NAG Library Function Document

nag_dgesdd (f08kdc)

1 Purpose

nag_dgesdd (f08kdc) computes the singular value decomposition (SVD) of a real m by n matrix A, optionally computing the left and/or right singular vectors, by using a divide-and-conquer method.

2 Specification

```
#include <nag.h>
#include <nagf08.h>
void nag_dgesdd (Nag_OrderType order, Nag_JobType job, Integer m, Integer n,
     double a[], Integer pda, double s[], double u[], Integer pdu,
     double vt[], Integer pdvt, NagError *fail)
```

3 Description

The SVD is written as

$$A = U\Sigma V^{\mathrm{T}},$$

where Σ is an m by n matrix which is zero except for its $\min(m,n)$ diagonal elements, U is an m by m orthogonal matrix, and V is an n by n orthogonal matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in descending order. The first $\min(m,n)$ columns of U and V are the left and right singular vectors of A.

Note that the function returns V^{T}, not V.

4 References

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J J, Du Croz J J, Greenbaum A, Hammarling S, McKenney A and Sorensen D (1999) *LAPACK Users' Guide* (3rd Edition) SIAM, Philadelphia http://www.netlib.org/lapack/lug

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: **job** – Nag_JobType *Input*

On entry: specifies options for computing all or part of the matrix U.

job = Nag_DoAll
All m columns of U and all n rows of V^{T} are returned in the arrays **u** and **vt**.

job = Nag_DoSquare
The first $\min(m, n)$ columns of U and the first $\min(m, n)$ rows of V^{T} are returned in the arrays **u** and **vt**.

job = Nag_DoOverwrite
If $\mathbf{m} \geq \mathbf{n}$, the first n columns of U are overwritten on the array **a** and all rows of V^{T} are returned in the array **vt**. Otherwise, all columns of U are returned in the array **u** and the first m rows of V^{T} are overwritten in the array **vt**.

job = Nag_DoNothing
No columns of U or rows of V^{T} are computed.

Constraint: **job** = Nag_DoAll, Nag_DoSquare, Nag_DoOverwrite or Nag_DoNothing.

3: **m** – Integer *Input*

On entry: m, the number of rows of the matrix A.

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

4: **n** – Integer *Input*

On entry: n, the number of columns of the matrix A.

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

5: $\mathbf{a}[dim]$ – double *Input/Output*

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

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

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

$\mathbf{a}[(j-1) \times \mathbf{pda} + i - 1]$ when **order** = Nag_ColMajor;
$\mathbf{a}[(i-1) \times \mathbf{pda} + j - 1]$ when **order** = Nag_RowMajor.

On entry: the m by n matrix A.

On exit: if **job** = Nag_DoOverwrite, **a** is overwritten with the first n columns of U (the left singular vectors, stored column-wise) if $\mathbf{m} \geq \mathbf{n}$; **a** is overwritten with the first m rows of V^{T} (the right singular vectors, stored row-wise) otherwise.

If **job** $\neq$ Nag_DoOverwrite, the contents of **a** are destroyed.

6: **pda** – Integer *Input*

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

Constraints:

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

7: $\mathbf{s}[\min(\mathbf{m}, \mathbf{n})]$ – double *Output*

On exit: the singular values of A, sorted so that $\mathbf{s}[i-1] \geq \mathbf{s}[i]$.

8: $\mathbf{u}[dim]$ – double *Output*

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

$\max(1, \mathbf{pdu} \times \mathbf{m})$ when **job** = Nag_DoAll or **job** = Nag_DoOverwrite and $\mathbf{m} < \mathbf{n}$;
$\max(1, \mathbf{pdu} \times \min(\mathbf{m}, \mathbf{n}))$ when **job** = Nag_DoSquare and **order** = Nag_ColMajor;
$\max(1, \mathbf{m} \times \mathbf{pdu})$ when **job** = Nag_DoSquare and **order** = Nag_RowMajor;
$\max(1, \mathbf{m})$ otherwise.

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

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

On exit:

If **job** = Nag_DoAll or **job** = Nag_DoOverwrite and **m** < **n**, **u** contains the m by m orthogonal matrix U.

If **job** = Nag_DoSquare, **u** contains the first $\min(m, n)$ columns of U (the left singular vectors, stored column-wise).

If **job** = Nag_DoOverwrite and **m** $\geq$ **n**, or **job** = Nag_DoNothing, **u** is not referenced.

9: **pdu** – Integer *Input*

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

Constraints:

> if **order** = Nag_ColMajor,
>
> > if **job** = Nag_DoAll or **job** = Nag_DoOverwrite and **m** < **n**, **pdu** $\geq \max(1,$ **m**$)$;
> > if **job** = Nag_DoSquare, **pdu** $\geq \max(1,$ **m**$)$;
> > otherwise **pdu** ≥ 1.;
>
> if **order** = Nag_RowMajor,
>
> > if **job** = Nag_DoAll or **job** = Nag_DoOverwrite and **m** < **n**, **pdu** $\geq \max(1,$ **m**$)$;
> > if **job** = Nag_DoSquare, **pdu** $\geq \max(1, \min($**m**, **n**$))$;
> > otherwise **pdu** ≥ 1..

10: **vt**[*dim*] – double *Output*

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

> $\max(1,$ **pdvt** $\times$ **n**$)$ when **job** = Nag_DoAll or **job** = Nag_DoOverwrite and **m** $\geq$ **n**;
> $\max(1,$ **pdvt** $\times$ **n**$)$ when **job** = Nag_DoSquare and **order** = Nag_ColMajor;
> $\max(1, \min($**m**, **n**$) \times$ **pdvt**$)$ when **job** = Nag_DoSquare and **order** = Nag_RowMajor;
> $\max(1, \min($**m**, **n**$))$ otherwise.

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

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

On exit: if **job** = Nag_DoAll or **job** = Nag_DoOverwrite and **m** $\geq$ **n**, **vt** contains the n by n orthogonal matrix V^{T}.

If **job** = Nag_DoSquare, **vt** contains the first $\min(m, n)$ rows of V^{T} (the right singular vectors, stored row-wise).

If **job** = Nag_DoOverwrite and **m** < **n**, or **job** = Nag_DoNothing, **vt** is not referenced.

11: **pdvt** – Integer *Input*

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

Constraints:

> if **order** = Nag_ColMajor,
>
> > if **job** = Nag_DoAll or **job** = Nag_DoOverwrite and **m** $\geq$ **n**, **pdvt** $\geq \max(1,$ **n**$)$;
> > if **job** = Nag_DoSquare, **pdvt** $\geq \max(1, \min($**m**, **n**$))$;
> > otherwise **pdvt** ≥ 1.;

if **order** $=$ Nag_RowMajor,

if **job** $=$ Nag_DoAll or **job** $=$ Nag_DoOverwrite and $\mathbf{m} \geq \mathbf{n}$, $\mathbf{pdvt} \geq \max(1, \mathbf{n})$;
if **job** $=$ Nag_DoSquare, $\mathbf{pdvt} \geq \max(1, \mathbf{n})$;
otherwise $\mathbf{pdvt} \geq 1$..

12: **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.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_BAD_PARAM

On entry, argument $\langle value \rangle$ had an illegal value.

NE_CONVERGENCE

nag_dgesdd (f08kdc) did not converge, the updating process failed.

NE_ENUM_INT_3

On entry, **job** $= \langle value \rangle$, **pdu** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: if **job** $=$ Nag_DoAll or **job** $=$ Nag_DoOverwrite and $\mathbf{m} < \mathbf{n}$, $\mathbf{pdu} \geq \max(1, \mathbf{m})$;
if **job** $=$ Nag_DoSquare, $\mathbf{pdu} \geq \max(1, \mathbf{m})$;
otherwise $\mathbf{pdu} \geq 1$.

On entry, **job** $= \langle value \rangle$, **pdu** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: if **job** $=$ Nag_DoAll or **job** $=$ Nag_DoOverwrite and $\mathbf{m} < \mathbf{n}$, $\mathbf{pdu} \geq \max(1, \mathbf{m})$;
if **job** $=$ Nag_DoSquare, $\mathbf{pdu} \geq \max(1, \min(\mathbf{m}, \mathbf{n}))$;
otherwise $\mathbf{pdu} \geq 1$.

On entry, **job** $= \langle value \rangle$, **pdvt** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: if **job** $=$ Nag_DoAll or **job** $=$ Nag_DoOverwrite and $\mathbf{m} \geq \mathbf{n}$, $\mathbf{pdvt} \geq \max(1, \mathbf{n})$;
if **job** $=$ Nag_DoSquare, $\mathbf{pdvt} \geq \max(1, \min(\mathbf{m}, \mathbf{n}))$;
otherwise $\mathbf{pdvt} \geq 1$.

On entry, **job** $= \langle value \rangle$, **pdvt** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: if **job** $=$ Nag_DoAll or **job** $=$ Nag_DoOverwrite and $\mathbf{m} \geq \mathbf{n}$, $\mathbf{pdvt} \geq \max(1, \mathbf{n})$;
if **job** $=$ Nag_DoSquare, $\mathbf{pdvt} \geq \max(1, \mathbf{n})$;
otherwise $\mathbf{pdvt} \geq 1$.

NE_INT

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

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

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

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

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

NE_INT_2

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

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

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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

7 Accuracy

The computed singular value decomposition is nearly the exact singular value decomposition for a nearby matrix $(A + E)$, where

$$\|E\|_2 = O(\epsilon)\|A\|_2,$$

and ϵ is the ***machine precision***. In addition, the computed singular vectors are nearly orthogonal to working precision. See Section 4.9 of Anderson *et al.* (1999) for further details.

8 Parallelism and Performance

nag_dgesdd (f08kdc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_dgesdd (f08kdc) 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 X06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

The total number of floating-point operations is approximately proportional to mn^2 when $m > n$ and m^2n otherwise.

The singular values are returned in descending order.

The complex analogue of this function is nag_zgesvd (f08kpc).

10 Example

This example finds the singular values and left and right singular vectors of the 4 by 6 matrix

$$A = \begin{pmatrix} 2.27 & 0.28 & -0.48 & 1.07 & -2.35 & 0.62 \\ -1.54 & -1.67 & -3.09 & 1.22 & 2.93 & -7.39 \\ 1.15 & 0.94 & 0.99 & 0.79 & -1.45 & 1.03 \\ -1.94 & -0.78 & -0.21 & 0.63 & 2.30 & -2.57 \end{pmatrix},$$

together with approximate error bounds for the computed singular values and vectors.

The example program for nag_dgesvd (f08kbc) illustrates finding a singular value decomposition for the case $m \geq n$.

10.1 Program Text

```
/* nag_dgesdd (f08kdc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 23, 2011.
 */

#include <stdio.h>
#include <nag.h>
#include <nagx04.h>
#include <nag_stdlib.h>
#include <nagf08.h>
#include <nagx02.h>

int main(void)
{

  /* Scalars */
  double        eps, serrbd;
  Integer       exit_status = 0, i, j, m, n, pda, pdu;

  /* Arrays */
  double        *a = 0, *rcondu = 0, *rcondv = 0, *s = 0, *u = 0;
  double        *uerrbd = 0, *verrbd = 0;
  double        dummy[1];

  /* Nag Types */
  NagError      fail;
  Nag_OrderType order;

#ifdef NAG_COLUMN_MAJOR
#define A(I, J) a[(J - 1) * pda + I - 1]
  order = Nag_ColMajor;
#else
#define A(I, J) a[(I - 1) * pda + J - 1]
  order = Nag_RowMajor;
#endif

  INIT_FAIL(fail);

  printf("nag_dgesdd (f08kdc) Example Program Results\n\n");

  /* Skip heading in data file */
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif

#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%*[^\n]", &m, &n);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%*[^\n]", &m, &n);
#endif
  if (m < 0 && n < 0)
    {
      printf("Invalid m or n\n");
      exit_status = 1;
      goto END;
    }

  /* Allocate memory */
  if (!(a      = NAG_ALLOC(m * n, double)) ||
      !(rcondu = NAG_ALLOC(m, double)) ||
      !(rcondv = NAG_ALLOC(m, double)) ||
```

```
      !(s      = NAG_ALLOC(MIN(m, n), double)) ||
      !(u      = NAG_ALLOC(m * m, double)) ||
      !(uerrbd = NAG_ALLOC(m, double)) ||
      !(verrbd = NAG_ALLOC(m, double))
      )
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

#ifdef NAG_COLUMN_MAJOR
  pda = m;
  pdu = m;
#else
  pda = n;
  pdu = MIN(m, n);
#endif

  /* Read the m by n matrix A from data file */
  for (i = 1; i <= m; ++i)
#ifdef _WIN32
    for (j = 1; j <= n; ++j) scanf_s("%lf", &A(i, j));
#else
    for (j = 1; j <= n; ++j) scanf("%lf", &A(i, j));
#endif
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif

  /* nag_dgesdd (f08kdc).
   * Compute the singular values and left and right singular vectors
   * of A (A = U*S*(V**T), m.le.n)
   */
  nag_dgesdd(order, Nag_DoOverwrite, m, n, a, pda, s, u, pdu, dummy, 1, &fail);

  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_dgesdd (f08kdc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Print singular values */
  printf("Singular values\n");
  for (i = 0; i < m; ++i) printf(" %7.4f%s", s[i], i%8 == 7?"\n":"");
  printf("\n\n");

  /* Print left and right singular vectors using
   * nag_gen_real_mat_print (x04cac).
   */
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, m, m, u,
                         pdu, "Left singular vectors", 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("\n");

  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, m, n, a,
                         pda, "Right singular vectors by row",
                         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;
    }

  /* Get the machine precision, eps using nag_machine_precision (x02ajc). */
  eps = nag_machine_precision;

  /* compute the approximate error bound for the computed singular values.
   * Note that for the 2-norm, s[0] = ||A||
   */
  serrbd = eps * s[0];

  /* Call nag_ddisna (f08flc) to estimate reciprocal condition numbers for
   * the singular vectors.
   */
  nag_ddisna(Nag_LeftSingVecs, m, n, s, rcondu, &fail);
  nag_ddisna(Nag_RightSingVecs, m, n, s, rcondv, &fail);

  /* Compute the error estimates for the singular vectors. */
  for (i = 0; i < m; ++i)
    {
      uerrbd[i] = serrbd / rcondu[i];
      verrbd[i] = serrbd / rcondv[i];
    }

  /* Print the approximate error bounds for the singular values and vectors */
  printf("\nError estimate for the singular values\n%11.1e\n", serrbd);

  printf("\nError estimates for the left singular vectors\n");
  for (i = 0; i < m; ++i) printf(" %10.1e%s", uerrbd[i], i%6 == 5?"\n":"");

  printf("\n\nError estimates for the right singular vectors\n");
  for (i = 0; i < m; ++i) printf(" %10.1e%s", verrbd[i], i%6 == 5?"\n":"");
  printf("\n");

 END:
  NAG_FREE(a);
  NAG_FREE(rcondu);
  NAG_FREE(rcondv);
  NAG_FREE(s);
  NAG_FREE(u);
  NAG_FREE(uerrbd);
  NAG_FREE(verrbd);

  return exit_status;
}
#undef A
```

10.2 Program Data

```
nag_dgesdd (f08kdc) Example Program Data

   4      6                                : m and n

   2.27   0.28  -0.48   1.07  -2.35   0.62
  -1.54  -1.67  -3.09   1.22   2.93  -7.39
   1.15   0.94   0.99   0.79  -1.45   1.03
  -1.94  -0.78  -0.21   0.63   2.30  -2.57  : matrix A
```

10.3 Program Results

```
nag_dgesdd (f08kdc) Example Program Results

Singular values
  9.9966  3.6831  1.3569  0.5000

 Left singular vectors
          1       2       3       4
 1  -0.1921  0.8030 -0.0041  0.5642
 2   0.8794  0.3926  0.0752 -0.2587
```

```
 3  -0.2140  0.2980 -0.7827 -0.5027
 4   0.3795 -0.3351 -0.6178  0.6017

 Right singular vectors by row
         1       2       3       4       5       6
 1  -0.2774 -0.2020 -0.2918  0.0938  0.4213 -0.7816
 2   0.6003  0.0301 -0.3348  0.3699 -0.5266 -0.3353
 3   0.1277 -0.2805 -0.6453 -0.6781 -0.0413  0.1645
 4  -0.1323 -0.7034 -0.1906  0.5399  0.0575  0.3957

Error estimate for the singular values
    1.1e-15

Error estimates for the left singular vectors
    1.8e-16    4.8e-16    1.3e-15    1.3e-15

Error estimates for the right singular vectors
    1.8e-16    4.8e-16    1.3e-15    2.2e-15
```