NAG Library Function Document

nag_sparse_nherm_precon_bdilu_solve (f11duc)

1 Purpose

nag_sparse_nherm_precon_bdilu_solve (f11duc) solves a complex sparse non-Hermitian 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 Specification

```
#include <nag.h>
#include <nagf11.h>
```

```
void nag_sparse_nherm_precon_bdilu_solve (Nag_SparseNsym_Method method,
     Integer n, Integer nnz, const Complex a[], Integer la,
     const Integer irow[], const Integer icol[], Integer nb,
     const Integer istb[], const Integer indb[], Integer lindb,
     const Integer ipivp[], const Integer ipivq[], const Integer istr[],
     const Integer idiag[], const Complex b[], Integer m, double tol,
     Integer maxitn, Complex x[], double *rnorm, Integer *itn,
     NagError *fail)
```

3 Description

nag_sparse_nherm_precon_bdilu_solve (f11duc) solves a complex sparse non-Hermitian 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_nherm_precon_bdilu_solve (f11duc) uses the incomplete (possibly overlapping) block LU factorization determined by nag_sparse_nherm_precon_bdilu (f11dtc) as the preconditioning matrix. A call to nag_sparse_nherm_precon_bdilu_solve (f11duc) must always be preceded by a call to nag_sparse_nherm_precon_bdilu (f11dtc). Alternative preconditioners for the same storage scheme are available by calling nag_sparse_nherm_fac_sol (f11dqc) or nag_sparse_nherm_sol (f11dsc).

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_nherm_precon_bdilu (f11dtc). The array **a** holds the nonzero entries in these matrices, while **irow** and **icol** hold the corresponding row and column indices.

nag_sparse_nherm_precon_bdilu_solve (f11duc) is a Black Box function which calls nag_sparse_nherm_basic_setup (f11brc), nag_sparse_nherm_basic_solver (f11bsc) and nag_sparse_nherm_basic_diagnostic (f11btc). 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

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 Arguments

1: **method** – Nag_SparseNsym_Method *Input*

On entry: specifies the iterative method to be used.

method = Nag_SparseNsym_RGMRES
Restarted generalized minimum residual method.

method = Nag_SparseNsym_CGS
Conjugate gradient squared method.

method = Nag_SparseNsym_BiCGSTAB
Bi-conjugate gradient stabilized (ℓ) method.

method = Nag_SparseNsym_TFQMR
Transpose-free quasi-minimal residual method.

Constraint: **method** = Nag_SparseNsym_RGMRES, Nag_SparseNsym_CGS, Nag_SparseNsym_BiCGSTAB or Nag_SparseNsym_TFQMR.

2: **n** – Integer *Input*
3: **nnz** – Integer *Input*
4: **a**[**la**] – const Complex *Input*
5: **la** – Integer *Input*
6: **irow**[**la**] – const Integer *Input*
7: **icol**[**la**] – const Integer *Input*
8: **nb** – Integer *Input*
9: **istb**[**nb** + **1**] – const Integer *Input*
10: **indb**[**lindb**] – const Integer *Input*
11: **lindb** – Integer *Input*
12: **ipivp**[**lindb**] – const Integer *Input*
13: **ipivq**[**lindb**] – const Integer *Input*
14: **istr**[**lindb** + **1**] – const Integer *Input*
15: **idiag**[**lindb**] – const Integer *Input*

On entry: the values returned in arrays **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** by a previous call to nag_sparse_nherm_precon_bdilu (f11dtc).

The arrays **istb**, **indb** and **a** together with the scalars **n**, **nnz**, **la**, **nb** and **lindb** must be the same values that were supplied in the preceding call to nag_sparse_nherm_precon_bdilu (f11dtc).

On entry: the values returned in arrays **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** by a previous call to nag_sparse_nherm_precon_bdilu (f11dtc).

The arrays **istb**, **indb** and the scalars **nb** and **lindb** must be the same values that were supplied in the preceding call to nag_sparse_nherm_precon_bdilu (f11dtc).

16: **b**[**n**] – const Complex *Input*

On entry: the right-hand side vector b.

17: **m** – Integer *Input*

On entry: if **method** $=$ Nag_SparseNsym_RGMRES, **m** is the dimension of the restart subspace.

If **method** $=$ Nag_SparseNsym_BiCGSTAB, **m** is the order ℓ of the polynomial Bi-CGSTAB method.

Otherwise, **m** is not referenced.

Constraints:

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

18: **tol** – double *Input*

On entry: 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 $\mathbf{tol} \leq 0.0$, $\tau = \max(\sqrt{\epsilon}, \sqrt{n}\epsilon)$ is used, where ϵ is the ***machine precision***. Otherwise $\tau = \max(\mathbf{tol}, 10\epsilon, \sqrt{n}\epsilon)$ is used.

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

19: **maxitn** – Integer *Input*

On entry: the maximum number of iterations allowed.

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

20: **x**[**n**] – Complex *Input/Output*

On entry: an initial approximation to the solution vector x.

On exit: an improved approximation to the solution vector x.

21: **rnorm** – double * *Output*

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

22: **itn** – Integer * *Output*

On exit: the number of iterations carried out.

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

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

6 Error Indicators and Warnings

NE_ACCURACY

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.

NE_ALG_FAIL

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

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

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

NE_CONVERGENCE

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

NE_INT

On entry, **istb**$[0] = \langle value\rangle$.
Constraint: **istb**$[0] \geq 1$.

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

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

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

NE_INT_2

On entry, **la** $= \langle value\rangle$ and **nnz** $= \langle value\rangle$.
Constraint: **la** $\geq 2 \times$ **nnz**.

On entry, **nb** $= \langle value\rangle$ and **n** $= \langle value\rangle$.
Constraint: $1 \leq$ **nb** $\leq$ **n**.

On entry, **nnz** $= \langle value\rangle$ and **n** $= \langle value\rangle$.
Constraint: **nnz** $\leq$ $\mathbf{n}^2$.

NE_INT_3

On entry, **lindb** $= \langle value\rangle$, **istb**[**nb**] $- 1 = \langle value\rangle$ and **nb** $= \langle value\rangle$.
Constraint: **lindb** $\geq$ **istb**[**nb**] $- 1$.

On entry, **m** $= \langle value\rangle$ and **n** $= \langle value\rangle$.
Constraint: if **method** = Nag_SparseNsym_RGMRES, $1 \leq$ **m** $\leq \min($**n**$, \langle value\rangle)$.
If **method** = Nag_SparseNsym_BiCGSTAB, $1 \leq$ **m** $\leq \min($**n**$, \langle value\rangle)$.

NE_INT_ARRAY

On entry, **indb**$[\langle value\rangle] = \langle value\rangle$ and **n** $= \langle value\rangle$.
Constraint: $1 \leq$ **indb**$[m-1] \leq$ **n**, for $m =$ **istb**$[0]$, **istb**$[0] + 1, \ldots,$ **istb**[**nb**] $- 1$.

NE_INTERNAL_ERROR

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

NE_INVALID_CS

On entry, **icol**$[\langle value\rangle] = \langle value\rangle$ and **n** $= \langle value\rangle$.
Constraint: $1 \leq$ **icol**$[i-1] \leq$ **n**, for $i = 1, 2, \ldots,$ **nnz**.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_nherm_precon_bdilu (f11dtc) and nag_sparse_nherm_precon_bdilu_solve (f11duc).

On entry, **irow**[$\langle value\rangle$] $= \langle value\rangle$ and **n** $= \langle value\rangle$.
Constraint: $1 \leq$ **irow**[$i-1$] $\leq$ **n**, for $i = 1, 2, \ldots,$ **nnz**.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_nherm_precon_bdilu (f11dtc) and nag_sparse_nherm_precon_bdilu_solve (f11duc).

NE_INVALID_CS_PRECOND

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_nherm_precon_bdilu (f11dtc) and nag_sparse_nherm_precon_bdilu_solve (f11duc).

NE_NOT_STRICTLY_INCREASING

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_nherm_precon_bdilu (f11dtc) and nag_sparse_nherm_precon_bdilu_solve (f11duc).

On entry, for $b = \langle value\rangle$, **istb**[b] $= \langle value\rangle$ and **istb**[$b-1$] $= \langle value\rangle$.
Constraint: **istb**[b] $>$ **istb**[$b-1$], for $b = 1, 2, \ldots,$ **nb**.

On entry, location $\langle value\rangle$ of (**irow**, **icol**) was a duplicate.
Check that **a**, **irow**, **icol**, **ipivp**, **ipivq**, **istr** and **idiag** have not been corrupted between calls to nag_sparse_nherm_precon_bdilu (f11dtc) and nag_sparse_nherm_precon_bdilu_solve (f11duc).

NE_REAL

On entry, **tol** $= \langle value\rangle$.
Constraint: **tol** < 1.0.

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 Parallelism and Performance

nag_sparse_nherm_precon_bdilu_solve (f11duc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_sparse_nherm_precon_bdilu_solve (f11duc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

The time taken by nag_sparse_nherm_precon_bdilu_solve (f11duc) for each iteration is roughly proportional to the value of **nnzc** returned from the preceding call to nag_sparse_nherm_precon_bdilu (f11dtc).

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 $\bar{A} = M^{-1}A$.

10 Example

This example program reads in a sparse matrix A and a vector b. It calls nag_sparse_nherm_precon_bdilu (f11dtc), 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_nherm_precon_bdilu_solve (f11duc) which uses the RGMRES method to solve $Ax = b$.

10.1 Program Text

```
/* nag_sparse_nherm_precon_bdilu_solve (f11duc) Example Program.
 *
 * Copyright 2013 Numerical Algorithms Group.
 *
 * Mark 24, 2013.
 */

#include <nag.h>
#include <nagf11.h>
#include <nag_stdlib.h>

static void overlap(Integer *n, Integer *nnz, Integer *irow, Integer *icol,
                    Integer *nb, Integer *istb, Integer *indb, Integer *lindb,
                    Integer *nover, Integer *iwork);

int main(void) {

  /* Scalars */
  double  dtolg, rnorm, tol;
  Integer i, itn, k, la, lfillg, lindb, liwork, m, maxitn, mb, n, nb, nnz;
  Integer nnzc, nover, exit_status = 0;
  Nag_SparseNsym_Method method;
  Nag_SparseNsym_Piv    pstrag;
  Nag_SparseNsym_Fact   milug;

  /* Arrays */
  char    nag_enum_arg[40];
  double  *dtol;
  Complex *a, *b, *x;
  Integer *icol, *idiag, *indb, *ipivp, *ipivq, *irow, *istb, *istr, *iwork;
  Integer *lfill, *npivm;
  Nag_SparseNsym_Piv  *pstrat;
  Nag_SparseNsym_Fact *milu;

  /* Nag Types */
  NagError              fail;

  /* Print example header */
  printf("nag_sparse_nherm_precon_bdilu_solve (f11duc) Example Program ");
  printf("Results\n\n");

  /* Skip heading in data file */
  scanf("%*[^\n] ");

  /* Get the matrix order and number of non-zero entries. */
  scanf("%ld %*[^\n]", &n);
  scanf("%ld %*[^\n]", &nnz);

  la     = 20 * nnz;
  lindb  = 3 * n;
  liwork = 9 * n + 3;

  /* Allocate arrays */
  b = NAG_ALLOC( n, Complex);
  x = NAG_ALLOC( n, Complex);

  a    = NAG_ALLOC( la, Complex);
  irow = NAG_ALLOC( la, Integer);
  icol = NAG_ALLOC( la, Integer);
```

```
idiag = NAG_ALLOC( lindb,   Integer);
indb  = NAG_ALLOC( lindb,   Integer);
ipivp = NAG_ALLOC( lindb,   Integer);
ipivq = NAG_ALLOC( lindb,   Integer);
istr  = NAG_ALLOC( lindb+1, Integer);

iwork = NAG_ALLOC( liwork, Integer);

if ( (!b) || (!x) || (!a) || (!irow) || (!icol) || (!idiag) || (!indb) ||
     (!ipivp) || (!ipivq) || (!istr) || (!iwork) ) {
  printf("Allocation failure!\n");
  exit_status = -1;
}

/* Initialise arrays */
for ( i = 0; i < n; i++ ) {
  b[i].re = 0.0;
  b[i].im = 0.0;
  x[i].re = 0.0;
  x[i].im = 0.0;
}

for ( i = 0; i < la; i++ ) {
  a[i].re = 0.0;
  a[i].im = 0.0;
  irow[i] = 0;
  icol[i] = 0;
}

for( i = 0; i < lindb; i++ ) {
  indb[i]  = 0;
  ipivp[i] = 0;
  ipivq[i] = 0;
  istr[i]  = 0;
  idiag[i] = 0;
}
istr[lindb] = 0;

for( i = 0; i < liwork; i++ ) {
  iwork[i] = 0;
}

/* Read the matrix A */
for ( i = 0; i < nnz; i++) {
  scanf(" (%lf, %lf) %ld %ld",
        &a[i].re, &a[i].im, &irow[i], &icol[i] );
}
scanf("%*[^\n] ");

/* Read the RHS b */
for ( i = 0; i < n; i++) {
  scanf(" (%lf, %lf)", &b[i].re, &b[i].im );
}
scanf("%*[^\n] ");

/* nag_enum_name_to_value (x04nac): Converts NAG enum member name to value */
scanf("%39s %*[^\n]", nag_enum_arg);
method = (Nag_SparseNsym_Method) nag_enum_name_to_value( nag_enum_arg );

/* Read algorithmic parameters */
scanf("%ld %lf %*[^\n]", &lfillg, &dtolg);

/* nag_enum_name_to_value (x04nac): Converts NAG enum member name to value */
scanf("%39s %*[^\n]", nag_enum_arg);
pstrag = (Nag_SparseNsym_Piv) nag_enum_name_to_value( nag_enum_arg );

/* nag_enum_name_to_value (x04nac): Converts NAG enum member name to value */
scanf("%39s %*[^\n]", nag_enum_arg);
milug = (Nag_SparseNsym_Fact) nag_enum_name_to_value( nag_enum_arg );
```

```
  /* Read algorithmic parameters */
  scanf("%ld %lf %ld %*[^\n]", &m, &tol, &maxitn);
  scanf("%ld %ld %*[^\n]", &nb, &nover);

  /* Allocate arrays */
  dtol  = NAG_ALLOC( nb,   double );
  istb  = NAG_ALLOC( nb+1, Integer);
  lfill = NAG_ALLOC( nb,   Integer);
  npivm = NAG_ALLOC( nb,   Integer);

  pstrat = (Nag_SparseNsym_Piv *) NAG_ALLOC(nb, Nag_SparseNsym_Piv);
  milu   = (Nag_SparseNsym_Fact *) NAG_ALLOC(nb, Nag_SparseNsym_Fact);

  if ( (!dtol) || (!istb) || (!lfill) || (!npivm) || (!pstrat) || (!milu) ) {
    printf("Allocation failure!\n");
    exit_status = -1;
  }

  /* Initialise arrays */
  for( i = 0; i < nb; i++ ) {
    dtol[i]  = 0.0;
    istb[i]  = 0;
    lfill[i] = 0;
    npivm[i] = 0;

    pstrat[i]  = 0;
    milu[i]    = 0;
  }
  istb[nb] = 0;

  /* Define diagonal block indices.
   * In this example use blocks of MB consecutive rows and initialise
   * assuming no overlap.
   */
  mb = (n + nb - 1)/nb;
  for ( k = 0; k < nb; k++) {
    istb[k] = k * mb + 1;
  }
  istb[nb] = n + 1;

  for ( i = 0; i < n; i++) {
    indb[i] = i + 1;
  }

  /* Modify INDB and ISTB to account for overlap. */
  overlap(&n, &nnz, irow, icol, &nb, istb, indb, &lindb, &nover, iwork);

  /* Set algorithmic parameters for each block from global values */
  for (k = 0; k < nb; k++) {
    lfill[k]  =  lfillg;
    dtol[k]   =  dtolg;
    pstrat[k] =  pstrag;
    milu[k]   =  milug;
  }

  /* Initialise fail */
  INIT_FAIL(fail);

  /* Calculate factorization
   *
   * nag_sparse_nherm_precon_bdilu (f11dtc). Calculates incomplete LU
   * factorization of local or overlapping diagonal blocks, mostly used
   * as incomplete LU preconditioner for complex sparse matrix.
   */
  nag_sparse_nherm_precon_bdilu(n, nnz, a, la, irow, icol, nb, istb, indb,
                                lindb, lfill, dtol, pstrat, milu, ipivp,
                                ipivq, istr, idiag, &nnzc, npivm, &fail);

  if( fail.code != NE_NOERROR ) {
    printf("Error from nag_sparse_nherm_precon_bdilu (f11dtc).\n%s\n\n",
           fail.message);
    exit(-2);
  }

  /* Initialise fail */
  INIT_FAIL(fail);

  /* Solve Ax = b using nag_sparse_nherm_precon_bdilu_solve (f11duc)
   *
   * nag_sparse_nherm_precon_bdilu_solve (f11duc): Solves complex sparse
   * nonsymmetric linear system, using block-jacobi preconditioner
   * generated by f11dtc.
   */
  nag_sparse_nherm_precon_bdilu_solve(method, n, nnz, a, la, irow, icol, nb,
                                      istb, indb, lindb, ipivp, ipivq, istr,
                                      idiag, b, m, tol, maxitn, x, &rnorm,
                                      &itn, &fail);

  if( fail.code != NE_NOERROR ) {
    printf("Error from nag_sparse_nherm_precon_bdilu_solve (f11duc).\n\n%s",
           fail.message);
    exit(-3);
  }

  /* Print output */
  printf(" Converged in %9ld iterations\n", itn);
  printf(" Final residual norm = %15.6E\n", rnorm);

  /* Output x */
  printf(" Solution vector  x\n");
  printf(" -----------------------\n");
  for (i = 0; i < n; i++) {
    printf(" ( %f,  %f )\n", x[i].re, x[i].im );
  }
  printf("\n");

  NAG_FREE(b);
  NAG_FREE(x);
  NAG_FREE(a);
  NAG_FREE(irow);
  NAG_FREE(icol);
  NAG_FREE(idiag);
  NAG_FREE(indb);
  NAG_FREE(ipivp);
  NAG_FREE(ipivq);
  NAG_FREE(istr);
  NAG_FREE(iwork);
  NAG_FREE(dtol);
  NAG_FREE(istb);
  NAG_FREE(lfill);
  NAG_FREE(npivm);
  NAG_FREE(pstrat);
  NAG_FREE(milu);

  return exit_status;
}

/* ************************************************************************ */

static void overlap(Integer *n, Integer *nnz, Integer *irow, Integer *icol,
                    Integer *nb, Integer *istb, Integer *indb, Integer *lindb,
                    Integer *nover, Integer *iwork) {
  /* Purpose
   * =======
   *
   * This routine takes a set of row indices INDB defining the diagonal blocks
   * to be used in nag_sparse_nherm_precon_bdilu (f11dtc) 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 on to
   * nag_sparse_nherm_precon_bdilu (f11dtc) to define overlapping diagonal
   * blocks.
   *
   * ------------------------------------------------------------------------ */

  /* Scalars */
  Integer i, ik, ind, iover, j, k, l, n21, nadd, row;

  /* Find the number of nonzero elements in each row of the matrix A, and start
   * address of each row. Store the start addresses in iwork(n,...,2*n-1).
   */

  for ( i = 0; i < (*n); i++) {
    iwork[i] =  0;
  }

  for ( i = 0; i < (*nnz); i++) {
    iwork[irow[i]-1] = iwork[irow[i]-1] + 1;
  }
  iwork[ (*n) ] = 1;

  for ( i = 0; i < (*n); i++) {
    iwork[(*n)+i+1] = iwork[(*n)+i] + iwork[i];
  }

  /* Loop over blocks. */
  for ( k = 0; k < (*nb); k++) {

    /* Initialize marker array. */
    for (j = 0; j < (*n); j++) {
      iwork[j] =  0;
    }

    /* Mark the rows already in block K in the workspace array. */
    for ( l = istb[k]; l < istb[k+1]; l++) {
      iwork[indb[l-1]-1] = 1;
    }

    /* Loop over levels of overlap. */
    for ( iover = 1; iover <= (*nover); iover++) {

      /* Initialize counter of new row indices to be added. */
      ind = 0;

      /* Loop over the rows currently in the diagonal block. */
      for ( l = istb[k]; l < istb[k+1]; l++ ) {
        row = indb[l-1];

        /* Loop over non-zero elements in row ROW. */
        for ( i = iwork[(*n)+row-1]; i < iwork[(*n)+row]; i++ ) {

          /* 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-1]-1]==0 ) {
            iwork[icol[i-1]-1] = 1;
            iwork[2*(*n)+1+ind] = icol[i-1];
            ind = ind + 1;
          }
        }
      }

      /* Shift the indices in INDB and add the new entries for block K.
       * Change ISTB accordingly.
       */
      nadd = ind;
      if ( istb[(*nb)]+nadd-1 > (*lindb) ) {
        printf("**** lindb too small, lindb = %ld ****\n", *lindb);
        exit(-1);
      }
      for ( i = istb[(*nb)] - 1; i >= istb[k+1]; i-- ) {
        indb[i+nadd-1] = indb[i-1];
      }

      n21 = 2 * (*n) + 1;
      ik = istb[k+1] - 1;

      for ( j = 0; j < nadd; j++ ) {
        indb[ik + j] =  iwork[n21 + j];
      }

      for ( j = k+1; j < (*nb)+1; j++) {
        istb[j] =  istb[j] + nadd;
      }
    }
  }
  return;
}
```

10.2 Program Data

```
nag_sparse_nherm_precon_bdilu_solve (f11duc) Example Program Data
 9                                  :n
 33                                 :nnz
 ( 96.0, -64.0)     1     1
 (-20.0,  22.0)     1     2
 (-36.0,  14.0)     1     4
 (-12.0,  10.0)     2     1
 ( 96.0, -64.0)     2     2
 (-20.0,  22.0)     2     3
 (-36.0,  14.0)     2     5
 (-12.0,  10.0)     3     2
 ( 96.0, -64.0)     3     3
 (-36.0,  14.0)     3     6
 (-28.0,  18.0)     4     1
 ( 96.0, -64.0)     4     4
 (-20.0,  22.0)     4     5
 (-36.0,  14.0)     4     7
 (-28.0,  18.0)     5     2
 (-12.0,  10.0)     5     4
 ( 96.0, -64.0)     5     5
 (-20.0,  22.0)     5     6
 (-36.0,  14.0)     5     8
 (-28.0,  18.0)     6     3
 (-12.0,  10.0)     6     5
 ( 96.0, -64.0)     6     6
 (-36.0,  14.0)     6     9
 (-28.0,  18.0)     7     4
 ( 96.0, -64.0)     7     7
 (-20.0,  22.0)     7     8
 (-28.0,  18.0)     8     5
 (-12.0,  10.0)     8     7
 ( 96.0, -64.0)     8     8
 (-20.0,  22.0)     8     9
 (-28.0,  18.0)     9     6
 (-12.0,  10.0)     9     8
 ( 96.0, -64.0)     9     9         :a(i), irow(i), icol(i) for i=1,nnz
 (100.0, 4.0)
 (100.0, 4.0)
 (100.0, 4.0)
 (100.0, 4.0)
 (100.0, 4.0)
 (100.0, 4.0)
 (100.0, 4.0)
 (100.0, 4.0)
 (100.0, 4.0)                       :b(i) for i=1,n
 Nag_SparseNsym_RGMRES              :method
 0   0.0                            :lfillg, dtolg
 Nag_SparseNsym_NoPiv               :pstrag
 Nag_SparseNsym_UnModFact          :milug
 2   1.0E-6  100                   :m, tol, maxitn
 3   1                             :nb, nover
```

10.3 Program Results

```
nag_sparse_nherm_precon_bdilu_solve (f11duc) Example Program Results

 Converged in         8 iterations
 Final residual norm =    6.492468E-04
 Solution vector  x
 -----------------------
 ( 2.203988,  1.697175 )
 ( 2.351101,  1.927544 )
 ( 1.593125,  1.436842 )
 ( 2.864079,  1.976215 )
 ( 3.068697,  2.264468 )
 ( 2.046736,  1.694760 )
 ( 2.206508,  1.324434 )
 ( 2.372372,  1.517040 )
 ( 1.625419,  1.178329 )
```