NAG Library Function Document

nag_multid_quad_monte_carlo_1 (d01xbc)

1 Purpose

nag_multid_quad_monte_carlo_1 (d01xbc) evaluates an approximation to the integral of a function over a hyper-rectangular region, using a Monte–Carlo method. An approximate relative error estimate is also returned. This function is suitable for low accuracy work.

2 Specification

```
#include <nag.h>
#include <nagd01.h>
void nag_multid_quad_monte_carlo_1 (Integer ndim,
     double  (*f)(Integer ndim, const double x[], Nag_User *comm),
     Nag_MCMethod method, Nag_Start cont, const double a[], const double b[],
     Integer *mincls, Integer maxcls, double eps, double *finest,
     double *acc, double **comm_arr, Nag_User *comm, NagError *fail)
```

3 Description

nag_multid_quad_monte_carlo_1 (d01xbc) uses an adaptive Monte–Carlo method based on the algorithm described by Lautrup (1971). It is implemented for integrals of the form:

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_n}^{b_n} f(x_1,x_2,\ldots,x_n)dx_n\cdots dx_2dx_1.$$

Upon entry, unless the argument **method** = Nag_OneIteration, the function subdivides the integration region into a number of equal volume subregions. Inside each subregion the integral and the variance are estimated by means of pseudorandom sampling. All contributions are added together to produce an estimate for the whole integral and total variance. The variance along each coordinate axis is determined and the function uses this information to increase the density and change the widths of the sub-intervals along each axis, so as to reduce the total variance. The total number of subregions is then increased by a factor of two and the program recycles for another iteration. The program stops when a desired accuracy has been reached or too many integral evaluations are needed for the next cycle.

4 References

Lautrup B (1971) An adaptive multi-dimensional integration procedure *Proc. 2nd Coll. Advanced Methods in Theoretical Physics, Marseille*

5 Arguments

1: **ndim** – Integer *Input*

On entry: the number of dimensions of the integral, n.

Constraint: **ndim** ≥ 1.

2: **f** – function, supplied by the user *External Function*

f must return the value of the integrand f at a given point.

The specification of **f** is:

```
double  f (Integer ndim, const double x[], Nag_User *comm)
```

1: **ndim** – Integer *Input*

On entry: the number of dimensions of the integral.

2: **x**[**ndim**] – const double *Input*

On entry: the coordinates of the point at which the integrand must be evaluated.

3: **comm** – Nag_User *

Pointer to a structure of type Nag_User with the following member:

p – Pointer

On entry/exit: the pointer **comm**→**p** should be cast to the required type, e.g., `struct user *s = (struct user *)comm → p`, to obtain the original object's address with appropriate type. (See the argument **comm** below.)

3: **method** – Nag_MCMethod *Input*

On entry: the method to be used.

method = Nag_OneIteration
The function uses only one iteration of a crude Monte–Carlo method with **maxcls** sample points.

method = Nag_ManyIterations
The function subdivides the integration region into a number of equal volume subregions.

Constraint: **method** = Nag_OneIteration or Nag_ManyIterations.

4: **cont** – Nag_Start *Input*

On entry: the continuation state of the evaluation of the integrand.

cont = Nag_Cold
Indicates that this is the first call to the function with the current integrand and arguments **ndim**, **a** and **b**.

cont = Nag_Hot
Indicates that a previous call has been made with the same arguments **ndim**, **a** and **b** with the same integrand. Please note that **method** must not be changed.

cont = Nag_Warm
Indicates that a previous call has been made with the same arguments **ndim**, **a** and **b** but that the integrand is new. Please note that **method** must not be changed.

Constraint: **cont** = Nag_Cold, Nag_Warm or Nag_Hot.

5: **a**[**ndim**] – const double *Input*

On entry: the lower limits of integration, a_i, for $i = 1, 2, \ldots, n$.

6: **b**[**ndim**] – const double *Input*

On entry: the upper limits of integration, b_i, for $i = 1, 2, \ldots, n$.

7: **mincls** – Integer * *Input/Output*

On entry: **mincls** must be set to the minimum number of integrand evaluations to be allowed.

Constraint: $0 \leq$ **mincls** $<$ **maxcls**.

On exit: **mincls** contains the total number of integrand evaluations actually used by nag_multid_quad_monte_carlo_1 (d01xbc).

8: **maxcls** – Integer *Input*

On entry: the maximum number of integrand evaluations to be allowed. In the continuation case this is the number of new integrand evaluations to be allowed. These counts do not include zero integrand values.

Constraints:

maxcls $>$ **mincls**;
maxcls $\geq 4 \times (\mathbf{ndim} + 1)$.

9: **eps** – double *Input*

On entry: the relative accuracy required.

Constraint: **eps** ≥ 0.0.

10: **finest** – double * *Output*

On exit: the best estimate obtained for the integral.

11: **acc** – double * *Output*

On exit: the estimated relative accuracy of **finest**.

12: **comm_arr** – double ** *Input/Output*

On entry: if **cont** = Nag_Warm or Nag_Hot, the memory pointed to and allocated by a previous call of nag_multid_quad_monte_carlo_1 (d01xbc) must be unchanged.

If **cont** = Nag_Cold then appropriate memory is allocated internally by nag_multid_quad_monte_carlo_1 (d01xbc).

On exit: **comm_arr** contains information about the current sub-interval structure which could be used in later calls of nag_multid_quad_monte_carlo_1 (d01xbc). In particular, **comm_arr**$[j-1]$ gives the number of sub-intervals used along the jth coordinate axis.

When this information is no longer useful, or before a subsequent call to nag_multid_quad_monte_carlo_1 (d01xbc) with **cont** = Nag_Cold is made, you should free the storage contained in this pointer using the NAG macro `NAG_FREE`. Note this memory will have been allocated and needs to be freed only if the error exit NE_NOERROR or NE_QUAD_-MAX_INTEGRAND_EVAL occurs. Otherwise, no memory needs to be freed.

13: **comm** – Nag_User *

Pointer to a structure of type Nag_User with the following member:

p – Pointer

On entry/exit: the pointer **comm**→**p**, of type Pointer, allows you to communicate information to and from **f**(). An object of the required type should be declared, e.g., a structure, and its address assigned to the pointer **comm**→**p** by means of a cast to Pointer in the calling program, e.g., `comm.p = (Pointer)&s`. The type Pointer is `void *`.

14: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_2_INT_ARG_GE

On entry, **mincls** $= \langle value \rangle$ while **maxcls** $= \langle value \rangle$. These arguments must satisfy **mincls** $<$ **maxcls**.

NE_2_INT_ARG_LT

On entry, **maxcls** $= \langle value \rangle$ while **ndim** $= \langle value \rangle$. These arguments must satisfy **maxcls** $\geq 4 \times ($**ndim** $+ 1)$.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument **cont** had an illegal value.

On entry, argument **method** had an illegal value.

NE_INT_ARG_LE

On entry, **mincls** $= \langle value \rangle$.
Constraint: **mincls** > 0.

NE_INT_ARG_LT

On entry, **ndim** $= \langle value \rangle$.
Constraint: **ndim** ≥ 1.

NE_QUAD_MAX_INTEGRAND_EVAL

maxcls was too small to obtain the required accuracy.
In this case nag_multid_quad_monte_carlo_1 (d01xbc) returns a value of **finest** with estimated relative error **acc**, but **acc** will be greater than **eps**. This error exit may be taken before **maxcls** nonzero integrand evaluations have actually occurred, if the function calculates that the current estimates could not be improved before **maxcls** was exceeded.

NE_REAL_ARG_LT

On entry, **eps** must not be less than 0.0: **eps** $= \langle value \rangle$.

7 Accuracy

A relative error estimate is output through the argument **acc**. The confidence factor is set so that the actual error should be less than **acc** 90% of the time. If you desire a higher confidence level then a smaller value of **eps** should be used.

8 Parallelism and Performance

Not applicable.

9 Further Comments

The running time for nag_multid_quad_monte_carlo_1 (d01xbc) will usually be dominated by the time used to evaluate the integrand **f**, so the maximum time that could be used is approximately proportional to **maxcls**.

For some integrands, particularly those that are poorly behaved in a small part of the integration region, this function may terminate with a value of **acc** which is significantly smaller than the actual relative error. This should be suspected if the returned value of **mincls** is small relative to the expected difficulty

of the integral. Where this occurs, nag_multid_quad_monte_carlo_1 (d01xbc) should be called again, but with a higher entry value of **mincls** (e.g., twice the returned value) and the results compared with those from the previous call.

9.1 Additional Information

The exact values of **finest** and **acc** on return will depend (within statistical limits) on the sequence of random numbers generated within this function.

If desired, you may ensure the identity or difference between runs of the results returned by this function by calling nag_random_init_repeatable (g05cbc) or nag_random_init_nonrepeatable (g05ccc) immediately prior to calling this function.

nag_random_init_repeatable (g05cbc) has the prototype

```
void g05cbc(Integer seed)
```

where `seed` is a scalar value used to initialize the underlying random number generator. Using the same value for `seed` will ensure that the same sequence of random values are generated and hence that the same result from this function will be obtained.

nag_random_init_nonrepeatable (g05ccc) has the prototype

```
void g05ccc()
```

Each time nag_random_init_nonrepeatable (g05ccc) is called the underlying random number generator will be reinitialized using a random seed, ensuring a different sequence of values being used. Consequently this function may return different numerical results.

10 Example

This example calculates the integral

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{4x_1x_3^2 \exp(2x_1x_3)}{(1+x_2+x_4)^2} dx_1 dx_2 dx_3 dx_4 = 0.575364.$$

10.1 Program Text

```
/* nag_multid_quad_monte_carlo_1 (d01xbc) Example Program.
 *
 * Copyright 1998 Numerical Algorithms Group.
 *
 * Mark 5, 1998.
 * Mark 6 revised, 2000.
 * Mark 7 revised, 2001.
 * Mark 8 revised, 2004.
 *
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <math.h>
#include <nagd01.h>

#ifdef __cplusplus
extern "C" {
#endif
static double NAG_CALL f(Integer  ndim, const double x[], Nag_User *comm);
#ifdef __cplusplus
}
#endif

#define MAXCLS 20000

int main(void)
{
```

```
  static Integer use_comm[1] = {1};
  Integer      exit_status = 0, k, maxcls = MAXCLS, mincls, ndim = 4;
  NagError     fail;
  Nag_MCMethod method;
  Nag_Start    cont;
  Nag_User     comm;
  double       *a = 0, acc, *b = 0, *comm_arr = 0, eps, finest;

  INIT_FAIL(fail);

  printf(
          "nag_multid_quad_monte_carlo_1 (d01xbc) Example Program Results\n");

  /* For communication with user-supplied functions: */
  comm.p = (Pointer)

  if (ndim >= 1)
    {
      if (!(a = NAG_ALLOC(ndim, double)) ||
          !(b = NAG_ALLOC(ndim, double)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
    }
  else
    {
      printf("Invalid ndim.\n");
      exit_status = 1;
      return exit_status;
    }
  for (k = 0; k < ndim; ++k)
    {
      a[k] = 0.0;
      b[k] = 1.0;
    }
  eps = 0.01;
  mincls = 1000;
  method = Nag_ManyIterations;
  cont = Nag_Cold;

  /* nag_multid_quad_monte_carlo_1 (d01xbc).
   * Multi-dimensional quadrature, using Monte Carlo method,
   * thread-safe
   */
  nag_multid_quad_monte_carlo_1(ndim, f, method, cont, a, b, &mincls, maxcls,
                                eps, &finest, &acc, &comm_arr, &comm, &fail);
  if (fail.code == NE_NOERROR || fail.code == NE_QUAD_MAX_INTEGRAND_EVAL)
    {
      if (fail.code == NE_QUAD_MAX_INTEGRAND_EVAL)
        {
          printf(
                  "Error from nag_multid_quad_monte_carlo_1 (d01xbc).\n%s\n",
                  fail.message);
          exit_status = 2;
        }
      printf("Requested accuracy    = %11.2e\n", eps);
      printf("Estimated value       = %10.5f\n", finest);
      printf("Estimated accuracy    = %11.2e\n", acc);
      printf("Number of evaluations = %5ld\n", mincls);
    }
  else
    {
      printf(
              "Error from nag_multid_quad_monte_carlo_1 (d01xbc).\n%s\n",
              fail.message);
      printf("%s\n", fail.message);
      exit_status = 1;
    }
 END:
```

```
  NAG_FREE(a);
  NAG_FREE(b);
  /* Free memory allocated internally */
  NAG_FREE(comm_arr);
  return exit_status;
}


static double NAG_CALL f(Integer ndim, const double x[], Nag_User *comm)
{
  Integer *use_comm = (Integer *)comm->p;

  if (use_comm[0])
    {
      printf("(User-supplied callback f, first invocation.)\n");
      use_comm[0] = 0;
    }

  return x[0]*4.0*(x[2]*x[2])*exp(x[0]*2.0*x[2])/
         ((x[1]+1.0+x[ndim-1])*(x[1]+1.0+x[ndim-1]));
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_multid_quad_monte_carlo_1 (d01xbc) Example Program Results
(User-supplied callback f, first invocation.)
Requested accuracy    =    1.00e-02
Estimated value       =    0.57554
Estimated accuracy    =    8.20e-03
Number of evaluations =  1728
```