Chapter 4
Matrix and Vector Operations

1 Scope of the Chapter

This chapter provides procedures for matrix and vector operations.

This chapter (and Chapters 5 and 6) can handle general matrices, matrices with special structure and sparse matrices. Using the special structure of matrices offers possibilities for greater efficiency, more economical storage and increased reliability.

All the procedures in this chapter are generic procedures which can handle either real or complex data.

2 Available Modules

Module 4.1: `nag_mat_norm` — **Matrix norms**

Provides procedures to compute the 1-norm, ∞-norm, Frobenius (Euclidean) norm or the element of largest absolute value, of a real or complex matrix. It caters for different types of matrices and storage schemes.

Module 4.2: `nag_mat_inv` — **Matrix inversion**

provides procedures for matrix inversion.

Module 4.3: `nag_sparse_mat` — **Sparse matrix utilities**

provides procedures for initialization and manipulation of sparse matrices.

3 Storage of Matrices

In this section we assume that A is a matrix and `a` is the corresponding argument. For symmetric or triangular matrices, it is assumed that the argument `uplo` is used to specify that the elements of either the upper or the lower triangle are referenced.

3.1 Symmetric Matrices

There are two storage schemes for the symmetric or Hermitian matrix A: conventional storage or packed storage. The choice is determined by the rank of the corresponding argument `a`.

Conventional storage

`a` is a rank-2 array, of shape (n,n). Matrix element a_{ij} is stored in `a`(i,j). Only the elements of either the upper or the lower triangle need be stored in `a` as specified by the argument `uplo`; the remaining elements need not be set.

This storage scheme is more straightforward and carries less risk of user error than packed storage; on some machines it may result in more efficient execution. It requires almost twice as much memory as packed storage, although the unused triangle of `a` may be used to store other data.

Packed storage

`a` is a rank-1 array of shape $(n(n+1)/2)$. The elements of either the upper or the lower triangle of A, as specified by `uplo`, are packed by *columns* into contiguous elements of `a`.

Packed storage is more economical in use of memory than conventional storage, but may result in less efficient execution on some machines.

The details of packed storage are as follows:

- if `uplo` = `'u'` or `'U'`, a_{ij} is stored in `a(`$i + j(j-1)/2$`)`, for $i \le j$;
- if `uplo` = `'l'` or `'L'`, a_{ij} is stored in `a(`$i + (2n-j)(j-1)/2$`)`, for $i \ge j$.

For example,

`uplo`	Hermitian Matrix A	Packed storage in array `a`
`'u'` or `'U'`	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \overline{a}_{12} & a_{22} & a_{23} & a_{24} \\ \overline{a}_{13} & \overline{a}_{23} & a_{33} & a_{34} \\ \overline{a}_{14} & \overline{a}_{24} & \overline{a}_{34} & a_{44} \end{pmatrix}$	$a_{11} \quad \underbrace{a_{12}\,a_{22}} \quad \underbrace{a_{13}\,a_{23}\,a_{33}} \quad \underbrace{a_{14}\,a_{24}\,a_{34}\,a_{44}}$
`'l'` or `'L'`	$\begin{pmatrix} a_{11} & \overline{a}_{21} & \overline{a}_{31} & \overline{a}_{41} \\ a_{21} & a_{22} & \overline{a}_{32} & \overline{a}_{42} \\ a_{31} & a_{32} & a_{33} & \overline{a}_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}\,a_{21}\,a_{31}\,a_{41}} \quad \underbrace{a_{22}\,a_{32}\,a_{42}} \quad \underbrace{a_{33}\,a_{43}} \quad a_{44}$

Note that for symmetric matrices, packing the upper triangle by columns is equivalent to packing the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the upper triangle by rows. For Hermitian matrices, packing the upper triangle by columns is equivalent to packing the conjugate of the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the conjugate of the upper triangle by rows.

3.2 Triangular Matrices

There are two storage schemes for the triangular matrix A: conventional storage or packed storage. The choice is determined by the rank of the corresponding argument `a`.

Conventional storage

`a` is a rank-2 array, of shape (n,n). Matrix element a_{ij} is stored in `a(`i,j`)`. If A is upper triangular, only the elements of the upper triangle ($i \le j$) need be stored; if A is lower triangular, only the elements of the lower triangle ($i \ge j$) need be stored; the remaining elements of `a` need not be set.

This storage scheme is more straightforward and carries less risk of user error than packed storage; on some machines it may result in more efficient execution. It requires almost twice as much memory as packed storage, although the other triangle of `a` may be used to store other data.

Packed storage

`a` is a rank-1 array of shape $(n(n+1)/2)$. The elements of either the upper or the lower triangle of A, as specified by `uplo`, are packed by *columns* into contiguous elements of `a`.

Packed storage is more economical in use of memory than conventional storage, but may result in less efficient execution on some machines.

The details of packed storage are as follows:

- if `uplo` = `'u'` or `'U'`, a_{ij} is stored in `a(`$i + j(j-1)/2$`)`, for $i \le j$;
- if `uplo` = `'l'` or `'L'`, a_{ij} is stored in `a(`$i + (2n-j)(j-1)/2$`)`, for $i \ge j$.

For example,

`uplo`	Triangular Matrix A	Packed storage in array `a`
`'u'` or `'U'`	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$a_{11} \quad \underbrace{a_{12}\, a_{22}} \quad \underbrace{a_{13}\, a_{23}\, a_{33}} \quad \underbrace{a_{14}\, a_{24}\, a_{34}\, a_{44}}$
`'l'` or `'L'`	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}\, a_{21}\, a_{31}\, a_{41}} \quad \underbrace{a_{22}\, a_{32}\, a_{42}} \quad \underbrace{a_{33}\, a_{43}} \quad a_{44}$

Unit triangular matrices

A *unit* triangular matrix is a triangular matrix whose diagonal elements are known to be unity. Some procedures have an optional argument `unit_diag` which can be used to specify that the matrix is unit triangular, and then the diagonal elements do not need to be stored; the storage of the other elements of the matrix is not affected.

3.3 Square Banded Matrices

The following storage scheme is used for the general band matrix A with k_l sub-diagonals and k_u super-diagonals:

- a_{ij} is stored in `a(`$k_u + i - j + 1, j$`)`, for $\max(j - k_u, 1) \leq i \leq \min(j + k_l, n)$.

For example,

General band matrix A	Band storage in array `a`
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

The elements marked by * in the upper left and lower right corners of `a` are not referenced and need not be set.

3.4 Symmetric Banded Matrices

The following storage scheme is used for the symmetric or Hermitian band matrix A with k super-diagonals or sub-diagonals:

- if `uplo` = `'u'` or `'U'`, a_{ij} is stored in `a(`$k + i - j + 1, j$`)`, for $\max(j - k, 1) \leq i \leq j$;
- if `uplo` = `'l'` or `'L'`, a_{ij} is stored in `a(`$i - j + 1, j$`)`, for $j \leq i \leq \min(j + k, n)$.

For example,

uplo	Hermitian band matrix A	Band storage in array `a`
`'u'` or `'U'`	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ \overline{a}_{12} & a_{22} & a_{23} & a_{24} & \\ \overline{a}_{13} & \overline{a}_{23} & a_{33} & a_{34} & a_{35} \\ & \overline{a}_{24} & \overline{a}_{34} & a_{44} & a_{45} \\ & & \overline{a}_{35} & \overline{a}_{45} & a_{55} \end{pmatrix}$	$\begin{matrix} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \end{matrix}$
`'l'` or `'L'`	$\begin{pmatrix} a_{11} & \overline{a}_{21} & \overline{a}_{31} & & \\ a_{21} & a_{22} & \overline{a}_{32} & \overline{a}_{42} & \\ a_{31} & a_{32} & a_{33} & \overline{a}_{43} & \overline{a}_{53} \\ & a_{42} & a_{43} & a_{44} & \overline{a}_{54} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

The elements marked by * in the upper left and lower right corners of `a` are not referenced and need not be set.

3.5 Triangular Banded Matrices

The following storage scheme is used for the triangular band matrix A with k super-diagonals or sub-diagonals:

- if `uplo` = `'u'` or `'U'`, a_{ij} is stored in `a(`$k+i-j+1,j$`)`, for $\max(j-k,1) \leq i \leq j$;
- if `uplo` = `'l'` or `'L'`, a_{ij} is stored in `a(`$i-j+1,j$`)`, for $j \leq i \leq \min(j+k,n)$.

For example,

uplo	Triangular band matrix A	Band storage in array `a`
`'u'` or `'U'`	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ & a_{22} & a_{23} & a_{24} & \\ & & a_{33} & a_{34} & a_{35} \\ & & & a_{44} & a_{45} \\ & & & & a_{55} \end{pmatrix}$	$\begin{matrix} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \end{matrix}$
`'l'` or `'L'`	$\begin{pmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ & a_{42} & a_{43} & a_{44} & \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

The elements marked by * in the upper left and lower right corners of `a` are not referenced and need not be set.

Unit triangular matrices

A *unit* triangular banded matrix is a triangular banded matrix whose diagonal elements are known to be unity. Some procedures have an optional argument `unit_diag` which can be used to specify that the matrix is unit triangular banded, and then the diagonal elements do not need to be stored; the storage of the other elements of the matrix is not affected.

4 Sparse Vectors

A vector is *sparse* when it is beneficial to identify which entries have non-zero values. Sparse vectors are represented by a pair of conventional vectors, one denoting the non-zero values and the other denoting the indices. That is, if a is a sparse vector, then it is represented by a one-dimensional array of the non-zero entries of a and an integer vector of equal length whose values indicate the location in a of the corresponding floating point value. For example, the sparse vector

$$a = \begin{pmatrix} 11.0 & 0.0 & 13.0 & 14.0 & 0.0 \end{pmatrix}$$

can be represented by two vectors as

$$\begin{array}{rl} \mathit{value} = & \begin{pmatrix} 11.0 & 13.0 & 14.0 \end{pmatrix} \\ \mathit{indx} = & \begin{pmatrix} 1 & 3 & 4 \end{pmatrix} \end{array}$$

4.1 Storage Scheme

Two derived types `nag_sparse_vec_real_`*wp* and `nag_sparse_vec_cmplx_`*wp* are defined by the Library to store real and complex sparse vectors. A structure of one of the derived types will contain all the information needed to define the sparse vector. The module `nag_sparse_vec` contains the definition of the derived types and procedures to manipulate these types.

5 Sparse Matrices

A matrix is *sparse* when it is beneficial to identify which entries have non-zero values. Many problems arising from engineering and scientific computing require the solution of large, sparse systems, hence their importance in numerical linear algebra. Typically, sparse matrices provide an opportunity to conserve storage and reduce computational requirements by storing only the significant (typically, non-zero) entries.
The sparse matrix formats include:

Point entry:

COO - Coordinate

CSC - Compressed sparse column

CSR - Compressed sparse row

DIA - Sparse diagonal

Block entry:

BCO - Block coordinate

BSC - Block compressed sparse column

BSR - Block compressed sparse row

BDI - Block sparse diagonal

VBR - Variable block compressed sparse row

Each of these formats is intended to support an important class of problems or algorithms. The following is a list of the point entry data structure and one or more of its intended uses. The intended uses for block entry data structures are analogous, with the added property that they exploit the property of multiple unknowns per grid point or related properties.

COO - Coordinate: Most flexible data structure when constructing or modifying a sparse matrix.

CSC - Compressed sparse column: Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.

CSR - Compressed sparse row: Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.

DIA - Sparse diagonal: Particularly useful for matrices coming from finite difference approximations to partial differential equations on uniform grids.

VBR - Variable Block Row: It is often the case for problems with multiple unknowns per node, that the number of unknowns per node will vary from node to node, generating a block entry matrix with a block structure where the size of the blocks varies correspondingly. If the variation in size is small, it may be advantageous to make all block rows the same size by adding identity equations to smaller blocks. However, this is an unnatural restriction and may lead to significant wasted storage space if the variation is large. For this situation, we provide the variable block compressed sparse row (VBR) data structure.

The VBR data structure handles variations in block size in an efficient and natural way. Also, in addition to supporting natural variations due to problem characteristics, VBR allows for the agglomeration of neighbouring nodes, or the splitting of equations at a node in a natural way. This can be an important capability for developing robust preconditioners. Although more difficult to work with than constant block entry data structures like BSR, VBR is essential for application that have multiphysics capabilities or other complicated coupled equation formulations.

Point entry data structures

In this section we describe the supported point entry data structures for an m by n matrix A.

5.1 COO - Coordinate

The point-entry form of coordinate storage (COO) stores the entries of the matrix, along with their corresponding row and column indices. Three arrays are required for the COO format:

value - a scalar array of length *nnz* consisting of non-zero the entries of A, in any order.

row_indx - an integer array of length *nnz* consisting of the row indices of the entries in *value*.

col_indx - an integer array of length *nnz* consisting of the column indices of the entries in *value*.

For example, suppose

$$A = \begin{pmatrix} 11 & 0 & 13 & 14 & 0 \\ 0 & 0 & 23 & 24 & 0 \\ 31 & 32 & 33 & 34 & 0 \\ 0 & 42 & 0 & 44 & 0 \\ 51 & 52 & 0 & 0 & 55 \end{pmatrix}, \tag{1}$$

then one representation of A in COO format is:

$$\begin{array}{rlcccccccccccccl} value = & (& 11 & 51 & 31 & 32 & 34 & 52 & 13 & 23 & 33 & 14 & 24 & 42 & 55 & 44 &), \\ row\_indx = & (& 1 & 5 & 3 & 3 & 3 & 5 & 1 & 2 & 3 & 1 & 2 & 4 & 5 & 4 &), \\ col\_indx = & (& 1 & 1 & 1 & 2 & 4 & 2 & 3 & 3 & 3 & 4 & 4 & 2 & 5 & 4 &). \end{array}$$

It is possible to assert that index values are zero-based instead of one-based. In this case, one representation of A is:

$$\begin{array}{rlcccccccccccccl} value = & (& 11 & 51 & 31 & 32 & 34 & 52 & 13 & 23 & 33 & 14 & 24 & 42 & 55 & 44 &), \\ row\_indx = & (& 0 & 4 & 2 & 2 & 2 & 4 & 0 & 1 & 2 & 0 & 1 & 3 & 4 & 3 &), \\ col\_indx = & (& 0 & 0 & 0 & 1 & 3 & 1 & 2 & 2 & 2 & 3 & 3 & 1 & 4 & 3 &). \end{array}$$

If A is symmetric (or Hermitian or triangular) then we only need to store the lower (or upper) triangle. In this case one representation of the lower triangle of A is:

$$\begin{array}{rlccccccccccl} value = & (& 11 & 51 & 31 & 32 & 52 & 33 & 42 & 55 & 44 &), \\ row\_indx = & (& 1 & 5 & 3 & 3 & 5 & 3 & 4 & 5 & 4 &), \\ col\_indx = & (& 1 & 1 & 1 & 2 & 2 & 3 & 2 & 5 & 4 &). \end{array}$$

If A is Hermitian and the diagonal of A is stored, then we will assume that the imaginary part of the diagonal is zero.

5.2 CSC - Compressed Sparse Column

The point-entry form of compressed sparse column storage (CSC) stores the matrix entries in each of the columns A_{*j} as a sparse vector. A matrix A is stored in the CSC format using four arrays.

value - a scalar array of length *nnz* consisting of non-zero the entries of A stored column by column, in any order:

$$value = (A_{*\sigma(1)}, A_{*\sigma(2)}, \ldots, A_{*\sigma(n)}).$$

row_indx - an integer array of length *nnz* consisting of the row indices of the entries in *value*. As in the COO format, this can be one-based or zero-based.

col_begin - an integer array of length n such that $col\_begin(j) - \alpha + 1$, where $\alpha = \min_{i=1}^{n} col\_begin(i)$, points to the location (within the range $1 : nnz$) in *value* and *row_indx* of the first element of column j.

col_end - an integer array of length n such that $col\_end(j) - \alpha$, where $\alpha = \min_{i=1}^{n} col\_begin(i)$, points to the location (within the range $1 : nnz$) in *value* and *row_indx* of the last element of column j.

For example, one CSC representation of the matrix in (1) would be:

$$\begin{array}{rcccccccccccccccl}
value = & (& 11 & 31 & 51 & 32 & 42 & 52 & 13 & 23 & 33 & 14 & 24 & 34 & 44 & 55 &), \\
row\_indx = & (& 1 & 3 & 5 & 3 & 4 & 5 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 5 &), \\
col\_begin = & (& 1 & 4 & 7 & 10 & 14 &), \\
col\_end = & (& 4 & 7 & 10 & 14 & 15 &),
\end{array}$$

The mapping $\sigma()$ used above is a permutation of the first n integers. Much of the time $\sigma()$ will be the identity mapping. However, for technical accuracy we wanted to indicate that the columns of A can be arranged in any order within *value*.

The actual values in *col_begin* and *col_end* are not important, only their relative position from $\alpha = \min_{i=1}^{n} col\_begin(i)$. This allows greater flexibility for the user. In particular, it is common to construct pointer arrays starting at 0. Also, again we are able to assert a zero-based index vector. For (1), we could then define *row_indx*, *col_begin* and *col_end* as

$$\begin{array}{rcccccccccccccccl}
row\_indx = & (& 0 & 2 & 4 & 2 & 3 & 4 & 0 & 1 & 2 & 0 & 1 & 2 & 3 & 4 &), \\
col\_begin = & (& 0 & 3 & 6 & 9 & 13 &), \\
col\_end = & (& 3 & 6 & 9 & 13 & 14 &).
\end{array}$$

If column j is empty, then $col\_begin(j) = col\_end(j)$.

If the columns are in consecutive order, one can represent *col_end* in terms of *col_begin*. In this case $col\_end(1 : n-1) = col\_begin(2 : n)$ and $col\_end(n) = nnz + 1$

The two-array approach to pointers offers much more flexibility than the one array approach. For example, if we define the array

$$diag\_ptr = (\; 1 \quad 4 \quad 9 \quad 13 \quad 14 \;),$$

to point to the diagonal elements, then letting $col\_begin = diag\_ptr$ we can use just the lower triangular part of the general CSC representation without modifying or copying the other data structures.

A second example of increased flexibility is that there is no longer an implicit storage association between contiguous columns. The columns can be specified in any order. In particular, for the example above it is possible to put column one as the last column stored:

$$\begin{array}{rcccccccccccccccl}
value = & (& 32 & 42 & 52 & 13 & 23 & 33 & 14 & 24 & 34 & 44 & 55 & 11 & 31 & 51 &), \\
row\_indx = & (& 3 & 4 & 5 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 5 & 1 & 3 & 5 &), \\
col\_begin = & (& 12 & 1 & 4 & 7 & 11 &), \\
col\_end = & (& 15 & 4 & 7 & 11 & 12 &),
\end{array}$$

If A is symmetric then we only need to store the lower (or upper) triangle. In this case we have (for the lower triangle)

$$\begin{array}{rlllllllllll} \textit{value} = & (& 11 & 31 & 51 & 32 & 42 & 52 & 33 & 44 & 55 &), \\ \textit{row\_indx} = & (& 1 & 3 & 5 & 3 & 4 & 5 & 3 & 4 & 5 &), \\ \textit{col\_begin} = & (& 1 & 4 & 7 & 8 & 9 &), \\ \textit{col\_end} = & (& 4 & 7 & 8 & 9 & 10 &). \end{array}$$

5.3 CSR - Compressed Sparse Row

The point-entry form of compressed sparse row storage (CSR) stores the matrix entries in each of the rows A_{i*} as a sparse vector. A matrix A is stored in the CSR format using four arrays.

value - a scalar array of length *nnz* consisting of non-zero the entries of A stored row by row, in any order:

$$\textit{value} = (A_{\sigma(1)*}, A_{\sigma(2)*}, \ldots, A_{\sigma(n)*}).$$

col_indx - an integer array of length *nnz* consisting of the column indices of the entries in *value*. As in the COO format, this can be one-based or zero-based.

row_begin - an integer array of length m such that $\textit{row\_begin}(i) - \alpha + 1$, where $\alpha = \min_{j=1}^{m} \textit{row\_begin}(j)$, points to the location (within the range $1 : nnz$) in *value* and *col_indx* of the first element of row i.

row_end - an integer array of length m such that $\textit{row\_end}(i) - \alpha$, where $\alpha = \min_{j=1}^{m} \textit{row\_begin}(j)$, points to the location (within the range $1 : nnz$) in *value* and *col_indx* of the last element of row i.

For example, one CSR representation of the matrix in (1) would be:

$$\begin{array}{rlllllllllllllll} \textit{value} = & (& 11 & 13 & 14 & 23 & 24 & 31 & 32 & 33 & 34 & 42 & 44 & 51 & 52 & 55 &), \\ \textit{col\_indx} = & (& 1 & 3 & 4 & 3 & 4 & 1 & 2 & 3 & 4 & 2 & 4 & 1 & 2 & 5 &), \\ \textit{row\_begin} = & (& 1 & 4 & 6 & 10 & 12 &), \\ \textit{row\_end} = & (& 4 & 6 & 10 & 12 & 15 &), \end{array}$$

The discussion on the use $\sigma()$, *row_begin* and *row_end* follows that of the use of $\sigma()$, *col_begin* and *col_end* of the CSC format.
If A is symmetric then we only need to store the lower (or upper) triangle. In this case we have (for the lower triangle)

$$\begin{array}{rlllllllllll} \textit{value} = & (& 11 & 31 & 32 & 33 & 42 & 44 & 51 & 52 & 55 &), \\ \textit{col\_indx} = & (& 1 & 1 & 2 & 3 & 2 & 4 & 1 & 2 & 5 &), \\ \textit{row\_begin} = & (& 1 & 2 & 2 & 5 & 7 &), \\ \textit{row\_end} = & (& 2 & 2 & 5 & 7 & 10 &). \end{array}$$

5.4 DIA - Sparse Diagonal (Point entry form)

The point-entry form of diagonal storage (DIA) stores each supplied diagonal of A along with its position relative to the main diagonal. Let $l = \min(m, n)$, and let *n_diag* denote the number of non-zero diagonals of A. Two arrays are required for the DIA format:

value - a two-dimensional $(l, \textit{n\_diag})$ scalar array consisting of the *n_diag* non-zero diagonals of A in any order.

diag_indx - an integer array of length *n_diag* consisting of the corresponding indices of the nonzero diagonals of A in *value*. Thus, if $\textit{diag\_indx}(j) = i$ then the j^{th} column of *value* contains the i^{th} diagonal of A.

For example, let

$$A = \begin{pmatrix} 11 & 0 & 13 & 0 & 0 \\ 21 & 0 & 0 & 24 & 0 \\ 31 & 32 & 33 & 0 & 35 \\ 0 & 42 & 0 & 44 & 0 \\ 0 & 0 & 53 & 0 & 55 \end{pmatrix},$$

then A would be stored in DIA format as follows.

$$value = \begin{pmatrix} * & * & 11 & 13 \\ * & 21 & 0 & 24 \\ 31 & 32 & 33 & 35 \\ 42 & 0 & 44 & * \\ 53 & 0 & 55 & * \end{pmatrix},$$

The elements marked by * are not referenced and need not be set.

$$diag\_indx = \begin{pmatrix} -2 & -1 & 0 & 2 \end{pmatrix}.$$

Let $diag\_indx(i) = j$, then $j = 0$, implies that $value(:, i)$ (the i^{th} column of $value$) contains the main diagonal of A;

For $m \leq n$, then:

$j < 0$, implies that in the column $value(:, i)$, 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 column $value(:, i)$, 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 $m > n$, then:

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

If A is symmetric or Hermitian only the diagonals of lower (or upper) triangle of A are stored.

5.5 Storage Scheme

Two derived types `nag_sparse_mat_real_`*wp* and `nag_sparse_mat_cmplx_`*wp* are defined by the Library to store real and complex sparse matrices. A structure of one of the derived types will contain all the information needed to define the sparse matrix. The module `nag_sparse_mat` contains the definition of the derived types and procedures to manipulate these types.