NAG Toolbox

nag_rand_bb_make_bridge_order (g05xe)

1 Purpose

nag_rand_bb_make_bridge_order (g05xe) 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 (g05xa) or nag_rand_bb_inc_init (g05xc) to initialize the Brownian bridge generators with the chosen bridge construction order.

2 Syntax

```
[times, ifail] = nag_rand_bb_make_bridge_order(t0, tend, intime, move, 'bgord', bgord, 'ntimes', ntimes, 'nmove', nmove)
[times, ifail] = g05xe(t0, tend, intime, move, 'bgord', bgord, 'ntimes', ntimes, 'nmove', nmove)
```

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 (g05xe) 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:

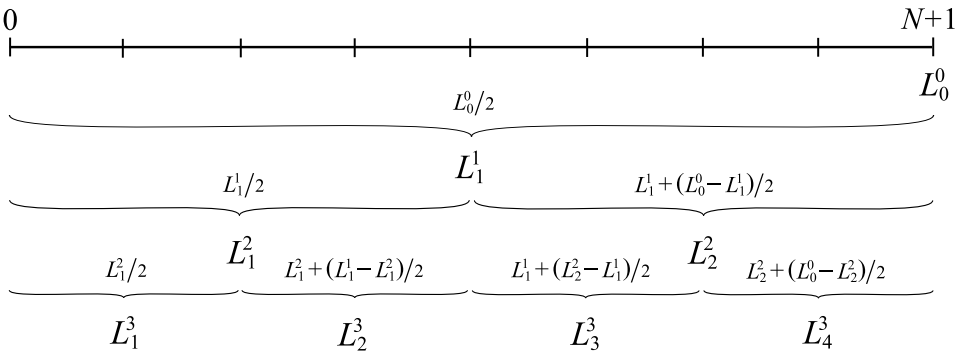

Figure 1

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

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

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:

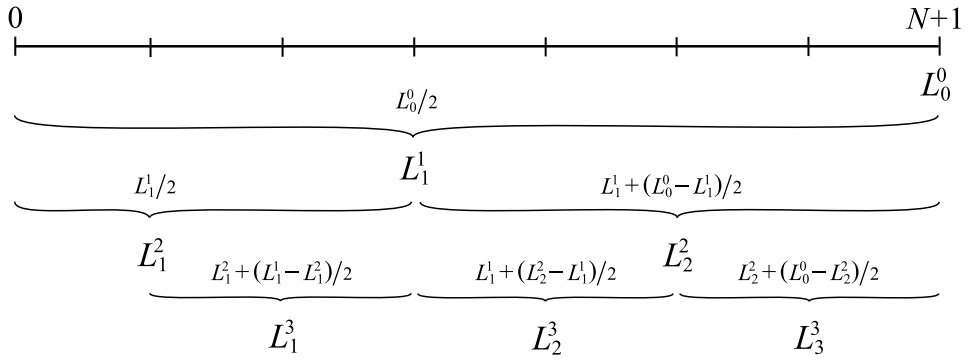

Figure 2

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 (g05xe) 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 $= 1$ (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 $= 2$ (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 $= 3$ (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 $= 4$ (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 (g05xe) when **nmove** $= 0$ (i.e., the bridge construction order as specified by **bgord** only). Let

$$B = \{t_j : j = \mathbf{move}(i), i = 1, 2, \ldots, \mathbf{nmove}\}$$

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 (g05xe) 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 Parameters

5.1 Compulsory Input Parameters

1: **t0** – REAL (KIND=nag_wp)

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

2: **tend** – REAL (KIND=nag_wp)

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

3: **intime**(**ntimes**) – REAL (KIND=nag_wp) array

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:

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

4: **move**(**nmove**) – INTEGER array

The indices of the entries in **intime** which should be moved to the front of the **times** array, with $\mathbf{move}(j) = i$ 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.

Constraint: $1 \leq \mathbf{move}(j) \leq \mathbf{ntimes}$, for $j = 1, 2, \ldots, \mathbf{nmove}$.

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

5.2 Optional Input Parameters

1: **bgord** – INTEGER

Default: 1

The bridge construction order to use.

Constraint: **bgord** = 1, 2, 3 or 4.

2: **ntimes** – INTEGER

Default: the dimension of the array **intime**.

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

Constraint: **ntimes** ≥ 1.

3: **nmove** – INTEGER

Default: the dimension of the array **move**.

The number of elements in the array **move**.

Constraint: $0 \leq$ **nmove** $\leq$ **ntimes**.

5.3 Output Parameters

1: **times**(**ntimes**) – REAL (KIND=nag_wp) array

The output bridge construction order. This should be passed to nag_rand_bb_init (g05xa) or nag_rand_bb_inc_init (g05xc).

2: **ifail** – INTEGER

ifail $= 0$ unless the function detects an error (see Section 5).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

ifail $= 1$

Constraint: **bgord** $= 1, 2, 3$ or 4

ifail $= 2$

Constraint: **ntimes** ≥ 1.

ifail $= 3$

Constraint: $0 \leq$ **nmove** $\leq$ **ntimes**.

ifail $= 4$

Constraint: **intime**$(1) >$ **t0**.

Constraint: **intime**(**ntimes**) $<$ **tend**.

Constraint: the elements in **intime** must be in increasing order.

ifail $= 5$

Constraint: **move**$(i) \leq$ **ntimes** for all i.

Constraint: **move**$(i) \geq 1$ for all i.

ifail $= 6$

Constraint: all elements in **move** must be unique.

ifail $= -99$

An unexpected error has been triggered by this routine. Please contact NAG.

ifail $= -399$

Your licence key may have expired or may not have been installed correctly.

ifail $= -999$

Dynamic memory allocation failed.

7 Accuracy

Not applicable.

8 Further Comments

None.

9 Example

This example calls nag_rand_bb_make_bridge_order (g05xe), nag_rand_bb_init (g05xa) and nag_rand_bb (g05xb) 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.

9.1 Program Text

```
function g05xe_example

fprintf('g05xe example results\n\n');

% Get information required to set up the bridge
[bgord,t0,tend,ntimes,intime,nmove,move] = get_bridge_init_data();

% Make the bridge construction bgord
[times, ifail] = g05xe( ...
                        t0, tend, intime, move, 'bgord', bgord);

% Initialize the Brownian bridge generator
[rcomm, ifail] = g05xa( ...
                        t0, tend, times);

% Get additional information required by the bridge generator
[npaths,d,start,a,term,c] = get_bridge_gen_data();

% Generate the Z values
[z] = get_z(npaths, d, a, ntimes);

% Call the Brownian bridge generator routine
[z, b, ifail] = g05xb( ...
                       npaths, start, term, z, c, rcomm, 'a', a);

% Display the results
for i = 1:npaths
  fprintf('Weiner Path %d, %d time steps, %d dimensions\n', i, ntimes+1, d);
  w = transpose(reshape(b(:,i), d, ntimes+1));

  ifail = x04ca('G', ' ', w, '');

  fprintf('\n');
end

function [bgord,t0,tend,ntimes,intime,nmove,move] = get_bridge_init_data()
  % Set the basic parameters for a Wiener process
  t0 = 0;
  ntimes = nag_int(10);

  % We want to generate the Wiener process at these time points
  intime = 1.71*double(1:ntimes) + t0;
  tend = t0 + 1.71*double(ntimes + 1);
```

```
  % We suppose the following 3 times are very important and should be
  % constructed first. Note: these are indices into intime
  nmove= nag_int(3);
  move = [nag_int(3), 5, 4];
  bgord = nag_int(3);

function [npaths,d,start,a,term,c] = get_bridge_gen_data();
  % Set the basic parameters for a non-free Wiener process
  npaths = nag_int(2);
  d = 3;
  a = nag_int(0);
  start = zeros(d, 1);
  term  = zeros(d, 1);

  % As a = 0, term need not be initialized

  % We want the following covariance matrix
  c = [ 6,   1,  -0.2;
        1,   5,   0.3;
       -0.2, 0.3, 4  ];

  % Cholesky factorize of the covariance matrix c
  [c, info] = f07fd('l', c);

function [z] = get_z(npaths, d, a, ntimes)
  idim = d*(ntimes+1-a);

  % We now need to generate the input pseudorandom points

  % First initialize the base pseudorandom number generator
  state = initialize_prng(nag_int(6), nag_int(0), [nag_int(1023401)]);

  % Scrambled quasi-random sequences preserve the good discrepancy
  % properties of quasi-random sequences while counteracting the bias
  % some applications experience when using quasi-random sequences.
  % Initialize the scrambled quasi-random generator.
  [iref, state] = initialize_scrambled_qrng(nag_int(1), nag_int(2), ...
                                            idim, state);

  % Generate the quasi-random points from N(0,1)
  xmean = zeros(idim, 1);
  std   = ones(idim, 1);
  [z, iref, ifail] = g05yj( ...
                            xmean, std, npaths, iref);
  z = z';

function [state] = initialize_prng(genid, subid, seed)
  % Initialize the generator to a repeatable sequence
  [state, ifail] = g05kf( ...
                          genid, subid, seed);

function [iref, state] = initialize_scrambled_qrng(genid,stype,idim,state)
  iskip = nag_int(0);
  nsdigits = nag_int(32);
  [iref, state, ifail] = g05yn( ...
                                genid, stype, nag_int(idim), ...
                                iskip, nsdigits, state);
```

9.2 Program Results

```
     g05xe example results

Weiner 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

Weiner 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
```