NAG Library Function Document

nag_rand_bb_init (g05xac)

1 Purpose

nag_rand_bb_init (g05xac) initializes the Brownian bridge generator nag_rand_bb (g05xbc). It must be called before any calls to nag_rand_bb (g05xbc).

2 Specification

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

```
void nag_rand_bb_init (double t0, double tend, const double times[],
     Integer ntimes, double rcomm[], NagError *fail)
```

3 Description

3.1 Brownian Bridge Algorithm

Details on the Brownian bridge algorithm and the Brownian bridge process (sometimes also called a non-free Wiener process) can be found in Section 2.6 in the g05 Chapter Introduction. We briefly recall some notation and definitions.

Fix two times $t_0 < T$ and let $(t_i)_{1 \leq i \leq N}$ be any set of time points satisfying $t_0 < t_1 < t_2 < \cdots < t_N < T$. Let $(X_{t_i})_{1 \leq i \leq N}$ denote a d-dimensional Wiener sample path at these time points, and let C be any d by d matrix such that CC^{T} is the desired covariance structure for the Wiener process. Each point X_{t_i} of the sample path is constructed according to the Brownian bridge interpolation algorithm (see Glasserman (2004) or Section 2.6 in the g05 Chapter Introduction). We always start at some fixed point $X_{t_0} = x \in \mathbb{R}^d$. If we set $X_T = x + C\sqrt{T - t_0}Z$ where Z is any d-dimensional standard Normal random variable, then X will behave like a normal (free) Wiener process. However if we fix the terminal value $X_T = w \in \mathbb{R}^d$, then X will behave like a non-free Wiener process.

3.2 Implementation

Given the start and end points of the process, the order in which successive interpolation times t_j are chosen is called the *bridge construction order*. The construction order is given by the array **times**. Further information on construction orders is given in Section 2.6.2 in the g05 Chapter Introduction. For clarity we consider here the common scenario where the Brownian bridge algorithm is used with quasi-random points. If pseudorandom numbers are used instead, these details can be ignored.

Suppose we require P Wiener sample paths each of dimension d. The main input to the Brownian bridge algorithm is then an array of quasi-random points $Z^1, Z^2, \ldots, Z^P$ where each point $Z^p = \left(Z^p_1, Z^p_2, \ldots, Z^p_D\right)$ has dimension $D = d(N+1)$ or $D = dN$ respectively, depending on whether a free or non-free Wiener process is required. When nag_rand_bb (g05xbc) is called, the pth sample path for $1 \leq p \leq P$ is constructed as follows: if a non-free Wiener process is required set X_T equal to the terminal value w, otherwise construct X_T as

$$X_T = X_{t_0} + C\sqrt{T - t_0}\begin{bmatrix} Z^p_1 \\ \vdots \\ Z^p_d \end{bmatrix}$$

where C is the matrix described in Section 3.1. The array **times** holds the remaining time points $t_1, t_2, \ldots t_N$ in the order in which the bridge is to be constructed. For each $j = 1, \ldots, N$ set $r = \textbf{times}[j-1]$, find

$$q = \max\{t_0, \textbf{times}[i-1] : 1 \leq i < j, \textbf{times}[i-1] < r\}$$

and

$$s = \min\{T, \textbf{times}[i-1] : 1 \leq i < j, \textbf{times}[i-1] > r\}$$

and construct the point X_r as

$$X_r = \frac{X_q(s-r) + X_s(r-q)}{s-q} + C\sqrt{\frac{(s-r)(r-q)}{(s-q)}}\begin{bmatrix} Z^p_{jd-ad+1} \\ \vdots \\ Z^p_{jd-ad+d} \end{bmatrix}$$

where $a = 0$ or $a = 1$ respectively depending on whether a free or non-free Wiener process is required. Note that in our discussion j is indexed from 1, and so X_r is interpolated between the nearest (in time) Wiener points which have already been constructed. The function nag_rand_bb_make_bridge_order (g05xec) can be used to initialize the **times** array for several predefined bridge construction orders.

4 References

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

5 Arguments

1: **t0** – double *Input*

On entry: the starting value t_0 of the time interval.

2: **tend** – double *Input*

On entry: the end value T of the time interval.

Constraint: **tend** $>$ **t0**.

3: **times**[**ntimes**] – const double *Input*

On entry: the points in the time interval (t_0, T) at which the Wiener process is to be constructed. The order in which points are listed in **times** determines the bridge construction order. The function nag_rand_bb_make_bridge_order (g05xec) can be used to create predefined bridge construction orders from a set of input times.

Constraints:

t0 $<$ **times**$[i-1]$ $<$ **tend**, for $i = 1, 2, \ldots,$ **ntimes**;
times$[i-1] \neq$ **times**$[j-1]$, for $i, j = 1, 2, \ldots$ **ntimes** and $i \neq j$.

4: **ntimes** – Integer *Input*

On entry: the length of **times**, denoted by N in Section 3.1.

Constraint: **ntimes** ≥ 1.

5: **rcomm**[**12** $\times$ (**ntimes** + **1**)] – double *Communication Array*

On exit: communication array, used to store information between calls to nag_rand_bb (g05xbc). This array MUST NOT be directly modified.

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

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

6 Error Indicators and Warnings

NE_BAD_PARAM

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

NE_INT

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

NE_REAL

On entry, **tend** $= \langle value \rangle$ and **t0** $= \langle value \rangle$.
Constraint: **tend** $>$ **t0**.

NE_REAL_ARRAY

On entry, **times**$[\langle value \rangle] = \langle value \rangle$, **t0** $= \langle value \rangle$ and **tend** $= \langle value \rangle$.
Constraint: **t0** $<$ **times**$[i-1] <$ **tend** for all i.

On entry, **times**$[\langle value \rangle]$ and **times**$[\langle value \rangle]$ both equal $\langle value \rangle$.
Constraint: all elements of **times** must be unique.

7 Accuracy

Not applicable.

8 Parallelism and Performance

Not applicable.

9 Further Comments

The efficient implementation of a Brownian bridge algorithm requires the use of a workspace array called the *working stack*. Since previously computed points will be used to interpolate new points, they should be kept close to the hardware processing units so that the data can be accessed quickly. Ideally the whole stack should be held in hardware cache. Different bridge construction orders may require different amounts of working stack. Indeed, a naive bridge algorithm may require a stack of size $\frac{N}{4}$ or even $\frac{N}{2}$, which could be very inefficient when N is large. nag_rand_bb_init (g05xac) performs a detailed analysis of the bridge construction order specified by **times**. Heuristics are used to find an execution strategy which requires a small working stack, while still constructing the bridge in the order required.

10 Example

This example calls nag_rand_bb_init (g05xac), nag_rand_bb (g05xbc) and nag_rand_bb_make_bridge_order (g05xec) to generate two sample paths of a three-dimensional free Wiener process. Pseudorandom variates are used to construct the sample paths.

See Section 10 in nag_rand_bb (g05xbc) and nag_rand_bb_make_bridge_order (g05xec) for additional examples.

10.1 Program Text

```
/* nag_rand_bb_init (g05xac) Example Program.
 *
 * Copyright 2013 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(Integer nelements, double * z);
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 calling %s\n%s\n",name,fail.message); exit_status=-1; goto END; }

int main(void)
{
#define C(I,J) c[(J-1)*pdc + I-1]
  Integer exit_status = 0;
  NagError fail;
  /*  Scalars */
  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;
  INIT_FAIL(fail);

  /* Parameters which determine the bridge */
  ntimes = 10;
  t0 = 0.0;
  npaths = 2;
  a = 0;
  nmove = 0;
  d = 3;
  pdz = d*(ntimes+1-a);
  pdb = d*(ntimes+1);
  pdc = d;
  /* 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(pdc * d, double)) ||
          !( z = NAG_ALLOC(pdz * npaths, double)) ||
          !( b = NAG_ALLOC(pdb * 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);

  /* g05xec.  Creates a Brownian bridge construction order */
  /* out of a set of input times */
  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);

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

  /* We want the following covariance matrix*/
  C( 1,1 ) = 6.0;
  C( 2,1 ) = 1.0;
  C( 3,1 ) = -0.2;
  C( 1,2 ) = 1.0;
  C( 2,2 ) = 5.0;
  C( 3,2 ) = 0.3;
  C( 1,3 ) = -0.2;
  C( 2,3 ) = 0.3;
  C( 3,3 ) = 4.0;
  /* nag_rand_bb uses the Cholesky factorization of the covariance matrix C */
  /* f07fdc. Cholesky factorization of real positive definite matrix */
  nag_dpotrf(Nag_ColMajor, Nag_Lower, d, c, pdc, &fail);
  CHECK_FAIL("nag_dpotrf",fail);

  /* Generate the random numbers */
  if( get_z(npaths*d*(ntimes+1-a), z) != 0)
  {
      printf("Error generating random numbers\n");
      exit_status = -1;
      goto END;
  }

  for(i=0; i<d; i++) start[i] = 0.0;
  /* g05xbc. Generate paths for a free or non-free Wiener process using the */
  /* Brownian bridge algorithm   */
  nag_rand_bb(Nag_RowMajor, npaths, d, start, a, term, z, pdz, c, pdc, b, pdb,
              rcomm, &fail);
  CHECK_FAIL("nag_rand_bb",fail);

  /* Display the results*/
  display_results(Nag_RowMajor, 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(Integer nelements, double * z)
{
  NagError fail;
  Integer lseed, lstate, exit_status=0;
  /*  Arrays */
  Integer seed[1];
  Integer  state[80];
  lstate = 80;
  lseed = 1;
  INIT_FAIL(fail);

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

  /* g05skc.  Generates a vector of pseudorandom numbers from */
  /* a Normal distribution */
  nag_rand_normal(nelements, 0.0, 1.0, state, z, &fail);
  CHECK_FAIL("nag_rand_normal",fail);

END: 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_init (g05xac) Example Program Results\n\n");
  for ( p=1; p<=npaths; p++)
    {
      printf("Wiener Path ");
      printf("%1ld ", p);
      printf(",  ");
      printf("%1ld ",  ntimes + 1);
      printf(" time steps, ");
      printf("%1ld ",  d);
      printf(" dimensions \n");

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


      for (i=1; i<= ntimes+1; i++)
        {
              printf("%2ld ", i);
              for (k=1; k<=d; k++)
                  {
                      printf("%10.4f", B(p, k + (i-1)*d));

                  }
              printf("\n");

        }
      printf("\n");
    }
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_rand_bb_init (g05xac) Example Program Results

 Wiener Path  1 ,  11  time steps,  3  dimensions
         1          2          3
 1     1.6020    0.5611    1.6975
 2     1.2767    0.3972   -1.7199
 3    -0.1895   -0.8812   -5.1908
 4    -2.8083   -4.4484   -6.7697
 5    -5.6251   -6.0375   -3.2551
 6    -6.5404   -6.2009   -5.5638
 7    -4.6398   -4.9675   -7.4454
 8    -5.3501   -4.8563   -9.9002
 9    -7.1683   -7.2638   -9.7825
10    -1.9440   -7.0725  -10.7577
11    -4.9941   -3.5442  -10.1561

Wiener Path  2 ,  11  time steps,  3  dimensions
         1          2          3
 1     2.6097    6.2430    0.0316
 2     3.5442    4.2836    2.5742
 3     1.3068    6.1511    4.5362
 4     2.7487    8.6021    2.6880
 5     3.4584    6.1778   -0.6274
 6     0.5965    8.3014    0.5933
 7    -3.2701    5.4787    1.0727
 8    -4.7527    7.0988    0.9120
 9    -4.9375    7.9486    0.7657
10    -7.1302    7.3180    0.2706
11    -0.6289    9.8866   -2.2762
```