NAG Library Function Document

nag_reorder_vector (m01esc)

1 Purpose

nag_reorder_vector (m01esc) rearranges a vector of arbitrary type data objects into the order specified by a vector of indices.

2 Specification

```
#include <nag.h>
#include <nagm01.h>
void nag_reorder_vector (Pointer vec, size_t n, size_t size,
     ptrdiff_t stride, size_t indices[], NagError *fail)
```

3 Description

nag_reorder_vector (m01esc) uses a variant of list merging as described by Knuth (1973). The function rearranges a set of n data objects of arbitrary type, which are stored in an array at intervals of length **stride**, into the order specified by an array of indices.

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 rearranged.

On exit: the objects rearranged according to array **indices**.

2: **n** – size_t *Input*

On entry: the number, n, of objects to be rearranged.

Constraint: **n** ≥ 0.

3: **size** – size_t *Input*

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

Constraint: **size** ≥ 1.

4: **stride** – ptrdiff_t *Input*

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

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

Constraint: $|\mathbf{stride}| \geq \mathbf{size}$.

5: **indices**[**n**] – size_t *Input*

On entry: the indices specifying the order in which the elements of vector are to be rearranged.

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

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

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

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

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_RANK

Invalid **indices** vector.

NE_INT_ARG_GT

On entry, |**stride**| must not be greater than ⟨*value*⟩: |**stride**| = ⟨*value*⟩.

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

On entry, **size** = ⟨*value*⟩.
Constraint: **size** ≤ ⟨*value*⟩.
These arguments are limited to an implementation-dependent size which 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

Not applicable.

9 Further Comments

The average time taken by the function is approximately proportional to **n**.

10 Example

The example program.

10.1 Program Text

```
/* nag_reorder_vector (m01esc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 2 revised, 1992.
 * Mark 5 revised, 1998.
 * Mark 7 revised, 2001.
```

```
 * Mark 8 revised, 2004.
 *
 */

#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 A(I, J) a[(I) *tda + J]
int main(void)
{
  Integer  exit_status = 0, tda;
  NagError fail;
  double   *a = 0;
  size_t   i, *indices = 0, j, k, m, n;

  INIT_FAIL(fail);

  /* Skip heading in data file */
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif
  printf("nag_reorder_vector (m01esc) Example Program Results\n");
#ifdef _WIN32
  scanf_s("%"NAG_UFMT"%"NAG_UFMT"%"NAG_UFMT, &m, &n, &k);
#else
  scanf("%"NAG_UFMT"%"NAG_UFMT"%"NAG_UFMT, &m, &n, &k);
#endif
  if (m >= 1 && n >= 1 && k >= 1 && k <= n)
    {
      if (!(a = NAG_ALLOC(m*n, double)) ||
          !(indices = NAG_ALLOC(m, size_t)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
      tda = 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", &A(i, j));
#else
      scanf("%lf", &A(i, j));
#endif
  /* nag_rank_sort (m01dsc).
   * Rank sort of set of values of arbitrary data type
   */
  nag_rank_sort((Pointer) &A(0, k-1), m, (ptrdiff_t)(n*sizeof(double)),
                compare, Nag_Ascending, indices, &fail);
  if (fail.code != NE_NOERROR)
```

```
    {
      printf("Error from nag_rank_sort (m01dsc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  /* nag_make_indices (m01zac).
   * Inverts a permutation converting a rank vector to an
   * index vector or vice versa
   */
  nag_make_indices(indices, m, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_make_indices (m01zac).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }
  for (j = 0; j < n; ++j)
    {
      /* nag_reorder_vector (m01esc).
       * Reorders set of values of arbitrary data type into the
       * order specified by a set of indices
       */
      nag_reorder_vector((Pointer) &A(0, j), m, sizeof(double),
                         (ptrdiff_t)(n*sizeof(double)), indices, &fail);
      if (fail.code != NE_NOERROR)
        {
          printf("Error from nag_reorder_vector (m01esc).\n%s\n",
                  fail.message);
          exit_status = 1;
          goto END;
        }
    }
  printf("\nMatrix with column %"NAG_UFMT" sorted\n", k);
  for (i = 0; i < m; ++i)
    {
      for (j = 0; j < n; ++j)
        printf("  %7.1f  ", A(i, j));
      printf("\n");
    }
 END:
  NAG_FREE(a);
  NAG_FREE(indices);
  return exit_status;
}

static Integer NAG_CALL compare(const Nag_Pointer a, const Nag_Pointer b)
{
  double x = *((const double *) a) - *((const double *) b);
  return(x < 0.0?-1:(x == 0.0?0:1));
}
```

10.2 Program Data

```
nag_reorder_vector (m01esc) 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_reorder_vector (m01esc) Example Program Results

Matrix with column 1 sorted
     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
```