Module 13.2: nag_pde_ell_mg

Multigrid Solution of Elliptic Partial Differential Equations

nag_pde_ell_mg provides procedures for generation and multigrid solution of seven-diagonal systems of linear equations which arise from discretization of two-dimensional elliptic partial differential equations.

Contents

Introduction

This module contains procedures for discretization and solving elliptic partial differential equations on rectangular regions using a standard seven-point finite difference discretization and a multigrid solver.

`nag_pde_ell_rect` discretizes a second order linear elliptic partial differential equation of the form

$$\alpha(x,y)\frac{\partial^2 U}{\partial x^2}+\beta(x,y)\frac{\partial^2 U}{\partial x\partial y}+\gamma(x,y)\frac{\partial^2 U}{\partial y^2}+\delta(x,y)\frac{\partial U}{\partial x}+\epsilon(x,y)\frac{\partial U}{\partial y}+\phi(x,y)U=\psi(x,y) \tag{1}$$

on a rectangular region

$$x_A \le x \le x_B, \quad y_A \le y \le y_B$$

subject to boundary conditions of the form

$$a(x,y)U+b(x,y)\frac{\partial U}{\partial n}=c(x,y),$$

where $\dfrac{\partial U}{\partial n}$ denotes the outward pointing normal derivative on the boundary. Equation (1) is said to be *elliptic* if

$$4\alpha(x,y)\gamma(x,y) \ge \beta^2(x,y)$$

for all points in the rectangular region. The linear equations produced are in a form suitable for passing directly to the multigrid procedure `nag_pde_ell_mg_sol`.

The equation is discretized on a rectangular grid, with n_x grid points in the x-direction and n_y grid points in the y-direction. The grid spacing used is therefore

$$h_x=(x_B-x_A)/(n_x-1), \quad h_y=(y_B-y_A)/(n_y-1)$$

and the coordinates of the grid points (x_i,y_j) are

$$x_i=x_A+(i-1)h_x, \quad y_j=y_A+(j-1)h_y, \quad i=1,2,\ldots,n_x, \quad j=1,2,\ldots,n_y$$

at each grid point (x_i,y_j) six neighbouring grid points are used to approximate the partial differential equation, so that the equation is discretized on the seven-point stencil:

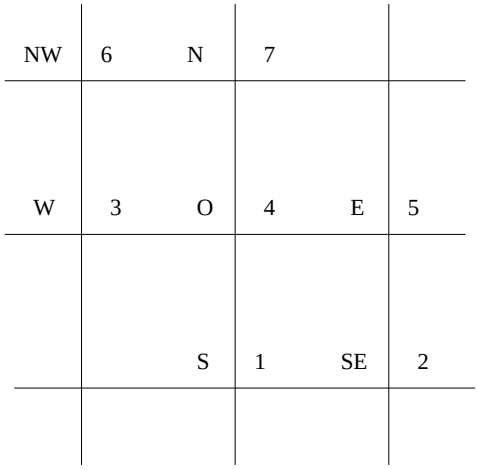

Figure 1. Seven-point stencil

For convenience the approximation $u_{i,j}$ to the exact solution $U(x_i,y_j)$ is denoted by u_{O}, and the neighbouring approximations are labelled according to points of the compass as shown. Where numerical labels for the seven points are required, these are also shown.

The following approximations are used for the second derivatives:

$$\frac{\partial^2 U}{\partial x^2} \simeq \frac{1}{h_x^2}(u_{\mathrm{E}} - 2u_{\mathrm{O}} + u_{\mathrm{W}}),$$

$$\frac{\partial^2 U}{\partial y^2} \simeq \frac{1}{h_y^2}(u_{\mathrm{N}} - 2u_{\mathrm{O}} + u_{\mathrm{S}}),$$

$$\frac{\partial^2 U}{\partial x \partial y} \simeq \frac{1}{2h_x h_y}(u_{\mathrm{N}} - u_{\mathrm{NW}} + u_{\mathrm{E}} - 2u_{\mathrm{O}} + u_{\mathrm{W}} - u_{\mathrm{SE}} + u_{\mathrm{S}}).$$

Two possible schemes may be used to approximate the first derivatives:

Central Differences

$$\frac{\partial U}{\partial x} \simeq \frac{1}{2h_x}(u_{\mathrm{E}} - u_{\mathrm{W}}),$$

$$\frac{\partial U}{\partial y} \simeq \frac{1}{2h_y}(u_{\mathrm{N}} - u_{\mathrm{S}}).$$

Upwind Differences

$$\frac{\partial U}{\partial x} \simeq \frac{1}{h_x}(u_{\mathrm{O}} - u_{\mathrm{W}}) \quad \text{if} \quad \delta(x,y) > 0,$$

$$\frac{\partial U}{\partial x} \simeq \frac{1}{h_x}(u_{\mathrm{E}} - u_{\mathrm{O}}) \quad \text{if} \quad \delta(x,y) < 0,$$

$$\frac{\partial U}{\partial y} \simeq \frac{1}{h_y}(u_{\mathrm{N}} - u_{\mathrm{O}}) \quad \text{if} \quad \epsilon(x,y) > 0,$$

$$\frac{\partial U}{\partial y} \simeq \frac{1}{h_y}(u_{\mathrm{O}} - u_{\mathrm{S}}) \quad \text{if} \quad \epsilon(x,y) < 0.$$

Central differences are more accurate than upwind differences, but upwind differences may lead to a more diagonally dominant matrix for those problems where the coefficients of the first derivatives are significantly larger than the coefficients of the second derivatives.

The approximations used for the first derivatives may be written in a more compact form as follows:

$$\frac{\partial U}{\partial x} \simeq \frac{1}{2h_x}\left((k_x - 1)u_{\mathrm{W}} - 2k_x u_{\mathrm{O}} + (k_x + 1)u_{\mathrm{E}}\right),$$

$$\frac{\partial U}{\partial y} \simeq \frac{1}{2h_y}\left((k_y - 1)u_{\mathrm{S}} - 2k_y u_{\mathrm{O}} + (k_y + 1)u_{\mathrm{N}}\right),$$

where $k_x = \operatorname{sign}\delta$ and $k_y = \operatorname{sign}\epsilon$ for upwind differences, and $k_x = k_y = 0$ for central differences.

At all points in the rectangular domain, including the boundary, the coefficients in the partial differential equation are evaluated by calling the user-supplied subroutine `pde_coeff`, and applying the approximations. This leads to a seven-diagonal system of linear equations of the form:

$$\begin{aligned} & A^6_{ij}u_{i-1,j+1} + A^7_{ij}u_{i,j+1} \\ + \; & A^3_{ij}u_{i-1,j} + A^4_{ij}u_{i,j} + A^5_{ij}u_{i+1,j} \\ & + A^1_{ij}u_{i,j-1} + A^2_{ij}u_{i+1,j-1} = f_{ij}, \quad i = 1,2,\ldots,n_x \text{ and } j = 1,2,\ldots,n_y; \end{aligned}$$

where the coefficients are given by

$$\begin{aligned}
A^1_{ij} &= \beta(x_i,y_j)\frac{1}{2h_xh_y} + \gamma(x_i,y_j)\frac{1}{h_y^2} + \epsilon(x_i,y_j)\frac{1}{2h_y}(k_y-1),\\
A^2_{ij} &= -\beta(x_i,y_j)\frac{1}{2h_xh_y},\\
A^3_{ij} &= \alpha(x_i,y_j)\frac{1}{h_x^2} + \beta(x_i,y_j)\frac{1}{2h_xh_y} + \delta(x_i,y_j)\frac{1}{2h_x}(k_x-1),\\
A^4_{ij} &= -\alpha(x_i,y_j)\frac{2}{h_x^2} - \beta(x_i,y_j)\frac{1}{h_xh_y} - \gamma(x_i,y_j)\frac{2}{h_y^2} - \delta(x_i,y_j)\frac{k_y}{h_x} - \epsilon(x_i,y_j)\frac{k_y}{h_y} - \phi(x_i,y_j),\\
A^5_{ij} &= \alpha(x_i,y_j)\frac{1}{h_x^2} + \beta(x_i,y_j)\frac{1}{2h_xh_y} + \delta(x_i,y_j)\frac{1}{2h_x}(k_x+1),\\
A^6_{ij} &= -\beta(x_i,y_j)\frac{1}{2h_xh_y},\\
A^7_{ij} &= \beta(x_i,y_j)\frac{1}{2h_xh_y} + \gamma(x_i,y_j)\frac{1}{h_y^2} + \epsilon(x_i,y_j)\frac{1}{2h_y}(k_y+1),\\
f_{ij} &= \psi(x_i,y_j).
\end{aligned}$$

These equations then have to be modified to take account of the boundary conditions. These may be Dirichlet (where the solution is given), Neumann (where the derivative of the solution is given), or mixed (where a linear combination of solution and derivative is given).

If the boundary conditions are Dirichlet, there is an infinity of possible equations which may be applied:

$$\mu u_{i,j} = \mu f_{ij}\ ,\ \mu \neq 0. \tag{2}$$

If the procedure `nag_pde_ell_mg_sol` is used to solve the discretized equations, it turns out that the choice of μ can have a dramatic effect on the rate of convergence, and the obvious choice $\mu = 1$ is not always the best. Some choices may even cause the multigrid method to fail altogether. In practice it has been found that a value of the same order as the other diagonal elements of the matrix is best, and the following value has been found to work well in practice:

$$\mu = \min_{ij}\left(-\left\{\frac{2}{h_x^2}+\frac{2}{h_y^2}\right\}, A^4_{ij}\right).$$

If the boundary conditions are either mixed or Neumann (i.e., `b` $\neq 0$ on return from the user-supplied subroutine `bound_cond`), then one of the points in the seven-point stencil lies outside the domain. In this case the normal derivative in the boundary conditions is used to eliminate the 'fictitious' point, u_{outside}:

$$\frac{\partial U}{\partial n} \simeq \frac{1}{2h}(u_{\text{outside}} - u_{\text{inside}}). \tag{3}$$

It should be noted that if the boundary conditions are Neumann and $\phi(x,y) \equiv 0$, then there is no unique solution. The procedure returns with `error%code` $= 102$ in this case, and the seven-diagonal matrix is singular.

The four corners are treated separately. The user-supplied subroutine `bound_cond` is called twice, once along each of the edges meeting at the corner. If both boundary conditions at this point are Dirichlet and the prescribed solution values agree, then this value is used in an equation of the form (2). If the prescribed solution is discontinuous at the corner, then the average of the two values is used. If one boundary condition is Dirichlet and the other is mixed, then the value prescribed by the Dirichlet condition is used in an equation of the form given above. Finally, if both conditions are mixed or Neumann, then two 'fictitious' points are eliminated using two equations of the form (3).

It is possible that equations for which the solution is known at all points on the boundary, have coefficients which are not defined on the boundary. Since this procedure calls the user-supplied subroutine `pde_coeff`

at *all* points in the domain, including boundary points, arithmetic errors may occur in the user's procedure `pde_coeff` which this procedure cannot trap. If the user has an equation with Dirichlet boundary conditions (i.e., `b` = 0 at all points on the boundary), but with PDE coefficients which are singular on the boundary, then the procedure `nag_pde_ell_mg_sol` could be called directly only using interior grid points with the user's own discretization.

After the equations have been set up as described above, they are checked for diagonal dominance. That is to say,

$$|A_{ij}^4| \geq \sum_{k \neq 4} |A_{ij}^k|, \quad i = 1, 2, \ldots, n_x \text{ and } j = 1, 2, \ldots, n_y.$$

If this condition is not satisfied then the procedure returns with `error%code` = 103. The multigrid procedure `nag_pde_ell_mg_sol` may still converge in this case, but if the coefficients of the first derivatives in the partial differential equation are large compared with the coefficients of the second derivatives, the user should consider using upwind differences (`upwind_diff` = `.true.`).

Since this procedure is designed primarily for use with `nag_pde_ell_mg_sol`, this document should be read in conjunction with the document for that procedure.

Procedure: nag_pde_ell_rect

1 Description

nag_pde_ell_rect discretizes a second order linear elliptic partial equation of the form

$$\alpha(x,y)\frac{\partial^2 U}{\partial x^2}+\beta(x,y)\frac{\partial^2 U}{\partial x \partial y}+\gamma(x,y)\frac{\partial^2 U}{\partial y^2}+\delta(x,y)\frac{\partial U}{\partial x}+\epsilon(x,y)\frac{\partial U}{\partial y}+\phi(x,y)U=\psi(x,y) \tag{4}$$

on a rectangular region

$$x_A \le x \le x_B, \quad y_A \le y \le y_B;$$

subject to boundary conditions of the form

$$a(x,y)U+b(x,y)\frac{\partial U}{\partial n}=c(x,y)$$

where $\dfrac{\partial U}{\partial n}$ denotes the outward pointing normal derivative on the boundary. Equation 4 is said to be elliptic if

$$4\alpha(x,y)\gamma(x,y) \ge \beta^2(x,y)$$

for all points in the rectangular region. The linear equations produced are in a form suitable for passing directly to the multigrid procedure nag_pde_ell_mg_sol.

The equation is discretized on a rectangular grid, with n_x grid points in the x-direction and n_y grid points in the y-direction. At all points in the rectangular domain, including the boundary, the coefficients in the partial differential equation are evaluated by calling the user-supplied procedure pde_coeff, and applying the approximations (see the Module Introduction). This leads to a seven-diagonal system of linear equations of the form:

$$\begin{aligned} & A^6_{ij}u_{i-1,j+1} + A^7_{ij}u_{i,j+1} \\ + \; & A^3_{ij}u_{i-1,j} + A^4_{ij}u_{i,j} + A^5_{ij}u_{i+1,j} \\ & + A^1_{ij}u_{i,j-1} + A^2_{ij}u_{i+1,j-1} = f_{ij}, \quad i=1,2,\ldots,n_x \text{ and } j=1,2,\ldots,n_y. \end{aligned}$$

These equations then have to be modified to take account of the boundary conditions. These may be Dirichlet (where the solution is given), Neumann (where the derivative of the solution is given), or mixed (where a linear combination of solution and derivative is given). Those modifications are evaluated by calling the user-supplied procedure bound_cond. (See the Module Introduction for further details.)

2 Usage

```
USE nag_pde_ell_mg

CALL nag_pde_ell_rect(pde_coeff, bound_cond, nx, ny, x_min, x_max, y_min, y_max, a, &
                      rhs  [, optional arguments])
```

3 Arguments

3.1 Mandatory Arguments

pde_coeff — subroutine

The user-supplied procedure `pde_coeff` must evaluate the functions $\alpha(x,y)$, $\beta(x,y)$, $\gamma(x,y)$, $\delta(x,y)$, $\epsilon(x,y)$, $\phi(x,y)$ and $\psi(x,y)$ which define the equation at a general point (x,y).

Its specification is:

```
subroutine pde_coeff(x, y, coeff, i_comm, r_comm)

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

Input: the x and y co-ordinates of the point at which the coefficients of the partial differential equation are to be evaluated.

```
real(kind=wp), intent(out) ::  coeff(7)
```

Output: `coeff`(1:7) must be set to the value of $\alpha(x,y)$, $\beta(x,y)$, $\gamma(x,y)$, $\delta(x,y)$, $\epsilon(x,y)$, $\phi(x,y)$ and $\psi(x,y)$ respectively at the point specified by `x` and `y`.

```
integer, intent(in), optional ::  i_comm(:)
real(kind=wp), intent(in), optional ::  r_comm(:)
```

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

bound_cond — subroutine

The user-supplied procedure `bound_cond` must evaluate the functions $a(x,y)$, $b(x,y)$ and $c(x,y)$ involved in the boundary conditions.

Its specification is:

```
subroutine bound_cond(x, y, a, b, c, bnd, i_comm, r_comm)

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

Input: the x and y co-ordinates of the point at which the boundary conditions are to be evaluated.

```
real(kind=wp), intent(out) ::  a
real(kind=wp), intent(out) ::  b
real(kind=wp), intent(out) ::  c
```

Output: `a`, `b` and `c` must be set to the value of $a(x,y)$, $b(x,y)$ and $c(x,y)$ respectively at the point specified by `x` and `y`.

```
integer, intent(in) ::  bnd
```

Input: `bnd` specifies on which boundary the point (`x`,`y`) lies. `bnd` $= 0, 1, 2$ or 3 according to whether the point lies on the bottom, right, top or left boundary.

```
integer, intent(in), optional ::  i_comm(:)
real(kind=wp), intent(in), optional ::  r_comm(:)
```

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

nx — integer, intent(in)

ny — integer, intent(in)

Input: the number of interior grid points n_x and n_y in the x- and y-directions respectively. If the seven-diagonal equations are to be solved by the procedure nag_pde_ell_mg_sol then nx−1 and ny−1 should preferably be divisible by as high a power of 2 as possible.

Constraints: nx ≥ 3, ny ≥ 3.

x_min — real(kind=*wp*), intent(in)

x_max — real(kind=*wp*), intent(in)

Input: the lower and upper bounds x_A and x_B of the range of x respectively.

Constraints: x_min $<$ x_max.

y_min — real(kind=*wp*), intent(in)

y_max — real(kind=*wp*), intent(in)

Input: the lower and upper bounds y_A and y_B of the range of y, respectively.

Constraints: y_min $<$ y_max.

a($n_x \times n_y, 7$) — real(kind=*wp*), intent(out)

Output: a(i, j), for $i = 1, 2, \ldots, n_x \times n_y$ and $j = 1, 2, \ldots, 7$; contains the seven-diagonal linear equations produced by the discretization described in the Module Introduction. The array a can then be passed directly to the procedure nag_pde_ell_mg_sol to solve the system.

rhs($n_x \times n_y$) — real(kind=*wp*), intent(out)

Output: the right hand sides of the seven-diagonal linear equations produced by the discretization described in the Module Introduction, which may be passed directly to the procedure nag_pde_ell_mg_sol to solve the system.

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.

upwind_diff — logical, intent(in), optional

Input: the type of approximation to be used for the first derivatives which occur in the partial differential equation (see the Module Introduction for more details about approximation types).

If upwind_diff = .false., then central differences are used;

if upwind_diff = .true., then upwind differences are used.

Default: upwind_diff = .false..

Note: generally speaking, if at least one of the coefficients multiplying the first derivatives ($\delta(x, y)$ or $\epsilon(x, y)$ returned by pde_coeff as coeff(4) and coeff(5) respectively) is large compared with the coefficients multiplying the second derivatives, then upwind differences may be more appropriate. Upwind differences are less accurate that central differences, but may result in more rapid convergence for strongly convective equations. The easiest test is to try both schemes.

i_comm(:) — integer, intent(in), optional

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

Input: these arrays are not used by this procedure, but they are passed directly from the calling (sub)program to the user-supplied procedures `pde_coeff` and/or `bound_cond`, and hence may be used to pass information to them.

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.

Failures (error%level = 2):

error%code	Description
201	At some point on the boundary there is a derivative in the boundary conditions and there is a non-zero coefficient of the mixed derivative $\dfrac{\partial^2 U}{\partial x \partial y}$. `b` $\neq 0$ on return from `bound_cond` and $\beta \neq 0$ on return from `pde_coeff`.
202	A null boundary has been specified. That means that at some point both `a` and `b` are zero on return from a call to `bound_cond`.

Warnings (error%level = 1):

error%code	Description
101	The equation is not elliptic, i.e., $4\alpha\gamma < \beta^2$ after a call to `pde_coeff`. The discretization has been completed, but the convergence of the procedure `nag_pde_ell_mg_sol` cannot be guaranteed.
102	The boundary conditions are purely Neumann. Only the derivative is specified, and there is in general no unique solution.
103	The equations are not diagonally dominant. See the Module Introduction.

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

If this procedure is used as a pre-processor to the multigrid procedure nag_pde_ell_mg_sol it should be noted that the rate of convergence of that procedure is strongly dependent upon the number of levels in the multigrid scheme, and thus the choice of nx and ny is very important.

Procedure: nag_pde_ell_mg_sol

1 Description

nag_pde_ell_mg_sol solves, by multigrid iteration, the seven-point scheme

$$\begin{aligned} A^6_{i,j}u_{i-1,j+1} &+ A^7_{i,j}u_{i,j+1} &+ \\ A^3_{i,j}u_{i-1,j} &+ A^4_{i,j}u_{i,j} &+ A^5_{i,j}u_{i+1,j} \\ &+ A^1_{i,j}u_{i,j-1} &+ A^2_{i,j}u_{i+1,j-1} = f_{ij}, \quad i = 1,2,\ldots,n_x \text{ and } j = 1,2,\ldots,n_y, \end{aligned}$$

which arises from the discretization of an elliptic partial differential equation of the form

$$\alpha(x,y)\frac{\partial^2 U}{\partial x^2} + \beta(x,y)\frac{\partial^2 U}{\partial x \partial y} + \gamma(x,y)\frac{\partial^2 U}{\partial y^2} + \delta(x,y)\frac{\partial U}{\partial x} + \epsilon(x,y)\frac{\partial U}{\partial y} + \phi(x,y)U = \psi(x,y)$$

and its boundary conditions, on a rectangular domain. This may be written in matrix form as $Au = f$.

The algorithm is described in separate reports by Wesseling [2], Wesseling [3] and McCarthy [1].

Systems of linear equations, matching the seven-point stencil defined above (see also Figure 1 in the Module Introduction), are solved by a multigrid iteration. An initial estimate of the solution must be provided by the user. A zero guess may be supplied if no better approximation is available.

A 'smoother' based on incomplete LU decomposition is used to eliminate the high frequency components of the error. A restriction operator is then used to map the system on to a sequence of coarser grids. The errors are then smoothed and prolongated (mapped onto successively finer grids). When the finest cycle is reached, the approximation to the solution is corrected. The cycle is repeated for max_iter iterations or until the required accuracy (acc) is reached.

This procedure will automatically determine the number l of possible coarse grids for a particular problem. In other words, this procedure determines the maximum integer l so that n_x and n_y can be expressed in the form

$$n_x = m2^{l-1} + 1, \quad n_y = n2^{l-1} + 1, \quad \text{with} \quad m \geq 2 \quad \text{and} \quad n \geq 2.$$

It should be noted that the rate of convergence improves significantly with the number of levels used (see McCarthy [1]), so that n_x and n_y should be carefully chosen so that $n_x - 1$ and $n_y - 1$ have factors of the form 2^l, with l as large as possible. For good convergence l should be at least 2.

2 Usage

```
USE nag_pde_ell_mg
CALL nag_pde_ell_mg_sol(nx, ny, a, rhs, u  [, optional arguments])
```

3 Arguments

3.1 Mandatory Arguments

nx — integer, intent(in)

Input: the number of grid points n_x in the x-direction. nx−1 should preferably be divisible by as high a power of 2 as possible.

Constraints: nx ≥ 3.

ny — integer, intent(in)

Input: the number of grid points n_y in the y-direction. ny-1 should preferably be divisible by as high a power of 2 as possible.

Constraints: ny ≥ 3.

a$(n_x \times n_y, 7)$ — real(kind=*wp*), intent(inout)

Input: the values of the left-hand side matrix, as follows:

$$\texttt{a}(i+(j-1)n_x, k) = A_{ij}^k, \quad i = 1, 2, \ldots, n_x,\ j = 1, 2, \ldots, n_y \text{ and } k = 1, \ldots, 7.$$

Output: a is overwritten by the incomplete LU factorization (on the finest mesh).

rhs$(n_x \times n_y)$ — real(kind=*wp*), intent(in)

Input: the values of the right-hand side f:

$$\texttt{rhs}(i+(j-1)n_x) = f_{ij}, \quad i = 1, 2, \ldots, n_x \text{ and } j = 1, 2, \ldots, n_y.$$

u$(n_x \times n_y)$ — real(kind=*wp*), intent(inout)

Input: the values of the initial estimate u^0 for the solution u:

$$\texttt{u}(i+(j-1)n_x) = u_{ij}^0, \quad i = 1, 2, \ldots, n_x \text{ and } j = 1, 2, \ldots, n_y.$$

Output: the computed solution u:

$$\texttt{u}(i+(j-1)n_x) = u_{i,j}, \quad i = 1, 2, \ldots, n_x \text{ and } j = 1, 2, \ldots, n_y.$$

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.

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

Output: the residual 2-norm.

max_iter — integer, intent(in), optional

Input: the maximum permitted number of multigrid iterations. If max_iter $= 0$, no multigrid iterations are performed but the coarse-grid approximations and incomplete LU decompositions are computed.

Constraints: max_iter ≥ 0.

Default: max_iter $= 100$.

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

Input: the required tolerance for convergence of the residual 2-norm:

$$\|r\|_2 = \sqrt{\sum_{k=1}^{n_x \times n_y} (r_k)^2}$$

where $r = f - Au$ and u is the computed solution. Note that the norm is not scaled by the number of equations. The procedure will stop after fewer than max_iter iterations if the residual 2-norm is less than the specified tolerance. (If max_iter > 0, at least one iteration is always performed).

Constraints: acc $\geq$ EPSILON(1.0_*wp*).

Default: acc $=$ EPSILON(1.0_*wp*).

print_level — integer, intent(in), optional

Input: controls the amount of output produced by nag_pde_ell_mg_sol. The following output is sent to the Fortran unit number defined by the optional argument unit:

print_level = 0, no output;

print_level = 1, the solution $u_{i,j}$, for $i = 1, 2, \ldots, n_x$ and $j = 1, 2, \ldots, n_y$;

print_level = 2, the residual 2-norm after each iteration, with the reduction factor over the previous iteration;

print_level = 3, as for print_level = 1 and print_level = 2;

print_level = 4, as for print_level = 3, plus the final residual;

print_level = 5, as for print_level = 4, plus the initial element of a and rhs;

print_level = 6, as for print_level = 5, plus the coarse grid approximations on all grids;

print_level = 7, as for print_level = 6, plus the incomplete LU decompositions on all grids;

print_level = 8, as for print_level = 7, plus the residual after each iteration.

The element $\mathtt{a}(p, k)$, the coarse grid approximations and the incomplete LU decompositions are output in the format:

Y-index = j

X-index = i $\quad \mathtt{a}(p,1) \quad \mathtt{a}(p,2) \quad \mathtt{a}(p,3) \quad \mathtt{a}(p,4) \quad \mathtt{a}(p,5) \quad \mathtt{a}(p,6) \quad \mathtt{a}(p,7)$

where $p = 1 + (j-1) \times n_x$, for $i = 1, 2, \ldots, n_x$ and $j = 1, 2, \ldots, n_y$.

The vectors u and rhs are output in matrix form with ny rows and nx columns. Where nx > 10, the nx values for a given j-value are produced in rows of 10. Values of print_level > 4 may therefore produce considerable amounts of output.

Constraints: $0 \leq$ print_level ≤ 8.

Default: print_level = 0.

unit — integer, intent(in), optional

Input: specifies the Fortran unit number which identifies the file to be written to.

Constraints: unit ≥ 0.

Default: unit = the default output unit number for the implementation.

num_iter — integer, intent(out), optional

Output: the number of iterations performed.

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.

Warnings (error%level = 1):

error%code	Description
101	Unable to achieve the required tolerance within `max_iter` iterations. `max_iter` iterations have been performed with the residual 2-norm decreasing at each iteration but it has not been reduced to less than the specified tolerance `acc`. Examine the progress of the iterations by setting `print_level` ≥ 2. You could try increasing `max_iter` or `acc`.
102	Unable to achieve the required tolerance (non-decreasing residuals). `max_iter` iterations have been performed but the residual 2-norm has not been reduced to less than the specified tolerance `acc`. At one or more iterations the residual 2-norm did not decrease. It is likely that the method fails to converge for the given matrix A.
103	Unused optional input argument. The optional argument `unit` is present while no output is required (the optional argument `print_level` is not present). `unit` will not be used.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 3 of this module document. They show that the number of iterations is essentially independent of the size of the problem.

6 Further Comments

6.1 Algorithmic Detail

This procedure has been found to be robust in applications, but being an iterative method the problem of divergence can arise. For a strictly diagonally dominant matrix A

$$|A_{ij}^4| \geq \sum_{k \neq 4} |A_{ij}^k|, \quad i = 1, 2, \ldots, n_x \text{ and } j = 1, 2, \ldots, n_y;$$

no such problem is foreseen. The diagonal dominance of A is not a necessary condition, but should this condition be strongly violated then divergence may occur. The quickest test is to try the procedure.

The rate of convergence of this procedure is strongly dependent upon the number of levels, l, in the multigrid scheme, and thus the choice of n_x and n_y is very important. The user is advised to experiment with different values of n_x and n_y to see the effect they have on the rate of convergence; e.g., by using a value such as $n_x = 65(2^6 + 1)$ followed by $n_x = 64$ (for which $l = 1$).

Example 1: Solves the Laplace Equation With an Exact Discretization

The following program solves the elliptic partial differential equation

$$-(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}) = 0,$$

on the unit square $0 \leq x, y \leq 1$, with boundary conditions

$$U \text{ given on } x = 1, \ x = 0, \ y = 0 \text{ and } y = 1.$$

Dirichlet boundary conditions derived from the exact solution $U(x, y) = x^2 - y^2$ are applied.

As expected the results show that the multigrid method is achieving grid-independent convergence rates.

For some machines the largest problem solved in this example may exhaust the available memory. In this case, reduce the value of the parameter `level_up` accordingly.

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_pde_ell_mg_ex01

  ! Example Program Text for nag_pde_ell_mg
  ! NAG f190, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_pde_ell_mg, ONLY : nag_pde_ell_mg_sol
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EPSILON, KIND, MAX, MIN, PRECISION, REAL, SQRT, SUM
  ! .. Parameters ..
  INTEGER, PARAMETER :: level_low = 3
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  INTEGER, PARAMETER :: level_up = MIN(8,PRECISION(0.0_wp))
  INTEGER, PARAMETER :: maxsize = (2**level_up+1)**2
  REAL (wp), PARAMETER :: one = 1.0_wp
  REAL (wp), PARAMETER :: two = 2.0_wp
  REAL (wp), PARAMETER :: zero = 0.0_wp
  REAL (wp), PARAMETER :: mone = -one
  ! .. Local Scalars ..
  INTEGER :: i, ix, iy, j, k, level, max_iter, num_iter, nx, nxy, ny
  REAL (wp) :: acc, hx, hx2, hy, hy2, mu, resid_norm, rms_err, xi, xj, xj2
  ! .. Local Arrays ..
  REAL (wp) :: a(maxsize,7), rhs(maxsize), sol(maxsize), u(maxsize)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_pde_ell_mg_ex01'

  max_iter = 1000

  ! Do-loop on the level

  DO level = level_low, level_up
    nx = 2**level + 1
    ny = nx
    nxy = nx*ny
    acc = MAX(1.0E-6_wp,SQRT(nx*EPSILON(zero)))
```

```
! Set-up operator, right-hand side and initial guess

hx = one/REAL(nx-1,kind=wp)
hx2 = hx*hx
hy = one/REAL(ny-1,kind=wp)
hy2 = hy*hy
mu = two/hx2 + two/hy2

a(1:nxy,1) = mone/hy2
a(1:nxy,2) = zero
a(1:nxy,3) = mone/hx2
a(1:nxy,4) = mu
a(1:nxy,5) = mone/hx2
a(1:nxy,6) = zero
a(1:nxy,7) = mone/hy2

u(1:nxy) = zero
rhs(1:nxy) = zero

! Exact solution "sol"
DO j = 1, ny
  xj = REAL(j-1,kind=wp)*hy
  xj2 = xj*xj
  DO i = 1, nx
    xi = REAL(i-1,kind=wp)*hx
    k = i + (j-1)*nx
    sol(k) = xi*xi - xj2
  END DO
END DO
! Correction for the boundary conditions
! Horizontal boundaries
DO i = 1, nx
  ! Boundary condition Y = 0
  ix = i
  u(ix) = sol(ix)
  rhs(ix) = mu*sol(ix)
  a(ix,1:7:2) = zero
  ! Boundary condition Y = 1
  ix = i + (ny-1)*nx
  u(ix) = sol(ix)
  rhs(ix) = mu*sol(ix)
  a(ix,1:7:2) = zero
END DO
! Vertical boundaries
DO j = 1, ny
  ! Boundary condition X = 0
  iy = (j-1)*nx + 1
  u(iy) = sol(iy)
  rhs(iy) = mu*sol(iy)
  a(iy,1:7:2) = zero
  ! Boundary condition X = 1
  iy = j*nx
  u(iy) = sol(iy)
  rhs(iy) = mu*sol(iy)
  a(iy,1:7:2) = zero
END DO
! Solve the equation
CALL nag_pde_ell_mg_sol(nx,ny,a(1:nxy,1:7),rhs(1:nxy),u(1:nxy), &
 resid_norm=resid_norm,acc=acc,num_iter=num_iter,max_iter=max_iter)
WRITE (nag_std_out,*) ' '
WRITE (nag_std_out,999) ' Level ', level
WRITE (nag_std_out,999) ' Order of the matrix ', nxy
```

```
        WRITE (nag_std_out,998) ' Accuracy ', acc
        WRITE (nag_std_out,999) ' Number of iteration ', num_iter
        WRITE (nag_std_out,998) ' Residual norm ', resid_norm

        rms_err = SUM((u(:nxy)-sol(:nxy))**2)
        rms_err = SQRT(rms_err/REAL(nxy,kind=wp))

        WRITE (nag_std_out,998) ' RMS Error ', rms_err
      END DO

999   FORMAT (1X,A,I7)
998   FORMAT (1X,A,1P,E10.4)

    END PROGRAM nag_pde_ell_mg_ex01
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_pde_ell_mg_ex01

  Level       3
  Order of the matrix      81
  Accuracy 1.0000E-06
  Number of iteration       7
  Residual norm 1.2358E-07
  RMS Error 1.1239E-10

  Level       4
  Order of the matrix     289
  Accuracy 1.0000E-06
  Number of iteration       8
  Residual norm 3.3638E-07
  RMS Error 4.1703E-11

  Level       5
  Order of the matrix    1089
  Accuracy 1.0000E-06
  Number of iteration       9
  Residual norm 2.2809E-07
  RMS Error 3.9178E-12

  Level       6
  Order of the matrix    4225
  Accuracy 1.0000E-06
  Number of iteration      10
  Residual norm 1.5476E-07
  RMS Error 3.3047E-13

  Level       7
  Order of the matrix   16641
  Accuracy 1.0000E-06
  Number of iteration      10
  Residual norm 8.8075E-07
  RMS Error 3.0404E-13

  Level       8
  Order of the matrix   66049
  Accuracy 1.0000E-06
```

```
Number of iteration      11
Residual norm 5.8939E-07
RMS Error 8.5490E-14
```

Example 2: Solves an Elliptic Partial Differential Equation With Convection Terms

The following program solves the elliptic partial differential equation

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + 50\left\{\frac{\partial U}{\partial x} + \frac{\partial U}{\partial y}\right\} = f(x,y)$$

on the unit square $0 \leq x,\, y \leq 1$, with boundary conditions

$$\begin{array}{ll} \dfrac{\partial U}{\partial n} & \text{given on} \quad x = 0 \quad \text{and} \quad y = 0, \\ U & \text{given on} \quad x = 1 \quad \text{and} \quad y = 1. \end{array}$$

The function $f(x,y)$ and the exact form of the boundary conditions are derived from the exact solution $U(x,y) = \sin x \sin y$.

The equation is first solved using central differences. Because of the first derivative terms, the linear equations are not diagonally dominant, and (as expected) convergence is slow. The equation is solved a second time with upwind differences, showing that convergence is more rapid, but the solution is less accurate.

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 nag_pde_ell_mg_ex02_mod

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

CONTAINS

  SUBROUTINE pde_coeff(x,y,coeff,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC COS, SIN
    ! .. Scalar Arguments ..
    REAL (wp), INTENT (IN) :: x, y
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), INTENT (OUT) :: coeff(7)
    REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:)
    ! .. Executable Statements ..
    coeff(1:6) = r_comm(1:6)

    ! PSI = (-ALPHA-GAMMA+PHI)*SIN(X)*SIN(Y) + BETA*COS(X)*COS(Y) +
    ! .      + DELTA*COS(X)*SIN(Y) + EPSLON*SIN(X)*COS(Y)

    coeff(7) = (-coeff(1)-coeff(3)+coeff(6))*SIN(x)*SIN(y) + &
     coeff(2)*COS(x)*COS(y) + coeff(4)*COS(x)*SIN(y) + &
     coeff(5)*SIN(x)*COS(y)

  END SUBROUTINE pde_coeff
```

```
  SUBROUTINE bound_cond(x,y,a,b,c,bnd,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIN
    ! .. Parameters ..
    INTEGER, PARAMETER :: bottom = 0, left = 3, right = 1, top = 2
    REAL (wp), PARAMETER :: one = 1.0_wp
    REAL (wp), PARAMETER :: zero = 0.0_wp
    ! .. Scalar Arguments ..
    INTEGER, INTENT (IN) :: bnd
    REAL (wp), INTENT (OUT) :: a, b, c
    REAL (wp), INTENT (IN) :: x, y
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:)
    ! .. Executable Statements ..

    IF (bnd==top .OR. bnd==right) THEN
      ! Solution prescribed
      a = one
      b = zero
      c = SIN(x)*SIN(y)
    ELSE IF (bnd==bottom) THEN
      ! Derivative prescribed
      a = zero
      b = one
      c = -SIN(x)
    ELSE IF (bnd==left) THEN
      ! Derivative prescribed
      a = zero
      b = one
      c = -SIN(y)
    END IF

  END SUBROUTINE bound_cond

END MODULE nag_pde_ell_mg_ex02_mod
PROGRAM nag_pde_ell_mg_ex02

  ! Example Program Text for nag_pde_ell_mg
  ! NAG f190, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_math_constants, ONLY : nag_pi
  USE nag_pde_ell_mg, ONLY : nag_pde_ell_mg_sol, nag_pde_ell_rect
  USE nag_pde_ell_mg_ex02_mod, ONLY : pde_coeff, bound_cond, wp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EPSILON, MAX, REAL, SIN, SQRT
  ! .. Parameters ..
  INTEGER, PARAMETER :: levels = 3
  INTEGER, PARAMETER :: nx = 2**levels + 1
  INTEGER, PARAMETER :: ny = nx
  INTEGER, PARAMETER :: nxy = nx*ny
  REAL (wp), PARAMETER :: fifty = 50.0_wp
  REAL (wp), PARAMETER :: one = 1.0_wp
  REAL (wp), PARAMETER :: zero = 0.0_wp
  ! .. Local Scalars ..
```

```
INTEGER :: i, j, k1, k2, max_iter, num_iter
REAL (wp) :: acc, hx, hy, pi, rms_err, x_max, x_min, y_max, y_min
LOGICAL :: upwind_diff
! .. Local Arrays ..
REAL (wp) :: a(nxy,7), rhs(nxy), r_comm(6), u(nxy), x(nxy), y(nxy)
! .. Executable Statements ..
WRITE (nag_std_out,*) 'Example Program Results for nag_pde_ell_mg_sol'
WRITE (nag_std_out,*) ' '
pi = nag_pi(0.0_wp)
! r_comm(1:6) contains the coefficient alpha, beta, gamma, delta,
! epsilon and phi appearing in the example PDE.
! They are stored for the use in subroutine pde_coeff
r_comm(1) = one
r_comm(2) = zero
r_comm(3) = one
r_comm(4) = fifty
r_comm(5) = fifty
r_comm(6) = zero

x_min = zero
x_max = one
y_min = zero
y_max = one
hx = (x_max-x_min)/REAL(nx-1,kind=wp)
hy = (y_max-y_min)/REAL(ny-1,kind=wp)

y(1:nx) = y_min
x(1:nx) = x_min + hx*(/ (REAL(i-1,kind=wp),i=1,nx) /)
x(nx) = x_max

y(nxy+1-nx:nxy) = y_max
x(nxy+1-nx:nxy) = x(1:nx)

DO j = 2, ny - 1
  k2 = j*nx
  k1 = k2 + 1 - nx
  y(k1:k2) = y_min + REAL(j-1,kind=wp)*hy
  x(k1:k2) = x(1:nx)
END DO

! Discretize the equations :
! Set-up operator, right-hand side

CALL nag_pde_ell_rect(pde_coeff,bound_cond,nx,ny,x_min,x_max,y_min, &
 y_max,a,rhs,r_comm=r_comm)

! Set-up initial guess

u(1:nxy) = zero

! Solve the equation

acc = MAX(1.0E-6_wp,SQRT(nx*EPSILON(zero)))
max_iter = 50
CALL nag_pde_ell_mg_sol(nx,ny,a,rhs,u,acc=acc,max_iter=max_iter, &
 num_iter=num_iter)

! Print out the Solution

WRITE (nag_std_out,*) ' '
WRITE (nag_std_out,*) ' Exact solution above computed solution'
WRITE (nag_std_out,*) ' '
```

```
      WRITE (nag_std_out,999) '  I/J', (i,i=1,nx)
      rms_err = zero
      DO j = ny, 1, -1
        WRITE (nag_std_out,*) ' '
        WRITE (nag_std_out,998) j, (SIN(x(i+(j-1)*nx))*SIN(y(i+(j-1)*nx)),i=1, &
         nx)
        WRITE (nag_std_out,998) j, (u(i+(j-1)*nx),i=1,nx)
        DO i = 1, nx
          rms_err = rms_err + (SIN(x(i+(j-1)*nx))*SIN(y(i+ &
           (j-1)*nx))-u(i+(j-1)*nx))**2
        END DO
      END DO
      rms_err = SQRT(rms_err/REAL(nx*ny,kind=wp))
      WRITE (nag_std_out,*) ' '
      WRITE (nag_std_out,997) ' Number of Iteration ', num_iter
      WRITE (nag_std_out,996) ' RMS Error ', rms_err

      ! Now discretize and solve the equations using upwinding differences
      ! Set-up operator, right-hand side

      upwind_diff = .TRUE.
      CALL nag_pde_ell_rect(pde_coeff,bound_cond,nx,ny,x_min,x_max,y_min, &
       y_max,a,rhs,r_comm=r_comm,upwind_diff=upwind_diff)

      ! Set-up initial guess

      u(1:nxy) = zero

      ! Solve the equation

      CALL nag_pde_ell_mg_sol(nx,ny,a,rhs,u,acc=acc,max_iter=max_iter, &
       num_iter=num_iter)

      ! Print out the Solution

      WRITE (nag_std_out,*) ' '
      WRITE (nag_std_out,*) ' Exact solution above computed solution'
      WRITE (nag_std_out,*) ' '
      WRITE (nag_std_out,999) '  I/J', (i,i=1,nx)
      rms_err = zero
      DO j = ny, 1, -1
        WRITE (nag_std_out,*) ' '
        WRITE (nag_std_out,998) j, (SIN(x(i+(j-1)*nx))*SIN(y(i+(j-1)*nx)),i=1, &
         nx)
        WRITE (nag_std_out,998) j, (u(i+(j-1)*nx),i=1,nx)
        DO i = 1, nx
          rms_err = rms_err + (SIN(x(i+(j-1)*nx))*SIN(y(i+ &
           (j-1)*nx))-u(i+(j-1)*nx))**2
        END DO
      END DO
      rms_err = SQRT(rms_err/REAL(nx*ny,kind=wp))
      WRITE (nag_std_out,*) ' '
      WRITE (nag_std_out,997) ' Number of Iteration ', num_iter
      WRITE (nag_std_out,996) ' RMS Error ', rms_err

999   FORMAT (1X,A,10I7:/(6X,10I7))
998   FORMAT (1X,I3,2X,10F7.3:/(6X,10F7.3))
997   FORMAT (1X,A,I3)
996   FORMAT (1X,A,1P,E10.2)

    END PROGRAM nag_pde_ell_mg_ex02
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_pde_ell_mg_sol

 ****************** Warning reported by NAG Fortran 90 Library *****************
 Procedure nag_pde_ell_rect                 Level = 1  Code = 103
 The equations are not diagonally dominant.
 See the description section of the procedure document.
 *************************** Execution continued ******************************

  Exact solution above computed solution

   I/J      1      2      3      4      5      6      7      8      9

   9    0.000  0.105  0.208  0.308  0.403  0.492  0.574  0.646  0.708
   9   -0.000  0.105  0.208  0.308  0.403  0.492  0.574  0.646  0.708

   8    0.000  0.096  0.190  0.281  0.368  0.449  0.523  0.589  0.646
   8   -0.000  0.095  0.190  0.281  0.368  0.449  0.523  0.589  0.646

   7    0.000  0.085  0.169  0.250  0.327  0.399  0.465  0.523  0.574
   7   -0.000  0.084  0.168  0.249  0.326  0.398  0.464  0.523  0.574

   6    0.000  0.073  0.145  0.214  0.281  0.342  0.399  0.449  0.492
   6   -0.001  0.072  0.144  0.213  0.280  0.342  0.398  0.449  0.492

   5    0.000  0.060  0.119  0.176  0.230  0.281  0.327  0.368  0.403
   5   -0.001  0.059  0.118  0.174  0.229  0.280  0.326  0.368  0.403

   4    0.000  0.046  0.091  0.134  0.176  0.214  0.250  0.281  0.308
   4   -0.001  0.044  0.089  0.133  0.174  0.213  0.249  0.281  0.308

   3    0.000  0.031  0.061  0.091  0.119  0.145  0.169  0.190  0.208
   3   -0.001  0.029  0.060  0.089  0.118  0.144  0.168  0.190  0.208

   2    0.000  0.016  0.031  0.046  0.060  0.073  0.085  0.096  0.105
   2   -0.001  0.014  0.029  0.044  0.059  0.072  0.084  0.095  0.105

   1    0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
   1   -0.001 -0.001 -0.001 -0.001 -0.001 -0.001 -0.000 -0.000 -0.000

 Number of Iteration  10
 RMS Error   7.92E-04

 Exact solution above computed solution

   I/J      1      2      3      4      5      6      7      8      9

   9    0.000  0.105  0.208  0.308  0.403  0.492  0.574  0.646  0.708
   9   -0.000  0.105  0.208  0.308  0.403  0.492  0.574  0.646  0.708

   8    0.000  0.096  0.190  0.281  0.368  0.449  0.523  0.589  0.646
   8   -0.002  0.093  0.186  0.276  0.362  0.443  0.517  0.585  0.646

   7    0.000  0.085  0.169  0.250  0.327  0.399  0.465  0.523  0.574
   7   -0.005  0.078  0.160  0.239  0.316  0.388  0.455  0.517  0.574

   6    0.000  0.073  0.145  0.214  0.281  0.342  0.399  0.449  0.492
```

```
  6   -0.008  0.063  0.132  0.200  0.266  0.329  0.388  0.443  0.492

  5    0.000  0.060  0.119  0.176  0.230  0.281  0.327  0.368  0.403
  5   -0.011  0.047  0.103  0.159  0.214  0.266  0.316  0.362  0.403

  4    0.000  0.046  0.091  0.134  0.176  0.214  0.250  0.281  0.308
  4   -0.013  0.030  0.074  0.117  0.159  0.200  0.239  0.276  0.308

  3    0.000  0.031  0.061  0.091  0.119  0.145  0.169  0.190  0.208
  3   -0.015  0.014  0.044  0.074  0.103  0.132  0.160  0.186  0.208

  2    0.000  0.016  0.031  0.046  0.060  0.073  0.085  0.096  0.105
  2   -0.016 -0.001  0.014  0.030  0.047  0.063  0.078  0.093  0.105

  1    0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
  1   -0.016 -0.016 -0.015 -0.013 -0.011 -0.008 -0.005 -0.002 -0.000

 Number of Iteration   4
 RMS Error   1.05E-02
```

Example 3: Solves the Poisson Equation

The following program solves the elliptic partial differential equation

$$-(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}) = f(x, y)$$

on the unit square $0 \leq x, y \leq 1$, with boundary conditions

$$U \text{ given on } x = 1, \ x = 0, \ y = 0 \text{ and } y = 1.$$

The function $f(x, y)$ and the boundary conditions are derived from the exact solution $U(x, y) = \sin(\pi x)\sin(\pi y)$.

As expected the results show that the multigrid method is achieving grid-independent convergence rates and that the accuracy is quadratic (decreases by a factor of 4 for each mesh refinement).

For some machines the largest problem solved in this example may exhaust the available memory. In this case, reduce the value of the parameter `level_up` accordingly.

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_pde_ell_mg_ex03

  ! Example Program Text for nag_pde_ell_mg
  ! NAG f190, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_pde_ell_mg, ONLY : nag_pde_ell_mg_sol
  USE nag_math_constants, ONLY : nag_pi
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EPSILON, KIND, MAX, MIN, PRECISION, REAL, SIN, SQRT, SUM
  ! .. Parameters ..
  INTEGER, PARAMETER :: level_low = 3
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  INTEGER, PARAMETER :: level_up = MIN(8,PRECISION(0.0_wp))
  INTEGER, PARAMETER :: maxsize = (2**level_up+1)**2
  REAL (wp), PARAMETER :: one = 1.0_wp
  REAL (wp), PARAMETER :: two = 2.0_wp
  REAL (wp), PARAMETER :: zero = 0.0_wp
  REAL (wp), PARAMETER :: mone = -one
  ! .. Local Scalars ..
  INTEGER :: i, ix, iy, j, k1, k2, level, num_iter, nx, nxy, ny
  REAL (wp) :: acc, hx, hx2, hy, hy2, mu, pi, resid_norm, rms_err, tpi2, &
   xj
  ! .. Local Arrays ..
  REAL (wp) :: a(maxsize,7), rhs(maxsize), sol(maxsize), u(maxsize)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_pde_ell_mg_ex03'

  pi = nag_pi(zero)
  tpi2 = two*pi*pi

  ! Do-loop on the level

  DO level = level_low, level_up
    nx = 2**level + 1
```

```
ny = nx
nxy = nx*ny
acc = MAX(1.0E-6_wp,SQRT(nx*EPSILON(zero)))

! Set-up operator, right-hand side and
! initial guess

hx = one/REAL(nx-1,kind=wp)
hx2 = hx*hx
hy = one/REAL(ny-1,kind=wp)
hy2 = hy*hy
mu = two/hx2 + two/hy2

a(1:nxy,1) = mone/hy2
a(1:nxy,2) = zero
a(1:nxy,3) = mone/hx2
a(1:nxy,4) = mu
a(1:nxy,5) = mone/hx2
a(1:nxy,6) = zero
a(1:nxy,7) = mone/hy2

u(1:nxy) = zero

! Exact solution "sol"
DO j = 1, ny
  xj = REAL(j-1,kind=wp)*hy
  k2 = j*nx
  k1 = k2 + 1 - nx
  sol(k1:k2) = SIN(pi*xj)*SIN(pi*hx*(/(REAL(i-1,kind=wp),i=1,nx)/))
END DO

rhs(1:nxy) = tpi2*sol(1:nxy)

! Correction for the boundary conditions
! Horizontal boundaries
DO i = 1, nx
  ! Boundary condition Y = 0
  ix = i
  u(ix) = sol(ix)
  rhs(ix) = mu*sol(ix)
  a(ix,1:7:2) = zero
  ! Boundary condition Y = 1
  ix = i + (ny-1)*nx
  u(ix) = sol(ix)
  rhs(ix) = mu*sol(ix)
  a(ix,1:7:2) = zero
END DO
! Vertical boundaries
DO j = 1, ny
  ! Boundary condition X = 0
  iy = (j-1)*nx + 1
  u(iy) = sol(iy)
  rhs(iy) = mu*sol(iy)
  a(iy,1:7:2) = zero
  ! Boundary condition X = 1
  iy = j*nx
  u(iy) = sol(iy)
  rhs(iy) = mu*sol(iy)
  a(iy,1:7:2) = zero
END DO
! Solve the equation
CALL nag_pde_ell_mg_sol(nx,ny,a(1:nxy,1:7),rhs(1:nxy),u(1:nxy), &
```

```
         resid_norm=resid_norm,acc=acc,num_iter=num_iter)
        WRITE (nag_std_out,*) ' '
        WRITE (nag_std_out,999) ' Level ', level
        WRITE (nag_std_out,999) ' Order of the matrix ', nxy
        WRITE (nag_std_out,998) ' Accuracy ', acc
        WRITE (nag_std_out,999) ' Number of iteration ', num_iter
        WRITE (nag_std_out,998) ' Residual norm ', resid_norm

        rms_err = SUM((u(:nxy)-sol(:nxy))**2)
        rms_err = SQRT(rms_err/REAL(nxy,kind=wp))

        WRITE (nag_std_out,998) ' RMS Error ', rms_err
      END DO

999   FORMAT (1X,A,I7)
998   FORMAT (1X,A,1P,E10.4)

    END PROGRAM nag_pde_ell_mg_ex03
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_pde_ell_mg_ex03

  Level       3
  Order of the matrix      81
  Accuracy 1.0000E-06
  Number of iteration       6
  Residual norm 2.6324E-07
  RMS Error 5.7559E-03

  Level       4
  Order of the matrix     289
  Accuracy 1.0000E-06
  Number of iteration       7
  Residual norm 1.9001E-07
  RMS Error 1.5148E-03

  Level       5
  Order of the matrix    1089
  Accuracy 1.0000E-06
  Number of iteration       8
  Residual norm 8.1103E-08
  RMS Error 3.8961E-04

  Level       6
  Order of the matrix    4225
  Accuracy 1.0000E-06
  Number of iteration       8
  Residual norm 2.6200E-07
  RMS Error 9.8866E-05

  Level       7
  Order of the matrix   16641
  Accuracy 1.0000E-06
  Number of iteration       8
  Residual norm 6.5546E-07
  RMS Error 2.4906E-05
```

```
Level       8
Order of the matrix   66049
Accuracy 1.0000E-06
Number of iteration       9
Residual norm 1.0175E-07
RMS Error 6.2506E-06
```

References

[1] McCarthy G J (1983) Investigation into the multigrid code MGD1 *Report AERE-R 10889* Harwell

[2] Wesseling P (1982) MGD1 – A robust and efficient multigrid method *Multigrid Methods. Lecture Notes in Mathematics* **960** Springer-Verlag 614–630

[3] Wesseling P (1982) Theoretical aspects of a multigrid method *SIAM J. Sci. Statist. Comput.* **3** 387–407