RANDLIB90
Fortran 95 Routines for Random Number Generation
User's Guide

Barry W. Brown
James Lovato

1 Technicalities

1.1 Obtaining the Code

The source for this code (and all code written by this group) can be obtained from the following URL:

http://odin.mdacc.tmc.edu/anonftp/

1.2 Legalities

1.3 References

1.3.1 Base Generator

The base generator and all code in the ecuyer_cote_mod come from the reference:
P. L'Ecuyer and S. Cote. (1991) "Implementing a Random Number Package with Splitting Facilities." *ACM Trans. on Math. Softw. 17*:1, pp 98-111.
We transliterated the Pascal of the reference to Fortran 95.

1.4 The Beta Random Number Generator

R. C. H. Cheng (1978) "Generating Beta Variates with Nonintegral Shape Parameters." *Communications of the ACM, 21*:317-322 (Algorithms B and BC)

1.5 The Binomial Random Number Generator

Kachitvichyanukul, V. and Schmeiser, B. W. (1988) "Binomial Random Variate Generation." *Communications of the ACM, 31*: 216. (Algorithm BTPE.)

1.6 The Standard Exponential Random Number Generator

Ahrens, J.H. and Dieter, U. (1972) "Computer Methods for Sampling from the Exponential and Normal Distributions." *Communications of the ACM, 15*: 873-882. (Algorithm SA.)

1.7 The Standard Gamma Random Number Generator

Ahrens, J.H. and Dieter, U. (1982) "Generating Gamma Variates by a Modified Rejection Technique." *Communications of the ACM, 25*: 47-54. (Algorithm SA.)

1.8 The Standard Normal Random Number Generator

Ahrens, J.H. and Dieter, U. (1973) "Extensions of Forsythe's Method for Random Sampling from the Normal Distribution." *Math. Comput., 27*:927 - 937. Algorithm FL (method=5)

1.9 ACM Policy on Use of Code

Here is the software Policy of the ACM.

> Submittal of an algorithm for publication in one of the ACM Transactions implies that unrestricted use of the algorithm within a computer is permissible. General permission to copy and distribute the algorithm without fee is granted provided that the copies are not made or distributed for direct commercial advantage. The ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
>
> Krogh, F. (1997) "Algorithms Policy." *ACM Tran. Math. Softw. 13*, 183-186.

Here is our standard disclaimer.

> NO WARRANTY
>
> WE PROVIDE ABSOLUTELY NO WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY

AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THIS PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT SHALL THE UNIVERSITY OF TEXAS OR ANY OF ITS COMPONENT INSTITUTIONS INCLUDING M. D. ANDERSON HOSPITAL BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA OR ITS ANALYSIS BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES) THE PROGRAM.

(Above NO WARRANTY modified from the GNU NO WARRANTY statement.)

2 Introduction: The ecuyer_cote_mod module

The base random generator for this set of programs contains 32 virtual random number generators. Each generator can provide 1,048,576 blocks of numbers, and each block is of length 1,073,741,824. Any generator can be set to the beginning or end of the current block or to its starting value. The methods are from the paper cited immediately below, and most of the code is a transliteration from the Pascal of the paper into Fortran.
Most users won't need the sophisticated capabilities of this package, and will desire a single generator. This single generator (which will have a non-repeating length of 2.3×10^18 numbers) is the default.

3 Initializing the Random Number Generator

You almost certainly want to start the random number generation by setting the seed. (Defaults are provided if the seed is not set, but the defaults will produce the same set of numbers on each run of the program which is generally not what is wanted.)
Make sure that the program unit that sets the seed has
USE user_set_generator at its top.
The most straightforward way of setting the values need by the generator is a call to USER_SET_ALL.

SUBROUTINE USER_SET_ALL(SEED1, SEED2, GENERATOR)

ARGUMENTS

- **INTEGER, INTENT(IN) :: SEED1** The first number used to set the generator.
- **INTEGER, INTENT(IN) :: SEED2** The second number used to set the generator.
- **INTEGER, INTENT(IN), OPTIONAL :: GENERATOR** The current generator. If this argument is not present, the current generator is set to 1. Input Range: [1 : 32]

It is easier to remember an English phrase, such as your name, than it is to remember two large integer values. The following routine prompts interactively for a phrase (non-blank character string – only the first 80 characters are used) and hashes the phrase to set the seeds of the random number generator. The current generator is set to 1.

SUBROUTINE INTER_PHRASE_SET_SEEDS()

This routine calls the following routine which is the non-interactive version of the same action.

SUBROUTINE PHRASE_SET_SEEDS(PHRASE)
CHARACTER(LEN=*), INTENT(IN) :: PHRASE

This in turn calls the lowest level routine:

CALL PHRASE_TO_SEED(PHRASE, SEED1, SEED2)
CHARACTER(LEN=*), INTENT(IN) :: PHRASE
INTEGER, INTENT(OUT) :: SEED1, SEED2

In this call, phrase is an (input) character string of arbitrary length and seed1 and seed2 are two (output) integer values that may be used to initialize the seeds of the random number generator.
Setting the seeds to a fixed value is extremely useful in the debugging phase of a project where exact runs are to be replicated. However, there are occasions in which a random appearing sequence is desired – an example is computer games. The following routine sets the seeds of the generator using the system clock.

SUBROUTINE TIME_SET_SEEDS()

If the computer doesn't have a clock (unlikely) then the routine queries the user for a phrase to use to set the seeds. The current generator is set to 1.
The next described routine allows the programmer to select setting seeds by time or from a phrase entered interactively.

SUBROUTINE SET_SEEDS(WHICH)
INTEGER, OPTIONAL, INTENT(IN) :: WHICH

If WHICH is not present, the routine queries the user as to whether time or a phrase is to be used to set the random number seeds. If WHICH is present in the calling list and equals 1 then the seeds are set from the time; if WHICH is present and not equal to 1 then the seeds are set for a phrase promted from the user..
The current generator is set through

SUBROUTINE SET_CURRENT_GENERATOR(G)
INTEGER, INTENT(IN) :: G

The integer argument G sets the number of the current generator to its value which must be between 1 and 32.

4 random_beta_mod

FUNCTION RANDOM_BETA(A,B)

4.1 The Distribution

The density of the beta distribution is defined on x in $[0, 1]$ and is proportional to

$$x^a(1-x)^b$$

4.2 Arguments

- **REAL, INTENT(IN) :: A.** The first parameter of the beta distribution (a above). Input Range: $[10^{-37} : \infty]$

- **REAL, INTENT(IN) :: B.** The second parameter of the beta distribution (b above). Input Range: $[10^{-37} : \infty]$

5 random_binomial_mod

REAL FUNCTION RANDOM_BINOMIAL(N,PR)

5.1 The Distribution

The density of the binomial distribution provides the probability of S successes in N independent trials, each with probability of success PR.
The density is proportional to

$$PR^S(1-PR)^{N-S}$$

The routine returns values of S drawn from this distribution.

5.2 Arguments

- **INTEGER, INTENT(IN) :: N.** The number of binomial trials. Input Range: $[0:\infty]$

- **REAL, INTENT(IN) :: PR.** The probability of success at each trial. Input Range: $[0:1]$

6 random_chisq_mod

REAL FUNCTION RANDOM_CHISQ(DF)

6.1 The Distribution

The chi-squared distribution is the distribution of the sum of squares of DF independent unit (mean 0, sd 1) normal deviates.
The density is defined on x in $[0, \infty)$ and is proportional to

$$x^{(DF-2)/2} \exp(-x/2$$

6.2 Argument

- **REAL, INTENT(IN) :: DF.** The degrees of freedom of the chi-square distribution. Input Range: $[10^{-37} : \infty]$

7 random_exponential_mod

REAL FUNCTION RANDOM_EXPONENTIAL(AV)

7.1 The Distribution

The density is:

$$1/AV \exp(-x/AV)$$

7.2 Argument

- **REAL, INTENT(IN) :: AV.** The mean of the exponential; see density above. Input Range: $[10^{-37} : \infty]$

8 random_gamma_mod

REAL FUNCTION RANDOM_GAMMA(scale,shape)

8.1 The Distribution

The density of the GAMMA distribution is proportional to:

$$(x/SCALE)^{SHAPE-1} \exp(-x/SCALE)$$

8.2 Arguments

- **REAL (dpkind), INTENT(IN) :: SHAPE.** The shape parameter of the distribution (See above.)
  Input Range: $[10^{-37} : \infty]$

- **REAL (dpkind), INTENT(IN) :: SCALE.** The scale parameter of the distribution.
  Input Range: $[10^{-37} : \infty]$

9 random_multinomial_mod

SUBROUTINE RANDOM_MULTINOMIAL(N,P,NCAT,IX)

9.1 The Distribution

Each observation falls into one category where the categories are numbered $1 \ldots NCAT$. The observations are independent and the probability that the outcome will be category i is $P(i)$. There are N observations altogether, and the number falling in category i is IX(i).

9.2 Arguments

- **INTEGER, INTENT(IN) :: N** The number of observations from the multinomial distribution. Input Range: $[1 : \infty]$

- **REAL, DIMENSION(NCAT-1), INTENT(IN) :: P** P(i) is the probability that an observation falls into category i. NOTE: The P(i) must be non-negative and less than or equal to 1. Only the first NCAT-1 P(i) are used, the final one is obtained from the restriction that all NCAT P's must add to 1.

- **INTEGER, INTENT(IN) :: NCAT** The number of outcome categories. NCAT must be at least 2.

- **INTEGER, INTENT(OUT), IX(NCAT)**

10 random_multivariate_normal_mod

11 The Distribution

The multivariate normal density is:

$$(2\pi)^{-(k/2)}|COVM|^{-1/2}\exp(\left[-(1/2)(X-MEANV)^{T}COVM^{-1}(X-MEANV)\right])$$

where superscript T is the transpose operator.

12 Use of Routines

First, CALL SET_RANDOM_MULTIVARIATE_NORMAL. This routine processes and saves information needed to generate multivariate normal deviates. Successive calls to RANDOM_MULTIVARIATE_NORMAL generate deviates from values specified in the most recent SET routine.
SUBROUTINE SET_RANDOM_MULTIVARIATE_NORMAL(MEANV,COVM,P)

12.1 Arguments

- **REAL, INTENT(IN), DIMENSION(:) :: MEANV.** The mean of the multivariate normal. The size of MEANV must be at least P; if it is more than P, entries following the P'th are ignored.
- **REAL, INTENT(IN), DIMENSION(:,:) :: COVM.** The variance covariance matrix of the multivariate normal. Both dimensions must be at least P; if the matrix is bigger than PXP, extra entries are ignored.
- **INTEGER, INTENT(IN) :: P.** The dimension of the multivariate normal deviates to be generated.

SUBROUTINE RANDOM_MULTIVARIATE_NORMAL(X)

12.2 Arguments

- **REAL, INTENT(OUT), DIMENSION(:) :: X.** Contains the generated multivariate normal deviate. NOTE. No check on the size of X is made – the programmer must assure that it is big enough.

13 random_nc_chisq_mod

FUNCTION random_nc_chisq(df,pnonc)

13.1 The Distribution

The noncentral chi-squared distribution is the sum of DF independent normal distributions with unit standard deviations, but possibly non-zero means . Let the mean of the ith normal be δ_i. Then PNONC = $\sum_i \delta_i$.

13.2 Arguments

- **REAL, INTENT(IN) :: DF.** The degrees of freedom of the noncentral chi-squared.
- **REAL, INTENT(IN) :: PNONC.** The noncentrality parameter of the noncentral chi-squared.

14 random_negative_binomial_mod

FUNCTION random_negative_binomial(s,pr)

14.1 The Distribution

The density of the negative binomial distribution provides the probability of precisely F failures before the S'th success in independent binomial trials, each with probability of success PR.
The density is

$$\left(\begin{array}{c} F+S-1 \\ S-1 \end{array} \right) PR^S (1-PR)^F$$

This routine returns a random value of F given values of S and P.

14.2 Arguments

- **INTEGER, INTENT(IN) :: S.** The number of succcesses after which the number of failures is not counted.
- **REAL, INTENT(IN) :: PR.** The probability of success in each binomial trial.

15 random_normal_mod

FUNCTION random_normal(mean,sd)

15.1 The Distribution

The density of the normal distribution is proportional to

$$\exp\left(-\frac{(X-MEAN)^2}{2SD^2}\right)$$

15.2 Arguments

- **REAL, INTENT(IN) :: MEAN.** The mean of the normal distribution.
- **REAL, INTENT(IN) :: SD.** The standard deviation of the normal distribution.

16 random_permutation_mod

SUBROUTINE RANDOM_PERMUTATION(ARRAY,LARRAY)

16.1 Function

Returns a random permutation of (integer) array.

16.2 Arguments

- **INTEGER, INTENT(INOUT) :: ARRAY.** On input an array of integers (usually 1..larray). On output a random permutation of this array.
- **INTEGER, INTENT(IN), OPTIONAL:: LARRAY.** If present the length of ARRAY to be used . If absent, all of ARRAY is used – i.e., the SIZE function is invoked to determine the dimensioned size of ARRAY

17 random_poisson_mod

FUNCTION random_poisson(lambda)

17.1 The Distribution

The density of the Poisson distribution (probability of observing S events) is:

$$\frac{LAMBDA^S}{S!}\exp(-LAMBDA)$$

17.2 Argument

- **REAL, INTENT(IN) :: LAMBDA.** The mean of the Poisson distribution.

17.3 random_uniform_integer

FUNCTION RANDOM_UNIFORM_INTEGER(LOW,HIGH)

17.4 Function

Returns a random integer between LOW and HIGH (inclusive of the limits).

17.5 Arguments

- **INTEGER, INTENT(IN) :: LOW.** Low limit of uniform integer.
- **INTEGER, INTENT(IN) :: HIGH.** High limit of uniform integer.

18 random_uniform_mod

FUNCTION random_uniform(low,high)

18.1 Function

Returns a random real between LOW and HIGH.

18.2 Arguments

- **REAL, INTENT(IN) :: LOW.** Low limit of uniform real.
- **REAL, INTENT(IN) :: HIGH.** High limit of uniform real.

19 The "Standard" Generator Modules

It is not really intended that these modules be used directly. However, for completeness, here is a listing of them.

19.1 random_standard_exponential_mod

FUNCTION random_standard_exponential()

19.1.1 Function

Returns a random value from an exponential distribution with mean one.

19.2 random_standard_normal_mod

FUNCTION random_standard_normal()

19.2.1 Function

Returns a random value from a normal distribution with mean 0 and standard deviation one.

19.3 random_standard_uniform_mod

FUNCTION random_standard_uniform()

19.3.1 Function

Returns a random value from a uniform distribution on zero to one.

20 Advanced Use of the ecuyer_cote_mod Module

Recall the following information from the Introduction.

> The base random generator for this set of programs contains 32 virtual random number generators. Each generator can provide 1,048,576 blocks of numbers, and each block is of length 1,073,741,824. Any generator can be set to the beginning or end of the current block or to its starting value.

20.1 Setting the Virtual Random Number Generator

SUBROUTINE set_current_generator(g)
Sets the current virtual generator to the integer value g, where $1 \leq g \leq 32$. Before this routine is called, the current generator is 1.

20.1.1 Arguments

- **INTEGER, INTENT(IN) :: G.** The number of the virtual generator to be used.

20.2 Getting the Virtual Random Number Generator

SUBROUTINE get_current_generator(g)

20.2.1 Arguments

- **INTEGER, INTENT(OUT) :: G.** The number of the virtual generator currently in use.

20.3 Reinitializing the Current Virtual Generator

SUBROUTINE reinitialize_current_generator(isdtyp)
Reinitializes the state of the current generator.

20.3.1 Arguments

- **INTEGER, INTENT(IN) :: ISDTYP.** The initialization action to be performed.

  -1 Sets the seeds of the current generator to their value at the beginning of the current run.

  0 Sets the seeds to the first value of the current block of seeds.

  1 Sets the seeds to the first value of the next block of seeds.

21 Getting the Current Seeds of the Virtual Random Number Generator

SUBROUTINE get_current_seeds(iseed1,iseed2)

21.1 Arguments

Gets the current value of the two integer seeds of the current generator.

- **INTEGER, INTENT(OUT) :: ISEED1.** The value of the first seed.
- **INTEGER, INTENT(OUT) :: ISEED2.** The value of the second seed.

22 Setting the Current Seeds of the Virtual Random Number Generator

SUBROUTINE set_current_seeds(iseed1,iseed2)

22.1 Arguments

Sets the current value of the two integer seeds of the current generator.

- **INTEGER, INTENT(IN) :: ISEED1.** The value of the first seed.
- **INTEGER, INTENT(IN) :: ISEED2.** The value of the second seed.

23 Producing Antithetic Random Numbers from the Current Generator

SUBROUTINE set_antithetic(qvalue)
Sets the current generator to produce antithetic values or not. If X is the value normally returned from a uniform [0,1] random number generator then 1 - X is the antithetic value. If X is the value normally returned from a uniform [0,N] random number generator then N - 1 - X is the antithetic value. All generators are initialized to NOT generate antithetic values.

23.1 Arguments

- **LOGICAL, INTENT (IN) :: qvalue** If .TRUE., antithetic values are generated. If .FALSE., non-antithetic (ordinary) values are generated.

24 Bottom Level Support Routines

SUBROUTINE advance_state(k)
Advances the current state of the seeds of the current virtual random number generator by 2^k values.

- **INTEGER, INTENT (IN) :: k** The seeds of the current virtual generator are set ahead by s^k values.

FUNCTION multiply_modulo(a,s,m)
Returns (A*S) modulo M.

- **INTEGER, INTENT(IN) :: A** First component of product. Must be $<$ M.
- **INTEGER, INTENT(IN) :: S** Second component of product. Must be $\leq$ M.
- **INTEGER, INTENT(IN) :: M** Integer with respect to which the modulo operation is performed.