3.1 Uniform Random Numbers

A. Purpose

Generate pseudorandom numbers from the uniform distribution. Capabilities are also provided for optionally setting and fetching the "seed" of the generator.

B. Usage

B.1 Generating uniform pseudorandom numbers

Three subprograms are provided for generation of single precision uniform random numbers:

X = SRANU() Returns one random number in [0, 1].

call SRANUA(XTAB, N) Returns an array of N random numbers in [0, 1].

call SRANUS(XTAB, N, A, B) Returns an array of N numbers scaled as A + B × U where U is random in [0, 1].

Corresponding double precision subprograms are also provided.

B.1.a Program Prototype, A Single Random Number, Single Precision

REAL SRANU, X

X = SRANU()

Argument Definitions

SRANU [out] The function returns a pseudorandom number from the uniform distribution on [0.0, 1.0].

B.1.b Program Prototype, An Array of Random Numbers, Single Precision

INTEGER N

REAL XTAB(≥N)

Assign a value to N.

CALL SRANUA(XTAB, N)

Computed values will be returned in XTAB().

Argument Definitions

XTAB() [out] Array into which the subroutine will store N pseudorandom samples from the uniform distribution on [0.0, 1.0].

N [in] Number of pseudorandom numbers requested. The subroutine returns immediately if N ≤ 0.

B.1.c Program Prototype, An Array of Scaled Random Numbers, Single Precision

INTEGER N

REAL XTAB(≥N)**, A, B**

Assign values to N, A, and B.

CALL SRANUS(XTAB, N, A, B)

Computed values will be returned in XTAB().

Argument Definitions

XTAB() [out] Array into which the subroutine will store N numbers computed as A + B × U, where for each number, U is a pseudorandom sample from a uniform distribution on [0.0, 1.0].

N [in] Number of pseudorandom numbers requested. The subroutine returns immediately if N ≤ 0.

A, B [in] Numbers defining the linear transformation (A + B × U) to be applied to the random numbers.

B.2 Modifications for Double Precision

For double precision usage change the REAL type statements above to DOUBLE PRECISION and change the initial "S" of the function and subroutine names to "D." Note particularly that if the function name, DRANU, is used it must be typed DOUBLE PRECISION either explicitly or via an IMPLICIT statement.

B.3 Operations relating to the seed

The handling of the seed is modeled on the function RANDOM_SEED which is a new intrinsic function introduced in Fortran 90. Random number generation does not require any initialization calls by the user, but initialization capabilities are provided in case they are wanted.

The seed for random number generation is a set of KSIZE numbers of type INTEGER. The value of KSIZE depends on the algorithm and implementation used for generating uniform random numbers.

call RANSIZ(KSIZE) Returns the value of KSIZE for the current library implementation.

call RAN1 Sets the seed to its default initial value.

call RANPUT(KSEED) Sets the seed to the array of KSIZE values given in KSEED().

call RANGET(KSEED) Fetches the current seed into the array KSEED().

Besides resetting the seed, RAN1 and RANGET set values in common that have the effect of reinitializing all of the pseudorandom number generators of Chapters 3.1, 3.2 and 3.3.

If one needs to produce the same sequence of pseudorandom numbers more than once within the same run, a suggested approach is to initialize the package at each point in the computation where the sequence is to be started or restarted. One could either use RAN1 to initialize the package to its standard starting seed or use RANPUT to initialize the package to a seed selected by the user.

The seed returned by RANGET may be the seed associated with the next uniform number that will be returned by the package, but generally this will not be the case. Due to buffering within the package this seed may be associated with a uniform number that will be returned some tens of requests later.

A potential use for the RANGET function would be to assure a different set of random numbers on a subsequent run. Thus one could use RANGET at the end of a run and write to a file the seed value returned by RANGET. Then on a subsequent run one could read this seed from the file and use RANPUT to initialize the package to this seed value. This would assure a new sequence of numbers.

B.3.a Program Prototype, Get the value of KSIZE

INTEGER KSIZE

CALL RANSIZ(KSIZE)

A value will be returned in KSIZE.

Argument Definitions

KSIZE [out] The subroutine sets KSIZE to the number of integers needed to constitute a seed for the current library implementation of a random number generation algorithm. The user should use this information to verify that the dimension of the array KSEED() is adequate before calling RANPUT or RANGET. The preferred algorithm in the MATH77 library has KSIZE = 2. If this is replaced by a different algorithm KSIZE could change.

B.3.b Program Prototype, Set seed to default value

CALL RAN1

Argument Definitions

This subroutine has no arguments. It causes the seed stored in the random number generation code to be reset to its default initial value. It also sets values in common that have the effect of reinitializing all of the pseudorandom number generators of Chapters 3.1, 3.2, and 3.3.

B.3.c Program Prototype, Set the seed

INTEGER KSEED(≥KSIZE)

Assign values to KSEED().

CALL RANPUT(KSEED)

Argument Definitions

KSEED() [in] Array of KSIZE integers to be used to set a new seed value. Any integer values are acceptable. If the given values do not conform to internal requirements the subroutine will derive usable values from the given values.

In the preferred MATH77 implementation the internal integer sequence consists of numbers in the range from 1 to 68719476502. For example, to set the seed to the value 10987654321 one should set KSEED(1) = 109876 and KSEED(2) = 54321. In general RANPUT will compute KSEED(1) $\times 10^5$ + KSEED(2) using either single precision or double precision arithmetic, depending on the "mode" described in Section D, and then alter the result, if necessary, to obtain a seed in the range from 1 to 68719476502.

This subroutine also sets values in common that have the effect of reinitializing all of the pseudorandom number generators of Chapters 3.1, 3.2, and 3.3.

B.3.d Program Prototype, Get the seed

INTEGER KSEED(≥KSIZE)

CALL RANGET(KSEED)

Values will be returned in KSEED(). See the discussion at the beginning of Section B.2 for information on the applicability of this subprogram.

Argument Definitions

KSEED() [out] Array into which the subroutine will store the KSIZE integers constituting the current seed.

C. Examples and Remarks

DRSRANU demonstrates the use of SRANU to compute uniform random numbers and uses SSTAT1 and SSTAT2 to compute and print statistics and a histogram based on a sample of 10000 numbers delivered by SRANU.

The uniform distribution on [0, 1] has mean 0.5 and standard deviation $\sqrt{1/12} \approx 0.288675$.

The smallest number that can be produced by SRANU, DRANU, SRANUA, or DRANUA is approximately 0.15×10^{-10}. The largest value that can be produced

is approximately $1.0 - 0.15 \times 10^{-10}$. The single precision subprograms SRANU and SRANUA will return this largest value as exactly 1.0 on many computer systems.

DRDRAN provides a critical test of the correct performance of the core integer sequence generator on whatever host system it is run. The seed values are set to cause the generation of the largest and smallest numbers possible in the underlying integer sequence. This program is expected to generate exactly the same values in the column headed "Integer sequence" on all compiler/computer systems. DRDRAN also calls RN2 to show the value of MODE (described below in Section D) being used on the host system. See the output listing, ODDRAN, for results.

To compute random numbers, uniform in [C, D], one can use the statement

```
X =C + (D − C) * SRANU()
```

or to put N such numbers into an array XTAB() one can write

```
call SRANUS(XTAB, N, C, D − C)
```

To compute random INTEGER's in the range from I1 through I2, (I1 < I2), with equal probability, one can write

```
FAC = real(I2 − I1 + 1)
K = min(I2, I1 + int(FAC * SRANU() ) )
```

The min function is used in the above statement because SRANU will, with very low probability, return the exact value 1.0.

If one needs to compute many random numbers and execution time is critical, one should note that one call to SRANUA with a sizeable value of N will take less execution time than N references to SRANU. For even greater efficiency one could write the random number generation in line, since it only requires a few declarations and a few executable statements. When and if Fortran 90 compilers come into widespread usage it will probably be more efficient to use the new intrinsic subroutine RANDOM_NUMBER, although this subroutine is not specified to generate the same sequence on different computers.

D. Functional Description

D.1 The core algorithm for generation of uniform pseudorandom numbers

A sequence of integer values, k_i, is generated by the equation

$$k_i = ak_{i-1} \bmod m \qquad (1)$$

The rational number, k_i/m, is returned as a pseudorandom number from the uniform distribution on [0, 1].

When m is prime and a is a primitive root of m, this integer sequence has period $m - 1$, attaining all integer values in the range [1, $m - 1$]. According to [1], this sequence will have good equidistribution properties, at least in dimensions up to d, if the numbers ν_i and μ_i, $i = 2, ..., d$, that are functions of m and a, are not exceptionally small. Finding satisfactory values of ν_i and μ_i is simplified by having m large and a not too small. The size of m and a is limited however by the requirement of computing Eq. (1) exactly at a reasonable cost.

We have determined a pair of integers m and a that satisfy all these requirements. The values of ν_i and μ_i, $i = 2, ..., 6$, attained are excellent compared with any of the 30 (m, a) pairs listed in Table 1, pp. 102–103 of [1], which includes pairs used in a number of widely distributed random number generation subprograms. We use the values

$$\text{MDIV} = m = 6\_87194\_76503 = 2^{36} - 233$$

and

$$\text{AFAC} = a = 612\_662 \approx 0.58 \times 2^{20}$$

The number $m - 1$ is the product of three primes, $p(i)$, listed here with other relevant number-theoretic values.

i	$p(i)$	$q(i) = (m-1)/p(i)$	$a^{q(i)} \bmod m$
1	2	3_43597_38251	$m - 1$
2	43801	15_68902	2_49653_21011
3	784451	87602	1_44431_31136

The fact that values in the last column above are not 1 verifies that a is a primitive root of m.

The values of μ_i and log base 2 of ν_i are

$$(Log_2\nu_i,\ i = 2, 6) = 18.00,\ 12.00,\ 8.60,\ 7.30,\ 6.00$$
$$(\mu_i,\ i = 2, 6) = 3.00,\ 3.05,\ 3.39,\ 4.55,\ 6.01$$

These values may be compared with Table 1, pp. 102–103, [1], that lists the same measures for a number of other random number generators.

This package contains both a short and a long algorithm to implement Eq. (1). Let XCUR be the program variable containing the current value of k_i of Eq. (1). The short algorithm for advancing XCUR is

```
XCUR = mod(AFAC * XCUR, MDIV).
```

The long algorithm, using ideas from [2], is

```
Q = aint(XCUR/B)
R = XCUR − Q * B
XCUR = AFAC * R − C * Q
do while(XCUR .lt. 0.0)
   XCUR = XCUR + MDIV
end do
```

where B and C are constants related to MDIV and AFAC by MDIV = B × AFAC + C. We use B = 112165 and C = 243273. The average number of executions of the statement XCUR = XCUR + MDIV is 1.09 and the maximum number of executions is 3.

The largest number that must be handled in the short algorithm is the product of AFAC with the maximum value of XCUR, *i.e.*, $612\_662 \times 6\_87194\_76502 = 42\_10181\_19126\_68324 \approx 0.58 \times 2^{56}$. Thus, the short algorithm requires arithmetic exact to at least 56 bits.

The largest number that must be handled in the long algorithm is the product of C with the maximum value of aint(XCUR/B), *i.e.*, $243273 \times 612664 \approx 0.14904 \times 10^{12} \approx 0.54 \times 2^{38}$. Thus the long algorithm requires arithmetic exact to at least 38 bits.

To accommodate different compiler/computer systems this program unit contains code for 3 different ways of computing the new XCUR from the old XCUR, each producing the same sequence of values. Initially we have MODE = 1. When MODE = 1 the code does tests to see which of the three implementation methods will be used, and sets MODE = 2, 3, or 4 to indicate the choice.

Mode 2 will be used in machines such as the Cray that have at least a 38 bit significand in single precision arithmetic. XCUR will be advanced using the long algorithm in single precision arithmetic.

Mode 3 will be used on machines that don't meet the Mode 2 test, but can maintain at least 56 bits exactly in computing mod(AFAC × XCUR, MDIV) in double precision arithmetic. This includes VAX, UNISYS, IBM 30xx, and some IEEE machines that have clever compilers that keep an extended precision representation of the product AFAC × XCUR within the math processor for use in the division by MDIV. XCUR will be advanced using the short algorithm in double precision arithmetic.

Mode 4 will be used on machines that don't meet the Mode 2 or 3 tests, but have at least a 38 bit significand in double precision arithmetic. This includes IEEE machines that have not-so-clever compilers. XCUR is advanced using the long algorithm in double precision arithmetic.

If a user wishes to know which mode has been selected, the statement

CALL RN2(MODE)

can be used after at least one access has been made to one of the random number generators or to RANPUT or RANGET. This call will set the integer variable MODE to the mode value of 2, 3, or 4 that the package is using.

D.2 Remarks on alternative algorithms

From [3] and [4] it appears that a multiplicative congruential generator of the form of Eq. (1) using the values MDIV = 2147483647 = $(2^{31}) - 1$ and AFAC = 16807 = $7^5 \approx 0.513 \times 2^{15}$ has been widely used, and is quite satisfactory. However, the associated values of μ_i and ν_i are smaller than those associated with the MDIV and AFAC we are using. With these values, XCUR would take all integer values in [1, MDIV − 1]. The maximum value of the product, AFAC × XCUR would be approximately $0.361 \times 10^{14} \approx 0.51 \times 2^{46}$, so at least 46-bit arithmetic would be needed.

The method of [2] is interesting in that it illustrates techniques for getting a very long period generator ($0.36 \times 10^{14} \approx 0.51 \times 2^{46}$) using relatively low-precision arithmetic. This method uses at least three integer mod operations, three integer multiplications, three floating divisions, two floating additions, and a floating mod; there does not appear to be any theoretical measure of the quality of the sequence, such as the μ_i and ν_i described in [1].

D.3 Organization of the package

The basic uniform pseudorandom number generation algorithm for the MATH77 library is contained in the program unit RANPK2 that returns an array of N numbers when called at any one of the entry points, SRANUA, SRANUS, DRANUA, or DRANUS. A single sequence of pseudorandom integers is managed within this program unit. Calling any of these four entry points will cause updating of this single pseudorandom integer sequence. The other random number subprograms in Chapters 3.2 and 3.3 depend on uniform pseudorandom numbers obtained by calling SRANUA or DRANUA.

The function SRANU is a separate program unit. SRANU references common blocks that contain a REAL buffer array of length 97, and an index into the array. If the value of the index indicates the buffer contains unused random numbers, SRANU simply decrements the index and returns the next number from the buffer. If the index indicates the buffer is empty, SRANU calls SRANUA to fill the buffer with uniform random numbers and then reinitializes the index and returns one random number.

The common blocks are also referenced and used in this way by other random number subprograms needing single-precision uniform random numbers: SRANE, SRANG, SRANR, and IRANP.

The double-precision subprograms, DRANU, DRANE, DRANG, and DRANR share reference to a different common block that is used similarly to buffer an array of double-precision uniform pseudorandom numbers.

RAN1 and RANPUT are entries in a program unit RANPK1. When either of these entries is called, the pointers in common will be set to indicate the empty state and a call will be made to the appropriate one of two private entry points in program unit RANPK2 to make the requested change in the seed.

RANSIZ and RANGET are entries in RANPK2 that simply return stored values, a constant in the case of RANSIZ, and a variable integer array in the case of RANGET.

D.4 Accuracy tests

We have run tests of equidistribution in 1, 2, and 3 dimensions, as well as the run test and gap test described in [1]. Results were satisfactory. The main basis for confidence in this algorithm is the number-theoretic properties of the pair (m, a) described above.

Values returned as double-precision random numbers will have random bits throughout the word, however the quality of randomness should not be expected to be as good in a low-order segment of the word as in a high-order part.

References

1. Donald E. Knuth, **Seminumerical Algorithms**, volume 2 of *The Art of Computer Programming*, Addison Wesley, Reading, Mass., second edition (1981).
2. B. A. Wichmann and I. D. Hill, *An efficient and portable pseudo-random number generator*, **Applied Statistics** (1982) 188–190.
3. G. P. Learmonth and P. A. W. Lewis. *Statistical tests of some widely used and recently proposed uniform random number generators.* in **Proc. Seventh Interface Symposium on Computer Science and Statistics**, (1973). Also see NPS-55L@-73111A, Naval Postgraduate School, Monterey, 20 pp.
4. G. P. Learmonth. *Empirical tests of multipliers for the prime-modulus random number generator,* $x_{i+1} \equiv ax_i \mod 2^{31} - 1$. in **Proc. Ninth Interface Symposium on Computer Science and Statistics**, 178–183, (1976).

E. Error Procedures and Restrictions

If the argument N in SRANUA, SRANUS, DRANUA, or DRANUS is nonpositive the subroutine will return immediately and make no reference to the array XTAB().

When using RANPUT or RANGET, the user must assure that the array, KSEED(), has an adequate dimension. Violation of this condition will have unpredictable effects. The user can call RANSIZ to determine the required dimension.

If none of the three modes of computation (See MODE = 2, 3, or 4 in Section D.) succeeds, the program unit RANPK2 will write an error message directly to the system output unit and stop. This is unlikely, as it would only happen if the host system cannot at least do exact 38-bit arithmetic in double precision.

F. Supporting Information

The source language is ANSI Fortran 77.

Entry	Required Files
DRANU	DRANU, ERFIN, ERMSG, RANPK1, RANPK2
DRANUA	ERFIN, ERMSG, RANPK2
DRANUS	ERFIN, ERMSG, RANPK2
RAN1	ERFIN, ERMSG, RANPK1, RANPK2
RANGET	ERFIN, ERMSG, RANPK2
RANPUT	ERFIN, ERMSG, RANPK1, RANPK2
RANSIZ	ERFIN, ERMSG, RANPK2
RN2	ERFIN, ERMSG, RANPK2
SRANU	ERFIN, ERMSG, RANPK1, RANPK2, SRANU
SRANUA	ERFIN, ERMSG, RANPK2
SRANUS	ERFIN, ERMSG, RANPK2

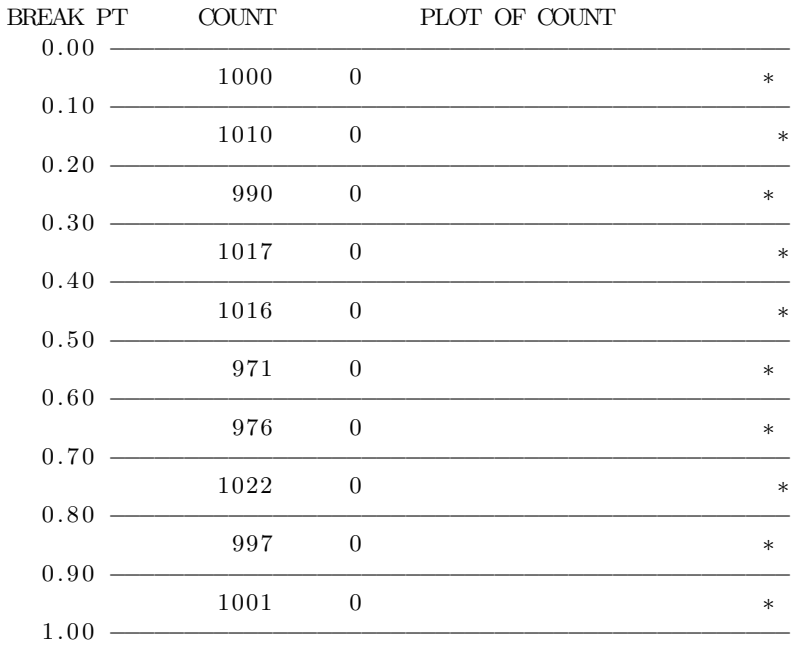
Designed by C. L. Lawson and F. T. Krogh, JPL, April 1987. Programmed by C. L. Lawson and S. Y. Chiu, JPL, April, 1987. November 1991: Lawson redesigned RANPK2 to have MODES 2, 3, and 4 for better portability. Also reorganized and renamed common blocks.

DRSRANU

```
c      program DRSRANU
c>> 1996-05-28 DRSRANU Krogh Added external statement.
c>> 1994-10-19 DRSRANU  Krogh  Changes to use M77CON
c>> 1992-03-13 DRSRANU  CLL
c>> 1987-12-09 DRSRANU  Lawson  Initial Code.
c
c      Driver to demonstrate use of SRANU to generate random numbers
c      from the uniform distribution on [0.0, 1.0].
c      Program computes histogram for N numbers
c     ------------------------------------------------------------------
c--S replaces "?": DR?RANU, ?RANU, ?STAT1, ?STAT2
c     ------------------------------------------------------------------
      integer N, NCELLS
      parameter(N = 10000, NCELLS = 10+2)
      external SRANU
      real              SRANU, STATS(5), YTAB(1), Y1, Y2
      integer I, IHIST(NCELLS)
      data Y1, Y2 / 0.0E0, 1.0E0 /
c     ------------------------------------------------------------------
      STATS(1) = 0.0E0
      do 20 I = 1, N
c                             Get random number
         YTAB(1) = SRANU( )
c                             Accumulate statistics and histogram.
c
        call SSTAT1(YTAB(1), 1, STATS, IHIST, NCELLS, Y1, Y2)
   20 continue
c                             Print the statistics and histogram.
c
      write(*,'(13x,a//)') 'Uniform random numbers from SRANU'
      call SSTAT2(STATS, IHIST, NCELLS, Y1, Y2)
      stop
      end
```

ODSRANU

Uniform random numbers from SRANU



Count	Minimum	Maximum	Mean	Std. Deviation
10000	0.28041E−03	0.99990	0.49944	0.28929

DRDRAN

```
c      program drdran
c>> 2001-07-16 DRDRAN Krogh  Added comma in two fomrats.
c>> 1996-06-19 DRDRAN Krogh  Minor format change for C conversion.
c>> 1994-10-19 DRDRAN Krogh  Changes to use M77CON
c>> 1992-02-24 DRDRAN CLL
c>> 1987-12-10 Original time stamp
c--D replaces "?": DR?RAN, ?RANUA
c++S Default NDIG = 6
c++  Default NDIG = 12
c++  Replace "f15.12" = "f"//NDIG+3//"."//NDIG
c      Reports MODE for host system, and prints a few integers in the
c      integer sequence underlying the pseudorandom number package for
c      the MATH77 library.  These integers should be exactly the
c      same on all host systems.  The listed integers include the
c      smallest and largest in the entire sequence.
c     ------------------------------------------------------------------
      integer I, KASE, KSEED(2,2), MODE
      double precision X(1)
      data KSEED(1,1), KSEED(2,1) / 249979, 65550 /
      data KSEED(1,2), KSEED(2,2) / 437215, 10953 /
c     ------------------------------------------------------------------
      do 20 KASE = 1,2
         print'(1x//4x,''Integer sequence'',10x,''Number returned''/1x)'

         print'(7x,i7,'','','',i5)',KSEED(1,KASE),KSEED(2,KASE)
         call RANPUT(KSEED(1,KASE))
         do 10 I = 1,10
            call DRANUA(X,1)
            call RANGET(KSEED(1,KASE))
         print'(7x,i7,'','','',i5,10x,f15.12)',KSEED(1,KASE),KSEED(2,KASE),
     *           X(1)
   10    continue
   20 continue
      call RN2(MODE)
      print'(/1x,a,i2)',
     *   'MODE may be 2, 3, or 4.  On the current host it is ',MODE
      stop
      end
```

ODDRAN

Integer sequence	Number returned
249979,65550	
687194,76502	0.999999999985
687188,63841	0.999991084594
369621,32774	0.537869824611
276585,97792	0.402485571769
422879,97043	0.615371350234
441300,56424	0.642176842282
239519,29877	0.348546454308
115303,75451	0.167789046683
668584,81671	0.972918960872
327383,74992	0.476406059213

Integer sequence	Number returned
437215,10953	
0, 1	0.000000000015
6,12662	0.000008915406
317573,43729	0.462130175389
410608,78711	0.597514428231
264314,79460	0.384628649766
245894,20079	0.357823157718
447675,46626	0.651453545692
571891, 1052	0.832210953317
18609,94832	0.027081039128
359811, 1511	0.523593940787

MODE may be 2, 3, or 4. On the current host it is 4