NAG Library Function Document

nag_quad_md_sgq_multi_vec (d01esc)

1 Purpose

nag_quad_md_sgq_multi_vec (d01esc) approximates a vector of definite integrals $\mathbf{F}$ over the unit hypercube $\Omega = [0,1]^d$, given the vector of integrands $\mathbf{f}(\mathbf{x})$.

$$\mathbf{F} = \int_\Omega \mathbf{f}(\mathbf{x}) d\mathbf{x} = \int_0^1 \int_0^1 \dots \int_0^1 \mathbf{f}(x_1, x_2, \dots, x_d) d_{x_1} d_{x_2} \dots d_{x_d}.$$

The function uses a sparse grid discretisation, allowing for computationally feasible estimations of integrals of high dimension ($d \sim O(100)$).

2 Specification

```
#include <nag.h>
#include <nagd01.h>

void nag_quad_md_sgq_multi_vec (Integer ni, Integer ndim,
     void (*f)(Integer ni, Integer ndim, Integer nx, double xtr,
          Integer nntr, const Integer icolzp[], const Integer irowix[],
          const double xs[], const Integer qs[], double fm[], Integer *iflag,
          Nag_Comm *comm),
     const Integer maxdlv[], double dinest[], double errest[],
     Integer ivalid[], const Integer iopts[], const double opts[],
     Nag_Comm *comm, NagError *fail)
```

3 Description

nag_quad_md_sgq_multi_vec (d01esc) uses a sparse grid to generate a vector of approximations $\hat{\mathbf{F}}$ to a vector of integrals $\mathbf{F}$ over the unit hypercube $\Omega = [0,1]^d$, that is,

$$\hat{\mathbf{F}} \approx \mathbf{F} = \int_{[0,1]^d} \mathbf{f}(\mathbf{x}) d\mathbf{x}.$$

3.1 Comparing Quadrature Over Full and Sparse Grids

Before illustrating the sparse grid construction, it is worth comparing integration over a sparse grid to integration over a full grid.

Given a one-dimensional quadrature rule with N abscissae, which accurately evaluates a polynomial of order P_N, a full tensor product over d dimensions, a full grid, may be constructed with N^d multidimensional abscissae. Such a product will accurately integrate a polynomial where the maximum power of any dimension is P_N. For example if $d = 2$ and $P_N = 3$, such a rule will accurately integrate any polynomial whose highest order term is $x_1^3 x_2^3$. Such a polynomial may be said to have a maximum combined order of P_N^d, provided no individual dimension contributes a power greater than P_N. However, the number of multidimensional abscissae, or points, required increases exponentially with the dimension, rapidly making such a construction unusable.

The sparse grid technique was developed by Smolyak (Smolyak (1963)). In this, multiple one-dimensional quadrature rules of increasing accuracy are combined in such a way as to provide a multidimensional quadrature rule which will accurately evaluate the integral of a polynomial whose maximum order appears as a monomial. Hence a sparse grid construction whose highest level quadrature rule has polynomial order P_N will accurately integrate a polynomial whose maximum combined order is also P_N. Again taking $P_N = 3$, one may, theoretically, accurately integrate a polynomial such as $x^3 + x^2y + y^3$, but not a polynomial such as $x^3y^3 + xy$. Whilst this has a lower maximum combined order than the full tensor product, the number of abscissae required increases significantly slower than the equivalent full grid, making some classes of integrals of dimension $d \sim O(100)$ tractable. Specifically, if a one-dimensional quadrature rule of level ℓ has $N \sim O(2^{\ell})$ abscissae, the corresponding full grid will have $O\left(\left(2^{\ell}\right)^{d}\right)$ multidimensional abscissae, whereas the sparse grid will have $O\left(2^{\ell} d^{\ell-1}\right)$. Figure 1 demonstrates this using a Gauss–Patterson rule with 15 points in 3 dimensions. The full grid requires 3375 points, whereas the sparse grid only requires 111.

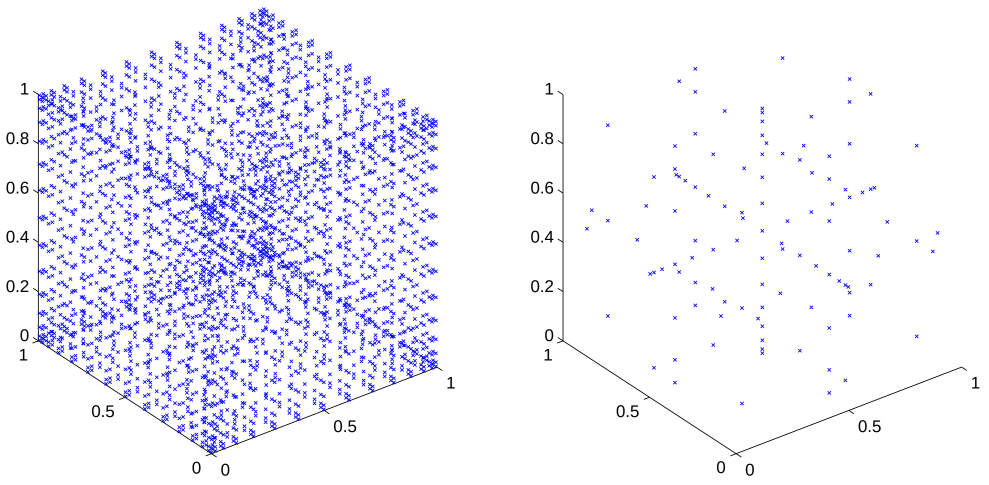

Figure 1
Three-dimensional full (left) and sparse (right) grids, constructed from the 15 point Gauss–Patterson rule

3.2 Sparse Grid Quadrature

We now include a brief description of the sparse grid construction, sufficient for the understanding of the use of this routine. For a more detailed analysis, see Gerstner and Griebel (1998).

Consider a one-dimensional n_{ℓ}-point quadrature rule of level ℓ, Q_{ℓ}. The action of this rule on a integrand f is to approximate its definite one-dimensional integral ${}_1F$ as,

$$ {}_1F = \int_0^1 f(x)dx \approx Q_{\ell}(f) = \sum_{i=1}^{n_{\ell}} w_{\ell,i} \times f\left(x_{\ell,i}\right), $$

using weights $w_{\ell,i}$ and abscissae $x_{\ell,i}$, for $i = 1,2,\ldots,n_{\ell}$.

Now construct a set of one-dimensional quadrature rules, $\{Q_{\ell} \mid \ell = 1,\ldots,L\}$, such that the accuracy of the quadrature rule increases with the level number. In this routine we exclusively use quadrature rules which are completely nested, implying that if an abscissae $x_{\ell,k}$ is in level ℓ, it is also in level $\ell + 1$. The quantity L denotes some maximum level appropriate to the rules that have been selected.

Now define the action of the tensor product of d rules as,

$$ \left(Q_{\ell_1} \otimes \cdots \otimes Q_{\ell_d}\right)(f) = \sum_{i_1=1}^{n_{\ell_1}} \cdots \sum_{i_d=1}^{n_{\ell_d}} w_{\ell_1,i_1} \cdots w_{\ell_d,i_d} f\left(x_{\ell_1,i_1},\ldots,x_{\ell_d,i_d}\right), $$

where the individual level indices ℓ_j are not necessarily ordered or unique. Each tensor product of d rules defines an action of the quadrature rules $Q_{\boldsymbol{\ell}}$, $\boldsymbol{\ell} = (\ell_1,\ell_2,\ldots,\ell_d)$ over a subspace, which is given a level $|\boldsymbol{\ell}| = \sum_{j=1}^{d} \ell_j$. If all rule levels are equal, this is the full tensor product of that level.

The sparse grid construction of level ℓ can then be declared as the sum of all actions of the quadrature differences $\Delta_k = \left(Q_k - Q_{k-1}\right)$, over all subspaces having a level at most $\ell - d + 1$,

$$ {}_dF \approx Q_\ell^d(f) = \sum_{\text{level at most } \ell-d+1} (\Delta_{k_1} \otimes \cdots \otimes \Delta_{k_d})(f). \tag{1}$$

By definition, all subspaces used for level $\ell - 1$ must also be used for level ℓ, and as such the difference between the result of all actions over subsequent sparse grid constructions may be used as an error estimate.

Let L be the maximum level allowable in a sparse grid construction. The classical sparse grid construction of $\ell = L$ allows each dimension to support a one-dimensional quadrature rule of level at most L, with such a quadrature rule being used in every dimension at least once. Such a construction lends equal weight to each dimension of the integration, and is termed here 'isotropic'.

Define the set $\mathbf{m} = (m_j, j = 1, 2, \ldots, d)$, where m_j is the maximum quadrature rule allowed in the jth dimension, and m_q to be the maximum quadrature rule used by any dimension. Let a subspace be identified by its quadrature difference levels, $\mathbf{k} = (k_1, k_2, \ldots, k_d)$.

The classical construction may be extended by allowing different dimensions to have different values m_j, and by allowing $m_q \leq L$. This creates non-isotropic constructions. These are especially useful in higher dimensions, where some dimensions contribute more than others to the result, as they can drastically reduce the number of function evaluations required.

For example, consider the two-dimensional construction with $L = 4$. The classical isotropic construction would have the following subspaces.

Subspaces generated by a classical sparse grid with $L = 4$.

Level	**Subspaces**
1	$(1,1)$
2	$(2,1)$, $(1,2)$
3	$(3,1)$, $(2,2)$, $(1,3)$
4	$(4,1)$, $(3,2)$, $(2,3)$, $(1,4)$

If the variation in the second dimension is sufficiently accurately described by a quadrature rule of level 2, the contributions of the subspaces $(1,3)$ and $(1,4)$ are probably negligible. Similarly, if the variation in the first dimension is sufficiently accurately described by a quadrature rule of level 3, the subspace $(4,1)$ is probably negligible. Furthermore the subspace $(2,3)$ would also probably have negligible impact, whereas the subspaces $(2,2)$ and $(3,2)$ would not. Hence restricting the first dimension to a maximum level of 3, and the second dimension to a maximum level of 2 would probably give a sufficiently acceptable estimate, and would generate the following subspaces.

Subspaces generated by a non-isotropic sparse grid with $L = 4$, $m_q = 3$ and $\mathbf{m} = (3,2)$.

Level	**Subspaces**
1	$(1,1)$
2	$(2,1)$, $(1,2)$
3	$(3,1)$, $(2,2)$
4	$(4,1)$, $(3,2)$

Taking this to the extreme, if the variation in the first and second dimensions are sufficiently accurately described by a level 2 quadrature rule, restricting the maximum level of both dimensions to 2 would generate the following subspaces.

Subspaces generated by a sparse grid construction with $L = 4$, $m_q = 2$ and $\mathbf{m} = (2,2)$.

Level	**Subspaces**
1	$(1,1)$
2	$(2,1)$, $(1,2)$
3	$(2,2)$
4	None

Hence one subspace is generated at level 3, and no subspaces are generated at level 4. The level 3 subspace $(2,2)$ actually indicates that this is the full grid of level 2.

3.3 Using nag_quad_md_sgq_multi_vec (d01esc)

nag_quad_md_sgq_multi_vec (d01esc) uses optional arguments, supplied in the option arrays **iopts** and **opts**. Before calling nag_quad_md_sgq_multi_vec (d01esc), these option arrays must be initialized using nag_quad_opt_set (d01zkc). Once initialized, the required options may be set and queried using nag_quad_opt_set (d01zkc) and nag_quad_opt_get (d01zlc) respectively. A complete list of the options available may be found in Section 11.

You may control the maximum level required, L, using the optional argument **Maximum Level**. Furthermore, you may control the first level at which the error comparison will take place using the optional argument **Minimum Level**, allowing for the forced evaluation of a predetermined number of levels before the routine attempts to complete. Completion is flagged when the error estimate is sufficiently small:

$$\left|\hat{F}_d^k - \hat{F}_d^{k-1}\right| \leq \max\left(\epsilon_a, \epsilon_r \times \hat{F}_d^k\right),$$

where ϵ_a and ϵ_r are the absolute and relative error tolerances, respectively, and $k \leq L$ is the highest level at which computation was performed. The tolerances ϵ_a and ϵ_r can be controlled by setting the optional arguments **Absolute Tolerance** and **Relative Tolerance**.

Owing to the interlacing nature of the quadrature rules used herein, abscissae **x** required in lower level subspaces will also appear in higher-level subspaces. This allows for calculations which will be repeated later to be stored and re-used. However, this is naturally at the expense of memory. It may also be at the expense of computational time, depending on the complexity of the integrands, as the lookup time for a given value is (at worst) $O(d)$. Furthermore, as the sparse grid level increases, fewer subsequent levels will require values from the current level. You may control the number of levels for which values are stored by setting the optional argument **Index Level**.

Two different sets of interlacing quadrature rules are selectable using the optional argument **Quadrature Rule**: Gauss–Patterson and Clenshaw–Curtis. Gauss–Patterson rules offer greater polynomial accuracy, whereas Clenshaw–Curtis rules are often effective for oscillatory integrands. Clenshaw–Curtis rules require function values to be evaluated on the boundary of the hypercube, whereas Gauss–Patterson rules do not. Both of these rules use precomputed weights, and as such there is an effective limit on m_q; see the description of the optional argument **Quadrature Rule**. The value of m_q is returned by the queriable optional argument **Maximum Quadrature Level**.

nag_quad_md_sgq_multi_vec (d01esc) also allows for non-isotropic sparse grids to be constructed. This is done by appropriately setting the array **maxdlv**. It should be emphasised that a non-isometric construction should only be used if the integrands behave in a suitable way. For example, they may decay toward zero as the lesser dimensions approach their bounds of Ω. It should also be noted that setting **maxdlv**$[k-1] = 1$ will not reduce the dimension of the integrals, it will simply indicate that only one point in dimension k should be used. It is also advisable to approximate the integrals several times, once with an isometric construction of some level, and then with a non-isometric construction with higher levels in various dimensions. If the difference between the solutions is significantly more than the returned error estimates, the assumptions of dimensional importance are probably incorrect.

The abscissae in each subspace are generally expressible in a sparse manner, because many of the elements of each abscissa will in fact be the centre point of the dimension, which is termed here the ‘trivial’ element. In this function the trivial element is always returned as 0.5 owing to the restriction to the $[0,1]$ hypercube. As such, the function **f** returns the abscissae in Compressed Column Storage (CCS) format (see the f11 Chapter Introduction). This has particular advantages when using accelerator hardware to evaluate the required functions, as much less data must be forwarded. It also, potentially, allows for calculations to be computed faster, as any sub-calculations dependent upon the trivial value may be potentially re-used. See the example in Section 10.

4 References

Caflisch R E, Morokoff W and Owen A B (1997) Valuation of mortgage backed securities using Brownian bridges to reduce effective dimension *Journal of Computational Finance* **1** 27–46

Gerstner T and Griebel M (1998) Numerical integration using sparse grids *Numerical Algorithms* **18** 209–232

Smolyak S A (1963) Quadrature and interpolation formulas for tensor products of certain classes of functions *Dokl. Akad. Nauk SSSR* **4** 240–243

5 Arguments

1: **ni** – Integer *Input*

On entry: n_i, the number of integrands.

Constraint: **ni** ≥ 1.

2: **ndim** – Integer *Input*

On entry: d, the number of dimensions.

Constraint: **ndim** ≥ 1.

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

f must return the value of the integrands f_j at a set of n_x d-dimensional points $\mathbf{x}_i$, implicitly supplied as columns of a matrix $X(d, n_x)$. If X was supplied explicitly you would find that most of the elements attain the same value, x_{tr}; the larger the number of dimensions, the greater the proportion of elements of X would be equal to x_{tr}. So, X is effectively a sparse matrix, except that the 'zero' elements are replaced by elements that are all equal to the value x_{tr}. For this reason X is supplied, as though it were a sparse matrix, in compressed column storage (CCS) format (see the f11 Chapter Introduction).

Individual entries $x_{k,i}$ of X, for $k = 1, 2, \ldots, d$, are either trivially valued, in which case $x_{k,i} = x_{\text{tr}}$, or are non-trivially valued. For point i, the non-trivial row indices and corresponding abscissae values are supplied in elements $c(i) = \textbf{icolzp}[i-1], \ldots, \textbf{icolzp}[i] - 1$, for $i = 1, 2, \ldots, n_x$, of the arrays **irowix** and **xs**, respectively. Hence the ith column of the matrix X is retrievable as

$$X(\textbf{irowix}[c(i) - 1], i) = \textbf{xs}[c(i) - 1],$$

$$X(k \notin \textbf{irowix}[c(i) - 1], i) = x_{\text{tr}}.$$

An equivalent integer valued matrix Q is also implicitly provided. This contains the unique indices $q_{k,i}$ of the underlying one-dimensional quadrature rule corresponding to the individual abscissae $x_{k,i}$. For trivial abscissae, the implicit index $q_{k,i} = 1$. Q is supplied in the same CCS format as X, with the non-trivial values supplied in **qs**.

Note: the values returned in **icolzp** and **irowix** are one-based.

The specification of **f** is:

```
void f (Integer ni, Integer ndim, Integer nx, double xtr,
     Integer nntr, const Integer icolzp[], const Integer irowix[],
     const double xs[], const Integer qs[], double fm[],
     Integer *iflag, Nag_Comm *comm)
```

1: **ni** – Integer *Input*

On entry: n_i, the number of integrands.

2: **ndim** – Integer *Input*

On entry: d, the number of dimensions.

3: **nx** – Integer *Input*

On entry: n_x, the number of points x_i, corresponding to the number of columns of X, at which the set of integrands must be evaluated.

4: **xtr** – double *Input*

On entry: x_{tr}, the value of the trivial elements of X.

5: **nntr** – Integer *Input*

On entry: if **iflag** > 0, the number of non-trivial elements of X.

If **iflag** $= 0$, the total number of abscissae from the underlying one-dimensional quadrature.

6: **icolzp**[**nx** + **1**] – const Integer *Input*

On entry: the set $\{$**icolzp**$[i-1], \dots,$ **icolzp**$[i] - 1\}$ contains the indices of **irowix** and **xs** corresponding to the non-trivial elements of column i of X and hence of the point $\mathbf{x}_i$, for $i = 1, 2, \dots, n_x$.

7: **irowix**[**nntr**] – const Integer *Input*

On entry: the row indices corresponding to the non-trivial elements of X.

8: **xs**[**nntr**] – const double *Input*

On entry: $x_{k,i} \neq x_{\text{tr}}$, the non-trivial entries of X.

9: **qs**[**nntr**] – const Integer *Input*

On entry: $q_{k,i} \neq 1$, the indices of the underlying quadrature rules corresponding to $x_{k,i} \neq x_{\text{tr}}$.

10: **fm**[**ni** $\times$ **nx**] – double *Output*

On exit: **fm**$[(i-1) \times$ **ni** $+ p - 1] = f_p(\mathbf{x}_i)$, for $i = 1, 2, \dots, n_x$ and $p = 1, 2, \dots, n_i$.

11: **iflag** – Integer * *Input/Output*

On entry: if **iflag** $= 0$, this is the first call to **f**. $n_x = 1$, and the entire point $\mathbf{x}_1$ will satisfy $x_{k,1} = x_{\text{tr}}$, for $k = 1, 2, \dots, d$. In addition, **nntr** contains the total number of abscissae from the underlying one-dimensional quadrature; **xs** contains the complete set of abscissae and **qs** contains the corresponding quadrature indices, with **xs**$[0] = x_{\text{tr}}$ and **qs**$[0] = 1$. This will always be called in serial.

In subsequent calls to **f**, **iflag** $= 1$. Subsequent calls may be made from within an OpenMP parallel region. See Section 8 for details.

On exit: set **iflag** < 0 if you wish to force an immediate exit from nag_quad_md_sgq_multi_vec (d01esc) with **fail.code** = NE_USER_STOP.

12: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **f**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_quad_md_sgq_multi_vec (d01esc) you may allocate memory and initialize these pointers with various quantities for use by **f** when called from nag_quad_md_sgq_multi_vec (d01esc) (see Section 3.2.1.1 in the Essential Introduction).

4: **maxdlv**[**ndim**] – const Integer *Input*

On entry: **m**, the array of maximum levels for each dimension. **maxdlv**$[j-1]$, for $j = 1, 2, \ldots, d$, contains m_j, the maximum level of quadrature rule dimension j will support.

The default value, $\min(m_q, L)$ will be used if either **maxdlv**$[j-1] \leq 0$ or **maxdlv**$[j-1] \geq \min(m_q, L)$ (for details on the default values for m_q and L and on how to change these values see the options **Maximum Level**, **Maximum Quadrature Level** and **Quadrature Rule**).

If **maxdlv**$[j-1] = 1$ for all j, only one evaluation will be performed, and as such no error estimation will be possible.

Suggested value: **maxdlv**$[j-1] = 0$ for all $j = 1, 2, \ldots, d$.

Note: setting non-default values for some dimensions makes the assumption that the contribution from the omitted subspaces is 0. The integral and error estimates will only be based on included subspaces, which if the 0 contribution assumption is not valid will be erroneous.

5: **dinest**[**ni**] – double *Output*

On exit: **dinest**$[p-1]$ contains the final estimate $\hat{F}_p$ of the definite integral F_p, for $p = 1, 2, \ldots, n_i$.

6: **errest**[**ni**] – double *Output*

On exit: **errest**$[p-1]$ contains the final error estimate E_p of the definite integral F_p, for $p = 1, 2, \ldots, n_i$.

7: **ivalid**[**ni**] – Integer *Output*

On exit: **ivalid**$[p-1]$ indicates the final state of integral p, for $p = 1, 2, \ldots, n_i$.

ivalid$[p-1] = 0$
  The error estimate for integral p was below the requested tolerance.

ivalid$[p-1] = 1$
  The error estimate for integral p was below the requested tolerance. The final level used was non-isotropic.

ivalid$[p-1] = 2$
  The error estimate for integral p was above the requested tolerance.

ivalid$[p-1] = 3$
  The error estimate for integral p was above $\max(0.1|\hat{F}_p|, 0.01)$.

ivalid$[p-1] < 0$
  You aborted the evaluation before an error estimate could be made.

8: **iopts**[**100**] – const Integer *Communication Array*
9: **opts**[**100**] – const double *Communication Array*

The arrays **iopts** and **opts** MUST NOT be altered between calls to any of the functions nag_quad_md_sgq_multi_vec (d01esc), nag_quad_opt_set (d01zkc) and nag_quad_opt_get (d01zlc).

10: **comm** – Nag_Comm *

The NAG communication argument (see Section 3.2.1.1 in the Essential Introduction).

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

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

6 Error Indicators and Warnings

NE_ACCURACY

The requested accuracy was not achieved for at least one integral.

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_BAD_PARAM

On entry, argument $\langle value \rangle$ had an illegal value.

NE_INT

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

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

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

NE_INVALID_OPTION

Either the option arrays **iopts** and **opts** have not been initialized for nag_quad_md_sgq_multi_vec (d01esc), or they have become corrupted.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

NE_TOTAL_PRECISION_LOSS

No accuracy was achieved for at least one integral.

NE_USER_STOP

Exit requested from **f** with **iflag** $= \langle value \rangle$.

7 Accuracy

For each integral p, an error estimate E_p is returned, where,

$$E_p = \left| \hat{F}_p^k - \hat{F}_p^{k-1} \right| \approx \left| \hat{F}_p - F_p \right|,$$

where $k \leq L$ is the highest level at which computation was performed.

8 Parallelism and Performance

8.1 Direct Threading

nag_quad_md_sgq_multi_vec (d01esc) is directly threaded for parallel execution. For each level, at most n_t threads will evaluate the integrands over independent subspaces of the construction, and will construct a partial sum of the level's contribution. Once all subspaces from a given level have been processed, the partial sums are combined to give the total contribution of the level, which is in turn added to the total solution. For a given number of threads, the division of subspaces between the threads, and the order in which a single thread operates over its assigned subspaces, is fixed. However, the order in which all subspaces are combined will necessarily be different to the single threaded case, which may result in some discrepency in the results between parallel and serial execution.

To mitigate this discrepency, it is recommended that nag_quad_md_sgq_multi_vec (d01esc) be instructed to use higher-than-double precision to accumulate the actions over the subspaces. This is done by setting the option **Summation Precision** $=$ HIGHER, which is the default behaviour. This has some computational cost, although this is typically negligible in comparison to the overall runtime, particularly for non-trivial integrands.

If **Summation Precision** $=$ WORKING, then the accumulation will be performed using double precision, which may provide some increase in performance. Again, this is probably negligible in comparison to the overall runtime.

For some problems, typically of lower dimension, there may be insufficient work to warrant direct threading at lower levels. For example, a three-dimensional problem will require at most 3 subspaces to be evaluated at level 2, and at most 6 subspaces at level 3. Furthermore, level 2 subspaces typically contain only 2 new multidimensional abscissae, while level 3 subspaces typically contain 2 or 4 new multidimensional abscissae depending on the **Quadrature Rule**. If there are more threads than the number of available subspaces at a given level, or the amount of work in each subspace is outweighed by the amount of work required to generate the parallel region, parallel efficiency will be decreased. This may be mitigated to some extent by evaluating the first s_l levels in serial. The value of s_l may be altered using the optional argument **Serial Levels**. If $s_l \geq L$, then all levels will be evaluated in serial and no direct threading will be utilized.

If you use direct threading in the manner just described, you must ensure any access to the communication structure **comm** is done in a thread-safe manner. This is classed as OpenMP SHARED, and is passed directly to the function **f** for every thread.

8.2 Parallelization of f

The vectorized interface also allows for parallelization inside the function **f** by evaluating the required integrands in parallel. Provided the values returned by **f** match those that would be returned without parallelizing **f**, the final result should match the serial result, barring any discrepencies in accumulation. If you wish to parallelize **f**, it is advisable to set a large value for **Maximum Nx**, although be aware that increasing **Maximum Nx** will increase the memory requirement of the function. In general, parallelization of **f** should not be necessary, as the higher-level parallelism over different subspaces scales well for many problems.

9 Further Comments

Not applicable.

10 Example

The example program evaluates an estimate to the set of integrals

$$\mathbf{F} = \int_{\Omega} \begin{pmatrix} \sin(1+|\mathbf{x}|) \\ \vdots \\ \sin(n_i+|\mathbf{x}|) \end{pmatrix} \log|\mathbf{x}| d\mathbf{x}$$

where $|\mathbf{x}| = \sum_{j=1}^{d} jx_j$. It also demonstrates a simple method to safely use **comm** as workspace for sub-calculations when running in parallel.

10.1 Program Text

```
/* nag_quad_md_sgq_multi_vec (d01esc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 25, 2014.
 */

#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagd01.h>
#include <nagx06.h>

#ifdef __cplusplus
extern "C" {
#endif
  static void NAG_CALL f(Integer ni,Integer ndim, Integer nx, double xtr,
                         Integer nntr, const Integer *icolzp,
                         const Integer *irowix, const double *xs,
                         const Integer *qs, double *fm,
                         Integer *iflag, Nag_Comm *comm);
#ifdef __cplusplus
}
#endif

/* We define some structures to serve as a demonstration of safely operating
 * with the NAG communication structure comm when running in parallel.
 */

/* par_sh: any information to be shared between threads in the function f. */
typedef struct {
  double s_tr;
} par_sh;

/* par_pr: any private workspace that a single thread will require in the
 * execution of the function f.
 */
typedef struct {
  double *s;
  double *logs;
} par_pr;

/* parallel_comm: a container for par_sh and par_pr. */
typedef struct {
  par_sh shared;
  par_pr *private;
} parallel_comm;

int main(void) {
  Integer     exit_status = 0;
  Integer     ndim, ni;
  Integer     maxnx, smpthd, lcvalue;
```

```
double      rvalue;
char        cvalue[16];
Integer     *ivalid, *iopts, *maxdlv;
double      *dinest, *errest, *opts;
parallel_comm parcom;
int         i, thdnum;
/* Nag Types */
Nag_VariableType optype;
Nag_Comm    comm;
NagError    fail;

INIT_FAIL(fail);

printf("nag_quad_md_sgq_multi_vec (d01esc) Example Program Results\n");

ni = 10;
ndim = 4;
lcvalue = 16;
if (!(iopts = NAG_ALLOC(100, Integer)) ||
    !(opts = NAG_ALLOC(100, double)) ||
    !(maxdlv = NAG_ALLOC(ndim, Integer)) ||
    !(dinest = NAG_ALLOC(ni, double)) ||
    !(errest = NAG_ALLOC(ni, double)) ||
    !(ivalid = NAG_ALLOC(ni, Integer) )) {
  printf("Allocation failure\n");
  exit_status = -1;
  goto END;
}

/* Initialize option arrays. */
nag_quad_opt_set("Initialize = d01esc",iopts,100,opts,100,&fail);
if (fail.code != NE_NOERROR) {
  printf("Error from nag_quad_opt_set (d01zkc).\n%s\n",fail.message);
  exit_status = 1;
  goto END;
}

/* Set any required options. */
nag_quad_opt_set("Absolute Tolerance = 0.0",iopts,100,opts,100,&fail);
if (fail.code != NE_NOERROR) {
  printf("Error from nag_quad_opt_set (d01zkc).\n%s\n",fail.message);
  exit_status = 1;
  goto END;
}
nag_quad_opt_set("Relative Tolerance = 1.0e-3",iopts,100,opts,100,&fail);
if (fail.code != NE_NOERROR) {
  printf("Error from nag_quad_opt_set (d01zkc).\n%s\n",fail.message);
  exit_status = 1;
  goto END;
}
nag_quad_opt_set("Maximum Level = 6",iopts,100,opts,100,&fail);
if (fail.code != NE_NOERROR) {
  printf("Error from nag_quad_opt_set (d01zkc).\n%s\n",fail.message);
  exit_status = 1;
  goto END;
}
nag_quad_opt_set("Index Level = 5",iopts,100,opts,100,&fail);
if (fail.code != NE_NOERROR) {
  printf("Error from nag_quad_opt_set (d01zkc).\n%s\n",fail.message);
  exit_status = 1;
  goto END;
}

/* Set any required maximum dimension levels. */
for (i = 0; i < ndim; ++i) maxdlv[i] = 0;

/* As a demonstration of safely operating with the communication structure
 * comm when running in parallel, we will create an instance of our
 * parallel_comm structure with fields indexed by the current thread number.
 * The size of the array subfields in this structure are a function of
 * Maximum Nx.
```

```
   */
  nag_quad_opt_get("Maximum Nx",&maxnx,&rvalue,cvalue,lcvalue,&optype,iopts,
                   opts,&fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_quad_opt_get (d01zlc).\n%s\n",fail.message);
    exit_status = 1;
    goto END;
  }

  smpthd = nag_omp_get_max_threads();

  /* Allocate an array of smpthd pointers to private structures. */
  if (!(parcom.private = NAG_ALLOC(smpthd, par_pr))) {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

  /* For each thread, allocate a double array of size maxnx in the
   * private component of the parallel structure.
   */
  for (thdnum = 0; thdnum<smpthd; thdnum++) {
    if (!(parcom.private[thdnum].s = NAG_ALLOC(maxnx, double)) ||
        !(parcom.private[thdnum].logs = NAG_ALLOC(maxnx, double))) {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
  }

  /* Finally, store the parallel structure in comm for communication to f. */
  comm.p = &parcom;

  /* Approximate the integrals. */
  nag_quad_md_sgq_multi_vec(ni,ndim,f,maxdlv,dinest,errest,ivalid,iopts,opts,
                            &comm,&fail);
  if (fail.code != NE_NOERROR && fail.code != NE_ACCURACY &&
      fail.code != NE_TOTAL_PRECISION_LOSS && fail.code != NE_USER_STOP)
    {
      /* If internal memory allocation failed consider reducing the options
       * 'Maximum Nx' and 'Index Level', or run with fewer threads.
       */
      printf("Error from nag_quad_md_sgq_multi_vec (d01esc).\n%s\n",
             fail.message);
      exit_status = 2;
      goto END;
    }

  /* NE_NOERROR:
   *   The result returned satisfies the requested accuracy requirements.
   * NE_ACCURACY, NE_TOTAL_PRECISION_LOSS:
   *   The result returned is inaccurate for at least one integral.
   * NE_USER_STOP:
   *   Exit was requested by setting iflag negative in f.
   *   A result will be returned if at least one call to f was successful.
   */
  printf("Integral # | Estimated value | Error estimate |"
         " Final state of integral\n");
  for (i = 0; i < ni; ++i)
    printf("%11d|%17.5e|%16.5e|%8"NAG_IFMT"\n",
           i, dinest[i], errest[i], ivalid[i]);

 END:
  NAG_FREE(maxdlv);
  NAG_FREE(dinest);
  NAG_FREE(errest);
  NAG_FREE(ivalid);
  NAG_FREE(iopts);
  NAG_FREE(opts);
  for (thdnum = 0; thdnum < smpthd; ++thdnum) {
    NAG_FREE(parcom.private[thdnum].s);
    NAG_FREE(parcom.private[thdnum].logs);
  }
  NAG_FREE(parcom.private);
  return exit_status;
}

static void NAG_CALL f(Integer ni, Integer ndim, Integer nx, double xtr,
                       Integer nntr, const Integer *icolzp,
                       const Integer *irowix, const double *xs,
                       const Integer *qs, double *fm,
                       Integer *iflag, Nag_Comm *comm)
{
  Integer i, j, k, tid;
  double s_tr, s_ntr;
  double *s, *logs;
  parallel_comm *parcom;

  /* For each evaluation point x_i, i = 1, ..., nx, return in fm the computed
   * values of the ni integrals f_j, j = 1, ..., ni defined by
   *
   *   fm(j,i) = f_j(x_i)
   *                                                       ndim
   *           = sin(j + S(i))*log(S(i)), where S(i) =    Σ   k*x_i(k).
   *                                                       k=1
   *
   * Split the S expression into two components, one involving only the
   * 'trivial' value xtr:
   *
   *          ndim             ndim
   *   S(i) =   Σ   (k*xtr)  +   Σ (k*(x_i(k)-xtr))
   *           k=1              k=1
   *
   *                ndim*(ndim+1)   ndim
   *        = xtr * ------------- +   Σ (k*(x_i(k)-xtr))
   *                      2          k=1
   *
   *       := s_tr                + s_ntr(i)
   *
   * By definition the summands in the s_ntr(i) term on the right-hand side
   * are zero for those k outside the range of indices defined in irowix.
   */

  /* As a demonstration of safely operating with the communication structure
   * comm when running in parallel, the p field of comm is itself (a pointer
   * to) an instance of our parallel_comm structure 'partitioned' by the current
   * thread number. Store some of the s_tr and s_ntr computations in these.
   */

  /* The thread number. */
  tid = nag_omp_get_thread_num();

  /* The S and log(S) terms from above, extracted from comm. */
  parcom = comm->p;
  s = (*parcom).private[tid].s;
  logs = (*parcom).private[tid].logs;

  if (*iflag == 0) {
    /* First call: nx=1, no non-trivial dimensions.
     * The constant s_tr can be reused by all subsequent calculations.
     */
    s_tr = 0.5*xtr*((double)(ndim*(ndim+1)));
    parcom->shared.s_tr = s_tr;
    s[0] = s_tr;
    logs[0] = log(s_tr);
  } else {
    /* Calculate S(i) = s_tr + s_ntr(i). */
    s_tr = parcom->shared.s_tr;
    for (i = 0; i < nx; ++i) {
      s_ntr = 0.0;
      for (k = icolzp[i]; k < icolzp[i+1]; ++k)
 s_ntr = s_ntr + ((double)irowix[k-1])*(xs[k-1]-xtr);
```

```
      s[i] = s_tr + s_ntr;
      logs[i] = log(s[i]);
    }
  }
  /* Finally we obtain fm(j,:) = sin(j+S(:))*log(S(:)). */
  for (j = 0; j < ni; ++j)
    for (i = 0; i < nx; ++i)
      fm[i*ni + j] = sin(((double)j+1) + s[i])*logs[i];
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_quad_md_sgq_multi_vec (d01esc) Example Program Results
Integral # | Estimated value | Error estimate | Final state of integral
          0|      3.83522e-02|     2.39770e-05|       0
          1|      4.01177e-01|     1.69503e-05|       0
          2|      3.95161e-01|     5.66045e-06|       0
          3|      2.58363e-02|     2.30670e-05|       0
          4|     -3.67242e-01|     1.92659e-05|       0
          5|     -4.22680e-01|     2.24822e-06|       0
          6|     -8.95077e-02|     2.16953e-05|       0
          7|      3.25958e-01|     2.11959e-05|       0
          8|      4.41739e-01|     1.20901e-06|       0
          9|      1.51388e-01|     1.98894e-05|       0
```

11 Optional Arguments

Several optional arguments in nag_quad_md_sgq_multi_vec (d01esc) control aspects of the algorithm, methodology used, logic or output. Their values are contained in the arrays **iopts** and **opts**; these must be initialized before calling nag_quad_md_sgq_multi_vec (d01esc) by first calling nag_quad_opt_set (d01zkc) with **optstr** set to `"Initialize = d01esc"`.

Each optional argument has an associated default value; to set any of them to a non-default value, or to reset any of them to the default value, use nag_quad_opt_set (d01zkc). The current value of an optional argument can be queried using nag_quad_opt_get (d01zlc).

The remainder of this section can be skipped if you wish to use the default values for all optional arguments.

The following is a list of the optional arguments available. A full description of each optional argument is provided in Section 11.1.

Absolute Tolerance

Index Level

Maximum Level

Maximum Nx

Maximum Quadrature Level

Minimum Level

Quadrature Rule

Relative Tolerance

Serial Levels

Summation Precision

11.1 Description of the Optional Arguments

For each option, we give a summary line, a description of the optional argument and details of constraints.

The summary line contains:

the keywords, where the minimum abbreviation of each keyword is underlined;

a parameter value, where the letters a, i and r denote options that take character, integer and real values respectively.

the default value.

The following symbol represent various machine constants:

ϵ represents the ***machine precision*** (see nag_machine_precision (X02AJC)).

All options accept the value 'DEFAULT' in order to return single options to their default states.

Keywords and character values are case insensitive, however they must be separated by at least one space.

Queriable options will return the appropriate value when queried by calling nag_quad_opt_get (d01zlc). They will have no effect if passed to nag_quad_opt_set (d01zkc).

For nag_quad_md_sgq_multi_vec (d01esc) the maximum length of the argument **cvalue** used by nag_quad_opt_get (d01zlc) is 15.

Absolute Tolerance r Default $= \sqrt{\epsilon}$

$r = \epsilon_a$, the absolute tolerance required.

Index Level i Default $= 4$

The maximum level at which function values are stored for later use. Larger values use increasingly more memory, and require more time to access specific elements. Lower levels result in more repeated computation.

Constraint: $i \geq 1$.

Maximum Level i Default $= 5$

$i = L$, the maximum number of levels to evaluate.

Constraint: $1 < i \leq 20$.

Note: the maximum allowable level in any single dimension, m_q, is governed by the **Quadrature Rule** selected. If a value greater than m_q is set, only a subset of subspaces from higher levels will be used. Should this subset be empty for a given level, computation will consider the preceding level to be the maximum level and will terminate.

Maximum Nx i Default $= 128$

$i = \max n_x$, the maximum number of points to evaluate in a single call to **f**.

Constraint: $1 \leq i \leq 16384$.

Maximum Quadrature Level i Queriable only

$i = m_q$, the maximum level of the underlying one-dimensional quadrature rule (see **Quadrature Rule**).

Minimum Level i Default $= 2$

The minimum number of levels which must be evaluated before an error estimate is used to determine convergence.

Constraint: $i > 1$.

Note: if the minimum level is greater than the maximum computable level, the maximum level will be used.

Quadrature Rule a Default $=$ Gauss–Patterson

The underlying one-dimensional quadrature rule to be used in the construction. Open rules do not require evaluations at boundaries.

Quadrature Rule $=$ Gauss–Patterson or GP
The interlacing Gauss–Patterson rules. Level ℓ uses $2^{\ell} - 1$ abscissae. All levels are open. These rules provide high order accuracy. $m_q = 9$.

Quadrature Rule $=$ Clenshaw–Curtis or CC
The interlacing Clenshaw–Curtis rules. Level ℓ uses $2^{\ell-1} + 1$ abscissae. All levels above level 1 are closed. $m_q = 12$.

Relative Tolerance r Default $= \sqrt{\epsilon}$

$r = \epsilon_a$, the relative tolerance required.

Summation Precision a Default $=$ HIGHER

Determines whether nag_quad_md_sgq_multi_vec (d01esc) uses double precision or higher-than-double precision to accumulate the actions over subspaces.

Summation Precision $=$ HIGHER or H
Higher-than-double precision is used to accumulate the action over a subspace, and for the accumulation of all such actions. This is more expensive computationally, although this is probably negligible in comparison to the cost of evaluating the integrands and the overall runtime. This significantly reduces variation in the result when changing the number of threads.

Summation Precision $=$ WORKING or W
Double precision is used to accumulate the actions over subspaces. This may provide some speedup, particularly if n_i or n_t is large. The results of parallel simulations will however be more prone to variation.

Note: the following option is relevant only to multithreaded implementations of the NAG Library..

Serial Levels i Default $= 1$

$i = s_l$, the number of levels to be evaluated in serial before initializing parallelization. For relatively trivial integrands, this may need to be set greater than the default to reduce parallel overhead.