Module 11.1: nag_quad_1d
Numerical Integration over a Finite Interval

`nag_quad_1d` provides procedures for computing the value of a one-dimensional definite integral over a *finite* interval.

Contents

Introduction

The procedures in this module are designed to estimate the value of a one-dimensional definite integral of the form:

$$\int_a^b f(x)\,dx,$$

where the limits of integration a and b are finite, and $f(x)$ is defined by the user, either in the form of a function, or at a set of distinct points x_i, for $i = 1, 2, \ldots, n$.

The module also provides procedures for integrands of the form

$$f(x) = w(x)g(x),$$

which contain a factor $w(x)$, called the *weight function*, of a specific form. These procedures take full account of any peculiar behaviour attributable to the $w(x)$ factor. For further details see the Mathematical Background section of this module document.

Procedure: nag_quad_1d_gen

1 Description

nag_quad_1d_gen is a general-purpose integrator which computes an approximation to the integral

$$I = \int_a^b f(x)\,dx$$

of a function $f(x)$ over a finite interval $[a, b]$. The procedure is suitable for integrands which have:

- singularities, especially when these are of algebraic or logarithmic type;
- local singular behaviour at a known finite number of points within the integration interval;
- oscillatory behaviour, but are non-singular.

It is an adaptive procedure, offering a choice of six Gauss–Kronrod rules. See Section 6.1 for more details.

This procedure requires a vector valued user-supplied function to evaluate the integrand at an array of points, and will therefore be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

2 Usage

```
USE nag_quad_1d
CALL nag_quad_1d_gen(f, a, b, result  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 1$ — the number of break-points

3.1 Mandatory Arguments

f — function

f must return the values of the integrand f at a set of points.

```
function f(x)

real(kind=wp), intent(in) ::  x(:)
```

Input: the points at which the integrand f must be evaluated.

```
real(kind=wp) ::  f(SIZE(x))
```

Result: f(i) must contain the value of f at x(i), for $i = 1, 2, \ldots,$SIZE(x).

a — real(kind=*wp*), intent(in)

b — real(kind=*wp*), intent(in)

Input: the lower and upper limits of integral I respectively.

Note: it is not necessary that `a` < `b`.

result — real(kind=*wp*), intent(out)

Output: the approximation to the integral I.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

abs_acc — real(kind=*wp*), intent(in), optional

Input: the absolute accuracy required.

Default: `abs_acc` = `SQRT(EPSILON(1.0_wp))`.

Constraints: `abs_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

rel_acc — real(kind=*wp*), intent(in), optional

Input: the relative accuracy required.

Default: `rel_acc` = 10^{-4}.

Constraints: `rel_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

abs_err — real(kind=*wp*), intent(out), optional

Output: an estimate of the modulus of the absolute error, which should be an upper bound for $| I - $ `result` $|$.

rule — integer, intent(in), optional

Input: the integration rule to be used:

if `rule` = 1 for the Gauss 7-point and Kronrod 15-point rule;
if `rule` = 2 for the Gauss 10-point and Kronrod 21-point rule;
if `rule` = 3 for the Gauss 15-point and Kronrod 31-point rule;
if `rule` = 4 for the Gauss 20-point and Kronrod 41-point rule;
if `rule` = 5 for the Gauss 25-point and Kronrod 51-point rule;
if `rule` = 6 for the Gauss 30-point and Kronrod 61-point rule.

Default: `rule` = 2.

Recommended Value: for an oscillatory, but non-singular integrand, `rule` = 6.

Constraints: `rule` = 1, 2, 3, 4, 5, or 6.

brk_pts(n) — real(kind=*wp*), intent(in), optional

Input: the user-specified break-points.

Constraints: $n \geq 1$. The break-points must all lie within the interval of integration (but may be supplied in any order).

max_num_subint — integer, intent(in), optional

Input: the maximum number of subintervals to be used in the subdivision strategy.

Default: `max_num_subint` = max(500,2×(`SIZE(brk_pts)`+1)).

Constraints: `max_num_subint` ≥ 1.

num_subint_used — integer, intent(out), optional

Output: the final number of subintervals used in the subdivision strategy.

num_fun_eval — integer, intent(out), optional

Output: the actual number of integrand evaluations.

subint_info(:,:) — real(kind=*wp*), pointer, optional

Output: details of the computation which may be examined in the event of a failure. This array contains the end-points of the subinterval used by the procedure, along with the integral contribution and error estimates over these subintervals. Specifically, for $i = 1, 2, \ldots, m$, let r_i denote the approximation to the value of the integral over the subinterval $[x_i, x_{i+1}]$ in the partition of $[a, b]$, and let e_i be the corresponding absolute error estimate. Then

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \simeq r_i \quad \text{and} \quad \texttt{result} = \sum_{i=1}^{m} r_i,$$

unless this procedure terminates while testing for divergence of the integral. In this case `result` (and `abs_err`) are taken to be the values returned from the extrapolation process. The value of m is returned in `num_subint_used`, and the values of x_i, r_i and e_i are stored in the array `subint_info`, that is:

x_i = `subint_info`$(i, 1)$, $i \neq m + 1$, $x_{m+1} = b$,
r_i = `subint_info`$(i, 2)$,
e_i = `subint_info`$(i, 3)$.

Note: this array is allocated by `nag_quad_1d_gen`. It should be deallocated when no longer required.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Maximum number of subdivisions allowed has been reached. The accuracy requirements have not been achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g., a singularity of the integrand or its derivative, a peak, a discontinuity, etc.) you will probably gain from splitting up the interval at this point and calling the integrator on the subintervals, or supplying this point to the procedure as an element of the optional argument `brk_pts`. If necessary, another integrator should be used, which is designed for handling the type of difficulty involved. Alternatively, consider relaxing the accuracy requirements specified by the optional arguments `abs_acc` and `rel_acc`, or increasing the value of the optional argument `max_num_subint`.

202 Round-off error prevents the requested accuracy from being achieved.

The error may be under-estimated. Consider requesting less accuracy.

203 Extremely bad local integrand behaviour.

This causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case `error%code` = 201.

204 The requested accuracy cannot be achieved.

The extrapolation does not increase the accuracy satisfactorily; the result returned is the best which can be obtained. The same advice applies as in the case `error%code` = 201.

205 The integral is probably divergent, or slowly convergent.

Note that divergence can also occur with any other value of `error%code` when `error%level` = 2.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

6 Further Comments

6.1 Algorithmic Detail

This procedure is a modified version of the QUADPACK procedure QAGP (Piessens *et al.* [6]). It is an adaptive procedure, offering a choice of six Gauss–Kronrod rules. The algorithm described by De Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [3]) together with the ϵ-algorithm (Wynn [9]) to perform extrapolation. The local error estimation is described in Piessens *et al.* [6].

This procedure can cope with singularities of several types. It is very reliable, particularly where the integrand has singularities other than at an end-point, or has discontinuities or cusps, and it is therefore recommended where the integrand is known to be badly behaved, or where its nature is completely unknown.

If $f(x)$ is known to be free of singularities, though it may be oscillatory, this procedure may be used with `rule` = 6. If $f(x)$ has singularities of certain types, discontinuities or sharp peaks *occurring at known points*, the integral should be evaluated separately over each of the subintervals between these points. Alternatively, the points may be specified in the optional argument `brk_pts`.

6.2 Accuracy

The procedure cannot guarantee, but in practice usually achieves, the following accuracy:

$$| I - \texttt{result} | \leq tol,$$

where

$$tol = \max(\texttt{abs\_acc}, \texttt{rel\_acc} \times | I |)$$

and `abs_acc` and `rel_acc` are the specified absolute and relative accuracies, or their default values. The optional output argument `abs_err` normally satisfies

$$| I - \texttt{result} | \leq \texttt{abs\_err} \leq tol.$$

Procedure: nag_quad_1d_wt_trig

1 Description

nag_quad_1d_wt_trig computes an approximation to the integral

$$I = \int_a^b w(x)g(x)\,dx,$$

over a finite interval $[a, b]$, where $w(x) = \sin(\omega x)$ or $\cos(\omega x)$, for a user-specified ω.

It is an adaptive procedure, and uses a modified Clenshaw–Curtis integration of orders 12 and 24 and Gauss 7-point and Kronrod 15-point rules. A description of the algorithm is given in Section 6.1.

This procedure requires a vector valued user-supplied function to evaluate g at an array of points, and will therefore be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

2 Usage

```
USE nag_quad_1d
CALL nag_quad_1d_wt_trig(g, a, b, omega, trig_wt, result  [, optional arguments])
```

3 Arguments

3.1 Mandatory Arguments

g — function

g must return the values of the function g at a set of points.

```
function g(x)

real(kind=wp), intent(in) ::  x(:)
```
Input: the points at which g must be evaluated.

```
real(kind=wp) ::  g(SIZE(x))
```
Result: g(i) must contain the value of g at x(i), for $i = 1, 2, \ldots,$SIZE(x).

a — real(kind=*wp*), intent(in)

b — real(kind=*wp*), intent(in)

Input: the lower and upper limits of integral I respectively.

Note: it is not necessary that a < b.

omega — real(kind=*wp*), intent(in)

Input: the parameter ω in the weight function $w(x)$.

trig_wt — character(len=1), intent(in)

Input: indicates which type of weight function is to be used:

if trig_wt = 'c' or 'C', $w(x) = \cos(\omega x)$;
if trig_wt = 's' or 'S', $w(x) = \sin(\omega x)$.

Constraints: trig_wt = 'c', 'C', 's' or 'S'.

result — real(kind=*wp*), intent(out)

Output: the approximation to the integral I.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

abs_acc — real(kind=*wp*), intent(in), optional

Input: the absolute accuracy required.

Default: `abs_acc` = `SQRT(EPSILON(1.0_wp))`.

Constraints: `abs_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

rel_acc — real(kind=*wp*), intent(in), optional

Input: the relative accuracy required.

Default: `rel_acc` $= 10^{-4}$.

Constraints: `rel_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

abs_err — real(kind=*wp*), intent(out), optional

Output: an estimate of the modulus of the absolute error, which should be an upper bound for $| I -$ `result` $|$.

max_num_subint — integer, intent(in), optional

Input: the maximum number of subintervals to be used in the subdivision strategy.

Default: `max_num_subint` = 500.

Constraints: `max_num_subint` ≥ 2.

num_subint_used — integer, intent(out), optional

Output: the final number of subintervals used in the subdivision strategy.

num_fun_eval — integer, intent(out), optional

Output: the actual number of integrand evaluations.

subint_info(:,:) — real(kind=*wp*), pointer, optional

Output: details of the computation which may be examined in the event of a failure. This array contains the end-points of the subinterval used by the procedure, along with the integral contribution and error estimates over these subintervals. Specifically, for $i = 1, 2, \ldots, m$, let r_i denote the approximation to the value of the integral over the subinterval $[x_i, x_{i+1}]$ in the partition of $[a, b]$, and let e_i be the corresponding absolute error estimate. Then

$$\int_{x_i}^{x_{i+1}} w(x)g(x)\,dx \simeq r_i \quad \text{and} \quad \texttt{result} = \sum_{i=1}^{m} r_i,$$

unless this procedure terminates while testing for divergence of the integral. In this case `result` (and `abs_err`) are taken to be the values returned from the extrapolation process. The value of m is returned in `num_subint_used`, and the values of x_i, r_i and e_i are stored in the array `subint_info`, that is:

$x_i =$ `subint_info`$(i, 1)$, $i \neq m + 1$, $x_{m+1} = b$,

$r_i =$ `subint_info`$(i, 2)$,

$e_i =$ `subint_info`$(i, 3)$.

Note: this array is allocated by `nag_quad_1d_wt_trig`. It should be deallocated when no longer required.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Maximum number of subdivisions allowed has been reached. The accuracy requirements have not been achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g., a singularity of the integrand or its derivative, a peak, a discontinuity, etc.) you will probably gain from splitting up the interval at this point and calling the integrator on the subintervals. If necessary, another integrator should be used, which is designed for handling the type of difficulty involved. Alternatively, consider relaxing the accuracy requirements specified by the optional arguments abs_acc and rel_acc, or increasing the value of the optional argument max_num_subint.
202	Round-off error prevents the requested accuracy from being achieved. The error may be under-estimated. Consider requesting less accuracy.
203	Extremely bad local integrand behaviour. This causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case error%code = 201.
204	The requested accuracy cannot be achieved. The extrapolation does not increase the accuracy satisfactorily; the result returned is the best which can be obtained. The same advice applies as in the case error%code = 201.
205	The integral is probably divergent, or slowly convergent. Note that divergence can also occur with any other value of error%code when error%level = 2.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The procedure is a modified version of the QUADPACK procedure QFOUR (Piessens *et al.* [6]). It is an adaptive procedure, designed to integrate a function of the form $w(x)g(x)$, where $w(x)$ is either $\sin(\omega x)$ or $\cos(\omega x)$.

Consider a subinterval of length

$$L = \mid b - a \mid 2^{-l}.$$

The integration over this subinterval is performed by means of a modified Clenshaw–Curtis procedure (Piessens and Branders [5]), if $L\omega > 4$ and $l \leq 20$. In this case a Chebyshev series approximation of degree 24 is used to approximate $g(x)$, while an error estimate is computed from this approximation together with that obtained using a Chebyshev series of degree 12. If the above conditions do not hold the Gauss 7-point and Kronrod 15-point rules are used. The algorithm described in Piessens *et al.* [6], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [3]) together with the ϵ-algorithm (Wynn [9]) to perform extrapolation. The local error estimation is described in Piessens *et al.* [6].

6.2 Accuracy

The procedure cannot guarantee, but in practice usually achieves, the following accuracy:

$$\mid I - \texttt{result} \mid \leq tol,$$

where

$$tol = \max(\texttt{abs\_acc},\ \texttt{rel\_acc} \times \mid I \mid)$$

and `abs_acc` and `rel_acc` are the specified absolute and relative accuracies, or their default values. The optional output argument `abs_err` normally satisfies

$$\mid I - \texttt{result} \mid \leq \texttt{abs\_err} \leq tol.$$

Procedure: nag_quad_1d_wt_end_sing

1 Description

nag_quad_1d_wt_end_sing computes an approximation to the integral

$$I = \int_a^b w(x)g(x)\,dx,$$

over a finite interval $[a, b]$. The weight $w(x)$ has end-point algebraico-logarithmic singularities of the form:

$$w(x) = \begin{cases} (x-a)^\alpha (x-b)^\beta, & \text{or} \\ (x-a)^\alpha (x-b)^\beta \ln(x-a), & \text{or} \\ (x-a)^\alpha (x-b)^\beta \ln(b-x), & \text{or} \\ (x-a)^\alpha (x-b)^\beta \ln(x-a)\ln(b-x), & \end{cases}$$

where $\alpha > -1$, $\beta > -1$.

It is an adaptive procedure, and uses a modified Clenshaw–Curtis integration of orders 12 and 24 and Gauss 7-point and Kronrod 15-point rules. A description of the algorithm is given in Section 6.1.

This procedure requires a vector valued user-supplied function to evaluate g at an array of points, and will therefore be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

2 Usage

```
USE nag_quad_1d

CALL nag_quad_1d_wt_end_sing(g, a, b, alpha, beta, log_wt, result &
                             [, optional arguments])
```

3 Arguments

3.1 Mandatory Arguments

g — function

g must return the values of the function g at a set of points.

```
function g(x)

real(kind=wp), intent(in) ::  x(:)
```
Input: the points at which g must be evaluated.

```
real(kind=wp) ::  g(SIZE(x))
```
Result: g(i) must contain the value of g at x(i), for $i = 1, 2, \ldots,$SIZE(x).

a — real(kind=*wp*), intent(in)

b — real(kind=*wp*), intent(in)

Input: the lower and upper limits of integral I respectively.

Constraints: a < b.

alpha — real(kind=*wp*), intent(in)

beta — real(kind=*wp*), intent(in)

Input: the parameters α and β in the weight function.

Constraints: `alpha` > -1, `beta` > -1.

log_wt — character(len=1), intent(in)

Input: indicates which type of weight function is to be used:

if `log_wt` = `'n'` or `'N'`, $w(x) = (x-a)^{\alpha}(x-b)^{\beta}$;
if `log_wt` = `'l'` or `'L'`, $w(x) = (x-a)^{\alpha}(x-b)^{\beta}\ln(x-a)$;
if `log_wt` = `'u'` or `'U'`, $w(x) = (x-a)^{\alpha}(x-b)^{\beta}\ln(b-x)$;
if `log_wt` = `'b'` or `'B'`, $w(x) = (x-a)^{\alpha}(x-b)^{\beta}\ln(x-a)\ln(b-x)$.

Constraints: `log_wt` = `'n'`, `'N'`, `'l'`, `'L'`, `'u'`, `'U'`, `'b'` or `'B'`.

result — real(kind=*wp*), intent(out)

Output: the approximation to the integral I.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

abs_acc — real(kind=*wp*), intent(in), optional

Input: the absolute accuracy required.

Default: `abs_acc` = `SQRT(EPSILON(1.0_wp))`.

Constraints: `abs_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

rel_acc — real(kind=*wp*), intent(in), optional

Input: the relative accuracy required.

Default: `rel_acc` $= 10^{-4}$.

Constraints: `rel_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

abs_err — real(kind=*wp*), intent(out), optional

Output: an estimate of the modulus of the absolute error, which should be an upper bound for $|\ I -$ `result` $|$.

max_num_subint — integer, intent(in), optional

Input: the maximum number of subintervals to be used in the subdivision strategy.

Default: `max_num_subint` = 500.

Constraints: `max_num_subint` ≥ 2.

num_subint_used — integer, intent(out), optional

Output: the final number of subintervals used in the subdivision strategy.

num_fun_eval — integer, intent(out), optional

Output: the actual number of integrand evaluations.

subint_info(:,:) — real(kind=*wp*), pointer, optional

Output: details of the computation which may be examined in the event of a failure. This array contains the end-points of the subinterval used by the procedure, along with the integral contribution and error estimates over these subintervals. Specifically, for $i = 1, 2, \ldots, m$, let r_i denote the approximation to the value of the integral over the subinterval $[x_i, x_{i+1}]$ in the partition of $[a, b]$, and let e_i be the corresponding absolute error estimate. Then

$$\int_{x_i}^{x_{i+1}} w(x)g(x)\,dx \simeq r_i \quad \text{and} \quad \texttt{result} = \sum_{i=1}^{m} r_i,$$

unless this procedure terminates while testing for divergence of the integral. In this case `result` (and `abs_err`) are taken to be the values returned from the extrapolation process. The value of m is returned in `num_subint_used`, and the values of x_i, r_i and e_i are stored in the array `subint_info`, that is:

$x_i =$ `subint_info`$(i, 1)$, $i \neq m+1$, $x_{m+1} = b$,
$r_i =$ `subint_info`$(i, 2)$,
$e_i =$ `subint_info`$(i, 3)$.

Note: this array is allocated by `nag_quad_1d_wt_end_sing`. It should be deallocated when no longer required.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Maximum number of subdivisions allowed has been reached. The accuracy requirements have not been achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g., a singularity of the integrand or its derivative, a peak, a discontinuity, etc.) you will probably gain from splitting up the interval at this point and calling the integrator on the subintervals. If necessary, another integrator should be used, which is designed for handling the type of difficulty involved. Alternatively, consider relaxing the accuracy requirements specified by the optional arguments `abs_acc` and `rel_acc`, or increasing the value of the optional argument `max_num_subint`.
202	Round-off error prevents the requested accuracy from being achieved. The error may be under-estimated. Consider requesting less accuracy.
203	Extremely bad local integrand behaviour. This causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case `error%code` = 201.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 3 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The procedure is a modified version of the QUADPACK procedure QAWSE (Piessens *et al.* [6]). It is an adaptive procedure, designed to integrate a function of the form $w(x)g(x)$, where $w(x)$ may have algebraico-logarithmic singularities at the end points a and/or b. We start by bisecting the original interval and applying modified Clenshaw–Curtis integration of orders 12 and 24 to both halves. Clenshaw–Curtis integration is then used on all subintervals which have a or b as one of their end-points (Piessens *et al.* [7]). On the other subintervals the Gauss 7-point and Kronrod 15-point rules are used. A global acceptance criterion (as defined by Malcolm and Simpson [3]) is used. The local error estimation is described in Piessens *et al.* [6].

6.2 Accuracy

The procedure cannot guarantee, but in practice usually achieves, the following accuracy:

$$| I - \texttt{result} | \leq tol,$$

where

$$tol = \max(\texttt{abs\_acc},\ \texttt{rel\_acc} \times | I |)$$

and `abs_acc` and `rel_acc` are the specified absolute and relative accuracies, or their default values. The optional output argument `abs_err` normally satisfies

$$| I - \texttt{result} | \leq \texttt{abs\_err} \leq tol.$$

Procedure: nag_quad_1d_wt_hilb

1 Description

nag_quad_1d_wt_hilb computes an approximation to the Hibert transform of a function $g(x)$:

$$I = \int_a^b \frac{g(x)}{x-c}\,dx,$$

over a finite interval $[a, b]$ where $c \neq a$ and $c \neq b$.

It is an adaptive procedure, and uses a modified Clenshaw–Curtis integration of orders 12 and 24 and Gauss 7-point and Kronrod 15-point rules. A description of the algorithm is given in Section 6.1.

This procedure requires a vector valued user-supplied function to evaluate g at an array of points, and will therefore be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

2 Usage

```
USE nag_quad_1d
CALL nag_quad_1d_wt_hilb(g, a, b, c, result  [, optional arguments])
```

3 Arguments

3.1 Mandatory Arguments

g — function

g must return the values of the function g at a set of points.

```
function g(x)

real(kind=wp), intent(in) ::  x(:)
```
Input: the points at which g must be evaluated.

```
real(kind=wp) ::  g(SIZE(x))
```
Result: g(i) must contain the value of g at x(i), for $i = 1, 2, \ldots$,SIZE(x).

a — real(kind=*wp*), intent(in)

b — real(kind=*wp*), intent(in)

Input: the lower and upper limits of integral I respectively.

Note: it is not necessary that a < b.

c — real(kind=*wp*), intent(in)

Input: the parameter c in the weight function.

Constraints: c ≠ a and c ≠ b.

result — real(kind=*wp*), intent(out)

Output: the approximation to the integral I.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

abs_acc — real(kind=*wp*), intent(in), optional

Input: the absolute accuracy required.

Default: `abs_acc` = `SQRT(EPSILON(1.0_wp))`.

Constraints: `abs_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

rel_acc — real(kind=*wp*), intent(in), optional

Input: the relative accuracy required.

Default: `rel_acc` $= 10^{-4}$.

Constraints: `rel_acc` ≥ 0.0. Both `abs_acc` and `rel_acc` cannot be zero.

abs_err — real(kind=*wp*), intent(out), optional

Output: an estimate of the modulus of the absolute error, which should be an upper bound for $| I - $ `result` $|$.

max_num_subint — integer, intent(in), optional

Input: the maximum number of subintervals to be used in the subdivision strategy.

Default: `max_num_subint` $= 500$.

Constraints: `max_num_subint` ≥ 2.

num_subint_used — integer, intent(out), optional

Output: the final number of subintervals used in the subdivision strategy.

num_fun_eval — integer, intent(out), optional

Output: the actual number of integrand evaluations.

subint_info(:,:) — real(kind=*wp*), pointer, optional

Output: details of computation which may be examined in the event of a failure. This array contains the end-points of the subinterval used by the procedure along with the integral contribution and error estimates over these subintervals. Specifically, for $i = 1, 2, \ldots, m$, let r_i denote the approximation to the value of the integral over the subinterval $[x_i, x_{i+1}]$ in the partition of $[a, b]$, and let e_i be the corresponding absolute error estimate. Then

$$\int_{x_i}^{x_{i+1}} \frac{g(x)}{x-c}\,dx \simeq r_i \quad \text{and} \quad \texttt{result} = \sum_{i=1}^{m} r_i,$$

unless this procedure terminates while testing for divergence of the integral. In this case `result` (and `abs_err`) are taken to be the values returned from the extrapolation process. The value of m is returned in `num_subint_used`, and the values of x_i, r_i and e_i are stored in the array `subint_info`, that is:

x_i = `subint_info`$(i, 1)$, $i \neq m+1$, $x_{m+1} = b$,

r_i = `subint_info`$(i, 2)$,

e_i = `subint_info`$(i, 3)$.

Note: this array is allocated by `nag_quad_1d_wt_hilb`. It should be deallocated when no longer required.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Maximum number of subdivisions allowed has been reached. The accuracy requirements have not been achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g., a singularity of the integrand or its derivative, a peak, a discontinuity, etc.) you will probably gain from splitting up the interval at this point and calling the integrator on the subintervals. If necessary, another integrator should be used, which is designed for handling the type of difficulty involved. Alternatively, consider relaxing the accuracy requirements specified by the optional arguments abs_acc and rel_acc, or increasing the value of the optional argument max_num_subint.
202	Round-off error prevents the requested accuracy from being achieved. The error may be under-estimated. Consider requesting less accuracy.
203	Extremely bad local integrand behaviour. This causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case error%code = 201.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 4 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The procedure is a modified version of the QUADPACK procedure QAWS (Piessens *et al.* [6]). It is an adaptive procedure, designed to integrate a function of the form $w(x)g(x)$, where the weight function $w(x) = 1/(x-c)$ is that of Hilbert transform. (If c is in the interval (a, b), the integral is to be interpreted in the sense of a Cauchy principal value.) A global acceptance criterion (as defined by Malcolm and Simpson [3]) is used. Special care is taken to ensure that c is never the end-point of a subinterval (Piessens *et al.* [8]). On each subinterval (c_1, c_2) modified Clenshaw–Curtis integration of orders 12 and 24 is performed if $c_1 - d \leq c \leq c_2 + d$ where $d = (c_2 - c_1)/20$. Otherwise the Gauss 7-point and Kronrod 15-point rules are used. The local error estimation is described in Piessens *et al.* [6].

6.2 Accuracy

The procedure cannot guarantee, but in practice usually achieves, the following accuracy:

$$| I - \texttt{result} | \leq tol,$$

where

$$tol = \max(\texttt{abs\_acc},\ \texttt{rel\_acc} \times | I |)$$

and `abs_acc` and `rel_acc` are the specified absolute and relative accuracies, or their default values. The optional output argument `abs_err` normally satisfies

$$| I - \texttt{result} | \leq \texttt{abs\_err} \leq tol.$$

Procedure: nag_quad_1d_data

1 Description

nag_quad_1d_data evaluates the definite integral

$$I = \int_{x_1}^{x_n} y(x)\,dx,$$

where the function $y(x)$ is specified at the n distinct points $x_1, x_2, \ldots, x_n$, $n \geq 4$, which are either in ascending or descending order. The method is due to Gill and Miller [2]. See Section 6.1 for more details.

2 Usage

```
USE nag_quad_1d
CALL nag_quad_1d_data(x, y, result  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 4$ — the number of data points

3.1 Mandatory Arguments

x(n) — real(kind=*wp*), intent(in)

Input: the values $x_1, x_2, \ldots, x_n$ of the independent variable.

Constraints: either x(1) $<$ x(2) $<\cdots<$ x(n), or x(1) $>$ x(2) $>\cdots>$ x(n).

y(n) — real(kind=*wp*), intent(in)

Input: the values y_i of the dependent variable at the points x_i, for $i = 1, 2, \ldots, n$.

result — real(kind=*wp*), intent(out)

Output: an estimate of the integral.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

err_estimate — real(kind=*wp*), intent(out), optional

Output: an estimate of the uncertainty in result.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 5 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The integral between successive points is calculated by the four-point finite-difference formula centred on the interval concerned, except in the case of the first and last intervals, where four-point forward and backward difference formulae respectively are employed. An approximation to the truncation error is integrated and added to the result. It is also returned separately to give an estimate of the uncertainty in the result. The method is due to Gill and Miller [2].
In their paper, Gill and Miller [2] do not add the quantity `err_estimate` to `result`. However, extensive tests have shown that a dramatic reduction in the error often results from such addition. In other cases, it does not make an improvement, but these tend to be cases of low accuracy in which the modified answer is not significantly inferior to the unmodified one. You have the option of recovering the result of Gill and Miller [2] by subtracting `err_estimate` from `result` on return from the procedure.

In order to check results independently, and so as to provide an alternative technique, you may fit the data with a cubic spline using `nag_spline_1d_lsq_fit` and then evaluate its integral using `nag_spline_1d_intg`; both these procedures are provided in the module `nag_spline_1d` (8.2).

6.2 Accuracy

The procedure does not allow you to specify an accuracy level, but on return | `err_estimate` | is an approximation to, but not necessarily a bound for, | I − `result` |. If on exit `error%code` has a non-zero value, both `result` and `err_estimate` are returned as zero.

Example 1: Singular integral using general integrator

The integral

$$\int_0^{2\pi} \frac{x\sin(30x)}{\sqrt{1-\frac{x^2}{4\pi^2}}}\,dx$$

is computed using nag_quad_1d_gen.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE quad_1d_ex01_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  REAL (wp) :: pi

CONTAINS

  FUNCTION f(x)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIN, SIZE, SQRT
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Function Return Value ..
    REAL (wp) :: f(SIZE(x))
    ! .. Executable Statements ..

    f = x*SIN(30.0_wp*x)/SQRT(1.0_wp-x*x/(4.0_wp*pi*pi))

  END FUNCTION f

END MODULE quad_1d_ex01_mod

PROGRAM nag_quad_1d_ex01

  ! Example Program Text for nag_quad_1d
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_math_constants, ONLY : nag_pi
  USE nag_quad_1d, ONLY : nag_quad_1d_gen
  USE quad_1d_ex01_mod, ONLY : wp, f, pi
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Local Scalars ..
  REAL (wp) :: a, b, result
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_quad_1d_ex01'
```

```
      pi = nag_pi(0.0_wp)
      a = 0.0_wp
      b = 2.0_wp*pi

      CALL nag_quad_1d_gen(f,a,b,result)

      WRITE (nag_std_out,'(/,1X,A,F10.4)') &
       'a  - lower limit of integration = ', a
      WRITE (nag_std_out,'(1X,A,F10.4)') &
       'b  - upper limit of integration = ', b
      WRITE (nag_std_out,'(1X,A,F9.5)') &
       'result - approximation to the integral =', result

   END PROGRAM nag_quad_1d_ex01
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_quad_1d_ex01

 a  - lower limit of integration =     0.0000
 b  - upper limit of integration =     6.2832
 result - approximation to the integral = -2.54326
```

Example 2: Integrand with oscillatory weight function

The integral

$$\int_0^1 \ln(x)\sin(10\pi x)\,dx$$

is computed using `nag_quad_1d_wt_trig`.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE quad_1d_ex02_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  FUNCTION g(x)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC LOG, SIZE
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Function Return Value ..
    REAL (wp) :: g(SIZE(x))
    ! .. Executable Statements ..

    WHERE (x>0.0_wp)
      g = LOG(x)
    ELSEWHERE
      g = 0.0_wp
    END WHERE

  END FUNCTION g

END MODULE quad_1d_ex02_mod

PROGRAM nag_quad_1d_ex02

  ! Example Program Text for nag_quad_1d
  ! NAG f190, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_math_constants, ONLY : nag_pi
  USE nag_quad_1d, ONLY : nag_quad_1d_wt_trig
  USE quad_1d_ex02_mod, ONLY : wp, g
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Local Scalars ..
  REAL (wp) :: a, b, omega, pi, result
  CHARACTER (1) :: trig_wt
  ! .. Executable Statements ..
```

```
      WRITE (nag_std_out,*) 'Example Program Results for nag_quad_1d_ex02'

      pi = nag_pi(0.0_wp)
      a = 0.0_wp
      b = 1.0_wp
      omega = 10.0_wp*pi
      trig_wt = 'sine'

      CALL nag_quad_1d_wt_trig(g,a,b,omega,trig_wt,result)

      WRITE (nag_std_out,'(/,1X,A,F10.4)') &
       'a  - lower limit of integration = ', a
      WRITE (nag_std_out,'(1X,A,F10.4)') &
       'b  - upper limit of integration = ', b
      WRITE (nag_std_out,'(1X,A,F9.5)') &
       'result - approximation to the integral =', result

    END PROGRAM nag_quad_1d_ex02
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_quad_1d_ex02

 a  - lower limit of integration =     0.0000
 b  - upper limit of integration =     1.0000
 result - approximation to the integral = -0.12814
```

Example 3: Integrands with algebraico-logarithmic singularities

The integrals

$$\int_0^1 \ln(x)\cos(10\pi x)\,dx \quad \text{and} \quad \int_0^1 \frac{\sin(10x)}{\sqrt{x(1-x)}}\,dx$$

are computed using `nag_quad_1d_wt_end_sing`.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE quad_1d_ex03_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  REAL (wp) :: pi

CONTAINS

  FUNCTION g1(x)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC COS, SIZE
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Function Return Value ..
    REAL (wp) :: g1(SIZE(x))
    ! .. Executable Statements ..

    g1 = COS(10.0_wp*pi*x)

  END FUNCTION g1

  FUNCTION g2(x)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIN, SIZE
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Function Return Value ..
    REAL (wp) :: g2(SIZE(x))
    ! .. Executable Statements ..

    g2 = SIN(10.0_wp*x)

  END FUNCTION g2

END MODULE quad_1d_ex03_mod
```

```
PROGRAM nag_quad_1d_ex03

  ! Example Program Text for nag_quad_1d
  ! NAG f190, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_math_constants, ONLY : nag_pi
  USE nag_quad_1d, ONLY : nag_quad_1d_wt_end_sing
  USE quad_1d_ex03_mod, ONLY : wp, g1, g2, pi
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Local Scalars ..
  REAL (wp) :: a, alpha1, alpha2, b, beta1, beta2, result1, result2
  CHARACTER (1) :: log_wt
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_quad_1d_ex03'

  pi = nag_pi(0.0_wp)
  a = 0.0_wp
  b = 1.0_wp
  alpha1 = 0.0_wp
  beta1 = 0.0_wp
  log_wt = 'lower'

  WRITE (nag_std_out,'(/,1X,A,F10.4)') &
   'a  - lower limit of integration = ', a
  WRITE (nag_std_out,'(1X,A,F10.4)') &
   'b  - upper limit of integration = ', b

  CALL nag_quad_1d_wt_end_sing(g1,a,b,alpha1,beta1,log_wt,result1)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,'(1X,A,F10.4)') &
   'alpha1  - parameter in the first weight function = ', alpha1
  WRITE (nag_std_out,'(1X,A,F10.4)') &
   'beta1  - parameter in the first weight function  = ', beta1
  WRITE (nag_std_out,'(1X,A,F9.5)') &
   'result1 - approximation to the first integral =', result1

  alpha2 = -0.5_wp
  beta2 = -0.5_wp
  log_wt = 'none'

  CALL nag_quad_1d_wt_end_sing(g2,a,b,alpha2,beta2,log_wt,result2)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,'(1X,A,F10.4)') &
   'alpha2  - parameter in the second weight function = ', alpha2
  WRITE (nag_std_out,'(1X,A,F10.4)') &
   'beta2  - parameter in the second weight function  = ', beta2
  WRITE (nag_std_out,'(1X,A,F9.5)') &
   'result2 - approximation to the second integral =', result2

END PROGRAM nag_quad_1d_ex03
```

2 Program Data

None.

3 Program Results

```
Example Program Results for nag_quad_1d_ex03

a  - lower limit of integration =     0.0000
b  - upper limit of integration =     1.0000

alpha1  - parameter in the first weight function =     0.0000
beta1  - parameter in the first weight function  =     0.0000
result1 - approximation to the first integral = -0.04899

alpha2  - parameter in the second weight function =    -0.5000
beta2  - parameter in the second weight function  =    -0.5000
result2 - approximation to the second integral =  0.53502
```

Example 4: Hilbert transform

The integral

$$\int_{-1}^{1} \frac{1}{(x^2+0.01^2)(x-0.5)}\,dx$$

is computed using `nag_quad_1d_wt_hilb`.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE quad_1d_ex04_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  FUNCTION g(x)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIZE
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Function Return Value ..
    REAL (wp) :: g(SIZE(x))
    ! .. Executable Statements ..

    g = 1.0_wp/(x*x+0.01_wp*0.01_wp)

  END FUNCTION g

END MODULE quad_1d_ex04_mod

PROGRAM nag_quad_1d_ex04

  ! Example Program Text for nag_quad_1d
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_quad_1d, ONLY : nag_quad_1d_wt_hilb
  USE quad_1d_ex04_mod, ONLY : wp, g
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Local Scalars ..
  REAL (wp) :: a, b, c, result
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_quad_1d_ex04'

  a = -1.0_wp
  b = 1.0_wp
```

```
      c = 0.5_wp

      CALL nag_quad_1d_wt_hilb(g,a,b,c,result)

      WRITE (nag_std_out,'(/,1X,A,F10.4)') &
       'a  - lower limit of integration = ', a
      WRITE (nag_std_out,'(1X,A,F10.4)') &
       'b  - upper limit of integration = ', b
      WRITE (nag_std_out,'(1X,A,F10.4)') &
       'c  - parameter in the weight function = ', c
      WRITE (nag_std_out,'(1X,A,F12.3)') &
       'result - approximation to the integral =', result

    END PROGRAM nag_quad_1d_ex04
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_quad_1d_ex04

 a  - lower limit of integration =    -1.0000
 b  - upper limit of integration =     1.0000
 c  - parameter in the weight function =     0.5000
 result - approximation to the integral =    -628.462
```

Example 5: Integration of a function defined by data values

The integral

$$\int_0^1 \frac{1}{1+x^2}\,dx = \pi$$

is computed using `nag_quad_1d_data` by reading in the function values at 21 unequally spaced points.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
    PROGRAM nag_quad_1d_ex05

      ! Example Program Text for nag_quad_1d
      ! NAG fl90, Release 3. NAG Copyright 1997.

      ! .. Use Statements ..
      USE nag_examples_io, ONLY : nag_std_in, nag_std_out
      USE nag_quad_1d, ONLY : nag_quad_1d_data
      ! .. Implicit None Statement ..
      IMPLICIT NONE
      ! .. Intrinsic Functions ..
      INTRINSIC KIND
      ! .. Parameters ..
      INTEGER, PARAMETER :: wp = KIND(1.0D0)
      ! .. Local Scalars ..
      INTEGER :: i, n
      REAL (wp) :: result
      ! .. Local Arrays ..
      REAL (wp), ALLOCATABLE :: x(:), y(:)
      ! .. Executable Statements ..

      WRITE (nag_std_out,*) 'Example Program Results for nag_quad_1d_ex05'

      READ (nag_std_in,*)          ! Skip heading in data file
      READ (nag_std_in,*) n

      ALLOCATE (x(n),y(n))         ! Allocate storage

      READ (nag_std_in,*) (x(i),y(i),i=1,n)

      CALL nag_quad_1d_data(x,y,result)

      WRITE (nag_std_out,'(/,1X,A,F9.5)') &
       'result - approximation to the integral =', result

      DEALLOCATE (x,y)             ! Deallocate storage

    END PROGRAM nag_quad_1d_ex05
```

2 Program Data

```
Example Program Data for nag_quad_1d_ex05
   21
   0.00   4.0000
   0.04   3.9936
   0.08   3.9746
   0.12   3.9432
   0.22   3.8153
```

```
  0.26   3.7467
  0.30   3.6697
  0.38   3.4943
  0.39   3.4719
  0.42   3.4002
  0.45   3.3264
  0.46   3.3014
  0.60   2.9412
  0.68   2.7352
  0.72   2.6344
  0.73   2.6094
  0.83   2.3684
  0.85   2.3222
  0.88   2.2543
  0.90   2.2099
  1.00   2.0000
```

3 Program Results

```
Example Program Results for nag_quad_1d_ex05

result - approximation to the integral =  3.14141
```

Additional Examples

Not all example programs supplied with NAG *fl*90 appear in full in this module document. The following additional examples, associated with this module, are available.

`nag_quad_1d_ex06`

Oscillatory smooth integrand using general integrator.

`nag_quad_1d_ex07`

Integrand at known point of singularity using general integrator.

`nag_quad_1d_ex08`

Singular integral using general integrator with user specified accuracy.

`nag_quad_1d_ex09`

Integrand with oscillatory weight function with specified accuracy.

Mathematical Background

1 Quadrature

To estimate the value of a one-dimensional integral, a quadrature rule uses an approximation in the form of a weighted sum of integrand values, i.e.,

$$\int_a^b f(x)\,dx \simeq \sum_{i=1}^{N} w_i f(x_i), \tag{1}$$

where x_i and w_i are the abscissae and the weights of the quadrature rule, respectively.

More generally, if the integrand has the form $f(x) = w(x)g(x)$, the corresponding formula becomes

$$\int_a^b w(x)g(x)\,dx \simeq \sum_{i=1}^{N} w_i g(x_i). \tag{2}$$

If the integrand is known only at a fixed set of points, these points must be used as the abscissae, and the weighted sum is calculated using finite-difference methods. However, if the functional form of the integrand is known, so that its value can easily be evaluated at any abscissae, then a wide variety of quadrature rules are available, each characterised by its choice of abscissae and the corresponding weights.

The choice of an appropriate rule depends on the interval $[a, b]$ and the form of any $w(x)$ factor. A suitable value of N depends on the behaviour of $f(x)$ (or of $g(x)$, if there is a $w(x)$ factor present).

Among possible rules, we mention particularly the Gaussian formulae, which employ a distribution of abscissae which is optimal for $f(x)$ or $g(x)$ of polynomial form.

Generally, quadrature algorithms are divided into two categories: *automatic* and *non-automatic*. In a non-automatic algorithm, a fixed number of abscissae, N, is used. This number and the particular rule chosen uniquely determine the weights and abscissae. No estimate is made of the accuracy of the result. However, in an automatic algorithm, the number of abscissae, N, within $[a, b]$ is gradually increased until consistency is achieved to within a level of accuracy (absolute or relative) requested by the user. There are essentially two ways of doing this, *non-adaptive* or *adaptive*; hybrid forms of these two methods are also possible.

1.1 Non-adaptive Algorithms

A series of rules using increasing values of N are successively applied over the whole interval $[a, b]$. It is clearly more economical if abscissae already used for a lower value of N can be used again as part of a higher-order formula. This principle is known as *optimal extension*. There is no overlap between the abscissae used in Gaussian formulae of different orders. However, the Kronrod formulae are designed to give an optimal $(2N + 1)$-point formula by adding $(N + 1)$ points to an N-point Gauss formula.

1.2 Adaptive Algorithms

The interval $[a, b]$ is repeatedly divided into a number of subintervals, and integration rules are applied separately to each subinterval. Typically, the subdivision process will be carried further in the neighbourhood of a sharp peak in the integrand, than where the curve is smooth. Thus, the distribution of abscissae is adapted to the shape of the integrand.

Subdivision raises the problem of what constitutes an acceptable accuracy in each subinterval. The usual *global acceptability criterion* demands that the sum of the absolute values of the error estimates in the subintervals should meet the conditions required of the error over the whole interval. Automatic extrapolation over several levels of subdivision may eliminate the effects of some types of singularities.

An ideal general-purpose method would be an automatic method which could be used for a wide variety of integrands, was efficient (i.e., required the use of as few abscissae as possible), and was reliable (i.e., always gave results within the requested accuracy). Complete reliability is unobtainable, and generally

higher reliability is obtained at the expense of efficiency, and vice versa. *It must therefore be emphasised that the automatic procedures in this module cannot be assumed to be 100% reliable*. In general, however, the reliability is very high.

References

[1] De Doncker E (1978) An adaptive extrapolation algorithm for automatic integration *SIGNUM Newsl.* **13 (2)** 12–18

[2] Gill P E and Miller G F (1972) An algorithm for the integration of unequally spaced data *Comput. J.* **15** 80–83

[3] Malcolm M A and Simpson R B (1976) Local versus global strategies for adaptive quadrature *ACM Trans. Math. Software* **1** 129–146

[4] Piessens R (1973) An algorithm for automatic integration *Angew. Inf.* **15** 399–401

[5] Piessens R and Branders M (1975) Algorithm 002. Computation of oscillating integrals *J. Comput. Appl. Math.* **1** 153–164

[6] Piessens R, De Doncker-Kapenga E, Überhuber C and Kahaner D (1983) *QUADPACK, A Subroutine Package for Automatic Integration* Springer-Verlag

[7] Piessens R, Mertens I and Branders M (1974) Integration of functions having end-point singularities *Angew. Inf.* **16** 65–68

[8] Piessens R, Van Roy-Branders M and Mertens I (1976) The automatic evaluation of Cauchy principal value integrals *Angew. Inf.* **18** 31–35

[9] Wynn P (1956) On a device for computing the $e_m(S_n)$ transformation *Math. Tables Aids Comput.* **10** 91–96