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 **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 **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

Knuth D E (1973) *The Art of Computer Programming (Volume 3)* (2nd Edition) Addison–Wesley

5 Arguments

1: **vec**[**n**] – Pointer *Input/Output*

On entry: the array of objects to be sorted.

On exit: the objects rearranged into sorted order.

2: **n** – size_t *Input*

On entry: the number n of objects to be sorted.

Constraint: $0 \leq$ **n** $\leq$ MAX_LENGTH, where MAX_LENGTH is an implementation-dependent value for the maximum size of an array.

3: **size** – size_t *Input*

On entry: the size of each object to be sorted.

Constraint: $1 \leq$ **size** $\leq p$, where p is an implementation-dependent value for the maximum size_t size on the system, divided by **n** if **n** is positive.

4: **stride** – ptrdiff_t *Input*

On entry: the increment between data items in **vec** to be sorted.

Note: if **stride** is positive, **vec** should point at the first data object; otherwise **vec** should point at the last data object.

Constraint: **size** $\leq$ |**stride**| $\leq p$, where p is an implementation-dependent value for the maximum size_t size on the system, divided by **n** if **n** is positive.

5: **compare** – function, supplied by the user *External Function*

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:

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

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

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

The specification of **compare** is:

```
Integer compare (const Nag_Pointer a, const Nag_Pointer b)
```

1: **a** – const Nag_Pointer *Input*

On entry: the first data field.

2: **b** – const Nag_Pointer *Input*

On entry: the second data field.

6: **order** – Nag_SortOrder *Input*

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

Constraint: **order** = Nag_Ascending or Nag_Descending.

7: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

On entry, |**stride**| = ⟨*value*⟩ while **size** = ⟨*value*⟩. These arguments must satisfy |**stride**| $\geq$ **size**.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument **order** had an illegal value.

NE_INT_ARG_GT

On entry, **n** = ⟨*value*⟩.
Constraint: **n** $\leq$ ⟨*value*⟩, an implementation-dependent size that is printed in the error message.

On entry, **size** = ⟨*value*⟩.
Constraint: **size** $\leq$ ⟨*value*⟩, an implementation-dependent size that is printed in the error message.

On entry, **stride** = ⟨*value*⟩.
Constraint: |**stride**| ≤ ⟨*value*⟩, an implementation-dependent size that is printed in the error message.

NE_INT_ARG_LT

On entry, **n** = ⟨*value*⟩.
Constraint: **n** ≥ 0.

On entry, **size** = ⟨*value*⟩.
Constraint: **size** ≥ 1.

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_stable_sort (m01ctc) is not threaded in any implementation.

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

```
/* nag_stable_sort (m01ctc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#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]");
#else
  scanf("%*[^\n]");
#endif
  printf("nag_stable_sort (m01ctc) Example Program Results\n");
#ifdef _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)
#ifdef _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
```