NAG Library Function Document

nag_rand_bb (g05xbc)

1 Purpose

nag_rand_bb (g05xbc) uses a Brownian bridge algorithm to construct sample paths for a free or non-free Wiener process. The initialization function nag_rand_bb_init (g05xac) must be called prior to the first call to nag_rand_bb (g05xbc).

2 Specification

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

```
void nag_rand_bb (Nag_OrderType order, Integer npaths, Integer d,
     const double start[], Integer a, const double term[], double z[],
     Integer pdz, const double c[], Integer pdc, double b[], Integer pdb,
     const double rcomm[], NagError *fail)
```

3 Description

For details on the Brownian bridge algorithm and the bridge construction order see Section 2.6 in the g05 Chapter Introduction and Section 3 in nag_rand_bb_init (g05xac). Recall that the terms Wiener process (or free Wiener process) and Brownian motion are often used interchangeably, while a non-free Wiener process (also known as a Brownian bridge process) refers to a process which is forced to terminate at a given point.

4 References

Glasserman P (2004) *Monte Carlo Methods in Financial Engineering* Springer

5 Arguments

Note: the following variable is used in the parameter descriptions: $N =$ **ntimes**, the length of the array **times** passed to the initialization function nag_rand_bb_init (g05xac).

1: **order** – Nag_OrderType *Input*

On entry: the **order** argument specifies the two-dimensional storage scheme being used, i.e., row-major ordering or column-major ordering. C language defined storage is specified by **order** = Nag_RowMajor. See Section 3.2.1.3 in the Essential Introduction for a more detailed explanation of the use of this argument.

Constraint: **order** = Nag_RowMajor or Nag_ColMajor.

2: **npaths** – Integer *Input*

On entry: the number of Wiener sample paths to create.

Constraint: **npaths** ≥ 1.

3: **d** – Integer *Input*

On entry: the dimension of each Wiener sample path.

Constraint: **d** ≥ 1.

4: **start**[**d**] – const double *Input*

On entry: the starting value of the Wiener process.

5: **a** – Integer *Input*

On entry: if $\mathbf{a} = 0$, a free Wiener process is created beginning at **start** and **term** is ignored.

If $\mathbf{a} = 1$, a non-free Wiener process is created beginning at **start** and ending at **term**.

Constraint: $\mathbf{a} = 0$ or 1.

6: **term**[**d**] – const double *Input*

On entry: the terminal value at which the non-free Wiener process should end. If $\mathbf{a} = 0$, **term** is ignored.

7: **z**[*dim*] – double *Input/Output*

Note: the dimension, *dim*, of the array **z** must be at least

$\mathbf{pdz} \times \mathbf{npaths}$ when $\mathbf{order} = \text{Nag\_RowMajor}$;
$\mathbf{pdz} \times (\mathbf{d} \times (N + 1 - \mathbf{a}))$ when $\mathbf{order} = \text{Nag\_ColMajor}$.

The (i, j)th element of the matrix Z is stored in

$\mathbf{z}[(j - 1) \times \mathbf{pdz} + i - 1]$ when $\mathbf{order} = \text{Nag\_ColMajor}$;
$\mathbf{z}[(i - 1) \times \mathbf{pdz} + j - 1]$ when $\mathbf{order} = \text{Nag\_RowMajor}$.

On entry: the Normal random numbers used to construct the sample paths.

If quasi-random numbers are used, the $\mathbf{d} \times (N + 1 - \mathbf{a})$-dimensional quasi-random points should be stored in successive rows of Z.

On exit: the Normal random numbers premultiplied by C.

8: **pdz** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **z**.

Constraints:

if $\mathbf{order} = \text{Nag\_RowMajor}$, $\mathbf{pdz} \geq \mathbf{d} \times (N + 1 - \mathbf{a})$;
if $\mathbf{order} = \text{Nag\_ColMajor}$, $\mathbf{pdz} \geq \mathbf{npaths}$.

9: **c**[*dim*] – const double *Input*

Note: the dimension, *dim*, of the array **c** must be at least $\mathbf{pdc} \times \mathbf{d}$.

The (i, j)th element of the matrix C is stored in $\mathbf{c}[(j - 1) \times \mathbf{pdc} + i - 1]$.

On entry: the lower triangular Cholesky factorization C such that CC^{T} gives the covariance matrix of the Wiener process. Elements of C above the diagonal are not referenced.

10: **pdc** – Integer *Input*

On entry: the stride separating matrix row elements in the array **c**.

Constraint: $\mathbf{pdc} \geq \mathbf{d}$.

11: **b**[*dim*] – double *Output*

Note: the dimension, *dim*, of the array **b** must be at least

$\mathbf{pdb} \times \mathbf{npaths}$ when $\mathbf{order} = \text{Nag\_RowMajor}$;
$\mathbf{pdb} \times (\mathbf{d} \times (N + 1))$ when $\mathbf{order} = \text{Nag\_ColMajor}$.

The (i,j)th element of the matrix B is stored in

b$[(j-1)\times$ **pdb** $+ i - 1]$ when **order** = Nag_ColMajor;
b$[(i-1)\times$ **pdb** $+ j - 1]$ when **order** = Nag_RowMajor.

On exit: the values of the Wiener sample paths.

Let $X^k_{p,i}$ denote the kth dimension of the ith point of the pth sample path where $1 \leq k \leq$ **d**, $1 \leq i \leq N+1$ and $1 \leq p \leq$ **npaths**. The point $X^k_{p,i}$ is stored at $B(p, k + (i-1) \times$ **d**$)$. The starting value **start** is never stored, whereas the terminal value is always stored.

12: **pdb** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **b**.

Constraints:

if **order** = Nag_RowMajor, **pdb** $\geq$ **d** $\times (N+1)$;
if **order** = Nag_ColMajor, **pdb** $\geq$ **npaths**.

13: **rcomm**[*dim*] – const double *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 **rcomm** in the previous call to nag_rand_bb_init (g05xac) or nag_rand_bb (g05xbc).

On entry: communication array as returned by the last call to nag_rand_bb_init (g05xac) or nag_rand_bb (g05xbc). This array MUST NOT be directly modified.

14: **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.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_ARRAY_SIZE

On entry, **pdb** = ⟨*value*⟩ and **d** × (**ntimes** + 1) = ⟨*value*⟩.
Constraint: **pdb** ≥ **d** × (**ntimes** + 1).

On entry, **pdb** = ⟨*value*⟩ and **npaths** = ⟨*value*⟩.
Constraint: **pdb** ≥ **npaths**.

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

On entry, **pdz** = ⟨*value*⟩ and **d** × (**ntimes** + 1 − **a**) = ⟨*value*⟩.
Constraint: **pdz** ≥ **d** × (**ntimes** + 1 − **a**).

On entry, **pdz** = ⟨*value*⟩ and **npaths** = ⟨*value*⟩.
Constraint: **pdz** ≥ **npaths**.

NE_BAD_PARAM

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

NE_ILLEGAL_COMM

On entry, **rcomm** was not initialized or has been corrupted.

NE_INT

On entry, **a** = $\langle value \rangle$.
Constraint: **a** = 0 or 1.

On entry, **d** = $\langle value \rangle$.
Constraint: **d** ≥ 1.

On entry, **npaths** = $\langle value \rangle$.
Constraint: **npaths** ≥ 1.

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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_rand_bb (g05xbc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_rand_bb (g05xbc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the X06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

None.

10 Example

This example calls nag_rand_bb (g05xbc), nag_rand_bb_init (g05xac) and nag_rand_bb_make_bridge_order (g05xec) to generate two sample paths of a three-dimensional non-free Wiener process. The process starts at zero and each sample path terminates at the point $(1.0, 0.5, 0.0)$. Quasi-random numbers are used to construct the sample paths.

See Section 10 in nag_rand_bb_init (g05xac) and nag_rand_bb_make_bridge_order (g05xec) for additional examples.

10.1 Program Text

```
/* nag_rand_bb (g05xbc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 24, 2013.
 */
#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagg05.h>
#include <nagf07.h>

int get_z(Nag_OrderType order, Integer ntimes, Integer d, Integer a,
          Integer npaths, double *z, Integer pdz);
void display_results(Nag_OrderType order, Integer npaths, Integer  ntimes,
                     Integer  d, double  *b, Integer pdb);

#define CHECK_FAIL(name,fail) if(fail.code != NE_NOERROR) { \
    printf("Error from %s.\n%s\n",name,fail.message); \
    exit_status = -1; goto END; }

int main(void)
{
  /*  Scalars */
  Integer       exit_status = 0;
  double        t0, tend;
  Integer       a, d, pdb, pdc, pdz, nmove, npaths, ntimes, i;
  /*  Arrays */
  double        *b = 0, *c = 0, *intime = 0, *rcomm = 0, *start = 0, *term = 0,
                *times = 0, *z = 0;
  Integer       *move = 0;
  /* Nag Types */
  NagError      fail;
  Nag_OrderType order;

  INIT_FAIL(fail);

  /* Parameters which determine the bridge. */
  ntimes = 10;
  t0 = 0.0;
  npaths = 2;
  /* Create a non-free bridge. */
  a = 1;
  nmove = 0;
  d = 3;
#ifdef NAG_COLUMN_MAJOR
  order = Nag_ColMajor;
  pdz = npaths;
  pdb = npaths;
#else
  order = Nag_RowMajor;
  pdz = d*(ntimes+1-a);
  pdb = d*(ntimes+1);
#endif
  pdc = d;
#define C(I,J) c[(J-1)*pdc+I-1]

  /* Allocate memory */
  if (
      !( intime = NAG_ALLOC((ntimes), double)) ||
      !( times = NAG_ALLOC((ntimes), double)) ||
      !( rcomm = NAG_ALLOC((12*(ntimes+1)), double)) ||
      !( start = NAG_ALLOC(d, double))  ||
      !( term = NAG_ALLOC(d, double))  ||
      !( c = NAG_ALLOC(d*pdc, double)) ||
      !( z = NAG_ALLOC(d*(ntimes+1-a)*npaths, double)) ||
      !( b = NAG_ALLOC(d*(ntimes+1)*npaths, double)) ||
      !( move = NAG_ALLOC(nmove, Integer))
```

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

  /* Fix the time points at which the bridge is required */
  for ( i=0; i<ntimes; i++)
    intime[i] =  t0 + (double)(i+1);
  tend =  t0 + (double)(ntimes + 1);

  /* Create a Brownian bridge construction order out of a set of input times
   * using nag_rand_bb_make_bridge_order (g05xec).
   */
  nag_rand_bb_make_bridge_order(Nag_RLRoundDown, t0, tend, ntimes, intime,
                                nmove, move, times, &fail);
  CHECK_FAIL("nag_rand_bb_make_bridge_order",fail);

  /* Initialize the Brownian bridge generator using
   * nag_rand_bb_init (g05xac).
   */
  nag_rand_bb_init(t0, tend, times, ntimes, rcomm, &fail);
  CHECK_FAIL("nag_rand_bb_init (g05xac)",fail);

  /* We want the following covariance matrix ... */
  C(1,1) = 6.0;
  C(2,1) = C(1,2) = 1.0;
  C(3,1) = C(1,3) = -0.2;
  C(2,2) = 5.0;
  C(3,2) = C(2,3) = 0.3;
  C(3,3) = 4.0;
  /* Cholesky factorize the covariance matrix C, as required by
   * nag_rand_bb (g05xbc), using nag_dpotrf (f07fdc).
   */
  nag_dpotrf(Nag_ColMajor, Nag_Lower, d, c, pdc, &fail);
  CHECK_FAIL("nag_dpotrf",fail);

  /* Generate the random numbers z.*/
  if( get_z(order, ntimes, d, a, npaths, z, pdz) != 0)
  {
      printf("Error generating random numbers\n");
      exit_status = -1;
      goto END;
  }
  /* Give start and terminal values of pinned bridge */
  start[0] = start[1] = start[2] = 0.0;
  term[0] = 1.0;
  term[1] = 0.5;
  term[2] = 0.0;

  /* Generate paths for a free or non-free Wiener process using the
   * Brownian bridge algorithm: nag_rand_bb (g05xbc).
   */
  nag_rand_bb(order, npaths, d, start, a, term, z, pdz, c, pdc,
              b, pdb, rcomm, &fail);
  CHECK_FAIL("nag_rand_bb",fail);

  /* Display the results*/
  display_results(order, npaths, ntimes, d, b, pdb);
 END:
  NAG_FREE(b);
  NAG_FREE(c);
  NAG_FREE(intime);
  NAG_FREE(rcomm);
  NAG_FREE(start);
  NAG_FREE(term);
  NAG_FREE(times);
  NAG_FREE(z);
  NAG_FREE(move);
  return exit_status;
```

```
}

int get_z(Nag_OrderType order, Integer ntimes, Integer d, Integer a,
          Integer npaths, double * z, Integer pdz)
{
  /* Scalars */
  Integer exit_status=0;
  Integer lseed, lstate, idim, liref, i;
  /* Arrays */
  Integer  seed[1], *iref=0, state[80];
  double   *xmean = 0, *stdev = 0;
  /* Nag Types */
  NagError fail;

  INIT_FAIL(fail);

  lstate = 80;
  lseed = 1;
  idim = d*(ntimes + 1 - a);
  liref = 32*idim + 7;
  if (
      !( iref = NAG_ALLOC((liref), Integer)) ||
      !( xmean = NAG_ALLOC((idim), double)) ||
      !( stdev = NAG_ALLOC((idim), double)) )
    {
      printf("Allocation failure in get_z\n");
      exit_status = -1;
      goto END;
    }

  /* We now need to generate the input pseudorandom numbers. */
  seed[0] = 1023401;
  /* Initialize a pseudorandom number generator to give a repeatable sequence
   * using nag_rand_init_repeatable (g05kfc).
   */
  nag_rand_init_repeatable(Nag_MRG32k3a,0,seed, lseed, state, &lstate, &fail);
  CHECK_FAIL("nag_rand_init_repeatable (g05kfc)",fail);

  /* Initialize a scrambled quasi-random number generator using
   * nag_quasi_init_scrambled (g05ync).
   */
  nag_quasi_init_scrambled(Nag_QuasiRandom_Sobol, Nag_FaureTezuka, idim,
        iref, liref, 0, 32, state, &fail);
  CHECK_FAIL("nag_quasi_init_scrambled (g05ync)",fail);

  for(i=0; i<idim; i++) {
    xmean[i] = 0.0;
    stdev[i] = 1.0;
  }
  /* Generate a (repeatable) Normal quasi-random number sequence using
   * nag_quasi_rand_normal (g05yjc).
   */
  nag_quasi_rand_normal(order, xmean, stdev, npaths, z, pdz, iref, &fail);
  CHECK_FAIL("nag_quasi_rand_normal (g05yjc)",fail);

END:
  NAG_FREE(iref);
  NAG_FREE(xmean);
  NAG_FREE(stdev);
  return exit_status;
}

void display_results(Nag_OrderType order, Integer npaths, Integer  ntimes,
                     Integer  d, double  *b, Integer pdb)
{
#define B(I,J) (order==Nag_RowMajor ? b[(I-1)*pdb + J-1]:b[(J-1)*pdb + I-1])
  Integer i, p, k;

  printf("nag_rand_bb (g05xbc) Example Program Results\n\n");
  for ( p=1; p<=npaths; p++) {
    printf("Wiener Path  %1"NAG_IFMT",  %1"NAG_IFMT"", p, ntimes + 1);
```

```
    printf(" time steps,  %1"NAG_IFMT" dimensions\n",  d);

    for ( k=1; k<= d; k++)
      printf("%10"NAG_IFMT" ", k);
    printf("\n");

    for (i=0; i<ntimes+1; i++) {
      printf("%2"NAG_IFMT" ", i+1);
      for (k=1; k<=d; k++)
        printf("%10.4f", B(p, k+i*d));
      printf("\n");
    }
    printf("\n");
  }
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_rand_bb (g05xbc) Example Program Results

Wiener Path  1,  11 time steps,  3 dimensions
         1          2          3
 1    -1.0602   -2.8701   -0.9415
 2    -3.0575   -1.9502    0.2596
 3    -6.8274   -2.4434    0.4597
 4    -5.2855   -3.4475    0.0795
 5    -8.1784   -5.2296   -0.0921
 6    -4.6874   -5.0220    1.4862
 7    -3.0959   -4.8623   -4.4076
 8    -2.9605   -1.8936   -3.9539
 9    -5.4685   -2.3856   -3.2031
10     0.1205   -5.0520   -1.0385
11     1.0000    0.5000    0.0000

Wiener Path  2,  11 time steps,  3 dimensions
         1          2          3
 1     0.6564    3.5142    1.5911
 2    -2.3773    3.1618    3.0316
 3     0.3020    6.8815    2.0875
 4    -0.2169    4.6026    1.1982
 5    -2.0684    4.1503    2.4758
 6    -5.1075    3.7303    2.7563
 7    -3.8497    3.6682    2.4827
 8    -1.8292    4.4153    0.1916
 9    -2.0649    0.6952   -2.1201
10     0.1962    1.7769   -5.7685
11     1.0000    0.5000    0.0000
```