Module 1.4: nag_sort
Sorting

nag_sort is concerned with sorting numeric or character data.

Contents

Introduction

This module is concerned with sorting numeric or character data. It handles only the simplest types of data structure and it is concerned only with *internal* sorting — that is, sorting a set of data which can all be stored within the program.

If you wish to sort large files of data or complex data structures you should use a comprehensive sorting program or package.

Procedure: nag_sort_vec

1 Description

`nag_sort_vec` is a procedure which rearranges a vector of integer or real numbers into ascending or descending order, or rearranges a vector of character data, such that a specified substring is in ASCII or reverse ASCII sequence.

For character data, only a substring of each element of the vector is used to determine the sorted order, but the entire elements are rearranged into sorted order.

2 Usage

```
USE nag_sort
CALL nag_sort_vec(a  [, 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.

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

n — the number of data elements

3.1 Mandatory Argument

a(n) — integer / real(kind=*wp*) / character(len=*), intent(inout)

Input: the data to be sorted.

Output: the sorted data.

Constraints: for character data *only*, the length of each element of `a` must not exceed 255.

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.

ascend — logical, intent(in), optional

Input: specifies the sorting order:

if `ascend` = `.true.`, then data will be sorted in ascending order;

if `ascend` = `.false.`, then data will be sorted in descending order.

Default: `ascend` = `.true.`.

pos1 — integer, intent(in), optional

pos2 — integer, intent(in), optional

Input: specify the substring to be used in sorting character data; only the substring (`pos1`:`pos2`) of each element of `a` is to be used in determining the sorted order.

Note: `pos1` and `pos2` should *only* be specified when sorting character data.

Default: `pos1` = 1, `pos2` = LEN(`a`).

Constraints: $0 <$ `pos1` $\leq$ `pos2` $\leq$ LEN(a).

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`. 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. *Note:* This argument should *only* be used when sorting character data.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
306	A character argument has an invalid length.
	The length of `a` must not exceed 255.

5 Examples of Usage

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

6 Further Comments

6.1 Algorithmic Detail

The procedure is based on Singleton's implementation of the 'median-of-three' Quicksort algorithm (see Singleton [3]), but with two additional modifications. First, small subfiles are sorted by an insertion sort on a separate final pass (see Sedgewick [2]). Second, if a subfile is partitioned into two very unbalanced subfiles, the larger of them is flagged for special treatment: before it is partitioned, its end-points are swapped with two random points within it; this makes the worst case behaviour extremely unlikely.

6.2 Timing

The average time taken by the procedure is approximately proportional to $n \log\ n$. The worst case time is proportional to n^2, but this is extremely unlikely to occur.

Procedure: nag_rank_vec

1 Description

`nag_rank_vec` is a procedure which ranks a vector of integer or real numbers into ascending or descending order, or ranks a vector of character data in ASCII or reverse ASCII order.

2 Usage

```
USE nag_sort
CALL nag_rank_vec(a, rank  [, 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.

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

n — the number of data elements

3.1 Mandatory Arguments

a(n) — integer / real(kind=*wp*) / character(len=*), intent(in)

Input: the data to be ranked.

Constraints: for character data *only*, the length of each element of `a` must not exceed 255.

rank(n) — integer, intent(out)

Output: the ranks of the corresponding elements of `a`. For example, if the ith element of `a` is the first in the rank order, then `rank(`i`)` is set to 1. `rank` will be a valid permutation of the integers 1 to 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.

ascend — logical, intent(in), optional

Input: specifies the ranking order:

if `ascend` = `.true.`, then data will be ranked in ascending order;

if `ascend` = `.false.`, then data will be ranked in descending order.

Default: `ascend` = `.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
303	Array arguments have inconsistent shapes.
306	A character argument has an invalid length.
	The length of `a` must not exceed 255.

5 Examples of Usage

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

6 Further Comments

6.1 Algorithmic Detail

This procedure uses a variant of list-merging, as described by Knuth [1]. The procedure takes advantage of natural ordering in the data, and uses a simple list insertion in a preparatory pass to generate ordered lists of length at least 10. The ranking is stable; equal elements preserve their ordering in the input data.

6.2 Timing

The average time taken by the procedure is approximately proportional to $n \log\, n$.

Procedure: nag_reorder_vec

1 Description

nag_reorder_vec is a procedure which is designed to be used typically in conjunction with nag_rank_vec and nag_rank_mat. After one of the ranking procedures has been called to determine a vector of ranks, this procedure can be called to rearrange a vector of numeric or character data into the rank order.

If the vector of ranks has been generated in some other way, then nag_check_perm should be called to check the validity before this procedure is called.

2 Usage

```
USE nag_sort
CALL nag_reorder_vec(a, rank  [, 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.

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

n — the number of data elements

3.1 Mandatory Arguments

a(n) — integer / real(kind=*wp*) / complex(kind=*wp*) / character(len=*), intent(inout)

Input: the data to be rearranged.

Output: the rearranged data according to the rank order.

Constraints: for character data *only*, the length of each element of a must not exceed 255.

rank(n) — integer, intent(in)

Input: the ranks of the corresponding elements of a.

Constraints: rank must be a valid permutation of the integers 1 to n, i.e.,

$1 \leq \texttt{rank}(i) \leq n$, for $i = 1, 2, \ldots, n$ and $\texttt{rank}(i) \neq \texttt{rank}(j)$ when $i \neq j$.

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.
303	Array arguments have inconsistent shapes.
306	A character argument has an invalid length.
	The length of `a` must not exceed 255.

5 Examples of Usage

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

6 Further Comments

6.1 Timing

The average time taken by the procedure is approximately proportional to $n \log\ n$.

Procedure: nag_rank_mat

1 Description

`nag_rank_mat` is a procedure which ranks the rows or columns of a matrix of integer or real data into ascending or descending order. The ordering is determined by first ranking the data in column 1 (or row 1), then ranking any tied rows according to the data in column 2 (or row 2), and so on up to the last column (or row).

2 Usage

```
USE nag_sort
CALL nag_rank_mat(a, rank  [, 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.

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

m	— the number of rows of the data matrix
n	— the number of columns of the data matrix

3.1 Mandatory Arguments

a(m, n) — integer / real(kind=*wp*), intent(in)

Input: the data to be ranked.

rank(m) / **rank**(n) — integer, intent(out)

Output: the ranks of the rows (or columns) of `a`.

If the optional argument `row` = `.true.` (the default), `rank` must be of shape (m) and will contain the ranks of the rows of `a`. For example, if the ith row of `a` is the first in the rank order, then `rank`(i) is set to 1. `rank` will be a valid permutation of the integers 1 to m.

If the optional argument `row` = `.false.`, `rank` must be of shape (n) and will contain the ranks of the columns of `a`. For example, if the ith column of `a` is the first in the rank order, then `rank`(i) is set to 1. `rank` will be a valid permutation of the integers 1 to 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.

ascend — logical, intent(in), optional

Input: specifies the sorting order:

if `ascend` = `.true.`, then data will be sorted in ascending order;

if `ascend` = `.false.`, then data will be sorted in descending order.

Default: `ascend` = `.true.`.

row — logical, intent(in), optional

Input: specifies whether ranking should be performed by row or by column:

if `row` = `.true.`, then the rows of the matrix `a` will be ranked;

if `row` = `.false.`, then the columns of `a` will be ranked.

Default: `row` = `.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
303	Array arguments have inconsistent shapes.

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

This procedure uses a variant of list-merging, as described by Knuth [1]. The procedure takes advantage of natural ordering in the data, and uses a simple list insertion in a preparatory pass to generate ordered lists of length at least 10. The ranking is stable; equal rows (or columns) preserve their ordering in the input data.

6.2 Timing

The average time taken by the procedure is approximately proportional to $n \log n$.

Procedure: nag_rank_arb_data

1 Description

`nag_rank_arb_data` is a general purpose procedure for ranking arbitrary data. The procedure does not access the data directly; instead it calls a user-supplied function `compare` to determine the relative ordering of any two data items.

2 Usage

```
USE nag_sort
CALL nag_rank_arb_data(compare, rank)
```

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.

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

n — the number of data elements

3.1 Mandatory Arguments

compare — function

`compare` is a user-supplied function that must specify the relative ordering of any two data items.

```
function compare(i,j)
```

```
integer, intent(in) ::  i, j
```

Input: `i` and `j` identify the data items to be compared.

```
logical ::  compare
```

Result: if item `i` must come strictly *after* item `j` in the rank ordering, then `compare` must return `.true.`, otherwise `compare` must return `.false.`.

rank(n) — integer, intent(out)

Output: the ranks of the data items. For example, if item i is the first in the rank order, then `rank`(i) is set to 1. `rank` will be a valid permutation of the integers 1 to n.

4 Error Codes

None.

5 Examples of Usage

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

6 Further Comments

6.1 Algorithmic Detail

This procedure uses a variant of list-merging, as described by Knuth [1]. The procedure takes advantage of natural ordering in the data, and uses a simple list insertion in a preparatory pass to generate ordered lists of length at least 10.

6.2 Timing

The average time taken by the procedure is approximately proportional to $n \log n$; this will usually be dominated by the time taken in the user-supplied function `compare`.

Procedure: nag_invert_perm

1 Description

There are two common ways of describing a permutation using an integer vector p.

- The first uses ranks: $p(i)$ holds the position to which the ith data element should be moved in order to sort the data, in other words its rank in the sorted order.
- The second uses indices: $p(i)$ holds the current position of the data element which would occur in ith position in sorted order.

For example, given the values

$$3.5 \quad 5.9 \quad 2.9 \quad 0.5$$

to be sorted in ascending order, the ranks would be

$$3 \quad 4 \quad 2 \quad 1$$

and the indices would be

$$4 \quad 3 \quad 1 \quad 2$$

The procedures `nag_rank_vec` and `nag_rank_mat` generate ranks, and `nag_reorder_vec` requires ranks to be supplied to specify the reordering. However, if it is desired simply to refer to the data in sorted order without actually reordering them, indices are more convenient than ranks. This procedure can be used to convert ranks to indices, or indices to ranks, as the two permutations are inverses of one another.

2 Usage

```
USE nag_sort
CALL nag_invert_perm(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 '$\mathtt{x}(n)$' 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 — the number of data elements

3.1 Mandatory Argument

p(n) — integer, intent(inout)

Input: a permutation of the integers 1 to n.

Output: the inverse permutation of the input array.

Constraints: `p` must be a valid permutation of the integers 1 to n, i.e.,

$$1 \leq \mathtt{p}(i) \leq n, \text{ for } i = 1, 2, \ldots, n \text{ and } \mathtt{p}(i) \neq \mathtt{p}(j) \text{ when } i \neq j.$$

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.

5 Examples of Usage

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

Procedure: nag_check_perm

1 Description

`nag_check_perm` can be used to check the validity of user-supplied ranks or indices, without the ranks or indices being corrupted.

2 Usage

```
USE nag_sort
[value =] nag_check_perm(p  [, optional arguments])
```

This function returns a logical value.

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 '$\mathtt{x}(n)$' 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	— the number of data elements

3.1 Mandatory Argument

p(n) — integer, intent(in)

Input: the values which are supposed to be a permutation of the integers 1 to n. If `p` is a valid permutation of the integers 1 to n, i.e.,

$1 \leq \mathtt{p}(i) \leq n$, for $i = 1, 2, \ldots, n$ and $\mathtt{p}(i) \neq \mathtt{p}(j)$ when $i \neq j$,

then the function will return the value `.true.` on exit, otherwise it will return `.false.`.

Constraints: $1 \leq \mathtt{p}(i) \leq n$, for $i = 1, 2, \ldots, n$.

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.

5 Examples of Usage

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

Procedure: nag_decomp_perm

1 Description

`nag_decomp_perm` is provided as an aid to reordering data structures without using additional storage. However, you should consider carefully whether it is necessary to rearrange the data, or whether it would be simpler and more efficient to refer to it in sorted order using an index vector, or to create a copy of the data in sorted order.

Given a vector `rank` which specifies the ranks of the data (as generated by the procedures `nag_rank_vec` or `nag_rank_mat`, for example), this procedure generates a new vector `cycles`, in which the permutation is represented in its component cycles, with the first element of each cycle negated.

This output argument `cycles` can then be used to perform a cyclic permutation of the data without using additional storage, as illustrated in Section 5. This is the method used by `nag_reorder_vec`.

2 Usage

```
USE nag_sort
CALL nag_decomp_perm(p, cycles  [, 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.

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

n — the number of data elements

3.1 Mandatory Arguments

p(n) — integer, intent(in)

Input: a permutation of the integers 1 to n.

Constraints: `p` must be a valid permutation of the integers 1 to n, i.e.,

$1 \leq \texttt{p}(i) \leq \text{n}$, for $i = 1, 2, \ldots, n$ and $\texttt{p}(i) \neq \texttt{p}(j)$ when $i \neq j$.

cycles(n) — integer, intent(out)

Output: a representation of the permutation as a list of cycles, with the first integer in each cycle negated.

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.
303	Array arguments have inconsistent shapes.

5 Examples of Usage

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

To rearrange the data into a different order without using additional storage, the simplest method is to decompose the permutation which specifies the new order into cycles, and then to do a cyclic permutation of the data items in each cycle.

Given a vector `rank` which specifies the ranks of the data, this procedure generates a new vector `cycles`, in which the permutation is represented in its component cycles, with the first element of each cycle negated. For example, the permutation

$$5 \quad 7 \quad 4 \quad 2 \quad 1 \quad 6 \quad 3$$

is composed of the cycles

$$(1 \quad 5) \quad (2 \quad 7 \quad 3 \quad 4) \quad (6)$$

and the vector `cycles` generated by this procedure contains

$$-1 \quad 5 \quad -2 \quad 7 \quad 3 \quad 4 \quad -6.$$

In order to rearrange the data according to the specified ranks:

items 1 and 5 must be interchanged;

items 4, 2, 7 and 3 must be moved one place round the cycle;

item 6 must be left in place.

The complete rearrangement can be achieved by the following code:

```
DO k=1, n
   i = cycles(k)
   IF ( i < 0 ) THEN
        j = -i
   ELSE
       [ Swap items i and j ]
   END IF
END DO
```

Example 1: Sorting a Vector of Character Data

This example program reads a list of 12-character records, and sorts them into reverse ASCII order on characters 7 to 11. The sorting is performed in two ways, the first using the procedure `nag_sort_vec` directly, and the second using `nag_rank_vec` and `nag_reorder_vec`.

1 Program Text

```
PROGRAM nag_sort_ex01

  ! Example Program Text for nag_sort
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sort, ONLY : nag_sort_vec, nag_rank_vec, nag_reorder_vec
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Local Scalars ..
  INTEGER :: n
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: rank(:)
  CHARACTER (12), ALLOCATABLE :: a(:), a1(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sort_ex01'
  WRITE (nag_std_out,*)

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

  ALLOCATE (a(n),a1(n),rank(n)) ! Allocate storage

  READ (nag_std_in,'(A)') a
  a1 = a

  ! Sort data directly

  CALL nag_sort_vec(a,pos1=7,pos2=11,ascend=.FALSE.)

  WRITE (nag_std_out,*) 'Data sorted (directly) on columns 7 to 11'
  WRITE (nag_std_out,'(2X,A)') a
  WRITE (nag_std_out,*)

  ! Sort data indirectly (by ranking and reordering)

  CALL nag_rank_vec(a1(:)(7:11),rank,ascend=.FALSE.)

  CALL nag_reorder_vec(a1,rank)

  WRITE (nag_std_out,*) 'Vector of ranks'
  WRITE (nag_std_out,'(2X,15I4)') rank
  WRITE (nag_std_out,*)

  WRITE (nag_std_out,*) 'Data sorted (indirectly) on columns 7 to 11'
  WRITE (nag_std_out,'(2X,A)') a1

  DEALLOCATE (a,a1,rank)       ! Deallocate storage

END PROGRAM nag_sort_ex01
```

2 Program Data

```
Example Program Data for nag_sort_ex01
7              : n (number of data elements)
VIOLET 123CS
INDIGO 563AV
BLUE   151KJ
GREEN  205AX
YELLOW  99UI
ORANGE 132PO
RED    225LE  : End of a (data vector)
```

3 Program Results

```
 Example Program Results for nag_sort_ex01

 Data sorted (directly) on columns 7 to 11
  INDIGO 563AV
  RED    225LE
  GREEN  205AX
  BLUE   151KJ
  ORANGE 132PO
  VIOLET 123CS
  YELLOW  99UI

 Vector of ranks
     6   1   4   3   7   5   2

 Data sorted (indirectly) on columns 7 to 11
  INDIGO 563AV
  RED    225LE
  GREEN  205AX
  BLUE   151KJ
  ORANGE 132PO
  VIOLET 123CS
  YELLOW  99UI
```

Example 2: Rearranging a Matrix of Real Data

This example program reads a matrix of real data and rearranges its rows so that the elements of the kth column are in ascending order. To do this, the program first calls `nag_rank_vec` to rank the elements of the kth column, and then calls `nag_reorder_vec` to rearrange each column into the order specified by the ranks.

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_sort_ex02

  ! Example Program Text for nag_sort
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_write_mat, ONLY : nag_write_gen_mat
  USE nag_sort, ONLY : nag_rank_vec, nag_reorder_vec
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, j, k, m, n
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: rank(:)
  REAL (wp), ALLOCATABLE :: a(:,:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sort_ex02'
  WRITE (nag_std_out,*)

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

  ALLOCATE (rank(m),a(m,n))    ! Allocate storage

  READ (nag_std_in,*) (a(i,:),i=1,m)

  ! Rank the kth column of matrix in ascending order

  CALL nag_rank_vec(a(:,k),rank)

  ! Rearrange the columns of the matrix
  DO j = 1, n

    CALL nag_reorder_vec(a(:,j),rank)

  END DO

  CALL nag_write_gen_mat(a,format='F6.1',title='Data sorted on column 1')

  DEALLOCATE (a,rank)          ! Deallocate storage

END PROGRAM nag_sort_ex02
```

2 Program Data

```
Example Program Data for nag_sort_ex02
12  3  1     : m, n (size of data matrix) , k (column index)
6.0 5.0 4.0
5.0 2.0 1.0
2.0 4.0 9.0
4.0 9.0 6.0
4.0 9.0 5.0
4.0 1.0 2.0
3.0 4.0 1.0
2.0 4.0 6.0
1.0 6.0 4.0
9.0 3.0 2.0
6.0 2.0 5.0
4.0 9.0 6.0  : End of a (data matrix)
```

3 Program Results

```
 Example Program Results for nag_sort_ex02

 Data sorted on column 1
     1.0   6.0   4.0
     2.0   4.0   9.0
     2.0   4.0   6.0
     3.0   4.0   1.0
     4.0   9.0   6.0
     4.0   9.0   5.0
     4.0   1.0   2.0
     4.0   9.0   6.0
     5.0   2.0   1.0
     6.0   5.0   4.0
     6.0   2.0   5.0
     9.0   3.0   2.0
```

Example 3: Ranking a Matrix, Checking and Decomposing Ranks

This example program reads a matrix of numeric data and rearranges its columns so that the elements of the lth row are in ascending order. To do this, the program first calls nag_rank_mat to rank the elements of the lth row, and then checks the ranks by calling nag_check_perm and finally calls nag_decomp_perm to decompose the rank vector into cycles. It then rearranges the columns using the framework of code suggested in Section 5 of the procedure document for nag_decomp_perm.

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_sort_ex03

  ! Example Program Text for nag_sort
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_write_mat, ONLY : nag_write_gen_mat
  USE nag_sort, ONLY : nag_check_perm, nag_rank_mat, nag_decomp_perm
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, j, k, l, m, n
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: cycles(:), rank(:), tmp(:)
  REAL (wp), ALLOCATABLE :: a(:,:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_sort_ex03'
  WRITE (nag_std_out,*)

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

  ALLOCATE (a(m,n),rank(n),cycles(n),tmp(m)) ! Allocate storage

  READ (nag_std_in,*) (a(i,:),i=1,m)

  ! Rank the matrix by row, in ascending order

  CALL nag_rank_mat(a(l:l,:),rank,row=.FALSE.)

  ! Check the validity of the permutation
  IF (nag_check_perm(rank)) THEN

    ! Decompose rank into cycles

    CALL nag_decomp_perm(rank,cycles)

    ! Rearrange matrix
    DO k = 1, n
      i = cycles(k)
      IF (i<0) THEN
        j = -i
      ELSE
        ! Swap columns i and j
        tmp = a(:,j)
        a(:,j) = a(:,i)
        a(:,i) = tmp
      END IF
    END DO

    CALL nag_write_gen_mat(a,format='F6.1',title='Matrix sorted on row 3')

  END IF

  DEALLOCATE (a,rank,cycles,tmp) ! Deallocate storage

END PROGRAM nag_sort_ex03
```

2 Program Data

```
Example Program Data for nag_sort_ex03
3 12 3 : m,n (size of data matrix) , l (row index)
5.0 4.0 3.0 2.0 2.0 1.0 9.0 4.0 4.0 2.0 2.0 1.0
3.0 8.0 2.0 5.0 5.0 6.0 9.0 8.0 9.0 5.0 4.0 1.0
9.0 1.0 6.0 1.0 2.0 4.0 8.0 1.0 2.0 2.0 6.0 2.0 : End of a (data matrix)
```

3 Program Results

```
 Example Program Results for nag_sort_ex03

 Matrix sorted on row 3
     4.0   2.0   4.0   2.0   4.0   2.0   1.0   1.0   3.0   2.0   9.0   5.0
     8.0   5.0   8.0   5.0   9.0   5.0   1.0   6.0   2.0   4.0   9.0   3.0
     1.0   1.0   1.0   2.0   2.0   2.0   2.0   4.0   6.0   6.0   8.0   9.0
```

Example 4: Ranking Arbitrary Data, Inverting Ranks

This example program reads records, each of which contains an integer key, `a`, and a real number, `b`. The program ranks the records first of all in ascending order of the integer key; records with equal keys are ranked in descending order of the real number if the key is negative, in ascending order of the real number if the key is positive, and in their original order if the key is zero. After calling `nag_rank_arb_data`, the program calls `nag_invert_perm` to convert the ranks to indices, and prints the records in rank order.

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.

```
MODULE sort_ex04_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: a(:)
  REAL (wp), ALLOCATABLE :: b(:)

CONTAINS

  FUNCTION compare(i,j)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Scalar Arguments ..
    INTEGER, INTENT (IN) :: i, j
    ! .. Function Return Value ..
    LOGICAL :: compare
    ! .. Executable Statements ..
    IF (a(i)/=a(j)) THEN
      compare = a(i) > a(j)
    ELSE
      IF (a(i)<0) THEN
        compare = b(i) < b(j)
      ELSE IF (a(i)>0) THEN
        compare = b(i) > b(j)
      ELSE
        compare = i < j
      END IF
    END IF

  END FUNCTION compare

END MODULE sort_ex04_mod

PROGRAM nag_sort_ex04

  ! Example Program Text for nag_sort
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sort, ONLY : nag_invert_perm, nag_rank_arb_data
```

```
      USE sort_ex04_mod, ONLY : compare, a, b
      ! .. Implicit None Statement ..
      IMPLICIT NONE
      ! .. Intrinsic Functions ..
      INTRINSIC KIND
      ! .. Local Scalars ..
      INTEGER :: i, n
      ! .. Local Arrays ..
      INTEGER, ALLOCATABLE :: rank(:)
      ! .. Executable Statements ..
      WRITE (nag_std_out,*) 'Example Program Results for nag_sort_ex04'
      WRITE (nag_std_out,*)

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

      ALLOCATE (a(n),b(n),rank(n)) ! Allocate storage

      READ (nag_std_in,*) (a(i),b(i),i=1,n)

      ! Rank data

      CALL nag_rank_arb_data(compare,rank)

      ! Convert the ranks to indices

      CALL nag_invert_perm(rank)

      WRITE (nag_std_out,*) 'Data in sorted order'
      WRITE (nag_std_out,'(1X,I4,F6.1)') (a(rank(i)),b(rank(i)),i=1,n)

      DEALLOCATE (a,b,rank)        ! Deallocate storage

    END PROGRAM nag_sort_ex04
```

2 Program Data

```
Example Program Data for nag_sort_ex04
 12      : n (number of data elements)
  2  3.0
  1  4.0
 -1  6.0
  0  5.0
  2  2.0
 -2  7.0
  0  4.0
  1  3.0
  1  5.0
 -1  2.0
  1  0.0
  2  1.0 : End of a, b (integer and real data vectors)
```

3 Program Results

```
 Example Program Results for nag_sort_ex04

 Data in sorted order
   -2   7.0
   -1   6.0
   -1   2.0
    0   4.0
    0   5.0
    1   0.0
    1   3.0
    1   4.0
    1   5.0
    2   1.0
    2   2.0
    2   3.0
```

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_sort_ex05`

Ranking a matrix, checking and inverting the permutation.

`nag_sort_ex06`

Sorting a vector of real data.

`nag_sort_ex07`

Ranking a matrix of real data.

`nag_sort_ex08`

Ranking a matrix of complex data.

References

[1] Knuth D E (1981) *The Art of Computer Programming (Volume 2)* Addison-Wesley (2nd Edition)

[2] Sedgewick R (1978) Implementing quicksort programs *Comm. ACM* **21** 847–857

[3] Singleton R C (1969) An efficient algorithm for sorting with minimal storage: Algorithm 347 *Comm. ACM* **12** 185–187