NAG Library Function Document

nag_quicksort (m01csc)

1 Purpose

nag_quicksort (m01csc) rearranges a vector of arbitrary type objects into ascending or descending order.

2 Specification

```
#include <nag.h>
#include <nagm01.h>
void nag_quicksort (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_quicksort (m01csc) 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.

nag_quicksort (m01csc) is based on Singleton's implementation of the 'median-of-three' Quicksort algorithm, Singleton (1969), but with two additional modifications. First, small subfiles are sorted by an insertion sort on a separate final pass, Sedgewick (1978). 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.

4 References

Maclaren N M (1985) *Comput. J.* **28** 448

Sedgewick R (1978) Implementing Quicksort programs *Comm. ACM* **21** 847–857

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

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: **n** ≥ 0.

3: **size** – size_t *Input*

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

Constraint: **size** ≥ 1.

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

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

nag_quicksort (m01csc) 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.

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 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

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

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

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

NE_INT_ARG_GT

On entry, $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{n} \leq \langle value \rangle$.

On entry, $\mathbf{size} = \langle value \rangle$.
Constraint: $\mathbf{size} \leq \langle value \rangle$.
These arguments are limited to an implementation-dependent size which is printed in the error message.

NE_INT_ARG_LT

On entry, $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{n} \geq 0$.

On entry, **size** $= \langle value \rangle$.
Constraint: **size** ≥ 1.
The absolute value of **stride** must not be less than **size**.

7 Accuracy

Not applicable.

8 Parallelism and Performance

Not applicable.

9 Further Comments

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

10 Example

The example program reads a two-dimensional array of numbers and sorts the second column into ascending order.

10.1 Program Text

```
/* nag_quicksort (m01csc) Example Program.
 *
 * Copyright 1990 Numerical Algorithms Group.
 *
 * Mark 2 revised, 1992.
 * 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 VEC(I, J) vec[(I) *tdvec + J]
int main(void)
{
  Integer  exit_status = 0, i, j, k, m, n, tdvec;
  double   *vec = 0;
  NagError fail;

  INIT_FAIL(fail);

  /* Skip heading in data file */
  scanf("%*[^\n]");
  printf("nag_quicksort (m01csc) Example Program Results\n");
```

```
  scanf("%ld", &m);
  scanf("%ld", &n);
  scanf("%ld", &k);
  if (m >= 1 && n >= 1 && k >= 1)
    {
      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)
      scanf("%lf", &VEC(i, j));
  /* nag_quicksort (m01csc).
   * Quicksort of set of values of arbitrary data type
   */
  nag_quicksort((Pointer) &VEC(0, k-1), (size_t) m, sizeof(double),
                (ptrdiff_t)(sizeof(double)*n), compare, Nag_Ascending, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_quicksort (m01csc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  printf("\nMatrix with column %ld sorted \n", k);
  for (i = 0; i < m; ++i)
    {
      for (j = 0; j < n; ++j)
        printf(" %8.1f ", 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_quicksort (m01csc) Example Program Data
12 3 2
      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_quicksort (m01csc) Example Program Results

Matrix with column 2 sorted
      6.0       1.0       4.0
      5.0       2.0       1.0
      2.0       2.0       9.0
      4.0       3.0       6.0
      4.0       4.0       5.0
      4.0       4.0       2.0
      3.0       4.0       1.0
      2.0       5.0       6.0
      1.0       6.0       4.0
      9.0       9.0       2.0
      6.0       9.0       5.0
      4.0       9.0       6.0
```