NAG Library Function Document

nag_quasi_init_scrambled (g05ync)

1 Purpose

nag_quasi_init_scrambled (g05ync) initializes a scrambled quasi-random generator prior to calling nag_quasi_rand_normal (g05yjc), nag_quasi_rand_lognormal (g05ykc) or nag_quasi_rand_uniform (g05ymc). It must be preceded by a call to one of the pseudorandom initialization functions nag_rand_init_repeatable (g05kfc) or nag_rand_init_nonrepeatable (g05kgc).

2 Specification

```
#include <nag.h>
#include <nagg05.h>
```

```
void nag_quasi_init_scrambled (Nag_QuasiRandom_Sequence genid,
     Nag_QuasiRandom_Scrambling stype, Integer idim, Integer iref[],
     Integer liref, Integer iskip, Integer nsdigi, Integer state[],
     NagError *fail)
```

3 Description

nag_quasi_init_scrambled (g05ync) selects a quasi-random number generator through the input value of **genid**, a method of scrambling through the input value of **stype** and initializes the **iref** communication array for use in the functions nag_quasi_rand_normal (g05yjc), nag_quasi_rand_lognormal (g05ykc) or nag_quasi_rand_uniform (g05ymc).

Scrambled quasi-random sequences are an extension of standard quasi-random sequences that attempt to eliminate the bias inherent in a quasi-random sequence whilst retaining the low-discrepancy properties. The use of a scrambled sequence allows error estimation of Monte–Carlo results by performing a number of iterates and computing the variance of the results.

This implementation of scrambled quasi-random sequences is based on TOMS Algorithm 823 and details can be found in the accompanying paper, Hong and Hickernell (2003). Three methods of scrambling are supplied; the first a restricted form of Owen's scrambling (Owen (1995)), the second based on the method of Faure and Tezuka (2000) and the last method combines the first two.

Scrambled versions of the Niederreiter sequence and two sets of Sobol sequences are provided. The first Sobol sequence is obtained using **genid** = Nag_QuasiRandom_Sobol. The first 10000 direction numbers for this sequence are based on the work of Joe and Kuo (2008). For dimensions greater than 10000 the direction numbers are randomly generated using the pseudorandom generator specified in **state** (see Jäckel (2002) for details). The second Sobol sequence is obtained using **genid** = Nag_QuasiRandom_SobolA659 and referred to in the documentation as 'Sobol (A659)'. The first 1111 direction numbers for this sequence are based on Algorithm 659 of Bratley and Fox (1988) with the extension proposed by Joe and Kuo (2003). For dimensions greater than 1111 the direction numbers are once again randomly generated. The Niederreiter sequence is obtained by setting **genid** = Nag_QuasiRandom_Nied.

4 References

Bratley P and Fox B L (1988) Algorithm 659: implementing Sobol's quasirandom sequence generator *ACM Trans. Math. Software* **14(1)** 88–100

Faure H and Tezuka S (2000) Another random scrambling of digital (t,s)-sequences *Monte Carlo and Quasi-Monte Carlo Methods* Springer-Verlag, Berlin, Germany (eds K T Fang, F J Hickernell and H Niederreiter)

Hong H S and Hickernell F J (2003) Algorithm 823: implementing scrambled digital sequences *ACM Trans. Math. Software* **29:2** 95–109

Jäckel P (2002) *Monte Carlo Methods in Finance* Wiley Finance Series, John Wiley and Sons, England

Joe S and Kuo F Y (2003) Remark on Algorithm 659: implementing Sobol's quasirandom sequence generator *ACM Trans. Math. Software (TOMS)* **29** 49–57

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

Niederreiter H (1988) Low-discrepancy and low dispersion sequences *Journal of Number Theory* **30** 51–70

Owen A B (1995) Randomly permuted (t,m,s)-nets and (t,s)-sequences *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, Lecture Notes in Statistics* **106** Springer-Verlag, New York, NY 299–317 (eds H Niederreiter and P J-S Shiue)

5 Arguments

1: **genid** – Nag_QuasiRandom_Sequence *Input*

On entry: must identify the quasi-random generator to use.

genid = Nag_QuasiRandom_Sobol
Sobol generator.

genid = Nag_QuasiRandom_SobolA659
Sobol (A659) generator.

genid = Nag_QuasiRandom_Nied
Niederreiter generator.

Constraint: **genid** = Nag_QuasiRandom_Sobol, Nag_QuasiRandom_SobolA659 or Nag_QuasiRandom_Nied.

2: **stype** – Nag_QuasiRandom_Scrambling *Input*

On entry: must identify the scrambling method to use.

stype = Nag_NoScrambling
No scrambling. This is equivalent to calling nag_quasi_init (g05ylc).

stype = Nag_OwenLike
Owen like scrambling.

stype = Nag_FaureTezuka
Faure–Tezuka scrambling.

stype = Nag_OwenFaureTezuka
Owen and Faure–Tezuka scrambling.

Constraint: **stype** = Nag_NoScrambling, Nag_OwenLike, Nag_FaureTezuka or Nag_OwenFaureTezuka.

3: **idim** – Integer *Input*

On entry: the number of dimensions required.

Constraints:

if **genid** = Nag_QuasiRandom_Sobol, $1 \leq$ **idim** ≤ 50000;
if **genid** = Nag_QuasiRandom_SobolA659, $1 \leq$ **idim** ≤ 50000;
if **genid** = Nag_QuasiRandom_Nied, $1 \leq$ **idim** ≤ 318.

4: **iref**[**liref**] – Integer *Communication Array*

On exit: contains initialization information for use by the generator functions nag_quasi_rand_normal (g05yjc), nag_quasi_rand_lognormal (g05ykc) and nag_quasi_rand_uniform (g05ymc). **iref** must not be altered in any way between initialization and calls of the generator functions.

5: **liref** – Integer *Input*

On entry: the dimension of the array **iref**.

Constraint: $\textbf{liref} \geq 32 \times \textbf{idim} + 7$.

6: **iskip** – Integer *Input*

On entry: the number of terms of the sequence to skip on initialization for the Sobol and Niederreiter generators.

Constraint: $0 \leq \textbf{iskip} \leq 2^{30}$.

7: **nsdigi** – Integer *Input*

On entry: controls the number of digits (bits) to scramble when **genid** = Nag_QuasiRandom_Sobol or Nag_QuasiRandom_SobolA659, otherwise **nsdigi** is ignored. If $\textbf{nsdigi} < 1$ or $\textbf{nsdigi} > 30$ then all the digits are scrambled.

8: **state**[*dim*] – Integer *Communication Array*

Note: the dimension, *dim*, of this array is dictated by the requirements of associated functions that must have been previously called. This array MUST be the same array passed as argument **state** in the previous call to nag_rand_init_repeatable (g05kfc) or nag_rand_init_nonrepeatable (g05kgc).

On entry: contains information on the selected base generator and its current state.

On exit: contains updated information on the state of the generator.

9: **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, **idim** = ⟨*value*⟩.
Constraint: $1 \leq \textbf{idim} \leq$ ⟨*value*⟩.

On entry, **iskip** = ⟨*value*⟩.
Constraint: $0 \leq \textbf{iskip} \leq 2^{30}$.

On entry, **liref** = ⟨*value*⟩.
Constraint: $\textbf{liref} \geq 32 \times \textbf{idim} + 7$.

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.

NE_INVALID_STATE

On entry, **state** vector has been corrupted or not initialized.

7 Accuracy

Not applicable.

8 Parallelism and Performance

Not applicable.

9 Further Comments

The additional computational cost in using a scrambled quasi-random sequence over a non-scrambled one comes entirely during the initialization. Once nag_quasi_init_scrambled (g05ync) has been called the computational cost of generating a scrambled sequence and a non-scrambled one is identical.

10 Example

This example calls nag_rand_init_repeatable (g05kfc), nag_quasi_rand_uniform (g05ymc) and nag_quasi_init_scrambled (g05ync) to estimate the value of the integral

$$\int_0^1 \cdots \int_0^1 \prod_{i=1}^{s} |4x_i - 2| dx_1, dx_2, \ldots, dx_s = 1,$$

where s, the number of dimensions, is set to 8.

10.1 Program Text

```
/* nag_quasi_init_scrambled (g05ync) Example Program.
 *
 * Copyright 2008, Numerical Algorithms Group.
 *
 * Mark 9, 2009.
 */
#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagg05.h>
#define QUAS(I, J) quas[(order == Nag_ColMajor)?(J*pdquas + I):(I*pdquas + J)]

int main(void)
{
  /*Integer scalar and array declarations */
  Integer                    exit_status = 0;
  Integer                    liref, d, i, j, lstate, q_size;
  Integer                    *iref = 0, *state = 0;

  /* NAG structures */
  Integer                    pdquas;
  NagError                   fail;

  /*Double scalar and array declarations */
  double                     sum, tmp, vsbl;
  double                     *quas = 0;

  /* Number of dimensions */
  Integer                    idim = 8;

  /* Set the sample size */
  Integer                    n = 200;
  /* Skip the first 1000 variates */
  Integer                    iskip = 1000;

  /* Use row major order */
  Nag_OrderType              order = Nag_RowMajor;

  /* Choose the base pseudo generator */
  Nag_BaseRNG                pgenid = Nag_Basic;
  Integer                    psubid = 0;

  /* Set the seed */
  Integer                    seed[] = { 1762543 };
  Integer                    lseed = 1;

  /* Choose the quasi generator */
  Nag_QuasiRandom_Sequence   genid = Nag_QuasiRandom_Sobol;

  /* Use Owen type scrambling */
  Nag_QuasiRandom_Scrambling stype = Nag_OwenLike;

  /* Scramble the default number of digits */
  Integer                    nsdigi = 0;

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

  printf("nag_quasi_init_scrambled (g05ync) Example Program Results\n");

  /* Get the length of the state array */
  lstate = -1;
  nag_rand_init_repeatable(pgenid, psubid, 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;
    }

  pdquas = (order == Nag_RowMajor)?idim:n;
  q_size = (order == Nag_RowMajor)?pdquas * n:pdquas * idim;

  /* Calculate the size of the reference vector */
  liref = (genid == Nag_QuasiRandom_Faure)?407:32 * idim + 7;

  /* Allocate arrays */
  if (!(quas = NAG_ALLOC(q_size, double)) ||
      !(iref = NAG_ALLOC(liref, Integer)) ||
      !(state = NAG_ALLOC(lstate, Integer)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Initialise the pseudo-random generator used in the
     scrambling to a repeatable sequence */
  nag_rand_init_repeatable(pgenid, psubid, 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;
    }

  /* Initialise the quasi-random sequence */
  nag_quasi_init_scrambled(Nag_QuasiRandom_Sobol, stype, idim, iref, liref,
                           iskip, nsdigi, state, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_quasi_init_scrambled (g05ync).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }
  /* Generate n quasi-random variates */
  nag_quasi_rand_uniform(order, n, quas, pdquas, iref, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_quasi_rand_uniform (g05ymc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Estimate integral by evaluating function at each variate and summing */
  sum = 0.0e0;
  for (i = 0; i < n; i++)
    {
      tmp = 1.0e0;
      for (d = 0; d < idim; d++)
        tmp *= fabs(4.0e0 * QUAS(i, d) - 2.0e0);
      sum += tmp;
    }

  /* Convert sum to mean value*/
  vsbl = sum / (double) n;

  /* Print the estimated value of the integral */
  printf("Value of integral = %8.4f\n\n", vsbl);

  /* Display the first 10 variates used */
  printf("First 10 variates\n");
  for (i = 0; i < 10; i++)
    {
      printf(" %3ld", i + 1);
      for (j = 0; j < idim; j++)
        printf("%8.4f%s", QUAS(i, j), ((j+1)%20)?" ":"\n");
      if (idim%20) printf("\n");
    }

 END:
  NAG_FREE(quas);
  NAG_FREE(iref);
  NAG_FREE(state);

  return exit_status;
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_quasi_init_scrambled (g05ync) Example Program Results
Value of integral =   1.0169

First 10 variates
   1  0.8688   0.9719   0.5375   0.0876   0.4721   0.3800   0.2977   0.1010
   2  0.6287   0.3611   0.4963   0.8648   0.0753   0.0174   0.7011   0.2532
   3  0.1244   0.5349   0.8645   0.2621   0.7523   0.7212   0.0538   0.6231
   4  0.1353   0.4013   0.6656   0.4691   0.9096   0.9272   0.5481   0.4164
   5  0.6154   0.6962   0.0321   0.9000   0.2307   0.3186   0.1989   0.7102
   6  0.8870   0.0880   0.9947   0.1775   0.3148   0.2059   0.8033   0.9249
   7  0.3603   0.7579   0.3633   0.6995   0.5127   0.5328   0.4496   0.2013
   8  0.3304   0.1096   0.5034   0.3596   0.0137   0.3643   0.1719   0.8774
   9  0.9207   0.7834   0.1357   0.7596   0.8138   0.8825   0.5831   0.2493
```

```
  10  0.5828    0.4226    0.8287    0.0370    0.7336    0.5189    0.4143    0.4015
```