
Dept. of Sport Science, University of Innsbruck

In this paper we develop grid-like techniques for fast generation of nonuniform random numbers. The given method is applicable to (continuous) bounded densities and is demonstrated for normal and exponential random number generation. Parameter selection is governed by algorithmic efficiency and computer oriented implementation. The proposed generators are compared to several well known algorithms.

Categories and Subject Descriptors: AMS: 65C10, GAMS: L6a14 [**Random Number Generation**]: Normal and Exponential Distribution

Additional Key Words and Phrases: Mathematics of Computing, Probability and Statistics, Random Number Generation, Normal and Exponential Distribution

## 1. INTRODUCTION

In simulation it is common practice to model data errors with random numbers (RN). Because of unknown distribution properties of the distribution of errors and because of the central limit theorem researchers often use normal distributed RNs. In many cases the mean is assumed to be zero and the standard deviation is known from experimental experience. Therefore good and fast RN generators (RNG) for normal distributed RNs are needed. The exponential distribution is used, for instance, to model decaying processes. Both normal and exponential distributed RNs are used to sample from other distributions.

There exist several algorithms for generating normal and exponential distributed RNs. Simple implementations are given by inversion of the distribution function: Let $U$ and $V$ be uniform distributed, then $E = -\log(U)$ is exponential distributed and $\sqrt{2E}\ \sin(2\pi V)$ as well as $\sqrt{2E}\ \cos(2\pi V)$ are normal distributed RNs (Box-Muller [6]). The evaluation of the trigonometric functions can be avoided by using the polar method [4]. Another well known method is motivated by the central limit theorem: $N = \sum_{i=1}^{12} U_i - 6$.

Address: Dept. of Sport Science, Fürstenweg 185, University of Innsbruck, A-6020 Innsbruck, Austria. email: Martin.Moessner@uibk.ac.at. URI: http://sport1.uibk.ac.at/isw/mm/

Brent [9] uses sampling from exponential decaying distributions. Forsythe [15], resp., Ahrens and Dieter [1; 2] use a comparison method. Leva [23; 24] uses the ratio of uniforms method with special squeeze steps. Marsaglia et al. have introduced a rejection algorithm [27] and the 'ziggurat' method [28]. The NAG library [36] implements Brent's algorithm for normal and inversion for exponential distributed RNs and the LAPACK library [3; 14] Box-Mullers method only.

In the following we develop grid based techniques that are applicable for (continuous) bounded densities and apply these methods for normal and exponential random sampling.

## 2. METHOD

**2.1 Uniform distributed RNs.** We need three kinds of random numbers 1) uniform distributed random numbers (RN) in the range of $(0, 1)$ 2) random indices (RI) from 0 to $2^n - 1$, and 3) random signs (RS). With $F$ the distribution function, $F$-distributed RNs are given by $F^{-1}(U)$. Since $F$ usually is a smooth function, any nonrandomness or dependence in the sequence of uniform RNs is mapped to nonuniform RNs. Therefore the underlying uniform generator has to be of high quality. For efficiency both the uniform generator and the inversion of $F$ have to be fast.

Since there exist no rigorous criteria, what a good generator should be, several authors have developed their own testing procedures and hope to have the ultimate test. Nevertheless each of these tests inspects some sort of nonrandomness. As minimal requirement a modern generator needs to have a large period ($\geq 10^{20}$) and has to pass common available testing suites (e.g. the DIEHARD package [12]). Furthermore tests, known to be stringent, should be passed (e.g. birthday test [26], monkey tests [30], lattice and spectral tests [18; 22], ...). Overviews on RNGs were given by Marsaglia [26], L'Ecuyer [22], and Hellekalek [16].

Because of 2) and 3) we use a generator that returns 32-bit RNs. Generators based on prime moduli are, although theoretically superior, of poor help. Since the performance of the nonuniform generator linearly depends on that of the uniform generator, we focus on the fast ones among the good ones. Candidates are the KISS[1] generator (Marsaglia [12]), Marsaglias SWB[2][3] generator (Marsaglia and Zaman [29; 12]), the combined Tausworthe generator taus88 (L'Ecuyer [20]), the TGFSR[4] generator tt800 (Matsumoto and Kurita [32; 33]), and the MT[5] generator mt19937b (Matsumoto and Nishimura [34]). Especially the last one astonishes because of its excellent theoretical characteristics.

We use a Fortran version of mt19937b. It's period is $2^{19937} - 1$, that is about $10^{6000}$. Never a computer will be able to exhaust a significant part of the period. The generator is equidistributed up to 623 dimensions and performs well in the tests of the DIEHARD package. The algorithm is well suited for implementation in 32-bit arithmetic and results in a rather fast code. A parallel implementation may

[1] Keep It Simple Stupid.
[2] Subtract With Borrow.
[3] $x_n = x_{n-24} - x_{n-37} - borrow \equiv 2^{32}$, $y_n = 30903\, y_{n-1} + carry \equiv 2^{16}$, $u_n = x_n + y_n$.
[4] Twisted Generalized Feedback Shift Register.
[5] Mersene Twister.

be obtained by the suggestions of Masuda and Zimmermann (see sec. 2.3 of [31]).

In order to get a correct inversion of the distribution function, we use full precision uniform RNs, that means all mantissa bits are uniformly distributed. According to IEEE 754 standard a d.p.[6] number consists of 53 mantissa bits, a sign bit, and 11 exponent bits[7]. Therefore, we need two RNs $u$ and $v$ for conversion to uniform distribution:

$$U = (ur + v)r + \frac{r+1}{2}, \qquad r = 2^{-32}. \tag{1}$$

The lower 11 bits of $u$ underflow in conversion. These bits are reused for RSs and RIs. Above this, operations on uniform RNs result in further bit losses. For instance, IEEE 754 forces 48 bits for transcendental math functions, such as the logarithm, only. Analogously we can do the conversion for s.p. numbers. One has 24 bits for mantissa, one for sign, and 8 for exponent. The conversion reads $U = ru + \frac{1}{2}$.

**2.2 Reduction to finite intervals.** For simplicity we assume to have the distribution $f$ defined on the positive real axis $x > 0$. Let $p$ be the tail probability

$$p = \int_e^\infty f(x)\, dx = 1 - F(e). \tag{2}$$

Then the discrete mixture algorithm returns with probability $p$ a random deviate of the tail distribution $\frac{1}{p}f$, $x > e$ and else a deviate from the restricted distribution $\frac{1}{1-p}f$, $x \in [0, e]$. In order to substitute the uniform RN, needed by the mixture technique, by a RI, we restrict $p$ to $n_t 2^{-n}$ for a proper $n = 1, 2, \ldots$ and small $n_t = 1, 2, \ldots$ The root $e$ of $F(e) = 1 - n_t 2^{-n}$ can be computed by Brent's zero finding technique [7].

**2.3 Method I.** We rewrite the restricted distribution as $cg(x)$ with $c > 0$ an arbitrary constant. This constant can be used, for instance, to simplify the evaluation of $g$. The specific value of $c$ is not needed for the algorithm. Next we divide the $x$-axis in $n_x$ and the $y$-axis in $n_y$ intervals (see Fig. 1). This division defines a grid of $n_g$ 'good' and $n_b$ 'bad' rectangles, each of size $\delta_x \times \delta_y$, $\delta_x = \frac{e}{n_x}$ and $\delta_y = \frac{M}{n_y}$, $M = \max_{x\in[0,e]} g(x)$. The good rectangles are under the curve $g$ and the bad ones are those that intersect. Because of practical reasons we allow further $n_u$ unused rectangles which can be thought as rectangles above the curve. We number the rectangles, first the bad followed by the good and the unused ones, by $z$ and tabulize the left lower corner $(x_z, y_z)$.

RN sampling is done by rejection using as comparison function the envelope of the rectangles. Getting a random point under the comparison function divides into two steps: 1) selecting a rectangle randomly and 2) selecting a random point in the rectangle. The first step is done by choosing a RI $z$ and the second by computing uniform distributed $U$, $V$ and setting $(X, Y) = (x_z + U\delta_x, y_z + V\delta_y)$. For good rectangles the rejection condition $Y > g(X)$ is false in any case. Therefore $X$ can be accepted immediately and $Y$ is not needed. We arrive at

[6] s.p. single precision, d.p. double precision

[7] The leading bit of the normalized mantissa is not stored.

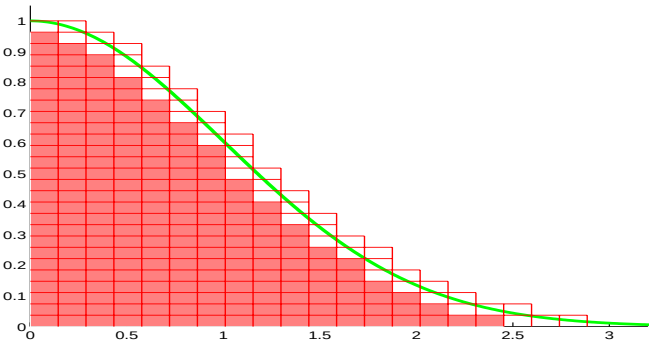


Fig. 1. Grid discretization for method I

**do**
  get a RI $z$ between 0 and $n_b + n_g + n_u + n_t - 1$
  **if** $(z < n_b + n_g)$
    get uniform distributed $U$. $X = x_z + U\delta_x$.
    **if** $(z >= n_b)$ **exit**
    get uniform distributed $V$. $Y = y_z + V\delta_y$
    **if** $(Y < g(X))$ **exit**
  **else if** $(z < n_b + n_g + n_u)$
    **cycle**
  **else**
    get a random deviate $X$ from the tail density
  **end if**
**end do**

The algorithm needs RIs $z$ between 0 and $n_b + n_g + n_u + n_t - 1$. For a fast method we need $2^n = n_b + n_g + n_u + n_t$ and a large quick acceptance rate $r = n_g\ 2^{-n}$. If possible, there should be no unused rectangles, $n_u = 0$, and the tail probability should be kept small, $n_t = 1$, $n$ large.

**Normal distributed RNs**. The free constants were selected by doing a computer search for $n = 6, 7, 8$, and 9. With $n = 6, 7$, and 8 we reached rates $r$ of $0.67, 0.75$, and 0.81. For $n = 9$ we did not find any reasonable values of $n_x$ and $n_y$ that result in a high rate. Therefore we decided to implement the method with $n = 8$. We have $n_x = 20$ and $n_y = 27$, which leads to $n_g = 208$ good and $n_b = 46$ bad rectangles (see Fig. 1). Furthermore we have $n_t = 1$. In order to get a table of 256 elements we need one unused rectangle, $n_u = 1$. The portion of good rectangles is 81%. The tail probability $p$ is $\frac{1}{256}$ and the end-point $e$ is $\sqrt{2}\ \mathrm{erf}^{-1}(\frac{255}{256}) = 2.885634912426757147 38\ldots$

**2.4 Method II.** Although method I results in a quite fast algorithm, it is difficult to transfer to other distributions. Therefore we developed a second approach that parts the rejection area in $m - 1$ equal areas (see Fig. 2).

Let $0 = x_1 < x_2 < \ldots < x_m = e$ be the grid points along the $x$-axis and $h_z$ the heights of the rectangles. The area of the $z$-th rectangle is $A_z = h_z(x_{z+1} - x_z)$. We denote by $l_z = \min_{x\in[x_z,x_{z+1}]} g(x)$ and $u_z = \max_{x\in[x_z,x_{z+1}]} g(x)$ the lower and upper bounds of the scaled distribution function $g$. For setup we have to find numbers $x_z, h_z$ with $A_1 = \ldots = A_{m-1}$ that obey $g(x) < h_z$ and $h_z$ gets as small

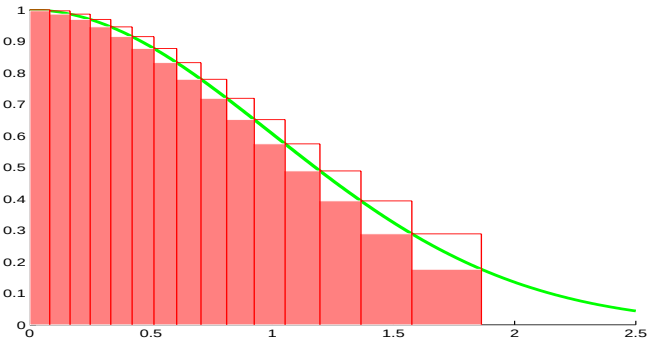


Fig. 2. Grid discretization for method II

as possible. It is clear that $h_z = u_z$ is the best choice, but $h_z \in [u_z, u_z + \epsilon]$ with a small constant $\epsilon$, say 0.001, will work too. Any constant less equal $l_z$ can be used for quick acceptance steps.

For decaying densities, such as the halfnormal or the exponential density, one has $l_z = g(x_{z+1})$ and $u_z = g(x_z)$. The numbers $x_z$ are solutions to the optimization problem: $g(x_z)(x_{z+1} - x_z) = x_2 M$. To apply the Newton technique (e.g. Deuflhard [10]) we had to implement the residuals and their Jacobian. Because of the simplicity of the model we could supply analytical expressions for the Jacobian. With the initial guess $x_z = \frac{(z-1)e}{m-1}$ the damped Newton steps converge after some few iterations to the solution. The optimization process is very stable and results in residuals near the relative machine precision.

In the case of unimodal distributions the situation is similar. As long as $x_{z+1} < mode$ one has $l_z = g(x_z)$, $u_z = g(x_{z+1})$, for $x_z < mode < x_{z+1}$ we have $l_z = \min(g(x_z), g(x_{z+1}))$, $u_z = g(mode)$ and for $mode < x_z$ $l_z = g(x_{z+1})$, $u_z = g(x_z)$. The general case is tricky since the heights $h_z$ of the rectangles depend on the locations of the grid points $x_z$. In this case one can start with $x_z = \frac{(z-1)e}{m-1}$, $h_z = M$ and setup an optimization process for both $x_z$ and $h_z$.

RN sampling is done in the same way as in method I. Selecting a random point in the rejection area requires 1) random selection of one of the $m-1$ rectangles and 2) getting a random point in the rectangle. The first step needs a RI $z$ and the second one uniform distributed $U, V$. Then $(X, Y) = (x_z + U(x_{z+1} - x_z), h_z V)$ is a random point in the rejection area. A quick acceptance step can be done by comparing $Y$ with $l_z$. Summarizing we have the algorithm

```
do
    get a RI z between 0 and m − 1
    if (z > 0)
        get uniform distributed U. X = x_z + U(x_{z+1} − x_z)
        get uniform distributed V. Y = h_z V
        if (Y < l_z) exit
        if (Y < g(X)) exit
    else
        get a random deviate X from the tail density
    end if
end do
```

For good performance one selects $2^n = m$. We have implemented the method for the normal and exponential distribution with table sizes of 64 and 256 elements.

**2.5 Tail of the distribution.** To complete the algorithm we have to give a method for calculating RNs of the tail densities. Since this branch will be called in mean in $n_t$ of $2^n$ cases only, we do not need an optimal method. For the tail of the exponential distribution one can use inversion: $X = e - \log(U)$ and for the normal distribution rejection from the exponential density (Devroye [11]): Compute exponential distributed $E$ and $F$ until $E^2 < 2e^2F$, then set $X = e + \frac{E}{e}$. Marsaglia [28] describes a rather general technique for rejection from exponential ($f(x) < ce^{-ax}$) and polynomial ($f(x) < c(1+bx)^{-a}$) decaying distributions.

**2.6 Further Improvements.** There exist several improvements that speed up the overall performance considerably.

**Premultiplication of constants**. Whenever products like $cU$ with $U = (ur + v)r + \frac{r+1}{2}$ have to be formed the products $c\,r$ and $c\,\frac{r+1}{2}$ can be precalculated. In case of the comparison $c_1U < c_2$, it is faster to compare $ur + v$ with $(\frac{c_2}{c_1} - \frac{r+1}{2})/r$. Sometimes it is even sufficient to compare $v$ with a slightly larger constant.

**Squeezing** can be used to avoid the evaluation of the rejection condition $Y > g(X)$. In each grid interval we sandwich $g$ using a polynomial function: $s_z(x) - \epsilon < g(x) < s_z(x) + \epsilon$ of degree $k$ (see Fig. 3). For implementation we consider linear ($k = 1$) and quadratic ($k = 2$) polynomials only.

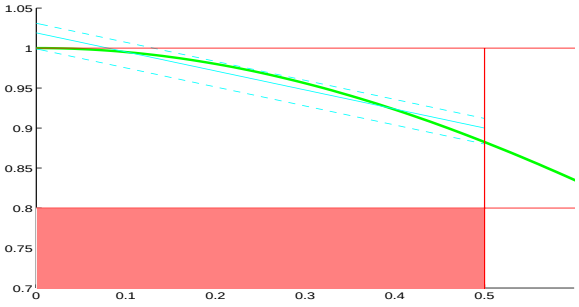


Fig. 3. Linear squeezing functions

For each grid interval $[x_z, x_z + \delta_z]$ (method I: $\delta_z = \delta_x$ method II: $\delta_z = x_{z+1} - x_z$) we compute coefficients $a_{z,i}$, $i = 0, \ldots k$ that minimize

$$\min_{a_{z,i}, i=0,\ldots k} \int_{x_z}^{x_z+\delta_z} \left( g(x) - \sum_{i=0}^{k} a_{z,i} x^i \right)^2 dx. \tag{3}$$

Setting

$$I_{z,i} = \int_{x_z}^{x_z+\delta_z} x^i g(x)\, dx \tag{4}$$

we have to solve the linear system

$$\left( \frac{1}{i+j+1} ((x_z + \delta_z)^{i+j+1} - x_z^{i+j+1}) \right)_{i,j=0,\ldots k} (a_{z,i})_{i=0,\ldots k} = (I_{z,i})_{i=0,\ldots k} \tag{5}$$

to get the coefficients $a_{z,i}$ for the polynomial approximations $s_z$. Finally we use Brents minimization technique [8] to calculate

$$\epsilon_z = \max_{x \in [x_z, x_z + \delta_z]} |g(x) - s_z(x)| \tag{6}$$

and set $\epsilon = \max_z \epsilon_z$. Since this type of squeezing bounds is very sharp the rejection condition $Y > g(x)$ will be evaluated hardly anytime.

## 3. IMPLEMENTATION

**3.1 Implementational details**. For runtime comparison we obtained several implementations from the net (GAMS [5], netlib [13]). Using the original codes, it is scarcely possible to judge the various algorithms against each other. Every RNG uses its own uniform RNG. Some of them do a function call for every RN, some are vectorized, few are even parallel. Last but not least most generators are given in s.p. Almost all use reduced precision for uniform RNs.

Since we are coding in Fortran 90/95 at least some porting had to be done. Therefore we decided to unify all algorithms. For portability and multiprecision coding we introduced the module `global` containing the statement `INTEGER, PARAMETER :: kn = KIND (1.0d0)`. By changing `1.0d0` to `1.0` and recompiling all sources one obtains a s.p. version of our sources. Whenever appropriate we use code segments like

```
IF (kn == KIND (1.0)) THEN
   ... ! code for s.p.
ELSE
   ... ! code for d.p.
end IF
```

Current compilers evaluate, during compilation, the if condition, notice it is a constant expression, and eliminate the branch that is never executed. Therefore the given coding technique does not result in any performance loss. However, one has to take care not to introduce expressions, that will not be optimized away during compilation.

Random bits are generated by the generator mt19937b of Matsumoto and Nishimura [34] and are saved in a pool `uu` of size `nu = 624`. The generator may be initialized either by default initialization: `CALL set_rng_seed ()` or by supplying a single integer seed `s`: `CALL set_rng_seed (seed=s)` or by giving the whole seed array `seeds`: `CALL set_rng_seed (seed_vec=seeds).` The actual seed state may be obtained by calling: `CALL get_rng_seed (seeds).` Whenever the pool is exhausted a service routine is called to fill the pool: `CALL get_uni ().` For conversion to uniform distribution we use the constants `r1 = 0.5_kn**32` and `r2 = (r1 + 1) / 2`. Uniform RNs $U$, RIs $z$ in the range of 0 to 255, and RSs $s$ are formed by

```
IF (kn == KIND (1.0)) THEN
   IF (iu > nu-1) CALL get_uni ()
   u = uu(iu) * r1 + 0.5_kn
   z = IAND (uu(iu), 255); s = BTEST (uu(iu), 8)
   iu = iu + 1
ELSE
   IF (iu > nu-2) CALL get_uni ()
```

```
      u = (uu(iu) * r1 + uu(iu+1)) * r1 + r2
      z = IAND (uu(iu), 255); s = BTEST (uu(iu), 8)
      iu = iu + 2
   end IF
```

The typical code of a RNG looks like

```
   SUBROUTINE rng (rn)
      REAL (kind=kn), INTENT (out), DIMENSION (:) :: rn
      INTEGER l, ...; REAL (kind=kn) ...
      DO l = 1, SIZE (rn)
         ... ! code for computing one RN
         rn(l) = ...
      end DO
   end SUBROUTINE rng
```

**3.2 Calling sample**. We give a code fragment, that explains how to use the RNGs. Whenever ellipsis occurs the user may insert his own code.

```
MODULE user_module
   USE global, ONLY : kn   ! kn = KIND (1.0d0)
   USE UniformMod, ONLY : set_rng_seed, get_rng_seed
   USE NormalMod, ONLY : normal_01_rn
   USE ExponentialMod, ONLY : exponential_01_rn
   IMPLICIT NONE
   ...
CONTAINS
   SUBROUTINE user_sub (e, n, mu, sigma, ...)
      REAL (kind=kn), INTENT (in) :: mu, sigma
      REAL (kind=kn), INTENT (out), DIMENSION (:) :: e, n
      INTEGER, SAVE :: s = 123456789
      INTEGER, SAVE, DIMENSION (624) :: seeds
      ...
      ! default initialization of the RNG
      CALL set_rng_seed ()
      ! initialize RNG with integer seed s
      CALL set_rng_seed (s)
      ! get normal distributed random numbers
      CALL normal_01_rn (n)
      ! convert to   normal(mu,sigma)
      n = n * sigma + mu
      ! get seed state of RNG
      CALL get_rng_seed (seeds)
      ...
      ! initialize generator with seed vector
      CALL set_rng_seed (seed_vec=seeds)
      ! get exponential distributed random numbers
      CALL exponential_01_rn (e)
      ...
   end SUBROUTINE user_sub
   ...
end MODULE user_module
```

## 4. TESTING - RESULTS

**4.1 Validation**. In contrast to uniform RNs there exist few tests for assessing the quality of a sequence of nonuniform RNs. A first class of tests computes the empirical distribution function and tests it against the underlying distribution. This can be done by chi-square or Kolmogorov-type tests [18]. Using the Anderson-Darling test statistic [35] one has a good tool for investigating whether the tails of the distribution are correct or not. On the other side these tests are far to weak. They will reveal errors in implementation, but will not detect weakness of a particular RNG. Another class of tests on RNs, the (serial) correlation tests [18], investigate the independence of subsequent RNs. These tests are satisfied by the given RNGs.

Another idea is to transform the nonuniform RNs to uniform ones and supply a testing package for uniform RNs. For exponential distributed $E$ and $F$ we have $\frac{E}{E+F}$ uniform distributed, and for normal distributed $X$ and $Y$ we have uniform $\exp(-\frac{X^2+Y^2}{2})$. However, results based on this technique have to be taken with care, as following example shows: The LAPACK generator [3; 14] for uniform RNs fails some of the tests in the DIEHARD package but backtransformations of normal distributed RNs pass all of the tests.

**4.2 Runtime measurements**. In order to get processor and compiler independent information we did runtime measurements on following systems:

**SGI 1**: SGI server, 4 processors, CPU R4400, FPU R4010, 150 MHz, 16 KB data-, 16 KB instr.-, and 1 MB sec. cache, 512 MB memory, IRIX 5.3, NAGWare f90 ver. 2.1(591), option: -O.

**SGI 2**: SGI workstation, CPU R4000, FPU R4010, 100 MHz, 8 KB data-, 8 KB instr.-, and 1 MB sec. cache, 64 MB memory, IRIX 5.3, NAGWare f90 ver. 2.2(305), option: -O, Nag Fortran Library fl90, rel. 1.

**HP 1**: HP 9000/735 workstation, PA7100, 99 MHz, 256 KB data- and 256 instr.-cache, 80 MB memory, HP-UX A.09.01, NAGWare f90 ver. 2.1(561), option: -O.

**HP 2**: HP 9000/715 workstation, PA7100LC, 75 MHz, 256 KB data- and 256 KB instr.-cache, 96 MB memory, HP-UX A.09.05, NAGWare f90 ver. 2.1(561), option: -O.

**PC 1**: PC, Pentium, 120 MHz, 8 KB data-, 8 KB instr.-, and 512 KB sec. cache, 32 MB memory, MS-DOS 6.22 with Phar Lap DOS extender, ver. 7.0, Lahey lf90, ver. 2.00a, options: -o2 -tp (-o3 performs interprocedural optimizations).

**PC 2**: PC, Pentium, 120 MHz, 8 KB data-, 8 KB instr.-, and 512 KB sec. cache, 32 MB memory, DOS 6.22 with DBOS DOS extender, ver. 3.00, Salford ftn90, ver. 2.05, option: /optimise.

These machines range from mid range server over slow workstations to standard PCs. Modern workstations are 3 to 5 times faster.

In praxis it is valuable to know the total cpu-time needed to produce a certain amount of RNs. For that purpose we measure the cpu-time required for providing the RNs in an user defined array, i.e. we add up time for generation, function calling and, perhaps, copying. Since computing time for few RNs is negligible, we observe runtimes for large samples only. With respect to function calling and caching it is advisable to calculate RNs in blocks of medium size.

In Fig. 4 we show the performance of uniform random sampling with respect to $n$, the number of RNs computed in a single call to the RNG.

The overhead for computing single RNs is considerable large. Additional computing times range form 30 to 160 percent. The situation is even worse for nonuniform RN generation. The effect of function calling is observable up to vector length

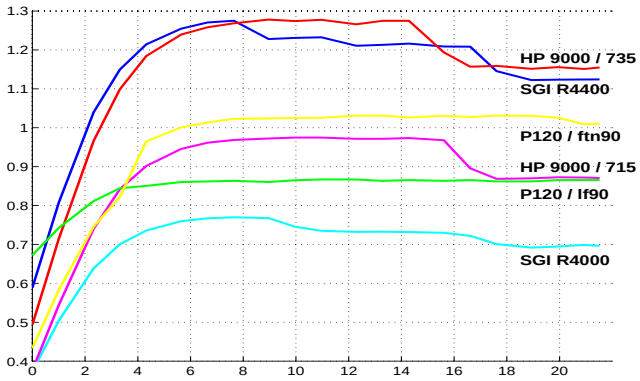


Fig. 4. Performance of the d.p. version of the uniform RNG mt199937 with respect to the vector length. Shown is the number of RNs per $\mu$s versus $\log_2(n)$, with $n$ the number of RNs computed per function call.

of about 64 elements. Remarkable is the difference for the two PC configurations which use the same hardware but different compiler technology. ftn90 is a port of NAGWares UNIX compiler, which himself is organized as preprocessor to C. Its performance relies on well tested (hosted) C technology. On the other side lf90 calls himself to be a native compiler. This explains the low costs for function calling but contradicts to the smaller overall efficiency of the compiler.

On the other side, for large data sizes, the caches will overflow and, hence, result in a particular loss of performance ($10 \leftrightarrow 8$ KB, $15 \leftrightarrow 256$ KB, $17 \leftrightarrow 1$ MB). This effect amounts to approximately 10 percent. On the PCs the influence of the DOS extenders probably exceeds the influence due to caching.

In Tab. 1 and 2 we collect runtime measurements for d.p. and s.p. RNGs. We give absolute timings in $\mu$s per random deviate and relative timings, i.e. ratios of cpu-time for nonuniform RN per uniform one. Absolute time measurements are mean values of 25 runs of a particular generator. In each run we computed 1000 vectors of 1000 RNs. Standard statistics on the 25 values indicate good repeatability of the time measurements (s.d. $< 0.02$ $\mu$s).

## 5. DISCUSSION

Uniform RN generation needs, depending on machine, 0.50 to 0.82 $\mu$s for s.p., resp., 0.78 to 1.34 $\mu$s for d.p. This is about twice as fast as common available library routines, such as the LAPACK or the NAG generator. mt19937 is a high quality generator with an extraordinary large period. On the other side both the LAPACK[8] and the NAG[9] generator are LCGs[10]. These type of generators is known to show a typical lattice structure [16; 22]. The LAPACK generator additionally fails some tests of the DIEHARD package. In Fortran one could use the intrinsic RNG. We do not use this generator, since for any compiler one does not know

[8] $u_{n+1} = au_n \equiv 2^{48}$, $a = 33952834046453$.
[9] $u_{n+1} = au_n \equiv 2^{59}$, $a = 13^{13}$.
[10] Linear Congruental Generator.

see moetab.tex

Table 1. Runtimes for d.p. RNGs

see moetab.tex

Table 2. Runtimes for s.p. RNGs

anything about the quality of the intrinsic RNG. A further drawback is, that all given compilers do not make a difference between s.p. and d.p. RNs.

The best algorithms for normal and exponential RN generation need few additional computing time with respect to uniform RN generation. Computing times for normal RNs range from 0.68 to 1.21 $\mu$s for s.p., resp., 0.94 - 1.80 $\mu$s for d.p. and for exponential RNs from 0.65 to 1.21 $\mu$s for s.p., resp., 0.93 - 1.77 $\mu$s for d.p. Computing times do not differ significantly, since our method is uniformly fast with respect to distribution function. Additional computing times range from 20 to 50 percent and are somewhat larger for s.p. RN generation. The efficiency of our generator is markably better than most well known realizations. Library solutions, such as NAG's implementation, are considerable less efficient. Beyond that, recently published algorithms, such as Leva's method, can't stand with.

The only method that is competitive is Marsaglia and Tsang's 'ziggurat' method. This method is a sort of grid technique, too. In contrast to our method it uses horizontal stripes. Because of this the method is applicable to decaying distributions and its symmetric counterparts, only. Random sampling with our method may be implemented efficiently for (continuous) bounded distributions. Setup is fast for the class of unimodal distributions. The speed of the 'ziggurat' and our methods is comparatively fast with a small profit of our 2nd method on workstations. Compared to Marsaglias original suggestion (64 element table) we got performance improvements up to 22 percent on HP workstations.

For further improvement of absolute computing times one has to search for a faster uniform RNG. As long as one does not release quality properties such as 'independence' and 'randomness' of the uniform bitstream, there is few hope to get further improvements for the nonuniform part of the generator. Efficiency improvements for the uniform generator may be obtained either by new algorithms or by implementation. Coding the RNG in assembly language or even realizing it by hardware usually results in high performance gains. On the other side coding the generator in C does not result in any runtime improvement. The d.p. version of mt19937 needs 1.356 $\mu$s in Fortran and 1.342 $\mu$s in C. Finally one could switch to a parallel computing environment (e.g. [31]). This technique will get growing attention soon, since dual Pentium PCs running under Windows NT are available already yet.

The choice of the underlying uniform RNG makes no difference for of our method. Relative timing results do not depend significantly on the particular choice of the uniform RNG. If one likes another generator he should use it. For efficiency we recommend to use a generator, that returns 32-bit integers. During development we tried out various uniform generators. From that experience we favor the generators mentioned in sec. 2. Finally we decided to use mt19937 because of its quality (period, equidistribution properties, passes all tests of the DIEHARD battery) and since its realization is comparatively fast. It is barely slower than the simple LCG. The only drawback is the relative large size of the internal seed pool. If one likes smaller seed pools, we recommend to use tt800 [32; 33] or even simpler taus88 [20]. Other generators, like RANLUX [17; 25] seem to be, at least at safe luxury levels, to slow, or use prime moduli, such as L'Ecuyers combined generators [19; 21], or do not stand our quality demands. On a 64-bit processor it probably is sufficient

to use a good 64 bit MWC[11] generator, or even simpler a LCG[12] generator.

From Fig. 4 we obtain, that RNs should be calculated in packages of at least 64 numbers. Most applications need vectors of RNs, but there are situation where one needs single numbers or even arrays of RNs. For that purpose it is convenient to implement some interfaces that invokes the generator for the various shapes used in praxis. One further could think to implement the generator as function and not as subroutine. This improves readability of user code but causes allocation and copying of a temporary array. Since the uniform RNG needs data within a save statement, it is not possible to implement the generator as elemental routine.

The investigated processors are optimized for different operations. The SGI's and HP's use RISC processors, that means they use a small instruction set. They are particularly good in array references and slow in complicated actions, such as evaluation of transcendental functions. The P9000 is optimized for floating point calculations and uses a special pipelining technique for loading elements of an array. On the other hand the Pentium is optimized on integer, resp., index calculations and has a fast math-unit for transcendental functions. From that we expect, especially for table methods, an increasing efficiency from the PC to the SGI and to the HP (compare Table 1). In contrast the simple algorithms need several evaluations of transcendental functions and, hence, are most efficient on the PC and the HP.

Table methods need random access to arrays and therefore produce a lot of data traffic in the processor. These methods tend to fill up the inner caching pools of the processor. Whenever this occurs the performance of the generator brakes down. This may be, perhaps, the reason for the slightly larger cpu-time of our 2nd method with the 256 element table on the PC.

With some additional considerations the setup of method II can be implemented independent of distribution for the class of monotoniously decaying and unimodal densities. Up to now we have investigated the behavior of the technique in the case of the exponential power distribution, a distribution family that contains exponential and normal distribution. On the SGI R4000 the generator needs about 0.06 s for initialization and 2.0 $\mu$s per RN. The initialization time does not matter, it has to be done only once. The given technique results in an implementation that is uniformly fast with respect to distribution parameter and, we claim, with respect to distribution function. The setup is fast as long as the density and the distribution function is computable. The generator is fast as long as the density can be evaluated efficiently. However this generalization needs further work and investigations. It will be published elsewhere.

## 6. ACKNOWLEDGEMENTS

We thank the Institute of Astronomy, the Faculty of engineering and the local computing site for providing the various machines for testing. Furthermore we thank P. Kaps for the help in preparation of the paper and P. Hellekalek for comments on the paper.

[11] Multiply With Carry.
[12] Linear Congruental Generator.

REFERENCES


[1] J.H. Ahrens and U. Dieter, *Computer methods for sampling from the exponential and normal distributions*, Communications of the ACM **15** (1972), 873–882.

[2] J.H. Ahrens, K.D. Kohrt, and U. Dieter, *Algorithm 599: Sampling from gamma and Poisson distributions*, ACM Transactions on Mathematical Software **9** (1983), 255–257.

[3] E. Anderson, Z. Bai, C. Bischof, J.W. Demmel, J.J. Dongarra, J. du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992.

[4] J.R. Bell, *Algorithm 334: Normal random deviates*, Communications of the ACM **11** (1968), 498.

[5] R.F. Boisvert, S.E. Howe, and D.K. Kahaner, *GAMS: A framework for the management of scientific software*, ACM Transactions on Mathematical Software (1985), 313–355.

[6] G.E.P. Box and M.E. Muller, *A note on the generation of random normal deviates*, Annals of Mathematical Statistics **28** (1958), 610–611.

[7] R.P. Brent, *An algorithm with guaranteed convergence for finding a zero of a function*, The Computer Journal **14** (1971), 422–425.

[8] R.P. Brent, *Algorithms for minimization without derivatives*, Prentice Hall International, Inc., Englewood Cliffs, New Jersey, 1973.

[9] R.P. Brent, *Algorithm 488: A Gaussian pseudo-random number generator*, Communications of the ACM **17** (1974), 704–706.

[10] P. Deuflhard, *A Modified Newton Method for the Solution of Ill-Conditioned Systems of Nonlinear Equations with Applications to Multiple Shooting*,, Numerische Mathematik **22** (1974), 289–315.

[11] L. Devroye, *Non-uniform random variate generation*, Springer, New York, 1986.

[12] DIEHARD, *A battery of tests of randomness*, available by anonymous ftp from stat.fsu.edu directory /pup/diehard.

[13] J.J. Dongarra and E. Grosse, *Distribution of mathematical software via electronic mail*, Communications of the ACM **30** (1987), 403–407.

[14] G.S. Fishman, *Multiplicative congruental random number generators with modulus $2^\beta$: An exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$*, Mathematics of Computation **54** (1990), 331–344.

[15] G.E. Forsythe, *Von Neumann's comparison method for random sampling from the normal and other distributions*, Mathematics of Computation **26** (1972), 817–826.

[16] P. Hellekalek, *Good random number generators are (not so) easy to find*, Proceedings of the 2nd IMACS Symposium on Mathematical Modelling (Vienna), 199?, ?–??

[17] F. James, *RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Lüscher*, Computer Physics Communications **79** (1993), 111–114.

[18] D.E. Knuth, *The art of computer programming. Volume 2: Seminumerical algorithms*, 2nd ed., Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[19] P. L'Ecuyer, *Combined multiple-recursive random number generators*, Operations Research **44** (1996), 812–822.

[20] P. L'Ecuyer, *Maximally equidistributed combined Tausworthe generators*, Mathematics of Computation **65** (1996), 203–213.

[21] P. L'Ecuyer, *A random number generator based on combination of four LCG's*, Mathematics of Computers in Simulation **??** (1997), ???

[22] P. L'Ecuyer, *Random number generation*, In: Handbook on simulation (J. Banks, ed.), John Wiley & Sons, New York, (to appear, 1997).

[23] J.L. Leva, *A fast normal random number generator*, ACM Transactions on Mathematical Software **18** (1992), 449–453.

[24] J.L. Leva, *Algorithm 712: A normal random number generator*, ACM Transactions on Mathematical Software **18** (1992), 454–455.

[25] M. Lüscher, *A portable high-quality random number generator for lattice field theory simulations*, Computer Physics Communications **79** (1993), 100–110.

[26] G. Marsaglia (ed.), *A current view of random number generators*, Elsevier, Computer Science and Statistics: 16th Symposium on the Interface, 1985.

[27] G. Marsaglia and T.A. Bray, *A convenient method for generating normal variables*, SIAM Review **6** (1964), 260–264.

[28] G. Marsaglia and W.W. Tsang, *A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions*, SIAM Journal on Scientific and Statistical Computing **5** (1984), 349–359.

[29] G. Marsaglia and A. Zaman, *A new class of random number generators*, Annals of Applied Probability **1** (1991), 462–480.

[30] G. Marsaglia and A. Zaman, *Monkey tests for random number generators*, Computers and Mathematics with Applications **26** (1993), 1–10.

[31] N. Masuda and F. Zimmermann, *PRNGlib: A parallel random number generator library*, Technical Report TR-96-08, Swiss Center for Scientific Computing (CSCS-SCSC), Manno, Switzerland, 1996, URI: http://www.cscs.ch/Official/PubTR96.html.

[32] M. Matsumoto and Y. Kurita, *Twisted GFSR generators*, ACM Transactions on Modeling and Computer Simulation **2** (1992), 179–194.

[33] M. Matsumoto and Y. Kurita, *Twisted GFSR generators II*, ACM Transactions on Modeling and Computer Simulation **4** (1994), 254–266.

[34] M. Matsumoto and T. Nishimura, *A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation (1997, submitted).

[35] M. Mössner, J. Pfleiderer, N. Netzer, *AD* dept. Astro

[36] Numerical Algorithms Group (NAG), *NAG Fortran 90 library, Release 1*, NAG Ltd., Oxford, England, UK, 1994.

| generator | SGI 1 | | SGI 2 | | HP 1 | | HP 2 | | PC 1 | | PC 2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | time | ratio | time | ratio | time | ratio | time | ratio | time | ratio | time | ratio |
| uniform distributed random numbers | | | | | | | | | | | | |
| mt19937 | 0.801 | | 1.342 | | 0.789 | | 1.026 | | 1.155 | | 0.975 | |
| LAPACK | 1.435 | | 2.318 | | 0.926 | | 1.480 | | 1.658 | | 1.775 | |
| NAG | | | 2.502 | | | | | | | | | |
| normal distributed random numbers | | | | | | | | | | | | |
| Box-Muller | 3.47 | 4.33 | 5.26 | 3.91 | 1.98 | 2.50 | 2.61 | 2.54 | 2.85 | 2.46 | 2.76 | 2.83 |
| polar method | 3.18 | 3.97 | 4.20 | 3.12 | 2.06 | 2.61 | 2.69 | 2.62 | 2.65 | 2.29 | 3.71 | 3.80 |
| sum of 12 uniforms | 6.93 | 8.65 | 10.86 | 8.09 | 7.84 | 9.93 | 10.28 | 10.01 | 11.73 | 10.15 | 10.12 | 10.37 |
| Brent | 2.11 | 2.63 | 3.25 | 2.42 | 1.87 | 2.37 | 2.47 | 2.40 | 2.90 | 2.51 | 3.08 | 3.15 |
| Ahrens-Dieter | 1.81 | 2.25 | 2.78 | 2.07 | 1.47 | 1.86 | 1.94 | 1.89 | 2.34 | 2.02 | 2.65 | 2.71 |
| Leva | 3.35 | 4.18 | 4.24 | 3.15 | 2.77 | 3.51 | 3.64 | 3,54 | 3.48 | 3.01 | 4.94 | 5.06 |
| Marsaglia-Bray | 2.52 | 3.14 | 4.02 | 2.99 | 2.32 | 2.94 | 3.02 | 2.94 | 3.49 | 3.02 | 3.42 | 3.50 |
| Marsaglia Tsang (64 el.t.) | 1.25 | 1.56 | 1.96 | 1.46 | 1.14 | 1.44 | 1.50 | 1.46 | 1.48 | 1.28 | 1.52 | 1.55 |
| Marsaglia Tsang (256 el.t.) | 1.21 | 1.51 | 1.91 | 1.42 | 1.08 | 1.36 | 1.42 | 1.38 | 1.40 | 1.21 | 1.43 | 1.46 |
| Mössner I (lin. sq.) | 1.27 | 1.58 | 2.02 | 1.50 | 1.10 | 1.39 | 1.44 | 1.40 | 1.63 | 1.41 | 1.74 | 1.78 |
| Mössner I (quad. sq.) | 1.17 | 1.46 | 1.81 | 1.34 | 1.05 | 1.33 | 1.37 | 1.33 | 1.56 | 1.35 | 1.59 | 1.63 |
| Mössner II (64 el.t.) | 1.17 | 1.46 | 1.84 | 1.37 | 1.00 | 1.26 | 1.32 | 1.28 | 1.53 | 1.32 | 1.48 | 1.51 |
| Mössner II (256 el.t.) | 1.14 | 1.42 | 1.80 | 1.34 | 0.94 | 1.19 | 1.25 | 1.21 | 1.53 | 1.32 | 1.44 | 1.47 |
| LAPACK | 6.89 | 4.80 | 10.84 | 4.67 | 3.83 | 4.13 | 5.00 | 3.37 | 5.70 | 3.43 | 6.10 | 3.43 |
| NAG | | | 7.67 | 3.06 | | | | | | | | |
| exponential distributed random numbers | | | | | | | | | | | | |
| inversion | 2.03 | 2.53 | 3.20 | 2.38 | 1.65 | 2.09 | 2.17 | 2.11 | 2.10 | 1.81 | 2.93 | 3.00 |
| uniform spaces | 1.68 | 2.09 | 2.60 | 1.93 | 1.47 | 1.86 | 1.88 | 1.83 | 2.08 | 1.80 | 2.12 | 2.17 |
| Ahrens-Dieter | 1.82 | 2.27 | 2.79 | 2.07 | 1.69 | 2.14 | 2.22 | 2.16 | 2.49 | 2.15 | 2.44 | 2.50 |
| Marsaglia Tsang (64 el.t.) | 1.15 | 1.43 | 1.83 | 1.36 | 1.13 | 1.43 | 1.49 | 1,45 | 1.48 | 1.28 | 1.43 | 1.46 |
| Marsaglia Tsang (256 el.t.) | 1.11 | 1.38 | 1.71 | 1.27 | 1.07 | 1.35 | 1.42 | 1.38 | 1.41 | 1.22 | 1.35 | 1.38 |
| Mössner II (64 el.t.) | 1.08 | 1.34 | 1.77 | 1.31 | 0.97 | 1.22 | 1.28 | 1.24 | 1.40 | 1.21 | 1.32 | 1.35 |
| Mössner II (256 el.t.) | 1.09 | 1.36 | 1.77 | 1.31 | 0.93 | 1.17 | 1.22 | 1.18 | 1.45 | 1.25 | 1.31 | 1.34 |
| NAG | | | 4.76 | 1.90 | | | | | | | | |

Table 1: Timing results for normal and exponential RNGs

Table 2: Timing results for normal and exponential RNGs

| generator | SGI 1 | | SGI 2 | | HP 1 | | HP 2 | | PC 1 | | PC 2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | time | ratio | time | ratio | time | ratio | time | ratio | time | ratio | time | ratio |
| uniform distributed random numbers | | | | | | | | | | | | |
| mt19937 | 0.506 | | 0.821 | | 0.543 | | 0.702 | | 1x155 | 1x00 | 0x975 | 1x00 |
| normal distributed random numbers | | | | | | | | | | | | |
| Box-Muller | 2.53 | 5.00 | 3.81 | 4.64 | 1.70 | 3.13 | 2.24 | 3.19 | 2x85 | 2x46 | 2x76 | 2x83 |
| polar method | 2.28 | 4.50 | 2.82 | 3.43 | 1.73 | 3.18 | 2.27 | 3.23 | 2x65 | 2x29 | 3x71 | 3x80 |
| sum of 12 uniforms | 3.68 | 7.27 | 5.77 | 7.02 | 4.17 | 7.67 | 5.48 | 7.80 | 11x73 | 10x15 | 10x12 | 10x37 |
| Brent | 1.55 | 3.06 | 2.36 | 2.87 | 1.36 | 2.50 | 1.79 | 2.54 | 2x90 | 2x51 | 3x08 | 3x15 |
| Ahrens-Dieter | 1.40 | 2.76 | 2.13 | 2.59 | 1.14 | 2.09 | 1.50 | 2.13 | 2x34 | 2x02 | 2x65 | 2x71 |
| Leva | 2.26 | 4.46 | 2.55 | 3.10 | 2.06 | 3.79 | 2.71 | 3,86 | 3x48 | 3x01 | 4x94 | 5x06 |
| Marsaglia-Bray | 1.51 | 2.98 | 2.29 | 2.78 | 1.35 | 2.48 | 1.77 | 2.52 | 3x49 | 3x02 | 3x42 | 3x50 |
| Marsaglia Tsang (64 el.t.) | 0.93 | 1.83 | 1.32 | 1.60 | 0.83 | 1.52 | 1.10 | 1.56 | 1x48 | 1x28 | 1x52 | 1x55 |
| Marsaglia Tsang (256 el.t.) | 0.83 | 1.64 | 1.21 | 1.47 | 0.78 | 1.43 | 1.03 | 1.46 | 1x40 | 1x21 | 1x43 | 1x46 |
| Mössner I (lin. sq.) | 0.91 | 1.79 | 1.36 | 1.65 | 0.76 | 1.39 | 1.00 | 1.42 | 1x63 | 1x41 | 1x74 | 1x78 |
| Mössner I (quad. sq.) | 0.82 | 1.62 | 1.26 | 1.53 | 0x70 | 1.28 | 0.92 | 1.31 | 1x56 | 1x35 | 1x59 | 1x63 |
| Mössner II (64 el.t.) | 0.83 | 1.64 | 1.23 | 1.49 | 0.73 | 1.34 | 0.93 | 1.32 | 1x53 | 1x32 | 1x48 | 1x51 |
| Mössner II (256 el.t.) | 0.75 | 1.48 | 1.21 | 1.47 | 0.68 | 1.25 | 0.90 | 1.28 | 1x53 | 1x32 | 1x44 | 1x47 |
| exponential distributed random numbers | | | | | | | | | | | | |
| inversion | 1.66 | 3.28 | 2.53 | 3.08 | 1.21 | 2.22 | 1.60 | 2.27 | 2x10 | 1x81 | 2x93 | 3x00 |
| uniform spaces | 1.17 | 2.31 | 1.82 | 2.21 | 0.95 | 1.74 | 1.25 | 1.78 | 2x08 | 1x80 | 2x12 | 2x17 |
| Ahrens-Dieter | 1.29 | 2.54 | 1.95 | 2.37 | 1.20 | 2.20 | 1.58 | 2.25 | 2x49 | 2x15 | 2x44 | 2x50 |
| Marsaglia Tsang (64 el.t.) | 0.91 | 1.79 | 1.35 | 1.64 | 0.80 | 1.47 | 1.06 | 1,50 | 1x48 | 1x28 | 1x43 | 1x46 |
| Marsaglia Tsang (256 el.t.) | 0.83 | 1.64 | 1.21 | 1.47 | 0.76 | 1.39 | 1.00 | 1.42 | 1x41 | 1x22 | 1x35 | 1x38 |
| Mössner II (64 el.t.) | 0.76 | 1.50 | 1.18 | 1.43 | 0.68 | 1.25 | 0.89 | 1.26 | 1x40 | 1x21 | 1x32 | 1x35 |
| Mössner II (256 el.t.) | 0.73 | 1.44 | 1.12 | 1.36 | 0x65 | 1x19 | 0.85 | 1.21 | 1x45 | 1x25 | 1x31 | 1x34 |