NAG Library Function Document

nag_rand_init_repeatable (g05kfc)

1 Purpose

nag_rand_init_repeatable (g05kfc) initializes the selected base generator, as used by the group of pseudorandom number functions (see g05khc–g05kjc, g05ncc, g05ndc, g05pdc–g05pjc, g05pxc–g05pzc, g05rcc, g05rdc, g05ryc, g05rzc and g05sac–g05tlc), so as to generate a repeatable sequence of variates and the quasi-random scrambled sequence initialization function, nag_quasi_init_scrambled (g05ync).

2 Specification

```
#include <nag.h>
#include <nagg05.h>
void nag_rand_init_repeatable (Nag_BaseRNG genid, Integer subid,
     const Integer seed[], Integer lseed, Integer state[], Integer *lstate,
     NagError *fail)
```

3 Description

nag_rand_init_repeatable (g05kfc) selects a base generator through the input value of the arguments **genid** and **subid**, and then initializes it based on the values given in the array **seed**.

A given base generator will yield different sequences of random numbers if initialized with different values of **seed**. Alternatively, the same sequence of random numbers will be generated if the same value of **seed** is used. It should be noted that there is no guarantee of statistical properties between sequences, only within sequences.

A definition of some of the terms used in this description, along with details of the various base generators can be found in the g05 Chapter Introduction.

4 References

L'Ecuyer P and Simard R (2002) *TestU01: a software library in ANSI C for empirical testing of random number generators* Departement d'Informatique et de Recherche Operationnelle, Universite de Montreal http://www.iro.umontreal.ca/~lecuyer

Maclaren N M (1989) The generation of multiple independent sequences of pseudorandom numbers *Appl. Statist.* **38** 351–359

Matsumoto M and Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator *ACM Transactions on Modelling and Computer Simulations*

Wichmann B A and Hill I D (2006) Generating good pseudo-random numbers *Computational Statistics and Data Analysis* **51** 1614–1622

Wikramaratna R S (1989) ACORN - a new method for generating sequences of uniformly distributed pseudo-random numbers *Journal of Computational Physics* **83** 16–31

5 Arguments

1: **genid** – Nag_BaseRNG *Input*

On entry: must contain the type of base generator to use.

genid = Nag_Basic
NAG basic generator.

genid = Nag_WichmannHill_I
Wichmann Hill I generator.

genid = Nag_MersenneTwister
Mersenne Twister.

genid = Nag_WichmannHill_II
Wichmann Hill II generator.

genid = Nag_ACORN
ACORN generator.

genid = Nag_MRG32k3a
L'Ecuyer MRG32k3a generator.

See the g05 Chapter Introduction for details of each of the base generators.

Constraint: **genid** = Nag_Basic, Nag_WichmannHill_I, Nag_MersenneTwister, Nag_WichmannHill_II, Nag_ACORN or Nag_MRG32k3a.

2: **subid** – Integer *Input*

On entry: if **genid** = Nag_WichmannHill_I, **subid** indicates which of the 273 sub-generators to use. In this case, the $((|\mathbf{subid}| + 272)\ mod\ 273) + 1$ sub-generator is used.

If **genid** = Nag_ACORN, **subid** indicates the values of k and p to use, where k is the order of the generator, and p controls the size of the modulus, M, with $M = 2^{(p \times 30)}$. If **subid** < 1, the default values of $k = 10$ and $p = 2$ are used, otherwise values for k and p are calculated from the formula, **subid** $= k + 1000(p - 1)$.

If **genid** = Nag_MRG32k3a and **subid** mod $2 = 0$ the range of the generator is set to $(0, 1]$, otherwise the range is set to $(0, 1)$; in this case the sequence is identical to the implementation of MRG32k3a in TestU01 (see L'Ecuyer and Simard (2002)) for identical seeds.

For all other values of **genid**, **subid** is not referenced.

3: **seed**[**lseed**] – const Integer *Input*

On entry: the initial (seed) values for the selected base generator. The number of initial values required varies with each of the base generators.

If **genid** = Nag_Basic, one seed is required.

If **genid** = Nag_WichmannHill_I, four seeds are required.

If **genid** = Nag_MersenneTwister, 624 seeds are required.

If **genid** = Nag_WichmannHill_II, four seeds are required.

If **genid** = Nag_ACORN, $(k + 1)p$ seeds are required, where k and p are defined by **subid**. For the ACORN generator it is recommended that an odd value is used for **seed**[0].

If **genid** = Nag_MRG32k3a, six seeds are required.

If insufficient seeds are provided then the first **lseed** $- 1$ values supplied in **seed** are used and the remaining values are randomly generated using the NAG basic generator. In such cases the NAG basic generator is initialized using the value supplied in **seed**[**lseed** $- 1$].

Constraint: **seed**$[i - 1] \geq 1$, for $i = 1, 2, \ldots,$ **lseed**.

4: **lseed** – Integer *Input*

On entry: the size of the **seed** array.

Constraint: **lseed** ≥ 1.

5: **state**[**lstate**] – Integer *Communication Array*

On exit: contains information on the selected base generator and its current state. If **lstate** < 1 then **state** may be **NULL**.

6: **lstate** – Integer * *Input/Output*

On entry: the dimension of the **state** array, or a value < 1. If the Mersenne Twister (**genid** = Nag_MersenneTwister) is being used and the skip ahead function nag_rand_skip_ahead (g05kjc) or nag_rand_skip_ahead_power2 (g05kkc) will be called subsequently, then you must ensure that **lstate** ≥ 1260.

On exit: if **lstate** < 1 on entry, then the required length of the **state** array for the chosen base generator, otherwise **lstate** is unchanged. When **genid** = Nag_MersenneTwister (Mersenne Twister) a value of 1260 is returned, allowing for the skip ahead function to be subsequently called. In all other cases the minimum length, as documented in the constraints below, is returned.

Constraints:

if **genid** = Nag_Basic, **lstate** ≥ 17;
if **genid** = Nag_WichmannHill_I, **lstate** ≥ 21;
if **genid** = Nag_MersenneTwister, **lstate** ≥ 633;
if **genid** = Nag_WichmannHill_II, **lstate** ≥ 29;
if **genid** = Nag_ACORN, **lstate** $\geq \max((k+1) \times p + 9, 14) + 3$, where k and p are defined by **subid**;
if **genid** = Nag_MRG32k3a, **lstate** ≥ 61;
otherwise **lstate** < 1.

7: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_INT

On entry, **lseed** = ⟨*value*⟩.
Constraint: **lseed** ≥ 1.

On entry, **lstate** = ⟨*value*⟩.
Constraint: **lstate** ≤ 0 or **lstate** $\geq$ ⟨*value*⟩.

NE_INT_ARRAY

On entry, invalid **seed**.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

7 Accuracy

Not applicable.

8 Parallelism and Performance

Not applicable.

9 Further Comments

None.

10 Example

This example prints the first five pseudorandom real numbers from a uniform distribution between 0 and 1, generated by nag_rand_basic (g05sac) after initialization by nag_rand_init_repeatable (g05kfc).

10.1 Program Text

```
/* nag_rand_init_repeatable (g05kfc) Example Program.
 *
 * Copyright 2008, Numerical Algorithms Group.
 *
 * Mark 9, 2009.
 */
/* Pre-processor includes */
#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagg05.h>

int main(void)
{
  /* Integer scalar and array declarations */
  Integer     exit_status = 0;
  Integer     i, lstate;
  Integer     *state = 0;
  /* NAG structures */
  NagError    fail;
  /* Double scalar and array declarations */
  double      *x = 0;
  /* Set the sample size */
  Integer     n = 5;
  /* Choose the base generator */
  Nag_BaseRNG genid = Nag_Basic;
  Integer     subid = 0;
  /* Set the seed */
  Integer     seed[] = { 1762543 };
  Integer     lseed = 1;

  /* Initialise the error structure */
  INIT_FAIL(fail);

  printf(
          "nag_rand_init_repeatable (g05kfc) Example Program Results\n\n");

  /* Get the length of the state array */
  lstate = -1;
  nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Allocate arrays */
  if (!(x = NAG_ALLOC(n, double)) ||
      !(state = NAG_ALLOC(lstate, Integer)))
```

```
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Initialise the generator to a repeatable sequence */
  nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Generate the variates*/
  nag_rand_basic(n, state, x, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_basic (g05sac).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Display the variates */
  for (i = 0; i < n; i++)
    {
      printf("%10.4f\n", x[i]);
    }

 END:
  NAG_FREE(x);
  NAG_FREE(state);

  return exit_status;
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_rand_init_repeatable (g05kfc) Example Program Results

    0.6364
    0.1065
    0.7460
    0.7983
    0.1046
```