NAG Library Function Document

nag_rand_bb_make_bridge_order (g05xec)

1 Purpose

nag_rand_bb_make_bridge_order (g05xec) takes a set of input times and permutes them to specify one of several predefined Brownian bridge construction orders. The permuted times can be passed to nag_rand_bb_init (g05xac) or nag_rand_bb_inc_init (g05xcc) to initialize the Brownian bridge generators with the chosen bridge construction order.

2 Specification

```
#include <nag.h>
#include <nagg05.h>
void nag_rand_bb_make_bridge_order (Nag_BridgeOrder bgord, double t0,
     double tend, Integer ntimes, const double intime[], Integer nmove,
     const Integer move[], double times[], NagError *fail)
```

3 Description

The Brownian bridge algorithm (see Glasserman (2004)) is a popular method for constructing a Wiener process at a set of discrete times, $t_0 < t_1 < t_2 <, \ldots, < t_N < T$, for $N \geq 1$. To ease notation we assume that T has the index $N+1$ so that $T = t_{N+1}$. Inherent in the algorithm is the notion of a *bridge construction order* which specifies the order in which the $N+2$ points of the Wiener process, X_{t_0}, X_T and X_{t_i}, for $i = 1, 2, \ldots, N$, are generated. The value of X_{t_0} is always assumed known, and the first point to be generated is always the final time X_T. Thereafter, successive points are generated iteratively by an interpolation formula, using points which were computed at previous iterations. In many cases the bridge construction order is not important, since any construction order will yield a correct process. However, in certain cases, for example when using quasi-random variates to construct the sample paths, the bridge construction order can be important.

3.1 Supported Bridge Construction Orders

nag_rand_bb_make_bridge_order (g05xec) accepts as input an array of time points $t_1, t_2, \ldots, t_N, T$ at which the Wiener process is to be sampled. These time points are then permuted to construct the bridge. In all of the supported construction orders the first construction point is T which has index $N+1$. The remaining points are constructed by iteratively bisecting (sub-intervals of) the *time indices* interval $[0, N+1]$, as Figure 1 illustrates:

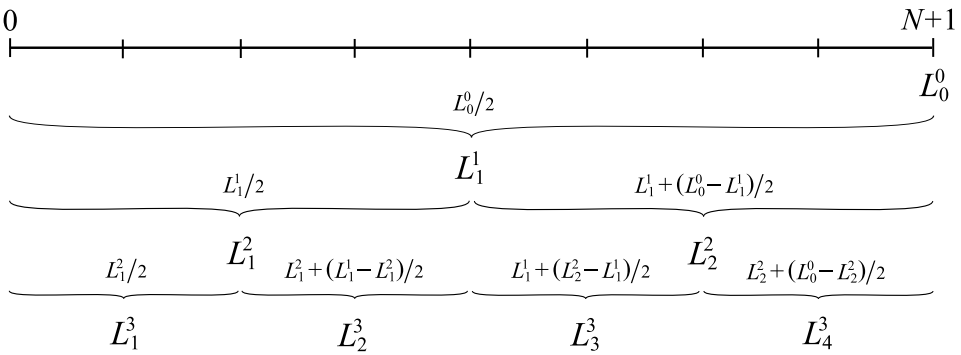

Figure 1

The time indices interval is processed in levels L^i, for $i = 1, 2, \ldots$. Each level L^i contains n_i points $L_1^i, \ldots, L_{n_i}^i$ where $n_i \leq 2^{i-1}$. The number of points at each level depends on the value of N. The points

L_j^i for $i \geq 1$ and $j = 1, 2, \ldots n_i$ are computed as follows: define $L_0^0 = N + 1$ and set

$$L_j^i = J + (K - J)/2 \quad \text{where}$$
$$J = \max\left\{L_k^p : 1 \leq k \leq n_p,\ 0 \leq p < i \text{ and } L_k^p < L_j^i\right\} \quad \text{and}$$
$$K = \min\left\{L_k^p : 1 \leq k \leq n_p,\ 0 \leq p < i \text{ and } L_k^p > L_j^i\right\}$$

By convention the maximum of the empty set is taken to be to be zero. Figure 1 illustrates the algorithm when $N + 1$ is a power of two. When $N + 1$ is not a power of two, one must decide how to round the divisions by 2. For example, if one rounds down to the nearest integer, then one could get the following:

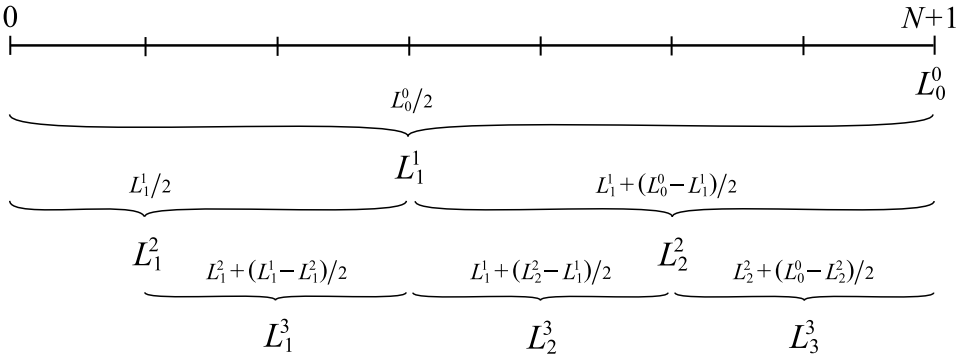

Figure 2

From the series of bisections outlined above, two ways of ordering the time indices L_j^i are supported. In both cases, levels are always processed from coarsest to finest (i.e., increasing i). Within a level, the time indices can either be processed left to right (i.e., increasing j) or right to left (i.e., decreasing j). For example, when processing left to right, the sequence of time indices could be generated as:

$$N+1 \quad L_1^1 \quad L_1^2 \quad L_2^2 \quad L_1^3 \quad L_2^3 \quad L_3^3 \quad L_4^3 \quad \cdots$$

while when processing right to left, the same sequence would be generated as:

$$N+1 \quad L_1^1 \quad L_2^2 \quad L_1^2 \quad L_4^3 \quad L_3^3 \quad L_2^3 \quad L_1^3 \quad \cdots$$

nag_rand_bb_make_bridge_order (g05xec) therefore offers four bridge construction methods; processing either left to right or right to left, with rounding either up or down. Which method is used is controlled by the **bgord** argument. For example, on the set of times

$$t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad t_8 \quad t_9 \quad t_{10} \quad t_{11} \quad t_{12} \quad T$$

the Brownian bridge would be constructed in the following orders:

bgord = Nag_LRRoundDown (processing left to right, rounding down)

$$T \quad t_6 \quad t_3 \quad t_9 \quad t_1 \quad t_4 \quad t_7 \quad t_{11} \quad t_2 \quad t_5 \quad t_8 \quad t_{10} \quad t_{12}$$

bgord = Nag_LRRoundUp (processing left to right, rounding up)

$$T \quad t_7 \quad t_4 \quad t_{10} \quad t_2 \quad t_6 \quad t_9 \quad t_{12} \quad t_1 \quad t_3 \quad t_5 \quad t_8 \quad t_{11}$$

bgord = Nag_RLRoundDown (processing right to left, rounding down)

$$T \quad t_6 \quad t_9 \quad t_3 \quad t_{11} \quad t_7 \quad t_4 \quad t_1 \quad t_{12} \quad t_{10} \quad t_8 \quad t_5 \quad t_2$$

bgord = Nag_RLRoundUp (processing right to left, rounding up)

$$T \quad t_7 \quad t_{10} \quad t_4 \quad t_{12} \quad t_9 \quad t_6 \quad t_2 \quad t_{11} \quad t_8 \quad t_5 \quad t_3 \quad t_1 .$$

The four construction methods described above can be further modified through the use of the input array **move**. To see the effect of this argument, suppose that an array A holds the output of nag_rand_bb_make_bridge_order (g05xec) when **nmove** = 0 (i.e., the bridge construction order as specified by **bgord** only). Let

$$B = \{t_j : j = \mathbf{move}[i-1], i = 1, 2, \ldots, \mathbf{nmove}\}$$

be the array of all times identified by **move**, and let C be the array A with all the elements in B removed, i.e.,

$$C = \{A(i) : A(i) \neq B(j), i = 1, 2, \ldots, \mathbf{ntimes}, j = 1, 2, \ldots, \mathbf{nmove}\}.$$

Then the output of nag_rand_bb_make_bridge_order (g05xec) when $\mathbf{nmove} > 0$ is given by

$$B(1) \quad B(2) \quad \cdots \quad B(\mathbf{nmove}) \quad C(1) \quad C(2) \quad \cdots \quad C(\mathbf{ntimes} - \mathbf{nmove})$$

When the Brownian bridge is used with quasi-random variates, this functionality can be used to allow specific sections of the bridge to be constructed using the lowest dimensions of the quasi-random points.

4 References

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

5 Arguments

1: **bgord** – Nag_BridgeOrder *Input*

On entry: the bridge construction order to use.

Constraint: **bgord** = Nag_LRRoundDown, Nag_LRRoundUp, Nag_RLRoundDown or Nag_RLRoundUp.

2: **t0** – double *Input*

On entry: t_0, the start value of the time interval on which the Wiener process is to be constructed.

3: **tend** – double *Input*

On entry: T, the largest time at which the Wiener process is to be constructed.

4: **ntimes** – Integer *Input*

On entry: N, the number of time points in the Wiener process, excluding t_0 and T.

Constraint: $\mathbf{ntimes} \geq 1$.

5: **intime**[**ntimes**] – const double *Input*

On entry: the time points, $t_1, t_2, \ldots, t_N$, at which the Wiener process is to be constructed. Note that the final time T is not included in this array.

Constraints:

$\mathbf{t0} < \mathbf{intime}[i-1]$ and $\mathbf{intime}[i-1] < \mathbf{intime}[i]$, for $i = 1, 2, \ldots, \mathbf{ntimes} - 1$;
$\mathbf{intime}[\mathbf{ntimes} - 1] < \mathbf{tend}$.

6: **nmove** – Integer *Input*

On entry: the number of elements in the array **move**.

Constraint: $0 \leq \mathbf{nmove} \leq \mathbf{ntimes}$.

7: **move**[**nmove**] – const Integer *Input*

On entry: the indices of the entries in **intime** which should be moved to the front of the **times** array, with $\mathbf{move}[j-1] = i$ setting the jth element of **times** to t_i. Note that i ranges from 1 to **ntimes**. When $\mathbf{nmove} = 0$, **move** is not referenced and may be **NULL**.

Constraint: $1 \leq \mathbf{move}[j-1] \leq \mathbf{ntimes}$, for $j = 1, 2, \ldots, \mathbf{nmove}$.

The elements of **move** must be unique.

8: **times**[**ntimes**] – double *Output*

On exit: the output bridge construction order. This should be passed to nag_rand_bb_init (g05xac) or nag_rand_bb_inc_init (g05xcc).

9: **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 3.2.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

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

NE_INT

On entry, **nmove** = ⟨*value*⟩ and **ntimes** = ⟨*value*⟩.
Constraint: $0 \leq$ **nmove** $\leq$ **ntimes**.

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

NE_INT_ARRAY

On entry, **move**[⟨*value*⟩] = ⟨*value*⟩.
Constraint: **move**$[i] \geq 1$ for all i.

On entry, **move**[⟨*value*⟩] = ⟨*value*⟩ and **ntimes** = ⟨*value*⟩.
Constraint: **move**$[i] \leq$ **ntimes** for all i.

On entry, **move**[⟨*value*⟩] and **move**[⟨*value*⟩] both equal ⟨*value*⟩.
Constraint: all elements in **move** must be unique.

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 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 3.6.5 in How to Use the NAG Library and its Documentation for further information.

NE_NOT_STRICTLY_INCREASING

On entry, **intime**[⟨*value*⟩] = ⟨*value*⟩ and **intime**[⟨*value*⟩] = ⟨*value*⟩.
Constraint: the elements in **intime** must be in increasing order.

NE_REAL_2

On entry, **intime**[0] = ⟨*value*⟩ and **t0** = ⟨*value*⟩.
Constraint: **intime**[0] > **t0**.

On entry, **ntimes** = ⟨*value*⟩, **intime**[**ntimes** − 1] = ⟨*value*⟩ and **tend** = ⟨*value*⟩.
Constraint: **intime**[**ntimes** − 1] < **tend**.

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_rand_bb_make_bridge_order (g05xec) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

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' Notefor your implementation for any additional implementation-specific information.

9 Further Comments

None.

10 Example

This example calls nag_rand_bb_make_bridge_order (g05xec), nag_rand_bb_init (g05xac) and nag_rand_bb (g05xbc) to generate two sample paths of a three-dimensional free Wiener process. The array **move** is used to ensure that a certain part of the sample path is always constructed using the lowest dimensions of the input quasi-random points. For further details on using quasi-random points with the Brownian bridge algorithm, please see Section 2.6 in the g05 Chapter Introduction.

10.1 Program Text

```
/* nag_rand_bb_make_bridge_order (g05xec) 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(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)
{
  Integer exit_status = 0;
  NagError fail;
  /*  Scalars */
  double t0, tend;
  Integer a, d, pdb, pdc, pdz, nmove, npaths, ntimes, i;
  Nag_OrderType order;
  /*  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 = 3; /* We modify the first 3 points in the construction order */
  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(pdc * d, 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 + 1.71 * (double) (i + 1);
  }
  tend = t0 + 1.71 * (double) (ntimes + 1);
  /* We will use the bridge construction order Nag_RLRoundDown. However
   * we modify this construction order by moving points 3, 5 and 4
   * to the front.  Modifying a construction order would typically only
   * be considered when using quasi-random numbers */
  move[0] = 3;
  move[1] = 5;
  move[2] = 4;

  /* nag_rand_bb_make_bridge_order (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);

  /* 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;
  /* g05xbc works with 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(order, ntimes, d, a, npaths, z, pdz) != 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(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)
{
  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 number 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;
  }
  /* g05yjc. 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(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_make_bridge_order (g05xec) Example Program Results\n\n");
  for (p = 1; p <= npaths; p++) {
    printf("%s ", "Wiener Path ");
    printf("%1" NAG_IFMT " ", p);
    printf("%s ", ", ");
    printf("%1" NAG_IFMT " ", ntimes + 1);
    printf("%s ", " time steps, ");
    printf("%1" NAG_IFMT " ", d);
    printf("%s ", " dimensions");
    printf("\n");

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

    for (i = 1; i <= ntimes + 1; i++) {
      printf("%2" NAG_IFMT " ", 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_make_bridge_order (g05xec) Example Program Results

Wiener Path  1 ,  11  time steps,  3  dimensions
         1          2          3
 1    -2.1275   -2.4995   -6.0191
 2    -6.1589   -1.3257   -3.7378
 3    -5.1917   -3.1653   -6.2291
 4   -11.5557   -5.9183   -5.9062
 5    -9.2492   -5.7497   -4.2989
 6    -6.7853  -13.9759   -0.8990
 7   -12.7642  -15.6386   -3.6481
 8   -12.5245  -11.8142    3.3504
 9   -15.1995  -15.5145    0.5355
10   -16.0360  -14.4140    0.0104
11   -22.6719  -14.3308   -0.2418
```

```
Wiener Path  2 ,  11  time steps,  3  dimensions
          1           2           3
 1    -0.0973    3.7229    0.8640
 2     0.8027    8.5041   -0.9103
 3    -3.8494    6.1062    0.1231
 4    -6.6643    4.9936   -0.1329
 5    -6.8095    9.3508    4.7022
 6    -7.7178   10.9577   -1.4262
 7    -8.0711   12.7207    4.4744
 8   -12.8353    8.8296    7.6458
 9    -7.9795   12.2399    7.3783
10    -6.4313   10.0770    5.5234
11    -6.6258   10.3026    6.5021
```