NAG Toolbox

nag_sparse_real_gen_solve_bdilu (f11dg)

1 Purpose

nag_sparse_real_gen_solve_bdilu (f11dg) solves a real sparse nonsymmetric system of linear equations, represented in coordinate storage format, using a restarted generalized minimal residual (RGMRES), conjugate gradient squared (CGS), stabilized bi-conjugate gradient (Bi-CGSTAB), or transpose-free quasi-minimal residual (TFQMR) method, with block Jacobi or additive Schwarz preconditioning.

2 Syntax

```
[x, rnorm, itn, ifail] = nag_sparse_real_gen_solve_bdilu(method, nnz, a, irow, icol, nb, istb, indb, ipivp, ipivq, istr, idiag, b, m, tol, maxitn, x, 'n', n, 'la', la, 'lindb', lindb)
[x, rnorm, itn, ifail] = f11dg(method, nnz, a, irow, icol, nb, istb, indb, ipivp, ipivq, istr, idiag, b, m, tol, maxitn, x, 'n', n, 'la', la, 'lindb', lindb)
```

3 Description

nag_sparse_real_gen_solve_bdilu (f11dg) solves a real sparse nonsymmetric linear system of equations:

$$Ax = b,$$

using a preconditioned RGMRES (see Saad and Schultz (1986)), CGS (see Sonneveld (1989)), Bi-CGSTAB(ℓ) (see Van der Vorst (1989) and Sleijpen and Fokkema (1993)), or TFQMR (see Freund and Nachtigal (1991) and Freund (1993)) method.

nag_sparse_real_gen_solve_bdilu (f11dg) uses the incomplete (possibly overlapping) block LU factorization determined by nag_sparse_real_gen_precon_bdilu (f11df) as the preconditioning matrix. A call to nag_sparse_real_gen_solve_bdilu (f11dg) must always be preceded by a call to nag_sparse_real_gen_precon_bdilu (f11df). Alternative preconditioners for the same storage scheme are available by calling nag_sparse_real_gen_solve_ilu (f11dc) or nag_sparse_real_gen_solve_jacssor (f11de).

The matrix A, and the preconditioning matrix M, are represented in coordinate storage (CS) format (see Section 2.1.1 in the F11 Chapter Introduction) in the arrays **a**, **irow** and **icol**, as returned from nag_sparse_real_gen_precon_bdilu (f11df). The array **a** holds the nonzero entries in these matrices, while **irow** and **icol** hold the corresponding row and column indices.

nag_sparse_real_gen_solve_bdilu (f11dg) is a Black Box function which calls nag_sparse_real_gen_basic_setup (f11bd), nag_sparse_real_gen_basic_solver (f11be) and nag_sparse_real_gen_basic_diag (f11bf). If you wish to use an alternative storage scheme, preconditioner, or termination criterion, or require additional diagnostic information, you should call these underlying functions directly.

4 References

Freund R W (1993) A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems *SIAM J. Sci. Comput.* **14** 470–482

Freund R W and Nachtigal N (1991) QMR: a Quasi-Minimal Residual Method for Non-Hermitian Linear Systems *Numer. Math.* **60** 315–339

Saad Y and Schultz M (1986) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **7** 856–869

Salvini S A and Shaw G J (1996) An evaluation of new NAG Library solvers for large sparse unsymmetric linear systems *NAG Technical Report TR2/96*

Sleijpen G L G and Fokkema D R (1993) BiCGSTAB(ℓ) for linear equations involving matrices with complex spectrum *ETNA* **1** 11–32

Sonneveld P (1989) CGS, a fast Lanczos-type solver for nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **10** 36–52

Van der Vorst H (1989) Bi-CGSTAB, a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **13** 631–644

5 Parameters

5.1 Compulsory Input Parameters

1: **method** – CHARACTER(*)

Specifies the iterative method to be used.

method = 'RGMRES'
Restarted generalized minimum residual method.

method = 'CGS'
Conjugate gradient squared method.

method = 'BICGSTAB'
Bi-conjugate gradient stabilized (ℓ) method.

method = 'TFQMR'
Transpose-free quasi-minimal residual method.

Constraint: **method** = 'RGMRES', 'CGS', 'BICGSTAB' or 'TFQMR'.

2: **nz** – INTEGER
3: **a**(**la**) – REAL (KIND=nag_wp) array
4: **irow**(**la**) – INTEGER array
5: **icol**(**la**) – INTEGER array
6: **nb** – INTEGER
7: **istb**(**nb** + **1**) – INTEGER array
8: **indb**(**lindb**) – INTEGER array
9: **ipivp**(**lindb**) – INTEGER array
10: **ipivq**(**lindb**) – INTEGER array
11: **istr**(**lindb** + **1**) – INTEGER array
12: **idiag**(**lindb**) – INTEGER array

The values returned in arrays **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** by a previous call to nag_sparse_real_gen_precon_bdilu (f11df).

The arrays **istb**, **indb** and **a** together with the the scalars **n**, **nnz**, **la**, **nb** and **lindb** must be the same values that were supplied in the preceding call to nag_sparse_real_gen_precon_bdilu (f11df).

13: **b**(**n**) – REAL (KIND=nag_wp) array

The right-hand side vector b.

14: **m** – INTEGER

If **method** = 'RGMRES', **m** is the dimension of the restart subspace.

If **method** = 'BICGSTAB', **m** is the order ℓ of the polynomial Bi-CGSTAB method. Otherwise, **m** is not referenced.

Constraints:

if **method** = 'RGMRES', $0 < \mathbf{m} \leq \min(\mathbf{n}, 50)$;
if **method** = 'BICGSTAB', $0 < \mathbf{m} \leq \min(\mathbf{n}, 10)$.

15: **tol** – REAL (KIND=nag_wp)

The required tolerance. Let x_k denote the approximate solution at iteration k, and r_k the corresponding residual. The algorithm is considered to have converged at iteration k if

$$\|r_k\|_\infty \leq \tau \times \left(\|b\|_\infty + \|A\|_\infty \|x_k\|_\infty\right).$$

If **tol** ≤ 0.0, $\tau = \max\sqrt{\epsilon}, \sqrt{n\epsilon}$ is used, where ϵ is the ***machine precision***. Otherwise $\tau = \max(\textbf{tol}, 10\epsilon, \sqrt{n\epsilon})$ is used.

Constraint: **tol** < 1.0.

16: **maxitn** – INTEGER

The maximum number of iterations allowed.

Constraint: **maxitn** ≥ 1.

17: **x**(**n**) – REAL (KIND=nag_wp) array

An initial approximation to the solution vector x.

5.2 Optional Input Parameters

1: **n** – INTEGER
2: **la** – INTEGER
3: **lindb** – INTEGER

Default: the dimension of the arrays **b**, **x**, **a**, **irow**, **icol**. (An error is raised if these dimensions are not equal.)For **lindb**, the dimension of the arrays **ipivp**, **ipivq**, **idiag**, **indb**. (An error is raised if these dimensions are not equal.)

The values returned in arrays **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** by a previous call to nag_sparse_real_gen_precon_bdilu (f11df).

The arrays **istb**, **indb** and **a** together with the the scalars **n**, **nnz**, **la**, **nb** and **lindb** must be the same values that were supplied in the preceding call to nag_sparse_real_gen_precon_bdilu (f11df).

5.3 Output Parameters

1: **x**(**n**) – REAL (KIND=nag_wp) array

An improved approximation to the solution vector x.

2: **rnorm** – REAL (KIND=nag_wp)

The final value of the residual norm $\|r_k\|_\infty$, where k is the output value of **itn**.

3: **itn** – INTEGER

The number of iterations carried out.

4: **ifail** – INTEGER

ifail $= 0$ unless the function detects an error (see Section 5).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

ifail $= 1$

Constraint: $1 \leq$ **nb** $\leq$ **n**.

Constraint: $\mathbf{istb}(b+1) > \mathbf{istb}(b)$, for $b = 1, 2, \ldots, \mathbf{nb}$.

Constraint: if $\mathbf{method} =$ 'RGMRES', $1 \leq \mathbf{m} \leq \min(\mathbf{n}, \langle value \rangle)$.

Constraint: $\mathbf{istb}(1) \geq 1$.

Constraint: $\mathbf{la} \geq 2 \times \mathbf{nnz}$.

Constraint: $\mathbf{lindb} \geq \mathbf{istb}(\mathbf{nb}+1) - 1$.

Constraint: $1 \leq \mathbf{indb}(m) \leq \mathbf{n}$, for $m = 1, 2, \ldots, \mathbf{istb}(\mathbf{nb}+1) - 1$

Constraint: $\mathbf{maxitn} \geq 1$.

Constraint: $\mathbf{method} =$ 'RGMRES', 'CGS' or 'BICGSTAB'.

Constraint: $\mathbf{nnz} \leq \mathbf{n}^2$.

Constraint: $\mathbf{nnz} \geq 1$.

Constraint: $\mathbf{n} \geq 1$.

Constraint: $\mathbf{tol} < 1.0$.

$lwork$ is too small.

ifail $= 2$

Constraint: $1 \leq \mathbf{icol}(i) \leq \mathbf{n}$, for $i = 1, 2, \ldots, \mathbf{nnz}$.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_real_gen_precon_bdilu (f11df) and nag_sparse_real_gen_solve_bdilu (f11dg).

Constraint: $1 \leq \mathbf{irow}(i) \leq \mathbf{n}$, for $i = 1, 2, \ldots, \mathbf{nnz}$.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_real_gen_precon_bdilu (f11df) and nag_sparse_real_gen_solve_bdilu (f11dg).

On entry, element $\langle value \rangle$ of **a** was out of order.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_real_gen_precon_bdilu (f11df) and nag_sparse_real_gen_solve_bdilu (f11dg).

On entry, location $\langle value \rangle$ of $(\mathbf{irow}, \mathbf{icol})$ was a duplicate.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_real_gen_precon_bdilu (f11df) and nag_sparse_real_gen_solve_bdilu (f11dg).

ifail $= 3$

The CS representation of the preconditioner is invalid.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_real_gen_precon_bdilu (f11df) and nag_sparse_real_gen_solve_bdilu (f11dg).

ifail $= 4$

The required accuracy could not be obtained. However a reasonable accuracy may have been achieved. You should check the output value of **rnorm** for acceptability. This error code usually implies that your problem has been fully and satisfactorily solved to within or close to the accuracy available on your system. Further iterations are unlikely to improve on this situation.

ifail $= 5$

The solution has not converged after $\langle value \rangle$ iterations.

ifail $= 6$

Algorithmic breakdown. A solution is returned, although it is possible that it is completely inaccurate.

ifail $= -99$

An unexpected error has been triggered by this routine. Please contact NAG.

ifail $= -399$

Your licence key may have expired or may not have been installed correctly.

ifail $= -999$

Dynamic memory allocation failed.

7 Accuracy

On successful termination, the final residual $r_k = b - Ax_k$, where $k =$ **itn**, satisfies the termination criterion

$$\|r_k\|_\infty \leq \tau \times \left(\|b\|_\infty + \|A\|_\infty \|x_k\|_\infty\right).$$

The value of the final residual norm is returned in **rnorm**.

8 Further Comments

The time taken by nag_sparse_real_gen_solve_bdilu (f11dg) for each iteration is roughly proportional to the value of **nnzc** returned from the preceding call to nag_sparse_real_gen_precon_bdilu (f11df).

The number of iterations required to achieve a prescribed accuracy cannot be easily determined *a priori*, as it can depend dramatically on the conditioning and spectrum of the preconditioned coefficient matrix $\bar{A} = M^{-1}A$.

Some illustrations of the application of nag_sparse_real_gen_solve_bdilu (f11dg) to linear systems arising from the discretization of two-dimensional elliptic partial differential equations, and to random-valued randomly structured linear systems, can be found in Salvini and Shaw (1996).

9 Example

This example program reads in a sparse matrix A and a vector b. It calls nag_sparse_real_gen_precon_bdilu (f11df), with the array **lfill** $= 0$ and the array **dtol** $= 0.0$, to compute an overlapping incomplete LU factorization. This is then used as an additive Schwarz preconditioner on a call to nag_sparse_real_gen_solve_bdilu (f11dg) which uses the Bi-CGSTAB method to solve $Ax = b$.

9.1 Program Text

```
function f11dg_example

fprintf('f11dg example results\n\n');

n   = nag_int(9);
nz  = nag_int(33);
a    = zeros(20*nz, 1);
irow = zeros(20*nz, 1, nag_int_name);
icol = zeros(20*nz, 1, nag_int_name);
a(1:nz)    = [64; -20; -20; -12;  64; -20; -20; -12;  64; -20; -12;
              64; -20; -20; -12; -12;  64; -20; -20; -12; -12;  64;
             -20; -12;  64; -20; -12; -12;  64; -20; -12; -12;  64];
irow(1:nz) = [ 1;   1;   1;   2;   2;   2;   2;   3;   3;   3;   4;
               4;   4;   4;   5;   5;   5;   5;   5;   6;   6;   6;
               6;   7;   7;   7;   8;   8;   8;   8;   9;   9;   9];
icol(1:nz) = [ 1;   2;   4;   1;   2;   3;   5;   2;   3;   6;   1;
               4;   5;   7;   2;   4;   5;   6;   8;   3;   5;   6;
               9;   4;   7;   8;   5;   7;   8;   9;   6;   8;   9];

% 3 Blocks
nb     = nag_int(3);
nover  = 1;
lfill  = [nag_int(0); 0; 0];
dtol   = [  0;   0;   0];
pstrat = {'n'; 'n'; 'n'};
milu   = {'n'; 'n'; 'n'};
```

```
% Define diagonal block indices.
% In this example use blocks of mb consecutive rows and initialise
% assuming no overlap.
mb    = idivide(n+nb-1, nb);
istb  = zeros(nb+1, 1, nag_int_name);
indb  = zeros(3*n, 1, nag_int_name);
ipivp = zeros(3*n, 1, nag_int_name);
ipivq = zeros(3*n, 1, nag_int_name);
istb(1:nb) = [1:mb:nb*mb];
istb(nb+1) = n+1;
indb(1:n)  = [1:n];

% Modify indb and istb to account for overlap.
[istb, indb, ifail] = f11df_overlap(n, nz, irow, icol, nb, ...
                                    istb, indb, 3*n, nover);
if (ifail == -999)
  error('indb is too small, size of indb = %d', numel(indb));
end

% Calculate Factorisation
[a, irow, icol, ipivp, ipivq, istr, idiag, nnzc, npivm, ifail] = ...
f11df( ...
       n, nz, a, irow, icol, istb, indb, ...
       lfill, dtol, milu, ipivp, ipivq, 'pstrat', pstrat);

% RHS b and initial guess x
b = 100*ones(n,1);
x = zeros(n, 1);

% Solver algorithmic parameters
method = 'bicgstab';
m      = nag_int(2);
tol    = 1e-6;
maxitn = nag_int(100);

% Solve Ax = b using f11dg
[x, rnorm, itn, ifail] = f11dg( ...
                              method, nz, a, irow, icol, nb, istb, indb, ...
                              ipivp, ipivq, istr, idiag, b, m, tol, ...
                              maxitn, x);

fprintf('Converged in %d iterations\n', itn);
fprintf('Final redidual norm = %16.3d\n\n', rnorm);
disp('Solution');
disp(x);

function [istb, indb, ifail] = f11df_overlap(n, nz, irow, icol, nb, ...
                                             istb, indb, lindb, nover)

  ifail = 0;

  % This function takes a set of row indices indb defining the diagonal
  % blocks to be used in f11df to define a block Jacobi or additive Schwarz
  % preconditioner, and expands them to allow for nover levels of overlap.
  % The pointer array istb is also updated accordingly, so that the returned
  % values of istb and indb can be passed to f11df to define overlapping
  % diagonal blocks.
  iwork = zeros(3*n+1, 1, nag_int_name);

  % Find the number of non-zero elements in each row of the matrix A, and
  % the start address of each row. Store the start addresses in
  % iwork(n+1,...,2*n+1).
  for k=1:nz
    iwork(irow(k)) = iwork(irow(k)) + 1;
  end
  iwork(n+1) = 1;
  for i = 1:n
    iwork(n+i+1) = iwork(n+i) + iwork(i);
  end
```

```
  % Loop over blocks
  for k=1:nb
    % Initialize marker array.
    iwork(1:n) = 0;

    % Mark the rows already in block k in the workspace array.
    for l = istb(k):istb(k+1)-1
      iwork(indb(l)) = 1;
    end

    % Loop over levels of overlap.
    for iover=1:nover
      % Initialize counter of new row indices to be added.
      ind = 0;

      % Loop over the rows currently in the diagonal block.
      for l = istb(k):istb(k+1)-1
        row = indb(l);

        % Loop over non-zero elements in row
        for i = iwork(n+row):iwork(n+row+1)-1

          % If the column index of the non-zero element is not in the
          % existing set for this block, store it to be added later, and
          % mark it in the marker array.
          if (iwork(icol(i))==0)
            iwork(icol(i)) = 1;
            ind = ind + 1;
            iwork(2*n+1+ind) = icol(i);
          end
        end
      end

      % Shift the indices in indb and add the new entries for block k.
      % Change istb accordingly.
      nadd = ind;
      if (istb(nb+1)+nadd-1>lindb) Then
        ifail = -999;
        return;
      end

      for i = istb(nb+1) - 1:-1:istb(k+1)
        indb(i+nadd) = indb(i);
      end
      n21 = 2*n + 1;
      ik = istb(k+1) - 1;
      indb(ik+1:ik+nadd) = iwork(n21+1:n21+nadd);
      istb(k+1:nb+1) = istb(k+1:nb+1) + nadd;
    end
  end
```

9.2 Program Results

```
     f11dg example results

Converged in 4 iterations
Final redidual norm =        1.106e-05

Solution
    5.2603
    5.9165
    4.1131
    5.9165
    6.6636
    4.6119
    4.1131
    4.6119
    3.2919
```