

# Random numbers

November 24, 2009  
13:32

## Contents

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

# 1 Introduction

\$Id: random.web,v 1.13 2000/04/01 17:33:30 dstotler Exp \$

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

$$X_n = (X_{n-l} + X_{n-k}) \bmod 1, \quad (1)$$

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 accomodate 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  $2^{e-1}(2^k - 1) \approx 8.9 \times 10^{43}$ . 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  $10^{685}$  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

$$T(x) \equiv (a_1x + c_1) \bmod 2^{112}, \quad (2)$$

We choose  $a_1 = 31167285 \times 2^{64} + 6364136223646793005$ ,  $c_1 = 1$ . The multiplier  $a_1$  is obtained by concatenating two “good” multipliers  $31167285 \bmod 2^{48}$  and  $6364136223646793005 \bmod 2^{64}$  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 coverage 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 be 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

$$L(n_0, n_1, n_2) = g_0n_0 + g_1n_1 + g_2n_2 \quad (3)$$

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 `rn_seed_decl(seed)`. *seed* should not be altered by the user. Similarly the state of the random number generator is stored in `rn_args(tag)`, where *tag* is an arbitrary tag. This should not be altered by the user. `rn_decl(tag)` declares `rn_args(tag)` and, in a top-level routine, allocates space for them.

If you're not using FWEB, then replace `rn_args(tag)` by the argument pair `ran_index, ran_array` and then declare the seed and the state as follows.

```
"random.f" 2 ≡

@#if 0
    /* ran_s, ran_c, and ran_k are defined in: */
    include 'random.h'

    /* Seeds are specified by an argument, seed, which should be declared as: */
    integer seed(ran_s)

    /* The external representation of the seed is via a string of decimal digits. This should be declared
       as: */
    character*(ran_c) string

    /* The state of the random number generator is stored in two variables which are passed as
       arguments: ran_index and ran_array. These should be declared as: */
    integer ran_index
    double precision ran_array(ran_k)

    /* Finally, you may need to declare the external routines random and srandom: */
    double precision random
    real srandom
    external random, srandom

    /* In the above, you will need to change double precision to 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 as '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 0
  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*.

$x = \text{random}(\text{rn\_args}(\text{tag}))$  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*.

$sx = \text{srandom}(\text{rn\_args}(\text{tag}))$  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 0

```

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, ierr)
    // Determine and print the seed
    if (id == 0) then
        call date_and_time(cdate, ctime, czone, time)
        call set_random_seed(time, seed)

```

```

    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 · 100D) 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, ' is 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 ≡ 0) then
    pi = (4.0 · 100D * sum) / tot
    print *, 'pi is approximately', 4 * sum, '/', tot, '=', pi
endif

// Terminate MPI
call mpi_finalize(ierr)

stop
end
@#endif

```

The unnamed module.

```

"random.f" 3.1 ≡
    ⟨Functions and Subroutines 5⟩

```

Define constants used in the basic random number generator. *ran\_k* is defined in the header file to be 100.

```
"random.f" 3.2 ≡
  @m FILE 'random.web'

  @m ran_l 63 // Set l
  @m ran_p 1009 // We use k out of p numbers
  @m ran_e 47 // Bits of precision with double precision
  @m ran_es 23 // Bits of precision with single precision

  @m rand_center(x) x + ran_ulp2 // Center returned numbers in (0,1)
@#if ~HIPREC // Likewise for single_precision
  @m rand_center_s(x) (int(ran_mult * x) + 0.5) * ran_ulps
@#endif
```

## 4 Random definitions

[/u/dstotler/degas2/src/random.hweb]

\$Id: random.hweb,v 1.11 1999/08/09 15:18:38 karney Exp \$

[/u/dstotler/degas2/src/random.hweb] A random number sequence is identified by a string tag.

```
"random.f" 4.1 ≡
  @f rn_decl integer
  @f rn_decls integer
  @f rn_seed_decl integer
  @f single_precision integer

  @m rn_args(x) rn_index(x), rn_array(x)_0
  @m rn_dummy(x) rn_index(x), rn_array(x)
  @m rn_decl(x) integer rn_index(x)
    real rn_array(x)_0:ran_k-1
  @m rn_decls real random
    external random

  @m rn_copy(x,y) rn_index(y) = rn_index(x)
    do ran_temp = 0, ran_k - 1
      rn_array(y)_ran_temp = rn_array(x)_ran_temp
    end do

  @m rn_index(x) rn_index_##x // Accessor routines
  @m rn_array(x) rn_array_##x
```

[/u/dstotler/degas2/src/random.hweb] Length specifications.

```
"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 = random(rn_args(x))
  @m rn_init(seed,x) call random_init(seed, rn_args(x))
  @m rn_seed_copy(x,y) rn_seed_copy1(x, y)
  @m rn_seed_copy1(x,y) $DO(I, 0, ran_s - 1)
    {
      y_I = x_I
    }
  @m rn_seed_decl(x) integer x_0:ran_s-1
  @m rn_seed_args(x) x_0
  @m rn_iso_next(v,x) call random_isodist(v, 1, rn_args(x))
  @m rn_cos_next(v,x) call random_cosdist(v, 1, rn_args(x))
  @m rn_gauss_next(y,x) call random_gauss(y, 1, 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.

⟨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 = tworan_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(rn_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(rn_array(x),rn_index(x))
    rn_index(x) = rn_index(x) + 1

    return
  @#endif
end

```

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 *random* which fills an array. This is perhaps more efficient than separate calls to *random*. *srandom\_array* is an entry point providing a single precision result.

⟨Functions and Subroutines 5⟩ +≡

```

subroutine random_array(y, n, 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 = tworan_es)
  @#endif
  integer n // Input
  real y(0 : n - 1) // Output
  @#if -HIPREC
    single_precision ys(0 : n - 1) // Output
  @#endif
  rn_decl(x) // RNG state
  integer i, k, j // Local
  external rand_batch // External

  @#if HIPREC
    entry srandom_array(y, n, rn_dummy(x))
  @#endif

  if (n ≤ 0)
    return
  k = min(n, ran_k - rn_index(x))
  do i = 0, k - 1
    yi = 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
      yi = 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 ≤ 0)
      return
    k = min(n, ran_k - rn_index(x))
    do i = 0, k - 1
      ysi = rand_center_s(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
        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 Calculate the next batch

The fills the next  $k$  elements of  $rn\_array(x)$ . 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 ≡
@#if USEINT
  @m ran_assign(x,y) tmp = y
    x = tmp - int(tmp)
@#else
  @m ran_assign(x,y) tmp = y
    if (tmp ≥ one) then
      x = tmp - one
    else
      x = tmp
    end if
@#endif

⟨Functions and Subroutines 5⟩ +≡
  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) ≡ 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*( $i$ )  $rn\_array(x)_i = (((s_7 * ran\_del + s_6) * ran\_del + s_5) * ran\_del + \text{int}(s_4 / 512)) * 512 * ran\_del$

⟨Functions and Subroutines 5⟩ +≡

**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  $a_1$  and  $c_1$

**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**  $i$ ,  $j$ ,  $s_0:ran\_s-1$  // Local

**logical**  $odd$  // At least one seed is odd so far

**integer**  $z_0:ran\_s-1$ ,  $t$

**do**  $i = 0$ ,  $ran\_s - 1$

*assert*( $0 \leq seed_i \wedge seed_i < b$ )

$s_i = seed_i$

**end do**

$odd = \text{mod}(s_7, 2) \neq 0$

*ran\_set*(0)

**do**  $j = 1$ ,  $ran\_k - 1$

⟨Step the linear congruential generator forward one step 7.2⟩

$odd = odd \vee (\text{mod}(s_7, 2) \neq 0)$ ;

*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**

⟨Step the linear congruential generator forward one step 7.2⟩

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

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

**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  $c$ .)

⟨Step the linear congruential generator forward one step 7.2⟩ ≡

```

 $z_0 = c0 + a0 * s_0$ 
 $z_1 = a0 * s_1 + a1 * s_0$ 
 $z_2 = a0 * s_2 + a1 * s_1 + a2 * s_0$ 
 $z_3 = a0 * s_3 + a1 * s_2 + a2 * s_1 + a3 * s_0$ 
 $z_4 = a0 * s_4 + a1 * s_3 + a2 * s_2 + a3 * s_1 + a4 * s_0$ 
 $z_5 = a0 * s_5 + a1 * s_4 + a2 * s_3 + a3 * s_2 + a4 * s_1 + a5 * s_0$ 
 $z_6 = a0 * s_6 + a1 * s_5 + a2 * s_4 + a3 * s_3 + a4 * s_2 + a5 * s_1 + a6 * s_0$ 
 $z_7 = a0 * s_7 + a1 * s_6 + a2 * s_5 + a3 * s_4 + a4 * s_3 + a5 * s_2 + a6 * s_1$ 

 $t = 0$ 
do  $i = 0, ran\_s - 1$ 
   $t = \text{int}(t / b) + z_i$ 
   $s_i = \text{mod}(t, b)$ 
end do

```

This code is used in section 7.1.

Here's the vectorizing version of *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
  subroutine random_init(seed, rn_dummy(x))
    implicit_none_f77
    implicit_none_f90
    integer b
    real ran_del, ran_ulp
    parameter(ran_del = two-14, ran_ulp = two-ran_e, b = 214)
    rn_seed_decl(seed) // Input
    rn_decl(x) // Output
    integer i, j, k, s0:ran_s-1, 0:ran_k1-1 // Local
    logical odd0:ran_k1-1 // At lease one seed is odd so far
    integer z0:ran_s-1, 0:ran_k1-1, t0:ran_k1-1
    integer a0, a1, a2, a3, a4, a5, a6, c0 // The base 214 representation of a1 and c1
    data a0, a1, a2, a3, a4, a5, a6, c0/15661, 678, 724, 5245, 13656, 11852, 29, 1/
    integer ak20:ran_s-1, ck20:ran_s-1 // Tk2
    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 ≤ seedi ∧ seedi < b)
      si, 0 = seedi
    end do

    do k = 1, ran_k1 - 1
      do i = 0, ran_s - 1
        si, k = si, k-1
      end do
      call rand_axc(ak2, s0, k, ck2)
    end do

    do k = 0, ran_k1 - 1
      oddk = (mod(s7, k, 2) ≠ 0)
      rn_array(x)k*ran_k2 = (((s7, k*ran_del + s6, k)*ran_del + s5, k)*ran_del + int(s4, k/512))*512*ran_del
    end do

    do j = 1, ran_k2 - 1
      do k = 0, ran_k1 - 1
        z0, k = c0 + a0 * s0, k
        z1, k = a0 * s1, k + a1 * s0, k
        z2, k = a0 * s2, k + a1 * s1, k + a2 * s0, k
        z3, k = a0 * s3, k + a1 * s2, k + a2 * s1, k + a3 * s0, k
        z4, k = a0 * s4, k + a1 * s3, k + a2 * s2, k + a3 * s1, k + a4 * s0, k
        z5, k = a0 * s5, k + a1 * s4, k + a2 * s3, k + a3 * s2, k + a4 * s1, k + a5 * s0, k
        z6, k = a0 * s6, k + a1 * s5, k + a2 * s4, k + a3 * s3, k + a4 * s2, k + a5 * s1, k + a6 * s0, k
        z7, k = a0 * s7, k + a1 * s6, k + a2 * s5, k + a3 * s4, k + a4 * s3, k + a5 * s2, k + a6 * s1, k
        tk = 0
      end do
    end do
  end do
#endif
```

```

do i = 0, ran_s - 1
  do k = 0, ran_k1 - 1
    t_k = int(t_k / b) + z_{i, k}
    s_{i, k} = mod(t_k, b)
  end do
end do

do k = 0, ran_k1 - 1
  odd_k = odd_k ∨ (mod(s_{7, k}, 2) ≠ 0)
  rn_array(x)_{k*ran_k2+j} = (((s_{7, k} * ran_del + s_{6, k}) * ran_del + s_{5, k}) * ran_del + int(s_{4, k} / 512)) *
    512 * ran_del
end do
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. */

do k = 0, ran_k1 - 1
  if (odd_k)
    return
  end do

call rand_axc(ak2, s_{0, 0}, ck2)
j = int((s_{ran_s-1, 0} * ran_k) / b)
rn_array(x)_j += ran_ulp

return
end
@#endif

```

## 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.

(Functions and Subroutines 5) +≡

```

subroutine decimal_to_seed(decimal, seed)
  implicit_none_f77
  implicit_none_f90
  character*(*) decimal
  rn_seed_decl(seed)
  external rand_axc
  integer i, ten:ran_s-1, c0:ran_s-1, ch
  data ten/10, 7*0/

  do i = 0, ran_s - 1
    seedi = 0
    ci = 0
  end do

  do i = 1, len(decimal)
    ch = ichar(decimal(i : i))
    if (ch ≥ ichar('0') ∧ ch ≤ ichar('9')) then
      c0 = ch - ichar('0')
      call rand_axc(ten, seed, c)
    end if
  end do

  return
end

```

Set the seed from an arbitrary ASCII string. Only printable characters are considered. Each such character causes the operation  $seed = rotr(seed) + char$ , where  $rotr$  is a circular right shift by one bit; this is the operation carried out by the BSD sum utility.

⟨Functions and Subroutines 5⟩ +=

```

subroutine string_to_seed(string, seed)
  implicit none_f77
  implicit none_f90
  integer b
  parameter (b = 214)
  character*(*) string
  rn_seed_decl(seed)
  external rand_axc
  integer t, i, k, unity0:ran_s-1, c0:ran_s-1, ch
  data unity/1, 7*0/

  do i = 0, ran_s - 1
    seedi = 0
    ci = 0
  end do

  do i = 1, len(string)
    ch = ichar(string(i:i))
    if (ch > ichar('␣') ∧ ch < 127) then
      t = mod(seed0, 2) * (b / 2)
      do k = 0, ran_s - 1
        seedk = int(seedk / 2)
        if (k < ran_s - 1) then
          seedk += mod(seedk+1, 2) * (b / 2)
        else
          seedk += t
        end if
      end do
      c0 = ch
      call rand_axc(unity, seed, c) // else the character is whitespace and we skip it.
    end if
  end do

  return
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 is the zone offset is negative.

```
"random.f" 8.2 ≡
```

```
@m SP $UNQUOTE('␣')
```

```
⟨Functions and Subroutines 5⟩ +≡
```

```
subroutine set_random_seed(time, seed)
  implicit_none_f77
  implicit_none_f90
  integer time8 // 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), '(3i2.2,i3.3)') t5, t6, t7, t8
  call decimal_to_seed(c, seed)

  return
end
```

## 9 Convert a seed to its decimal equivalent

*decimal* 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 = 10pow, b = 214)
  character*(*) decimal
  rn_seed_decl(seed)
  integer z0:ran_s-1, i, t, j, k
  character*36 str

  k = -1
  do i = 0, ran_s - 1
    z_i = seed_i
    assert(0 ≤ z_i ∧ z_i < b)
    if (z_i > 0)
      k = i
  end do

  str = '␣'
  /* 2112 < 1036 */

  i = 9
loop: continue
  i--
  t = 0
  do j = k, 0, -1
    z_j = z_j + t * b
    t = mod(z_j, decbase)
    z_j = int(z_j / decbase)
  end do
  if (z_max(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) = '␣␣␣0'
  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
 $k = \min(j + 1, \text{len}(str))$ 
if ( $\text{len}(decimal) \geq \text{len}(str(k :))$ ) then
   $decimal = str(k :)$ 
else
   $decimal = str(\text{len}(str(k :)) - \text{len}(decimal) + 1 :)$ 
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

$$T^2(x) \equiv (a_2x + c_2) \pmod{2^{112}}.$$

where  $a_2 = a_1^2 \pmod{2^{112}}$  and  $c_2 = (a_1 + 1)c_1 \pmod{2^{112}}$ . The time is proportional to  $\log n$ .

(Functions and Subroutines 5) +≡

```

subroutine rand_next_seed(n, ax, cx, y)
  implicit_none_f77
  implicit_none_f90
  integer n, ax:ran_s-1, cx:ran_s-1 // Input
  rn_seed_decl(y) // Input/Output
  external rand_axc
  integer a:ran_s-1, c:ran_s-1, z:ran_s-1, t: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
    ci = cxi
  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' = a2 */
  goto loop
end

```

*next\_seed3* steps the seed forward by  $L(n_0, n_1, n_2)$  steps from equation 3. In this routine *afk* and *cfk* are precomputed coefficients for equation 2 iterated  $g_k$  times (i.e., forwards) and *abk* and *cbk* are precomputed coefficients for equation 2 iterated  $-g_k$  times (i.e., backwards). This routine works but invoking *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.

(Functions and Subroutines 5) +≡

```

subroutine next_seed3(n0, n1, n2, seed)
  implicit none_f77
  implicit none_f90
  integer n0, n1, n2    // Input
  rn_seed_decl(seed)    // Input/Output
  external rand_next_seed
  integer af0_{0:ran_s-1}, cf0_{0:ran_s-1}
  integer ab0_{0:ran_s-1}, cb0_{0:ran_s-1}
  integer af1_{0:ran_s-1}, cf1_{0:ran_s-1}
  integer ab1_{0:ran_s-1}, cb1_{0:ran_s-1}
  integer af2_{0:ran_s-1}, cf2_{0:ran_s-1}
  integer ab2_{0:ran_s-1}, cb2_{0:ran_s-1}
  data af0/15741, 8689, 9280, 4732, 12011, 7130, 6824, 12302/
  data cf0/16317, 10266, 1198, 331, 10769, 8310, 2779, 13880/
  data ab0/9173, 9894, 15203, 15379, 7981, 2280, 8071, 429/
  data cb0/8383, 3616, 597, 12724, 15663, 9639, 187, 4866/
  data af1/8405, 4808, 3603, 6718, 13766, 9243, 10375, 12108/
  data cf1/13951, 7170, 9039, 11206, 8706, 14101, 1864, 15191/
  data ab1/6269, 3240, 9759, 7130, 15320, 14399, 3675, 1380/
  data cb1/15357, 5843, 6205, 16275, 8838, 12132, 2198, 10330/
  data af2/445, 10754, 1869, 6593, 385, 12498, 14501, 7383/
  data cf2/2285, 8057, 3864, 10235, 1805, 10614, 9615, 15522/
  data ab2/405, 4903, 2746, 1477, 3263, 13564, 8139, 2362/
  data cb2/8463, 575, 5876, 2220, 4924, 1701, 9060, 5639/

  if (n2 > 0) then
    call rand_next_seed(n2, af2, cf2, seed)
  else if (n2 < 0) then
    call rand_next_seed(-n2, ab2, cb2, seed)
  end if

  if (n1 > 0) then
    call rand_next_seed(n1, af1, cf1, seed)
  else if (n1 < 0) then
    call rand_next_seed(-n1, ab1, cb1, seed)
  end if

entry next_seed(n0, seed)

  if (n0 > 0) then
    call rand_next_seed(n0, af0, cf0, seed)
  else if (n0 < 0) then
    call rand_next_seed(-n0, ab0, cb0, seed)
  end if

  return
end

```

A utility routine. Implement  $x = a * x + c$ .

⟨Functions and Subroutines 5⟩ +≡

```

subroutine rand_axc(a, x, c)
  implicit_none_f77
  implicit_none_f90
  integer b
  parameter (b = 214)
  integer a0:ran_s-1, c0:ran_s-1 // Input
  integer x0:ran_s-1 // Input/output
  integer z0:ran_s-1, i, j, t // Local

  do i = 0, ran_s - 1
    z_i = c_i
  end do

  /* These do loops were originally do i = 0, 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. */
  do j = 0, ran_s - 1
    do i = j, ran_s - 1
      z_i += a_j * x_{i-j}
    end do
  end do

  t = 0
  do i = 0, ran_s - 1
    t = int(t / b) + z_i
    x_i = mod(t, b)
  end do

  return
end

```

## 11 INDEX

- a*: [3](#), [10](#), [10.2](#).  
*abk*: [10.1](#).  
*abs*: [8.2](#).  
*ab0*: [10.1](#).  
*ab1*: [10.1](#).  
*ab2*: [10.1](#).  
*afk*: [10.1](#).  
*af0*: [10.1](#).  
*af1*: [10.1](#).  
*af2*: [10.1](#).  
*ak2*: [7.3](#).  
*assert*: [6](#), [7.1](#), [7.3](#), [9](#), [10](#).  
*ax*: [10](#).  
*a0*: [7.1](#), [7.2](#), [7.3](#).  
*a1*: [7.1](#), [7.2](#), [7.3](#).  
*a2*: [7.1](#), [7.2](#), [7.3](#).  
*a3*: [7.1](#), [7.2](#), [7.3](#).  
*a4*: [7.1](#), [7.2](#), [7.3](#).  
*a5*: [7.1](#), [7.2](#), [7.3](#).  
*a6*: [7.1](#), [7.2](#), [7.3](#).  
*b*: [7.1](#), [7.3](#), [8.1](#), [9](#), [10.2](#).  
*c*: [8](#), [8.1](#), [8.2](#), [10](#), [10.2](#).  
*cbk*: [10.1](#).  
*cb0*: [10.1](#).  
*cb1*: [10.1](#).  
*cb2*: [10.1](#).  
*cdate*: [3](#).  
*cfk*: [10.1](#).  
*cf0*: [10.1](#).  
*cf1*: [10.1](#).  
*cf2*: [10.1](#).  
*ch*: [8](#), [8.1](#).  
*char*: [8.1](#), [9](#).  
*ck2*: [7.3](#).  
*ctime*: [3](#).  
*cx*: [10](#).  
*czone*: [3](#).  
*c0*: [7.1](#), [7.2](#), [7.3](#).  
*date\_and\_time*: [2.1](#), [3](#), [8.2](#).  
*date\_time*: [2.1](#), [8.2](#).  
*decbase*: [9](#).  
*decimal*: [2.1](#), [8](#), [9](#).  
*decimal\_to\_seed*: [2.1](#), [8](#), [8.2](#).  
*digit*: [8](#).  
*FILE*: [3.2](#).  
*HIPREC*: [3.2](#), [5](#), [5.1](#).  
*hostname*: [3](#).  
*i*: [3](#), [5.1](#), [6](#), [7.1](#), [7.3](#), [8](#), [8.1](#), [9](#), [10](#), [10.2](#).  
*ichar*: [8](#), [8.1](#), [9](#).  
*id*: [3](#).  
*ierr*: [3](#).  
*iierr*: [3](#).  
*implicit\_none\_f77*: [5](#), [5.1](#), [6](#), [7.1](#), [7.3](#), [8](#), [8.1](#), [8.2](#), [9](#), [10](#), [10.1](#), [10.2](#).  
*implicit\_none\_f90*: [5](#), [5.1](#), [6](#), [7.1](#), [7.3](#), [8](#), [8.1](#), [8.2](#), [9](#), [10](#), [10.1](#), [10.2](#).  
*include*: [2](#), [2.1](#), [3](#).  
*int*: [3.2](#), [6](#), [7.1](#), [7.2](#), [7.3](#), [8.1](#), [9](#), [10](#), [10.2](#).  
*integer*: [4.1](#).  
*iterations*: [3](#).  
*j*: [3](#), [5.1](#), [7.1](#), [7.3](#), [9](#), [10.2](#).  
*k*: [5.1](#), [7.3](#), [8.1](#), [9](#).  
*len*: [3](#), [8](#), [8.1](#), [9](#).  
*loop*: [9](#), [10](#).  
*loop1*: [9](#).  
*m*: [3](#), [10](#).  
*max*: [9](#).  
*min*: [5.1](#), [9](#).  
*mod*: [7.1](#), [7.2](#), [7.3](#), [8.1](#), [8.2](#), [9](#), [10](#), [10.2](#).  
*mpi\_bcast*: [3](#).  
*mpi\_comm\_rank*: [3](#).  
*mpi\_comm\_size*: [3](#).  
*mpi\_comm\_world*: [3](#).  
*mpi\_finalize*: [3](#).  
*mpi\_get\_processor\_name*: [3](#).  
*mpi\_init*: [3](#).  
*mpi\_integer*: [3](#).  
*mpi\_max\_processor\_name*: [3](#).  
*mpi\_reduce*: [3](#).  
*mpi\_sum*: [3](#).  
*n*: [5.1](#), [10](#).  
*next\_seed*: [2.2](#), [3](#), [10.1](#).  
*next\_seed3*: [2.2](#), [10.1](#).  
*numprocs*: [3](#).  
*n0*: [2.2](#), [10.1](#).  
*n1*: [2.2](#), [10.1](#).  
*n2*: [2.2](#), [10.1](#).  
*odd*: [7.1](#), [7.3](#).  
*one*: [6](#).  
*pi*: [3](#).  
*pirandom*: [3](#).  
*pow*: [9](#).  
*ran\_array*: [1](#), [2](#), [3](#), [5](#).  
*ran\_array\_*: [4.1](#).  
*ran\_assign*: [6](#).

*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, 4.1, 4.2, 5, 5.1, 6, 7.1, 7.3.  
*ran\_k1*: 7.3.  
*ran\_k2*: 7.3.  
*ran\_l*: 3.2, 6.  
*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\_ulp*: 7.1, 7.3.  
*ran\_ulps*: 3.2, 5, 5.1.  
*ran\_ulp2*: 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.  
*rn\_args*: 2, 2.3, 4.1, 4.3, 5, 5.1.  
*rn\_array*: 4.1, 5, 5.1, 6, 7.1, 7.3.  
*rn\_copy*: 4.1.  
*rn\_cos\_next*: 4.3.  
**rn\_decl**: 2, 4.1.  
**rn\_decls**: 4.1.  
*rn\_dummy*: 4.1, 5, 5.1, 6, 7.1, 7.3.  
*rn\_gauss\_next*: 4.3.  
*rn\_index*: 4.1, 5, 5.1, 6, 7.1, 7.3.  
*rn\_init*: 4.3.  
*rn\_iso\_next*: 4.3.  
*rn\_next*: 4.3.  
*rn\_seed\_args*: 4.3.  
*rn\_seed\_copy*: 4.3.  
*rn\_seed\_copy1*: 4.3.  
**rn\_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.  
*srancom\_array*: 2.3.  
*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.  
*sum*: 3.  
*sx*: 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.  
*yyyymmddzzzzhhmmssmmm*: 8.2.  
  
*z*: 3, 7.1, 7.3, 9, 10, 10.2.  
*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.