NAG Library Function Document

nag_dgejsv (f08khc)

1 Purpose

nag_dgejsv (f08khc) computes the singular value decomposition (SVD) of a real m by n matrix A where $m \geq n$, and optionally computes the left and/or right singular vectors. **nag_dgejsv (f08khc)** implements the preconditioned Jacobi SVD of Drmac and Veselic. This is the expert driver function that calls **nag_dgesvj (f08kjc)** after certain preconditioning. In most cases **nag_dgesvd (f08kbc)** or **nag_dgesdd (f08kdc)** is sufficient to obtain the SVD of a real matrix. These are much simpler to use and also handle the case $m < n$.

2 Specification

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

```
void nag_dgejsv (Nag_OrderType order, Nag_Preprocess joba,
     Nag_LeftVecsType jobu, Nag_RightVecsType jobv, Nag_ZeroCols jobr,
     Nag_TransType jobt, Nag_Perturb jobp, Integer m, Integer n, double a[],
     Integer pda, double sva[], double u[], Integer pdu, double v[],
     Integer pdv, double work[], Integer iwork[], 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 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 in descending order of magnitude. The columns of U and V are the left and the right singular vectors of A. The diagonal of Σ is computed and stored in the array **sva**.

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

Drmac Z and Veselic K (2008a) New fast and accurate Jacobi SVD algorithm I *SIAM J. Matrix Anal. Appl.* **29 4**

Drmac Z and Veselic K (2008b) New fast and accurate Jacobi SVD algorithm II *SIAM J. Matrix Anal. Appl.* **29 4**

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 3.3.1.3 in How to Use the NAG Library and its Documentation for a more detailed explanation of the use of this argument.

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

2: **joba** – Nag_Preprocess *Input*

On entry: specifies the form of pivoting for the QR factorization stage; whether an estimate of the condition number of the scaled matrix is required; and the form of rank reduction that is performed.

joba = Nag_ColpivRrank
The initial QR factorization of the input matrix is performed with column pivoting; no estimate of condition number is computed; and, the rank is reduced by only the underflowed part of the triangular factor R. This option works well (high relative accuracy) if $A = BD$, with well-conditioned B and arbitrary diagonal matrix D. The accuracy cannot be spoiled by column scaling. The accuracy of the computed output depends on the condition of B, and the procedure aims at the best theoretical accuracy.

joba = Nag_ColpivRrankCond
Computation as with **joba** = Nag_ColpivRrank with an additional estimate of the condition number of B. It provides a realistic error bound.

joba = Nag_FullpivRrank
The initial QR factorization of the input matrix is performed with full row and column pivoting; no estimate of condition number is computed; and, the rank is reduced by only the underflowed part of the triangular factor R. If $A = D_1 \times C \times D_2$ with ill-conditioned diagonal scalings D_1, D_2, and well-conditioned matrix C, this option gives higher accuracy than the **joba** = Nag_ColpivRrank option. If the structure of the input matrix is not known, and relative accuracy is desirable, then this option is advisable.

joba = Nag_FullpivRrankCond
Computation as with **joba** = Nag_FullpivRrank with an additional estimate of the condition number of B, where $A = DB$ (i.e., $B = C \times D_2$). If A has heavily weighted rows, then using this condition number gives too pessimistic an error bound.

joba = Nag_ColpivSVrankAbs
Computation as with **joba** = Nag_ColpivRrank except in the treatment of rank reduction. In this case, small singular values are to be considered as noise and, if found, the matrix is treated as numerically rank deficient. The computed SVD $A = U\Sigma V^{\mathrm{T}}$ restores A up to $f(m,n) \times \epsilon \times \|A\|$, where ϵ is ***machine precision***. This gives the procedure licence to discard (set to zero) all singular values below $\mathbf{n} \times \epsilon \times \|A\|$.

joba = Nag_ColpivSVrankRel
Similar to **joba** = Nag_ColpivSVrankAbs. The rank revealing property of the initial QR factorization is used to reveal (using the upper triangular factor) a gap $\sigma_{r+1} < \epsilon\sigma_r$ in which case the numerical rank is declared to be r. The SVD is computed with absolute error bounds, but more accurately than with **joba** = Nag_ColpivSVrankAbs.

Constraint: **joba** = Nag_ColpivRrank, Nag_ColpivRrankCond, Nag_FullpivRrank, Nag_FullpivRrankCond, Nag_ColpivSVrankAbs or Nag_ColpivSVrankRel.

3: **jobu** – Nag_LeftVecsType *Input*

On entry: specifies options for computing the left singular vectors U.

jobu = Nag_LeftSpan
The first n left singular vectors (columns of U) are computed and returned in the array **u**.

jobu = Nag_LeftVecs
All m left singular vectors are computed and returned in the array **u**.

jobu = Nag_NotLeftWork
No left singular vectors are computed, but the array **u** (with **pdu** $\geq$ **m** and second dimension at least **n**) is available as workspace for computing right singular values. See the description of **u**.

jobu = Nag_NotLeftVecs
No left singular vectors are computed. **u** is not referenced when **jobv** = Nag_NotRightVecs or Nag_NotRightWork.

Constraint: **jobu** = Nag_LeftSpan, Nag_LeftVecs, Nag_NotLeftWork or Nag_NotLeftVecs.

4: **jobv** – Nag_RightVecsType *Input*

On entry: specifies options for computing the right singular vectors V.

jobv = Nag_RightVecs
the n right singular vectors (columns of V) are computed and returned in the array **v**; Jacobi rotations are not explicitly accumulated.

jobv = Nag_RightVecsJRots
the n right singular vectors (columns of V) are computed and returned in the array **v**, but they are computed as the product of Jacobi rotations. This option is allowed only if **jobu** = Nag_LeftSpan or Nag_LeftVecs, i.e., in computing the full SVD.

This is equivalent to multiplying the input matrix, on the right, by the matrix V.

jobv = Nag_NotRightWork
No right singular values are computed, but the array **v** (with **pdv** $\geq$ **n** and second dimension at least **n**) is available as workspace for computing left singular values. See the description of **v**.

jobv = Nag_NotRightVecs
No right singular vectors are computed. **v** is not referenced when **jobu** = Nag_NotLeftVecs or Nag_NotLeftWork or **jobt** = Nag_NoTrans or **m** $\neq$ **n**.

Constraints:

jobv = Nag_RightVecs, Nag_RightVecsJRots, Nag_NotRightWork or Nag_NotRightVecs;
if **jobu** = Nag_NotLeftWork or Nag_NotLeftVecs, **jobv** $\neq$ Nag_RightVecsJRots.

5: **jobr** – Nag_ZeroCols *Input*

On entry: specifies the conditions under which columns of A are to be set to zero. This effectively specifies a lower limit on the range of singular values; any singular values below this limit are (through column zeroing) set to zero. If $A \neq 0$ is scaled so that the largest column (in the Euclidean norm) of cA is equal to the square root of the overflow threshold, then **jobr** allows the function to kill columns of A whose norm in cA is less than $\sqrt{sfmin}$ (for **jobr** = Nag_ZeroColsRestrict), or less than $sfmin/\epsilon$ (otherwise). $sfmin$ is the safe range argument, as returned by function **nag_real_safe_small_number (X02AMC)**.

jobr = Nag_ZeroColsNormal
Only set to zero those columns of A for which the norm of corresponding column of $cA < sfmin/\epsilon$, that is, those columns that are effectively zero (to ***machine precision***) anyway. If the condition number of A is greater than the overflow threshold λ, where λ is the value returned by **nag_real_largest_number (X02ALC)**, you are recommended to use function **nag_dgesvj (f08kjc)**.

jobr = Nag_ZeroColsRestrict
Set to zero those columns of A for which the norm of the corresponding column of $cA < \sqrt{sfmin}$. This approximately represents a restricted range for $\sigma(cA)$ of $\left[\sqrt{sfmin}, \sqrt{\lambda}\right]$.

For computing the singular values in the full range from the safe minimum up to the overflow threshold use **nag_dgesvj (f08kjc)**.

Suggested value: **jobr** = Nag_ZeroColsRestrict.

Constraint: **jobr** = Nag_ZeroColsNormal or Nag_ZeroColsRestrict.

6: **jobt** – Nag_TransType *Input*

On entry: specifies, in the case $n = m$, whether the function is permitted to use the transpose of A for improved efficiency. If the matrix is square then the procedure may use transposed A if A^{T} seems to be better with respect to convergence. If the matrix is not square, **jobt** is ignored. The decision is based on two values of entropy over the adjoint orbit of $A^{\mathrm{T}}A$. See the descriptions of **work**[5] and **work**[6].

jobt = Nag_Trans
: If $n = m$, perform an entropy test and then transpose if the test indicates possibly faster convergence of the Jacobi process if A^{T} is taken as input. If A is replaced with A^{T}, then the row pivoting is included automatically.

jobt = Nag_NoTrans
: No entropy test and no transposition is performed.

The option **jobt** = Nag_Trans can be used to compute only the singular values, or the full SVD (U, Σ and V). In the case where only one set of singular vectors (U or V) is required, the caller must still provide both **u** and **v**, as one of the matrices is used as workspace if the matrix A is transposed. See the descriptions of **u** and **v**.

Constraint: **jobt** = Nag_Trans or Nag_NoTrans.

7: **jobp** – Nag_Perturb *Input*

On entry: specifies whether the function should be allowed to introduce structured perturbations to drown denormalized numbers. For details see Drmac and Veselic (2008a) and Drmac and Veselic (2008b). For the sake of simplicity, these perturbations are included only when the full SVD or only the singular values are requested.

jobp = Nag_PerturbOn
: Introduce perturbation if A is found to be very badly scaled (introducing denormalized numbers).

jobp = Nag_PerturbOff
: Do not perturb.

Constraint: **jobp** = Nag_PerturbOn or Nag_PerturbOff.

8: **m** – Integer *Input*

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

Constraint: **m** ≥ 0.

9: **n** – Integer *Input*

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

Constraint: **m** $\geq$ **n** ≥ 0.

10: **a**[*dim*] – double *Input/Output*

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

max(1, **pda** × **n**) when **order** = Nag_ColMajor;
max(1, **m** × **pda**) when **order** = Nag_RowMajor.

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

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

On entry: the m by n matrix A.

On exit: the contents of **a** are overwritten.

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

12: **sva**[**n**] – double *Output*

On exit: the, possibly scaled, singular values of A.

The singular values of A are $\sigma_i = \alpha\mathbf{sva}[i-1]$, for $i = 1, 2, \ldots, n$, where $\alpha = \mathbf{work}[0]/\mathbf{work}[1]$. Normally $\alpha = 1$ and no scaling is required to obtain the singular values. However, if the largest singular value of A overflows or if small singular values have been saved from underflow by scaling the input matrix A, then $\alpha \neq 1$.

If **jobr** = Nag_ZeroColsRestrict then some of the singular values may be returned as exact zeros because they are below the numerical rank threshold or are denormalized numbers.

13: **u**[*dim*] – double *Output*

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

$\max(1, \mathbf{pdu} \times \mathbf{m})$ when **jobu** = Nag_LeftVecs;
$\max(1, \mathbf{pdu} \times \mathbf{n})$ when **jobu** = Nag_LeftSpan or Nag_NotLeftWork and **order** = Nag_ColMajor;
$\max(1, \mathbf{m} \times \mathbf{pdu})$ when **jobu** = Nag_LeftSpan or Nag_NotLeftWork and **order** = Nag_RowMajor;
$\max(1, \mathbf{m})$ otherwise.

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

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

On exit: if **jobu** = Nag_LeftSpan, **u** contains the m by n matrix of the left singular vectors.

If **jobu** = Nag_LeftVecs, **u** contains the m by m matrix of the left singular vectors, including an orthonormal basis of the orthogonal complement of Range(A).

u is not referenced when **jobu** = Nag_NotLeftWork or Nag_NotLeftVecs and one of the following is satisfied:

jobv = Nag_NotRightWork or Nag_NotRightVecs, or

n = 1, or

A is the zero matrix.

14: **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 **jobu** = Nag_LeftVecs, **pdu** $\geq \max(1, \mathbf{m})$;
if **jobu** = Nag_LeftSpan or Nag_NotLeftWork, **pdu** $\geq \max(1, \mathbf{m})$;
otherwise **pdu** ≥ 1.;

if **order** = Nag_RowMajor,

if **jobu** = Nag_LeftVecs, **pdu** $\geq \max(1, \mathbf{m})$;
if **jobu** = Nag_LeftSpan or Nag_NotLeftWork, **pdu** $\geq \max(1, \mathbf{n})$;
otherwise **pdu** ≥ 1..

15: **v**[*dim*] – double *Output*

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

max$(1, \mathbf{pdv} \times \mathbf{n})$ when **jobv** = Nag_RightVecs, Nag_RightVecsJRots or Nag_NotRightWork;
1 otherwise.

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

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

On exit: if **jobv** = Nag_RightVecs or Nag_RightVecsJRots, **v** contains the n by n matrix of the right singular vectors.

v is not referenced when **jobv** = Nag_NotRightWork or Nag_NotRightVecs and one of the following is satisfied:

jobu = Nag_LeftSpan or Nag_LeftVecs and **jobt** = Nag_Trans, or

$\mathbf{n} = 1$, or

A is the zero matrix.

16: **pdv** – Integer *Input*

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

Constraints:

if **jobv** = Nag_RightVecs, Nag_RightVecsJRots or Nag_NotRightWork, $\mathbf{pdv} \geq \max(1, \mathbf{n})$;
otherwise $\mathbf{pdv} \geq 1$.

17: **work**[7] – double *Output*

On exit: contains information about the completed job.

work[0]
$\alpha = \mathbf{work}[0]/\mathbf{work}[1]$ is the scaling factor such that $\sigma_i = \alpha\mathbf{sva}[i-1]$, for $i = 1, 2, \ldots, n$ are the computed singular values of A. (See the description of **sva**.)

work[1]
See the description of **work**[0].

work[2]
sconda, an estimate for the condition number of column equilibrated A (if **joba** = Nag_ColpivRrankCond or Nag_FullpivRrankCond). *sconda* is an estimate of $\sqrt{\left(\left\|(R^{\mathrm{T}}R)^{-1}\right\|_1\right)}$. It is computed using **nag_dpocon (f07fgc)**. It satisfies $n^{-\frac{1}{4}} \times sconda \leq \left\|R^{-1}\right\|_2 \leq n^{\frac{1}{4}} \times sconda$ where R is the triangular factor from the QR factorization of A. However, if R is truncated and the numerical rank is determined to be strictly smaller than n, *sconda* is returned as -1, thus indicating that the smallest singular values might be lost.

If full SVD is needed, and you are familiar with the details of the method, the following two condition numbers are useful for the analysis of the algorithm.

work[3]
An estimate of the scaled condition number of the triangular factor in the first QR factorization.

work[4]
An estimate of the scaled condition number of the triangular factor in the second QR factorization.

The following two parameters are computed if **jobt** = Nag_Trans.

work[5]
The entropy of $A^{T}A$: this is the Shannon entropy of diag $A^{T}A/$ trace $A^{T}A$ taken as a point in the probability simplex.

work[6]
The entropy of AA^{T}.

18: **iwork**[**3**] – Integer *Output*

On exit: contains information about the completed job.

iwork[0]
The numerical rank of A determined after the initial QR factorization with pivoting. See the descriptions of **joba** and **jobr**.

iwork[1]
The number of computed nonzero singular values.

iwork[2]
If nonzero, a warning message: If **iwork**[2] = 1 then some of the column norms of A were denormalized (tiny) numbers. The requested high accuracy is not warranted by the data.

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

The NAG error argument (see Section 3.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

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

NE_CONSTRAINT

On entry, **jobv** = ⟨*value*⟩ and **jobu** = ⟨*value*⟩.
Constraint: **jobv** = Nag_RightVecs, Nag_RightVecsJRots, Nag_NotRightWork or Nag_NotRightVecs and
if **jobu** = Nag_NotLeftWork or Nag_NotLeftVecs, **jobv** ≠ Nag_RightVecsJRots.

NE_CONVERGENCE

nag_dgejsv (f08khc) did not converge in the allowed number of iterations (30). The computed values might be inaccurate.

NE_ENUM_INT_2

On entry, **jobu** = ⟨*value*⟩, **m** = ⟨*value*⟩ and **pdu** = ⟨*value*⟩.
Constraint: if **jobu** = Nag_LeftVecs, **pdu** $\geq \max(1, \mathbf{m})$;
if **jobu** = Nag_LeftSpan or Nag_NotLeftWork, **pdu** $\geq \max(1, \mathbf{m})$;
otherwise **pdu** ≥ 1.

On entry, **jobv** = ⟨*value*⟩, **pdv** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: if **jobv** = Nag_RightVecs, Nag_RightVecsJRots or Nag_NotRightWork, **pdv** $\geq \max(1, \mathbf{n})$;
otherwise **pdv** ≥ 1.

NE_ENUM_INT_3

On entry, **jobu** = $\langle value \rangle$, **pdu** = $\langle value \rangle$, **m** = $\langle value \rangle$ and **n** = $\langle value \rangle$.
Constraint: if **jobu** = Nag_LeftVecs, **pdu** $\geq \max(1, \mathbf{m})$;
if **jobu** = Nag_LeftSpan or Nag_NotLeftWork, **pdu** $\geq \max(1, \mathbf{n})$;
otherwise **pdu** ≥ 1.

NE_INT

On entry, **m** = $\langle value \rangle$.
Constraint: **m** ≥ 0.

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

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

On entry, **pdv** = $\langle value \rangle$.
Constraint: **pdv** > 0.

NE_INT_2

On entry, **m** = $\langle value \rangle$ and **n** = $\langle value \rangle$.
Constraint: **m** $\geq$ **n** ≥ 0.

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.
See Section 2.7.6 in How to Use the NAG Library and its Documentation for further information.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 2.7.5 in How to Use the NAG Library and its Documentation 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_dgejsv (f08khc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_dgejsv (f08khc) 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

nag_dgejsv (f08khc) implements a preconditioned Jacobi SVD algorithm. It uses **nag_dgeqrf (f08aec)**, **nag_dgelqf (f08ahc)** and **nag_dgeqp3 (f08bfc)** as preprocessors and preconditioners. Optionally, an additional row pivoting can be used as a preprocessor, which in some cases results in much higher accuracy. An example is matrix A with the structure $A = D_1CD_2$, where D_1, D_2 are arbitrarily ill-conditioned diagonal matrices and C is a well-conditioned matrix. In that case, complete pivoting in the first QR factorizations provides accuracy dependent on the condition number of C, and independent of D_1, D_2. Such higher accuracy is not completely understood theoretically, but it works well in practice. Further, if A can be written as $A = BD$, with well-conditioned B and some diagonal D, then the high accuracy is guaranteed, both theoretically and in software, independent of D.

10 Example

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

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

together with the condition number of A and approximate error bounds for the computed singular values and vectors.

10.1 Program Text

```
/* nag_dgejsv (f08khc) Example Program.
 *
 * Copyright 2017 Numerical Algorithms Group.
 *
 * Mark 26.1, 2017.
 */

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

int main(void)
{
  /* Scalars */
  double eps, serrbd;
  Integer exit_status = 0;
  Integer pda, pdu, pdv;
  Integer i, j, m, n, n_uvecs, n_vvecs;
  /* Arrays */
  double *a = 0, *rcondu = 0, *rcondv = 0, *s = 0, *u = 0, *v = 0;
  double work[7];
  Integer iwork[3];
  char nag_enum_arg[40];

  /* Nag Types */
  Nag_OrderType order;
  Nag_Preprocess joba;
  Nag_LeftVecsType jobu;
  Nag_RightVecsType jobv;
  Nag_ZeroCols jobr;
  Nag_TransType jobt;
  Nag_Perturb jobp;
  NagError fail;
```

```
#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_dgejsv (f08khc) 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 (n < 0 || m < n) {
    printf("Invalid n or nrhs\n");
    exit_status = 1;
    goto END;;
  }

  /* Read Nag type arguments by name and convert to value */
#ifdef _WIN32
  scanf_s(" %39s%*[^\n]", nag_enum_arg, (unsigned)_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
   */
  joba = (Nag_Preprocess) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
  scanf_s(" %39s%*[^\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
  scanf(" %39s%*[^\n]", nag_enum_arg);
#endif
  jobu = (Nag_LeftVecsType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
  scanf_s(" %39s%*[^\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
  scanf(" %39s%*[^\n]", nag_enum_arg);
#endif
  jobv = (Nag_RightVecsType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
  scanf_s(" %39s%*[^\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
  scanf(" %39s%*[^\n]", nag_enum_arg);
#endif
  jobr = (Nag_ZeroCols) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
  scanf_s(" %39s%*[^\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
  scanf(" %39s%*[^\n]", nag_enum_arg);
#endif
  jobt = (Nag_TransType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
  scanf_s(" %39s%*[^\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
  scanf(" %39s%*[^\n]", nag_enum_arg);
#endif
  jobp = (Nag_Perturb) nag_enum_name_to_value(nag_enum_arg);
```

```
  /* Size of u and v depends on some of the above Nag type arguments. */
  n_uvecs = 1;
  if (jobu == Nag_LeftVecs) {
    n_uvecs = m;
  }
  else if (jobu == Nag_LeftSpan) {
    n_uvecs = n;
  }
  else if (jobu == Nag_NotLeftWork && jobv == Nag_RightVecs &&
           jobt == Nag_Trans && m == n) {
    n_uvecs = m;
  }
  if (jobv == Nag_NotRightVecs) {
    n_vvecs = 1;
  }
  else {
    n_vvecs = n;
  }
#ifdef NAG_COLUMN_MAJOR
  pda = m;
  pdu = m;
  pdv = n;
#else
  pda = n;
  pdu = n_uvecs;
  pdv = n_vvecs;
#endif

  if (!(a = NAG_ALLOC(m * n, double)) ||
      !(rcondu = NAG_ALLOC(m, double)) ||
      !(rcondv = NAG_ALLOC(m, double)) ||
      !(s = NAG_ALLOC(n, double)) ||
      !(u = NAG_ALLOC(m * n_uvecs, double)) ||
      !(v = NAG_ALLOC(n_vvecs * n_vvecs, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

  /* 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_dgejsv (f08khc)
   * Compute the singular values and left and right singular vectors
   * of A (A = U*S*V^T, m>=n).
   */
  nag_dgejsv(order, joba, jobu, jobv, jobr, jobt, jobp, m, n, a, pda, s, u,
             pdu, v, pdv, work, iwork, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dgejsv (f08khc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Get the machine precision, eps and compute the approximate
   * error bound for the computed singular values. Note that for
   * the 2-norm, s[0] = norm(A).
   */
```

```
  eps = nag_machine_precision;
  serrbd = eps * s[0];

  /* Print (possibly scaled) singular values. */
  if (fabs(work[0] - work[1]) < 2.0 * eps) {
    /* No scaling required */
    printf("Singular values\n");
    for (j = 0; j < n; j++)
      printf("%8.4f", s[j]);
  }
  else {
    printf("Scaled singular values\n");
    for (j = 0; j < n; j++)
      printf("%8.4f", s[j]);
    printf("\nFor true singular values, multiply by a/b,\n");
    printf("where a = %f and b = %f", work[0], work[1]);
  }
  printf("\n\n");

  /* Print left and right (spanning) singular vectors, if requested.  using
   * nag_gen_real_mat_print (x04cac)
   * Print real general matrix (easy-to-use)
   */
  if (jobu == Nag_LeftVecs || jobu == Nag_LeftSpan) {
    fflush(stdout);
    nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, m, n, 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;
    }
  }
  if (jobv == Nag_RightVecs || jobv == Nag_RightVecsJRots) {
    printf("\n");
    fflush(stdout);
    nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n, v,
                           pdv, "Right 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;
    }
  }

  /* nag_ddisna (f08flc)
   * Estimate reciprocal condition numbers for the singular vectors.
   */
  nag_ddisna(Nag_LeftSingVecs, m, n, s, rcondu, &fail);
  if (fail.code == NE_NOERROR)
    nag_ddisna(Nag_RightSingVecs, m, n, s, rcondv, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ddisna (f08flc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  if (joba == Nag_ColpivRrankCond || joba == Nag_FullpivRrankCond) {
    printf("\n\nEstimate of the condition number of column equilibrated A\n");
    printf("%11.1e", work[2]);
  }

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

  printf("\n\nError estimates for left singular vectors\n");
  for (i = 0; i < n; i++)
    printf("%11.1e", serrbd / rcondu[i]);
```

```
  printf("\n\nError estimates for right singular vectors\n");
  for (i = 0; i < n; i++)
    printf("%11.1e", serrbd / rcondv[i]);
  printf("\n");

END:
  NAG_FREE(a);
  NAG_FREE(rcondu);
  NAG_FREE(rcondv);
  NAG_FREE(s);
  NAG_FREE(u);
  NAG_FREE(v);

  return exit_status;
}
```

10.2 Program Data

```
nag_dgejsv (f08khc) Example Program Data

   6      4                   : m and n

   Nag_ColpivRrankCond        : joba
   Nag_LeftSpan               : jobu
   Nag_RightVecs              : jobv
   Nag_ZeroColsRestrict       : jobr
   Nag_NoTrans                : jobt
   Nag_PerturbOff             : jobp

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

10.3 Program Results

```
nag_dgejsv (f08khc) Example Program Results

Singular values
  9.9966  3.6831  1.3569  0.5000

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

 Right 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


Estimate of the condition number of column equilibrated A
    9.0e+00

Error estimate for the singular values
    1.1e-15
```

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

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