

NAG Numerical Functions for GPUs Function Document

nag_gpu_depthbb

1 Purpose

nag_gpu_depthbb constructs sample paths for a Brownian bridge or for a free Brownian motion using a depth-order bridge interpolation algorithm. It must be preceded by a call to the initialization function `nag_gpu_depthbb_init`, and must finally be followed by a call to `nag_gpu_depthbb_cleanup`.

All references to `bridgeisfree` and `nTimes` below refer to the values that were passed to the initialization function `nag_gpu_depthbb_init`.

2 Specification

```
#include <nag_gpu.h>
```

```
void nag_gpu_depthbb (int npath, int dim, float bridge_start, float bridge_end,
    const float *d_Z, float *d_W, const float *h_cov, Nag_Gpu_Comm *comm,
    bool copyCov = true)
```

3 Description

3.1 Background

Fix $T > 0$ and let $W = (W_t)_{0 \leq t \leq T}$ be a standard d -dimensional Wiener process. A *standard* d -dimensional Brownian bridge $B = (B_t)_{0 \leq t \leq T}$ is defined (see Revuz and Yor (1999)) as

$$B_t = W_t - \frac{t}{T}W_T$$

for all $t \in [0, T]$. This process is continuous, starts at zero at time 0 and ends at zero at time T . It is Gaussian, has zero mean and has a covariance structure given by

$$(B_s B_t') = s \left(1 - \frac{t}{T}\right) I_d = \frac{s(T-t)}{T} I_d$$

for any $s \leq t$ in $[0, T]$ where I_d is the d -dimensional identity matrix. The Brownian bridge is often called a non-free or ‘pinned’ Brownian motion, since it is forced to be equal to 0 at time T but is otherwise very similar to a standard Brownian motion.

We can generalize this construction as follows. Fix points $x, w \in \mathbb{R}^d$, let Σ be a $d \times d$ covariance matrix and choose any $d \times d$ matrix C such that $CC' = \Sigma$. We will define the *generalized* d -dimensional Brownian bridge $X = (X_t)_{0 \leq t \leq T}$ by setting

$$X_t = \frac{tw + (T-t)x}{T} + CB_t = \frac{tw + (T-t)x}{T} + CW_t - \frac{t}{T}CW_T$$

for all $t \in [0, T]$. The process X is therefore continuous, starts at x at time zero and ends at w at time T . It has time-dependent mean $(tw + (T-t)x)/T$ and has the covariance structure

$$(X_s - X_s)(X_t - X_t)' = (CB_s B_t' C') = \frac{s(T-t)}{T} CC' = \frac{s(T-t)}{T} \Sigma$$

for all $s \leq t$ in $[0, T]$. This is a non-free bridge since it is forced to be equal to w at time T . However if we set $w = x + CW_T$, then X simplifies to

$$X_t = x + CW_t$$

for all $t \in [0, T]$ which is a free d -dimensional Brownian motion with covariance given by Σ .

3.2 Implementation

The bridge is generated in a modified depth-first order. Suppose there are N time points t_1, \dots, t_N at which the bridge is to be computed. The algorithm starts by taking the known values $X_{t_0} = x$ and $X_{t_N} = w$ and then generating

$$X_{t_{\lfloor N/2 \rfloor}}, X_{t_{\lfloor N/4 \rfloor}}, X_{t_{\lfloor N/8 \rfloor}}, \dots, X_{t_1}$$

according to the standard Brownian bridge interpolation formula (see Glasserman (2004)). Once X_{t_1} is reached, the algorithm moves upwards from t_1 searching for an interval $[t_i, t_k]$ such that both X_{t_i} and X_{t_k} are already known, but all X_{t_j} for $i < j < k$ are not. This interval is then treated in the same way as the interval $[t_0, t_N]$, and the process repeats until all points are computed.

The main input to the bridge algorithm is an array of standard Normal random numbers. If these come from a quasi-random generator (e.g., Sobol numbers), then the order in which these numbers are used becomes important. Suppose that the bridge is one-dimensional and that we have an N -dimensional quasi-random point. Roughly speaking, the algorithm uses the dimensions in this point in *breadth-first* order: the first dimension is used to compute $X_{t_{\lfloor N/2 \rfloor}}$, the second dimension is used to compute $X_{t_{\lfloor N/4 \rfloor}}$, the third to compute $X_{t_{\lfloor 3N/4 \rfloor}}$, the fourth to compute $X_{t_{\lfloor N/8 \rfloor}}$ and so on. For a d -dimensional bridge, and corresponding $N \times d$ dimensional quasi-random point, the first d dimensions are used to compute $X_{t_{\lfloor N/2 \rfloor}}$, the second d to compute $X_{t_{\lfloor N/4 \rfloor}}$, the third d to compute $X_{t_{\lfloor 3N/4 \rfloor}}$, and so on. If the bridge is free, in other words $X_t = x + CW_t$, then the first d dimensions are used to compute X_{t_N} , the second d to compute $X_{t_{\lfloor N/2 \rfloor}}$, the third d to compute $X_{t_{\lfloor N/4 \rfloor}}$, and so on.

The boolean parameter `bridgeisfree` controls whether a free or non-free Brownian sample path is created. Note that the final value w of the bridge is always stored, whereas the starting value x is never stored. The algorithm therefore only produces the values $X_{t_1}, X_{t_2}, X_{t_3}, \dots, X_{t_N}$.

4 References

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

Revuz D and Yor M (1999) *Continuous Martingales and Brownian Motion* Springer

5 Arguments

- 1: **npath** – int *Input*
On entry: the number of Brownian bridge sample paths to create.
Constraint: $\text{npath} \geq 1$.
- 2: **dim** – int *Input*
On entry: the dimension of each Brownian bridge sample path.
Constraint: $1 \leq \text{dim} \leq 8$.
- 3: **bridge_start** – float *Input*
On entry: the starting value x of the bridge.
- 4: **bridge_end** – float *Input*
On entry: the final value w of the bridge. If `bridgeisfree` is `Nag_TRUE`, this value is ignored and w is set equal to $x + CW_T$.
- 5: **d_z[nTimes × dim × npath]** – const float * *Input*
The array must be allocated in the *device memory space*.

On entry: the Normal random numbers used to construct the bridge.

Constraint: `d_Z` must contain `nTimes × dim × npath` values if `bridgeisfree`, otherwise it must contain `(nTimes - 1) × dim × npath` values. These should be laid out as a matrix with `dim × nTimes` rows and `npath` columns if `bridgeisfree`, otherwise as a matrix with `dim × (nTimes - 1)` rows and `npath` columns. If quasi-random numbers are to be used, the rows should correspond to the dimensions of each quasi-random point.

6: **d_w[`dim × nTimes × npath`]** – float * *Output*

The array must be allocated in the *device memory space*.

On exit: the values of the Brownian bridge. If $x_{p,i}^d$ denotes the d th dimension of the i th point of the p th sample path where $0 \leq d < \text{dim}$, $0 \leq i < \text{nTimes}$ and $0 \leq p < \text{npath}$, then $x_{p,i}^d$ will be stored at `d_w[p+npath(d+i*dim)]`. The starting value `bridge_start` is never stored, while the terminal value `bridge_end` is always stored.

7: **h_cov[`dim × dim`]** – const float * *Input*

The matrix `h_cov` must reside in the host memory space.

On entry: the matrix C which specifies the correlation structure of the Brownian bridge. C should be chosen such that $CC' = \Sigma$ where $\text{ov}(X_s, X_t) = s(T - t)/T\Sigma$ for all $s \leq t$ in $[0, T]$.

8: **comm** – Nag_Gpu_Comm * *Communication Data*

`Nag_Gpu_Comm` is a NAG defined type which holds state and communication information and must not be modified in any way.

On entry: the structure initialized by a previous call to `nag_gpu_depthbb_init`.

9: **copycov** – bool *Input*

Default: true

An optional flag specifying whether the matrix `h_cov` should be copied to the device or not.

On initial entry: following initialization by `nag_gpu_depthbb_init`, the flag `copyCov` must be `Nag_TRUE`.

On intermediate re-entry: `copyCov` may be set to `Nag_FALSE` and need only be set to `Nag_TRUE` again if the matrix `h_cov` changes.

6 Error Indicators and Warnings

No argument constraint checking is carried out by this function. You can insert a call to function `nag_gpu_utilCheckMsg` (contained in the header file `nag_gpu.h`) following the call to **nag_gpu_depthbb** to check for the last CUDA error message, if any, generated during execution.

7 Example

This example program uses **nag_gpu_depthbb** to print three Brownian bridge sample paths where each path is three dimensional. The bridge is pinned to end at a fixed value, and the inputs are quasi-random Normal numbers from the Sobol generator `nag_gpu_sobol_normal`.

7.1 Program Text

```
/* nag_gpu_sobol_test
 *
 * Copyright 2010, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.3, 2010.
 */
```

```

/////////////////////////////////////////////////////////////////
// pre-processor includes
/////////////////////////////////////////////////////////////////
#include <stdio.h>

/////////////////////////////////////////////////////////////////
// include libraries
/////////////////////////////////////////////////////////////////
#include <nag_gpu.h>

/////////////////////////////////////////////////////////////////
// Main program
/////////////////////////////////////////////////////////////////

int main(int argc, char **argv)
{
    FILE *fpout = stdout;

    // Number of time steps in the bridge - must be less than 4096
    const int nTimes = 30;
    // Dimension of the bridge - must be less than 8
    const int dim = 3;
    // Generate batchSize sample paths at a time on the GPU.
    // For large numbers of sample paths this can avoid out-of-memory
    // errors
    const int batchSize = 50;
    // Total number of batches to generate
    const int nBatches = 10;

    float
        // Covariance matrix
        cov[dim*dim],
        // Vector of times at which to compute bridge
        t[nTimes],
        // Host (PC) storage for batch-generated sample paths
        *h_GpuBatchPaths = 0,
        // Device (GPU) storage for required normal random numbers
        *d_z = 0,
        // Device (GPU) storage for batch-generated sample paths
        *d_GpuBatchPaths = 0,
        // Starting value of the bridge
        bridge_start = -0.5,
        // End value of bridge
        bridge_end = 1.5;

    bool copyCov = true;

    Nag_Gpu_Comm
        // Communication structure for GPU generator
        comm;

    // Scrambling vector for GPU Sobol
    unsigned int *v = 0;
    // Offset for Normal random number generator
    const int offset = 1234;

    // Number of sample paths to print
    int nPrint = 2;
    // Make sure we have enough sample paths
    nPrint =
        (nPrint > batchSize*nBatches
         ? batchSize*nBatches : nPrint);
    // Storage for numbers we'll print
    float *h_print = 0;
    // Number of paths stored so far
    int printPath = 0;

```

```

// For MRG generator, if required
//unsigned int v1[] = {1,2,3};
//unsigned int v2[] = {55,34,709};

// Greeting
fprintf(fpout, "\n\n \tNAG GPU Brownian Bridge Example "
        "Program\n\n");

// Set device with highest Gflops/s
cudaSetDevice( nag_gpu_GetMaxGflopsDeviceId() );

// Allocate memory
if (!(h_GpuBatchPaths =
(float*)malloc(sizeof(float)*dim*nTimes*batchSize)) ) {
    fprintf(stderr, "h_GpuBatchPaths allocation failure\n");
    goto END;
}
if (!(cudaSuccess == cudaMalloc((void **)&d_GpuBatchPaths,
    sizeof(float)*dim*nTimes*batchSize))) {
    fprintf(stderr, "d_GpuBatchPaths allocation failure\n");
    goto END;
}
if (!(cudaSuccess == cudaMalloc((void **)&d_Z,
    sizeof(float)*dim*nTimes*batchSize))) {
    fprintf(stderr, "d_Z allocation failure\n");
    goto END;
}
if (!(v =
(unsigned int*)malloc(sizeof(unsigned int)*dim*nTimes)) ) {
    fprintf(stderr, "v allocation failure\n");
    goto END;
}
if (!(h_print =
(float*)malloc(sizeof(float)*dim*nTimes*nPrint)) ) {
    fprintf(stderr, "h_print allocation failure\n");
    goto END;
}

// Set the time points
for(int i = 0; i < nTimes; i++) t[i] = (i+1)*0.477f;
// Cholesky factorisation of Covariance Matrix:
// for simplicity we merely use a correlation matrix,
// however statistically this is wrong
for(int i = 0; i < dim*dim; i++) cov[i] = 0.44f;
for(int i = 0; i < dim; i++) cov[i*(dim+1)] = 1.0f;

// Set scrambling vector to 0 - don't want scrambling
for (int i=0; i < dim*nTimes; i++) v[i] = 0;

// Initialise GPU Sobol generator
nag_gpu_sobol_init(dim*nTimes, offset, v, &comm);
// Initialise GPU Brownian Bridge generator:
// can share comm structure
// among different GPU routines, but not between
// GPU and CPU routines.
nag_gpu_depthbb_init(0, t, nTimes, false, &comm);

for(int batchCount = 0; batchCount < nBatches; batchCount++) {
    //MRG generator if required
    //nag_gpu_mrg32k3a_init(v1, v2,
    // batchCount*nTimes*dim*batchSize+offset);
    //nag_gpu_mrg32k3a_normal(nTimes, dim, batchSize, d_Z);

    // Launch GPU computations asynchronously
    nag_gpu_sobol_normal(batchSize, d_Z, &comm);
    nag_gpu_depthbb(batchSize, dim, bridge_start,
        bridge_end, d_Z, d_GpuBatchPaths, cov, &comm, copyCov);
}

```

```

copyCov = false;

/*
 * One can now launch other kernels to operate on the
 * Brownian Bridge sample paths, or copy the paths to
 * the host and operate on them there.
 * Here we simply copy them to the host in order to
 * print them
 */
cudaMemcpy(h_GpuBatchPaths, d_GpuBatchPaths,
           sizeof(float)*dim*nTimes*batchSize, cudaMemcpyDeviceToHost);

// Copy GPU values to print storage
// Path counter
for(int p = 0; p < batchCount; p++) {
    // Check if we have enough paths
    if (printPath < nPrint) {
// Time counter
for(int i = 0; i < nTimes; i++) {
    // BB dimension counter
    for( int d = 0; d < dim; d++) {
        // h_print is stored in display order
        h_print[d + printPath*dim +
            i*dim*nPrint] = h_GpuBatchPaths[p + batchSize*(d+i*dim)];
    }
}
printPath++;
    }
} // END print storage loop
} // END batching for loop

// Print Brownian Bridge Sample Paths
fprintf(fpout, "\nThe first %d Brownian Bridge sample paths "
        "of dimension %d:\n", nPrint, dim);
fprintf(fpout, "    \t ");
for(int p = 0; p < nPrint; p++) {
    int nspaces = (9*dim - 9)/2;
    for(int s = 0; s < nspaces; s++) fprintf(fpout, "-");
    fprintf(fpout, " Path%d ", p+1);
    for(int s = 0; s < nspaces; s++) fprintf(fpout, "-");
    fprintf(fpout, "\t ");
}
fprintf(fpout, "\n t_i\t");
for(int p = 0; p < nPrint; p++) {
    for(int d = 1; d<=dim; d++) fprintf(fpout,
        " dim%d ", d);
    fprintf(fpout, "\t");
}
for(int i = 0; i < nTimes; i++) {
    fprintf(fpout, "\n %d\t", i+1);
    for(int p = 0; p < nPrint; p++) {
        for(int d = 0; d < dim; d++) {
float val =
    h_print[i*dim*nPrint + p*dim + d];
if (val < 10 && val > -10)
    fprintf(fpout, "%.4f ", val);
else fprintf(fpout, "%.3f ", val);
        }
        fprintf(fpout, "\t");
    }
}
fprintf(fpout, "\n\n");

END:

// Release GPU and CPU memory
nag_gpu_sobol_cleanup(&comm);
nag_gpu_depthbb_cleanup(&comm);

```

```

if (d_GpuBatchPaths)
    nag_gpu_utilSafeCall( cudaFree(d_GpuBatchPaths) );

if (d_Z)
    nag_gpu_utilSafeCall( cudaFree(d_Z) );

if (h_GpuBatchPaths)
    free(h_GpuBatchPaths);

if (v)
    free(v);

if (h_print)
    free(h_print);

fflush(stdout);
fflush(stderr);

if (fpout!=stdout)
    fclose(fpout);

cudaThreadExit();
return 0;

}

```

7.2 Program Data

None.

7.3 Program Results

NAG GPU Brownian Bridge Example Program

The first 2 Brownian Bridge sample paths of dimension 3:

t_i	Path1			Path2		
	dim1	dim2	dim3	dim1	dim2	dim3
1	-0.4319	-0.5808	-0.2898	-0.8480	-0.4992	-0.4495
2	-2.7937	-2.9514	-2.1002	-1.0522	-0.6356	-1.0732
3	-3.3334	-2.8654	-2.3170	-2.0026	-1.5285	-1.2797
4	-1.4033	-1.4243	-0.8140	-0.5033	-0.2917	0.0213
5	-1.6910	-1.5270	-0.9744	-0.5964	-1.2117	-0.7582
6	-0.3886	-0.8542	0.5213	0.0870	-1.3075	-0.2395
7	1.1627	-0.1753	1.4755	0.5219	-1.1769	0.0842
8	2.9716	1.5756	3.9745	0.0489	-1.5908	0.0623
9	1.5040	0.7386	3.1853	0.0557	-1.3891	0.6072
10	2.2742	1.2883	2.7507	1.2465	-0.4790	2.3386
11	2.3454	1.5572	2.5536	0.9171	-0.5798	2.8336
12	1.6686	1.3529	1.8254	0.2452	-1.3949	1.4622
13	1.1460	0.3977	0.5882	1.1608	-1.2396	2.1604
14	-0.9138	-0.5229	-0.3301	1.4408	-0.8624	2.0340
15	-0.0744	0.2414	0.2336	1.1127	-1.7696	2.2644
16	-1.2231	-1.0024	-0.9659	2.2444	0.2407	3.8911
17	-1.0212	-0.3701	-1.0639	2.7101	0.6687	3.5573
18	-0.9228	-0.4152	-1.2231	3.3877	1.7935	4.0289
19	-1.8929	-1.5585	-2.6705	4.5592	2.2268	4.8547
20	-1.3380	-0.9218	-2.6492	4.9077	2.9009	5.7403
21	-1.2043	0.1941	-2.9113	5.2091	2.9556	6.3204
22	-0.7989	0.3286	-2.1112	5.2944	3.0768	6.5263
23	-1.5346	-0.2710	-2.1290	3.5146	1.7292	5.1454
24	-0.7368	0.3461	-1.1648	3.5580	1.7544	4.8543
25	-0.9513	0.4458	-0.4359	3.9576	1.9291	4.7411
26	-1.0345	-0.1895	-0.6200	4.0321	2.4064	4.2983

27	-0.7238	0.0604	0.4583	3.9061	3.8269	4.4716
28	-0.6229	-0.5132	0.0133	3.2623	3.1889	3.6857
29	0.6226	0.3169	0.5710	2.6916	2.7481	2.6390
30	1.5000	1.5000	1.5000	1.5000	1.5000	1.5000
