NAG Library Function Document

nag_zggrqf (f08ztc)

1 Purpose

nag_zggrqf (f08ztc) computes a generalized RQ factorization of a complex matrix pair (A, B), where A is an m by n matrix and B is a p by n matrix.

2 Specification

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

```
void nag_zggrqf (Nag_OrderType order, Integer m, Integer p, Integer n,
     Complex a[], Integer pda, Complex taua[], Complex b[], Integer pdb,
     Complex taub[], NagError *fail)
```

3 Description

nag_zggrqf (f08ztc) forms the generalized RQ factorization of an m by n matrix A and a p by n matrix B

$$A = RQ, \quad B = ZTQ,$$

where Q is an n by n unitary matrix, Z is a p by p unitary matrix and R and T are of the form

$$R = \begin{cases} \begin{array}{c} \\ m \end{array} \begin{array}{cc} n-m & m \\ \left(\begin{array}{c} 0 \end{array} \right. & \left. \begin{array}{c} R_{12} \end{array} \right) \end{array}; & \text{if } m \le n, \\ \begin{array}{c} \\ m-n \\ n \end{array} \begin{array}{c} n \\ \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix} \end{array}; & \text{if } m > n, \end{cases}$$

with R_{12} or R_{21} upper triangular,

$$T = \begin{cases} \begin{array}{c} \\ n \\ p-n \end{array} \begin{array}{c} n \\ \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \end{array}; & \text{if } p \ge n, \\ \begin{array}{c} \\ p \end{array} \begin{array}{cc} p & n-p \\ \left(T_{11} \right. & \left. T_{12} \right) \end{array}; & \text{if } p < n, \end{cases}$$

with T_{11} upper triangular.

In particular, if B is square and nonsingular, the generalized RQ factorization of A and B implicitly gives the RQ factorization of AB^{-1} as

$$AB^{-1} = \left(RT^{-1}\right)Z^{\mathrm{H}}.$$

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

Anderson E, Bai Z and Dongarra J (1992) Generalized *QR* factorization and its applications *Linear Algebra Appl. (Volume 162–164)* 243–271

Hammarling S (1987) The numerical solution of the general Gauss-Markov linear model *Mathematics in Signal Processing* (eds T S Durrani, J B Abbiss, J E Hudson, R N Madan, J G McWhirter and T A Moore) 441–456 Oxford University Press

Paige C C (1990) Some aspects of generalized QR factorizations . *In Reliable Numerical Computation* (eds M G Cox and S Hammarling) 73–91 Oxford University Press

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: **m** – Integer *Input*

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

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

3: **p** – Integer *Input*

On entry: p, the number of rows of the matrix B.

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

4: **n** – Integer *Input*

On entry: n, the number of columns of the matrices A and B.

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

5: **a**[*dim*] – Complex *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.

Where $\mathbf{A}(i,j)$ appears in this document, it refers to the array element

$\mathbf{a}[(j-1) \times \mathbf{pda} + i - 1]$ 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 $m \leq n$, the upper triangle of the subarray $\mathbf{A}(1:m, n-m+1:n)$ contains the m by m upper triangular matrix R_{12}.

If $m \geq n$, the elements on and above the $(m-n)$th subdiagonal contain the m by n upper trapezoidal matrix R; the remaining elements, with the array **taua**, represent the unitary matrix Q as a product of $\min(m,n)$ elementary reflectors (see Section 3.3.6 in the f08 Chapter Introduction).

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: **taua**[**min**(**m**, **n**)] – Complex *Output*

On exit: the scalar factors of the elementary reflectors which represent the unitary matrix Q.

8: **b**[*dim*] – Complex *Input/Output*

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

max(1, **pdb** × **n**) when **order** = Nag_ColMajor;
max(1, **p** × **pdb**) when **order** = Nag_RowMajor.

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

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

On entry: the p by n matrix B.

On exit: the elements on and above the diagonal of the array contain the $\min(p,n)$ by n upper trapezoidal matrix T (T is upper triangular if $p \geq n$); the elements below the diagonal, with the array **taub**, represent the unitary matrix Z as a product of elementary reflectors (see Section 3.3.6 in the f08 Chapter Introduction).

9: **pdb** – Integer *Input*

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

Constraints:

if **order** = Nag_ColMajor, **pdb** ≥ max(1, **p**);
if **order** = Nag_RowMajor, **pdb** ≥ max(1, **n**).

10: **taub**[**min**(**p**, **n**)] – Complex *Output*

On exit: the scalar factors of the elementary reflectors which represent the unitary matrix Z.

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

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

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

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

NE_INT

On entry, **m** = ⟨*value*⟩.
Constraint: **m** ≥ 0.

On entry, **n** = ⟨*value*⟩.
Constraint: **n** ≥ 0.

On entry, **p** = ⟨*value*⟩.
Constraint: **p** ≥ 0.

On entry, **pda** = ⟨*value*⟩.
Constraint: **pda** > 0.

On entry, **pdb** = ⟨*value*⟩.
Constraint: **pdb** > 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** $\geq \max(1, \mathbf{n})$.

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

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

NE_INTERNAL_ERROR

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

7 Accuracy

The computed generalized RQ factorization is the exact factorization for nearby matrices $(A+E)$ and $(B+F)$, where

$$\|E\|_2 = O\epsilon\|A\|_2 \quad \text{and} \quad \|F\|_2 = O\epsilon\|B\|_2,$$

and ϵ is the ***machine precision***.

8 Parallelism and Performance

nag_zggrqf (f08ztc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_zggrqf (f08ztc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

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

9 Further Comments

The unitary matrices Q and Z may be formed explicitly by calls to nag_zungrq (f08cwc) and nag_zungqr (f08atc) respectively. nag_zunmrq (f08cxc) may be used to multiply Q by another matrix and nag_zunmqr (f08auc) may be used to multiply Z by another matrix.

The real analogue of this function is nag_dggrqf (f08zfc).

10 Example

This example solves the general Gauss–Markov linear model problem

$$\min_x \|y\|_2 \quad \text{subject to} \quad d = Ax + By$$

where

$$A = \begin{pmatrix} 0.96 - 0.81i & -0.03 + 0.96i & -0.91 + 2.06i & -0.05 + 0.41i \\ -0.98 + 1.98i & -1.20 + 0.19i & -0.66 + 0.42i & -0.81 + 0.56i \\ 0.62 - 0.46i & 1.01 + 0.02i & 0.63 - 0.17i & -1.11 + 0.60i \\ 0.37 + 0.38i & 0.19 - 0.54i & -0.98 - 0.36i & 0.22 - 0.20i \\ 0.83 + 0.51i & 0.20 + 0.01i & -0.17 - 0.46i & 1.47 + 1.59i \\ 1.08 - 0.28i & 0.20 - 0.12i & -0.07 + 1.23i & 0.26 + 0.26i \end{pmatrix},$$

$$B = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}, \quad c = \begin{pmatrix} -2.54 + 0.09i \\ 1.65 - 2.26i \\ -2.11 - 3.96i \\ 1.82 + 3.30i \\ -6.41 + 3.77i \\ 2.07 + 0.66i \end{pmatrix} \quad \text{and} \quad d = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The constraints $Bx = d$ correspond to $x_1 = x_3$ and $x_2 = x_4$.

The solution is obtained by first obtaining a generalized QR factorization of the matrix pair (A, B). The example illustrates the general solution process, although the above data corresponds to a simple weighted least squares problem.

10.1 Program Text

```
/* nag_zggrqf (f08ztc) Example Program.
 *
 * Copyright 2011 Numerical Algorithms Group.
 *
 * Mark 23, 2011.
 */

#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <naga02.h>
#include <nagf07.h>
#include <nagf08.h>
#include <nagf16.h>

int main(void)
{
  /* Scalars */
  Complex       alpha, beta;
  double        rnorm;
  Integer       i, j, m, n, p, pda, pdb, pdc, pdd, pdx;
  Integer       y1rows, y2rows, y3rows;
  Integer       exit_status = 0;

  /* Arrays */
  Complex       *a = 0, *b = 0, *c = 0, *d = 0, *taua = 0, *taub = 0, *x = 0;

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

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

  INIT_FAIL(fail);

  printf("nag_zggrqf (f08ztc) Example Program Results\n\n");

  /* Skip heading in data file */
  scanf("%*[^\n]");
  scanf("%ld%ld%ld%*[^\n]", &m, &n, &p);
  if (n < 0 || m < 0 || p < 0)
    {
      printf("Invalid n, m or p\n");
      exit_status = 1;
      goto END;
    }

#ifdef NAG_COLUMN_MAJOR
  pda = m;
  pdb = p;
  pdc = m;
  pdd = p;
  pdx = n;
#else
  pda = n;
  pdb = n;
  pdc = 1;
  pdd = 1;
  pdx = 1;
#endif

  /* Allocate memory */
  if (!(a    = NAG_ALLOC(m*n, Complex)) ||
      !(b    = NAG_ALLOC(p*n, Complex)) ||
      !(c    = NAG_ALLOC(m, Complex)) ||
      !(d    = NAG_ALLOC(p, Complex)) ||
      !(taua = NAG_ALLOC(MIN(m, n), Complex)) ||
      !(taub = NAG_ALLOC(MIN(p, n), Complex)) ||
      !(x    = NAG_ALLOC(n, Complex)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Read A, B, c and d from data file for the problem
   * min{||c-Ax||_2, x in R^n and B x = d}.
   */
  for (i = 1; i <= m; ++i)
    for (j = 1; j <= n; ++j)
      scanf(" ( %lf , %lf )", &A(i, j).re, &A(i, j).im);
  scanf("%*[^\n]");
  for (i = 1; i <= p; ++i)
    for (j = 1; j <= n; ++j)
      scanf(" ( %lf , %lf )", &B(i, j).re, &B(i, j).im);
  scanf("%*[^\n]");
  for (i = 0; i < m; ++i) scanf(" ( %lf , %lf )", &c[i].re, &c[i].im);
  scanf("%*[^\n]");
  for (i = 0; i < p; ++i) scanf(" ( %lf , %lf )", &d[i].re, &d[i].im);
  scanf("%*[^\n]");


  /* First compute the generalized RQ factorization of (A,B) as
   * B = (0 R12)*Q,   A = Z*(T11 T12 T13)*Q = T*Q.
   *                        ( 0  T22 T23)
   * where R12, T11 and T22 are upper triangular,
   * using  nag_zggrqf (f08ztc).
   */
  nag_zggrqf(order, p, m, n, b, pdb, taub, a, pda, taua, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_zggrqf (f08ztc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Now, Z^H * (c-Ax) = Z^H * c - T*Q*x, and
   * let f = (f1) = Z^H * (c1) => minimize ||f - T*Q*x||
   *         (f2)         (c2)
   * Compute f using  nag_zunmqr (f08auc), storing result in c
   */
  nag_zunmqr(order, Nag_LeftSide, Nag_ConjTrans, m, 1, MIN(m, n), a, pda, taua,
             c, pdc, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_zunmqr (f08auc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Putting Q*x = (y1), B * x = d becomes (0 R12) (y1) = d;
   *              (w )                          (w )
   * => R12 * w = d.
   * Solve for w using nag_dtrtrs (f07tec), storing result in d;
   * R12 is (p by p) triangular submatrix starting at B(1,n-p+1).
   */
  nag_ztrtrs(order, Nag_Upper, Nag_NoTrans, Nag_NonUnitDiag, p, 1,
             &B(1, n - p + 1), pdb, d, pdd, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_ztrtrs (f07tsc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* The problem now reduces to finding the minimum norm of
   * g = (g1) = (f1) - T11*y1 - (T12 T13)*w
   *     (g2)   (f2)          - (T22 T23)*w.
   * Form c1 = f1 - (T12 T13)*w using  nag_zgemv (f16sac).
   */
  alpha = nag_complex(-1.0,0.0);
  beta = nag_complex(1.0,0.0);
  y1rows = n - p;
  nag_zgemv(order, Nag_NoTrans, y1rows, p, alpha, &A(1, n-p+1), pda, d, 1,
            beta, c, 1, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_zgemv (f16sac).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
   /* => now (g1) = c - T11*y1 and ||g1|| = 0 when T11 * y1 = c1.
    * Solve this for y1 using nag_ztrtrs (f07tsc) storing result in c1.
    */
  nag_ztrtrs(order, Nag_Upper, Nag_NoTrans, Nag_NonUnitDiag, y1rows, 1, a, pda,
             c, pdc, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_ztrtrs (f07tsc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* So now Q*x = (y1) is stored in (c1), which is now copied to x.
   *             (w )              (d )
   */
  for (i = 0; i < y1rows; ++i) x[i] = nag_complex(c[i].re,c[i].im);
  for (i = 0; i < n-y1rows; ++i) x[y1rows+i] = nag_complex(d[i].re,d[i].im);

  /* Compute x by applying Q inverse using nag_zunmrq (f08cxc). */
  nag_zunmrq(order, Nag_LeftSide, Nag_ConjTrans, n, 1, p, b, pdb, taub, x, pdx,
             &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_zunmrq (f08cxc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* It remains to minimize  ||g2||, g2 = f2 - (T22 T23)*w.
   * Putting w = (y2), gives g2 = f2 - T22*y2 - T23*y3
   *             (y3)
   * [y2 stored in d1, first y2rows of d; y3 stored in d2, next n-m rows of d.]
   *
   * First form T22*y2 using  nag_ztrmv (f16sfc) where y2 is held in d.
   */
  y2rows = MIN(m, n) - y1rows;
  alpha = nag_complex(1.0,0.0);
  nag_ztrmv(order, Nag_Upper, Nag_NoTrans, Nag_NonUnitDiag, y2rows, alpha,
            &A(n-p+1, n-p+1), pda, d, 1, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_ztrmv (f16sfc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Then, f2 - T22*y2 (c2 = c2 - d) */
  for (i = 0; i < y2rows; ++i)
    c[y1rows + i] = nag_complex_subtract(c[y1rows+i], d[i]);
  y2rows = m - y1rows;

  if (m < n)
    {
      y3rows = n - m;
      /* Then g2 = f2 - T22*y2 - T23*y3 (c2 = c2 - T23*d2) */
      alpha = nag_complex(-1.0,0.0);
      nag_zgemv(order, Nag_NoTrans, y2rows, y3rows, alpha, &A(n-p+1, m+1), pda,
                &d[y2rows], 1, beta, &c[y1rows], 1, &fail);
      if (fail.code != NE_NOERROR)
        {
          printf("Error from nag_zgemv (f16sac).\n%s\n", fail.message);
          exit_status = 1;
          goto END;
        }
    }

  /* Compute ||g|| = ||g2|| = norm(f2 - T22*y2 - T23*y3)
   * using nag_zge_norm (f16uac).
   */
  nag_zge_norm(Nag_ColMajor, Nag_FrobeniusNorm, y2rows, 1, &c[y1rows], y2rows,
               &rnorm, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_zge_norm (f16uac).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Print least squares solution x */
  printf("Constrained least squares solution\n");
  for (i = 0; i < n; ++i)
    printf(" (%7.4f, %7.4f)%s", x[i].re, x[i].im, i%4 == 3?"\n":"");

  /* Print the square root of the residual sum of squares */
  printf("\nSquare root of the residual sum of squares\n");
  printf("%11.2e\n", rnorm);

 END:
  NAG_FREE(a);
  NAG_FREE(b);
  NAG_FREE(c);
  NAG_FREE(d);
  NAG_FREE(taua);
  NAG_FREE(taub);
  NAG_FREE(x);

  return exit_status;
}
```

10.2 Program Data

```
nag_zggrqf (f08ztc) Example Program Data

  6            4            2                         : m, n and p

( 0.96,-0.81) (-0.03, 0.96) (-0.91, 2.06) (-0.05, 0.41)
(-0.98, 1.98) (-1.20, 0.19) (-0.66, 0.42) (-0.81, 0.56)
( 0.62,-0.46) ( 1.01, 0.02) ( 0.63,-0.17) (-1.11, 0.60)
( 0.37, 0.38) ( 0.19,-0.54) (-0.98,-0.36) ( 0.22,-0.20)
( 0.83, 0.51) ( 0.20, 0.01) (-0.17,-0.46) ( 1.47, 1.59)
( 1.08,-0.28) ( 0.20,-0.12) (-0.07, 1.23) ( 0.26, 0.26) : matrix A

( 1.00, 0.00) ( 0.00, 0.00) (-1.00, 0.00) ( 0.00, 0.00)
( 0.00, 0.00) ( 1.00, 0.00) ( 0.00, 0.00) (-1.00, 0.00) : matrix B

(-2.54, 0.09)
( 1.65,-2.26)
(-2.11,-3.96)
( 1.82, 3.30)
(-6.41, 3.77)
( 2.07, 0.66)                                         : vector c

( 0.00, 0.00)
( 0.00, 0.00)                                         : vector d
```

10.3 Program Results

```
nag_zggrqf (f08ztc) Example Program Results

Constrained least squares solution
 ( 1.0874, -1.9621) (-0.7409,  3.7297) ( 1.0874, -1.9621) (-0.7409,  3.7297)

Square root of the residual sum of squares
   1.59e-01
```