

# **NAG Numerical Routines for GPUs Manual, Mark 0.5**

## **Contents**

Copyright Statement

## **Chapters of the Library**

Linear Equations (LAPACK)

Random Number Generators

Error Handling

---



## **NAG Numerical Routines for GPUs**

© The Numerical Algorithms Group Limited, 2011

All rights reserved. Duplication of this Manual in printed form or by electronic means for the private and sole use of the software licensee is permitted provided that the individual copying the document is not

- selling or reselling the documentation;
- distributing the documentation to others, who have not licensed the software for professional use;
- using it for the purpose of critical review, publication in printed form or any electronic publication including the Internet without the prior written permission of the copyright owner.

The copyright owner gives no warranties and makes no representations about the contents of this Manual and specifically disclaims any implied warranties or merchantability or fitness for any purpose.

The copyright owner reserves the right to revise this Manual and to make changes from time to time in its contents without notifying any person of such revisions or changes.

Produced by NAG

Mark 0.5 released September 2011

NAG is a registered trademark of:

The Numerical Algorithms Group Limited  
The Numerical Algorithms Group Inc  
Nihon Numerical Algorithms Group KK

---



# NAG Numerical Routines for GPUs

## Table of Contents: Linear Equations (LAPACK)

### Chapter Introduction

#### Host-Callable Functions

naggpuLinAlgInitA  
naggpuDgetrfA  
naggpuDpotrfA  
naggpuLinAlgCleanupA

#### Serial CPU Functions

nagCPUDgetrfA  
nagCPUDpotrfA

#### List of Structures

NagGpuLinAlgComm

#### List of Enumerations

NagGpuMatrixUpLow

---



# NAG Numerical Routines for GPUs Chapter Introduction

## Linear Equations (LAPACK)

### Contents

<b>1</b>	<b>Scope of the Chapter</b> .....	2
<b>2</b>	<b>Background to the Problems</b> .....	2
2.1	Notation .....	2
2.2	Matrix Factorizations .....	2
2.3	Solution of Systems of Equations .....	2
<b>3</b>	<b>Recommendations on Choice and Use of Available Functions</b> .....	3
3.1	Available Functions .....	3
3.2	Matrix Storage Schemes .....	3
3.2.1	Conventional storage .....	3
3.3	Argument Conventions .....	3
3.3.1	Problem dimensions .....	3
<b>4</b>	<b>Functionality Index</b> .....	3
4.1	Host-Callable Linear Equation (LAPACK) Functions .....	3
4.2	Serial CPU Functions .....	4
<b>5</b>	<b>References</b> .....	4

## 1 Scope of the Chapter

This chapter provides functions for the solution of systems of simultaneous linear equations, and associated computations. It provides functions for matrix factorizations. Functions are provided for *real* data only.

The functions in this chapter handle only *dense* matrices (not matrices with more specialized structures, or general sparse matrices).

## 2 Background to the Problems

This section is only a brief introduction to the numerical solution of systems of linear equations. Consult a standard textbook, for example Golub and Van Loan (1996) for a more thorough discussion.

### 2.1 Notation

We use the standard notation for a system of simultaneous linear equations:

$$Ax = b \quad (1)$$

where  $A$  is the *coefficient matrix*,  $b$  is the *right-hand side*, and  $x$  is the *solution*.  $A$  is assumed to be a square matrix of order  $n$ .

If there are several right-hand sides, we write

$$AX = B \quad (2)$$

where the columns of  $B$  are the individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

### 2.2 Matrix Factorizations

If  $A$  is upper or lower triangular,  $Ax = b$  can be solved by a straightforward process of backward or forward substitution.

Otherwise, the solution is obtained after first factorizing  $A$ , as follows.

#### General matrices (*LU* factorization with partial pivoting)

$$A = PLU$$

where  $P$  is a permutation matrix,  $L$  is lower-triangular with diagonal elements equal to 1, and  $U$  is upper-triangular; the permutation matrix  $P$  (which represents row interchanges) is needed to ensure numerical stability.

#### Symmetric positive definite matrices (*Cholesky* factorization)

$$A = U^T U \quad \text{or} \quad A = LL^T$$

where  $U$  is upper triangular and  $L$  is lower triangular.

### 2.3 Solution of Systems of Equations

Given one of the above matrix factorizations, it is straightforward to compute a solution to  $Ax = b$  by solving two subproblems, as shown below, first for  $y$  and then for  $x$ . Each subproblem consists essentially of solving a triangular system of equations by forward or backward substitution; the permutation matrix  $P$  introduces only a little extra complication:

#### General matrices (*LU* factorization)

$$Ly = P^T b \\ Ux = y$$

#### Symmetric positive definite matrices (*Cholesky* factorization)

$$U^T y = b \quad \text{or} \quad Ly = b \\ Ux = y \quad \text{or} \quad L^T x = y$$

### 3 Recommendations on Choice and Use of Available Functions

#### 3.1 Available Functions

Functions are provided to perform Cholesky decomposition and *LU* factorization on dense, real-valued, double precision matrices:

naggpuDgetrfA computes the *LU* factorization of a general matrix.

naggpuDpotrfA computes the Cholesky decomposition of a symmetric positive definite matrix.

The initialization function naggpuLinAlgInitA must be called before the first call to any of the linear algebra functions. Once all calls to all linear algebra functions have completed, naggpuLinAlgCleanupA must be called to free allocated system resources.

#### 3.2 Matrix Storage Schemes

In this chapter matrices must be stored in the conventional way: packed storage schemes are not supported. In the examples below, \* indicates an array element which need not be set and is not referenced by the functions. The examples illustrate only the relevant part of the arrays; array arguments may of course have additional rows or columns.

##### 3.2.1 Conventional storage

Matrices must be stored in *column major order*: a matrix *A* is stored in a one-dimensional array **a**, with matrix element  $a_{i,j}$  stored in array element  $a((j - 1) \times pda + i - 1)$  where **pda** is the principal dimension of the array (i.e., the stride separating row elements of the matrix).

Functions which handle **symmetric** matrices allow for either the upper or lower triangle of the matrix (as specified by **uplow**) to be stored in the corresponding elements of the array; the remaining elements of the array need not be set.

For example, when  $n = 3$ :

uplow	Symmetric matrix <i>A</i>	Storage in array <b>a</b>
NAGGPUMATRIXUPLOW_UPPER	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}$	$a_{11} * * a_{12} a_{22} * a_{13} a_{23} a_{33}$
NAGGPUMATRIXUPLOW_LOWER	$\begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11} a_{21} a_{31} * a_{22} a_{32} * * a_{33}$

#### 3.3 Argument Conventions

##### 3.3.1 Problem dimensions

It is permissible for the problem dimensions (for example, **m** in naggpuDgetrfA, **n** in naggpuDpotrfA) to be passed as zero, in which case the computation is skipped. Negative dimensions are regarded as an error.

### 4 Functionality Index

#### 4.1 Host-Callable Linear Equation (LAPACK) Functions

Linear Equations

- Cholesky factorization of a real symmetric positive definite matrix ..... naggpuDpotrfA
- free system resources ..... naggpuLinAlgCleanupA
- initialise the linear equation functions ..... naggpuLinAlgInitA
- LU* factorization of a real *m* by *n* matrix ..... naggpuDgetrfA

## 4.2 Serial CPU Functions

### Linear Equations

Cholesky factorization of a real symmetric positive definite matrix .....	nagCPUDpotrfA
<i>LU</i> factorization of a real <i>m</i> by <i>n</i> matrix .....	nagCPUDgetrfA

## 5 References

Golub G H and Van Loan C F (1996) *Matrix Computations* (3rd Edition) Johns Hopkins University Press, Baltimore

---

# NAG Numerical Routines for GPUs Function Document

## naggpuLinAlgInitA

### 1 Purpose

**naggpuLinAlgInitA** initializes the GPU linear algebra (LAPACK) suite of functions. This function must be called before any call to the linear algebra functions (such as **naggpuDgetrfA**) and must ultimately be followed by a call to the cleanup function **naggpuLinAlgCleanupA** to release system resources.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuLinAlgInitA(NagGpuLinAlgComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

1: **comm** – NagGpuLinAlgComm \* *Communication Data*

NagGpuLinAlgComm is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the GPU linear algebra functions (such as **naggpuDgetrfA**). Once all calls to linear algebra functions have completed, **comm** must be passed to **naggpuLinAlgCleanupA** to free allocated system resources.

2: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

## 7 Example

There is no example program specifically for this function. For an example of how this function should be used, please see the example program for `naggpuDgetrfA`.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuDgetrfA

### 1 Purpose

**naggpuDgetrfA** computes the  $LU$  factorization of a real  $m$  by  $n$  matrix.

The initialization function `naggpuLinAlgInitA` must be called prior to the first call to **naggpuDgetrfA**. Once all calls to all linear algebra functions have been completed, the function `naggpuLinAlgCleanupA` must be called to free allocated system resources.

**Note:** the matrix must be stored in column major order.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuDgetrfA(int m, int n, double *d_a, int pda, int *ipiv,
                          NagGpuLinAlgComm *comm, NagGpuError *error)
```

### 3 Description

**naggpuDgetrfA** forms the  $LU$  factorization of a real  $m$  by  $n$  matrix  $A$  as  $A = PLU$ , where  $P$  is a permutation matrix,  $L$  is lower triangular with unit diagonal elements (lower trapezoidal if  $m > n$ ) and  $U$  is upper triangular (upper trapezoidal if  $m < n$ ). Usually  $A$  is square ( $m = n$ ), and both  $L$  and  $U$  are triangular.

**Note:** **naggpuDgetrfA** is currently not thread safe when used with multiple host threads and GPU devices.

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- |    |  |              |
|----|--|--------------|
| 1: | <b>m</b> – int   | <i>Input</i> |
|    | <i>On entry:</i> $m$ , the number of rows of the matrix $A$ .    |              |
|    | <i>Constraint:</i> $m \geq 0$ .                                  |              |
| 2: | <b>n</b> – int   | <i>Input</i> |
|    | <i>On entry:</i> $n$ , the number of columns of the matrix $A$ . |              |
|    | <i>Constraint:</i> $n \geq 0$ .                                  |              |

3: **d\_a**[*d*] – double \* Input/Output

This buffer must reside in the GPU memory space.

**Note:** the dimension *d* of the array **d\_a** must satisfy  $d \geq \max(1, n \times \text{pda})$ .

*On entry:* the *m* by *n* matrix *A*. The (*i*, *j*)th element  $a_{i,j}$  of the matrix *A* is stored in  $\text{d\_a}[(j-1) \times \text{pda} + i - 1]$ .

*On exit:* the factors *L* and *U* from the factorization  $A = PLU$ ; the unit diagonal elements of *L* are not stored. This means that

$u_{i,j}$  is stored in  $\text{d\_a}[(j-1) \times \text{pda} + i - 1]$  for  $j = 1, 2, \dots, n$  and  $i = 1, 2, \dots, \min(j, m)$

$l_{i,j}$  is stored in  $\text{d\_a}[(j-1) \times \text{pda} + i - 1]$  for  $i = 2, 3, \dots, m$  and  $j = 1, 2, \dots, \min(i-1, n)$

where  $u_{i,j}$  denotes the (*i*, *j*)th element of the matrix *U* and  $l_{i,j}$  denotes the (*i*, *j*)th element of the matrix *L*.

4: **pda** – int Input

*On entry:* the stride separating row elements of the matrix *A* in the array **d\_a**.

*Constraint:*  $\text{pda} \geq \max(1, m)$ .

5: **ipiv**[**min(m, n)**] – int \* Output

*On exit:* the pivot indices that define the permutation matrix. At the *i*th step, if  $\text{ipiv}[i-1] > i$  then row *i* of the matrix *A* was interchanged with row  $\text{ipiv}[i-1]$ , for  $i = 1, 2, \dots, \min(m, n)$ .  $\text{ipiv}[i-1] \leq i$  indicates that at the *i*th step, a row interchange was not required.

6: **comm** – NagGpuLinAlgComm \* Communication Data

NagGpuLinAlgComm is a structure which holds state and communication information and must not be modified in any way. Once all calls to linear algebra functions have completed, **comm** must be passed to naggpuLinAlgCleanupA to free allocated system resources.

7: **error** – NagGpuError \* Error Reporting

**Note:** on exit,  $\text{error} \rightarrow \text{subCodes}[0]$  will contain the LAPACK `info` parameter which is traditionally used to indicate errors or warnings.

This parameter contains error information and should not be modified directly. Errors are indicated through the value of  $\text{error} \rightarrow \text{code}$  which should be inspected after each call to this function. If  $\text{error} \rightarrow \text{code} = 0$  then no error occurred. If  $\text{error} \rightarrow \text{code} \neq 0$  then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

$\text{error} \rightarrow \text{code} = 1$

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

$\text{error} \rightarrow \text{code} = 2$

*During execution:* a CUDA runtime error was detected.

$\text{error} \rightarrow \text{code} = 3$

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

$\text{error} \rightarrow \text{code} = 100$

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $m < 0$

error → code = 111

*On entry:*  $n < 0$

error → code = 112

*On entry:* **d\_a** is NULL.

error → code = 113

*On entry:* **pda** does not satisfy the constraint listed above.

error → code = 114

*On entry:* **ipiv** is NULL.

error → code = 115

*On exit:*  $U(i, i) = 0$  where  $i = \text{error} \rightarrow \text{subCodes}[0]$ . The factorization has been completed, but the factor  $U$  is exactly singular, and division by zero will occur if it is used to solve a system of equations.

## 7 Example

This example computes the  $LU$  factorization of the matrix  $A$ , where

$$A = \begin{pmatrix} 1.80 & 2.88 & 2.05 & -0.89 \\ 5.25 & -2.95 & -0.95 & -3.80 \\ 1.58 & -2.69 & -2.90 & -1.04 \\ -1.11 & -0.66 & -0.59 & 0.80 \end{pmatrix}.$$

### 7.1 Program Text

```

/*
 * Example Program: naggpuDgetrfA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 */

#include <stdio.h>
#include
using namespace std;

#include <nag_gpu.h>

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for the matrix
    double *h_A = 0;
    // device (GPU) storage for the matrix
    double *d_A = 0;
    // host (CPU) storage for ipiv array
    int * ipiv = 0;

```

```

const int m = 4;
const int n = 4;
const int lda = n;

// NAG GPU structures
NagGpuLinAlgComm comm;
NagGpuError error;

cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpuDgetrfA";
cout << endl << endl;

// Allocate CPU and GPU memory
h_A = new double[m*n];
ipiv = new int[m];
cuError = cudaMalloc((void **)&d_A, sizeof(double)*m*n);
checkCudaError(cuError);

// Populate the matrix:
// NOTE: Column major order
h_A[0]=1.80; h_A[4]=2.88; h_A[8]=2.05; h_A[12]=-0.89;
h_A[1]=5.25; h_A[5]=-2.95; h_A[9]=-0.95; h_A[13]=-3.80;
h_A[2]=1.58; h_A[6]=-2.69; h_A[10]=-2.90; h_A[14]=-1.04;
h_A[3]=-1.11; h_A[7]=-0.66; h_A[11]=-0.59; h_A[15]=0.80;

// Copy data to GPU
cuError = cudaMemcpy(d_A, h_A, sizeof(double)*m*n,
                    cudaMemcpyHostToDevice);
checkCudaError(cuError);

// Initialise LAPACK
cout << "Initialising Linear Algebra routines ..." << endl << endl;
naggpuLinAlgInitA(&comm, &error);
checkNagError(&error);

// Do factorisation
naggpuDgetrfA(m, n, d_A, lda, ipiv, &comm, &error);
checkNagError(&error);

// Copy back values from the GPU for printing
cuError = cudaMemcpy(h_A, d_A, sizeof(double)*m*n,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print output
cout << "The LU factorisation: h_A = " << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(4);
for(int row = 0; row < m; row++)
{
    for(int col = 0; col < n; col++)
    {
        printf("    %.4f", h_A[row + col*lda]);
    }
    cout << endl;
}
cout << endl;
cout << "Pivoting information: ipiv = " << endl;
for(int row=0; row < m; row++)
{
    cout << "\t" << ipiv[row];
}
cout << endl << endl;

// Call cleanup for the NAG routine
naggpuLinAlgCleanupA(&comm, &error);

```

```
    checkNagError(&error);

    // Free CPU and GPU memory
    delete[] h_A;
    delete[] ipiv;
    if (d_A)
    {
        cuError = cudaFree(d_A);
        checkCudaError(cuError);
    }

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}
```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuDgetrfA

Initialising Linear Algebra routines ...

The LU factorisation: h\_A =

5.2500	-2.9500	-0.9500	-3.8000
0.3429	3.8914	2.3757	0.4129
0.3010	-0.4631	-1.5139	0.2948
-0.2114	-0.3299	0.0047	0.1314

Pivoting information: ipiv =  
2 2 3 4



# NAG Numerical Routines for GPUs Function Document

## naggpuDpotrfA

### 1 Purpose

**naggpuDpotrfA** computes the Cholesky factorization of a real symmetric positive definite matrix.

The initialization function `naggpuLinAlgInitA` must be called prior to the first call to **naggpuDpotrfA**. Once all calls to all linear algebra functions have been completed, the function `naggpuLinAlgCleanupA` must be called to free allocated system resources.

**Note:** the matrix must be stored in column major order.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuDpotrfA(NagGpuMatrixUpLow uplow, int n, double *d_a, int pda,
                          NagGpuLinAlgComm *comm, NagGpuError *error)
```

### 3 Description

**naggpuDpotrfA** forms the Cholesky factorization of a real symmetric positive definite matrix  $A$  either as  $A = U^T U$  if `uplow = NAGGPUMATRIXUPLOW_UPPER` or  $A = LL^T$  if `uplow = NAGGPUMATRIXUPLOW_LOWER`, where  $U$  is an upper triangular matrix and  $L$  is lower triangular.

**Note:** **naggpuDpotrfA** is currently not thread safe when used with multiple host threads and GPU devices.

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

1: **uplow** – NagGpuMatrixUpLow *Input*

*On entry:* specifies whether the upper or lower triangular part of  $A$  is stored and how  $A$  is to be factorized.

`uplow = NAGGPUMATRIXUPLOW_UPPER`

The upper triangular part of  $A$  is stored and  $A$  is factorized as  $U^T U$ , where  $U$  is upper triangular.

uplow = NAGGPUMATRIXUPLOW\_LOWER

The lower triangular part of  $A$  is stored and  $A$  is factorized as  $LL^T$ , where  $L$  is lower triangular.

*Constraint:* uplow = NAGGPUMATRIXUPLOW\_UPPER or  
uplow = NAGGPUMATRIXUPLOW\_LOWER.

2: **n** – int *Input*

*On entry:*  $n$ , the order of the matrix  $A$ .

*Constraint:*  $n \geq 0$ .

3: **d\_a**[ $d$ ] – double \* *Input/Output*

This buffer must reside in the GPU memory space.

**Note:** the dimension  $d$  of the array **d\_a** must satisfy  $d \geq \max(1, n \times \text{pda})$ .

*On entry:* the  $n$  by  $n$  symmetric positive definite matrix  $A$ . Let  $a_{i,j}$  denote the  $(i, j)$ th element of the matrix  $A$ :

if uplow = NAGGPUMATRIXUPLOW\_UPPER, the upper triangular part  $a_{i,j}$  for  $j = 1, 2, \dots, n$  and  $i = 1, 2, \dots, j$  must be stored in  $\text{d\_a}[(j-1) \times \text{pda} + i - 1]$  and the elements of the array below the diagonal are not referenced.

if uplow = NAGGPUMATRIXUPLOW\_LOWER, the lower triangular part  $a_{i,j}$  for  $i = 1, 2, \dots, n$  and  $j = 1, 2, \dots, i$  must be stored in  $\text{d\_a}[(j-1) \times \text{pda} + i - 1]$  and the elements of the array above the diagonal are not referenced.

*On exit:* the factor  $U$  or  $L$  from the Cholesky factorization  $A = U^T U$  or  $A = LL^T$ . Let  $u_{i,j}$  denote the  $(i, j)$ th element of the matrix  $U$  and let  $l_{i,j}$  denote the  $(i, j)$ th element of the matrix  $L$ :

if uplow = NAGGPUMATRIXUPLOW\_UPPER, the element  $u_{i,j}$  for  $j = 1, 2, \dots, n$  and  $i = 1, 2, \dots, j$  is stored in  $\text{d\_a}[(j-1) \times \text{pda} + i - 1]$

if uplow = NAGGPUMATRIXUPLOW\_LOWER, the element  $l_{i,j}$  for  $i = 1, 2, \dots, n$  and  $j = 1, 2, \dots, i$  is stored in  $\text{d\_a}[(j-1) \times \text{pda} + i - 1]$

4: **pda** – int *Input*

*On entry:* the stride separating row elements of the matrix  $A$  in the array **d\_a**.

*Constraint:*  $\text{pda} \geq \max(1, n)$ .

5: **comm** – NagGpuLinAlgComm \* *Communication Data*

NagGpuLinAlgComm is a structure which holds state and communication information and must not be modified in any way. Once all calls to linear algebra functions have completed, **comm** must be passed to naggpuLinAlgCleanupA to free allocated system resources.

6: **error** – NagGpuError \* *Error Reporting*

**Note:** on exit,  $\text{error} \rightarrow \text{subCodes}[0]$  will contain the LAPACK `info` parameter which is traditionally used to indicate errors or warnings.

This parameter contains error information and should not be modified directly. Errors are indicated through the value of  $\text{error} \rightarrow \text{code}$  which should be inspected after each call to this function. If  $\text{error} \rightarrow \text{code} = 0$  then no error occurred. If  $\text{error} \rightarrow \text{code} \neq 0$  then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:* **uplow** does not satisfy the constraint listed above.

error → code = 111

*On entry:*  $n < 0$

error → code = 112

*On entry:* **d\_a** is NULL.

error → code = 113

*On entry:* **pda** does not satisfy the constraint listed above.

error → code = 114

*On exit:* the leading minor of order  $i$  where  $i = \text{error} \rightarrow \text{subCodes}[0]$  is not positive definite, and the factorization could not be completed. Hence the matrix  $A$  itself is not positive definite. This may indicate an error in forming the matrix  $A$ .

## 7 Example

This example computes the Cholesky factorization of the matrix  $A$ , where

$$A = \begin{pmatrix} 4.16 & -3.12 & 0.56 & -0.10 \\ -3.12 & 5.03 & -0.83 & 1.18 \\ 0.56 & -0.83 & 0.76 & 0.34 \\ -0.10 & 1.18 & 0.34 & 1.18 \end{pmatrix}.$$

### 7.1 Program Text

```

/*
 * Example Program: naggpuDpotrfA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include <stdio.h>
#include
using namespace std;

```

```

#include <nag_gpu.h>

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for the matrix
    double *h_A = 0;
    // device (GPU) storage for the matrix
    double *d_A = 0;
    // host (CPU) storage for ipiv array
    int * ipiv = 0;

    const int n = 4;
    const int lda = n;

    // NAG GPU structures
    NagGpuLinAlgComm comm;
    NagGpuError error;

    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpuDpotrfA";
    cout << endl << endl;

    // Allocate CPU and GPU memory
    h_A = new double[n*n];
    cuError = cudaMalloc((void **)&d_A, sizeof(double)*n*n);
    checkCudaError(cuError);

    // Populate the matrix:
    // NOTE: Column major order
    h_A[0]=4.16; h_A[4]=-3.12; h_A[8]=0.56; h_A[12]=-0.10;
    h_A[1]=-3.12; h_A[5]=5.03; h_A[9]=-0.83; h_A[13]=1.18;
    h_A[2]=0.56; h_A[6]=-0.83; h_A[10]=0.76; h_A[14]=0.34;
    h_A[3]=-0.10; h_A[7]=1.18; h_A[11]=0.34; h_A[15]=1.18;

    // Copy data to GPU
    cuError = cudaMemcpy(d_A, h_A, sizeof(double)*n*n,
        cudaMemcpyHostToDevice);
    checkCudaError(cuError);

    // Initialise LAPACK
    cout << "Initialising Linear Algebra routines ..." << endl << endl;
    naggpuLinAlgInitA(&comm, &error);
    checkNagError(&error);

    // Do factorisation
    naggpuDpotrfA(NAGGPUMATRIXUPLOW_LOWER, n, d_A, lda, &comm, &error);
    checkNagError(&error);

    // Copy back values from the GPU for printing
    cuError = cudaMemcpy(h_A, d_A, sizeof(double)*n*n,
        cudaMemcpyDeviceToHost);
    checkCudaError(cuError);

    // Print output
    cout << "The Cholesky factorisation: h_A = " << endl;
    cout.setf(ios::fixed,ios::floatfield);
    cout.precision(4);
    for(int row = 0; row < n; row++)
    {
        for(int col = 0; col < n; col++)
        {

```

```

        if(col <= row) printf("    % .4f", h_A[row + col*lda]);
        else           printf("          ");
    }
    cout << endl;
}
cout << endl << endl;

// Call cleanup for the NAG routine
naggpuLinAlgCleanupA(&comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_A;
if (d_A)
{
    cuError = cudaFree(d_A);
    checkCudaError(cuError);
}

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuDpotrfA

Initialising Linear Algebra routines ...

The Cholesky factorisation: h\_A =

2.0396				
-1.5297	1.6401			
0.2746	-0.2500	0.7887		
-0.0490	0.6737	0.6617	0.5347	



# NAG Numerical Routines for GPUs Function Document

## naggpuLinAlgCleanupA

### 1 Purpose

**naggpuLinAlgCleanupA** frees system resources that were allocated by a previous call to **naggpuLinAlgInitA**.

### 2 Specification

```
#include <nag_gpu.h>
extern "C"
cudaError_t naggpuLinAlgCleanupA(NagGpuLinAlgComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- 1: **comm** – NagGpuLinAlgComm \* *Communication Data*  
*On entry:* the pointer that was passed to a previous call to **naggpuLinAlgInitA**.
- 2: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

### 6 Error Indicators and Warnings

`error → code = 1`

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error → code = 2`

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

## **7 Example**

There is no example program specifically for this function. For an example of how this function should be used, please see the example program for naggpuDgetrfA.

---

# NAG Numerical Routines for GPUs Function Document

## nagCPUDgetrfA

### 1 Purpose

**nagCPUDgetrfA** computes the  $LU$  factorization of a real  $m$  by  $n$  matrix.

**Note:** the matrix must be stored in column major order.

### 2 Specification

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

extern "C"
void nagCPUDgetrfA(int m, int n, double *a, int pda, int *ipiv, NagGpuError *error)
```

### 3 Description

**nagCPUDgetrfA** forms the  $LU$  factorization of a real  $m$  by  $n$  matrix  $A$  as  $A = PLU$ , where  $P$  is a permutation matrix,  $L$  is lower triangular with unit diagonal elements (lower trapezoidal if  $m > n$ ) and  $U$  is upper triangular (upper trapezoidal if  $m < n$ ). Usually  $A$  is square ( $m = n$ ), and both  $L$  and  $U$  are triangular.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

1: **m** – int *Input*

*On entry:*  $m$ , the number of rows of the matrix  $A$ .

*Constraint:*  $m \geq 0$ .

2: **n** – int *Input*

*On entry:*  $n$ , the number of columns of the matrix  $A$ .

*Constraint:*  $n \geq 0$ .

3: **a**[ $d$ ] – double \* *Input/Output*

**Note:** the dimension  $d$  of the array **a** must satisfy  $d \geq \max(1, n \times \text{pda})$ .

*On entry:* the  $m$  by  $n$  matrix  $A$ . The  $(i, j)$ th element  $a_{i,j}$  of the matrix  $A$  is stored in  $\text{a}[(j-1) \times \text{pda} + i - 1]$ .

*On exit:* the factors  $L$  and  $U$  from the factorization  $A = PLU$ ; the unit diagonal elements of  $L$  are not stored. This means that

$u_{i,j}$  is stored in  $\text{a}[(j-1) \times \text{pda} + i - 1]$  for  $j = 1, 2, \dots, n$  and  $i = 1, 2, \dots, \min(j, m)$

$l_{i,j}$  is stored in  $\text{a}[(j-1) \times \text{pda} + i - 1]$  for  $i = 2, 3, \dots, m$  and  $j = 1, 2, \dots, \min(i-1, n)$

where  $u_{i,j}$  denotes the  $(i,j)$ th element of the matrix  $U$  and  $l_{i,j}$  denotes the  $(i,j)$ th element of the matrix  $L$ .

4: **pda** – int *Input*

*On entry:* the stride separating row elements of the matrix  $A$  in the array **a**.

*Constraint:*  $pda \geq \max(1, m)$ .

5: **ipiv**[**min**(**m**, **n**)] – int \* *Output*

*On exit:* the pivot indices that define the permutation matrix. At the  $i$ th step, if  $ipiv[i - 1] > i$  then row  $i$  of the matrix  $A$  was interchanged with row  $ipiv[i - 1]$ , for  $i = 1, 2, \dots, \min(m, n)$ .  $ipiv[i - 1] \leq i$  indicates that at the  $i$ th step, a row interchange was not required.

6: **error** – NagGpuError \* *Error Reporting*

**Note:** on exit,  $error \rightarrow subCodes[0]$  will contain the LAPACK `info` parameter which is traditionally used to indicate errors or warnings.

This parameter contains error information and should not be modified directly. Errors are indicated through the value of  $error \rightarrow code$  which should be inspected after each call to this function. If  $error \rightarrow code = 0$  then no error occurred. If  $error \rightarrow code \neq 0$  then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

$error \rightarrow code = 110$

*On entry:*  $m < 0$

$error \rightarrow code = 111$

*On entry:*  $n < 0$

$error \rightarrow code = 112$

*On entry:* **a** is NULL.

$error \rightarrow code = 113$

*On entry:* **pda** does not satisfy the constraint listed above.

$error \rightarrow code = 114$

*On entry:* **ipiv** is NULL.

$error \rightarrow code = 115$

*On exit:*  $U(i, i) = 0$  where  $i = error \rightarrow subCodes[0]$ . The factorization has been completed, but the factor  $U$  is exactly singular, and division by zero will occur if it is used to solve a system of equations.

## 7 Example

None.

# NAG Numerical Routines for GPUs Function Document

## nagCPUDpotrfA

### 1 Purpose

**nagCPUDpotrfA** computes the Cholesky factorization of a real symmetric positive definite matrix.

**Note:** the matrix must be stored in column major order.

### 2 Specification

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

extern "C"
void nagCPUDpotrfA(NagGpuMatrixUpLow uplow, int n, double *a, int pda,
                  NagGpuError *error)
```

### 3 Description

**nagCPUDpotrfA** forms the Cholesky factorization of a real symmetric positive definite matrix  $A$  either as  $A = U^T U$  if `uplow = NAGGPUMATRIXUPLOW_UPPER` or  $A = LL^T$  if `uplow = NAGGPUMATRIXUPLOW_LOWER`, where  $U$  is an upper triangular matrix and  $L$  is lower triangular.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

- 1: **uplow** – NagGpuMatrixUpLow *Input*  
*On entry:* specifies whether the upper or lower triangular part of  $A$  is stored and how  $A$  is to be factorized.
- `uplow = NAGGPUMATRIXUPLOW_UPPER`  
 The upper triangular part of  $A$  is stored and  $A$  is factorized as  $U^T U$ , where  $U$  is upper triangular.
- `uplow = NAGGPUMATRIXUPLOW_LOWER`  
 The lower triangular part of  $A$  is stored and  $A$  is factorized as  $LL^T$ , where  $L$  is lower triangular.
- Constraint:* `uplow = NAGGPUMATRIXUPLOW_UPPER` or `uplow = NAGGPUMATRIXUPLOW_LOWER`.
- 2: **n** – int *Input*  
*On entry:*  $n$ , the order of the matrix  $A$ .  
*Constraint:*  $n \geq 0$ .

3: **a**[*d*] – double \* Input/Output

**Note:** the dimension *d* of the array **a** must satisfy  $d \geq \max(1, n \times \text{pda})$ .

*On entry:* the *n* by *n* symmetric positive definite matrix *A*. Let  $a_{i,j}$  denote the (*i*, *j*)th element of the matrix *A*:

if `uplow = NAGGPUMATRIXUPLOW_UPPER`, the upper triangular part  $a_{i,j}$  for  $j = 1, 2, \dots, n$  and  $i = 1, 2, \dots, j$  must be stored in `a[(j - 1) × pda + i - 1]` and the elements of the array below the diagonal are not referenced.

if `uplow = NAGGPUMATRIXUPLOW_LOWER`, the lower triangular part  $a_{i,j}$  for  $i = 1, 2, \dots, n$  and  $j = 1, 2, \dots, i$  must be stored in `a[(j - 1) × pda + i - 1]` and the elements of the array above the diagonal are not referenced.

*On exit:* the factor *U* or *L* from the Cholesky factorization  $A = U^T U$  or  $A = LL^T$ . Let  $u_{i,j}$  denote the (*i*, *j*)th element of the matrix *U* and let  $l_{i,j}$  denote the (*i*, *j*)th element of the matrix *L*:

if `uplow = NAGGPUMATRIXUPLOW_UPPER`, the element  $u_{i,j}$  for  $j = 1, 2, \dots, n$  and  $i = 1, 2, \dots, j$  is stored in `a[(j - 1) × pda + i - 1]`

if `uplow = NAGGPUMATRIXUPLOW_LOWER`, the element  $l_{i,j}$  for  $i = 1, 2, \dots, n$  and  $j = 1, 2, \dots, i$  is stored in `a[(j - 1) × pda + i - 1]`

4: **pda** – int Input

*On entry:* the stride separating row elements of the matrix *A* in the array **a**.

*Constraint:*  $\text{pda} \geq \max(1, n)$ .

5: **error** – NagGpuError \* Error Reporting

**Note:** on exit, `error → subCodes[0]` will contain the LAPACK `info` parameter which is traditionally used to indicate errors or warnings.

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

`error → code = 110`

*On entry:* **uplow** does not satisfy the constraint listed above.

`error → code = 111`

*On entry:*  $n < 0$

`error → code = 112`

*On entry:* **a** is NULL.

`error → code = 113`

*On entry:* **pda** does not satisfy the constraint listed above.

`error → code = 114`

*On exit:* the leading minor of order *i* where  $i = \text{error} \rightarrow \text{subCodes}[0]$  is not positive definite, and the factorization could not be completed. Hence the matrix *A* itself is not positive definite. This may indicate an error in forming the matrix *A*.

## **7 Example**

None.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuLinAlgComm

### 1 Purpose

**NagGpuLinAlgComm** is used by the library for communication between the GPU linear algebra (LAPACK) functions. It is for internal library use and should not be modified by the user in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuLinAlgComm {
    NagGpuTuneOrigin param1;
    void *param2;
    int param3;
    void *param4;
}
```

### 3 Description

### 4 References

None.

### 5 Members

The full structure definition is provided so that the library can be called from languages other than C/C++. The members of this structure not documented below are private to the library and must not be modified in any way.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuMatrixUpLow

### 1 Purpose

An enumeration to identify the upper or lower triangular parts of a matrix.

### 2 Specification

```
#include <nag_gpu.h>
enum NagGpuMatrixUpLow {
    NAGGPUMATRIXUPLOW_UPPER = 700,
    NAGGPUMATRIXUPLOW_LOWER
}
```

### 3 Description

### 4 References

None.

### 5 Symbols

- 1: **NAGGPUMATRIXUPLOW\_UPPER**  
Identifies the upper triangular part of a matrix
  - 2: **NAGGPUMATRIXUPLOW\_LOWER**  
Identifies the lower triangular part of a matrix
-



# NAG Numerical Routines for GPUs

## Table of Contents: Random Number Generators

### Chapter Introduction

#### Host-Callable Generator Functions

naggpuRandInitA  
naggpuRandExpA  
naggpuRandGammaA  
naggpuRandNormalA  
naggpuRandUniformA  
naggpuRandCleanupA

naggpuQuasiRandInitA  
naggpuQuasiRandExpA  
naggpuQuasiRandNormalA  
naggpuQuasiRandUniformA  
naggpuQuasiRandCleanupA

naggpuDepthBBInitA  
naggpuDepthBBA  
naggpuDepthBBIncInitA  
naggpuDepthBBIncA  
naggpuDepthBBCleanupA

#### Inline Device Function Generators

naggpuMrg32k3aDeviceInitA  
naggpudevMrg32k3aInitA  
naggpudevMrg32k3aExpA  
naggpudevMrg32k3aGammaA  
naggpudevMrg32k3aGammaSetParamsA  
naggpudevMrg32k3aNormalA  
naggpudevMrg32k3aUniformA  
naggpuMrg32k3aDeviceCleanupA

naggpuSobolDeviceInitA  
naggpudevSobolInitA  
naggpudevSobolExpA  
naggpudevSobolNormalA  
naggpudevSobolUniformA  
naggpuSobolDeviceCleanupA

#### Serial CPU Functions

nagCPURandInitA  
nagCPURandExpA  
nagCPURandGammaA  
nagCPURandNormalA  
nagCPURandUniformA  
nagCPURandCleanupA

nagCPUQuasiRandInitA  
nagCPUQuasiRandExpA  
nagCPUQuasiRandNormalA

nagCPUQuasiRandUniformA  
nagCPUQuasiRandCleanupA

nagCPUDepthBBInitA  
nagCPUDepthBBA  
nagCPUDepthBBIncInitA  
nagCPUDepthBBIncA  
nagCPUDepthBBCleanupA

### **List of Structures**

NagGpuDepthBBComm  
NagGpuMrg32k3aDeviceComm  
NagGpuQuasiRandComm  
NagGpuQuasiRandTune  
NagGpuRandComm  
NagGpuRandTune  
NagGpuSobolDeviceComm  
  
NagCPUDepthBBComm  
NagCPUQuasiRandComm  
NagCPURandComm

### **List of Enumerations**

NagGpuRandGen  
NagGpuQuasiGen  
NagGpuQuasiOrient  
NagGpuRandOrder  
NagGpuScramTypes  
NagGpuTuneOrigin

---

# NAG Numerical Routines for GPUs Chapter Introduction

## Random Number Generators

### Contents

<b>1</b>	<b>Scope of the Chapter</b> .....	2
<b>2</b>	<b>Background to the Problems</b> .....	2
2.1	Pseudorandom Numbers .....	2
2.1.1	MRG32k3a Generator .....	2
2.1.1.1	Implementation Changes Since Release 0.3 .....	3
2.1.2	Mersenne Twister MT19937 Generator .....	3
2.2	Quasi-random Numbers .....	4
2.3	Brownian Bridge .....	4
<b>3</b>	<b>Recommendations on Choice and Use of Available Functions</b> .....	4
3.1	Host-Callable Generator Functions .....	4
3.2	Inline Device Function Generators .....	5
<b>4</b>	<b>Functionality Index</b> .....	6
4.1	Host-Callable Generator Functions .....	6
4.2	Inline Device Function Generators .....	6
4.3	Serial CPU Functions .....	7
<b>5</b>	<b>References</b> .....	7

## 1 Scope of the Chapter

This chapter contains functions for the generation of sequences of pseudorandom and quasi-random numbers from various distributions.

## 2 Background to the Problems

### 2.1 Pseudorandom Numbers

A pseudorandom sequence is a sequence of numbers generated in some systematic way such that they are independent and statistically indistinguishable from a truly random sequence. A pseudorandom number generator (PRNG) is a mathematical algorithm that, given an initial state, produces a sequence of pseudorandom numbers. A PRNG has several advantages over a true random number generator in that the generated sequence is repeatable, has known mathematical properties and can be implemented without needing any specialist hardware. Many books on statistics and computer science have good introductions to PRNGs, for example Knuth (1981) or Banks (1998).

Suppose for a given seed, the PRNG produces the sequence of random numbers  $X_0, X_1, X_2, X_3, \dots$ . The library allows users to advance, or *skip ahead*, the seed by an amount

$$s = a_1 2^{b_1} + a_2 2^{b_2} + c \quad (1)$$

so that the generator will now produce the sequence of random numbers  $X_s, X_{s+1}, X_{s+2}, X_{s+3}, \dots$ . In applications where  $T$  separate copies of a generator are to operate independently, this technique is often used in one of two ways:

1. To split a sample of  $N$  consecutive values into  $T$  adjacent, non-overlapping blocks where each block typically has length  $N/T$ . This can only be done if  $N$  is known, i.e. if it is known in advance how many values are to be generated, and in this case the  $i$ -th generator simply skips the seed ahead by  $(i - 1)N/T$  steps for  $1 \leq i \leq T$ .
2. To produce  $T$  *independent substreams*. This is done if it is not known in advance how many random values are needed, however now the task of determining where each generator should start generating is non-trivial. Merely ensuring that the generators (or ‘substreams’) do not overlap (e.g., by choosing a very large  $s$ ) does not ensure statistical independence between substreams. In general, the task of determining good values of  $a_1, b_1, a_2, b_2$  and  $c$  for use in these computations is not a simple matter, and will depend heavily on the particular generator.

**Note:** when using multiple streams and substreams to create independent generators, care should be taken that the skip aheads  $s$  do not exceed the period of the generator and result in two streams inadvertently starting at the same point (or close enough to overlap). Therefore it is recommended that  $s$  never exceed the period of the generator being used. The periods of the generators are listed below.

Independent generators will most likely only be used by applications which use multiple GPUs simultaneously. In this case, each generator will have its own communication structure (i.e. NagGpuRandComm) which encapsulates all the information the generator needs to function. An array of communication structures with judiciously chosen skip aheads represents an array of independent generators. Note that the initialization function must access the same GPU device context on which the generate call will be made. Please consult the CUDA documentation for details on how to achieve this and how to access multiple GPUs simultaneously from a single application.

For the most common task of generating a block of pseudorandom numbers on a single GPU, users will typically only have a single GPU generator (i.e. only one communication structure) and the skip ahead  $s$  above can be set to zero.

#### 2.1.1 MRG32k3a Generator

The Multiple Recursive Generator MRG32k3a is described in L’Ecuyer (1999), where L’Ecuyer gives a very efficient serial implementation. The period of the generator is approximately  $2^{191}$ . Parallelization relies on the fact that the two recurrences defining this generator are of the same form, and can be represented as simple matrices leading to an efficient means of computing skip ahead points. The GPU implementation is fairly straightforward. The matrices defining the skip ahead can be pre-computed on the

host and copied to device memory. Each thread uses the skip ahead algorithm to compute the initial point and then uses the standard algorithm to compute a contiguous block of pseudorandom variates.

In a later paper L'Ecuyer *et al.* (2002), the authors consider the problem of partitioning the generator's period into independent substreams. They consider various values of  $b_1$  and  $b_2$  in (1) above, and use a spectral test to search for values where the resulting streams of numbers have good statistical properties. They conclude that good statistical properties are obtained when  $b_1$  or  $b_2$  are equal to 76 or 127. The  $i$ th independent *stream* of the generator is selected by setting  $s = i2^{127}$  for  $i = 0, 1, \dots$ , and within the  $i$ th *stream* the  $j$ th independent *substream* is selected by setting  $s = i2^{127} + j2^{76}$  for  $j = 0, 1, \dots$ . The reader is referred to L'Ecuyer *et al.* (2002) for further details. The library has been optimized for values of  $b_1$  or  $b_2$  equal to 76 or 127: skipping ahead with these parameters is significantly faster than skipping ahead with different values of  $b_1$  and  $b_2$ .

### 2.1.1.1 Implementation Changes Since Release 0.3

The MRG32k3a generator interface has changed significantly since release 0.3. The main changes are as follows:

1. The function names and argument lists have changed.
2. The seed has been unified into a single array. Previously, the generator was initialized via a call to `nag_gpu_mrg32k3a_init(v1, v2, offset)` where `v1` and `v2` were arrays of three unsigned integers. Now (see `naggpuRandInitA`) there is a single array `seed` which corresponds to `v1` and `v2` as follows: `seed[i] = v1[i]` and `seed[i + 3] = v2[i]` for  $i = 0, 1, 2$ .
3. Previously, the input seed was advanced by one step before the first point was generated. Now the input seed is used as-is to generate the first point.

To match the CPU generators in release 0.3, call `nagCPURandInitA` with

1. `genid = NAGGPURANDGEN_MRG32K3A`
2. `a1 = b1 = a2 = b2 = 0`
3. `c = offset + 1`
4. `seed` initialized from `v1, v2` as shown above

Then call the generator functions (such as `nagCPURandUniformA`) as usual, passing in the desired distribution parameters.

To match the GPU generators in release 0.3, call `naggpuRandInitA` in the same way as just described for `nagCPURandInitA`, and then call one of the generator functions (e.g. `naggpuRandUniformA`) and

1. `set order = NAGGPURANDORDER_OPTIMAL`
2. create a `NagGpuRandTune` structure `compat`, set `compat.mrgOptATHdsPerBlk = nt` and `compat.mrgOptAPtsPerThd = np`, and set `tune = compat`
3. `set n = nb * nt * np`

where `np`, `nt` and `nb` were the generator parameters release 0.3 of the library.

**Note:** if the GPU generators in release 0.3 were forced to match the serial ordering (i.e. by specifying `nt = 1`), then call `naggpuRandInitA` as described above, but call the generator function (e.g. `naggpuRandUniformA`) with `order = NAGGPURANDORDER_CONSISTENT`, `n = nb * np` and `tune = NULL`.

### 2.1.2 Mersenne Twister MT19937 Generator

The Mersenne Twister is described in Matsumoto and Nishimura (1998). It is a twisted generalized feedback shift register generator with very good multidimensional statistical properties and a period approximately equal to  $2^{19937}$ . The library implementation follows the implementation given in Matsumoto and Nishimura (1998). Parallelization uses the polynomial methods outlined in Haramoto *et al.* (2008).

The library allows skip aheads of the form (1) above for the Mersenne Twister, so that independent streams and substreams can be constructed. However there is currently no literature on recommended values for  $b_1$

and  $b_2$ . Users are urged to exercise some caution in this matter. Note also that for large values of  $b_1$  and  $b_2$ , the skip ahead calculation can take several seconds.

## 2.2 Quasi-random Numbers

Quasi-random numbers are intended primarily for use in Monte Carlo integration. Like pseudorandom numbers they are evenly distributed, but whereas pseudorandom sequences aim for statistical independence, quasi-random sequences aim for low discrepancy. Discrepancy is a measure of how evenly a sequence fills an area of multidimensional space. In a low discrepancy sequence, each point will tend to lie an equal distance from all its neighbouring points. This is typically not true for pseudorandom sequences, where one usually observes some form of clustering. For this reason quasi-random sequences are often more efficient in multidimensional Monte Carlo methods.

The low discrepancy sequence due to Sobol is provided here, based on the extension to higher dimensions described in Joe and Kuo (2003) and Joe and Kuo (2008). The digital scrambling described in Hong and Hickernell (2003) is provided. Please see `NagGpuScramTypes` for further details. Digital scrambling is an attempt to eliminate the bias inherent in a quasi-random sequence while retaining its low-discrepancy properties. It can be used to obtain error estimates of Monte Carlo integrals and can also alleviate systematic dependencies between dimensions, caused e.g. by poor choices of direction numbers.

## 2.3 Brownian Bridge

The term ‘Brownian bridge’ can mean one of two things: either it refers to a particular stochastic process which resembles a standard Brownian motion, but is forced to be zero at some time  $T > 0$ ; or it refers to a particular algorithm used to construct Brownian sample paths. The bridge algorithm constructs a Brownian sample path by first simulating its final value and then recursively filling in intermediate values through an interpolation formula. When compared with standard path construction methods, the bridge algorithm often displays advantages when solving stochastic differential equations or when pricing path dependent options. It is also possible to construct sample paths of the Brownian bridge process via the Brownian bridge algorithm, and this could be useful in certain stochastic models.

The Brownian bridge can be constructed in many different orderings. A modified depth-ordered algorithm is employed here, where the discretized Brownian motion  $X_k$ , for  $k = 0, 1, \dots, N$ , on the time interval  $[0, T]$  is generated in the order,  $X_0, X_T, X_{T/2}, X_{T/4}, X_{T/8}, \dots, X_1, \dots$  where for simplicity the case of equal time increments with  $N$  a power of 2 is shown. The algorithm provided allows for unequal time increments and does not restrict  $N$  to be a power of 2. However, for the purpose of introducing the samples from a Normal distribution, the Brownian bridge is structured internally by levels proceeding from the coarsest subdivisions of the time interval  $[0, T]$  to the finest. This ensures that largest part of the variance of the Brownian path is concentrated in the coarser levels which can be matched to the lowest dimensions of a low discrepancy sequence such as that of Sobol when these appear as the earliest entries in the input array. For further details please see the documentation for `naggpuDepthBBA`

## 3 Recommendations on Choice and Use of Available Functions

### 3.1 Host-Callable Generator Functions

A suite of pre-written CUDA kernels are provided which users can call from their CPU programs. These kernels will launch computations on the GPU device through the CUDA runtime. The available functionality is described below.

Prior to generating any pseudorandom variates the base generator being used must be initialized by calling the function `naggpuRandInitA`. Once initialized, a distributional generator can be called to obtain the variates required.

The random numbers computed can be chosen to be either double or single precision, however, double precision is recommended since the algorithms are designed for double precision storage and in single precision the sequences cannot be guaranteed to pass all statistical tests. If a sequence of random variates from a uniform distribution on the interval  $[a, b]$  is required, then the uniform distribution function `naggpuRandUniformA` should be called. Functions for Normal (`naggpuRandNormalA`), exponential (`naggpuRandExpA`) and gamma (`naggpuRandGammaA`) distributions are also included. Once all required

random variates have been obtained, `naggpuRandCleanupA` should be called to free allocated system resources.

`naggpuDepthBBIncInitA` initializes a binary tree which defines the Brownian bridge to be constructed. `naggpuDepthBBIncA` generates the Brownian bridge taking as input a previously computed array of pseudo or quasi-random numbers sampled from the standard Normal distribution. `naggpuDepthBBCleanupA` must be called to release system resources following generation of the Brownian bridge.

The Brownian bridge generated by `naggpuDepthBBIncA` can be used to solve a stochastic differential equation (SDE) driven by a Wiener process

$$dy(t) = f(t, y)dt + g(t, y)dW(t)$$

for

$$0 \leq t \leq T \quad \text{with} \quad y(0) = y_0,$$

where  $f(t, y)$  is the drift coefficient,  $g(t, y)$  is the diffusion coefficient and  $W(t)$  is the standard Wiener process whose increment is

$$\Delta W(t) = W(t + \Delta t) - W(t) = \sqrt{\Delta t}Z$$

and  $Z \sim \mathcal{N}(0, 1)$ .

For numerical solution, the given SDE can be discretized using the Euler-Maruyama method to give

$$y_{n+1} = y_n + f(t_n, y_n)\Delta t + g(t_n, y_n)\Delta W_n,$$

with  $t_n = n\Delta t$ .

The output values `d_bgIncs[i]`, for  $i = 0, 1, \dots, \text{dim} \times \text{nPaths} \times \text{nTimes}$  (see `naggpuDepthBBIncA` and `naggpuDepthBBIncInitA`), where **dim** is the dimension of the SDE, **nPaths** is the number of simulation paths and **nTimes** is the number of time steps, are of the form

$$\hat{W}_n = \frac{\Delta W_n}{\Delta t} = \frac{Z_n}{\sqrt{\Delta t}}$$

so that the discretized SDE is

$$y_{n+1} = y_n + \Delta t(f(t_n, y_n) + g(t_n, y_n)\hat{W}_n).$$

For other applications, `naggpuDepthBBA` is provided. This returns the values of a Brownian path at the specified interpolation points rather than the scaled increments supplied by `naggpuDepthBBIncA`. A call to the initialization function `naggpuDepthBBInitA` must be made before calling `naggpuDepthBBA` to generate the Brownian bridge and it should be followed by a call to `naggpuDepthBBCleanupA` to free resources.

The Brownian bridge construction is often used with a low discrepancy sequence, such as the Sobol sequence, to supply the Normal variates. The quasi-random sequence generator must be initialized by a call to `naggpuQuasiRandInitA`. The sequence of Normal variates can then be generated by `naggpuQuasiRandNormalA` and resources freed by `naggpuQuasiRandCleanupA`.

Uniform and exponential distributions for the Sobol sequence can be computed by `naggpuQuasiRandUniformA` and `naggpuQuasiRandExpA` respectively.

### 3.2 Inline Device Function Generators

A small set of inline GPU device functions are provided using PTX assembly. These are functions which can be called from *a user's own CUDA kernel*, in other words these functions live entirely on the GPU and can be used when writing CUDA kernels. Embedding a pseudorandom generator directly in a user's kernel has a number of advantages: most notably, it can reduce memory traffic by avoiding writing numbers to global memory only to read them back in again. However inlining a generator will typically require quite a few registers, and this can lead to register pressure in some applications. As a guideline, if a kernel does a large number of floating point calculations, inline generators may not be necessary since there is enough computation to hide the memory traffic. Inline device generators are provided for the MRG32k3a and the Sobol generators only.

The use of device functions called from a program executing on the GPU is described in NVIDIA CUDA (2011). The distribution functions provided here pass successive variates, as they are generated, to the user's CUDA threads as they execute on the GPU: each CUDA thread generates a contiguous segment of the quasi and pseudorandom number sequence. The functions are identified by the type qualifier `__device__`. The output of the distribution functions can be declared as double or float through the template argument FP.

Prior to generating any pseudorandom variates, the MRG32k3a inline generator must be initialized by a call to the host function `naggpuMrg32k3aDeviceInitA` followed by a call (in the user's CUDA kernel) to the GPU device function `naggpudevMrg32k3aInitA`. Successive variates from either uniform, Normal, exponential or gamma distributions are generated by each thread through calls to `naggpudevMrg32k3aUniformA`, `naggpudevMrg32k3aNormalA`, `naggpudevMrg32k3aExpA` or `naggpudevMrg32k3aGammaA`, respectively. Resources used in the initialization are freed by a final call to the host function `naggpuMrg32k3aDeviceCleanupA` after the user's kernel exits.

Before using the inline device functions for generating a Sobol sequence, the host initialization function `naggpuSobolDeviceInitA` must be called followed by a call (in the user's CUDA kernel) to the GPU device function `naggpudevSobolInitA`. The individual points in the Sobol sequence are then computed by repeated calls to `naggpudevSobolUniformA`, `naggpudevSobolNormalA` or `naggpudevSobolExpA`. Resources used in the initialization are freed by a final call to host function `naggpuSobolDeviceCleanupA` after the user's kernel exits. Note that generating of a set of Sobol numbers with the inline generators is slower than using the host generators such as `naggpuQuasiRandUniformA`.

## 4 Functionality Index

### 4.1 Host-Callable Generator Functions

Brownian bridge, depth-ordered construction,

free generator resources .....	<code>naggpuDepthBBCleanupA</code>
generate Brownian bridge .....	<code>naggpuDepthBBA</code>
generate Brownian bridge increments .....	<code>naggpuDepthBBIncA</code>
initialize bridge generator .....	<code>naggpuDepthBBInitA</code>
initialize incremental bridge generator .....	<code>naggpuDepthBBIncInitA</code>

Pseudorandom numbers,

free generator resources .....	<code>naggpuRandCleanupA</code>
initialize generator .....	<code>naggpuRandInitA</code>
variates from exponential distribution .....	<code>naggpuRandExpA</code>
variates from gamma distribution .....	<code>naggpuRandGammaA</code>
variates from Normal distribution .....	<code>naggpuRandNormalA</code>
variates from uniform distribution .....	<code>naggpuRandUniformA</code>

Quasi-random numbers,

free generator resources .....	<code>naggpuQuasiRandCleanupA</code>
initialize generator .....	<code>naggpuQuasiRandInitA</code>
variates from exponential distribution .....	<code>naggpuQuasiRandExpA</code>
variates from Normal distribution .....	<code>naggpuQuasiRandNormalA</code>
variates from uniform distribution .....	<code>naggpuQuasiRandUniformA</code>

### 4.2 Inline Device Function Generators

Pseudorandom numbers,

host cleanup for inline device function generator .....	<code>naggpuMrg32k3aDeviceCleanupA</code>
host setup for inline device function generator .....	<code>naggpuMrg32k3aDeviceInitA</code>
initialize inline device function generator .....	<code>naggpudevMrg32k3aInitA</code>
next variate from exponential distribution .....	<code>naggpudevMrg32k3aExpA</code>
next variate from gamma distribution .....	<code>naggpudevMrg32k3aGammaA</code>
next variate from Normal distribution .....	<code>naggpudevMrg32k3aNormalA</code>
next variate from uniform distribution .....	<code>naggpudevMrg32k3aUniformA</code>

sets parameters for gamma distribution ..... naggpudevMrg32k3aGammaSetParamsA

Quasi-random numbers,

host cleanup for inline device function generator ..... naggpuSobolDeviceCleanupA

host setup for inline device function generator ..... naggpuSobolDeviceInitA

initialize inline device function generator ..... naggpudevSobolInitA

next point from exponential distribution ..... naggpudevSobolExpA

next point from Normal distribution ..... naggpudevSobolNormalA

next point from uniform distribution ..... naggpudevSobolUniformA

### 4.3 Serial CPU Functions

Brownian bridge, depth-ordered construction,

free generator resources ..... nagCPUDepthBBCleanupA

generate Brownian bridge ..... nagCPUDepthBBA

generate Brownian bridge increments ..... nagCPUDepthBBIncA

initialize bridge generator ..... nagCPUDepthBBInitA

initialize incremental bridge generator ..... nagCPUDepthBBIncInitA

Pseudorandom numbers,

free generator resources ..... nagCPURandCleanupA

initialize generator ..... nagCPURandInitA

variates from exponential distribution ..... nagCPURandExpA

variates from gamma distribution ..... nagCPURandGammaA

variates from Normal distribution ..... nagCPURandNormalA

variates from uniform distribution ..... nagCPURandUniformA

Quasi-random numbers,

free generator resources ..... nagCPUQuasiRandCleanupA

initialize generator ..... nagCPUQuasiRandInitA

variates from exponential distribution ..... nagCPUQuasiRandExpA

variates from Normal distribution ..... nagCPUQuasiRandNormalA

variates from uniform distribution ..... nagCPUQuasiRandUniformA

## 5 References

Banks J (1998) *Handbook on Simulation* Wiley

Haramoto H, Matsumoto M, Nishimura T, Panneton F and L'Ecuyer P (2008) Efficient jump ahead for F2-linear random number generators *INFORMS J. on Computing* **20(3)** 385–390

Hong H S and Hickernell F J (2003) Algorithm 823: implementing scrambled digital sequences *ACM Trans. Math. Software* **29:2** 95–109

Joe S and Kuo F Y (2003) Remark on Algorithm 659: implementing Sobol's quasirandom sequence generator *ACM Trans. Math. Software (TOMS)* **29** 49–57

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

Knuth D E (1981) *The Art of Computer Programming (Volume 2)* (2nd Edition) Addison–Wesley

L'Ecuyer P (1999) Good parameter sets for combined multiple recursive random number generators *Operations Research* **47:1** 159–164

L'Ecuyer P, Simar R, Chen E J and Kelton W D (2002) An object-oriented random-number package with many long streams and substreams *Operations Research* **50:6** 1073–1075

Matsumoto M and Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator *ACM Transactions on Modelling and Computer Simulations*

NVIDIA CUDA (2011) *Programming Guide Version 4.0* <http://www.nvidia.com/cuda>



# NAG Numerical Routines for GPUs Function Document

## naggpuRandInitA

### 1 Purpose

**naggpuRandInitA** initializes a GPU pseudorandom number generator to give a repeatable sequence of pseudorandom numbers. This function must be called before any call to the GPU generator functions (such as **naggpuRandUniformA**) and must ultimately be followed by a call to the cleanup function **naggpuRandCleanupA** to release system resources. A base generator is selected through the **genid** parameter and initialized with the values given in the **seed** array.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuRandInitA(NagGpuRandGen genid, int a1, int b1, int a2, int b2,
    long long c, const unsigned int *seed, NagGpuRandComm *comm,
    NagGpuError *error)
```

### 3 Description

For different values of **seed**, a given generator will yield different sequences of random numbers. Alternatively, the same sequence of random numbers will be generated if the same value of **seed** is used. In general there is no guarantee of statistical properties between sequences, only within sequences. This is important when generators are used in parallel. This function can ‘skip ahead’ or advance the seed by an amount

$$s = a_1 2^{b_1} + a_2 2^{b_2} + c \quad (1)$$

so that the generator will produce the sequence of random numbers  $X_s, X_{s+1}, X_{s+2}, \dots$  instead of the original sequence  $X_0, X_1, X_2, \dots$ . This technique is useful to produce independent generators, often also called *independent streams and substreams*. Please see the Random Number Generators Chapter Introduction for further information.

Independent generators will be important mostly to applications which use multiple GPUs simultaneously. In this case, each generator will have its own **NagGpuRandComm** structure which encapsulates all the information the generator requires. An array of **NagGpuRandComm** structures with judiciously chosen skip aheads (see the Random Number Generators Chapter Introduction) represents an array of independent generators. Each structure must be initialized by a call to **naggpuRandInitA**. Note that when initializing a given communication structure, the call to **naggpuRandInitA** must access the same GPU device context which will be used when generating numbers with that communication structure. Please consult the CUDA documentation for details on how to achieve this and how to access multiple GPUs simultaneously from a single application.

For the most common task of generating a block of pseudorandom numbers on a single GPU, users will typically only have a single GPU generator (i.e. only one communication structure) and the skip ahead  $s$  above can be set to zero. The comments about arrays of communication structures and multiple GPU contexts can be ignored.

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

None.

## 5 Arguments

- 1: **genid** – NagGpuRandGen *Input*  
*On entry:* the type of generator to be used:  
`genid = NAGGPURANDGEN_MRG32K3A`  
`genid = NAGGPURANDGEN_MT19937`  
Please see the Random Number Generators Chapter Introduction for details about each of these base generators.  
*Constraint:* `genid = NAGGPURANDGEN_MRG32K3A` or `NAGGPURANDGEN_MT19937`.
  
- 2: **a1** – int *Input*  
*On entry:* the value of  $a_1$  in the skip ahead equation (1) above.  
*Constraint:*  $a_1 \geq 0$ .
  
- 3: **b1** – int *Input*  
*On entry:* the value of  $b_1$  in the skip ahead equation (1) above.  
*Constraints:*  
if `genid = NAGGPURANDGEN_MRG32K3A`,  $0 \leq b_1 \leq 191$ ;  
if `genid = NAGGPURANDGEN_MT19937`,  $0 \leq b_1 \leq 19937$ .
  
- 4: **a2** – int *Input*  
*On entry:* the value of  $a_2$  in the skip ahead equation (1) above.  
*Constraint:*  $a_2 \geq 0$ .
  
- 5: **b2** – int *Input*  
*On entry:* the value of  $b_2$  in the skip ahead equation (1) above.  
*Constraints:*  
if `genid = NAGGPURANDGEN_MRG32K3A`,  $0 \leq b_2 \leq 191$ ;  
if `genid = NAGGPURANDGEN_MT19937`,  $0 \leq b_2 \leq 19937$ .
  
- 6: **c** – long long *Input*  
*On entry:* the value of  $c$  in the skip ahead equation (1) above.  
*Constraint:*  $c \geq 0$ .
  
- 7: **seed**[ $n$ ] – const unsigned int \* *Input*  
*On entry:* an array of  $n$  32-bit unsigned integers to initialize the generator.  
*Constraints:*  
if `genid = NAGGPURANDGEN_MRG32K3A`,

$n = 6$   
 for  $i = 0, 1, 2$ ,  $\text{seed}[i] < 2^{32} - 209$  and  $\text{seed}(i) \neq 0$  for at least one  $i$   
 for  $i = 3, 4, 5$ ,  $\text{seed}[i] < 2^{32} - 22853$  and  $\text{seed}(i) \neq 0$  for at least one  $i$ ;

if  $\text{genid} = \text{NAGGPURANDGEN\_MT19937}$ ,

$n = 624$   
 for  $i = 0, 1, 2, \dots, 623$ ,  $\text{seed}(i) \neq 0$  for at least one  $i$ .

8: **comm** – NagGpuRandComm \* *Communication Data*

NagGpuRandComm is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator functions (such as naggpuRandUniformA). Once all required points have been obtained, **comm** must be passed to naggpuRandCleanupA to free allocated system resources.

9: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of  $\text{error} \rightarrow \text{code}$  which should be inspected after each call to this function. If  $\text{error} \rightarrow \text{code} = 0$  then no error occurred. If  $\text{error} \rightarrow \text{code} \neq 0$  then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

$\text{error} \rightarrow \text{code} = 1$

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

$\text{error} \rightarrow \text{code} = 2$

*During execution:* a CUDA runtime error was detected.

$\text{error} \rightarrow \text{code} = 100$

*On entry:* the value of **comm** is NULL.

$\text{error} \rightarrow \text{code} = 110$

*On entry:* **genid** does not specify a valid pseudorandom number generator. See NagGpuRandGen for permitted values.

$\text{error} \rightarrow \text{code} = 111$

*On entry:* the value of **a1** is negative.

$\text{error} \rightarrow \text{code} = 112$

*On entry:* the value of **b1** does not satisfy the constraints listed above.

$\text{error} \rightarrow \text{code} = 113$

*On entry:* the value of **a2** is negative.

$\text{error} \rightarrow \text{code} = 114$

*On entry:* the value of **b2** does not satisfy the constraints listed above.

$\text{error} \rightarrow \text{code} = 115$

*On entry:* the value of **c** is negative.

error → code = 116

*On entry:* the value of **seed** is NULL.

error → code = 117

*On entry:* the values in the **seed** array do not satisfy the constraints listed above.

## **7 Example**

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpuRandUniformA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuRandExpA

### 1 Purpose

**naggpuRandExpA** generates  $n$  values  $X_i$  from an exponential distribution with mean  $\lambda$ .

The initialization function `naggpuRandInitA` must be called prior to the first call to **naggpuRandExpA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function `naggpuRandCleanupA` must be called to free allocated system resources.

**Note:** To obtain the same values from **naggpuRandExpA** as from the function `nag_gpu_mrg32-k3a_exp(nb, nt, np, d_P)` in release 0.3 of the library, please see Section 2.1.1.1 in the Random Number Generators Chapter Introduction.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuRandExpA_sp(int n, NagGpuRandOrder order, float lambda,
    float *d_buff, const NagGpuRandTune *tune, cudaStream_t cstream,
    NagGpuRandComm *comm, NagGpuError *error)

extern "C"
cudaError_t naggpuRandExpA(int n, NagGpuRandOrder order, double lambda,
    double *d_buff, const NagGpuRandTune *tune, cudaStream_t cstream,
    NagGpuRandComm *comm, NagGpuError *error)
```

### 3 Description

The exponential distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\lambda}e^{-x/\lambda} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\lambda > 0$ . This function returns

$$X_i = -\lambda \ln Y_i$$

where  $Y_i$  are the next  $n$  values generated by the underlying uniform  $[0, 1]$  generator.

#### 3.1 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

## 5 Arguments

- 1: **n** – int *Input*  
*On entry:* the number of random values to be generated.  
*Constraint:*  $n \geq 1$ .
- 2: **order** – NagGpuRandOrder *Input*  
*On entry:* the ordering to be observed by the underlying GPU generator:  
order = NAGGPURANDORDER\_OPTIMAL  
order = NAGGPURANDORDER\_CONSISTENT  
See NagGpuRandOrder for further details.  
*Constraint:*  
order = NAGGPURANDORDER\_OPTIMAL or NAGGPURANDORDER\_CONSISTENT.
- 3: **lambda** – float *Input*  
4: **lambda** – double *Input*  
This parameter has type float or double depending on whether the single or double precision version of this function is called.  
*On entry:* the mean,  $\lambda$ , of the distribution.  
*Constraint:*  $\lambda > 0$ .
- 5: **d\_buff[n]** – float \* *Output*  
6: **d\_buff[n]** – double \* *Output*  
This parameter has type float or double depending on whether the single or double precision version of this function is called.  
This buffer must reside in the GPU memory space.  
*On exit:* the **n** pseudorandom numbers from the specified distribution. The output tuning structure `comm`  $\rightarrow$  `tuneParamsUsed` will contain the parameters used to launch the kernel. If `order = NAGGPURANDORDER_OPTIMAL`, these parameters may determine the output ordering (see NagGpuRandTune for details).
- 7: **tune** – const NagGpuRandTune \* *Input*  
This parameter is optional and may be set to NULL.  
*On entry:* if specified, points to a NagGpuRandTune structure containing launch parameters for the selected GPU kernel. Upon a successful return from this function, the relevant data will be copied to the output tuning structure `comm`  $\rightarrow$  `tuneParamsUsed`. Please see NagGpuRandTune for additional information about performance tuning.
- 8: **custream** – cudaStream\_t *Input*  
*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.
- 9: **comm** – NagGpuRandComm \* *Communication Data*  
NagGpuRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to naggpuRandCleanupA to free allocated system resources.  
Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through **comm**. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see NagGpuRandTune for details on performance

tuning, and consult the NagGpuRandComm documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling naggpuRandCleanupA.

10: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **order** does not specify a valid ordering. See NagGpuRandOrder for permitted values.

error → code = 112

*On entry:* **d\_buff** is NULL.

error → code = 114

*On entry:*  $\lambda \leq 0$ .

error → code = 200

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details), order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and the kernel launch requires too much shared memory. Try reducing the value of tune → mrgConAThdsPerBlk.

error → code = 250

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details), order = NAGGPURANDORDER\_OPTIMAL, tune ≠ NULL and tune → mrgOptAThdsPerBlk is out of bounds. See NagGpuRandTune for further details.

error → code = 251

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_OPTIMAL, tune ≠ NULL and  
 tune → mrgOptAPtsPerThd is out of bounds. See NagGpuRandTune for further details.

error → code = 252

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and  
 tune → mrgConAThdsPerBlk is out of bounds. See NagGpuRandTune for further details.

error → code = 253

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and  
 tune → mrgConAThdsPerBlk is not divisible by  $W$  where  $W = 16$  on devices of compute  
 capability 1.3 or lower and  $W = 32$  otherwise. See NagGpuRandTune for further details.

error → code = 254

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and  
 tune → mrgConANumLoops is out of bounds. See NagGpuRandTune for further details.

error → code = 300

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and tune → mtANumBlks is out of bounds. See NagGpuRandTune for further  
 details.

error → code = 301

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and tune → mtAGen is NULL.

error → code = 302

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and tune → mtAGen contains negative entries. See NagGpuRandTune for  
 further details.

error → code = 303

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and the entries in tune → mtAGen do not sum to  $n$ . See NagGpuRandTune  
 for further details.

## 7 Example

This example program uses **naggpuRandExpA** to print 50 pseudorandom numbers from an exponential distribution using the MRG32k3a generator. For usage of the MT19937 generator, including tuning aspects, please see the example program for naggpuRandNormalA.

### 7.1 Program Text

```

/*
 * Example Program: naggpu_rand_expA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

```

```

#include <nag_gpu.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void printNagTuningParamsUsed(NagGpuRandComm *comm);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for generated GPU random numbers
    FP *h_buff = 0;
    // device (GPU) storage for generated random numbers
    FP *d_buff = 0;

    // total number of points to generate
    int N = 1000000;

    // seed variables
    const int seed_length = 6;
    unsigned int seed[seed_length];

    // skip ahead variables
    int a1, a2, b1, b2;
    long long c;

    // distribution parameters
    FP lambda = 1.0;

    // NAG GPU structures
    NagGpuRandComm comm;
    NagGpuRandTune tune;
    NagGpuError error;

    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpuRandExpA";
    if (sizeof(FP)==sizeof(float)) cout << "_sp";
    cout << endl << endl;

    // Allocate CPU and GPU memory
    h_buff = new FP[N];
    cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
    checkCudaError(cuError);

    // Initialise the generator only once
    cout << "Initialising generator..." << endl << endl;

    // arbitrary seed and skip ahead parameters
    for (int i = 0; i < seed_length; i++) seed[i] = i;
    a1 = 14;
    b1 = 34;
    a2 = 2;
    b2 = 21;
    c = 123;

    naggpuRandInitA(NAGGPURANDGEN_MRG32K3A, a1, b1, a2, b2,
                   c, seed, &comm, &error);
    checkNagError(&error);
}

```

```

// Generate N/2 numbers using default tuning parameters
cout << "Generate with default tuning parameters..." << endl;
#ifdef SINGLEPRECISION
    naggpuRandExpA_sp(N/2, NAGGPURANDORDER_CONSISTENT, lambda,
                      d_buff, NULL, 0, &comm, &error);
#else
    naggpuRandExpA(N/2, NAGGPURANDORDER_CONSISTENT, lambda,
                   d_buff, NULL, 0, &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Generate (N- N/2) numbers using specified tuning parameters
cout << "Generate with user supplied tuning parameters..." << endl;
tune.mrgConAThdsPerBlk = 96;
tune.mrgConANumLoops = 30;
cout << "Tuning Parameters supplied: " << endl;
cout << "  tune.mrgConAThdsPerBlk = " << tune.mrgConAThdsPerBlk
    << endl;
cout << "  tune.mrgConANumLoops = " << tune.mrgConANumLoops
    << endl;
#ifdef SINGLEPRECISION
    naggpuRandExpA_sp(N - N/2, NAGGPURANDORDER_CONSISTENT, lambda,
                      d_buff + N/2, &tune, 0, &comm, &error);
#else
    naggpuRandExpA(N - N/2, NAGGPURANDORDER_CONSISTENT, lambda,
                   d_buff + N/2, &tune, 0, &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Copy back values from the GPU for printing
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The first 50 GPU random numbers:" << endl;
cout.setf(ios::fixed, ios::floatfield);
cout.precision(3);
for(int row = 0; row < 10; row++)
{
    for(int col = 0; col < 5; col++)
    {
        cout << h_buff[row*10 + col] << "\t";
    }
    cout << endl;
}

// Call cleanup for the NAG routine
naggpuRandCleanupA(&comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}

```

```

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void printNagTuningParamsUsed(NagGpuRandComm *comm)
{
    cout << "  comm.tuneOrigin = ";

    switch (comm->tuneOrigin)
    {
        case NAGGPUTUNEORIGIN_NA:
            cout << "NAGGPUTUNEORIGIN_NA";
            break;
        case NAGGPUTUNEORIGIN_DEFAULT:
            cout << "NAGGPUTUNEORIGIN_DEFAULT";
            break;
        case NAGGPUTUNEORIGIN_USER:
            cout << "NAGGPUTUNEORIGIN_USER";
            break;
        case NAGGPUTUNEORIGIN_AUTO:
            cout << "NAGGPUTUNEORIGIN_AUTO";
            break;
        default:
            cout << "Unrecognised tuneOrigin";
    }
    cout << endl;

    cout << "  comm.tuneParamsUsed->mrgConAThdsPerBlk = ";
    cout << comm->tuneParamsUsed->mrgConAThdsPerBlk << endl;
    cout << "  comm.tuneParamsUsed->mrgConANumLoops = ";
    cout << comm->tuneParamsUsed->mrgConANumLoops << endl;

    cout << endl;
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

### 7.3 Program Results

NAG GPU Example Program: naggpuRandExpA\_sp

Initialising generator...

Generate with default tuning parameters...

Tuning Parameters used:

```
comm.tuneOrigin = NAGGPOTUNEORIGIN_DEFAULT
comm.tuneParamsUsed->mrgConAThdsPerBlk = 32
comm.tuneParamsUsed->mrgConANumLoops = 8
```

Generate with user supplied tuning parameters...

Tuning Parameters supplied:

```
tune.mrgConAThdsPerBlk = 96
tune.mrgConANumLoops = 30
```

Tuning Parameters used:

```
comm.tuneOrigin = NAGGPOTUNEORIGIN_USER
comm.tuneParamsUsed->mrgConAThdsPerBlk = 96
comm.tuneParamsUsed->mrgConANumLoops = 30
```

The first 50 GPU random numbers:

```
1.030 0.043 1.340 0.433 0.352
1.427 0.664 1.182 2.276 0.313
0.019 1.246 0.671 0.111 1.340
0.352 0.921 0.456 0.658 0.581
0.941 0.235 0.400 0.353 0.506
1.579 0.439 2.200 0.518 0.064
0.274 0.110 2.678 0.597 0.903
0.725 0.427 1.912 1.212 0.340
0.151 2.020 0.992 0.347 5.351
0.975 1.483 2.598 0.150 1.632
```

---

# NAG Numerical Routines for GPUs Function Document

## naggpuRandGammaA

### 1 Purpose

**naggpuRandGammaA** generates  $n$  values  $X_i$  from a gamma distribution with shape parameter  $\alpha$  and scale parameter  $\beta$ .

The initialization function `naggpuRandInitA` must be called prior to the first call to **naggpuRandGammaA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function `naggpuRandCleanupA` must be called to free allocated system resources.

**Note:** currently only the MRG32k3a base generator is supported. Support for MT19937 will be added in future releases.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuRandGammaA_sp(int n, NagGpuRandOrder order, float alpha,
    float beta, float *d_buff, const NagGpuRandTune *tune,
    cudaStream_t cstream, NagGpuRandComm *comm, NagGpuError *error)

extern "C"
cudaError_t naggpuRandGammaA(int n, NagGpuRandOrder order, double alpha,
    double beta, double *d_buff, const NagGpuRandTune *tune,
    cudaStream_t cstream, NagGpuRandComm *comm, NagGpuError *error)
```

### 3 Description

The gamma distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\alpha, \beta > 0$ . The rejection algorithm described in Marsaglia and Tsang (2000) is used to generate the gamma pseudorandom variates when  $\alpha \geq 1$ . When  $0 < \alpha < 1$ , the scaling

$$\gamma_\alpha = \gamma_{1+\alpha} U^\alpha$$

is used where  $U$  denotes a uniform random variable in the interval  $[0, 1]$  and  $\gamma_\alpha$  denotes a gamma random variable with shape parameter  $\alpha$  and scale parameter  $\beta = 1$ . Please see the members of the `NagGpuRandTune` structure which pertain to this function for further details about the implementation. Note that currently only the MRG32k3a base generator is supported. In addition, when `order = NAGGPURANDORDER_OPTIMAL`, the sequences produced by GPUs with compute capability 1.3 and lower will differ from those produced by GPUs with compute capability 2.0 and higher, even when identical launch configurations are specified (via **tune**). This is due to different optimal orderings being used for the different GPU architectures.

**Note:** rejection algorithms are extremely sensitive to computational accuracy. When a variate is generated close to the rejection envelope, small differences in numerical values can lead to it being accepted in double precision while it is rejected in single precision (or vice versa). From this point on, the single and double precision sequences will be different. The same behaviour is seen when comparing single precision sequences generated on the CPU and the GPU: differences in the floating point calculations will lead to the sequences diverging after a certain number of variates. In double precision, the CPU and GPU sequences will take much longer (on average) before they diverge, agreeing to tens or even hundreds of millions of variates before numerical differences cause a variate to be accepted on one platform while it is rejected on the other.

### 3.1 Synchronization

When `order = NAGGPURANDORDER_CONSISTENT`, this function is *blocking*. Control will not return to the calling program until the function has completed. An iterative procedure is followed to generate all `n` pseudorandom variates in consistent order.

When `order = NAGGPURANDORDER_OPTIMAL`, this function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. Independent substreams are used to avoid the blocking required to produce pseudorandom variates in consistent order. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

Marsaglia G and Tsang W W (2000) A simple method for generating Gamma variables *ACM Trans. Math. Software* **26(3)** 363–372

## 5 Arguments

- 1: **n** – int *Input*  
*On entry:* the number of random values to be generated.  
*Constraint:*  $n \geq 1$ .
- 2: **order** – NagGpuRandOrder *Input*  
*On entry:* the ordering to be observed by the underlying GPU generator:  
`order = NAGGPURANDORDER_OPTIMAL`  
`order = NAGGPURANDORDER_CONSISTENT`  
 See NagGpuRandOrder for further details.  
*Constraint:*  
`order = NAGGPURANDORDER_OPTIMAL` or `NAGGPURANDORDER_CONSISTENT`.
- 3: **alpha** – float *Input*  
 4: **alpha** – double *Input*  
 This parameter has type float or double depending on whether the single or double precision version of this function is called.  
*On entry:* the shape parameter,  $\alpha$ , of the distribution.  
*Constraint:*  $\alpha > 0$ .
- 5: **beta** – float *Input*  
 6: **beta** – double *Input*  
 This parameter has type float or double depending on whether the single or double precision version of this function is called.  
*On entry:* the scale parameter,  $\beta$ , of the distribution.  
*Constraint:*  $\beta > 0$ .

- 7: **d\_buff[n]** – float \* *Output*  
 8: **d\_buff[n]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

This buffer must reside in the GPU memory space.

*On exit:* the **n** pseudorandom numbers from the specified distribution. The output tuning structure `comm → tuneParamsUsed` will contain the parameters used to launch the kernel. If `order = NAGGPURANDORDER_OPTIMAL`, these parameters may determine the output ordering (see `NagGpuRandTune` for details).

- 9: **tune** – const `NagGpuRandTune` \* *Input*

This parameter is optional and may be set to `NULL`.

*On entry:* if specified, points to a `NagGpuRandTune` structure containing launch parameters for the selected GPU kernel. Upon a successful return from this function, the relevant data will be copied to the output tuning structure `comm → tuneParamsUsed`. Please see `NagGpuRandTune` for additional information about performance tuning.

- 10: **estream** – `cudaStream_t` *Input*

*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details. When `order = NAGGPURANDORDER_CONSISTENT`, this function is blocking and **estream** is not used.

- 11: **comm** – `NagGpuRandComm` \* *Communication Data*

`NagGpuRandComm` is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to `naggpuRandCleanupA` to free allocated system resources.

Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through **comm**. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see `NagGpuRandTune` for details on performance tuning, and consult the `NagGpuRandComm` documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling `naggpuRandCleanupA`.

- 12: **error** – `NagGpuError` \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

`error → code = 1`

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error → code = 2`

*During execution:* a CUDA runtime error was detected.

error → code = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **order** does not specify a valid ordering. See NagGpuRandOrder for permitted values.

error → code = 112

*On entry:* **d\_buff** is NULL.

error → code = 113

*On entry:*  $\alpha \leq 0$

error → code = 114

*On entry:*  $\beta \leq 0$

error → code = 115

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for further details). Currently only the MRG32k3a base generator is supported.

error → code = 200

*On entry:*  $\text{order} = \text{NAGGPURANDORDER\_CONSISTENT}$ ,  $\text{tune} \neq \text{NULL}$  and the kernel launch requires too much shared memory. Try reducing the number of threads per block. The runtime error message obtained from naggpuErrorCopyMsg will contain additional diagnostic information.

error → code = 201

*On entry:*  $\text{order} = \text{NAGGPURANDORDER\_OPTIMAL}$ ,  $\text{tune} \neq \text{NULL}$  and the kernel launch requires too much shared memory. Try reducing the value of  $\text{tune} \rightarrow \text{mrgRejOptAThdsPerBlk}$ .

error → code = 250

*On entry:*  $\text{order} = \text{NAGGPURANDORDER\_OPTIMAL}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgRejOptAThdsPerBlk}$  is out of bounds. See NagGpuRandTune for further details.

error → code = 251

*On entry:*  $\text{order} = \text{NAGGPURANDORDER\_OPTIMAL}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgRejOptAThdsPerBlk}$  is not divisible by  $W$  where  $W = 16$  on devices of compute capability 1.3 or lower and  $W = 32$  otherwise. See NagGpuRandTune for further details.

error → code = 252

*On entry:*  $\text{order} = \text{NAGGPURANDORDER\_OPTIMAL}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgRejOptANumBlks}$  is out of bounds. See NagGpuRandTune for further details.

error → code = 253

*On entry:*  $\text{order} = \text{NAGGPURANDORDER\_CONSISTENT}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgRejConA}$  is NULL. See NagGpuRandTune for further details.

error → code = 254

*On entry:* order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and the number of threads per block returned by tune → mrgRejConA is out of bounds. See NagGpuRandTune for further details. The runtime error message obtained from naggpuErrorCopyMsg will contain additional diagnostic information.

error → code = 255

*On entry:* order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and the number of threads per block returned by tune → mrgRejConA is not divisible by  $W$  where  $W = 16$  on devices of compute capability 1.3 or lower and  $W = 32$  otherwise. See NagGpuRandTune for further details. The runtime error message obtained from naggpuErrorCopyMsg will contain additional diagnostic information.

error → code = 256

*On entry:* order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and the number of blocks returned by tune → mrgRejConA is out of bounds. See NagGpuRandTune for further details. The runtime error message obtained from naggpuErrorCopyMsg will contain additional diagnostic information.

## 7 Example

This example program uses **naggpuRandGammaA** to print 50 pseudorandom numbers from a gamma distribution using the MRG32k3a generator.

### 7.1 Program Text

```

/*
 * Example Program: naggpuRandGammaA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void printNagTuningParamsUsed(NagGpuRandComm *comm);
void checkCudaError(cudaError_t cuError);

// This function is for illustrative purposes only.
// Realistic tuning functions would have finer subdivisions
// of the npts range and would probably use higher values
// of nthds.
static void tuneFunc(int npts, int *nthds, int *nblks)
{
    if(npts < 10000) {
        (*nthds) = 128;
        (*nblks) = 10;
    } else {
        (*nthds) = 64;
        (*nblks) = 60;
    }
}

```

```

}

int main(int argc, char **argv)
{
    // host (CPU) storage for generated GPU random numbers
    FP *h_buff = 0;
    // device (GPU) storage for generated random numbers
    FP *d_buff = 0;

    // total number of points to generate
    int N = 1000000;

    // seed variables
    const int seed_length = 6;
    unsigned int seed[seed_length];

    // skip ahead variables
    int a1, a2, b1, b2;
    long long c;

    // distribution parameters
    FP alpha = 1.7;
    FP beta = 1.3;

    // NAG GPU structures
    NagGpuRandComm comm;
    NagGpuRandTune tune;
    NagGpuError error;

    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpuRandGammaA";
    if (sizeof(FP)==sizeof(float)) cout << "_sp";
    cout << endl << endl;

    // Allocate CPU and GPU memory
    h_buff = new FP[N];
    cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
    checkCudaError(cuError);

    // Initialise the generator only once
    cout << "Initialising generator..." << endl << endl;

    // arbitrary seed and skip ahead parameters
    for (int i = 0; i < seed_length; i++) seed[i] = i;
    a1 = 14;
    b1 = 34;
    a2 = 2;
    b2 = 21;
    c = 123;

    naggpuRandInitA(NAGGPURANDGEN_MRG32K3A, a1, b1, a2, b2,
                   c, seed, &comm, &error);
    checkNagError(&error);

    // Generate N/2 numbers using default tuning parameters
    cout << "Generate with default tuning parameters..." << endl;
#ifdef SINGLEPRECISION
    naggpuRandGammaA_sp(N/2, NAGGPURANDORDER_CONSISTENT, alpha, beta,
                       d_buff, NULL, 0, &comm, &error);
#else
    naggpuRandGammaA(N/2, NAGGPURANDORDER_CONSISTENT, alpha, beta,
                    d_buff, NULL, 0, &comm, &error);
#endif
    checkNagError(&error);
}

```

```

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Generate (N- N/2) numbers using specified tuning parameters
cout << "Generate with user supplied tuning parameters..." << endl;
tune.mrgRejConA =
cout << "Tuning Parameters supplied: " << endl;
cout << "  tune.mrgRejConA = " << (void*)tune.mrgRejConA
  << endl;

#ifdef SINGLEPRECISION
  naggpuRandGammaA_sp(N - N/2, NAGGPURANDORDER_CONSISTENT, alpha, beta,
    d_buff + N/2, &tune, 0, &comm, &error);
#else
  naggpuRandGammaA(N - N/2, NAGGPURANDORDER_CONSISTENT, alpha, beta,
    d_buff + N/2, &tune, 0, &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Copy back values from the GPU for printing
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
  cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The first 50 GPU random numbers:" << endl;
cout.setf(ios::fixed, ios::floatfield);
cout.precision(3);
for(int row = 0; row < 10; row++)
  {
    for(int col = 0; col < 5; col++)
      {
        cout << h_buff[row*10 + col] << "\t";
      }
    cout << endl;
  }

// Call cleanup for the NAG routine
naggpuRandCleanupA(&comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
  {
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
  }

return 0;
}

void checkNagError(NagGpuError *error)
{
  if (error->code != 0)
    {
      char *buff;
      buff = new char[error->msgLength];
      naggpuErrorCopyMsg(buff, error);
    }
}

```

```

        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void printNagTuningParamsUsed(NagGpuRandComm *comm)
{
    cout << "  comm.tuneOrigin = ";

    switch (comm->tuneOrigin)
    {
        case NAGGPUTUNEORIGIN_NA:
            cout << "NAGGPUTUNEORIGIN_NA";
            break;
        case NAGGPUTUNEORIGIN_DEFAULT:
            cout << "NAGGPUTUNEORIGIN_DEFAULT";
            break;
        case NAGGPUTUNEORIGIN_USER:
            cout << "NAGGPUTUNEORIGIN_USER";
            break;
        case NAGGPUTUNEORIGIN_AUTO:
            cout << "NAGGPUTUNEORIGIN_AUTO";
            break;
        default:
            cout << "Unrecognised tuneOrigin";
    }
    cout << endl;

    cout << "  comm.tuneParamsUsed->mrgRejConA = ";
    cout << (void*)comm->tuneParamsUsed->mrgRejConA << endl;

    cout << endl;
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuRandGammaA\_sp

Initialising generator...

Generate with default tuning parameters...

Tuning Parameters used:

```

    comm.tuneOrigin = NAGGPUTUNEORIGIN_DEFAULT
    comm.tuneParamsUsed->mrgRejConA = 0x2b755111cab0

```

Generate with user supplied tuning parameters...

Tuning Parameters supplied:

```

    tune.mrgRejConA = 0x401060

```

Tuning Parameters used:

```

    comm.tuneOrigin = NAGGPUTUNEORIGIN_USER
    comm.tuneParamsUsed->mrgRejConA = 0x401060

```

The first 50 GPU random numbers:

```

1.267 4.824 0.902 1.178 2.641
2.077 1.709 0.679 0.428 4.019

```

0.426 2.351 3.099 0.903 2.076  
2.829 1.440 4.640 3.622 1.027  
2.399 0.047 0.103 2.336 0.240  
4.441 8.641 5.222 2.708 0.926  
2.378 0.843 1.122 2.630 3.052  
0.553 2.121 3.696 4.009 1.528  
2.402 0.562 0.460 0.044 4.720  
2.188 1.400 0.405 2.820 0.029

---



# NAG Numerical Routines for GPUs Function Document

## naggpuRandNormalA

### 1 Purpose

**naggpuRandNormalA** generates  $n$  values  $X_i$  from a Normal distribution with mean  $\mu$  and variance  $\sigma^2$ . The initialization function **naggpuRandInitA** must be called prior to the first call to **naggpuRandNormalA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function **naggpuRandCleanupA** must be called to free allocated system resources.

**Note:** To obtain the same values from **naggpuRandNormalA** as from the function **nag\_gpu\_mrg32k3a\_normal(nb, nt, np, d\_P)** in release 0.3 of the library, please see Section 2.1.1.1 in the Random Number Generators Chapter Introduction.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuRandNormalA_sp(int n, NagGpuRandOrder order, float mu,
    float sigma, float *d_buff, const NagGpuRandTune *tune,
    cudaStream_t cstream, NagGpuRandComm *comm, NagGpuError *error)

extern "C"
cudaError_t naggpuRandNormalA(int n, NagGpuRandOrder order, double mu,
    double sigma, double *d_buff, const NagGpuRandTune *tune,
    cudaStream_t cstream, NagGpuRandComm *comm, NagGpuError *error)
```

### 3 Description

The Normal distribution has probability density function given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where  $\sigma > 0$  and  $\mu \in \mathbb{R}$ . This function uses a Box-Muller transform to convert a pair of uniform  $(0, 1)$  random numbers into a pair of Normal random numbers. Let  $X_0, X_1, X_2, \dots$  denote the sequence of uniform  $(0, 1)$  pseudorandom variates as specified by the base generator algorithm. When `order = NAGGPURANDORDER_CONSISTENT`, the function will always use successive pairs of uniform variates in the Box-Muller transform to produce successive pairs of Normal variates, i.e.  $(X_0, X_1) \mapsto (Z_0, Z_1)$ ,  $(X_2, X_3) \mapsto (Z_2, Z_3)$  where  $Z_0, Z_1, Z_2, \dots$  denotes the output sequence of Normal variates. When `order = NAGGPURANDORDER_OPTIMAL`:

- the MRG32k3a generator will continue to use successive pairs of uniform variates as described above, but will store the output Normal variates in the permuted order specified in `NagGpuRandTune`
- the MT19937 generator will select uniform variates and store the output Normal variates in an implementation-dependent manner

#### 3.1 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

None.

## 5 Arguments

- 1: **n** – int *Input*  
*On entry:* the number of random values to be generated.  
*Constraint:*  $n \geq 1$ .
- 2: **order** – NagGpuRandOrder *Input*  
*On entry:* the ordering to be observed by the underlying GPU generator:  
`order = NAGGPURANDORDER_OPTIMAL`  
`order = NAGGPURANDORDER_CONSISTENT`  
 See NagGpuRandOrder for further details.  
*Constraint:*  
`order = NAGGPURANDORDER_OPTIMAL` or `NAGGPURANDORDER_CONSISTENT`.
- 3: **mu** – float *Input*  
 4: **mu** – double *Input*  
 This parameter has type float or double depending on whether the single or double precision version of this function is called.  
*On entry:* the mean,  $\mu$ , of the distribution.
- 5: **sigma** – float *Input*  
 6: **sigma** – double *Input*  
 This parameter has type float or double depending on whether the single or double precision version of this function is called.  
*On entry:* the standard deviation,  $\sigma$ , of the distribution.  
*Constraint:*  $\text{sigma} > 0$ .
- 7: **d\_buff[n]** – float \* *Output*  
 8: **d\_buff[n]** – double \* *Output*  
 This parameter has type float or double depending on whether the single or double precision version of this function is called.  
 This buffer must reside in the GPU memory space.  
*On exit:* the **n** pseudorandom numbers from the specified distribution. The output tuning structure `comm`  $\rightarrow$  `tuneParamsUsed` will contain the parameters used to launch the kernel. If `order = NAGGPURANDORDER_OPTIMAL`, these parameters may determine the output ordering (see NagGpuRandTune for details).
- 9: **tune** – const NagGpuRandTune \* *Input*  
 This parameter is optional and may be set to NULL.  
*On entry:* if specified, points to a NagGpuRandTune structure containing launch parameters for the selected GPU kernel. Upon a successful return from this function, the relevant data will be copied

to the output tuning structure `comm` → `tuneParamsUsed`. Please see `NagGpuRandTune` for additional information about performance tuning.

10: **custream** – `cudaStream_t` *Input*

*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.

11: **comm** – `NagGpuRandComm *` *Communication Data*

`NagGpuRandComm` is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to `naggpuRandCleanupA` to free allocated system resources.

Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through **comm**. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see `NagGpuRandTune` for details on performance tuning, and consult the `NagGpuRandComm` documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling `naggpuRandCleanupA`.

12: **error** – `NagGpuError *` *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error` → `code` which should be inspected after each call to this function. If `error` → `code` = 0 then no error occurred. If `error` → `code` ≠ 0 then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

`error` → `code` = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error` → `code` = 2

*During execution:* a CUDA runtime error was detected.

`error` → `code` = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

`error` → `code` = 100

*On entry:* the value of **comm** is NULL.

`error` → `code` = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

`error` → `code` = 110

*On entry:*  $n \leq 0$ .

`error` → `code` = 111

*On entry:* **order** does not specify a valid ordering. See `NagGpuRandOrder` for permitted values.

`error` → `code` = 112

*On entry:* **d\_buff** is NULL.

error → code = 115

*On entry:*  $\sigma \leq 0$

error → code = 200

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  $\text{order} = \text{NAGGPURANDORDER\_CONSISTENT}$ ,  $\text{tune} \neq \text{NULL}$  and the kernel launch requires too much shared memory. Try reducing the value of  $\text{tune} \rightarrow \text{mrgConAThdsPerBlk}$ .

error → code = 250

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  $\text{order} = \text{NAGGPURANDORDER\_OPTIMAL}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgOptAThdsPerBlk}$  is out of bounds. See NagGpuRandTune for further details.

error → code = 251

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  $\text{order} = \text{NAGGPURANDORDER\_OPTIMAL}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgOptAPtsPerThd}$  is out of bounds. See NagGpuRandTune for further details.

error → code = 252

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  $\text{order} = \text{NAGGPURANDORDER\_CONSISTENT}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgConAThdsPerBlk}$  is out of bounds. See NagGpuRandTune for further details.

error → code = 253

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  $\text{order} = \text{NAGGPURANDORDER\_CONSISTENT}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgConAThdsPerBlk}$  is not divisible by  $W$  where  $W = 16$  on devices of compute capability 1.3 or lower and  $W = 32$  otherwise. See NagGpuRandTune for further details.

error → code = 254

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  $\text{order} = \text{NAGGPURANDORDER\_CONSISTENT}$ ,  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mrgConANumLoops}$  is out of bounds. See NagGpuRandTune for further details.

error → code = 300

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mtANumBlks}$  is out of bounds. See NagGpuRandTune for further details.

error → code = 301

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mtAGen}$  is NULL.

error → code = 302

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{mtAGen}$  contains negative entries. See NagGpuRandTune for further details.

error → code = 303

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  $\text{tune} \neq \text{NULL}$  and the entries in  $\text{tune} \rightarrow \text{mtAGen}$  do not sum to  $\mathbf{n}$ . See NagGpuRandTune for further details.

error → code = 304

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  $\text{order} = \text{NAGGPURANDORDER\_CONSISTENT}$ ,  $\text{tune} \neq \text{NULL}$  and some of the entries in  $\text{tune} \rightarrow \text{mtAGen}$  are odd. See NagGpuRandTune for further details.

## 7 Example

This example program uses **naggpuRandNormalA** to print 50 pseudorandom numbers from a uniform distribution using the MT19937 generator. For usage of the MRG32k3a generator, including tuning aspects, please see the example program for **naggpuRandUniformA**.

### 7.1 Program Text

```

/*
 * Example Program: naggpu_rand_normalA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 */

#include
using namespace std;

#include <nag_gpu.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void printNagTuningParamsUsed(NagGpuRandComm *comm);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for generated GPU random numbers
    FP *h_buff = 0;
    // device (GPU) storage for generated random numbers
    FP *d_buff = 0;

    // total number of points to generate
    int N = 1000001;

    // seed variables
    const int seed_length = 624;
    unsigned int seed[seed_length];

    // skip ahead variables
    int a1, a2, b1, b2;
    long long c;

    // distribution parameters
    FP mu = 0.0;
    FP sigma = 1.0;

    // NAG GPU structures
    NagGpuRandComm comm;
    NagGpuRandTune tune;
    NagGpuError error;

    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpuRandNormalA";
    if (sizeof(FP)==sizeof(float)) cout << "_sp";
    cout << endl << endl;
}

```

```

// Allocate CPU and GPU memory
h_buff = new FP[N];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
checkCudaError(cuError);

// Initialise the generator only once
cout << "Initialising generator..." << endl << endl;

// arbitrary seed and skip ahead parameters
for (int i = 0; i < seed_length; i++) seed[i] = i;
a1 = 14;
b1 = 34;
a2 = 2;
b2 = 21;
c = 123;

naggpuRandInitA(NAGGPURANDGEN_MT19937, a1, b1, a2, b2,
                c, seed, &comm, &error);
checkNagError(&error);

// Generate N/2 numbers using default tuning parameters
cout << "Generate with default tuning parameters..." << endl;
#ifdef SINGLEPRECISION
naggpuRandNormalA_sp(N/2, NAGGPURANDORDER_CONSISTENT, mu, sigma,
                    d_buff, NULL, 0, &comm, &error);
#else
naggpuRandNormalA(N/2, NAGGPURANDORDER_CONSISTENT, mu, sigma,
                 d_buff, NULL, 0, &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Generate (N- N/2) numbers using specified tuning parameters
cout << "Generate with user supplied tuning parameters..." << endl;
int N2 = N - N/2;

tune.mtANumBlks = 20;
tune.mtAGen = new int[tune.mtANumBlks];

// check pts_per_block is even
int pts_per_block = N2 / tune.mtANumBlks;
if (pts_per_block%2 == 1) pts_per_block -= 1;
for(int i = 0; i < tune.mtANumBlks - 1; i++)
{
    tune.mtAGen[i] = pts_per_block;
}

// last block does not have to generate an even number of points
int pts_last_block = N2 - (tune.mtANumBlks - 1)*pts_per_block;
tune.mtAGen[tune.mtANumBlks - 1] = pts_last_block;

cout << "Tuning Parameters supplied: " << endl;
cout << "  tune.mtANumBlks = ";
cout << tune.mtANumBlks << endl;

for(int i = 0; i < tune.mtANumBlks; i++)
{
    cout << "  tune.mtAGen[" << i << "] = ";
    cout << tune.mtAGen[i] << endl;
}
#ifdef SINGLEPRECISION
naggpuRandNormalA_sp(N2, NAGGPURANDORDER_CONSISTENT, mu, sigma,
                    d_buff + N/2, &tune, 0, &comm, &error);

```

```

#else
    naggpuRandNormalA(N2, NAGGPURANDORDER_CONSISTENT, mu, sigma,
                     d_buff + N/2, &tune, 0, &comm, &error);
#endif
    checkNagError(&error);

    delete[] tune.mtAGen;

    // Print out the tuning parameters used
    cout << "Tuning Parameters used: " << endl;
    printNagTuningParamsUsed(&comm);

    // Copy back values from the GPU for printing
    cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
                        cudaMemcpyDeviceToHost);
    checkCudaError(cuError);

    // Print random numbers
    cout << "The first 50 GPU random numbers:" << endl;
    cout.setf(ios::fixed, ios::floatfield);
    cout.precision(3);
    for(int row = 0; row < 10; row++)
    {
        for(int col = 0; col < 5; col++)
        {
            cout << h_buff[row*10 + col] << "\t";
        }
        cout << endl;
    }

    // Call cleanup for the NAG routine
    naggpuRandCleanupA(&comm, &error);
    checkNagError(&error);

    // Free CPU and GPU memory
    delete[] h_buff;
    if (d_buff)
    {
        cuError = cudaFree(d_buff);
        checkCudaError(cuError);
    }

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void printNagTuningParamsUsed(NagGpuRandComm *comm)
{
    cout << "  comm.tuneOrigin = ";

    switch (comm->tuneOrigin)
    {

```

```

    case NAGGPUTUNEORIGIN_NA:
        cout << "NAGGPUTUNEORIGIN_NA";
        break;
    case NAGGPUTUNEORIGIN_DEFAULT:
        cout << "NAGGPUTUNEORIGIN_DEFAULT";
        break;
    case NAGGPUTUNEORIGIN_USER:
        cout << "NAGGPUTUNEORIGIN_USER";
        break;
    case NAGGPUTUNEORIGIN_AUTO:
        cout << "NAGGPUTUNEORIGIN_AUTO";
        break;
    default:
        cout << "Unrecognised tuneOrigin";
    }
    cout << endl;

    cout << "  comm.tuneParamsUsed->mtANumBlks = ";
    cout << comm->tuneParamsUsed->mtANumBlks << endl;

    for(int i = 0; i < comm->tuneParamsUsed->mtANumBlks; i++)
    {
        cout << "  comm.tuneParamsUsed->mtAGen[" << i << "] = ";
        cout << comm->tuneParamsUsed->mtAGen[i] << endl;
    }

    cout << endl;
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuRandNormalA\_sp

Initialising generator...

Generate with default tuning parameters...

Tuning Parameters used:

```

comm.tuneOrigin = NAGGPUTUNEORIGIN_DEFAULT
comm.tuneParamsUsed->mtANumBlks = 18
comm.tuneParamsUsed->mtAGen[0] = 15000
comm.tuneParamsUsed->mtAGen[1] = 28528
comm.tuneParamsUsed->mtAGen[2] = 28528
comm.tuneParamsUsed->mtAGen[3] = 28528
comm.tuneParamsUsed->mtAGen[4] = 28528
comm.tuneParamsUsed->mtAGen[5] = 28528
comm.tuneParamsUsed->mtAGen[6] = 28528
comm.tuneParamsUsed->mtAGen[7] = 28528
comm.tuneParamsUsed->mtAGen[8] = 28528
comm.tuneParamsUsed->mtAGen[9] = 28528
comm.tuneParamsUsed->mtAGen[10] = 28528
comm.tuneParamsUsed->mtAGen[11] = 28528
comm.tuneParamsUsed->mtAGen[12] = 28528
comm.tuneParamsUsed->mtAGen[13] = 28528
comm.tuneParamsUsed->mtAGen[14] = 28528
comm.tuneParamsUsed->mtAGen[15] = 28528
comm.tuneParamsUsed->mtAGen[16] = 28528
comm.tuneParamsUsed->mtAGen[17] = 28552

```

Generate with user supplied tuning parameters...

Tuning Parameters supplied:

```
tune.mtANumBlks = 20
tune.mtAGen[0] = 25000
tune.mtAGen[1] = 25000
tune.mtAGen[2] = 25000
tune.mtAGen[3] = 25000
tune.mtAGen[4] = 25000
tune.mtAGen[5] = 25000
tune.mtAGen[6] = 25000
tune.mtAGen[7] = 25000
tune.mtAGen[8] = 25000
tune.mtAGen[9] = 25000
tune.mtAGen[10] = 25000
tune.mtAGen[11] = 25000
tune.mtAGen[12] = 25000
tune.mtAGen[13] = 25000
tune.mtAGen[14] = 25000
tune.mtAGen[15] = 25000
tune.mtAGen[16] = 25000
tune.mtAGen[17] = 25000
tune.mtAGen[18] = 25000
tune.mtAGen[19] = 25001
```

Tuning Parameters used:

```
comm.tuneOrigin = NAGGPOTUNEORIGIN_USER
comm.tuneParamsUsed->mtANumBlks = 20
comm.tuneParamsUsed->mtAGen[0] = 25000
comm.tuneParamsUsed->mtAGen[1] = 25000
comm.tuneParamsUsed->mtAGen[2] = 25000
comm.tuneParamsUsed->mtAGen[3] = 25000
comm.tuneParamsUsed->mtAGen[4] = 25000
comm.tuneParamsUsed->mtAGen[5] = 25000
comm.tuneParamsUsed->mtAGen[6] = 25000
comm.tuneParamsUsed->mtAGen[7] = 25000
comm.tuneParamsUsed->mtAGen[8] = 25000
comm.tuneParamsUsed->mtAGen[9] = 25000
comm.tuneParamsUsed->mtAGen[10] = 25000
comm.tuneParamsUsed->mtAGen[11] = 25000
comm.tuneParamsUsed->mtAGen[12] = 25000
comm.tuneParamsUsed->mtAGen[13] = 25000
comm.tuneParamsUsed->mtAGen[14] = 25000
comm.tuneParamsUsed->mtAGen[15] = 25000
comm.tuneParamsUsed->mtAGen[16] = 25000
comm.tuneParamsUsed->mtAGen[17] = 25000
comm.tuneParamsUsed->mtAGen[18] = 25000
comm.tuneParamsUsed->mtAGen[19] = 25001
```

The first 50 GPU random numbers:

```
0.300 -0.090 -1.365 0.532 -0.478
1.442 -0.657 -0.683 -0.370 0.298
1.323 0.447 0.226 -1.014 -0.255
0.045 -0.496 -0.693 -0.480 -0.360
0.519 1.637 0.333 0.148 0.207
-3.021 0.040 1.690 -0.718 -0.160
0.645 -0.865 -1.703 1.592 -0.573
0.406 0.276 0.222 1.039 0.295
0.130 1.745 0.592 0.797 0.505
-0.271 -0.619 -2.377 -0.023 0.258
```



# NAG Numerical Routines for GPUs Function Document

## naggpuRandUniformA

### 1 Purpose

**naggpuRandUniformA** generates  $n$  values  $X_i$  from a uniform distribution over the interval  $[a, b]$  for specified constants  $a$  and  $b$ .

The initialization function `naggpuRandInitA` must be called prior to the first call to **naggpuRandUniformA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function `naggpuRandCleanupA` must be called to free allocated system resources.

**Note:** To obtain the same values from **naggpuRandUniformA** as from the function `nag_gpu_mrg32-k3a_uniform(nb, nt, np, d_P)` in release 0.3 of the library, please see Section 2.1.1.1 in the Random Number Generators Chapter Introduction.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuRandUniformA_sp(int n, NagGpuRandOrder order, float a, float b,
    float *d_buff, const NagGpuRandTune *tune, cudaStream_t cstream,
    NagGpuRandComm *comm, NagGpuError *error)

extern "C"
cudaError_t naggpuRandUniformA(int n, NagGpuRandOrder order, double a, double b,
    double *d_buff, const NagGpuRandTune *tune, cudaStream_t cstream,
    NagGpuRandComm *comm, NagGpuError *error)
```

### 3 Description

If  $a = 0$  and  $b = 1$ , this function returns the next  $n$  values  $Y_i$  from a uniform  $[0, 1]$  generator. For other values of  $a$  and  $b$ , the function applies the transformation

$$X_i = a + (b - a)Y_i$$

to produce random numbers from the interval  $[a, b]$ .

#### 3.1 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

## 5 Arguments

- 1: **n** – int *Input*  
*On entry:* the number of random values to be generated.  
*Constraint:*  $n \geq 1$ .
- 2: **order** – NagGpuRandOrder *Input*  
*On entry:* the ordering to be observed by the underlying GPU generator:  
order = NAGGPURANDORDER\_OPTIMAL  
order = NAGGPURANDORDER\_CONSISTENT  
See NagGpuRandOrder for further details.  
*Constraint:*  
order = NAGGPURANDORDER\_OPTIMAL or NAGGPURANDORDER\_CONSISTENT.
- 3: **a** – float *Input*  
4: **a** – double *Input*  
This parameter has type float or double depending on whether the single or double precision version of this function is called.  
*On entry:* The lower bound for the uniform random values.
- 5: **b** – float *Input*  
6: **b** – double *Input*  
This parameter has type float or double depending on whether the single or double precision version of this function is called.  
*On entry:* The upper bound for the uniform random values.  
*Constraint:*  $b > a$ .
- 7: **d\_buff[n]** – float \* *Output*  
8: **d\_buff[n]** – double \* *Output*  
This parameter has type float or double depending on whether the single or double precision version of this function is called.  
This buffer must reside in the GPU memory space.  
*On exit:* the **n** pseudorandom numbers from the specified distribution. The output tuning structure `comm` → `tuneParamsUsed` will contain the parameters used to launch the kernel. If `order = NAGGPURANDORDER_OPTIMAL`, these parameters may determine the output ordering (see NagGpuRandTune for details).
- 9: **tune** – const NagGpuRandTune \* *Input*  
This parameter is optional and may be set to NULL.  
*On entry:* if specified, points to a NagGpuRandTune structure containing launch parameters for the selected GPU kernel. Upon a successful return from this function, the relevant data will be copied to the output tuning structure `comm` → `tuneParamsUsed`. Please see NagGpuRandTune for additional information about performance tuning.
- 10: **estream** – cudaStream\_t *Input*  
*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.

- 11: **comm** – NagGpuRandComm \* *Communication Data*

NagGpuRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to naggpuRandCleanupA to free allocated system resources.

Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through **comm**. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see NagGpuRandTune for details on performance tuning, and consult the NagGpuRandComm documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling naggpuRandCleanupA.

- 12: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **order** does not specify a valid ordering. See NagGpuRandOrder for permitted values.

error → code = 112

*On entry:* **d\_buff** is NULL.

error → code = 113

*On entry:*  $b \leq a$

error → code = 200

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details), order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and the kernel launch requires too much shared memory. Try reducing the value of tune → mrgConAThdsPerBlk.

error → code = 250

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_OPTIMAL, tune ≠ NULL and  
 tune → mrgOptAThdsPerBlk is out of bounds. See NagGpuRandTune for further details.

error → code = 251

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_OPTIMAL, tune ≠ NULL and  
 tune → mrgOptAPtsPerThd is out of bounds. See NagGpuRandTune for further details.

error → code = 252

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and  
 tune → mrgConAThdsPerBlk is out of bounds. See NagGpuRandTune for further details.

error → code = 253

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and  
 tune → mrgConAThdsPerBlk is not divisible by  $W$  where  $W = 16$  on devices of compute  
 capability 1.3 or lower and  $W = 32$  otherwise. See NagGpuRandTune for further details.

error → code = 254

*On entry:* the MRG32k3a base generator is selected (see naggpuRandInitA for details),  
 order = NAGGPURANDORDER\_CONSISTENT, tune ≠ NULL and  
 tune → mrgConANumLoops is out of bounds. See NagGpuRandTune for further details.

error → code = 300

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and tune → mtANumBlks is out of bounds. See NagGpuRandTune for further  
 details.

error → code = 301

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and tune → mtAGen is NULL.

error → code = 302

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and tune → mtAGen contains negative entries. See NagGpuRandTune for  
 further details.

error → code = 303

*On entry:* the MT19937 base generator is selected (see naggpuRandInitA for details),  
 tune ≠ NULL and the entries in tune → mtAGen do not sum to  $n$ . See NagGpuRandTune  
 for further details.

## 7 Example

This example program uses **naggpuRandUniformA** to print 50 pseudorandom numbers from a uniform distribution using the MRG32k3a generator. For usage of the MT19937 generator, including tuning aspects, please see the example program for naggpuRandNormalA.

## 7.1 Program Text

```

/*
 * Example Program: naggpu_rand_uniformA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void printNagTuningParamsUsed(NagGpuRandComm *comm);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for generated GPU random numbers
    FP *h_buff = 0;
    // device (GPU) storage for generated random numbers
    FP *d_buff = 0;

    // total number of points to generate
    int N = 1000000;

    // seed variables
    const int seed_length = 6;
    unsigned int seed[seed_length];

    // skip ahead variables
    int a1, a2, b1, b2;
    long long c;

    // distribution parameters
    FP a = 0.0;
    FP b = 1.0;

    // NAG GPU structures
    NagGpuRandComm comm;
    NagGpuRandTune tune;
    NagGpuError error;

    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpuRandUniformA";
    if (sizeof(FP)==sizeof(float)) cout << "_sp";
    cout << endl << endl;

    // Allocate CPU and GPU memory
    h_buff = new FP[N];
    cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
    checkCudaError(cuError);

    // Initialise the generator only once

```

```

cout << "Initialising generator..." << endl << endl;

// arbitrary seed and skip ahead parameters
for (int i = 0; i < seed_length; i++) seed[i] = i;
a1 = 14;
b1 = 34;
a2 = 2;
b2 = 21;
c = 123;

naggpuRandInitA(NAGGPURANDGEN_MRG32K3A, a1, b1, a2, b2,
                c, seed, &comm, &error);
checkNagError(&error);

// Generate N/2 numbers using default tuning parameters
cout << "Generate with default tuning parameters..." << endl;
#ifdef SINGLEPRECISION
    naggpuRandUniformA_sp(N/2, NAGGPURANDORDER_CONSISTENT, a, b,
                          d_buff, NULL, 0, &comm, &error);
#else
    naggpuRandUniformA(N/2, NAGGPURANDORDER_CONSISTENT, a, b,
                       d_buff, NULL, 0, &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Generate (N- N/2) numbers using specified tuning parameters
cout << "Generate with user supplied tuning parameters..." << endl;
tune.mrgConATHdsPerBlk = 96;
tune.mrgConANumLoops = 30;
cout << "Tuning Parameters supplied: " << endl;
cout << "    tune.mrgConATHdsPerBlk = " << tune.mrgConATHdsPerBlk
    << endl;
cout << "    tune.mrgConANumLoops = " << tune.mrgConANumLoops
    << endl;
#ifdef SINGLEPRECISION
    naggpuRandUniformA_sp(N - N/2, NAGGPURANDORDER_CONSISTENT, a, b,
                          d_buff + N/2, &tune, 0, &comm, &error);
#else
    naggpuRandUniformA(N - N/2, NAGGPURANDORDER_CONSISTENT, a, b,
                       d_buff + N/2, &tune, 0, &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Copy back values from the GPU for printing
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The first 50 GPU random numbers:" << endl;
cout.setf(ios::fixed, ios::floatfield);
cout.precision(3);
for (int row = 0; row < 10; row++)
{
    for (int col = 0; col < 5; col++)
    {
        cout << h_buff[row*10 + col] << "\t";
    }
}

```

```

        cout << endl;
    }

    // Call cleanup for the NAG routine
    naggpuRandCleanupA(&comm, &error);
    checkNagError(&error);

    // Free CPU and GPU memory
    delete[] h_buff;
    if (d_buff)
    {
        cuError = cudaFree(d_buff);
        checkCudaError(cuError);
    }

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void printNagTuningParamsUsed(NagGpuRandComm *comm)
{
    cout << "  comm.tuneOrigin = ";

    switch (comm->tuneOrigin)
    {
        case NAGGPPUTUNEORIGIN_NA:
            cout << "NAGGPPUTUNEORIGIN_NA";
            break;
        case NAGGPPUTUNEORIGIN_DEFAULT:
            cout << "NAGGPPUTUNEORIGIN_DEFAULT";
            break;
        case NAGGPPUTUNEORIGIN_USER:
            cout << "NAGGPPUTUNEORIGIN_USER";
            break;
        case NAGGPPUTUNEORIGIN_AUTO:
            cout << "NAGGPPUTUNEORIGIN_AUTO";
            break;
        default:
            cout << "Unrecognised tuneOrigin";
    }
    cout << endl;

    cout << "  comm.tuneParamsUsed->mrgConAThdsPerBlk = ";
    cout << comm->tuneParamsUsed->mrgConAThdsPerBlk << endl;
    cout << "  comm.tuneParamsUsed->mrgConANumLoops = ";
    cout << comm->tuneParamsUsed->mrgConANumLoops << endl;

    cout << endl;
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)

```

```
{
    cout << cudaGetErrorString(cuError) << endl;
    exit(1);
}
}
```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuRandUniformA\_sp

Initialising generator...

Generate with default tuning parameters...

Tuning Parameters used:

```
comm.tuneOrigin = NAGGPUTUNEORIGIN_DEFAULT
comm.tuneParamsUsed->mrgConAThdsPerBlk = 32
comm.tuneParamsUsed->mrgConANumLoops = 8
```

Generate with user supplied tuning parameters...

Tuning Parameters supplied:

```
tune.mrgConAThdsPerBlk = 96
tune.mrgConANumLoops = 30
```

Tuning Parameters used:

```
comm.tuneOrigin = NAGGPUTUNEORIGIN_USER
comm.tuneParamsUsed->mrgConAThdsPerBlk = 96
comm.tuneParamsUsed->mrgConANumLoops = 30
```

The first 50 GPU random numbers:

```
0.357 0.958 0.262 0.649 0.703
0.240 0.515 0.307 0.103 0.732
0.982 0.288 0.511 0.895 0.262
0.704 0.398 0.634 0.518 0.559
0.390 0.790 0.670 0.703 0.603
0.206 0.645 0.111 0.595 0.938
0.761 0.896 0.069 0.551 0.405
0.484 0.652 0.148 0.298 0.712
0.860 0.133 0.371 0.706 0.005
0.377 0.227 0.074 0.861 0.195
```

---

# NAG Numerical Routines for GPUs Function Document

## naggpuRandCleanupA

### 1 Purpose

**naggpuRandCleanupA** frees system resources that were allocated by a previous call to **naggpuRandInitA**.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuRandCleanupA(NagGpuRandComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- 1: **comm** – NagGpuRandComm \* *Communication Data*  
*On entry:* the pointer that was passed to a previous call to **naggpuRandInitA**.
- 2: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

### 6 Error Indicators and Warnings

`error → code = 1`

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error → code = 2`

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

## **7 Example**

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpuRandUniformA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuQuasiRandInitA

### 1 Purpose

**naggpuQuasiRandInitA** initializes a GPU quasi-random number generator. This function must be called before any call to the GPU generator functions (such as **naggpuQuasiRandUniformA**) and must ultimately be followed by a call to the cleanup function **naggpuQuasiRandCleanupA** to release system resources.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuQuasiRandInitA(NagGpuQuasiGen genid, NagGpuScramTypes stype,
    int dim, int offset, NagCPURandComm *pseudoComm, NagGpuQuasiRandComm *comm,
    NagGpuError *error)
```

### 3 Description

Low discrepancy (quasi-random) sequences are used in numerical integration, simulation and optimization. Like pseudorandom numbers they are uniformly distributed, but they are not statistically independent. Quasi-random sequences are designed to give a more even distribution in multidimensional space (uniformity), and are often more efficient than pseudorandom numbers in multidimensional Monte Carlo methods.

Let  $x^1, x^2, \dots, x^N$  be a sequence of  $d$ -dimensional points in the unit cube  $I^d = [0, 1]^d$ . Let  $G$  be a subset of  $I^d$  and define the counting function  $S_N(G)$  as the number of  $d$ -dimensional points  $x^i \in G$ . For each point  $x = (x_1, x_2, \dots, x_d) \in I^d$ , let  $G_x$  be the rectangular  $d$ -dimensional region

$$G_x = [0, x_1) \times [0, x_2) \times \dots \times [0, x_d)$$

with volume  $x_1 \cdot x_2 \cdot \dots \cdot x_d = \prod_{i=1}^d x_i$ . Then one measure of the uniformity of the points  $x^1, x^2, \dots, x^N$  is the so-called star discrepancy:

$$D_N^*(x^1, x^2, \dots, x^N) = \sup_{x \in I^d} \left| S_N(G_x) - N \prod_{i=1}^d x_i \right|$$

which satisfies the inequality

$$D_N^*(x^1, x^2, \dots, x^N) \leq C_d (\log N)^d + O((\log N)^{d-1}) \quad \text{for all } N \geq 2.$$

The principal aim in the construction of low-discrepancy sequences is to find sequences of points in  $I^d$  with a bound of this form where the constant  $C_d$  is as small as possible.

The type of low-discrepancy sequence generated by **naggpuQuasiRandInitA** depends on the value of **genid**, and the sequence can optionally be scrambled through the parameter **stype**. See **NagGpuQuasiGen** and **NagGpuScramTypes** respectively for further information.

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or **cudaSuccess** if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

None.

## 5 Arguments

- 1: **genid** – NagGpuQuasiGen *Input*  
*On entry:* the type of generator to be used:  
genid = NAGGPUQUASIGEN\_SOBOL  
*Constraint:* genid = NAGGPUQUASIGEN\_SOBOL.
- 2: **stype** – NagGpuScramTypes *Input*  
*On entry:* the type of scrambling to be used:  
stype = NAGGPUSCRAMTYPES\_NONE  
stype = NAGGPUSCRAMTYPES\_OWEN  
stype = NAGGPUSCRAMTYPES\_FAURE\_TEZUKA  
stype = NAGGPUSCRAMTYPES\_OWEN\_FAURE\_TEZUKA  
Please see NagGpuScramTypes for some of the benefits of scrambling and details about each of available scrambling types.  
*Constraint:* stype = NAGGPUSCRAMTYPES\_NONE or  
NAGGPUSCRAMTYPES\_OWEN or  
NAGGPUSCRAMTYPES\_FAURE\_TEZUKA or  
NAGGPUSCRAMTYPES\_OWEN\_FAURE\_TEZUKA
- 3: **dim** – int *Input*  
*On entry:* the dimension of the quasi-random sequence.  
*Constraint:*  $1 \leq \text{dim} \leq 50000$ .
- 4: **offset** – int *Input*  
*On entry:* the offset into the sequence at which to start generating.  
*Constraint:* offset  $\geq 0$ .
- 5: **pseudoComm** – NagCpurandComm \* *Input*  
*On entry:* a pointer to a NagCpurandComm structure which has already been initialized by the function nagCpurandInitA.  
*Constraint:* **pseudoComm** must be initialized before being passed to this function .
- 6: **comm** – NagGpuQuasiRandComm \* *Communication Data*  
NagGpuQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator functions (such as naggpuQuasiRandUniformA). Once all required points have been obtained, **comm** must be passed to naggpuQuasiRandCleanupA to free allocated system resources.
- 7: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of error  $\rightarrow$  code which should be inspected after each call to this function. If error  $\rightarrow$  code = 0 then no error occurred. If error  $\rightarrow$  code  $\neq 0$  then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 110

*On entry:* **genid** does not specify a valid quasi-random number generator. See `NagGpuQuasiGen` for permitted values.

error → code = 111

*On entry:* **stype** does not specify a valid scrambling type. See `NagGpuScramTypes` for permitted values.

error → code = 112

*On entry:* the value of **dim** does not satisfy the constraint listed above.

error → code = 113

*On entry:* the value of **offset** is negative.

error → code = 114

*On entry:* the value of **pseudoComm** is NULL.

error → code = 115

*On entry:* the pseudorandom generator `nagCPURandUniformA` returned an error when called by this function: **pseudoComm** is not initialized, or the internal state of **pseudoComm** is corrupted.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for `naggpuQuasiRandUniformA`.

---



# NAG Numerical Routines for GPUs Function Document

## naggpuQuasiRandExpA

### 1 Purpose

**naggpuQuasiRandExpA** generates  $n$  points  $x_i$  from a quasi-random exponential distribution with mean  $\lambda$ . The initialization function **naggpuQuasiRandInitA** must be called prior to the first call to **naggpuQuasiRandExpA**. Thereafter, this function may be called repeatedly to generate additional sets of quasi-random points. Once all desired points have been obtained, the function **naggpuQuasiRandCleanupA** must be called to free allocated system resources.

**Note:** Concerns were raised about the set of Sobol' direction numbers that were used in release 0.3 of the NAG Numerical Routines for GPUs. These concerns have been addressed by an amended set of direction numbers in Joe and Kuo (2008) which are used in this release. Consequently, the higher dimensions of this Sobol' generator may not match the higher dimensions of the generator in release 0.3 since the direction numbers are different.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuQuasiRandExpA_sp(int n, NagGpuQuasiOrient orient,
    float lambda, float *d_buff, const NagGpuQuasiRandTune *tune,
    cudaStream_t cstream, NagGpuQuasiRandComm *comm, NagGpuError *error)

extern "C"
cudaError_t naggpuQuasiRandExpA(int n, NagGpuQuasiOrient orient, double lambda,
    double *d_buff, const NagGpuQuasiRandTune *tune, cudaStream_t cstream,
    NagGpuQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

Quasi-random sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling **naggpuQuasiRandInitA** to initialize the generator. Below we will consider a  $d$ -dimensional quasi-random sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

The exponential distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\lambda} e^{-x/\lambda} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\lambda > 0$ . This function returns the next  $n$  points  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  for  $j = 1, 2, \dots, n$  where

$$x_i^j = -\lambda \ln(y_i^j + 2^{-32})$$

for each  $i = 1, 2, \dots, d$ . Here  $y^j = (y_1^j, y_2^j, \dots, y_d^j) \in [0, 1)^d$  are the next  $n$  points from the quasi-random generator.

#### 3.1 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this

direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

## 5 Arguments

1: **n** – int *Input*

*On entry:* the number of quasi-random points to be generated.

*Constraint:*  $n \geq 1$ .

2: **orient** – NagGpuQuasiOrient *Input*

*On entry:* specifies the orientation with which the generator will store the output points. Currently only **NAGGPUQUASIORIENT\_DIMVALS\_SCATT** is supported. See NagGpuQuasiOrient for further details on output orientation.

*Constraint:* orient = NAGGPUQUASIORIENT\_DIMVALS\_SCATT.

3: **lambda** – float *Input*

4: **lambda** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The mean,  $\lambda$ , of the exponential distribution.

*Constraint:* lambda > 0.

5: **d\_buff[n × d]** – float \* *Output*

6: **d\_buff[n × d]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

This buffer must reside in the GPU memory space.

The value  $d$  is the dimension **dim** of the sequence as specified to the initialization function `naggpuQuasiRandInitA`.

*On exit:* the **n** quasi-random points from the specified distribution. For a given point, the individual dimension values **are not** stored consecutively in memory. The  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location `d_buff[(i - 1) * n + j]` for every  $0 \leq j < n$  and  $1 \leq i \leq d$ . In other words, the first **n** values correspond to dimension 1, the second **n** to dimension 2, and so on.

7: **tune** – const NagGpuQuasiRandTune \* *Input*

This parameter is optional and may be set to NULL.

*On entry:* if specified, points to a NagGpuQuasiRandTune structure containing launch parameters for the selected GPU kernel. Please see NagGpuQuasiRandTune for additional information about performance tuning.

- 8: **istream** – cudaStream\_t *Input*  
*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.
- 9: **comm** – NagGpuQuasiRandComm \* *Communication Data*  
NagGpuQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to naggpuQuasiRandCleanupA to free allocated system resources.  
Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through **comm**. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see NagGpuQuasiRandTune for details on performance tuning, and consult the NagGpuQuasiRandComm documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling naggpuQuasiRandCleanupA.
- 10: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **orient** does not satisfy the constraint listed above.

error → code = 112

*On entry:* **d\_buff** is NULL.

error → code = 114

*On entry:*  $\lambda \leq 0$ .

error → code = 250

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune ≠ NULL and tune → sblAThdsPerBlk is out of bounds. See NagGpuQuasiRandTune for further details.

error → code = 251

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune ≠ NULL and tune → sblAThdsPerBlk is not a power of two.

error → code = 252

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune ≠ NULL and tune → sblABlksPerDim is out of bounds. See NagGpuQuasiRandTune for further details.

error → code = 253

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune ≠ NULL and tune → sblABlksPerDim is not a power of two.

## 7 Example

This example program uses **naggpuQuasiRandExpA** to print 5 quasi-random numbers of dimension 10 from an exponential distribution. The first point in the sequence is skipped and generation starts at the second point.

### 7.1 Program Text

```

/*
 * Example Program: naggpuQuasiRandExpA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void printNagTuningParamsUsed(NagGpuQuasiRandComm *comm);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for generated GPU random numbers
    FP *h_buff = 0;
    // device (GPU) storage for generated random numbers
    FP *d_buff = 0;

    // number of points to generate
    int n = 100000;
    int dim = 100;

    int offset = 1;

```

```

unsigned int pseudoSeed[] = {1, 2, 3, 4, 5, 6};

// distribution parameter
FP lambda = 1.0;

// NAG GPU structures
NagGpuQuasiRandComm comm;
NagGpuQuasiRandTune tune;
NagCPURandComm pseudoComm;
NagGpuError error;
NagGpuQuasiOrient orient = NAGGPUQUASIORIENT_DIMVALS_SCATT;

cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpuQuasiRandExpA";
if (sizeof(FP)==sizeof(float)) cout << "_sp";
cout << endl << endl;

// Allocate CPU and GPU memory
h_buff = new FP[n*dim];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*n*dim);
checkCudaError(cuError);

// Initialise the CPU pseudo-random generator
nagCPURandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, pseudoSeed,
                &pseudoComm, &error);
checkNagError(&error);

// Initialise the generator only once
cout << "Initialising generator..." << endl << endl;

naggpuQuasiRandInitA(NAGGPUQUASIGEN_SOBOL, NAGGPUSCRAMTYPES_NONE,
                    dim, offset, &pseudoComm, &comm, &error);
checkNagError(&error);

// Generate N numbers using default tuning parameters
cout << "Generate with default tuning parameters..." << endl;
#ifdef SINGLEPRECISION
    naggpuQuasiRandExpA_sp(n, orient, lambda, d_buff, NULL, 0, &comm, &error);
#else
    naggpuQuasiRandExpA(n, orient, lambda, d_buff, NULL, 0, &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Copy back values from the GPU for printing
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*n*dim,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The 5 GPU numbers from dimensions 1 to 10:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(3);
for(int d = 0; d < 10; d++)
{
    cout << "dim" << d+1 << "\t";
    for(int i = 0; i < 5; i++)

```

```

        {
            cout << h_buff[n*d + i] << "\t";
        }
        cout << endl;
    }
    cout << endl;

    // Generate n numbers using specified tuning parameters
    cout << "Generate with user supplied tuning parameters..." << endl;
    tune.sblAThdsPerBlk = 32;
    tune.sblABlksPerDim = 8;
    cout << "Tuning Parameters supplied: " << endl;
    cout << "  tune.sblAThdsPerBlk = " << tune.sblAThdsPerBlk << endl;
    cout << "  tune.sblABlksPerDim = " << tune.sblABlksPerDim << endl;
#ifdef SINGLEPRECISION
    naggpuQuasiRandExpA_sp(n, orient, lambda, d_buff, &tune, 0, &comm, &error);
#else
    naggpuQuasiRandExpA(n, orient, lambda, d_buff, &tune, 0, &comm, &error);
#endif
    checkNagError(&error);

    // Print out the tuning parameters used
    cout << "Tuning Parameters used: " << endl;
    printNagTuningParamsUsed(&comm);

    // Call cleanup for the NAG routine
    naggpuQuasiRandCleanupA(&comm, &error);
    checkNagError(&error);

    // Free CPU and GPU memory
    delete[] h_buff;
    if (d_buff)
    {
        cuError = cudaFree(d_buff);
        checkCudaError(cuError);
    }

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void printNagTuningParamsUsed(NagGpuQuasiRandComm *comm)
{
    cout << "  comm.tuneOrigin = ";

    switch (comm->tuneOrigin)
    {
        case NAGGPPUTUNEORIGIN_NA:
            cout << "NAGGPPUTUNEORIGIN_NA";
            break;
        case NAGGPPUTUNEORIGIN_DEFAULT:
            cout << "NAGGPPUTUNEORIGIN_DEFAULT";
            break;
    }
}

```

```

    case NAGGPUNEORIGIN_USER:
        cout << "NAGGPUNEORIGIN_USER";
        break;
    case NAGGPUNEORIGIN_AUTO:
        cout << "NAGGPUNEORIGIN_AUTO";
        break;
    default:
        cout << "Unrecognised tuneOrigin";
    }
    cout << endl;

    cout << "  comm.tuneParamsUsed->sblAThdsPerBlk = ";
    cout << comm->tuneParamsUsed->sblAThdsPerBlk << endl;
    cout << "  comm.tuneParamsUsed->sblABlksPerDim = ";
    cout << comm->tuneParamsUsed->sblABlksPerDim << endl;

    cout << endl;
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuQuasiRandExpA\_sp

Initialising generator...

Generate with default tuning parameters...

```

Tuning Parameters used:
  comm.tuneOrigin = NAGGPUNEORIGIN_DEFAULT
  comm.tuneParamsUsed->sblAThdsPerBlk = 64
  comm.tuneParamsUsed->sblABlksPerDim = 16

```

The 5 GPU numbers from dimensions 1 to 10:

```

dim1 0.693 0.288 1.386 0.981 0.134
dim2 0.693 1.386 0.288 0.981 0.134
dim3 0.693 1.386 0.288 0.470 2.079
dim4 0.693 1.386 0.288 0.134 0.981
dim5 0.693 0.288 1.386 0.981 0.134
dim6 0.693 0.288 1.386 2.079 0.470
dim7 0.693 1.386 0.288 0.981 0.134
dim8 0.693 0.288 1.386 0.134 0.981
dim9 0.693 0.288 1.386 0.134 0.981
dim10 0.693 0.288 1.386 0.470 2.079

```

Generate with user supplied tuning parameters...

```

Tuning Parameters supplied:
  tune.sblAThdsPerBlk = 32
  tune.sblABlksPerDim = 8
Tuning Parameters used:
  comm.tuneOrigin = NAGGPUNEORIGIN_USER
  comm.tuneParamsUsed->sblAThdsPerBlk = 32
  comm.tuneParamsUsed->sblABlksPerDim = 8

```



# NAG Numerical Routines for GPUs Function Document

## naggpuQuasiRandNormalA

### 1 Purpose

**naggpuQuasiRandNormalA** generates  $N$  points  $x_i$  from a quasi-random Normal distribution with mean  $\mu$  and variance  $\sigma^2$ .

The initialization function **naggpuQuasiRandInitA** must be called prior to the first call to **naggpuQuasiRandNormalA**. Thereafter, this function may be called repeatedly to generate additional sets of quasi-random points. Once all desired points have been obtained, the function **naggpuQuasiRandCleanupA** must be called to free allocated system resources.

**Note:** Concerns were raised about the set of Sobol' direction numbers that were used in release 0.3 of the NAG Numerical Routines for GPUs. These concerns have been addressed by an amended set of direction numbers in Joe and Kuo (2008) which are used in this release. Consequently, the higher dimensions of this Sobol' generator may not match the higher dimensions of the generator in release 0.3 since the direction numbers are different.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuQuasiRandNormalA_sp(int n, NagGpuQuasiOrient orient, float mu,
    float sigma, float *d_buff, const NagGpuQuasiRandTune *tune,
    cudaStream_t cstream, NagGpuQuasiRandComm *comm, NagGpuError *error)

extern "C"
cudaError_t naggpuQuasiRandNormalA(int n, NagGpuQuasiOrient orient, double mu,
    double sigma, double *d_buff, const NagGpuQuasiRandTune *tune,
    cudaStream_t cstream, NagGpuQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

Quasi-random sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling **naggpuQuasiRandInitA** to initialize the generator. Below we will consider a  $d$ -dimensional quasi-random sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

The Normal distribution has probability density function given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

where  $\sigma > 0$  and  $\mu \in \mathbb{R}$ . This function returns the next  $n$  points  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  for  $j = 1, 2, \dots, n$  where

$$x_i^j = \mu + \sigma\sqrt{2} \operatorname{erfinv}(z_i^j)$$

for each  $i = 1, 2, \dots, d$  and  $\operatorname{erfinv}$  is the inverse error function. Here each  $z^j = (z_1^j, z_2^j, \dots, z_d^j)$  is a low discrepancy point in the interval  $(-1, 1)^d$ .

#### 3.1 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU

code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

## 5 Arguments

1: **n** – int *Input*

*On entry:* the number of quasi-random points to be generated.

*Constraint:*  $n \geq 1$ .

2: **orient** – NagGpuQuasiOrient *Input*

*On entry:* specifies the orientation with which the generator will store the output points. Currently only **NAGGPUQUASIORIENT\_DIMVALS\_SCATT** is supported. See NagGpuQuasiOrient for further details on output orientation.

*Constraint:* `orient = NAGGPUQUASIORIENT_DIMVALS_SCATT`.

3: **mu** – float *Input*

4: **mu** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the mean,  $\mu$ , of the distribution.

5: **sigma** – float *Input*

6: **sigma** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the standard deviation,  $\sigma$ , of the distribution

*Constraint:*  $\sigma > 0$ .

7: **d\_buff[n × d]** – float \* *Output*

8: **d\_buff[n × d]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

This buffer must reside in the GPU memory space.

The value  $d$  is the dimension **dim** of the sequence as specified to the initialization function `naggpuQuasiRandInitA`.

*On exit:* the **n** quasi-random points from the specified distribution. For a given point, the individual dimension values **are not** stored consecutively in memory. The  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location `d_buff[(i - 1) * n + j]` for every  $0 \leq j < n$  and  $1 \leq i \leq d$ . In other words, the first **n** values correspond to dimension 1, the second **n** to dimension 2, and so on.

- 9: **tune** – const NagGpuQuasiRandTune \* *Input*  
 This parameter is optional and may be set to NULL.  
*On entry:* if specified, points to a NagGpuQuasiRandTune structure containing launch parameters for the selected GPU kernel. Please see NagGpuQuasiRandTune for additional information about performance tuning.
- 10: **estream** – cudaStream\_t *Input*  
*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.
- 11: **comm** – NagGpuQuasiRandComm \* *Communication Data*  
 NagGpuQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to naggpuQuasiRandCleanupA to free allocated system resources.  
 Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through **comm**. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see NagGpuQuasiRandTune for details on performance tuning, and consult the NagGpuQuasiRandComm documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling naggpuQuasiRandCleanupA.
- 12: **error** – NagGpuError \* *Error Reporting*  
 This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **orient** does not satisfy the constraint listed above.

error → code = 112

*On entry:* **d\_buff** is NULL.

error → code = 115

*On entry:*  $\sigma \leq 0$ .

error → code = 250

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details),  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{sblAThdsPerBlk}$  is out of bounds. See NagGpuQuasiRandTune for further details.

error → code = 251

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details),  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{sblAThdsPerBlk}$  is not a power of two.

error → code = 252

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details),  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{sblABlksPerDim}$  is out of bounds. See NagGpuQuasiRandTune for further details.

error → code = 253

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details),  $\text{tune} \neq \text{NULL}$  and  $\text{tune} \rightarrow \text{sblABlksPerDim}$  is not a power of two.

## 7 Example

This example program uses **naggpuQuasiRandNormalA** to print 5 quasi-random numbers of dimension 10 from a Normal distribution. The first point in the sequence is skipped and generation starts at the second point.

### 7.1 Program Text

```

/*
 * Example Program: naggpuQuasiRandNormalA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void printNagTuningParamsUsed(NagGpuQuasiRandComm *comm);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for generated GPU random numbers
    FP *h_buff = 0;

```

```

// device (GPU) storage for generated random numbers
FP *d_buff = 0;

// number of points to generate
int n = 100000;
int dim = 100;

int offset = 1;

unsigned int pseudoSeed[] = {1, 2, 3, 4, 5, 6};

// distribution parameters
FP mu = 0.0;
FP sigma = 1.0;

// NAG GPU structures
NagGpuQuasiRandComm comm;
NagGpuQuasiRandTune tune;
NagCpuRandComm pseudoComm;
NagGpuError error;
NagGpuQuasiOrient orient = NAGGPUQUASIORIENT_DIMVALS_SCATT;

cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpuQuasiRandNormalA";
if (sizeof(FP)==sizeof(float)) cout << "_sp";
cout << endl << endl;

// Allocate CPU and GPU memory
h_buff = new FP[n*dim];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*n*dim);
checkCudaError(cuError);

// Initialise the CPU pseudo-random generator
nagCpuRandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, 0, pseudoSeed,
               &pseudoComm, &error);
checkNagError(&error);

// Initialise the generator only once
cout << "Initialising generator..." << endl << endl;

naggpuQuasiRandInitA(NAGGPUQUASIGEN_SOBOL, NAGGPUSCRAMTYPES_NONE,
                    dim, offset, &pseudoComm, &comm, &error);
checkNagError(&error);

// Generate n numbers using default tuning parameters
cout << "Generate with default tuning parameters..." << endl;
#ifdef SINGLEPRECISION
    naggpuQuasiRandNormalA_sp(n, orient, mu, sigma, d_buff, NULL, 0,
                              &comm, &error);
#else
    naggpuQuasiRandNormalA(n, orient, mu, sigma, d_buff, NULL, 0,
                           &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Copy back values from the GPU for printing
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*n*dim,
                    cudaMemcpyDeviceToHost);

```

```

checkCudaError(cuError);

// Print random numbers
cout << "The 5 GPU numbers from dimensions 1 to 10:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(3);
for(int d = 0; d < 10; d++)
{
    cout << "dim" << d+1 << "\t";
    for(int i = 0; i < 5; i++)
    {
        cout << h_buff[n*d + i] << "\t";
    }
    cout << endl;
}
cout << endl;

// Generate n numbers using specified tuning parameters
cout << "Generate with user supplied tuning parameters..." << endl;
tune.sblATHdsPerBlk = 32;
tune.sblABlksPerDim = 8;
cout << "Tuning Parameters supplied: " << endl;
cout << "    tune.sblATHdsPerBlk = " << tune.sblATHdsPerBlk << endl;
cout << "    tune.sblABlksPerDim = " << tune.sblABlksPerDim << endl;
#ifdef SINGLEPRECISION
    naggpuQuasiRandNormalA_sp(n, orient, mu, sigma, d_buff, &tune, 0,
                              &comm, &error);
#else
    naggpuQuasiRandNormalA(n, orient, mu, sigma, d_buff, &tune, 0,
                           &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Call cleanup for the NAG routine
naggpuQuasiRandCleanupA(&comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

```

```

void printNagTuningParamsUsed(NagGpuQuasiRandComm *comm)
{
    cout << "  comm.tuneOrigin = ";

    switch (comm->tuneOrigin)
    {
        case NAGGPPUTUNEORIGIN_NA:
            cout << "NAGGPPUTUNEORIGIN_NA";
            break;
        case NAGGPPUTUNEORIGIN_DEFAULT:
            cout << "NAGGPPUTUNEORIGIN_DEFAULT";
            break;
        case NAGGPPUTUNEORIGIN_USER:
            cout << "NAGGPPUTUNEORIGIN_USER";
            break;
        case NAGGPPUTUNEORIGIN_AUTO:
            cout << "NAGGPPUTUNEORIGIN_AUTO";
            break;
        default:
            cout << "Unrecognised tuneOrigin";
    }
    cout << endl;

    cout << "  comm.tuneParamsUsed->sblAThdsPerBlk = ";
    cout << comm->tuneParamsUsed->sblAThdsPerBlk << endl;
    cout << "  comm.tuneParamsUsed->sblABlksPerDim = ";
    cout << comm->tuneParamsUsed->sblABlksPerDim << endl;

    cout << endl;
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        {
            cout << cudaGetErrorString(cuError) << endl;
            exit(1);
        }
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuQuasiRandNormalA\_sp

Initialising generator...

Generate with default tuning parameters...

Tuning Parameters used:

```

    comm.tuneOrigin = NAGGPPUTUNEORIGIN_DEFAULT
    comm.tuneParamsUsed->sblAThdsPerBlk = 64
    comm.tuneParamsUsed->sblABlksPerDim = 16

```

The 5 GPU numbers from dimensions 1 to 10:

```

dim1 0.000 0.674 -0.674 -0.319 1.150
dim2 0.000 -0.674 0.674 -0.319 1.150
dim3 0.000 -0.674 0.674 0.319 -1.150
dim4 0.000 -0.674 0.674 1.150 -0.319
dim5 0.000 0.674 -0.674 -0.319 1.150
dim6 0.000 0.674 -0.674 -1.150 0.319
dim7 0.000 -0.674 0.674 -0.319 1.150
dim8 0.000 0.674 -0.674 1.150 -0.319
dim9 0.000 0.674 -0.674 1.150 -0.319
dim10 0.000 0.674 -0.674 0.319 -1.150

```

Generate with user supplied tuning parameters...

Tuning Parameters supplied:

tune.sblAThdsPerBlk = 32

tune.sblABlksPerDim = 8

Tuning Parameters used:

comm.tuneOrigin = NAGGPOTUNEORIGIN\_USER

comm.tuneParamsUsed->sblAThdsPerBlk = 32

comm.tuneParamsUsed->sblABlksPerDim = 8

---

# NAG Numerical Routines for GPUs Function Document

## naggpuQuasiRandUniformA

### 1 Purpose

**naggpuQuasiRandUniformA** generates  $n$  points  $x_i$  from a quasi-random uniform distribution over the interval  $[a, b)$  for specified constants  $a$  and  $b$ .

The initialization function `naggpuQuasiRandInitA` must be called prior to the first call to **naggpuQuasiRandUniformA**. Thereafter, this function may be called repeatedly to generate additional sets of quasi-random points. Once all desired points have been obtained, the function `naggpuQuasiRandCleanupA` must be called to free allocated system resources.

**Note:** Concerns were raised about the set of Sobol' direction numbers that were used in release 0.3 of the NAG Numerical Routines for GPUs. These concerns have been addressed by an amended set of direction numbers in Joe and Kuo (2008) which are used in this release. Consequently, the higher dimensions of this Sobol' generator may not match the higher dimensions of the generator in release 0.3 since the direction numbers are different.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuQuasiRandUniformA_sp(int n, NagGpuQuasiOrient orient, float a,
    float b, float *d_buff, const NagGpuQuasiRandTune *tune,
    cudaStream_t cstream, NagGpuQuasiRandComm *comm, NagGpuError *error)

extern "C"
cudaError_t naggpuQuasiRandUniformA(int n, NagGpuQuasiOrient orient, double a,
    double b, double *d_buff, const NagGpuQuasiRandTune *tune,
    cudaStream_t cstream, NagGpuQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

Quasi-random sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling `naggpuQuasiRandInitA` to initialize the generator. Below we will consider a  $d$ -dimensional quasi-random sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

If  $a = 0$  and  $b = 1$ , this function returns the next  $n$  points  $y^j \in [0, 1)^d$  from the quasi-random generator. For other values of  $a$  and  $b$ , the function applies the transformation

$$x_i^j = a + (b - a)y_i^j$$

for each  $i = 1, 2, \dots, d$  to produce quasi-random points  $x^j$  from the interval  $[a, b)^d$  for each  $j = 1, 2, \dots, n$ .

#### 3.1 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

## 5 Arguments

1: **n** – int *Input*

*On entry:* the number of quasi-random points to be generated.

*Constraint:*  $n \geq 1$ .

2: **orient** – NagGpuQuasiOrient *Input*

*On entry:* specifies the orientation with which the generator will store the output points. Currently only **NAGGPUQUASIORIENT\_DIMVALS\_SCATT** is supported. See NagGpuQuasiOrient for further details on output orientation.

*Constraint:* `orient = NAGGPUQUASIORIENT_DIMVALS_SCATT`.

3: **a** – float *Input*

4: **a** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The lower bound for the uniform random values.

5: **b** – float *Input*

6: **b** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The upper bound for the uniform random values.

*Constraint:*  $b > a$ .

7: **d\_buff[n × d]** – float \* *Output*

8: **d\_buff[n × d]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

This buffer must reside in the GPU memory space.

The value  $d$  is the dimension **dim** of the sequence as specified to the initialization function `naggpuQuasiRandInitA`.

*On exit:* the **n** quasi-random points from the specified distribution. For a given point, the individual dimension values **are not** stored consecutively in memory. The  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location `d_buff[(i - 1) * n + j]` for every  $0 \leq j < n$  and  $1 \leq i \leq d$ . In other words, the first **n** values correspond to dimension 1, the second **n** to dimension 2, and so on.

9: **tune** – const NagGpuQuasiRandTune \* *Input*

This parameter is optional and may be set to `NULL`.

*On entry:* if specified, points to a NagGpuQuasiRandTune structure containing launch parameters for the selected GPU kernel. Please see NagGpuQuasiRandTune for additional information about performance tuning.

10: **istream** – cudaStream\_t *Input*

*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.

11: **comm** – NagGpuQuasiRandComm \* *Communication Data*

NagGpuQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to naggpuQuasiRandCleanupA to free allocated system resources.

Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through **comm**. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see NagGpuQuasiRandTune for details on performance tuning, and consult the NagGpuQuasiRandComm documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling naggpuQuasiRandCleanupA.

12: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 3

*On entry:* an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **orient** does not satisfy the constraint listed above.

error → code = 112

*On entry:* **d\_buff** is NULL.

error → code = 113

*On entry:*  $b \leq a$

error → code = 250

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune  $\neq$  NULL and tune → sblAThdsPerBlk is out of bounds. See NagGpuQuasiRandTune for further details.

error → code = 251

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune  $\neq$  NULL and tune → sblAThdsPerBlk is not a power of two.

error → code = 252

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune  $\neq$  NULL and tune → sblABlksPerDim is out of bounds. See NagGpuQuasiRandTune for further details.

error → code = 253

*On entry:* the Sobol' base generator is selected (see naggpuQuasiRandInitA for details), tune  $\neq$  NULL and tune → sblABlksPerDim is not a power of two.

## 7 Example

This example program uses **naggpuQuasiRandUniformA** to print 5 quasi-random numbers of dimension 10 from a uniform distribution. The first point in the sequence is skipped and generation starts at the second point.

### 7.1 Program Text

```

/*
 * Example Program: naggpuQuasiRandUniformA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void printNagTuningParamsUsed(NagGpuQuasiRandComm *comm);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // host (CPU) storage for generated GPU random numbers
    FP *h_buff = 0;
    // device (GPU) storage for generated random numbers
    FP *d_buff = 0;

    // number of points to generate

```

```

int n = 100000;
int dim = 100;

int offset = 1;

unsigned int pseudoSeed[] = {1, 2, 3, 4, 5, 6};

// distribution parameters
FP a = 0.0;
FP b = 1.0;

// NAG GPU structures
NagGpuQuasiRandComm comm;
NagGpuQuasiRandTune tune;
NagCPURandComm pseudoComm;
NagGpuError error;
NagGpuQuasiOrient orient = NAGGPUQUASIORIENT_DIMVALS_SCATT;

cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpuQuasiRandUniformA";
if (sizeof(FP)==sizeof(float)) cout << "_sp";
cout << endl << endl;

// Allocate CPU and GPU memory
h_buff = new FP[n*dim];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*n*dim);
checkCudaError(cuError);

// Initialise the CPU pseudo-random generator
nagCPURandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, 0, pseudoSeed,
                &pseudoComm, &error);
checkNagError(&error);

// Initialise the generator only once
cout << "Initialising generator..." << endl << endl;

naggpuQuasiRandInitA(NAGGPUQUASIGEN_SOBOL, NAGGPUSCRAMTYPES_NONE,
                    dim, offset, &pseudoComm, &comm, &error);
checkNagError(&error);

// Generate n numbers using default tuning parameters
cout << "Generate with default tuning parameters..." << endl;
#ifdef SINGLEPRECISION
    naggpuQuasiRandUniformA_sp(n, orient, a, b, d_buff, NULL, 0,
                              &comm, &error);
#else
    naggpuQuasiRandUniformA(n, orient, a, b, d_buff, NULL, 0,
                            &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Copy back values from the GPU for printing
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*n*dim,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers

```

```

cout << "The 5 GPU numbers from dimensions 1 to 10:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(3);
for(int d = 0; d < 10; d++)
{
    cout << "dim" << d+1 << "\t";
    for(int i = 0; i < 5; i++)
    {
        cout << h_buff[n*d + i] << "\t";
    }
    cout << endl;
}
cout << endl;

// Generate n numbers using specified tuning parameters
cout << "Generate with user supplied tuning parameters..." << endl;
tune.sblATHdsPerBlk = 32;
tune.sblABlksPerDim = 8;
cout << "Tuning Parameters supplied: " << endl;
cout << "  tune.sblATHdsPerBlk = " << tune.sblATHdsPerBlk << endl;
cout << "  tune.sblABlksPerDim = " << tune.sblABlksPerDim << endl;
#ifdef SINGLEPRECISION
    naggpuQuasiRandUniformA_sp(n, orient, a, b, d_buff, &tune, 0,
                               &comm, &error);
#else
    naggpuQuasiRandUniformA(n, orient, a, b, d_buff, &tune, 0,
                             &comm, &error);
#endif
checkNagError(&error);

// Print out the tuning parameters used
cout << "Tuning Parameters used: " << endl;
printNagTuningParamsUsed(&comm);

// Call cleanup for the NAG routine
naggpuQuasiRandCleanupA(&comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void printNagTuningParamsUsed(NagGpuQuasiRandComm *comm)
{
    cout << "  comm.tuneOrigin = ";

```

```

switch (comm->tuneOrigin)
{
case NAGGPUTUNEORIGIN_NA:
    cout << "NAGGPUTUNEORIGIN_NA";
    break;
case NAGGPUTUNEORIGIN_DEFAULT:
    cout << "NAGGPUTUNEORIGIN_DEFAULT";
    break;
case NAGGPUTUNEORIGIN_USER:
    cout << "NAGGPUTUNEORIGIN_USER";
    break;
case NAGGPUTUNEORIGIN_AUTO:
    cout << "NAGGPUTUNEORIGIN_AUTO";
    break;
default:
    cout << "Unrecognised tuneOrigin";
}
cout << endl;

cout << "  comm.tuneParamsUsed->sblAThdsPerBlk = ";
cout << comm->tuneParamsUsed->sblAThdsPerBlk << endl;
cout << "  comm.tuneParamsUsed->sblABlksPerDim = ";
cout << comm->tuneParamsUsed->sblABlksPerDim << endl;

cout << endl;
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuQuasiRandUniformA\_sp

Initialising generator...

Generate with default tuning parameters...

Tuning Parameters used:

```

comm.tuneOrigin = NAGGPUTUNEORIGIN_DEFAULT
comm.tuneParamsUsed->sblAThdsPerBlk = 64
comm.tuneParamsUsed->sblABlksPerDim = 16

```

The 5 GPU numbers from dimensions 1 to 10:

```

dim1 0.500 0.750 0.250 0.375 0.875
dim2 0.500 0.250 0.750 0.375 0.875
dim3 0.500 0.250 0.750 0.625 0.125
dim4 0.500 0.250 0.750 0.875 0.375
dim5 0.500 0.750 0.250 0.375 0.875
dim6 0.500 0.750 0.250 0.125 0.625
dim7 0.500 0.250 0.750 0.375 0.875
dim8 0.500 0.750 0.250 0.875 0.375
dim9 0.500 0.750 0.250 0.875 0.375
dim10 0.500 0.750 0.250 0.625 0.125

```

Generate with user supplied tuning parameters...

Tuning Parameters supplied:

```

tune.sblAThdsPerBlk = 32
tune.sblABlksPerDim = 8

```

Tuning Parameters used:

```
comm.tuneOrigin = NAGGPUNETUNEORIGIN_USER  
comm.tuneParamsUsed->sblAThdsPerBlk = 32  
comm.tuneParamsUsed->sblABlksPerDim = 8
```

---

# NAG Numerical Routines for GPUs Function Document

## naggpuQuasiRandCleanupA

### 1 Purpose

**naggpuQuasiRandCleanupA** frees system resources that were allocated by a previous call to **naggpuQuasiRandInitA**.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuQuasiRandCleanupA(NagGpuQuasiRandComm *comm,
    NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

1: **comm** – NagGpuQuasiRandComm \* *Communication Data*  
*On entry:* the pointer that was passed to a previous call to **naggpuQuasiRandInitA**.

2: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

### 6 Error Indicators and Warnings

`error → code = 1`

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error → code = 2`

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

## **7 Example**

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpuQuasiRandUniformA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuDepthBBInitA

### 1 Purpose

**naggpuDepthBBInitA** initializes the depth-order Brownian bridge generator **naggpuDepthBBA**. It must be called before any calls to **naggpuDepthBBA** and must finally be followed by a call to **naggpuDepthBBCleanupA**.

**Note:** after the first call to **naggpuDepthBBInitA**, all subsequent calls (for example, to change the time points) must be preceded by a call to **naggpuDepthBBCleanupA**.

### 2 Specification

```
#include <nag_gpu.h>
extern "C"
cudaError_t naggpuDepthBBInitA(float tStart, const float *times, int nTimes,
    bool isBridgeFree, NagGpuDepthBBComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- |    |   |              |
|----|---|--------------|
| 1: | <b>tStart</b> – float   | <i>Input</i> |
|    | <i>On entry:</i> the starting value of the time interval.   |              |
| 2: | <b>times</b> [ <b>nTimes</b> ] – const float *  | <i>Input</i> |
|    | <i>On entry:</i> the vector of times at which to compute the Brownian bridge.   |              |
|    | <i>Constraint:</i> the values in <b>times</b> must be in increasing order, and each must be greater than <b>tStart</b> .                |              |
| 3: | <b>nTimes</b> – int   | <i>Input</i> |
|    | <i>On entry:</i> the length of the vector <b>times</b> .  |              |
|    | <i>Constraint:</i> $1 \leq \text{nTimes} \leq 4095$ .   |              |
| 4: | <b>isBridgeFree</b> – bool  | <i>Input</i> |
|    | <i>On entry:</i> specifies whether a free or ‘pinned’ Brownian bridge is to be constructed. See <b>naggpuDepthBBA</b> for more details. |              |

If `isBridgeFree = true`, `naggpuDepthBBA` will construct a free Brownian motion via a depth-order Brownian bridge algorithm.

If `isBridgeFree = false`, `naggpuDepthBBA` will construct a non-free or ‘pinned’ Brownian motion.

5: **comm** – NagGpuDepthBBComm \* *Communication Data*

`NagGpuDepthBBComm` is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator function `naggpuDepthBBA`. Once all required bridge sample paths have been obtained, **comm** must be passed to `naggpuDepthBBCleanupA` to free allocated system resources.

6: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

`error → code = 1`

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error → code = 2`

*During execution:* a CUDA runtime error was detected.

`error → code = 100`

*On entry:* the value of **comm** is NULL.

`error → code = 110`

*On entry:* the value of **times** is NULL.

`error → code = 111`

*On entry:* the value of **nTimes** does not satisfy the constraint listed above.

`error → code = 112`

*On entry:* the values in the **times** array do not satisfy the constraints listed above.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for `naggpuDepthBBA`.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuDepthBBA

### 1 Purpose

**naggpuDepthBBA** 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 `naggpuDepthBBInitA`, and must finally be followed by a call to `naggpuDepthBBCleanupA`.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuDepthBBA(int nPaths, int dim, float bgStart, float bgEnd,
    const float *d_z, const float *d_cholCov, float *d_bgVals,
    cudaStream_t cstream, NagGpuDepthBBComm *comm, NagGpuError *error)
```

### 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

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

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  $\Sigma$  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

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

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  $\Sigma$ .

#### 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

$$X_{t_{[N/2]}}, X_{t_{[N/4]}}, X_{t_{[N/8]}}, \dots, X_{t_1}$$

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  $X_{t_{[N/2]}}$ , the second dimension is used to compute  $X_{t_{[N/4]}}$ , the third to compute  $X_{t_{[3N/4]}}$ , the fourth to compute  $X_{t_{[N/8]}}$  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  $X_{t_{[N/2]}}$ , the second  $d$  to compute  $X_{t_{[N/4]}}$ , the third  $d$  to compute  $X_{t_{[3N/4]}}$ , 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  $X_{t_{[N/2]}}$ , the third  $d$  to compute  $X_{t_{[N/4]}}$ , and so on.

The boolean parameter **isBridgeFree** in the initialization function naggpuDepthBBInitA 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  $X_{t_1}, X_{t_2}, X_{t_3}, \dots, X_{t_N}$ .

### 3.3 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

### 3.4 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 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: **nPaths** – int Input  
*On entry:* the number of Brownian bridge sample paths to create.  
*Constraint:*  $nPaths \geq 1$ .
- 2: **dim** – int Input  
*On entry:* the dimension of each Brownian bridge sample path.  
*Constraint:*  $1 \leq dim \leq 8$ .
- 3: **bgStart** – float Input  
*On entry:* the starting value  $x$  of the bridge.

- 4: **bgEnd** – float *Input*  
*On entry:* the final value  $w$  of the bridge. If `naggpuDepthBBInitA` was called with `isBridgeFree = true`, this value is ignored and  $w$  is set equal to  $x + CW_T$ .
- 5: **d\_z**[**dim** ×  $N$  × **nPaths**] – const float \* *Input*  
This buffer must reside in the GPU memory space.  
The variable  $N$  denotes the length **nTimes** of the **times** array passed to the initialization function `naggpuDepthBBInitA`.  
*On entry:* the Normal random numbers used to construct the bridge.  
*Constraints:*  
If `naggpuDepthBBInitA` was called with `isBridgeFree = true`, then **d\_z** must contain  $N \times \text{dim} \times \text{nPaths}$  values. The values should be laid out as a matrix with  $\text{dim} \times N$  rows and **nPaths** columns. If quasi-random numbers are to be used, successive  $\text{dim} \times N$ -dimensional points should be stored in successive columns of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_SCATT**;  
If `naggpuDepthBBInitA` was called with `isBridgeFree = false`, then **d\_z** must contain  $(N - 1) \times \text{dim} \times \text{nPaths}$  values. The values should be laid out as a matrix with  $\text{dim} \times (N - 1)$  rows and **nPaths** columns. If quasi-random numbers are to be used, successive  $\text{dim} \times N$ -dimensional points should be stored in successive columns of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_SCATT**.
- 6: **d\_cholCov**[**dim** × **dim**] – const float \* *Input*  
This buffer must reside in the GPU 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  $\text{Cov}(X_s, X_t) = s(T - t)/T\Sigma$  for all  $s \leq t$  in  $[0, T]$ .
- 7: **d\_bgVals**[**dim** ×  $N$  × **nPaths**] – float \* *Output*  
This buffer must reside in the GPU memory space.  
The variable  $N$  denotes the length **nTimes** of the **times** array passed to the initialization function `naggpuDepthBBInitA`.  
*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  $0 \leq d < \text{dim}$ ,  $0 \leq i < N$  and  $0 \leq p < \text{nPaths}$ , then  $x_{p,i}^d$  will be stored at `d_bgVals[p + nPaths(d + i * dim)]`. The starting value **bgStart** is never stored, while the terminal value **bgEnd** is always stored.
- 8: **custream** – `cudaStream_t` *Input*  
*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.
- 9: **comm** – `NagGpuDepthBBComm *` *Communication Data*  
`NagGpuDepthBBComm` is a structure which holds state and communication information and must not be modified in any way. Once all required bridge sample paths have been obtained, **comm** must be passed to `naggpuDepthBBCleanupA` to free allocated system resources.
- 10: **error** – `NagGpuError *` *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:* the value of **nPaths** does not satisfy the constraint listed above.

error → code = 111

*On entry:* the value of **dim** does not satisfy the constraint listed above.

error → code = 112

*On entry:* the value of **d\_z** is NULL.

error → code = 113

*On entry:* the value of **d\_cholCov** is NULL.

error → code = 115

*On entry:* the value of **d\_bgVals** is NULL.

## 7 Example

This example program uses **naggpuDepthBBA** to print two 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 generator `naggpuQuasiRandNormalA`.

### 7.1 Program Text

```

/*
 * Example Program: naggpu_depthbba
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 */

#include <stdio.h>
#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{

```

```

// Number of time steps bridge - less than 4096
const int nTimes = 30;
// Dimension of bridge - less than 8
const int dim = 3;
// Number of sample paths to generate
int N = 50;

// NAG structures
NagGpuError error;
NagCPURandComm pcomm;
NagGpuQuasiRandComm qcomm;
NagGpuDepthBBComm bbcomm;
NagGpuQuasiOrient orient = NAGGPUQUASIORIENT_DIMVALS_SCATT;

cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpuDepthBBA";
cout << endl << endl;

// Initialise the generator only once
cout << "Initialising generators ..." << endl << endl;

// Initialise the CPU pseudorandom generator
unsigned int seed[] = {1,2,3,4,5,6};
nagCPURandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, 0, seed,
                &pcomm, &error);
checkNagError(&error);
// Initialise the quasi-random generator
naggpuQuasiRandInitA(NAGGPUQUASIGEN_SOBOL, NAGGPUSCRAMTYPES_NONE,
                    dim*nTimes, 0, &pcomm, &qcomm, &error);
checkNagError(&error);
// Cleanup the pseudorandom generator
nagCPURandCleanupA(&pcomm, &error);
checkNagError(&error);

// Generate the Sobol numbers
float *d_z = NULL;
cuError = cudaMalloc((void **)&d_z, sizeof(float)*nTimes*dim*N);
checkCudaError(cuError);
naggpuQuasiRandNormalA_sp(N, orient, 0.0f, 1.0f, d_z, NULL, 0,
                          &qcomm, &error);
checkNagError(&error);

// Create the bridge time points
float times[nTimes];
for(int i=0; i<nTimes; i++) times[i] = (i+1)*0.477f;
// Specify bridge setup
float bridgeStart = -0.5;
float bridgeEnd = 1.5;
bool isBridgeFree = false;
// Initialise the bridge generator
naggpuDepthBBInitA(0.0f, times, nTimes, isBridgeFree,
                  &bbcomm, &error);
checkNagError(&error);

// Create covariance structure and copy to GPU
float cov[dim*dim];
for(int i=0; i<dim*dim; i++) cov[i] = 0.1f;
for(int i=0; i<dim; i++) cov[i*(dim+1)] = 0.31f;
float *d_cov = NULL;
cuError = cudaMalloc((void **)&d_cov, sizeof(float)*dim*dim);

```

```

checkCudaError(cuError);
cuError = cudaMemcpy(d_cov, cov, sizeof(float)*dim*dim,
                    cudaMemcpyHostToDevice);
checkCudaError(cuError);

cout << "Creating the bridge ..." << endl << endl;

// Generate bridge and copy back to host
float *d_buff = NULL;
cuError = cudaMalloc((void **)&d_buff, sizeof(float)*dim*nTimes*N);
checkCudaError(cuError);
float *h_buff = new float[dim*nTimes*N];
naggpuDepthBBA(N, dim, bridgeStart, bridgeEnd, d_z, d_cov, d_buff,
               0, &bbcomm, &error);
checkNagError(&error);
cuError = cudaMemcpy(h_buff, d_buff, sizeof(float)*dim*nTimes*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Cleaup quasi-generator and brige
naggpuQuasiRandCleanupA(&qcomm, &error);
checkNagError(&error);
naggpuDepthBBCleanupA(&bbcomm, &error);
checkNagError(&error);
cuError = cudaFree(d_z);
checkCudaError(cuError);
cuError = cudaFree(d_cov);
checkCudaError(cuError);
cuError = cudaFree(d_buff);
checkCudaError(cuError);

// Transpose the data in order to display it
const int nPrint = 2;
float *pPrint = new float[dim*nTimes*nPrint];
for(int p=0; p<nPrint; p++)
{
    for(int i=0; i<nTimes; i++)
    {
        for(int d=0; d<dim; d++)
        {
            print[d+p*dim+i*dim*nPrint] = h_buff[p+N*(d+i*dim)];
        }
    }
}
delete[] h_buff;

// Print Brownian Bridge Sample Paths
cout << "The first " << nPrint << " bridge paths of dimension "
      << dim << ":" << endl;

cout << " \t ";
for(int p = 0; p < nPrint; p++)
{
    int nspaces = (8*dim - 9)/2;
    for(int s=0; s<nspaces; s++) cout << "-";
    cout << " Path" << p+1 << " ";
    for(int s=0; s<nspaces; s++) cout << "-";
    cout << " \t ";
}
cout << endl;
cout << " t_i\t";
for(int p=0; p<nPrint; p++)
{
    for(int d=1; d<=dim; d++) cout << " dim" << d << " ";
}

```

```

        cout << "\t";
    }
    for(int i=0; i<nTimes; i++)
    {
        cout << "\n  " << i+1 << "\t";
        for(int p=0; p<nPrint; p++)
        {
            for(int d=0; d<dim; d++)
            {
                float val = print[i*dim*nPrint + p*dim + d];
                if (val < 10 && val > -10) printf("%.3f  ", val);
                else printf("%.2f  ", val);
            }
            cout << "\t";
        }
    }
    cout << endl << endl;
    delete[] print;

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuDepthBBA

Initialising generators ...

Creating the bridge ...

The first 2 bridge paths of dimension 3:

	----- Path1 -----			----- Path2 -----		
t_i	dim1	dim2	dim3	dim1	dim2	dim3
1	-3.701	-3.701	-3.701	-0.433	-0.433	-0.433
2	-5.913	-5.913	-5.913	-0.367	-0.367	-0.367
3	-5.426	-5.426	-5.426	-0.300	-0.300	-0.300
4	-7.915	-7.915	-7.915	-0.233	-0.233	-0.233

5	-7.703	-7.703	-7.703	-0.167	-0.167	-0.167
6	-8.282	-8.282	-8.282	-0.100	-0.100	-0.100
7	-6.162	-6.162	-6.162	-0.033	-0.033	-0.033
8	-8.962	-8.962	-8.962	0.033	0.033	0.033
9	-9.063	-9.063	-9.063	0.100	0.100	0.100
10	-9.954	-9.954	-9.954	0.167	0.167	0.167
11	-8.145	-8.145	-8.145	0.233	0.233	0.233
12	-9.595	-9.595	-9.595	0.300	0.300	0.300
13	-8.346	-8.346	-8.346	0.367	0.367	0.367
14	-7.887	-7.887	-7.887	0.433	0.433	0.433
15	-4.728	-4.728	-4.728	0.500	0.500	0.500
16	-7.232	-7.232	-7.232	0.567	0.567	0.567
17	-8.747	-8.747	-8.747	0.633	0.633	0.633
18	-7.563	-7.563	-7.563	0.700	0.700	0.700
19	-9.354	-9.354	-9.354	0.767	0.767	0.767
20	-8.446	-8.446	-8.446	0.833	0.833	0.833
21	-8.328	-8.328	-8.328	0.900	0.900	0.900
22	-5.510	-5.510	-5.510	0.967	0.967	0.967
23	-7.614	-7.614	-7.614	1.033	1.033	1.033
24	-7.017	-7.017	-7.017	1.100	1.100	1.100
25	-7.211	-7.211	-7.211	1.167	1.167	1.167
26	-4.705	-4.705	-4.705	1.233	1.233	1.233
27	-5.458	-5.458	-5.458	1.300	1.300	1.300
28	-3.512	-3.512	-3.512	1.367	1.367	1.367
29	-2.356	-2.356	-2.356	1.433	1.433	1.433
30	1.500	1.500	1.500	1.500	1.500	1.500

---

# NAG Numerical Routines for GPUs Function Document

## naggpuDepthBBIncInitA

### 1 Purpose

**naggpuDepthBBIncInitA** initializes the depth-order Brownian bridge increments generator `naggpuDepthBBIncA`. This function must be called before any calls to `naggpuDepthBBIncA` and must finally be followed by a call to `naggpuDepthBBCleanupA`.

**Note:** after the first call to **naggpuDepthBBIncInitA**, all subsequent calls (for example, to change the time points) must be preceded by a call to `naggpuDepthBBCleanupA`.

### 2 Specification

```
#include <nag_gpu.h>
extern "C"
cudaError_t naggpuDepthBBIncInitA(float tStart, const float *times, int nTimes,
    bool isBridgeFree, NagGpuDepthBBComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- |    |  |              |
|----|--|--------------|
| 1: | <b>tStart</b> – float  | <i>Input</i> |
|    | <i>On entry:</i> the starting value of the time interval.  |              |
| 2: | <b>times</b> [ <b>nTimes</b> ] – const float *   | <i>Input</i> |
|    | <i>On entry:</i> the vector of times at which to compute the Brownian bridge.  |              |
|    | <i>Constraint:</i> the values in <b>times</b> must be in increasing order, and each must be greater than <b>tStart</b> .   |              |
| 3: | <b>nTimes</b> – int  | <i>Input</i> |
|    | <i>On entry:</i> the length of the vector <b>times</b> .   |              |
|    | <i>Constraint:</i> $1 \leq \text{nTimes} \leq 4095$ .  |              |
| 4: | <b>isBridgeFree</b> – bool   | <i>Input</i> |
|    | <i>On entry:</i> specifies whether scaled increments for a free or ‘pinned’ Brownian bridge is to be constructed. See <code>naggpuDepthBBIncA</code> for more details. |              |

If `isBridgeFree = true`, `naggpuDepthBBIncA` will construct scaled increments of a free Brownian motion via a depth-order Brownian bridge algorithm.

If `isBridgeFree = false`, `naggpuDepthBBIncA` will construct scaled increments of a non-free or ‘pinned’ Brownian motion.

5: **comm** – NagGpuDepthBBComm \* *Communication Data*

`NagGpuDepthBBComm` is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator function `naggpuDepthBBIncA`. Once all required bridge increments have been obtained, **comm** must be passed to `naggpuDepthBBCleanupA` to free allocated system resources.

6: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

`error → code = 1`

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error → code = 2`

*During execution:* a CUDA runtime error was detected.

`error → code = 100`

*On entry:* the value of **comm** is NULL.

`error → code = 110`

*On entry:* the value of **times** is NULL.

`error → code = 111`

*On entry:* the value of **nTimes** does not satisfy the constraint listed above.

`error → code = 112`

*On entry:* the values in the **times** array do not satisfy the constraints listed above.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for `naggpuDepthBBIncA`.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuDepthBBInCA

### 1 Purpose

**naggpuDepthBBInCA** computes scaled increments of a depth-order Brownian bridge or free Brownian motion. It must be preceded by a call to the initialization function `naggpuDepthBBInCAInitA`, and must finally be followed by a call to `naggpuDepthBBCleanupA`.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuDepthBBInCA(int nPaths, int dim, float startEndDiff,
    const float *d_z, const float *d_cholCov, float *d_bgIncs,
    cudaStream_t cstream, NagGpuDepthBBComm *comm, NagGpuError *error)
```

### 3 Description

Fix  $T > 0$  and suppose that  $0 = t_0 < t_1 < \dots < t_N = T$ . Conceptually, this algorithm first constructs a depth-order Brownian bridge  $X = (X_t)_{0 \leq t \leq T}$  in the same way as `naggpuDepthBBA` and then computes

$$\frac{X_{t_1} - X_{t_0}}{t_1 - t_0}, \frac{X_{t_2} - X_{t_1}}{t_2 - t_1}, \dots, \frac{X_{t_N} - X_{t_{N-1}}}{t_N - t_{N-1}}.$$

Such increments can be useful when computing numerical solutions to stochastic differential equations driven by either a Brownian bridge or a free Brownian motion. For more details on the Brownian bridge and how it is constructed, see the discussion in `naggpuDepthBBA`.

We briefly recall some notation: for further details see `naggpuDepthBBA`. We let  $W = (W_t)_{0 \leq t \leq T}$  be a standard  $d$ -dimensional Wiener process, we let  $\Sigma$  be a  $d \times d$  covariance matrix, we choose  $C$  to be a  $d \times d$  matrix such that  $CC' = \Sigma$ , and we fix two points  $x$  and  $w$  in  $\mathbb{R}^d$ . The generalized Brownian bridge  $X = (X_t)_{0 \leq t \leq T}$  is defined as

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

for all  $t \in [0, T]$  so that  $X_0 = x$ ,  $X_T = w$  and  $\text{Cov}(X_s, X_t) = s(T - t)/T\Sigma$  for all  $s \leq t$  in  $[0, T]$ . This process is a non-free or ‘pinned’ Brownian motion since  $X_T = w$ . However if we set  $w = x + CW_T$  then  $X_t = x + CW_t$  becomes a standard, correlated  $d$ -dimensional Brownian motion. The boolean parameter **isBridgeFree** in the initialization routine `naggpuDepthBBInCAInitA` controls whether a free or non-free Brownian sample path is created.

#### 3.1 Synchronization

This function is non-blocking. Control will return immediately to the calling program while the computation is executed on the GPU. The user is responsible for synchronization between host and GPU code. Please see the synchronization chapter in the CUDA Programming Guide for further details in this direction. For example, a call to `cudaMemcpy` in the CUDA runtime library is enough to force the host to wait for the GPU to finish, and then copy the results from the GPU to the host.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

None.

## 5 Arguments

- 1: **nPaths** – int *Input*  
*On entry:* the number of Brownian bridge sample paths that are created.  
*Constraint:*  $nPaths \geq 1$ .
- 2: **dim** – int *Input*  
*On entry:* the dimension of each Brownian bridge sample path.  
*Constraint:*  $1 \leq dim \leq 8$ .
- 3: **startEndDiff** – float *Input*  
*On entry:* the difference between  $X_{t_N}$  and  $X_{t_0}$ . If naggpuDepthBBIncInitA was called with `isBridgeFree = true`, this value is ignored and  $X_{t_N}$  is set equal to  $x + CW_T$ .
- 4: **d\_z**[**dim** ×  $N$  × **nPaths**] – const float \* *Input*  
 This buffer must reside in the GPU memory space.  
 The variable  $N$  denotes the length **nTimes** of the **times** array passed to the initialization function naggpuDepthBBIncInitA.  
*On entry:* the Normal random numbers used to construct the bridge.  
*Constraints:*  
 If naggpuDepthBBIncInitA was called with `isBridgeFree = true`, then **d\_z** must contain  $N \times dim \times nPaths$  values. The values should be laid out as a matrix with  $dim \times N$  rows and **nPaths** columns. If quasi-random numbers are to be used, successive  $dim \times N$ -dimensional points should be stored in successive columns of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_SCATT**;  
 If naggpuDepthBBIncInitA was called with `isBridgeFree = false`, then **d\_z** must contain  $(N - 1) \times dim \times nPaths$  values. The values should be laid out as a matrix with  $dim \times (N - 1)$  rows and **nPaths** columns. If quasi-random numbers are to be used, successive  $dim \times N$ -dimensional points should be stored in successive columns of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_SCATT**.
- 5: **d\_cholCov**[**dim** × **dim**] – const float \* *Input*  
 This buffer must reside in the GPU memory space.  
*On entry:* the matrix  $C$  which specifies the correlation structure of the Brownian bridge.  $C$  should be chosen such that  $CC^T = \Sigma$  where  $\text{Cov}(X_s, X_t) = s(T - t)/T\Sigma$  for all  $s \leq t$  in  $[0, T]$ .
- 6: **d\_bgIncs**[**dim** ×  $N$  × **nPaths**] – float \* *Output*  
 This buffer must reside in the GPU memory space.  
 The variable  $N$  denotes the length **nTimes** of the **times** array passed to the initialization function naggpuDepthBBIncInitA.  
*On exit:* the scaled increments 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  $0 \leq d < dim$ ,  $0 \leq i < N$  and  $0 \leq p < nPaths$ , then the scaled increment  $(x_{p,i+1}^d - x_{p,i}^d)/(t_{i+1} - t_i)$  will be stored at `d_bgIncs[p + nPaths(d + i * dim)]`.

- 7: **ostream** – cudaStream\_t *Input*  
*On entry:* specifies the CUDA stream on which to launch the selected GPU kernel. If no streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.
- 8: **comm** – NagGpuDepthBBComm \* *Communication Data*  
NagGpuDepthBBComm is a structure which holds state and communication information and must not be modified in any way. Once all required bridge sample paths have been obtained, **comm** must be passed to naggpuDepthBBCleanupA to free allocated system resources.
- 9: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:* the value of **nPaths** does not satisfy the constraint listed above.

error → code = 111

*On entry:* the value of **dim** does not satisfy the constraint listed above.

error → code = 112

*On entry:* the value of **d\_z** is NULL.

error → code = 113

*On entry:* the value of **d\_cholCov** is NULL.

error → code = 114

*On entry:* the value of **d\_bgIncs** is NULL.

## 7 Example

This example program uses **naggpuDepthBBInCA** and naggpuDepthBBA to print a Brownian bridge sample path and the corresponding scaled path increments side by side. Each sample path is three dimensional and the bridge is pinned to end at a fixed value. To aid comparison, the distance between time points  $\Delta t_i := t_{i+1} - t_i = 1$  so that the scaled increments  $(X_{t_{i+1}} - X_{t_i}) / (t_{i+1} - t_i)$  are in fact the true bridge increments  $X_{t_{i+1}} - X_{t_i}$ .

## 7.1 Program Text

```

/*
 * Example Program: naggpu_depthbba
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include <stdio.h>
#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

int main(int argc, char **argv)
{
    // Number of time steps bridge - less than 4096
    const int nTimes = 30;
    // Dimension of bridge - less than 8
    const int dim = 3;
    // Number of sample paths to generate
    int N = 2;

    // NAG structures
    NagGpuError error;
    NagCPURandComm pcomm;
    NagGpuQuasiRandComm qcomm;
    NagGpuDepthBBComm bbcomm;
    NagGpuQuasiOrient orient = NAGGPUQUASIORIENT_DIMVALS_SCATT;

    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpuDepthBBIncA";
    cout << endl << endl;

    // Initialise the CPU pseudorandom generator
    unsigned int seed[] = {1,2,3,4,5,6};
    nagCPURandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, 0, seed,
                   &pcomm, &error);
    checkNagError(&error);
    // Initialise the quasi-random generator
    naggpuQuasiRandInitA(NAGGPUQUASIGEN_SOBOL, NAGGPUSCRAMTYPES_NONE,
                        dim*nTimes, 0, &pcomm, &qcomm, &error);
    checkNagError(&error);
    // Cleanup the pseudorandom generator
    nagCPURandCleanupA(&pcomm, &error);
    checkNagError(&error);

    // Generate the Sobol numbers and clean up
    float *d_z = NULL;
    cuError = cudaMalloc((void **)&d_z, sizeof(float)*nTimes*dim*N);
    checkCudaError(cuError);
    naggpuQuasiRandNormalA_sp(N, orient, 0.0f, 1.0f, d_z, NULL, 0,
                              &qcomm, &error);

    checkNagError(&error);
    naggpuQuasiRandCleanupA(&qcomm, &error);
    checkNagError(&error);
}

```

```

// Create the bridge time points
float times[nTimes];
for(int i=0; i<nTimes; i++) times[i] = (i+1)*1.0f;
// Specify bridge setup
float bridgeStart = -0.5;
float bridgeEnd = 1.5;
bool isBridgeFree = false;

// Create covariance structure and copy to GPU
float cov[dim*dim];
for(int i=0; i<dim*dim; i++) cov[i] = 0.1f;
for(int i=0; i<dim; i++) cov[i*(dim+1)] = 0.31f;
float *d_cov = NULL;
cuError = cudaMalloc((void **)&d_cov, sizeof(float)*dim*dim);
checkCudaError(cuError);
cuError = cudaMemcpy(d_cov, cov, sizeof(float)*dim*dim,
                    cudaMemcpyHostToDevice);
checkCudaError(cuError);

// Create storage for bridge numbers on host and device
float *d_buff = NULL;
cuError = cudaMalloc((void **)&d_buff, sizeof(float)*dim*nTimes*N);
checkCudaError(cuError);
float *h_bb = new float[dim*nTimes*N];
float *h_bbinc = new float[dim*nTimes*N];

// Initialise the bridge generator, generate and clean up
naggpuDepthBBInitA(0.0f, times, nTimes, isBridgeFree,
                  &bbcomm, &error);
checkNagError(&error);
naggpuDepthBBA(N, dim, bridgeStart, bridgeEnd, d_z, d_cov, d_buff,
              0, &bbcomm, &error);
checkNagError(&error);
cuError = cudaMemcpy(h_bb, d_buff, sizeof(float)*dim*nTimes*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);
naggpuDepthBBCleanupA(&bbcomm, &error);
checkNagError(&error);

// Initialise the bridge increments generator, generate and clean up
naggpuDepthBBIncInitA(0.0f, times, nTimes, isBridgeFree,
                     &bbcomm, &error);
checkNagError(&error);
naggpuDepthBBIncA(N, dim, bridgeEnd-bridgeStart, d_z, d_cov,
                 d_buff, 0, &bbcomm, &error);
checkNagError(&error);
cuError = cudaMemcpy(h_bbinc, d_buff, sizeof(float)*dim*nTimes*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);
naggpuDepthBBCleanupA(&bbcomm, &error);
checkNagError(&error);

// Free GPU memory
cuError = cudaFree(d_z);
checkCudaError(cuError);
cuError = cudaFree(d_buff);
checkCudaError(cuError);
cuError = cudaFree(d_cov);
checkCudaError(cuError);

// Print Brownian Bridge Sample Paths and increments

```

```

printf("\nThe %d Brownian Bridge paths of dimension %d and "
      "corresponding increments:\n", N, dim);
printf("Bridge starts at x = %g, and t_{i+1}-t_i = dt = %g\n",
      bridgeStart, times[0]);

// Top level grouping
for(int p=0; p<N; p++)
{
    int nspaces = (9*dim - 9)/2;
    printf("      \t ");
    for(int s = 0; s < nspaces; s++) printf("-");
    printf(" Path%d ", p+1);
    for(int s = 0; s < nspaces; s++) printf("-");

    printf("-----");

    for(int s = 0; s < nspaces; s++) printf("-");
    printf(" Incr%d ", p+1);
    for(int s = 0; s < nspaces; s++) printf("-");

    // Time and Dimension column headings
    printf("\n t_i\t");
    for(int d=1; d<=dim; d++) printf(" dim%d ", d);
    printf(" ");
    for(int d=1; d<=dim; d++) printf(" dim%d ", d);
    // Values
    for(int i = -1; i < nTimes; i++)
    {
        printf("\n %d\t", i+1);
        // Print path
        for(int d=0; d<dim; d++)
        {
            float val = 0;
            if (i < 0) val = bridgeStart;
            else val = h_bb[p+N*(d+i*dim)];
            if (val < 10 && val > -10) printf("% .4f ", val);
            else printf("% .3f ", val);
        }
        printf(" ");
        // Print increments
        for(int d=0; d<dim; d++)
        {
            if (i < 0) printf(" ");
            else
            {
                float val = h_bbinc[p+N*(d+i*dim)];
                if (val < 10 && val > -10) printf("% .4f ", val);
                else printf("% .3f ", val);
            }
        }
        printf("\n\n\n");
    }
}

delete[] h_bb;
delete[] h_bbinc;

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)

```

```

    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpuDepthBBInCA

The 2 Brownian Bridge paths of dimension 3 and corresponding increments:  
 Bridge starts at  $x = -0.5$ , and  $t_{i+1} - t_i = dt = 1$

t_i	dim1	dim2	dim3	dim1	dim2	dim3
0	-0.5000	-0.5000	-0.5000			
1	-5.1643	-5.1643	-5.1643	-4.6643	-4.6643	-4.6643
2	-8.3978	-8.3978	-8.3978	-3.2335	-3.2335	-3.2335
3	-7.7222	-7.7222	-7.7222	0.6757	0.6757	0.6757
4	-11.355	-11.355	-11.355	-3.6329	-3.6329	-3.6329
5	-11.079	-11.079	-11.079	0.2763	0.2763	0.2763
6	-11.947	-11.947	-11.947	-0.8687	-0.8687	-0.8687
7	-8.9069	-8.9069	-8.9069	3.0405	3.0405	3.0405
8	-12.991	-12.991	-12.991	-4.0844	-4.0844	-4.0844
9	-13.166	-13.166	-13.166	-0.1752	-0.1752	-0.1752
10	-14.487	-14.487	-14.487	-1.3202	-1.3202	-1.3202
11	-11.898	-11.898	-11.898	2.5890	2.5890	2.5890
12	-14.027	-14.027	-14.027	-2.1298	-2.1298	-2.1298
13	-12.248	-12.248	-12.248	1.7794	1.7794	1.7794
14	-11.614	-11.614	-11.614	0.6344	0.6344	0.6344
15	-7.0700	-7.0700	-7.0700	4.5436	4.5436	4.5436
16	-10.725	-10.725	-10.725	-3.6550	-3.6550	-3.6550
17	-12.949	-12.949	-12.949	-2.2241	-2.2241	-2.2241
18	-11.264	-11.264	-11.264	1.6850	1.6850	1.6850
19	-13.888	-13.888	-13.888	-2.6235	-2.6235	-2.6235
20	-12.602	-12.602	-12.602	1.2856	1.2856	1.2856
21	-12.461	-12.461	-12.461	0.1407	0.1407	0.1407
22	-8.4116	-8.4116	-8.4116	4.0498	4.0498	4.0498
23	-11.487	-11.487	-11.487	-3.0750	-3.0750	-3.0750
24	-10.652	-10.652	-10.652	0.8341	0.8341	0.8341
25	-10.963	-10.963	-10.963	-0.3108	-0.3108	-0.3108
26	-7.3650	-7.3650	-7.3650	3.5983	3.5983	3.5983
27	-8.4854	-8.4854	-8.4854	-1.1204	-1.1204	-1.1204
28	-5.6967	-5.6967	-5.6967	2.7887	2.7887	2.7887
29	-4.0529	-4.0529	-4.0529	1.6438	1.6438	1.6438
30	1.5000	1.5000	1.5000	5.5529	5.5529	5.5529

  

t_i	dim1	dim2	dim3	dim1	dim2	dim3
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						

0	-0.5000	-0.5000	-0.5000			
1	-0.4333	-0.4333	-0.4333	0.0667	0.0667	0.0667
2	-0.3667	-0.3667	-0.3667	0.0667	0.0667	0.0667
3	-0.3000	-0.3000	-0.3000	0.0667	0.0667	0.0667
4	-0.2333	-0.2333	-0.2333	0.0667	0.0667	0.0667
5	-0.1667	-0.1667	-0.1667	0.0667	0.0667	0.0667
6	-0.1000	-0.1000	-0.1000	0.0667	0.0667	0.0667
7	-0.0333	-0.0333	-0.0333	0.0667	0.0667	0.0667
8	0.0333	0.0333	0.0333	0.0667	0.0667	0.0667
9	0.1000	0.1000	0.1000	0.0667	0.0667	0.0667
10	0.1667	0.1667	0.1667	0.0667	0.0667	0.0667
11	0.2333	0.2333	0.2333	0.0667	0.0667	0.0667
12	0.3000	0.3000	0.3000	0.0667	0.0667	0.0667
13	0.3667	0.3667	0.3667	0.0667	0.0667	0.0667
14	0.4333	0.4333	0.4333	0.0667	0.0667	0.0667
15	0.5000	0.5000	0.5000	0.0667	0.0667	0.0667
16	0.5667	0.5667	0.5667	0.0667	0.0667	0.0667
17	0.6333	0.6333	0.6333	0.0667	0.0667	0.0667
18	0.7000	0.7000	0.7000	0.0667	0.0667	0.0667
19	0.7667	0.7667	0.7667	0.0667	0.0667	0.0667
20	0.8333	0.8333	0.8333	0.0667	0.0667	0.0667
21	0.9000	0.9000	0.9000	0.0667	0.0667	0.0667
22	0.9667	0.9667	0.9667	0.0667	0.0667	0.0667
23	1.0333	1.0333	1.0333	0.0667	0.0667	0.0667
24	1.1000	1.1000	1.1000	0.0667	0.0667	0.0667
25	1.1667	1.1667	1.1667	0.0667	0.0667	0.0667
26	1.2333	1.2333	1.2333	0.0667	0.0667	0.0667
27	1.3000	1.3000	1.3000	0.0667	0.0667	0.0667
28	1.3667	1.3667	1.3667	0.0667	0.0667	0.0667
29	1.4333	1.4333	1.4333	0.0667	0.0667	0.0667
30	1.5000	1.5000	1.5000	0.0667	0.0667	0.0667

---

# NAG Numerical Routines for GPUs Function Document

## naggpuDepthBBCleanupA

### 1 Purpose

**naggpuDepthBBCleanupA** frees system resources which were allocated by a previous call to **naggpuDepthBBInitA** or **naggpuDepthBBIncInitA**.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuDepthBBCleanupA(NagGpuDepthBBComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- 1: **comm** – NagGpuDepthBBComm \* *Communication Data*  
The structure which was initialized by a previous call to **naggpuDepthBBInitA** or **naggpuDepthBBIncInitA**.
- 2: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

### 6 Error Indicators and Warnings

`error → code = 1`

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

`error → code = 2`

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

## **7 Example**

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpuDepthBBA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuMrg32k3aDeviceInitA

### 1 Purpose

**naggpuMrg32k3aDeviceInitA** creates initialization data on the GPU for the MRG32k3a device function generators. It must be called prior to a call to `naggpudevMrg32k3aInitA` and must ultimately be followed by a call to `naggpuMrg32k3aDeviceCleanupA` to free allocated system resources.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuMrg32k3aDeviceInitA(int a1, int b1, int a2, int b2, long long c,
    unsigned int *seed, NagGpuMrg32k3aDeviceComm **devComm,
    NagGpuRandComm *comm, NagGpuError *error)
```

### 3 Description

To use the MRG32k3a device function generators, the memory address of a `NagGpuMrg32k3aDeviceComm` structure must be obtained from the device generator initialization function `naggpuMrg32k3aDeviceInitA`. This structure will reside in the GPU memory space and will contain communication data for use by the device function generator. This GPU memory address must then be passed to the MRG32k3a device generator initialization function `naggpudevMrg32k3aInitA`. Once all values have been obtained from the device function generators, the same `NagGpuMrg32k3aDeviceComm` structure memory address must be passed to `naggpuMrg32k3aDeviceCleanupA` to free allocated system resources.

This function will create the `NagGpuMrg32k3aDeviceComm` communication structure in the GPU memory space and will assign its address to `devComm`. It also requires the seed for the generator to be passed in for error checking and verification. The seed can optionally be skipped ahead (see below) by a user-specified amount. Note that this function **does NOT copy the seed to the device**: the supplied seed is verified and skipped ahead (if required), but is left on the host. The user must transfer the seed to the device and ultimately pass it to `naggpudevMrg32k3aInitA`. This is for performance reasons: users will typically wish to exploit the GPU's memory hierarchy and place the seed in a specific location, be it registers, shared memory, or elsewhere.

#### 3.1 Parallelization

For different values of `seed`, a given generator will yield different sequences of random numbers. Alternatively, the same sequence of random numbers will be generated if the same value of `seed` is used. In general there is no guarantee of statistical properties between sequences, only within sequences. This is important when generators are used in parallel. This function can 'skip ahead' or advance the seed by an amount

$$s = a_1 2^{b_1} + a_2 2^{b_2} + c \quad (1)$$

so that the generator will produce the sequence of random numbers  $X_s, X_{s+1}, X_{s+2}, \dots$  instead of the original sequence  $X_0, X_1, X_2, \dots$ . This technique is useful to produce independent generators, often also called *independent streams and substreams*. Please see the Random Number Generators Chapter Introduction for further information.

The skip ahead functionality provided by **naggpuMrg32k3aDeviceInitA** will typically only be important to applications which use multiple GPUs simultaneously. Users would write their own kernel, embedding the inline MRG32k3a device function generator, and then launch the kernel on multiple GPUs simultaneously to distribute the computation. In this case, each user kernel would require its own `NagGpuRandComm` and `NagGpuMrg32k3aDeviceComm` structures to pass to the MRG32k3a device

functions. Each NagGpuRandComm and NagGpuMrg32k3aDeviceComm pair must be initialized by a call to **naggpuMrg32k3aDeviceInitA**. Note that when initializing a given pair, the call to **naggpuMrg32k3aDeviceInitA** must access the same GPU device context which will be used to execute the instance of the user's kernel associated with that pair. Please consult the CUDA documentation for details on how to achieve this and how to access multiple GPUs simultaneously from a single application.

Note that the initialization function `naggpudevMrg32k3aInitA` provides another opportunity to advance the seed before generating values. The skip ahead performed by **naggpuMrg32k3aDeviceInitA** therefore corresponds to the GPU upon which the user's kernel will run (e.g. determined by an OpenMP thread number and/or a GPU device index), while the skip ahead performed by `naggpudevMrg32k3aInitA` will correspond to the thread number of the CUDA thread in the user's GPU kernel.

If the user's kernel is not to be distributed across multiple GPUs, the skip ahead  $s$  above can be set to zero and the comments about arrays of communication structures can be ignored.

### 3.2 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 3.3 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

## 4 References

L'Ecuyer P (1999) Good parameter sets for combined multiple recursive random number generators *Operations Research* **47:1** 159–164

## 5 Arguments

- |    |   |              |
|----|---|--------------|
| 1: | <b>a1</b> – int   | <i>Input</i> |
|    | <i>On entry:</i> the value of $a_1$ in the skip ahead equation (1) above. |              |
|    | <i>Constraint:</i> $a_1 \geq 0$ .   |              |
| 2: | <b>b1</b> – int   | <i>Input</i> |
|    | <i>On entry:</i> the value of $b_1$ in the skip ahead equation (1) above. |              |
|    | <i>Constraint:</i> $0 \leq b_1 \leq 191$ .                                |              |
| 3: | <b>a2</b> – int   | <i>Input</i> |
|    | <i>On entry:</i> the value of $a_2$ in the skip ahead equation (1) above. |              |
|    | <i>Constraint:</i> $a_2 \geq 0$ .   |              |
| 4: | <b>b2</b> – int   | <i>Input</i> |
|    | <i>On entry:</i> the value of $b_2$ in the skip ahead equation (1) above. |              |
|    | <i>Constraint:</i> $0 \leq b_2 \leq 191$ .                                |              |
| 5: | <b>c</b> – long long  | <i>Input</i> |
|    | <i>On entry:</i> the value of $c$ in the skip ahead equation (1) above.   |              |
|    | <i>Constraint:</i> $c \geq 0$ .   |              |

6: **seed**[6] – unsigned int \* *Input/Output*

The device function generator seed. Note that this function **does NOT copy the seed to the device**. This is left to the user, who may wish to control where the seed is placed in the GPU memory hierarchy.

*On entry:* the seed which is to be used.

*On exit:* the seed skipped ahead by  $s = a_1 2^{b_1} + a_2 2^{b_2} + c$  steps.

*Constraints:*

for  $i = 0, 1, 2$ ,  $\text{seed}[i] < 2^{32} - 209$  and  $\text{seed}[i] \neq 0$  for at least one  $i$ ;  
for  $i = 3, 4, 5$ ,  $\text{seed}[i] < 2^{32} - 22853$  and  $\text{seed}[i] \neq 0$  for at least one  $i$ .

7: **devComm** – NagGpuMrg32k3aDeviceComm \*\* *Communication Data*

NagGpuMrg32k3aDeviceComm is a structure which holds state and communication information and must not be modified in any way. A structure will be allocated in the GPU memory space and its address will be assigned to the location pointed to by **devComm**. This address must be passed to the MRG32k3a device generator initialization function naggpudevMrg32k3aInitA. Once all required values have been obtained, this address must be passed to naggpuMrg32k3aDeviceCleanupA to free allocated system resources.

8: **comm** – NagGpuRandComm \* *Communication Data*

NagGpuRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required values have been obtained from the MRG32k3a device function generators, **comm** must be passed to naggpuMrg32k3aDeviceCleanupA to free allocated system resources.

9: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of  $\text{error} \rightarrow \text{code}$  which should be inspected after each call to this function. If  $\text{error} \rightarrow \text{code} = 0$  then no error occurred. If  $\text{error} \rightarrow \text{code} \neq 0$  then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

$\text{error} \rightarrow \text{code} = 1$

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

$\text{error} \rightarrow \text{code} = 2$

*During execution:* a CUDA runtime error was detected.

$\text{error} \rightarrow \text{code} = 100$

*On entry:* the value of **comm** is NULL.

$\text{error} \rightarrow \text{code} = 111$

*On entry:* the value of **a1** is negative.

$\text{error} \rightarrow \text{code} = 112$

*On entry:* the value of **b1** does not satisfy the constraints listed above.

$\text{error} \rightarrow \text{code} = 113$

*On entry:* the value of **a2** is negative.

error → code = 114

*On entry:* the value of **b2** does not satisfy the constraints listed above.

error → code = 115

*On entry:* the value of **c** is negative.

error → code = 116

*On entry:* the value of **seed** is NULL.

error → code = 117

*On entry:* the values in the **seed** array do not satisfy the constraints listed above.

error → code = 118

*On entry:* the value of **devComm** is NULL.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpudevMrg32k3aUniformA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpudevMrg32k3aInitA

### 1 Purpose

**naggpudevMrg32k3aInitA** initializes the MRG32k3a device function generators. It must be preceded by a call to **naggpuMrg32k3aDeviceInitA** and must ultimately be followed by a call to **naggpuMrg32k3aDeviceCleanupA** to free allocated system resources. **naggpudevMrg32k3aInitA** must be called prior to calling any of the device generator functions such as **naggpudevMrg32k3aUniformA**.

### 2 Specification

```
#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

__device__ void naggpudevMrg32k3aInitA(unsigned int a1, unsigned int b1,
    unsigned int a2, unsigned int b2, unsigned long long c, unsigned int *seed,
    const NagGpuMrg32k3aDeviceComm *devComm)
```

### 3 Description

The MRG32k3a generator is a ‘light weight’ generator with a small state and very good statistical properties. Due to its small state, it is well suited for use on a GPU and can be embedded in a user’s GPU kernel. Typically each CUDA thread will have its own copy of state and will run as an independent generator. The NVIDIA compiler will often store the state in registers, provided the array is handled with some care (see the CUDA Programming Guide section on Local Memory). There is no need for threads to communicate in order to generate values, and ideally threads should consume the values as they come off the generator: this avoids any storage overhead and maximizes speed.

#### 3.1 Parallelization

For different values of **seed**, a given generator will yield different sequences of random numbers. Alternatively, the same sequence of random numbers will be generated if the same value of **seed** is used. In general there is no guarantee of statistical properties between sequences, only within sequences. This is important when generators are used in parallel. This function can ‘skip ahead’ or advance the seed by an amount

$$s = a_1 2^{b_1} + a_2 2^{b_2} + c \quad (1)$$

so that the generator will produce the sequence of random numbers  $X_s, X_{s+1}, X_{s+2}, \dots$  instead of the original sequence  $X_0, X_1, X_2, \dots$ . This technique is useful to produce independent generators, often also called *independent streams and substreams*. Please see the Random Number Generators Chapter Introduction for further information.

Each CUDA thread can skip the seed ahead according to the formula above. Which skip aheads are chosen depends on the application and on how the CUDA threads are to cooperate. Applications where the number of random variates required is known in advance will probably use a block splitting scheme as described in the Random Number Generators Chapter Introduction. Others may wish to use independent streams.

#### 3.2 Error Handling

This is a GPU device function, and no error handling is performed.

## 4 References

L'Ecuyer P (1999) Good parameter sets for combined multiple recursive random number generators *Operations Research* **47:1** 159–164

## 5 Arguments

- 1: **a1** – unsigned int *Input*  
*On entry:* the value of  $a_1$  in the skip ahead equation (1) above.
- 2: **b1** – unsigned int *Input*  
*On entry:* the value of  $b_1$  in the skip ahead equation (1) above.  
*Constraint:*  $b_1 \leq 191$ .
- 3: **a2** – unsigned int *Input*  
*On entry:* the value of  $a_2$  in the skip ahead equation (1) above.
- 4: **b2** – unsigned int *Input*  
*On entry:* the value of  $b_2$  in the skip ahead equation (1) above.  
*Constraint:*  $b_2 \leq 191$ .
- 5: **c** – unsigned long long *Input*  
*On entry:* the value of  $c$  in the skip ahead equation (1) above.
- 6: **seed[6]** – unsigned int \* *Input/Output*  
*On entry:* the seed which is to be used.  
*On exit:* the state of the generator. This is the seed skipped ahead by  $s = a_1 2^{b_1} + a_2 2^{b_2} + c$  steps.  
*Constraints:*  
     for  $i = 0, 1, 2$ ,  $\text{seed}[i] < 2^{32} - 209$  and  $\text{seed}[i] \neq 0$  for at least one  $i$ ;  
     for  $i = 3, 4, 5$ ,  $\text{seed}[i] < 2^{32} - 22853$  and  $\text{seed}[i] \neq 0$  for at least one  $i$ .
- 7: **devComm** – const NagGpuMrg32k3aDeviceComm \* *Communication Data*  
 The NagGpuMrg32k3aDeviceComm memory address obtained from the host initialization function naggpuMrg32k3aDeviceInitA

## 6 Error Indicators and Warnings

None.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpudevMrg32k3aUniformA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpudevMrg32k3aExpA

### 1 Purpose

**naggpudevMrg32k3aExpA** generates the next value from an exponential distribution with mean  $\lambda$ .

The initialization function **naggpudevMrg32k3aInitA** must be called prior to the first call to **naggpudevMrg32k3aExpA**. Thereafter, this function may be called repeatedly to generate the next value in the sequence. Care should be taken to ensure that sequences from successive CUDA threads do not overlap.

### 2 Specification

```
#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

template <typename FP>
__device__ void naggpudevMrg32k3aExpA(FP &x, FP lambda, unsigned int *state)
```

### 3 Description

The exponential distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\lambda} e^{-x/\lambda} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\lambda > 0$ . This function returns

$$X = -\lambda \ln Y$$

where  $Y$  is the next value generated by the MRG32k3a uniform  $(0, 1]$  generator.

#### 3.1 Error Handling

This is a GPU device function, and no error handling is performed.

### 4 References

None.

### 5 Arguments

- |    |  |               |
|----|--|---------------|
| 1: | <b>x</b> – FP &  | <i>Output</i> |
|    | The template argument parameter FP can take type double or float.<br><i>On exit:</i> the next pseudorandom value from the sequence.  |               |
| 2: | <b>lambda</b> – FP   | <i>Input</i>  |
|    | The template argument parameter FP can take type double or float.<br><i>On entry:</i> the mean, $\lambda$ , of the exponential distribution.<br><i>Constraint:</i> lambda > 0. |               |

3: **state** – unsigned int \*

*Input/Output*

*On entry:* the output state from the previous call to any of the MRG32k3a device function generators, or if this is the first call to any of the MRG32k3a device generators, the value of **seed** obtained from the initialization call naggpudevMrg32k3aInitA

*On exit:* the output state of the generator

## 6 Error Indicators and Warnings

None.

## 7 Example

This example program uses **naggpudevMrg32k3aExpA** to print 50 pseudorandom numbers from an exponential distribution using the MRG32k3a device function generator.

### 7.1 Program Text

```

/*
 * Example Program: naggpu_devMrg32k3a_expA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

/*
 * Test kernel to try out device level functions
 * Note: this will NOT output numbers in the same order
 * as the CPU routine
 */
__global__ void mrg32k3a_device_functions_exponential_test(
    int npts,      // Number of points per thread
    int N,        // Total number of points to create
    FP *d_buff,   // memory address to store points
    unsigned int *d_seed,
    NagGpuMrg32k3aDeviceComm *devComm)
{
    /*
     * Assumptions:
     * Grid is one dimensional
     * Thread block is one dimensional
     */

    const FP lambda = 1.0;

    unsigned int state[6];
    for (int i = 0; i < 6; i++) state[i] = d_seed[i];

```

```

// Generation offset
int offset = ((blockIdx.x * blockDim.x) + threadIdx.x)*npts;

// Storage offset
int store = threadIdx.x + npts*blockIdx.x*blockDim.x;

FP x = 0;

// Initialise state for this thread
naggpudevMrg32k3aInitA(0, 0, 0, 0, offset, state, devComm);

// Compute only as many points as requested
for (int i = 0; i < npts; i++)
{
    if (store < N)
    {
        naggpudevMrg32k3aExpA(x, lambda, state);
        d_buff[store] = x;
        store += blockDim.x;
    }
    else
    {
        break;
    }
}
}

int main(int argc, char **argv)
{
    const int N = 2621440;

    FP *h_buff = 0, *d_buff = 0;

    unsigned int seed[] = {1, 2, 3, 1, 2, 3};
    unsigned int *d_seed;

    unsigned long long offset = 1;

    NagGpuMrg32k3aDeviceComm *devComm;
    NagGpuRandComm comm;
    NagGpuError error;
    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpudevMrg32k3aExpA, ";
    if (sizeof(FP)==sizeof(float))
        cout << "single precision";
    else
        cout << "double precision";
    cout << endl << endl;

    // Allocate CPU and GPU memory
    h_buff = new FP[N];
    cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
    checkCudaError(cuError);
    cuError = cudaMalloc((void **)&d_seed, sizeof(unsigned int) * 6);
    checkCudaError(cuError);

    // Host initialisation call
    naggpudevMrg32k3aDeviceInitA(0, 0, 0, 0, offset, seed,
                                &devComm, &comm, &error);
    checkNagError(&error);

    // Copy seed to device
    cuError = cudaMemcpy(d_seed, seed, sizeof(unsigned int)*6,
                        cudaMemcpyHostToDevice);
}

```

```

checkCudaError(cuError);

// Launch GPU computation asynchronously. Feel free to experiment
// with nblks=#thread blocks, nthds=#threads/block & ppt=#points/thread
const int ppt = 1024;
const int nthds = 64;
const int nblks = N/(ppt*nthds) + 1;
mrg32k3a_device_functions_exponential_test<<<nblks, nthds>>>
(ppt, N, d_buff, d_seed, devComm);

/*
 * One can now launch other kernels to operate on the random numbers,
 * or copy the numbers to the host and operate on them there.
 * Here we simply copy them to the host in order to print them
 */
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The first 50 GPU random numbers:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(4);
for(int row = 0; row < 10; row++)
{
    for(int col = 0; col < 5; col++)
    {
        cout << h_buff[row*10 + col] << "\t";
    }
    cout << endl;
}

// Call cleanup for the NAG routine
naggpuMrg32k3aDeviceCleanupA(devComm, &comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}
if (d_seed)
{
    cuError = cudaFree(d_seed);
    checkCudaError(cuError);
}

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

```

```
void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}
```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpudevMrg32k3aExpA, single precision

The first 50 GPU random numbers:

0.0000	0.7015	0.3593	1.0333	0.0692
1.9998	1.5521	2.3616	0.6293	2.1551
0.9912	0.9287	1.0029	3.8324	0.4377
0.2346	0.2850	1.8797	0.8697	0.0256
0.1589	0.9164	0.2278	0.6564	0.1516
0.1700	0.7241	1.1626	0.2227	0.3614
0.2547	0.5586	0.0025	0.5932	7.7862
1.4530	1.1058	1.4885	0.4088	0.2544
0.8717	0.5517	0.0337	0.9418	0.5588
2.1155	1.3872	3.2928	1.2697	1.4017

---



# NAG Numerical Routines for GPUs Function Document

## naggpudevMrg32k3aGammaA

### 1 Purpose

**naggpudevMrg32k3aGammaA** generates the next value from a gamma distribution with shape parameter  $\alpha$  and scale parameter  $\beta$ . The function **naggpudevMrg32k3aGammaSetParamsA** must be called first to set the parameters of the distribution and to compute certain constants which are used by this function. Each time the parameters of the distribution change, **naggpudevMrg32k3aGammaSetParamsA** must be called to recompute these constants, which must then be passed to **naggpudevMrg32k3aGammaA**.

The initialization function **naggpudevMrg32k3aInitA** must be called prior to the first call to **naggpudevMrg32k3aGammaA**. Thereafter, this function may be called repeatedly to generate the next value in the sequence. Care should be taken to ensure that sequences from successive CUDA threads do not overlap.

### 2 Specification

```
#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

template <typename FP>
__device__ void naggpudevMrg32k3aGammaA(FP &x, unsigned int *state,
    FP gammaComm1, FP gammaComm2, FP gammaComm3, FP gammaComm4, FP &normalComm)
```

### 3 Description

The gamma distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\alpha, \beta > 0$ . The rejection algorithm described in Marsaglia and Tsang (2000) is used to generate the gamma pseudorandom variates when  $\alpha \geq 1$ . When  $0 < \alpha < 1$ , the scaling

$$\gamma_\alpha = \gamma_{1+\alpha} U^\alpha$$

is used where  $U$  denotes a uniform random variable in the interval  $[0, 1]$  and  $\gamma_\alpha$  denotes a gamma random variable with shape parameter  $\alpha$  and scale parameter  $\beta = 1$ .

This function uses a rejection algorithm to generate variates from the required distribution. A key feature of rejection algorithms is that a *random number* of uniform variates is required to generate a single variate from the target distribution. Creating independent generators for use in a parallel setting is therefore not simple: for each of these generators one does not know how far to skip ahead the state to ensure that the generators do not overlap. Care must also be taken to ensure the independent generators preserve the good statistical properties of the underlying uniform generator. The recommended way of using **naggpudevMrg32k3aGammaA** is to place each CUDA thread on a different *independent stream or substream*. Please see the Random Number Generators Chapter Introduction for recommendations on choices of streams and substreams, and **naggpuMrg32k3aDeviceInitA** and **naggpudevMrg32k3aInitA** for details on how to create such streams and substreams.

Using **naggpudevMrg32k3aGammaA** will lead to a certain amount of *warp divergence* (please see NVIDIA CUDA (2011) for a thorough discussion of this concept and its performance implications). The probability of warp divergence is closely linked to the acceptance probability of the rejection algorithm, which in the case of Marsaglia and Tsang (2000) is approximately 0.951, 0.981 and 0.992 when  $\alpha = 1, 2$  and 4 respectively. In the worst case when  $\alpha = 1$ , the probability of warp divergence is slightly less than  $1 - 0.951^{32} = 0.80$ . However the separate code branches executed in divergent warps are relatively small and not computationally demanding, so that the impact of the divergence should not be great.

### 3.1 Error Handling

This is a GPU device function, and no error handling is performed.

## 4 References

Marsaglia G and Tsang W W (2000) A simple method for generating Gamma variables *ACM Trans. Math. Software* **26(3)** 363–372

NVIDIA CUDA (2011) *Programming Guide Version 4.0* <http://www.nvidia.com/cuda>

## 5 Arguments

- 1: **x** – FP & *Output*  
 The template argument parameter FP can take type double or float.  
*On exit:* the next pseudorandom value from the sequence.
- 2: **state** – unsigned int \* *Input/Output*  
*On entry:* the output state from the previous call to any of the MRG32k3a device function generators, or if this is the first call to any of the MRG32k3a device generators, the value of **seed** obtained from the initialization call naggpudevMrg32k3aInitA  
*On exit:* the output state of the generator
- 3: **gammaComm1** – FP *Communication Data*  
 The template argument parameter FP can take type double or float.  
 This parameter is for internal use and must not be modified in any way.  
*On entry:* the value of **gammaComm1** obtained from naggpudevMrg32k3aGammaSetParamsA
- 4: **gammaComm2** – FP *Communication Data*  
 The template argument parameter FP can take type double or float.  
 This parameter is for internal use and must not be modified in any way.  
*On entry:* the value of **gammaComm2** obtained from naggpudevMrg32k3aGammaSetParamsA
- 5: **gammaComm3** – FP *Communication Data*  
 The template argument parameter FP can take type double or float.  
 This parameter is for internal use and must not be modified in any way.  
*On entry:* the value of **gammaComm3** obtained from naggpudevMrg32k3aGammaSetParamsA
- 6: **gammaComm4** – FP *Communication Data*  
 The template argument parameter FP can take type double or float.  
 This parameter is for internal use and must not be modified in any way.  
*On entry:* the value of **gammaComm4** obtained from naggpudevMrg32k3aGammaSetParamsA
- 7: **normalComm** – FP & *Communication Data*  
 The template argument parameter FP can take type double or float.  
 This parameter is for internal use and must not be modified in any way.  
 The argument **normalComm** contains information required by both naggpudevMrg32k3aNormalA and naggpudevMrg32k3aGammaA. If both these functions are referenced in a GPU kernel, a single instance of **normalComm** must be created and must be used when calling either function. On the very first call to either naggpudevMrg32k3aNormalA or naggpudevMrg32k3aGammaA,

**normalComm** must be set to (FP)nanf(""). Thereafter the variable must be passed unchanged to every subsequent call to these functions.

## 6 Error Indicators and Warnings

None.

## 7 Example

This example program uses **naggpudevMrg32k3aGammaA** to print 50 pseudorandom numbers from a gamma distribution using the MRG32k3a device function generator.

### 7.1 Program Text

```

/*
 * Example Program: naggpudevMrg32k3aGammaA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

/*
 * Test kernel to try out device level functions
 * Note: this will NOT output numbers in the same order
 * as the CPU routine
 */
__global__ void mrg32k3a_device_functions_gamma_test(
    int ppt,          // Number of points per thread
    int N,           // Total number of points
    FP *d_buff,      // memory address to store points
    unsigned int *d_seed,
    NagGpuMrg32k3aDeviceComm *devComm)
{
    /*
     * Assumptions:
     * Grid is one dimensional
     * Thread block is one dimensional
     */
    const FP alpha = FP(1.3);
    const FP beta = FP(3.7);

    FP gammaComm1, gammaComm2, gammaComm3, gammaComm4;
    FP normComm = (FP)nanf("");

    unsigned int state[6];
    for (int i = 0; i < 6; i++) state[i] = d_seed[i];

    // Initialise state for this thread

```

```

// Place each CUDA thread on a separate substream
int tid = threadIdx.x + blockDim.x*blockIdx.x;
naggpudevMrg32k3aInitA(tid, 76, 0, 0, 0, state, devComm);

// Storage offset
int store = threadIdx.x + ppt*blockIdx.x*blockDim.x;

// Set parameters of gamma distribution
naggpudevMrg32k3aGammaSetParamsA(alpha, beta,
    gammaComm1, gammaComm2, gammaComm3, gammaComm4);

FP x = 0;
// Compute only as many points as requested
for (int i = 0; i < ppt; i++)
{
    if (store < N)
    {
        naggpudevMrg32k3aGammaA(x, state,
            gammaComm1, gammaComm2, gammaComm3, gammaComm4, normComm);

        d_buff[store] = x;
        store += blockDim.x;
    }
    else
    {
        break;
    }
}
}

int main(int argc, char **argv)
{
    const int N = 2621440;

    FP *h_buff = 0, *d_buff = 0;

    unsigned int seed[] = {1, 2, 4, 1, 2, 3};
    unsigned int *d_seed;

    NagGpuMrg32k3aDeviceComm *devComm;
    NagGpuRandComm comm;
    NagGpuError error;
    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpudevMrg32k3aGammaA, ";
    if (sizeof(FP)==sizeof(float))
        cout << "single precision";
    else
        cout << "double precision";
    cout << endl << endl;

    // Allocate CPU and GPU memory
    h_buff = new FP[N];
    cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
    checkCudaError(cuError);
    cuError = cudaMalloc((void **)&d_seed, sizeof(unsigned int) * 6);
    checkCudaError(cuError);

    // Host initialisation call
    naggpudevMrg32k3aDeviceInitA(0, 0, 0, 0, 0, seed,
        &devComm, &comm, &error);
    checkNagError(&error);

    // Copy seed to device
    cuError = cudaMemcpy(d_seed, seed, sizeof(unsigned int)*6,

```

```

        cudaMemcpyHostToDevice);
checkCudaError(cuError);

// Launch GPU computation asynchronously. Feel free to experiment
// with nblks=#thread blocks, nthds=#threads/block & ppt=#points/thread
const int ppt = 1024;
const int nthds = 64;
const int nblks = N/(ppt*nthds) + 1;
mrg32k3a_device_functions_gamma_test<<<nblks, nthds>>>
(ppt, N, d_buff, d_seed, devComm);

/*
 * One can now launch other kernels to operate on the random numbers,
 * or copy the numbers to the host and operate on them there.
 * Here we simply copy them to the host in order to print them
 */
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The first 50 GPU random numbers:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(4);
for(int row = 0; row < 10; row++)
{
    for(int col = 0; col < 5; col++)
    {
        cout << h_buff[row*10 + col] << "\t";
    }
    cout << endl;
}

// Call cleanup for the NAG routine
naggpuMrg32k3aDeviceCleanupA(devComm, &comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}
if (d_seed)
{
    cuError = cudaFree(d_seed);
    checkCudaError(cuError);
}

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

```

```
void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}
```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpudevMrg32k3aGammaA, single precision

The first 50 GPU random numbers:  
3.6403 0.9104 6.8153 4.3084 4.1830  
0.7710 10.8153 16.9506 1.5821 1.8948  
2.9317 4.6541 0.2354 1.6106 3.0009  
4.0389 3.4582 11.5590 1.3871 3.2133  
1.2653 2.6843 4.3136 5.7665 2.3449  
5.5406 4.3038 5.9099 1.9624 0.8370  
0.4379 7.3254 4.8952 11.3394 2.7389  
0.6569 5.6433 0.7745 0.0692 0.1111  
2.4630 2.6579 0.5987 4.1963 4.0539  
6.5393 11.2430 5.1479 0.3441 4.6715

---

# NAG Numerical Routines for GPUs Function Document

## naggpudevMrg32k3aGammaSetParamsA

### 1 Purpose

**naggpudevMrg32k3aGammaSetParamsA** accepts the shape parameter  $\alpha$  and scale parameter  $\beta$  of a gamma distribution and computes constants which are used by the generator function **naggpudevMrg32k3aGammaA**.

### 2 Specification

```
#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

template <typename FP>
__device__ void naggpudevMrg32k3aGammaSetParamsA(FP alpha, FP beta,
          FP &gammaComm1, FP &gammaComm2, FP &gammaComm3, FP &gammaComm4)
```

### 3 Description

The function **naggpudevMrg32k3aGammaA** uses certain constants which only change when the parameters of the gamma distribution change. For performance reasons, these constants should only be recomputed when necessary and not each time **naggpudevMrg32k3aGammaA** is called. This function should therefore be called before the first call to **naggpudevMrg32k3aGammaA** and each time the gamma distribution's parameters change, and the output must be passed to **naggpudevMrg32k3aGammaA**.

#### 3.1 Error Handling

This is a GPU device function, and no error handling is performed.

### 4 References

None.

### 5 Arguments

- |    |  |               |
|----|--|---------------|
| 1: | <b>alpha</b> – FP<br>The template argument parameter FP can take type double or float.<br><i>On entry:</i> the shape parameter, $\alpha$ , of the distribution.<br><i>Constraint:</i> $\alpha > 0$ . | <i>Input</i>  |
| 2: | <b>beta</b> – FP<br>The template argument parameter FP can take type double or float.<br><i>On entry:</i> the scale parameter, $\beta$ , of the distribution.<br><i>Constraint:</i> $\beta > 0$ .    | <i>Input</i>  |
| 3: | <b>gammaComm1</b> – FP &<br>The template argument parameter FP can take type double or float.<br><i>On exit:</i> a constant which must be passed to <b>naggpudevMrg32k3aGammaA</b> .                 | <i>Output</i> |

- 4: **gammaComm2** – FP & *Output*  
The template argument parameter FP can take type double or float.  
*On exit:* a constant which must be passed to naggpudevMrg32k3aGammaA.
- 5: **gammaComm3** – FP & *Output*  
The template argument parameter FP can take type double or float.  
*On exit:* a constant which must be passed to naggpudevMrg32k3aGammaA.
- 6: **gammaComm4** – FP & *Output*  
The template argument parameter FP can take type double or float.  
*On exit:* a constant which must be passed to naggpudevMrg32k3aGammaA.

## 6 Error Indicators and Warnings

None.

## 7 Example

There is no example program specifically for this function. For an example of how this function should be used, please see the example program for naggpudevMrg32k3aGammaA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpudevMrg32k3aNormalA

### 1 Purpose

**naggpudevMrg32k3aNormalA** generates the next value from a Normal distribution with mean  $\mu$  and variance  $\sigma^2$ .

The initialization function **naggpudevMrg32k3aInitA** must be called prior to the first call to **naggpudevMrg32k3aNormalA**. Thereafter, this function may be called repeatedly to generate the next value in the sequence. Care should be taken to ensure that sequences from successive CUDA threads do not overlap.

### 2 Specification

```
#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

template <typename FP>
__device__ void naggpudevMrg32k3aNormalA(FP &x, FP mu, FP sigma,
    unsigned int *state, FP &normalComm)
```

### 3 Description

The Normal distribution has probability density function given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where  $\sigma > 0$  and  $\mu \in \mathbb{R}$ . This function uses a Box-Muller transform to convert a pair of uniform  $(0, 1)$  random numbers into a pair of Normal random numbers. Let  $X_0, X_1, X_2, \dots$  denote the sequence of uniform  $(0, 1)$  pseudorandom variates as specified by the MRG32k3a algorithm. This function uses successive pairs of uniform variates in the Box-Muller transform to produce successive pairs of Normal variates, i.e.  $(X_0, X_1) \mapsto (Z_0, Z_1)$ ,  $(X_2, X_3) \mapsto (Z_2, Z_3)$  where  $Z_0, Z_1, Z_2, \dots$  denotes the output sequence of Normal variates.

#### 3.1 Error Handling

This is a GPU device function, and no error handling is performed.

### 4 References

None.

### 5 Arguments

- |    |   |               |
|----|---|---------------|
| 1: | <b>x</b> – FP &   | <i>Output</i> |
|    | The template argument parameter FP can take type double or float.<br><i>On exit:</i> the next pseudorandom value from the sequence. |               |
| 2: | <b>mu</b> – FP  | <i>Input</i>  |
|    | The template argument parameter FP can take type double or float.<br><i>On entry:</i> the mean, $\mu$ , of the distribution.        |               |

- 3: **sigma** – FP *Input*  
 The template argument parameter FP can take type double or float.  
*On entry:* the standard deviation,  $\sigma$ , of the distribution  
*Constraint:*  $\text{sigma} > 0$ .
- 4: **state** – unsigned int \* *Input/Output*  
*On entry:* the output state from the previous call to any of the MRG32k3a device function generators, or if this is the first call to any of the MRG32k3a device generators, the value of **seed** obtained from the initialization call naggpudevMrg32k3aInitA  
*On exit:* the output state of the generator
- 5: **normalComm** – FP & *Communication Data*  
 The template argument parameter FP can take type double or float.  
 This parameter is for internal use and must not be modified in any way.  
 The argument **normalComm** contains information required by both naggpudevMrg32k3aNormalA and naggpudevMrg32k3aGammaA. If both these functions are referenced in a GPU kernel, a single instance of **normalComm** must be created and must be used when calling either function. On the very first call to either naggpudevMrg32k3aNormalA or naggpudevMrg32k3aGammaA, **normalComm** must be set to (FP)nanf(""). Thereafter the variable must be passed unchanged to every subsequent call to these functions.

## 6 Error Indicators and Warnings

None.

## 7 Example

This example program uses **naggpudevMrg32k3aNormalA** to print 50 pseudorandom numbers from a Normal distribution using the MRG32k3a device function generator.

### 7.1 Program Text

```

/*
 * Example Program: naggpu_devMrg32k3a_normalA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

/*

```

```

* Test kernel to try out device level functions
* Note: this will NOT output numbers in the same order
* as the CPU routine
*/
__global__ void mrg32k3a_device_functions_normal_test(
    int npts,          // Number of points per thread
    int N,             // Total number of points to create
    FP *d_buff,        // memory address to store points
    unsigned int *d_seed,
    NagGpuMrg32k3aDeviceComm *devComm)
{
    /*
    * Assumptions:
    * Grid is one dimensional
    * Thread block is one dimensional
    */

    const FP mu = 0.0, sigma = 1.0;
    FP normComm = (FP)nanf("");

    unsigned int state[6];
    for (int i = 0; i < 6; i++) state[i] = d_seed[i];

    // Generation offset
    int offset = ((blockIdx.x * blockDim.x) + threadIdx.x)*npts;

    // Storage offset
    int store = threadIdx.x + npts*blockIdx.x*blockDim.x;

    FP x = 0;

    // Initialise state for this thread
    naggpudevMrg32k3aInitA(0, 0, 0, 0, offset, state, devComm);

    // Compute only as many points as requested
    for (int i = 0; i < npts; i++)
    {
        if (store < N)
        {
            naggpudevMrg32k3aNormalA(x, mu, sigma, state, normComm);
            d_buff[store] = x;
            store += blockDim.x;
        }
        else
        {
            break;
        }
    }
}

int main(int argc, char **argv)
{
    const int N = 2621440;

    FP *h_buff = 0, *d_buff = 0;

    unsigned int seed[] = {1, 2, 3, 1, 2, 3};
    unsigned int *d_seed;

    unsigned long long offset = 1;

    NagGpuMrg32k3aDeviceComm *devComm;
    NagGpuRandComm comm;
    NagGpuError error;
    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpudevMrg32k3aNormalA, ";

```

```

if (sizeof(FP)==sizeof(float))
    cout << "single precision";
else
    cout << "double precision";
cout << endl << endl;

// Allocate CPU and GPU memory
h_buff = new FP[N];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
checkCudaError(cuError);
cuError = cudaMalloc((void **)&d_seed, sizeof(unsigned int) * 6);
checkCudaError(cuError);

// Host initialisation call
naggpuMrg32k3aDeviceInitA(0, 0, 0, 0, offset, seed,
                          &devComm, &comm, &error);
checkNagError(&error);

// Copy seed to device
cuError = cudaMemcpy(d_seed, seed, sizeof(unsigned int)*6,
                    cudaMemcpyHostToDevice);
checkCudaError(cuError);

// Launch GPU computation asynchronously. Feel free to experiment
// with nblks=#thread blocks, nthds=#threads/block & ppt=#points/thread
const int ppt = 1024;
const int nthds = 64;
const int nblks = N/(ppt*nthds) + 1;
mrg32k3a_device_functions_normal_test<<<nblks, nthds>>>
(ppt, N, d_buff, d_seed, devComm);

/*
 * One can now launch other kernels to operate on the random numbers,
 * or copy the numbers to the host and operate on them there.
 * Here we simply copy them to the host in order to print them
 */
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The first 50 GPU random numbers:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(4);
for(int row = 0; row < 10; row++)
{
    for(int col = 0; col < 5; col++)
    {
        cout << h_buff[row*10 + col] << "\t";
    }
    cout << endl;
}

// Call cleanup for the NAG routine
naggpuMrg32k3aDeviceCleanupA(devComm, &comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}
if (d_seed)

```

```
    {
        cuError = cudaFree(d_seed);
        checkCudaError(cuError);
    }

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}
```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpudevMrg32k3aNormalA, single precision

The first 50 GPU random numbers:  
0.0000 0.3201 -0.7592 -0.4116 -0.0888  
-1.9745 -0.3208 -2.0949 -0.1084 1.4794  
-0.6145 0.9015 0.2775 -1.4955 0.7982  
0.6848 -0.5189 -0.2508 -0.4190 0.0253  
0.5189 0.8332 -0.5497 -1.0040 0.2467  
-0.0231 -1.1487 -1.3612 0.4385 0.1324  
-0.6566 -0.9416 -0.0496 0.8405 0.0003  
0.1454 -0.4827 0.1190 -0.5212 0.3178  
-1.5864 -0.5816 0.7089 -0.5835 -1.2668  
2.0041 0.0033 0.9920 -0.1001 0.0164

---



# NAG Numerical Routines for GPUs Function Document

## naggpudevMrg32k3aUniformA

### 1 Purpose

**naggpudevMrg32k3aUniformA** generates the next value from a uniform distribution over the interval  $(a, b]$  for specified constants  $a$  and  $b$ .

The initialization function **naggpudevMrg32k3aInitA** must be called prior to the first call to **naggpudevMrg32k3aUniformA**. Thereafter, this function may be called repeatedly to generate the next value in the sequence. Care should be taken to ensure that sequences from successive CUDA threads do not overlap.

### 2 Specification

```
#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

template <typename FP>
__device__ void naggpudevMrg32k3aUniformA(FP &x, FP a, FP b, unsigned int *state)
```

### 3 Description

If  $a = 0$  and  $b = 1$ , this function returns the next value  $Y$  from the MRG32k3a uniform  $(0, 1]$  sequence. For other values of  $a$  and  $b$ , the function applies the transformation

$$X = a + (b - a)Y$$

to produce random numbers from the interval  $(a, b]$ .

#### 3.1 Error Handling

This is a GPU device function, and no error handling is performed.

### 4 References

None.

### 5 Arguments

- |    |  |               |
|----|--|---------------|
| 1: | <b>x</b> – FP &  | <i>Output</i> |
|    | The template argument parameter FP can take type double or float.<br><i>On exit:</i> the next pseudorandom value from the sequence.                                  |               |
| 2: | <b>a</b> – FP  | <i>Input</i>  |
|    | The template argument parameter FP can take type double or float.<br><i>On entry:</i> The lower bound for the uniform random values.                                 |               |
| 3: | <b>b</b> – FP  | <i>Input</i>  |
|    | The template argument parameter FP can take type double or float.<br><i>On entry:</i> The upper bound for the uniform random values.<br><i>Constraint:</i> $b > a$ . |               |

4: **state** – unsigned int \*

*Input/Output*

*On entry:* the output state from the previous call to any of the MRG32k3a device function generators, or if this is the first call to any of the MRG32k3a device generators, the value of **seed** obtained from the initialization call naggpudevMrg32k3aInitA

*On exit:* the output state of the generator

## 6 Error Indicators and Warnings

None.

## 7 Example

This example program uses **naggpudevMrg32k3aUniformA** to print 50 pseudorandom numbers from a uniform distribution using the MRG32k3a device function generator.

### 7.1 Program Text

```

/*
 * Example Program: naggpu_devMrg32k3a_uniformA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_mrg32k3aDevFuncs.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

/*
 * Test kernel to try out device level functions
 * Note: this will NOT output numbers in the same order
 * as the CPU routine
 */
__global__ void mrg32k3a_device_functions_uniform_test(
    int npts,      // Number of points per thread
    int N,        // Total number of points to create
    FP *d_buff,   // memory address to store points
    unsigned int *d_seed,
    NagGpuMrg32k3aDeviceComm *devComm)
{
    /*
     * Assumptions:
     * Grid is one dimensional
     * Thread block is one dimensional
     */

    const FP a = 0.0, b = 1.0;

    unsigned int state[6];
    for (int i = 0; i < 6; i++) state[i] = d_seed[i];

```

```

// Generation offset
int offset = ((blockIdx.x * blockDim.x) + threadIdx.x)*npts;

// Storage offset
int store = threadIdx.x + npts*blockIdx.x*blockDim.x;

FP x = 0;

// Initialise state for this thread
naggpudevMrg32k3aInitA(0, 0, 0, 0, offset, state, devComm);

// Compute only as many points as requested
for (int i = 0; i < npts; i++)
{
    if (store < N)
    {
        naggpudevMrg32k3aUniformA(x, a, b, state);
        d_buff[store] = x;
        store += blockDim.x;
    }
    else
    {
        break;
    }
}
}

int main(int argc, char **argv)
{
    const int N = 2621440;

    FP *h_buff = 0, *d_buff = 0;

    unsigned int seed[] = {1, 2, 3, 1, 2, 3};
    unsigned int *d_seed;

    unsigned long long offset = 1;

    NagGpuMrg32k3aDeviceComm *devComm;
    NagGpuRandComm comm;
    NagGpuError error;
    cudaError_t cuError;

    // Print the title
    cout << "NAG GPU Example Program: ";
    cout << "naggpudevMrg32k3aUniformA, ";
    if (sizeof(FP)==sizeof(float))
        cout << "single precision";
    else
        cout << "double precision";
    cout << endl << endl;

    // Allocate CPU and GPU memory
    h_buff = new FP[N];
    cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*N);
    checkCudaError(cuError);
    cuError = cudaMalloc((void **)&d_seed, sizeof(unsigned int) * 6);
    checkCudaError(cuError);

    // Host initialisation call
    naggpudevMrg32k3aDeviceInitA(0, 0, 0, 0, offset, seed,
                                &devComm, &comm, &error);
    checkNagError(&error);

    // Copy seed to device
    cuError = cudaMemcpy(d_seed, seed, sizeof(unsigned int)*6,
                        cudaMemcpyHostToDevice);
}

```

```

checkCudaError(cuError);

// Launch GPU computation asynchronously. Feel free to experiment
// with nblks=#thread blocks, nthds=#threads/block & ppt=#points/thread
const int ppt = 1024;
const int nthds = 64;
const int nblks = N/(ppt*nthds) + 1;
mrg32k3a_device_functions_uniform_test<<<nblks, nthds>>>
(ppt, N, d_buff, d_seed, devComm);

/*
 * One can now launch other kernels to operate on the random numbers,
 * or copy the numbers to the host and operate on them there.
 * Here we simply copy them to the host in order to print them
 */
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*N,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The first 50 GPU random numbers:" << endl;
cout.setf(ios::fixed, ios::floatfield);
cout.precision(4);
for(int row = 0; row < 10; row++)
{
    for(int col = 0; col < 5; col++)
    {
        cout << h_buff[row*10 + col] << "\t";
    }
    cout << endl;
}

// Call cleanup for the NAG routine
naggpuMrg32k3aDeviceCleanupA(devComm, &comm, &error);
checkNagError(&error);

// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}
if (d_seed)
{
    cuError = cudaFree(d_seed);
    checkCudaError(cuError);
}

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

```

```
void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}
```

## 7.2 Program Data

None.

## 7.3 Program Results

NAG GPU Example Program: naggpudevMrg32k3aUniformA, single precision

The first 50 GPU random numbers:

1.0000	0.4958	0.6982	0.3558	0.9332
0.1354	0.2118	0.0943	0.5330	0.1159
0.3711	0.3951	0.3668	0.0217	0.6455
0.7909	0.7520	0.1526	0.4191	0.9747
0.8531	0.3999	0.7963	0.5187	0.8593
0.8437	0.4848	0.3127	0.8003	0.6967
0.7751	0.5720	0.9975	0.5525	0.0004
0.2339	0.3310	0.2257	0.6644	0.7754
0.4182	0.5760	0.9669	0.3899	0.5719
0.1206	0.2498	0.0371	0.2809	0.2462

---



# NAG Numerical Routines for GPUs Function Document

## naggpuMrg32k3aDeviceCleanupA

### 1 Purpose

**naggpuMrg32k3aDeviceCleanupA** frees system resources that were allocated by a previous call to **naggpuMrg32k3aDeviceInitA**.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuMrg32k3aDeviceCleanupA(NagGpuMrg32k3aDeviceComm *devComm,
    NagGpuRandComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- 1: **devComm** – NagGpuMrg32k3aDeviceComm \* *Input*  
*On entry:* the GPU memory address obtained from a prior call to **naggpuMrg32k3aDeviceInitA**.
- 2: **comm** – NagGpuRandComm \* *Communication Data*  
*On entry:* the pointer that was passed to a previous call to **naggpuMrg32k3aDeviceInitA**.
- 3: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:* **devComm** is NULL.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for `naggpudevMrg32k3aUniformA`.

---

# NAG Numerical Routines for GPUs Function Document

## naggpuSobolDeviceInitA

### 1 Purpose

**naggpuSobolDeviceInitA** creates initialization data on the GPU for the Sobol' device function generators. It must be called prior to a call to **naggpudevSobolInitA** and must ultimately be followed by a call to **naggpuSobolDeviceCleanupA** to free allocated system resources.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuSobolDeviceInitA(NagGpuScramTypes stype, int maxDevGenDim,
    NagGpuSobolDeviceComm **devComm, NagCPURandComm *pseudoComm,
    NagGpuQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

To use the Sobol' device function generators, the memory address of a **NagGpuSobolDeviceComm** structure must be obtained from the device generator initialization function **naggpuSobolDeviceInitA**. This structure will reside in the GPU memory space and will contain communication data for use by the device function generator. This GPU memory address must then be passed to the Sobol' device generator initialization function **naggpudevSobolInitA**. Once all values have been obtained from the device function generators, the same **NagGpuSobolDeviceComm** structure memory address must be passed to **naggpuSobolDeviceCleanupA** to free allocated system resources.

This function will create the **NagGpuSobolDeviceComm** communication structure in the GPU memory space and will assign its address to **devComm**. It also requires the maximum dimension **maxDevGenDim** of any Sobol' sequence which is to be generated by the GPU device function generators. Using the device function generators to generate sequences of dimension higher than **maxDevGenDim** will result in undefined behaviour.

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or **cudaSuccess** if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- 1: **stype** – NagGpuScramTypes *Input*  
*On entry:* the type of scrambling to be used:  
 stype = NAGGPUSCRAMTYPES\_NONE  
 stype = NAGGPUSCRAMTYPES\_OWEN  
 stype = NAGGPUSCRAMTYPES\_FAURE.TEZUKA

stype = NAGGPUSCRAMTYPES\_OWEN\_FAURE\_TEZUKA

Please see NagGpuScramTypes for some of the benefits of scrambling and details about each of available scrambling types.

*Constraint:* stype = NAGGPUSCRAMTYPES\_NONE or  
NAGGPUSCRAMTYPES\_OWEN or  
NAGGPUSCRAMTYPES\_FAURE\_TEZUKA or  
NAGGPUSCRAMTYPES\_OWEN\_FAURE\_TEZUKA

2: **maxDevGenDim** – int *Input*

*On entry:* the largest dimension of any quasi-random sequence which will be generated by the Sobol' device function generators such as naggpudevSobolUniformA.

*Constraint:*  $1 \leq \text{maxDevGenDim} \leq 50000$ .

3: **devComm** – NagGpuSobolDeviceComm \*\* *Communication Data*

NagGpuSobolDeviceComm is a structure which holds state and communication information and must not be modified in any way. A structure will be allocated in the GPU memory space and its address will be assigned to the location pointed to by **devComm**. This address must be passed to the Sobol device generator initialization function naggpudevSobolInitA. Once all required points have been obtained, this address must be passed to naggpuSobolDeviceCleanupA to free allocated system resources.

4: **pseudoComm** – NagCPURandComm \* *Input*

*On entry:* a pointer to a NagCPURandComm structure which has already been initialized by the function nagCPURandInitA.

*Constraint:* **pseudoComm** must be initialized before being passed to this function .

5: **comm** – NagGpuQuasiRandComm \* *Communication Data*

NagGpuQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained from the Sobol' device function generators, **comm** must be passed to naggpuSobolDeviceCleanupA to free allocated system resources.

6: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 111

*On entry:* **stype** does not specify a valid scrambling type. See NagGpuScramTypes for permitted values.

error → code = 114

*On entry:* the value of **pseudoComm** is NULL.

error → code = 115

*On entry:* the pseudorandom generator nagCPURandUniformA returned an error when called by this function: **pseudoComm** is not initialized, or the internal state of **pseudoComm** is corrupted.

error → code = 116

*On entry:* the value of **devComm** is NULL.

error → code = 117

*On entry:* the value of **maxDevGenDim** does not satisfy the constraint listed above.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpudevSobolUniformA.

---



# NAG Numerical Routines for GPUs Function Document

## naggpudevSobolInitA

### 1 Purpose

**naggpudevSobolInitA** initializes the Sobol' device function generator. It must be preceded by a call to **naggpuSobolDeviceInitA** and must ultimately be followed by a call to **naggpuSobolDeviceCleanupA** to free allocated system resources. **naggpudevSobolInitA** must be called prior to calling any of the Sobol' device generator functions such as **naggpudevSobolUniformA**.

### 2 Specification

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

__device__ void naggpudevSobolInitA(int dim, unsigned int offset, int &comm1,
    unsigned int &comm2, unsigned int *comm3,
    const NagGpuSobolDeviceComm *devComm)
```

### 3 Description

The Sobol' generator is an efficient quasi-random generator which is relatively light on resources. As such it is natural to try to embed it within a GPU kernel. A per-thread parallelization strategy has been adopted, whereby each CUDA thread has an independent Sobol' generator. Starting points are controlled through the **offset** parameter.

High dimensional Sobol' sequences require a lot of storage for each quasi-random point. Since shared memory on the GPU is limited, it may be necessary to store these points in local memory. On the older NVIDIA hardware this will result in relatively poor performance, however on the newer 'Fermi' architecture the performance should be much better. Please see the section on Local Memory in the CUDA Programming Guide.

### 4 Error Handling

This is a GPU device function, and no error handling is performed.

### 5 References

None.

### 6 Arguments

- 1: **dim** – int *Input*  
*On entry:* the dimension of the Sobol' sequence to be generated.  
*Constraint:*  $1 \leq \text{dim} \leq D$  where  $D$  denotes the value of the **maxDevGenDim** parameter in the initialization call to **naggpuSobolDeviceInitA**.
- 2: **offset** – unsigned int *Input*  
*On entry:* the offset into the Sobol' sequence at which to start generating.
- 3: **comm1** – int & *Communication Data*  
the parameter will be initialized with communication information and must be passed to subsequent calls to the device function generators such as **naggpudevSobolUniformA**. It must not be modified in any way.

- 4: **comm2** – unsigned int & *Communication Data*  
the parameter will be initialized with communication information and must be passed to subsequent calls to the device function generators such as naggpudevSobolUniformA. It must not be modified in any way.
- 5: **comm3[dim]** – unsigned int \* *Communication Array*  
the parameter will be initialized with communication information and must be passed to subsequent calls to the device function generators such as naggpudevSobolUniformA. It must not be modified in any way. This array will contain state information which will be accessed frequently.
- 6: **devComm** – const NagGpuSobolDeviceComm \* *Communication Data*  
The NagGpuSobolDeviceComm memory address obtained from the host initialization function naggpuSobolDeviceInitA.

## 7 Error Indicators and Warnings

None.

## 8 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for naggpudevSobolUniformA.

---

# NAG Numerical Routines for GPUs Function Document

## naggpudevSobolExpA

### 1 Purpose

**naggpudevSobolExpA** generates the next point from a quasi-random exponential distribution with parameter  $\lambda$ .

The initialization function **naggpudevSobolInitA** must be called prior to the first call to **naggpudevSobolExpA**. Thereafter, this function may be called repeatedly to generate the next point in the sequence. Care should be taken to ensure that sequences from successive CUDA threads do not overlap.

### 2 Specification

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

template <typename FP>
__device__ void naggpudevSobolExpA(FP *x, FP lambda, const int comm1,
    unsigned int &comm2, unsigned int *comm3,
    const NagGpuSobolDeviceComm *devComm)
```

### 3 Description

Sobol' sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling **naggpudevSobolInitA** to initialize the generator. Below we will consider a  $d$ -dimensional Sobol' sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

The exponential distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\lambda} e^{-x/\lambda} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\lambda > 0$ . This function returns the next point  $x = (x_1, x_2, \dots, x_d)$  where

$$x_i = -\lambda \ln(y_i + 2^{-32})$$

for each  $i = 1, 2, \dots, d$  and  $y = (y_1, y_2, \dots, y_d) \in [0, 1)^d$  is the next point in the Sobol' sequence.

### 4 Error Handling

This is a GPU device function, and no error handling is performed.

### 5 References

None.

### 6 Arguments

1:  $\mathbf{x}[d]$  – FP \* *Output*

The template argument parameter FP can take type double or float.

The value  $d$  is the dimension of the sequence as specified to the initialization function **naggpudevSobolInitA**.

*On exit:* the next  $d$ -dimensional point from the specified distribution.

- 2: **lambda** – FP *Input*  
 The template argument parameter FP can take type double or float.  
*On entry:* the mean,  $\lambda$ , of the exponential distribution.  
*Constraint:* lambda > 0.
- 3: **comm1** – const int *Communication Data*  
 The value of **comm1** from the initialization call naggpudevSobolInitA.
- 4: **comm2** – unsigned int & *Communication Data*  
 The value of **comm2** obtained from the previous call to any of the Sobol' device function generators, or if this is the first call to any of the Sobol' device function generators, the value of **comm2** from the initialization call naggpudevSobolInitA.
- 5: **comm3**[*d*] – unsigned int \* *Communication Array*  
 The value *d* is the dimension of the sequence as specified to the initialization function naggpudevSobolInitA.  
 The value of **comm3** obtained from the previous call to any of the Sobol' device function generators, or if this is the first call to any of the Sobol' device function generators, the value of **comm3** from the initialization call naggpudevSobolInitA.
- 6: **devComm** – const NagGpuSobolDeviceComm \* *Communication Data*  
 The NagGpuSobolDeviceComm memory address obtained from the host initialization function naggpuSobolDeviceInitA.

## 7 Error Indicators and Warnings

None.

## 8 Example

This example program uses **naggpudevSobolExpA** to print 5 quasi-random numbers of dimension 10 from an exponential distribution using the Sobol' device function generator. The first point in the sequence is skipped and generation starts at the second point.

### 8.1 Program Text

```

/*
 * Example Program: naggpudevSobolExpA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>
#include <nag_gpu_sobolDevFuncs.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

```

```

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

#define MAX_SOBOL_DIMENSION 40

/*
 * Test kernel to try out device level functions
 * Note: this will store numbers in TRANSPOSE order
 * to naggpuQuasiRandExpA
 */
__global__ void sobol_device_functions_exp_test(
    int n, int ppt, int dim, FP *d_buff,
    NagGpuSobolDeviceComm *devComm, int offset)
{
    /* Assumptions:
     * Grid is two dimensional - run along x dimension first, then along y
     * Thread block is one dimensional with number of threads in blockDim.x.
     * Numbers are stored as npts rows of dim columns each. Have no thread
     * coalescing *at all* in this kernel
     */

    // Compute thread number for this thread
    int blockIdx = blockIdx.y*gridDim.x + blockIdx.x;
    int threadIdx = blockIdx*blockDim.x + threadIdx.x;

    // Not shared among threads - jbeing used as a cache to speed up reads
    extern __shared__ unsigned int comm3[];

    const FP lambda = 1.0;

    // Index of point we are currently creating. From 0 to npts-1
    int idx = threadIdx*ppt;
    // Storage offset
    d_buff += idx*dim;

    if (idx < n)
    {
        // Workspace variable for device Sobol functions
        int comm1 = 0;
        unsigned int comm2 = 0;

        // Initialise stream generator
        naggpuSobolInitA(dim, offset + idx, comm1, comm2,
            &comm3[threadIdx.x*dim], devComm);

        // Loop over all points we're required to create
        for(int i = 0; i < ppt && idx + i < n; i++)
        {
            naggpuSobolExpA(d_buff, lambda, comm1, comm2,
                &comm3[threadIdx.x*dim], devComm);
            d_buff += dim;
        }
    }
}

int main(int argc, char **argv)
{
    const int dim = 40;
    const int n = 101;

    FP *h_buff = 0, *d_buff = 0;

    int offset = 1;

    unsigned int pseudoSeed[] = {1, 2, 3, 4, 5, 6};

    NagGpuSobolDeviceComm *devComm;
    NagCPURandComm pseudoComm;

```

```

NagGpuQuasiRandComm comm;
NagGpuError error;
cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpudevSobolExpA, ";
if (sizeof(FP)==sizeof(float))
    cout << "single precision";
else
    cout << "double precision";
cout << endl << endl;

// Allocate CPU and GPU memory
h_buff = new FP[n*dim];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*n*dim);
checkCudaError(cuError);

// Initialise the CPU pseudo-random generator
nagCPURandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, 0, pseudoSeed,
                &pseudoComm, &error);
checkNagError(&error);

// Initialise GPU Sobol generator
naggpuSobolDeviceInitA(NAGGPUSCRAMTYPES_NONE, dim, &devComm,
                       &pseudoComm, &comm, &error);
checkNagError(&error);

// Launch GPU computation asynchronously
// Use 96 threads per block and 2 points per thread.
const int ppt = 2;
const int nthds = 96;
const int nblks = n/(ppt*nthds) + 1;
sobol_device_functions_exp_test
<<<nblks, nthds, nthds*MAX_SOBOL_DIMENSION*sizeof(unsigned int)>>>
(n, ppt, dim, d_buff, devComm, offset);

/*
 * One can now launch other kernels to operate on the Sobol numbers,
 * or copy the numbers to the host and operate on them there.
 * Here we simply copy them to the host in order to print them
 */

// Copy results to host once completed
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*n*dim,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The 5 GPU numbers from dimensions 1 to 10:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(3);
for(int d = 0; d < 10; d++)
{
    cout << "dim" << d+1 << "\t";
    for(int i = 0; i < 5; i++)
    {
        cout << h_buff[i*dim + d] << "\t";
    }
    cout << endl;
}
cout << endl;

// Call cleanup for the NAG routine
naggpuSobolDeviceCleanupA(devComm, &comm, &error);
checkNagError(&error);

```

```
// Free CPU and GPU memory
delete[] h_buff;
if (d_buff)
{
    cuError = cudaFree(d_buff);
    checkCudaError(cuError);
}

return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}
```

## 8.2 Program Data

None.

## 8.3 Program Results

NAG GPU Example Program: naggpudevSobolExpA, single precision

The 5 GPU numbers from dimensions 1 to 10:

```
dim1 0.693 0.288 1.386 0.981 0.134
dim2 0.693 1.386 0.288 0.981 0.134
dim3 0.693 1.386 0.288 0.470 2.079
dim4 0.693 1.386 0.288 0.134 0.981
dim5 0.693 0.288 1.386 0.981 0.134
dim6 0.693 0.288 1.386 2.079 0.470
dim7 0.693 1.386 0.288 0.981 0.134
dim8 0.693 0.288 1.386 0.134 0.981
dim9 0.693 0.288 1.386 0.134 0.981
dim10 0.693 0.288 1.386 0.470 2.079
```

---



# NAG Numerical Routines for GPUs Function Document

## naggpudevSobolNormalA

### 1 Purpose

**naggpudevSobolNormalA** generates the next point from a quasi-random Normal distribution with mean  $\mu$  and variance  $\sigma^2$ .

The initialization function **naggpudevSobolInitA** must be called prior to the first call to **naggpudevSobolNormalA**. Thereafter, this function may be called repeatedly to generate the next point in the sequence. Care should be taken to ensure that sequences from successive CUDA threads do not overlap.

### 2 Specification

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

template <typename FP>
__device__ void naggpudevSobolNormalA(FP *x, FP mu, FP sigma, const int comm1,
    unsigned int &comm2, unsigned int *comm3,
    const NagGpuSobolDeviceComm *devComm)
```

### 3 Description

Sobol' sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling **naggpudevSobolInitA** to initialize the generator. Below we will consider a  $d$ -dimensional Sobol' sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

The Normal distribution has probability density function given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

where  $\sigma > 0$  and  $\mu \in \mathbb{R}$ . This function returns the next point  $x = (x_1, x_2, \dots, x_d)$  where

$$x_i = \mu + \sigma\sqrt{2} \operatorname{erfinv}(z_i)$$

for each  $i = 1, 2, \dots, d$  and  $\operatorname{erfinv}$  is the inverse error function. Here  $z = (z_1, z_2, \dots, z_d)$  is a low discrepancy point in the interval  $(-1, 1)^d$ .

### 4 Error Handling

This is a GPU device function, and no error handling is performed.

### 5 References

None.

### 6 Arguments

1:  $\mathbf{x}[d]$  – FP \* *Output*

The template argument parameter FP can take type double or float.

The value  $d$  is the dimension of the sequence as specified to the initialization function **naggpudevSobolInitA**.

*On exit:* the next  $d$ -dimensional point from the specified distribution.

- 2: **mu** – FP *Input*  
 The template argument parameter FP can take type double or float.  
*On entry:* the mean,  $\mu$ , of the Normal distribution.
- 3: **sigma** – FP *Input*  
 The template argument parameter FP can take type double or float.  
*On entry:* the standard deviation,  $\sigma$ , of the Normal distribution.  
*Constraint:*  $\text{sigma} > 0$ .
- 4: **comm1** – const int *Communication Data*  
 The value of **comm1** from the initialization call naggpudevSobolInitA.
- 5: **comm2** – unsigned int & *Communication Data*  
 The value of **comm2** obtained from the previous call to any of the Sobol' device function generators, or if this is the first call to any of the Sobol' device function generators, the value of **comm2** from the initialization call naggpudevSobolInitA.
- 6: **comm3**[ $d$ ] – unsigned int \* *Communication Array*  
 The value  $d$  is the dimension of the sequence as specified to the initialization function naggpudevSobolInitA.  
 The value of **comm3** obtained from the previous call to any of the Sobol' device function generators, or if this is the first call to any of the Sobol' device function generators, the value of **comm3** from the initialization call naggpudevSobolInitA.
- 7: **devComm** – const NagGpuSobolDeviceComm \* *Communication Data*  
 The NagGpuSobolDeviceComm memory address obtained from the host initialization function naggpuSobolDeviceInitA.

## 7 Error Indicators and Warnings

None.

## 8 Example

This example program uses **naggpudevSobolNormalA** to print 5 quasi-random numbers of dimension 10 from a Normal distribution using the Sobol' device function generator. The first point in the sequence is skipped and generation starts at the second point.

### 8.1 Program Text

```

/*
 * Example Program: naggpudevSobolNormalA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 *
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>

```

```

#include <nag_gpu_sobolDevFuncs.h>
#include <nag_gpu_serial.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else
#define FP double
#endif

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

#define MAX_SOBOL_DIMENSION 40

/*
 * Test kernel to try out device level functions
 * Note: this will store numbers in TRANSPOSE order
 * to naggpuQuasiRandNormalA
 */
__global__ void sobol_device_functions_normal_test(
    int n, int ppt, int dim, FP *d_buff,
    NagGpuSobolDeviceComm *devComm, int offset)
{
    /* Assumptions:
     * Grid is two dimensional - run along x dimension first, then along y
     * Thread block is one dimensional with number of threads in blockDim.x.
     * Numbers are stored as npts rows of dim columns each. Have no thread
     * coalescing *at all* in this kernel
     */

    // Compute thread number for this thread
    int blockNum = blockIdx.y*gridDim.x + blockIdx.x;
    int thdNum = blockNum*blockDim.x + threadIdx.x;

    // Not shared among threads - jbeing used as a cache to speed up reads
    extern __shared__ unsigned int comm3[];

    const FP mu = 0.0, sigma = 1.0;

    // Index of point we are currently creating. From 0 to npts-1
    int idx = thdNum*ppt;
    // Storage offset
    d_buff += idx*dim;

    if (idx < n)
    {
        // Workspace variable for device Sobol functions
        int comm1 = 0;
        unsigned int comm2 = 0;

        // Initialise stream generator
        naggpudevSobolInitA(dim, offset + idx, comm1, comm2,
            &comm3[threadIdx.x*dim], devComm);

        // Loop over all points we're required to create
        for(int i = 0; i < ppt && idx + i < n; i++)
        {
            naggpudevSobolNormalA(d_buff, mu, sigma, comm1, comm2,
                &comm3[threadIdx.x*dim], devComm);
            d_buff += dim;
        }
    }
}

int main(int argc, char **argv)
{
    const int dim = 40;
    const int n = 101;

```

```

FP *h_buff = 0, *d_buff = 0;

int offset = 1;

unsigned int pseudoSeed[] = {1, 2, 3, 4, 5, 6};

NagGpuSobolDeviceComm *devComm;
NagCPURandComm pseudoComm;
NagGpuQuasiRandComm comm;
NagGpuError error;
cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpudevSobolNormalA, ";
if (sizeof(FP)==sizeof(float))
    cout << "single precision";
else
    cout << "double precision";
cout << endl << endl;

// Allocate CPU and GPU memory
h_buff = new FP[n*dim];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*n*dim);
checkCudaError(cuError);

// Initialise the CPU pseudo-random generator
nagCPURandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, pseudoSeed,
                &pseudoComm, &error);
checkNagError(&error);

// Initialise GPU Sobol generator
naggpuSobolDeviceInitA(NAGGPUSCRAMTYPES_NONE, dim, &devComm,
                       &pseudoComm, &comm, &error);
checkNagError(&error);

// Launch GPU computation asynchronously
// Use 96 threads per block and 2 points per thread.
const int ppt = 2;
const int nthds = 96;
const int nblks = n/(ppt*nthds) + 1;
sobol_device_functions_normal_test
<<<nblks, nthds, nthds*MAX_SOBOL_DIMENSION*sizeof(unsigned int)>>>
(n, ppt, dim, d_buff, devComm, offset);

/*
 * One can now launch other kernels to operate on the Sobol numbers,
 * or copy the numbers to the host and operate on them there.
 * Here we simply copy them to the host in order to print them
 */

// Copy results to host once completed
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*n*dim,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The 5 GPU numbers from dimensions 1 to 10:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(3);
for(int d = 0; d < 10; d++)
{
    cout << "dim" << d+1 << "\t";
    for(int i = 0; i < 5; i++)
    {
        cout << h_buff[i*dim + d] << "\t";
    }
}

```

```

        cout << endl;
    }
    cout << endl;

    // Call cleanup for the NAG routine
    naggpuSobolDeviceCleanupA(devComm, &comm, &error);
    checkNagError(&error);

    // Free CPU and GPU memory
    delete[] h_buff;
    if (d_buff)
    {
        cuError = cudaFree(d_buff);
        checkCudaError(cuError);
    }

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}

```

## 8.2 Program Data

None.

## 8.3 Program Results

NAG GPU Example Program: naggpudevSobolNormalA, single precision

The 5 GPU numbers from dimensions 1 to 10:

```

dim1 0.000 0.674 -0.674 -0.319 1.150
dim2 0.000 -0.674 0.674 -0.319 1.150
dim3 0.000 -0.674 0.674 0.319 -1.150
dim4 0.000 -0.674 0.674 1.150 -0.319
dim5 0.000 0.674 -0.674 -0.319 1.150
dim6 0.000 0.674 -0.674 -1.150 0.319
dim7 0.000 -0.674 0.674 -0.319 1.150
dim8 0.000 0.674 -0.674 1.150 -0.319
dim9 0.000 0.674 -0.674 1.150 -0.319
dim10 0.000 0.674 -0.674 0.319 -1.150

```



# NAG Numerical Routines for GPUs Function Document

## naggpudevSobolUniformA

### 1 Purpose

**naggpudevSobolUniformA** generates the next point from a quasi-random uniform distribution over the interval  $[a, b)$  for specified constants  $a$  and  $b$ .

The initialization function `naggpudevSobolInitA` must be called prior to the first call to **naggpudevSobolUniformA**. Thereafter, this function may be called repeatedly to generate the next point in the sequence. Care should be taken to ensure that sequences from successive CUDA threads do not overlap.

### 2 Specification

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

template <typename FP>
__device__ void naggpudevSobolUniformA(FP *x, FP a, FP b, const int comm1,
    unsigned int &comm2, unsigned int *comm3,
    const NagGpuSobolDeviceComm *devComm)
```

### 3 Description

Sobol' sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling `naggpudevSobolInitA` to initialize the generator. Below we will consider a  $d$ -dimensional Sobol' sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

If  $a = 0$  and  $b = 1$ , this function returns the next point  $y \in [0, 1)^d$  from the Sobol' sequence. For other values of  $a$  and  $b$ , the function applies the transformation

$$x_i = a + (b - a)y_i$$

for each  $i = 1, 2, \dots, d$  to produce a quasi-random point  $x$  in the interval  $[a, b)^d$ .

### 4 Error Handling

This is a GPU device function, and no error handling is performed.

### 5 References

None.

### 6 Arguments

1: **x**[ $d$ ] – FP \* *Output*

The template argument parameter FP can take type double or float.

The value  $d$  is the dimension of the sequence as specified to the initialization function `naggpudevSobolInitA`.

*On exit:* the next  $d$ -dimensional point from the specified distribution.

2: **a** – FP *Input*

The template argument parameter FP can take type double or float.

*On entry:* the lower bound for the quasi-random values.

3: **b** – FP *Input*

The template argument parameter FP can take type double or float.

*On entry:* the upper bound for the quasi-random values.

*Constraint:*  $b > a$ .

4: **comm1** – const int *Communication Data*

The value of **comm1** from the initialization call naggpudevSobolInitA.

5: **comm2** – unsigned int & *Communication Data*

The value of **comm2** obtained from the previous call to any of the Sobol' device function generators, or if this is the first call to any of the Sobol' device function generators, the value of **comm2** from the initialization call naggpudevSobolInitA.

6: **comm3**[*d*] – unsigned int \* *Communication Array*

The value *d* is the dimension of the sequence as specified to the initialization function naggpudevSobolInitA.

The value of **comm3** obtained from the previous call to any of the Sobol' device function generators, or if this is the first call to any of the Sobol' device function generators, the value of **comm3** from the initialization call naggpudevSobolInitA.

7: **devComm** – const NagGpuSobolDeviceComm \* *Communication Data*

The NagGpuSobolDeviceComm memory address obtained from the host initialization function naggpuSobolDeviceInitA.

## 7 Error Indicators and Warnings

None.

## 8 Example

This example program uses **naggpudevSobolUniformA** to print 5 quasi-random numbers of dimension 10 from a uniform distribution using the Sobol' device function generator. The first point in the sequence is skipped and generation starts at the second point.

### 8.1 Program Text

```

/*
 * Example Program: naggpudevSobolUniformA
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.4, 2011.
 */

#include
using namespace std;

#include <nag_gpu.h>
#include <nag_gpu_serial.h>
#include <nag_gpu_sobolDevFuncs.h>

// precision defined by Makefile
#ifdef SINGLEPRECISION
#define FP float
#else

```

```

#define FP double
#endif

void checkNagError(NagGpuError *error);
void checkCudaError(cudaError_t cuError);

#define MAX_SOBOL_DIMENSION 40

/*
 * Test kernel to try out device level functions
 * Note: this will store numbers in TRANSPOSE order
 * to naggpuQuasiRandUniformA
 */
__global__ void sobol_device_functions_uniform_test(
    int n, int ppt, int dim, FP *d_buff,
    NagGpuSobolDeviceComm *devComm, int offset)
{
    /* Assumptions:
     * Grid is two dimensional - run along x dimension first, then along y
     * Thread block is one dimensional with number of threads in blockDim.x.
     * Numbers are stored as npts rows of dim columns each. Have no thread
     * coalescing *at all* in this kernel
     */

    // Compute thread number for this thread
    int blockNum = blockIdx.y*gridDim.x + blockIdx.x;
    int thdNum = blockNum*blockDim.x + threadIdx.x;

    // Not shared among threads - jbeing used as a cache to speed up reads
    extern __shared__ unsigned int comm3[];

    const FP a = 0.0, b = 1.0;

    // Index of point we are currently creating. From 0 to npts-1
    int idx = thdNum*ppt;
    // Storage offset
    d_buff += idx*dim;

    if (idx < n)
    {
        // Workspace variable for device Sobol functions
        int comm1 = 0;
        unsigned int comm2 = 0;

        // Initialise stream generator
        naggpudevSobolInitA(dim, offset + idx, comm1, comm2,
            &comm3[threadIdx.x*dim], devComm);

        // Loop over all points we're required to create
        for(int i = 0; i < ppt && idx + i < n; i++)
        {
            naggpudevSobolUniformA(d_buff, a, b, comm1, comm2,
                &comm3[threadIdx.x*dim], devComm);
            d_buff += dim;
        }
    }
}

int main(int argc, char **argv)
{
    const int dim = 40;
    const int n = 101;

    FP *h_buff = 0, *d_buff = 0;

    int offset = 1;

    unsigned int pseudoSeed[] = {1, 2, 3, 4, 5, 6};

```

```

NagGpuSobolDeviceComm *devComm;
NagCPURandComm pseudoComm;
NagGpuQuasiRandComm comm;
NagGpuError error;
cudaError_t cuError;

// Print the title
cout << "NAG GPU Example Program: ";
cout << "naggpudevSobolUniformA, ";
if (sizeof(FP)==sizeof(float))
    cout << "single precision";
else
    cout << "double precision";
cout << endl << endl;

// Allocate CPU and GPU memory
h_buff = new FP[n*dim];
cuError = cudaMalloc((void **)&d_buff, sizeof(FP)*n*dim);
checkCudaError(cuError);

// Initialise the CPU pseudo-random generator
nagCPURandInitA(NAGGPURANDGEN_MRG32K3A, 0, 0, 0, 0, 0, pseudoSeed,
                &pseudoComm, &error);
checkNagError(&error);

// Initialise GPU Sobol generator
naggpuSobolDeviceInitA(NAGGPUSCRAMTYPES_NONE, dim, &devComm,
                      &pseudoComm, &comm, &error);
checkNagError(&error);

// Launch GPU computation asynchronously
// Use 96 threads per block and 2 points per thread.
const int ppt = 2;
const int nthds = 96;
const int nblks = n/(ppt*nthds) + 1;
sobol_device_functions_uniform_test
<<<nblks, nthds, nthds*MAX_SOBOL_DIMENSION*sizeof(unsigned int)>>>
(n, ppt, dim, d_buff, devComm, offset);

/*
 * One can now launch other kernels to operate on the Sobol numbers,
 * or copy the numbers to the host and operate on them there.
 * Here we simply copy them to the host in order to print them
 */

// Copy results to host once completed
cuError = cudaMemcpy(h_buff, d_buff, sizeof(FP)*n*dim,
                    cudaMemcpyDeviceToHost);
checkCudaError(cuError);

// Print random numbers
cout << "The 5 GPU numbers from dimensions 1 to 10:" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(3);
for(int d = 0; d < 10; d++)
{
    cout << "dim" << d+1 << "\t";
    for(int i = 0; i < 5; i++)
    {
        cout << h_buff[i*dim + d] << "\t";
    }
    cout << endl;
}
cout << endl;

// Call cleanup for the NAG routine
naggpuSobolDeviceCleanupA(devComm, &comm, &error);

```

```
    checkNagError(&error);

    // Free CPU and GPU memory
    delete[] h_buff;
    if (d_buff)
    {
        cuError = cudaFree(d_buff);
        checkCudaError(cuError);
    }

    return 0;
}

void checkNagError(NagGpuError *error)
{
    if (error->code != 0)
    {
        char *buff;
        buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        exit(1);
    }
}

void checkCudaError(cudaError_t cuError)
{
    if (cuError != cudaSuccess)
    {
        cout << cudaGetErrorString(cuError) << endl;
        exit(1);
    }
}
```

## 8.2 Program Data

None.

## 8.3 Program Results

NAG GPU Example Program: naggpudevSobolUniformA, single precision

The 5 GPU numbers from dimensions 1 to 10:

```
dim1 0.500 0.750 0.250 0.375 0.875
dim2 0.500 0.250 0.750 0.375 0.875
dim3 0.500 0.250 0.750 0.625 0.125
dim4 0.500 0.250 0.750 0.875 0.375
dim5 0.500 0.750 0.250 0.375 0.875
dim6 0.500 0.750 0.250 0.125 0.625
dim7 0.500 0.250 0.750 0.375 0.875
dim8 0.500 0.750 0.250 0.875 0.375
dim9 0.500 0.750 0.250 0.875 0.375
dim10 0.500 0.750 0.250 0.625 0.125
```



# NAG Numerical Routines for GPUs Function Document

## naggpuSobolDeviceCleanupA

### 1 Purpose

**naggpuSobolDeviceCleanupA** frees system resources that were allocated by a previous call to **naggpuSobolDeviceInitA**.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
cudaError_t naggpuSobolDeviceCleanupA(NagGpuSobolDeviceComm *devComm,
    NagGpuQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking and will force synchronization between host and device. Control will not return to the calling program before this function has terminated.

#### 3.2 Return Value

Any CUDA runtime errors that were encountered, or `cudaSuccess` if no CUDA runtime errors were encountered. Please see the Error Handling Chapter Introduction for further details on error handling.

### 4 References

None.

### 5 Arguments

- 1: **devComm** – NagGpuSobolDeviceComm \* *Input*  
*On entry:* the GPU memory address obtained from a prior call to **naggpuSobolDeviceInitA**.
- 2: **comm** – NagGpuQuasiRandComm \* *Communication Data*  
*On entry:* the pointer that was passed to a previous call to **naggpuSobolDeviceInitA**.
- 3: **error** – NagGpuError \* *Error Reporting*  
 This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 1

*On entry:* the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call `cudaGetLastError()` in the CUDA runtime library to clear the runtime error status.

error → code = 2

*During execution:* a CUDA runtime error was detected.

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:* the value of **devComm** is NULL.

## 7 Example

There is no example program specifically for this function. For examples of how this function should be used, please see the example program for `naggpudevSobolUniformA`.

---

# NAG Numerical Routines for GPUs Function Document

## nagCPURandInitA

### 1 Purpose

**nagCPURandInitA** initializes a serial, host-only pseudorandom number generator to give a repeatable sequence of pseudorandom numbers. This function must be called before any call to the serial generator functions (such as **nagCPURandUniformA**) and must ultimately be followed by a call to the cleanup function **nagCPURandCleanupA** to release system resources. A base generator is selected through the **genid** parameter and initialized with the values given in the **seed** array.

### 2 Specification

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

extern "C"
void nagCPURandInitA(NagGpuRandGen genid, int a1, int b1, int a2, int b2,
    long long c, const unsigned int *seed, NagCPURandComm *comm,
    NagGpuError *error)
```

### 3 Description

For different values of **seed**, a given generator will yield different sequences of random numbers. Alternatively, the same sequence of random numbers will be generated if the same value of **seed** is used. In general there is no guarantee of statistical properties between sequences, only within sequences. This is important when generators are used in parallel. This function can ‘skip ahead’ or advance the seed by an amount

$$s = a_1 2^{b_1} + a_2 2^{b_2} + c \quad (1)$$

so that the generator will produce the sequence of random numbers  $X_s, X_{s+1}, X_{s+2}, \dots$  instead of the original sequence  $X_0, X_1, X_2, \dots$ . This technique is useful to produce independent generators, often also called *independent streams and substreams*. Please see the Random Number Generators Chapter Introduction for further information.

Independent generators will be important mostly to applications which use multiple CPU cores simultaneously. In this case, each generator will have its own **NagCPURandComm** structure which encapsulates all the information the generator requires. An array of **NagCPURandComm** structures with judiciously chosen skip aheads (see the Random Number Generators Chapter Introduction) represents an array of independent generators. Each structure must be initialized by a call to **nagCPURandInitA**.

For the most common task of generating a block of pseudorandom numbers on a single CPU core, users will typically only have a single generator (i.e. only one communication structure) and the skip ahead  $s$  above can be set to zero. The comments about arrays of communication structures can be ignored.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

## 5 Arguments

- 1: **genid** – NagGpuRandGen *Input*  
*On entry:* the type of generator to be used:  
genid = NAGGPURANDGEN\_MRG32K3A  
genid = NAGGPURANDGEN\_MT19937  
Please see the Random Number Generators Chapter Introduction for details about each of these base generators.  
*Constraint:* genid = NAGGPURANDGEN\_MRG32K3A or NAGGPURANDGEN\_MT19937.
- 2: **a1** – int *Input*  
*On entry:* the value of  $a_1$  in the skip ahead equation (1) above.  
*Constraint:*  $a_1 \geq 0$ .
- 3: **b1** – int *Input*  
*On entry:* the value of  $b_1$  in the skip ahead equation (1) above.  
*Constraints:*  
if genid = NAGGPURANDGEN\_MRG32K3A,  $0 \leq b_1 \leq 191$ ;  
if genid = NAGGPURANDGEN\_MT19937,  $0 \leq b_1 \leq 19937$ .
- 4: **a2** – int *Input*  
*On entry:* the value of  $a_2$  in the skip ahead equation (1) above.  
*Constraint:*  $a_2 \geq 0$ .
- 5: **b2** – int *Input*  
*On entry:* the value of  $b_2$  in the skip ahead equation (1) above.  
*Constraints:*  
if genid = NAGGPURANDGEN\_MRG32K3A,  $0 \leq b_2 \leq 191$ ;  
if genid = NAGGPURANDGEN\_MT19937,  $0 \leq b_2 \leq 19937$ .
- 6: **c** – long long *Input*  
*On entry:* the value of  $c$  in the skip ahead equation (1) above.  
*Constraint:*  $c \geq 0$ .
- 7: **seed**[ $n$ ] – const unsigned int \* *Input*  
*On entry:* an array of  $n$  32-bit unsigned integers to initialize the generator.  
*Constraints:*  
if genid = NAGGPURANDGEN\_MRG32K3A,  
 $n = 6$   
for  $i = 0, 1, 2$ , seed[ $i$ ] <  $2^{32} - 209$  and seed( $i$ )  $\neq 0$  for at least one  $i$   
for  $i = 3, 4, 5$ , seed[ $i$ ] <  $2^{32} - 22853$  and seed( $i$ )  $\neq 0$  for at least one  $i$ ;  
if genid = NAGGPURANDGEN\_MT19937,  
 $n = 624$   
for  $i = 0, 1, 2, \dots, 623$ , seed( $i$ )  $\neq 0$  for at least one  $i$ .

8: **comm** – NagCPURandComm \* *Communication Data*

NagCPURandComm is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator functions (such as nagCPURandUniformA). Once all required points have been obtained, **comm** must be passed to nagCPURandCleanupA to free allocated system resources.

9: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 110

*On entry:* **genid** does not specify a valid pseudorandom number generator. See NagGpuRandGen for permitted values.

error → code = 111

*On entry:* the value of **a1** is negative.

error → code = 112

*On entry:* the value of **b1** does not satisfy the constraints listed above.

error → code = 113

*On entry:* the value of **a2** is negative.

error → code = 114

*On entry:* the value of **b2** does not satisfy the constraints listed above.

error → code = 115

*On entry:* the value of **c** is negative.

error → code = 116

*On entry:* the value of **seed** is NULL.

error → code = 117

*On entry:* the values in the **seed** array do not satisfy the constraints listed above.

## 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPURandExpA

### 1 Purpose

**nagCPURandExpA** generates  $n$  values  $x_i$  from an exponential distribution with mean  $\lambda$ .

The initialization function **nagCPURandInitA** must be called prior to the first call to **nagCPURandExpA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function **nagCPURandCleanupA** must be called to free allocated system resources.

**Note:** To obtain the same values from **nagCPURandExpA** as from the function **nag\_CPU\_mrg32-k3a\_exp(N, P)** in release 0.3 of the library, please see Section 2.1.1.1 in the Random Number Generators Chapter Introduction.

### 2 Specification

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

extern "C"
void nagCPURandExpA_sp(int n, float lambda, float *buff, NagCPURandComm *comm,
    NagGpuError *error)

extern "C"
void nagCPURandExpA(int n, double lambda, double *buff, NagCPURandComm *comm,
    NagGpuError *error)
```

### 3 Description

The exponential distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\lambda}e^{-x/\lambda} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\lambda > 0$ . This function returns

$$X_i = -\lambda \ln Y_i$$

where  $Y_i$  are the next  $n$  values generated by the underlying uniform  $[0, 1]$  generator.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

- 1: **n** – int *Input*  
*On entry:* the number of random values to be generated.  
*Constraint:*  $n \geq 1$ .

- 2: **lambda** – float *Input*  
 3: **lambda** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the mean,  $\lambda$ , of the distribution.

*Constraint:*  $\lambda > 0$ .

- 4: **buff[n]** – float \* *Output*  
 5: **buff[n]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On exit:* the **n** pseudorandom numbers from the specified distribution.

- 6: **comm** – NagCPURandComm \* *Communication Data*

NagCPURandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to nagCPURandCleanupA to free allocated system resources.

- 7: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of **error** → code which should be inspected after each call to this function. If **error** → code = 0 then no error occurred. If **error** → code  $\neq$  0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

**error** → code = 100

*On entry:* the value of **comm** is NULL.

**error** → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

**error** → code = 110

*On entry:*  $n \leq 0$ .

**error** → code = 112

*On entry:* **buff** is NULL.

**error** → code = 114

*On entry:*  $\lambda \leq 0$ .

## 7 Example

None.

# NAG Numerical Routines for GPUs Function Document

## nagCPURandGammaA

### 1 Purpose

**nagCPURandGammaA** generates  $n$  values  $X_i$  from a gamma distribution with shape parameter  $\alpha$  and scale parameter  $\beta$ .

The initialization function **nagCPURandInitA** must be called prior to the first call to **nagCPURandGammaA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function **nagCPURandCleanupA** must be called to free allocated system resources.

**Note:** currently only the MRG32k3a base generator is supported. Support for MT19937 will be added in future releases.

### 2 Specification

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

extern "C"
void nagCPURandGammaA_sp(int n, float alpha, float beta, float *buff,
    NagCPURandComm *comm, NagGpuError *error)

extern "C"
void nagCPURandGammaA(int n, double alpha, double beta, double *buff,
    NagCPURandComm *comm, NagGpuError *error)
```

### 3 Description

The gamma distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\alpha, \beta > 0$ . The rejection algorithm described in Marsaglia and Tsang (2000) is used to generate the gamma pseudorandom variates when  $\alpha \geq 1$ . When  $0 < \alpha < 1$ , the scaling

$$\gamma_\alpha = \gamma_{1+\alpha} U^\alpha$$

is used where  $U$  denotes a uniform random variable in the interval  $[0, 1]$  and  $\gamma_\alpha$  denotes a gamma random variable with shape parameter  $\alpha$  and scale parameter  $\beta = 1$ . Note that currently only the MRG32k3a base generator is supported.

**Note:** rejection algorithms are extremely sensitive to computational accuracy. When a variate is generated close to the rejection envelope, small differences in numerical values can lead to it being accepted in double precision while it is rejected in single precision (or vice versa). From this point on, the single and double precision sequences will be different. The same behaviour is seen when comparing single precision sequences generated on the CPU and the GPU: differences in the floating point calculations will lead to the sequences diverging after a certain number of variates. In double precision, the CPU and GPU sequences will take much longer (on average) before they diverge, agreeing to tens or even hundreds of millions of variates before numerical differences cause a variate to be accepted on one platform while it is rejected on the other.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

## 4 References

Marsaglia G and Tsang W W (2000) A simple method for generating Gamma variables *ACM Trans. Math. Software* **26(3)** 363–372

## 5 Arguments

1: **n** – int *Input*

*On entry:* the number of random values to be generated.

*Constraint:*  $n \geq 1$ .

2: **alpha** – float *Input*

3: **alpha** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the shape parameter,  $\alpha$ , of the distribution.

*Constraint:*  $\alpha > 0$ .

4: **beta** – float *Input*

5: **beta** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the scale parameter,  $\beta$ , of the distribution.

*Constraint:*  $\beta > 0$ .

6: **buff[n]** – float \* *Output*

7: **buff[n]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On exit:* the **n** pseudorandom numbers from the specified distribution.

8: **comm** – NagCPURandComm \* *Communication Data*

NagCPURandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to nagCPURandCleanupA to free allocated system resources.

9: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of **error** → code which should be inspected after each call to this function. If **error** → code = 0 then no error occurred. If **error** → code  $\neq$  0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

**error** → code = 100

*On entry:* the value of **comm** is NULL.

**error** → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 112

*On entry:* **buff** is NULL.

error → code = 113

*On entry:*  $\alpha \leq 0$

error → code = 114

*On entry:*  $\beta \leq 0$

error → code = 115

*On entry:* the MT19937 base generator is selected (see nagCPURandInitA for further details).  
Currently only the MRG32k3a base generator is supported.

## 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPURandNormalA

### 1 Purpose

**nagCPURandNormalA** generates  $n$  values  $x_i$  from a Normal distribution with mean  $\mu$  and variance  $\sigma^2$ .

The initialization function `nagCPURandInitA` must be called prior to the first call to **nagCPURandNormalA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function `nagCPURandCleanupA` must be called to free allocated system resources.

**Note:** To obtain the same values from **nagCPURandNormalA** as from the function `nag_CPU_mrg32k3a_normal(N, P)` in release 0.3 of the library, please see Section 2.1.1.1 in the Random Number Generators Chapter Introduction.

### 2 Specification

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

extern "C"
void nagCPURandNormalA_sp(int n, float mu, float sigma, float *buff,
    NagCPURandComm *comm, NagGpuError *error)

extern "C"
void nagCPURandNormalA(int n, double mu, double sigma, double *buff,
    NagCPURandComm *comm, NagGpuError *error)
```

### 3 Description

The Normal distribution has probability density function given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where  $\sigma > 0$  and  $\mu \in \mathbb{R}$ . This function uses a Box-Muller transform to convert a pair of uniform  $(0, 1)$  random numbers into a pair of Normal random numbers. Let  $X_0, X_1, X_2, \dots$  denote the sequence of uniform  $(0, 1)$  pseudorandom variates as specified by the base generator algorithm. This function uses successive pairs of uniform variates in the Box-Muller transform to produce successive pairs of Normal variates, i.e.  $(X_0, X_1) \mapsto (Z_0, Z_1)$ ,  $(X_2, X_3) \mapsto (Z_2, Z_3)$  where  $Z_0, Z_1, Z_2, \dots$  denotes the output sequence of Normal variates.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

- 1: **n** – int *Input*  
*On entry:* the number of random values to be generated.  
*Constraint:*  $n \geq 1$ .

- 2: **mu** – float *Input*  
 3: **mu** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the mean,  $\mu$ , of the distribution.

- 4: **sigma** – float *Input*  
 5: **sigma** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the standard deviation,  $\sigma$ , of the distribution

*Constraint:*  $\sigma > 0$ .

- 6: **buff[n]** – float \* *Output*  
 7: **buff[n]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On exit:* the **n** pseudorandom numbers from the specified distribution.

- 8: **comm** – NagCPURandComm \* *Communication Data*

NagCPURandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to nagCPURandCleanupA to free allocated system resources.

- 9: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of **error** → code which should be inspected after each call to this function. If **error** → code = 0 then no error occurred. If **error** → code  $\neq 0$  then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

**error** → code = 100

*On entry:* the value of **comm** is NULL.

**error** → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

**error** → code = 110

*On entry:*  $n \leq 0$ .

**error** → code = 112

*On entry:* **buff** is NULL.

**error** → code = 115

*On entry:*  $\sigma \leq 0$

## 7 Example

None.

# NAG Numerical Routines for GPUs Function Document

## nagCPURandUniformA

### 1 Purpose

**nagCPURandUniformA** generates  $n$  values  $X_i$  from a uniform distribution over the interval  $[a, b]$  for specified constants  $a$  and  $b$ .

The initialization function `nagCPURandInitA` must be called prior to the first call to **nagCPURandUniformA**. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function `nagCPURandCleanupA` must be called to free allocated system resources.

**Note:** To obtain the same values from **nagCPURandUniformA** as from the function `nag_CPU_mrg32-k3a_uniform(N, P)` in release 0.3 of the library, please see Section 2.1.1.1 in the Random Number Generators Chapter Introduction.

### 2 Specification

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

extern "C"
void nagCPURandUniformA_sp(int n, float a, float b, float *buff,
    NagCPURandComm *comm, NagGpuError *error)

extern "C"
void nagCPURandUniformA(int n, double a, double b, double *buff,
    NagCPURandComm *comm, NagGpuError *error)
```

### 3 Description

If  $a = 0$  and  $b = 1$ , this function returns the next  $n$  values  $Y_i$  from a uniform  $[0, 1]$  generator. For other values of  $a$  and  $b$ , the function applies the transformation

$$X_i = a + (b - a)Y_i$$

to produce random numbers from the interval  $[a, b]$ .

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

- 1: **n** – int *Input*  
*On entry:* the number of random values to be generated.  
*Constraint:*  $n \geq 1$ .

- 2: **a** – float *Input*  
 3: **a** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The lower bound for the uniform random values.

- 4: **b** – float *Input*  
 5: **b** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The upper bound for the uniform random values.

*Constraint:*  $b > a$ .

- 6: **buff[n]** – float \* *Output*  
 7: **buff[n]** – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On exit:* the **n** pseudorandom numbers from the specified distribution.

- 8: **comm** – NagCPURandComm \* *Communication Data*

NagCPURandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to nagCPURandCleanupA to free allocated system resources.

- 9: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 112

*On entry:* **buff** is NULL.

error → code = 113

*On entry:*  $b \leq a$

## 7 Example

None.

# NAG Numerical Routines for GPUs Function Document

## nagCPURandCleanupA

### 1 Purpose

**nagCPURandCleanupA** frees system resources that were allocated by a previous call to **nagCPURandInitA**.

### 2 Specification

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

extern "C"
void nagCPURandCleanupA(NagCPURandComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

1: **comm** – NagCPURandComm \* *Communication Data*  
*On entry:* the pointer that was passed to a previous call to **nagCPURandInitA**.

2: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of **error** → code which should be inspected after each call to this function. If **error** → code = 0 then no error occurred. If **error** → code ≠ 0 then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

### 6 Error Indicators and Warnings

**error** → code = 100

*On entry:* the value of **comm** is NULL.

**error** → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

### 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPUQuasiRandInitA

### 1 Purpose

**nagCPUQuasiRandInitA** initializes a serial, host-only quasi-random number generator. This function must be called before any call to the serial quasi-random generator functions (such as **nagCPUQuasiRandUniformA**) and must ultimately be followed by a call to the cleanup function **nagCPUQuasiRandCleanupA** to release system resources.

### 2 Specification

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

extern "C"
void nagCPUQuasiRandInitA(NagGpuQuasiGen genid, NagGpuScramTypes stype, int dim,
    int offset, NagCPURandComm *pseudoComm, NagCPUQuasiRandComm *comm,
    NagGpuError *error)
```

### 3 Description

Low discrepancy (quasi-random) sequences are used in numerical integration, simulation and optimization. Like pseudorandom numbers they are uniformly distributed, but they are not statistically independent. Quasi-random sequences are designed to give a more even distribution in multidimensional space (uniformity), and are often more efficient than pseudorandom numbers in multidimensional Monte Carlo methods.

Let  $x^1, x^2, \dots, x^N$  be a sequence of  $d$ -dimensional points in the unit cube  $I^d = [0, 1]^d$ . Let  $G$  be a subset of  $I^d$  and define the counting function  $S_N(G)$  as the number of  $d$ -dimensional points  $x^i \in G$ . For each point  $x = (x_1, x_2, \dots, x_d) \in I^d$ , let  $G_x$  be the rectangular  $d$ -dimensional region

$$G_x = [0, x_1) \times [0, x_2) \times \dots \times [0, x_d)$$

with volume  $x_1 \cdot x_2 \cdot \dots \cdot x_d = \prod_{i=1}^d x_i$ . Then one measure of the uniformity of the points  $x^1, x^2, \dots, x^N$  is the so-called star discrepancy:

$$D_N^*(x^1, x^2, \dots, x^N) = \sup_{x \in I^d} \left| S_N(G_x) - N \prod_{i=1}^d x_i \right|$$

which satisfies the inequality

$$D_N^*(x^1, x^2, \dots, x^N) \leq C_d (\log N)^d + O((\log N)^{d-1}) \quad \text{for all } N \geq 2.$$

The principal aim in the construction of low-discrepancy sequences is to find sequences of points in  $I^d$  with a bound of this form where the constant  $C_d$  is as small as possible.

The type of low-discrepancy sequence generated by **nagCPUQuasiRandInitA** depends on the value of **genid**, and the sequence can optionally be scrambled through the parameter **stype**. See **NagGpuQuasiGen** and **NagGpuScramTypes** respectively for further information.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

## 4 References

None.

## 5 Arguments

- 1: **genid** – NagGpuQuasiGen *Input*  
*On entry:* the type of generator to be used:  
genid = NAGGPUQUASIGEN\_SOBOL  
*Constraint:* genid = NAGGPUQUASIGEN\_SOBOL.
- 2: **stype** – NagGpuScramTypes *Input*  
*On entry:* the type of scrambling to be used:  
stype = NAGGPUSCRAMTYPES\_NONE  
stype = NAGGPUSCRAMTYPES\_OWEN  
stype = NAGGPUSCRAMTYPES\_FAURE\_TEZUKA  
stype = NAGGPUSCRAMTYPES\_OWEN\_FAURE\_TEZUKA  
Please see NagGpuScramTypes for some of the benefits of scrambling and details about each of available scrambling types.  
*Constraint:* stype = NAGGPUSCRAMTYPES\_NONE or  
NAGGPUSCRAMTYPES\_OWEN or  
NAGGPUSCRAMTYPES\_FAURE\_TEZUKA or  
NAGGPUSCRAMTYPES\_OWEN\_FAURE\_TEZUKA
- 3: **dim** – int *Input*  
*On entry:* the dimension of the quasi-random sequence.  
*Constraint:*  $1 \leq \text{dim} \leq 50000$ .
- 4: **offset** – int *Input*  
*On entry:* the offset into the sequence at which to start generating.  
*Constraint:* offset  $\geq 0$ .
- 5: **pseudoComm** – NagCPURandComm \* *Input*  
*On entry:* a pointer to a NagCPURandComm structure which has already been initialized by the function nagCPURandInitA.  
*Constraint:* **pseudoComm** must be initialized before being passed to this function .
- 6: **comm** – NagCPUQuasiRandComm \* *Communication Data*  
NagCPUQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator functions (such as nagCPUQuasiRandUniformA). Once all required points have been obtained, **comm** must be passed to nagCPUQuasiRandCleanupA to free allocated system resources.
- 7: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of error  $\rightarrow$  code which should be inspected after each call to this function. If error  $\rightarrow$  code = 0 then no error occurred. If error  $\rightarrow$  code  $\neq 0$  then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 110

*On entry:* **genid** does not specify a valid quasi-random number generator. See NagGpuQuasiGen for permitted values.

error → code = 111

*On entry:* **stype** does not specify a valid scrambling type. See NagGpuScramTypes for permitted values.

error → code = 112

*On entry:* the value of **dim** does not satisfy the constraint listed above.

error → code = 113

*On entry:* the value of **offset** is negative.

error → code = 114

*On entry:* the value of **pseudoComm** is NULL.

error → code = 115

*On entry:* the pseudorandom generator nagCPURandUniformA returned an error when called by this function: **pseudoComm** is not initialized, or the internal state of **pseudoComm** is corrupted.

## 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPUQuasiRandExpA

### 1 Purpose

**nagCPUQuasiRandExpA** generates  $n$  points  $x_i$  from a quasi-random exponential distribution with mean  $\lambda$ .

The initialization function **nagCPUQuasiRandInitA** must be called prior to the first call to **nagCPUQuasiRandExpA**. Thereafter, this function may be called repeatedly to generate additional sets of quasi-random points. Once all desired points have been obtained, the function **nagCPUQuasiRandCleanupA** must be called to free allocated system resources.

**Note:** Concerns were raised about the set of Sobol' direction numbers that were used in release 0.3 of the NAG Numerical Routines for GPUs. These concerns have been addressed by an amended set of direction numbers in Joe and Kuo (2008) which are used in this release. Consequently, the higher dimensions of this Sobol' generator may not match the higher dimensions of the generator in release 0.3 since the direction numbers are different.

### 2 Specification

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

extern "C"
void nagCPUQuasiRandExpA_sp(int n, NagGpuQuasiOrient orient, float lambda,
    float *buff, NagCPUQuasiRandComm *comm, NagGpuError *error)

extern "C"
void nagCPUQuasiRandExpA(int n, NagGpuQuasiOrient orient, double lambda,
    double *buff, NagCPUQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

Quasi-random sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling **nagCPUQuasiRandInitA** to initialize the generator. Below we will consider a  $d$ -dimensional quasi-random sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

The exponential distribution has probability density function given by

$$f(x) = \begin{cases} \frac{1}{\lambda} e^{-x/\lambda} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\lambda > 0$ . This function returns the next  $n$  points  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  for  $j = 1, 2, \dots, n$  where

$$x_i^j = -\lambda \ln(y_i^j + 2^{-32})$$

for each  $i = 1, 2, \dots, d$ . Here  $y^j = (y_1^j, y_2^j, \dots, y_d^j) \in [0, 1)^d$  are the next  $n$  points from the quasi-random generator.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

## 4 References

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

## 5 Arguments

1: **n** – int *Input*  
*On entry:* the number of quasi-random points to be generated.  
*Constraint:*  $n \geq 1$ .

2: **orient** – NagGpuQuasiOrient *Input*  
*On entry:* specifies the orientation with which the generator will store the output points. See NagGpuQuasiOrient for further details.  
**orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC  
**orient** = NAGGPUQUASIORIENT\_DIMVALS\_SCATT  
*Constraint:* **orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC or NAGGPUQUASIORIENT\_DIMVALS\_SCATT

3: **lambda** – float *Input*  
 4: **lambda** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The mean,  $\lambda$ , of the exponential distribution.

*Constraint:*  $\lambda > 0$ .

5: **buff**[ $n \times d$ ] – float \* *Output*  
 6: **buff**[ $n \times d$ ] – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

The value  $d$  is the dimension **dim** of the sequence as specified to the initialization function nagCPUQuasiRandInitA.

*On exit:* the **n** quasi-random points from the specified distribution.

When **orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC, the  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location  $\text{buff}[j * d + (i - 1)]$  for every  $0 \leq j < n$  and  $1 \leq i \leq d$ .

When **orient** = NAGGPUQUASIORIENT\_DIMVALS\_SCATT, the  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location  $\text{buff}[(i - 1) * n + j]$  for every  $0 \leq j < n$  and  $1 \leq i \leq d$ .

7: **comm** – NagCPUQuasiRandComm \* *Communication Data*  
 NagCPUQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to nagCPUQuasiRandCleanupA to free allocated system resources.

8: **error** – NagGpuError \* *Error Reporting*  
 This parameter contains error information and should not be modified directly. Errors are indicated through the value of **error** → code which should be inspected after each call to this function. If **error** → code = 0 then no error occurred. If **error** → code  $\neq$  0 then an error was detected and a call

to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **orient** does not satisfy the constraint listed above.

error → code = 112

*On entry:* **buff** is NULL.

error → code = 114

*On entry:*  $\lambda \leq 0$ .

## 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPUQuasiRandNormalA

### 1 Purpose

**nagCPUQuasiRandNormalA** generates  $n$  points  $x_i$  from a quasi-random Normal distribution with mean  $\mu$  and variance  $\sigma^2$ .

The initialization function **nagCPUQuasiRandInitA** must be called prior to the first call to **nagCPUQuasiRandNormalA**. Thereafter, this function may be called repeatedly to generate additional sets of quasi-random points. Once all desired points have been obtained, the function **nagCPUQuasiRandCleanupA** must be called to free allocated system resources.

**Note:** Concerns were raised about the set of Sobol' direction numbers that were used in release 0.3 of the NAG Numerical Routines for GPUs. These concerns have been addressed by an amended set of direction numbers in Joe and Kuo (2008) which are used in this release. Consequently, the higher dimensions of this Sobol' generator may not match the higher dimensions of the generator in release 0.3 since the direction numbers are different.

### 2 Specification

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

extern "C"
void nagCPUQuasiRandNormalA_sp(int n, NagGpuQuasiOrient orient, float mu,
    float sigma, float *buff, NagCPUQuasiRandComm *comm, NagGpuError *error)

extern "C"
void nagCPUQuasiRandNormalA(int n, NagGpuQuasiOrient orient, double mu,
    double sigma, double *buff, NagCPUQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

Quasi-random sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling **nagCPUQuasiRandInitA** to initialize the generator. Below we will consider a  $d$ -dimensional quasi-random sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

The Normal distribution has probability density function given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where  $\sigma > 0$  and  $\mu \in \mathbb{R}$ . This function returns the next  $n$  points  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  for  $j = 1, 2, \dots, n$  where

$$x_i^j = \mu + \sigma\sqrt{2} \operatorname{erfinv}(z_i^j)$$

for each  $i = 1, 2, \dots, d$  and  $\operatorname{erfinv}$  is the inverse error function. Here each  $z^j = (z_1^j, z_2^j, \dots, z_d^j)$  is a low discrepancy point in the interval  $(-1, 1)^d$ .

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

## 4 References

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

## 5 Arguments

1: **n** – int *Input*

*On entry:* the number of quasi-random points to be generated.

*Constraint:*  $n \geq 1$ .

2: **orient** – NagGpuQuasiOrient *Input*

*On entry:* specifies the orientation with which the generator will store the output points. See NagGpuQuasiOrient for further details.

**orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC

**orient** = NAGGPUQUASIORIENT\_DIMVALS\_SCATT

*Constraint:* **orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC or  
NAGGPUQUASIORIENT\_DIMVALS\_SCATT

3: **mu** – float *Input*

4: **mu** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the mean,  $\mu$ , of the distribution.

5: **sigma** – float *Input*

6: **sigma** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* the standard deviation,  $\sigma$ , of the distribution

*Constraint:*  $\text{sigma} > 0$ .

7: **buff**[ $n \times d$ ] – float \* *Output*

8: **buff**[ $n \times d$ ] – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

The value  $d$  is the dimension **dim** of the sequence as specified to the initialization function nagCPUQuasiRandInitA.

*On exit:* the **n** quasi-random points from the specified distribution.

When **orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC, the  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location  $\text{buff}[j * d + (i - 1)]$  for every  $0 \leq j < n$  and  $1 \leq i \leq d$ .

When **orient** = NAGGPUQUASIORIENT\_DIMVALS\_SCATT, the  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location  $\text{buff}[(i - 1) * n + j]$  for every  $0 \leq j < n$  and  $1 \leq i \leq d$ .

9: **comm** – NagCPUQuasiRandComm \* *Communication Data*

NagCPUQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to nagCPUQuasiRandCleanupA to free allocated system resources.

10: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **orient** does not satisfy the constraint listed above.

error → code = 112

*On entry:* **buff** is NULL.

error → code = 115

*On entry:*  $\sigma \leq 0$ .

## 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPUQuasiRandUniformA

### 1 Purpose

**nagCPUQuasiRandUniformA** generates  $n$  points  $x_i$  from a quasi-random uniform distribution over the interval  $[a, b)$  for specified constants  $a$  and  $b$ .

The initialization function **nagCPUQuasiRandInitA** must be called prior to the first call to **nagCPUQuasiRandUniformA**. Thereafter, this function may be called repeatedly to generate additional sets of quasi-random points. Once all desired points have been obtained, the function **nagCPUQuasiRandCleanupA** must be called to free allocated system resources.

**Note:** Concerns were raised about the set of Sobol' direction numbers that were used in release 0.3 of the NAG Numerical Routines for GPUs. These concerns have been addressed by an amended set of direction numbers in Joe and Kuo (2008) which are used in this release. Consequently, the higher dimensions of this Sobol' generator may not match the higher dimensions of the generator in release 0.3 since the direction numbers are different.

### 2 Specification

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

extern "C"
void nagCPUQuasiRandUniformA_sp(int n, NagGpuQuasiOrient orient, float a,
    float b, float *buff, NagCPUQuasiRandComm *comm, NagGpuError *error)

extern "C"
void nagCPUQuasiRandUniformA(int n, NagGpuQuasiOrient orient, double a, double b,
    double *buff, NagCPUQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

Quasi-random sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. The dimensionality of the sequence is specified when calling **nagCPUQuasiRandInitA** to initialize the generator. Below we will consider a  $d$ -dimensional quasi-random sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values.

If  $a = 0$  and  $b = 1$ , this function returns the next  $n$  points  $y^j \in [0, 1)^d$  from the quasi-random generator. For other values of  $a$  and  $b$ , the function applies the transformation

$$x_i^j = a + (b - a)y_i^j$$

for each  $i = 1, 2, \dots, d$  to produce quasi-random points  $x^j$  from the interval  $[a, b)^d$  for each  $j = 1, 2, \dots, n$ .

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

## 4 References

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

## 5 Arguments

1: **n** – int *Input*

*On entry:* the number of quasi-random points to be generated.

*Constraint:*  $n \geq 1$ .

2: **orient** – NagGpuQuasiOrient *Input*

*On entry:* specifies the orientation with which the generator will store the output points. See NagGpuQuasiOrient for further details.

**orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC

**orient** = NAGGPUQUASIORIENT\_DIMVALS\_SCATT

*Constraint:* **orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC or  
NAGGPUQUASIORIENT\_DIMVALS\_SCATT

3: **a** – float *Input*

4: **a** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The lower bound for the uniform random values.

5: **b** – float *Input*

6: **b** – double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

*On entry:* The upper bound for the uniform random values.

*Constraint:*  $b > a$ .

7: **buff**[ $n \times d$ ] – float \* *Output*

8: **buff**[ $n \times d$ ] – double \* *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

The value  $d$  is the dimension **dim** of the sequence as specified to the initialization function nagCPUQuasiRandInitA.

*On exit:* the **n** quasi-random points from the specified distribution.

When **orient** = NAGGPUQUASIORIENT\_DIMVALS\_CONSEC, the  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location  $\text{buff}[j * d + (i - 1)]$  for every  $0 \leq j < n$  and  $1 \leq i \leq d$ .

When **orient** = NAGGPUQUASIORIENT\_DIMVALS\_SCATT, the  $i$ -th dimension  $x_i^j$  of the  $j$ -th quasi-random point will be stored at location  $\text{buff}[(i - 1) * n + j]$  for every  $0 \leq j < n$  and  $1 \leq i \leq d$ .

9: **comm** – NagCPUQuasiRandComm \* *Communication Data*

NagCPUQuasiRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, **comm** must be passed to nagCPUQuasiRandCleanupA to free allocated system resources.

10: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:*  $n \leq 0$ .

error → code = 111

*On entry:* **orient** does not satisfy the constraint listed above.

error → code = 112

*On entry:* **buff** is NULL.

error → code = 113

*On entry:*  $b \leq a$

## 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPUQuasiRandCleanupA

### 1 Purpose

**nagCPUQuasiRandCleanupA** frees system resources that were allocated by a previous call to **nagCPUQuasiRandInitA**.

### 2 Specification

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

extern "C"
void nagCPUQuasiRandCleanupA(NagCPUQuasiRandComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

1: **comm** – NagCPUQuasiRandComm \* *Communication Data*  
*On entry:* the pointer that was passed to a previous call to **nagCPUQuasiRandInitA**.

2: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of **error** → code which should be inspected after each call to this function. If **error** → code = 0 then no error occurred. If **error** → code ≠ 0 then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

### 6 Error Indicators and Warnings

**error** → code = 100

*On entry:* the value of **comm** is NULL.

**error** → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

### 7 Example

None.



# NAG Numerical Routines for GPUs Function Document

## nagCPUDepthBBInitA

### 1 Purpose

**nagCPUDepthBBInitA** initializes the serial, host-only depth-order Brownian bridge generator **nagCPUDepthBBA**. It must be called before any calls to **nagCPUDepthBBA** and must finally be followed by a call to **nagCPUDepthBBCleanupA**.

**Note:** after the first call to **nagCPUDepthBBInitA**, all subsequent calls (for example, to change the time points) must be preceded by a call to **nagCPUDepthBBCleanupA**.

### 2 Specification

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

extern "C"
void nagCPUDepthBBInitA(float tStart, const float *times, int nTimes,
    bool isBridgeFree, NagCPUDepthBBComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

- 1: **tStart** – float *Input*  
*On entry:* the starting value of the time interval.
- 2: **times[nTimes]** – const float \* *Input*  
*On entry:* the vector of times at which to compute the Brownian bridge.  
*Constraint:* the values in **times** must be in increasing order, and each must be greater than **tStart**.
- 3: **nTimes** – int *Input*  
*On entry:* the length of the vector **times**.  
*Constraint:*  $1 \leq nTimes \leq 4095$ .
- 4: **isBridgeFree** – bool *Input*  
*On entry:* specifies whether a free or ‘pinned’ Brownian bridge is to be constructed. See **nagCPUDepthBBA** for more details.  
  
If **isBridgeFree** = **true**, **nagCPUDepthBBA** will construct a free Brownian motion via a depth-order Brownian bridge algorithm.  
  
If **isBridgeFree** = **false**, **nagCPUDepthBBA** will construct a non-free or ‘pinned’ Brownian motion.

5: **comm** – NagCPUDepthBBComm \* *Communication Data*

NagCPUDepthBBComm is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator function nagCPUDepthBBA. Once all required bridge sample paths have been obtained, **comm** must be passed to nagCPUDepthBBCleanupA to free allocated system resources.

6: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 110

*On entry:* the value of **times** is NULL.

error → code = 111

*On entry:* the value of **nTimes** does not satisfy the constraint listed above.

error → code = 112

*On entry:* the values in the **times** array do not satisfy the constraints listed above.

## 7 Example

None.

---

# NAG Numerical Routines for GPUs Function Document

## nagCPUDepthBBA

### 1 Purpose

**nagCPUDepthBBA** is a serial, host-only function which 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 `nagCPUDepthBBInitA`, and must finally be followed by a call to `nagCPUDepthBBCleanupA`.

### 2 Specification

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

extern "C"
void nagCPUDepthBBA(int nPaths, int dim, float bgStart, float bgEnd,
    const float *z, const float *cholCov, float *bgVals,
    NagCPUDepthBBComm *comm, NagGpuError *error)
```

### 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

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

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  $\Sigma$  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

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

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  $\Sigma$ .

### 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 **isBridgeFree** in the initialization function nagCPUDepthBBInitA 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  $X_{t_1}, X_{t_2}, X_{t_3}, \dots, X_{t_N}$ .

### 3.3 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

## 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: **nPaths** – int *Input*  
*On entry:* the number of Brownian bridge sample paths to create.  
*Constraint:* nPaths  $\geq$  1.
- 2: **dim** – int *Input*  
*On entry:* the dimension of each Brownian bridge sample path.  
*Constraint:*  $1 \leq$  dim  $\leq$  8.
- 3: **bgStart** – float *Input*  
*On entry:* the starting value  $x$  of the bridge.
- 4: **bgEnd** – float *Input*  
*On entry:* the final value  $w$  of the bridge. If nagCPUDepthBBInitA was called with isBridgeFree = **true**, this value is ignored and  $w$  is set equal to  $x + CW_T$ .

5: **z**[**dim** × *N* × **nPaths**] – const float \* *Input*

The variable *N* denotes the length **nTimes** of the **times** array passed to the initialization function nagCPUDepthBBInitA.

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

*Constraints:*

If nagCPUDepthBBInitA was called with isBridgeFree = **true**, then **z** must contain  $N \times \text{dim} \times \text{nPaths}$  values. The values should be laid out as a matrix with **nPaths** rows and  $\text{dim} \times N$  columns. If quasi-random numbers are to be used, successive  $\text{dim} \times N$ -dimensional points should be stored in successive rows of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_CONSEC**;

If nagCPUDepthBBInitA was called with isBridgeFree = **false**, then **z** must contain  $(N - 1) \times \text{dim} \times \text{nPaths}$  values. The values should be laid out as a matrix with **nPaths** rows and  $\text{dim} \times (N - 1)$  columns. If quasi-random numbers are to be used, successive  $\text{dim} \times N$ -dimensional points should be stored in successive rows of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_CONSEC**.

6: **cholCov**[**dim** × **dim**] – const float \* *Input*

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

7: **bgVals**[**dim** × *N* × **nPaths**] – float \* *Output*

The variable *N* denotes the length **nTimes** of the **times** array passed to the initialization function nagCPUDepthBBInitA.

*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  $0 \leq d < \text{dim}$ ,  $0 \leq i < N$  and  $0 \leq p < \text{nPaths}$ , then  $x_{p,i}^d$  will be stored at **bgVals**[ $d + i * \text{dim} + p * \text{dim} * N$ ].

**Note:** output is *transposed* relative to the equivalent GPU function.

8: **comm** – NagCPUDepthBBComm \* *Communication Data*

NagCPUDepthBBComm is a structure which holds state and communication information and must not be modified in any way. Once all required bridge sample paths have been obtained, **comm** must be passed to nagCPUDepthBBCleanupA to free allocated system resources.

9: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:* the value of **nPaths** does not satisfy the constraint listed above.

error → code = 111

*On entry:* the value of **dim** does not satisfy the constraint listed above.

error → code = 112

*On entry:* the value of **z** is NULL.

error → code = 113

*On entry:* the value of **cholCov** is NULL.

error → code = 115

*On entry:* the value of **bgVals** is NULL.

## 7 Example

None.

---

# NAG Numerical Routines for GPUs Function Document

## nagCPUDepthBBIncInitA

### 1 Purpose

**nagCPUDepthBBIncInitA** initializes the serial, host-only depth-order Brownian bridge increments generator nagCPUDepthBBIncA. This function must be called before any calls to nagCPUDepthBBIncA and must finally be followed by a call to nagCPUDepthBBCleanupA.

**Note:** after the first call to **nagCPUDepthBBIncInitA**, all subsequent calls (for example, to change the time points) must be preceded by a call to nagCPUDepthBBCleanupA.

### 2 Specification

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

extern "C"
void nagCPUDepthBBIncInitA(float tStart, const float *times, int nTimes,
    bool isBridgeFree, NagCPUDepthBBComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

- 1: **tStart** – float *Input*  
*On entry:* the starting value of the time interval.
- 2: **times**[**nTimes**] – const float \* *Input*  
*On entry:* the vector of times at which to compute the Brownian bridge.  
*Constraint:* the values in **times** must be in increasing order, and each must be greater than **tStart**.
- 3: **nTimes** – int *Input*  
*On entry:* the length of the vector **times**.  
*Constraint:*  $1 \leq \text{nTimes} \leq 4095$ .
- 4: **isBridgeFree** – bool *Input*  
*On entry:* specifies whether scaled increments for a free or ‘pinned’ Brownian bridge is to be constructed. See nagCPUDepthBBIncA for more details.  
  
If **isBridgeFree** = **true**, nagCPUDepthBBIncA will construct scaled increments of a free Brownian motion via a depth-order Brownian bridge algorithm.  
  
If **isBridgeFree** = **false**, nagCPUDepthBBIncA will construct scaled increments of a non-free or ‘pinned’ Brownian motion.

5: **comm** – NagCPUDepthBBComm \* *Communication Data*

NagCPUDepthBBComm is a structure which holds state and communication information and must not be modified in any way. The structure will be initialized and must be passed to the generator function nagCPUDepthBBIncA. Once all required bridge increments have been obtained, **comm** must be passed to nagCPUDepthBBCleanupA to free allocated system resources.

6: **error** – NagGpuError \* *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 110

*On entry:* the value of **times** is NULL.

error → code = 111

*On entry:* the value of **nTimes** does not satisfy the constraint listed above.

error → code = 112

*On entry:* the values in the **times** array do not satisfy the constraints listed above.

## 7 Example

None.

---

# NAG Numerical Routines for GPUs Function Document

## nagCPUDepthBBInCA

### 1 Purpose

**nagCPUDepthBBInCA** is a serial, host-only function which computes scaled increments of a depth-order Brownian bridge or free Brownian motion. It must be preceded by a call to the initialization function `nagCPUDepthBBInCAInitA`, and must finally be followed by a call to `nagCPUDepthBBInCACleanupA`.

### 2 Specification

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

extern "C"
void nagCPUDepthBBInCA(int nPaths, int dim, float startEndDiff, const float *z,
    const float *cholCov, float *bgIncs, NagCPUDepthBBComm *comm,
    NagGpuError *error)
```

### 3 Description

Fix  $T > 0$  and suppose that  $0 = t_0 < t_1 < \dots < t_N = T$ . Conceptually, this algorithm first constructs a depth-order Brownian bridge  $X = (X_t)_{0 \leq t \leq T}$  in the same way as `nagCPUDepthBBA` and then computes

$$\frac{X_{t_1} - X_{t_0}}{t_1 - t_0}, \frac{X_{t_2} - X_{t_1}}{t_2 - t_1}, \dots, \frac{X_{t_N} - X_{t_{N-1}}}{t_N - t_{N-1}}.$$

Such increments can be useful when computing numerical solutions to stochastic differential equations driven by either a Brownian bridge or a free Brownian motion. For more details on the Brownian bridge and how it is constructed, see the discussion in `nagCPUDepthBBA`.

We briefly recall some notation: for further details see `nagCPUDepthBBA`. We let  $W = (W_t)_{0 \leq t \leq T}$  be a standard  $d$ -dimensional Wiener process, we let  $\Sigma$  be a  $d \times d$  covariance matrix, we choose  $C$  to be a  $d \times d$  matrix such that  $CC' = \Sigma$ , and we fix two points  $x$  and  $w$  in  $\mathbb{R}^d$ . The generalized Brownian bridge  $X = (X_t)_{0 \leq t \leq T}$  is defined as

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

for all  $t \in [0, T]$  so that  $X_0 = x$ ,  $X_T = w$  and  $\text{Cov}(X_s, X_t) = s(T - t)/T\Sigma$  for all  $s \leq t$  in  $[0, T]$ . This process is a non-free or ‘pinned’ Brownian motion since  $X_T = w$ . However if we set  $w = x + CW_T$  then  $X_t = x + CW_t$  becomes a standard, correlated  $d$ -dimensional Brownian motion. The boolean parameter **isBridgeFree** in the initialization routine `nagCPUDepthBBInCAInitA` controls whether a free or non-free Brownian sample path is created.

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

## 5 Arguments

- 1: **nPaths** – int *Input*  
*On entry:* the number of Brownian bridge sample paths that are created.  
*Constraint:*  $nPaths \geq 1$ .
- 2: **dim** – int *Input*  
*On entry:* the dimension of each Brownian bridge sample path.  
*Constraint:*  $1 \leq dim \leq 8$ .
- 3: **startEndDiff** – float *Input*  
*On entry:* the difference between  $X_{t_N}$  and  $X_{t_0}$ . If nagCPUDepthBBIncInitA was called with isBridgeFree = **true**, this value is ignored and  $X_{t_N}$  is set equal to  $x + CW_T$ .
- 4: **z[dim × N × nPaths]** – const float \* *Input*  
The variable  $N$  denotes the length **nTimes** of the **times** array passed to the initialization function nagCPUDepthBBIncInitA.  
*On entry:* the Normal random numbers used to construct the bridge.  
*Constraints:*  
If nagCPUDepthBBIncInitA was called with isBridgeFree = **true**, then **z** must contain  $N \times dim \times nPaths$  values. The values should be laid out as a matrix with **nPaths** rows and  $dim \times N$  columns. If quasi-random numbers are to be used, successive  $dim \times N$ -dimensional points should be stored in successive rows of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_CONSEC**;  
If nagCPUDepthBBIncInitA was called with isBridgeFree = **false**, then **z** must contain  $(N - 1) \times dim \times nPaths$  values. The values should be laid out as a matrix with **nPaths** rows and  $dim \times (N - 1)$  columns. If quasi-random numbers are to be used, successive  $dim \times N$ -dimensional points should be stored in successive rows of the matrix, i.e. an ordering corresponding to **NAGGPUQUASIORIENT\_DIMVALS\_CONSEC**.
- 5: **cholCov[dim × dim]** – const float \* *Input*  
*On entry:* the matrix  $C$  which specifies the correlation structure of the Brownian bridge.  $C$  should be chosen such that  $CC' = \Sigma$  where  $Cov(X_s, X_t) = s(T - t)/T\Sigma$  for all  $s \leq t$  in  $[0, T]$ .
- 6: **bgIncs[dim × N × nPaths]** – float \* *Output*  
The variable  $N$  denotes the length **nTimes** of the **times** array passed to the initialization function nagCPUDepthBBIncInitA.  
*On exit:* the scaled increments 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  $0 \leq d < dim$ ,  $0 \leq i < N$  and  $0 \leq p < nPaths$ , then the scaled increment  $(x_{p,i+1}^d - x_{p,i}^d)/(t_{i+1} - t_i)$  will be stored at **bgIncs**[ $d + i * dim + p * dim * N$ ].  
**Note:** output is *transposed* relative to the equivalent GPU function.
- 7: **comm** – NagCPUDepthBBComm \* *Communication Data*  
NagCPUDepthBBComm is a structure which holds state and communication information and must not be modified in any way. Once all required bridge sample paths have been obtained, **comm** must be passed to nagCPUDepthBBCleanupA to free allocated system resources.
- 8: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call

to `naggpuErrorCopyMsg` will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

## 6 Error Indicators and Warnings

error → code = 100

*On entry:* the value of **comm** is NULL.

error → code = 101

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

error → code = 110

*On entry:* the value of **nPaths** does not satisfy the constraint listed above.

error → code = 111

*On entry:* the value of **dim** does not satisfy the constraint listed above.

error → code = 112

*On entry:* the value of **z** is NULL.

error → code = 113

*On entry:* the value of **cholCov** is NULL.

error → code = 114

*On entry:* the value of **bgIncs** is NULL.

## 7 Example

None.

---



# NAG Numerical Routines for GPUs Function Document

## nagCPUDepthBBCleanupA

### 1 Purpose

**nagCPUDepthBBCleanupA** frees system resources which were allocated by a previous call to **nagCPUDepthBBInitA** or **nagCPUDepthBBIncInitA**.

### 2 Specification

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

extern "C"
void nagCPUDepthBBCleanupA(NagCPUDepthBBComm *comm, NagGpuError *error)
```

### 3 Description

#### 3.1 Synchronization

This function is blocking but *will not* force synchronization between host and device. Control will not return to the calling program before this function has terminated.

### 4 References

None.

### 5 Arguments

- 1: **comm** – NagCPUDepthBBComm \* *Communication Data*  
The structure which was initialized by a previous call to **nagCPUDepthBBInitA** or **nagCPUDepthBBIncInitA**.
- 2: **error** – NagGpuError \* *Error Reporting*  
This parameter contains error information and should not be modified directly. Errors are indicated through the value of `error → code` which should be inspected after each call to this function. If `error → code = 0` then no error occurred. If `error → code ≠ 0` then an error was detected and a call to **naggpuErrorCopyMsg** will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

### 6 Error Indicators and Warnings

`error → code = 100`

*On entry:* the value of **comm** is NULL.

`error → code = 101`

*On entry:* **comm** has not been initialized, or the internal state of **comm** is corrupted.

### 7 Example

None.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuDepthBBComm

### 1 Purpose

**NagGpuDepthBBComm** is used by the library for communication between the GPU Brownian bridge generator functions. It is for internal library use and should not be modified by the user in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuDepthBBComm {
    int param1;
    void * param2;

    int param3;
    int *param4;
    int *param5;
    float *param6;
    int param7;
    bool param8;
}
```

### 3 Description

#### 3.1 Monitoring Launch Parameters

It is currently not possible to change or observe the launch parameters of the GPU Brownian bridge kernels. This will be added in a future release.

### 4 References

None.

### 5 Members

The structure definition is provided so that the library can be called from languages other than C/C++. All the members of this structure are private to the library and must not be modified in any way.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuMrg32k3aDeviceComm

### 1 Purpose

**NagGpuMrg32k3aDeviceComm** is used by the library for communication between the host and the GPU MRG32k3a pseudorandom number generator functions such as `naggpudevMrg32k3aUniformA`. It is for internal library use and should not be modified in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuMrg32k3aDeviceComm {
    int param1;
    int *param2;
    unsigned int *param3;
    unsigned int *param4;
    unsigned int *param5;
}
```

### 3 Description

To use the MRG32k3a device function generators, the memory address of a `NagGpuMrg32k3aDeviceComm` structure must be obtained from the device generator initialization function `naggpuMrg32k3aDeviceInitA`. This structure will reside in the GPU memory space and will contain communication data for use by the device function generator. This GPU memory address must then be passed to the MRG32k3a device generator initialization function `naggpudevMrg32k3aInitA`. Once all values have been obtained from the device function generators, the same `NagGpuMrg32k3aDeviceComm` structure memory address must be passed to `naggpuMrg32k3aDeviceCleanupA` to free allocated system resources.

### 4 References

None.

### 5 Members

The structure definition is provided so that the library can be called from languages other than C/C++. All the members of this structure are private to the library and must not be modified in any way.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuQuasiRandComm

### 1 Purpose

**NagGpuQuasiRandComm** is used by the library for communication between the GPU quasi-random number generator functions. It is for internal library use and should not be modified by the user in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuQuasiRandComm {
    NagGpuTuneOrigin tuneOrigin;
    NagGpuQuasiRandTune *tuneParamsUsed;

    int param1;
    int param2;
    int param3;
    unsigned int param4;
    unsigned int *param5;
    unsigned int *param6;
    NagGpuSobolDeviceComm *param7;
}
```

### 3 Description

#### 3.1 Monitoring Launch Parameters

The first two members of **NagGpuQuasiRandComm** may be used to monitor the launch configuration which was used for a particular GPU kernel. This will typically only be of interest to users wanting to fine tune the performance of the library's GPU functions.

Immediately following a successful call to one of the GPU quasi-random generator functions (such as `naggpuQuasiRandUniformA`), the **tuneParamsUsed** member of the **NagGpuQuasiRandComm** structure will contain the parameters which were used to launch the kernel. Only the members of **tuneParamsUsed** relevant to the kernel that was launched should be inspected. See `NagGpuQuasiRandTune` for further details on these members and their meanings.

Note that the **tuneParamsUsed** pointer is no longer valid after calling `naggpuQuasiRandCleanupA`. Parameters must be observed before calling the cleanup function.

### 4 References

None.

### 5 Members

The full structure definition is provided so that the library can be called from languages other than C/C++. The members of this structure not documented below are private to the library and must not be modified in any way.

1: **tuneOrigin** – NagGpuTuneOrigin

Indicates where the tuning data in **tuneParamsUsed** originated: was it supplied by the user (by passing a `NagGpuQuasiRandTune` to the generator routine), or was it a default value used by the library. This value must not be modified in any way.

2: **tuneParamsUsed** – NagGpuQuasiRandTune \*

Contains the members used to launch the GPU kernel. See NagGpuQuasiRandTune for further details. This value, and the values inside the NagGpuQuasiRandTune structure, must not be modified in any way.

**Note:** **tuneParamsUsed** is no longer valid after calling naggpuQuasiRandCleanupA.

---

# NAG Numerical Routines for GPUs Data Type Document

## NagGpuQuasiRandTune

### 1 Purpose

**NagGpuQuasiRandTune** provides parameters to tune the performance of the GPU quasi-random generators (such as `naggpuQuasiRandUniformA`).

The generators are based on parameterized kernels, and these parameters are given in this structure. This structure represents advanced features of the library **which are optional**: users do not need to provide launch parameters in order to use the GPU functions, since default values will be used.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuQuasiRandTune {
    int sblAThdsPerBlk;
    int sblABlksPerDim;
}
```

### 3 Description

The members of this structure control the launch parameters for the GPU quasi-random generator kernels. These parameters are passed to the kernels through the generator functions such as `naggpuQuasiRandUniformA`. The parameters used in the subsequent kernel launch may be queried through the **tuneParamsUsed** member of the `NagGpuQuasiRandComm` communication structure. If the call to the generator function succeeded, they will be the same as the parameters passed in by the user. If an error occurred, the values in **tuneParamsUsed** are unset and should be ignored.

As a starting point in tuning a function, it may be useful to launch a kernel with a given problem size and specify a NULL **NagGpuQuasiRandTune** pointer. In this case the library will use default launch parameters, which can be queried through the **tuneParamsUsed** member of the `NagGpuQuasiRandComm` communication structure. This could give some indication of where a search could begin.

#### 3.1 Background on Performance Tuning

CUDA compute kernels partition the computational load into independent blocks, each block being made up of a number of threads. Blocks cannot communicate, and all blocks have the same number of threads. The blocks can optionally be arranged in a 2D grid, although this is only to aid the programmer and does not impact performance. To launch a CUDA kernel on a graphics device, one specifies to the CUDA runtime system which kernel to launch, how many blocks to launch it with, and how many threads each block will have. The graphics device is (loosely speaking) made up of a number of processors, where each processor can run one or more blocks. Two or more processors cannot combine efforts to run a single block. The kernel launch can fail for a number of reasons, the most common being:

1. A badly written kernel – buffer overruns, underruns or segmentation faults
2. Requesting too many blocks. Graphics devices have a limited number of blocks that can be launched, which varies from card to card.
3. Requesting too many threads per block. Graphics devices have a limited number of threads each block can run, which varies from card to card.
4. Requesting too many resources. Each processor on the graphics device has a limited number of registers and shared memory. A CUDA kernel will use a given number of registers per thread (which depends on the kernel). If too many threads are launched, the block will need more registers than a processor can provide, and the launch will fail.

By specifying the number of blocks and the number of threads, the user is determining how the computational load is spread across the device and this will determine the runtime. For example, if only one block is launched on a device with several processors, the runtime performance will be poor since much of the device is idle. If too many blocks are launched, performance will also be poor due to the overhead of block scheduling and setup, and the fact that each block does very little work before being replaced by another.

It should be evident that tuning a CUDA kernel for performance is not always simple. In general the optimal launch parameters depend on

1. The particular CUDA kernel being considered, including the number of registers and shared memory it uses.
2. The graphics device the kernel is run on.
3. The computational load of the kernel.

The last point above means that optimal launch parameters can be dependent on problem size, e.g. launching a kernel with a smaller computational load may have different optimal launch parameters to the same kernel for a larger computational load.

### 3.2 Tuning and CUDA Errors

Note that tuning a kernel **can produce CUDA errors**: while the NAG functions will endeavour to indicate when a launch would have required too much shared memory, no attempt is made to determine whether sufficient registers are available for a launch to succeed. Passing a high number of threads to a register-heavy kernel (typically the double precision kernels) may result in a launch failure and a CUDA runtime error such as `cudaErrorLaunchOutOfResources`. When tuning a function, users should check the error codes to see whether a resource-related error occurred (such as a launch requiring too much shared memory or too many registers). If such an error occurred, the launch configuration should be discarded and `cudaGetLastError()` should be called to clear the CUDA error status. A new launch configuration can then be tried.

## 4 References

None.

## 5 Members

- 1: **sblAThdsPerBlk** – int

The number of threads per block for the Sobol' generator.

*Constraints:*

$32 \leq \text{sblAThdsPerBlk} \leq T$  where  $T$  is the maximum number of threads per block as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`);

**sblAThdsPerBlk** must be a power of 2.

- 2: **sblABlksPerDim** – int

The number of thread blocks which will cooperate to generate numbers from each dimension. By default the generator will launch one block for each dimension. If many points are generated from a low dimensional sequence, increasing **sblABlksPerDim** may lead to better work distribution on the GPU.

If  $d$  is the dimensionality of the sequence, the total number of blocks  $B$  that is launched is given by

$$B = \text{sblABlksPerDim} * d$$

*Constraints:*

$1 \leq \text{sblABlksPerDim} \leq B_x$  where  $B_x$  is the maximum number of blocks in the  $x$  dimension of the grid, as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`);

**sblABlksPerDim** must be a power of 2.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuRandComm

### 1 Purpose

**NagGpuRandComm** is used by the library for communication between the GPU pseudorandom number generator functions. It is for internal library use and should not be modified by the user in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuRandComm {
    NagGpuTuneOrigin tuneOrigin;
    NagGpuRandTune *tuneParamsUsed;

    int param1;
    int param2;
    unsigned int *param3;
    unsigned int *param4;
    unsigned int *param5;
    unsigned int *param6;
    int *param7;
    int param8;
    NagGpuMrg32k3aDeviceComm *param9;
    void *param10;
}
```

### 3 Description

#### 3.1 Monitoring Launch Parameters

The first two members of **NagGpuRandComm** may be used to monitor the launch configuration which was used for a particular GPU kernel. This will typically only be of interest to users wanting to fine tune the performance of the library's GPU functions.

Immediately following a successful call to one of the GPU pseudorandom generator functions (such as `naggpuRandUniformA`), the **tuneParamsUsed** member of the **NagGpuRandComm** structure will contain the parameters which were used to launch the kernel. Only the members of **tuneParamsUsed** relevant to the kernel that was launched should be inspected. See `NagGpuRandTune` for further details on these members and their meanings.

Note that the **tuneParamsUsed** pointer is no longer valid after calling `naggpuRandCleanupA`. Parameters must be observed before calling the cleanup function.

### 4 References

None.

### 5 Members

The full structure definition is provided so that the library can be called from languages other than C/C++. The members of this structure not documented below are private to the library and must not be modified in any way.

1: **tuneOrigin** – NagGpuTuneOrigin

Indicates where the tuning data in **tuneParamsUsed** originated: was it supplied by the user (by passing a NagGpuRandTune to the generator routine), or was it a default value used by the library. This value must not be modified in any way.

2: **tuneParamsUsed** – NagGpuRandTune \*

Contains the members used to launch the GPU kernel. See NagGpuRandTune for further details. This value, and the values inside the NagGpuRandTune structure, must not be modified in any way.

**Note:** **tuneParamsUsed** is no longer valid after calling naggpuRandCleanupA.

---

# NAG Numerical Routines for GPUs Data Type Document

## NagGpuRandTune

### 1 Purpose

**NagGpuRandTune** provides parameters to tune the performance of the GPU pseudorandom number generators (such as `naggpuRandUniformA`).

The generators are based on parameterized kernels, and these parameters are given in this structure. This structure represents advanced features of the library **which are optional**: users do not need to provide launch parameters in order to use the GPU functions, since default values will be used.

### 2 Specification

```
#include <nag_gpu.h>
struct NagGpuRandTune {
    int mrgOptATHdsPerBlk;
    int mrgOptAPtsPerThd;

    int mrgConATHdsPerBlk;
    int mrgConANumLoops;

    int mtANumBlks;
    int *mtAGen;

    int mrgRejOptATHdsPerBlk;
    int mrgRejOptANumBlks;
    void (*mrgRejConA)(int npts, int *nthds, int *nblks);
}
```

### 3 Description

The members of this structure control the launch parameters for the GPU pseudorandom number generator kernels. These parameters are passed to the kernels through the generator functions such as `naggpuRandUniformA`. The parameters used in the subsequent kernel launch may be queried through the **tuneParamsUsed** member of the `NagGpuRandComm` communication structure. If the call to the generator function succeeded, they will be the same as the parameters passed in by the user. If an error occurred, the values in **tuneParamsUsed** are unset and should be ignored.

Note that only the members of **NagGpuRandTune** relevant to the kernel that is to be launched need be set. For example, when launching the MRG32k3a consistent order kernel for uniform pseudorandom numbers, only **mrgConATHdsPerBlk** and **mrgConANumLoops** need be set.

As a starting point in tuning a function, it may be useful to launch a kernel with a given problem size and specify a NULL **NagGpuRandTune** pointer. In this case the library will use default launch parameters, which can be queried through the **tuneParamsUsed** member of the `NagGpuRandComm` communication structure. This could give some indication of where a search could begin.

#### 3.1 Background on Performance Tuning

CUDA compute kernels partition the computational load into independent blocks, each block being made up of a number of threads. Blocks cannot communicate, and all blocks have the same number of threads. The blocks can optionally be arranged in a 2D grid, although this is only to aid the programmer and does not impact performance. To launch a CUDA kernel on a graphics device, one specifies to the CUDA runtime system which kernel to launch, how many blocks to launch it with, and how many threads each block will have. The graphics device is (loosely speaking) made up of a number of processors, where each

processor can run one or more blocks. Two or more processors cannot combine efforts to run a single block. The kernel launch can fail for a number of reasons, the most common being:

1. A badly written kernel – buffer overruns, underruns or segmentation faults
2. Requesting too many blocks. Graphics devices have a limited number of blocks that can be launched, which varies from card to card.
3. Requesting too many threads per block. Graphics devices have a limited number of threads each block can run, which varies from card to card.
4. Requesting too many resources. Each processor on the graphics device has a limited number of registers and shared memory. A CUDA kernel will use a given number of registers per thread (which depends on the kernel). If too many threads are launched, the block will need more registers than a processor can provide, and the launch will fail.

By specifying the number of blocks and the number of threads, the user is determining how the computational load is spread across the device and this will determine the runtime. For example, if only one block is launched on a device with several processors, the runtime performance will be poor since much of the device is idle. If too many blocks are launched, performance will also be poor due to the overhead of block scheduling and setup, and the fact that each block does very little work before being replaced by another.

It should be evident that tuning a CUDA kernel for performance is not always simple. In general the optimal launch parameters depend on

1. The particular CUDA kernel being considered, including the number of registers and shared memory it uses.
2. The graphics device the kernel is run on.
3. The computational load of the kernel.

The last point above means that optimal launch parameters can be dependent on problem size, e.g. launching a kernel with a smaller computational load may have different optimal launch parameters to the same kernel for a larger computational load.

### 3.2 Tuning and CUDA Errors

Note that tuning a kernel **can produce CUDA errors**: while the NAG functions will endeavour to indicate when a launch would have required too much shared memory, no attempt is made to determine whether sufficient registers are available for a launch to succeed. Passing a high number of threads to a register-heavy kernel (typically the double precision kernels) may result in a launch failure and a CUDA runtime error such as `cudaErrorLaunchOutOfResources`. When tuning a function, users should check the error codes to see whether a resource-related error occurred (such as a launch requiring too much shared memory or too many registers). If such an error occurred, the launch configuration should be discarded and `cudaGetLastError()` should be called to clear the CUDA error status. A new launch configuration can then be tried.

## 4 References

None.

## 5 Members

- 1: **mrgOptAThdsPerBlk** – int

The number of threads per block for the MRG32k3a kernel with ordering `NAGGPURANDORDER_OPTIMAL`.

*Relevant distributions:* `naggpuRandUniformA`, `naggpuRandExpA` and `naggpuRandNormalA`.

*Constraint:*  $1 \leq \text{mrgOptAThdsPerBlk} \leq T$  where  $T$  is the maximum number of threads per block as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`).

2: **mrgOptAPtsPerThd** – int

The number of random variates each CUDA thread calculates in the MRG32k3a kernel with ordering **NAGGPURANDORDER\_OPTIMAL**.

*Relevant distributions:* naggpuRandUniformA, naggpuRandExpA and naggpuRandNormalA.

Let  $X_0, X_1, X_2, \dots$  be the random variates in **NAGGPURANDORDER\_CONSISTENT** order. The parallelization strategy of this kernel is simply to split the sequence into adjacent, non-overlapping subsequences, and to assign each subsequence to a CUDA thread. Let  $T = \text{mrgOptAThdsPerBlk}$  and  $P = \text{mrgOptAPtsPerThd}$ , let **d\_buff** be the storage buffer and let  $N$  be the number of random variates required. The number of blocks  $B_{xy}$  that is launched is approximately  $B_{xy} = N/TP$ . Let  $B = N/TP$ . Then for all integers  $0 \leq t < T$  and  $0 \leq p < P$  and  $0 \leq b < B$ , we have that

$$\mathbf{d\_buff}[t + pT + bPT] = X_{p+tP+bPT} \quad (1)$$

The ordering of the variates **d\_buff**[ $i$ ] for  $PTB \leq i < N$  is implementation specific. For the Normal distribution, the relation (1) will only hold if subsequences start at *even* offsets in the generator sequence: a sufficient condition for this is that **mrgOptAThdsPerBlk** be even.

*Constraint:*  $\text{mrgOptAPtsPerThd} \geq 1$ .

3: **mrgConAThdsPerBlk** – int

The number of threads per block for the MRG32k3a kernel with ordering **NAGGPURANDORDER\_CONSISTENT**.

*Relevant distributions:* naggpuRandUniformA, naggpuRandExpA and naggpuRandNormalA.

*Constraints:*

$1 \leq \text{mrgConAThdsPerBlk} \leq T$  where  $T$  is the maximum number of threads per block as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`);

**mrgConAThdsPerBlk** must be divisible by  $W$  where  $W = 16$  on devices of compute capability 1.3 or lower, and  $W = 32$  otherwise. See the CUDA runtime function `cudaGetDeviceProperties()` for details of determining the compute capability of a given device.

4: **mrgConANumLoops** – int

Controls the number of random variates computed by each block of the MRG32k3a kernel with ordering **NAGGPURANDORDER\_CONSISTENT**.

*Relevant distributions:* naggpuRandUniformA, naggpuRandExpA and naggpuRandNormalA.

For a given sample size  $N$ , the number of blocks  $B_{xy}$  that is launched is approximately

$$B_{xy} = \frac{N}{\text{mrgConAThdsPerBlk} * \text{mrgConANumLoops} * W}$$

where  $W = 16$  on devices of compute capability 1.3 or lower, and  $W = 32$  otherwise. See `cudaGetDeviceProperties()` for details of determining the compute capability of a given device. The number of random variates generated by each block is therefore approximately  $N/B_{xy}$ .

*Constraint:*  $\text{mrgConANumLoops} \geq 1$ .

5: **mtANumBlks** – int

The number of thread blocks to be launched for the MT19937 generator. This parameter is used for both optimal and consistent ordering.

*Relevant distributions:* naggpuRandUniformA, naggpuRandExpA and naggpuRandNormalA.

*Constraint:*  $1 \leq \text{mtANumBlks} \leq B_{\max_x}$  where  $B_{\max_x}$  is the maximum number of thread blocks in the  $x$  dimension of the grid, as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`).

6: **mtAGen**[**mtANumBlks**] – int \*

The number of random variates each CUDA thread block of the MT19937 generator will calculate. This parameter is used for both optimal and consistent ordering.

*Relevant distributions:* naggpuRandUniformA, naggpuRandExpA and naggpuRandNormalA.

Let  $X_0, X_1, X_2, \dots$  denote the sequence of uniform pseudorandom variates specified by the MT19937 algorithm. The algorithm is based on a recurrence where generating  $X_n$  requires the values of  $X_{n-624}$ ,  $X_{n-623}$  and  $X_{n-227}$  for each  $n \geq 0$  (we assume the seed has negative indices). This means that a maximum of 226 new values can be updated in parallel before there is a need to wait for the previous values to be computed (a synchronization point). In this form there is an inherent limit on the amount of parallelization which is possible. To scale the problem beyond this limit, it must be subdivided into several subproblems. If  $N$  random variates are required, then the sequence can be split into  $B$  subsequences where the  $i$ th subsequence is of length  $n_i$ , subject to the condition that  $\sum_{i=1}^B n_i = N$ . The subsequences start at locations corresponding to  $X_0, X_{n_1}, X_{n_1+n_2}, \dots$  in the original sequence.

For distributions where a single input uniform variate is mapped to a single output variate (e.g. through the inverse CDF method), the **NAGGPURANDORDER\_OPTIMAL** and **NAGGPURANDORDER\_CONSISTENT** orderings for MT19937 will agree. In other words, *there is no distinction between optimal and consistent order* and both options will adhere to the order specified by the MT19937 algorithm. For more complex output distributions where two or more uniforms are mapped to one or more output values, there will typically be no mapping between the output for optimal and consistent ordering, such as there is for the MRG32k3a.

The CUDA kernel launches **mtANumBlks** thread blocks where block  $i$  generates **mtAGen**[ $i$ ] variates for  $0 \leq i < \text{mtANumBlks}$ . Each thread block uses the skipahead functionality to advance the state of the generator to the location where it needs to start generating. Thus the first block can start generating immediately, while the second is performing a skip ahead of **mtAGen**[0] before it is able to start, etc. The values in **mtAGen** therefore control the skip ahead strategy followed by the generator.

The final requirement below that the members of **mtAGen** be even is due to the Box-Muller transform that is used to compute Normal random variates.

*Constraints:*

$\text{mtAGen}(i) \geq 0$  for all  $0 \leq i \leq \text{mtANumBlks} - 1$ ;  
 $\sum_{i=0}^{\text{mtANumBlks}-1} \text{mtAGen}(i) = N$  where  $N$  is the required number of random variates to be generated;  
 When calling naggpuRandNormalA with `order = NAGGPURANDORDER_CONSISTENT`, **mtAGen**( $i$ ) must be even for all  $0 \leq i \leq \text{mtANumBlks} - 2$ .

7: **mrgRejOptAThdsPerBlk** – int

The number of threads per block to be launched for the MRG32k3a rejection kernel with ordering **NAGGPURANDORDER\_OPTIMAL**.

*Relevant distributions:* naggpuRandGammaA.

*Constraints:*

$1 \leq \text{mrgRejOptAThdsPerBlk} \leq T$  where  $T$  is the maximum number of threads per block as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`);  
**mrgRejOptAThdsPerBlk** must be divisible by  $W$  where  $W = 16$  on devices of compute capability 1.3 or lower, and  $W = 32$  otherwise. See the CUDA runtime function `cudaGetDeviceProperties()` for details of determining the compute capability of a given device.

8: **mrgRejOptANumBlks** – int

The number of thread blocks to be launched for the MRG32k3a rejection kernel with ordering **NAGGPURANDORDER\_OPTIMAL**.

*Relevant distributions:* naggpuRandGammaA.

This kernel uses *independent substreams* (please see the Random Number Generators Chapter Introduction for a brief introduction to streams and substreams) to parallelise the rejection algorithm. Each CUDA thread takes the MRG32k3a state and advances it by  $2^{76i}$  steps, where  $i$  is the thread's index. For performance reasons, warps of threads cooperate when writing variates to global memory. As such there is no simple mapping between the output in optimal and consistent order. The state of each CUDA thread is stored after the kernel completes so that calls to this kernel can be mixed safely with calls to the other MRG32k3a kernels.

*Constraint:*  $1 \leq \text{mrgRejOptANumBlks} \leq B_{\max_x}$  where  $B_{\max_x}$  is the maximum number of thread blocks in the  $x$  dimension of the grid, as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`).

9: **mrgRejConA** – void (\*)(int npts, int \*nthds, int \*nblks)

A function returning the number of threads per block and number of thread blocks to be used for the MRG32k3a rejection kernel with ordering **NAGGPURANDORDER\_CONSISTENT**.

*Relevant distributions:* naggpuRandGammaA.

This kernel uses an iterative method to generate random variates in consistent order. An attempt is first made to generate the required number of variates, say  $n$ ; when the generator completes, the actual number of variates  $g$  that were generated is subtracted from the required number of variates, and an attempt is made to generate remaining  $n - g$  variates. The process continues until all  $n$  variates have been generated. This gives rise to the need for a tuning *function* since the generator will be called repeatedly with decreasing sample sizes. A single set of launch parameters used across all these different sample sizes will almost certainly be suboptimal.

The parameters for **mrgRejConA** are as follows:

**npts**, *input*: the number of pseudorandom variates to be generated.

**nthds**, *output*: the number of threads per block to be used.

**nblks**, *output*: the number of thread blocks to launch.

*Constraints:*

`mrgRejConA`  $\neq$  NULL;

$1 \leq \text{nthds} \leq T$  where  $T$  is the maximum number of threads per block as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`);

**nthds** must be divisible by  $W$  where  $W = 16$  on devices of compute capability 1.3 or lower, and  $W = 32$  otherwise. See the CUDA runtime function `cudaGetDeviceProperties()` for details of determining the compute capability of a given device;

$1 \leq \text{nblks} \leq B_{\max_x}$  where  $B_{\max_x}$  is the maximum number of thread blocks in the  $x$  dimension of the grid, as reported by the CUDA runtime system (see the CUDA runtime function `cudaGetDeviceProperties()`).



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuSobolDeviceComm

### 1 Purpose

**NagGpuSobolDeviceComm** is used by the library for communication between the host and the GPU Sobol' quasi-random generator functions such as `naggpudevSobolUniformA`. It is for internal use and should not be modified in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuSobolDeviceComm {
    int param1;
    const unsigned int *param2;
    const unsigned int *param3;
}
```

### 3 Description

To use the Sobol' device function generators, the memory address of a `NagGpuSobolDeviceComm` structure must be obtained from the device generator initialization function `naggpuSobolDeviceInitA`. This structure will reside in the GPU memory space and will contain communication data for use by the device function generator. This GPU memory address must then be passed to the Sobol' device generator initialization function `naggpudevSobolInitA`. Once all values have been obtained from the device function generators, the same `NagGpuSobolDeviceComm` structure memory address must be passed to `naggpuSobolDeviceCleanupA` to free allocated system resources.

### 4 References

None.

### 5 Members

The structure definition is provided so that the library can be called from languages other than C/C++. All the members of this structure are private to the library and must not be modified in any way.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagCPUDepthBBComm

### 1 Purpose

**NagCPUDepthBBComm** is used by the library for communication between the serial, host-only Brownian bridge generator functions. It is for internal library use and should not be modified by the user in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagCPUDepthBBComm {
    int param1;
    int *param2;
    int *param3;
    float *param4;
    int param5;
    int param6;
    bool param7;
}
```

### 3 Description

None.

### 4 References

None.

### 5 Members

The structure definition is provided so that the library can be called from languages other than C/C++. All the members of this structure are private to the library and must not be modified in any way.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagCPUQuasiRandComm

### 1 Purpose

**NagCPUQuasiRandComm** is used by the library for communication between the serial, host-only quasi-random number generator functions. It is for internal library use and should not be modified by the user in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagCPUQuasiRandComm {
    int param1;
    int param2;
    int param3;
    unsigned int param4;
    unsigned int *param5;
    unsigned int *param6;
}
```

### 3 Description

None.

### 4 References

None.

### 5 Members

The structure definition is provided so that the library can be called from languages other than C/C++. All the members of this structure are private to the library and must not be modified in any way.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagCPURandComm

### 1 Purpose

**NagCPURandComm** is used by the library for communication between the serial, host-only pseudorandom number generator functions. It is for internal library use and should not be modified by the user in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagCPURandComm {
    int param1;
    int param2;
    unsigned int *param3;
    int param4;
    int param5;
    int param6;
    int param7;
}
```

### 3 Description

None.

### 4 References

None.

### 5 Members

The structure definition is provided so that the library can be called from languages other than C/C++. All the members of this structure are private to the library and must not be modified in any way.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuRandGen

### 1 Purpose

Identifies the base pseudorandom number generators that are available. A specific generator is selected by passing one of the symbols given below to the generator initialization functions such as `naggpuRandInitA`.

### 2 Specification

```
#include <nag_gpu.h>
enum NagGpuRandGen {
    NAGGPURANDGEN_MRG32K3A = 200,
    NAGGPURANDGEN_MT19937
}
```

### 3 Description

The various pseudorandom number generators that are available are listed in the Random Number Generators Chapter Introduction together with any relevant implementation details.

### 4 References

L'Ecuyer P (1999) Good parameter sets for combined multiple recursive random number generators *Operations Research* **47:1** 159–164

Matsumoto M and Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator *ACM Transactions on Modelling and Computer Simulations*

### 5 Symbols

1: **NAGGPURANDGEN\_MRG32K3A**

Identifies L'Ecuyer's multiple recursive generator MRG32k3a discussed in L'Ecuyer (1999).

2: **NAGGPURANDGEN\_MT19937**

Identifies the pseudorandom number generator MT19937 developed in Matsumoto and Nishimura (1998).

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuQuasiGen

### 1 Purpose

Identifies the base quasi-random number generators available in the library. A specific generator is selected by passing one of the symbols given below to the generator initialization functions such as naggpuQuasiRandInitA.

### 2 Specification

```
#include <nag_gpu.h>
enum NagGpuQuasiGen {
    NAGGPUQUASIGEN_SOBOL = 300
}
```

### 3 Description

Low discrepancy (quasi-random) sequences are used in numerical integration, simulation and optimization. Like pseudorandom numbers they are uniformly distributed, but they are not statistically independent. Quasi-random sequences are designed to give a more even distribution in multidimensional space (uniformity), and are often more efficient than pseudorandom numbers in multidimensional Monte Carlo methods.

Let  $x^1, x^2, \dots, x^N$  be a sequence of  $d$ -dimensional points in the unit cube  $I^d = [0, 1]^d$ . Let  $G$  be a subset of  $I^d$  and define the counting function  $S_N(G)$  as the number of  $d$ -dimensional points  $x^i \in G$ . For each point  $x = (x_1, x_2, \dots, x_d) \in I^d$ , let  $G_x$  be the rectangular  $d$ -dimensional region

$$G_x = [0, x_1] \times [0, x_2] \times \dots \times [0, x_d]$$

with volume  $x_1 \cdot x_2 \cdot \dots \cdot x_d = \prod_{i=1}^d x_i$ . Then one measure of the uniformity of the points  $x^1, x^2, \dots, x^N$  is the so-called star discrepancy:

$$D_N^*(x^1, x^2, \dots, x^N) = \sup_{x \in I^d} \left| S_N(G_x) - N \prod_{i=1}^d x_i \right|$$

which satisfies the inequality

$$D_N^*(x^1, x^2, \dots, x^N) \leq C_d (\log N)^d + O((\log N)^{d-1}) \quad \text{for all } N \geq 2.$$

The principal aim in the construction of low-discrepancy sequences is to find sequences of points in  $I^d$  with a bound of this form where the constant  $C_d$  is as small as possible.

### 4 References

Bratley P and Fox B L (1988) Algorithm 659: implementing Sobol's quasirandom sequence generator *ACM Trans. Math. Software* **14(1)** 88–100

Jäckel P (2002) *Monte Carlo Methods in Finance* Wiley Finance Series, John Wiley and Sons, England

Joe S and Kuo F Y (2003) Remark on Algorithm 659: implementing Sobol's quasirandom sequence generator *ACM Trans. Math. Software (TOMS)* **29** 49–57

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

## 5 Symbols

### 1: NAGGPUQUASIGEN\_SOBOL

Identifies the Sobol' quasi-random generator. The implementation follows that given in Bratley and Fox (1988). The generator can produce sequences in up to 50000 dimensions. For the first 21201 dimensions, the direction numbers proposed by Joe and Kuo (2008) and Joe and Kuo (2003) are used. Dimensions higher than 21201 follow a proposal by Jäckel (2002) and use a pseudorandom number generator to create random direction numbers.

---

# NAG Numerical Routines for GPUs Data Type Document

## NagGpuQuasiOrient

### 1 Purpose

Identifies the orientation of output from the quasi-random number generators. If the generator output is viewed as a matrix, these options effectively allow one to transpose the matrix.

### 2 Specification

```
#include <nag_gpu.h>
enum NagGpuQuasiOrient {
    NAGGPUQUASIORIENT_DIMVALS_CONSEC = 500,
    NAGGPUQUASIORIENT_DIMVALS_SCATT
}
```

### 3 Description

Quasi-random sequences are made up of one or more multidimensional points, with each point composed of several one dimensional values. Fix some integer  $d \geq 1$  and consider a  $d$ -dimensional quasi-random sequence  $x^0, x^1, \dots$  so that each point  $x^j = (x_1^j, x_2^j, \dots, x_d^j)$  is composed of  $d$  one dimensional values. When stored in memory, these points can typically be laid out in two ways: either  $d$ -dimensional points are stored one after the other, so that the one dimensional values of each point are laid out *consecutively*; or dimensions can be grouped across all the  $d$ -dimensional points so that the one dimensional values of any given  $d$ -dimensional point are *scattered*.

The symbols below select which memory layout is used by the quasi-random generators.

### 4 References

None.

### 5 Symbols

#### 1: NAGGPUQUASIORIENT\_DIMVALS\_CONSEC

Specifies that one dimensional values from a  $d$ -dimensional point are laid out consecutively in memory. Consider  $N$  points from a  $d$ -dimensional quasi-random sequence stored in a linear array  $A$ . The  $j$ -th dimension  $x_n^j$  of the  $n$ -th point will be stored at location  $A[n * d + (j - 1)]$  for every  $0 \leq n < N$  and  $1 \leq j \leq d$ .

#### 2: NAGGPUQUASIORIENT\_DIMVALS\_SCATT

Specifies that dimensions are grouped across all the  $d$ -dimensional points. All points from dimension 1 will be stored first, followed by all points from dimension 2, and so on. Consider  $N$  points from a  $d$ -dimensional quasi-random sequence stored in a linear array  $A$ . The  $j$ -th dimension  $x_n^j$  of the  $n$ -th point will be stored at location  $A[N * (j - 1) + n]$  for every  $0 \leq n < N$  and  $1 \leq j \leq d$ .



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuRandOrder

### 1 Purpose

Identifies an output ordering for the variates produced by the pseudorandom number generators.

### 2 Specification

```
#include <nag_gpu.h>
enum NagGpuRandOrder {
    NAGGPURANDORDER_OPTIMAL = 600,
    NAGGPURANDORDER_CONSISTENT
}
```

### 3 Description

#### 3.1 Permutations

Pseudorandom number generators are called ‘pseudo’ random since the numbers they produce are not truly random. Indeed, there is a strict deterministic algorithm for computing the next number from one or more preceding numbers, and so the numbers are not independent in the true sense of the word. The art of constructing a good generator lies in choosing this deterministic algorithm such that, for most practical applications where the generator is to be used, the numbers are ‘sufficiently random’ in the sense that the application cannot determine the difference between the pseudorandom numbers and truly random numbers.

Pseudorandom generators produce numbers in a long sequence which we denote  $X_0, X_1, X_2, X_3, \dots$ . For good generators, most applications will perform equally well if given the sequence  $X_1, X_0, X_3, X_2, \dots$ , or the sequence  $X_{10}, X_{15}, X_{20}, X_{25}, \dots$ , as they would if given the original sequence  $X_0, X_1, X_2, \dots$ . The application cannot distinguish between the three sequences, and will produce more or less the same result. For good generators therefore, the output order can be permuted (somewhat) without adversely affecting the majority of applications which use the generator.

Clearly some constraints should be placed on the types of permutations allowed: a permutation which rearranges all the pseudorandom numbers to be in increasing order will no doubt have a serious impact on the application. However there should be a reasonably large class of benign permutations which do not adversely affect most applications. Such permutations would probably result in a slightly different numerical answer, but this should be in line with the variability inherent in using a (pseudo)random number generator in the first place. For example, initializing a generator with two different seeds and computing two Monte Carlo integrals of a given function will yield two different answers: the difference should be in line with the standard deviation of the Monte Carlo integral estimate.

#### 3.2 Performance Implications

For computer implementations of the generator, it is not always efficient to output the numbers in the original order  $X_0, X_1, X_2, \dots$ . This is often the case for GPUs. Due to the design of the hardware, it may be faster to output the numbers in some permuted order rather than trying to maintain the original order. This *performance-optimal* order is dependent on the launch configuration of the CUDA kernel, so that launching with different numbers of threads and blocks could result in different permutations of the output values. The types of permutations are typically of the form

$$\begin{array}{cccccc} X_0, & X_n, & X_{2n}, & \dots, & X_{Tn}, & \\ X_1, & X_{n+1}, & X_{2n+1}, & \dots, & X_{Tn+1}, & \\ \dots, & & & & & \\ X_{n-1}, & X_{2n-1}, & X_{3n-1}, & \dots, & X_{(T+1)n-1}, & X_{(T+1)n}, & X_{(T+2)n}, & \dots \end{array}$$

for some given integers  $n$  and  $T$ . (The table-style layout above is simply to ease presentation: the rows should be concatenated one after another to obtain the layout in memory.)

### 3.3 Statistical Properties and Correctness

For many applications such a permutation will be benign and should not affect interpretations of the results. However it should be emphasized that permuting the output of a generator in effect creates a *new* pseudorandom generator, the statistical properties of which are not necessarily known. Some users may be uncomfortable with this, and may prefer to maintain the original ordering of the generator on the grounds of statistical correctness.

In other cases, a GPU implementation of an application may have to be compared to a reference CPU implementation, with the aim of obtaining the same results (to some appropriate tolerance) from both. In this case the order in which the pseudorandom numbers are *used* becomes important: for example for some interest rate models, constructing a sample path from the sequence  $X_0, X_1, X_2, X_3, \dots$  will yield a different path than if the sequence  $X_1, X_0, X_3, X_2, \dots$  were used. In this case it may be easier to maintain the original ordering of the sequence simply so that numbers can be read back and used in the correct order.

### 3.4 Implementation

The GPU pseudorandom number generators can output numbers either in an ordering aimed at peak performance, or in the original order specified by the generator algorithm. The performance ordering is referred to as *optimal ordering*, while the original generator order is referred to as *consistent ordering*, and the choices are indicated to the generator functions by passing one of the symbols below. Much work has gone into optimising the performance of the consistent order generators, and in many cases it is comparable to that of the optimal order versions. The double precision functions are typically worst affected.

Both the consistent and optimal order generators are based on parameterized GPU kernels. The parameters are called *tuning parameters*. They control how the compute load is distributed across the GPU and hence control the performance. The consistent order generators will produce output in the same order regardless of the tuning parameter values. For the optimal order generators, the values of the tuning parameters affect the output ordering. The tuning parameters are contained in the NagGpuRandTune structure: please see the documentation there for further information.

By default, the library will choose tuning parameters based on distribution, precision, base generator, problem size (number of random values required) and GPU architecture, with the aim of maximizing performance. For optimal ordering therefore, when allowing the library to select default tuning parameters (by passing a null NagGpuRandTune pointer to the generator functions such as naggpuRandUniformA), the following behaviour can be expected: when generating  $n$  points from a given distribution and base generator, the output ordering may vary between one GPU and another, between single and double precision on the same GPU, and the ordering for  $n$  points may be different from the ordering for  $n + 1$  points for the same precision on the same GPU. By contrast, the consistent order generators will always maintain the same ordering, even though the library may select different default tuning parameters based on distribution, precision, base generator, problem size and GPU, in order to maximize performance.

Tuning parameters can be specified directly to the generator functions (see NagGpuRandTune). For consistent order, these parameters will affect performance only, but for optimal order they may affect performance and output order. User supplied tuning parameters will override any default library choices: for optimal order generators, the associated output ordering will thus be maintained across different platforms.

It is worth noting that for optimal order generators, fixing an ordering by specifying a set of tuning parameters could also be useful when comparing GPU and CPU applications to ensure that numbers are read back in a certain order. For example, one could launch an optimal order MRG32k3a generator with  $T$  threads and  $B$  blocks and then write a kernel with  $T$  threads and  $B$  blocks which reads the numbers back so that each CUDA thread reads a segment  $X_i, X_{i+1}, X_{i+2}, X_{i+3}, \dots$  from the original generator sequence. This could help achieve coalesced memory transfers.

### 3.5 Ordering and Non-Uniform Distributions

The notion of ‘original order of the generator’ only exists for the uniform distribution, since pseudorandom generators are only designed to produce uniform pseudorandom numbers. There are several methods of transforming uniform numbers into other distributions. Where such a transform requires only a single uniform random number (e.g. through the inverse CDF method), the notion of ‘original order’ is unambiguous and means that each uniform random number is replaced by its transformed value.

However where more than one uniform number is required (e.g. through a rejection method), the notion is less clear. In this case the documentation for the relevant generator function *may* state how the uniform numbers were used and what the output ordering is, but if these relationships are too involved the information will be omitted. In all cases, the output from consistent order GPU generators will agree with the output from their serial CPU counterparts.

## 4 References

None.

## 5 Symbols

1: **NAGGPURANDORDER\_OPTIMAL**

Specifies that the output order may be permuted to improve performance.

2: **NAGGPURANDORDER\_CONSISTENT**

Specifies that output must be in the order specified by the generator algorithm. The output ordering will remain unchanged (*consistent*) across different GPU architectures, problem sizes and launch configurations. This is the ordering used by all the serial CPU generator functions such as nagCPURandUniformA.

---



# NAG Numerical Routines for GPUs Data Type Document

## NagGpuScramTypes

### 1 Purpose

Identifies the types of bit-scrambling which can be applied to the quasi-random number generators. A specific type of scrambling is selected by passing one of the symbols given below to the generator initialization functions such as `naggpuQuasiRandInitA`.

### 2 Specification

```
#include <nag_gpu.h>

enum NagGpuScramTypes {
    NAGGPUSCRAMTYPES_NONE = 400,
    NAGGPUSCRAMTYPES_OWEN,
    NAGGPUSCRAMTYPES_FAURE_TEZUKA,
    NAGGPUSCRAMTYPES_OWEN_FAURE_TEZUKA
}
```

### 3 Description

Scrambled quasi-random sequences are an extension of standard quasi-random sequences. They attempt to eliminate the bias inherent in a quasi-random sequence while retaining its low-discrepancy properties. The use of a scrambled sequence allows error estimation of Monte Carlo results by performing a number of iterates (i.e. computing several Monte Carlo integrations of the same function) and then computing the variance of the sample of different results. Scrambling can also improve the high dimensional performance of a quasi-random generator by destroying systematic dependencies between dimensions (e.g. as is sometimes observed in two or three dimensional projections). These dependencies are often caused by poor choices of initialization data.

This implementation of scrambled quasi-random sequences is based on TOMS Algorithm 823 and details can be found in the accompanying paper, Hong and Hickernell (2003). Three methods of scrambling are supplied; the first is a restricted form of Owen's scrambling (Owen (1995)), the second is based on the method of Faure and Tezuka (2000) and the third method is a combination the first two.

### 4 References

Faure H and Tezuka S (2000) Another random scrambling of digital (t,s)-sequences *Monte Carlo and Quasi-Monte Carlo Methods* Springer-Verlag, Berlin, Germany (eds K T Fang, F J Hickernell and H Niederreiter)

Hong H S and Hickernell F J (2003) Algorithm 823: implementing scrambled digital sequences *ACM Trans. Math. Software* **29:2** 95–109

Owen A B (1995) Randomly permuted (t,m,s)-nets and (t,s)-sequences *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, Lecture Notes in Statistics* **106** Springer-Verlag, New York, NY 299–317 (eds H Niederreiter and P J-S Shiue)

### 5 Symbols

- 1: **NAGGPUSCRAMTYPES\_NONE**  
No scrambling is to be performed.

**2: NAGGPUSCRAMTYPES\_OWEN**

The restricted Owen's scrambling described in Hong and Hickernell (2003) is to be performed.

**3: NAGGPUSCRAMTYPES\_FAURE\_TEZUKA**

The Faure-Tezuka scrambling described in Hong and Hickernell (2003) is to be performed.

**4: NAGGPUSCRAMTYPES\_OWEN\_FAURE\_TEZUKA**

The combined Owen-Faure-Tezuka scrambling described in Hong and Hickernell (2003) is to be performed.

---

# NAG Numerical Routines for GPUs Data Type Document

## NagGpuTuneOrigin

### 1 Purpose

Identifies the origin of the parameters used to launch a GPU kernel.

### 2 Specification

```
#include <nag_gpu.h>
enum NagGpuTuneOrigin {
    NAGGPOTUNEORIGIN_NA = 100,
    NAGGPOTUNEORIGIN_DEFAULT,
    NAGGPOTUNEORIGIN_USER,
    NAGGPOTUNEORIGIN_AUTO
}
```

### 3 Description

Several of the GPU functions in the library are based on parameterized kernels. Default values for these parameters are stored in the library, but users are allowed to override these and specify different launch configurations. This can improve performance since the optimal parameter values are highly dependent on the type of graphics card that is being used. For more information on performance tuning, consult the documentation for any of the tuning structures, e.g. `NagGpuRandTune`.

The symbols below identify where a set of tuning parameters originated and distinguish default values stored in the library from values specified by the user.

### 4 References

None.

### 5 Symbols

- 1: **NAGGPOTUNEORIGIN\_NA**  
The function does not use tuning parameters.
  - 2: **NAGGPOTUNEORIGIN\_DEFAULT**  
The tuning parameters are default values used by the library.
  - 3: **NAGGPOTUNEORIGIN\_USER**  
Tuning parameters were supplied by the user.
  - 4: **NAGGPOTUNEORIGIN\_AUTO**  
This value is reserved and is currently unused.
-



# **NAG Numerical Routines for GPUs**

## **Table of Contents: Error Handling**

### **Chapter Introduction**

### **Error Handling Functions**

naggpuErrorCopyMsg

### **List of Structures**

NagGpuError

---



# NAG Numerical Routines for GPUs Chapter Introduction

## Error Handling

### Contents

<b>1</b>	<b>Scope of the Chapter</b> .....	2
<b>2</b>	<b>Library Error Handling</b> .....	2
<b>3</b>	<b>Functionality Index</b> .....	2
3.1	Error Handling Functions .....	2
<b>4</b>	<b>References</b> .....	2

## 1 Scope of the Chapter

This chapter contains functions and structures for reporting errors.

## 2 Library Error Handling

All error handling is implemented through the `NagGpuError` structure which contains information relating to errors that occur at runtime. This structure does not need to be initialized and a single instance can be used across multiple library functions calls. Note that passing a NULL `NagGpuError` pointer to a library function will result in undefined behaviour, possibly a segmentation fault.

On exit from a library function, a non-zero value of `NagGpuError.code` indicates that some error occurred, and the documentation for that function should be consulted for the causes of the error. The documentation will list the possible errors, the causes, and the corresponding values of `NagGpuError.code`.

Alternatively, a null terminated ANSI C string describing the error can be obtained at runtime. On exit from a function, the `msgLength` member of `NagGpuError` will return the length (including null terminator) of the error message. A buffer of that size should then be allocated and passed, together with the `NagGpuError` structure, to `naggpuErrorCopyMsg`. Please see the `naggpuErrorCopyMsg` documentation for further details.

Functions which interact with the GPU can generate CUDA errors. Any CUDA errors that these functions encounter will be indicated in the `NagGpuError` structure and will also be returned to the user. On entry, these functions will check the CUDA runtime error status, and if any prior error is encountered, it will be indicated to the user and the function will return. The CUDA error checking will *not* clear the CUDA runtime error status: this is equivalent to calling `cudaPeekAtLastError()` and not `cudaGetLastError()` in the CUDA runtime library. Clearing the CUDA error status is left to the user. Note that **CUDA errors returned by library functions may relate to previous calls to the library, another GPU library, the CUDA runtime library, or the user's own GPU code.** Please see NVIDIA CUDA (2011) for detailed information on the CUDA runtime error system, and how to trap CUDA errors correctly when executing code asynchronously on the GPU.

## 3 Functionality Index

### 3.1 Error Handling Functions

Error handling,  
 retrieve an error message ..... `naggpuErrorCopyMsg`

## 4 References

NVIDIA CUDA (2011) *Programming Guide Version 4.0* <http://www.nvidia.com/cuda>

---

# NAG Numerical Routines for GPUs Function Document

## naggpuErrorCopyMsg

### 1 Purpose

**naggpuErrorCopyMsg** copies a character string describing an error message into a buffer allocated by the user.

### 2 Specification

```
#include <nag_gpu.h>

extern "C"
void naggpuErrorCopyMsg(char *buff, const NagGpuError *error)
```

### 3 Description

The `NagGpuError` structure contains information relating to errors that occur at runtime. It does not need to be initialized and a single instance can be used across multiple library functions calls. On exit from a library function, a non-zero value of `NagGpuError.code` indicates that some error occurred, and the documentation for that function should be consulted. The documentation will list the possible errors, the causes, and the corresponding values of `NagGpuError.code`.

Alternatively, a null terminated ANSI C string describing the error can be obtained at runtime. On exit from a library function, the **msgLength** member of `NagGpuError` will return the length (including null terminator) of the error message. A buffer of that size should then be allocated and passed, together with the `NagGpuError` structure, to `naggpuErrorCopyMsg`.

This function copies the character string describing the error to the user allocated buffer. The following code snippet illustrates a typical use of this function:

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

void checkError(const NagGpuError *error)
{
    using namespace std;

    if(error->code != 0) {
        char * buff = new char[error->msgLength];
        naggpuErrorCopyMsg(buff, error);
        cout << buff << endl;
        delete[] buff;
        // A simple solution: exit
        exit(-1);
    }
}

void myFunc()
{
    NagGpuError err;

    // ... user code

    naggpuCallToSomeNagFunc(..., &err);
    checkError(&err);

    // ... further calls to GPU library
}
```

**Note:** Passing a NULL NagGpuError pointer to **error** will result in undefined behaviour, possibly a segmentation fault. Passing a NULL pointer to **buff** is safe, but will result in the function returning without doing anything.

### 3.1 Synchronization

This function is blocking, but will **not** force synchronization between the host and the device.

## 4 References

None.

## 5 Arguments

- 1: **buff**[**error** → **msgLength**] – char \* *Output*  
A character buffer into which the message will be copied.  
*On exit:* contains a null terminated message describing the error.
- 2: **error** – const NagGpuError \* *Input*  
*On entry:* the structure describing the error. If this is NULL, the behaviour is undefined and a segmentation fault may occur.  
*Constraint:* error ≠ NULL.

## 6 Error Indicators and Warnings

None.

## 7 Example

There is no example program specifically for this function since it is used by all the other example programs. Please see any of the example programs for usage of this function.

---

# NAG Numerical Routines for GPUs Data Type Document

## NagGpuError

### 1 Purpose

**NagGpuError** is used to indicate errors to the user, and is used in conjunction with `naggpuErrorCopyMsg`. It must not be modified directly in any way.

### 2 Specification

```
#include <nag_gpu.h>

struct NagGpuError {
    int code;
    int msgLength;
    int subCodes[10];

    char param2[256];
    int param3;
    void (*param4)(char*, const NagGpuError*);
}
```

### 3 Description

The `NagGpuError` structure contains information relating to errors that occur at runtime. It does not need to be initialized and a single instance can be used across multiple library functions calls. On exit from a library function, a non-zero value of `NagGpuError.code` indicates that some error occurred, and the documentation for that function should be consulted. The documentation will list the possible errors, the causes, and the corresponding values of `NagGpuError.code`.

Alternatively, a null terminated ANSI C string describing the error can be obtained at runtime. On exit from a library function, the **msgLength** member of `NagGpuError` will return the length (including null terminator) of the error message. A buffer of that size should then be allocated and passed, together with the `NagGpuError` structure, to `naggpuErrorCopyMsg`.

Only the members **code** and **msgLength** should be queried by the user - the remaining members are private to the library and should not be modified in any way.

### 4 References

None.

### 5 Members

The full structure definition is provided so that the library can be called from languages other than C/C++. The members of this structure not documented below are private to the library and must not be modified in any way.

1: **code** – int

The error flag. Zero means no error occurred, while a non-zero value indicates that some error occurred during execution. This value must not be altered directly by the user.

2: **msgLength** – int

The length (including null terminator) of the ANSI C character string which describes the error indicated by **code**. This value must not be altered directly by the user. The string can be obtained by calling `naggpuErrorCopyMsg`.

3: **subCodes**[10] – int

Elements of this array may be used by some functions in the library to provide additional information. The documentation for a function will state whether this array is used, and if so what information it contains.

---