NAG Library Function Document

nag_dsyrfs (f07mhc)

1 Purpose

nag_dsyrfs (f07mhc) returns error bounds for the solution of a real symmetric indefinite system of linear equations with multiple right-hand sides, $AX = B$. It improves the solution by iterative refinement, in order to reduce the backward error as much as possible.

2 Specification

```
#include <nag.h>
#include <nagf07.h>
```

```
void nag_dsyrfs (Nag_OrderType order, Nag_UploType uplo, Integer n,
     Integer nrhs, const double a[], Integer pda, const double af[],
     Integer pdaf, const Integer ipiv[], const double b[], Integer pdb,
     double x[], Integer pdx, double ferr[], double berr[], NagError *fail)
```

3 Description

nag_dsyrfs (f07mhc) returns the backward errors and estimated bounds on the forward errors for the solution of a real symmetric indefinite system of linear equations with multiple right-hand sides $AX = B$. The function handles each right-hand side vector (stored as a column of the matrix B) independently, so we describe the function of nag_dsyrfs (f07mhc) in terms of a single right-hand side b and solution x.

Given a computed solution x, the function computes the *component-wise backward error* β. This is the size of the smallest relative perturbation in each element of A and b such that x is the exact solution of a perturbed system

$$(A + \delta A)x = b + \delta b$$
$$\left|\delta a_{ij}\right| \leq \beta\left|a_{ij}\right| \quad \text{and} \quad |\delta b_i| \leq \beta|b_i|.$$

Then the function estimates a bound for the *component-wise forward error* in the computed solution, defined by:

$$\max_i|x_i - \hat{x}_i|/\max_i|x_i|$$

where $\hat{x}$ is the true solution.

For details of the method, see the f07 Chapter Introduction.

4 References

Golub G H and Van Loan C F (1996) *Matrix Computations* (3rd Edition) Johns Hopkins University Press, Baltimore

5 Arguments

1: **order** – Nag_OrderType *Input*

On entry: the **order** argument specifies the two-dimensional storage scheme being used, i.e., row-major ordering or column-major ordering. C language defined storage is specified by **order** = Nag_RowMajor. See Section 3.2.1.3 in the Essential Introduction for a more detailed explanation of the use of this argument.

Constraint: **order** = Nag_RowMajor or Nag_ColMajor.

2: **uplo** – Nag_UploType *Input*

On entry: specifies whether the upper or lower triangular part of A is stored and how A is to be factorized.

uplo = Nag_Upper
The upper triangular part of A is stored and A is factorized as $PUDU^{\mathrm{T}}P^{\mathrm{T}}$, where U is upper triangular.

uplo = Nag_Lower
The lower triangular part of A is stored and A is factorized as $PLDL^{\mathrm{T}}P^{\mathrm{T}}$, where L is lower triangular.

Constraint: **uplo** = Nag_Upper or Nag_Lower.

3: **n** – Integer *Input*

On entry: n, the order of the matrix A.

Constraint: $\mathbf{n} \geq 0$.

4: **nrhs** – Integer *Input*

On entry: r, the number of right-hand sides.

Constraint: $\mathbf{nrhs} \geq 0$.

5: **a**$[dim]$ – const double *Input*

Note: the dimension, *dim*, of the array **a** must be at least $\max(1, \mathbf{pda} \times \mathbf{n})$.

On entry: the n by n original symmetric matrix A as supplied to nag_dsytrf (f07mdc).

6: **pda** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) of the matrix in the array **a**.

Constraint: $\mathbf{pda} \geq \max(1, \mathbf{n})$.

7: **af**$[dim]$ – const double *Input*

Note: the dimension, *dim*, of the array **af** must be at least $\max(1, \mathbf{pdaf} \times \mathbf{n})$.

On entry: details of the factorization of A, as returned by nag_dsytrf (f07mdc).

8: **pdaf** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) of the matrix in the array **af**.

Constraint: $\mathbf{pdaf} \geq \max(1, \mathbf{n})$.

9: **ipiv**$[dim]$ – const Integer *Input*

Note: the dimension, *dim*, of the array **ipiv** must be at least $\max(1, \mathbf{n})$.

On entry: details of the interchanges and the block structure of D, as returned by nag_dsytrf (f07mdc).

10: **b**$[dim]$ – const double *Input*

Note: the dimension, *dim*, of the array **b** must be at least

$\max(1, \mathbf{pdb} \times \mathbf{nrhs})$ when **order** = Nag_ColMajor;
$\max(1, \mathbf{n} \times \mathbf{pdb})$ when **order** = Nag_RowMajor.

The (i,j)th element of the matrix B is stored in

b$[(j-1) \times$ **pdb** $+ i - 1]$ when **order** = Nag_ColMajor;
b$[(i-1) \times$ **pdb** $+ j - 1]$ when **order** = Nag_RowMajor.

On entry: the n by r right-hand side matrix B.

11: **pdb** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **b**.

Constraints:

if **order** = Nag_ColMajor, **pdb** $\geq \max(1, \mathbf{n})$;
if **order** = Nag_RowMajor, **pdb** $\geq \max(1, \mathbf{nrhs})$.

12: **x**$[dim]$ – double *Input/Output*

Note: the dimension, *dim*, of the array **x** must be at least

$\max(1, \mathbf{pdx} \times \mathbf{nrhs})$ when **order** = Nag_ColMajor;
$\max(1, \mathbf{n} \times \mathbf{pdx})$ when **order** = Nag_RowMajor.

The (i,j)th element of the matrix X is stored in

x$[(j-1) \times$ **pdx** $+ i - 1]$ when **order** = Nag_ColMajor;
x$[(i-1) \times$ **pdx** $+ j - 1]$ when **order** = Nag_RowMajor.

On entry: the n by r solution matrix X, as returned by nag_dsytrs (f07mec).

On exit: the improved solution matrix X.

13: **pdx** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **x**.

Constraints:

if **order** = Nag_ColMajor, **pdx** $\geq \max(1, \mathbf{n})$;
if **order** = Nag_RowMajor, **pdx** $\geq \max(1, \mathbf{nrhs})$.

14: **ferr**[**nrhs**] – double *Output*

On exit: **ferr**$[j-1]$ contains an estimated error bound for the jth solution vector, that is, the jth column of X, for $j = 1, 2, \ldots, r$.

15: **berr**[**nrhs**] – double *Output*

On exit: **berr**$[j-1]$ contains the component-wise backward error bound β for the jth solution vector, that is, the jth column of X, for $j = 1, 2, \ldots, r$.

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

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

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_INT

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

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

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

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

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

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

NE_INT_2

On entry, $\mathbf{pda} = \langle value \rangle$ and $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{pda} \geq \max(1, \mathbf{n})$.

On entry, $\mathbf{pdaf} = \langle value \rangle$ and $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{pdaf} \geq \max(1, \mathbf{n})$.

On entry, $\mathbf{pdb} = \langle value \rangle$ and $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{pdb} \geq \max(1, \mathbf{n})$.

On entry, $\mathbf{pdb} = \langle value \rangle$ and $\mathbf{nrhs} = \langle value \rangle$.
Constraint: $\mathbf{pdb} \geq \max(1, \mathbf{nrhs})$.

On entry, $\mathbf{pdx} = \langle value \rangle$ and $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{pdx} \geq \max(1, \mathbf{n})$.

On entry, $\mathbf{pdx} = \langle value \rangle$ and $\mathbf{nrhs} = \langle value \rangle$.
Constraint: $\mathbf{pdx} \geq \max(1, \mathbf{nrhs})$.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

7 Accuracy

The bounds returned in **ferr** are not rigorous, because they are estimated, not computed exactly; but in practice they almost always overestimate the actual error.

8 Parallelism and Performance

nag_dsyrfs (f07mhc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_dsyrfs (f07mhc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations. Each step of iterative refinement involves an additional $6n^2$ operations. At most five steps of iterative refinement are performed, but usually only one or two steps are required.

Estimating the forward error involves solving a number of systems of linear equations of the form $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution involves approximately $2n^2$ operations.

The complex analogues of this function are nag_zherfs (f07mvc) for Hermitian matrices and nag_zsyrfs (f07nvc) for symmetric matrices.

10 Example

This example solves the system of equations $AX = B$ using iterative refinement and to compute the forward and backward error bounds, where

$$A = \begin{pmatrix} 2.07 & 3.87 & 4.20 & -1.15 \\ 3.87 & -0.21 & 1.87 & 0.63 \\ 4.20 & 1.87 & 1.15 & 2.06 \\ -1.15 & 0.63 & 2.06 & -1.81 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} -9.50 & 27.85 \\ -8.38 & 9.90 \\ -6.07 & 19.25 \\ -0.96 & 3.93 \end{pmatrix}.$$

Here A is symmetric indefinite and must first be factorized by nag_dsytrf (f07mdc).

10.1 Program Text

```
/* nag_dsyrfs (f07mhc) Example Program.
 *
 * Copyright 2001 Numerical Algorithms Group.
 *
 * Mark 7, 2001.
 */

#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagf07.h>
#include <nagx04.h>

int main(void)
{
  /* Scalars */
  Integer       berr_len, ferr_len, i, j, n, nrhs;
  Integer       pda, pdaf, pdb, pdx;
  Integer       exit_status = 0;
  Nag_UploType  uplo;
  NagError      fail;
  Nag_OrderType order;
  /* Arrays */
  char          nag_enum_arg[40];
  Integer       *ipiv = 0;
  double        *a = 0, *af = 0, *b = 0, *berr = 0, *ferr = 0, *x = 0;

#ifdef NAG_COLUMN_MAJOR
#define A(I, J)  a[(J-1)*pda + I - 1]
#define AF(I, J) af[(J-1)*pdaf + I - 1]
#define B(I, J)  b[(J-1)*pdb + I - 1]
#define X(I, J)  x[(J-1)*pdx + I - 1]
  order = Nag_ColMajor;
#else
#define A(I, J)  a[(I-1)*pda + J - 1]
#define AF(I, J) af[(I-1)*pdaf + J - 1]
#define B(I, J)  b[(I-1)*pdb + J - 1]
#define X(I, J)  x[(I-1)*pdx + J - 1]
  order = Nag_RowMajor;
#endif
```

```
  INIT_FAIL(fail);

  printf("nag_dsyrfs (f07mhc) Example Program Results\n\n");

  /* Skip heading in data file */
  scanf("%*[^\n] ");
  scanf("%ld%ld%*[^\n] ", &n, &nrhs);
#ifdef NAG_COLUMN_MAJOR
  pda = n;
  pdaf = n;
  pdb = n;
  pdx = n;
#else
  pda = n;
  pdaf = n;
  pdb = nrhs;
  pdx = nrhs;
#endif
  ferr_len = nrhs;
  berr_len = nrhs;

  /* Allocate memory */
  if (!(ipiv = NAG_ALLOC(n, Integer)) ||
      !(a = NAG_ALLOC(n * n, double)) ||
      !(af = NAG_ALLOC(n * n, double)) ||
      !(b = NAG_ALLOC(n * nrhs, double)) ||
      !(berr = NAG_ALLOC(berr_len, double)) ||
      !(ferr = NAG_ALLOC(ferr_len, double)) ||
      !(x = NAG_ALLOC(n * nrhs, double)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Read A and B from data file, and copy A to AF and B to X */
  scanf(" %39s%*[^\n] ", nag_enum_arg);
  /* nag_enum_name_to_value (x04nac).
   * Converts NAG enum member name to value
   */
  uplo = (Nag_UploType) nag_enum_name_to_value(nag_enum_arg);

  if (uplo == Nag_Upper)
    {
      for (i = 1; i <= n; ++i)
        {
          for (j = i; j <= n; ++j)
            scanf("%lf", &A(i, j));
        }
      scanf("%*[^\n] ");
    }
  else
    {
      for (i = 1; i <= n; ++i)
        {
          for (j = 1; j <= i; ++j)
            scanf("%lf", &A(i, j));
        }
      scanf("%*[^\n] ");
    }

  for (i = 1; i <= n; ++i)
    {
      for (j = 1; j <= nrhs; ++j)
        scanf("%lf", &B(i, j));
    }
  scanf("%*[^\n] ");
  /* Copy A to AF and B to X */
  if (uplo == Nag_Upper)
    {
```

```
        for (i = 1; i <= n; ++i)
          {
            for (j = i; j <= n; ++j)
              AF(i, j) = A(i, j);
          }
      }
    else
      {
        for (i = 1; i <= n; ++i)
          {
            for (j = 1; j <= i; ++j)
              AF(i, j) = A(i, j);
          }
      }
    for (i = 1; i <= n; ++i)
      {
        for (j = 1; j <= nrhs; ++j)
          X(i, j) = B(i, j);
      }
    /* Factorize A in the array AF */
    /* nag_dsytrf (f07mdc).
     * Bunch-Kaufman factorization of real symmetric indefinite
     * matrix
     */
    nag_dsytrf(order, uplo, n, af, pdaf, ipiv, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from nag_dsytrf (f07mdc).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
    /* Compute solution in the array X */
    /* nag_dsytrs (f07mec).
     * Solution of real symmetric indefinite system of linear
     * equations, multiple right-hand sides, matrix already
     * factorized by nag_dsytrf (f07mdc)
     */
    nag_dsytrs(order, uplo, n, nrhs, af, pdaf, ipiv, x, pdx,
               &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from nag_dsytrs (f07mec).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
    /* Improve solution, and compute backward errors and */
    /* estimated bounds on the forward errors */
    /* nag_dsyrfs (f07mhc).
     * Refined solution with error bounds of real symmetric
     * indefinite system of linear equations, multiple
     * right-hand sides
     */
    nag_dsyrfs(order, uplo, n, nrhs, a, pda, af, pdaf, ipiv,
               b, pdb, x, pdx, ferr, berr, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from nag_dsyrfs (f07mhc).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }

    /* Print solution */
    /* nag_gen_real_mat_print (x04cac).
     * Print real general matrix (easy-to-use)
     */
    fflush(stdout);
    nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, nrhs, x,
                           pdx, "Solution(s)", 0, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n",
```

```
                fail.message);
        exit_status = 1;
        goto END;
      }

  printf("\nBackward errors (machine-dependent)\n");
  for (j = 1; j <= nrhs; ++j)
    printf("%11.1e%s", berr[j-1], j%7 == 0?"\n":" ");
  printf("\nEstimated forward error bounds"
          "(machine-dependent)\n");
  for (j = 1; j <= nrhs; ++j)
    printf("%11.1e%s", ferr[j-1], j%7 == 0 || j == nrhs?"\n":" ");
 END:
  NAG_FREE(ipiv);
  NAG_FREE(a);
  NAG_FREE(af);
  NAG_FREE(b);
  NAG_FREE(berr);
  NAG_FREE(ferr);
  NAG_FREE(x);
  return exit_status;
}
```

10.2 Program Data

```
nag_dsyrfs (f07mhc) Example Program Data
  4  2                         :Values of n and nrhs
  Nag_Lower                    :Value of uplo
  2.07
  3.87  -0.21
  4.20   1.87   1.15
 -1.15   0.63   2.06  -1.81    :End of matrix A
 -9.50  27.85
 -8.38   9.90
 -6.07  19.25
 -0.96   3.93                  :End of matrix B
```

10.3 Program Results

```
nag_dsyrfs (f07mhc) Example Program Results

 Solution(s)
            1          2
 1     -4.0000     1.0000
 2     -1.0000     4.0000
 3      2.0000     3.0000
 4      5.0000     2.0000

Backward errors (machine-dependent)
    5.7e-17     1.0e-16
Estimated forward error bounds(machine-dependent)
    2.3e-14     3.4e-14
```