

# NAG Toolbox for MATLAB

## d03ed

### 1 Purpose

d03ed solves seven-diagonal systems of linear equations which arise from the discretization of an elliptic partial differential equation on a rectangular region. This function uses a multigrid technique.

### 2 Syntax

```
[a, rhs, ub, us, u, numit, ifail] = d03ed(ngx, ngy, a, rhs, ub, maxit,
acc, iout)
```

### 3 Description

d03ed solves, by multigrid iteration, the seven-point scheme

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

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

$$\alpha(x, y) U_{xx} + \beta(x, y) U_{xy} + \gamma(x, y) U_{yy} + \delta(x, y) U_x + \epsilon(x, y) U_y + \phi(x, y) U = \psi(x, y)$$

and its boundary conditions, defined on a rectangular region. This we write in matrix form as

$$Au = f.$$

The algorithm is described in separate reports by Wesseling 1982a, Wesseling 1982b and McCarthy 1983.

Systems of linear equations, matching the seven-point stencil defined above, are solved by a multigrid iteration. An initial estimate of the solution must be provided by you. A zero guess may be supplied if no better approximation is available.

A ‘smoother’ based on incomplete Crout 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 prolonged (mapped onto successively finer grids). When the finest cycle is reached, the approximation to the solution is corrected. The cycle is repeated for **maxit** iterations or until the required accuracy, **acc**, is reached.

d03ed will automatically determine the number  $l$  of possible coarse grids, ‘levels’ of the multigrid scheme, for a particular problem. In other words, d03ed determines the maximum integer  $l$  so that  $n_x$  and  $n_y$  can be expressed in the form

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

It should be noted that the rate of convergence improves significantly with the number of levels used (see McCarthy 1983), 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 the integer  $l$  should be at least 2.

d03ed has been found to be robust in application, but being an iterative method the problem of divergence can arise. For a strictly diagonally dominant matrix  $A$

$$|A_{ij}^4| > \sum_{k \neq 4} |A_{ij}^k|, \quad i = 1, 2, \dots, n_x; j = 1, 2, \dots, n_y$$

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 function.

## 4 References

McCarthy G J 1983 Investigation into the multigrid code MGD1 *Report AERE-R 10889* Harwell

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

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

## 5 Parameters

### 5.1 Compulsory Input Parameters

1: **ngx** – int32 scalar

The number of interior grid points in the  $x$ -direction,  $n_x$ . **ngx** – 1 should preferably be divisible by as high a power of 2 as possible.

*Constraint:* **ngx**  $\geq 3$ .

2: **ngy** – int32 scalar

The number of interior grid points in the  $y$ -direction,  $n_y$ . **ngy** – 1 should preferably be divisible by as high a power of 2 as possible.

*Constraint:* **ngy**  $\geq 3$ .

3: **a(lda,7)** – double array

**lda**, the first dimension of the array, must be at least .

**a**( $i + (j - 1) \times \text{ngx}$ ,  $k$ ) must be set to  $\mathbf{a}_{ij}^k$ , for  $i = 1, 2, \dots, \text{ngx}$  and  $j = 1, 2, \dots, \text{ngy}$  and  $k = 1, 2, \dots, 7$ .

4: **rhs(lda)** – double array

**lda**, the first dimension of the array, must be at least .

**rhs**( $i + (j - 1) \times \text{ngx}$ ) must be set to  $f_{ij}$ , for  $i = 1, 2, \dots, \text{ngx}$  and  $j = 1, 2, \dots, \text{ngy}$ .

5: **ub(ngx  $\times$  ngy)** – double array

**ub**( $i + (j - 1) \times \text{ngx}$ ) must be set to the initial estimate for the solution  $u_{ij}$ .

6: **maxit** – int32 scalar

The maximum permitted number of multigrid iterations. If **maxit** = 0, no multigrid iterations are performed, but the coarse-grid approximations and incomplete Crout decompositions are computed, and may be output if **iout** is set accordingly.

*Constraint:* **maxit**  $\geq 0$ .

7: **acc** – double scalar

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

$$\|r\|_2 = \sqrt{\sum_{k=1}^{\text{ngx} \times \text{ngy}} (r_k)^2}$$

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

If on entry  $\mathbf{acc} = 0.0$ , then the *machine precision* is used as a default value for the tolerance; if  $\mathbf{acc} > 0.0$ , but  $\mathbf{acc}$  is less than the *machine precision*, then the function will stop when the residual 2-norm is less than the *machine precision* and **ifail** will be set to 4.

Constraint:  $\mathbf{acc} \geq 0.0$ .

8: **iout – int32 scalar**

Controls the output of printed information to the advisory message unit as returned by x04ab:

**iout** = 0

No output.

**iout** = 1

The solution  $u_{ij}$ , for  $i = 1, 2, \dots, \mathbf{ngx}$  and  $j = 1, 2, \dots, \mathbf{ngy}$ .

**iout** = 2

The residual 2-norm after each iteration, with the reduction factor over the previous iteration.

**iout** = 3

As for **iout** = 1 and **iout** = 2.

**iout** = 4

As for **iout** = 3, plus the final residual (as returned in **ub**).

**iout** = 5

As for **iout** = 4, plus the initial elements of **a** and **rhs**.

**iout** = 6

As for **iout** = 5, plus the Galerkin coarse grid approximations.

**iout** = 7

As for **iout** = 6, plus the incomplete Crout decompositions.

**iout** = 8

As for **iout** = 7, plus the residual after each iteration.

The elements  $\mathbf{a}(p, k)$ , the Galerkin coarse grid approximations and the incomplete Crout decompositions are output in the format:

Y-index =  $j$

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

where  $p = i + (j - 1) \times \mathbf{ngx}$ ,  $i = 1, 2, \dots, \mathbf{ngx}$  and  $j = 1, 2, \dots, \mathbf{ngy}$ .

The vectors  $\mathbf{u}(p)$ ,  $\mathbf{ub}(p)$ ,  $\mathbf{rhs}(p)$  are output in matrix form with **ngy** rows and **ngx** columns. Where **ngx** > 10, the **ngx** values for a given  $j$  value are produced in rows of 10. Values of **iout** > 4 may yield considerable amounts of output.

Constraint:  $0 \leq \mathbf{iout} \leq 8$ .

## 5.2 Optional Input Parameters

None.

## 5.3 Input Parameters Omitted from the MATLAB Interface

lda

## 5.4 Output Parameters

1: **a(lda,7)** – double array

Is overwritten.

2: **rhs(lda)** – double array

The first  $\mathbf{ngx} \times \mathbf{ngy}$  elements are unchanged and the rest of the array is used as workspace.

3: **ub(ngx × ngy)** – double array

The corresponding component of the residual  $r = f - \mathbf{au}$ .

4: **us(lda)** – double array

The residual 2-norm, stored in element **us**(1).

5: **u(lda)** – double array

The computed solution  $u_{ij}$  is returned in **u**( $i + (j - 1) \times \mathbf{ngx}$ ), for  $i = 1, 2, \dots, \mathbf{ngx}$  and  $j = 1, 2, \dots, \mathbf{ngy}$ .

6: **numit** – int32 scalar

The number of iterations performed.

7: **ifail** – int32 scalar

0 unless the function detects an error (see Section 6).

## 6 Error Indicators and Warnings

Errors or warnings detected by the function:

**ifail** = 1

On entry, **ngx** < 3,  
or **ngy** < 3,  
or **lda** is too small,  
or **acc** < 0.0,  
or **maxit** < 0,  
or **iout** < 0,  
or **iout** > 8.

**ifail** = 2

**maxit** iterations have been performed with the residual 2-norm decreasing at each iteration but the residual 2-norm has not been reduced to less than the specified tolerance (see **acc**). Examine the progress of the iteration by setting **iout** ≥ 2.

**ifail** = 3

As for **ifail** = 2, except that 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  $A$ .

**ifail** = 4

On entry, **acc** is less than the *machine precision*. The function terminated because the residual norm is less than the *machine precision*.

## 7 Accuracy

See **acc** (Section 5).

## 8 Further Comments

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

## 9 Example

```

ngx = int32(9);
ngy = int32(9);
a = zeros(133,7);
rhs = zeros(133, 1);
ub = zeros(81, 1);
maxit = int32(15);
acc = 0.0001;
iout = int32(0);

hx = 1/double(ngx+1);
hy = 1/double(ngy+1);
alpha = 1.7;

% Set up operator, right-hand side and initial guess for
% step-lengths hx and hy
for i = 1:ngy
    for j = 1:ngx
        k = (j-1)*ngx + i;
        a(k,1) = 1.0 - 0.5*alpha;
        a(k,2) = 0.5*alpha;
        a(k,3) = 1.0 - 0.5*alpha;
        a(k,4) = -4.0 + alpha;
        a(k,5) = 1.0 - 0.5*alpha;
        a(k,6) = 0.5*alpha;
        a(k,7) = 1.0 - 0.5*alpha;
        rhs(k) = -4.0*hx*hy;
    end
end

% Correction for the boundary conditions
% Horizontal boundaries --
for i = 2:ngx-1
    % Boundary condition on y=0 -- u=0
    ix = i;
    a(ix,1) = 0.0;
    a(ix,2) = 0.0;
    % Boundary condition on y=1 -- u=0
    ix = i + (ngy-1)*ngx;
    a(ix,6) = 0.0;
    a(ix,7) = 0.0;
end
% Vertical boundaries --
for j = 2:ngy-1
    % Boundary condition on x=0 -- u=0
    iy = (j-1)*ngx + 1;
    a(iy,3) = 0.0;
    a(iy,6) = 0.0;
    % Boundary condition on x=1 -- u=1
    iy = j*ngx;
    rhs(iy) = rhs(iy) - a(iy,5) - a(iy,2);
    a(iy,2) = 0.0;
    a(iy,5) = 0.0;
end

```

```
end

% Now the four corners --
% Bottom left corner
k = 1;
a(k,1) = 0.0;
a(k,2) = 0.0;
a(k,3) = 0.0;
a(k,6) = 0.0;
% Top left corner
k = 1 + (ngy-1)*ngx;
a(k,3) = 0.0;
a(k,6) = 0.0;
a(k,7) = 0.0;
% Bottom right corner
% Use average value at discontinuity ( = 0.5 )
k = ngx;
rhs(k) = rhs(k) - a(k,2)*0.5 - a(k,5);
a(k,1) = 0.0;
a(k,2) = 0.0;
a(k,5) = 0.0;
% Top right corner
k = ngx*ngy;
rhs(k) = rhs(k) - a(k,2) - a(k,5);
a(k,2) = 0.0;
a(k,5) = 0.0;
a(k,6) = 0.0;
a(k,7) = 0.0;

[aOut, rhsOut, ubOut, us, u, numit, ifail] = ...
    d03ed(ngx, ngy, a, rhs, ub, maxit, acc, iout)

aOut =
    array elided
rhsOut =
    array elided
ubOut =
    array elided
us =
    array elided
u =
    array elided
numit =
    4
ifail =
    0
```