Internal Note SUF–PFY/88–01
Stockholm, October 6, 1988

PROPOSAL

FOR A NEW RANDOM NUMBER GENERATOR FOR THE DELPHI SIMULATION PROGRAM

by
Christian Walck
Department of Physics
University of Stockholm

1 INTRODUCTION

The period of the basic random number generator **RNDM** used in the DELPHI simulation program DELSIM [1] is much too short to be ideal. We propose here the use of a new generator based on the CERN library routines **RNDM2** and **NORRAN** which has a sufficiently long period and which also gives efficient generation of standard normal random numbers. This work was induced by a suggestion from H.Forsbach of Wuppertal. In fact, some of the work had already been done, mainly in the process of installing the CERN libraries on the new Alliant FX/80 computer in Stockholm but also in connection with work done by the author in the area of random number generation [2]. Below we mainly describe work done on computers in Stockholm and at CERN. Per Steinar Iversen of Bergen was very helpful and efficient in preparing a version of the routines developed for a NORD 570/CX computer.

We have no intention to cover the whole area of random number generation in this report but, due to the interest within and outside the collaboration, we felt it necessary to describe in some detail the theory behind the subject.

This document is prepared using the LaTeX program and figures are made using the HLaTeX program which has been developed by colleagues in Copenhagen [3].

2 UNIFORM RANDOM NUMBERS

In the DELPHI simulation program the basic random number generator used in order to obtain uniform pseudorandom numbers between zero and one is the CERN library routine **RNDM** [4]. Special effort has been made in order to use the random number sequence of **RNDM** throughout the program by interfacing other random number routines wherever necessary. Two separate sequences are kept in parallel, one for the generation part and one for the simulation part. This is done in order to be able to generate the same sequence of events regardless of the specific simulation requested.

The CERN library routine **RNDM** (entry V104) is based on the multiplicative congruential method i.e. subsequent seeds are obtained by

$$x_{n+1} = \lambda x_n \bmod 2^\beta$$

i.e. multiplying the previous seed by the multiplier λ and extracting the β least significant bits in the binary representation[1]. Such generators has a maximum period of $P = 2^{\beta-2}$ (for $\beta \geq 3$) which is obtained for multipliers $\lambda = 3$ or 5 (mod 8) and all odd start values [5]. The sequences generated are permutations of the following values:

λ (mod 8)	Start value	values in sequence	
3	1,3,9,11 (mod 16)	$8\nu+1, 8\nu+3$	$\nu = 0,1,\ldots,2^{\beta-3}-1$
3	5,7,13,15 (mod 16)	$8\nu+5, 8\nu+7$	$\nu = 0,1,\ldots,2^{\beta-3}-1$
5	1 (mod 4)	$4\nu+1$	$\nu = 0,1,\ldots,2^{\beta-2}-1$
5	3 (mod 4)	$4\nu+3$	$\nu = 0,1,\ldots,2^{\beta-2}-1$

For **RNDM** $\beta = 32$ and $\lambda = 690069$ (on VAX and NORD) or $\lambda = 69069$ (on IBM and Apollo) and by the default the start seed is 12345 corresponding to the third row in the table above. The random number sequence generated by **RNDM** thus has a period of $2^{30} \approx 1.1 \cdot 10^9$ random numbers. This is easily verified on a VAX computer but according to the CERN library manual IBM computers give only half of this [4]. The sequence of 32-bit integers generated is converted into real numbers between zero and one essentially by extracting 24 bits and multiplying this number by 2^{-24}. The multiplication is on VAX computers made by adding a shift into the exponent which has the virtue that even if the 24 bits are all zero the result is non-zero. This

[1] A very convenient choice on 32-bit computers is $\beta = 32$ where the modulus 2^{32} is obtained by simply ignoring the integer overflow in the multiplication.

is useful e.g. when generating random numbers from exponential distributions involving the logarithm of a uniform random number. On a VAX computer the minimum and maximum value returned by `RNDM` is $2^{-25} \approx 0.000000030$ and $1 - 2^{-24} \approx 0.999999940$[2], respectively.

In fact one even sometimes defines the "effective period" as the smallest integer t such that $\lambda^t - 1$ is a multiple of the period P [6]. This implies that the sequence of seeds consists of blocks of t numbers with translates of the preceding block. For `RNDM` the effective period defined in this way becomes $2^{28} = 268435456$ and the `RNDM` sequence consists of four blocks each being a translation of the previous by 0.75 (modulus 1.0).

We note that the multiplier chosen for `RNDM` is quite small. Although `RNDM` passes many quality tests this implies that when we have obtained a value close to zero from `RNDM` the next value is quite small as well. The effect is not large enough to be seen in correlation studies but might be unwanted. In a paper by Ahrens, Dieter and Grube [7] it is argued that a better choice for the multiplier of a multiplicative congruential generator is $2^{\beta-2}\xi$ where ξ is the golden section number ($\xi = \frac{\sqrt{5}-1}{2}$). This would imply a multiplicator of $\lambda = 663608941$ for $\beta = 32$. However, we are aware that extensive studies for optimal multipliers have been performed and that surely the one used in `RNDM` are not chosen randomly.

Although not used here we also want to warn for the usage of multiplicative congruential generators corresponding to the first two rows in the previous table. Such generators have very bad behaviour by e.g. having too many completely ascending or descending sequences of random numbers.

The short period of the intrinsic random number sequence of `RNDM` is worrying. Especially since it is never recommendable to exhaust more than a few percent of the period of any such generator to assure good randomness (if we were to exhaust a full period a completely flat distribution would result). The full simulation of a $e^+e^- \to Z^0/\gamma \to c\bar{c}$-event with the DELPHI simulation program consumes on average about one million uniform random numbers and thus the entire `RNDM`-sequence would only suffice for one thousand events.

A better choice than `RNDM` as concerns the period of the sequence would be another CERN library routine, namely `RNDM2` (entry V107). This routine has been around for a long time and in the CERN library manual it is recommended to replace `RNDM` for applications requiring many random numbers. This routine combines two independent generators each with its own sequence (the routine thus has two separate seeds).

The first is a multiplicative congruential generator very similar to the one of `RNDM` (the only difference being that $\lambda = 69069$) having a period of $2^{30} \approx 1.1 \cdot 10^9$ random numbers.

The second is a shift register generator having a period of 4.292.868.097 random numbers. This is close to 2^{32}, or more precisely $2^{32} - 2^{21} - 2^{11} + 1$, and thus almost all possible bit patterns are included in its sequence. For a more general description of shift register generators see [6]. In our specific case the shift register generator updates the seed `SRGN` by

```
R0   = IEOR ( SRGN, ISHFT(SRGN,-15) )
SRGN = IEOR (   R0, ISHFT(R0,17)    )
```

where `IEOR` is the bit by bit exclusive-or operation between the 32 bits of its arguments and `ISHFT(R,N)` shifts the 32 bits in `R` left `N` bits (or right `-N` bits if `N` is negative) with zero filled from right (left). The default start seed is 1073. As for the multiplicative congruential generator one should not invent seeds but use seeds from previous generations if not using the default value. The risk to enter sequences with shorter periods than the default sequence are, however, small. Besides the default sequence which is entered with 1073 (or 1) three other sequences exist. Starting with 2581 enters a sequence with a period of $2097151 = 2^{21} - 1$, starting with 2816784 a sequence with a period of $2047 = 2^{11} - 1$ and starting with zero always give zero.

Finally a bit by bit exclusive-or between the two seeds are used. Since the two random number sequences are independent this implies a total period for `RNDM2` of $4.609.432.020.664.188.928 \approx 4.6 \cdot 10^{18}$ random numbers which is quite sufficient for our application. The conversion to real

[2] The largest number less than one representable in single precision on a VAX computer.

numbers between zero and one is done in a similar way as for **RNDM** and the minimum and maximum values returned by `RNDM2` are therefore the same. Note that although the period of the random sequence is $4.6 \cdot 10^{18}$ the number of possible outcomes of a call to `RNDM2` (and `RNDM`) on a VAX computer is $2^{24} \approx 1.7 \cdot 10^{7}$ due to the fact that only 24 bits are significant. On computers with less than 24 effective bits in the mantissa the number of possible outcomes are even fewer.

A good description of different uniform random number generators are given in [6]. The technique used in `RNDM2`, namely to exclusive-or the results of two different generators, are mentioned and as an example exactly the `RNDM2` generator are presented. The origin of this is from G.Marsaglia and collaborators at McGill University who during the sixties and seventies developed a package of fast random number generators ("Super-Duper"-package).

3 STANDARD NORMAL RANDOM NUMBERS

There are many methods available in order to generate random numbers from a normal distribution (see e.g. [2]). When generating random numbers from normal distributions the DELPHI simulation program now use the CERN library routine **RANNOR** (entry V100) based on the exact transformation method of Box and Muller [8]. This routine returns two standard normal random numbers, z_1 and z_2, on each call using the well-known transformation

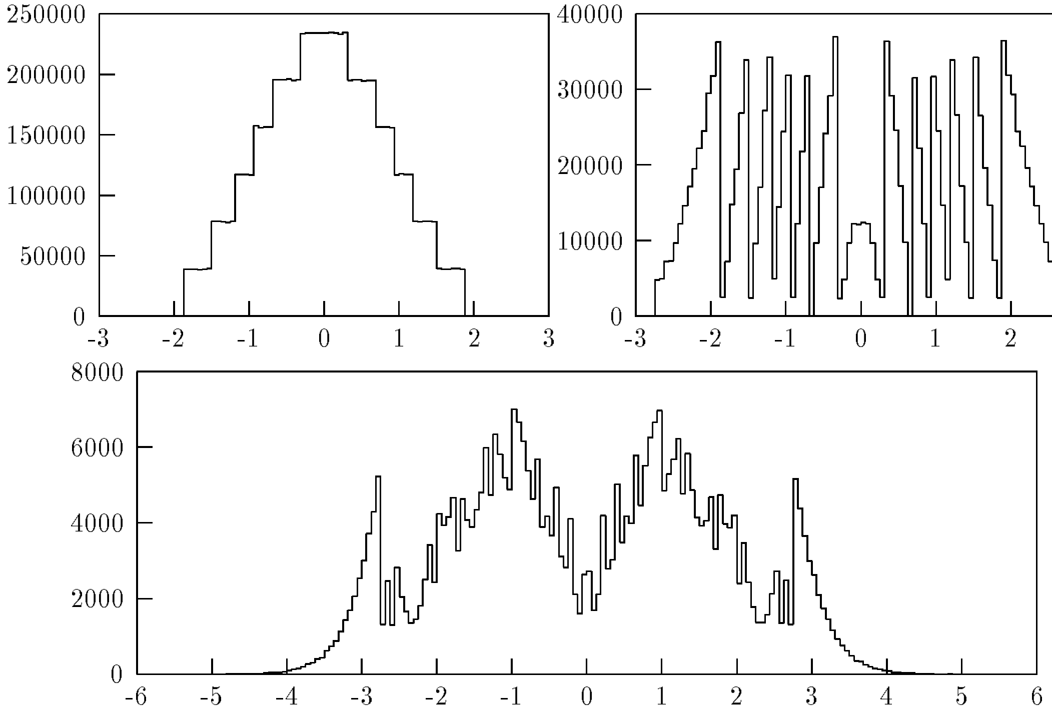
$$z_1 = \sqrt{-2\ln x}\cos 2\pi y \quad \text{and} \quad z_2 = \sqrt{-2\ln x}\sin 2\pi y$$

where x and y are two uniform random numbers between zero and one. One may easily speed this method up by using the polar method of Marsaglia [9] where the sine and cosine terms are effectively done by a simple accept-reject method.

On the CERN library exists a much faster generator called **NORRAN** (entry V101) based on algorithms developed by G.Marsaglia, K.Ananthanarayanan, N.Paul, J.H.Ahrens and U.Dieter [10, 11, 12]. The method used in **NORRAN** is almost a factor of two faster than any of the alternatives. However, this routine has not been used sofar in the DELPHI simulation program since its random number sequence is independent of that of **RNDM**. In fact, the intrinsic random number sequence in **NORRAN** is the same as the one used in `RNDM2` which opens the possibility to merge these two routines. For a description of the technique used in **NORRAN** see [12]. A brief description of **NORRAN** is as follows

- in 81.2 % of all cases random numbers are generated from an histogram inscribed under the normal curve,
- in 14.8 % of all cases triangular regions between the normal curve and the histogram are used and
- in the remaining 4.0 % of all cases random numbers are generated from the "teeth-function" left between the standard normal curve and the histogram plus triangular regions and from the tails using reject-accept techniques.

In the figures the three distributions corresponding to the three cases above are shown. These are empirical distributions obtained by flagging, in a special **NORRAN** version, which part of the code were used. In the first figure to the left we show the histogram part, in the second figure to the right the triangular parts and in the third figure the remaining region are shown. These three distributions add up to the standard normal distribution.



The technique is exact and only limited by the machine precision in giving random numbers following the standard normal distribution. In order to make the routine as fast as possible the 12 first bits of a seed are used to select which part of the code to be used. Thereafter the remaining 20 bits are used to make a random number with the result of a slight degradation in the accuracy.

4 A NEW RANDOM NUMBER GENERATOR

By combining `RNDM2` and `NORRAN` so that they use the same random number sequence we obtain a good generator with a sufficiently long period. The loss in time per call between `RNDM` and `RNDM2` are most probably compensated by the gain in using `NORRAN` as compared to `RANNOR` (especially taking into account that in many places in the simulation program the second random number returned by `RANNOR` is simply disregarded). This, however, depend on the frequency with which normal random numbers are used as compared to uniform random numbers.

The first obstacle is that `RNDM2` and `NORRAN` normally are written in assembler language. For VAX computers we have prepared a joint routine simply by merging the assembler code of the two individual routines. In fact `NORRAN` contains an entry `UNI` which is identical to `RNDM2`. With the aid of the functions `IAND`, `IEOR` and `ISHFT` it was possible to prepare an identical routine in `FORTRAN`. Such routines written in `FORTRAN` had already been prepared for `RNDM2` and `NORRAN` separately and it was straightforward to merge these into one routine. The `FORTRAN` version turned out to be only marginally slower than the assembler version (see section on timing).

Please note that the functions `IAND`, `IEOR` and `ISHFT` are not intrinsic or inline functions on all computers. In such cases there exists on the CERN library a BITPAK package (entry M441) with these routines. However, this would make the routine much slower and we therefore try to find inline alternatives for all computers if possible.

The bit manipulations involved, although in principle possible to solve for any computer, turns out to be an obstacle. To achieve such versions the floating point representation of each computer (number of bits in exponent and mantissa and their positions) has to be specified. In addition the trick avoiding the zero (adding a shift into the exponent) does not work properly on all computers. In view of this we tried the very direct approach of simply multiplying the extracted 24-bit integer with 2^{-24}. This, however, kills the trick avoiding the zero so we have to add a test for this as well. In fact, even so this routine became faster than the bit-manipulating version.

The next obstacle encountered was that in `NORRAN` and also in our joint version the main routine calls a subroutine (`RNORTH`) which in turns calls entry points in the main routine to get uniform random numbers from the same sequence. This recursive type of calls work nicely on e.g. VAX and Alliant computers but turned out not to work on NORD computers. In view of this we had to introduce a `BLOCK DATA` and a `COMMON`-block which is annoying if one wants to place the routine on a general library. For the DELPHI simulation program it would not be important to include the `BLOCK DATA` since we always handle the initialization of the seeds from the main program.

In the final routine we traded beauty for efficiency and replaced all the calls from the `RNORTH`-code with explicit generation of the required random numbers. In this way this part of the code gets much longer but we may then write one routine containing all entry points avoiding recursive calls as well as `COMMON`-blocks.

The final routine has as its entry point for uniform random numbers `RNDUNI` and for standard normal random numbers `RNDNRM`. Note that `RNDNRM` in our final version is a function subprogram while `NORRAN` was a subroutine subprogram. Some of the tests avoiding zeros in the code of `RNDNRM` seem unnecessary and only slows the routine down. However, they have been kept in order to make the results of the routine *identical* to that of `NORRAN` on VAX computers. The final `FORTRAN` routine is listed in appendix A. The routine `RNDUNI` has the following entry points

- `R=RNDUNI(DUMMY)` gives pseudorandom numbers from a uniform distribution between zero and one (end-points excluded). `RNDUNI` is identical to `RNDM2` and the entry point `UNI` in `NORRAN`.
- `R=RNDVNI(DUMMY)` gives uniform pseudorandom numbers from a uniform distribution between -1 and +1. `RNDVNI` is identical to the entry `VNI` in `NORRAN`. In our final version this entry is not needed. It is however kept for historical reasons and may be used for somewhat more efficient generation when pseudorandom numbers between -1 and 1 are required.
- `IR=IRNDUN(DUMMY)` gives discrete (integer) pseudorandom numbers between 0 and $2^{31}-1$. `IRNDUN` is identical to `IRNDM2`.
- `R=RNDNRM(DUMMY)` gives a standard normal pseudorandom number. `R=RNDNRM(DUMMY)` is identical to `CALL NORRAN(R)`.
- `CALL RNDIN(ISEED1,ISEED2)` sets the seeds of the random sequence. `RNDIN` has the same function as `RD2IN` and `NORRIN` has for `RNDM2` and `NORRAN`. Preferably use seeds obtained by a previous call to `RNDOUT` when starting a new run.
- `CALL RNDOUT(ISEED1,ISEED2)` accesses the seeds of the random sequence. `RNDOUT` has the same function as `RD2OUT` and `NORRUT` has for `RNDM2` and `NORRAN`.

The multiplicative congruential generator in the routines works by ignoring the integer overflow. On some computers this may have to be handled with by compiling the routines with specific switches (on a VAX computer the option `/CHECK=NOOVERFLOW` is required).

It is to be noted that the integer sequences of the two seeds are identical on every machine and thus start seeds for different simulation runs can be exchanged between different installations. However, the way in which this integer sequence is turned into pseudorandom numbers between zero and one is machine dependent. In fact the present way of extracting 24 bits is slightly VAXish and is chosen because of the 24 significant bits (one hidden) in the mantissa in the single precision floating point representation on a VAX computer. This is probably done in order to get a uniform "lattice" of discrete points (the exception being the zero which has been deliberately shifted a half step to 0.5). If we, like in the two parameter version of the VAX function RAN, would use 31 bits (only masking off the sign bit) and then multiply this number by 2^{-31} we would get steps of 2^{-24} between 0.5 and 1.0, of 2^{-25} between 0.25 and 0.5 etc. If it is considered important we can easily adjust the routine to take into account the actual number of bits in the mantissa for each computer.

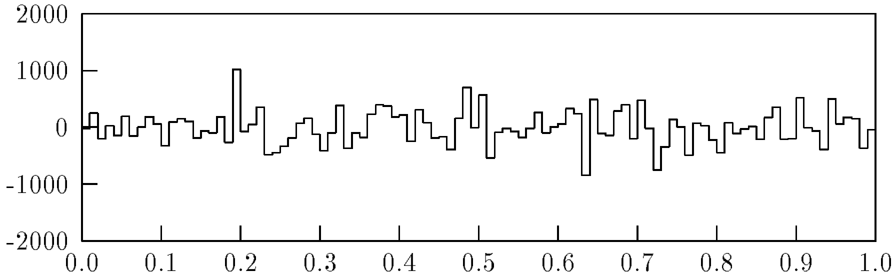
As a matter of fact we are close to a completely machine independent generator (similar to the CERN library routine RN32) for uniform random numbers. However, due to the different representation of floating point numbers on different computers round-off errors will soon cause one of the many accept-reject (or hit-miss) and composition-rejection loops[3] used in our program (and in RNDNRM) to get out of phase.

5 QUALITY

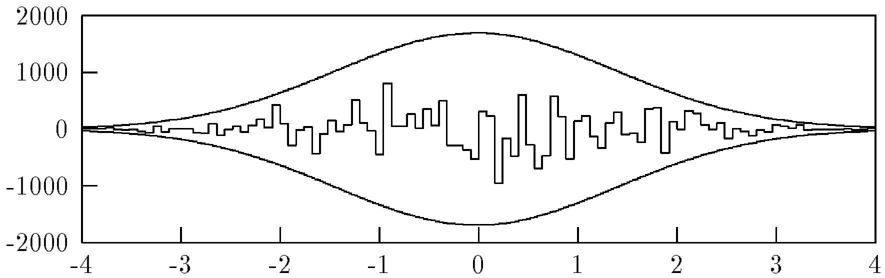
For general discussions of the quality of different generators of uniform random numbers we refer the reader to [5, 6]. The origin of the generator in RNDM2 used in our new routine is from G.Marsaglia and collaborators at McGill University which seems to guarantee that the generator has been thoroughly tested. However, we have performed some tests which we present in this section.

5.1 Frequency

First we compare frequencies in 100 bins found by generating 10^7 random numbers with RNDUNI and RNDNRM, respectively. In the first figure we show the difference between a sample of 10^7 uniform random numbers generated by RNDUNI and the ideal distribution. In this case the expected number of events in each bin is 10^5 corresponding to a standard error of $\sqrt{10^5} \approx 316$.



In the next figure we show the corresponding plot for standard normal random numbers. In this case the standard errors vary along the scale and we show by the two curves the upper and lower three standard deviation levels.



As can be seen in both cases the generators give satisfactorily results. Calculating χ^2-probabilities we find $\chi^2/df = 90/99$ and $\chi^2/df = 75/99$ corresponding to probability contents of 76 % and 97 % in the uniform and standard normal case, respectively.

[3]The acception-rejection (hit-miss) and the composition-rejection methods are the two most common techniques to achieve random numbers from arbitrary distributions (see e.g. [2]).

5.2 Moments

The moments of the distributions obtained by generating a sample of 10^7 random numbers are given in the tables below where we have calculated the algebraic and central moments up to eighth order (up to sixteenth order calculated in order to estimate the errors). There are excellent agreement with the ideal values.

Algebraic moments

	RNDUNI		RNDNRM	
order	sample	theory	sample	theory
1	0.49998 ± 0.00009	1/2	0.0000 ± 0.0003	0
2	0.33331 ± 0.00009	1/3	1.0003 ± 0.0004	1
3	0.24998 ± 0.00009	1/4	0.0009 ± 0.0012	0
4	0.19998 ± 0.00008	1/5	3.000 ± 0.003	3
5	0.16666 ± 0.00008	1/6	0.009 ± 0.010	0
6	0.14285 ± 0.00008	1/7	14.99 ± 0.03	15
7	0.12500 ± 0.00007	1/8	0.12 ± 0.12	0
8	0.11111 ± 0.00007	1/9	$104.8 \pm .5$	105

Central moments

	RNDUNI		RNDNRM	
order	sample	theory	sample	theory
2	0.08333 ± 0.00002	1/12	1.0003 ± 0.0004	1
3	0.000004 ± 0.000011	0	0.0009 ± 0.0008	0
4	0.012501 ± 0.000005	1/80	3.000 ± 0.003	3
5	0.000001 ± 0.000003	0	0.009 ± 0.008	0
6	0.002232 ± 0.000001	1/448	14.99 ± 0.03	15
7	0.000000 ± 0.000001	0	0.12 ± 0.11	0
8	0.000434 ± 0.0000003	1/2304	104.8 ± 0.5	105

5.3 Serial correlation

The serial correlation, or autocorrelation coefficients, expresses the correlation between random numbers in the sequence k step apart i.e.

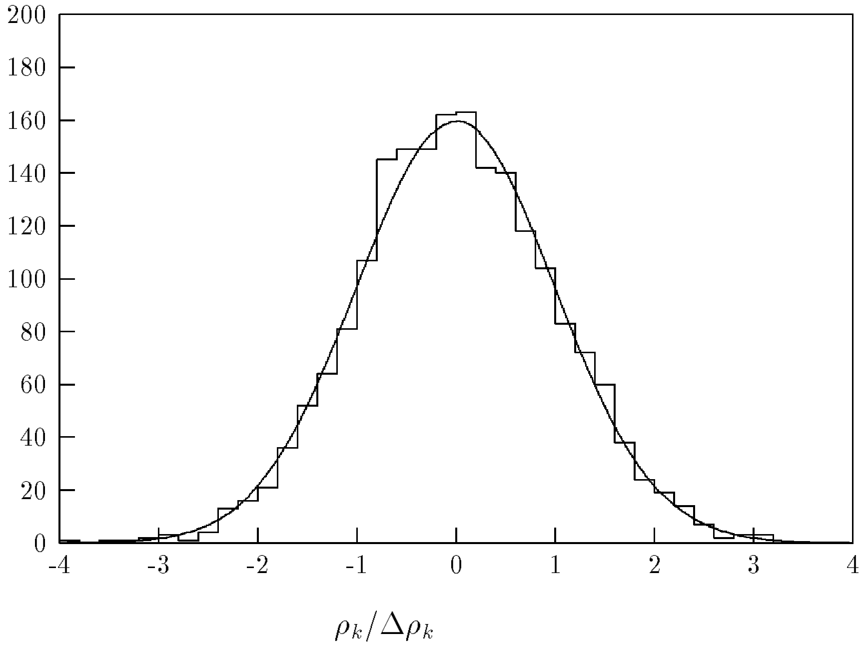
$$\rho_k = \frac{Cov(x_i, x_{i+k})}{V(x_i)}$$

where x_j is the j:th random number in the sequence and Cov and V are the covariance and variance operators, respectively. Sampling N random numbers the standard error of ρ_k is given by $1/\sqrt{N}$ for large enough N.

Generating 10^7 random numbers with `RNDUNI`, corresponding to a standard error of 0.0003 in ρ_k, we found

ρ_1=0.0002	ρ_8=0.0000	ρ_{15}=0.0001	ρ_{22}=-0.0002	ρ_{29}=-0.0000
ρ_2=0.0002	ρ_9=-0.0001	ρ_{16}=-0.0004	ρ_{23}=-0.0001	ρ_{30}=0.0005
ρ_3=0.0000	ρ_{10}=-0.0001	ρ_{17}=0.0001	ρ_{24}=0.0005	ρ_{31}=-0.0002
ρ_4=0.0001	ρ_{11}=0.0001	ρ_{18}=0.0003	ρ_{25}=-0.0003	ρ_{32}=-0.0001
ρ_5=-0.0001	ρ_{12}=0.0005	ρ_{19}=0.0003	ρ_{26}=0.0000	ρ_{33}=-0.0001
ρ_6=-0.0004	ρ_{13}=0.0003	ρ_{20}=0.0004	ρ_{27}=-0.0003	ρ_{34}=0.0003
ρ_7=0.0005	ρ_{14}=-0.0003	ρ_{21}=0.0003	ρ_{28}=0.0001	ρ_{35}=-0.0002

This is in perfect agreement with the ideal expectation of zero serial correlations. Indeed, determining ρ_1 to ρ_{2000} for a sample of one million random numbers from `RNDUNI` and dividing these by the expected standard error ($\Delta\rho_k = 0.001$) we get excellent agreement with the standard normal distribution which is shown in the figure below.



5.4 n-tuples

In this test we generate samples from `RNDUNI` or `RNDM2` of n-tuples, non-overlapping sequences of n uniform pseudorandom numbers, and classify these according to their ordering in terms of ascending/descending. As an example, for $n = 2$ we have two possible outcomes either the sequence is ascending (i.e. the second random number in the doublet is larger than the first) or descending. Generally we have 2^{n-1} possible orderings for a sequence with n random numbers. Theoretical frequencies for these outcomes can be calculated and compared to the generated samples. In the table below we give the χ^2, degrees of freedom (*df*) and χ^2-probability for test runs with samples of 10^7 n-tuples for n values ranging from 2 to 6. We show the results for the `RNDUNI` generator as well as for the multiplicative congruential generator (MCG) and the shift register generator (SRG) separately. The results for the `RNDUNI` generator and the multiplicative congruential generator are good while the shift register generator is very bad.

		`RNDUNI`		MCG		SRG	
n	*df*	χ^2	%	χ^2	%	χ^2	%
2	1	0.1	76.5	5.6	1.8	4.6	3.3
3	3	1.6	65.2	4.5	21.3	1300	0.0
4	7	10.7	15.0	12.6	8.3	16800	0.0
5	15	12.4	65.3	16.6	34.1	12000	0.0
6	31	27.4	65.4	33.5	34.5	228000	0.0

As an example of a specific case let us take quadruples and denote with D and U if the sequence is going down or up in each step. In the table below we show the result and give the number of outcomes in the sample of 10^7 quadruples, the theoretical expectation, the difference between these two numbers, the number of standard deviations (s.d.) this difference amounts to and finally the χ^2-contribution for each class ($\Delta\chi^2$=(s.d.)^2).

Quadruples from `RNDUNI`

class	sample	theory	diff	s.d.	$\Delta\chi^2$
DDD	416226	416667	-441	-0.68	0.47
DDU	1248579	1250000	-1421	-1.27	1.62
DUD	2085349	2083333	2016	1.40	1.95
DUU	1250961	1250000	961	0.86	0.74
UDD	1249846	1250000	-154	-0.14	0.02
UDU	2080502	2083333	-2831	-1.96	3.85
UUD	1251350	1250000	1350	1.21	1.46
UUU	417187	416667	520	0.81	0.65

6 TIMING

When constructing a pseudorandom number routine it is of interest to speed it up as much as possible. However, we note that in the present DELPHI simulation of $c\bar{c}$-events only slightly above one percent of the total time is spent in `RNDM` (the actual time spent in generating random numbers from specific distributions including the code at higher levels is longer). It might thus seem that timing is of minor importance for us. However, in our simulation program few routines takes more than a few percent of the total time and thus savings at this level are not unimportant.

In this section we give the approximate time per call in microseconds for calls to `RNDM`, `RNDM2` and `NORRAN` as well as for the new generator written in assembler and `FORTRAN` for some different computers in Stockholm and elsewhere. In the table below we give the computers on which the routines has been tested and the operative systems and `FORTRAN` versions used on these.

Computers

Computer system	Location	Operating system	`FORTRAN` version
VAX750	Stockholm	VMS V4.5	VAX `FORTRAN` V4.3-145
MicroVAX II	Stockholm	MicroVMS V4.4	VAX `FORTRAN` V4.3-145
VAX780	Stockholm	VMS V4.5	VAX `FORTRAN` V4.5-219
VAX8530	Stockholm	VMS V4.5	VAX `FORTRAN` V4.5-219
VAX8650	CERN	VMS V4.6	VAX `FORTRAN` V4.7-271
VAX8800	CERN	VMS V4.6	VAX `FORTRAN` V4.7-271
Alliant FX/80	Stockholm	Concentrix 3.0	FX/Fortran 3.1.33
NORD 570/CX	Bergen	SINTRAN III vsn K	`FORTRAN`-500-K02

The different routines tested were

- The original CERN routines `RNDM2` and `NORRAN` as well as `RNDM` for the uniform distribution.
- An assembler version obtained by merging the CERN library routines `RNDM2` and `NORRAN` also including a copy of the routine `RNORTH` called by `NORRAN` (VAX version only).
- A `FORTRAN` version obtained by direct translation of the assembler versions of `RNDM2` and `NORRAN` using the `FORTRAN` functions `IAND`, `IEOR` and `ISHFT` (VAX version only).
- A `FORTRAN` version using multiplications (plus tests for zero) and not bit-manipulations thus becoming more machine-independent (not working on NORD computers). This version includes a slightly modified version of `RNORTH` called `RNDTOO`.
- A `FORTRAN` version using a `COMMON`-block and dividing `RNDUNI` and `RNDNRM` into two routines (but instead merging the `RNORTH` code into `RNDNRM`). In this version `RNDNRM` was made a function subprogram and not a subroutine as in the previous versions and in `NORRAN`.

- A `FORTRAN` version where we have avoided the `COMMON`-block as well as the original recursive calls by making the required generation in the old `RNORTH`-code directly in this code. This makes this part of the code long and less elegant but is worth it in order to become as machine independent and efficient as possible.

The timing results in μs/call for the generation of uniform and standard normal random numbers for the different routines are given in the tables below.

Uniform random numbers

Computer	`RNDM`	`RNDM2` assembler original	`RNDUNI` assembler	`RNDUNI` `FORTRAN` bit-manip	`RNDUNI` `FORTRAN` multiply	`RNDUNI` `FORTRAN` `COMMON`	`RNDUNI` `FORTRAN` final
VAX 750	46	72	74	87	76	72	80
MicroVAX II	35	51	53	61	54	54	56
VAX 780	29	43	43	56	47	45	
NORD 570/CX	9.4	25.2					17.9
Alliant FX/80	14.1	16.8			17.0	16.5	17.2
VAX 8530	8.2	13.8	14.0	15.0	13.0	12.0	12.6
VAX 8650	6.1	9.6	9.9	10.7	9.1	8.7	9.3
VAX 8800	6.0	9.6	9.3	10.6	9.2	9.4	9.3

Standard normal random numbers

Computer	`NORRAN` original	`RNDNRM` assembler	`RNDNRM` `FORTRAN` bit-manip	`RNDNRM` `FORTRAN` multiply	`RNDNRM` `FORTRAN` `COMMON`	`RNDNRM` `FORTRAN` final
VAX 750	111	113	125	108	113	113
MicroVAX II	81	81	89	80	81	78
VAX 780	67	67	80	69	69	
NORD 570/CX						25.7
Alliant FX/80	23.5			23.9	15.6	22.1
VAX 8530	20.4	20.7	21.0	17.9	18.0	19.0
VAX 8650	14.7	14.6	15.0	12.9	13.8	12.5
VAX 8800	13.3	14.0	14.5	12.1	13.2	13.9

These times are obtained by timing a loop with calls to the routine under investigation afterwards subtracting the time for an empty loop[4]. They are reproducible within a few percent from run to run for a certain job. However, we have noted that e.g. the time per call for `RNDM2` is affected by the way the routine is loaded. In another job where `RNDM2` was located across a page boundary in virtual memory the 43 μs/call on a VAX 780 suddenly became 50 μs/call. We have not investigated systematically for such effects but all times above originates from the same job ran on all machines.

The figures for the NORD 570/CX, having a NORD 110 computer as front-end machine, were obtained in Bergen by P.S.Iversen who also had prepared the versions of `RNDM`, `RNDM2` and `NORRAN` for this machine.

For the Alliant figures for `RNDM`, `RNDM2` and `NORRAN` are for routines written in `FORTRAN` in the same spirit as our new routines. These routines ran on a VAX computer gives identical random number sequences as the CERN originals. Note that for the version with the `COMMON`-block the compiler manage to optimize the code such that calls to `RNDNRM` are even faster than calls to `RNDUNI`.

[4]In fact an almost empty loop were used since some modern compilers may be too smart to be fooled to perform a completely empty loop.

We have checked that the routine also compiles on Apollos and IBMs and on the latter we have checked that it works properly. For the moment, however, we have no detailed timing for these machines.

7 SEQUENCE OF SEEDS

If we are going to run large amounts of simulated events of the same type at different laboratories, or in parallel processors on the same computer, we want to control the start seeds of each run. As stated above the integer sequences of the two seeds in the new generator are identical on all 32-bit computers and thus can be freely exchanged although different computers will give different sequences of events starting from the same values for the seeds. Note that it is not recommendable to invent seeds since this may severly affect the period and the quality of the generated random numbers.

To achieve this we first want to know how many random numbers, or rather how many seeds[5], are used in e.g. the "worst" fully simulated event. As given above a fully simulated $c\bar{c}$-event consumes about one million seed pairs. Adding a safety margin on this we would can decide at what intervals we could start different sequences.

To exhaust a full sequence of the generator in order to get start values would take more than a million years[6] on the fastest computer tested here (some gain could be achieved by extracting the seed updating code only). However, as we described above the two intrinsic seed sequences are independent from each other and each one of them has a period which is possible to exhaust. This means that we can predict seeds for any position in the sequence without running through the whole sequence until that point.

To predict the seeds is thus possible. It is, however, still quite time consuming and also involves handling very big integers. Because of this we developed a program which given any number of values, for which we want to find the seeds, loops through the two sequences once only.

As an example the seeds for the multiplicative congruential generator (MCG) and the shift register generator (SRG) after N updates for some values of N are given in the table below.

N	MCG seed	SRG seed
0	12345	1073
10^{18}	-1357893575	1744470304
$2 \cdot 10^{18}$	1579167801	1002460372
$3 \cdot 10^{18}$	221261881	428909540
$4 \cdot 10^{18}$	-1136644039	-1540004162

8 MACHINE INDEPENDENCE

The final routine written follows the rules of standard FORTRAN 77. There are, however, still some problems in making a machine independent and efficient routine. As was pointed out previously we are using the functions `IAND`, `IEOR` and `ISHFT`. These are not intrinsic or inline functions on all computers. Where they do not exist the CERN package BITPAK (M441) could be used. This, however, would slow the routine down and we would prefer to find inline solutions.

Note also that some computers do not like entry points with different argument lists. Indeed the entry points for setting and obtaining the seeds of the sequence are treated as subroutines with two arguments while the entries for random numbers are treated as functions with one argument. This is simply solved by either reintroducing a `COMMON`-block or by making specific entries for setting and obtaining each of the two intrinsic seeds. In either case the two entry points `RNDIN` and `RNDOUT` would become two separate subroutines.

[5] A call to `RNDUNI` consumes one seed pair while a call to `RNDNRM` consumes on average 1.16 seed pairs.

[6] As compared to less than two hours for our present generator `RNDM`.

The situation on various computers are as follows

- On Alliant, NORD and VAX computers the bit manipulating functions are intrinsic and the routine are well tested.
- On Apollo computers the routine compiles but we have not sofar made any detailed tests on it. Calls to `IAND`, `IEOR` and `ISHFT` should be exchanged to `AND`, `XOR`, `LSHFT` and `RSHFT` on Apollo.
- On the CERN IBM 3090 the bitmanipulating routines should exist as inline functions for the VS-compiler according to the manual. We have compiled and tested the routine on the CERN IBM. No detailed timing has been done sofar but it seems that the BITPAK routines are used with the result of a quite slow performance of the routine.
- On the CERN Siemens 7890 the bit manipulating routines might exist as well but the original names of these routines, which might be preferable in any case, are `LAND`, `LXOR`, `SHFTL` and `SHFTR`.
- For CDC and Cray we have not yet prepared any versions of the routine. It should not be any problem doing this adding a few instructions masking out 32 bits only. This should be tested in order to have the routine working on "all" computers. There must most certainly, however, exist better generators specially made for these computers with good enough periods which could be used instead. We recall that having 64-bit word length implies that a multiplicative congruential generator, which would be easy to construct, will have a period of $2^{62} \approx 4.6 \cdot 10^{18}$ (i.e. slightly longer than the period for the routine presented here). Also the precision of such a generator will naturally be better than what can be achieved on 32-bit computers. If precision were our problem one may construct such a generator using double precision on 32-bit machines [2].

Note that the shift instructions we need should not be circular and that they should fill with zeros. One way of introducing a machine dependence would be to use statement functions interfacing `IAND`, `IEOR` and `ISHFT` to the local routines (at least the two first, the `ISHFT` instruction is less easy to interface this way since e.g. depending on the sign of the second argument one should select `RSHFT` or `LSHFT` on an Apollo computer). However, this is not the most efficient way on all computers and to get the best performance we have to consider changing in the code. Using PATCHY this can be done by SELF-sequences and some parts of the code used in several places can be placed in KEEP-blocks.

9 SUMMARY

We have constructed a new random number generator for the generation of uniform and standard normal random numbers. The main emphasis has been to obtain a generator with a longer period than the present $1.1 \cdot 10^9$ period of `RNDM`. The new generator is based on the CERN library routines `RNDM2` and `NORRAN` and has a period of $4.6 \cdot 10^{18}$ random numbers. It gives about 50 % slower generation of uniform random numbers as compared to `RNDM` (on VAX computers) but allows the most efficient generation of standard normal random numbers known to us. The latter fact will probably make the new generator competitive even in terms of its speed. The final routine presented here has been tested on some computers (VAX, Alliant, NORD, Apollo and IBM) and is believed to be sufficiently machine independent that it should work on other computers as well. Start seeds can be found at arbitrary points in the sequence and distributed to different laboratories if needed.

A soft changeover to the new routines would be obtained by interfacing calls to `RNDM`, `RANNOR` etc to the new routine just as was once done in order to use only the `RNDM` sequence. In the long run, however, we should change the program to call the new routine directly.

Please send any suggestions, questions or requests to user WALCK on node SESUF51 at the BITNET/EARN network.

References

[1] *DELSIM 3.1 DELPHI Event Generation and Detector Simulation.* DELPHI 87-96 PROG-99, 7 January 1988.

[2] Ch.Walck. *Random Number Generation.* University of Stockholm, USIP 87-15, December 1987.

[3] N.J.Kjær and K.Spang. *The HLaTeX Data Presentation Program.* 12 April 1988.

[4] *CERN Program Library Manual.*

[5] Birger Jansson. *Random Number Generators.* Victor Pettersons Bokindustri AB, Stockholm (1966).

[6] John M. Chambers. *Computational Methods for Data Analysis.* John Wileys & Sons, New York-Chichester-Brisbane-Toronto (1977).

[7] J.H.Ahrens and U.Dieter and A.Grube. *Pseudo-Random Numbers: A New Proposal for the Choice of Multipliers.* Computing **6** (1970) 121-138.

[8] G.E.P.Box and M.E.Muller. *A Note on the Generation of Random Normal Deviates.* Annals of Math. Stat **29** (1958) 610-611.

[9] G.Marsaglia and T.A.Bray. *A Convenient Method for Generating Normal Variables.* SIAM review **6** (1964) 260-264.

[10] J.H.Ahrens and U.Dieter. *Computer Methods for Sampling from the Exponential and Normal Distributions.* Comm. ACM **15** (1972) 873-882.

[11] U.Dieter and J.H.Ahrens. *Extensions of Forsythe's Method for Random Sampling from the Normal Distribution.* Math. Comp. **27** (1973) 927-937.

[12] G.Marsaglia and M.D.MacLaren and T.A.Bray. *A Fast Procedure for Generating Normal Random Variables.* Comm. ACM **7** (1964) 4-10.

10 APPENDIX A

Below we give the final FORTRAN routine obtained by merging RNDM2 and NORRAN into one routine using the same random number sequence.

```
      FUNCTION RNDUNI ( DUMMY )
C-----------------------------------------------------------------------
C
C...       FUNCTION RNDUNI ( DUMMY )                       CH.WALCK    880506
C.
C.    Combined version of CERN routines NORRAN (GENLIB V101) and RNDM2
C.    (GENLIB V107) for efficient generation of uniform (0 to 1) and
C.    standard normal pseudorandom numbers.
C.
C.    The generator is using a common random number sequence with a
C.    period of 4.609.432.020.664.188.928 = 4.6*10**18. In this version
C.    the old "bit manipulations" have been replaced by straightforward
C.    multiplications (plus a test for zero) which, in fact, is faster
C.    (on VAXes). The function RNORTH used by NORRAN has been merged
C.    into this routine and in order to avoid recursive calls as well
C.    as COMMON-blocks we have, in this version, explicitly made the
C.    generation needed within the RNORTH-code. This makes the routine
C.    longer and maybe less elegant but we feel it is worth it in order
C.    to become as machine independent and efficient as possible.
C.
C.    The code is written in standard FORTRAN 77. However, the functions
C.    IAND, IEOR and ISHFT referenced may not be intrinsic or inline
C.    functions on some computers. In these cases the BITPAK package on
C.    the CERN library (entry M441) can be used. This, however, makes
C.    the routine much slower and it is better if inline alternatives
C.    can be found.
C.
C.    Calling sequences:
C.    R = RNDUNI ( DUMMY )           Continuous uniform r.n. 0 to 1
C.    R = RNDVNI ( DUMMY )           Continuous uniform r.n. -1 to 1
C.    R = RNDNRM ( DUMMY )           Standard normal r.n.
C.    IR = IRNDUN ( DUMMY )          Discrete uniform r.n. 0 to 2**31-1
C.    CALL RNDIN ( ISEED1, ISEED2 )  Initialize seeds
C.    CALL RNDOUT ( ISEED1, ISEED2 ) Access seeds
C.
C.    The arithmetics involved causes integer overflows which is handled
C.    on a VAX by calling ERRSET once from the main program as follows:
C.       CALL ERRSET(70,.TRUE.,.FALSE.,.FALSE.,.FALSE.,100)
C.    or by compiling this routine with /CHECK=NOOVERFLOW
C.
C.----------------------------------------------------------------------
      SAVE C, I1, I2, MCGN, SRGN, TBL
      PARAMETER ( M20=2**20-1, M24=2**24-1, XM24=2.0**(-24)
     +          , XM28=2.0**(-28) )
      DIMENSION C(45), TBL(0:326)
      INTEGER R0, R2, SEED1, SEED2, SIG, SRGN
      EQUIVALENCE (MAN,XMAN)
      DATA MCGN /12345/, SRGN /1073/
```

```
      DATA I1 /-71085056/, I2 /-25595858/
      DATA C / 0.9889430404, 0.9889430404, 0.9791515470, 0.9595685005
     +       , 0.9497770071, 0.9301939607, 0.9008194208, 0.8812363744
     +       , 0.8518618345, 0.8224872947, 0.7833212614, 0.7539466619
     +       , 0.7147806287, 0.6756145358, 0.6364485025, 0.5972824097
     +       , 0.5679078698, 0.5287418365, 0.4895757437, 0.4504097104
     +       , 0.4210351706, 0.3818690777, 0.3524945378, 0.3231199980
     +       , 0.2937454581, 0.2643709183, 0.2349963784, 0.2154133320
     +       , 0.1860387921, 0.1664557457, 0.1468726993, 0.1272896528
     +       , 0.1174981594, 0.0979151130, 0.0881236196, 0.0783321261
     +       , 0.0685405731, 0.0587490946, 0.0489575788, 0.0391660631
     +       , 0.0293745473, 0.0293745473, 0.0195830315, 0.0195830315
     +       , 0.0195830315 /
      DATA TBL/   0.0000,   0.0625, 2*0.1250, 4*0.1875, 5*0.2500
     +        ,   0.5625, 5*0.6250, 3*0.8750,   1.1250,   1.4375
     +        , 5*0.0000, 5*0.0625, 4*0.1250, 2*0.1875,   0.2500
     +        , 5*0.3125, 5*0.3750, 5*0.4375, 5*0.5000, 4*0.5625
     +        , 4*0.6875, 4*0.7500, 4*0.8125,   0.8750, 3*0.9375
     +        , 3*1.0000, 3*1.0625, 2*1.1250, 2*1.1875, 2*1.2500
     +        , 2*1.3125, 2*1.3750,   1.4375,   1.5000,   1.5625
     +        ,   1.6250,   1.6875,   1.7500,   1.8125,10*0.3125
     +        , 7*0.3750, 5*0.4375, 2*0.5000, 9*0.6875, 5*0.7500
     +        ,   0.8125,10*0.9375, 7*1.0000, 3*1.0625,12*1.1875
     +        , 9*1.2500, 5*1.3125, 2*1.3750,13*1.5000,10*1.5625
     +        , 7*1.6250, 5*1.6875, 2*1.7500,15*1.8750,13*1.9375
     +        ,12*2.0000,10*2.0625, 9*2.1250, 8*2.1875, 7*2.2500
     +        , 6*2.3125, 5*2.3750, 4*2.4375, 3*2.5000, 3*2.5625
     +        , 2*2.6250, 2*2.6875 /
C
C     RNDUNI entry (old RNDM2 and old UNI entry for RNORTH) for the
C     generation of uniform pseudorandom numbers between 0 and 1.
C     Not used by RNORTH-code in this version in order to avoid
C     recursive calls and COMMON-block.
C
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      MAN  = ISHFT ( IEOR(SRGN,MCGN) , -8 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      RNDUNI = XM24 * XMAN
      RETURN
C
C     RNDVNI entry (old VNI entry for RNORTH) for the generation of
C     uniform pseudorandom numbers between -1 and 1. Not used by
C     RNORTH-code in this version in order to avoid recursive calls
C     and COMMON-block.
C
      ENTRY RNDVNI ( DUMMY )
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      R2   = IEOR ( SRGN, MCGN )
```

```
      MAN  = IAND ( ISHFT(R2,-7), M24 )
      SIG  = 1 - 2 * ISHFT(R2,-31)
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      RNDVNI = FLOAT(SIG) * XM24 * XMAN
      RETURN
C
C     IRNDUN (old IRNDM2) entry for the generation of uniform discrete
C     pseudorandom numbers from 0 to 2**31-1.
C
      ENTRY IRNDUN ( DUMMY )
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      IRNDUN = ISHFT ( IEOR(MCGN,SRGN) , -1 )
      RETURN
C
C     RNDNRM (old NORRAN) entry for the generation of standard
C     normal pseudorandom numbers. Note that RNDNRM is a function
C     subprogram and not a subroutine subprogram as NORRAN is.
C
      ENTRY RNDNRM ( DUMMY )
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      R2   = IEOR(SRGN,MCGN)
      R0   = ISHFT(R2,-24)
      IF ( R0 .GE. 104 ) GO TO 20
      MAN  = IAND(R2,M24)
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      RNDNRM = XM28 * XMAN + TBL(R0)
      RETURN
C
   20 IF ( R0 .GE. 208 ) GO TO 30
      R0   = R0 - 104
      MAN  = IAND(R2,M24)
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      RNDNRM = - ( XM28 * XMAN + TBL(R0) )
      RETURN
C
   30 R0   = ISHFT(R2,-20)
      IF ( R0 .GE. 3631 ) GO TO 40
      R0   = R0 - 3304
      MAN  = ISHFT ( IAND(R2,M20), 4 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      RNDNRM = XM28 * XMAN + TBL(R0)
      RETURN
C
   40 IF ( R0 .GE. 3934 ) GO TO 50
      R0   = R0 - 3607
```

```
      MAN  = ISHFT ( IAND(R2,M20), 4 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      RNDNRM = - ( XM28 * XMAN + TBL(R0) )
      RETURN
C
C     Old RNORTH code inserted here
C     -----------------------------
C     In this version all previous calls to RNDUNI and RNDVNI (which
C     then where in a separate deck) have been replaced by a direct
C     generation of such random numbers in the code. This makes this
C     part of the code long and "less attractive". However, it saves
C     us the trouble of the COMMON-block and avoids the recursive calls
C     present in the original NORRAN code. The old statements affected
C     are left commented preceding the new code.
C
   50 IF ( R2 .GT. I1 ) GO TO 60
C
C     S = RNDUNI(0)
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      MAN  = ISHFT ( IEOR(SRGN,MCGN) , -8 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      S    = XM24 * XMAN
C
C     T = RNDUNI(0)
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      MAN  = ISHFT ( IEOR(SRGN,MCGN) , -8 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      T    = XM24 * XMAN
C
      B    = AINT ( 7.0*(S+T) + 37.0*ABS(S-T) )
C
C     X    = RNDUNI(0) - RNDUNI(1)
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      MAN  = ISHFT ( IEOR(SRGN,MCGN) , -8 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      X0 = XM24 * XMAN
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      MAN  = ISHFT ( IEOR(SRGN,MCGN) , -8 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      X    = X0 - XM24 * XMAN
```

```
      RNDNRM = 0.0625 * ( X + SIGN (B,X) )
      RETURN
   60 IF ( R2 .GT. I2 ) GO TO 110
C
C  70 RNDNRM = 2.75 * RNDVNI(0)
   70 R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      R2   = IEOR ( SRGN, MCGN )
      MAN  = IAND ( ISHFT(R2,-7), M24 )
      SIG  = 1 - 2 * ISHFT(R2,-31)
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      X0   = FLOAT(SIG) * XM24 * XMAN
      RNDNRM = 2.75 * X0
      J    = 16.0 * ABS (RNDNRM) + 1.0
      IF ( J - 14 ) 80, 80, 90
   80 P    = FLOAT(J+J-1) * 0.1497466E-2
      GO TO 100
   90 P    = FLOAT(89-J-J) * 0.698817E-3
C
C 100 IF ( RNDUNI(0) .GT. 79.78846 * (EXP (-0.5 * RNDNRM * RNDNRM)
C    1       - C(J) - P * ( J - 16.0*ABS(RNDNRM) ) ) ) GO TO 70
  100 R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      MAN  = ISHFT ( IEOR(SRGN,MCGN) , -8 )
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      X0   = XM24 * XMAN
      IF ( X0 .GT. 79.78846 * (EXP (-0.5 * RNDNRM * RNDNRM)
     1       - C(J) - P * ( J - 16.0*ABS(RNDNRM) ) ) ) GO TO 70
      RETURN
C
C     This is the tail-part for the region outside -2.75 to +2.75
C 110 V    = RNDVNI(0)
  110 R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      R2   = IEOR ( SRGN, MCGN )
      MAN  = IAND ( ISHFT(R2,-7), M24 )
      SIG  = 1 - 2 * ISHFT(R2,-31)
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      V    = FLOAT(SIG) * XM24 * XMAN
      IF ( V .EQ. 0.0 ) GO TO 110
      X    = SQRT ( 7.5625 - 2.0*LOG(ABS(V)) )
C
C     IF ( RNDUNI(0)*X .GT. 2.75 ) GO TO 110
      R0   = IEOR ( ISHFT(SRGN,-15), SRGN )
      SRGN = IEOR ( R0, ISHFT(R0,17) )
      MCGN = 69069 * MCGN
      MAN  = ISHFT ( IEOR(SRGN,MCGN) , -8 )
```

```
      XMAN = FLOAT(MAN)
      IF ( XMAN .EQ. 0.0 ) XMAN = 0.5
      X0   = XM24 * XMAN
      IF ( X0*X .GT. 2.75 ) GO TO 110
      RNDNRM = SIGN ( X, V )
      RETURN
C
C     Set seeds
C
      ENTRY RNDIN ( SEED1, SEED2 )
      MCGN = SEED1
      SRGN = SEED2
      RETURN
C
C     Get seeds
C
      ENTRY RNDOUT ( SEED1, SEED2 )
      SEED1 = MCGN
      SEED2 = SRGN
      RETURN
      END
```