Module 5.6: nag_sparse_prec
Sparse Matrix Preconditioner Set-up and Solve

`nag_sparse_prec` provides procedures for setting up preconditioners for sparse linear systems together with solvers for the preconditioned systems.

Contents

Introduction

1 Background

This module contains procedures for setting up the preconditioners to be used with the iterative solvers of module `nag_sparse_lin_sys`. At this release methods for real non-symmetric and complex non-Hermitian systems are available. It also includes direct solvers for the resulting preconditioned systems $Mz = r$. The preconditioner information is stored as a sparse matrix.

Preconditioning involves the replacement of a given linear system

$$Ax = b \tag{1}$$

by the modified system

$$\bar{A}\bar{x} = \bar{b}. \tag{2}$$

A *left* preconditioner M^{-1} can be used by the GMRES(m), CGS and TFQMR methods, such that $\bar{A} = M^{-1}A \sim I_n$ in (2), where I_n is the identity matrix of order n; a *right* preconditioner M^{-1} can be used by the Bi-CGSTAB (ℓ) method, such that $\bar{A} = AM^{-1} \sim I_n$.

2 Choice of Procedures

Procedures for Jacobi, Symmetric Successive Overrelaxation (SSOR) and incomplete LU factorization (ILU) preconditioning methods are provided. These procedures initialize a preconditioner which can be supplied as an argument to the iterative solvers provided by the module `nag_sparse_lin_sys`. The simplest preconditioner is the Jacobi method, initialized by `nag_sparse_prec_init_jac`, which consists of the diagonal part of the matrix. However, more sophisticated methods usually give faster convergence of the iterative solvers. The SSOR preconditioner is initialized by `nag_sparse_prec_init_ssor` and the value of the relaxation parameter, ω, should be supplied. The cost of initializing and applying these two preconditioners is small when compared to the overall costs. For incomplete factorization methods, more work is required and this must be offset against the expected improvement in the convergence of the iterative solver. The ILU factorization is performed by `nag_sparse_prec_init_ilu`.

The procedure `nag_sparse_prec_sol` computes the solution vector z of the linear system

$$Mz = r,$$

where M is a preconditioning matrix described above, which is defined by a call to one of the initialization procedures.

The procedure `nag_sparse_prec_sol` is not specifically designed for direct solution of sparse linear systems. However, the arguments of the incomplete factorization initialization procedure `nag_sparse_prec_init_ilu` can be chosen in such a way so as to produce a direct solution. For example, in this case, procedure `nag_sparse_prec_sol` solves a linear system involving the incomplete LU preconditioning matrix

$$M = PLDUQ = A - R,$$

where P and Q are permutation matrices, L is unit lower triangular, U is unit upper triangular, D is diagonal and R is a remainder matrix. If A is non-singular, then setting the optional argument `drop_tol` = 0.0 in a call to `nag_sparse_prec_init_ilu`, results in a zero remainder matrix R and a complete factorization. A subsequent call to `nag_sparse_prec_sol` will therefore result in a direct method for solving the linear system (1).

Procedure: nag_sparse_prec_init_jac

1 Description

nag_sparse_prec_init_jac is a set-up procedure which forms the Jacobi preconditioner for a previously initialized n by n sparse matrix A. The preconditioning matrix is defined as $M = D$ where D is the diagonal part of A. You should ensure that the diagonal of A is full and does not contain any zero entries.

2 Usage

```
USE nag_sparse_prec
CALL nag_sparse_prec_init_jac(a,p  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

3.1 Mandatory Arguments

a — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(in)

Input: a structure containing details of the representation of the sparse matrix A.

Constraints: a must be as output from a call to one of the procedures nag_sparse_mat_init_coo, nag_sparse_mat_init_csc, nag_sparse_mat_init_csr or nag_sparse_mat_init_dia (see module nag_sparse_mat).

p — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(out)

Output: a structure containing details of the representation of the sparse preconditioning matrix M.

Constraints: p must be of the same type as a.

Note: to reduce the risk of corrupting the preconditioner accidentally, the components of this structure are private.

If you wish to deallocate this storage when the structure is no longer required, you must call the procedure nag_deallocate, as illustrated in Example 1 of this module document.

3.2 Optional Argument

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.
320	The procedure was unable to allocate enough memory.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

Procedure: nag_sparse_prec_init_ssor

1 Description

nag_sparse_prec_init_ssor is a set-up procedure which forms the symmetric successive-over-relaxation (SSOR) preconditioner for a previously initialized n by n sparse matrix A with nnz non-zero entries. The preconditioning matrix, M, is defined as

$$M = \frac{1}{\omega(2-\omega)}(D+\omega L)D^{-1}(D+\omega U),$$

where D is the diagonal part of A, L is the strictly lower triangular part of A, U is the strictly upper triangular part of A, and ω is a user-defined relaxation parameter. You should ensure that the diagonal of A is full and does not contain any zero entries.

2 Usage

```
USE nag_sparse_prec
CALL nag_sparse_prec_init_ssor(a,p  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

3.1 Mandatory Arguments

a — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(in)

Input: a structure containing details of the representation of the sparse matrix A.

Constraints: a must be as output from a call to one of the procedures nag_sparse_mat_init_coo, nag_sparse_mat_init_csc, nag_sparse_mat_init_csr or nag_sparse_mat_init_dia (see module nag_sparse_mat).

p — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(out)

Output: a structure containing details of the representation of the sparse preconditioning matrix M.

Constraints: p must be of the same type as a.

Note: to reduce the risk of corrupting the preconditioner accidentally, the components of this structure are private.

If you wish to deallocate this storage when the structure is no longer required, you must call the procedure nag_deallocate, as illustrated in Example 2 of this module document.

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.

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

Input: the relaxation parameter, ω.

Default: `omega` = 1.0.

Constraints: 0.0 < `omega` < 2.0.

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.
320	The procedure was unable to allocate enough memory.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

Procedure: nag_sparse_prec_init_ilu

1 Description

nag_sparse_prec_init_ilu computes an incomplete LU factorization (see Meijerink and van der Vorst [2], Meijerink and van der Vorst [4]) of a previously initialized n by n real non-symmetric or complex non-Hermitian sparse matrix A with nnz non-zero entries.

The decomposition is written in the form

$$A = M + R,$$

where

$$M = PLDUQ$$

and L is unit lower triangular, D is diagonal, U is unit upper triangular, P and Q are permutation matrices, and R is a remainder matrix.

The amount of fill-in occurring in the factorization can vary from zero to complete fill, and can be controlled by specifying either the maximum level of fill (fill_level) or the drop tolerance (drop_tol).

The optional argument pivoting defines the pivoting strategy to be used. 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 matrix.

2 Usage

```
USE nag_sparse_prec
CALL nag_sparse_prec_init_ilu(a,p  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array x must have exactly n elements.

3.1 Mandatory Arguments

a — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(in)

Input: a structure containing details of the representation of the sparse matrix A.

Constraints: a must be as output from a call to one of the procedures nag_sparse_mat_init_coo, nag_sparse_mat_init_csc, nag_sparse_mat_init_csr or nag_sparse_mat_init_dia (see module nag_sparse_mat).

p — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(out)

Output: a structure containing details of the representation of the sparse preconditioning matrix M.

Constraints: p must be of the same type as a.

Note: to reduce the risk of corrupting the preconditioner accidentally, the components of this structure are private.

If you wish to deallocate this storage when the structure is no longer required, you must call the procedure nag_deallocate, as illustrated in Example 3 of this module document.

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.

prec_max_nnz — integer, intent(in), optional

Input: an upper bound on the maximum number of non-zero entries in the incomplete LU factorization of A. This determines the size of storage arrays and must be large enough to include all added fill elements.

Default: `prec_max_nnz` $= nnz + \max(10, nnz/10)$, where nnz is the number of non-zero entries in A.

Constraints: `prec_max_nnz` $\geq nnz$.

fill_level — integer, intent(in), optional

Input: the required level of fill of the decomposition (for more details see Section 6.1).

Default: `fill_level` $= 0$.

Constraints: `fill_level` ≥ 0; `drop_tol` must not be present if `fill_level` is present.

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

Input: the drop tolerance of the decomposition. A fill element a_{ij} will be dropped if $|a_{ij}| <$ `drop_tol` $\times \max_{1\leq k,\ell\leq n} |a_{k\ell}|$ (for more details see Section 6.1).

Default: `drop_tol` $= 0.0$.

Constraints: `drop_tol` ≥ 0.0; `fill_level` must not be present if `drop_tol` is present.

pivoting — character(len=1), intent(in), optional

Input: specifies the pivoting strategy when this is not determined by the arguments `pivot_row` and `pivot_col`.

If `pivoting` = `'N'` or `'n'`, no pivoting is performed;

if `pivoting` = `'P'` or `'p'`, partial pivoting by columns for stability is performed;

if `pivoting` = `'C'` or `'c'`, complete pivoting by rows for sparsity and by columns for stability is performed.

Default: `pivoting` = `'C'`.

Constraints: `pivoting` = `'n'`, `'N'`, `'p'`, `'P'`, `'c'` or `'C'`; `pivot_row` and `pivot_col` must not be present if `pivoting` is present.

pivot_row(*n*) — integer, intent(in), optional

pivot_col(*n*) — integer, intent(in), optional

Input: `pivot_row`(k) and `pivot_col`(k) must specify the row and column indices of the pivot element to be used at elimination stage k.

Constraints: `pivot_row` and `pivot_col` must both contain valid permutations of the integers [1,n]; `pivoting` must not be present if `pivot_row` and `pivot_col` are present.

row_sum — logical, intent(in), optional

Input: indicates whether the factorization is to be modified to preserve row sums.

If `row_sum` = `.true.`, row sums are preserved;

if `row_sum` = `.false.`, row sums are not preserved.

Default: `row_sum` = `.false.`.

num_pivot — integer, intent(out), optional

Output: pivot information.

If num_pivot > 0, this is the number of pivots which were modified during the factorization to ensure that M exists;

if num_pivot $= 0$, no pivot modifications or local restarts were required;

if num_pivot $= -1$, no pivot modifications were required, but a local restart occurred.

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.
304	Invalid presence of an optional argument.
305	Invalid absence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	The storage defined by prec_max_nnz is too small to contain the fill. Either increase prec_max_nnz or reduce the values of fill_level or drop_tol.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 3 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The factorization is constructed row by row. At each elimination stage a row index is chosen. In the case of complete pivoting this index is chosen in order to reduce fill-in. Otherwise the rows are treated in the order given, or some user-defined order.

The chosen row is copied from the original matrix A and modified according to those previous elimination stages which affect it. During this process any fill-in elements are either dropped or kept according to the values of fill_level or drop_tol. In the case of a modified factorization (row_sum = .true.) the sum of the dropped terms for the given row is stored.

Finally, the pivot element for the row is chosen and the multipliers are computed for this elimination stage. For partial or complete pivoting the pivot element is chosen in the interests of stability as the

element of largest absolute value in the row. Otherwise the pivot element is chosen in the order given, or some user-defined order.

If the factorization breaks down because the chosen pivot element is zero, or there is no non-zero pivot available, a local restart recovery process is implemented. The modification of the given pivot row according to previous elimination stages is repeated, but this time keeping all fill. Note that in this case the final factorization will include more fill than originally specified by the user-supplied value of `fill_level` or `drop_tol`. The local restart usually results in a suitable non-zero pivot arising. The original criteria for dropping fill-in elements is then resumed for the next elimination stage (hence the local nature of the restart process). Should this restart process also fail to produce a non-zero pivot element an arbitrary unit pivot is introduced in an arbitrarily chosen column. An integer parameter `num_pivot` is optionally returned, which gives the number of these arbitrary unit pivots introduced. If no pivots were modified, but local restarts occurred, `num_pivot` is returned with a value of -1.

There is unfortunately no choice of the various algorithmic parameters which is optimal for all types of matrix, and some experimentation will generally be required for each new type of matrix encountered. The recommended approach is to start with `fill_level` = 0 and `pivoting` = 'C' (the default). If the value returned for `num_pivot` is significantly larger than zero, i.e., a large number of pivot modifications were required to ensure that M existed, the preconditioner is unlikely to be satisfactory. In this case increase `fill_level` until `num_pivot` falls to a value close to zero.

For certain classes of matrices (typically those arising from the discretisation of elliptic or parabolic partial differential equations) the convergence rate of the preconditioned iterative solver can sometimes be significantly improved by using an incomplete factorization which preserves the row-sums of the original matrix. In such cases, the setting `row_sum` = `.true.` is recommended.

Although it is not the primary purpose, `nag_sparse_prec_init_ilu` and `nag_sparse_prec_sol` may be used together to obtain a direct solution to a non-singular sparse non-Hermitian linear system. To achieve this the call to `nag_sparse_prec_sol` should be preceded by a complete LU factorization

$$A = PLDUQ = M.$$

A complete factorization is obtained from a call to `nag_sparse_prec_init_ilu` with `drop_tol` = 0.0, provided `num_pivot` = 0 or -1 on exit. A positive value of `num_pivot` indicates that A is singular or ill-conditioned. A factorization with `num_pivot`> 0 may serve as a preconditioner, but will not result in a direct solution. It is therefore essential to check the output value of `num_pivot` if a direct solution is required.

If `fill_level` ≥ 0, the amount of fill-in occurring in the incomplete factorization is controlled by limiting the maximum level of fill-in to `fill_level`. The original non-zero elements of A are defined to be of level 0. The fill level of a new non-zero location occurring during the factorization is defined as

$$k = \max(k_e, k_c) + 1,$$

where k_e is the level of fill of the element being eliminated, and k_c is the level of fill of the element causing the fill-in.

The fill-in can also be controlled by means of `drop_tol`, the drop tolerance. A potential fill-in element a_{ij} occurring in row i and column j will not be included if

$$|a_{ij}| < \texttt{drop\_tol} \times \alpha,$$

where α is the maximum modulus element in the matrix A.

For either method of control, any elements which are not included are discarded unless `row_sum` = `.true.`, in which case their contributions are subtracted from the pivot element in the relevant elimination row, in order to preserve the row-sums of the original matrix.

Should the factorization break down a local restart process is implemented as described above. This will affect the amount of fill present in the final factorization.

6.2 Timing

The time taken for a call to `nag_sparse_prec_init_ilu` is roughly proportional to $(prec\_nnz)^2/n$, where $prec\_nnz$ is the number of non-zero entries in the incomplete factorization matrix.

Procedure: nag_sparse_prec_sol

1 Description

nag_sparse_prec_sol is a generic procedure which computes the solution vector, z, of the system $Mz = r$ or its transpose $M^T z = r$ according to the value of the argument trans. The matrix M must be supplied as a structure of the sparse matrix derived type and previously initialized by a call to one of the set-up procedures contained in this module. nag_sparse_prec_sol may be used in combination with nag_sparse_prec_init_ilu to solve a sparse system of linear equations directly. See also the description of nag_sparse_prec_init_ilu.

2 Usage

```
USE nag_sparse_prec
CALL nag_sparse_prec_sol(p,r,z  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the following two cases.

Real / complex data

Real data: the arguments r and z are of type real(kind=*wp*), p is of type nag_sparse_mat_real_*wp*.

Complex data: the arguments r and z are of type complex(kind=*wp*), p is of type nag_sparse_mat_cmplx_*wp*.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

3.1 Mandatory Arguments

p — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(in)
Input: a structure containing details of the representation of the sparse preconditioning matrix M.
Constraints: p must be as output from a call to one of the procedures nag_sparse_prec_init_jac, nag_sparse_prec_init_ssor or nag_sparse_prec_init_ilu.

r(n) — real(kind=*wp*)/complex(kind=*wp*), intent(in)
Input: the right-hand-side of the linear system.

z(n) — real(kind=*wp*)/complex(kind=*wp*), intent(out)
Output: the solution of the linear system.
Constraints: z must be of the same type as r.

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.

trans — logical, intent(in), optional

Input: specifies which system is solved.

If `trans` = `.false.`, the preconditioned system $Mz = r$ is solved;

if `trans` = `.true.`, the transposed preconditioned system $M^T z = r$ is solved.

Default: `trans` = `.false.`.

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
302	An array argument has an invalid shape.

Warnings (error%level = 1):

error%code	Description
101	The argument `trans` is ignored when the Jacobi preconditioner is used.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 3 of this module document.

6 Further Comments

6.1 Accuracy

For the SSOR method, initialized by a call to `nag_sparse_prec_init_ssor`, the matrix M is

$$M = \frac{1}{\omega(2-\omega)}(D + \omega L)D^{-1}(D + \omega U).$$

If `trans` = `.false.`(the default), the computed solution z is the exact solution of a perturbed system of equations $(M + \delta M)z = r$, where

$$|\delta M| \leq c(n)\epsilon|D + \omega L||D^{-1}||D + \omega U|,$$

$c(n)$ is a modest linear function of n and ϵ is the machine precision. An equivalent result holds when `trans` = `.true.`.

For the ILU method initialized by a call to `nag_sparse_prec_init_ilu`, the matrix M is

$$M = PLDUQ.$$

If `trans` = `.false.`, the computed solution x is the exact solution of a perturbed system of equations $(M + \delta M)x = y$, where

$$|\delta M| \leq c(n)\epsilon P|L||D||U|Q,$$

where $c(n)$ and ϵ are as described above, with an equivalent result when `trans` = `.true.`.

6.2 Timing

The time taken for a call to nag_sparse_prec_sol is proportional to nnz, the number of non-zero entries in the matrix M.

Example 1: Solution of a Real, Non-symmetric Sparse Linear System Using Jacobi Preconditioned TFQMR

This example program initializes the sparse matrix A from the supplied data and sets up the Jacobi preconditioner by a call to `nag_sparse_prec_init_jac`. The preconditioned linear system is solved iteratively by the TFQMR method using the procedure `nag_sparse_gen_lin_sol`.

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_sparse_prec_ex01

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sparse_mat, ONLY : nag_sparse_mat_init_csc, &
   nag_sparse_mat_real_wp => nag_sparse_mat_real_dp, nag_deallocate
  USE nag_sparse_prec, ONLY : nag_sparse_prec_init_jac
  USE nag_sparse_lin_sys, ONLY : nag_sparse_gen_lin_sol
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n, nnz
  TYPE (nag_sparse_mat_real_wp) :: a, c_jac
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col_ptr(:), row(:)
  REAL (wp), ALLOCATABLE :: b(:), value(:), x(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sparse_prec_ex01'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, nnz

  ALLOCATE (row(nnz),col_ptr(n),value(nnz),b(n),x(n))

  DO i = 1, nnz
    READ (nag_std_in,*) value(i), row(i)
  END DO
  READ (nag_std_in,*) col_ptr
  READ (nag_std_in,*) b

  x = 0.0_wp

  CALL nag_sparse_mat_init_csc(a,value,row,col_ptr)

  CALL nag_sparse_prec_init_jac(a,c_jac)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Method: TFQMR with Jacobi preconditioner'
  WRITE (nag_std_out,*)

  CALL nag_sparse_gen_lin_sol(a,b,x,method='t',p=c_jac)

  ! Output results
```

```
      WRITE (nag_std_out,*) ' Solution'
      WRITE (nag_std_out,'(10F7.1)') x

      CALL nag_deallocate(a)
      CALL nag_deallocate(c_jac)
      DEALLOCATE (row,col_ptr,value,b,x)

    END PROGRAM nag_sparse_prec_ex01
```

2 Program Data

```
Example Program Data for nag_sparse_prec_ex01
  5   14          : n, nnz
  5.   1          : value(1), col(1)
  1.   2
  1.   4
  3.   2
  1.   3
  1.   1
  4.   3
 -1.   4
  1.   5
  1.   2
 -1.   3
  3.   4
 -1.   2
  5.   5          : value(nnz), col(nnz)
  1  4  6  10 13  : col_ptr(1:n)
  8.0  8.0  12.0  4.0  8.0     : b(1:n)
```

3 Program Results

```
 Example Program Results for nag_sparse_prec_ex01

 Method: TFQMR with Jacobi preconditioner

  Solution
    1.0    2.0    3.0    2.0    1.0
```

Example 2: Solution of a Complex, Non-Hermitian Sparse Linear System Using SSOR Preconditioned BiCGSTAB(ℓ)

This example program initializes the sparse matrix A, using the supplied data and sets up the corresponding SSOR preconditioner by a call to `nag_sparse_prec_init_ssor`. The preconditioned linear system is solved iteratively by the BiCGSTAB(ℓ) method using the procedure `nag_sparse_gen_lin_sol`.

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_sparse_prec_ex02

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sparse_mat, ONLY : nag_sparse_mat_init_coo, &
   nag_sparse_mat_cmplx_wp => nag_sparse_mat_cmplx_dp, nag_deallocate
  USE nag_sparse_prec, ONLY : nag_sparse_prec_init_ssor
  USE nag_sparse_lin_sys, ONLY : nag_sparse_gen_lin_sol
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC CMPLX, KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n, nnz, num_iter
  REAL (wp) :: resid_norm
  TYPE (nag_sparse_mat_cmplx_wp) :: a, c_ssor
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col(:), row(:)
  COMPLEX (wp), ALLOCATABLE :: b(:), value(:), x(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sparse_prec_ex02'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, nnz

  ALLOCATE (row(nnz),col(nnz),value(nnz),b(n),x(n))

  DO i = 1, nnz
    READ (nag_std_in,*) value(i), row(i), col(i)
  END DO
  READ (nag_std_in,*) b

  x = CMPLX(0.0_wp,0.0_wp,kind=wp)

  CALL nag_sparse_mat_init_coo(a,n,value,row,col)

  CALL nag_sparse_prec_init_ssor(a,c_ssor)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Method: BiCGSTAB with SSOR preconditioner'
  WRITE (nag_std_out,*)

  CALL nag_sparse_gen_lin_sol(a,b,x,method='b',p=c_ssor, &
   resid_norm=resid_norm,num_iter=num_iter)
```

```
      ! Output results

      WRITE (nag_std_out,*) ' Solution'
      WRITE (nag_std_out,'(3X,''('',F4.1,'','',F4.1,'')'')') x
      WRITE (nag_std_out,'(2x,''residual norm . . .  ='',1PE9.1)') resid_norm
      WRITE (nag_std_out,'(2x,''number of iterations ='',I4)') num_iter

      CALL nag_deallocate(a)
      CALL nag_deallocate(c_ssor)
      DEALLOCATE (row,col,value,b,x)

    END PROGRAM nag_sparse_prec_ex02
```

2 Program Data

```
Example Program Data for nag_sparse_prec_ex02
  5  16               : n, nnz
 ( 1., 0.)   3    5   : value(1), row(1), col(1)
 ( 0., 2.)   2    2
 (-2., 0.)   5    3
 (-2., 1.)   2    3
 ( 2., 3.)   1    1
 ( 1., 0.)   2    5
 (-6., 1.)   5    5
 ( 0.,-1.)   3    1
 (-1., 0.)   1    4
 ( 3.,-1.)   3    4
 (-2., 2.)   4    1
 ( 4.,-2.)   5    2
 (-3., 1.)   4    4
 ( 1.,-1.)   1    2
 ( 0., 3.)   4    5
 ( 5., 4.)   3    3   : value(nnz), row(nnz), col(nnz)
 (-3., 3.) (-11., 5.) ( 23.,48.) (-41., 2.)  (-28.,-31.)  : b(1:n)
```

3 Program Results

```
 Example Program Results for nag_sparse_prec_ex02

 Method: BiCGSTAB with SSOR preconditioner

  Solution
   ( 1.0, 2.0)
   ( 2.0, 3.0)
   ( 3.0, 4.0)
   ( 4.0, 5.0)
   ( 5.0, 6.0)
  residual norm . . .  =  4.3E-14
  number of iterations =   4
```

Example 3: Solution of a Real, Non-symmetric Sparse Linear System Using LU Factorization

This example program computes a complete LU factorization of a sparse matrix by supplying the optional argument `drop_tol` = 0.0 to `nag_sparse_prec_init_ilu`. This is followed by a call to `nag_sparse_prec_sol` to solve the linear system.

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_sparse_prec_ex03

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sparse_mat, ONLY : nag_sparse_mat_init_coo, &
   nag_sparse_mat_real_wp => nag_sparse_mat_real_dp, nag_deallocate
  USE nag_sparse_prec, ONLY : nag_sparse_prec_init_ilu, &
   nag_sparse_prec_sol
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n, nnz, num_pivot
  REAL (wp) :: drop_tol
  TYPE (nag_sparse_mat_real_wp) :: a, c_ilu
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col(:), row(:)
  REAL (wp), ALLOCATABLE :: r(:), value(:), z(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sparse_prec_ex03'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, nnz

  ALLOCATE (row(nnz),col(nnz),value(nnz),r(n),z(n))

  DO i = 1, nnz
    READ (nag_std_in,*) value(i), row(i), col(i)
  END DO
  READ (nag_std_in,*) r

  CALL nag_sparse_mat_init_coo(a,n,value,row,col)

  drop_tol = 0.0_wp
  CALL nag_sparse_prec_init_ilu(a,c_ilu,drop_tol=drop_tol, &
   num_pivot=num_pivot)

  IF (num_pivot>0) THEN
    WRITE (nag_std_out,*) 'Factorization is not complete'
  ELSE

    CALL nag_sparse_prec_sol(c_ilu,r,z)

    ! Output results
```

```
        WRITE (nag_std_out,*)
        WRITE (nag_std_out,*) 'Solution of linear system'
        WRITE (nag_std_out,'(10F7.1)') z

      END IF

      CALL nag_deallocate(a)
      CALL nag_deallocate(c_ilu)
      DEALLOCATE (row,col,value,r,z)

    END PROGRAM nag_sparse_prec_ex03
```

2 Program Data

```
Example Program Data for nag_sparse_prec_ex03
  5   13         : n, nnz
  1.   1    1    : value(1), row(1), col(1)
  1.   1    3
  2.   2    2
  1.   2    4
  1.   2    5
  1.   3    2
  3.   3    3
 -1.   3    5
  2.   4    1
  1.   4    4
  1.   5    1
  1.   5    3
  2.   5    5    : value(nnz), row(nnz), col(nnz)
  4.0  13.0  6.0  6.0  14.0     : b(1:n)
```

3 Program Results

```
 Example Program Results for nag_sparse_prec_ex03

 Solution of linear system
    1.0    2.0    3.0    4.0    5.0
```

Additional Examples

Not all example programs supplied with NAG *fl*90 appear in full in this module document. The following additional examples, associated with this module, are available.

`nag_sparse_prec_ex04`

Iterative solution of a complex, non-Hermitian, sparse system of linear equations by method BiCGSTAB with Jacobi preconditioning.

`nag_sparse_prec_ex05`

Iterative solution of a real, non-symmetric, sparse system of linear equations by method CG with SSOR preconditioning.

`nag_sparse_prec_ex06`

Direct solution of a complex, non-Hermitan, sparse system of linear equations using LU factorization.

References

[1] Barrett R, Berry M, Chan T F, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C and van der Vorst H (1994) *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* SIAM, Philadelphia

[2] 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

[3] Young D (1971) *Iterative Solution of Large Linear Systems* Academic Press, New York

[4] 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