Title: Is the Standard Monte Carlo Power Iteration Approach the Wrong Approach? Part 2

Author(s): Booth, Thomas E.

Intended for: share with colleagues
Report
Web

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

Part 2

Thomas E. Booth

LA-UR-12-DRAFT, July 10, 2012

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

Abstract

The recent work "Is the Standard Monte Carlo Power Iteration Approach the Wrong Approach?" speculated that the second eigenfunction could be built using essentially the same "building brick" approach that obtained the first eigenfunction in LA-UR-12-21928. This note shows that the speculation was at least partially correct, but not complete.

1 Introduction

Small modifications of the procedure used in LA-UR-12-21928 [1] for the first eigenfunction allow estimation of the second eigenfunction. For the second eigenfunction, one uses negative as well as positive weight source neutrons. Note that by definition each component of a vector is a "state" and a "region" is a collections of states. One computes global K_S's for both negative and positive regions; that is, one computes a K_{S-} for the region containing negative state weights and similarly a K_{S+} for the region containing positive state weights. Specifically, one computes both a K_{S-} and a K_{S+} corresponding to Eq. 8 in LA-UR-12-21928, depending upon whether the region's weight is negative or positive. The global region having the smaller K has neutrons of the appropriate sign dropped into the states where the local k estimate is larger than the global K estimate in the region. Restating more mathematically, with $I(T) = 0$ when T is false and $I(T) = 1$ when T is true, one has

$$K_{S-} = \frac{\sum_i I(R_i < 0)R_i}{\sum_j I(Q_j < 0)Q_j} \tag{1}$$

$$K_{S+} = \frac{\sum_i I(R_i > 0) R_i}{\sum_j I(Q_j > 0) Q_j} \tag{2}$$

If $K_{S-} > K_{S+}$ then the procedure in section 3 of LA-UR-12-21928 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 of LA-UR-12-21928 is followed with negative weight neutrons for all states for which $I(R_j < 0)$.

One thing different from the procedure in section 3 of LA-UR-12-21928 is that one must decide what to do when the source (Q_i) and response (R_i) do not agree in sign. That is, $R_i/Q_i < 0$. This problem never comes up when calculating the fundamental because there are no negative weights. If $Q_i \neq 0$ and $R_i = 0$ then it is not yet clear whether there will be a mismatch in signs. Here, a guess is tried. If $R_i \times Q_i \leq 0$, then state i is taken as belonging to the global region having the smaller K and a neutron appropriate to this global region (i.e., of correct sign) is dropped into state i. This guess will be discussed further in the section on future work.

2 Modification of the Algorithm in LA-UR-12-21928

In this section the algorithm in LA-UR-12-21928 is modified to obtain the second eigenfunction. The differences are mainly due to the necessity of dealing with neutrons of signed weight. As in LA-UR-12-21928, N_t is the total number of neutrons to run and the indicator function $I(A)$ is used. Specifically, $I(A) = 0$ when A is false and $I(A) = 1$ when A is true.

Before specifying the procedure, a few comments are perhaps worthwhile. The purpose of this report was to investigate the convergence to the eigenfunction as a function of the total number (N_t) of neutrons used. The stopping criterion reflects this. If one were doing a more typical calculation and simply trying to *get* the second eigenfunction rather than trying to *study* the convergence, one would probably change the stopping criterion in step 7 to something such as "when the local eigenvalue estimates v_j are all close enough to being the same." For a fixed N_t, there will be some difference between the v_j ($j = 1, \ldots, 10$), K_{S-}, and K_{S+}. In the limit as $N_t \to \infty$ the v_j ($j = 1, \ldots, 10$), K_{S-}, and K_{S+} should all be equal to the true second eigenvalue.

The procedure for obtaining the second eigenfunction is:

1. Initially $Q_j = 0$ and $R_i = 0$ for all i and all j.

2. Start 50 negative weight neutrons, $w = -1$, in one user-specified state j. For each neutron, increment (by $w = -1$) the source weight $Q_j \leftarrow Q_j - 1$. The fission state i (or termination) is sampled from A_{ij} and the resulting fission

distribution updated; i.e., $R_i \leftarrow R_i - 1$. (At the end of this step $Q_j = -50$.)

3. Start 50 positive weight neutrons, $w = +1$, in one user-specified state m. (m is different from j in the previous step.) For each neutron, increment (by $w = +1$) the source weight $Q_m \leftarrow Q_m + 1$. The fission state i (or termination) is sampled from A_{im} and the resulting fission distribution updated; i.e., $R_i \leftarrow R_i + 1$. (At the end of this step $Q_m = 50$.)

4. $j \leftarrow 0$

5. Calculate the negative global system eigenvalue estimate $K_{S-} = \frac{\sum_i I(R_i<0)R_i}{\sum_j I(Q_j<0)Q_j}$ and the individual negative eigenvalue estimates $v_j = \frac{R_j}{Q_j}$ when $R_j < 0$.

6. Calculate the positive global system eigenvalue estimate $K_{S+} = \frac{\sum_i I(R_i>0)R_i}{\sum_j I(Q_j>0)Q_j}$ and the individual positive eigenvalue estimates $v_j = \frac{R_j}{Q_j}$ when $R_j > 0$.

7. If $N = N_t$, then all neutrons have been run and all eigenvalue estimates have been made. The calculation is done. Go to 18.

8. Choose the appropriate signed weight to continue. If $K_{S-} < K_{S+}$ then choose $w = -1$ or if $K_{S-} \geq K_{S+}$ then choose $w = +1$.

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

10. If R_j and w have different signs, then try the next state instead. That is, go to 9.

11. Depending on the sign of the neutron to be added, the global eigenvalue estimate of that sign will be compared to the local eigenvalue estimate in step 12 or 13. If $w = -1$ go to 12 or if $w = +1$ go to 13.

12. Negative weight neutron. Check local eigenvalue estimate in state j against global negative eigenvalue estimate.

    If $w = -1$ and $v_j \geq K_{S-}$, drop a negative weight neutron in state j. Go to 14.

    If $w = -1$ and $v_j < K_{S-}$, do nothing. Go to 15.

13. Positive weight neutron. Check local eigenvalue estimate in state j against global positive eigenvalue estimate.

    If $w = +1$ and $v_j \geq K_{S+}$, drop a positive weight neutron in state j. Go to 14.

    If $w = +1$ and $v_j < K_{S+}$, do nothing. Go to 15.

14. Update

    $Q_j \leftarrow Q_j + w$ (another neutron is being sourced into state j)

    $N \leftarrow N + 1$ (another neutron is being sourced in somewhere)

    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

    $R_i \leftarrow R_i + w$

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

15. If $N = N_t$, then done adding neutrons. Need to compute final eigenvalues. Go to 5.

16. If $j < 10$ go to 9.

17. If $j \geq 10$ go to 4. (Once all the source states have been processed, the global and local eigenvalues need to be recalculated.)

18. End calculation

3 Discrete Ten State Transport Problem

This section gives results for the second eigenfunction on the same matrix as in LA-UR-12-21928. That is, the transport operator A is the matrix with elements A_{ij}:

$$\begin{pmatrix}
.95000000 & .02900000 & .00097000 & .00002600 & .00000084 & .00000003 & .00000001 & .00000001 & .00000001 & .00000001 \\
.03000000 & .93000000 & .03000000 & .00089000 & .00003000 & .00000084 & .00000003 & .00000001 & .00000001 & .00000001 \\
.00087000 & .03000000 & .85000000 & .02700000 & .00086000 & .00003000 & .00000092 & .00000003 & .00000001 & .00000001 \\
.00002600 & .00086000 & .02800000 & .89000000 & .03100000 & .00093000 & .00002700 & .00000081 & .00000003 & .00000001 \\
.00000094 & .00002600 & .00086000 & .03000000 & .92000000 & .03100000 & .00093000 & .00002600 & .00000081 & .00000003 \\
.00000002 & .00000090 & .00002900 & .00089000 & .03000000 & .93000000 & .03000000 & .00098000 & .00002800 & .00000086 \\
.00000001 & .00000003 & .00000085 & .00002700 & .00096000 & .02900000 & .88000000 & .03100000 & .00087000 & .00002900 \\
.00000001 & .00000001 & .00000003 & .00000093 & .00002800 & .00086000 & .02800000 & .91000000 & .02800000 & .00098000 \\
.00000001 & .00000001 & .00000001 & .00000003 & .00000087 & .00002700 & .00091000 & .02900000 & .90000000 & .02600000 \\
.00000001 & .00000001 & .00000001 & .00000001 & .00000003 & .00000081 & .00002800 & .00084000 & .07100000 & .90000000
\end{pmatrix}$$

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

$$s_j = \sum_{i=1}^{10} A_{ij} < 1 \tag{3}$$

and the termination probability from state j is

$$t_j = 1 - s_j = 1 - \sum_{i=1}^{10} A_{ij} > 0 \tag{4}$$

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:

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

4 The Good News

Figure 1 shows results from 100 runs starting initially with 50 $w = -1$ neutrons dropped into state 1 and 50 $w = +1$ neutrons dropped into state 3. Note all 100 runs converge. Figure 1 plots results after a 100 thousand, 1 million, 10 million, and 100 million neutrons.

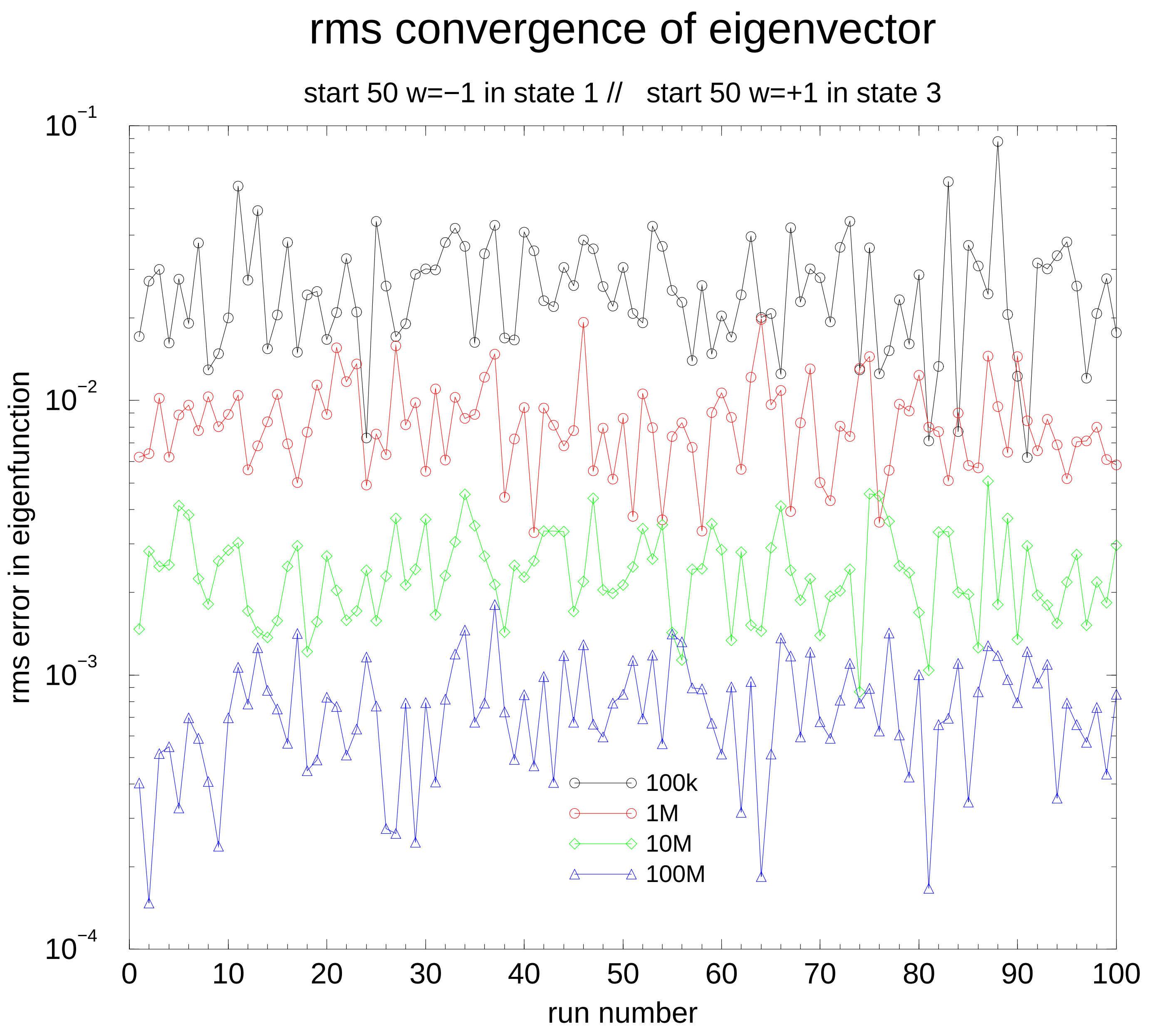

Figure 1: Pleasing Convergence to Eigenfunction

5 The Bad News

Figure 2 shows results from 100 runs starting initially with 50 $w = -1$ neutrons dropped into state 1 and 50 $w = +1$ neutrons dropped into state 10. Note that most, but not all, of the 100 runs converge. Again, Figure 2 plots results after a 100 thousand, 1 million, 10 million, and 100 million neutrons.

Note from Figure 2 that run 46 is one of the runs that does not converge well. Figure 3 shows a detailed look at run 46 and adds results for 1 billion and 2 billion neutrons. Figure 4 compares the estimated eigenfunctions to the exact eigenfunction. Note that at two billion neutrons, the eigenfunction matches pretty well except perhaps for states 1-3. Figure 5 gets rid of the clutter and only compares the two billion eigenfunction estimate with the exact eigenfunction. Note the mismatch in states 1-3. Figure 6 shows that the small mismatch in the eigenfunction appears to be because the local eigenfunction estimate k_3 is *way* off. The guessed algorithm is not making the necessary sign change for state 3. Maybe it is worthwhile displaying these functions in nongraphical form as well. The exact eigenfunction is:

```
 i     R_i
================
 1   -2.1763322613E-01
 2   -1.4799075109E-01
 3    1.5071633084E-02
 4    2.2038750945E-01
 5    5.3735552864E-01
 6    6.3770501364E-01
 7    2.9094175322E-01
 8    2.2315011151E-01
 9    1.5836103611E-01
10    1.6430692516E-01
```

The estimated eigenfunction is:

```
 i     R_i
================
 1   -2.5000114348E-01

 2   -1.8487205437E-01

 3   -3.2134165198E-04

 4    2.1058781363E-01

 5    5.2735410045E-01

 6    6.2958021269E-01

 7    2.8806708416E-01

 8    2.2226069256E-01

 9    1.5831382616E-01

10    1.6448435124E-01
```

The exact local k_i's are (they have to be identical by definition of an eigenfunction)

```
 i     k_i

=================

1 0.969624316657604

2 0.969624316657604

3 0.969624316657604

4 0.969624316657604

5 0.969624316657604

6 0.969624316657604

7 0.969624316657604

8 0.969624316657604

9 0.969624316657604

10 0.969624316657604
```

The estimated local k_i's are

```
 i     k_i

=================
1 0.9714808170426541

2 0.9695144612869399

3 0.3765475758097095

4 0.9695144369990029

5 0.9695144379627307

6 0.9695144451179467

7 0.9695144396277504

8 0.969514435108306

9 0.9695144479870336

10 0.9695144388612372
```

Note that the estimated eigenfunction component $R_3 = -3.2134165198E-04$ is getting very near zero, but remains negative. That is, the algorithm cannot seem to flip the sign and approach the true eigenfunction component $R_3 = 1.5071633084E-02$.

6 Comments on the Successes and Failures and Future Work

Despite some unacceptable failures, the important thing to note is that *most* of the time, the second eigenfunction *is* successfully obtained. Thus, probably all that is necessary is to uncover some small tweak to ensure that the second eigenfunction is *always* successfully obtained. Fortunately, there is an aspect of building the second eigenfunction that is different from building the first eigenfunction. This aspect has not yet been exploited nor investigated.

Note that when building the first eigenfunction if the local eigenvalue estimate:

$$v_i = \frac{R_i}{Q_i} \tag{5}$$

is not consistent with the global eigenvalue K_S estimate then when $v_i > K_S$ then one tries to make v_i smaller by increasing Q_i by adding one more neutron to state i. If $v_i < K_S$ then one eschews increasing Q_i and simply waits until transfers from neutrons sourced into other states increase R_i, thus increasing v_i.

When building the second eigenfunction there is now a *choice* of adding either a positive or a negative neutron. This choice has not been exploited in the algorithm herein. Future work will investigate the implications of this choice in developing an algorithm that always converges. In particular, this choice might be used to deal with the problem of states having uncertain sign ($R_i \times Q_i \leq 0$) mentioned in section 1. (As shown herein, the guess in section 1 did not always solve this problem.)

Roger Martz (XCP-3) posed a different question. Roger asked whether it would be possible to estimate both the first and second eigenvalues simultaneously with this method. The answer is probably. One would use both "first eigenvalue" neutrons and "second eigenvalue" neutrons. The states would be swept through as in the procedure herein and for each state one could either add or not add a "first eigenvalue" neutron as well as add or not add a "second eigenvalue" neutron. How much work would be saved relative to two independent calculations would need further study. At the moment, it is probably good to focus on modifying the method so that the second eigenfunction is *always* produced before studying producing the first and second eigenfunctions simultaneously.

References

[1] "Is the Standard Monte Carlo Power Iteration Approach the Wrong Approach?", Thomas E. Booth, Los Alamos Report LA-UR-12-21928 (May 29, 2012)

7 Acknowledgement

Thanks to Roger Martz for making one crucial correction and many useful suggestions that improved the explanations in this report.

8 Appendix - Source Code for Calculations

```
       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)




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

  integer(i8knd):: &
    & RN_seed_input,      &  != user input, starting RN seed
```

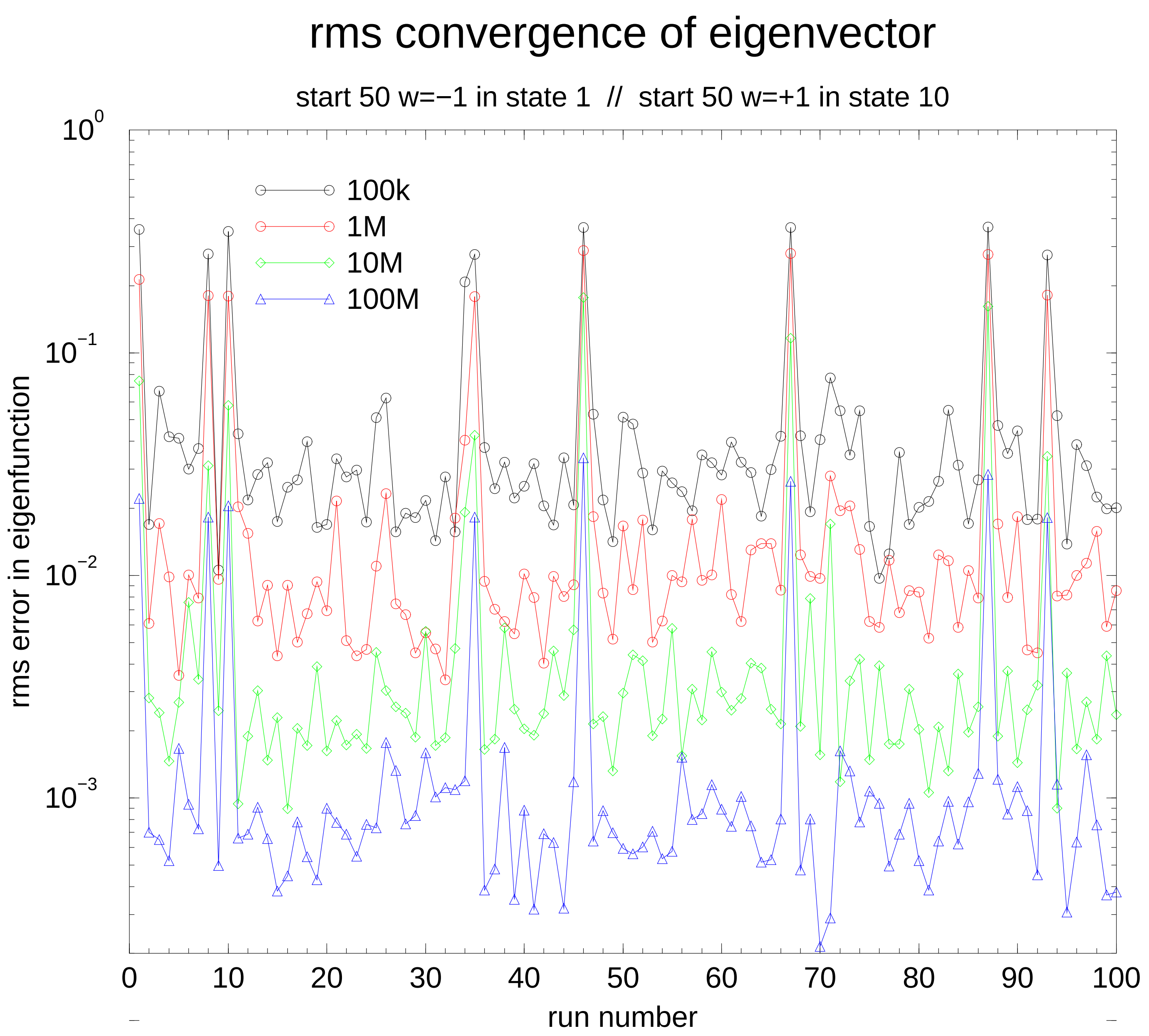

Figure 2: Problematical Convergence to Eigenfunction

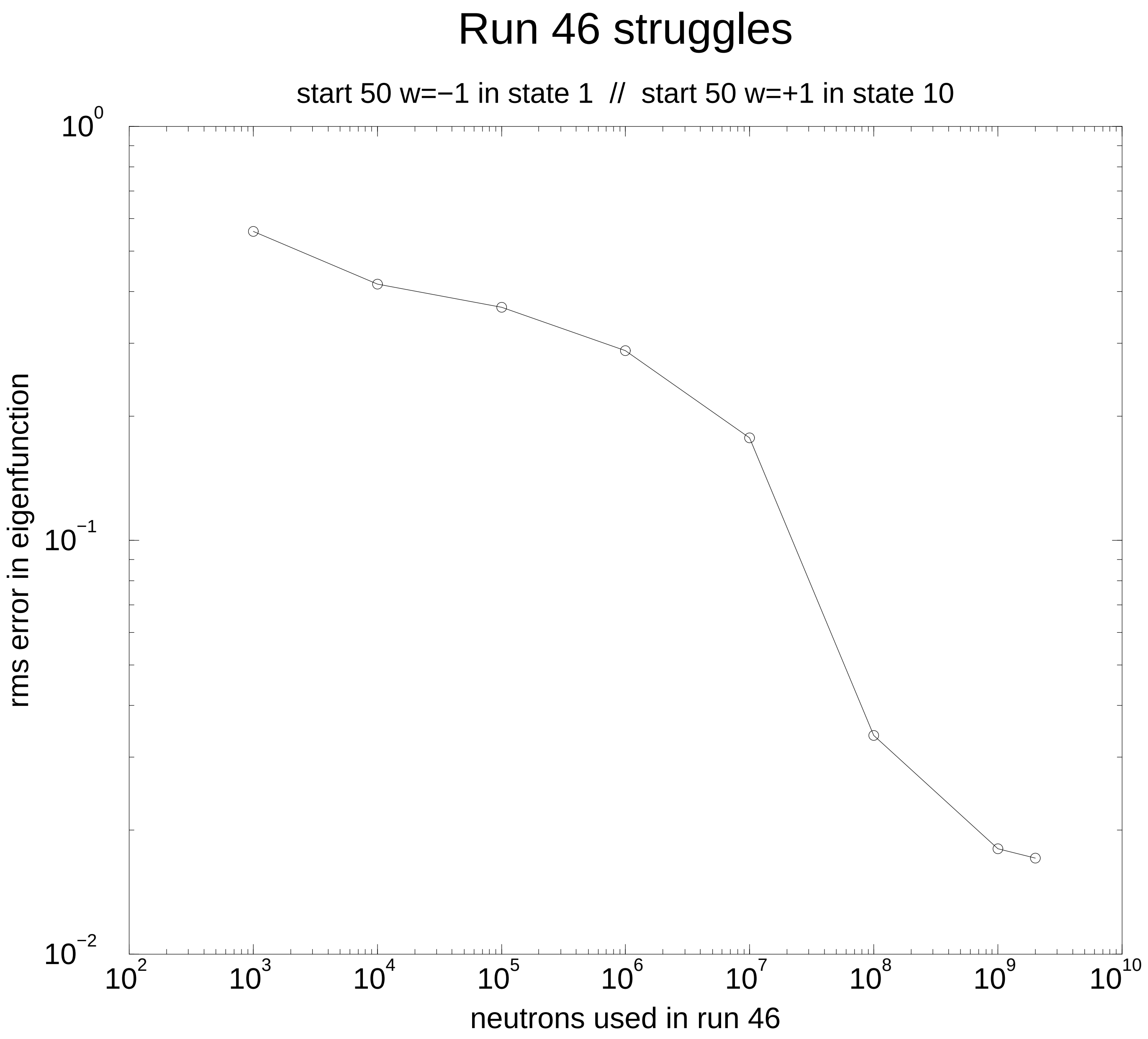

Figure 3: Detailed look at problematical run number 46

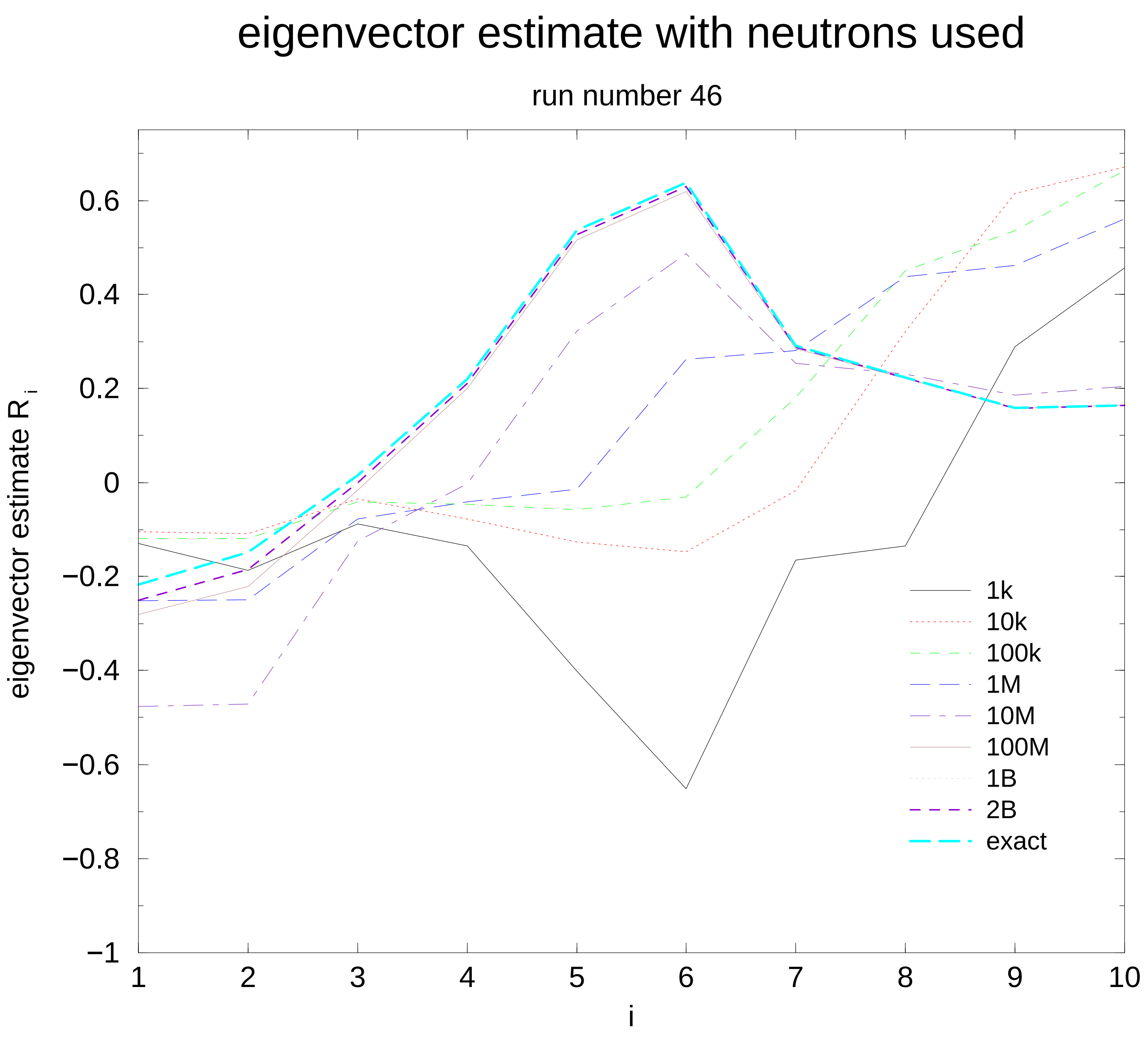

Figure 4: Eigenfunction nonconvergence on run number 46

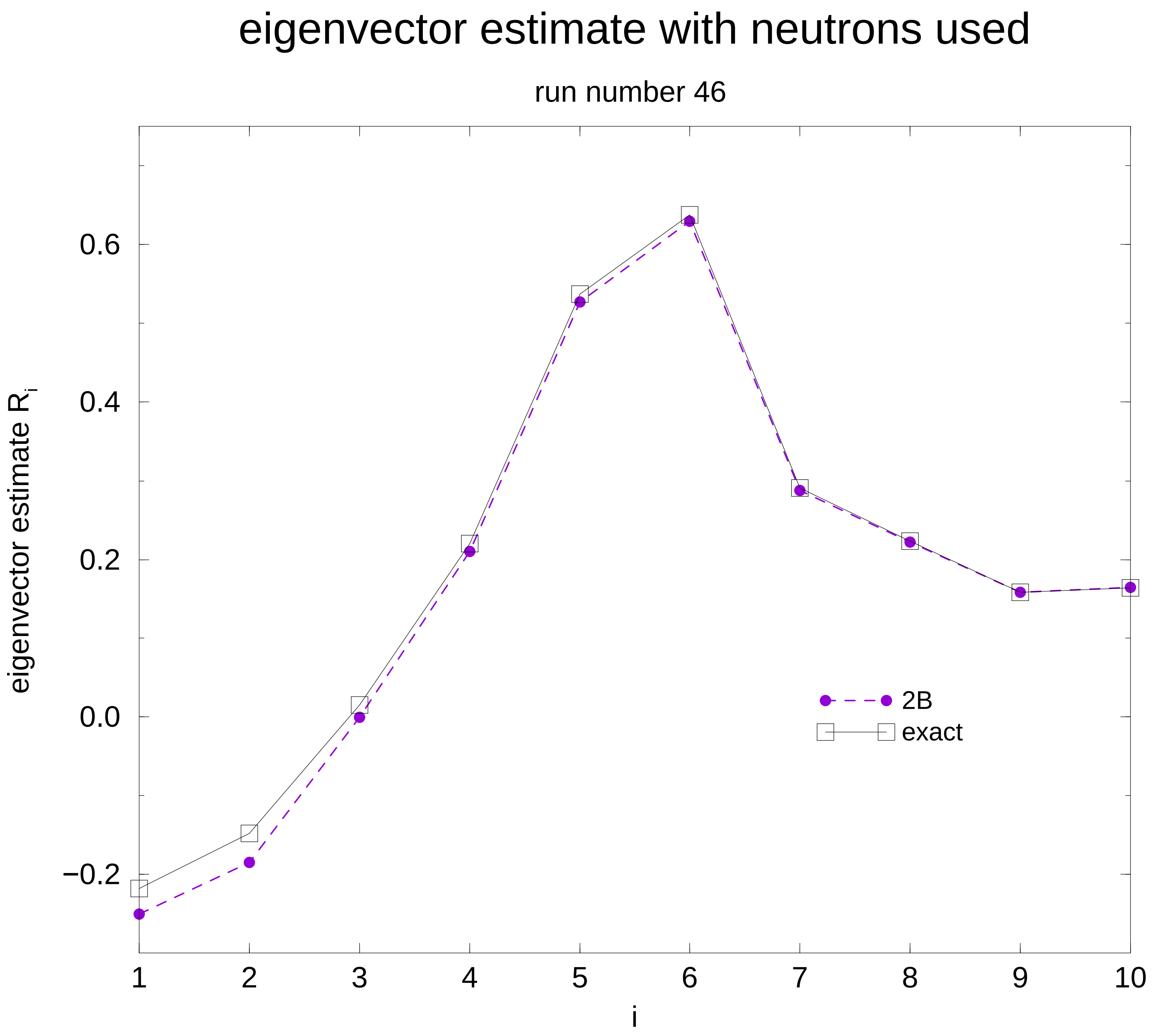

Figure 5: Detail of nonconvergence on run number 46

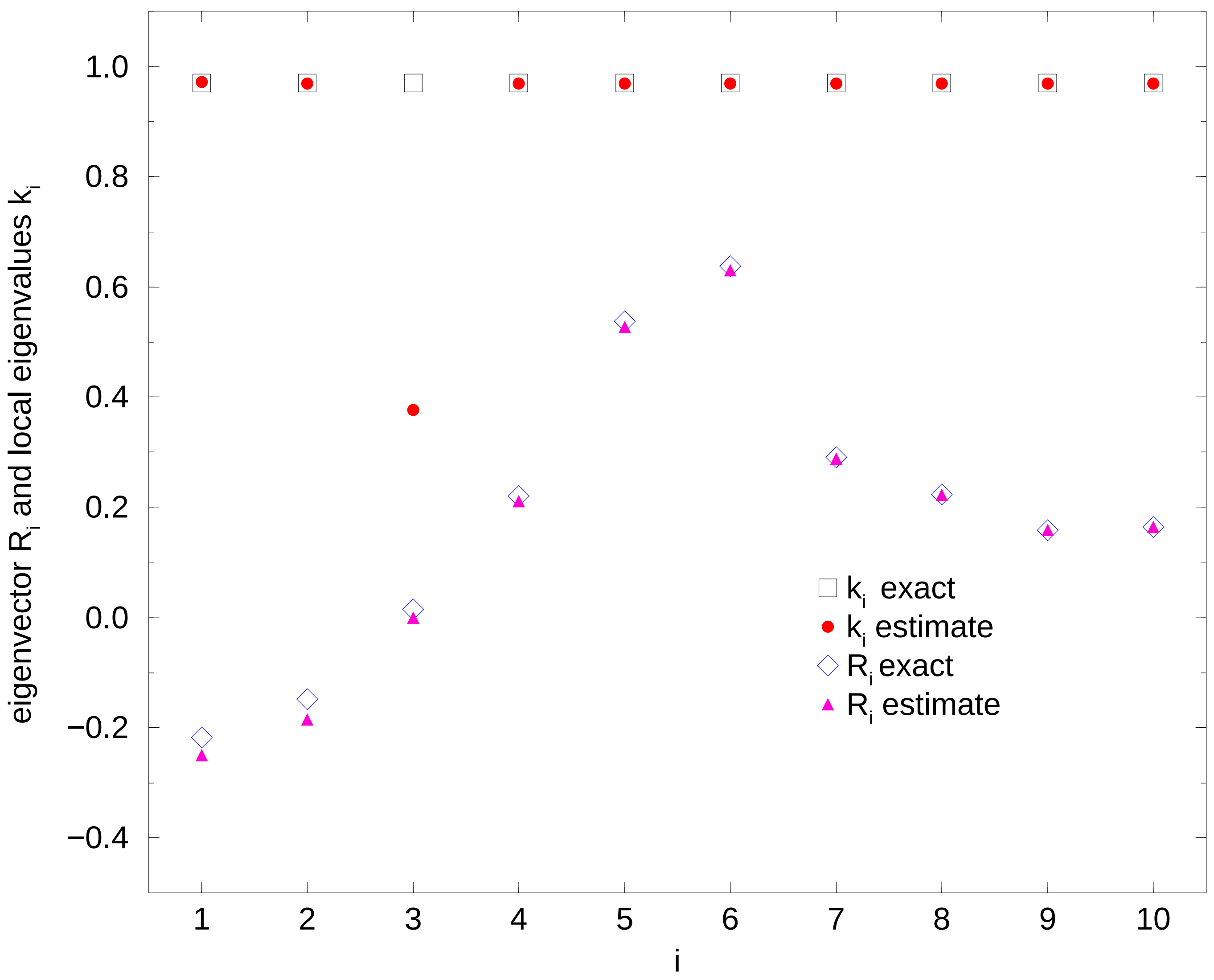

Figure 6: Run 46 mismatches

```
   & 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)
```

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

      npmod=1000

   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)= 8.2E-10_dknd

p( 2, 9)= 2.8E-11_dknd

p( 2,10)= 8.8E-13_dknd

p( 3, 1)= 8.7E-04_dknd

p( 3, 2)= 3.0E-02_dknd

p( 3, 3)= 8.5E-01_dknd

p( 3, 4)= 2.7E-02_dknd

p( 3, 5)= 8.6E-04_dknd

p( 3, 6)= 3.0E-05_dknd

p( 3, 7)= 9.2E-07_dknd

p( 3, 8)= 2.6E-08_dknd

p( 3, 9)= 8.8E-10_dknd

p( 3,10)= 2.5E-11_dknd

p( 4, 1)= 2.6E-05_dknd

p( 4, 2)= 8.6E-04_dknd

p( 4, 3)= 2.8E-02_dknd

p( 4, 4)= 8.9E-01_dknd

p( 4, 5)= 3.1E-02_dknd

p( 4, 6)= 9.3E-04_dknd

p( 4, 7)= 2.7E-05_dknd

p( 4, 8)= 8.1E-07_dknd

p( 4, 9)= 3.0E-08_dknd

p( 4,10)= 7.8E-10_dknd

p( 5, 1)= 9.4E-07_dknd

p( 5, 2)= 2.6E-05_dknd
```

```
p( 5, 3)= 8.6E-04_dknd

p( 5, 4)= 3.0E-02_dknd

p( 5, 5)= 9.2E-01_dknd

p( 5, 6)= 3.1E-02_dknd

p( 5, 7)= 9.3E-04_dknd

p( 5, 8)= 2.6E-05_dknd

p( 5, 9)= 8.1E-07_dknd

p( 5,10)= 2.8E-08_dknd

p( 6, 1)= 2.5E-08_dknd

p( 6, 2)= 9.0E-07_dknd

p( 6, 3)= 2.9E-05_dknd

p( 6, 4)= 8.9E-04_dknd

p( 6, 5)= 3.0E-02_dknd

p( 6, 6)= 9.3E-01_dknd

p( 6, 7)= 3.0E-02_dknd

p( 6, 8)= 9.8E-04_dknd

p( 6, 9)= 2.8E-05_dknd

p( 6,10)= 8.6E-07_dknd

p( 7, 1)= 8.1E-10_dknd

p( 7, 2)= 2.6E-08_dknd

p( 7, 3)= 8.5E-07_dknd

p( 7, 4)= 2.7E-05_dknd

p( 7, 5)= 9.6E-04_dknd

p( 7, 6)= 2.9E-02_dknd

p( 7, 7)= 8.8E-01_dknd

p( 7, 8)= 3.1E-02_dknd

p( 7, 9)= 8.7E-04_dknd
```

```
p( 7,10)= 2.9E-05_dknd

p( 8, 1)= 2.8E-11_dknd

p( 8, 2)= 8.0E-10_dknd

p( 8, 3)= 2.6E-08_dknd

p( 8, 4)= 9.3E-07_dknd

p( 8, 5)= 2.8E-05_dknd

p( 8, 6)= 8.6E-04_dknd

p( 8, 7)= 2.8E-02_dknd

p( 8, 8)= 9.1E-01_dknd

p( 8, 9)= 2.8E-02_dknd

p( 8,10)= 9.8E-04_dknd

p( 9, 1)= 9.1E-13_dknd

p( 9, 2)= 2.9E-11_dknd

p( 9, 3)= 9.0E-10_dknd

p( 9, 4)= 2.8E-08_dknd

p( 9, 5)= 8.7E-07_dknd

p( 9, 6)= 2.7E-05_dknd

p( 9, 7)= 9.1E-04_dknd

p( 9, 8)= 2.9E-02_dknd

p( 9, 9)= 9.0E-01_dknd

p( 9,10)= 2.6E-02_dknd

p(10, 1)= 2.7E-14_dknd

p(10, 2)= 7.8E-13_dknd

p(10, 3)= 2.6E-11_dknd

p(10, 4)= 8.3E-10_dknd

p(10, 5)= 3.0E-08_dknd

p(10, 6)= 8.1E-07_dknd
```

```
     p(10, 7)= 2.8E-05_dknd

     p(10, 8)= 8.4E-04_dknd

     p(10, 9)= 7.1E-02_dknd

     p(10,10)= 9.0E-01_dknd


      do i=1,10

      do j=1,10

        if(p(i,j) < .00000001_dknd)p(i,j)=0.00000001_dknd

      enddo

      enddo



      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

      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
      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

nruns=1000

write(*,*)'npart=?'

read(*,*)npart

do 900 irun=1,nruns


do i=i,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


  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)
        enddo
        rktot=btot/atot
        sum=0
        do i=1,nstate
          sum=sum+b(i)**2
        enddo
        tn2=sqrt(sum)
        do i=1,nstate
          bnorm(i)=b(i)/tn2
        enddo
        sum=0
        do i=1,nstate
          sum=sum+(bt(i)-bnorm(i))**2
        enddo
      rms=sqrt(sum/nstate)
      rkdiff=rktot-rktrue1
      write(4,*)n,rms,abs(rkdiff)
      endif
      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
      enddo
  500 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)
      enddo
      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
      enddo
      tn1=sqrt(sum)
      do i=1,nstate
        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

      rkdiff2=rkdiff2+rkdiff**2

  900 continue


      avgrms=rmssum/nruns

      avgrms2=rmssum2/nruns

      var=avgrms2-avgrms**2

      sdm=sqrt(var/(nruns-1))

      avgrkdif=rkdiff1/nruns

      avgrkdif2=rkdiff2/nruns

      vardif=avgrkdif2-avgrkdif**2

      sdmrkdif=sqrt(vardif/(nruns-1))

      write(*,*)'nruns=',nruns

      write(*,*)'npart,avgrms,sdm=',npart,avgrms,sdm

      write(*,*)'npart,avgrkdif,sdmrkdif=',npart,avgrkdif,sdmrkdif

      end


!+ $Id: mcnp_random.F90,v 1.10 2009/09/15 16:58:24 hgh Exp $

! 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 mcnp_iofiles, 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

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

  ! Kinds for LONG INTEGERS (64-bit) & REAL*8 (64-bit)

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

  integer, parameter :: R8 = DKND

  integer, parameter :: I8 = I8KND


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

  ! 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
!
!   * 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,     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

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

    !$OMP CRITICAL (RN_NEXT_PART)
    RN_COUNT_ADVANCES = np_run + skip
    RN_SEED  = RN_skip_ahead( RN_SEED0, RN_COUNT_ADVANCES*RN_STRIDE )
    !$OMP END CRITICAL (RN_NEXT_PART)
    RN_COUNT = 0
    RN_NPS   = nps

    return
  end subroutine RN_next_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 == "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
```

```
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


!-------------------------------------------------------------------
!###################################################################
!#
!#  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, 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
  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
```