

# NAG Library Function Document

## nag\_dbdsdc (f08mdc)

### 1 Purpose

nag\_dbdsdc (f08mdc) computes the singular values and, optionally, the left and right singular vectors of a real  $n$  by  $n$  (upper or lower) bidiagonal matrix  $B$ .

### 2 Specification

```
#include <nag.h>
#include <nagf08.h>

void nag_dbdsdc (Nag_OrderType order, Nag_UploType uplo,
                Nag_ComputeSingularVecsType compq, Integer n, double d[], double e[],
                double u[], Integer pdu, double vt[], Integer pdvt, double q[],
                Integer iq[], NagError *fail)
```

### 3 Description

nag\_dbdsdc (f08mdc) computes the singular value decomposition (SVD) of the (upper or lower) bidiagonal matrix  $B$  as

$$B = USV^T,$$

where  $S$  is a diagonal matrix with non-negative diagonal elements  $s_{ii} = s_i$ , such that

$$s_1 \geq s_2 \geq \dots \geq s_n \geq 0,$$

and  $U$  and  $V$  are orthogonal matrices. The diagonal elements of  $S$  are the singular values of  $B$  and the columns of  $U$  and  $V$  are respectively the corresponding left and right singular vectors of  $B$ .

When only singular values are required the function uses the  $QR$  algorithm, but when singular vectors are required a divide and conquer method is used. The singular values can optionally be returned in compact form, although currently no function is available to apply  $U$  or  $V$  when stored in compact form.

### 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: **uplo** – Nag\_UploType *Input*  
*On entry:* indicates whether  $B$  is upper or lower bidiagonal.  
**uplo** = Nag\_Upper  
 $B$  is upper bidiagonal.  
**uplo** = Nag\_Lower  
 $B$  is lower bidiagonal.  
*Constraint:* **uplo** = Nag\_Upper or Nag\_Lower.
- 3: **compq** – Nag\_ComputeSingularVecsType *Input*  
*On entry:* specifies whether singular vectors are to be computed.  
**compq** = Nag\_NotSingularVecs  
 Compute singular values only.  
**compq** = Nag\_PackedSingularVecs  
 Compute singular values and compute singular vectors in compact form.  
**compq** = Nag\_SingularVecs  
 Compute singular values and singular vectors.  
*Constraint:* **compq** = Nag\_NotSingularVecs, Nag\_PackedSingularVecs or Nag\_SingularVecs.
- 4: **n** – Integer *Input*  
*On entry:*  $n$ , the order of the matrix  $B$ .  
*Constraint:*  $n \geq 0$ .
- 5: **d**[*dim*] – double *Input/Output*  
**Note:** the dimension,  $dim$ , of the array **d** must be at least  $\max(1, n)$ .  
*On entry:* the  $n$  diagonal elements of the bidiagonal matrix  $B$ .  
*On exit:* if **fail.code** = NE\_NOERROR, the singular values of  $B$ .
- 6: **e**[*dim*] – double *Input/Output*  
**Note:** the dimension,  $dim$ , of the array **e** must be at least  $\max(1, n - 1)$ .  
*On entry:* the  $(n - 1)$  off-diagonal elements of the bidiagonal matrix  $B$ .  
*On exit:* the contents of **e** are destroyed.
- 7: **u**[*dim*] – double *Output*  
**Note:** the dimension,  $dim$ , of the array **u** must be at least  
 $\max(1, pdu \times n)$  when **compq** = Nag\_SingularVecs;  
 1 otherwise.  
 The  $(i, j)$ th element of the matrix  $U$  is stored in  
 $\mathbf{u}[(j - 1) \times pdu + i - 1]$  when **order** = Nag\_ColMajor;  
 $\mathbf{u}[(i - 1) \times pdu + j - 1]$  when **order** = Nag\_RowMajor.  
*On exit:* if **compq** = Nag\_SingularVecs, then if **fail.code** = NE\_NOERROR, **u** contains the left singular vectors of the bidiagonal matrix  $B$ .  
 If **compq**  $\neq$  Nag\_SingularVecs, **u** is not referenced.
- 8: **pdu** – Integer *Input*  
*On entry:* the stride separating row or column elements (depending on the value of **order**) in the array **u**.

*Constraints:*

if **compq** = Nag\_SingularVecs, **pdu**  $\geq$  max(1, **n**);  
otherwise **pdu**  $\geq$  1.

9: **vt**[*dim*] – double

*Output*

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

max(1, **pdvt**  $\times$  **n**) when **compq** = Nag\_SingularVecs;  
1 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 **compq** = Nag\_SingularVecs, then if **fail.code** = NE\_NOERROR, the rows of **vt** contain the right singular vectors of the bidiagonal matrix *B*.

If **compq**  $\neq$  Nag\_SingularVecs, **vt** is not referenced.

10: **pdvt** – Integer

*Input*

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

*Constraints:*

if **compq** = Nag\_SingularVecs, **pdvt**  $\geq$  max(1, **n**);  
otherwise **pdvt**  $\geq$  1.

11: **q**[*dim*] – double

*Output*

**Note:** the dimension, *dim*, of the array **q** must be at least max(1, **n**<sup>2</sup> + 5**n**, *ldq*).

*On exit:* if **compq** = Nag\_PackedSingularVecs, then if **fail.code** = NE\_NOERROR, **q** and **iq** contain the left and right singular vectors in a compact form, requiring  $\bar{O}(\mathbf{n} \log_2 \mathbf{n})$  space instead of  $2 \times \mathbf{n}^2$ . In particular, **q** contains all the real data in the first *ldq* = **n**  $\times$  (11 + 2  $\times$  *smlsiz* + 8  $\times$  int( $\log_2(\mathbf{n}/(\mathit{smlsiz} + 1))$ )) elements of **q**, where *smlsiz* is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25).

If **compq**  $\neq$  Nag\_PackedSingularVecs, **q** is not referenced.

12: **iq**[*dim*] – Integer

*Output*

**Note:** the dimension, *dim*, of the array **iq** must be at least max(1, *ldiq*).

*On exit:* if **compq** = Nag\_PackedSingularVecs, then if **fail.code** = NE\_NOERROR, **q** and **iq** contain the left and right singular vectors in a compact form, requiring  $\bar{O}(\mathbf{n} \log_2 \mathbf{n})$  space instead of  $2 \times \mathbf{n}^2$ . In particular, **iq** contains all integer data in the first *ldiq* = **n**  $\times$  (3 + 3  $\times$  int( $\log_2(\mathbf{n}/(\mathit{smlsiz} + 1))$ )) elements of **iq**, where *smlsiz* is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25).

If **compq**  $\neq$  Nag\_PackedSingularVecs, **iq** is not referenced.

13: **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\_ENUM\_INT\_2

On entry, **compq** =  $\langle value \rangle$ , **pdu** =  $\langle value \rangle$  and **n** =  $\langle value \rangle$ .

Constraint: if **compq** = Nag\_SingularVecs, **pdu**  $\geq \max(1, \mathbf{n})$ ;  
otherwise **pdu**  $\geq 1$ .

On entry, **compq** =  $\langle value \rangle$ , **pdvt** =  $\langle value \rangle$  and **n** =  $\langle value \rangle$ .

Constraint: if **compq** = Nag\_SingularVecs, **pdvt**  $\geq \max(1, \mathbf{n})$ ;  
otherwise **pdvt**  $\geq 1$ .

### NE\_INT

On entry, **n** =  $\langle value \rangle$ .

Constraint: **n**  $\geq 0$ .

On entry, **pdu** =  $\langle value \rangle$ .

Constraint: **pdu**  $> 0$ .

On entry, **pdvt** =  $\langle value \rangle$ .

Constraint: **pdvt**  $> 0$ .

### 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.

### NE\_SINGULAR

The algorithm failed to compute a singular value. The update process of divide-and-conquer failed.

## 7 Accuracy

Each computed singular value of  $B$  is accurate to nearly full relative precision, no matter how tiny the singular value. The  $i$ th computed singular value,  $\hat{s}_i$ , satisfies the bound

$$|\hat{s}_i - s_i| \leq p(n)\epsilon s_i$$

where  $\epsilon$  is the *machine precision* and  $p(n)$  is a modest function of  $n$ .

For bounds on the computed singular values, see Section 4.9.1 of Anderson *et al.* (1999). See also nag\_ddisna (f08flc).

## 8 Parallelism and Performance

nag\_dbdsdc (f08mdc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag\_dbdsdc (f08mdc) 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

If only singular values are required, the total number of floating-point operations is approximately proportional to  $n^2$ . When singular vectors are required the number of operations is bounded above by approximately the same number of operations as nag\_dbdsqr (f08mec), but for large matrices nag\_dbdsdc (f08mdc) is usually much faster.

There is no complex analogue of nag\_dbdsdc (f08mdc).

## 10 Example

This example computes the singular value decomposition of the upper bidiagonal matrix

$$B = \begin{pmatrix} 3.62 & 1.26 & 0 & 0 \\ 0 & -2.41 & -1.53 & 0 \\ 0 & 0 & 1.92 & 1.19 \\ 0 & 0 & 0 & -1.43 \end{pmatrix}.$$

### 10.1 Program Text

```

/* nag_dbdsdc (f08mdc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 23, 2011.
 */

#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nagx02.h>
#include <nagx04.h>
#include <nag_stdlib.h>
#include <nagf08.h>
#include <nagf16.h>

int main(void)
{
    /* Scalars */
    double      alpha, beta, eps, norm;
    Integer     abi, i, j, k1, k2, leniq, lenq, mlvl, n, pdab, pdb, pdu, pdvt;
    Integer     exit_status = 0, smlsiz = 25;

    /* Arrays */
    double      *ab = 0, *b = 0, *d = 0, *e = 0, *q = 0, *u = 0, *vt = 0;
    Integer     *iq = 0;
    char        nag_enum_arg[40];

    /* Nag Types */
    NagError    fail;
    Nag_OrderType order;
    Nag_UploType uplo;
    Nag_ComputeSingularVecsType compq;

```

```

#ifdef NAG_COLUMN_MAJOR
#define B(I, J) b[(J - 1) * pdb + I - 1]
#define U(I, J) u[(J - 1) * pdu + I - 1]
    order = Nag_ColMajor;
#else
#define B(I, J) b[(I - 1) * pdb + J - 1]
#define U(I, J) u[(I - 1) * pdu + J - 1]
    order = Nag_RowMajor;
#endif

    INIT_FAIL(fail);

    printf("nag_dbdsdc (f08mdc) 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"%*[\n]", &n);
#else
    scanf("%"NAG_IFMT"%*[\n]", &n);
#endif
    if (n < 0)
    {
        printf("Invalid n\n");
        exit_status = 1;
        goto END;;
    }
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, _countof(nag_enum_arg));
#else
    scanf(" %39s%*[\n]", nag_enum_arg);
#endif
    /* nag_enum_name_to_value (x04nac).
     * Converts NAG enum member name to value
     */
    uplo = (Nag_UploType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, _countof(nag_enum_arg));
#else
    scanf(" %39s%*[\n]", nag_enum_arg);
#endif
    /* Starting index for main diagonal in banded storage format = abi. */
    if ((order==Nag_ColMajor && uplo==Nag_Lower) ||
        (order==Nag_RowMajor && uplo==Nag_Upper)) {
        abi = 0;
    } else {
        abi = 1;
    }

    compq = (Nag_ComputeSingularVecsType) nag_enum_name_to_value(nag_enum_arg);
    /* size of u, vt, q and iq depends on value of compq input */
    if (compq==Nag_SingularVecs) {
        pdu = n;
        pdvt = n;
    } else {
        pdu = 1;
        pdvt = 1;
    }
    if (compq==Nag_PackedSingularVecs) {
        mlvl = (Integer) (log(n/(smlsiz+1.0))/log(2.0)) + 1;
        if (mlvl < 1) mlvl = 1;
        lenq = MAX(n*n+5*n, n*(3 + 2*smlsiz + 8*mlvl));
        leniq = n*3*mlvl;
    } else {
        lenq = 1;
        leniq = 1;
    }

```

```

}

pdb = n;
pdab = 2;
/* Allocate memory */
if (!(b = NAG_ALLOC(n*n, double)) ||
    !(ab = NAG_ALLOC(pdab*n, double)) ||
    !(d = NAG_ALLOC(n, double)) ||
    !(e = NAG_ALLOC(n-1, double)) ||
    !(q = NAG_ALLOC(lenq, double)) ||
    !(u = NAG_ALLOC(pdu*pdu, double)) ||
    !(vt = NAG_ALLOC(pdvt*pdvt, double)) ||
    !(iq = NAG_ALLOC(leniq, Integer)))
{
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
}

/* Read the bidiagonal matrix B from data file,
 * first the diagonal elements, and then the off-diagonal elements.
 */
#ifdef _WIN32
    for (i = 0; i < n; ++i) scanf_s("%lf", &d[i]);
#else
    for (i = 0; i < n; ++i) scanf("%lf", &d[i]);
#endif
#ifdef _WIN32
    scanf_s("%*[\n]");
#else
    scanf("%*[\n]");
#endif
#ifdef _WIN32
    for (i = 0; i < n - 1; ++i) scanf_s("%lf", &e[i]);
#else
    for (i = 0; i < n - 1; ++i) scanf("%lf", &e[i]);
#endif
#ifdef _WIN32
    scanf_s("%*[\n]");
#else
    scanf("%*[\n]");
#endif

/* Store diagonal arrays in banded format in ab for printing */
for(i=0; i<n; i++) ab[2*i+abi] = d[i];
for(i=0; i<n-1; i++) ab[2*i+abi+1] = e[i];

/* k1 = lower bandwidth, k2 = upper bandwidth */
k1 = (uplo==Nag_Upper ? 0 : 1);
k2 = 1 - k1;

/* Print Bidiagonal Matrix B stored in ab.
 * nag_band_real_mat_print (x04cec).
 * Print real packed banded matrix (easy-to-use)
 */
fflush(stdout);
nag_band_real_mat_print(order, n, n, k1, k2, ab, 2, "Matrix B", 0, &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error from nag_band_real_mat_print (x04cec).\n%s\n",
        fail.message);
    exit_status = 1;
    goto END;
}

/* Calculate the singular values and left and right singular vectors of B
 * using nag_dbdsdc (f08mdc).
 */
nag_dbdsdc(order, uplo, compq, n, d, e, u, pdu, vt, pdvt, q, iq, &fail);
if (fail.code != NE_NOERROR)
{

```

```

    printf("Error from nag_dbdsdc (f08mdc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

printf("\nSingular values\n");
for (i = 0; i < n; ++i) printf(" %8.4f%s", d[i], i%8 == 7?"\n":""");
printf("\n\n");

if (compq==Nag_SingularVecs) {
    /* Reconstruct bidiagonal matrix from decomposition:
     * first, U <- U*S, then Compute B = U*S*V^T.
     * nag_dgemm (f16yac).
     */
    for(i = 1; i <= n; i++)
        for(j = 1; j <= n; j++) U(i, j) = U(i, j) * d[j-1];
    alpha = 1.0;
    beta = 0.0;
    nag_dgemm(order, Nag_NoTrans, Nag_NoTrans, n, n, n, alpha, u, pdu,
              vt, pdvt, beta, b, pdb, &fail);
    if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_dgemm (f16yac).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
    }

    /* Subtract original bidiagonal matrix:
     * this should give a matrix close to zero.
     */
    for(i=1; i<=n; i++) B(i,i) -= ab[2*i-2+abi];
    if (uplo==Nag_Upper)
        for(i=1; i<=n-1; i++) B(i,i+1) -= ab[2*i-1+abi];
    else
        for(i=1; i<=n-1; i++) B(i+1,i) -= ab[2*i-1+abi];

    /* nag_dge_norm (f16rac): Find norm of matrix B and print warning if
     * it is too large.
     */
    nag_dge_norm(order, Nag_OneNorm, n, n, b, pdb, &norm, &fail);
    if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_dge_norm (f16rac).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
    }

    /* Get the machine precision, using nag_machine_precision (x02ajc) */
    eps = nag_machine_precision;
    if (norm > pow(eps,0.8))
    {
        printf("Norm of B-(U*S*V^T) is much greater than 0.\nSchur "
              "factorization has failed.\n norm = %13.4e\n", norm);
    }
}
}
END:
NAG_FREE(ab);
NAG_FREE(b);
NAG_FREE(d);
NAG_FREE(e);
NAG_FREE(q);
NAG_FREE(u);
NAG_FREE(vt);
NAG_FREE(iq);

return exit_status;
}

```

## 10.2 Program Data

nag\_dbdsdc (f08mdc) Example Program Data

```
4 : n
Nag_Upper : uplo
Nag_SingularVecs : compq
3.62 -2.41 1.92 -1.43 : diagonal elements
1.26 -1.53 1.19 : off-diagonal elements
```

## 10.3 Program Results

nag\_dbdsdc (f08mdc) Example Program Results

```
Matrix B
      1      2      3      4
1      3.6200      1.2600
2      -2.4100     -1.5300
3      1.9200      1.1900
4      -1.4300
Singular values
 4.0001  3.0006  1.9960  0.9998
```

---