Is the Standard Monte Carlo Power Iteration Approach the Wrong Approach?

Thomas E. Booth

LA-UR Draft May 29, 2012

Thomas E. Booth, Mail Stop A143, Los Alamos National Laboratory, Los Alamos, New Mexico 87545 USA

Abstract

The power iteration method is the standard Monte Carlo approach for obtaining the eigenfunctions of a nuclear system, but the power method sometimes converges very slowly. Most discussions give a mathematical reason for the slow convergence of the Monte Carlo power method using the same concepts and terminology as when the power method is applied to a deterministic problem.

This note first looks at why the convergence is slow from an intuitive Monte Carlo neutron perspective. Second, this note proposes building an eigenfunction intuitively in a cumulative (and noniterative) neutron by neutron manner that tends to better direct neutrons to where the neutrons need to be. Third, a very similar method for building the second eigenfunction is speculatively proposed.

1 Introduction

Most Monte Carlo transport codes (e.g. MCNP [1]) obtain the eigenfunction via a power iteration method, as explained below. Let $Q(P)$ be an arbitrary density of source neutrons at phase-space point $P$ and let $A$ be a transport operator. Let $S_i(P)$ and $k_i$ be the eigenfunctions and eigenvalues of $A$. Any arbitrary function $Q(P)$ can be written as a linear combination of the eigenfunctions

$$Q(P) = \sum_{i=1}^{\infty} a_i S_i(P)$$

where $a_i$ are constants.

The eigenvalue ($k_i$) and eigenfunction ($S_i(P)$) relation is

$$AS_i(P) = k_i S_i(P)$$

(2)
Applying the operator $A$ $n$ times using Eqs. 1 and 2 gives

$$A^n Q(P) = \sum_{i=1}^{\infty} a_i k_i^n S_i(P)$$  \hspace{1cm} (3)

If $|k_1| > |k_2| \geq |k_3| \geq |k_4| \cdots$ then as $n \to \infty$

$$A^n Q(P) = k_1^n \left( a_1 S_1(P) + \sum_{i=2}^{\infty} a_i \left( \frac{k_i}{k_1} \right)^n S_i(P) \right) \to k_1^n a_1 S_1(P)$$  \hspace{1cm} (4)

so that only the fundamental eigenfunction remains. (Normally after application of $A$ the eigenfunction estimate is normalized in some convenient way, but that is an unimportant detail here.)

In this paper, unless otherwise specified, it is assumed that

1. $A$ is everywhere real and nonnegative

2. All neutrons have positive weight

## 2 Intuitive Inefficiencies in the Monte Carlo Power Method

Note that if $Q(P)$ were the true eigenfunction ($Q(P) = S_1(P)$) then

$$k_1 = \frac{A S_1(P)}{S_1(P)}$$  \hspace{1cm} (5)

is satisfied at every point $P$ for which $S_1(P) \neq 0$. Note that this is a local rather than a global eigenvalue relationship.

In general $Q(P)$ is not the true eigenfunction and this can be expressed as

$$v(P) = \frac{A Q(P)}{Q(P)}$$  \hspace{1cm} (6)

where $v(P)$ is not constant. When the transport problem is continuous, define a global eigenvalue estimate

$$K_S = \frac{\int (AQ(P)) dP}{\int Q(P) dP}$$  \hspace{1cm} (7)

and when the transport problem is discrete, define a global eigenvalue estimate

$$K_S = \frac{\sum_i (AQ)_i}{\sum_j Q_j}$$  \hspace{1cm} (8)

Note that the only direct control we have on $v(P)$ is to adjust $Q(P)$; we cannot directly adjust $AQ(P)$ because this is a dependent quantity. So, if $v(P)$ is too large (say $v(P) > K_S$) we can attempt to reduce $v(P)$ by increasing the denominator of Eq. 6; that is, by increasing $Q(P)$. (Unless there is an unphysical $\delta$-function component of $A$, the probability of a neutron sourced in at $P$ producing a fission at $P$ is zero. For a large discrete system it is near zero. Thus, one can be pretty sure that when the denominator in Eq. 6 is increased by sourcing a neutron in at $P$, the numerator stays the same and $v(P)$
decreases.) On the other hand, if \( v(P) \) is too small (say \( v(P) < K_S \)), then it makes little sense to add a neutron at \( P \) because that only makes \( v(P) \) smaller. The power method ignores this fact and simply sources in new neutrons proportional to \( Q(P) \), whether or not it is counterproductive.

The standard Monte Carlo power method (e.g. in MCNP [1]) samples source neutrons proportional to \( Q(P) \). Thus, computational resources are expended proportional to \( Q(P) \), even when the current eigenvalue estimate at \( P \) is already too low. That is,

\[
v(P) = \frac{AQ(P)}{Q(P)} < K_S
\]

Note that adding another source neutron at \( P \) tends to make \( v(P) \) even smaller than the already too low value.

3 New Procedure on a Discrete Ten State Transport Problem

A new procedure that does not source neutrons counterproductively is described and illustrated with a discrete ten state transport problem. Let the transport operator \( A \) be the matrix with elements \( A_{ij} \):

\[
\begin{pmatrix}
0.9500000 & 0.0290000 & 0.0009700 & 0.0000260 & 0.0000084 & 0.0000003 & 0.0000001 & 0.0000001 & 0.0000001 \\
0.0300000 & 0.9300000 & 0.0300000 & 0.0008900 & 0.0003000 & 0.0000084 & 0.0000003 & 0.0000001 & 0.0000001 \\
0.0008700 & 0.0300000 & 0.8500000 & 0.0270000 & 0.0086000 & 0.0003000 & 0.0000092 & 0.0000001 & 0.0000001 \\
0.0002600 & 0.0008600 & 0.0280000 & 0.8900000 & 0.0310000 & 0.0093000 & 0.0002700 & 0.0000081 & 0.0000003 \\
0.0000094 & 0.0002600 & 0.0086000 & 0.3000000 & 0.9200000 & 0.0310000 & 0.0093000 & 0.0002600 & 0.0000081 \\
0.0000002 & 0.0000090 & 0.0002900 & 0.0089000 & 0.3000000 & 0.9300000 & 0.3000000 & 0.0009800 & 0.0002800 \\
0.0000001 & 0.0000003 & 0.0000085 & 0.0002700 & 0.0096000 & 0.0290000 & 0.8800000 & 0.0310000 & 0.0008700 \\
0.0000001 & 0.0000001 & 0.0000003 & 0.0000093 & 0.0002800 & 0.0086000 & 0.0280000 & 0.9100000 & 0.0280000 \\
0.0000001 & 0.0000001 & 0.0000001 & 0.0000003 & 0.0000087 & 0.0002700 & 0.0091000 & 0.2900000 & 0.9000000 \\
0.0000001 & 0.0000001 & 0.0000001 & 0.0000003 & 0.0000081 & 0.0002800 & 0.0084000 & 0.0710000 & 0.9000000
\end{pmatrix}
\]

Define the probability \( s_j \) that the neutron sourced into state \( j \) survives and reaches one of the ten states. For this particular matrix

\[
s_j = \sum_{i=1}^{10} A_{ij} < 1
\]

and the termination probability from state \( j \) is

\[
t_j = 1 - s_j = 1 - \sum_{i=1}^{10} A_{ij} > 0
\]

and so \( A_{ij} \) can be interpreted as the probability that a neutron sourced into state \( j \) produces 1 fission neutron in state \( i \). (For this problem the fission multiplicity is \( \nu = 1 \).) Call the termination state “state 0” for convenience.
This particular operator $A$ has a dominance ratio of 0.995 and eigenvalues:

$$
\begin{pmatrix}
0.974674 & 0.969624 & 0.955625 & 0.928092 & 0.917311 & 0.90804 & 0.876805 & 0.858909 & 0.844076 & 0.826844
\end{pmatrix}
$$

Let $N_t$ be the total number of neutrons in the entire calculation and $N$ be the running count used so far. Let $Q_j$ be the number of neutrons sourced into state $j$ and let $R_i$ be the number of fission neutrons produced in state $i$ via transport of all the source neutrons from all the states. Stated mathematically, ($T$ indicates transposing from a row vector to a column vector):

$$
R = (R_1, \ldots, R_{10})^T = AQ = A(Q_1, \ldots, Q_{10})^T
$$

or

$$
R_i = \sum_{j=1}^{10} A_{ij}Q_j
$$

(13)

The procedure proposed here for building the fundamental eigenfunction using $N_t$ neutrons in a noniterative and cumulative way is to calculate local eigenvalues and a global eigenvalue and then compare each of the local eigenvalues with the global eigenvalue. Neutrons are sourced into a state if and only if the local eigenvalue exceeds the global eigenvalue. Once all the states have been processed, new local eigenvalues and a global eigenvalue are computed and the process continues.

Algorithmically, this neutron by neutron “building brick” procedure can be expressed as

1. Initially $Q = 0$ and $R = 0$. One neutron is sourced into each of the ten states so that $Q_j = 1$ for $j = 1, \ldots, 10$.

   For each $j$, the fission state $i$ (or termination) is sampled from $A_{ij}$ and the resulting fission distribution updated; i.e.,
   
   $$
   R_i \leftarrow R_i + 1.
   $$

2. $j \leftarrow 0$

3. Calculate the global system eigenvalue estimate $K_S = \frac{\sum R_i}{\sum Q_i}$ and the individual eigenvalue estimates $v_j = \frac{R_j}{Q_j}$.

4. If $N = N_t$, go to 9.

5. update $j \leftarrow j + 1$ (i.e., check the next source state)

6. If $j \geq 10$ go to 2. (Once all the source states have been processed, the global and local eigenvalues need to be recalculated.)

7. If $v_j < K_S$ go to 5 (Compare the eigenvalue estimate in state $j$ with the global eigenvalue estimate. If the local estimate is too low, do not source a neutron into this state. Instead, proceed to the next state.)

8. Update

   $$
   N \leftarrow N + 1 \text{ (another neutron is being sourced in)}
   $$

   go to 1.
\( Q_j \leftarrow Q_j + 1 \) (another neutron is being sourced into state \( j \))

Sample for either the fission state \( i \) reached from state \( j \) with probability \( A_{ij} \) or the termination (state 0) with probability \( t_j \). Update the fission neutrons in state \( i \)

\[ R_i \leftarrow R_i + 1 \]

(Note that system has fission multiplicity \( \nu = 1 \) otherwise one updates \( R_i \leftarrow R_i + \nu \).)

If \( N = N_i \) go to 3, else go to 5

9. End calculation

(Note an alternative procedure would be to calculate \( K_S \) after each neutron. If computer time were not a factor, this is probably a slightly better convergence procedure until \( N \) gets large. On the other hand, calculating \( K_S \) after each neutron does take more time, and as \( N \) gets large there is very little advantage in convergence. Additionally, this alternative procedure is probably more problematical for parallel computing. This alternative procedure will not be discussed further.)

Figure 1 shows the rms convergence of the eigenvector to the true eigenvector as a function of the cumulative number of neutrons. The plot shows averages over 1000 runs and the associated error bars. Note that because the procedure is cumulative, the source can never entirely disappear in any region because there will always be at least one neutron that has been sourced into the region. Furthermore, time is not wasted on source neutrons that move the eigenfunction estimate away from the true eigenfunction.

4 Future Work: Continuous Problems and the Second Eigenfunction

Note that for continuous problems, the problem can be divided into continuous regions and the regionwise eigenvalues compared with the global eigenvalue. As before, neutrons are then sourced into a region if and only if the region’s eigenvalue is higher than the global eigenvalue. This leaves the question of where the neutron is sourced into the region. Preliminary work, not reported here, indicates that sampling the neutron’s source position from the current cumulative Monte Carlo estimate of \( AQ(P) = R(P) \) within the region is a good solution. Other solutions are possible as well, but they have not yet been tested even in a preliminary test.

Although it is not shown here, based on previous experience with estimating higher eigenfunctions, it is a good guess that the second (and higher) eigenfunction can be obtained by small modifications of the above procedure for the first eigenfunction. For the second eigenfunction, one uses negative as well as positive weight source neutrons. One computes global \( K_S \)’s for both negative and positive regions. That is both a \( K_{S-} \) and a \( K_{S+} \) corresponding to Eq. 8. With \( I(T) = 0 \)
Convergence to Eigenfunction

\[ y = 6.928 x^{(-0.5014)} \]

Figure 1: Convergence to Eigenfunction
when $T$ is false and $I(T) = 1$ when $T$ is true, one has

$$K_{S-} = \frac{\sum_i I(R_i < 0)R_i}{\sum_j I(R_j < 0)Q_j}$$  \hspace{1cm} (14)$$

$$K_{S+} = \frac{\sum_i I(R_i > 0)R_i}{\sum_j I(R_j > 0)Q_j}$$  \hspace{1cm} (15)$$

If $K_{S-} > K_{S+}$ then the procedure in section 3 is followed with positive weight neutrons for all states for which $I(R_j > 0)$.

Similarly, if $K_{S-} < K_{S+}$ then the procedure in section 3 is followed with negative weight neutrons for all states for which $I(R_j < 0)$.

References


5 Acknowledgement

Thanks to Roger Martz for making a number of useful suggestions to improve the clarity of this report.

6 Appendix - Source Code for Calculations

```fortran
program add
! ns-state criticality problem eigenvalue and vector
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
use mcnp_random, only : rn_init_problem
use mcnp_random, only : rang
use mcnp_random, only : rn_init_problem, rn_set
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
implicit real(selected_real_kind(15,307)) (a-h,o-z)
integer,parameter :: dknd = selected_real_kind(15,307)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

7
integer, parameter:: i8knd = selected_int_kind(18) ! 8-byte integer kind

integer(i8knd):: &
  & RN_seed_input,       & != user input, starting RN seed
  & RN_stride_input,    & != user input, RN stride
  & RN_hist_input      & != user input, start RN sequence with this history

integer::RN_gen_input

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

integer, parameter :: nstate=10

common/teb/p(1:nstate,1:nstate),a(nstate),b(nstate),rat(nstate) &
& ,bt(nstate),cp(0:nstate,1:nstate),bnorm(nstate)

open(4,file='out')

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

RN_gen_input=2
RN_seed_input = 717715_i8knd
RN_stride_input=0_i8knd
RN_hist_input=0_i8knd

call RN_init_problem( RN_gen_input, RN_seed_input, &
  & RN_stride_input, RN_hist_input, 1)
14 continue

do i=1,nstate

do j=1,nstate

   rn=rang()

   qj=0.84_dknd*(1+0.2*rn)

   p(i,j)=qj*2**( -(5.0_dknd*abs(i-j)) )

   write(*,1121)i,j,p(i,j)

1121 format(' p[',i5,',',i5,']=',f30.20,';')

enddo

enddo

p( 1, 1)= 9.5E-01_dknd
p( 1, 2)= 2.9E-02_dknd
p( 1, 3)= 9.7E-04_dknd
p( 1, 4)= 2.6E-05_dknd
p( 1, 5)= 8.4E-07_dknd
p( 1, 6)= 2.9E-08_dknd
p( 1, 7)= 9.0E-10_dknd
p( 1, 8)= 2.6E-11_dknd
p( 1, 9)= 8.8E-13_dknd
p( 1,10)= 2.8E-14_dknd
p( 2, 1)= 3.0E-02_dknd
p( 2, 2)= 9.3E-01_dknd
p( 2, 3)= 3.0E-02_dknd
p( 2, 4)= 8.9E-04_dknd
p( 2, 5)= 3.0E-05_dknd
p( 2, 6)= 8.4E-07_dknd
p( 2, 7)= 3.0E-08_dknd


\( p(2, 8) = 8.2 \times 10^{-10} \text{dknd} \)
\( p(2, 9) = 2.8 \times 10^{-11} \text{dknd} \)
\( p(2,10) = 8.8 \times 10^{-13} \text{dknd} \)
\( p(3, 1) = 8.7 \times 10^{-4} \text{dknd} \)
\( p(3, 2) = 3.0 \times 10^{-2} \text{dknd} \)
\( p(3, 3) = 8.5 \times 10^{-1} \text{dknd} \)
\( p(3, 4) = 2.7 \times 10^{-2} \text{dknd} \)
\( p(3, 5) = 8.6 \times 10^{-4} \text{dknd} \)
\( p(3, 6) = 3.0 \times 10^{-5} \text{dknd} \)
\( p(3, 7) = 9.2 \times 10^{-7} \text{dknd} \)
\( p(3, 8) = 2.6 \times 10^{-8} \text{dknd} \)
\( p(3, 9) = 8.8 \times 10^{-10} \text{dknd} \)
\( p(3,10) = 2.5 \times 10^{-11} \text{dknd} \)
\( p(4, 1) = 2.6 \times 10^{-5} \text{dknd} \)
\( p(4, 2) = 8.6 \times 10^{-4} \text{dknd} \)
\( p(4, 3) = 2.8 \times 10^{-2} \text{dknd} \)
\( p(4, 4) = 8.9 \times 10^{-1} \text{dknd} \)
\( p(4, 5) = 3.1 \times 10^{-2} \text{dknd} \)
\( p(4, 6) = 9.3 \times 10^{-4} \text{dknd} \)
\( p(4, 7) = 2.7 \times 10^{-5} \text{dknd} \)
\( p(4, 8) = 8.1 \times 10^{-7} \text{dknd} \)
\( p(4, 9) = 3.0 \times 10^{-8} \text{dknd} \)
\( p(4,10) = 7.8 \times 10^{-10} \text{dknd} \)
\( p(5, 1) = 9.4 \times 10^{-7} \text{dknd} \)
\( p(5, 2) = 2.6 \times 10^{-5} \text{dknd} \)
\( p(5, 3) = 8.6 \times 10^{-4} \text{dknd} \)
\( p(5, 4) = 3.0 \times 10^{-2} \text{dknd} \)
\( p(5, 5) = 9.2 \times 10^{-1} \text{dknd} \)
\( p(5, 6) = 3.1 \times 10^{-2} \text{dknd} \)
\( p(5, 7) = 9.3 \times 10^{-4} \text{dknd} \)
\[ p(5,8) = 2.6 \times 10^{-5} \text{dknd} \]
\[ p(5,9) = 8.1 \times 10^{-7} \text{dknd} \]
\[ p(5,10) = 2.8 \times 10^{-8} \text{dknd} \]
\[ p(6,1) = 2.5 \times 10^{-8} \text{dknd} \]
\[ p(6,2) = 9.0 \times 10^{-7} \text{dknd} \]
\[ p(6,3) = 2.9 \times 10^{-5} \text{dknd} \]
\[ p(6,4) = 8.9 \times 10^{-4} \text{dknd} \]
\[ p(6,5) = 3.0 \times 10^{-2} \text{dknd} \]
\[ p(6,6) = 9.3 \times 10^{-1} \text{dknd} \]
\[ p(6,7) = 3.0 \times 10^{-2} \text{dknd} \]
\[ p(6,8) = 9.8 \times 10^{-4} \text{dknd} \]
\[ p(6,9) = 2.8 \times 10^{-5} \text{dknd} \]
\[ p(6,10) = 8.6 \times 10^{-7} \text{dknd} \]
\[ p(7,1) = 8.1 \times 10^{-10} \text{dknd} \]
\[ p(7,2) = 2.6 \times 10^{-8} \text{dknd} \]
\[ p(7,3) = 8.5 \times 10^{-7} \text{dknd} \]
\[ p(7,4) = 2.7 \times 10^{-5} \text{dknd} \]
\[ p(7,5) = 9.6 \times 10^{-4} \text{dknd} \]
\[ p(7,6) = 2.9 \times 10^{-2} \text{dknd} \]
\[ p(7,7) = 8.8 \times 10^{-1} \text{dknd} \]
\[ p(7,8) = 3.1 \times 10^{-2} \text{dknd} \]
\[ p(7,9) = 8.7 \times 10^{-4} \text{dknd} \]
\[ p(7,10) = 2.9 \times 10^{-5} \text{dknd} \]
\[ p(8,1) = 2.8 \times 10^{-11} \text{dknd} \]
\[ p(8,2) = 8.0 \times 10^{-10} \text{dknd} \]
\[ p(8,3) = 2.6 \times 10^{-8} \text{dknd} \]
\[ p(8,4) = 9.3 \times 10^{-7} \text{dknd} \]
\[ p(8,5) = 2.8 \times 10^{-5} \text{dknd} \]
\[ p(8,6) = 8.6 \times 10^{-4} \text{dknd} \]
\[ p(8,7) = 2.8 \times 10^{-2} \text{dknd} \]
\begin{verbatim}
 do i=1,10
     do j=1,10
         if(p(i,j) < .00000001_dknd)p(i,j)=0.00000001_dknd
     enddo
 enddo
enddo
\end{verbatim}
do i=1,10
    write(*,1127)(p(i,j),j=1,10)
1127 format(10f11.8)
enddo

! form cumulative probability
do j=1,nstate
    cp(0,j)=0.
enddo
do j=1,nstate
do i=1,nstate
    cp(i,j)=cp(i-1,j)+p(i,j)
enddo
do j=1,nstate
    if(cp(nstate,j)>1.0_dknd) then
        ! write(*,*)'reject j,cp(nstate,j)=',j,cp(nstate,j)
go to 14
endif
enddo
do j=1,nstate
do i=1,nstate
    write(*,1129)i,j,p(i,j)
1129 format(' p[',i5,',',i5,']=',f11.8,';')
enddo
do j=1,nstate
do i=1,nstate
    write(*,1127)(p(i,j),j=1,10)
1127 format(10f11.8)
enddo
enddo
bt(1)=0.7242049396583518_dknd
bt(2)=0.6100224460514354_dknd
bt(3)=0.18029221875652388_dknd
bt(4)=0.12611919130915963_dknd
bt(5)=0.1594989592519969_dknd
bt(6)=0.15184328748549905_dknd
bt(7)=0.0613638578560001_dknd
bt(8)=0.039595201578962465_dknd
bt(9)=0.02443470204159862_dknd
bt(10)=0.023702878885358013_dknd
rktrue1=0.9746738906813877_dknd
rktrue2=0.969624316657604_dknd
domratio=rktrue2/rktrue1
write(*,*)'domratio=',domratio

rmssum=0
rmssum2=0
rkdiff1=0
rkdiff2=0
nrunc=1000
write(*,*)'npart=?'
read(*,*)npart

do 900 irun=1,nruns

do i=1,nstate
    a(i)=0
    b(i)=0
enddo
num=1
nprint=0

do 500 n=1,npart
new=0
nprint=nprint+1
if(n <= 1*nstate) then
    ns=mod(n-1,nstate)+1
endif
a(ns)=a(ns)+1.0_dknd
rn=rang()
if(rn > cp(nstate,ns))go to 490
! sample next state
ic=0
ib=nstate
10 continue
if(ib-ic.eq.1)go to 30
ih=(ic+ib)/2
if(rn.le.cp(ih,ns))then
    ib=ih
go to 10
else
    ic=ih
    go to 10
endif
30 continue
new=ib
b(new)=b(new)+num
go to 490
continue

nt=mod(nprint,npmod)
if(nt==0) then
if(npmod < 1952257800)npmod=npmod*1.1
atot=0.0_dknd
btot=0.0_dknd
do i=1,nstate
rat(i)=b(i)/a(i)
atot=atot+a(i)
btot=btot+b(i)
!
write(*,*)'i,k(i)=',i,rat(i)
dendo
rktot=btot/atot
sum=0
do i=1,nstate
sum=sum+b(i)**2
dendo

tn2=sqrt(sum)
do i=1,nstate
  bnorm(i)=b(i)/tn2
endo
sum=0
do i=1,nstate
  sum=sum+(bt(i)-bnorm(i))**2
endo
rms=sqrt(sum/nstate)
rkdiff=rktot-rktrue1
write(4,*)n,rms,abs(rkdiff)
dendif
if(n.le.nstate) go to 500
ratmx=-1.e23
do i=1,nstate
   rat(i)=b(i)/a(i)
   ! write(*,*)'i,a(i),b(i),k(i)=',i,a(i),b(i),rat(i)
   if(rat(i)>ratmx) then
      ratmx=rat(i)
      ns=i
   endif
endo
d500 continue
atot=0.0_dknd
btot=0.0_dknd
do i=1,nstate
   rat(i)=b(i)/a(i)
   atot=atot+a(i)
   btot=btot+b(i)
   write(*,*)'i,k(i)=',i,rat(i)
endo
rktot=btot/atot
write(*,2000)(a(i),i=1,nstate)
write(*,2000)(b(i),i=1,nstate)
2000 format(1p5e15.6)
! normalize
   sum=0
do i=1,nstate
      sum=sum+a(i)**2
endo
tn1=sqrt(sum)
doi=1,nstate
\begin{verbatim}
a(i)=a(i)/tn1
enddo
write(*,2200)(a(i),i=1,nstate)
sum=0
do i=1,nstate
    sum=sum+b(i)**2
enddo
tn2=sqrt(sum)
do i=1,nstate
    b(i)=b(i)/tn2
enddo
write(*,2201)(b(i),i=1,nstate)
2201 format('b=',1p5e20.10)
2200 format('a=',1p5e20.10)
do i=1,nstate
    write(*,3010)i,b(i),b(i)-bt(i)
3010 format(' b(',i2,')=',1p2e20.10)
enddo
sum=0
do i=1,nstate
    sum=sum+(bt(i)-b(i))**2
enddo
rms=sqrt(sum/nstate)
rkdiff=abs(rktot-rktrue1)
write(*,*)'irun,nrun=',irun,nruns,float(irun)/nruns
write(*,*)'npart,rms,abs(rkdiff)=',npart,rms,abs(rkdiff)

rmssum=rmssum+rms
rmssum2=rmssum2+rms**2
rkdiff1=rkdiff1+rkdiff
\end{verbatim}
rkdiff2=rkdiff2+rkdif**2
900 continue

avgrms=rmssum/nruns
avgrms2=rmssum2/nruns
var=avgrms2-avgrms**2
sdm=sqrt(var/(nruns-1))
avgrkdf2=rkdiff1/nruns
avgrkdif2=rkdiff2/nruns
vardif=avgrkdf2-avgrkdif**2
sdmrkdif=sqrt(vardif/(nruns-1))
write(*,*)'nruns=',nruns
write(*,*)'npart,avgrms,sdm=',npart,avgrms,sm
write(*,*)'npart,avgrkdif,sdmrkdif=',npart,avgrkdif,smrkdif
end

! Copyright LANS/LANL/DOE - see file COPYRIGHT_INFO

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
- RN init for particle, special: RN_next_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
* 12-12-2006 - C Zeeb, added subroutine RN_next_particle

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

!-------------------
MCNP output units
!-------------------

!!!!!!!  teb use mcnp_params, only: iuo, I8KND, DKND
!!!!!!! teb use mcnpiofiles, only: jtty

integer jtty !!!!!!!! teb

integer, parameter, public :: i8knd = selected_int_kind(18)  ! 8-byte integer kind !!! teb
integer, parameter, public :: iuo    = 32    ! I/O unit for problem output file.
integer(i8knd), parameter, public :: i8limit = huge(1_i8knd)  ! Max integer*8 ~1E20
integer, parameter, public :: dknd = selected_real_kind(15,307)  ! 8-byte real kind

PRIVATE

!-----------------------------------
! Public functions and subroutines for this module
!-----------------------------------

PUBLIC :: rang
PUBLIC :: RN_init_problem
PUBLIC :: RN_init_particle
PUBLIC :: RN_next_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
PUBLIC :: jteb1sub  ! teb
! 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

22
! * 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
integer(I8), SAVE :: RN_COUNT_ADVANCES= 0 ! Used by RN_next_particle

!---------------------------------------------------------------------
! 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, 26025100190209_I8, &
& 106001385730621_I8, 232883458246025_I8, 979348506190209_I8, &
& 163056893025873_I8, &
! ***** 2 *****
& 9219741426499971446_I8, 666764808255707375_I8, 4935109208453540924_I8, &
& 7076815037777023853_I8, 5594070487082964434_I8, 7069484152921594561_I8, &
& 8424485724631928902_I8, 19322398608391599_I8, 863975969196973212_I8, &
& 81813158193753227437_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, 665145400413087106_I8, 2732760390316414145_I8, &
& 2067727651689204870_I8, 2707840203503213343_I8, 6009142246302485212_I8, &
& 6678916955629521741_I8, &
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
endo

! 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 )n = 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

!!! teb if( new_standard_gen<1 .or. new_standard_gen>n_RN_GEN ) then
! 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
RN_COUNT_ADVANCES = 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
  !!! teb call expire( 0, 'RN_init_problem', &
  !!! teb & ' ***** 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
  !!! teb call expire( 0, 'RN_init_problem', &
  !!! teb & ' ***** 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

dend 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_next_particle( nps, skip, np_run )

! advance the MCNP random number parameters to the next particle
!
!
! * generate a new particle seed from the base seed & particle index
! * set the RN count to zero

implicit none

integer(I8), intent(in) :: nps
integer(I8), intent(in) :: skip
integer(I8), intent(in) :: np_run
 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") then 
    RN_COUNT_TOTAL = value 
  endif 

  if(key == "count_stride") then 
    RN_COUNT_STRIDE = value 
  endif 

  if(key == "count_max") then 
    RN_COUNT_MAX = value 
  endif 

  if(key == "count_max_nps") then 
    RN_COUNT_MAX_NPS = value 
  endif 

  if(key == "seed") then 

end subroutine RN_set
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 == "count_advances" ) RN_query = RN_COUNT_ADVANCES
    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
dend 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
dend subroutine RN_update_stats
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, 0 )

! 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
endo
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 *****"

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

  !-------------------------------------------------------------------
end subroutine RN_test_mixed
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
endo

return

end subroutine RN_test_mixed
end module mcnp_random