NAG Library Function Document

nag_sparse_nsym_precon_bdilu (f11dfc)

1 Purpose

nag_sparse_nsym_precon_bdilu (f11dfc) computes a block diagonal incomplete LU factorization of a real sparse nonsymmetric matrix, represented in coordinate storage format. The diagonal blocks may be composed of arbitrary rows and the corresponding columns, and may overlap. This factorization can be used to provide a block Jacobi or additive Schwarz preconditioner, for use in combination with nag_sparse_nsym_basic_solver (f11bec) or nag_sparse_nsym_precon_bdilu_solve (f11dgc).

2 Specification

```
#include <nag.h>
#include <nagf11.h>
void nag_sparse_nsym_precon_bdilu (Integer n, Integer nnz, double a[],
     Integer la, Integer irow[], Integer icol[], Integer nb,
     const Integer istb[], const Integer indb[], Integer lindb,
     const Integer lfill[], const double dtol[],
     const Nag_SparseNsym_Piv pstrat[], const Nag_SparseNsym_Fact milu[],
     Integer ipivp[], Integer ipivq[], Integer istr[], Integer idiag[],
     Integer *nnzc, Integer npivm[], NagError *fail)
```

3 Description

nag_sparse_nsym_precon_bdilu (f11dfc) computes an incomplete LU factorization (see Meijerink and Van der Vorst (1977) and Meijerink and Van der Vorst (1981)) of the (possibly overlapping) diagonal blocks A_b, for $b = 1, 2, \ldots,$ **nb**, of a real sparse nonsymmetric n by n matrix A. The factorization is intended primarily for use as a block Jacobi or additive Schwarz preconditioner (see Saad (1996)), with one of the iterative solvers nag_sparse_nsym_basic_solver (f11bec) and nag_sparse_nsym_precon_bdi lu_solve (f11dgc).

The **nb** diagonal blocks need not consist of consecutive rows and columns of A, but may be composed of arbitrarily indexed rows, and the corresponding columns, as defined in the arguments **indb** and **istb**. Any given row or column index may appear in more than one diagonal block, resulting in overlap. Each diagonal block A_b, for $b = 1, 2, \ldots,$ **nb**, is factorized as:

$$A_b = M_b + R_b$$

where

$$M_b = P_b L_b D_b U_b Q_b$$

and L_b is lower triangular with unit diagonal elements, D_b is diagonal, U_b is upper triangular with unit diagonals, P_b and Q_b are permutation matrices, and R_b is a remainder matrix.

The amount of fill-in occurring in the factorization of block b can vary from zero to complete fill, and can be controlled by specifying either the maximum level of fill **lfill**$[b-1]$, or the drop tolerance **dtol**$[b-1]$.

The parameter **pstrat**$[b-1]$ defines the pivoting strategy to be used in block b. The options currently available are no pivoting, user-defined pivoting, partial pivoting by columns for stability, and complete pivoting by rows for sparsity and by columns for stability. The factorization may optionally be modified to preserve the row-sums of the original block matrix.

The sparse matrix A is represented in coordinate storage (CS) format (see Section 2.1.1 in the f11 Chapter Introduction). The array **a** stores all the nonzero elements of the matrix A, while arrays **irow** and **icol** store the corresponding row and column indices respectively. Multiple nonzero elements may not be specified for the same row and column index.

The preconditioning matrices M_b, for $b = 1, 2, \ldots,$ **nb**, are returned in terms of the CS representations of the matrices

$$C_b = L_b + {D^{-1}}_b + U_b - 2I.$$

4 References

Meijerink J and Van der Vorst H (1977) An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix *Math. Comput.* **31** 148–162

Meijerink J and Van der Vorst H (1981) Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems *J. Comput. Phys.* **44** 134–155

Saad Y (1996) *Iterative Methods for Sparse Linear Systems* PWS Publishing Company, Boston, MA

5 Arguments

1: **n** – Integer *Input*

On entry: n, the order of the matrix A.

Constraint: **n** ≥ 1.

2: **nnz** – Integer *Input*

On entry: the number of nonzero elements in the matrix A.

Constraint: $1 \leq$ **nnz** $\leq$ **n**^{2}.

3: **a**[**la**] – double *Input/Output*

On entry: the nonzero elements in the matrix A, ordered by increasing row index, and by increasing column index within each row. Multiple entries for the same row and column indices are not permitted. The function nag_sparse_nsym_sort (f11zac) may be used to order the elements in this way.

On exit: the first **nnz** entries of **a** contain the nonzero elements of A and the next **nnzc** entries contain the elements of the matrices C_b, for $b = 1, 2, \ldots,$ **nb** stored consecutively. Within each block the matrix elements are ordered by increasing row index, and by increasing column index within each row.

4: **la** – Integer *Input*

On entry: the dimension of the arrays **a**, **irow** and **icol**. These arrays must be of sufficient size to store both A (**nnz** elements) and C (**nnzc** elements).

Note: the minimum value for **la** is only appropriate if **lfill** and **dtol** are set such that minimal fill-in occurs. If this is not the case then we recommend that **la** is set much larger than the minimum value indicated in the constraint.

Constraint: **la** $\geq 2 \times$ **nnz**.

5: **irow**[**la**] – Integer *Input/Output*

6: **icol**[**la**] – Integer *Input/Output*

On entry: the row and column indices of the nonzero elements supplied in **a**.

Constraints:

irow and **icol** must satisfy these constraints (which may be imposed by a call to nag_sparse_nsym_sort (f11zac)):

$1 \leq$ **irow**$[i-1] \leq$ **n** and $1 \leq$ **icol**$[i-1] \leq$ **n**, for $i = 1, 2, \ldots,$ **nnz**;
either **irow**$[i-1] <$ **irow**$[i]$ or both **irow**$[i-1] =$ **irow**$[i]$ and **icol**$[i-1] <$ **icol**$[i]$, for $i = 1, 2, \ldots,$ **nnz**.

On exit: the row and column indices of the nonzero elements returned in **a**.

7: **nb** – Integer *Input*

On entry: the number of diagonal blocks to factorize.

Constraint: $1 \leq \mathbf{nb} \leq \mathbf{n}$.

8: **istb**[**nb** + **1**] – const Integer *Input*

On entry: **istb**$[b-1]$, for $b = 1, 2, \ldots, \mathbf{nb}$, holds the indices in arrays **indb**, **ipivp**, **ipivq** and **idiag** that, on successful exit from this function, define block b. Let r_b denote the number of rows in block b; then $\mathbf{istb}[b] = \mathbf{istb}[b-1] + r_b$, for $b = 1, 2, \ldots, \mathbf{nb}$. Thus, **istb**[**nb**] holds the sum of the number of rows in all blocks plus **istb**[0].

Constraint: $\mathbf{istb}[0] \geq 1, \mathbf{istb}[b-1] < \mathbf{istb}[b],$ for $b = 1, 2, \ldots, \mathbf{nb}$.

9: **indb**[**lindb**] – const Integer *Input*

On entry: **indb** must hold the row indices appearing in each diagonal block, stored consecutively. Thus the elements $\mathbf{indb}[k_b - 1],$ for $k_b = \mathbf{istb}[b-1], \mathbf{istb}[b-1]+1, \ldots, \mathbf{istb}[b]-2, \mathbf{istb}[b]-1$, are the row indices in the bth block, for $b = 1, 2, \ldots, \mathbf{nb}$.

Constraint: $1 \leq \mathbf{indb}[m-1] \leq \mathbf{n},$ for $m = \mathbf{istb}[0], \mathbf{istb}[0]+1, \ldots, \mathbf{istb}[\mathbf{nb}]-1$.

10: **lindb** – Integer *Input*

On entry: the dimension of the arrays **indb**, **ipivp**, **ipivq** and **idiag**.

Constraint: $\mathbf{lindb} \geq \mathbf{istb}[\mathbf{nb}] - 1$.

11: **lfill**[**nb**] – const Integer *Input*

On entry: if $\mathbf{lfill}[b-1] \geq 0$ its value is the maximum level of fill allowed in the decomposition of the block (see Section 9.2 in nag_sparse_nsym_fac (f11dac)). A negative value of $\mathbf{lfill}[b-1]$ indicates that $\mathbf{dtol}[b-1]$ will be used to control the fill in the block instead.

12: **dtol**[**nb**] – const double *Input*

On entry: if $\mathbf{lfill}[b-1] < 0$ then $\mathbf{dtol}[b-1]$ is used as a drop tolerance in the block to control the fill-in (see Section 9.2 in nag_sparse_nsym_fac (f11dac)); otherwise $\mathbf{dtol}[b-1]$ is not referenced.

Constraint: if $\mathbf{lfill}[b-1] < 0,$ $\mathbf{dtol}[b-1] \geq 0.0,$ for $b = 1, 2, \ldots, \mathbf{nb}$.

13: **pstrat**[**nb**] – const Nag_SparseNsym_Piv *Input*

On entry: **pstrat**$[b-1]$, for $b = 1, 2, \ldots, \mathbf{nb}$, specifies the pivoting strategy to be adopted in the block as follows:

pstrat$[b-1]$ = Nag_SparseNsym_NoPiv
No pivoting is carried out.

pstrat$[b-1]$ = Nag_SparseNsym_UserPiv
Pivoting is carried out according to the user-defined input values of **ipivp** and **ipivq**.

pstrat$[b-1]$ = Nag_SparseNsym_PartialPiv
Partial pivoting by columns for stability is carried out.

pstrat$[b-1]$ = Nag_SparseNsym_CompletePiv
Complete pivoting by rows for sparsity, and by columns for stability, is carried out.

Suggested value: $\mathbf{pstrat}[b-1] =$ Nag_SparseNsym_CompletePiv, for $b = 1, 2, \ldots, \mathbf{nb}$.

Constraint: **pstrat**$[b-1]$ = Nag_SparseNsym_NoPiv, Nag_SparseNsym_UserPiv, Nag_SparseNsym_PartialPiv or Nag_SparseNsym_CompletePiv, for $b = 1, 2, \ldots, \mathbf{nb}$.

14: **milu**[**nb**] – const Nag_SparseNsym_Fact *Input*

On entry: **milu**$[b-1]$, for $b = 1, 2, \ldots,$ **nb**, indicates whether or not the factorization in the block should be modified to preserve row-sums (see Section 9.4 in nag_sparse_nsym_fac (f11dac)).

milu$[b-1]$ = Nag_SparseNsym_ModFact
The factorization is modified.

milu$[b-1]$ = Nag_SparseNsym_UnModFact
The factorization is not modified.

Constraint: **milu**$[b-1]$ = Nag_SparseNsym_ModFact or Nag_SparseNsym_UnModFact, for $b = 1, 2, \ldots,$ **nb**.

15: **ipivp**[**lindb**] – Integer *Input/Output*

16: **ipivq**[**lindb**] – Integer *Input/Output*

On entry: if **pstrat**$[b-1]$ = Nag_SparseNsym_UserPiv, then **ipivp**[**istb**$[b-1]+k-2$] and **ipivq**[**istb**$[b-1]+k-2$] must specify the row and column indices of the element used as a pivot at elimination stage k of the factorization of the block. Otherwise **ipivp** and **ipivq** need not be initialized.

Constraint: if **pstrat**$[b-1]$ = Nag_SparseNsym_UserPiv, the elements **istb**$[b-1]-1$ to **istb**$[b]-2$ of **ipivp** and **ipivq** must both hold valid permutations of the integers on $[1,$ **istb**$[b]-$ **istb**$[b-1]]$.

On exit: the row and column indices of the pivot elements, arranged consecutively for each block, as for **indb**. If **ipivp**[**istb**$[b-1]+k-2] = i$ and **ipivq**[**istb**$[b-1]+k-2] = j$, then the element in row i and column j of A_b was used as the pivot at elimination stage k.

17: **istr**[**lindb** + **1**] – Integer *Output*

On exit: **istr**[**istb**$[b-1]+k-2$], gives the index in the arrays **a**, **irow** and **icol** of row k of the matrix C_b, for $b = 1, 2, \ldots,$ **nb** and $k = 1, 2, \ldots,$ **istb**$[b]-$ **istb**$[b-1]$.

istr[**istb**[**nb**] $- 1$] contains **nnz** + **nnzc** + 1.

18: **idiag**[**lindb**] – Integer *Output*

On exit: **idiag**[**istb**$[b-1]+k-2$], gives the index in the arrays **a**, **irow** and **icol** of the diagonal element in row k of the matrix C_b, for $b = 1, 2, \ldots,$ **nb** and $k = 1, 2, \ldots,$ **istb**$[b]-$ **istb**$[b-1]$.

19: **nnzc** – Integer * *Output*

On exit: the sum total number of nonzero elements in the matrices C_b, for $b = 1, 2, \ldots,$ **nb**.

20: **npivm**[**nb**] – Integer *Output*

On exit: if **npivm**$[b-1] > 0$ it gives the number of pivots which were modified during the factorization to ensure that M_b exists.

If **npivm**$[b-1] = -1$ no pivot modifications were required, but a local restart occurred (see Section 9.3 in nag_sparse_nsym_fac (f11dac)). The quality of the preconditioner will generally depend on the returned values of **npivm**$[b-1]$, for $b = 1, 2, \ldots,$ **nb**.

If **npivm**$[b-1]$ is large, for some block, the preconditioner may not be satisfactory. In this case it may be advantageous to call nag_sparse_nsym_precon_bdilu (f11dfc) again with an increased value of **lfill**$[b-1]$, a reduced value of **dtol**$[b-1]$, or **pstrat**$[b-1]$ = Nag_SparseNsym_CompletePiv.

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

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_INT

On entry, **istb**[0] = ⟨*value*⟩.
Constraint: **istb**[0] ≥ 1.

On entry, **n** = ⟨*value*⟩.
Constraint: **n** ≥ 1.

On entry, **nnz** = ⟨*value*⟩.
Constraint: **nnz** ≥ 1.

NE_INT_2

On entry, **la** = ⟨*value*⟩ and **nnz** = ⟨*value*⟩.
Constraint: **la** ≥ 2 × **nnz**.

On entry, **nb** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: 1 ≤ **nb** ≤ **n**.

On entry, **nnz** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **nnz** ≤ $\mathbf{n}^2$.

NE_INT_3

On entry, **lindb** = ⟨*value*⟩, **istb**[**nb**] − 1 = ⟨*value*⟩ and **nb** = ⟨*value*⟩.
Constraint: **lindb** ≥ **istb**[**nb**] − 1.

NE_INT_ARRAY

On entry, **indb**[⟨*value*⟩] = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: $1 \leq \mathbf{indb}[m-1] \leq \mathbf{n}$, for $m = \mathbf{istb}[0], \mathbf{istb}[0]+1, \ldots, \mathbf{istb}[\mathbf{nb}]-1$.

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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 2.7.6 in How to Use the NAG Library and its Documentation for further information.

NE_INVALID_CS

On entry, **icol**[⟨*value*⟩] = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: $1 \leq \mathbf{icol}[j-1] \leq \mathbf{n}$, for $j = 1, 2, \ldots, \mathbf{nnz}$.

On entry, **irow**[⟨*value*⟩] = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: $1 \leq \mathbf{irow}[i-1] \leq \mathbf{n}$, for $i = 1, 2, \ldots, \mathbf{nnz}$.

NE_INVALID_ROWCOL_PIVOT

On entry, the user-supplied value of **ipivp** for block ⟨*value*⟩ lies outside its range.

On entry, the user-supplied value of **ipivp** for block ⟨*value*⟩ was repeated.

On entry, the user-supplied value of **ipivq** for block ⟨*value*⟩ lies outside its range.

On entry, the user-supplied value of **ipivq** for block $\langle value \rangle$ was repeated.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 2.7.5 in How to Use the NAG Library and its Documentation for further information.

NE_NOT_STRICTLY_INCREASING

On entry, element $\langle value \rangle$ of **a** was out of order.

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.

NE_REAL_ARRAY

On entry, **dtol**$[\langle value \rangle] = \langle value \rangle$.
Constraint: **dtol**$[b-1] \geq 0.0$, for $b = 1, 2, \ldots,$ **nb**.

NE_TOO_SMALL

The number of nonzero entries in the decomposition is too large.
The decomposition has been terminated before completion.
Either increase **la**, or reduce the fill by reducing **lfill**, or increasing **dtol**.

7 Accuracy

The accuracy of the factorization of each block A_b will be determined by the size of the elements that are dropped and the size of any modifications made to the pivot elements. If these sizes are small then the computed factors will correspond to a matrix close to A_b. The factorization can generally be made more accurate by increasing the level of fill **lfill**$[b-1]$, or by reducing the drop tolerance **dtol**$[b-1]$ with **lfill**$[b-1] < 0$.

If nag_sparse_nsym_precon_bdilu (f11dfc) is used in combination with nag_sparse_nsym_basic_solver (f11bec) or nag_sparse_nsym_precon_bdilu_solve (f11dgc), the more accurate the factorization the fewer iterations will be required. However, the cost of the decomposition will also generally increase.

8 Parallelism and Performance

nag_sparse_nsym_precon_bdilu (f11dfc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

Please consult the x06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

nag_sparse_nsym_precon_bdilu (f11dfc) calls nag_sparse_nsym_fac (f11dac) internally for each block A_b. The comments and advice provided in Section 9 in nag_sparse_nsym_fac (f11dac) on timing, control of fill, algorithmic details, and choice of parameters, are all therefore relevant to nag_sparse_nsym_precon_bdilu (f11dfc), if interpreted blockwise.

10 Example

This example program reads in a sparse matrix A and then defines a block partitioning of the row indices with a user-supplied overlap and computes an overlapping incomplete LU factorization suitable for use as an additive Schwarz preconditioner. Such a factorization is used for this purpose in the example program of nag_sparse_nsym_precon_bdilu_solve (f11dgc).

10.1 Program Text

```
/* nag_sparse_nsym_precon_bdilu (f11dfc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#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;
  Integer i, j, k, la, lfillg, lindb, liwork, minval, mb, n, nb, nnz, nnzc,
         nover;
  Integer exit_status = 0, maxval_ret = 9999;
  Nag_SparseNsym_Piv pstrag;
  Nag_SparseNsym_Fact milug;

  /* Arrays */
  char nag_enum_arg[40];
  double *a = 0, *dtol = 0;
  Integer *icol = 0, *idiag = 0, *indb = 0, *ipivp = 0, *ipivq = 0, *irow = 0;
  Integer *istb = 0, *istr = 0, *iwork = 0, *lfill = 0, *npivm = 0;
  Nag_SparseNsym_Piv *pstrat;
  Nag_SparseNsym_Fact *milu;

  /* Nag Types */
  NagError fail;

  /* Print example header */
  printf("nag_sparse_nsym_precon_bdilu (f11dfc) Example Program Results\n\n");

  /* Skip heading in data file */
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Get the matrix order and number of nonzero entries. */
#ifdef _WIN32
  scanf_s("%" NAG_IFMT " %*[^\n]", &n);
#else
  scanf("%" NAG_IFMT " %*[^\n]", &n);
#endif
#ifdef _WIN32
  scanf_s("%" NAG_IFMT " %*[^\n]", &nnz);
#else
  scanf("%" NAG_IFMT " %*[^\n]", &nnz);
#endif

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

  /* Allocate arrays */
  a = NAG_ALLOC(la, double);
  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 ((!a) || (!irow) || (!icol) || (!idiag) || (!indb) || (!ipivp) ||
      (!ipivq) || (!istr) || (!iwork)) {
    printf("Allocation failure!\n");
    exit_status = -1;
    goto END;
  }

  /* Initialize arrays */
  for (i = 0; i < la; i++) {
    a[i] = 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++) {
#ifdef _WIN32
    scanf_s("%lf %" NAG_IFMT " %" NAG_IFMT, &a[i], &irow[i], &icol[i]);
#else
    scanf("%lf %" NAG_IFMT " %" NAG_IFMT, &a[i], &irow[i], &icol[i]);
#endif
  }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read algorithmic parameters */
#ifdef _WIN32
  scanf_s("%" NAG_IFMT " %lf %*[^\n]", &lfillg, &dtolg);
#else
  scanf("%" NAG_IFMT " %lf %*[^\n]", &lfillg, &dtolg);
#endif

  /* nag_enum_name_to_value (x04nac): Converts NAG enum member name to value */
#ifdef _WIN32
  scanf_s("%39s %*[^\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
  scanf("%39s %*[^\n]", nag_enum_arg);
#endif
  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 */
#ifdef _WIN32
  scanf_s("%39s %*[^\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
  scanf("%39s %*[^\n]", nag_enum_arg);
#endif
```

```
  milug = (Nag_SparseNsym_Fact) nag_enum_name_to_value(nag_enum_arg);

  /* Read algorithmic parameters */
#ifdef _WIN32
  scanf_s("%" NAG_IFMT " %" NAG_IFMT " %*[^\n]", &nb, &nover);
#else
  scanf("%" NAG_IFMT " %" NAG_IFMT " %*[^\n]", &nb, &nover);
#endif

  if (nb < 1)
    {
      printf("Value read for nb is out of range\n");
      exit_status = -4;
      goto END;
    }

  /* 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;
    goto END;
  }

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

  /* Define diagonal block indices.
   * In this example use blocks of MB consecutive rows and initialize
   * 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);

  /* Output matrix and blocking details */
  printf(" Original Matrix\n");
  printf(" n     = %4" NAG_IFMT "\n", n);
  printf(" nnz   = %4" NAG_IFMT "\n", nnz);
  printf(" nb    = %4" NAG_IFMT "\n", nb);

  for (k = 0; k < nb; k++) {
    printf(" Block = %4" NAG_IFMT ",%12s = %4" NAG_IFMT ",", k + 1, "order",
           istb[k + 1] - istb[k]);
    minval = indb[istb[k] - 1];
    for (j = istb[k]; j < istb[k + 1] - 1; j++) {
      minval = MIN(minval, indb[j]);
    }
    printf("%13s = %4" NAG_IFMT "\n", "start row", minval);
```

```
}
printf("\n");

/* 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;
}

/* Initialize fail */
INIT_FAIL(fail);

/* Calculate factorization
 *
 * nag_sparse_nsym_precon_bdilu (f11dfc). Calculates incomplete LU
 * factorization of local or overlapping diagonal blocks, mostly used
 * as incomplete LU preconditioner for real sparse matrix.
 */
nag_sparse_nsym_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_nsym_precon_bdilu (f11dfc).\n%s\n\n",
         fail.message);
  exit_status = -2;
  goto END;
}

/* Output details of the factorization */
printf(" Factorization\n");
printf(" nnzc  = %4" NAG_IFMT "\n\n", nnzc);
printf(" Elements of factorization\n");
printf("        i    j           c(i,j)   Index\n");

for (k = 0; k < nb; k++) {
  printf("  C_%1" NAG_IFMT "  --------------------------------\n", k + 1);

  /* Elements of the k-th block */
  for (i = istr[istb[k] - 1] - 1; i < istr[istb[k + 1] - 1] - 1; i++) {
    printf("%9" NAG_IFMT " %4" NAG_IFMT " %16.5e %7" NAG_IFMT "\n",
           irow[i], icol[i], a[i], i + 1);
  }
}

k = 0;
maxval_ret = npivm[k];
for (k = 1; k < nb; k++) {
  maxval_ret = MAX(maxval_ret, npivm[k]);
}

printf("\n Details of factorized blocks\n");
if (maxval_ret > 0) {

  /* Including pivoting details. */
  printf("  k    i    istr(i)   idiag(I)    indb(i)  ipivp(i)  ipivq(i)\n");
  for (k = 0; k < nb; k++) {
    i = istb[k] - 1;
    printf("%3" NAG_IFMT " %3" NAG_IFMT " %10" NAG_IFMT " ", k + 1, i + 1,
           istr[i]);
    printf("%10" NAG_IFMT " %10" NAG_IFMT " %10" NAG_IFMT " %10" NAG_IFMT
           "\n", idiag[i], indb[i], ipivp[i], ipivq[i]);

    for (i = istb[k]; i < istb[k + 1] - 1; i++) {
      printf("%3" NAG_IFMT " %10" NAG_IFMT " %10" NAG_IFMT " ",
             i + 1, istr[i], idiag[i]);
      printf("%10" NAG_IFMT " %10" NAG_IFMT " %10" NAG_IFMT "\n",
             indb[i], ipivp[i], ipivq[i]);
    }
```

```
      printf(" ------------------------------------------------\n");
    }
  }
  else {

    /* No pivoting on any block. */
    printf("  k   i     istr(i)   idiag(i)    indb(i)\n");
    for (k = 0; k < nb; k++) {
      i = istb[k] - 1;
      printf("%3" NAG_IFMT " %3" NAG_IFMT " %10" NAG_IFMT " ", k + 1, i + 1,
             istr[i]);
      printf("%10" NAG_IFMT " %10" NAG_IFMT "\n", idiag[i], indb[i]);

      for (i = istb[k]; i < istb[k + 1] - 1; i++) {
        printf("%7" NAG_IFMT " %10" NAG_IFMT " %10" NAG_IFMT " %10" NAG_IFMT
               "\n", i + 1, istr[i], idiag[i], indb[i]);
      }
      printf(" ------------------------------------\n");
    }
  }

END:
  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(dtol);
  NAG_FREE(istb);
  NAG_FREE(lfill);
  NAG_FREE(npivm);
  NAG_FREE(pstrat);
  NAG_FREE(milu);
  NAG_FREE(iwork);

  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_nsym_precon_bdilu (f11dfc) 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
   * nag_sparse_nsym_precon_bdilu (f11dfc) 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 nonzero elements in row ROW. */
        for (i = iwork[(*n) + row - 1]; i < iwork[(*n) + row]; i++) {

          /* If the column index of the nonzero 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 = %" NAG_IFMT " ****\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_nsym_precon_bdilu (f11dfc) Example Program Data
  9                                   :n
  33                                  :nnz
  64.0      1       1
 -20.0      1       2
 -20.0      1       4
 -12.0      2       1
  64.0      2       2
 -20.0      2       3
 -20.0      2       5
 -12.0      3       2
  64.0      3       3
 -20.0      3       6
 -12.0      4       1
  64.0      4       4
 -20.0      4       5
 -20.0      4       7
 -12.0      5       2
 -12.0      5       4
  64.0      5       5
 -20.0      5       6
 -20.0      5       8
 -12.0      6       3
 -12.0      6       5
  64.0      6       6
 -20.0      6       9
 -12.0      7       4
  64.0      7       7
 -20.0      7       8
 -12.0      8       5
 -12.0      8       7
  64.0      8       8
 -20.0      8       9
 -12.0      9       6
 -12.0      9       8
  64.0      9       9                  :a(i), irow(i), icol(i) for i=1,nnz
  0   0.0                               :lfillg, dtolg
 Nag_SparseNsym_NoPiv                   :pstrag
 Nag_SparseNsym_UnModFact               :milug
  3   1                                 :nb, nover
```

10.3 Program Results

```
nag_sparse_nsym_precon_bdilu (f11dfc) Example Program Results

 Original Matrix
 n     =    9
 nnz   =   33
 nb    =    3
 Block =    1,       order =    6,    start row =    1
 Block =    2,       order =    9,    start row =    1
 Block =    3,       order =    6,    start row =    4

 Factorization
 nnzc  =   73

 Elements of factorization
        i    j           c(i,j)   Index
  C_1  ---------------------------------
        1    1      1.56250e-02      34
        1    2     -3.12500e-01      35
        1    4     -3.12500e-01      36
```

```
        2    1     -1.87500e-01      37
        2    2      1.65975e-02      38
        2    3     -3.31950e-01      39
        2    5     -3.31950e-01      40
        3    2     -1.99170e-01      41
        3    3      1.66621e-02      42
        3    6     -3.33241e-01      43
        4    1     -1.87500e-01      44
        4    4      1.65975e-02      45
        4    5     -3.31950e-01      46
        5    2     -1.99170e-01      47
        5    4     -1.99170e-01      48
        5    5      1.78466e-02      49
        5    6     -3.56931e-01      50
        6    3     -1.99945e-01      51
        6    5     -2.14159e-01      52
        6    6      1.79475e-02      53
C_2     ------------------------------
        1    1      1.56250e-02      54
        1    2     -3.12500e-01      55
        1    4     -1.87500e-01      56
        1    5     -3.12500e-01      57
        2    1     -1.87500e-01      58
        2    2      1.65975e-02      59
        2    3     -3.31950e-01      60
        2    6     -1.99170e-01      61
        2    7     -3.31950e-01      62
        3    2     -1.99170e-01      63
        3    3      1.66621e-02      64
        3    8     -1.99945e-01      65
        3    9     -3.33241e-01      66
        4    1     -3.12500e-01      67
        4    4      1.65975e-02      68
        4    6     -3.31950e-01      69
        5    1     -1.87500e-01      70
        5    5      1.65975e-02      71
        5    7     -3.31950e-01      72
        6    2     -3.31950e-01      73
        6    4     -1.99170e-01      74
        6    6      1.78466e-02      75
        6    8     -3.56931e-01      76
        7    2     -1.99170e-01      77
        7    5     -1.99170e-01      78
        7    7      1.78466e-02      79
        7    9     -3.56931e-01      80
        8    3     -3.33241e-01      81
        8    6     -2.14159e-01      82
        8    8      1.79475e-02      83
        9    3     -1.99945e-01      84
        9    7     -2.14159e-01      85
        9    9      1.79475e-02      86
C_3     ------------------------------
        1    1      1.56250e-02      87
        1    2     -3.12500e-01      88
        1    4     -1.87500e-01      89
        2    1     -1.87500e-01      90
        2    2      1.65975e-02      91
        2    3     -3.31950e-01      92
        2    5     -1.99170e-01      93
        3    2     -1.99170e-01      94
        3    3      1.66621e-02      95
        3    6     -1.99945e-01      96
        4    1     -3.12500e-01      97
        4    4      1.65975e-02      98
        4    5     -3.31950e-01      99
        5    2     -3.31950e-01     100
        5    4     -1.99170e-01     101
        5    5      1.78466e-02     102
        5    6     -3.56931e-01     103
        6    3     -3.33241e-01     104
        6    5     -2.14159e-01     105
```

```
        6    6      1.79475e-02     106

 Details of factorized blocks
  k    i     istr(i)   idiag(i)    indb(i)
  1    1         34         34          1
       2         37         38          2
       3         41         42          3
       4         44         45          4
       5         47         49          5
       6         51         53          6
 ---------------------------------------
  2    7         54         54          4
       8         58         59          5
       9         63         64          6
      10         67         68          1
      11         70         71          7
      12         73         75          2
      13         77         79          8
      14         81         83          3
      15         84         86          9
 ---------------------------------------
  3   16         87         87          7
      17         90         91          8
      18         94         95          9
      19         97         98          4
      20        100        102          5
      21        104        106          6
 ---------------------------------------
```