

NAG Numerical Functions for GPUs Function Document

nag_gpu_sobol_uniform

1 Purpose

nag_gpu_sobol_uniform generates a scrambled, uniformly distributed, low discrepancy sequence in the S -dimensional unit cube $I^S = [0, 1]^S$, as proposed by Sobol.

The initialization function `nag_gpu_sobol_init` must be called prior to the first call to **nag_gpu_sobol_uniform**. Successive calls to **nag_gpu_sobol_uniform** will generate successive values along the sequence. Once all desired values have been obtained, the function `nag_gpu_sobol_cleanup` must be called to free allocated system resources.

The dimension of the Sobol sequence is given by the `dim` parameter in `nag_gpu_sobol_init`. All references to `dim` below refer to that value.

2 Specification

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

```
void nag_gpu_sobol_uniform (int npts, float *d_P, Nag_Gpu_Comm *comm)
```

```
void nag_gpu_sobol_uniform (int npts, double *d_P, Nag_Gpu_Comm *comm)
```

3 References

None.

4 Arguments

1: **npts** – int *Input*

On entry: the number of quasi-random points required.

Constraint: $1 \leq \text{npts}$.

2: **d_p[npts × dim]** – float * *Output*

3: **d_p[npts × dim]** – double * *Output*

This overloaded parameter can take type double or float.

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

On exit: the `npts` quasi-random numbers of dimension `dim` from a uniform distribution. For the n th Sobol point $x_n = (x_n^1, x_n^2, \dots, x_n^{\text{dim}})$ with $0 \leq n < \text{npts}$, the d th dimension x_n^d will be stored at location `d_P[(d-1)*npts+n]`. In other words, the first `npts` values correspond to dimension 1, the second `npts` to dimension 2, and so on.

4: **comm** – Nag_Gpu_Comm * *Communication Data*

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

The structure initialized by a previous call to `nag_gpu_sobol_init`.

5 Error Indicators and Warnings

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

6 Example

This example prints the first 64 Sobol numbers of dimension 5 generated by `nag_gpu_sobol_uniform` after initialization by `nag_gpu_sobol_init`.

6.1 Program Text

```

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

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

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

////////////////////////////////////
// define precision
////////////////////////////////////
#define PRS float

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

int main(int argc, char **argv)
{
    FILE *fpout = stdout;
    //fopen("nag_gpu_sobol_uniform_example.results.txt", "w");

    // Max dims supported by Sobol generator is 50,000
    const int dim = 40;
    // Generate batchSize numbers at a time on the GPU. For large sample
    // sizes this can avoid out-of-memory errors
    const int batchSize = 50;

    // Total number of batches to generate
    const int nBatches = 10;

    PRS
    // Host (PC) storage for batch-generated GPU Sobol numbers
    *h_GpuBatchNums = 0,
    // Device (GPU) storage for batch-generated Sobol numbers
    *d_GpuBatchNums = 0;

    Nag_Gpu_Comm
    // Communication structure pointer for GPU generator
    comm;

    // Scrambling vector for GPU Sobol
    unsigned int *v = 0;
    // Offset into Sobol sequence
    const int offset = 0;

    // Number of dimensions to print: must be less than dim

```

```

const int printDim = 5;
// Number of Sobol values to print
int nPrint = 64;
// Make sure we have enough numbers
nPrint =
    (nPrint > batchSize*nBatches
     ? batchSize*nBatches : nPrint);
// Storage for numbers we'll print
PRS *h_print = 0;
// Offset into h_print at which to store GPU numbers
int printOffset = 0;

// Greeting
fprintf(fpout, "\n\n NAG GPU Sobol Example Program: uniform\n\n");

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

// Allocate memory
if (!(h_GpuBatchNums = (PRS*)malloc(sizeof(PRS)*dim*batchSize)) ) {
    fprintf(stderr, "h_GpuBatchNums allocation failure\n");
    goto END;
}
if (!(cudaSuccess == cudaMalloc((void **)&d_GpuBatchNums,
    sizeof(PRS)*dim*batchSize))) {
    fprintf(stderr, "d_GpuBatchNums allocation failure\n");
    goto END;
}
if (!(v = (unsigned int*)malloc(sizeof(unsigned int)*dim)) ) {
    fprintf(stderr, "v allocation failure\n");
    goto END;
}
if (!(h_print = (PRS*)malloc(sizeof(PRS)*printDim*nPrint)) ) {
    fprintf(stderr, "h_print allocation failure\n");
    goto END;
}

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

// Initialise GPU Sobol generator
nag_gpu_sobol_init(dim, offset, v, &comm);

for(int batchCount = 0; batchCount < nBatches; batchCount++) {
    // Launch GPU computation asynchronously
    nag_gpu_sobol_uniform(batchSize, d_GpuBatchNums, &comm);

    /*
     * 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
     */

    cudaMemcpy(h_GpuBatchNums, d_GpuBatchNums,
        sizeof(PRS)*dim*batchSize, cudaMemcpyDeviceToHost);

    // Copy GPU values to print storage
    for (int i = 0; i < batchSize; i++) {
        if (i+printOffset < nPrint) {
for (int j = 0; j < printDim; j++) {
            // We transpose the GPU data for easier display
            h_print[(printOffset+i)*printDim + j] =
                h_GpuBatchNums[i+j*batchSize];
        }
        } // END print storage loop
        printOffset += batchSize;
    } // END batching for loop
}

```

```
// Print Sobol numbers
fprintf(fpout,
        "\nThe first 64 GPU numbers from dimensions 1 to 5:\n");
fprintf(fpout, "      n ");
for(int d = 1; d<=printDim; d++) fprintf(fpout, " dim%-8d", d);
for(int i = 0; i < nPrint; i++) {
    fprintf(fpout, "\n%5d", i+1);
    for(int d = 0; d < printDim; d++)
        fprintf(fpout, " % .6f ", h_print[i*printDim+d]);
}
fprintf(fpout, "\n\n");
```

END:

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

if (d_GpuBatchNums)
    nag_gpu_utilSafeCall( cudaFree(d_GpuBatchNums) );

if (h_GpuBatchNums)
    free(h_GpuBatchNums);

if (v)
    free(v);

if (h_print)
    free(h_print);

fflush(stdout);
fflush(stderr);

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

cudaThreadExit();
return 0;
}
```

6.2 Program Data

None.

6.3 Program Results

NAG GPU Sobol Example Program: uniform

The first 64 GPU numbers from dimensions 1 to 5:

n	dim1	dim2	dim3	dim4	dim5
1	0.000000	0.000000	0.000000	0.000000	0.000000
2	0.500000	0.500000	0.500000	0.500000	0.500000
3	0.750000	0.250000	0.250000	0.250000	0.750000
4	0.250000	0.750000	0.750000	0.750000	0.250000
5	0.375000	0.375000	0.625000	0.875000	0.375000
6	0.875000	0.875000	0.125000	0.375000	0.875000
7	0.625000	0.125000	0.875000	0.625000	0.625000
8	0.125000	0.625000	0.375000	0.125000	0.125000
9	0.187500	0.312500	0.937500	0.437500	0.562500
10	0.687500	0.812500	0.437500	0.937500	0.062500
11	0.937500	0.062500	0.687500	0.187500	0.312500
12	0.437500	0.562500	0.187500	0.687500	0.812500
13	0.312500	0.187500	0.312500	0.562500	0.937500
14	0.812500	0.687500	0.812500	0.062500	0.437500

15	0.562500	0.437500	0.062500	0.812500	0.187500
16	0.062500	0.937500	0.562500	0.312500	0.687500
17	0.093750	0.468750	0.468750	0.656250	0.281250
18	0.593750	0.968750	0.968750	0.156250	0.781250
19	0.843750	0.218750	0.218750	0.906250	0.531250
20	0.343750	0.718750	0.718750	0.406250	0.031250
21	0.468750	0.093750	0.843750	0.281250	0.156250
22	0.968750	0.593750	0.343750	0.781250	0.656250
23	0.718750	0.343750	0.593750	0.031250	0.906250
24	0.218750	0.843750	0.093750	0.531250	0.406250
25	0.156250	0.156250	0.531250	0.843750	0.843750
26	0.656250	0.656250	0.031250	0.343750	0.343750
27	0.906250	0.406250	0.781250	0.593750	0.093750
28	0.406250	0.906250	0.281250	0.093750	0.593750
29	0.281250	0.281250	0.156250	0.218750	0.718750
30	0.781250	0.781250	0.656250	0.718750	0.218750
31	0.531250	0.031250	0.406250	0.468750	0.468750
32	0.031250	0.531250	0.906250	0.968750	0.968750
33	0.046875	0.265625	0.703125	0.546875	0.140625
34	0.546875	0.765625	0.203125	0.046875	0.640625
35	0.796875	0.015625	0.953125	0.796875	0.890625
36	0.296875	0.515625	0.453125	0.296875	0.390625
37	0.421875	0.140625	0.078125	0.421875	0.265625
38	0.921875	0.640625	0.578125	0.921875	0.765625
39	0.671875	0.390625	0.328125	0.171875	0.515625
40	0.171875	0.890625	0.828125	0.671875	0.015625
41	0.234375	0.078125	0.265625	0.984375	0.703125
42	0.734375	0.578125	0.765625	0.484375	0.203125
43	0.984375	0.328125	0.015625	0.734375	0.453125
44	0.484375	0.828125	0.515625	0.234375	0.953125
45	0.359375	0.453125	0.890625	0.109375	0.828125
46	0.859375	0.953125	0.390625	0.609375	0.328125
47	0.609375	0.203125	0.640625	0.359375	0.078125
48	0.109375	0.703125	0.140625	0.859375	0.578125
49	0.078125	0.234375	0.796875	0.140625	0.421875
50	0.578125	0.734375	0.296875	0.640625	0.921875
51	0.828125	0.484375	0.546875	0.390625	0.671875
52	0.328125	0.984375	0.046875	0.890625	0.171875
53	0.453125	0.359375	0.421875	0.765625	0.046875
54	0.953125	0.859375	0.921875	0.265625	0.546875
55	0.703125	0.109375	0.171875	0.515625	0.796875
56	0.203125	0.609375	0.671875	0.015625	0.296875
57	0.140625	0.421875	0.234375	0.328125	0.984375
58	0.640625	0.921875	0.734375	0.828125	0.484375
59	0.890625	0.171875	0.484375	0.078125	0.234375
60	0.390625	0.671875	0.984375	0.578125	0.734375
61	0.265625	0.046875	0.609375	0.703125	0.609375
62	0.765625	0.546875	0.109375	0.203125	0.109375
63	0.515625	0.296875	0.859375	0.953125	0.359375
64	0.015625	0.796875	0.359375	0.453125	0.859375
