NAG Library Function Document

nag_zuncsd (f08rnc)

1 Purpose

nag_zuncsd (f08rnc) computes the CS decomposition of a complex m by m unitary matrix X, partitioned into a 2 by 2 array of submatrices.

2 Specification

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

```
void nag_zuncsd (Nag_OrderType order, Nag_ComputeUType jobu1,
     Nag_ComputeUType jobu2, Nag_ComputeVTType jobv1t,
     Nag_ComputeVTType jobv2t, Nag_SignsType signs, Integer m, Integer p,
     Integer q, Complex x11[], Integer pdx11, Complex x12[], Integer pdx12,
     Complex x21[], Integer pdx21, Complex x22[], Integer pdx22,
     double theta[], Complex u1[], Integer pdu1, Complex u2[], Integer pdu2,
     Complex v1t[], Integer pdv1t, Complex v2t[], Integer pdv2t,
     NagError *fail)
```

3 Description

The m by m unitary matrix X is partitioned as

$$X = \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix}$$

where X_{11} is a p by q submatrix and the dimensions of the other submatrices X_{12}, X_{21} and X_{22} are such that X remains m by m.

The CS decomposition of X is $X = U\Sigma_p V^{\mathrm{T}}$ where U, V and Σ_p are m by m matrices, such that

$$U = \begin{pmatrix} U_1 & \mathbf{0} \\ \mathbf{0} & U_2 \end{pmatrix}$$

is a unitary matrix containing the p by p unitary matrix U_1 and the $(m-p)$ by $(m-p)$ unitary matrix U_2;

$$V = \begin{pmatrix} V_1 & \mathbf{0} \\ \mathbf{0} & V_2 \end{pmatrix}$$

is a unitary matrix containing the q by q unitary matrix V_1 and the $(m-q)$ by $(m-q)$ unitary matrix V_2; and

$$\Sigma_p = \left(\begin{array}{ccc|ccc} I_{11} & & \mathbf{0} & & \mathbf{0} & \mathbf{0} \\ & C & \mathbf{0} & \mathbf{0} & -S & \\ \mathbf{0} & \mathbf{0} & & \mathbf{0} & & -I_{12} \\ \hline & \mathbf{0} & \mathbf{0} & I_{22} & & \mathbf{0} \\ \mathbf{0} & S & & & C & \mathbf{0} \\ \mathbf{0} & & I_{21} & \mathbf{0} & \mathbf{0} & \end{array}\right)$$

contains the r by r non-negative diagonal submatrices C and S satisfying $C^2 + S^2 = I$, where $r = \min(p, m-p, q, m-q)$ and the top left partition is p by q.

The identity matrix I_{11} is of order $\min(p,q) - r$ and vanishes if $\min(p,q) = r$.

The identity matrix I_{12} is of order $\min(p, m-q) - r$ and vanishes if $\min(p, m-q) = r$.

The identity matrix I_{21} is of order $\min(m-p, q) - r$ and vanishes if $\min(m-p, q) = r$.

The identity matrix I_{22} is of order $\min(m-p, m-q) - r$ and vanishes if $\min(m-p, m-q) = r$.

In each of the four cases $r = p, q, m-p, m-q$ at least two of the identity matrices vanish.

The indicated zeros represent augmentations by additional rows or columns (but not both) to the square diagonal matrices formed by I_{ij} and C or S.

Σ_p does not need to be stored in full; it is sufficient to return only the values θ_i for $i = 1, 2, \ldots, r$ where $C_{ii} = \cos(\theta_i)$ and $S_{ii} = \sin(\theta_i)$.

The algorithm used to perform the complete CS decomposition is described fully in Sutton (2009) including discussions of the stability and accuracy of the algorithm.

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 (2012) *Matrix Computations* (4th Edition) Johns Hopkins University Press, Baltimore

Sutton B D (2009) Computing the complete *CS* decomposition *Numerical Algorithms (Volume 50)* **1017–1398** Springer US 33–65 http://dx.doi.org/10.1007/s11075-008-9215-6

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 2.3.1.3 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: **jobu1** – Nag_ComputeUType *Input*

On entry:

if **jobu1** = Nag_AllU, U_1 is computed;

if **jobu1** = Nag_NotU, U_1 is not computed.

Constraint: **jobu1** = Nag_AllU or Nag_NotU.

3: **jobu2** – Nag_ComputeUType *Input*

On entry:

if **jobu2** = Nag_AllU, U_2 is computed;

if **jobu2** = Nag_NotU, U_2 is not computed.

Constraint: **jobu2** = Nag_AllU or Nag_NotU.

4: **jobv1t** – Nag_ComputeVTType *Input*

On entry:

if **jobv1t** = Nag_AllVT, V_1^T is computed;

if **jobv1t** = Nag_NotVT, V_1^T is not computed.

Constraint: **jobv1t** = Nag_AllVT or Nag_NotVT.

5: **jobv2t** – Nag_ComputeVTType *Input*

On entry:

if **jobv2t** = Nag_AllVT, V_2^{T} is computed;

if **jobv2t** = Nag_NotVT, V_2^{T} is not computed.

Constraint: **jobv2t** = Nag_AllVT or Nag_NotVT.

6: **signs** – Nag_SignsType *Input*

On entry:

if **signs** = Nag_LowerMinus, the lower-left block is made nonpositive (the other convention);

if **signs** = Nag_UpperMinus, the upper-right block is made nonpositive (the default convention).

Constraint: **signs** = Nag_LowerMinus or Nag_UpperMinus.

7: **m** – Integer *Input*

On entry: m, the number of rows and columns in the unitary matrix X.

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

8: **p** – Integer *Input*

On entry: p, the number of rows in X_{11} and X_{12}.

Constraint: $0 \leq \mathbf{p} \leq \mathbf{m}$.

9: **q** – Integer *Input*

On entry: q, the number of columns in X_{11} and X_{21}.

Constraint: $0 \leq \mathbf{q} \leq \mathbf{m}$.

10: **x11**[*dim*] – Complex *Input/Output*

Note: the dimension, *dim*, of the array **x11** must be at least

$\max(1, \mathbf{pdx11} \times \mathbf{p})$ when **order** = Nag_RowMajor;
$\max(1, \mathbf{pdx11} \times \mathbf{q})$ when **order** = Nag_ColMajor.

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

x11$[(j-1) \times \mathbf{pdx11} + i - 1]$ when **order** = Nag_ColMajor;
x11$[(i-1) \times \mathbf{pdx11} + j - 1]$ when **order** = Nag_RowMajor.

On entry: the upper left partition of the unitary matrix X whose CSD is desired.

On exit: contains details of the unitary matrix used in a simultaneous bidiagonalization process.

11: **pdx11** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **x11**.

Constraints:

if **order** = Nag_RowMajor, $\mathbf{pdx11} \geq \max(1, \mathbf{q})$;
if **order** = Nag_ColMajor, $\mathbf{pdx11} \geq \max(1, \mathbf{p})$.

12: **x12**[*dim*] – Complex *Input/Output*

Note: the dimension, *dim*, of the array **x12** must be at least

max$(1, \mathbf{pdx12} \times \mathbf{p})$ when **order** = Nag_RowMajor;
max$(1, \mathbf{pdx12} \times (\mathbf{m} - \mathbf{q}))$ when **order** = Nag_ColMajor.

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

x12$[(j - 1) \times \mathbf{pdx12} + i - 1]$ when **order** = Nag_ColMajor;
x12$[(i - 1) \times \mathbf{pdx12} + j - 1]$ when **order** = Nag_RowMajor.

On entry: the upper right partition of the unitary matrix X whose CSD is desired.

On exit: contains details of the unitary matrix used in a simultaneous bidiagonalization process.

13: **pdx12** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **x12**.

Constraints:

if **order** = Nag_RowMajor, $\mathbf{pdx12} \geq \max(1, \mathbf{m} - \mathbf{q})$;
if **order** = Nag_ColMajor, $\mathbf{pdx12} \geq \max(1, \mathbf{p})$.

14: **x21**[*dim*] – Complex *Input/Output*

Note: the dimension, *dim*, of the array **x21** must be at least

max$(1, \mathbf{pdx21} \times (\mathbf{m} - \mathbf{p}))$ when **order** = Nag_RowMajor;
max$(1, \mathbf{pdx21} \times \mathbf{q})$ when **order** = Nag_ColMajor.

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

x21$[(j - 1) \times \mathbf{pdx21} + i - 1]$ when **order** = Nag_ColMajor;
x21$[(i - 1) \times \mathbf{pdx21} + j - 1]$ when **order** = Nag_RowMajor.

On entry: the lower left partition of the unitary matrix X whose CSD is desired.

On exit: contains details of the unitary matrix used in a simultaneous bidiagonalization process.

15: **pdx21** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **x21**.

Constraints:

if **order** = Nag_RowMajor, $\mathbf{pdx21} \geq \max(1, \mathbf{q})$;
if **order** = Nag_ColMajor, $\mathbf{pdx21} \geq \max(1, \mathbf{m} - \mathbf{p})$.

16: **x22**[*dim*] – Complex *Input/Output*

Note: the dimension, *dim*, of the array **x22** must be at least

max$(1, \mathbf{pdx22} \times (\mathbf{m} - \mathbf{p}))$ when **order** = Nag_RowMajor;
max$(1, \mathbf{pdx22} \times (\mathbf{m} - \mathbf{q}))$ when **order** = Nag_ColMajor.

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

x22$[(j - 1) \times \mathbf{pdx22} + i - 1]$ when **order** = Nag_ColMajor;
x22$[(i - 1) \times \mathbf{pdx22} + j - 1]$ when **order** = Nag_RowMajor.

On entry: the lower right partition of the unitary matrix X CSD is desired.

On exit: contains details of the unitary matrix used in a simultaneous bidiagonalization process.

17: **pdx22** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **x22**.

Constraints:

if **order** = Nag_RowMajor, **pdx22** $\geq \max(1, \mathbf{m} - \mathbf{q})$;
if **order** = Nag_ColMajor, **pdx22** $\geq \max(1, \mathbf{m} - \mathbf{p})$.

18: **theta**[*dim*] – double *Output*

Note: the dimension, *dim*, of the array **theta** must be at least $\min(\mathbf{p}, \mathbf{m} - \mathbf{p}, \mathbf{q}, \mathbf{m} - \mathbf{q})$.

On exit: the values θ_i for $i = 1, 2, \dots, r$ where $r = \min(p, m - p, q, m - q)$. The diagonal submatrices C and S of Σ_p are constructed from these values as

$C = \mathrm{diag}(\cos(\mathbf{theta}[0]), \dots, \cos(\mathbf{theta}[r-1]))$ and

$S = \mathrm{diag}(\sin(\mathbf{theta}[0]), \dots, \sin(\mathbf{theta}[r-1]))$.

19: **u1**[*dim*] – Complex *Output*

Note: the dimension, *dim*, of the array **u1** must be at least

$\max(1, \mathbf{pdu1} \times \mathbf{p})$ when **jobu1** = Nag_AllU;
otherwise **u1** may be **NULL**.

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

u1$[(j - 1) \times \mathbf{pdu1} + i - 1]$ when **order** = Nag_ColMajor;
u1$[(i - 1) \times \mathbf{pdu1} + j - 1]$ when **order** = Nag_RowMajor.

On exit: if **jobu1** = Nag_AllU, **u1** contains the p by p unitary matrix U_1.

20: **pdu1** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **u1**.

Constraint: if **jobu1** = Nag_AllU, **pdu1** $\geq \max(1, \mathbf{p})$

21: **u2**[*dim*] – Complex *Output*

Note: the dimension, *dim*, of the array **u2** must be at least

$\max(1, \mathbf{pdu2} \times (\mathbf{m} - \mathbf{p}))$ when **jobu2** = Nag_AllU;
otherwise **u2** may be **NULL**.

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

u2$[(j - 1) \times \mathbf{pdu2} + i - 1]$ when **order** = Nag_ColMajor;
u2$[(i - 1) \times \mathbf{pdu2} + j - 1]$ when **order** = Nag_RowMajor.

On exit: if **jobu2** = Nag_AllU, **u2** contains the $m - p$ by $m - p$ unitary matrix U_2.

22: **pdu2** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **u2**.

Constraint: if **jobu2** = Nag_AllU, **pdu2** $\geq \max(1, \mathbf{m} - \mathbf{p})$

23: **v1t**[*dim*] – Complex *Output*

Note: the dimension, *dim*, of the array **v1t** must be at least

$\max(1, \mathbf{pdv1t} \times \mathbf{q})$ when **jobv1t** = Nag_AllVT;
otherwise **v1t** may be **NULL**.

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

> $\mathbf{v1t}[(j-1)\times\mathbf{pdv1t}+i-1]$ when **order** = Nag_ColMajor;
> $\mathbf{v1t}[(i-1)\times\mathbf{pdv1t}+j-1]$ when **order** = Nag_RowMajor.

On exit: if **jobv1t** = Nag_AllVT, **v1t** contains the q by q unitary matrix ${V_1}^{\mathrm{H}}$.

24: **pdv1t** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **v1t**.

Constraint: if **jobv1t** = Nag_AllVT, $\mathbf{pdv1t}\geq\max(1,\mathbf{q})$

25: **v2t**[*dim*] – Complex *Output*

Note: the dimension, *dim*, of the array **v2t** must be at least

> $\max(1,\mathbf{pdv2t}\times(\mathbf{m}-\mathbf{q}))$ when **jobv2t** = Nag_AllVT;
> otherwise **v2t** may be **NULL**.

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

> $\mathbf{v2t}[(j-1)\times\mathbf{pdv2t}+i-1]$ when **order** = Nag_ColMajor;
> $\mathbf{v2t}[(i-1)\times\mathbf{pdv2t}+j-1]$ when **order** = Nag_RowMajor.

On exit: if **jobv2t** = Nag_AllVT, **v2t** contains the $m-q$ by $m-q$ unitary matrix ${V_2}^{\mathrm{H}}$.

26: **pdv2t** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **v2t**.

Constraint: if **jobv2t** = Nag_AllVT, $\mathbf{pdv2t}\geq\max(1,\mathbf{m}-\mathbf{q})$

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

The NAG error argument (see Section 2.7 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_CONVERGENCE

The Jacobi-type procedure failed to converge during an internal reduction to bidiagonal-block form. The process requires convergence to $\min(\mathbf{p},\mathbf{m}-\mathbf{p},\mathbf{q},\mathbf{m}-\mathbf{q})$ values, the value of **fail.errnum** gives the number of converged values.

NE_ENUM_INT_2

On entry, **jobu1** = ⟨*value*⟩, **pdu1** = ⟨*value*⟩ and **p** = ⟨*value*⟩.
Constraint: if **jobu1** = Nag_AllU, $\mathbf{pdu1}\geq\max(1,\mathbf{p})$.

On entry, **jobu1** = ⟨*value*⟩, **pdu1** = ⟨*value*⟩, **p** = ⟨*value*⟩.
Constraint: if **jobu1** = Nag_AllU, $\mathbf{pdu1}\geq\mathbf{p}$.

On entry, **jobv1t** $= \langle value \rangle$, **pdv1t** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **jobv1t** = Nag_AllVT, **pdv1t** $\geq \max(1, \mathbf{q})$.

On entry, **jobv1t** $= \langle value \rangle$, **pdv1t** $= \langle value \rangle$, **q** $= \langle value \rangle$.
Constraint: if **jobv1t** = Nag_AllVT, **pdv1t** $\geq$ **q**.

NE_ENUM_INT_3

On entry, **jobu2** $= \langle value \rangle$, **pdu2** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **p** $= \langle value \rangle$.
Constraint: if **jobu2** = Nag_AllU, **pdu2** $\geq \max(1, \mathbf{m} - \mathbf{p})$.

On entry, **jobu2** $= \langle value \rangle$, **pdu2** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **p** $= \langle value \rangle$.
Constraint: if **jobu2** = Nag_AllU, **pdu2** $\geq \mathbf{m} - \mathbf{p}$.

On entry, **jobv2t** $= \langle value \rangle$, **pdv2t** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **jobv2t** = Nag_AllVT, **pdv2t** $\geq \max(1, \mathbf{m} - \mathbf{q})$.

On entry, **jobv2t** $= \langle value \rangle$, **pdv2t** $= \langle value \rangle$, **m** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **jobv2t** = Nag_AllVT, **pdv2t** $\geq \mathbf{m} - \mathbf{q}$.

On entry, **order** $= \langle value \rangle$, **pdx11** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx11** $\geq \max(1, \mathbf{p})$;
if **order** = Nag_ColMajor, **pdx11** $\geq \max(1, \mathbf{q})$.

On entry, **order** $= \langle value \rangle$, **pdx11** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx11** $\geq \max(1, \mathbf{q})$;
if **order** = Nag_ColMajor, **pdx11** $\geq \max(1, \mathbf{p})$.

NE_ENUM_INT_4

On entry, **order** $= \langle value \rangle$, **pdx12** $= \langle value \rangle$, **m** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx12** $\geq \max(1, \mathbf{m} - \mathbf{q})$;
if **order** = Nag_ColMajor, **pdx12** $\geq \max(1, \mathbf{p})$.

On entry, **order** $= \langle value \rangle$, **pdx12** $= \langle value \rangle$, **m** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx12** $\geq \max(1, \mathbf{p})$;
if **order** = Nag_ColMajor, **pdx12** $\geq \max(1, \mathbf{m} - \mathbf{q})$.

On entry, **order** $= \langle value \rangle$, **pdx21** $= \langle value \rangle$, **m** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx21** $\geq \max(1, \mathbf{m} - \mathbf{p})$;
if **order** = Nag_ColMajor, **pdx21** $\geq \max(1, \mathbf{q})$.

On entry, **order** $= \langle value \rangle$, **pdx21** $= \langle value \rangle$, **m** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx21** $\geq \max(1, \mathbf{q})$;
if **order** = Nag_ColMajor, **pdx21** $\geq \max(1, \mathbf{m} - \mathbf{p})$.

On entry, **order** $= \langle value \rangle$, **pdx22** $= \langle value \rangle$, **m** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx22** $\geq \max(1, \mathbf{m} - \mathbf{p})$;
if **order** = Nag_ColMajor, **pdx22** $\geq \max(1, \mathbf{m} - \mathbf{q})$.

On entry, **order** $= \langle value \rangle$, **pdx22** $= \langle value \rangle$, **m** $= \langle value \rangle$, **p** $= \langle value \rangle$ and **q** $= \langle value \rangle$.
Constraint: if **order** = Nag_RowMajor, **pdx22** $\geq \max(1, \mathbf{m} - \mathbf{q})$;
if **order** = Nag_ColMajor, **pdx22** $\geq \max(1, \mathbf{m} - \mathbf{p})$.

NE_INT

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

NE_INT_2

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

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

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 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 CS decomposition is nearly the exact CS decomposition for the nearby matrix $(X + E)$, where

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

and ϵ is the ***machine precision***.

8 Parallelism and Performance

nag_zuncsd (f08rnc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_zuncsd (f08rnc) 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 required to perform the full CS decomposition is approximately $2m^3$.

The real analogue of this function is nag_dorcsd (f08rac).

10 Example

This example finds the full CS decomposition of a unitary 6 by 6 matrix X (see Section 10.2) partitioned so that the top left block is 2 by 3.

The decomposition is performed both on submatrices of the unitary matrix X and on separated partition matrices. Code is also provided to perform a recombining check if required.

10.1 Program Text

```
/* nag_zuncsd (f08rnc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#include <stdio.h>
#include <math.h>
#include <nag.h>
```

```
#include <nag_stdlib.h>
#include <naga02.h>
#include <nagf08.h>
#include <nagf16.h>
#include <nagx04.h>

int main(void)
{
  /* Scalars */
  Integer exit_status = 0;
  Integer pdx, pdu, pdv, pdx11, pdx12, pdx21, pdx22, pdw;
  Integer i, j, m, p, q, n11, n12, n21, n22, r;
  Integer recombine = 0, reprint = 0;
  Complex cone = { 1.0, 0.0 }, czero = {
  0.0, 0.0};
  /* Arrays */
  Complex *u = 0, *u1 = 0, *u2 = 0, *v = 0, *v1t = 0, *v2t = 0, *w = 0,
         *x = 0, *x11 = 0, *x12 = 0, *x21 = 0, *x22 = 0;
  double *theta = 0;
  /* Nag Types */
  Nag_OrderType order;
  NagError fail;

#ifdef NAG_COLUMN_MAJOR
#define X(I,J) x[(J-1)*pdx + I-1]
#define U(I,J) u[(J-1)*pdu + I-1]
#define V(I,J) v[(J-1)*pdv + I-1]
#define W(I,J) w[(J-1)*pdw + I-1]
#define X11(I,J) x11[(J-1)*pdx11 + I-1]
#define X12(I,J) x12[(J-1)*pdx12 + I-1]
#define X21(I,J) x21[(J-1)*pdx21 + I-1]
#define X22(I,J) x22[(J-1)*pdx22 + I-1]
  order = Nag_ColMajor;
#else
#define X(I,J) x[(I-1)*pdx + J-1]
#define U(I,J) u[(I-1)*pdu + J-1]
#define V(I,J) v[(I-1)*pdv + J-1]
#define W(I,J) w[(I-1)*pdw + J-1]
#define X11(I,J) x11[(I-1)*pdx11 + J-1]
#define X12(I,J) x12[(I-1)*pdx12 + J-1]
#define X21(I,J) x21[(I-1)*pdx21 + J-1]
#define X22(I,J) x22[(I-1)*pdx22 + J-1]
  order = Nag_RowMajor;
#endif

  INIT_FAIL(fail);

  printf("nag_zuncsd (f08rnc) Example Program Results\n\n");
  fflush(stdout);

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

  r = MIN(MIN(p, q), MIN(m - p, m - q));

  if (!(x = NAG_ALLOC(m * m, Complex)) ||
      !(u = NAG_ALLOC(m * m, Complex)) ||
      !(v = NAG_ALLOC(m * m, Complex)) ||
      !(w = NAG_ALLOC(m * m, Complex)) ||
      !(theta = NAG_ALLOC(r, double)) ||
      !(x11 = NAG_ALLOC(p * q, Complex)) ||
      !(x12 = NAG_ALLOC(p * (m - q), Complex)) ||
      !(x21 = NAG_ALLOC((m - p) * q, Complex)) ||
      !(x22 = NAG_ALLOC((m - p) * (m - q), Complex)) ||
      !(u1 = NAG_ALLOC(p * p, Complex)) ||
      !(u2 = NAG_ALLOC((m - p) * (m - p), Complex)) ||
      !(v1t = NAG_ALLOC(q * q, Complex)) ||
      !(v2t = NAG_ALLOC((m - q) * (m - q), Complex)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }
  pdx = m;
  pdu = m;
  pdv = m;
  pdw = m;
#ifdef NAG_COLUMN_MAJOR
  pdx11 = p;
  pdx12 = p;
  pdx21 = m - p;
  pdx22 = m - p;
#else
  pdx11 = q;
  pdx12 = m - q;
  pdx21 = q;
  pdx22 = m - q;
#endif

  /* Read (by column) and print unitary X from data file
   * (as, say, generated by a generalized singular value decomposition).
   */
  for (i = 1; i <= m; i++) {
    for (j = 1; j <= m; j++)
#ifdef _WIN32
      scanf_s(" ( %lf , %lf ) ", &X(j, i).re, &X(j, i).im);
#else
      scanf(" ( %lf , %lf ) ", &X(j, i).re, &X(j, i).im);
#endif
#ifdef _WIN32
    scanf_s("%*[^\n] ");
#else
    scanf("%*[^\n] ");
#endif
  }
  /* Store partitions of X in separate matrices */
  for (j = 1; j <= p; j++) {
    for (i = 1; i <= q; i++)
      X11(j, i) = X(j, i);
    for (i = 1; i <= m - q; i++)
      X12(j, i) = X(j, i + q);
  }
  for (j = 1; j <= m - p; j++) {
    for (i = 1; i <= q; i++)
      X21(j, i) = X(j + p, i);
    for (i = 1; i <= m - q; i++)
      X22(j, i) = X(j + p, i + q);
  }

  /* nag_gen_complx_mat_print_comp (x04dbc).
   * Print least squares solutions.
   */
  nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag, m,
                                m, x, pdx, Nag_BracketForm, "%7.4f",
                                "Unitary matrix X", Nag_IntegerLabels,
                                0, Nag_IntegerLabels, 0, 80, 0, 0, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_complx_mat_print_comp (x04dbc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }
  printf("\n");
  fflush(stdout);

  /* nag_zuncsd (f08rnc).
   * Compute the complete CS factorization of X:
   * X11 is stored in X(1:p,    1:q),  X12 is stored in X(1:p,   q+1:m)
   * X21 is stored in X(p+1:m,  1:q),  X22 is stored in X(p+1:m, q+1:m)
   * U1  is stored in U(1:p,    1:p),  U2  is stored in U(p+1:m, p+1:m)
   * V1  is stored in V(1:q,    1:q),  V2  is stored in V(q+1:m, q+1:m)
   */

  /* This is how you might pass partitions as sub-matrices */
  nag_zuncsd(order, Nag_AllU, Nag_AllU, Nag_AllVT, Nag_AllVT, Nag_UpperMinus,
             m, p, q, x, pdx, &X(1, q + 1), pdx, &X(p + 1, 1), pdx, &X(p + 1,
                                                                     q + 1),
             pdx, theta, u, pdu, &U(p + 1, p + 1), pdu, v, pdv, &V(q + 1,
                                                               q + 1),
             pdv, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zuncsd (f08rnc).\n%s\n", fail.message);
    exit_status = 2;
    goto END;
  }

  /* Print Theta using matrix printing routine
   * nag_gen_real_mat_print (x04cac).
   * Note: U1, U2, V1T, V2T not printed since these may differ by a sign
   * change in columns of U1, U2 and corresponding rows of V1T, V2T.
   */
  printf(" Component of CS factorization of X:\n\n");
  fflush(stdout);
  nag_gen_real_mat_print(Nag_ColMajor, Nag_GeneralMatrix, Nag_NonUnitDiag, r,
                         1, theta, r, "    Theta", 0, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n", fail.message);
    exit_status = 3;
    goto END;
  }
  printf("\n");
  fflush(stdout);

  /* And this is how you might pass partitions as separate matrices. */
  nag_zuncsd(order, Nag_AllU, Nag_AllU, Nag_AllVT, Nag_AllVT, Nag_UpperMinus,
             m, p, q, x11, pdx11, x12, pdx12, x21, pdx21, x22, pdx22, theta,
             u1, p, u2, m - p, v1t, q, v2t, m - q, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zuncsd (f08rnc).\n%s\n", fail.message);
    exit_status = 4;
    goto END;
  }
  if (reprint != 0) {
    printf("Component of CS factorization of X using separate matrices:\n");
    fflush(stdout);
    nag_gen_real_mat_print(Nag_ColMajor, Nag_GeneralMatrix, Nag_NonUnitDiag,
                           r, 1, theta, r, "    Theta", 0, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n",
             fail.message);
      exit_status = 5;
      goto END;
    }
    printf("\n");
    fflush(stdout);
  }

  if (recombine != 0) {
    /* Recombining should return the original matrix.
       Assemble Sigma_p into X
     */
    for (i = 1; i <= m; i++) {
      for (j = 1; j <= m; j++) {
        X(i, j) = czero;
      }
    }
    n11 = MIN(p, q) - r;
    n12 = MIN(p, m - q) - r;
    n21 = MIN(m - p, q) - r;
    n22 = MIN(m - p, m - q) - r;

    /* top half */
    for (j = 1; j <= n11; j++) {
      X(j, j) = cone;
    }
    for (j = 1; j <= r; j++) {
      X(j + n11, j + n11).re = cos(theta[j - 1]);
      X(j + n11, j + n11).im = 0.0;
      X(j + n11, j + n11 + r + n21 + n22).re = -sin(theta[j - 1]);
      X(j + n11, j + n11 + r + n21 + n22).im = 0.0;
    }
    for (j = 1; j <= n12; j++) {
      X(j + n11 + r, j + n11 + r + n21 + n22 + r).re = -1.0;
      X(j + n11 + r, j + n11 + r + n21 + n22 + r).im = 0.0;
    }
    /* bottom half */
    for (j = 1; j <= n22; j++) {
      X(p + j, q + j) = cone;
    }
    for (j = 1; j <= r; j++) {
      X(p + n22 + j, j + n11).re = sin(theta[j - 1]);
      X(p + n22 + j, j + n11).im = 0.0;
      X(p + n22 + j, j + r + n21 + n22).re = cos(theta[j - 1]);
      X(p + n22 + j, j + r + n21 + n22).im = 0.0;
    }
    for (j = 1; j <= n21; j++) {
      X(p + n22 + r + j, n11 + r + j) = cone;
    }

    /* multiply U * Sigma_p into w */
    nag_zgemm(order, Nag_NoTrans, Nag_NoTrans, m, m, m, cone,
              &U(1, 1), pdu, &X(1, 1), pdx, czero, &W(1, 1), pdw, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_zgemm (f16zac).\n%s\n", fail.message);
      exit_status = 6;
      goto END;
    }
    /* form U * Sigma_p * V^T into u */
    nag_zgemm(order, Nag_NoTrans, Nag_NoTrans, m, m, m, cone,
              &W(1, 1), pdw, &V(1, 1), pdv, czero, &U(1, 1), pdu, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_zgemm (f16zac).\n%s\n", fail.message);
      exit_status = 7;
      goto END;
    }
    /* nag_gen_complx_mat_print_comp (x04dbc).
     * Print least squares solutions.
     */
    nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag,
                                  m, m, &U(1, 1), pdu, Nag_BracketForm,
                                  "%7.4f", "    U * Sigma_p * V^T",
                                  Nag_IntegerLabels, 0, Nag_IntegerLabels, 0,
                                  80, 0, 0, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_gen_complx_mat_print_comp (x04dbc).\n%s\n",
             fail.message);
      exit_status = 8;
      goto END;
    }
    printf("\n");
    fflush(stdout);
  }

END:
  NAG_FREE(x);
  NAG_FREE(u);
  NAG_FREE(v);
  NAG_FREE(w);
  NAG_FREE(theta);
  NAG_FREE(x11);
  NAG_FREE(x12);
  NAG_FREE(x21);
  NAG_FREE(x22);
  NAG_FREE(u1);
  NAG_FREE(u2);
  NAG_FREE(v1t);
  NAG_FREE(v2t);
  return exit_status;
}
```

10.2 Program Data

```
nag_zuncsd (f08rnc) Example Program Data

   6        2       3        : m,  p,  q

 ( -1.3038e-02, -3.2595e-01)
 (  4.2764e-01, -6.2582e-01)
 ( -3.2595e-01,  1.6428e-01)
 (  1.5906e-01, -5.2151e-03)
 ( -1.7210e-01, -1.3038e-02)
 ( -2.6336e-01, -2.4772e-01) : column 1 of unitary matrix X

 ( -1.4039e-01, -2.6167e-01)
 (  8.6298e-02, -3.8174e-02)
 (  3.8163e-01, -1.8219e-01)
 ( -2.8207e-01,  1.9732e-01)
 ( -5.0942e-01, -5.0319e-01)
 ( -1.0861e-01,  2.8474e-01) : column 2 of unitary matrix X

 (  2.5177e-01, -7.9789e-01)
 ( -3.2188e-01,  1.6112e-01)
 (  1.3231e-01, -1.4565e-02)
 (  2.1598e-01,  1.8813e-01)
 (  3.6488e-02,  2.0316e-01)
 (  1.0906e-01, -1.2712e-01) : column 3 of unitary matrix X

 ( -5.0956e-02, -2.1750e-01)
 (  1.1979e-01,  1.6319e-01)
 ( -5.0671e-01,  1.8615e-01)
 ( -4.0163e-01,  2.6787e-01)
 (  1.9271e-01,  1.5574e-01)
 ( -8.8159e-02,  5.6169e-01) : column 4 of unitary matrix X

 ( -4.5947e-02,  1.4052e-04)
 ( -8.0311e-02, -4.3605e-01)
 (  5.9714e-02, -5.8974e-01)
 ( -4.6443e-02,  3.0864e-01)
 (  5.7843e-01, -1.2439e-01)
 (  1.5763e-02,  4.7130e-02) : column 5 of unitary matrix X

 ( -5.2773e-02, -2.2492e-01)
 ( -3.8117e-02, -2.1907e-01)
 ( -1.3850e-01, -9.0941e-02)
 ( -3.7354e-01, -5.5148e-01)
 ( -1.8815e-02, -5.5686e-02)
 (  6.5007e-01,  4.9173e-03) : column 6 of unitary matrix X
```

10.3 Program Results

```
nag_zuncsd (f08rnc) Example Program Results

 Unitary matrix X
                   1                  2                  3                  4
 1  (-0.0130,-0.3260)  (-0.1404,-0.2617)  ( 0.2518,-0.7979)  (-0.0510,-0.2175)
 2  ( 0.4276,-0.6258)  ( 0.0863,-0.0382)  (-0.3219, 0.1611)  ( 0.1198, 0.1632)
 3  (-0.3260, 0.1643)  ( 0.3816,-0.1822)  ( 0.1323,-0.0146)  (-0.5067, 0.1862)
 4  ( 0.1591,-0.0052)  (-0.2821, 0.1973)  ( 0.2160, 0.1881)  (-0.4016, 0.2679)
 5  (-0.1721,-0.0130)  (-0.5094,-0.5032)  ( 0.0365, 0.2032)  ( 0.1927, 0.1557)
 6  (-0.2634,-0.2477)  (-0.1086, 0.2847)  ( 0.1091,-0.1271)  (-0.0882, 0.5617)

                   5                  6
 1  (-0.0459, 0.0001)  (-0.0528,-0.2249)
 2  (-0.0803,-0.4360)  (-0.0381,-0.2191)
 3  ( 0.0597,-0.5897)  (-0.1385,-0.0909)
 4  (-0.0464, 0.3086)  (-0.3735,-0.5515)
 5  ( 0.5784,-0.1244)  (-0.0188,-0.0557)
 6  ( 0.0158, 0.0471)  ( 0.6501, 0.0049)

 Component of CS factorization of X:

     Theta
         1
 1   0.3146
 2   0.5760
```