NAG Library Function Document

nag_rank_sort (m01dsc)

1 Purpose

nag_rank_sort (m01dsc) ranks a vector of arbitrary data type objects in ascending or descending order.

2 Specification

```
#include <nag.h>
#include <nagm01.h>
void nag_rank_sort (const Pointer vec, size_t n, ptrdiff_t stride,
     Integer (*compare)(const Nag_Pointer a, const Nag_Pointer b),
     Nag_SortOrder order, size_t ranks[], NagError *fail)
```

3 Description

nag_rank_sort (m01dsc) ranks a set of n data objects of arbitrary type, which are stored in the elements of an array at intervals of length **stride**. The ranks are in the range 0 to $n-1$.

Either ascending or descending ranking order may be specified.

nag_rank_sort (m01dsc) uses a variant of list merging as described by Knuth (1973).

4 References

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

5 Arguments

1: **vec**[**n**] – const Pointer *Input*

On entry: the array of objects to be ranked.

2: **n** – size_t *Input*

On entry: the number n of objects.

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

3: **stride** – ptrdiff_t *Input*

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

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

Constraint: $0 < |\mathbf{stride}| \leq p$, where p is an implementation-dependent value for the maximum size_t size on the system, divided by **n** if **n** is positive.

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

nag_rank_sort (m01dsc) 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 **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.

5: **order** – Nag_SortOrder *Input*

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

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

6: **ranks**[**n**] – size_t *Output*

On exit: the ranks of the corresponding data elements in **vec**.

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_BAD_PARAM

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

NE_INT_ARG_EQ

On entry, **stride** = ⟨*value*⟩.
Constraint: **stride** $\neq 0$.

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

NE_INT_ARG_LT

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

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_rank_sort (m01dsc) is not threaded in any implementation.

9 Further Comments

The time taken by nag_rank_sort (m01dsc) is approximately proportional to $n\log(n)$.

10 Example

The example program reads a list of real numbers and ranks them into ascending order.

10.1 Program Text

```
/* nag_rank_sort (m01dsc) 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

int main(void)
{
  Integer exit_status = 0;
  NagError fail;
  double *vec = 0;
  ptrdiff_t step;
  size_t i, n, *rank = 0, step_u;

  INIT_FAIL(fail);

  /* Skip heading in data file */
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif
  printf("nag_rank_sort (m01dsc) Example Program Results\n\n");
#ifdef _WIN32
  scanf_s("%" NAG_UFMT "%" NAG_UFMT "", &n, &step_u);
#else
  scanf("%" NAG_UFMT "%" NAG_UFMT "", &n, &step_u);
#endif
  step = (ptrdiff_t) step_u;
  if (n >= 1) {
    if (!(vec = NAG_ALLOC(n, double)) || !(rank = NAG_ALLOC(n, size_t))) {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
```

```
    }
  }
  else {
    printf("Invalid n or step.\n");
    exit_status = 1;
    return exit_status;
  }
  for (i = 0; i < n; ++i)
#ifdef _WIN32
    scanf_s("%lf", &vec[i]);
#else
    scanf("%lf", &vec[i]);
#endif
  /* nag_rank_sort (m01dsc).
   * Rank sort of set of values of arbitrary data type
   */
  nag_rank_sort((Pointer) vec, n, step * (ptrdiff_t) (sizeof(double)),
                compare, Nag_Ascending, rank, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_rank_sort (m01dsc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  printf("    Data      Rank\n");
  for (i = 0; i < n; ++i)
    printf("   %7.4f   %4" NAG_UFMT "\n", vec[i], rank[i]);
END:
  NAG_FREE(vec);
  NAG_FREE(rank);
  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_rank_sort (m01dsc) Example Program Data
12
1
5.3 4.6 7.8 1.7 5.3 9.9 3.2 4.3 7.8 4.5 1.2 7.6
```

10.3 Program Results

```
nag_rank_sort (m01dsc) Example Program Results

    Data      Rank
    5.3000      6
    4.6000      5
    7.8000      9
    1.7000      1
    5.3000      7
    9.9000     11
    3.2000      2
    4.3000      3
    7.8000     10
    4.5000      4
    1.2000      0
    7.6000      8
```