NAG Library Function Document
nag_stable_sort (m01ctc)

1 Purpose
nag_stable_sort (m01ctc) rearranges a vector of arbitrary type objects into ascending or descending order.

2 Specification
#include <nag.h>
#include <nagm01.h>
void nag_stable_sort (Pointer vec, size_t n, size_t size, ptrdiff_t stride,
                    Integer (*compare)(const Nag_Pointer a, const Nag_Pointer b),
                    Nag_SortOrder order, NagError *fail)

3 Description
nag_stable_sort (m01ctc) sorts a set of \( n \) data objects of arbitrary type, which are stored in the elements of an array at intervals of length \( \text{stride} \). The function may be used to sort a column of a two-dimensional array. Either ascending or descending sort order may be specified.

A stable sort is one which preserves the order of distinct data items that compare equal. This function uses nag_rank_sort (m01dsc), nag_make_indices (m01zac) and nag_reorder_vector (m01esc) in order to carry out a stable sort with the same specification as nag_quicksort (m01csc). nag_stable_sort (m01ctc) will be faster than nag_quicksort (m01csc) if the comparison function \( \text{compare} \) is slow or the data items are large. Internally a large amount of workspace may be required compared with nag_quicksort (m01csc).

4 References

5 Arguments
1: \( \text{vec} \rightarrow \text{Pointer} \) \( \quad \text{Input/Output} \)
   \( \text{On entry:} \) the array of objects to be sorted.
   \( \text{On exit:} \) the objects rearranged into sorted order.

2: \( \text{n} \rightarrow \text{size_t} \) \( \quad \text{Input} \)
   \( \text{On entry:} \) the number \( n \) of objects to be sorted.
   \( \text{Constraint:} \ n \geq 0. \)

3: \( \text{size} \rightarrow \text{size_t} \) \( \quad \text{Input} \)
   \( \text{On entry:} \) the size of each object to be sorted.
   \( \text{Constraint:} \ \text{size} \geq 1. \)

4: \( \text{stride} \rightarrow \text{ptrdiff_t} \) \( \quad \text{Input} \)
   \( \text{On entry:} \) the increment between data items in \( \text{vec} \) to be sorted.
Note: if \texttt{stride} is positive, \texttt{vec} should point at the first data object; otherwise \texttt{vec} should point at the last data object.

\textit{Constraint:} $|\texttt{stride}| \geq \texttt{size}$.

5: \texttt{compare} – function, supplied by the user

External Function \texttt{nag\_stable\_sort} (m01ctc) compares two data objects. If its arguments are pointers to a structure, this function must allow for the offset of the data field in the structure (if it is not the first).

The function must return:

-1 if the first data field is less than the second,

0 if the first data field is equal to the second,

1 if the first data field is greater than the second.

The specification of \texttt{compare} is:

\begin{verbatim}
Integer compare (const Nag_Pointer a, const Nag_Pointer b)
1:    a – const Nag_Pointer [Input]
    \textit{On entry:} the first data field.
2:    b – const Nag_Pointer [Input]
    \textit{On entry:} the second data field.
\end{verbatim}

6: \texttt{order} – Nag_SortOrder

\textit{Input}

\textit{On entry:} specifies whether the array is to be sorted into ascending or descending order.

\textit{Constraint:} \texttt{order} = Nag_Ascending or Nag_Descending.

7: \texttt{fail} – NagError *

\textit{Input/Output}

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

\textbf{NE\_2\_INT\_ARG\_LT}

On entry, $|\texttt{stride}| = \langle\text{value}\rangle$ while \texttt{size} = $\langle\text{value}\rangle$. These arguments must satisfy $|\texttt{stride}| \geq \texttt{size}$.

\textbf{NE\_ALLOC\_FAIL}

Dynamic memory allocation failed.

\textbf{NE\_BAD\_PARAM}

On entry, argument \texttt{order} had an illegal value.

\textbf{NE\_INT\_ARG\_GT}

On entry, $\texttt{n} = \langle\text{value}\rangle$.

Constraint: $\texttt{n} \leq \langle\text{value}\rangle$.

On entry, $\texttt{size} = \langle\text{value}\rangle$.

Constraint: $\texttt{size} \leq \langle\text{value}\rangle$.

On entry, $\texttt{stride} = \langle\text{value}\rangle$.

Constraint: $|\texttt{stride}| \leq \langle\text{value}\rangle$.

These arguments are limited to an implementation-dependent size which is printed in the error message.
On entry, \( n = \langle \text{value} \rangle \).
Constraint: \( n \geq 0 \).

On entry, \( \text{size} = \langle \text{value} \rangle \).
Constraint: \( \text{size} \geq 1 \).
The absolute value of \( \text{stride} \) must not be less than \( \text{size} \).

7 Accuracy
Not applicable.

8 Parallelism and Performance
Not applicable.

9 Further Comments
The time taken by nag_stable_sort (m01ctc) is approximately proportional to \( n \log(n) \).

10 Example
The example program reads a three column matrix of real numbers and sorts the first column into ascending order.

10.1 Program Text
#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nag_stddef.h>
#include <nagm01.h>

#ifdef __cplusplus
extern "C" {
#endif
static Integer NAG_CALL compare(const Nag_Pointer a, const Nag_Pointer b);
#ifdef __cplusplus
}
#endif
#define VEC(I, J) vec[(I) *tdvec + J]

int main(void)
{
    Integer exit_status = 0, i, j, k, m, n, tdvec;
    NagError fail;
    double *vec = 0;

    INIT_FAIL(fail);

    /* Skip heading in data file */
    #ifdef _WIN32
    scanf_s("%*[\n]");
    
    printf("\n\n%8s%8s%8s\n", "i", "j", "k");
    
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf_s("%f", &VEC(i, j));
        }
    }

    nagm01c3_stable_sort(n, m, vec, tdvec, &fail);
    
    if (fail) { nagerror(fail, fail); } else {
        printf("\n\n\n%8s%8s%8s\n", "i", "j", "k");
        
        for (i = 0; i < m; i++)
        {
            for (j = 0; j < n; j++)
            {
                printf("%8.3f", VEC(i, j));
                if (j % n) printf("\t");
            }
            printf("\n");
        }
    }

    if (exit_status) {
        return exit_status;
    } else {
        return 0;
    }

    #endif
}

"}
```c
#ifndef _WIN32
    scanf("%*
#else
    scanf("%*
#endif
#ifndef _WIN32
    scanf_s("%NAG_IFMT"%NAG_IFMT"%NAG_IFMT"", &m, &n, &k);
#else
    scanf("%NAG_IFMT"%NAG_IFMT"%NAG_IFMT"", &m, &n, &k);
#endif
if (m >= 0 && n >= 0 && k >= 0 && k <= n)
{
    if (!(vec = NAG_ALLOC(m*n, double)))
    {
        printf("Allocation failure\n");
        exit_status = -1;
        goto END;
    }
    tdvec = n;
} else
{
    printf("Invalid m or n or k.\n");
    exit_status = 1;
    return exit_status;
}
for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
#ifndef _WIN32
    scanf_s("%lf", &VEC(i, j));
#else
    scanf("%lf", &VEC(i, j));
#endif
/* nag_stable_sort (m01ctc).
* Stable sort of set of values of arbitrary data type
*/
nag_stable_sort((Pointer) &VEC(0, k-1), (size_t) m, sizeof(double),
    (ptrdiff_t)(n*sizeof(double)), compare, Nag_Ascending,
    &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error from nag_stable_sort (m01ctc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
printf("\nMatrix with column %"NAG_IFMT" sorted\n", k);
for (i = 0; i < m; ++i)
{
    for (j = 0; j < n; ++j)
        printf(" %7.1f ", VEC(i, j));
    printf("\n");
} END:
NAG_FREE(vec);
return exit_status;
}
static Integer NAG_CALL compare(const Nag_Pointer a, const Nag_Pointer b)
{
    double x = *((const double *) a);
    double y = *((const double *) b);
    return(x < y?-1:(x == y?0:1));
}
```

10.2 Program Data

nag_stable_sort (m01ctc) Example Program Data
12 3 1
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

10.3 Program Results

nag_stable_sort (m01ctc) Example Program Results

Matrix with column 1 sorted
1.0 5.0 4.0
2.0 2.0 1.0
2.0 4.0 9.0
3.0 9.0 6.0
4.0 9.0 5.0
4.0 1.0 2.0
4.0 4.0 1.0
4.0 4.0 6.0
5.0 6.0 4.0
6.0 3.0 2.0
6.0 2.0 5.0
9.0 9.0 6.0