LA-UR-07-7961

Approved for public release;
distribution is unlimited.

Title:	The MCNP5 Random Number Generator
Author(s):	Forrest Brown
Intended for:	MCNP Reference Material

Form 836 (7/06)

The MCNP5 Random Number Generator

Forrest Brown

Monte Carlo Codes

X-3-MCC, Los Alamos National Laboratory

Monte Carlo particle transport codes use random number generators to produce random variates from a uniform distribution on the interval (0,1). These random variates are then used in subsequent sampling from probability distributions to simulate the physical behavior of particles during the transport process. This paper describes the new random number generator developed for MCNP Version 5. The new generator will optionally preserve the exact random sequence of previous versions of MCNP and is entirely conformant to the Fortran-90 standard, hence completely portable. In addition, skip-ahead algorithms have been implemented to efficiently initialize the generator for new histories, a capability that greatly simplifies parallel algorithms. Further, the precision of the generator has been increased, extending the period by a factor of 10^5. Finally, the new generator has been subjected to 3 different sets of rigorous and extensive statistical tests to verify that it produces a sufficiently random sequence.

Overview & Theory:

- The MCNP5 Random Number Generator
- Random Number Generation with Arbitrary Strides

Application & Usage

- Using the MCNP Random Number Generator

Fortran Coding

Random Number Generator:	mcnp_random.f90
Sample Main Program for Testing:	main.f90

C Coding

Random Number Generator:	mcnp_random.c
Sample Main Program for Testing:	main.c

The MCNP5 Random Number Generator

INTRODUCTION

MCNP [1] and other Monte Carlo particle transport codes use random number generators to produce random variates from a uniform distribution on the interval [0,1). These random variates are then used in subsequent sampling from probability distributions to simulate the physical behavior of particles during the transport process. This paper describes the new random number generator developed for MCNP Version 5 [2]. The new generator will optionally preserve the exact random sequence of previous versions and is entirely conformant to the Fortran-90 standard, hence completely portable. In addition, skip-ahead algorithms have been implemented to efficiently initialize the generator for new histories, a capability that greatly simplifies parallel algorithms. Further, the precision of the generator has been increased, extending the period by a factor of 10^5. Finally, the new generator has been subjected to 3 different sets of rigorous and extensive statistical tests to verify that it produces a sufficiently random sequence.

BACKGROUND

The random number generator in MCNP and most other Monte Carlo codes for particle transport (e.g,, RACER, MORSE, KENO, VIM, RCP, MCPP) is based on algorithms called linear congruential generators (LCGs) [3]. The basic LCG in these codes has been in use for at least 40 years, and has several desirable properties:

1. The sequence is deterministic, so that repeated calculations will produce identical results.
2. LCGs are very fast, involving only a small number of arithmetic operations.
3. Initialization is trivial, and the state information to specify the sequence for a history is small (1 word).
4. A simple algorithm exists for skipping ahead to any given point in the random sequence.
5. If at least 48 bits of precision are used in the LCG, the period is large ($>10^{14}$) and serial correlation is entirely negligible.
6. The algorithm is robust – it cannot fail.
7. An extensive body of literature exists for LCGs, providing a sound theoretical basis and guidance for the proper choice of algorithm parameters.

The LCG traditionally used by MCNP and other codes has the form

$$S_{k+1} = g\,S_k + c \bmod 2^M \qquad (1)$$

$$r_{k+1} = S_{k+1} / 2^M$$

where S_k, g, and c are integers expressible in M bits or fewer, and r_{k+1} is a floating pointing number – the "random number" in the interval [0,1). The initial value of S_k, S_0, is called the initial seed for the generator. If c=0 and (g mod 8) = 3 or 5, then the generator has a period 2^{M-2}. If $c \neq 0$ and (g mod 4)=1 and c is odd, then the period is 2^M. The traditional LCG for MCNP uses $g=5^{19}$, c=0, $S_0=5^{19}$, and M=48. We will refer to generators in the form LCG(g, c, S_0, M), so that the traditional MCNP generator is LCG(5^{19}, 0, 5^{19}, 48).

Repeated application of Eq. (1) permits expressing the k^{th} seed in terms of the initial seed:

$$S_k = g^k S_0 + c\,(g^k-1)/(g-1) \bmod 2^M \qquad (2)$$

Eq. (2) must be computed using exact integer arithmetic. An algorithm for doing so was previously presented in [4] and has been incorporated into the new MCNP random number generator. Eq. (2) can be used to skip ahead in the random sequence by an arbitrary number of steps. This is particularly useful in initializing the random seed for a history in MCNP, since the skip-distance, or "stride" between starting random seeds for successive histories is fixed.

NEW RANDOM NUMBER GENERATOR

As part of a general upgrade to the MCNP code, a new random number package has been developed and implemented in MCNP Version 5. Because of the long experience with LCGs and the desirable features noted in the previous section, this new package is based on the LCG algorithm. This new package preserves all previously existing capabilities and provides many important new features:

Coding Considerations

The coding is entirely standard Fortran-90 and is completely portable. It has been tested on SGI, Sun, IBM, Compaq, HP, and Intel systems using a variety of Fortran-90 compilers. No special options or "#ifdefs" are required, and identical coding executes successfully on all systems. The generator has been implemented in a modular fashion, such that all parameters are private to the module and not subject to inadvertent side effects from other portions of the code. The module is thread-safe for parallel calculations using OpenMP threading. The module contains the LCG parameters, functions for generating random numbers, functions for skip-ahead in the random sequence and initialization for histories, unit tests for verifying correctness of the functions, and reference test results. A C version of the generator is also provide, with functionality identical to the Fortran version.

Algorithm Considerations

Parameters for 7 standard LCGs are included in the new random number package. These are listed in Table (1). The default LCG parameters are those for the traditional MCNP random number generator, set 1 in the table. Using the default parameters, the standard 48-bit LCG algorithm from previous versions of MCNP is preserved, yielding a bit-for-bit identical stream of random numbers for verification against previous versions of the code.

The standard 48-bit LCG algorithm in MCNP has a period of 2^{46} (~7 x 10^{13}). With modern parallel computers, large calculations may simulate 10^9 or more particle histories (with ~10^5 random numbers per history), resulting in the reuse of portions of the random number sequence. To avoid this problem, the LCG algorithm has been extended to use up to 63-bits and incorporate an additive term, so that a period of 2^{63} (~9.2 x 10^{18}) is achieved. The use of a 63-bit integer LCG algorithm, rather than 64-bits, was deliberate, to avoid coding complications arising from the 2's complement form for integers (i.e., the leading bit is interpreted as a sign bit for Fortran on most systems). The LCG parameter sets 2-7 in Table (1) are natural extensions of the traditional 48-bit MCNP LCG to 63 bits. The LCG parameter sets 2-7 are recommended based on [5], where a search for "best" parameters was performed.

An algorithm for fast "skip-ahead" using arbitrary strides in the sequence [4] has been implemented in the MCNP5 random number generator, greatly simplifying the LCG initialization for new histories. This feature is invaluable for reducing the complexity of parallel calculations.

Both the standard and extended LCG algorithms have been subjected to thorough testing, including:

- The Unix utility "bc" was used to perform extended-precision integer arithmetic in order to generate reference values for the random streams generated by Eq. (1) for each of the sets of LCG parameters listed in Table (1). These reference values were compared to those generated by the Fortran-90 random number module. In addition, the reference values were included in the module data section for use in unit testing routines.

- The standard statistical tests for random number generators described by Knuth [3] were applied to the random streams generated by each of the LCGs listed in Table (1). Using test parameters from [6,7], the test suite comprised 29 variations on Knuth's 9 statistical tests. Coding for these tests was obtained from the SPRNG package [8]. In a few cases, a single test was flagged as suspect. Repeating the test with a different initial seed (which should not affect test results) resulted in passing, indicating that some tests are sensitive to statistical fluctuations. None of the LCGs from Table (1) consistently failed any of the tests for randomness. (Note that these statistical tests cannot be used to prove randomness; consistent failure of several tests, however, is a good indicator of non-randomness.)

- Marsaglia's DIEHARD test suite for random number generators [9] was applied to each of the LCGs listed in Table (1). This test suite involves running over 200 variations on 18 different statistical tests. Only one of the LCGs from Table (1) failed any of the tests, LCG set 1, the traditional 48-bit MCNP generator. For 3 tests – the overlapping pairs sparse occupancy, the overlapping quadruples sparse occupancy test, and the DNA test – the generator failed when the least-significant 10-12 bits of the random numbers were used for testing. The generator passed these tests when higher-order bits were used. These failures are not deemed serious, since it is well-known and understood that the least-significant bits of LCGs may be

non-random and should not be used. In fact, these bits are not used directly in any portion of MCNP; only the higher-order bits are important.

CONCLUSIONS

As a result of the above testing, we believe that any of the 7 LCGs listed in Table (1) may be reliably used for Monte Carlo particle transport calculations. The traditional MCNP generator will remain the default, to provide consistency with older calculations. For new work, we are currently recommending sets 2-7. Any of these should be satisfactory and robust. Sets 2-4 include an additive constant, so that the period of the generator will be 2^{63}, a factor of ~10^5 longer than the traditional MCNP generator. Sets 5-7 do not use an additive constant, hence have a period of 2^{61}.

The extended LCG algorithm is the foundation for future planned work in MCNP – providing independent random number generators for different particle types (in order to allow reproducibility when particle physics options are turned on/off). The use of different carefully selected additive constants has been shown [10] both in theory and practice to be a reliable method of producing different independent and uncorrelated random streams.

The new random number generator package for MCNP is entirely self-contained and portable. It can be used directly in other Monte Carlo codes or stand-alone packages; there is no dependence on other portions of MCNP. It is highly recommended that other code developers make use of this thoroughly tested, well-proven package if new random number generator capabilities are needed.

REFERENCES

[1] J.F. Briesmeister, Ed., “MCNP – A General Monte Carlo N-Particle Transport Code – Version 4C,” LA-13709-M, Los Alamos National Laboratory (March, 2000)

[2] X-5 Monte Carlo Team, “MCNP – A General N-Particle Transport Code, Version 5 – Volume I: Overview and Theory”, LA-UR-03-1987, Los Alamos National Laboratory (April, 2003)

[3] D.E. Knuth, The Art of Computer Programming – Volume 2, Seminumerical Algorithms, 3rd Edition, pp. 1-170, Addison-Wesley (1991).

[4] F.B. Brown, “Random Number Generation with Arbitrary Strides,” *Trans. Am. Nucl. Soc.* (Nov., 1994)

[5] P. L’Ecuyer, “Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure,” *Math. of Comp*., **68**, 225, pp 249-260 (1999)

[6] P. L'Ecuyer, “Efficient and Portable Combined Random Number Generators,” *Comm. ACM* **31**(6): 742-749, 774 (1988)

[7] I. Vattulainen, et al., “A comparative study of some pseudorandom number generators,” *Comput. Phys. Comm*. **86**, 209 (1995)

[8] M. Mascagni, et al., “The Scalable Parallel Random Number Generators Library (SPRNG) for ASCI Monte Carlo Computations,” http://sprng.cs.fsu.edu

[9] G.S. Marsaglia, “The DIEHARD Battery of Tests of Randomness,” http://stat.fsu.edu/pub/diehard

[10] O.E. Percus and M.H. Kalos “Random number generators for MIMD processors,” *J. Parallel and Distributed*

Computing, **6**, 477-497 (1989)

Table 1. Standard LCG parameters for new MCNP5 random number generator

Index	g	c	M	S_0	stride	Comments
1	5^{19}	0	48	5^{19}	152,917	Traditional MCNP generator
2	9219741426499971445	1	63	1	152,917	From Ref. [5]
3	2806196910506780709	1	63	1	152,917	From Ref. [5]
4	3249286849523012805	1	63	1	152,917	From Ref. [5]
5	3512401965023503517	0	63	1	152,917	From Ref. [5]
6	2444805353187672469	0	63	1	152,917	From Ref. [5]
7	1987591058829310733	0	63	1	152,917	From Ref. [5]

An earlier version of this document was published as

F.B. Brown & Y. Nagaya, “The MCNP5 Random Number Generator”, *Trans. Am. Nucl. Soc.* [also, LA-UR-02-3782] (November, 2002)

Random Number Generation with Arbitrary Strides

(Summary prepared for the 1994 ANS Winter Meeting, Washington D. C., Nov. 13-17, 1994)

In this paper, we present techniques for the generalized initialization of random number generators for parallel Monte Carlo calculations. In particular, we develop and prove an algorithm for a fast, direct method for skipping-ahead in the random sequence by an arbitrarily large positive or negative "stride." This algorithm is currently used in three different production-level Monte Carlo codes [1-3] on a variety of parallel, vector, and distributed computer systems to generate the initial seeds for different particle histories. In the extreme cases, the algorithm can reduce several years of "brute-force" computation to a few hundred microseconds.

Most of the production-level Monte Carlo codes used for radiation transport analysis rely on linear congruential generators for producing a uniform-(0,1) distribution of basic random numbers. These generators have the form

$$S_{k+1} = g\,S_k + c \bmod 2^M \qquad (1)$$

where g and c are integer constants in the range $[0, 2^M]$, and M is the number of bits used for the "seeds" S_k. If the values of M, g, and c are chosen properly [4], then the period of the sequence is 2^M for c≠0, or 2^{M-2} for c=O. For a particular code or computer, M is typically 32, 48, 63, or 64. The seeds are converted to fractions on (0,1) by multiplying by 2^{-M}. Generators of the form of Equation (1) have been used for more than 40 years in Monte Carlo codes for radiation transport, and are considered well-proven and very robust for these applications.

Special attention to random number generation is one of the principle requirements for performing Monte Carlo calculations on parallel computers. To permit reproducibility of calculations [5] regardless of the number of processors used, each particle in a batch is given its own random seed. Random number generation is then performed in parallel for a number of active particles. The most common procedure for choosing the seeds for each particle is to skip-ahead in the random sequence by a fixed number of steps or "stride" [6] for each particle. That is, given an initial seed of S_0 and a stride of L, the initial seed for particle N would be found by iterating Equation (1) for k =NL times, to get the k-th term of the sequence:

$$S_k = g^k S_0 + c\,(g^k-1)/(g-1) \bmod 2^M \qquad (2)$$

Equation (2) cannot be naively evaluated using ordinary computer arithmetic, since roundoff and truncation effects would rapidly corrupt the random sequence. If M is chosen as 48, then exact 96-bit integer arithmetic is required. In many Monte Carlo codes [6], Equation (2) is evaluated in practice by a "brute-force" method, simply iterating with Equation (1) L times for each successive particle seed. This can be expensive for large strides. For a fixed value of L, the constants needed for Equation (2) are often pre-computed and "hard-wired" into a code, thus prohibiting any investigation of alternate strides.

Fortunately, simple and fast algorithms have been developed for direct evaluation of Equation (2), greatly simplifying the initialization of parallel random number generators. Denoting the j-th bit of the binary representation of k as k(j) (with j=0 the least-significant bit), then the multiplier in the first term of Equation (2) may be computed by:

$$G = g^k \bmod 2^m = g^{\sum_{j=0}^{m-1} k_{(j)} 2^j} \bmod 2^m = \prod_{j=0}^{m-1} \left(g^{2^j} \right)^{k_{(j)}} \bmod 2^m \qquad (3)$$

A computational algorithm using recursion to evaluate G according to Equation (3) is

Algorithm G:

G ← 1, h ← g, i ← k + 2^M mod 2^M

while i > 0

if i = odd: G ← G h mod 2^M

h ← h^2 mod 2^M

i ←floor(i/2)

Thus Algorithm G can be evaluated in *O*(M) steps, rather than *O*(k) steps, which can be a significant savings when k is a very large number. Using periodicity, negative strides can be trivially handled as the equivalent positive stride. In a similar manner (although the proof is more complicated), the second term

$C = c\,(g^k - 1)/(g-1) \bmod 2^M = c\,[\,1 + g + g^2 + g^3 + \ldots + g^{k-1}\,] \bmod 2^M \qquad (4)$

from Equation (2) can be evaluated in *O*(m) steps via the algorithm:

Algorithm C:

C ← 0, f ← c, h ← g, i ← k + 2^M mod 2^M

while i > 0

if i = odd: C ← C h + f mod 2^M

f ← f (h+1) mod 2^M

h ← h^2 mod 2^M

i ←floor(i/2)

For many Monte Carlo codes the constant c is chosen to be 0, so that Algorithm C is not needed.

When c≠0, Algorithms G and C are readily combined.

To demonstrate the usefulness of Algorithms G and C, consider the specific random generator used in VIM and MCNP, where $g=5^{19}$, c=0, M=48. To compute a new particle seed directly using Algorithm G for the default MCNP stride of 152,917 requires about 80 μsec on an IBM rs6000/350 workstation, which is greater than the 2 μsec required if a pre-computed value of G is used. Changing the stride to 1,152,917 requires about 90 μsec for Algorithm G, versus about 2.5 sec for the brute-force iteration of Equation (1). A more extreme difference is seen for negative strides: To use a stride of -152,917 (i.e., $k \sim 7 \times 10^{13}$) would require about 2151 μsec using Algorithm G, but about 5 years using brute-force. Timing measurements and estimates for other computers show similar advantages of Algorithm G when large or negative strides are desired. More important, use of these new algorithms permits the fast and convenient use of arbitrary strides, completely user-selectable rather than "hard-wired" into the Monte Carlo codes.

For "ordinary" Monte Carlo calculations (if there are any), the algorithms presented here for initializing the particle seeds will have little or no impact on computer running times. For specialized calculations where a nonstandard, unusually large stride or negative stride is needed, the present algorithms can provide considerable flexibility and computer time savings. In addition, they are very convenient for initializing the particle seeds on different processors in a parallel calculation - sequential portions of the calculation are eliminated and no special communication is required. The techniques described in this paper for initializing the Monte Carlo random number generators are currently being used successfully in a number of production-level Monte Carlo codes on a variety of computers: RACER [1] (KAPL) on a Cray-C90 vector/parallel supercomputer and a Meiko CSI parallel computer, a parallel version of VIM [2] (ANL) on a workstation network and the IBM-SP1 parallel computer, and a parallel version of KENO-Va [3] (CSN-Spain) for a Convex-C3440 vector/parallel computer.

References

[1] F. B. Brown & T. M. Sutton, "Parallel Monte Carlo for Reactor Calculations," proc. ANS topical meeting on *Advances in Reactor Physics*, April 11-15, 1994, Knoxville TN (1994).
[2] R. N. Blomquist & F. B. Brown, "Parallel Monte Carlo Reactor Neutronics," Proc. Soc. Comp. Sim. meeting on *High Performance Computing '94*, April 11-15, 1994, La Jolla, CA (April 1994).
[3] J. Pena, Consejo de Segurldad Nuclear -Spain, private communication (May 9, 1994).
[4] D. E. Knuth, **The Art of Computer Programming -Vol. 2**, 2nd ed., pp. 9-25, Addison-Wesley (1981).
[5] F. B. Brown & T. M. Sutton, "Reproducibility and Monte Carlo Eigenvalue Calculations," Trans. Am. Nucl. Soc., 65, 234 (1992).
[6] J. S. Hendricks, "Random Number Stride in Monte Carlo Calculations," Trans. Am. Nucl. Soc., **62**, 283 (1990)

Using the MCNP Random Number Generator

For Fortran-90 programs:

- Every program, subroutine, or function that references any of the random number routines should include a Fortran-90 use statement for the module:

  ```
  use mcnp_random
  ```

  When compiling, the mcnp_random.f90 module should be compiled **before** any of the subprograms that use it. That is, do this

  ```
  g95 mcnp_random.f90 main.f90
  ```

  and not this

  ```
  g95 main.f90 mcnp_random.f90
  ```

- Some arguments to the routines must be INTEGER(4), while other arguments must be INTEGER(8). In the routine descriptions below, these will be abbreviated as I4 and I8. Be sure to check below for the proper argument types for each routine.

- To compile & execute with g95:

  ```
  g95   -o rntest      mcnp_random.f90   main.f90

  ./rntest
  ```

For C/C++ programs:

- The routines were developed and tested with the gcc compiler, which provides full support for UNSIGNED LONG LONG integers (64-bit unsigned integers). Other C/C++ compilers may use a different name for this data type, and some C/C++ compilers may not support them (in which case you're out of luck - switch to gcc).

- Some arguments to the routines must be LONG integers, while other arguments must be UNSIGNED LONG LONG. In the routine descriptions below, these will be abbreviated as L and ULL. Be sure to check below for the proper argument types for each routine.

- Arguments for all routines must be passed by **reference** (as for Fortran), not by value.

- To compile & execute with gcc:

  ```
  gcc     -o rntest    mcnp_random.c    main.c

  ./rntest
  ```

Usage for either Fortran-90 or C/C++

- Before generating any random numbers, it is generally a good idea to initialize the random number package. (If this is not done, you will get the default generator, initial seed, and stride.) This is done by calling **RN_init_problem**:

```
F90: call RN_init_problem( gen, seed, stride,
part1, prt )

C:   RN_init_problem( &gen, &seed, &stride,
&part1, &prt );
```

where the arguments are:

`gen {I4,L}`

An integer selecting one of the generators.

1 = The traditional MCNP 48-bit generator, period = 2^46.

Results are in the range (0,1).

2,3,4 = LeCuyer's 63-bit mixed generators, period = 2^63.

Results are in the range [0,1).

5,6,7 = LeCuyer's 63-bit multiplicative generators, period = 2^61.

Results are in the range (0,1).

`seed {I8,ULL}`

The initial seed for the problem. Default = 1. If the seed is 0 or negative, the default seed will be used.

`stride {I8,ULL}`

The stride, or number of RNs allotted to each particle history. Default = 152917. If the stride is 0 or negative, the default will be used.

`part1 {I8,ULL}`

The history number associated with the initial problem seed. Default=1. If part1 is 0 or negative, then the default will be used.

`prt {I4,L}`

Print control: 0 = do not print RN info, 1 = print RN info

- For particle transport problems, **RN_init particle** should be called at the start of each particle history to advance the random seed to the correct starting point for the history. This routine need not be called for other types of calculations, such as testing the random number routines or games which do not involve particle histories (e.g., "dart games").

```
F90: call RN_init_particle( n )

C:   RN_init_particle( &n );
```

where the argument is:

`n {I8/ULL}`

The history number for this particle.

- The random number routine **rang()** returns a random number uniformly distributed in the range (0,1). For gen= 1, 5, 6, or 7, the values returned do **not** include the endpoints of the range, 0.0 or 1.0. For gen= 2, 3, or 4, the value 0.0 may be returned, but not 1.0. There are no arguments to **rang()**. The result is a 64-bit real type, `REAL(8)` in Fortran-90 or `double` in C.

```
F90: r = rang()

C:   r = rang();
```

- To obtain the current value of any of the parameters for the random number package, use the **RN_query** function. The return value of this function is of type `INTEGER(8)` in Fortran-90 or `UNSIGNED LONG LONG` in C.

```
F90: i8 = RN_query( "key" )

C:   i8 = RN_query( "key" );
```

where the argument is:

key {character variable or string} is one of the following

- "seed" - current value of the random seed
- "stride" - current value of the stride
- "mult" - the multiplier for the RN generator
- "add" - the additive constant for the RN generator
- "count" - the number of times the RN generator has been called for this history

- "period" - the period of the RN generator
- "count_total" - total number of times the RN generator has been called for all particle histories
- "count_max" - the max number of RN calls for any history
- "count_nps" - the history number having count_max RN calls
- "first" - the initial RN seed for the first history

- To determine the first RN seed for the nth history, use the function **RN_query_first**, which returns an `INTEGER(8)` for Fortran-90 or `UNSIGNED LONG LONG` for C.

```
F90: i8 = RN_query_first( n )

C:   i8 = RN_query_first( &n );
```

where the argument is

`n {I8/ULL}`

A history number.

- The **RN_set** routine can be called to set or alter the value of some of the RN package parameters.

```
F90: call RN_set( "key", value )

C:   RN_set( "key", &value );
```

where "`key`" is a character string and `value` is the corresponding new value. value must be of type `INTEGER(8)` for Fortran-90 or `UNSIGNED LONG LONG` for C.

This routine should generally not be used except for special purposes. Do not use it just to set the seed or stride - use the **RN_init_problem** routine.

The "key" must be one of the following:

"stride", "count_total", "count_stride", "count_max", "count_max_nps", "seed", "part1"

- When using these routines for particle transport problems, RN_update_stats should be called at the completion of each particle history, prior to starting the next history.

```
F90: call RN_update_stats

C:   RN_update_stats();
```

- For testing the RN routines, there are three test routines supplied: **RN_test_basic**, **RN_test_skip**, **RN_test_mixed**.

```
F90: call RN_test_basic( gen )

C:   RN_test_basic( &gen )
```

and similar calls for **RN_test_skip** and **RN_test_mixed**, where the argument is

`gen {I4/L}`

The number of the generator (see **RN_init_problem**) to use for the test.

Random Number Generator

Fortran Coding

```
module mcnp_random
  !======================================================================
  ! Description:
  !  mcnp_random.F90 -- random number generation routines
  !======================================================================
  !  This module contains:
  !
  !   * Constants for the RN generator, including initial RN seed for the
  !     problem & the current RN seed
  !
  !   * MCNP interface routines:
  !     - random number function:           rang()
  !     - RN initialization for problem:    RN_init_problem
  !     - RN initialization for particle:   RN_init_particle
  !     - get info on RN parameters:        RN_query
  !     - get RN seed for n-th history:     RN_query_first
  !     - set new RN parameters:            RN_set
  !     - skip-ahead in the RN sequence:    RN_skip_ahead
  !     - Unit tests:        RN_test_basic, RN_test_skip, RN_test_mixed
  !
  !   * For interfacing with the rest of MCNP, arguments to/from these
  !     routines will have types of I8 or I4.
  !     Any args which are to hold random seeds, multipliers,
  !     skip-distance will be type I8, so that 63 bits can be held without
  !     truncation.
  !
  ! Revisions:
  ! * 10-04-2001 - F Brown, initial mcnp version
  ! * 06-06-2002 - F Brown, mods for extended generators
  ! * 12-21-2004 - F Brown, added 3 of LeCuyer's 63-bit mult. RNGs
  ! * 01-29-2005 - J Sweezy, Modify to use mcnp modules prior to automatic
  !                io unit numbers.
  ! * 12-02-2005 - F Brown, mods for consistency with C version
  !======================================================================

  !-------------------
  ! MCNP output units
  !-------------------
  integer, parameter ::   iuo = 6
  integer, parameter ::  jtty = 6

  PRIVATE
  !-----------------------------------------------------
  ! Kinds for LONG INTEGERS (64-bit) & REAL*8 (64-bit)
  !-----------------------------------------------------
  integer, parameter :: R8 = selected_real_kind(15,307)
  integer, parameter :: I8 = selected_int_kind(18)

  !-------------------------------------
  ! Public functions and subroutines for this module
  !-------------------------------------
  PUBLIC ::  rang
  PUBLIC ::  RN_init_problem
  PUBLIC ::  RN_init_particle
  PUBLIC ::  RN_set
  PUBLIC ::  RN_query
  PUBLIC ::  RN_query_first
  PUBLIC ::  RN_update_stats
  PUBLIC ::  RN_test_basic
  PUBLIC ::  RN_test_skip
  PUBLIC ::  RN_test_mixed

  !-------------------------------------
  ! Constants for standard RN generators
  !-------------------------------------
  type :: RN_GEN
    integer          :: index
    integer(I8)      :: mult        ! generator (multiplier)
    integer(I8)      :: add         ! additive constant
    integer          :: log2mod     ! log2 of modulus, must be <64
    integer(I8)      :: stride      ! stride for particle skip-ahead
    integer(I8)      :: initseed    ! default seed for problem
    character(len=8) :: name
  end type RN_GEN

  ! parameters for standard generators
  integer,      parameter :: n_RN_GEN = 7
  type(RN_GEN), SAVE      :: standard_generator(n_RN_GEN)
  data standard_generator / &
    & RN_GEN( 1,      19073486328125_I8, 0_I8, 48, 152917_I8, 19073486328125_I8 , 'mcnp std' ), &
    & RN_GEN( 2, 9219741426499971445_I8, 1_I8, 63, 152917_I8, 1_I8,               'LEcuyer1' ), &
    & RN_GEN( 3, 2806196910506780709_I8, 1_I8, 63, 152917_I8, 1_I8,               'LEcuyer2' ), &
    & RN_GEN( 4, 3249286849523012805_I8, 1_I8, 63, 152917_I8, 1_I8,               'LEcuyer3' ), &
    & RN_GEN( 5, 3512401965023503517_I8, 0_I8, 63, 152917_I8, 1_I8,               'LEcuyer4' ), &
    & RN_GEN( 6, 2444805353187672469_I8, 0_I8, 63, 152917_I8, 1_I8,               'LEcuyer5' ), &
    & RN_GEN( 7, 1987591058829310733_I8, 0_I8, 63, 152917_I8, 1_I8,               'LEcuyer6' )  &
    & /

  !----------------------------------------------------------------
  !   * Linear multiplicative congruential RN algorithm:
  !
  !            RN_SEED = RN_SEED*RN_MULT + RN_ADD  mod RN_MOD
  !
  !   * Default values listed below will be used, unless overridden
  !----------------------------------------------------------------
  integer,     SAVE :: RN_INDEX  = 1
  integer(I8), SAVE :: RN_MULT   =  19073486328125_I8
  integer(I8), SAVE :: RN_ADD    =                0_I8
  integer,     SAVE :: RN_BITS   = 48
  integer(I8), SAVE :: RN_STRIDE =          152917_I8
  integer(I8), SAVE :: RN_SEED0  =  19073486328125_I8
  integer(I8), SAVE :: RN_MOD    = 281474976710656_I8
  integer(I8), SAVE :: RN_MASK   = 281474976710655_I8
  integer(I8), SAVE :: RN_PERIOD =  70368744177664_I8
  real(R8),    SAVE :: RN_NORM   = 1._R8 / 281474976710656._R8

  !-------------------------------------
  ! Private data for a single particle
```

```
  !--------------------------------------
  integer(I8) :: RN_SEED   = 19073486328125_I8 ! current seed
  integer(I8) :: RN_COUNT  = 0_I8              ! current counter
  integer(I8) :: RN_NPS    = 0_I8              ! current particle number

  common                /RN_THREAD/   RN_SEED, RN_COUNT, RN_NPS
  save                  /RN_THREAD/
  !$OMP THREADprivate ( /RN_THREAD/ )

  !-----------------------------------------
  ! Shared data, to collect info on RN usage
  !-----------------------------------------
  integer(I8), SAVE :: RN_COUNT_TOTAL   = 0  ! total RN count all particles
  integer(I8), SAVE :: RN_COUNT_STRIDE  = 0  ! count for stride exceeded
  integer(I8), SAVE :: RN_COUNT_MAX     = 0  ! max RN count all particles
  integer(I8), SAVE :: RN_COUNT_MAX_NPS = 0  ! part index for max count

  !---------------------------------------------------------------------
  ! Reference data:  Seeds for case of init.seed = 1,
  !                  Seed numbers for index 1-5, 123456-123460
  !---------------------------------------------------------------------
  integer(I8), dimension(10,n_RN_GEN) ::  RN_CHECK
  data  RN_CHECK /  &
    ! ***** 1 ***** mcnp standard gen *****
    &      19073486328125_I8,      29763723208841_I8,     187205367447973_I8, &
    &     131230026111313_I8,     264374031214925_I8,     260251000190209_I8, &
    &     106001385730621_I8,     232883458246025_I8,      97934850615973_I8, &
    &     163056893025873_I8, &
    ! ***** 2 *****
    & 9219741426499971446_I8,  666764808255707375_I8, 4935109208453540924_I8, &
    & 7076815037777023853_I8, 5594070487082964434_I8, 7069484152921594561_I8, &
    & 8424485724631982902_I8,   19322398608391599_I8, 8639759691969673212_I8, &
    & 8181315819375227437_I8, &
    ! ***** 3 *****
    & 2806196910506780710_I8, 6924308458965941631_I8, 7093833571386932060_I8, &
    & 4133560638274335821_I8,  678653069250352930_I8, 6431942287813238977_I8, &
    & 4489310252323546086_I8, 2001863356968247359_I8,  966581798125502748_I8, &
    & 1984113134431471885_I8, &
    ! ***** 4 *****
    & 3249286849523012806_I8, 4366192626284999775_I8, 4334967208229239068_I8, &
    & 6386614828577350285_I8, 6651454004113087106_I8, 2732760390316414145_I8, &
    & 2067727651689204870_I8, 2707840203503213343_I8, 6009142246302485212_I8, &
    & 6678916955629521741_I8, &
    ! ***** 5 *****
    & 3512401965023503517_I8, 5461769869401032777_I8, 1468184805722937541_I8, &
    & 5160872062372652241_I8, 6637647758174943277_I8,  794206257475890433_I8, &
    & 4662153896835267997_I8, 6075201270501039433_I8,  889694366662031813_I8, &
    & 7299299962545529297_I8, &
    ! ***** 6 *****
    & 2444805353187672469_I8,  316616515307798713_I8, 4805819485453690029_I8, &
    & 7073529708596135345_I8, 3727902566206144773_I8, 1142015043749161729_I8, &
    & 8632479219692570773_I8, 2795453530630165433_I8, 5678973088636679085_I8, &
    & 3491041423396061361_I8, &
    ! ***** 7 *****
    & 1987591058829310733_I8, 5032889449041854121_I8, 4423612208294109589_I8, &
    & 3020985922691845009_I8, 5159892747138367837_I8, 8387642107983542529_I8, &
    & 8488178996095934477_I8,  708540881389133737_I8, 3643160883363532437_I8, &
    & 4752976516470772881_I8  /
  !---------------------------------------------------------------------

CONTAINS

  !---------------------------------------------------------------------

  function rang()
    ! MCNP random number generator
    !
    ! ***************************************
    ! ***** modifies RN_SEED & RN_COUNT *****
    ! ***************************************
    implicit none
    real(R8) ::  rang

    RN_SEED  = iand( iand( RN_MULT*RN_SEED, RN_MASK) + RN_ADD,  RN_MASK)
    rang     = RN_SEED * RN_NORM
    RN_COUNT = RN_COUNT + 1

    return
  end function rang

  !---------------------------------------------------------------------

  function RN_skip_ahead( seed, skip )
    ! advance the seed "skip" RNs:   seed*RN_MULT^n mod RN_MOD
    implicit none
    integer(I8) :: RN_skip_ahead
    integer(I8), intent(in)  :: seed, skip
    integer(I8) :: nskip, gen, g, inc, c, gp, rn, seed_old

    seed_old = seed
    ! add period till nskip>0
    nskip = skip
    do while( nskip<0_I8 )
      if( RN_PERIOD>0_I8 ) then
        nskip = nskip + RN_PERIOD
      else
        nskip = nskip + RN_MASK
        nskip = nskip + 1_I8
      endif
    enddo

    ! get gen=RN_MULT^n,  in log2(n) ops, not n ops !
    nskip = iand( nskip, RN_MASK )
    gen   = 1
    g     = RN_MULT
    inc   = 0
    c     = RN_ADD
    do while( nskip>0_I8 )
      if( btest(nskip,0) )  then
```

```
      gen = iand( gen*g, RN_MASK )
      inc = iand( inc*g, RN_MASK )
      inc = iand( inc+c, RN_MASK )
    endif
    gp    = iand( g+1,  RN_MASK )
    g     = iand( g*g,  RN_MASK )
    c     = iand( gp*c, RN_MASK )
    nskip = ishft( nskip, -1 )
  enddo
  rn = iand( gen*seed_old, RN_MASK )
  rn = iand( rn + inc, RN_MASK )
  RN_skip_ahead = rn
  return
end function RN_skip_ahead

!---------------------------------------------------------------------

subroutine RN_init_problem( new_standard_gen, new_seed, &
  &                         new_stride, new_part1,  print_info )
  ! * initialize MCNP random number parameters for problem,
  !   based on user input.  This routine should be called
  !   only from the main thread, if OMP threading is being used.
  !
  ! * for initial & continue runs, these args should be set:
  !     new_standard_gen - index of built-in standard RN generator,
  !                        from RAND gen=   (or dbcn(14)
  !     new_seed   - from RAND seed=        (or dbcn(1))
  !     output     - logical, print RN seed & mult if true
  !
  !     new_stride - from RAND stride=      (or dbcn(13))
  !     new_part1  - from RAND hist=        (or dbcn(8))
  !
  ! * for continue runs only, these should also be set:
  !     new_count_total   - from "rnr"   at end of previous run
  !     new_count_stride  - from nrnh(1) at end of previous run
  !     new_count_max     - from nrnh(2) at end of previous run
  !     new_count_max_nps - from nrnh(3) at end of previous run
  !
  ! * check on size of long-ints & long-int arithmetic
  ! * check the multiplier
  ! * advance the base seed for the problem
  ! * set the initial particle seed
  ! * initialize the counters for RN stats
  implicit none
  integer,     intent(in) :: new_standard_gen
  integer(I8), intent(in) :: new_seed
  integer(I8), intent(in) :: new_stride
  integer(I8), intent(in) :: new_part1
  integer,     intent(in) :: print_info
  character(len=20) :: printseed
  integer(I8)       ::  itemp1, itemp2, itemp3, itemp4

  if( new_standard_gen<1 .or. new_standard_gen>n_RN_GEN ) then
    call expire( 0, 'RN_init_problem', &
      & ' ***** ERROR: illegal index for built-in RN generator')
  endif

  ! set defaults, override if input supplied: seed, mult, stride
  RN_INDEX   = new_standard_gen
  RN_MULT    = standard_generator(RN_INDEX)%mult
  RN_ADD     = standard_generator(RN_INDEX)%add
  RN_STRIDE  = standard_generator(RN_INDEX)%stride
  RN_SEED0   = standard_generator(RN_INDEX)%initseed
  RN_BITS    = standard_generator(RN_INDEX)%log2mod
  RN_MOD     = ishft( 1_I8,       RN_BITS )
  RN_MASK    = ishft( not(0_I8),  RN_BITS-64 )
  RN_NORM    = 2._R8**(-RN_BITS)
  if( RN_ADD==0_I8) then
    RN_PERIOD  = ishft( 1_I8, RN_BITS-2 )
  else
    RN_PERIOD  = ishft( 1_I8, RN_BITS )
  endif
  if( new_seed>0_I8 ) then
    RN_SEED0  = new_seed
  endif
  if( new_stride>0_I8 ) then
    RN_STRIDE = new_stride
  endif
  RN_COUNT_TOTAL   = 0
  RN_COUNT_STRIDE  = 0
  RN_COUNT_MAX     = 0
  RN_COUNT_MAX_NPS = 0

  if( print_info /= 0 ) then
    write(printseed,'(i20)') RN_SEED0
    write( iuo,1) RN_INDEX, RN_SEED0, RN_MULT, RN_ADD, RN_BITS, RN_STRIDE
    write(jtty,2) RN_INDEX, adjustl(printseed)
1   format( &
      & /,' ************************************************', &
      & /,' * Random Number Generator  = ',i20,             ' *', &
      & /,' * Random Number Seed       = ',i20,             ' *', &
      & /,' * Random Number Multiplier = ',i20,             ' *', &
      & /,' * Random Number Adder      = ',i20,             ' *', &
      & /,' * Random Number Bits Used  = ',i20,             ' *', &
      & /,' * Random Number Stride     = ',i20,             ' *', &
      & /,' ************************************************',/)
2   format(' comment. using random number generator ',i2, &
      &    ', initial seed = ',a20)
  endif

  ! double-check on number of bits in a long int
  if( bit_size(RN_SEED)<64 ) then
    call expire( 0, 'RN_init_problem', &
      & ' ***** ERROR: <64 bits in long-int, can-t generate RN-s')
  endif
  itemp1 = 5_I8**25
  itemp2 = 5_I8**19
  itemp3 = ishft(2_I8**62-1_I8,1) + 1_I8
  itemp4 = itemp1*itemp2
```

```
    if( iand(itemp4,itemp3)/=8443747864978395601_I8 ) then
      call expire( 0, 'RN_init_problem', &
        & ' ***** ERROR: can-t do 64-bit integer ops for RN-s')
    endif

    if( new_part1>1_I8 ) then
      ! advance the problem seed to that for part1
      RN_SEED0 = RN_skip_ahead( RN_SEED0, (new_part1-1_I8)*RN_STRIDE )
      itemp1   = RN_skip_ahead( RN_SEED0, RN_STRIDE )
      if( print_info /= 0 ) then
        write(printseed,'(i20)') itemp1
        write( iuo,3) new_part1,  RN_SEED0, itemp1
        write(jtty,4) new_part1,  adjustl(printseed)
3       format( &
          & /,' ************************************************', &
          & /,' * Random Number Seed will be advanced to that for *', &
          & /,' * previous particle number = ',i20,             ' *', &
          & /,' * New RN Seed for problem  = ',i20,             ' *', &
          & /,' * Next Random Number Seed  = ',i20,             ' *', &
          & /,' ************************************************',/)
4       format(' comment. advancing random number to particle ',i12, &
          &    ', initial seed = ',a20)
      endif
    endif

    ! set the initial particle seed
    RN_SEED  = RN_SEED0
    RN_COUNT = 0
    RN_NPS   = 0

    return
  end subroutine RN_init_problem

  !---------------------------------------------------------------------

  subroutine RN_init_particle( nps )
    ! initialize MCNP random number parameters for particle "nps"
    !
    !     * generate a new particle seed from the base seed
    !       & particle index
    !     * set the RN count to zero
    implicit none
    integer(I8), intent(in) :: nps

    RN_SEED  = RN_skip_ahead( RN_SEED0, nps*RN_STRIDE )
    RN_COUNT = 0
    RN_NPS   = nps

    return
  end subroutine RN_init_particle

  !---------------------------------------------------------------------

  subroutine RN_set(  key,  value )
    implicit none
    character(len=*), intent(in) :: key
    integer(I8),      intent(in) :: value
    character(len=20) :: printseed
    integer(I8) :: itemp1

    if( key == "stride"        ) then
      if( value>0_I8 ) then
        RN_STRIDE        = value
      endif
    endif
    if( key == "count_total"   )  RN_COUNT_TOTAL   = value
    if( key == "count_stride"  )  RN_COUNT_STRIDE  = value
    if( key == "count_max"     )  RN_COUNT_MAX     = value
    if( key == "count_max_nps" )  RN_COUNT_MAX_NPS = value
    if( key == "seed"          )  then
      if( value>0_I8 ) then
        RN_SEED0 = value
        RN_SEED  = RN_SEED0
        RN_COUNT = 0
        RN_NPS   = 0
      endif
    endif
    if( key == "part1" ) then
      if( value>1_I8 ) then
        ! advance the problem seed to that for part1
        RN_SEED0 = RN_skip_ahead( RN_SEED0, (value-1_I8)*RN_STRIDE )
        itemp1   = RN_skip_ahead( RN_SEED0, RN_STRIDE )
        write(printseed,'(i20)') itemp1
        write( iuo,3) value,  RN_SEED0, itemp1
        write(jtty,4) value,  adjustl(printseed)
3       format( &
          & /,' ************************************************', &
          & /,' * Random Number Seed will be advanced to that for *', &
          & /,' * previous particle number = ',i20,             ' *', &
          & /,' * New RN Seed for problem  = ',i20,             ' *', &
          & /,' * Next Random Number Seed  = ',i20,             ' *', &
          & /,' ************************************************',/)
4       format(' comment. advancing random number to particle ',i12, &
          &    ', initial seed = ',a20)
        RN_SEED  = RN_SEED0
        RN_COUNT = 0
        RN_NPS   = 0
      endif
    endif
    return
  end subroutine RN_set

  !---------------------------------------------------------------------

  function RN_query( key )
    implicit none
    integer(I8)                 :: RN_query
    character(len=*), intent(in) :: key
    RN_query = 0_I8
```

```
  if( key == "seed"          )  RN_query = RN_SEED
  if( key == "stride"        )  RN_query = RN_STRIDE
  if( key == "mult"          )  RN_query = RN_MULT
  if( key == "add"           )  RN_query = RN_ADD
  if( key == "count"         )  RN_query = RN_COUNT
  if( key == "period"        )  RN_query = RN_PERIOD
  if( key == "count_total"   )  RN_query = RN_COUNT_TOTAL
  if( key == "count_stride"  )  RN_query = RN_COUNT_STRIDE
  if( key == "count_max"     )  RN_query = RN_COUNT_MAX
  if( key == "count_max_nps" )  RN_query = RN_COUNT_MAX_NPS
  if( key == "first"         )  RN_query = RN_SEED0
  return
end function RN_query
!---------------------------------------------------------------------

function RN_query_first( nps )
  implicit none
  integer(I8)                 :: RN_query_first
  integer(I8),      intent(in) :: nps
  RN_query_first = RN_skip_ahead( RN_SEED0, nps*RN_STRIDE )
  return
end function RN_query_first

!---------------------------------------------------------------------

subroutine RN_update_stats()
  ! update overall RN count info
  implicit none

  !$OMP CRITICAL (RN_STATS)

  RN_COUNT_TOTAL = RN_COUNT_TOTAL + RN_COUNT

  if( RN_COUNT>RN_COUNT_MAX ) then
    RN_COUNT_MAX     = RN_COUNT
    RN_COUNT_MAX_NPS = RN_NPS
  endif

  if( RN_COUNT>RN_STRIDE ) then
    RN_COUNT_STRIDE = RN_COUNT_STRIDE + 1
  endif

  !$OMP END CRITICAL (RN_STATS)

  RN_COUNT = 0
  RN_NPS   = 0

  return
end subroutine RN_update_stats

!---------------------------------------------------------------------

subroutine expire( i, c1, c2 )
  integer,          intent(in) :: i
  character(len=*), intent(in) :: c1, c2
  write(*,*) ' ********** error: ',c1
  write(*,*) ' ********** error: ',c2
  stop '**error**'
end subroutine expire

!---------------------------------------------------------------------
!#####################################################################
!#
!#  Unit tests
!#
!#####################################################################

subroutine RN_test_basic( new_gen )
  ! test routine for basic random number generator
  implicit none
  integer, intent(in) :: new_gen
  real(R8)    :: s
  integer(I8) :: seeds(10)
  integer     :: i, j

  write(jtty,"(/,a)")  " ***** random number - basic test *****"

  ! set the seed
  call RN_init_problem( new_gen, 1_I8, 0_I8, 0_I8, 1 )

  ! get the first 5 seeds, then skip a few, get 5 more - directly
  s = 0.0_R8
  do  i = 1,5
    s = s + rang()
    seeds(i) = RN_query( "seed" )
  enddo
  do  i = 6,123455
    s = s + rang()
  enddo
  do  i = 6,10
    s = s + rang()
    seeds(i) = RN_query( "seed" )
  enddo

  ! compare
  do  i = 1,10
    j = i
    if( i>5  ) j = i + 123450
    write(jtty,"(1x,i6,a,i20,a,i20)") &
      &  j, "  reference: ", RN_CHECK(i,new_gen), "  computed: ", seeds(i)
    if( seeds(i)/=RN_CHECK(i,new_gen) ) then
      write(jtty,"(a)")  " ***** basic_test of RN generator failed:"
    endif
  enddo
  return
end subroutine RN_test_basic

!---------------------------------------------------------------------
```

```
  subroutine RN_test_skip( new_gen )
    ! test routine for basic random number generation & skip-ahead
    implicit none
    integer, intent(in) :: new_gen
    integer(I8) :: seeds(10)
    integer     :: i, j

    ! set the seed
    call RN_init_problem( new_gen, 1_I8, 0_I8, 0_I8, 0 )

    ! use the skip-ahead function to get first 5 seeds, then 5 more
    do i = 1,10
      j = i
      if( i>5 )  j = i + 123450
      seeds(i) = RN_skip_ahead( 1_I8, int(j,I8) )
    enddo

    ! compare
    write(jtty,"(/,a)")  " ***** random number - skip test *****"
    do i = 1,10
      j = i
      if( i>5  ) j = i + 123450
      write(jtty,"(1x,i6,a,i20,a,i20)") &
        &  j, "  reference: ", RN_CHECK(i,new_gen),  "  computed: ", seeds(i)
      if( seeds(i)/=RN_CHECK(i,new_gen) ) then
        write(jtty,"(a)")  " ***** skip_test of RN generator failed:"
      endif
    enddo
    return
  end subroutine RN_test_skip

  !---------------------------------------------------------------------

  subroutine RN_test_mixed( new_gen )
    ! test routine -- print RN's 1-5 & 123456-123460,
    !                 with reference vals
    implicit none
    integer, intent(in) :: new_gen
    integer(I8) :: r
    integer     :: i, j

    write(jtty,"(/,a)")  " ***** random number - mixed test *****"
    ! set the seed & set the stride to 1
    call RN_init_problem( new_gen, 1_I8, 1_I8, 0_I8, 0 )

    write(jtty,"(a,i20,z20)") " RN_MULT   = ", RN_MULT, RN_MULT
    write(jtty,"(a,i20,z20)") " RN_ADD    = ", RN_ADD,  RN_ADD
    write(jtty,"(a,i20,z20)") " RN_MOD    = ", RN_MOD,  RN_MOD
    write(jtty,"(a,i20,z20)") " RN_MASK   = ", RN_MASK, RN_MASK
    write(jtty,"(a,i20)")     " RN_BITS   = ", RN_BITS
    write(jtty,"(a,i20)")     " RN_PERIOD = ", RN_PERIOD
    write(jtty,"(a,es20.14)") " RN_NORM   = ", RN_NORM
    write(jtty,"(a)")  " "
    do i = 1,10
      j = i
      if( i>5  ) j = i + 123450
      call RN_init_particle( int(j,I8) )
      r = RN_query( "seed" )
      write(jtty,"(1x,i6,a,i20,a,i20)") &
        &  j, "  reference: ", RN_CHECK(i,new_gen),"  computed: ", r
      if( r/=RN_CHECK(i,new_gen) ) then
        write(jtty,"(a)")  " ***** mixed test of RN generator failed:"
      endif
    enddo
    return
  end subroutine RN_test_mixed

  !---------------------------------------------------------------------
end module mcnp_random
```

Sample Main Program for Testing

Fortran Coding

```
Program rn_test

  use mcnp_random

  implicit none
  integer, parameter :: I8 = selected_int_kind(18)
  integer, parameter :: R8 = selected_real_kind(15,307)
  integer, parameter :: ntry = 10000000
  integer  :: i, j
  real(R8) :: total, t1,t2, ave

  !-------------------------------------------------------------

  ! run the 3 test routines for each of the
  !   7 mcnp random number generators
  do i=1,7
    call RN_test_basic( i )
    call RN_test_skip( i )
    call RN_test_mixed( i )

    call cpu_time( t1 )
    do j=1,ntry
      ave = rang()
    enddo
    call cpu_time( t2 )
    write(*,1)  (t2-t1)*1.d9/real(ntry,R8)
1   format(/,'average time per RN = ',f6.2,' nanosec',/)

  enddo

  !-------------------------------------------------------------

  ! typical usage in a Monte Carlo code
  !   use generator 2, seed=1234567, default stride,
  !   default first particle, print info
  call RN_init_problem( 2, 1234567_I8, 0_I8, 0_I8, 1 )

  ! run particle histories
  total = 0
  do i=1,ntry

    ! advance to the seed for this history
    call RN_init_particle( int(i,I8) )

    ! do something that uses random numbers
    t1 = rang()
    t2 = rang()
    if( t1**2+t2**2 < 1.0_R8 ) then
      ! tally the "hits"
      total = total + 1
    endif
  enddo

  ! done with histories, get final results
  write(*,2) ntry, total*4./real(ntry,R8)
2 format(/,"After ",i8," trials, pi ~ ",f10.5 )

  !-------------------------------------------------------------


end program rn_test
```

Random Number Generator

C Coding

```
//=============================================================================
//
// C version of MCNP5 routines for random number generation
//
//   - These routine require the use of 'unsigned long long' integers,
//     as specified in the ANSI-C99 standard - 64-bit integers.
//
//   - These routines were developed & tested with the gcc and
//     g++ compiler. No special options are needed to compile & test.
//
//   - For other C/C++ compilers, some tweaking may be needed.
//     Be sure to run & examine the tests.
//
//   - NOTE: These routines are not thread-safe.
//           (OpenMP threading will be coming soon.)
//
//   - To mix these C routines with Fortran-90 compiled
//     with the Absoft compiler, use these options when
//     compiling the fortran:
//        f90 -YEXT_NAMES=LCS -YEXT_SFX=_ -YCFRL=1  ...
//
//   - To mix these C routines with Fortran-90 compiled
//     with the g95 compiler, use these options when
//     compiling the fortran:
//        g95 -funderscoring -fno-second-underscore ...
//
//   - To mix these C routines with Fortran-90 compiled
//     with the IBM xlf compiler, use these options when
//     compiling the fortran:
//        xlf -qextname ...
//
//
// Author:  FB Brown, 2005-12-02
//=============================================================================

//=============================================================================
// Description:
//  mcnp_random.F90 -- random number generation routines
//=============================================================================
//  This module contains:
//
//   * Constants for the RN generator, including initial RN seed for the
//     problem & the current RN seed
//
//   * MCNP interface routines:
//     - random number function:           rang()
//     - RN initialization for problem:    RN_init_problem
//     - RN initialization for particle:   RN_init_particle
//     - get info on RN parameters:        RN_query
//     - get RN seed for n-th history:     RN_query_first
//     - set new RN parameters:            RN_set
//     - skip-ahead in the RN sequence:    RN_skip_ahead
//     - Unit tests:        RN_test_basic, RN_test_skip, RN_test_mixed
//
//   * For interfacing with the rest of MCNP, arguments to/from these
//     routines will have types of I8 or I4.
//     Any args which are to hold random seeds, multipliers,
//     skip-distance will be type I8, so that 63 bits can be held without
//     truncation.
//
// Revisions:
// * 10-04-2001 - F Brown, initial mcnp version
// * 06-06-2002 - F Brown, mods for extended generators
// * 12-21-2004 - F Brown, added 3 of LeCuyer's 63-bit mult. RNGs
// * 01-29-2005 - J Sweezy, Modify to use mcnp modules prior to automatic
//                io unit numbers.
// * 12-02-2005 - F Brown, mods for consistency with C version
//=============================================================================

#ifdef __cplusplus
extern "C" {
#endif

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

// function names to match what g95 expects
#define  rang               rang_
#define  RN_init_problem    rn_init_problem_
#define  RN_init_particle   rn_init_particle_
#define  RN_set             rn_set_
#define  RN_query           rn_query_
#define  RN_query_first     rn_query_first_
#define  RN_update_stats    rn_update_stats_
#define  RN_test_basic      rn_test_basic_
#define  RN_test_skip       rn_test_skip_
#define  RN_test_mixed      rn_test_mixed_

  //-----------------------------------------------------
  // Kinds for LONG INTEGERS (64-bit) & REAL*8 (64-bit)
  //-----------------------------------------------------
#define     LONG  long long
#define    ULONG  unsigned long long
#define     REAL  double

  //-------------------------------------
  // Public interface for functions
  //-------------------------------------
  REAL   rang(void);
  ULONG  RN_skip_ahead( ULONG* seed, LONG* nskip );
  void   RN_init_problem( int*   new_standard_gen,
                          ULONG* new_seed,
                          ULONG* new_stride,
                          ULONG* new_part1,
                          int*   print_info );
  void   RN_init_particle( ULONG* nps );
  void   RN_set( char* key, ULONG* value );
```

```
  ULONG  RN_query( char* key );
  ULONG  RN_query_first( ULONG* nps );
  void   RN_update_stats( void );
  void   RN_test_basic( int* new_gen );
  void   RN_test_skip(  int* new_gen );
  void   RN_test_mixed( int* new_gen );

  //---------------------------------------
  // Constants for standard RN generators
  //---------------------------------------
  typedef struct {
    int          index;
    ULONG        mult;       // generator (multiplier)
    ULONG        add;        // additive constant
    int          log2mod;    // log2 of modulus, must be <64
    ULONG        stride;     // stride for particle skip-ahead
    ULONG        initseed;   // default seed for problem
    char         name[9];
  } RN_GEN;

  //  parameters for standard generators
  static const int      n_RN_GEN = 7;
  static const RN_GEN   standard_generator[7] = {
    1,      19073486328125ULL, 0ULL, 48, 152917ULL, 19073486328125ULL, "mcnp std",
    2, 92197414264999714 45ULL, 1ULL, 63, 152917ULL,              1ULL, "LEcuyer1",
    3, 28061969105067807 09ULL, 1ULL, 63, 152917ULL,              1ULL, "LEcuyer2",
    4, 32492868495230128 05ULL, 1ULL, 63, 152917ULL,              1ULL, "LEcuyer3",
    5, 35124019650235035 17ULL, 0ULL, 63, 152917ULL,              1ULL, "LEcuyer4",
    6, 24448053531876724 69ULL, 0ULL, 63, 152917ULL,              1ULL, "LEcuyer5",
    7, 19875910588293107 33ULL, 0ULL, 63, 152917ULL,              1ULL, "LEcuyer6"
  };

  //---------------------------------------------------------------------
  //   * Linear multiplicative congruential RN algorithm:
  //
  //            RN_SEED = RN_SEED*RN_MULT + RN_ADD  mod RN_MOD
  //
  //   * Default values listed below will be used, unless overridden
  //---------------------------------------------------------------------
  static int    RN_INDEX   = 1;
  static ULONG  RN_MULT    =  19073486328125ULL;
  static ULONG  RN_ADD     =               0ULL;
  static int    RN_BITS    = 48;
  static ULONG  RN_STRIDE  =          152917ULL;
  static ULONG  RN_SEED0   =  19073486328125ULL;
  static ULONG  RN_MOD     = 281474976710656ULL;
  static ULONG  RN_MASK    = 281474976710655ULL;
  static ULONG  RN_PERIOD  =  70368744177664ULL;
  static REAL   RN_NORM    = 1.0/281474976710656.;

  //--------------------------------------
  // Private data for a single particle
  //--------------------------------------
  static ULONG  RN_SEED    = 19073486328125ULL; // current seed
  static ULONG  RN_COUNT   = 0;                 // current counter
  static ULONG  RN_NPS     = 0;            // current particle number

  //-----------------------------------------
  // Shared data, to collect info on RN usage
  //-----------------------------------------
  static ULONG  RN_COUNT_TOTAL   = 0;     // total RN count all particles
  static ULONG  RN_COUNT_STRIDE  = 0;     // count for stride exceeded
  static ULONG  RN_COUNT_MAX     = 0;     // max RN count all particles
  static ULONG  RN_COUNT_MAX_NPS = 0;     // part index for max count

  //-------------------------------------------------------------------------
  // reference data:  seeds for case of init.seed = 1,
  //                  seed numbers for index 1-5, 123456-123460
  //-------------------------------------------------------------------------
  static const ULONG  RN_CHECK[7][10] = {
    // ***** 1 ***** mcnp standard gen *****
         19073486328125ULL,      29763723208841ULL,     187205367447973ULL,
        131230026111313ULL,     264374031214925ULL,     260251000190209ULL,
        106001385730621ULL,     232883458246025ULL,      97934850615973ULL,
        163056893025873ULL,
    // ***** 2 *****
    92197414264999714 46ULL,  66676480825570737 5ULL, 49351092084535409 24ULL,
    70768150377770238 53ULL, 55940704870829644 34ULL, 70694841529215945 61ULL,
    84244857246319829 02ULL,   19322398608391599ULL, 86397596919696732 12ULL,
    81813158193752274 37ULL,
    // ***** 3 *****
    28061969105067807 10ULL, 69243084589659416 31ULL, 70938335713869320 60ULL,
    41335606382743358 21ULL,  67865306925035293 0ULL, 64319422878132389 77ULL,
    44893102523235460 86ULL, 20018633569682475 59ULL,  96658179812550274 8ULL,
    19841131344314718 85ULL,
    // ***** 4 *****
    32492868495230128 06ULL, 43661926262849997 75ULL, 43349672082292390 68ULL,
    63866148285773502 85ULL, 66514540041130871 06ULL, 27327603903164141 45ULL,
    20677276516892048 70ULL, 27078402035032133 43ULL, 60091422463024852 12ULL,
    66789169556295217 41ULL,
    // ***** 5 *****
    35124019650235035 17ULL, 54617698694010327 77ULL, 14681848057229375 41ULL,
    51608720623726522 41ULL, 66376477581749432 77ULL,  79420625747589043 3ULL,
    46621538968352679 97ULL, 60752012705010394 33ULL,  88969436666203181 3ULL,
    72992999625455292 97ULL,
    // ***** 6 *****
    24448053531876724 69ULL,  31661651530779871 3ULL, 48058194854536900 29ULL,
    70735297085961353 45ULL, 37279025662061447 73ULL, 11420150437491617 29ULL,
    86324792196925707 73ULL, 27954535306301654 33ULL, 56789730886366790 85ULL,
    34910414233960613 61ULL,
    // ***** 7 *****
    19875910588293107 33ULL, 50328894490418541 21ULL, 44236122082941095 89ULL,
    30209859226918450 09ULL, 51598927471383678 37ULL, 83876421079835425 29ULL,
    84881789960959344 77ULL,  70854088138913373 7ULL, 36431608833635324 37ULL,
    47529765164707728 81ULL
  };

//-------------------------------------------------------------------------
//
```

```
  REAL rang( void ) {
    // MCNP random number generator
    //
    // ***************************************
    // ***** modifies RN_SEED & RN_COUNT *****
    // ***************************************
   RN_SEED   = (RN_MULT*RN_SEED + RN_ADD) & RN_MASK;
        RN_COUNT += 1;

        return  (REAL) (RN_SEED*RN_NORM);
  }

//------------------------------------------------------------------------
//
  ULONG   RN_skip_ahead( ULONG* s, LONG* n ) {
    //  skip ahead n RNs:   RN_SEED*RN_MULT^n mod RN_MOD
    ULONG seed  = *s;
    LONG  nskip = *n;
    while( nskip<0 )  nskip += RN_PERIOD;      // add period till >0
    nskip = nskip & RN_MASK;                   // mod RN_MOD
    ULONG  gen=1,  g=RN_MULT, gp, inc=0, c=RN_ADD, rn;
    // get gen=RN_MULT^n,  in log2(n) ops, not n ops !
    for( ; nskip; nskip>>=1 ) {
      if( nskip&1 ) {
        gen =  gen*g      & RN_MASK;
        inc = (inc*g + c) & RN_MASK;
      }
      c  = g*c+c & RN_MASK;
      g  = g*g   & RN_MASK;
    }
    rn = (gen*seed + inc) & RN_MASK;

    return (ULONG) rn;
  }
//------------------------------------------------------------------------
//
  void RN_init_problem( int*   new_standard_gen,
                        ULONG* new_seed,
                        ULONG* new_stride,
                        ULONG* new_part1,
                        int*   print_info ) {
    // * initialize MCNP random number parameters for problem,
    //   based on user input.  This routine should be called
    //   only from the main thread, if OMP threading is being used.
    //
    // * for initial & continue runs, these args should be set:
    //     new_standard_gen - index of built-in standard RN generator,
    //                        from RAND gen=   (or dbcn(14)
    //     new_seed   - from RAND seed=        (or dbcn(1))
    //     output     - logical, print RN seed & mult if true
    //
    //     new_stride - from RAND stride=      (or dbcn(13))
    //     new_part1  - from RAND hist=        (or dbcn(8))
    //
    // * for continue runs only, these should also be set:
    //     new_count_total   - from "rnr"   at end of previous run
    //     new_count_stride  - from nrnh(1) at end of previous run
    //     new_count_max     - from nrnh(2) at end of previous run
    //     new_count_max_nps - from nrnh(3) at end of previous run
    //
    // * check on size of long-ints & long-int arithmetic
    // * check the multiplier
    // * advance the base seed for the problem
    // * set the initial particle seed
    // * initialize the counters for RN stats

    if( *new_standard_gen<1 || *new_standard_gen>n_RN_GEN ) {
      printf("***** error: bad new_standard_gen\n");
      exit(1);
    }
    // set defaults, override if input supplied: seed, mult, stride
    RN_INDEX   = *new_standard_gen;
    RN_MULT    = standard_generator[RN_INDEX-1].mult;
    RN_ADD     = standard_generator[RN_INDEX-1].add;
    RN_STRIDE  = standard_generator[RN_INDEX-1].stride;
    RN_SEED0   = standard_generator[RN_INDEX-1].initseed;
    RN_BITS    = standard_generator[RN_INDEX-1].log2mod;
    RN_MOD     = 1ULL<<RN_BITS;
    RN_MASK    = (~0ULL) >> (64-RN_BITS);
    RN_NORM    = 1./(REAL)RN_MOD;
    RN_PERIOD  = (RN_ADD==0ULL)?  1ULL<<(RN_BITS-2) : 1ULL<<RN_BITS;
    if( *new_seed>0 ) {
      RN_SEED0 = *new_seed;
    }
    if( *new_stride>0 ) {
      RN_STRIDE = *new_stride;
    }
    RN_COUNT_TOTAL   = 0;
    RN_COUNT_STRIDE  = 0;
    RN_COUNT_MAX     = 0;
    RN_COUNT_MAX_NPS = 0;
    if( *print_info ) {
      printf( "\n%s\n%s%20d%s\n%s%20llu%s\n%s%20llu%s\n"
              "%s%20llu%s\n%s%20d%s\n%s%20llu%s\n%s\n\n",
          " ************************************************",
          " * Random Number Generator  = ", RN_INDEX,       " *",
          " * Random Number Seed       = ", RN_SEED0,       " *",
          " * Random Number Multiplier = ", RN_MULT,        " *",
          " * Random Number Adder      = ", RN_ADD,         " *",
          " * Random Number Bits Used  = ", RN_BITS,        " *",
          " * Random Number Stride     = ", RN_STRIDE,      " *",
          " ************************************************");
      printf(" comment. using random number generator %d,"
             " initial seed = %llu\n", RN_INDEX, RN_SEED0 );
    }
    // double-check on number of bits in a long long unsigned int
    if( sizeof(RN_SEED)<8 ) {
      printf("***** RN_init_problem ERROR:"
             " <64 bits in long-int, can-t generate RN-s\n");
```

```
      exit(1);
    }
    ULONG itemp1, itemp2, itemp3, itemp4;
    itemp1 =  (ULONG) pow(5.e0,25);
    itemp2 =  (ULONG) pow(5.e0,19);
    itemp3 = ((ULONG) pow(2.e0,62-1))<<1 + 1;
    itemp4 = itemp1*itemp2;
    if( itemp4&itemp3 != 844374786497839560lULL ) {
      printf("***** RN_init_problem ERROR:"
             " can-t do 64-bit integer ops for RN-s\n");
      exit(1);
    }
    if( *new_part1>1 ) {
      LONG n = (*new_part1-1)*RN_STRIDE;
      LONG k =                RN_STRIDE;
      RN_SEED0     = RN_skip_ahead( &RN_SEED0, &n );
      ULONG itemp1 = RN_skip_ahead( &RN_SEED0, &k );
      if( *print_info ) {
        printf("\n%s\n%s\n%s%20llu%s\n%s%20llu%s\n%s%20llu%s\n%s\n\n",
          " ***************************************************",
          " * Random Number Seed will be advanced to that for *",
          " * previous particle number = ", *new_part1,     " *",
          " * New RN Seed for problem  = ", RN_SEED0,       " *",
          " * Next Random Number Seed  = ", itemp1,         " *",
          " ***************************************************");
        printf(" comment. advancing random number to particle %20llu,"
               " initial seed = %20llu\n", *new_part1, RN_SEED0 );
      }
    }
    // set the initial particle seed
    RN_SEED    = RN_SEED0;
    RN_COUNT   = 0;
    RN_NPS     = 0;
    return;
  }
//---------------------------------------------------------------------
//
  void RN_init_particle( ULONG* nps ) {
    // initialize MCNP random number parameters for particle "nps"
    //
    //     * generate a new particle seed from the base seed
    //       & particle index
    //     * set the RN count to zero
        LONG  nskp = *nps * RN_STRIDE;
   RN_SEED  = RN_skip_ahead( &RN_SEED0, &nskp );
        RN_COUNT = 0;
        RN_NPS   = *nps;
  }
//---------------------------------------------------------------------
//
  void   RN_set( char* key, ULONG* value ) {
    if( !strcmp(key,"count_total")   ) { RN_COUNT_TOTAL   = *value; return; }
    if( !strcmp(key,"count_stride")  ) { RN_COUNT_STRIDE  = *value; return; }
    if( !strcmp(key,"count_max")     ) { RN_COUNT_MAX     = *value; return; }
    if( !strcmp(key,"count_max_nps") ) { RN_COUNT_MAX_NPS = *value; return; }
    if( !strcmp(key,"seed")          ) {
      RN_SEED0 = *value;
      RN_SEED  = RN_SEED0;
      RN_COUNT = 0;
      RN_NPS   = 0;
    }
    return;
  }
//---------------------------------------------------------------------
//
  ULONG   RN_query( char* key ) {
    if( !strcmp(key,"seed")          ) return  RN_SEED;
    if( !strcmp(key,"stride")        ) return  RN_STRIDE;
    if( !strcmp(key,"mult")          ) return  RN_MULT;
    if( !strcmp(key,"add")           ) return  RN_ADD;
    if( !strcmp(key,"count")         ) return  RN_COUNT;
    if( !strcmp(key,"period")        ) return  RN_PERIOD;
    if( !strcmp(key,"count_total")   ) return  RN_COUNT_TOTAL;
    if( !strcmp(key,"count_stride")  ) return  RN_COUNT_STRIDE;
    if( !strcmp(key,"count_max")     ) return  RN_COUNT_MAX;
    if( !strcmp(key,"count_max_nps") ) return  RN_COUNT_MAX_NPS;
    if( !strcmp(key,"first")         ) return  RN_SEED0;
    return 0ULL;
  }
//---------------------------------------------------------------------
//
  ULONG   RN_query_first( ULONG* nps ) {
    LONG n = *nps * RN_STRIDE;
    return  RN_skip_ahead( &RN_SEED0, &n);
  }
//---------------------------------------------------------------------
//
  void  RN_update_stats( void ) {
    // update overall RN count info

    RN_COUNT_TOTAL += RN_COUNT;

    if( RN_COUNT>RN_COUNT_MAX ) {
      RN_COUNT_MAX     = RN_COUNT;
      RN_COUNT_MAX_NPS = RN_NPS;
    }

    if( RN_COUNT>RN_STRIDE ) {
      RN_COUNT_STRIDE += 1;
    }

    RN_COUNT = 0;
    RN_NPS   = 0;

    return;
  }
//---------------------------------------------------------------------
//
  void RN_test_basic( int* new_gen ) {
```

```
    // test routine for basic random number generator
    //
    ULONG  seeds[10],  one=1ULL, z=0ULL;
    int i,j, k=1;
    double s = 0.0;

    printf("\n ***** random number - basic test *****\n");

    // set seed
    RN_init_problem( new_gen, &one, &z, &z, &k );

    // get the    5 seeds, then skip a few, get 5 more - directly
    for( i=0; i<5;      i++ ) { s += rang(); seeds[i] = RN_query("seed"); }
    for( i=5; i<123455; i++ ) { s += rang(); }
    for( i=5; i<10;     i++ ) { s += rang(); seeds[i] = RN_query("seed"); }

    // compare
    for( i=0; i<10; i++ ) {
      j = (i<5)? i+1 : i+123451;
      printf(" %6d  reference: %20llu  computed: %20llu\n",
               j, RN_CHECK[*new_gen-1][i], seeds[i] );
      if( seeds[i] != RN_CHECK[*new_gen-1][i] ) {
        printf(" ***** basic_test of RN generator failed\n");
      }
    }
  }
//---------------------------------------------------------------------
//
  void RN_test_skip( int* new_gen ) {
    // test routine for basic random number generation & skip-ahead
    //
    ULONG  seeds[10],  n=1ULL, z=0ULL;
    LONG j;
    int i, k=0;

    printf("\n ***** random number - skip test *****\n");

    // set the seed
    RN_init_problem( new_gen, &n,&z, &z, &k );

    // use the skp-ahead function to get the first 5 seeds, then 5 more
    for( i=0; i<10; i++ ) {
      j = (i<5)? i+1 : i+123451;
      seeds[i] = RN_skip_ahead( &n, &j );
    }

    // compare
    for( i=0; i<10; i++ ) {
      j = (i<5)? i+1 : i+123451;
      printf(" %6lld  reference: %20llu  computed: %20llu\n",
               j, RN_CHECK[*new_gen-1][i], seeds[i] );
      if( seeds[i] != RN_CHECK[*new_gen-1][i] ) {
        printf(" ***** skip_test of RN generator failed\n");
      }
    }
  }
//---------------------------------------------------------------------
//
  void RN_test_mixed( int* new_gen ) {
    // test routine -- print RN's 1-5 & 123456-123460, with reference vals
    ULONG  oldseed, s, n=0ULL, one=1ULL, j;
    int     i, z=0;

    printf("\n ***** random number - mixed test *****\n");

    // set the seed & set stride to 1
    RN_init_problem( new_gen, &one, &one, &n, &z );
    RN_set("stride", &one );

    printf(" RN_MULT   = %20llu\n", RN_MULT);
    printf(" RN_ADD    = %20llu\n", RN_ADD );
    printf(" RN_MOD    = %20llu\n", RN_MOD );
    printf(" RN_PERIOD = %20llu\n", RN_PERIOD);
    printf(" RN_MASK   = %20llu\n", RN_MASK);
    printf(" RN_STRIDE = %20llu\n", RN_STRIDE);
    printf(" RN_NORM   = %20.14e\n\n",   RN_NORM);
    for( i=0; i<10; i++ ) {
      rang();
      j = (i<5)? i+1 : i+123451;
      RN_init_particle( &j );
      s = RN_query("seed");
      printf(" %6llu  reference: %20llu  computed: %20llu\n",
             j, RN_CHECK[RN_INDEX-1][i], s );
      if( s!=RN_CHECK[RN_INDEX-1][i] ) {
        printf(" ***** mixed_test of RN generator failed\n");
      }
    }
  }
//---------------------------------------------------------------------
#ifdef __cplusplus
}
#endif
```

Sample Main Program for Testing

C Coding

```
#include <stdio.h>
#include <time.h>

#define RN_test_basic     rn_test_basic_
#define RN_test_skip      rn_test_skip_
#define RN_test_mixed     rn_test_mixed_
#define RN_init_problem   rn_init_problem_
#define RN_init_particle  rn_init_particle_
#define rang              rang_

double rang(void);

int main(  int argc, char** argv ) {

  long  ntry = 10000000;
  int  i, j;
  double  t1,t2, xxx, ave;
  //------------------------------------------------------------

  // run the 3 test routines for each of the
  //   7 mcnp random number generators
  for( i=1; i<8; i++ ) {
    RN_test_basic( &i );
    RN_test_skip( &i );
    RN_test_mixed( &i );

    t1 = clock();
    for( j=0; j<ntry; j++ ) {
      xxx = rang();
    }
    t2 = clock();
    ave = ((t2-t1)/CLOCKS_PER_SEC) * 1.e9 / ntry;
    printf("\naverage time per RN = %6.2f nanosec\n\n", ave );
  }

  //------------------------------------------------------------

  // typical usage in a Monte Carlo code
  //   use generator 2, seed=1234567, default stride,
  //   default first particle, print info
  int  gen=2,  prnt=1;
  unsigned long long  seed=1234567, zero=0;
  unsigned long long  nps;
  double total;

  RN_init_problem( &gen, &seed, &zero, &zero, &prnt );

  // run particle histories
  total = 0;
  for( nps=0; nps<ntry; nps++ ) {

    // advance to the seed for this history
    RN_init_particle( &nps );

    // do something that uses random numbers
    t1 = rang();
    t2 = rang();
    // tally the "hits"
    if( t1*t1+t2*t2 < 1.0 ) total += 1;
  }

  // done with histories, get final results
  double  pi = total*4./ntry;
  printf("\nAfter %d trials, pi ~ %10.5f\n", ntry, pi );

  //------------------------------------------------------------
}
```