NAG Library Function Document

nag_rand_bb_inc (g05xdc)

1 Purpose

nag_rand_bb_inc (g05xdc) computes scaled increments of sample paths of a free or non-free Wiener process, where the sample paths are constructed by a Brownian bridge algorithm. The initialization function nag_rand_bb_inc_init (g05xcc) must be called prior to the first call to nag_rand_bb_inc (g05xdc).

2 Specification

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

```
void nag_rand_bb_inc (Nag_OrderType order, Integer npaths, Integer d,
     Integer a, const double diff[], 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_inc_init (g05xcc). 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.

Fix two times $t_0 < T$, let $(t_i)_{1 \leq i \leq N}$ be any set of time points satisfying $t_0 < t_1 < t_2 < \cdots < t_N < T$, and let X_{t_0}, $(X_{t_i})_{1 \leq i \leq N}$, X_T denote a d-dimensional Wiener sample path at these time points.

The Brownian bridge increments generator uses the Brownian bridge algorithm to construct sample paths for the (free or non-free) Wiener process X, and then uses this to compute the *scaled Wiener increments*

$$\frac{X_{t_1} - X_{t_0}}{t_1 - t_0}, \frac{X_{t_2} - X_{t_1}}{t_2 - t_1}, \ldots, \frac{X_{t_N} - X_{t_{N-1}}}{t_N - t_{N-1}}, \frac{X_T - X_{t_N}}{T - t_N}$$

The example program in Section 10 shows how these increments can be used to compute a numerical solution to a stochastic differential equation (SDE) driven by a (free or non-free) Wiener process.

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_inc_init (g05xcc).

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 2.3.1.3 in How to Use the NAG Library and its Documentation 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.

Constraint: **npaths** ≥ 1.

3: **d** – Integer *Input*

On entry: the dimension of each Wiener sample path.

Constraint: **d** ≥ 1.

4: **a** – Integer *Input*

On entry: if **a** $= 0$, a free Wiener process is created and **diff** is ignored.

If **a** $= 1$, a non-free Wiener process is created where **diff** is the difference between the terminal value and the starting value of the process.

Constraint: **a** $= 0$ or 1.

5: **diff**[**d**] – const double *Input*

On entry: the difference between the terminal value and starting value of the Wiener process. If **a** $= 0$, **diff** is ignored.

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

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

> **pdz** $\times$ **npaths** when **order** $=$ Nag_RowMajor;
> **pdz** $\times$ (**d** $\times$ $(N + 1 -$ **a**$)$) when **order** $=$ Nag_ColMajor.

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

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

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

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

On exit: the Normal random numbers premultiplied by **c**.

7: **pdz** – Integer *Input*

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

Constraints:

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

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

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

The (i, j)th element of the matrix C is stored in **c**$[(j - 1) \times$ **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.

9: **pdc** – Integer *Input*

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

Constraint: **pdc** $\geq$ **d**.

10: **b**[dim] – double *Output*

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

pdb $\times$ **npaths** when **order** $=$ Nag_RowMajor;
pdb $\times$ (**d** $\times$ $(N+1)$) when **order** $=$ 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 scaled Wiener increments.

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 increment $\frac{\left(X^k_{p,i} - X^k_{p,i-1}\right)}{(t_i - t_{i-1})}$ is stored at $B(p, k + (i-1) \times$ **d**$)$.

11: **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**.

12: **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_inc_init (g05xcc) or nag_rand_bb_inc (g05xdc).

On entry: communication array as returned by the last call to nag_rand_bb_inc_init (g05xcc) or nag_rand_bb_inc (g05xdc). This array MUST NOT be directly modified.

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

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_ARRAY_SIZE

On entry, **pdb** $= \langle value \rangle$ and **d** $\times$ (**ntimes** $+ 1) = \langle value \rangle$.
Constraint: **pdb** $\geq$ **d** $\times$ (**ntimes** $+ 1$).

On entry, **pdb** $= \langle value \rangle$ and **npaths** $= \langle value \rangle$.
Constraint: **pdb** $\geq$ **npaths**.

On entry, **pdc** $= \langle value \rangle$.
Constraint: **pdc** $\geq \langle value \rangle$.

On entry, **pdz** $= \langle value \rangle$ and **d** $\times$ (**ntimes** $+ 1 -$ **a**$) = \langle value \rangle$.
Constraint: **pdz** $\geq$ **d** $\times$ (**ntimes** $+ 1 -$ **a**).

On entry, **pdz** $= \langle value \rangle$ and **npaths** $= \langle value \rangle$.
Constraint: **pdz** $\geq$ **npaths**.

NE_BAD_PARAM

On entry, argument $\langle value \rangle$ 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 2.7.6 in How to Use the NAG Library and its Documentation for further information.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 2.7.5 in How to Use the NAG Library and its Documentation for further information.

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_rand_bb_inc (g05xdc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_rand_bb_inc (g05xdc) 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

The scaled Wiener increments produced by this function can be used to compute numerical solutions to stochastic differential equations (SDEs) driven by (free or non-free) Wiener processes. Consider an SDE of the form

$$dY_t = f(t, Y_t)\mathrm{dt} + \sigma(t, Y_t)dX_t$$

on the interval $[t_0, T]$ where $(X_t)_{t_0 \le t \le T}$ is a (free or non-free) Wiener process and f and σ are suitable functions. A numerical solution to this SDE can be obtained by the Euler–Maruyama method. For any discretization $t_0 < t_1 < t_2 < \cdots < t_{N+1} = T$ of $[t_0, T]$, set

$$Y_{t_{i+1}} = Y_{t_i} + f(t_i, Y_{t_i})(t_{i+1} - t_i) + \sigma(t_i, Y_{t_i})\left(X_{t_{i+1}} - X_{t_i}\right)$$

for $i = 1, \ldots, N$ so that $Y_{t_{N+1}}$ is an approximation to Y_T. The scaled Wiener increments produced by nag_rand_bb_inc (g05xdc) can be used in the Euler–Maruyama scheme outlined above by writing

$$Y_{t_{i+1}} = Y_{t_i} + (t_{i+1} - t_i)\left(f(t_i, Y_{t_i}) + \sigma(t_i, Y_{t_i})\left(\frac{X_{t_{i+1}} - X_{t_i}}{t_{i+1} - t_i}\right)\right).$$

The following example program uses this method to solve the SDE for geometric Brownian motion

$$dS_t = rS_t \mathrm{dt} + \sigma S_t dX_t$$

where X is a Wiener process, and compares the results against the analytic solution

$$S_T = S_0 \exp\left(\left(r - \sigma^2/2\right)T + \sigma X_T\right).$$

Quasi-random variates are used to construct the Wiener increments.

10.1 Program Text

```
/* nag_rand_bb_inc (g05xdc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */
#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(Integer npaths, Integer ntimes,
                     double *St, double *analytic);

#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)
{
  Integer exit_status = 0;
  NagError fail;
  /*  Scalars */
  double t0, tend, r, S0, sigma;
  Integer a, d, pdb, pdc, pdz, nmove, npaths, ntimesteps, i, p;
  /*  Arrays */
  double *b = 0, c[1], *t = 0, *rcomm = 0, *diff = 0,
         *times = 0, *z = 0, *analytic = 0, *St = 0;
  Integer *move = 0;
  INIT_FAIL(fail);

  /* We wish to solve the stochastic differential equation (SDE)
   *      dSt = r * St * dt  +  sigma * St * dXt
   * where X is a one dimensional Wiener process.  This means we have
   *   a = 0
   *   d = 1
   *   c = 1
   * We now set the other parameters of the SDE and the Euler-Maruyama scheme
   *
   * Initial value of the process */
```

```
  S0 = 1.0;
  r = 0.05;
  sigma = 0.12;
  /* Number of paths to simulate */
  npaths = 3;
  /* The time interval [t0,T] on which to solve the SDE */
  t0 = 0.0;
  tend = 1.0;
  /* The time steps in the discretization of [t0,T] */
  ntimesteps = 20;
  /* Other bridge parameters */
  c[0] = 1.0;
  a = 0;
  nmove = 0;
  d = 1;
  pdz = d * (ntimesteps + 1 - a);
  pdb = d * (ntimesteps + 1);
  pdc = d;
  /* Allocate memory */
  if (!(t = NAG_ALLOC((ntimesteps), double)) ||
      !(times = NAG_ALLOC((ntimesteps), double)) ||
      !(rcomm = NAG_ALLOC((12 * (ntimesteps + 1)), double)) ||
      !(diff = NAG_ALLOC(d, double)) ||
      !(z = NAG_ALLOC(pdz * npaths, double)) ||
      !(b = NAG_ALLOC(pdb * npaths, double)) ||
      !(move = NAG_ALLOC(nmove, Integer)) ||
      !(St = NAG_ALLOC(npaths * (ntimesteps + 1), double)) ||
      !(analytic = NAG_ALLOC(npaths, double))
         )
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

  /* Fix the time points at which the bridge is required */
  for (i = 0; i < ntimesteps; i++) {
    t[i] = t0 + (i + 1) * (tend - t0) / (double) (ntimesteps + 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, ntimesteps, t,
                                nmove, move, times, &fail);
  CHECK_FAIL("nag_rand_bb_make_bridge_order", fail);

  /* g05xcc. Initializes the generator which backs out
   * the increments of sample paths generated by a Brownian bridge algorithm */
  nag_rand_bb_inc_init(t0, tend, times, ntimesteps, rcomm, &fail);
  CHECK_FAIL("nag_rand_bb_inc_init", fail);

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

  /* nag_rand_bb_inc (g05xdc). Backs out the increments from sample paths
   * generated by a Brownian bridge algorithm */
  nag_rand_bb_inc(Nag_RowMajor, npaths, d, a, diff, z, pdz, c, pdc,
                  b, pdb, rcomm, &fail);
  CHECK_FAIL("nag_rand_bb_inc", fail);

  /* Do the Euler-Maruyama time stepping for SDE
   *      dSt = r * St * dt  +  sigma * St * dXt */
  // Definitions consistent with Nag_RowMajor
#define B(I,J) b[(I-1)*pdb + J-1]
#define ST(I,J) St[(I-1)*(ntimesteps+1) + J-1]
  for (p = 1; p <= npaths; p++) {
    ST(p, 1) = S0 + (t[0] - t0) * (r * S0 + sigma * S0 * B(p, 1));
```

```
  }
  for (i = 2; i <= ntimesteps; i++) {
    for (p = 1; p <= npaths; p++) {
      ST(p, i) = ST(p, i - 1) + (t[i - 1] - t[i - 2]) *
              (r * ST(p, i - 1) + sigma * ST(p, i - 1) * B(p, i));
    }
  }
  for (p = 1; p <= npaths; p++) {
    ST(p, i) = ST(p, i - 1) + (tend - t[ntimesteps - 1]) *
            (r * ST(p, i - 1) + sigma * ST(p, i - 1) * B(p, i));
  }

  /* Compute the analytic solution:
   *       ST = S0*exp( (r-sigma*sigma/2)*T + sigma*WT )
   * where WT =law sqrt(T)Z is the Wiener process at time T.
   *
   * The first quasi-random point in z is always used to compute
   * the final value of WT (equivalently BT, the final value of the
   * Brownian bridge). Hence we have that
   *       WT = sqrt(tend-t0)*z[0]
   */
  for (p = 0; p < npaths; p++) {
    analytic[p] = S0 * exp((r - 0.5 * sigma * sigma) * (tend - t0) +
                           sigma * sqrt(tend - t0) * z[p * (ntimesteps + 1)]);
  }
  /* Display the results */
  display_results(npaths, ntimesteps, St, analytic);
END:
  ;
  NAG_FREE(b);
  NAG_FREE(t);
  NAG_FREE(rcomm);
  NAG_FREE(analytic);
  NAG_FREE(diff);
  NAG_FREE(St);
  NAG_FREE(times);
  NAG_FREE(z);
  NAG_FREE(move);
  return exit_status;
#undef C
}

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

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

  for (i = 0; i < idim; i++) {
    xmean[i] = 0.0;
    stdev[i] = 1.0;
  }
  /* g05skc. Generates a Normal quasi-random number sequence */
  nag_quasi_rand_normal(order, xmean, stdev, npaths, z, pdz, iref, &fail);
  CHECK_FAIL("nag_quasi_rand_normal", fail);

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

void display_results(Integer npaths, Integer ntimesteps,
                     double *St, double *analytic)
{
  Integer i, p;
  printf("nag_rand_bb_inc (g05xdc) Example Program Results\n\n");
  printf("Euler-Maruyama solution for Geometric Brownian motion\n");
  printf("      ");
  for (p = 1; p <= npaths; p++) {
    printf("Path");
    printf("%2" NAG_IFMT "    ", p);
  }
  printf("\n");
  for (i = 1; i <= ntimesteps + 1; i++) {
    printf("%2" NAG_IFMT " ", i);
    for (p = 1; p <= npaths; p++)
      printf("%10.4f", ST(p, i));
    printf("\n");
  }

  printf("\nAnalytic solution at final time step\n");
  printf("    ");
  for (p = 1; p <= npaths; p++) {
    printf("%7s", "Path");
    printf("%2" NAG_IFMT " ", p);
  }
  printf("\n   ");
  for (p = 0; p < npaths; p++)
    printf("%10.4f", analytic[p]);
  printf("\n");
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_rand_bb_inc (g05xdc) Example Program Results

Euler-Maruyama solution for Geometric Brownian motion
       Path 1    Path 2    Path 3
 1     0.9668    1.0367    0.9992
 2     0.9717    1.0254    1.0077
 3     0.9954    1.0333    1.0098
 4     0.9486    1.0226    0.9911
 5     0.9270    1.0113    1.0630
 6     0.8997    1.0127    1.0164
 7     0.8955    1.0138    1.0771
 8     0.8953    0.9953    1.0691
 9     0.8489    1.0462    1.0484
```

```
10     0.8449    1.0592    1.0429
11     0.8158    1.0233    1.0625
12     0.7997    1.0384    1.0729
13     0.8025    1.0138    1.0725
14     0.8187    1.0499    1.0554
15     0.8270    1.0459    1.0529
16     0.7914    1.0294    1.0783
17     0.8076    1.0224    1.0943
18     0.8208    1.0359    1.0773
19     0.8190    1.0326    1.0857
20     0.8217    1.0326    1.1095
21     0.8084    0.9695    1.1389

Analytic solution at final time step
       Path 1    Path 2    Path 3
       0.8079    0.9685    1.1389
```