Random numbers

November 24, 2009
13:32

Contents

1 Introduction 1
2 External interface to random 3
3 Typical calling sequence 6
4 Random definitions 8
5 Functions to return random numbers 10
6 Calculate the next batch 13
7 Initializing the random number sequence 14
8 Initializing the seed 19
9 Convert a seed to its decimal equivalent 22
10 Routines to step the seed forward 24
11 INDEX 27
1 Introduction

Random number routines for degas. We attempt to satisfy four objectives with these routines: (1) they should produce high-quality random numbers uniformly distributed in \((0, 1)\); (2) they should be fast; (3) they should be portable; (4) it should be easy to run many independent sequences of random numbers.

The methods used here are described in D. E. Knuth, *Art of Computer Programming* (3rd edition, Addison-Wesley, 1998), Vol. 2, *Seminumerical Algorithms*, Chapter 3, Random Numbers. The basic method is given by the routine `ran_array` in Section 3.6 and converted to floating method according to the Exercise 3.6-10. The Fortran version of this routine is available at

http://www-cs-faculty.stanford.edu/~knuth/programs/frngdb.f

The method uses the recurrence

\[
X_n = (X_{n-l} + X_{n-k}) \mod 1,
\]

where \(X_n\) are multiples of \(2^{-e}\).

The following changes have been made to Knuth’s version:

- In order to allow separate threads of random numbers to be generated, the state of the random number generator is passing to the random number routines via the argument list (instead of being stored in a common block).
- The number of significant bits, \(e\) has been reduced from 52 to 47 to accommodate the precision of Cray computers.
- We add \(2^{-(e+1)}\) to the random numbers returned by `random` and `random_array`. This centers the distribution of random numbers between 0 and 1 and ensures that the numbers lie strictly between 0 and 1. The latter property means, for example, that we can safely take the log of the random numbers.
- `random_array` can return any number of random numbers. (Knuth’s `ran_array` was coded to return at least \(k\) random numbers.)
- The lags \((k, l)\) are chosen to be \((63, 100)\) instead of \((37, 100)\). Because \(63 + 37 = 100\), this generates the sequence in reverse (with an alternating sign change). The larger value of \(k\) allows better vectorization (since the 63 numbers can be generated in parallel).
- We follow the suggestion of Exercise 3.6-15 using only \(k\) numbers out of each batch of \(p = 1009\).
The state of the random number generator is defined by \( k \) e-bit numbers. If we disallow the state where the least-significant bit of all these \( k \) numbers is zero, the period of equation \([1]\) is \( 2^{-1}(2^k - 1) \approx 8.9 \times 10^{43} \). All the bits of the random numbers are “good” (in contrast to linear congruential random generators where the less significant bits are less random). Employing the method of using only \( k \) out of \( p \) random numbers, we have a method with no known defects. It is also fairly fast since it involves only floating-point addition.

In order to start this random number generator, we need to specify its initial state. Knuth includes an algorithm to do this which accepts a single integer seed, which can take on approximately \( 10^9 \) different values and computes an initial state of the random number generator, in such a way that a particular seeds is guaranteed to produce at least \( 2^{70} \) different states before “colliding” with those produces by a different seed.

We don’t use this method here because (1) it’s rather slow (100 times slower than the method we use), and (2) \( 10^9 \) possible seeds seems to be rather restrictive in parallel applications which may need to consume many possible seeds in a given run.

The total number of allowable states this random generator can take is \( 2^{k(e-1)}(2^k - 1) \). Thus a single sequence generated by equation \([1]\) covers a tiny fraction \( 2^{-(e-1)(k-1)} \approx 10^{-1371} \) of allowable states. If we just picked two initial states “randomly,” it would be highly unlikely that they would belong to the same sequence. In fact, we would have to pick about \( 10^{985} \) initial states before having an appreciable probably of a collision.

We therefore adopt a strategy where we use a conventional linear congruential random generator to initialize equation \([1]\)

\[
T(x) \equiv (a_1x + c_1) \mod 2^{112},
\]

We choose \( a_1 = 31167285 \times 2^{64} + 6364136223646793005 \), \( c_1 = 1 \). The multiplier \( a_1 \) is obtained by concatenating two “good” multipliers \( 31167285 \mod 2^{48} \) and \( 6364136223646793005 \mod 2^{64} \) given in lines 23 and 26 of Knuth, Section 3.3.4, Table 1. This multiplier will give good results in the spectral test ensuring good convergence in multidimensional space. The state is obtained from the high \( e \) bits of \( k \) consecutive numbers from equation \([2]\). Actually we place the most significant (i.e., the “most random”) bits from equation \([2]\) into the least significant bits of the state vector where they can the most good (since they will result in carries into the more significant bits). (See the implementation of random_init for details.) A \( k+1 \)st number is also used to randomly set one of the least significant bits in the state vector to 1 in the (unlikely) event that they are all zero.

A second state vector can likewise be generated using the next \( k+1 \) numbers from equation \([2]\). Since \([2]\) is a high quality random number generator in its own right, this will be no correlations between the two resulting sequences. Furthermore, the discussion above implies that we are nearly guaranteed that the sequences will have no common points in \( k \)-dimensional space.

In order to obtain a potentially large set of independent random numbers we advance equation \([2]\) forward

\[
L(n_0, n_1, n_2) = g_0n_0 + g_1n_1 + g_2n_2
\]

steps, where \( g_0 = k + 1 = 101, g_1 = 375549701083, \) and \( g_2 = 1396411663216078567733. \) \( g_0 \) is the number of random numbers used by random_init. The other numbers are chosen such that

\[
\frac{g_1}{g_0} \approx \frac{g_2}{g_1} \approx \left( \frac{2^{112}}{1000g_0} \right)^{1/3} \approx 3.7 \times 10^9 > 2^{31}.
\]

This means that with \( n_0, n_1, n_2 \) varying between 0 and a billion we only use 0.1% (another of Knuth’s recommendation) of the numbers with no recycling. Since the coefficients, \( g_0, g_1, \) and \( g_2, \) are relatively prime, it wouldn’t be a big deal to go beyond these limits.

All these routines assume that the machine can handle real quantities with at least 48 bits of precision. On most machines this means double precision. One
2 External interface to random

If you’re using FWEB, the argument seed here always refers to an eight-element integer array which should be declared by \texttt{rn\_seed\_decl(seed)}. seed should not be altered by the user. Similarly the state of the random number generator is stored in \texttt{rn\_args(tag)}, where \texttt{tag} is an arbitrary tag. This should not be altered by the user. \texttt{rn\_decl(tag)} declares \texttt{rn\_args(tag)} and, in a top-level routine, allocates space for them.

If you’re not using FWEB, then replace \texttt{rn\_args(tag)} by the argument pair \texttt{ran\_index}, \texttt{ran\_array} and then declare the seed and the state as follows.

"random.f" 2 ≡

@#if 0
  /* \texttt{ran\_s}, \texttt{ran\_c}, and \texttt{ran\_k} are defined in: */
  include ’random.h’
  /* Seeds are specified by an argument, \texttt{seed}, which should be declared as: */
  integer \texttt{seed(\texttt{ran\_s})}
  /* The external representation of the seed is via a string of decimal digits. This should be declared as: */
  character*(\texttt{\texttt{ran\_c}) string}
  /* The state of the random number generator is stored in two variables which are passed as arguments: \texttt{ran\_index} and \texttt{ran\_array}. These should be declared as: */
  integer \texttt{ran\_index}
  double precision \texttt{ran\_array(\texttt{ran\_k})}
  /* Finally, you may need to declare the external routines \texttt{random} and \texttt{srandom}: */
  double precision \texttt{random}
  real \texttt{srandom}
  external \texttt{random}, \texttt{srandom}
  /* In the above, you will need to change \texttt{double precision} to \texttt{real} on the Crays. */
@#endif
The routines for initializing the seed are:

- **call** set_random_seed(time, seed) set the seed from an 8-element integer time array. Typically, time is set from **call** date_time(time) or using the Fortran 90 routine **call** date_and_time(values = time).

- **call** decimal_to_seed(decimal, seed) initializes the seed array from a decimal number in the character variable decimal. Only the digits in decimal are used, so ‘1999/07/30-18:55:33’ is the same is ‘19990730185533’. There is no limit on the length of decimal.

- **call** string_to_seed(string, seed) initializes the seed array from an arbitrary ASCII string string using a checksum algorithm. Only the printable characters in string are used. There is no limit on the length of string.

- **call** seed_to_decimal(seed, decimal) converted the seed array to a canonical printable representation in the character variable decimal which should be at least ran_c = 34 characters long.

Possible ways of setting seed are:

"random.f" 2.1

@@if(){"include ’random.h’
  integer seed(ran_s)
  integer time(8)
  character*(ran_c) string
    /* Set it based on the time of day */
  **call** date_and_time(values = time)
  **call** set_random_seed(time, seed)
    /* Set it during debugging */
  **call** decimal_to_seed(’0’, seed)
    /* Set it from a decimal number */
  **call** decimal_to_seed(’Run_number:12987’, seed)
    /* Set it from an ASCII string */
  **call** string_to_seed(’Pellet_injection_case_A’, seed)
    /* In all cases you should write out the seed */
  **call** seed_to_decimal(seed, string)
  print *, string
@@endif

Routines for advancing the seed. These are needed when independent random number sequences are required.

- **call** next_seed(n, seed) advances the seed n steps forward. This allows independent threads of random numbers to be initialized. This just invokes **call** next_seed3(n, 0, 0, seed).

- **call** next_seed3(n0, n1, n2, seed) advances the seed (n0,n1,n2) steps forward in 3-dimensional space. The steps along each axis can be up to at least $10^9$ allowing for a total of $10^{27}$ independent threads to be initialized. next_seed3 might be useful for selecting the seeds for a three-dimension domain decomposition.
Routines for initializing the state for random number generator and for returning random numbers.

**call** random_init(seed, rn_args(tag)) which initializes the random number generator from seed.

\[ x = \text{random}(\text{rn_args(tag)}) \] which returns the next random number in \( x \). \( x \) is a **double precision** random number uniformly distributed in \((0, 1)\).

**call** random_array(y, n, rn_args(tag)) which returns the next \( n \) random numbers in the **double precision** array \( y \).

\[ sx = \text{srandom}(\text{rn_args(tag)}) \] which returns the next random number in \( sx \). \( sx \) is a **real** random number uniformly distributed in \((0, 1)\).

**call** srandom_array(sy, n, rn_args(tag)) which returns the next \( n \) random numbers in the **real** array \( sy \).

The random numbers returned by random and random_array are of the form \((i + \frac{1}{2})/2^{47}\) for \(0 \leq i < 2^{47}\). The period of each thread is about \(8.9 \times 10^{43}\). Except for the Crays, the random numbers returned by srandom and srandom_array are of the form \((i + \frac{1}{2})/2^{23}\) for \(0 \leq i < 2^{23}\), with the same period. On the Crays, srandom and srandom_array are identical to random and random_array.
3 Typical calling sequence

The following is a more-or-less realistic example of using these routines in an MPI code. It computes $\pi$ by generating random points in a unit square and counting the number that lie in a circle of radius 1 within this square.

This example uses the Fortran 90 subroutine `date_and_time`. This is present in some recent Fortran 77 implementations (specifically, for Sun and Digital Unix). If it is not available initialize the random seed in some other way.

"random.f" 3 ≡

```
@#if ()

program pirandom
    // Compute pi by counting the number of randomly generated point lying in a circle. See Gropp et al., “Using MPI”, Section 3.7.
    // MPI declarations
    include ‘mpif.h’
    integer id, ierr, len, numprocs
    character*(mpi_max_processor_name) hostname
    // Declarations for random numbers
    include ‘random.h’
    integer seed(ran_s)
    character*(ran_c) string
    integer ran_index
    double precision ran_array(ran_k)
    // For date_and_time routine
    character cdate*8, ctime*10, czone*5
    integer time(8)
    // A place to store random numbers
    integer m
    parameter (m = 1000)
    double precision z(m)
    // Other local variables
    double precision pi
    integer iterations
    integer a, sum, tot, i, j
    // Set iteration count (each iteration computes m/2 points)
    iterations = 10000
    // Initialize MPI
    call mpi_init(ierr)
    call mpi_comm_rank(mpi_comm_world, id, ierr)
    call mpi_comm_size(mpi_comm_world, numprocs, iierr)
    // Determine and print the seed
    if (id == 0) then
        call date_and_time(cdate, ctime, czone, time)
        call set_random_seed(time, seed)
```

§3–§3.1 \#6–\#7  Random numbers

Typical calling sequence

```
call seed_to_decimal(seed, string)
print *, 'An inefficient way to compute \pi via Monte Carlo'
print *, 'Seed is set to', string
endif

// Tell everyone the value of seed
call mpi_bcast(seed, ran_s, mpi_integer, 0, mpi_comm_world, ierr)

// Advance the seed according to the MPI process number
call next_seed(id, seed)

// Initialize the random number generator
call random_init(seed, ran_index, ran_array)

// Determine the number of points lying inside a circle of radius 1
a = 0
do i = (iterations * id) / numprocs, (iterations * (id + 1)) / numprocs - 1
   call random_array(z, m, ran_index, ran_array)
   do j = 1, m, 2
      if (z(j)^2 + z(j + 1)^2 < 1.0 \cdot 10^0) then
         a = a + 1
      end if
   end do
end do

// Print some diagnostic information
call mpi_get_processor_name(hostname, len, ierr)
print *, 'Process', id, 'running on', hostname(1 : len), 'gives', a

// Assemble the results onto process 0
call mpi_reduce(a, sum, 1, mpi_integer, mpi_sum, 0, mpi_comm_world, ierr)

// Node 0 prints the answer.
tot = iterations * m / 2
if (id \equiv 0) then
   \pi = (4.0 \cdot 10^0 \cdot sum) / tot
   print *, '\pi is approximately', 4 * sum, '/', tot, '\approx', \pi
endif

// Terminate MPI
call mpi_finalize(ierr)
stop
@endif
```

The unnamed module.

"random.f" 3.1 =

(Functions and Subroutines 5)
Define constants used in the basic random number generator. \( \text{ran}_k \) is defined in the header file to be 100.

"random.f" 3.2 ≡
\[
\begin{align*}
\text{FILE } & \text{`random.web'} \\
\text{ran}_l & 63 \quad \text{// Set } l \\
\text{ran}_p & 1009 \quad \text{// We use } k \text{ out of } p \text{ numbers} \\
\text{ran}_e & 47 \quad \text{// Bits of precision with double precision} \\
\text{ran}_es & 23 \quad \text{// Bits of precision with single precision} \\
\text{rand}_\text{center}(x) & = x + \text{ran}_\text{ulp}2 \quad \text{// Center returned numbers in } (0,1) \\
\text{if } \neg \text{HIPREC} & \text{// Likewise for } \text{single\_precision} \\
\text{rand}_\text{center}_s(x) & = (\text{int}(\text{ran\_mult} \ast x) + 0.5) \ast \text{ran}_\text{ulps} \\
\text{endif}
\end{align*}
\]

4 Random definitions

"random.f" 4.1 ≡
\[
\begin{align*}
\text{rn\_decl} & \text{ integer} \\
\text{rn\_decls} & \text{ integer} \\
\text{rn\_seed\_decl} & \text{ integer} \\
\text{single\_precision} & \text{ integer} \\
\text{rn\_args}(x) & \text{rn\_index}(x), \text{rn\_array}(x)_0 \\
\text{rn\_dummy}(x) & \text{rn\_index}(x), \text{rn\_array}(x) \\
\text{rn\_decl}(x) & \text{integer} \text{rn\_index}(x) \\
\text{real} & \text{rn\_array}(x)_0:\text{rn\_k\_1} \\
\text{rn\_decls real} & \text{random} \\
\text{external} & \text{random} \\
\text{rn\_copy}(x, y) & \text{rn\_index}(y) = \text{rn\_index}(x) \\
\text{do} & \text{ran\_temp} = 0, \text{rn\_k\_1} \\
& \text{rn\_array}(y)\text{ran\_temp} = \text{rn\_array}(x)\text{ran\_temp} \\
\text{end do} \\
\text{rn\_index}(x) & \text{ran\_index\#x} \quad \text{// Accessor routines} \\
\text{rn\_array}(x) & \text{ran\_array\#x}
\end{align*}
\]
§4.2–§4.3 [#11–#12] Random numbers


"random.f" 4.2 ≡
@m ran_k 100  // The size of the batch of random numbers
@m ran_s 8   // The size of the seed.
@m ran_c 34  // The size of the decimal version of seed.

[u/dstotler/degas2/src/random.hweb] Inline calling routines.

/* An inline version of \( y = \text{random}(\text{rn_args}(x)) \) */

"random.f" 4.3 ≡
@m rn_next(y, x) y = \text{random}(\text{rn_args}(x))
@m rn_init(seed, x) call \text{random_init}(seed, \text{rn_args}(x))
@m rn_seed_copy(x, y) rn_seed_copy1(x, y)
@m rn_seed_copy1(x, y) $DO(I, 0, \text{ran}_s - 1)$
  { 
    y_I = x_I
  }
@m rn_seed_decl x0:ran_s−1
@m rn_seed_args(x) x0
@m rn_iso_next(v, x) call \text{random_isodist}(v, 1, \text{rn_args}(x))
@m rn_cos_next(v, x) call \text{random_cosdist}(v, 1, \text{rn_args}(x))
@m rn_gauss_next(y, x) call \text{random_gauss}(y, 1, \text{rn_args}(x))
5 Functions to return random numbers

We calculate the random numbers in batches, placing the results into the array $ran\_array$. The routine to return a single random number, $random$, can then merely return the next array element. It calls the routine to calculate the next batch if necessary. $ran\_index$ points to the next element of $ran\_array$ to be used. $ran\_max$ points to the first invalid element of $ran\_array$. $ran\_index$ and $ran\_array$ together describe the state of the random number generator. The header file packages these together into a single argument $rn\_args(x)$. $srandom$ is an entry point providing a single precision result.

\begin{verbatim}
(Functions and Subroutines 5) ≡
function random(rn_dummy(x))
  implicit none f77
  implicit none f90
  real ran_ulp2
  parameter (ran_ulp2 = two**-ran_e-1)
@# if ~HIPREC
  single_precision ran_ulps
  real ran_mult
  parameter (ran_ulps = 2.0**-ran_es, ran_mult = two**ran_es)
@# endif
  real random   // Function
  single_precision srandom   // Entry
  rn_decl(x)   // RNG state
  external rand_batch   // External
@# if HIPREC
  entry srandom(rn_dummy(x))
@# endif
  if (rn_index(x) ≥ ran_k) then
    call rand_batch(rn_args(x))
  end if
  random = rand_center(ran_array(x)rn_index(x))
@# if HIPREC
  srandom = random
@# endif
  rn_index(x) = rn_index(x) + 1
  return
@# if ~HIPREC
  entry srandom(rn_dummy(x))
  if (rn_index(x) ≥ ran_k) then
    call rand_batch(rn_args(x))
  end if
  srandom = rand_center_s(ran_array(x)rn_index(x))
  rn_index(x) = rn_index(x) + 1
  return
@# endif
end
\end{verbatim}

See also sections 5.1, 6, 7.1, 7.3, 8, 8.1, 8.2, 9, 10, 10.1, and 10.2.

This code is used in section 3.1.
Here is a version of \texttt{random} which fills an array. This is perhaps more efficient than separate calls to \texttt{random}. \texttt{srandom\_array} is an entry point providing a single precision result.

\begin{verbatim}
(Fun})
@#if ~HIPREC
!
subroutine random_array(y, n, rn\_dummy(x))
  ! Input
  ! Output
  integer n // Input
  real y(0 : n - 1) // Output
@#if ~HIPREC
  ! Output
  integer i, k, j // Local
  external rand\_batch // External
@#if HIPREC
  entry srandom\_array(y, n, rn\_dummy(x))
@#end

if (n \leq 0)
  return
k = min(n, ran\_k - rn\_index(x))
do i = 0, k - 1
  y_i = rand\_center(rn\_array(x)i+rn\_index(x))
end do
rn\_index(x) += k
do j = k, n - 1, ran\_k
  call rand\_batch(rn\_args(x))
do i = j, min(j + ran\_k, n) - 1
  y_i = rand\_center(rn\_array(x)\_i-j+rn\_index(x))
end do
rn\_index(x) += min(ran\_k, n - j)
end do
return
@#if ~HIPREC

entry srandom\_array(ys, n, rn\_dummy(x))
if (n \leq 0)
  return
k = min(n, ran\_k - rn\_index(x))
do i = 0, k - 1
  ys_i = rand\_center\_s(rn\_array(x)i+rn\_index(x))
end do
rn\_index(x) += k
do j = k, n - 1, ran\_k
\end{verbatim}
§5.1 [14] Random numbers

Functions to return random numbers

```c
    call rand_batch(rn_args(x))
    do i = j, min(j + ran_k, n) - 1
        ys_i = rand_center_s(rn_array(x)i-j+rn_index(x))
    end do
    rn_index(x) += min(ran_k, n - j)
    end do
    return
@#endif
end
```
§6 [15] Random numbers

6 Calculate the next batch

The fills the next $k$ elements of $rn\text{-}array(x)$. This uses an local array $w$ to skip over $p - k$ elements. For the implementation given here to work, we require $p \geq 2k$, which is, of course, satisfied for the parameters we use here.

On some machines it is better to implement the operation of taking the fractional part with a conditional. Both of these methods are equivalent since $0 \leq y < 2$.

"random.f" 6 \equiv
@#if USEINT
@m ran_assign(x, y) tmp = y
x = tmp - int(tmp)
@#else
@m ran_assign(x, y) tmp = y
if (tmp \geq one) then
x = tmp - one
else
x = tmp
end if
@endif

⟨Functions and Subroutines 5 ⟩\equiv
subroutine rand_batch(rn\_dummy(x))
  implicit none f77
  implicit none f90
  rn\_decl(x)  // RNG state
  integer i  // Local
  real w0, ran\_p - ran\_k - 1
  real tmp
   /* Sanity check on rn\_index(x). */
  assert(rn\_index(x) \equiv ran\_k)
  do i = 0, ran\_l - 1
    ran\_assign(wi, rn\_array(x)i + rn\_array(x)i+ran\_k-ran\_l)
  end do
  do i = ran\_l, ran\_k - 1
    ran\_assign(wi, rn\_array(x)i + wi-ran\_l)
  end do
  do i = ran\_k, ran\_p - ran\_k - 1
    ran\_assign(wi, wi-ran\_k + wi-ran\_l)
  end do
  do i = ran\_p - ran\_k, ran\_p - ran\_k + ran\_l - 1
    ran\_assign(rn\_array(x)i-ran\_p+ran\_k, wi-ran\_k + wi-ran\_l)
  end do
  do i = ran\_p - ran\_k + ran\_l, ran\_p - 1
    ran\_assign(rn\_array(x)i-ran\_p+ran\_k, wi-ran\_k + rn\_array(x)i-ran\_p+ran\_k-ran\_l)
  end do
  rn\_index(x) = 0
  return
end
7 Initializing the random number sequence

This initializes the random number array using the high 47 bits of successive numbers from a linear
congruential generator defined by equation 2.

The 112-bit numbers can be represented as an array of 8 14-bit integers which allows the multiplications
and additions necessary to perform 112-bit multiplies to be carried out on 32-bit machines. (We don’t use
15-bit integers since it is then necessary to do the multiplies more carefully.)
Here is random_init. This iterates $T$ 100 times. We might use the 101st iterate to turn on one of the least-significant bits in the unlikely event that they are all zero. We set $rn_array(x)$ from the $ran_e$ most significant bits of $s$. We place the most significant (i.e., the most random) byte of $s$ in the least significant position in $rn_array(x)$ where it can do the most good; and so on with the other bytes; finally we place the five most significant bits from the least significant byte in the five most significant bit positions in $rn_array(x)$.

The main loop here does not vectorize. It’s possible to remedy this by first jumping forward by multiples of ten using the tenth power of $T$ and then initializing the other elements in a parallel fashion. This results in a modest speed up on vector machines but slows down the running time on other machines; so we choose not to do this here.

"random.f" 7.1 ≡

@m ran_set(i) $rn_array(x)_i = (((s_7 * ran_del + s_6) * ran_del + s_5) * ran_del + int(s_4 / 512)) * 512 * ran_del$

 FUNCTIONS AND SUBROUTINES 

subroutine random_init(seed, $rn_dummy(x)$)
    implicit none f77
    implicit none f90
    integer b
    real ran_del, ran_ulp
    parameter (b = 2**14, ran_del = two**-14, ran_ulp = two**-ran_e)
    integer a0, a1, a2, a3, a4, a5, a6, c0  // The base 2**14 representation of a1 and c1
    parameter (a0 = 15661, a1 = 678, a2 = 724, a3 = 5245, a4 = 13656, a5 = 11852, a6 = 29)
    parameter (c0 = 1)
    $rn_seed_decl(seed)  // Input
    $rn_decl(x)  // Output
    integer i, j, s:ran_s-1  // Local
    logical odd  // At least one seed is odd so far
    integer z:ran_s-1, t
    do i = 0, ran_s - 1
        assert (0 <= seed_i ∧ seed_i < b)
        s_i = seed_i
    end do

    odd = mod(s_7, 2) ≠ 0
    ran_set(0)
    do j = 1, ran_k - 1
        (Step the linear congruential generator forward one step 7.2)
        odd = odd ∨ (mod(s_7, 2) ≠ 0);
        ran_set(j)
    end do
    $rn_index(x) = ran_k$

    /* If the least significant bit of all the seeds is zero, then randomly set one of them to one. This is very unlikely to happen. */

    if (odd)
        return
    end

    (Step the linear congruential generator forward one step 7.2)
    j = int((s_ran_s-1 * ran_k) / b)
    $rn_array(x)_j += ran_ulp$
    return
end
Here we step the linear congruential generator forward one step. We open code this to improve the speed somewhat. (We use the fact that the highest byte of a zero as are all but the lowest byte of $c$.)

\[
\begin{align*}
    z_0 &= c_0 + a_0 \cdot s_0 \\
    z_1 &= a_0 \cdot s_1 + a_1 \cdot s_0 \\
    z_2 &= a_0 \cdot s_2 + a_1 \cdot s_1 + a_2 \cdot s_0 \\
    z_3 &= a_0 \cdot s_3 + a_1 \cdot s_2 + a_2 \cdot s_1 + a_3 \cdot s_0 \\
    z_4 &= a_0 \cdot s_4 + a_1 \cdot s_3 + a_2 \cdot s_2 + a_3 \cdot s_1 + a_4 \cdot s_0 \\
    z_5 &= a_0 \cdot s_5 + a_1 \cdot s_4 + a_2 \cdot s_3 + a_3 \cdot s_2 + a_4 \cdot s_1 + a_5 \cdot s_0 \\
    z_6 &= a_0 \cdot s_6 + a_1 \cdot s_5 + a_2 \cdot s_4 + a_3 \cdot s_3 + a_4 \cdot s_2 + a_5 \cdot s_1 + a_6 \cdot s_0 \\
    z_7 &= a_0 \cdot s_7 + a_1 \cdot s_6 + a_2 \cdot s_5 + a_3 \cdot s_4 + a_4 \cdot s_3 + a_5 \cdot s_2 + a_6 \cdot s_1 \\
\end{align*}
\]

\[
t = 0 \\
\text{do } i = 0, \text{ ran}_{-s} - 1 \\
    t = \text{int}(t / b) + z_i \\
    s_i = \text{mod}(t, b) \\
\text{end do}
\]

This code is used in section 7.1.
Here’s the vectorizing version of \textit{random.init}. It’s commented out since it’s of marginal utility and it’s more complicated.

"random.f" 7.3 ≡
@m ran_k1 10 // Factor $k = k_1 k_2$.
@m ran_k2 10

⟨Functions and Subroutines 5⟩ +≡
@#if 0

\textbf{subroutine} random\_init(seed, \textit{rn} \_\textit{dummy}(x))
\begin{verbatim}
  implicit_none,f77
  implicit_none,f90
  integer b

  real ran\_del, ran\_ulp

  parameter (ran\_del = two\textsuperscript{-14}, ran\_ulp = two\textsuperscript{-ran\_e}, b = 2\textsuperscript{14})

  \textit{rn}\_\textit{seed} \_\textit{decl}(seed) // Input

  \textit{rn}\_\textit{decl}(x) // Output

  integer i, j, k, s0: ran\_s - 1, 0: ran\_k1 - 1 // Local

  logical odd: ran\_k1 - 1 // At least one seed is odd so far

  integer a0, a1, a2, a3, a4, a5, a6, c0 // The base 2\textsuperscript{14} representation of a1 and c1

  data a0, a1, a2, a3, a4, a5, a6, c0 = 15661, 678, 724, 5245, 13656, 11852, 29, 1/

  integer ak2: ran\_s - 1, ck2: ran\_s - 1 // $T_{k2}$

  data ak2 = 11273, 6813, 10723, 15580, 4310, 5243, 1020, 6133/

  data ck2 = 13766, 11135, 9779, 14128, 13492, 4815, 11925, 2137/

  do i = 0, ran\_s - 1
    assert (0 \leq seed \_i \land seed \_i < b)
    seed \_i = seed \_i
  end do

  do k = 1, ran\_k1 - 1
    do i = 0, ran\_s - 1
      seed \_i, k = seed \_i, k - 1
    end do

    \textit{call} rand\_\textit{axc}(ak2, s0, k, ck2)
  end do

  do k = 0, ran\_k1 - 1
    odd \_k = (mod(s7, k, 2) \neq 0)

    \textit{rn}\_\textit{array}(x) \_k: ran\_k2 = (((s7, k \_ran\_del + s6, k \_ran\_del + s5, k \_ran\_del + \textit{int}(s4, k/512)) \_512 \_ran\_del}
  end do

  do j = 1, ran\_k2 - 1
    do k = 0, ran\_k1 - 1
      seed \_k = c0 + a0 \_s0, k
      seed \_k = a0 \_s1, k + a1 \_s0, k
      seed \_k = a0 \_s2, k + a1 \_s1, k + a2 \_s0, k
      seed \_k = a0 \_s3, k + a1 \_s2, k + a2 \_s1, k + a3 \_s0, k
      seed \_k = a0 \_s4, k + a1 \_s3, k + a2 \_s2, k + a3 \_s1, k + a4 \_s0, k
      seed \_k = a0 \_s5, k + a1 \_s4, k + a2 \_s3, k + a3 \_s2, k + a4 \_s1, k + a5 \_s0, k
      seed \_k = a0 \_s6, k + a1 \_s5, k + a2 \_s4, k + a3 \_s3, k + a4 \_s2, k + a5 \_s1, k + a6 \_s0, k
      seed \_k = a0 \_s7, k + a1 \_s6, k + a2 \_s5, k + a3 \_s4, k + a4 \_s3, k + a5 \_s2, k + a6 \_s1, k
    end do

    \textit{tk} = 0
  end do
\end{verbatim}

\textbf{end subroutine}
do $i = 0$, $ran_s - 1$
  do $k = 0$, $ran_k1 - 1$
    $t_k = \text{int}(t_k / b) + z_i, k$
    $s_i, k = \text{mod}(t_k, b)$
  end do
end do

do $k = 0$, $ran_k1 - 1$
  $\text{odd}_k = \text{odd}_k \lor (\text{mod}(s_7, k, 2) \neq 0)$
  $\text{rn\_array}(x)_{k*ran_k2+j} = (((s_7, k*ran\_del + s_6, k)*ran\_del + s_5, k)*ran\_del + \text{int}(s_4, k / 512)) * 512 * ran\_del$
end do

$\text{rn\_index}(x) = \text{ran\_k}$

/* If the least significant bit of all the seeds is zero, then randomly set one of them to one. This is very unlikely to happen. */

do $k = 0$, $ran_k1 - 1$
  if ($\text{odd}_k$)
    return
  end if
end do

$\text{call \ rand\_axc}(ak2, s_0, 0, ck2)$

$j = \text{int}((s_{\text{ran\_s-1, 0}} * \text{ran\_k}) / b)$

$\text{rn\_array}(x)_j += \text{ran\_ulp}$

return
8 Initializing the seed

Set the seed from an decimal string. seed is initialized to zero. Digits 0–9 cause the operation seed = 10 * seed + digit. Other characters are ignored.

\begin{verbatim}
subroutine decimal_to_seed(decimal, seed)
    implicit none_f77
    implicit none_f90
    character(*) decimal
    rn_seed_decl(seed)
    external rand_axc
    integer i, ten0:ran_s−1, c0:ran_s−1, ch
    data ten/10, 7*0/
    do i = 0, ran_s − 1
        seed_i = 0
        c_i = 0
    end do
    do i = 1, len(decimal)
        ch = icchar(decimal(i : i))
        if (ch ≥ icchar('0') ∧ ch ≤ icchar('9')) then
            c_0 = ch - icchar('0')
            call rand_axc(ten, seed, c)
        end if
    end do
    return
end
\end{verbatim}
Set the seed from an arbitrary ASCII string. Only printable characters are considered. Each such character causes the operation \( seed = \text{rotr}(seed) + \text{char} \), where \( \text{rotr} \) is a circular right shift by one bit; this is the operation carried out by the BSD sum utility.

\[
\text{subroutine string_to_seed}(\text{string}, \ seed) \\
\text{implicit none}_\text{f77} \\
\text{implicit none}_\text{f90} \\
\text{integer} \ b \\
\text{parameter} (b = 2^{14}) \\
\text{character}(\_*) \text{ string} \\
\text{rn_seed_decl}(\ seed) \\
\text{external} \ rand\_axc \\
\text{integer} \ t, i, k, \text{ unity: ran_s-1, c0: ran_s-1}, \text{ ch} \\
\text{data} \ \text{ unity}/1, 7*0/ \\
\text{do} \ i = 0, \ \text{ ran_s-1} \\
\quad \text{seed}_i = 0 \\
\quad c_i = 0 \\
\text{end do} \\
\text{do} \ i = 1, \ \text{len} \ (\text{string}) \\
\quad ch = \text{ichar} \ (\text{string} \ (i : i)) \\
\quad \text{if} \ (\text{ch} > \text{ichar} \ ('\_') \land \text{ch} < 127) \ \text{then} \\
\quad \quad t = \text{mod} \ (\text{seed}_0, \ 2) \ast (b / 2) \\
\quad \quad \text{do} \ k = 0, \ \text{ ran_s-1} \\
\quad \quad \quad \text{seed}_k = \text{int} \ (\text{seed}_k / 2) \\
\quad \quad \quad \text{if} \ (k < \text{ ran_s-1}) \ \text{then} \\
\quad \quad \quad \quad \text{seed}_k += \text{mod} \ (\text{seed}_{k+1}, \ 2) \ast (b / 2) \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{seed}_k += t \\
\quad \quad \text{end if} \\
\quad \text{end do} \\
\quad c_0 = \text{ch} \\
\quad \text{call rand\_axc} (\text{unity}, \ \text{seed}, \ c) \ // \ \text{else the character is whitespace and we skip it.} \\
\text{end if} \\
\text{end do} \\
\text{return} \\
\text{end}
Choose random seed depending on time. *time* is the 8-element time returned by *date_time* (it can also be obtained in Fortran 90 with `call date_and_time(values = time)`). The seed is set to `yyyymmddzzzzhhmmssmmm` where `yyyy` is the year, etc. The first digit of `zzzz` is 1 if the zone offset is negative.

"random.f" 8.2 ≡

```fortran
$UNQUOTE('␣')

⟨Functions and Subroutines 5⟩ +=

```fortran
subroutine set_random_seed(time, seed)
  implicit none f77
  implicit none f90
  integer time // Input
  rn_seed_decl(seed) // Output
  character*26 c // Local
  integer t8
  external decimal_to_seed // make sure time is in range

  t1 = mod(mod(time1, 1000000000) + 1000000000, 1000000000)
  t2 = mod(mod(time2, 100) + 100, 100)
  t3 = mod(mod(time3, 100) + 100, 100)
  t4 = ((1 - sign(1, time4))/2) * 1000 + mod(abs(time4), 1000)
  t5 = mod(mod(time5, 100) + 100, 100)
  t6 = mod(mod(time6, 100) + 100, 100)
  t7 = mod(mod(time7, 100) + 100, 100)
  t8 = mod(mod(time7, 1000) + 1000, 1000)
  c = '␣'
  write(c(1 : 17), '('i9.9,2i2.2,i4.4')') t1, t2, t3, t4
  write(c(18 : 26), '('i2.2,i3.3')') t5, t6, t7, t8
  call decimal_to_seed(c, seed)

  return
end
```

9 Convert a seed to its decimal equivalent

Decimals should be at least 34 characters long; otherwise the high digits are lost. Internal conversion is done to a base of 10000.

(Functions and Subroutines 5) +≡

```
subroutine seed_to_decimal(seed, decimal)
  implicit none_f77
  implicit none_f90
  integer pow, decbase, b
  parameter(pow = 4, decbase = 10^p^ow, b = 2^14)
  character(*) decimal
  fn_seed_decl(seed)
  integer z0, ran_s-1, i, t, j, k
  character*36 str
  k = -1
  do i = 0, ran_s - 1
    zi = seed_i
    assert(0 ≤ zi ∧ zi < b)
    if (zi > 0)
      k = i
  end do
  str = ' 'u'
  /* 2^112 < 10^36 */
  i = 9
  loop: continue
  i--; t = 0
  do j = k, 0, -1
    zj = zj + t * b
    t = mod(zj, decbase)
    zj = int(zj / decbase)
  end do
  if (zmax(0, k) ≡ 0)
    k = k - 1
  j = pow * (i + 1)
  if (k ≥ 0) then
    str(j - (pow - 1) : j) = '0000'
  else
    str(j - (pow - 1) : j) = ' 'u00'
  end if
  loop1: continue
  if (t ≡ 0)
    goto skip
  str(j : j) = char(ichar('0') + mod(t, 10))
  j = j - 1
  t = int(t / 10)
  goto loop1
skip: continue
```
if \( k \geq 0 \)  
    goto loop  
\[ k = \min(j + 1, \ len(str)) \]

if \( \text{len}(\text{decimal}) \geq \text{len}(str(k:)) \) then  
    \( \text{decimal} = str(k : ) \)
else  
    \( \text{decimal} = str(\text{len}(str(k : )) - \text{len}(\text{decimal}) + 1 : ) \)
end if
return
end
10 Routines to step the seed forward

*rand_next_seed* steps the seed *y* forward *n* multiples of the generator specified by *ax* and *cx*. *n* is decomposed in binary and *y* is stepped forward by powers of 2. We do this by composing equation (2) with itself to give

\[ T^2(x) \equiv (a_2x + c_2) \mod 2^{112}. \]

where \( a_2 = a_1^2 \mod 2^{112} \) and \( c_1 = (a_1 + 1)c_1 \mod 2^{112} \). The time is proportional to \( \log n \).

```fortran
subroutine rand_next_seed(n, ax, cx, y)
  implicit none
  integer n, ax:ran_s-1, cx:ran_s-1   // Input
  external rand_axc
  integer a0:ran_s-1, c0:ran_s-1, z0:ran_s-1, t0:ran_s-1, m, i
  data z/ran_s*0/
  if (n == 0)
    return
  assert(n > 0)
  m = n
  do i = 0, ran_s - 1
    ai = axi
    c_i = cx_i
  end do
  loop: continue
  if (mod(m, 2) > 0) then
    call rand_axc(a, y, c)
  end if
  m = int(m / 2)
  if (m == 0)
    return
  do i = 0, ran_s - 1
    ti = ci
  end do
  call rand_axc(a, c, t)  // \( c' = (a + 1)c \)
  do i = 0, ran_s - 1
    ti = ai
  end do
  call rand_axc(t, a, z)  // \( a' = a^2 \)
  goto loop
end
```
next_seed3 steps the seed forward by \( L(n_0, n_1, n_2) \) steps from equation 3. In this routine \( a_fk \) and \( c_fk \) are precomputed coefficients for equation 2 iterated \( g_k \) times (i.e., forwards) and \( a_bk \) and \( c_bk \) are precomputed coefficients for equation 2 iterated \(-g_k\) times (i.e., backwards). This routine works but invoking \( \text{rand.next.seed} \) with the appropriate coefficients for each of the \( n_k \).

next_seed is an entry point where only \( n_0 \) is passed.

\[
\begin{align*}
\text{subroutine next_seed3}(n0, n1, n2, seed) \\
\text{implicit none f77} \\
\text{implicit none f90} \\
\text{integer n0, n1, n2} \quad // \text{ Input} \\
\text{rn_seed_decl(seed)} \quad // \text{ Input/Output} \\
\text{external rand.next_seed} \\
\text{integer a}_f0:0: \text{ran.s}−1, c_f0:0: \text{ran.s}−1 \\
\text{integer ab}_0:0: \text{ran.s}−1, cb_0:0: \text{ran.s}−1 \\
\text{integer a}_f1:0: \text{ran.s}−1, c_f1:0: \text{ran.s}−1 \\
\text{integer ab}_1:0: \text{ran.s}−1, cb_1:0: \text{ran.s}−1 \\
\text{integer a}_f2:0: \text{ran.s}−1, c_f2:0: \text{ran.s}−1 \\
\text{integer ab}_2:0: \text{ran.s}−1, cb_2:0: \text{ran.s}−1 \\
\text{data a}_f0/15741, 8689, 9280, 4732, 12011, 7130, 6824, 12302/ \\
\text{data c}_f0/16317, 10266, 1198, 3331, 10769, 8310, 2779, 13880/ \\
\text{data ab}_0/9173, 9894, 15203, 15379, 7981, 2280, 8071, 429/ \\
\text{data cb}_0/8338, 3616, 597, 12724, 15663, 9639, 187, 4866/ \\
\text{data a}_f1/8405, 4808, 3603, 6718, 13766, 9243, 10375, 12108/ \\
\text{data c}_f1/13951, 7170, 9039, 11206, 8706, 14101, 1864, 15191/ \\
\text{data ab}_1/6209, 3240, 9759, 7130, 15320, 14399, 3675, 1380/ \\
\text{data cb}_1/15357, 5843, 6205, 16275, 8838, 12132, 2198, 10330/ \\
\text{data a}_f2/445, 10754, 1869, 6593, 385, 12498, 14501, 7383/ \\
\text{data c}_f2/2285, 8057, 3864, 10235, 1805, 10614, 9615, 15522/ \\
\text{data ab}_2/405, 4903, 2746, 1477, 3263, 13564, 8139, 2362/ \\
\text{data cb}_2/8463, 575, 5876, 2220, 4924, 1701, 9060, 5639/ \\
\end{align*}
\]

if \( (n2 > 0) \) then
  call rand.next_seed\( n2, a_f2, c_f2, seed \)
else if \( (n2 < 0) \) then
  call rand.next_seed\( -n2, a_b2, c_b2, seed \)
end if

if \( (n1 > 0) \) then
  call rand.next_seed\( n1, a_b1, c_b1, seed \)
else if \( (n1 < 0) \) then
  call rand.next_seed\( -n1, a_b1, c_b1, seed \)
end if

entry next_seed\( n0, seed \)

if \( (n0 > 0) \) then
  call rand.next_seed\( n0, a_f0, c_f0, seed \)
else if \( (n0 < 0) \) then
  call rand.next_seed\( -n0, a_b0, c_b0, seed \)
end if

return
end
A utility routine. Implement \( x = a \times x + c \).

\[ \langle \text{Functions and Subroutines} \, 5 \rangle + \equiv \]

**subroutine** randoxc \((a, \, x, \, c)\)

\[
\begin{align*}
\text{implicit none}_{\text{f77}} \\
\text{implicit none}_{\text{f90}} \\
\text{integer} \; b \\
\text{parameter} \; (b = 2^{14}) \\
\text{integer} \; a_{: ran_s - 1}, \; c_{: ran_s - 1} \; \text{ // Input} \\
\text{integer} \; x_{: ran_s - 1} \; \text{ // Input/output} \\
\text{integer} \; z_{: ran_s - 1}, \; i, \; j, \; t \; \text{ // Local}
\end{align*}
\]

\[
\begin{align*}
do \; i = 0, \; \text{ran}_s - 1 \\
& \quad z_i = c_i 
\end{align*}
\]

/* These do loops were originally do \( i = 0, \; \text{ran}_s - 1 \); do \( j = 0, \; i \) and the assignment of \( z \) above was inside the outer loop. The order of the do loops has been interchanged, so that the inner one can vectorize. */

\[
\begin{align*}
do \; j = 0, \; \text{ran}_s - 1 \\
do \; i = j, \; \text{ran}_s - 1 \\
& \quad z_i + = a_j \times x_{i - j}
\end{align*}
\]

\end do

\[
\begin{align*}
t = 0 \\
do \; i = 0, \; \text{ran}_s - 1 \\
& \quad t = \text{int}(t / b) + z_i \\
& \quad x_i = \text{mod}(t, \; b)
\end{align*}
\]

\end do

\]

return

end
11 INDEX

\[
\begin{align*}
\text{a:} & \quad 3, 10, 10.2 \\
\text{abk:} & \quad 10.1. \\
\text{abs:} & \quad 8.2. \\
\text{ab0:} & \quad 10.1. \\
\text{ab1:} & \quad 10.1. \\
\text{ab2:} & \quad 10.1. \\
\text{afk:} & \quad 10.1. \\
\text{af0:} & \quad 10.1. \\
\text{af1:} & \quad 10.1. \\
\text{af2:} & \quad 10.1. \\
\text{ak2:} & \quad 7.3. \\
\text{assert:} & \quad 6, 7.1, 7.3, 9, 10. \\
\text{ax:} & \quad 10. \\
\text{a0:} & \quad 7.1, 7.2, 7.3. \\
\text{a1:} & \quad 7.1, 7.2, 7.3. \\
\text{a2:} & \quad 7.1, 7.2, 7.3. \\
\text{a3:} & \quad 7.1, 7.2, 7.3. \\
\text{a4:} & \quad 7.1, 7.2, 7.3. \\
\text{a5:} & \quad 7.1, 7.2, 7.3. \\
\text{a6:} & \quad 7.1, 7.2, 7.3. \\
\text{b:} & \quad 7.1, 7.3, 8.1, 9, 10.2. \\
\text{c:} & \quad 8, 8.1, 8.2, 10, 10.2. \\
\text{cbk:} & \quad 10.1. \\
\text{cb0:} & \quad 10.1. \\
\text{cb1:} & \quad 10.1. \\
\text{cb2:} & \quad 10.1. \\
\text{cdate:} & \quad 3. \\
\text{cfk:} & \quad 10.1. \\
\text{cf0:} & \quad 10.1. \\
\text{cf1:} & \quad 10.1. \\
\text{cf2:} & \quad 10.1. \\
\text{ch:} & \quad 8, 8.1. \\
\text{char:} & \quad 8.1, 9. \\
\text{ck2:} & \quad 7.3. \\
\text{ctime:} & \quad 3. \\
\text{cx:} & \quad 10. \\
\text{czone:} & \quad 3. \\
\text{c0:} & \quad 7.1, 7.2, 7.3. \\
\text{date_and_time:} & \quad 2.1, 3, 8.2. \\
\text{date_time:} & \quad 2.1, 8.2. \\
\text{decbase:} & \quad 9. \\
\text{decimal:} & \quad 2.1, 8, 9. \\
\text{decimal_to_seed:} & \quad 2.1, 8, 8.2. \\
\text{digit:} & \quad 8. \\
\text{FILE:} & \quad 3.2. \\
\text{HIPREC:} & \quad 3.2, 5, 5.1. \\
\text{hostname:} & \quad 3. \\
\text{i:} & \quad 3, 5.1, 6, 7.1, 7.3, 8, 8.1, 9, 10, 10.2. \\
\text{ichar:} & \quad 8, 8.1, 9. \\
\text{id:} & \quad 3. \\
\text{ ierr:} & \quad 3. \\
\text{ ierr:} & \quad 3. \\
\text{implicit_none_f77:} & \quad 5, 5.1, 6, 7.1, 7.3, 8, 8.1, 8.2, 9, 10, 10.1, 10.2. \\
\text{implicit_none_f90:} & \quad 5, 5.1, 6, 7.1, 7.3, 8, 8.1, 8.2, 9, 10, 10.1, 10.2. \\
\text{include:} & \quad 2, 2.1, 3. \\
\text{int:} & \quad 3.2, 6, 7.1, 7.2, 7.3, 8.1, 9, 10, 10.2. \\
\text{integer:} & \quad 4.1. \\
\text{iterations:} & \quad 3. \\
\text{j:} & \quad 3, 5.1, 7.1, 7.3, 9, 10.2. \\
\text{k:} & \quad 5.1, 7.3, 8.1, 9. \\
\text{len:} & \quad 3, 8, 8.1, 9. \\
\text{loop:} & \quad 9, 10. \\
\text{loop1:} & \quad 9. \\
\text{m:} & \quad 3, 10. \\
\text{max:} & \quad 9. \\
\text{min:} & \quad 5.1, 9. \\
\text{mod:} & \quad 7.1, 7.2, 7.3, 8.1, 8.2, 9, 10, 10.2. \\
\text{mpi_bcast:} & \quad 3. \\
\text{mpi_comm_rank:} & \quad 3. \\
\text{mpi_comm_size:} & \quad 3. \\
\text{mpi_comm_world:} & \quad 3. \\
\text{mpi_finalize:} & \quad 3. \\
\text{mpi_get_processor_name:} & \quad 3. \\
\text{mpi_init:} & \quad 3. \\
\text{mpi_integer:} & \quad 3. \\
\text{mpi_max_processor_name:} & \quad 3. \\
\text{mpi_reduce:} & \quad 3. \\
\text{mpi_sum:} & \quad 3. \\
\text{n:} & \quad 5.1, 10. \\
\text{next_seed:} & \quad 2.2, 3, 10.1. \\
\text{next_seed3:} & \quad 2.2, 10.1. \\
\text{numprocs:} & \quad 3. \\
\text{n0:} & \quad 2.2, 10.1. \\
\text{n1:} & \quad 2.2, 10.1. \\
\text{n2:} & \quad 2.2, 10.1. \\
\text{odd:} & \quad 7.1, 7.3. \\
\text{one:} & \quad 6. \\
\text{pi:} & \quad 3. \\
\text{pirandom:} & \quad 3. \\
\text{pow:} & \quad 9. \\
\text{ran_array:} & \quad 1, 2, 3, 5. \\
\text{ran_array:} & \quad 4.1. \\
\text{ran_assign:} & \quad 6. 
\end{align*}
\]
ran_c: 2, 2.1, 3, 4.2
ran_del: 7.1, 7.3
ran_e: 3.2, 5, 5.1, 7.1, 7.3.
ran_es: 3.2, 5, 5.1.
ran_index: 2, 3, 5.
ran_index: 4.1.
ran_k: 2, 3, 3.2, 4.2, 4.3, 5, 5.1, 6, 7.1, 7.3.
ran_k1: 7.3.
ran_k2: 7.3.
ran_max: 5.
ran_mult: 3.2, 5, 5.1.
ran_p: 3.2, 6.
ran_s: 2, 2.1, 3, 4.2, 4.3, 7.1, 7.2, 7.3, 8, 8.1, 9, 10, 10.1, 10.2.
ran_set: 7.1.
ran_temp: 4.1.
ran_ups: 7.1, 7.3.
ran_ups2: 3.2, 5, 5.1.
rand_axc: 7.3, 8, 8.1, 10, 10.2.
rand_batch: 5, 5.1, 6.
rand_center: 3.2, 5, 5.1.
rand_center_s: 3.2, 5, 5.1.
rand_next_seed: 10, 10.1.
random: 1, 2, 2.3, 4.1, 4.3, 5, 5.1.
random_array: 1, 2.3, 3, 5.1.
random_cosdist: 4.3.
random_gauss: 4.3.
random_init: 1, 2.3, 3, 4.3, 7.1, 7.3.
random_isodist: 4.3.
rm_args: 2, 2.3, 4.1, 4.3, 5, 5.1.
rm_array: 4.1, 5, 5.1, 6, 7.1, 7.3.
rm_copy: 4.1.
rm_cos_next: 4.3.
rm_decl: 2, 4.1.
rm_decls: 4.1.
rm_dummy: 4.1, 5, 5.1, 6, 7.1, 7.3.
rm_gauss: 4.3.
rm_gauss_next: 4.3.
rm_index: 4.1, 5, 5.1, 6, 7.1, 7.3.
rm_init: 4.3.
rm_iso_next: 4.3.
rm_next: 4.3.
rm_seed_args: 4.3.
rm_seed_copy: 4.3.
rm_seed_copy1: 4.3.
rm_seed_decl: 2, 4.1, 4.3.
rotr: 8.1.
s: 7.1, 7.3.
seed: 2, 2.1, 2.2, 2.3, 3, 4.3, 7.1, 7.3, 8, 8.1, 8.2, 9, 10.1.
seed_to_decimal: 2.1, 3, 9.
set_random_seed: 2.1, 3, 8.2.
sign: 8.2.
single_precision: 3.2, 4.1.
skip: 9.
SP: 8.2.
srandom: 2, 2.3, 5.
srandom_array: 2.3, 5.1.
str: 9.
string: 2, 2.1, 3, 8.1.
string_to_seed: 2.1, 8.1.
sun: 3.
sz: 2.3.
sy: 2.3.
t: 7.1, 7.3, 8.1, 8.2, 9, 10, 10.2.
tag: 2, 2.3.
ten: 8.
time: 2.1, 3, 8.2.
tmp: 6.
tot: 3.
two: 5, 5.1, 7.1, 7.3.
unity: 8.1.
values: 2.1, 8.2.
w: 6.
x: 4.3, 10.2.
y: 5.1.
ys: 5.1.
yyyy: 8.2.
yyyyMMddzzzzhhmmssmm: 8.2.
zzzz: 8.2.
⟨Functions and Subroutines 5, 5.1, 6, 7.1, 7.3, 8, 8.1, 8.2, 9, 10, 10.1, 10.2⟩ Used in section 3.1.
⟨Step the linear congruential generator forward one step 7.2⟩ Used in section 7.1.

COMMAND LINE: "fweave -f -i! -W[ -ykw700 -ytw40000 -j -n/ /u/dstotler/degas2/src/random.web".
WEB FILE: "/u/dstotler/degas2/src/random.web".
CHANGE FILE: (none).
GLOBAL LANGUAGE: Fortran.