Module 4.3: nag_sparse_mat
Sparse Matrix Utilities

`nag_sparse_mat` provides procedures for the initialization of matrices in the supported sparse formats together with matrix–vector multiply.

Contents

Introduction

This module defines the derived types `nag_sparse_mat_real_`*wp* or `nag_sparse_mat_cmplx_`*wp* used by the NAG *fl*90 Library to represent real and complex sparse matrices. It contains procedures for the initialization of structures of these types from any of the following commonly used formats:

coordinate (COO);

compressed sparse column (CSC);

compressed sparse row (CSR);

sparse diagonal (DIA).

Other formats will be added in future releases.

The sparse matrix derived type, once initialized, encapsulates the data structure to simplify the interface to procedures which use it.

This module contains utilities to deal with these derived types. The utilities are:

`nag_sparse_mat_extract` — Extracts details of a sparse matrix from a structure of type `nag_sparse_mat_real_`*wp* or `nag_sparse_mat_cmplx_`*wp*;

`nag_sparse_matvec` — Matrix–vector multiply for a sparse matrix.

Procedure: nag_sparse_mat_init_coo

1 Description

nag_sparse_mat_init_coo creates a structure containing the internal representation of the (real or complex) m by n sparse matrix A, where the entries of A are supplied in COO format.

2 Usage

```
USE nag_sparse_mat
CALL nag_sparse_mat_init_coo(a,m,value,row_indx,col_indx  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the following cases:

Real data: a is of type nag_sparse_mat_real_*wp* and value is of type real(kind=*wp*).

Complex data: a is of type nag_sparse_mat_cmplx_*wp* and value is of type complex(kind=*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.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$nnz \geq 1$ — the number of non-zero entries

3.1 Mandatory Arguments

a — 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 matrix A.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the sparse matrix may be extracted by calling nag_sparse_mat_extract.

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.

m — integer, intent(in)

Input: m, the number of rows of A.

Constraints: m ≥ 1.

value(nnz**)** — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the non-zero entries of A.

row_indx(nnz**)** — integer, intent(in)

col_indx(nnz**)** — integer, intent(in)

Input: the row and column indices of the non-zero entries of A. row_indx(i) and col_indx(i) *must* contain the row and column indices of the non-zero entry stored in value(i), for $i = 1, 2, \ldots, nnz$.

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.

n — integer, intent(in), optional

Input: n, the number of columns of A.

Constraints: `n` ≥ 1.

Default: the matrix is square, `n` = `m`.

max_nnz — integer, intent(in), optional

Input: the maximum number of entries allowed in `a`. Using `max_nnz` $> nnz$ allows the addition of more elements to the matrix without the need to create another structure.

Constraints: `max_nnz` $\geq nnz$.

Default: `max_nnz` $= nnz$.

zero_base — logical, intent(in), optional

Input: specifies whether the row and column indices start at zero i.e., `row_indx` and `col_indx` contain the indices of $A(0:m-1, 0:n-1)$ rather than of $A(1:m, 1:n)$.

If `zero_base` = `.false.`, `row_indx` and `col_indx` contain the indices of $A(1:m, 1:n)$;

if `zero_base` = `.true.`, `row_indx` and `col_indx` contain the indices of $A(0:m-1, 0:n-1)$.

Default: `zero_base` = `.false.`.

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

Input: specifies the matrix type.

If `mat_type` = `'g'` or `'G'`, the matrix is general;

if `mat_type` = `'s'` or `'S'`, the matrix is symmetric;

if `mat_type` = `'h'` or `'H'`, the matrix is Hermitian;

if `mat_type` = `'t'` or `'T'`, the matrix is triangular.

Constraints: `mat_type` = `'g'`, `'G'`, `'s'`, `'S'`, `'h'`, `'H'`, `'t'` or `'T'`.

Default: `mat_type` = `'g'`.

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

Input: specifies whether A is upper or lower triangular. If A is symmetric or Hermitian, `uplo` specifies whether the upper or lower triangle is supplied.

If `uplo` = `'u'` or `'U'`, either A is upper triangular or the upper triangle of a symmetric or Hermitian matrix is supplied;

if `uplo` = `'l'` or `'L'`, either A is lower triangular or the lower triangle of a symmetric or Hermitian matrix is supplied.

Note: `uplo` will be ignored if `mat_type` = `'g'`.

Constraints: `uplo` = `'u'`, `'U'`, `'l'` or `'L'`.

Default: `uplo` = `'l'`.

unit_diag — logical, intent(in), optional

Input: specifies whether the triangular, symmetric or Hermitian matrix has unit diagonal elements.

If `unit_diag` = `.false.`, the diagonal elements are supplied with the input data;

if `unit_diag` = `.true.`, the diagonal elements are *not* supplied and assumed to be unity.

Default: `unit_diag` = `.false.`.

check_indx — logical, intent(in), optional

Input: specifies whether the input data is to be checked.

If `check_indx` = `.true.`, the input data will be checked;

if `check_indx` = `.false.`, *no* checks will be performed on the input data.

Note: `check_indx` = `.false.` *must not* be used unless all indices within the input data are correct. Other Library procedures assume that the data are correct and unexpected errors may occur if that is not the case.

Default: `check_indx` = `.true.`.

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

Input: specifies how entries with zero entries are to be treated.

If `zero_entry` = `'r'` or `'R'`, zero entries are removed;

if `zero_entry` = `'k'` or `'K'`, zero entries are kept;

if `zero_entry` = `'e'` or `'E'`, on detecting a zero an error is raised.

Constraints: `zero_entry` = `'r'`, `'R'`, `'k'`, `'K'`, `'e'` or `'E'`.

Default: `zero_entry` = `'k'`.

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

Input: specifies how entries with duplicate row and column indices are to be treated.

If `dupl_entry` = `'n'` or `'N'`, duplicates are not checked;

if `dupl_entry` = `'f'` or `'F'`, duplicates are removed (keep first);

if `dupl_entry` = `'l'` or `'L'`, duplicates are removed (keep last);

if `dupl_entry` = `'s'` or `'S'`, duplicates are summed;

if `dupl_entry` = `'e'` or `'E'`, on detecting a duplicate an error is raised.

Note: some Library procedures assume that there are no duplicate entries. If any of these procedures will be used you *must not* use `dupl_entry` = `'n'` unless you are sure that there are no duplicate entries.

Constraints: `dupl_entry` = `'n'`, `'N'`, `'f'`, `'F'`, `'l'`, `'L'`, `'s'`, `'S'`, `'e'` or `'E'`.

Default: `dupl_entry` = `'l'`.

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

Input: specifies whether the input data is ordered.

If `ordered` = `'n'` or `'N'`, input data is not ordered;

if `ordered` = `'f'` or `'F'`, elements of input data are ordered in increasing row index and increasing column index within each row;

if `ordered` = `'i'` or `'I'`, elements of input data are ordered in increasing column index and increasing row index within each column;

if `ordered` = `'r'` or `'R'`, elements of input data are ordered in increasing row index but elements within each row are not ordered;

if `ordered` = `'c'` or `'C'`, elements of input data are ordered in increasing column index but elements within each column are not ordered.

Constraints: `ordered` = `'n'`, `'N'`, `'f'`, `'F'`, `'i'`, `'I'`, `'r'`, `'R'`, `'c'` or `'C'`.

Note: some Library procedures need the data ordered in a certain way. If the data is already ordered this information will help in avoiding unnecessary sorting.

Default: `ordered` = `'n'`.

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.
320	The procedure was unable to allocate enough memory.

Warnings (error%level = 1):

error%code	Description
101	Redundant optional argument. The optional argument `uplo` is not required for a general matrix. It will be ignored.
102	Redundant optional argument. The optional argument `unit_diag` is not allowed for a general matrix. It will be ignored.

5 Examples of Usage

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

6 Further Comments

This procedure deals with zero entries according to the optional argument `zero_entry` (default = keep zero entries) then deals with duplicate entries according to the optional argument `dupl_entry` (default = keep last entry).

Procedure: nag_sparse_mat_init_csc

1 Description

nag_sparse_mat_init_csc creates a structure containing the internal representation of the (real or complex) m by n sparse matrix A, where the entries of A are supplied in CSC format.

2 Usage

```
USE nag_sparse_mat
CALL nag_sparse_mat_init_csc(a,value,row_indx,col_begin  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the following cases:

Real data: a is of type nag_sparse_mat_real_*wp* and value is of type real(kind=*wp*).

Complex data: a is of type nag_sparse_mat_cmplx_*wp* and value is of type complex(kind=*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.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$nnz \geq 1$ — the number of non-zero entries

$n \geq 1$ — the number of columns of the matrix

3.1 Mandatory Arguments

a — 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 matrix A.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the sparse matrix may be extracted by calling nag_sparse_mat_extract.

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.

value(*nnz*) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the non-zero entries of A stored column by column (see the Module Introduction).

row_indx(*nnz*) — integer, intent(in)

Input: the row indices of the non-zero entries of A. row_indx(i) *must* contain the row index of the non-zero entry stored in value(i), for $i = 1, 2, \ldots, nnz$.

col_begin(n) — integer, intent(in)

Input: the index of the first entry of each column of the input data.

If `col_end` is *not* present, the columns *must* be in consecutive order, but need not be ordered by row within each column (see the optional argument `ordered`);

if `col_end` is present, the columns need not be in consecutive order and need not be ordered by row within each column. `col_begin`$(j) - \alpha + 1$, where $\alpha = \min_{i=1}^{n}$ `col_begin`(i), gives the location (within the range $1 : nnz$) in `value` and `row_indx` of the first entry of column j.

Note: an empty column is defined as follows:

if `col_end` is *not* present, column j is empty if `col_begin(`j`)` $=$ `col_begin(`$j+1$`)`, for $j = 1, 2, \ldots, n-1$ and column n is empty if `col_begin(`n`)` $= nnz + 1$;

if `col_end` is present, column j is empty if `col_end(`j`)` $=$ `col_begin(`j`)`, for $j = 1, 2, \ldots, n$.

Constraints:

if `col_end` is *not* present then `col_begin(`j`)` $\leq$ `col_begin(`$j+1$`)`, for $j = 1, 2, \ldots, n-1$ with `col_begin(1)` $= 1$ and `col_begin(`n`)` $\leq nnz + 1$;

if `col_end` is present, $\alpha \leq$ `col_begin(`j`)` $\leq nnz + \alpha$, for $j = 1, 2, \ldots, n$.

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.

m — integer, intent(in), optional

Input: m, the number of rows of A.

Constraints: `m` ≥ 1.

Default: the matrix is square, `m` $= n$.

max_nnz — integer, intent(in), optional

Input: the maximum number of entries allowed in `a`. Using `max_nnz` $> nnz$ allows the addition of more elements to the matrix without the need to create another structure.

Constraints: `max_nnz` $\geq nnz$.

Default: `max_nnz` $= nnz$.

col_end(n) — integer, intent(in), optional

Input: 1 + the index of the last entry of each column of the input data. `col_end`$(j) - \alpha$, where $\alpha = \min_{i=1}^{n}$ `col_begin`(i), gives the location (within the range $1 : nnz$) in `value` and `row_indx` of the last entry of column j.

Note: if column j is empty then `col_end(`j`)` $=$ `col_begin(`j`)`, for $j = 1, 2, \ldots, n$.

Constraints: $\alpha \leq$ `col_begin`$(j) \leq$ `col_end`$(j) \leq nnz + \alpha$, for $j = 1, 2, \ldots, n$.

Default: columns are provided in consecutive order, `col_end(`j`)` $=$ `col_begin(`$j+1$`)`, for $j = 1, 2, \ldots, n-1$ and `col_end(`n`)` $= nnz + 1$.

zero_base — logical, intent(in), optional

Input: specifies whether the row indices start at zero i.e., `row_indx` is in the range $(0 : m-1)$ rather than $(1 : m)$.

If `zero_base` $=$ `.false.`, `row_indx` is in the range $(1 : m)$;

if `zero_base` $=$ `.true.`, `row_indx` is in the range $(0 : m-1)$.

Default: `zero_base` $=$ `.false.`.

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

Input: specifies the matrix type.

If mat_type = 'g' or 'G', the matrix is general;

if mat_type = 's' or 'S', the matrix is symmetric;

if mat_type = 'h' or 'H', the matrix is Hermitian;

if mat_type = 't' or 'T', the matrix is triangular.

Constraints: mat_type = 'g', 'G', 's', 'S', 'h', 'H', 't' or 'T'.

Default: mat_type = 'g'.

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

Input: specifies whether A is upper or lower triangular. If A is symmetric or Hermitian, uplo specifies whether the upper or lower triangle is supplied.

If uplo = 'u' or 'U', either A is upper triangular or the upper triangle of a symmetric or Hermitian matrix is supplied;

if uplo = 'l' or 'L', either A is lower triangular or the lower triangle of a symmetric or Hermitian matrix is supplied.

Note: uplo will be ignored if mat_type = 'g'.

Constraints: uplo = 'u', 'U', 'l' or 'L'.

Default: uplo = 'l'.

unit_diag — logical, intent(in), optional

Input: specifies whether the triangular, symmetric or Hermitian matrix has unit diagonal elements.

If unit_diag = .false., the diagonal elements are supplied with the input data;

if unit_diag = .true., the diagonal elements are *not* supplied and assumed to be unity.

Default: unit_diag = .false..

check_indx — logical, intent(in), optional

Input: specifies whether the input data is to be checked.

If check_indx = .true., the input data will be checked;

if check_indx = .false., *no* checks will be performed on the input data.

Note: check_indx = .false. *must not* be used unless all indices within the input data are correct. Other Library procedures assume that the data are correct and unexpected errors may occur if that is not the case.

Default: check_indx = .true..

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

Input: specifies how entries with zero entries are to be treated.

If zero_entry = 'r' or 'R', zero entries are removed;

if zero_entry = 'k' or 'K', zero entries are kept;

if zero_entry = 'e' or 'E', on detecting a zero an error is raised.

Constraints: zero_entry = 'r', 'R', 'k', 'K', 'e' or 'E'.

Default: zero_entry = 'k'.

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

Input: specifies how entries with duplicate row and column indices are to be treated.

If `dupl_entry` = `'n'` or `'N'`, duplicates are not checked;

if `dupl_entry` = `'f'` or `'F'`, duplicates are removed (keep first);

if `dupl_entry` = `'l'` or `'L'`, duplicates are removed (keep last);

if `dupl_entry` = `'s'` or `'S'`, duplicates are summed;

if `dupl_entry` = `'e'` or `'E'`, on detecting a duplicate an error is raised.

Note: some Library procedures assume that there are no duplicate entries. If any of these procedures will be used you *must not* use `dupl_entry` = `'n'` unless you are sure that there are no duplicate entries.

Constraints: `dupl_entry` = `'n'`, `'N'`, `'f'`, `'F'`, `'l'`, `'L'`, `'s'`, `'S'`, `'e'` or `'E'`.

Default: `dupl_entry` = `'l'`.

ordered — logical, intent(in), optional

Input: specifies whether the entries in each column of the input data are ordered.

If `ordered` = `.false.`, row indices within each column are not ordered;

if `ordered` = `.true.`, each column is ordered in increasing row index.

Note: some Library procedures need the data ordered in a certain way. If the data is already ordered this information will help in avoiding unnecessary sorting.

Default: `ordered` = `.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
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Warnings (error%level = 1):

error%code	Description
101	Redundant optional argument. The optional argument `uplo` is not required for a general matrix. It will be ignored.
102	Redundant optional argument. The optional argument `unit_diag` is not allowed for a general matrix. It will be ignored.

5 Examples of Usage

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

Procedure: nag_sparse_mat_init_csr

1 Description

nag_sparse_mat_init_csr creates a structure containing the internal representation of the (real or complex) m by n sparse matrix A, where the entries of A are supplied in CSR format.

2 Usage

```
USE nag_sparse_mat
CALL nag_sparse_mat_init_csr(a,value,col_indx,row_begin  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the following cases:

Real data: a is of type nag_sparse_mat_real_*wp* and value is of type real(kind=*wp*).

Complex data: a is of type nag_sparse_mat_cmplx_*wp* and value is of type complex(kind=*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.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$nnz \geq 1$ — the number of non-zero entries

$m \geq 1$ — the number of rows of the matrix

3.1 Mandatory Arguments

a — 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 matrix A.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the sparse matrix may be extracted by calling nag_sparse_mat_extract.

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.

value(*nnz*) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the non-zero entries of A stored row by row (see the Module Introduction).

col_indx(*nnz*) — integer, intent(in)

Input: the column indices of the non-zero entries of A. col_indx(i) *must* contain the column index of the non-zero entry stored in value(i), for $i = 1, 2, \ldots, nnz$.

row_begin(m) — integer, intent(in)

Input: the index of the first entry of each row of the input data.

If `row_end` is *not* present, the rows *must* be in consecutive order, but need not be ordered by column within each row (see the optional argument `ordered`);

if `row_end` is present, the rows need not be in consecutive order and need not be ordered by column within each row. `row_begin`$(j)-\alpha+1$, where $\alpha = \min_{i=1}^{m}$ `row_begin`(i), gives the location (within the range $1:nnz$) in `value` and `col_indx` of the first entry of row j.

Note: an empty row is defined as follows:

if `row_end` is *not* present, row j is empty if `row_begin`(j) = `row_begin`$(j+1)$, for $j = 1, 2, \ldots, m-1$ and row m is empty if `row_begin`$(m) = nnz+1$;

if `row_end` is present, row j is empty if `row_end`(j) = `row_begin`(j), for $j = 1, 2, \ldots, m$.

Constraints:

if `row_end` is *not* present then `row_begin`$(j) \leq$ `row_begin`$(j+1)$, for $j = 1, 2, \ldots, m-1$ with `row_begin(1)` $= 1$ and `row_begin`$(m) \leq nnz+1$;

if `row_end` is present, $\alpha \leq$ `row_begin`$(j) \leq nnz+\alpha$ for $j = 1, 2, \ldots, m$.

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.

n — integer, intent(in), optional

Input: n, the number of columns of A.

Constraints: `n` ≥ 1.

Default: the matrix is square, `n` $= m$.

max_nnz — integer, intent(in), optional

Input: the maximum number of entries allowed in `a`. Using `max_nnz` $> nnz$ allows the addition of more elements to the matrix without the need to create another structure.

Constraints: `max_nnz` $\geq nnz$.

Default: `max_nnz` $= nnz$.

row_end(m) — integer, intent(in), optional

Input: 1+ the index of the last entry of each row of the input data. `row_end`$(j) - \alpha$, where $\alpha = \min_{i=1}^{m}$ `row_begin`(i), gives the location (within the range $1:nnz$) in `value` and `col_indx` of the last entry of row j.

Note: if row j is empty then `row_end`(j) = `row_begin`(j), for $j = 1, 2, \ldots, m$.

Constraints: $\alpha \leq$ `row_begin`$(j) \leq$ `row_end`$(j) \leq nnz+\alpha$, for $j = 1, 2, \ldots, m$.

Default: rows are provided in consecutive order, `row_end`(j) = `row_begin`$(j+1)$, for $j = 1, 2, \ldots, m-1$ and `row_end`$(m) = nnz+1$.

zero_base — logical, intent(in), optional

Input: specifies whether the column indices start at zero i.e., `col_indx` is in the range $(0 : n-1)$ rather than $(1 : n)$.

If `zero_base` = `.false.`, `col_indx` is in the range $(1 : n)$;

if `zero_base` = `.true.`, `col_indx` is in the range $(0 : n-1)$.

Default: `zero_base` = `.false.`.

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

Input: specifies the matrix type.

If mat_type = 'g' or 'G', the matrix is general;

if mat_type = 's' or 'S', the matrix is symmetric;

if mat_type = 'h' or 'H', the matrix is Hermitian;

if mat_type = 't' or 'T', the matrix is triangular.

Constraints: mat_type = 'g', 'G', 's', 'S', 'h', 'H', 't' or 'T'.

Default: mat_type = 'g'.

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

Input: specifies whether A is upper or lower triangular. If A is symmetric or Hermitian, uplo specifies whether the upper or lower triangle is supplied.

If uplo = 'u' or 'U', either A is upper triangular or the upper triangle of a symmetric or Hermitian matrix is supplied;

if uplo = 'l' or 'L', either A is lower triangular or the lower triangle of a symmetric or Hermitian matrix is supplied.

Note: uplo will be ignored if mat_type = 'g'.

Constraints: uplo = 'u', 'U', 'l' or 'L'.

Default: uplo = 'l'.

unit_diag — logical, intent(in), optional

Input: specifies whether the triangular, symmetric or Hermitian matrix has unit diagonal elements.

If unit_diag = .false., the diagonal elements are supplied with the input data;

if unit_diag = .true., the diagonal elements are *not* supplied and assumed to be unity.

Default: unit_diag = .false..

check_indx — logical, intent(in), optional

Input: specifies whether the input data is to be checked.

If check_indx = .true., the input data will be checked;

if check_indx = .false., *no* checks will be performed on the input data.

Note: check_indx = .false. *must not* be used unless all indices within the input data are correct. Other Library procedures assume that the data are correct and unexpected errors may occur if that is not the case.

Default: check_indx = .true..

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

Input: specifies how entries with zero entries are to be treated.

If zero_entry = 'r' or 'R', zero entries are removed;

if zero_entry = 'k' or 'K', zero entries are kept;

if zero_entry = 'e' or 'E', on detecting a zero an error is raised.

Constraints: zero_entry = 'r', 'R', 'k', 'K', 'e' or 'E'.

Default: zero_entry = 'k'.

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

Input: specifies how entries with duplicate row and column indices are to be treated.

If `dupl_entry` = `'n'` or `'N'`, duplicates are not checked;

if `dupl_entry` = `'f'` or `'F'`, duplicates are removed (keep first);

if `dupl_entry` = `'l'` or `'L'`, duplicates are removed (keep last);

if `dupl_entry` = `'s'` or `'S'`, duplicates are summed;

if `dupl_entry` = `'e'` or `'E'`, on detecting a duplicate an error is raised.

Note: some Library procedures assume that there are no duplicate entries. If any of these procedures will be used you *must not* use `dupl_entry` = `'n'` unless you are sure that there are no duplicate entries.

Constraints: `dupl_entry` = `'n'`, `'N'`, `'f'`, `'F'`, `'l'`, `'L'`, `'s'`, `'S'`, `'e'` or `'E'`.

Default: `dupl_entry` = `'l'`.

ordered — logical, intent(in), optional

Input: specifies whether the entries in each row of the input data are ordered.

If `ordered` = `.false.`, column indices within each row are not ordered;

if `ordered` = `.true.`, each row is ordered in increasing column index.

Note: some Library procedures need the data ordered in a certain way. If the data is already ordered this information will help in avoiding unnecessary sorting.

Default: `ordered` = `.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
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Warnings (error%level = 1):

error%code	Description
101	Redundant optional argument. The optional argument `uplo` is not required for a general matrix. It will be ignored.
102	Redundant optional argument. The optional argument `unit_diag` is not allowed for a general matrix. It will be ignored.

5 Examples of Usage

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

Procedure: nag_sparse_mat_init_dia

1 Description

nag_sparse_mat_init_dia creates a structure containing the internal representation of the (real or complex) m by n sparse matrix A, where the entries of A are supplied in DIA format.

2 Usage

```
USE nag_sparse_mat
CALL nag_sparse_mat_init_dia(a,m,value,diag_indx  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the following cases:

Real data: a is of type nag_sparse_mat_real_*wp* and value is of type real(kind=*wp*).

Complex data: a is of type nag_sparse_mat_cmplx_*wp* and value is of type complex(kind=*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.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n\_diag \geq 1$ — the number of non-zero diagonals

3.1 Mandatory Arguments

a — 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 matrix A.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the sparse matrix may be extracted by calling nag_sparse_mat_extract.

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 4 of this module document.

m — integer, intent(in)

Input: m, the number of rows of the matrix A.

Constraints: m ≥ 1.

value($\min(m,n), n\_diag$**)** — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the columns of value *must* contain the non-zero diagonals of A.

diag_indx(*n_diag*) — integer, intent(in)

Input: the indices of the non-zero diagonals of A as they appear in `value`. For the case $m \leq n$, let `diag_indx(`i`)` $= j$:

$j = 0$ implies that the i^{th} column of `value` contains the main diagonal of A;

$j < 0$ implies that in the i^{th} column of `value`, the first $|j|$ elements are not used and the next $m - |j|$ elements are the $|j|^{th}$ diagonal below the main diagonal in A;

$j > 0$ implies that in the i^{th} column of `value`, the first $\min(m, n - j)$ elements are the j^{th} diagonal above main diagonal in A and the remaining elements are not used.

For the case $m > n$, let `diag_indx(`i`)` $= j$:

$j = 0$ implies that the i^{th} column of `value` contains the main diagonal of A;

$j < 0$ implies that in the i^{th} column of `value`, the first $n - \min(n, m - |j|)$ elements are not used and the next $\min(n, m - |j|)$ elements are the $|j|^{th}$ diagonal below the main diagonal in A;

$j > 0$ implies that in the i^{th} column of `value`, the first $n - j$ elements are the j^{th} diagonal above main diagonal in A and the remaining elements are not used.

Constraints: no repeated indices are allowed.

For a general matrix: $-(m-1) \leq$ `diag_indx`$(i) \leq (n-1)$, for $i = 1, 2, \ldots, n\_diag$;

for an upper matrix: $0 \leq$ `diag_indx`$(i) \leq (n-1)$, for $i = 1, 2, \ldots, n\_diag$;

for an upper matrix with unit diagonals: $1 \leq$ `diag_indx`$(i) \leq (n-1)$, for $i = 1, 2, \ldots, n\_diag$;

for a lower matrix: $-(m-1) \leq$ `diag_indx`$(i) \leq 0$, for $i = 1, 2, \ldots, n\_diag$;

for a lower matrix with unit diagonals: $-(m-1) \leq$ `diag_indx`$(i) \leq 1$, for $i = 1, 2, \ldots, n\_diag$.

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.

max_nnz — integer, intent(in), optional

Input: the maximum number of entries allowed in `a`. Using `max_nnz` $> nnz$ allows the addition of more elements to the matrix without the need to create another structure.

Constraints: `max_nnz` $\geq nnz$.

Default: `max_nnz` $= nnz$.

n — integer, intent(in), optional

Input: n, the number of columns of the matrix A.

Constraints: `n` ≥ 1.

Default: the matrix is square, `n` $=$ `m`.

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

Input: specifies the matrix type.

If `mat_type` $=$ `'g'` or `'G'`, the matrix is general;

if `mat_type` $=$ `'s'` or `'S'`, the matrix is symmetric;

if `mat_type` $=$ `'h'` or `'H'`, the matrix is Hermitian;

if `mat_type` $=$ `'t'` or `'T'`, the matrix is triangular.

Constraints: `mat_type` $=$ `'g'`, `'G'`, `'s'`, `'S'`, `'h'`, `'H'`, `'t'` or `'T'`.

Default: `mat_type` $=$ `'g'`.

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

Input: specifies whether A is upper or lower triangular. If A is symmetric or Hermitian, `uplo` specifies whether the upper or lower triangle is supplied.

If `uplo` = `'u'` or `'U'`, either A is upper triangular or the upper triangle of a symmetric or Hermitian matrix is supplied;

if `uplo` = `'l'` or `'L'`, either A is lower triangular or the lower triangle of a symmetric or Hermitian matrix is supplied.

Note: `uplo` will be ignored if `mat_type` = `'g'`.

Constraints: `uplo` = `'u'`, `'U'`, `'l'` or `'L'`.

Default: `uplo` = `'l'`.

unit_diag — logical, intent(in), optional

Input: specifies whether a triangular, symmetric or Hermitian matrix has unit diagonal elements.

If `unit_diag` = `.false.`, the diagonal elements are supplied with the input data;

if `unit_diag` = `.true.`, the diagonal elements are *not* supplied and assumed to be unity.

Default: `unit_diag` = `.false.`.

check_indx — logical, intent(in), optional

Input: specifies whether the input data is to be checked.

If `check_indx` = `.true.`, the input data will be checked;

if `check_indx` = `.false.`, *no* checks will be performed on the input data.

Note: `check_indx` = `.false.` *must not* be used unless all indices within the input data are correct. Other Library procedures assume that the data are correct and unexpected errors may occur if that is not the case.

Default: `check_indx` = `.true.`.

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.
320	The procedure was unable to allocate enough memory.

Warnings (error%level = 1):

error%code	Description
101	Redundant optional argument. The optional argument `uplo` is not required for a general matrix. It will be ignored.
102	Redundant optional argument. The optional argument `unit_diag` is not allowed for a general matrix. It will be ignored.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 4 of this module document.

Procedure: nag_sparse_mat_extract

1 Description

Given a structure of type nag_sparse_mat_real_*wp* or nag_sparse_mat_cmplx_*wp* representing a m by n sparse matrix A with nnz non-zero entries, this procedure returns the dimensions of the matrix, the number of the non-zero entries and other information describing the matrix. It also returns arrays containing the non-zero entries in either COO or CSR formats.

In general you may need to call this procedure twice. Once to obtain the value of, nnz followed by a subsequent call to obtain the non-zero entries (see the Examples of Usage section of this procedure).

2 Usage

```
USE nag_sparse_mat
CALL nag_sparse_mat_extract(a,m  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the following cases:

Real data: a is of type nag_sparse_mat_real_*wp* and value is of type real(kind=*wp*).

Complex data: a is of type nag_sparse_mat_cmplx_*wp* and value is of type complex(kind=*wp*).

3 Arguments

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.

m — integer, intent(out)
Output: m, the number of rows of A.

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.

nnz — integer, intent(out), optional
Output: the number of non-zero entries in a.

n — integer, intent(out), optional
Output: n, the number of columns of A.

value(*nnz*) — real(kind=*wp*) / complex(kind=*wp*), intent(out), optional
Output: the non-zero entries in a.

row_indx(*nnz*) — integer, intent(out), optional
col_indx(*nnz*) — integer, intent(out), optional
Output: the row and column indices of the non-zero entries in a. row_indx(i) and col_indx(i) contain the row and column indices of the non-zero entry stored in value(i), for $i = 1, 2, \ldots, nnz$.
Note: these are one-base arrays i.e., $1 \leq$ row_indx(:) $\leq m$ and $1 \leq$ col_indx(:) $\leq n$.

row_begin$(m+1)$ — integer, intent(out)

Output: the index of the first entry of each row of the input data. `row_begin`(i) gives the location (within the range $1 : nnz$) in `value` and `col_indx` of the first entry of row i. `row_begin`$(m+1)$ will contain $nnz + 1$. The number of non-zero entries in row i is given by `row_begin`$(i+1) -$ `row_begin`(i).

mat_type — character(len=1), intent(out), optional

Output: the matrix type.

If A is general, `mat_type` = `'g'`;

if A is symmetric, `mat_type` = `'s'`;

if A is Hermitian, `mat_type` = `'h'`;

if A is triangular, `mat_type` = `'t'`.

uplo — character(len=1), intent(out), optional

Output: specifies which half of the matrix A is returned.

If A is upper triangular or the upper triangle of a symmetric or Hermitian matrix is returned, `uplo` = `'u'`;

if A is lower triangular or the lower triangle of a symmetric or Hermitian matrix is returned, `uplo` = `'l'`.

Note: `uplo` must not be used if `mat_type` = `'g'`.

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.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 4 of this module document.

Assume that all relevant arguments have been declared correctly as described in Section 3, and that input and input/output arguments have been appropriately initialized. This example shows how to call this procedure first to find the number of non-zero entries and the matrix type then to return the non-zero entries and other matrix information.

```
CALL nag_sparse_mat_extract (a,m,nnz=nnz,mat_type=mat_type)
SELECT CASE (mat_type)
CASE ('g','G')   ! general matrix
  CALL nag_sparse_mat_extract (a,m,value=v(1:nnz),row_indx=row_indx(1:nnz), &
   col_indx=col_indx(1:nnz),n=n)
CASE default
  CALL nag_sparse_mat_extract (a,m,value=v(1:nnz),row_indx=row_indx(1:nnz), &
   col_indx=col_indx(1:nnz),uplo=uplo)
END SELECT
```

Procedure: nag_sparse_matvec

1 Description

nag_sparse_matvec is a generic procedure which computes the the matrix-vector multiplication, $y = Ax$, involving an m by n sparse matrix A.

The procedure also provides an option to compute the transposed matrix-vector multiplication

$$y = A^T x \quad \text{or} \quad y = A^C x \ (= A^T x \text{ if } A \text{ is real}).$$

Any duplicate entries found in A will be summed. It is also assumed that the indices of the non-zero entries of A are correct.

2 Usage

```
USE nag_sparse_mat
CALL nag_sparse_matvec(a,x,y  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the following cases:

Real data: a is of type nag_sparse_mat_real_*wp*, and x and y are of type real(kind=*wp*).

Complex data: a is of type nag_sparse_mat_cmplx_*wp*, and x and y are of type complex(kind=*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.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

k — the size of the input vector

l — the size of the output vector

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.

x(k) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the vector x.

Constraints: if trans is not present or trans = 'n', then $k = n$, otherwise $k = m$.

y(l) — real(kind=*wp*) / complex(kind=*wp*), intent(out)

Output: y, the result of the matrix-vector multiplication.

Constraints: y must be of the same type as x and if trans is not present or trans = 'n', then $l = m$, otherwise $l = n$.

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

Input: specifies whether the multiplication involves A, its transpose A^T or its conjugate-transpose A^C (= A^T if A is real).

If `trans` = `'n'` or `'N'`, then $y = Ax$;
if `trans` = `'t'` or `'T'`, then $y = A^T x$;
if `trans` = `'c'` or `'C'`, then $y = A^C x$.

Default: `trans` = `'n'`.

Constraints: `trans` = `'n'`, `'N'`, `'t'`, `'T'`, `'c'` or `'C'`.

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.
303	Array arguments have inconsistent shapes.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 2 of this module document.

Derived Type: nag_sparse_mat_real_*wp*

Note. The names of derived types containing real/complex components are precision dependent. For double precision the name of this type is `nag_sparse_mat_real_dp`. For single precision the name is `nag_sparse_mat_real_sp`. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

The derived type `nag_sparse_mat_real_`*wp* is used to represent a sparse real matrix. Some Library procedures return structures of this type suitable for passing to other procedures.

The generation procedures allocate storage to the pointer components of the structure. The amount of storage allocated for this type depends on the format implemented by the Library procedures.

If you wish to deallocate the storage when the structure is no longer required, you must call the generic deallocation procedure `nag_deallocate`, which is described in the Module Document `nag_lib_support`(1.1).

The generation procedures check whether the structure has already had storage allocated to it in a previous call; if it has, they deallocate that storage before allocating the storage for the new call.

The components of this type are private.

2 Type Definition

```
type nag_sparse_mat_real_wp
  private
  .
  .
  .
end type nag_sparse_mat_real_wp
```

3 Components

In order to reduce the risk of accidental data corruption the components of this type are private and may not be accessed directly.

The procedures `nag_sparse_mat_init_coo` and `nag_sparse_mat_init_csc` may be used to initialize structures of the type.

Derived Type: nag_sparse_mat_cmplx_*wp*

Note. The names of derived types containing real/complex components are precision dependent. For double precision the name of this type is `nag_sparse_mat_cmplx_dp`. For single precision the name is `nag_sparse_mat_cmplx_sp`. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

The derived type `nag_sparse_mat_cmplx_`*wp* is used to represent a sparse complex matrix. Some Library procedures return structures of this type suitable for passing to other procedures.

The generation procedures allocate storage to the pointer components of the structure. The amount of storage allocated for this type depends on the format implemented by the Library procedures.

If you wish to deallocate the storage when the structure is no longer required, you must call the generic deallocation procedure `nag_deallocate`, which is described in the Module Document `nag_lib_support`(1.1).

The generation procedures check whether the structure has already had storage allocated to it in a previous call; if it has, they deallocate that storage before allocating the storage for the new call.

The components of this type are private.

2 Type Definition

```
type nag_sparse_mat_cmplx_wp
  private
  .
  .
  .
end type nag_sparse_mat_cmplx_wp
```

3 Components

In order to reduce the risk of accidental data corruption the components of this type are private and may not be accessed directly.

The procedures `nag_sparse_mat_init_coo` and `nag_sparse_mat_init_csc` may be used to initialize structures of the type.

Example 1: Initialization of a Sparse Matrix Structure from Data in COO Format

This example shows how to create a sparse structure from real data in COO format. The structure is then used in a call to `nag_sparse_matvec` to return the result of multiplying the sparse matrix by a vector.

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_mat_ex01

  ! Example Program Text for nag_sparse_mat
  ! NAG fl90, 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_sparse_matvec, &
   nag_deallocate
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, m, n, nnz
  TYPE (nag_sparse_mat_real_wp) :: a
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col_indx(:), row_indx(:)
  REAL (wp), ALLOCATABLE :: value(:), x(:), y(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sparse_mat_ex01'

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

  ! allocate required arrays
  ALLOCATE (row_indx(nnz),col_indx(nnz),value(nnz),x(n),y(m))

  ! Read values and indices
  DO i = 1, nnz
    READ (nag_std_in,*) value(i), row_indx(i), col_indx(i)
  END DO

  ! Read vector x
  READ (nag_std_in,*) x(1:n)

  CALL nag_sparse_mat_init_coo(a,m,value,row_indx,col_indx,n=n, &
   ordered='F',check_indx=.FALSE.,dupl_entry='n')

  CALL nag_sparse_matvec(a,x,y)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) ' The output of:'
  WRITE (nag_std_out,*) &
   '   a- creating a real sparse matrix from data in COO format'
  WRITE (nag_std_out,*) &
```

```
       '   b- multiplying the structure created in a by a vector'
      WRITE (nag_std_out,'(8F6.1)') y(1:m)

      CALL nag_deallocate(a)
      DEALLOCATE (x,y,value,row_indx,col_indx)

    END PROGRAM nag_sparse_mat_ex01
```

2 Program Data

```
Example Program Data for nag_sparse_mat_ex01
  8   9         : m, n
 25             : nnz
  2.0   1    1  : value(1), row_indx(1), col_indx(1)
 -1.0   1    4
  1.0   1    9
  4.0   2    1
 -3.0   2    2
  2.0   2    5
 -7.0   3    3
  2.0   3    6
  3.0   4    1
 -4.0   4    3
  5.0   4    4
  5.0   4    7
 -1.0   5    2
  8.0   5    5
 -3.0   5    7
 -6.0   6    1
  5.0   6    3
  2.0   6    6
 -5.0   7    3
 -1.0   7    5
  6.0   7    7
 -1.0   8    2
  2.0   8    6
  3.0   8    8
  6.0   8    9  : value(nnz), row_indx(nnz), col_indx(nnz)

  6.0   8.0  -9.0  -6.0  7.0  1.0  2.0  4.0  3.0  : x
```

3 Program Results

```
 Example Program Results for nag_sparse_mat_ex01

  The output of:
    a- creating a real sparse matrix from data in COO format
    b- multiplying the structure created in a by a vector
  21.0  14.0  65.0  34.0  42.0 -79.0  50.0  24.0
```

Example 2: Initialization of a Sparse Matrix Structure from Data in CSC Format

This example shows how to create a sparse structure from complex data in CSC format. The structure is then used in a call to `nag_sparse_matvec` to return the result of multiplying the sparse matrix by a vector.

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_mat_ex02

  ! Example Program Text for nag_sparse_mat
  ! 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_cmplx_wp => nag_sparse_mat_cmplx_dp, nag_sparse_matvec, &
   nag_deallocate
  ! .. 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_cmplx_wp) :: a
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col_begin(:), row_indx(:)
  COMPLEX (wp), ALLOCATABLE :: value(:), x(:), y(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sparse_mat_ex02'

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

  ! allocate required arrays
  ALLOCATE (row_indx(nnz),col_begin(n),value(nnz),x(n),y(n))

  ! Read values and row indices
  DO i = 1, nnz
    READ (nag_std_in,*) value(i), row_indx(i)
  END DO

  ! Read columns begin
  READ (nag_std_in,*) col_begin(1:n)

  ! Read vector x
  READ (nag_std_in,*) x(1:n)

  CALL nag_sparse_mat_init_csc(a,value,row_indx,col_begin,mat_type='h')

  CALL nag_sparse_matvec(a,x,y)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) ' The output of:'
  WRITE (nag_std_out,*) &
```

```
      '   a- creating a Hermitian sparse matrix from data in CSC format'
      WRITE (nag_std_out,*) &
       '   b- multiplying the structure created in a by a vector'
      WRITE (nag_std_out,'(3X,''('',F5.1,'','',F5.1,'')'')') y(1:n)

      CALL nag_deallocate(a)
      DEALLOCATE (x,y,value,row_indx,col_begin)

    END PROGRAM nag_sparse_mat_ex02
```

2 Program Data

```
Example Program Data for nag_sparse_mat_ex02
  7             : n
 11             : nnz
 ( 2.0,-1.0)  2 : value(1), row_indx(1)
 (-1.0, 1.0)  5
 ( 0.0, 2.0)  6
 ( 1.0, 0.0)  2
 (-2.0, 0.0)  3
 ( 4.0,-3.0)  7
 (-3.0, 3.0)  4
 ( 1.0,-2.0)  7
 ( 5.0, 0.0)  5
 ( 2.0, 0.0)  6
 ( 3.0, 0.0)  7 : value(nnz), row_indx(nnz)

 1  3  5  8  8  10  11  : col_begin

 ( 1.0, 0.0)  ( 2.0, 1.0)  ( 0.0, 3.0)  ( 4.0, 2.0)
 ( 1.0, 1.0)  ( 0.0, 2.0)  ( 3.0, 2.0)  : x
```

3 Program Results

```
 Example Program Results for nag_sparse_mat_ex02

  The output of:
    a- creating a Hermitian sparse matrix from data in CSC format
    b- multiplying the structure created in a by a vector
   (  3.0,  2.0)
   (  8.0,  0.0)
   (  0.0, -7.0)
   ( -9.0, -9.0)
   (  3.0, 14.0)
   ( -2.0,  8.0)
   ( 21.0, 17.0)
```

Example 3: Initialization of a Sparse Matrix Structure from Data in CSR Format

This example shows how to create a sparse structure from complex data in CSR format. The structure is then used in a call to `nag_sparse_matvec` to return the result of multiplying the sparse matrix by a vector.

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_mat_ex03

  ! Example Program Text for nag_sparse_mat
  ! 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_csr, &
   nag_sparse_mat_cmplx_wp => nag_sparse_mat_cmplx_dp, nag_sparse_matvec, &
   nag_deallocate
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, m, n, nnz
  TYPE (nag_sparse_mat_cmplx_wp) :: a
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col_indx(:), row_begin(:)
  COMPLEX (wp), ALLOCATABLE :: value(:), x(:), y(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sparse_mat_ex03'

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

  ! allocate required arrays
  ALLOCATE (col_indx(nnz),row_begin(m),value(nnz),x(m),y(n))

  ! Read values and column indices
  DO i = 1, nnz
    READ (nag_std_in,*) value(i), col_indx(i)
  END DO

  ! Read row begin
  READ (nag_std_in,*) row_begin(1:m)

  ! Read vector x
  READ (nag_std_in,*) x(1:m)

  CALL nag_sparse_mat_init_csr(a,value,col_indx,row_begin,n=n)

  CALL nag_sparse_matvec(a,x,y,trans='c')

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) ' The output of:'
  WRITE (nag_std_out,*) &
```

```
       '   a- creating a complex sparse matrix from data in CSR format'
      WRITE (nag_std_out,*) &
       '   b- multiplying the structure created in a by a vector'
      WRITE (nag_std_out,'(3X,''('',F5.1,'','',F5.1,'')'')') y(1:n)

      CALL nag_deallocate(a)
      DEALLOCATE (x,y,value,row_begin,col_indx)

    END PROGRAM nag_sparse_mat_ex03
```

2 Program Data

```
Example Program Data for nag_sparse_mat_ex03
  9 8           : m, n
 16             : nnz
 ( 2.0,-1.0)  6 : value(1), col_indx(1)
 (-1.0, 1.0)  3
 ( 0.0, 2.0)  1
 ( 1.0, 0.0)  8
 (-2.0, 0.0)  3
 ( 4.0,-3.0)  7
 (-3.0, 3.0)  5
 ( 1.0,-2.0)  1
 ( 5.0, 0.0)  6
 ( 2.0, 1.0)  2
 ( 1.0, 3.0)  7
 ( 3.0,-2.0)  5
 ( 1.0,-2.0)  1
 ( 2.0, 1.0)  3
 ( 2.0, 3.0)  2
 ( 3.0, 0.0)  8 : value(nnz), col_indx(nnz)

 1  3  5  8  10  12  14  14  15  : row_begin

 ( 2.0, 1.0)  ( 0.0, 1.0)  ( 2.0, 1.0)  ( 0.0, 2.0)  ( 1.0, 2.0)
 ( 1.0, 0.0)  ( 1.0, 2.0)  ( 4.0, 1.0)  ( 2.0, 2.0)  : x
```

3 Program Results

```
 Example Program Results for nag_sparse_mat_ex03

  The output of:
    a- creating a complex sparse matrix from data in CSR format
    b- multiplying the structure created in a by a vector
   ( -1.0,  4.0)
   ( 14.0,  1.0)
   (  4.0, -7.0)
   (  0.0,  0.0)
   (  0.0, -7.0)
   (  3.0, 14.0)
   ( 12.0,  9.0)
   (  6.0,  7.0)
```

Example 4: Initialization of a Sparse Matrix Structure from Data in DIA Format

This example shows how to create a sparse structure from complex data in DIA format. The components of the structure is then extracted (in the coordinate format) using `nag_sparse_mat_extract`.

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_mat_ex04

  ! Example Program Text for nag_sparse_mat
  ! 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_dia, &
   nag_sparse_mat_real_wp => nag_sparse_mat_real_dp, &
   nag_sparse_mat_extract, nag_deallocate
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC ABS, KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, m, nnz, n_diag, ret_m
  TYPE (nag_sparse_mat_real_wp) :: a
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: diag_indx(:), ret_col_indx(:), ret_row_indx(:)
  REAL (wp), ALLOCATABLE :: ret_value(:), value(:,:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sparse_mat_ex04'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) m, n_diag

  ! allocate required arrays
  ALLOCATE (diag_indx(n_diag),value(m,n_diag))

  ! Read values and indices
  DO i = 1, n_diag
    READ (nag_std_in,*) diag_indx(i)
    IF (diag_indx(i)<=0) THEN
      READ (nag_std_in,*) value(ABS(diag_indx(i))+1:,i)
    ELSE
      READ (nag_std_in,*) value(:m-diag_indx(i),i)
    END IF
  END DO

  CALL nag_sparse_mat_init_dia(a,m,value,diag_indx)

  CALL nag_sparse_mat_extract(a,ret_m,nnz=nnz)

  ! allocate required arrays
  ALLOCATE (ret_row_indx(nnz),ret_col_indx(nnz),ret_value(nnz))
  CALL nag_sparse_mat_extract(a,ret_m,value=ret_value, &
   row_indx=ret_row_indx,col_indx=ret_col_indx)

  WRITE (nag_std_out,*)
```

```
      WRITE (nag_std_out,*) ' The output of nag_sparse_mat_extract &
       &(the sparse matrix in COO format):'
      WRITE (nag_std_out,*)
      WRITE (nag_std_out,*) '   row index    col. index    value'
      DO i = 1, nnz
        WRITE (nag_std_out,'(I9,I14,F12.1)') ret_row_indx(i), ret_col_indx(i), &
         ret_value(i)
      END DO

      CALL nag_deallocate(a)
      DEALLOCATE (value,diag_indx,ret_row_indx,ret_col_indx,ret_value)

    END PROGRAM nag_sparse_mat_ex04
```

2 Program Data

```
Example Program Data for nag_sparse_mat_ex01
  8   3                          : m, n_diag
  2                              : diag_indx(1)
   1.3 -2.4  3.5  4.6 -5.7  6.8
 -3                              : diag_indx(2)
                 4.1  5.2 -6.3  7.4 -8.5
  0                              : diag_indx(3)
   1.1 -2.2  3.3  4.4 -5.5  6.6  7.7  8.8
```

3 Program Results

```
 Example Program Results for nag_sparse_mat_ex04

  The output of nag_sparse_mat_extract (the sparse matrix in COO format):

    row index    col. index    value
        1             1         1.1
        1             3         1.3
        2             2        -2.2
        2             4        -2.4
        3             3         3.3
        3             5         3.5
        4             1         4.1
        4             4         4.4
        4             6         4.6
        5             2         5.2
        5             5        -5.5
        5             7        -5.7
        6             3        -6.3
        6             6         6.6
        6             8         6.8
        7             4         7.4
        7             7         7.7
        8             5        -8.5
        8             8         8.8
```